

Software Engineering
Prof. Umesh Bellur
Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 13
Producing Quality Software- Introduction to
Software quality, Abstraction, Reuse, Modularity etc.

Welcome back. Today we are going to talk about software quality and this is very different from what we have already discussed in this course as part of quality management process of producing software. There are two things we should consider, the process and the product. Quality measurements, quality monitoring and quality improvement can be applied to both of these. For example, you can make process improvements, CMM metrics suite which exists that actually covers, how you measure the process efficiencies, how you improve process efficiencies and so on. There is also an aspect to be considered in terms of producing quality software. That aspect is that of the quality of software itself inherent to the design techniques that we use and the kind of abstraction that we use for building the software. This is going to be focus of today's discussion.

(Refer Slide Time: 02:00)



The slide features a blue background with a header banner showing circuitry and the text 'SOFTWARE ENGINEERING'. Below the banner, the title 'Software construction then and now' is displayed. A table compares various factors in 1965 and 2000. The table has three columns: 'Factor', '1965', and '2000'. The rows include 'Cost of Computer (256K)', 'Cost of programmer', and 'Software Projects'. The 'Software Projects' row is further divided into three sub-rows: 'Late', 'Buggy', and 'Inflexible'.

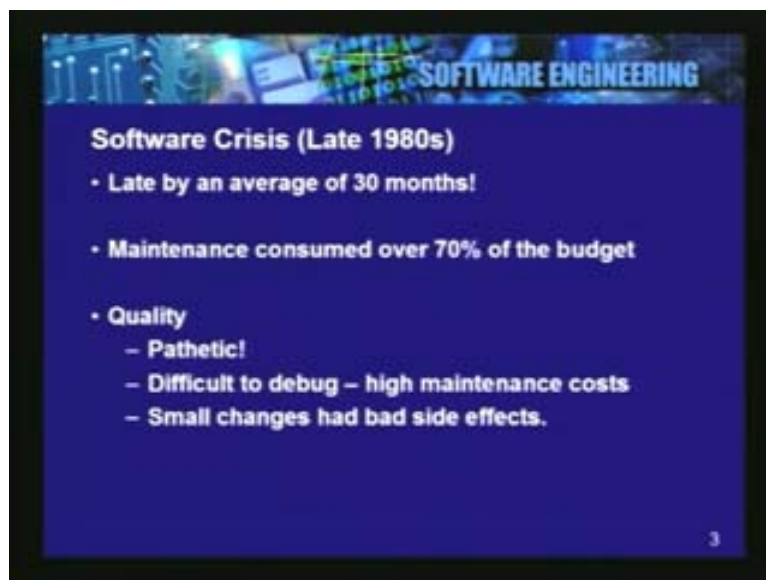
Factor	1965	2000
Cost of Computer (256K)	\$750000	\$100
Cost of programmer	\$2500/year	\$75000/year
Software Projects	Late	Late
	Buggy	Buggy
	Inflexible	Inflexible

2

All of this is motivated by taking a perspective on the changes that have happened in software and also the statistics about software development project themselves over the last twenty to thirty years. If you take a look at this graph or this chart, what it is basically trying to show is that, even though lot of things have changed about computers and software, there are some characteristics of software projects that have remained constant.

Basically, 35 or 40 years ago, software projects are to be constantly late. They were never delivered on time or never under budget or on budget. They were always over budget. They were very buggy in the sense that they had a lot of correctness issue with the software itself. They were very inflexible when we want to make a small change to the software. Inflexible in the sense it would cost a lot of money, it would cost lot of time, lot of effort, lot of people who are involved in making a change in the existing product. This characteristic of software has not changed even when this survey was done during the year 2000. Essentially when they did this survey of software projects in the late eighties, they came up with a term called software crisis. The software crisis was basically meant to illustrate the fact that the average software project was late by an average of almost 30 months, which was quite long.

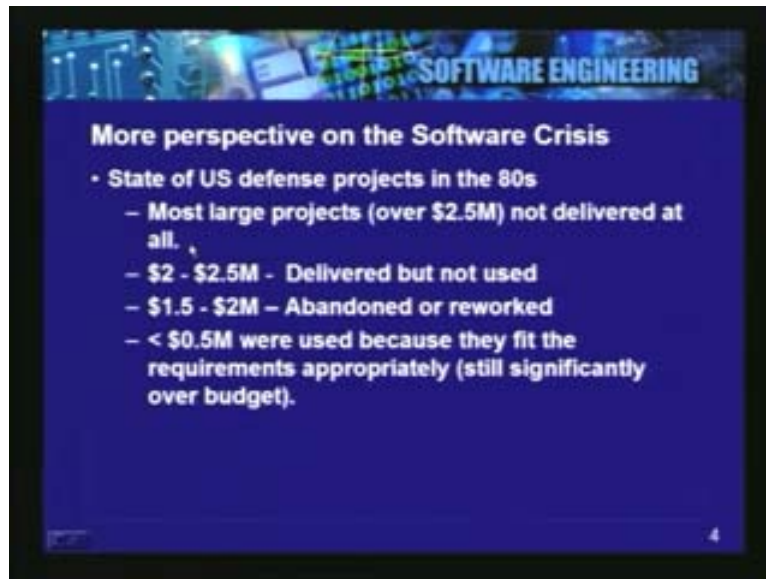
(Refer Slide Time: 03:15)



In fact today's project do not even last thirty months as before. More than 70% of the cost from the overall total cost of the software was taken up by the maintenance. Which means finding a bug, figuring out the solution and correcting the defect in the software as well as making small changes because of incremental requirements was very expensive. There were certain factors that we are going to go through today as why that occurs so.

The quality of the software itself was quite pathetic in the sense that there were lot of problems and issues of the software, there were lot of stability issues, there were lot of ease of use issues, there were maintenance issues, there were performance issues and there were functional correctness issues that it would not conform to the specifications that were written out when this entire project was started. Let us see some more perspective on the software crisis. I am taking a survey of the US defence projects in the eighties; they kind of broke it up into several categories.

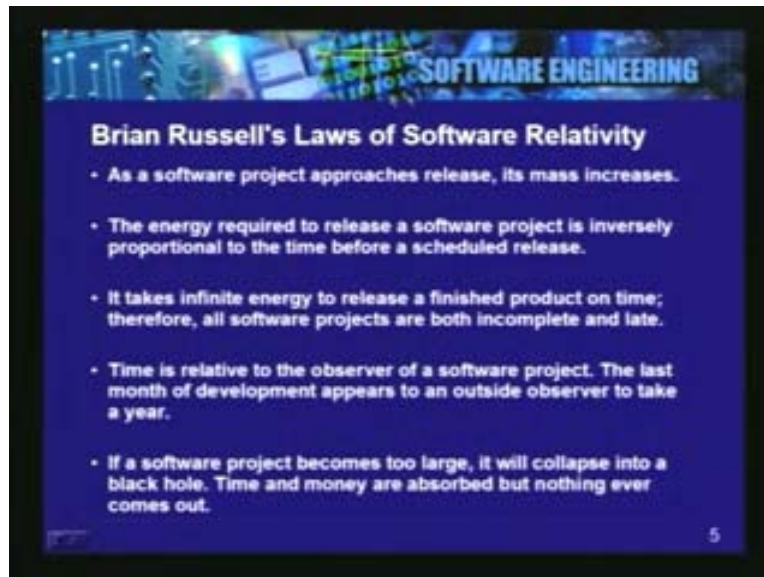
(Refer Slide Time: 04:30)



Among those, the first category is really large projects which were over \$2.5 million overall, most of these were not delivered at all. Over 85 to 90% of these projects eventually were not delivered and they became so late that the projects got cancelled. The second class of projects just slightly smaller than these was delivered, but they were never used. They were never used because of various reasons. The quality of the software was a major reason amongst them. Then if you go down, you see that some of these projects were completely abandoned or had to be reworked from scratch. Basically, the next category of projects which fell into the \$1.5 million to \$2 million range, pretty much had to be reworked from scratch because they do not conform to the specification. It took so long to complete the project that the requirements have changed in mean while and so on.

