Software Engineering

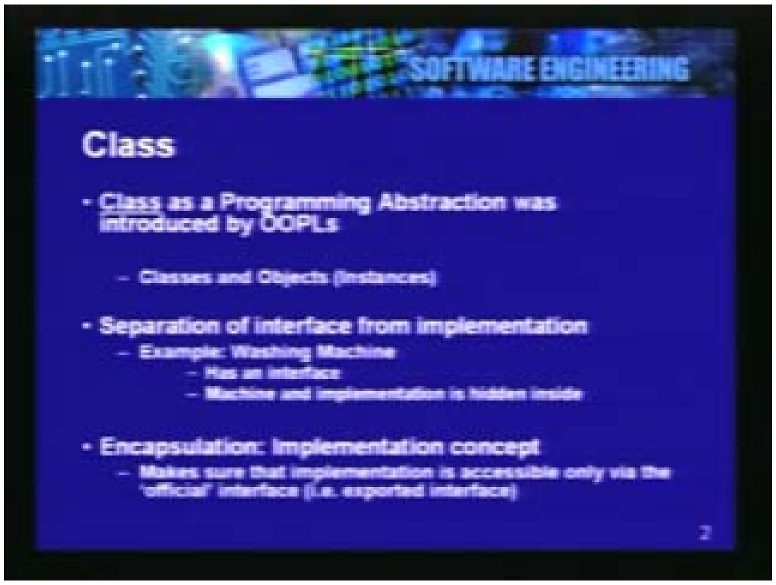
Prof. Rushikesh K. Joshi
Computer Science & Engineering
Indian Institute of Technology, Bombay

Lecture - 16

Class and Component Level Design

We have to design at different levels. Even for software requirements, we have to first analyze the requirement and then as we step on to design, we have to design our high level architecture and then we have to design modules packages at high levels. Then we have to look at class and component levels which are further at the lower layers. We will now discover the different issues which we have to take into account when we design at class and component level. As we know as shown in this slide, class is a programming abstraction which was mainly introduced by object oriented programming languages.

(Refer Slide Time: 01:25)


SOFTWARE ENGINEERING

Class

- Class as a Programming Abstraction was introduced by OOPLs
  - Classes and Objects (Instances)
- Separation of interface from implementation
  - Example: Washing Machine
    - Has an interface
    - Machine and implementation is hidden inside
- Encapsulation: Implementation concept
  - Makes sure that implementation is accessible only via the 'official' interface (i.e. exported interface)

2

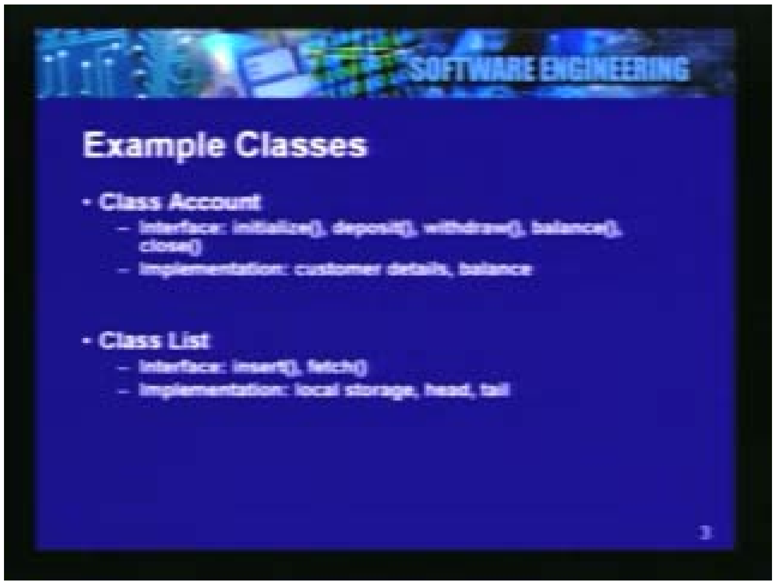
We have this notion of classes and objects. Classes provide us with definition of structure and the behavior of its instances. So when we say we have a class of a few objects and whatever is the behavior for the entire class that is described by the class description. When we use this term objects, we are mainly referring to instances of the classes, the end instances. The main concept in the class as an abstraction is, separation of interface from the implementation. We separate the externally observable behavior from its implementation. For example look at real life objects such as a washing machine; we operate the washing machine using an external interface. That is the observable interface of that object and the implementation that actually drives the machine, the logic inside the machine is hidden. So we have the interface and the implementation.

If you look at any real life object such as a data projector or a machine or any other real life object that you see or any manufactured machine, it has a nice interface through which you have to use the machine. If you open the machine and manipulate it directly, some of the interfaces may not be guaranteed because you might have changed the implementation and you cannot guarantee the interface. That property of enforcing the abstraction of the object, the externally observable behavior through implementation is called encapsulation. Encapsulation guarantees that the implementation, nice hidden inside and it is accessible only via the only official interface. Whatever is available to user to a given role, that interface gives the access to the object, so that interface is exported.

These are the important concepts when we talk of classes and abstraction. When we have the actual washing machine that is an object and the definition of that washing machine is a class. So we have a class of washing machines, in that class we may have many units. A class is major programming abstraction and the same can be used in design at design level. You can design systems of classes and then its implementation can be through an object oriented programming languages. Mostly when you do object oriented software engineering you come across this term called class.

UML has a notation for representing a class and interactions amongst the classes. You can design at the class level and then when you implement you convert those classes into programming language classes. Some example classes are given on this slide. For example, Class Account: when we want to design a class, we have to first think of its interface, which is very important. Design of individual classes means you first think of the abstractions of the classes.

(Refer Slide Time: 04:50)



SOFTWARE ENGINEERING

Example Classes

- Class Account
  - Interface: initialize(), deposit(), withdraw(), balance(), close()
  - Implementation: customer details, balance
- Class List
  - Interface: insert(), fetch()
  - Implementation: local storage, head, tail

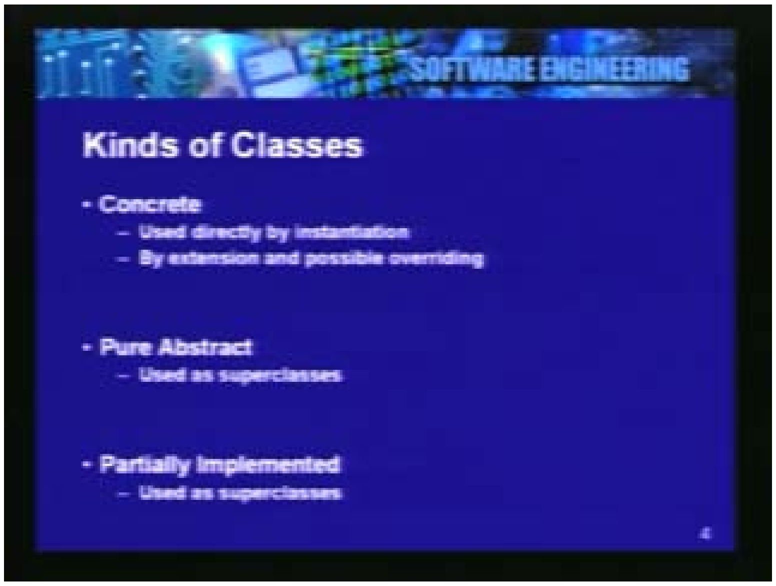
3

Now we are looking at two examples here. One is ‘Class Account’. Its interface is, first initialize the account. So when you initialize, what are your initial attributes? What is state of account when it is initialized? Then you can operate during a lifetime of the account, you can operate on this object of Class Account, any object of class account through this interface. You can invoke methods such as deposit (), withdraw (). You can check the balance using method balance () or you can close an account. This is an interface of Class Account through which the object is accessible to you and the implementation of Class Account that is the internal details involve data structures or data representation for say storing customer details, the actual amount or the balance etc that is hidden from the externally observable behavior.

You cannot directly manipulate this. You have to invoke the member functions provided to you by the class description and any instance of this class will provide you these interfaces. So instances are objects and classes are descriptions of those objects. There is another class ‘Class List’ which has an interface insert () and fetch (). You can insert into the list and you can fetch an item from the list. We can operate on this list through these member functions. Its implementation could be the local storage; how the list is stored internally, whether it has a static storage or is it dynamically allocated and so on.

The head and the tail pointers whatever are needed to maintain the state and to make these operations possible on the list. The implementation details are hidden. Class has implementation and interface. These are the main components and the guarantee is given by the programming language that the object will be accessible only through the interface. If such a guarantee is given, then you are enforcing abstraction by encapsulation. There can be different kinds of classes when you are thinking of classes, when you are designing your classes.

(Refer Slide Time: 07:21)


SOFTWARE ENGINEERING

Kinds of Classes

- Concrete
  - Used directly by instantiation
  - By extension and possible overriding
- Pure Abstract
  - Used as superclasses
- Partially Implemented
  - Used as superclasses

4

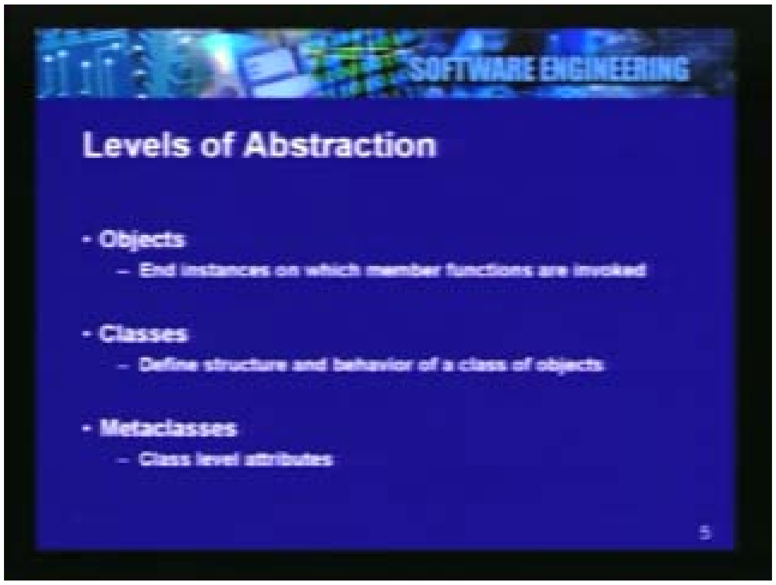
Mainly we can classify them into three categories: concrete classes, pure abstract classes, and partially implemented classes. Concrete classes are those which are used directly by instantiation. We will see these in different libraries. You can directly use such a class and instantiate it and invoke method. So it is fully implemented class or if you want to extend, you can extend it and possibly override a few member functions in that class, which is also possible through inheritance. But main property is that you will be able to directly instantiate it and access member function on it as it is completely implemented. One can of course override some of the member functions and specialize the class using inheritance.