It was noticed that only very small projects, relatively small by those standards were used because they fit the requirements appropriately, but they were still significantly over budget. There were nothing ever delivered on time and under budget in those days. So experiences such as this, essentially gave rise to the term the software crisis and people were wondering as to what were the causes behind the lack of quality within the software. In fact as a humorous aside Bryan Russell has come up with these laws of 'Software Relativity' as he calls them. Again these are not exactly laws, but kind of illustrate the point that it is trying to make about the software. The first thing says that as the software project approach its release the mass increases, that is the number of people that end up becoming a part of the project goes up dramatically. The energy required to release the software project is inversely proportional to the time before a scheduled release and so on. You can read the rest of these in the screen shot below.

(Refer Slide Time: 06:24)



What these are basically trying to put in perspective is the software crisis that was occurring in the eighties. Every time the project got delayed, there are lot of people got put on to it, there were manpower issues on this and that was not the way to solve this like Fred Brooks also said in the late eighties adding people to a late project is only going to make it more late. What are the properties of failing software? It is worth while to take a look at some of the causes that led to these kinds of failures, and that led to the poor quality of the software that was being produced in those days.

One of the first things that come up in this list is redundancy. Redundancy is basically referring to the fact that in a software project most of the code or the development takes place from scratch. And there is very little reuse of components that are taking place. How the redundancy affects the quality? Every time you write code, it has to be tested thoroughly and the number of testing combinations that are going to exist depending on number of input parameters that piece of module is going to take, it tends to be prohibitively large. As a result of which, either statistical techniques are employed or the software is not very well tested.

(Refer Slide Time: 09:30)

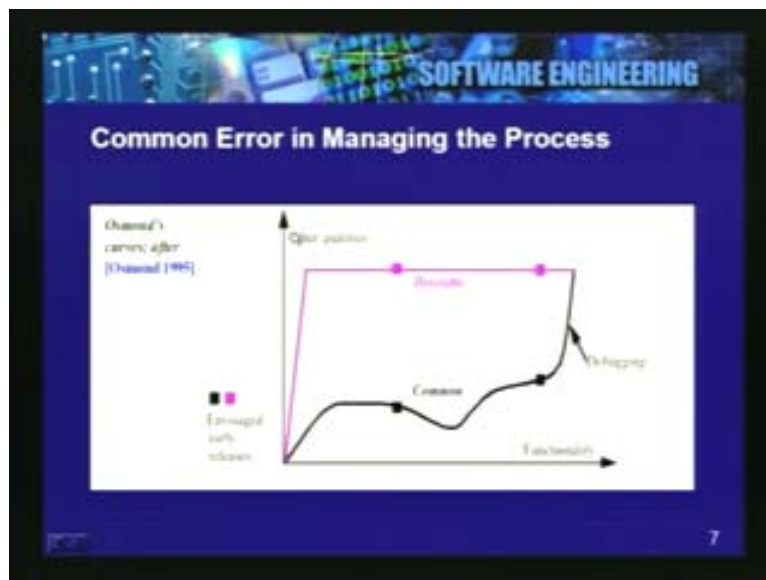


If we had reused and we were using pre built component that had been used in production situation several times before and you could be reasonably sure that these components could handle whatever is going to be thrown at them, because they had not just been tested, but they also be used in other applications. But that does not happen very often in software. There is a very high degree of inter-connectedness between the modules. This leads to a term called fragility of software and fragility basically refers to a fact that a small change that needs to be made is going to end up costing a lot in terms of time, in terms of effort, and in terms of money. This is because of the high degree of coupling that exists between the so called modules within the software.

Typically, a well modularized piece of software is supposed to have a very low degree of coupling between these two as we shall see going forward. One of the reasons why maintenance was so hard and it took so long to fix an issue or to add a certain requirement was the low understandability. The low understandability of the software came about because of the couple of reasons. One was that the abstractions were simply not there within the code. So there was lack of encapsulation. There was spaghetti code all over the place. Procedural abstractions were being used. Procedural abstractions basically consist of set of functions or set of global data that these set of functions manipulated and that just was not going to cut it as far as a second programmer who did not write the original piece of code trying to understand, makes sense of the spaghetti code and now he had to make changes in this environment. Of course documentation was always obsolete because software was changing much faster and the processes were not being followed. But that is not a focus of our talk here today.

Lastly, there were human factors. A human factor was man power estimation was closely inadequate. There was no effort estimation technique. But those are things that you have already seen so far in the course, in terms of how to attack a process, how to deal with man power estimation, how to deal with effort estimation of module and so on. Again that is some thing that we are not really going to get into in this particular lecture. These properties and the common error in managing the process of software development are well illustrated by this chart. What is this trying to show is that on the x axis is, the functionality that is being added and on the y axis is the quality properties of the software.

(Refer Slide Time: 10:01)



Things like robustness of software, the unit testing of the software, correctness, performance etc which are basically the qualities of non functional requirements. As the project went on suppose there were early releases before the project actually completed and these points indicate early releases. What normally happened was a lot of functionality was added initially all the way up to the point that the project came to its close and then all the testing and debugging got done in the last part of the project. Lot of the code was added without adequate testing getting done without adequate attention being paid to performance, being paid to ease of use, being paid to structural integrity of the software and so on.

Then most of the debugging took place in the last phase whereas what is actually desired is something that will set up the quality frame work very early in the process and then you add the functionality as you go along. You can make the same releases, with the same amount of functionality or to have lot more quality to it and that is the desired process which is shown as the pink colored line in the graph. Here is another example about how lack of encapsulation hurts maintenance.

(Refer Slide Time: 11:20)



Typically we have seen this chart before in terms of what is the break down of maintenance cost, where are the most bugs filed, where are the most change requests coming from. There are the only two important categories in this chart. The first one which is the largest category is the change in user requirement and it cost the most. It is not due to lot of changes, it is because of every change is very expensive to handle, it is because the software is fragile and not easily amendable for extension or not easily changeable or flexible, this happens.

Because the lack of encapsulation and lack of abstractions a change in the data format this is surprising statistic that almost twenty percent of the maintenance cost come from changes in data formats taking place. The Y2K bug is the famous example or infamous example of this. What are the quality factors in software that we need to be watching out for? The quality factors can be divided into two main categories.

One is external. External implies that it is visible to the user. Obvious example of this is performance. Ease of use is another example. The other factors are driven by internal needs, such a robustness of design. If I want to add a feature to the software, what is it going to cost me to do? This is driven by the robustness of design, flexibility of the design, openness of the design, software portability to other platform and documentation of the code etc. What building quality software is, the process, the mind set which is thinking about the internal factors and focusing on the internal factors in order to ensure that the external quality factors are being met. Basically, it is only the structural software with structural integrity that is going to allow you to make these changes quickly and which is going to allow you to give the kind of the performance that user is going to expect of this software.

(Refer Slide Time: 13:35)



SOFTWARE ENGINEERING

Quality Factors for Software

- External
 - Visible to the user
 - Example: Performance, ease of use
- Driven by Internal
 - Robustness of design
 - Documentation of code etc.
 - Building quality software is thinking about internal factors to ensure external requirements are met!
- Also can be thought of as:
 - Functional properties – profit, customer orientation etc.
 - Structural properties – Cost reduction

9

It can also be thought of as functional properties of software such as profit, customer orientation etc versus the structural properties of software which basically lead towards cost reduction. That was the categorization of quality features and if we actually dive into the quality factors themselves, correctness appears at the top of the list as it is to be expected.

(Refer Slide Time: 15:15)



SOFTWARE ENGINEERING

Quality Factors

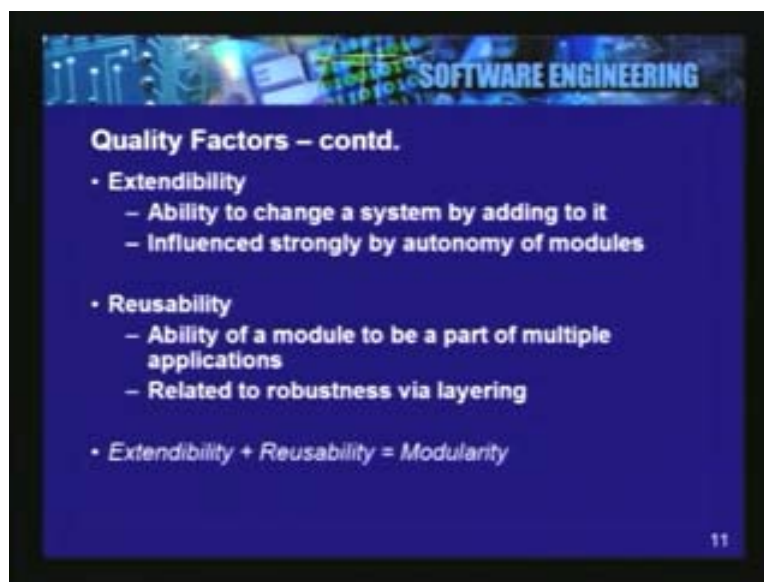
- Correctness
 - Conformance with specifications/requirements
 - Means of ensuring
 - Testing/Debugging
 - Layering (architectural approaches)
- Robustness
 - Ability to perform under abnormal conditions
- Correctness + Robustness = Reliability

10

What is correctness? Correctness has to do with fact that at least software has been built conforms to the specification, the requirements that were laid out in the beginning of the process. The means of ensuring the correctness are: testing and debugging. This is one straight forward mean. The other one is layering or architectural approaches towards ensuring correctness. So the changes in the lower layers essentially do not affect. They only affect the layer that is sitting on the top of it. For example when you change the operating system platform, software which manages say functional needs of say inventory management is not isolated from the operating system layer. Then you would have to make changes through out the software and more change to the code that is being made, the more perturbation that is being caused in the system, the more likely that it is going to be a quality problem in the software and in the near future.

The next quality factor that is very important is robustness and robustness basically refers to the ability the software to deal with abnormal condition. So those that were not specified for example or they might have been specified. Correctness and robustness when put together produces reliable software at the end of the day. These are factors that we should focus on.

(Refer Slide Time: 15:30)



However there are other factors as well that we do well take into account and some of these are expressed in this slide; extendibility of the software. The extendibility is the ability to not necessarily change the system but an addition to the system. For example we were building a module of inventory management and we would like to add ability to do logistic planning in that. For example you are managing inventory as well as you are going to have to manage shipments around to various warehouses. So logistic planning has to be added to this. When such a major requirement is brought into the picture, if the first piece of software, the inventory planning system was not built right in terms of design then it is not going to be very extensible.

Extending is going to cost a lot and may be new technologies would have to be used or entire system might have to be upgraded and so on. The autonomy of modules strongly influences. What this really means is the degree of coupling is got to be very low and the strength of cohesion or the degree of cohesion within a particular module has to be very high. We will go into the details of what exactly that means in a little while. The next one is reusability of the software. The ability of the module to be useful in multiple scenarios or the multiple situations directly is going to affect the quality of the software. This is also related to robustness, because if a module has been tested before, it is being used in multiple situation before, then it is likely that people will safely use it again and that would lead to a reduction in cost and it will lead to increase in quality.

Modularity essentially refers to the notion of extendibility and reusability. How well is it encapsulated is what we are talking about when we refer to modularity. Other factors are portability; the ability to execute on a wide variety of platform. For example, does it execute on Linux and on windows or does it work with the Netscape browser as well as with the IE browser and so on. The ability to execute on various configurations is what portability refers to.

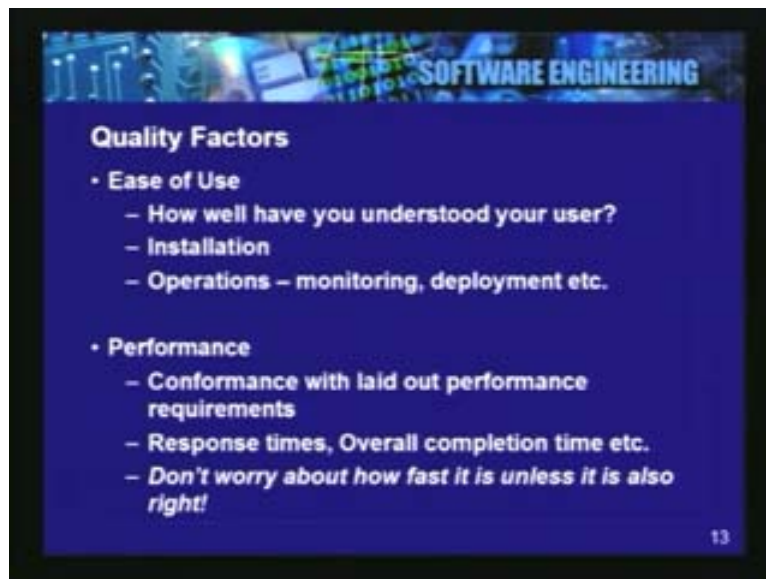
(Refer Slide Time: 18:45)



Efficiency refers to the usage of resources. For example, when I run this software, I have to think about whether I need a main frame to run it or a smaller box with one or two CPUs should be enough. And I should see whether it needs memory hog or 256 MB of memory would be enough. I should also consider I by O, bandwidth etc. All feature or resources that have to be carefully conserved and managed and the software should be efficient in dealing with it. The integrate-ability factor is basically refers to compliance with standards. For example, if my financial or accounting system has been bought from one vendor and my inventory management system has been bought from a second vendor, will these two talk to each other?

Every time I loose inventory for example I have to record it as lost inventory within the financial system that I have. Basically I should be able to automate this transaction, I should be able to pass this data from one software to another, and I should be able to control from one software to another. This is what integrate-ability of the software refers to. This is driven by compliance standard. If both of them work on a common standard, then integration would be a piece of cake. Otherwise it is going to be very hard and it would be taking out lot of time and resources. Then there are certain user perceivable factors like we talked about before: Ease of use. Basically how well have you understood your user? How easy is it to install the software?

(Refer Slide Time: 19:50)



The 'Ease of use' is not just a functional ease of use that we are talking about, but the non functional aspects of it. If it is going to take you two days, if it is going to require an expert system administrator to come in and install the software for you then it is not going to go out very well in certain situations. If it is a very complex piece or server side software it may be acceptable that the installation needs an expert. But if it is simple desktop software like word processor then it should be self installable completely.