Second type of classes that you can see is pure abstract classes. These are used as super classes. They can only be used as super classes because they are pure abstract. What does mean by pure abstractness of a class? It means that no member function is completely implemented in the class. In fact no member function is implemented in the classes is even more stronger. The third type of classes is partially implemented class. In this case we have a few member functions which could be implemented. We have concrete classes, pure abstract classes where you find no member function implemented. In partially implemented classes we will find a few member functions could be implemented and few are left out. They are not implemented in a class. So that class cannot be instantiated because those member functions are not available.

Hence when we design classes, we can think of these three kinds of classes. Concrete classes directly come forward from the application domain. Pure abstract classes come forward when we want to organize the classes into hierarchies which exhibit common behavior. Basically super class provides you the common abstraction. Partially implemented classes are very useful when you design frameworks. When you design frameworks you want to design a framework in terms of this partial implementation. So that the actual implementer of an application takes this framework and only provides the sub classes and a lot of logic is embedded in the framework itself.

These are the three main applications of these classes. The concrete class is mainly used as it is they are directly available in the real life domain or you can use them as library. The pure abstract classes are used for capturing the general behavior or the common behavior through inheritance. The partially implemented classes are very useful in frameworks. So we can see that the three levels of abstraction you talk of objects which are instances of classes. When you talk of classes, it defines structure and behavior of a class of object, a full category of objects. So that all instances of that class then guarantee whatever behavior that is guaranteed by the class description.

(Refer Slide Time: 10:25)


SOFTWARE ENGINEERING

Levels of Abstraction

- Objects
  - End instances on which member functions are invoked
- Classes
  - Define structure and behavior of a class of objects
- Metaclasses
  - Class level attributes

5

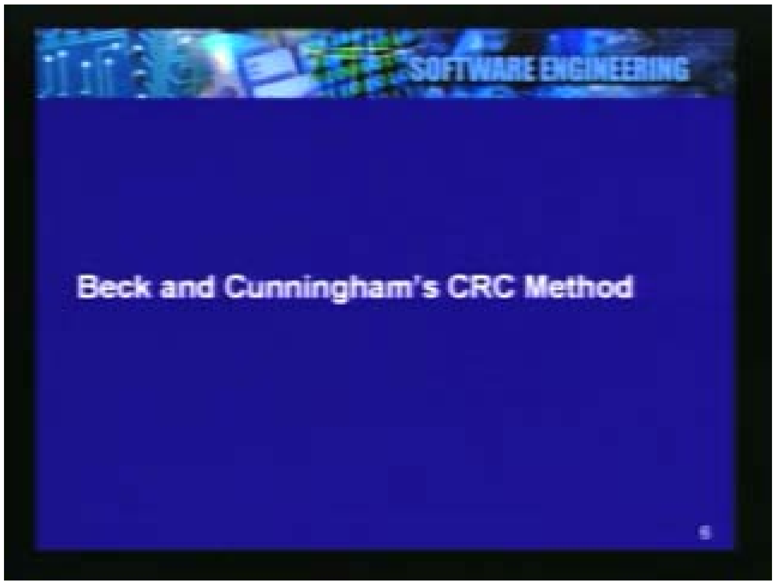
Then there are Meta classes which are classes of classes, if you want to describe class level attributes. For example, you want to keep the count of number of instances of a given class. Where are you going to keep that state? Are you going to keep that state in every object? That will be unnecessary redundancy and then you have to again maintain the consistency. Instead if you maintain that attribute, number of instances of given class within the class itself and you can ask the class to give the number of instances that it has created then you are talking of behavior of the class itself. You are going to invoke member functions on the class. That means the class is also going to be treated as an object, as an instance of its own class which is called Meta class.

When we want to model class level attributes, when you want to design some thing at Meta levels, we are talking of class level attributes. Then we need this modeling ability or Meta classes. These attributes are at Meta level and the member functions are at Meta level. The member functions invoke those attributes that describe that Meta level. They are invoked on classes themselves. In such a system, classes are treated as instances.

Not all programming languages provide you this facility of Meta classes directly through a notion of Meta class. But they give you some alternative. For example in Cplusplus and Java, you have static attributes and static member functions. You can invoke those member functions directly on the class. You do not need an instance for it. Those are the class level attributes and class level member functions. In programming language Smalltalk, you have a direct support of Meta classes and you can go on in the Meta class hierarchy. So there are Meta classes and so on. But languages like Cplusplus and Java, take care of some of these meta level attributes through these alternative construct such as static member functions, static attributes constructor and so on. Now we are going to look at an interesting method of designing classes.

Beck and Cunningham's CRC method: This was discussed in an OOPS law paper around ten fifteen years ago. What does CRC stand for in this term CRC method for object oriented thinking? First C stands for Classes, R stands for Responsibilities and the second C stands for Collaborators.

(Refer Slide Time: 12:57)

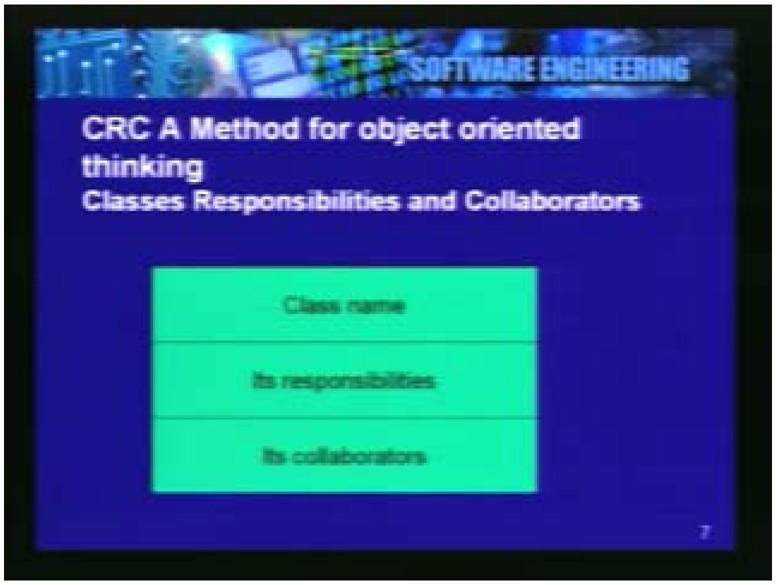


SOFTWARE ENGINEERING

Beck and Cunningham's CRC Method

6

(Refer Slide Time: 13:05)



SOFTWARE ENGINEERING

CRC A Method for object oriented thinking
Classes Responsibilities and Collaborators

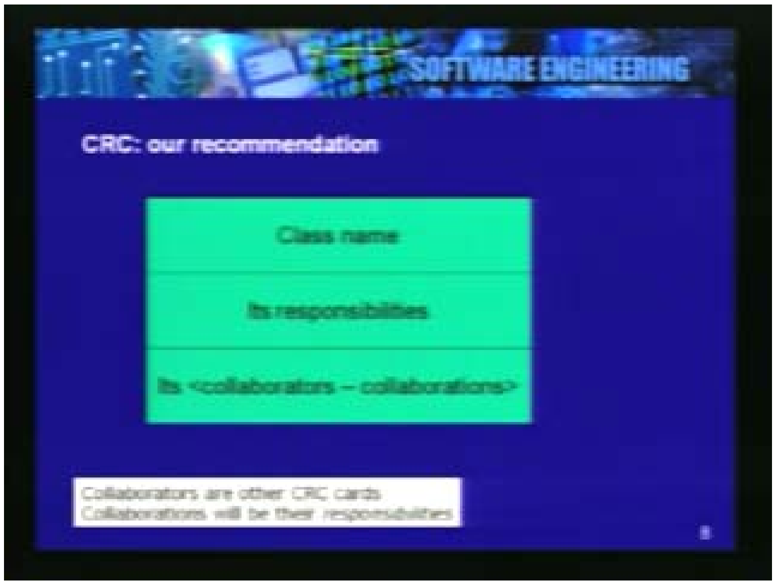
Class name
Its responsibilities
Its collaborators

7

We have this template which is shown on this slide. First we describe the class name, and then we want to list its responsibilities, the responsibilities of the class and then its collaborators with which class these classes collaborates.

The responsibilities of a given class are the member functions exported by the class, so that the class carries out responsibilities via the implementation of those member functions. And in order to complete these responsibilities who are the other collaborator classes with which classes does it need to collaborate? Many classes are stand alone and you can directly use them without having to connect them to other classes. But in a big system there are collaborations. Through those collaborations, collaborative actions are possible and you can implement higher level functionality. The collaborator of a given class also has to be listed. Our recommendation is that in the third section along with the collaborators you also list out collaboration.

(Refer Slide Time: 14:25)



SOFTWARE ENGINEERING

CRC: our recommendation

Class name
Its responsibilities
Its <collaborators – collaborations>

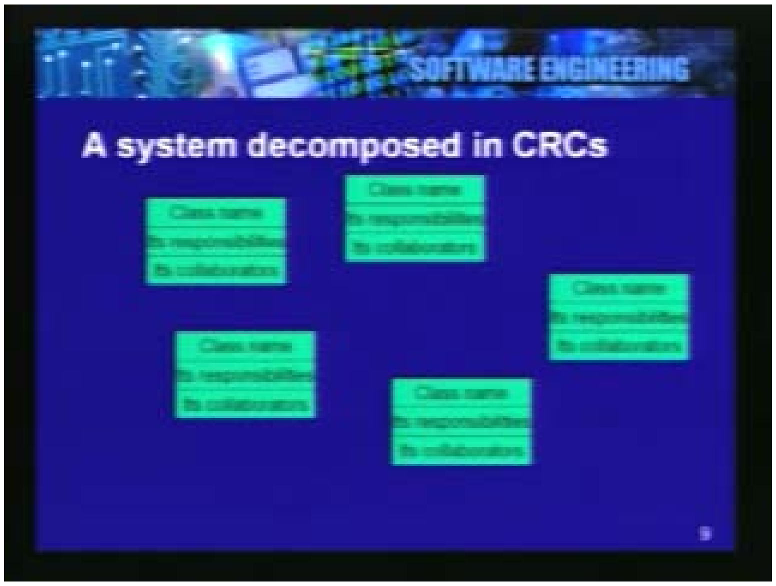
Collaborators are other CRC cards
Collaborations will be their responsibilities

For example, you need to list what is the collaboration? For what responsibilities you need these collaborators? Etc. The collaborators are all always the other CRC card. This is one card and this is called CRC card. It is a palm-top card you can make a small chip on which you can write class name, its responsibilities and its collaborators and also if you can talk of collaborations for which you want the collaborators.

This idea was initially projected as a method to teach novices, to teach about object oriented thinking. So when you break down a system into objects, you can use these CRC cards and you can break down a system into many classes. You can give one card to each team of say two members and they design these classes. Through which they give the classes the name, they talk about its responsibilities and also they identify the collaborators of a given class and collaborations. Every team is designing single class and then all these classes are working together, because every team also identify collaborators and over few iteration the system could get design.

When you look at a system, it consist of many such classes and they collaborate with each other and then once you identify these classes, you can go on to further detailing of each classes.

(Refer Slide Time: 16:00)

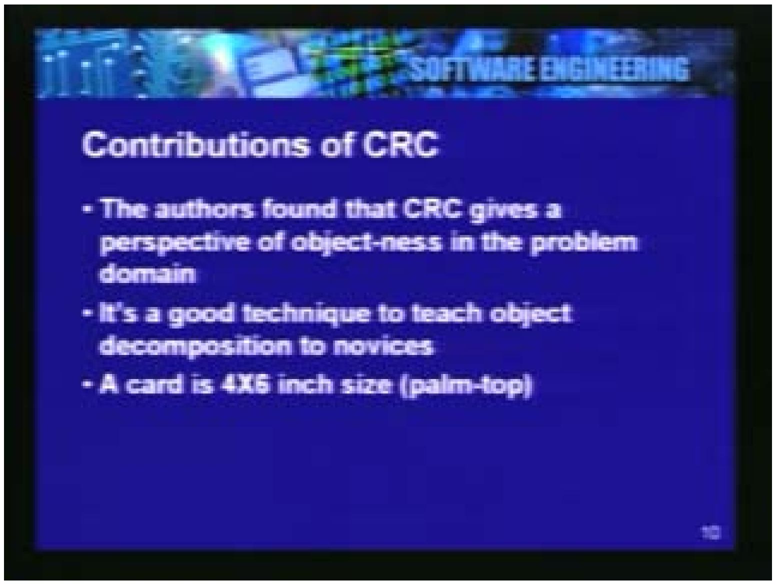

SOFTWARE ENGINEERING

A system decomposed in CRCs

9

The authors found that CRC gives the perspective of object ness in the problem domain. It is a good technique to teach object oriented decomposition to novices and a card could be of 4X6 inch size.

(Refer Slide Time: 16:13)


SOFTWARE ENGINEERING

Contributions of CRC

- The authors found that CRC gives a perspective of object-ness in the problem domain
- It's a good technique to teach object decomposition to novices
- A card is 4X6 inch size (palm-top)

10

This is a nice technique which can be practiced in class rooms. I often use this technique and ask the students to break them into many teams and take a problem. Then an individual team can design a single class at interface level mainly the responsibilities at this time. You do not think of implementation, lower level implementation or the actual code. But you are mainly looking at the interface of the classes and trying to find out what are the collaborators. Over a few iteration the systems gets refined. Now we will take a look at an interesting method of designing classes or even this can be also applied at system level. But now we are going to look at mainly class level design.

(Refer Slide Time: 17:07)

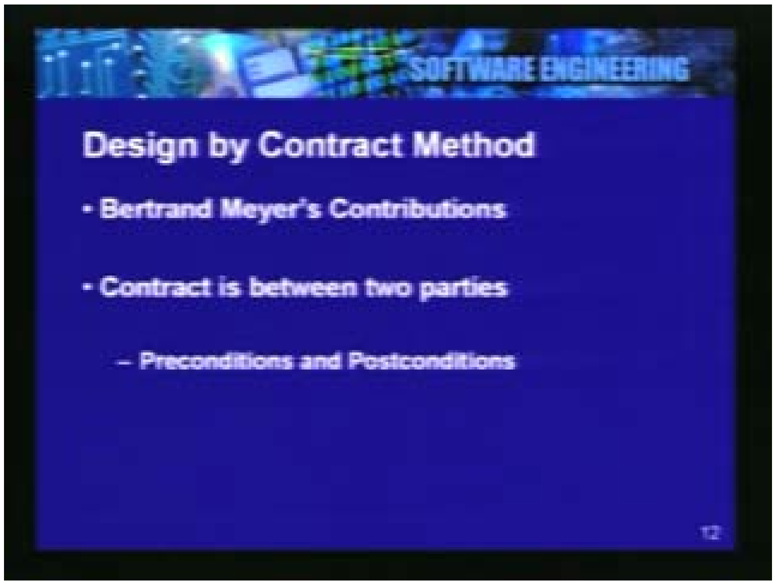
SOFTWARE ENGINEERING

Bertrand Meyer' Design-by-Contract

11

This is Bertrand Meyer' Design-by-Contract. This is our interesting method. Meyers contributions are that he discuss his method and provided this through a programming language support in Eiffel. Eiffel is some good programming language. Then it was designed from software engineering point of view. So this design by contract can be applied even at implementation level through Eiffel. Eiffel can also be used to describe it at higher level and then seamlessly you can migrate towards implementation from design. What is this design by contract method? What are contracts? Contract is a very important concept in object orientation. When we look at the interface, the interface is guaranteeing some contract to its users. Contract is always between two parties, the supplier and the consumer.

(Refer Slide Time: 17:19)


SOFTWARE ENGINEERING

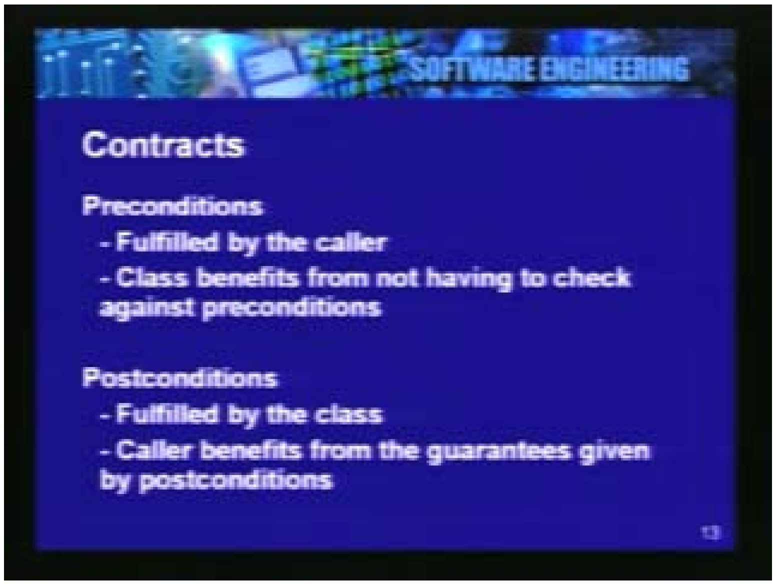
Design by Contract Method

- Bertrand Meyer's Contributions
- Contract is between two parties
  - Preconditions and Postconditions

12

The consumers are callers, they call member functions on a given class and supplier is a class. So in a contract there are pre conditions. For example, if you provide these resources, I will guarantee something. Pre conditions are to be guaranteed by the consumer and post conditions are to be provided or to be guaranteed by the supplier. So the contract can be mainly expressed in terms of pre conditions and post conditions. Now we will see how the design classes from these contracts point of view.

(Refer Slide Time: 18:50)


SOFTWARE ENGINEERING

Contracts

Preconditions
- Fulfilled by the caller
- Class benefits from not having to check against preconditions

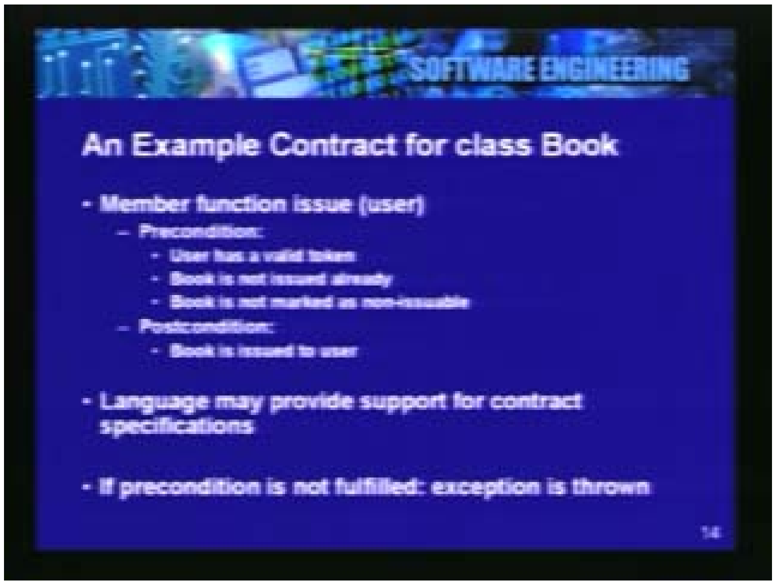
Postconditions
- Fulfilled by the class
- Caller benefits from the guarantees given by postconditions

13

We are just listing here that preconditions are fulfilled by the caller. Class benefits from not having to check against preconditions. You can say that in order to invoke a member function on the class, its parameters must satisfy these conditions. If they do not satisfy, we will not invoke or we will not actually dispatch the method on an instance of that class whichever has been selected for invocation. The implementation or the class benefits from not having to check against these preconditions. The supplier benefits from precondition. Preconditions have to be guaranteed by the consumer or the caller in this case. And post conditions are to be fulfilled by the class. They are to be guaranteed by the class.