'Ease of use' also means that, how well the software supports operation such as monitoring, deployment, redeployment, bringing it down, versioning etc. Another quality factor is performance. There are certain performance guide lines or performance requirements that would have been laid out at the beginning of the process and how well does the software conform to these performance requirements, response times, through puts etc. But certainly you do not have to worry about performance unless you meet the correctness criteria. The first two criteria that we have laid out which is correctness and robustness are absolutely key quality factors that we have to focus on. The question that we tend to ask ourselves is, we have looked at all these quality factors that affect software but, what are the keys to achieve the structural quality?

What is it that we can do in design? What is that we can do in our process that will help us produce quality software? The above questions will be the focus of today's lecture.

(Refer Slide Time: 22:40)



The first key is that of reuse and reuse was one of the main causes why software failed. So we have to attack that by making software modules be more reusable by having the appropriate degree of modularization or encapsulation within the software. We should give the confidence to the user that this module has been tested under wide variety of conditions, the module or the component is going to work on a wide variety of platform and therefore it is reusable and they do not have to develop their own piece of software for something like this. We have seen all the benefits of reuse in past lectures, but the key one are it increases reliability which means it is going to directly improve robustness, it reduces the development time.

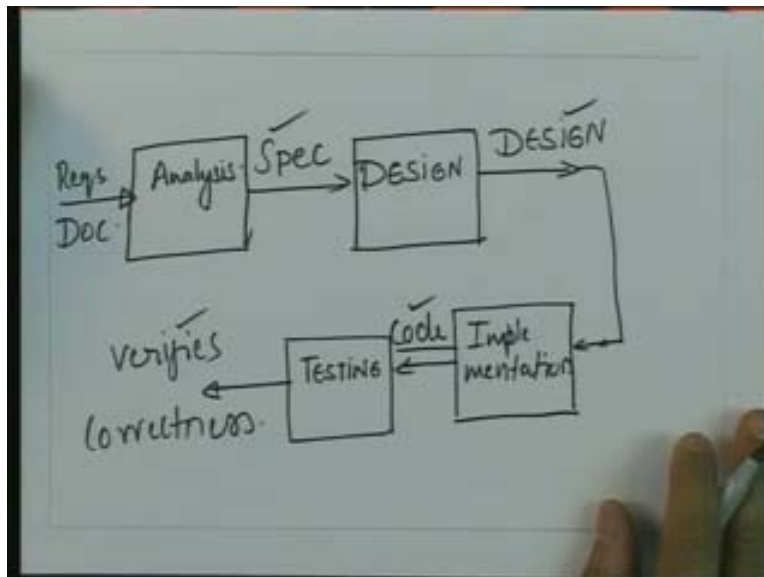
It is not just a software quality that gets improved, but also reduces the risk in terms of the process side of it and in terms of estimating how long a piece of software is going to take to construct can also be brought down by reuse. We have to distinguish between designing a piece of software by reusing component which is called design with reuse and a designing a component itself with in a your software construction process so that it can be reused across a wide variety of situation which is called design for reuse. The second key to achieving structural quality is that of abstraction.

The term is not very well defined, but we shall try to put a definition to the notion of what abstraction is within software development and there can be different levels of abstraction and that is what in fact object orientation is all about. Basically object orientation is about introducing abstraction at various levels within the software design process or with in the software construction process. The third key to achieving structure quality is continuity in the process.

We know that there are various phases in the software development lifecycle starting from requirement, going on to analysis and specification, design, implementation, testing and so on. Often what usually happens with software development projects of the past is that, English was the common denominator for specifying software requirements.

It was up to a human or a person to take the requirements specification documents, understand it and create a specification during the process of analysis. With that result, to check whether the analysis that had been done or the specification that had been written, conform to the requirements that were given in the first place. It was not possible to be done by a machine, it could not be verified that the analysis was indeed correct. The specification that has been turned out as a result of analysis could indeed be traced back to the requirements. So let us try to pictorially put this out in different phases. The first phase is that of analysis and in goes the requirements documents and outcomes a specification right.

(Refer Slide Time: 25:20)



Once this specification is then fed to a design phase and out comes a design. The design is fed to the implementation phase and outcomes code. The code is then fed to a testing phase which basically verifies correctness. So there is input in every module and an output from the module. This is a process and in the process there are artifacts that are created by every process. The analysis creates the artifacts as Specification. Design creates the artifact of design. Implementation creates the artifacts of code and testing basically gives you a Boolean answer to whether something is correct or wrong. In the traditional software engineering process, these artifacts that we have laid out here for example, the specification, the design and so on, do not get reused from step to step of the process. In other words there is a human who reads the requirements document, comes up with the specification and another human reads the specification and comes up with a design.

So to go back and verify whether the specification meets the requirement, whether the design meets the specifications, whether the code meets the design which in turn meets the specification and the requirements is simply not possible, because there is lack of continuity in this process.

If we can somehow achieve that continuity by keeping a common language across this process, that there is successive refinement going on all the way from requirement specification, analysis, design, implementation and then testing then we can ensure that there is traceability in this process. Testing should be really taking code and taking the requirements as an input and then with the help of a machine verification technique we should be able to match the requirements to the code and ensure that the code indeed meets the requirements that have been written out.

If we can achieve that we would have gone a long way towards improving the quality of software that gets constructed. We saw the 3 keys to achieving structural quality was reuse, abstraction at various levels and continuity within the process. Coming back to the abstraction, how do we define abstraction? What are the different levels of abstraction that we have talked about etc?

(Refer Slide Time: 27:21)



The abstraction is basically the process of separating out the properties that is inherent in some body from the body itself. An example of this could be that every vehicle is characterized by certain property and that is an example which is given out here. Every vehicle is characterized by certain property. The properties of a vehicle are it has an engine and it can move you from one place to another. It is not necessary that it is going to have wheels because not all vehicles have wheels. Land vehicles have wheels on them and that is another feature that we want to look at.

If we take a look at land vehicles, then we can say that land vehicles have wheels in addition to an engine and they have some means of steering etc. What we are trying to do in this process is to take a look at a specific object and in this case let us say we took a look at a car. We are trying to generalize some properties from this specific object and then apply it to all objects of that kind. No matter whether you are talking about Maruthi car or whether you are talking about Hyundai car you will notice that it is going to have an engine, it is going to have four wheels and it going to have four doors etc. Some properties could be relevant only to the specific instance, for example, a two door car is possible and some properties can be fixed, for example, four wheels on a car is typically fixed. No matter which make of car it is.

The process of abstraction is the process of more human like reasoning. It is a process of generalization from specificities. In that process what you are doing is, you are taking certain essential characteristics and you are taking them away from your particular instance of the whale that you saw, particular instance of the car that you saw Then you are creating this abstract representation saying, I am going to stay away from this car, but I know what all cars are now characterized by. In other words, it can also be thought of as distinguishing essence from detail.

What is important in considering all of these definitions is that we have to somehow relate them back to the notion of software quality. Remember the factors that affect the quality that we went through. There were factors such as extensibility, there were factors such as reusability, and there were factors such as maintainability, understandability and so on. We have to be able to relate all of these back to the definition and that is when it becomes important. The definition of the abstraction is the notion of distinguishing the essence from the detail What exactly does this mean? This means that what is important about it is not the representation or how it is constructed. But what can it do for us is what we would like to take a look at first. That is called the separation of interface from the implementation.

There can be different or multiple representations for a particular interface for some object and those different implementations become possible when you separate them to replace one implementation with another. If this level of abstraction does not exist, it becomes very difficult to change something into software. If the interface and the implementation were inter-connectedly mixed then the clients of that particular module would have to be changed if the module gets changed as well.