Who is the beneficiary, who benefits from post conditions? The caller benefits from the guarantees given by the post conditions. The operation is completed and the post conditions specify the guarantees that are given after the operation is completed. Unless a precondition is satisfied the member function is not dispatched to a method on the selected object. If a member function is dispatched, the code does not have to check against the parameters because the preconditions would have taken care of it. In that case the code is the core functional logic. You directly operate on the local state and complete the implementation of the method selected. After that the post conditions are guaranteed by the class and they are specified by the post condition classes. Look at an example contract for a class book and we will pick up a member function called issue.

(Refer Slide Time: 20:50)


SOFTWARE ENGINEERING

An Example Contract for class Book

- Member function issue (user)
  - Precondition:
    - User has a valid token
    - Book is not issued already
    - Book is not marked as non-issuable
  - Postcondition:
    - Book is issued to user
- Language may provide support for contract specifications
- If precondition is not fulfilled: exception is thrown

14

You can issue a book to a given user. User is a parameter.
What are the preconditions?

- The user must have a valid token. Otherwise we will not complete the issue. Issue method will not issue the book.
- The book is not issued already, then this method can be dispatched.
- And the book is not marked as non issue able. This is also precondition.

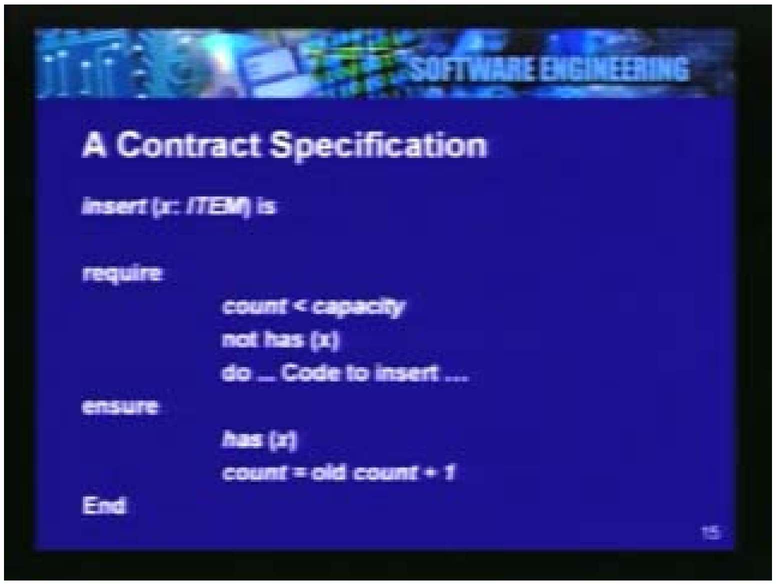
If these are satisfied, then the actual core business logic of this issue method can be dispatched. These constraints which must be satisfied before the member function is invoked are called preconditions. Then you have post condition which are the guarantees given by this member function after it is dispatched and completed that the book is issued to the user. Unless the preconditions are satisfied, this method is not dispatched. Issue is not done. Otherwise it is directly returned to the caller on failure of any of these preconditions. If the precondition is accepted, then post condition is guaranteed. That is, if the preconditions are acceptable then supplier must guarantee the contract to the consumer.

If the preconditions are not acceptable then the supplier should not undertake the implementation of the contract. This is the concept which you see in real life. Now the same is also applicable in design of software entities. Bertrand Meyer thought about this and he has written a few articles in his books on object oriented software development. And also the book on Eiffel you can see more description and more examples and the actual programming language constructs for describing these contracts and then seamlessly moving them on to implementation. So a language may provide support for contract specification.

For example, Eiffel provides direct support. If there is no support in the language for contracts, then you can use assertions in your programs. On paper, you can first design the classes and you can then map that to implementations using assertions. If there is no support for assertions in the programming language, then you write that code explicitly and separate it from business logic. It will be in the same member function code, but you have marked it separately. It is easy for the designer at first to provide the precondition, post condition code and the implementation can actually implement the code.

And if the preconditions are not satisfied exceptions can be thrown. If the post conditions are not satisfied it means that the implementations could be buggy. So this is an important method and it can be used at class levels designs by contract. Here is one example contract specification. Insert method on a set, insert ‘x’ an item and you could provide preconditions. This is similar to Eiffel. You can describe the precondition class as say it requires. The require keyword is used to specify the preconditions. The count is less than the capacity, and item ‘x’ is not in the object or in the set and the actual implementation code can be specified. When you are designing, this implementation code is not provided, but when you are implementing this code can be stuffed, the preconditions and post conditions. The post conditions described by ‘ensure’ class. It ensures that it has x and count is equal to old count plus 1.

(Refer Slide Time: 24:00)



A Contract Specification

```
insert (x: ITEM) is

require
            count < capacity
            not has (x)
            do ... Code to insert ....
ensure
            has (x)
            count = old count + 1
End
```

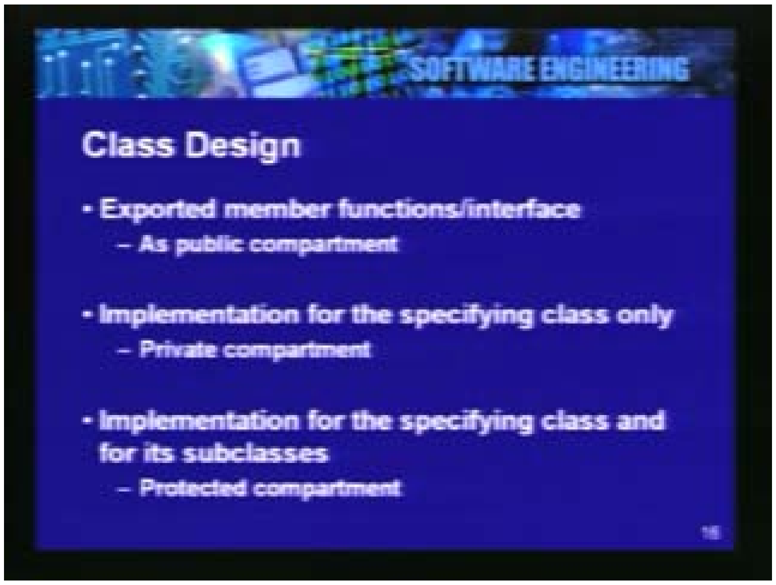
15

The require keyword is used to specify the preconditions. The count is less than the capacity, and item ‘x’ is not in the object or in the set and the actual implementation code can be specified. When you are designing, this implementation code is not provided, but when you are implementing this code can be stuffed, the preconditions and post conditions. The post conditions described by ‘ensure’ class. It ensures that it has x and count is equal to old count plus 1.

So these can be directly used in implementation. The class can be compiled without the code first when you design. In your design, you just translate into this template of contracts for every member function and then when the implementer implements this class, which is designed, using this ‘design by a contract’, he stuffs in the implementation there. The exceptions etc can be thrown by the programming language. If you do not have the support in the language use alternative constructs which are described few minutes ago.

Class design works for exported member functions, an interface, as public compartment, implementations for the specifying class only and implementation for this specifying class and for its subclasses. Till now we looked at exported member functions and the interface which is public compartment. The design and we looked at it design by contract and we looked at the CRC method of decomposing classes and looking at the responsibilities of the classes and collaborators. And then another two important aspects in class design when you want a detail or when you want to give the design of the implementation before the actual code is written. They are the implementation for the given class and implementations for the specifying the given class and for its sub classes.

(Refer Slide Time: 25:19)


SOFTWARE ENGINEERING

Class Design

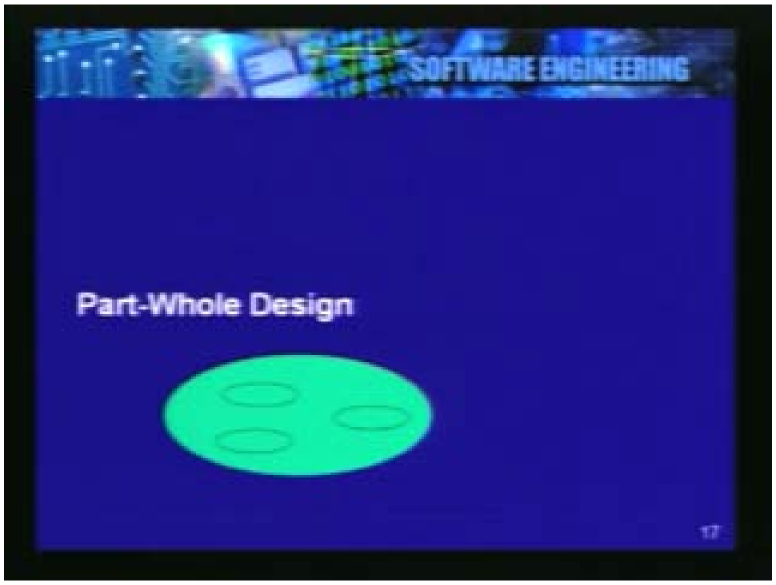
- Exported member functions/interface
  - As public compartment
- Implementation for the specifying class only
  - Private compartment
- Implementation for the specifying class and for its subclasses
  - Protected compartment

16

When you are designing classes there are two kinds of implementation again. One is for the methods of the class itself and other part of implementation can be used by the sub classes of the given class. So we have these private methods and protected methods or private attributes also and protected attributes also. So private and protected gives you two kinds of implementation one is for the specifying class only, that are private and other is for specifying class as well as for its subclasses. And the public compartment is the interface which is accessible for the external environment or the callers of the class.

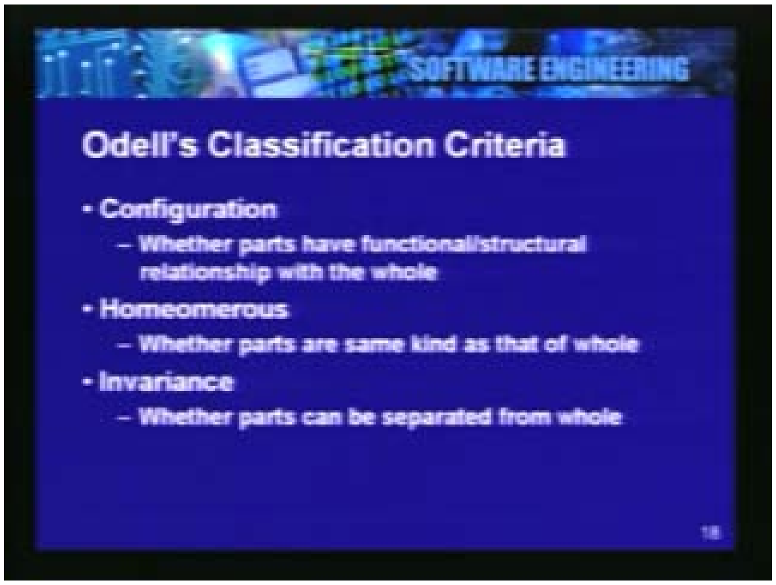
That summarized class level designs. Now we will look at another aspect of class design which is part-whole hierarchy. So you could have objects inside objects. When you are designing classes, you have to take into account this break up of class into many parts. This picture summarizes what we are going to talk about. You have objects inside objects and when you have such entities, you can describe them using classes. There has been a lot of discussion in literature about this part-whole hierarchies and the part whole relation as such. But we will look at one specific description given by Odell.

(Refer Slide Time: 27:00)



Part-Whole Design

(Refer Slide Time: 27:32)



Odell's Classification Criteria

- Configuration
  - Whether parts have functional/structural relationship with the whole
- Homeomerous
  - Whether parts are same kind as that of whole
- Invariance
  - Whether parts can be separated from whole

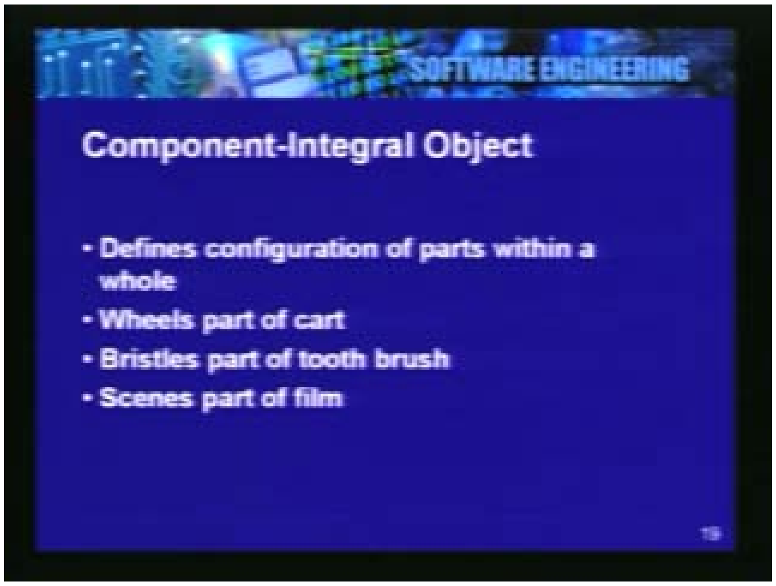
Odell applied these criteria to classify different kinds of part-whole relation. There are many kinds of part whole relation. For example we can say that I am part of such and such department or I am part of [‘unclear word’ 28:00] or whatever this such and such organization or you can say that lamp is part of projector. When one uses this keyword called part of, I am part of this country or a state is part of a country or the first hour part of a concert or first ten minutes which are part of a given lecture, the last five minutes which are part of given lecture, what do you do?

So we are using this part-whole relation where we are using this term called part of. But there are different kinds of part of relations and then there are consequences on the design and implementation. How you represent them in design and how you implement them? What are the semantics of a given part whole relation? When you delete the whole are all the parts deleted? When you invoke a method on a whole is that method also dispatched on to all its parts? These are various consequences. You have to find out what kind of part-whole relations are you trying to design. If you represent it properly, then you will be able to seamlessly implement it and also if you have a powerful notation to describe all different kinds of part-whole relation.

So Odell described different kinds of part-whole relations through these three parameters. First is 'Configuration': Whether parts have functional or structural relationship with the whole. Say I have a bag. In that bag there must be many items. Now it is just a collection and the items may not have any relation amongst themselves. But if you have a car and its parts, the parts are related they depend on each other. Their input by outputs are connected. So configuration is one important parameter to distinguish, can be used to distinguish different part-whole relations.

Another property is 'Homeomerous': Whether parts are of same kind of as that of the whole. You take slice of bread; the slice has the same properties of the bread, but it is a part of the whole bread. Then another property is 'Invariance': Whether parts can be separated from the whole. These are important parameters which can be used to differentiate or to distinguish between different kinds of part-whole relationship. We describe different kinds of part-whole relations and we will look in to six such kinds.
But researchers have found many different kinds, say UML give you two main variations of part-whole. You can capture them using aggregation, composition. But now we will look at these six different kinds of part-whole relation to give you an overview of how part-whole relation can be used in class level design.

(Refer Slide Time: 30:28)


SOFTWARE ENGINEERING

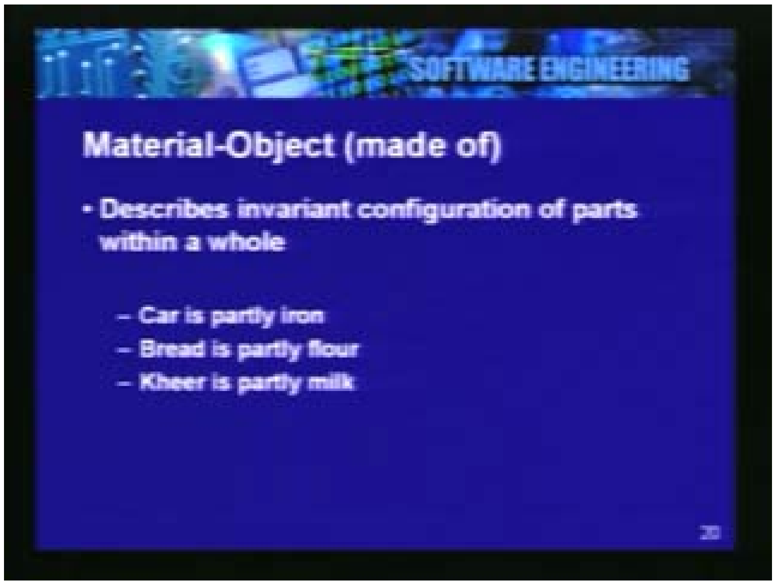
Component-Integral Object

- Defines configuration of parts within a whole
- Wheels part of cart
- Bristles part of tooth brush
- Scenes part of film

19

‘Component-integral object’: It defines configuration of parts within a whole. For example wheels are part of a cart or bristles part of tooth brush or scenes part of a film. Scenes they have a sequence, so the configuration has the structure. The wheels are at a particular place the configuration is important. The bristles part of tooth brush is in its configuration. Another example is ‘Material-object (made of)’: It describes invariant configuration of part within a whole. Car is made out of partly iron, but you cannot take out iron from the car.

(Refer Slide Time: 31:30)


SOFTWARE ENGINEERING

Material-Object (made of)

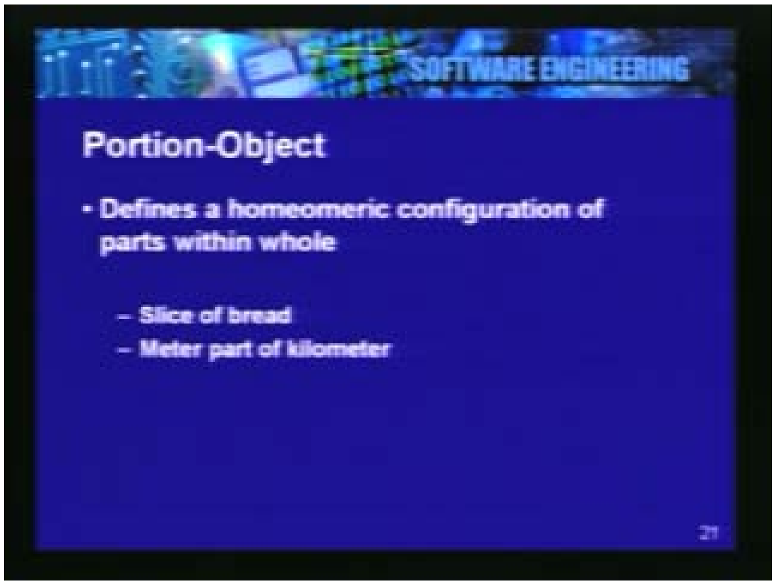
- Describes invariant configuration of parts within a whole
  - Car is partly iron
  - Bread is partly flour
  - Kheer is partly milk

20

Bread is partly flour, but you cannot take flour from the bread or kheer is partly milk if you take the milk from kheer, then it is no more kheer. So this is ‘Material-object (made of)’ relation. Then you have ‘Portion-object’: It defines a homeomeric configuration of parts within a whole. Parts are also like the whole. So when you say you have a homeomeric configuration, how will you translate it into your class if you have classes?

If there is homeomeric configuration, you are going to use inheritance over there. Examples: 1. Slice of bread: Slice of bread also has similar properties as that of bread. 2. Meter part of a kilometer. Then you have ‘Place-Area relation’; which is homeomeric and invariant configurations of parts within a whole.