Abstraction focuses on 'what' instead of the 'how'. 'What': is the process of analysis, the process specifying the interface of a module. 'How': is the process of design where you specify, how you build that functionality when the module is exported to the rest of the world. Why the abstraction is useful in producing quality software is something that we have to be convinced about before we move forward. One of the things of separating implementation from interface is that safeguards against implementation variations. What this means is that you can essentially change the implementation of something without affecting the clients that use that particular module. A good example is famous or now the in-famous Y2K bug that existed.

What happened there in that case was not the criticality was not caused by the fact that the representation of the date simply had two digits to represent the year instead of four. But that was not the heart of the problem. The heart of the problem was lack of abstraction in that internal implementation of the date which was exposed through out the software and they did not use an abstract interface to a date. As result of which when a change has to be made to the structure of that date module, it had to be changed and written in millions of places which have now used this particular internal structure directly without an abstract interface to it.

Therefore the separation of interface from implementation or separating the essence from detail is very critical. Another example can be that you can construct a stack as a list or as an array and it does not matter to the user of the stack on how you end up the building the stack. What he cares about is the operations that the stack supports.

The operations such as pushing, the popping, the top of the stack, creating and deleting a stack etc is that what he cares about. As long as you implement it, using an array or a list or whatever else it does not matter to the user of the stack.

It also promotes reuse. Abstraction will basically allow you to say a car has everything that of a vehicles. But it may have certain specific characteristic such as four wheels. The truck is also a vehicle, but it may have 8 or 10 wheels or 12 wheels. This notion of reusing whatever definition has been done up to date. But now extending it goes a long ways towards promoting reuse as well. It is important to try and figure out as to how we get the process of abstraction right and the three key quality characteristics that we have to look at in order to ensure that the abstractions we have come up with are just right.

First one is be 'precise', so there needs to be some formal way of putting down the properties of abstraction that we are trying or to write down or properties of the module that we are trying to write down. The second thing is that it should be 'complete'. It will not be reused unless it is complete. It has all the facets that are attached to it that could be used by various people. The third thing is that you do not have to go over board. You should not 'over specify', else the understandability feature is going to get seriously affected. If there are too many details that are coming out, it is not the essence of that object. There are different abstraction techniques. In fact if you take a look at this list, encapsulation, information hiding, polymorphism, inheritance, genericity and so on.

(Refer Slide Time: 34:56)



This is essentially when combined makes up what is called object orientation. The focus of object orientation as a process in general is that of abstraction. These techniques will explain how to maintain the continuity in the process as well as how to provide the right tools for the people to create these abstractions at appropriate levels. Let us go through some of these things in slightly greater detail and see what each of them mean.

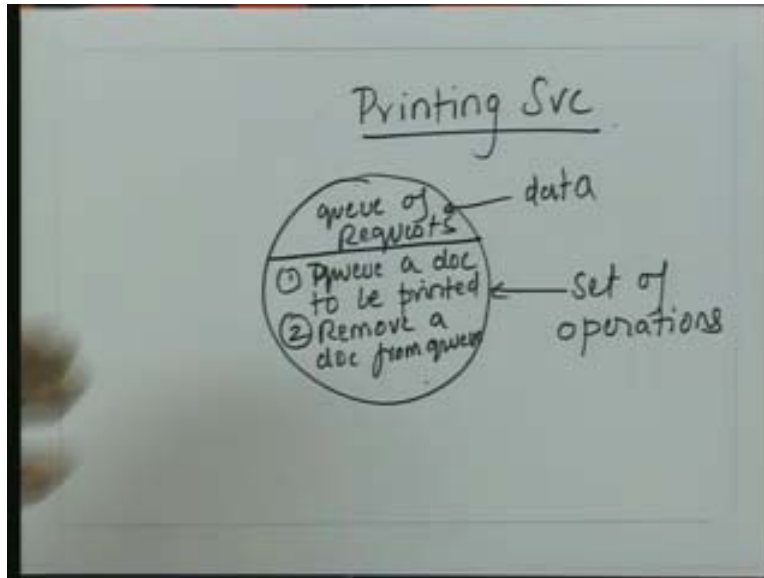
(Refer Slide Time: 35:38)



The first one is that of encapsulation. Encapsulation is also known as modularity. The encapsulation is all about putting together data and operations that operate on the data. Operations that operate directly on the data is the key part of the definition. It is not

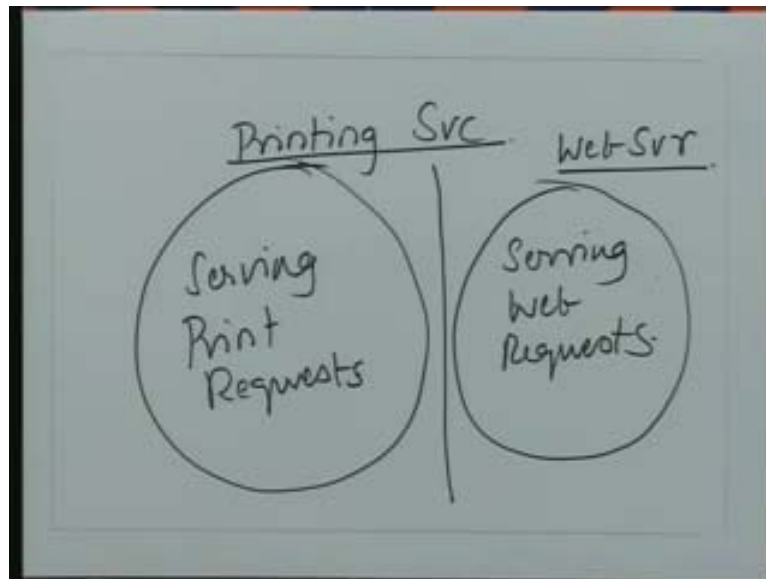
unrelated operations or operations that sound nice or vaguely related to it. They have to directly be manipulating the data that you are storing as part of that object or of that module or of that entity. To give an example, class in object oriented language provides you a means or a tool of creating an encapsulation. Let us take the example of a printing service. To create this module you would first specify that it has some data that is associated with it and the data could be marked down here as queue of requests. Then there are operations that manipulate this queue of requests.

(Refer Slide Time: 37:45)



The first operation could be queue a document to be printed. The second one could be remove a document from the queue etc. Here we have tried to create a module. An encapsulation is a piece of data and set of operations, which in this case happen to be the 'queue of requests' and the set of operations. But it is not just any set of operation it is a set of operation that directly end up manipulating this data. We bring them together in a cohesive unit that cannot be taken apart that is what encapsulation is all about and a related principle obviously is 'separation of concerns'. If you consider the printing service and you do not want to have more than one responsibility attached to the printing service. Let us say for example, printing service will not monitor the web server that is running at a site. It is completely an unrelated task and there is no reason to bring these two things together.

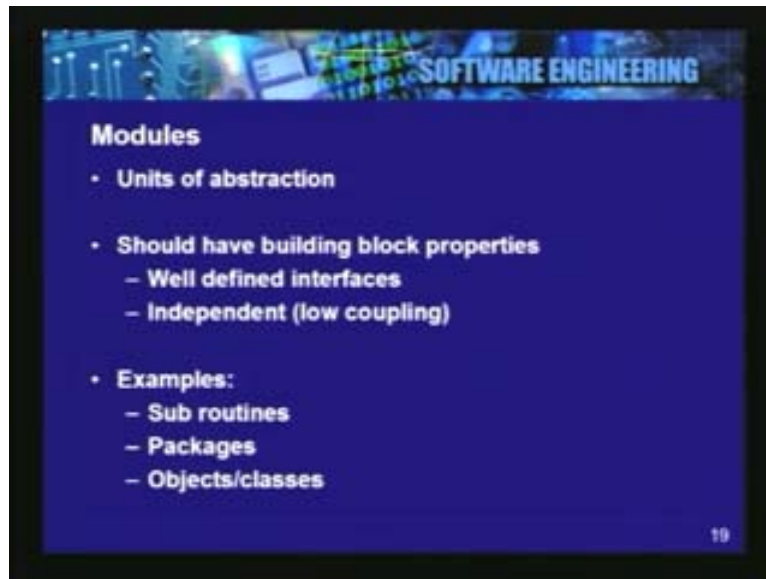
(Refer Slide Time: 39:05)



So here is the picture that explains the same. A web server (Web Svr) has the responsibility of serving web requests. The printing service (Printing Svc) has the responsibility of serving printing request or print requests. If you ever combine these it would make no sense. The separation of concerns, drawing a line between two concepts that are completely unrelated to one another is a part of encapsulation, is the part of making sure that only things that are related stay together but nothing more. Instead of having spaghetti of mixing all these things together, this helps you to manage the complexity. For example if you did not have a means of encapsulating and separating these concerns then all of the data would sit together, in this example it would be the data related to the web server and the data that is related to print server. And then there would be a set of function which is manipulated as global data.

The degree of understandability and the maintainability of the code go up significantly when it is well encapsulated and that is what it refers to. The modules are the units of encapsulation or units of abstraction. What are the characteristics of this module? The module should basically have what are called building blocks properties. Suppose you are building a wall and you are going to do it by putting bricks on top of one another and then cementing the bricks.

(Refer Slide Time: 40:10)



The bricks have certain essential properties being building blocks. They have well defined shapes. If bricks were all of odd shapes you could not possibly put them together. As you know that you can place one brick next to another and that is exactly the same thing with respect to software. They have well defined interfaces and therefore you know the two components can come together because of these well defined interfaces. The second property is that they are very independent. If I pull in a component, I do not have to pull in ten other related components. So there is a low degree of coupling between different components or between different modules or between different encapsulations. Examples of modules in various languages can be procedural languages have subroutines, object oriented language have classes and objects. There can be languages like ADA for example which is object based but not completely object oriented which has the notion of packages.

The desirable characteristics of modules are that it needs to have low degree of coupling and high degree of cohesion. What is it really mean? There are different levels of coupling within software that we can see and this chart lists it in the decreasing order of strength. This means that what you want is the lowest level of coupling that is data coupling.

(Refer Slide Time: 41:29)

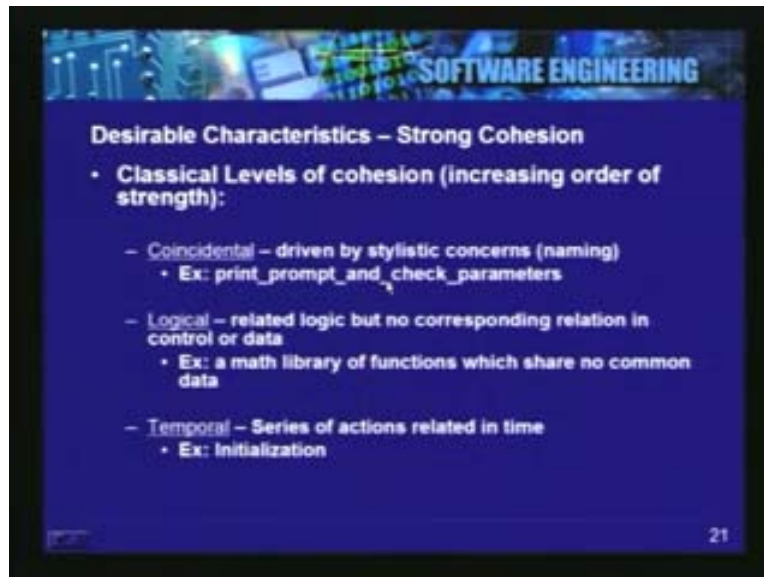


The content coupling refers to when a module directly refers to the contents of the code of another module. This is absolutely something to be avoided and example of this is a jump statement. If you write control that jumps from one place to another in a code or the jump for one function to another in the code, then it is not very good. The second pretty bad degree of coupling that we have is what is called common coupling. Common coupling occurs when you have access to the same global data. Every time you have global data then it is very hard to separate the two modules from each other.

Control coupling which directly affects the control of another module without reaching in for the code. Control coupling is basically making a module to do something which is not its responsibility. For example one module is requesting another to print an error message on behalf of itself. Actually printing the error message should be in the first module and not sending to another module for that to get that printed. Another type is parameter coupling, which is basically bound to happen. In signatures, it is bound to happen because you have to call different modules functions. There are parameters or data that has to be passed from one module to another. It is bound to happen and it is not an issue with this degree of coupling is acceptable because in strongly type programming languages there is a type checking that is done by the compiler.

The compiler tells you whether passing data are wrong type so that somebody else is going to manipulate it wrongly. So that cannot happen and it is okay to do this. And lastly the data coupling which we have went through already. Just like we had strong or weak coupling to be a desirable characteristic, we have a strong cohesion to be a desirable characteristic. There are six classical levels of cohesion that we can see. One is called coincidental cohesion which is driven by stylistic concerns. An example here is 'print_prompt_and_check_parameters'.

(Refer Slide Time: 43:43)

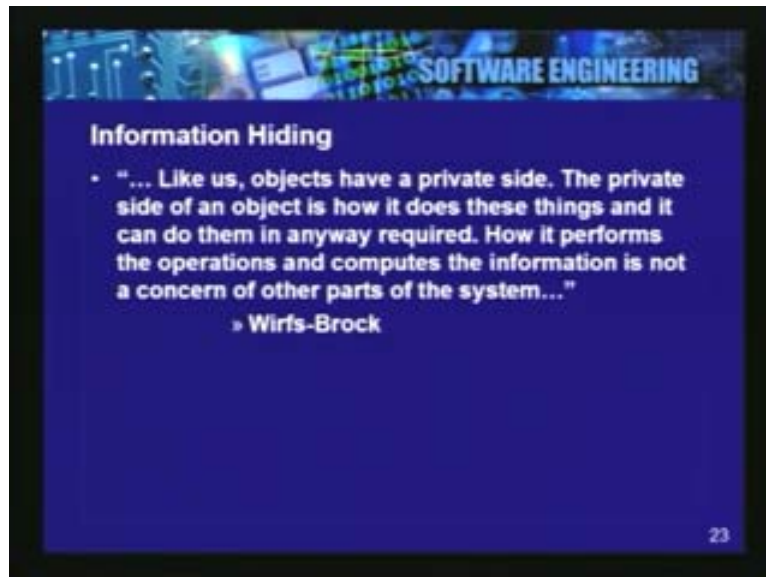


This is a shell function. This is trying to do two different things. Printing the prompt and checking the parameters that have been given to it. But they are trying to put these two together and that may or may not make sense. It typically does not make sense to do things like this which is why it is the first one. This list is in the increasing degree of strength and you want as a high degree of cohesion as you possibly can get. The second one is logical. It is related logic but no corresponding relation in control or data. Again something we do not want. Library of functions those are completely unrelated to one another. But they may be in a library but they all are math functions or something like that which you do not want again.

Temporal is the next characteristic. It is series of actions that are related in time. Initialization is a good example of this. There are four other levels of cohesion and what we really want is the final level of cohesion which is called data cohesion. Data cohesion refers to the fact that its central item around which the module is being constructed is a piece of data and then all the things that are surrounding the piece of data manipulate that data in some way. It is a set of operations that are related to a single piece of data. Objects and classes in object oriented design help us do that. Typically what we want is a low degree of coupling between modules and a high degree of cohesion within a module.