(Refer Slide Time: 31:55)

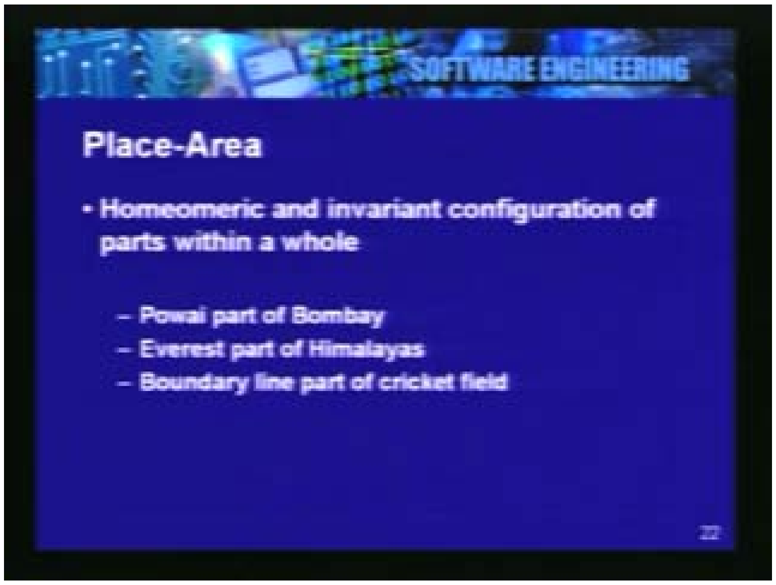

SOFTWARE ENGINEERING

Portion-Object

- Defines a homeomeric configuration of parts within whole
  - Slice of bread
  - Meter part of kilometer

21

(Refer Slide Time: 32:25)


SOFTWARE ENGINEERING

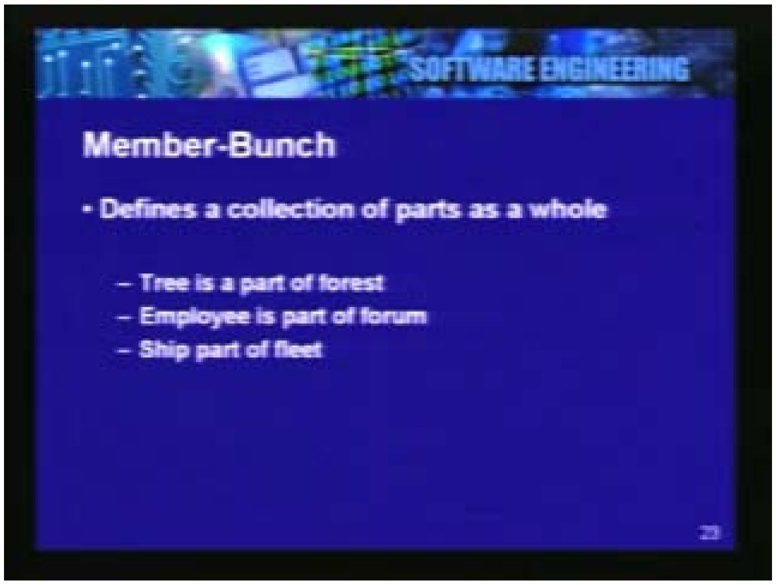
Place-Area

- Homeomeric and invariant configuration of parts within a whole
  - Powai part of Bombay
  - Everest part of Himalayas
  - Boundary line part of cricket field

22

Examples: Powai part of Bombay or Everest part of Himalayas or boundary line part of cricket field. If you are talking about a place and an area, you cannot remove the place from the area, you cannot separate it out. Then you have ‘Member-Bunch’ relation: It defines a collection of parts as a whole.

(Refer Slide Time: 32:47)


SOFTWARE ENGINEERING

Member-Bunch

- Defines a collection of parts as a whole
  - Tree is a part of forest
  - Employee is part of forum
  - Ship part of fleet

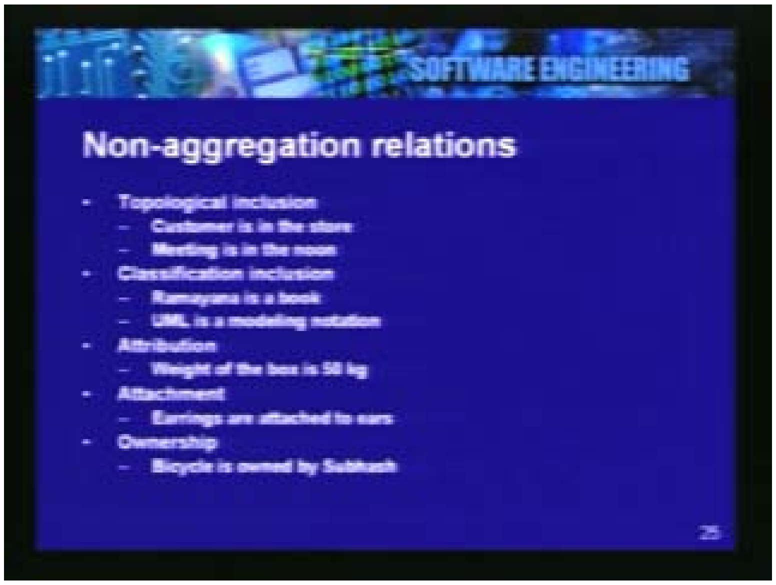
23

Examples: Tree is a part of forest. Employee is a part of a forum. Ship is part of a fleet or a person is part of an organization or is a member of a given club. Then you have ‘Member-Partnership’: Member of relation is a different relation and there is a slightly different member of relation where the members are related to each other. You cannot take out the members defines an invariant collection of parts as a whole. Laurel part of Laurel and Hardy or Puriya is partner in Puriya-Dhanashri which is a classical Hindustani raga, where if you take out one, then the raga breaks down. So the members in the member-partner are also related to each other. It is an invariant collection of parts in a whole.

What are ‘Non-aggregation relations’?

- Topological inclusion: For example, customer in the store.
  If you have such a requirement then you cannot design for a part-whole relation. And another example for topological inclusion is, meeting is in the noon.
- Classification inclusion: Ramayana is a book or UML is a modeling notation.
  This is not part-whole, so this is classification.
- Attribution: Weight of the box is 50 kg.
- Attachment: Earrings are attached to ears, so this is mainly attachment or association.
- Ownership: Bicycle is owned by Subhash. Ownership is not part-whole. Bicycle is not part of Subhash.

(Refer Slide Time: 33:45)



SOFTWARE ENGINEERING

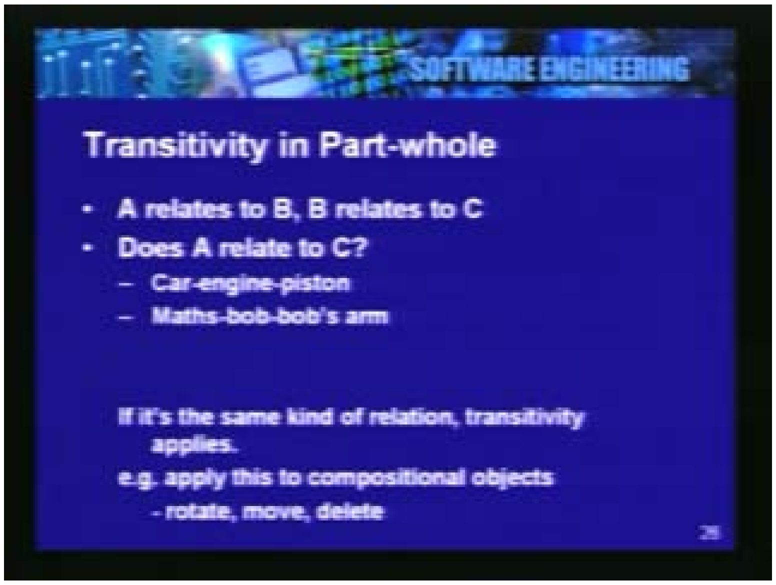
Non-aggregation relations

- Topological inclusion
  - Customer is in the store
  - Meeting is in the noon
- Classification inclusion
  - Ramayana is a book
  - UML is a modeling notation
- Attribution
  - Weight of the box is 50 kg
- Attachment
  - Earrings are attached to ears
- Ownership
  - Bicycle is owned by Subhash

25

These are different relations which sometimes might get confused for part-whole relation. When one decomposes a given class into part-whole, then we have to exclude some non part-whole relations. Then there is an important aspect in part-whole relation called transitivity. A relates B and B relates to C. Does A relate to C? For example, car-engine-piston; engine is part of car and piston is part of engine. So is piston also part of car? Then you can say Bob is in math class and Bob arm is part of Bob. So is Bob's arm part of math class?

(Refer Slide Time: 34:55)



SOFTWARE ENGINEERING

Transitivity in Part-whole

- A relates to B, B relates to C
- Does A relate to C?
  - Car-engine-piston
  - Maths-bob-bob's arm

If it's the same kind of relation, transitivity applies.

e.g. apply this to compositional objects

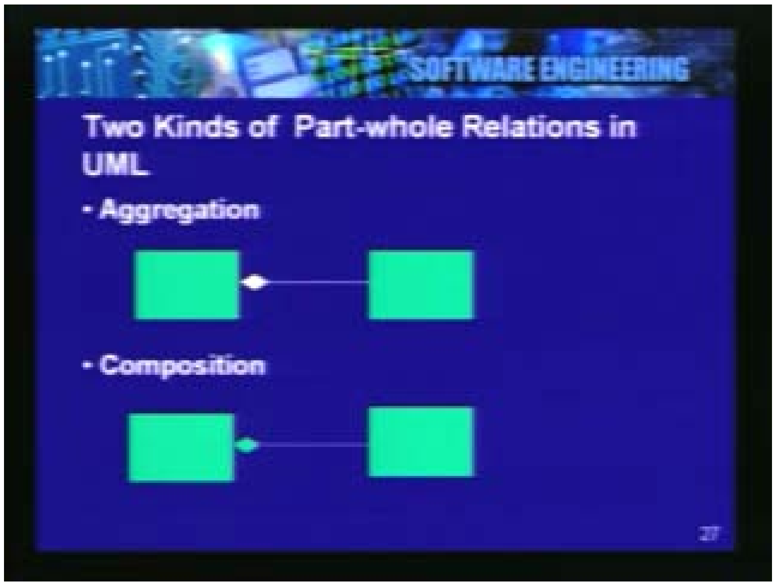
- rotate, move, delete

26

If you apply this criteria that if you find out these different kinds of relation, say math and Bob there is one kind of part-whole relation. Bob and Bob's arm have the different kind of part-whole relation. If that is the case, then transitivity will not apply. When you are designing your classes, the same is also applicable. Say for example if you have compositional objects in your document hierarchy, then the leaf documents or the composite documents are also documents and composites are whole for many internal documents. Then the operations that you invoke on the high level composites also can be propagated to the lower level and the transitivity applies.