There are different classical definitions of these that we have just gone over and you should take care to ensure that your design follow some of the principles. The third level of abstraction that we would like to go over is called 'information hiding'. Here as a definition by Wirfs Brock for information hiding. He is one of the people who did some pioneering work in area of object orientation.

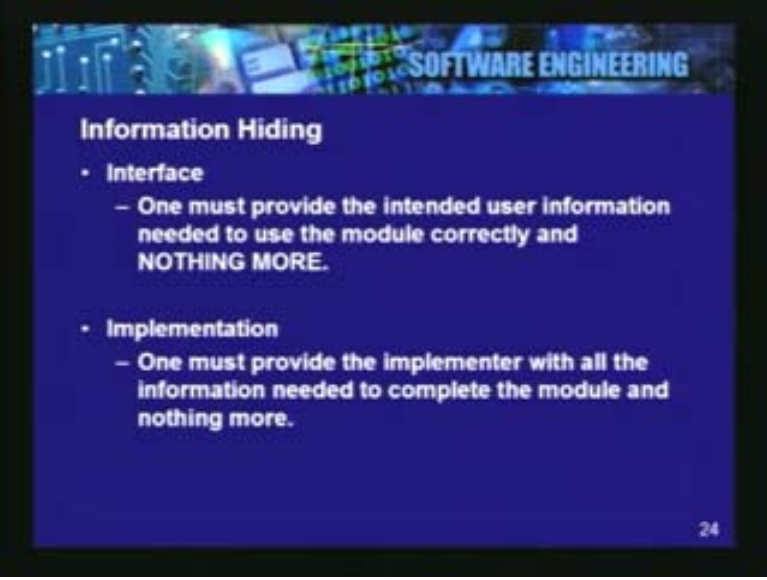
(Refer Slide Time: 45:55)



I am just going to quote this first before we talk about it. "...Like us objects have a private side. The private side of an object is how it does these things and it can do them in any way required. How it performs the operations and computes the information is not of concern to other parts of the system.." The last sentence in this particular quote is the key and this goes back to the notion of separating the essence from the detail. Separating the 'what' and the 'how' it performs. The 'what' is the one that is the public interface of the system. The 'how' is the information that is private. You have to somehow provide a tool for keeping this information private.

For example, consider the implementation of a stack as a 'list'. List representation of stack is to be kept private. It does not need to be exposed and there needs to be some tool within the language that you are working with, within the design notation that you are working with to ensure that this is private data, private representation as suppose to a public interface that other people can see. In a way this is an access control mechanism and that is what this slide is going to show. The interface is something that one must provide the intended user information about how to use the module correctly and nothing more. Implementation refers to as the information that is needed to complete the module to the implementer and nothing more is provided. So those two parts are kept separately. Another way of looking at it is that we keep the protocol separate from the behavior. The protocol is the envelope of the behavior and behavior typically it says what is the way the object acts and reacts.

(Refer Slide Time: 47:14)



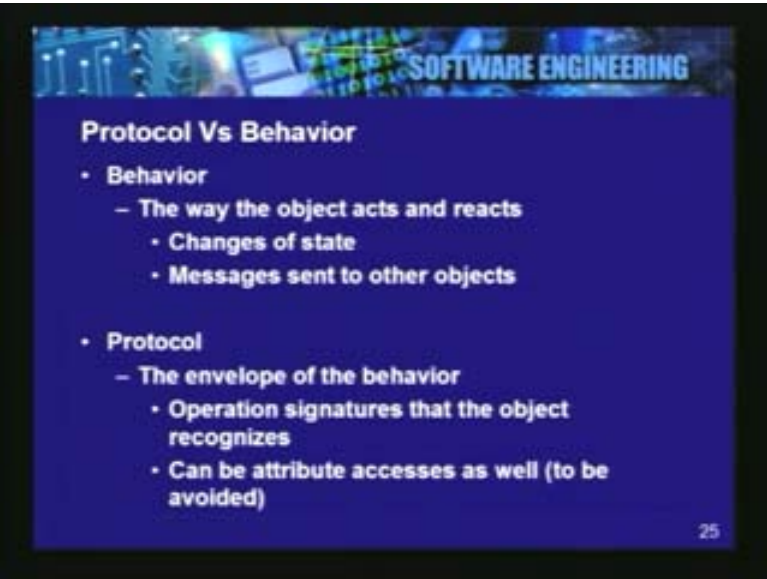
Information Hiding

- Interface
 - One must provide the intended user information needed to use the module correctly and **NOTHING MORE.**
- Implementation
 - One must provide the implementer with all the information needed to complete the module and **nothing more.**

24

Objects typically do changes of state. They send messages to other objects and so on. Envelopes of the behavior are simply the operational signatures that form part of the class definition.

(Refer Slide Time: 47:46)



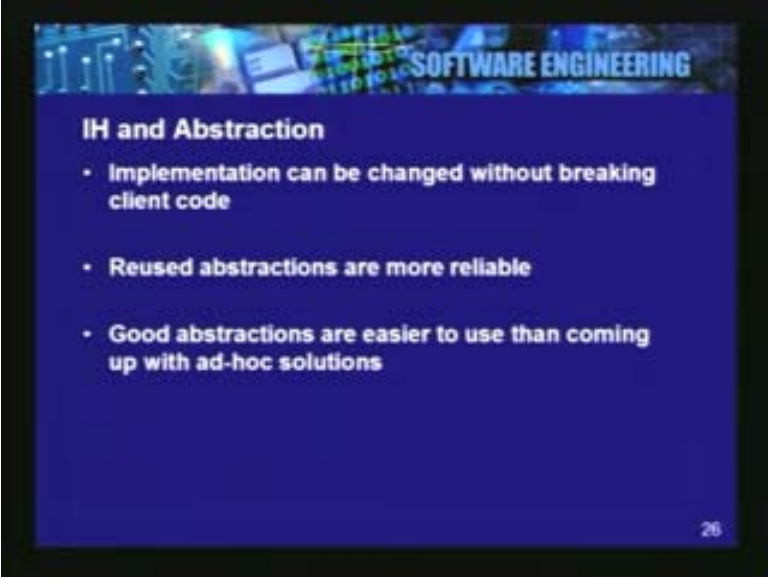
Protocol Vs Behavior

- Behavior
 - The way the object acts and reacts
 - Changes of state
 - Messages sent to other objects
- Protocol
 - The envelope of the behavior
 - Operation signatures that the object recognizes
 - Can be attribute accesses as well (to be avoided)

25

The obvious advantages of information hiding are that reused abstractions which becomes more reliable. As a result of which you want to keep the two things separate and therefore implementation can be changed without breaking client code.

(Refer Slide Time: 47:53)



SOFTWARE ENGINEERING


IH and Abstraction

- Implementation can be changed without breaking client code
- Reused abstractions are more reliable
- Good abstractions are easier to use than coming up with ad-hoc solutions

26

The last abstraction that we will consider today is that of inheritance. Here is one motivating example. Suppose you want to computerize personnel records. We start by considering two types of employees. Engineers and salesperson, both of them have name, birthday and salary as attributes. Remember these as essential properties and a salesperson has one additional attribute called commission. How do you model something like this? One way of modeling this is to have a class employee and the employee has the name, birth date and salary attributes attached to it.