For example, rotate when you rotate a shape all its components are also rotated internally. If you move it, all the components are moved. If you delete the high level shape, then the lower internal shapes are also deleted if you have a group. UML gives you this notation for capturing mainly two kinds of part whole relationship. One is called aggregation and another is called composition. In the below diagram, top one is shown with the hollow diamond with white color. And the bottom one is the filled diamond, it is dark filled diamond

(Refer Slide Time: 36:20)


SOFTWARE ENGINEERING

Two Kinds of Part-whole Relations in UML

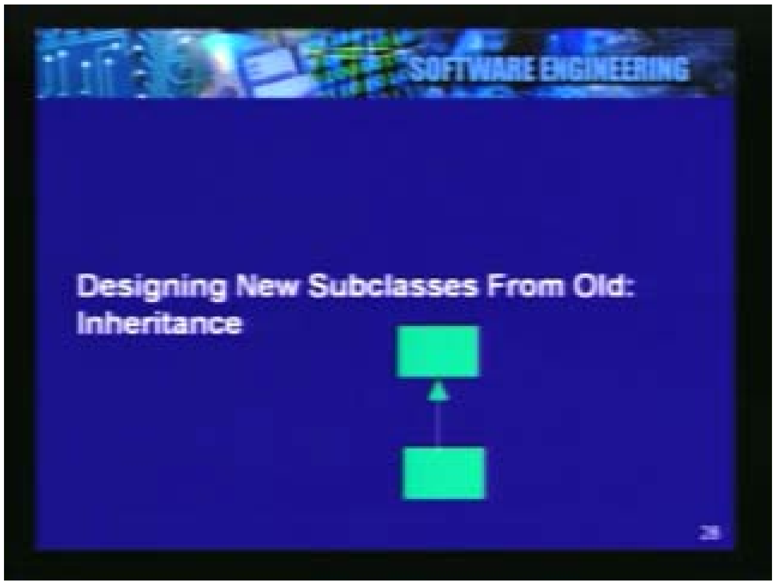
- Aggregation
- Composition

27

If you replace this by a black and white picture, then the distinction will be clear. When you are hid out whether it is a part whole relation, where it is a stronger relation where there is propagation of member functions, then you are going to use composition otherwise you can use aggregation. In aggregation, the part-whole is little bit weaker. If you are having relations like member function or a is a member of b and if you do not know whether there is strictly very configurational properties amongst the parts and they are all required for the functionality of whole component, if you do not have a component integral object, if you do not have a stronger part-whole relation, then you can go for a weaker part-whole relationships.

So you can say one is the weaker another is stronger part-whole relationship. These are the two main categories which are recognized by UML modeling notation. In this way we can use our part whole relations and design the part-whole hierarchies of classes. Now we will look at another aspect of class level design that is inheritance.

(Refer Slide Time: 37:55)

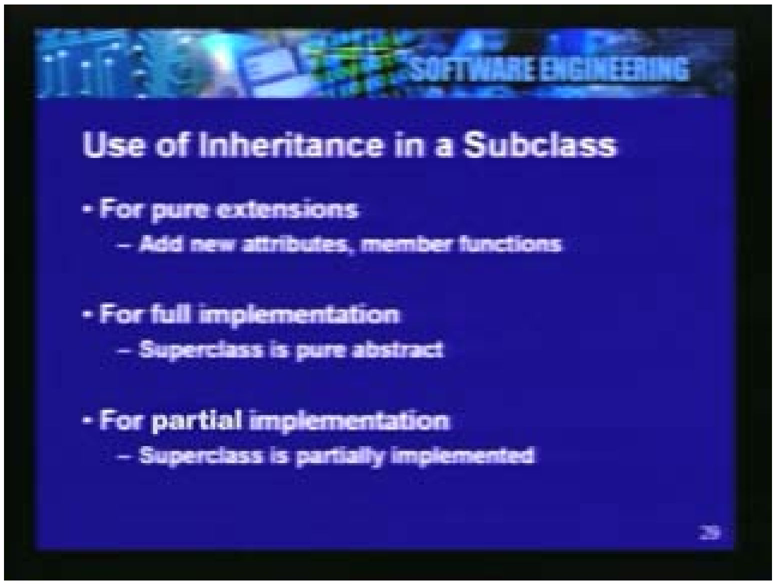


Designing New Subclasses From Old: Inheritance

We first saw individual classes, we looked at the CRC method, then we looked at design by contract method, then how to design classes from preconditions and post conditions point of view and from contracts point of view. Then we looked at how to break classes into part-whole or what are different kinds of part-whole relationship. Now we will look at this inheritance relation which is also very important relation in class level design. How to design new subclasses from old? What are the different kinds of uses of inheritance in subclasses? How do subclasses use inheritance?

If we first look at these different kinds of uses of inheritance, then we will apply these different kinds to our different class level design requirements. You can use sub classes as pure extensions to super classes. You can add new attributes; you can add new member functions. So your super classes may be concrete classes which we had described earlier. You can then add new attributes and new member functions to those super classes. You can use sub classes as full implementation. This is applicable when you have a pure abstract class as a super class.

(Refer Slide Time: 38:28)


SOFTWARE ENGINEERING

Use of Inheritance in a Subclass

- For pure extensions
  - Add new attributes, member functions
- For full implementation
  - Superclass is pure abstract
- For partial implementation
  - Superclass is partially implemented

29

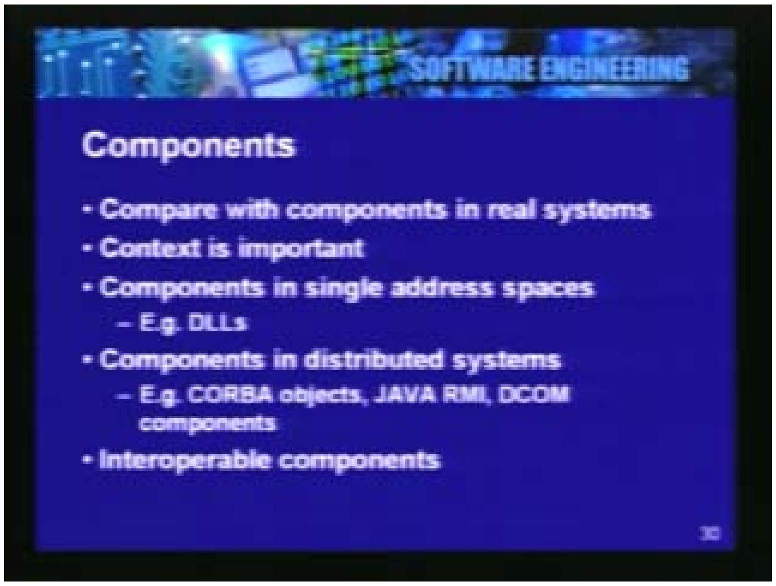
And thirdly you could use sub classes as partially implementation. These are applicable when your super classes are partially implemented. So they are partly concrete, partly abstract. These are again applications at same with peer extensions can be used where you have pure concrete library classes. Full implementations have to be used when you have intense hierarchies with abstract classes or they also have call interfaces mainly in component technologies. And you can use subclasses as partial implementers in frameworks. Inheritance for pure extension, inheritance for full implementation and inheritance for partial implementation: Mainly these three kinds of inheritance or these three kinds of subclasses can be used in class level designs with inheritance.

There is one aspect of inheritance which I have not listed here but which is also sometimes used, that is called mixings. Use inheritance to actually represent part-whole. You have an object as many parts and the parts can be used as sub classes. In this case, the whole is sub class which uses inheritance and using inheritance, one instance of each of those super classes becomes part of the object or the memory map of the object of the subclass. This is called mixing. But in this case one has to remember that the abstraction of the super classes is not exported by the sub class. That means the inheritance cannot be public inheritance. So the fact that the subclass is using the super class's parts only should be reflected in our implementation through private inheritance.

You have to also note this in your design that the designing your hierarchy or when you are designing your subclass as a mixing. Inheritance has to be private in such a case, where the parts which are modeled as super classes, have a different abstraction that of the whole. These are various important aspects of inheritance. We are not going to go into much detail of inheritance and then implementation levels, because we are mainly focusing on design aspects.

After the inheritance aspect we now go on to components. Till now we have explored various aspects of class level design, individual classes design by contract, part-whole relation, CRC and aspects of inheritance: how to design sub classes.

(Refer Slide Time: 40:04)


SOFTWARE ENGINEERING

Components

- Compare with components in real systems
- Context is important
- Components in single address spaces
  - E.g. DLLs
- Components in distributed systems
  - E.g. CORBA objects, JAVA RMI, DCOM components
- Interoperable components

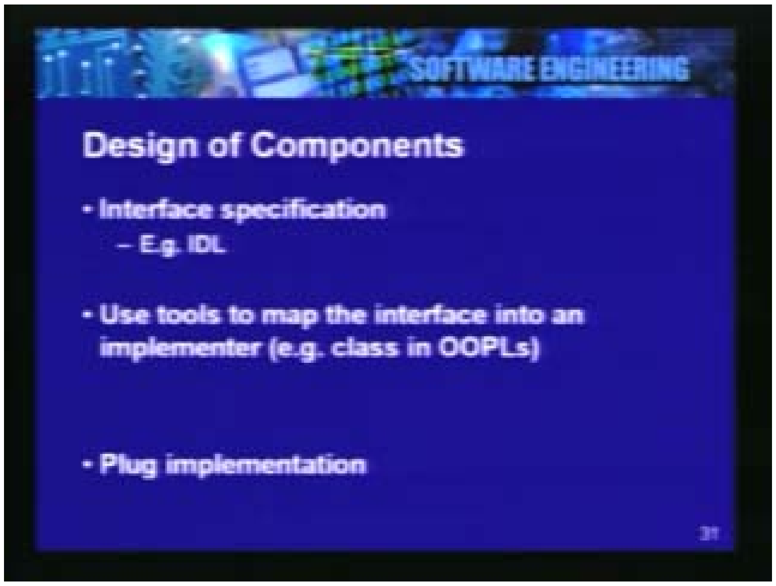
30

Now we will look at components. What are components and how are they different from classes? Compare this term with components in real systems. What happens to components in real system? For example you have a circuit board, you have a circuit and there are many components. You can remove a component; you can plug another component which is of the same kind and then the circuit will work. So in a given context, a particular entity can be a component. So context is very important. For example, components in single address space, with in a process, dynamically linked libraries. When the process is executing, you can change implementation, can change DLL and you can read it in. DLL is component in single address space.

If you talk of components in distributed systems, then you could have examples in CORBA, Java RMI, DCOM and so on. Then your context might need that your entities must be component in an interoperable system. For example I have a component which is implemented in Cplusplus and it works in a system where there are other languages like Java, Eiffel and so on. Will these components implemented in different languages be able to communicate with each other? That is called interoperability, because this is a communication among heterogeneous environments. When you talk of a system called components, your components or components in a given context and what is that context is very important. Now we will focus on this component. We will look at this details of what exactly the component and how it is different from a class.

When we talk of components, when we design components, we mainly focus on the interfaces of the component and the implementation can be in any given language or in any given context.

(Refer Slide Time: 44:30)



SOFTWARE ENGINEERING

Design of Components

- Interface specification
  - E.g. IDL
- Use tools to map the interface into an implementer (e.g. class in OOPLs)
- Plug implementation

31

Interface is very important for component. When we talk of component orientation you think of interfaces first. Class as a term provides the separation of interface and implementation. Component is a concept where you are talking of changeability, where you are talking of removing something and placing something else there. That aspect is very important. So component is very heavily used terms in technological context or in implementation context. When you are implementing, when you are designing your components, you first look at interface specification and when you are implementing them, you are looking at the programming language choice and so on.

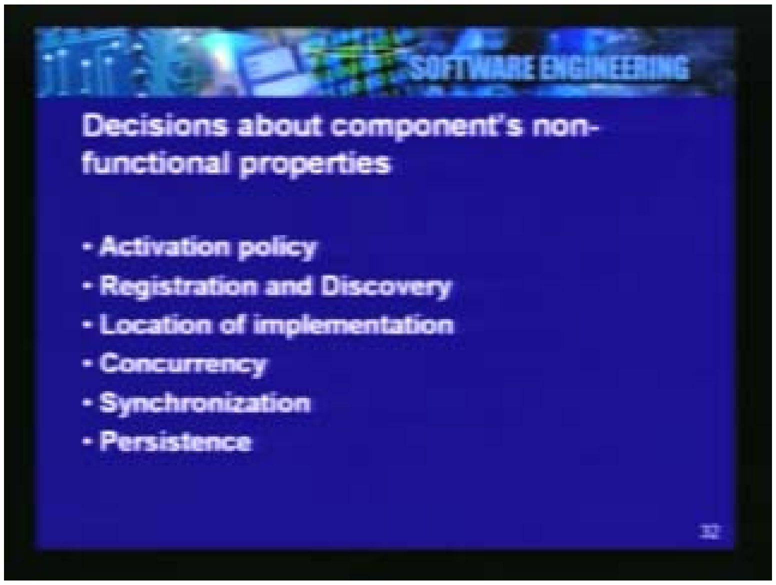
So systems component system such as CORBA gives you an interface specification language called ‘Interface Description Language’ - IDL. Using the IDL you can define these interfaces of your components and then you have to use tools to map this interface into an implementer and then into the implementer you have to plug the implementation. These are the three main steps. When you design the components, you first think of the interface and then when you want to implement the interface, when you want to provide the components implementation, then you map this interface into the implementer which is mainly a skeleton in a given language.

You might say that you have an account component which has the interface deposit, withdraw, and balance and so on. And now you want to implement the component in Cplusplus, in Java or in Eiffel, you can implement in any language.

The client implemented in any of these languages will be able to invoke these member functions on your component in spite of its implementation, in spite of you implementing the component in different programming language, because you are thinking of the component from the interface point of view. So standardization is very important in component orientation, in today's component technology. Interface description language has to be used. Different component technologies are there in existence and they have different interface description languages and you describe them. And then implement the component through the given implementer, generate it by the tool, or can also be generated by hand, but there are the tools which are available with almost all technologies.

It can generate skeletons from a given description and then in the skeleton you can plug implementation. The implementations can be provided through different mechanism such as inheritance or delegation and so on. But we are not going to look at implementation aspect. In a design level, you are mainly focusing on interfaces of the components and we know that components can be implemented in different languages. And in a context where interoperability is important, mechanisms are provided by the component platform. And the component technologies are for the interoperability aspect. The other decisions are about component's non functional properties.

(Refer Slide Time: 48:28)



SOFTWARE ENGINEERING

Decisions about component's non-functional properties

- Activation policy
- Registration and Discovery
- Location of implementation
- Concurrency
- Synchronization
- Persistence

32

These are also to be taken into account, when you design components. What are the activation policies? How do you start the component? Is the components started? So where is your system started? You are talking of actually instantiating or actually making the component active, so that it can execute the code against given request. What are the activation policies? As I started every time, a member function is invoked and then brought down as soon as the function is completed. Or it is started once and it shutdown by the end of the day. So you have to design for the activation policies of a component.

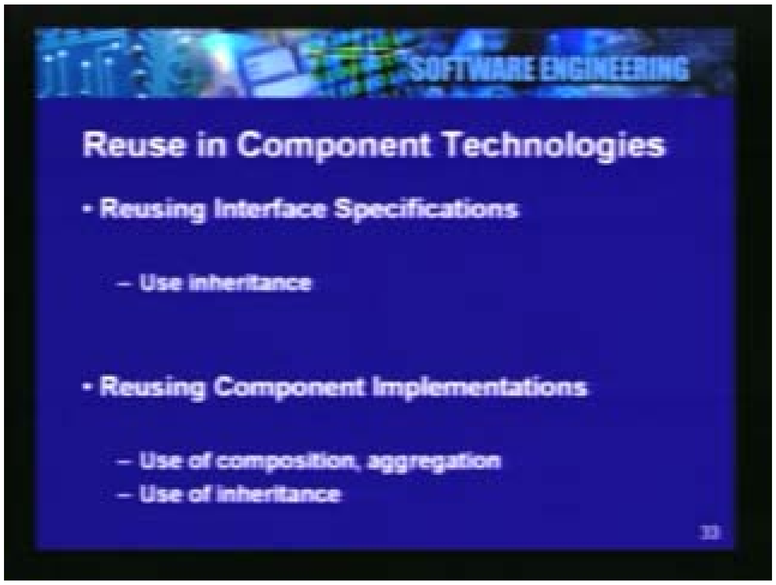
Registration and discovery: In a distributed system when there is an object which is available in the network of machines which is distributed, you need this registry and you need the ability of discovery. So that clients can discover these load objects through this registries and then invoke them. If component programming is made available for an object oriented environment you can use them as objects. Therefore, I am using this term called object in a distributed environment. You will come across this term called distributed objects. This term is also used to refer to component programming in distributed environment using object oriented programming language. For example in CORBA you can use Cplusplus, Java, Eiffel, or Smalltalk and interoperate.

These are object oriented programming languages which you can use in CORBA. You have java environment for making available this distributed object platform in java and also have similar technologies in Microsoft platform. For example you have com by dcom, dot net. These are some of the major component programming platforms and they also make possible this distributed object programming. And component orientation is also possible in object oriented domain in distributed environment. Then we are talking of registration and discovery this is an important decision one has to make: How many registries you are going to have or how are you going to organize your components?

Location and implementation: Where are they located? This sometimes closely related to activation policy? Then you have other decision such as concurrency within a component and synchronization within methods, amongst the methods of a given component. Say you may want to lock a component invoked when you are in the midst of the method. Persistence of the component: Is the state persistent when you bring down the component from the state it was before. When you bring down the machine is the component still persistent? These are some of the design decisions, one has to make when designing components.

How do you reuse components? We had seen inheritance relation in object technologies when you talk of components. Since we are decoupling interface from implementation, saying that interface description can be in one language and implementation can be of different programming language, you can reuse these two separately. You can reuse interfaces using inheritance and describe new component interfaces using old component interface. You can reuse component implementations using implementation specific reuse mechanisms. If you have Cplusplus implementation for a component description, you can use inheritance; you can use composition or aggregation and so on.

(Refer Slide Time: 52:16)


SOFTWARE ENGINEERING

Reuse in Component Technologies

- Reusing Interface Specifications
  - Use inheritance
- Reusing Component Implementations
  - Use of composition, aggregation
  - Use of inheritance

33

So reuse can be applied at interface levels separately and implementation level separately. This is about component oriented design, when you are designing component oriented systems. In summary, we had seen class level designs, we had seen individual classes, we had seen different compartments in a class, and then we had seen the method of design: the CRC method, we had seen design by contract, then we also looked at part-whole relation, we looked at inheritance; different kinds of inheritance and how to design sub classes and we looked at components: what are components? How they are different from classes, the design of components - interface is what very important in the component and then we looked at reuse of components such as interfaces separately and implementation separately.