(Refer Slide Time: 48:35)



SOFTWARE ENGINEERING

Factorization and Specialization

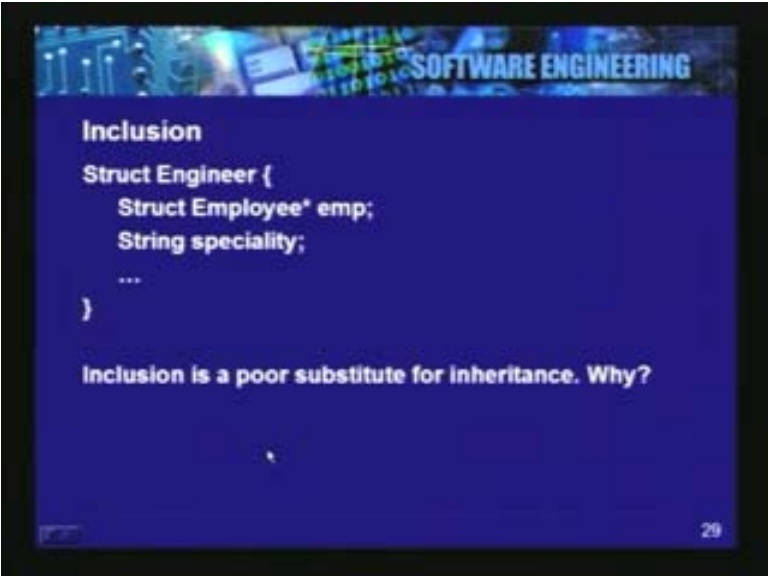
```
Class Employee {  
    private:  
        String name_;  
        Date birthdate_;  
        Integer salary_;  
    public:  
        void raise_salary(float);  
    ---  
}  
  
Class Engineer extends Employee {  
    private:  
        String speciality_;  
}  
  
Class SalesPerson extends Employee {  
    private:  
        Float commission_;  
}
```

28

Then there is an engineer class. Engineer is the new module that extends the employee. So it reuses everything the employee has and simply adds a thing called 'specialty' of the engineer. Similarly a class salesperson with merely extends what the employee has and extends it by adding commission which is the additional attribute. This is the way of factorization and specialization. You are trying to reuse most of what is there or all of what is there and trying to add to it the features that make something more specialized or more specific. Another way of doing that also could be by inclusion. It is a very poor substitute for inheritance and by the way this notation that you just saw is an object notation for inheritance.

Basically allows you to find what are called base classes and then extend them by using the inheritance mechanism. The 'extends' keyword being employed in this particular case. You can also do the inclusion. But it is not a very good substitute for inheritance, primarily because of the below reason. The benefits here of inheritance are that in the case the derived object also behaves as if it were an instance of base class or base object.

(Refer Slide Time: 49:45)



SOFTWARE ENGINEERING

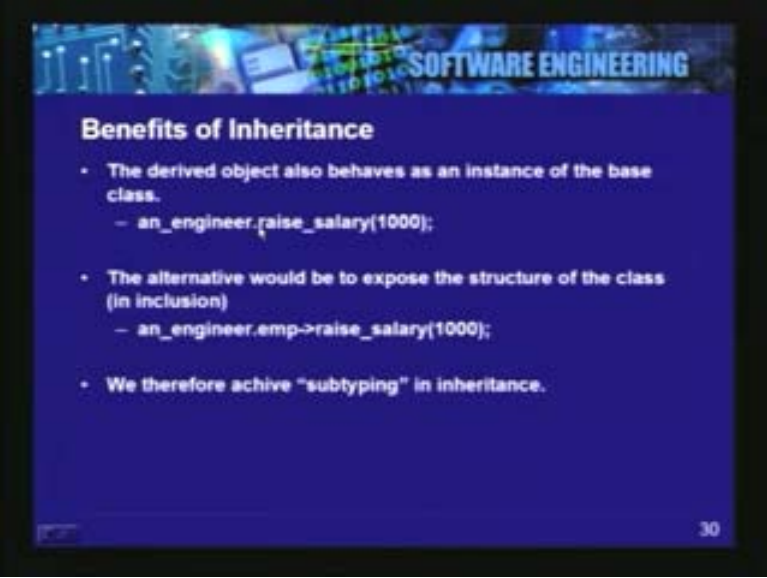
Inclusion

```
Struct Engineer {  
    Struct Employee* emp;  
    String speciality;  
    ...  
}
```

Inclusion is a poor substitute for inheritance. Why?

29

(Refer Slide Time 49:52)



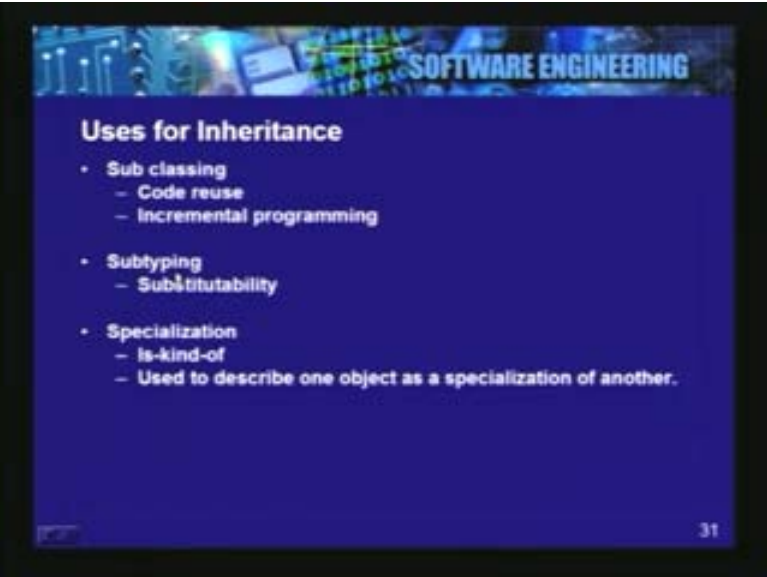
Benefits of Inheritance

- The derived object also behaves as an instance of the base class.
 - `an_engineer.raise_salary(1000);`
- The alternative would be to expose the structure of the class (in inclusion)
 - `an_engineer.emp->raise_salary(1000);`
- We therefore achieve "subtyping" in inheritance.

30

A sales person is also an employee, an engineer is also an employee and so any object of type engineer behaves as if it were an object of type employee as well. If you want it to call some operation say raise salary and raise salary was defined on the employee then you can call that even on engineer or a salesperson without having to worry about. But in the case of inclusion, you would have to expose the structure of the class which would be a bad thing to do. You would be breaking encapsulation at this point in time and therefore what you need to seek for is achieving sub typing in inheritance.

(Refer Slide Time 50:39)



Uses for Inheritance

- Sub classing
 - Code reuse
 - Incremental programming
- Subtyping
 - Substitutability
- Specialization
 - Is-kind-of
 - Used to describe one object as a specialization of another.

31

The various uses for inheritance here are, the first one is sub classing and second one is sub typing and third one is specialization. What this is really giving you is a tool for creating hierarchical abstraction. The first level in hierarchy in the example we saw was that of employee. We want to extend the hierarchy. The second level was that of a salesperson or an engineer, both of them were at the same level. You could obviously specialize that further by saying that these for electrical engineers versus mechanical engineers and may be there could be further specialization of salespeople as well.

Inheritance is also an abstraction and the abstraction that inheritance provides is a way of creating a hierarchical abstraction, because that helps us model the things that we see around us much better. What we have looked at today is the features that affect that quality of software. We have looked at the both the external features as well as the internal, the structural integrity of the software types of features. Then we have seen how we can we effect these features in a better way. We saw the abstraction and the reuse were the keys to achieving structural integrity within software. We took look at the various levels of abstraction such as encapsulation, inheritance, modularity, information hiding and so on. That helps us achieve a certain design style that can contribute to much better quality software.