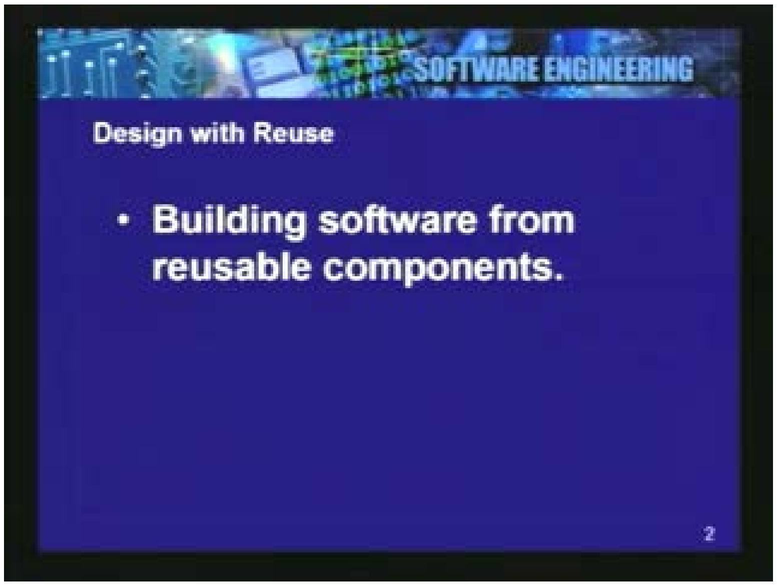
Software Engineering

Prof. Umesh Bellur
Department of Computer Science & Engineering
Indian Institute of Technology, Bombay

Lecture - 27
Software Reuse, CBSE

Hello and welcome back to the course on software engineering. Today we are going to be talking about a subject that has been receiving increasing attention of late of software reuse. If you take a look at the traditional engineering methodologies for building other kinds of systems for example hardware, for building civil engineering, construction, mechanical etc you will find that a lot of work that has been is going to be done to build a new system component, artifact, entity whatever borrows a lot of the ideas the design ideas, borrows a lot of….. even prebuilt components from other from previously done work but in software because it is a relatively new field that practice of reusing software what goes into making the software in the software itself is not very well established and we will try to take a look at some of the notions in the field of software reuse; now what is reuse, what are the problems of reuse, what are the advantages of reuse, what are the different ways of…… what can be reused, what are the different ways of doing it is the main objective of this lecture.

(Refer Slide Time: 1:57)


SOFTWARE ENGINEERING

Design with Reuse

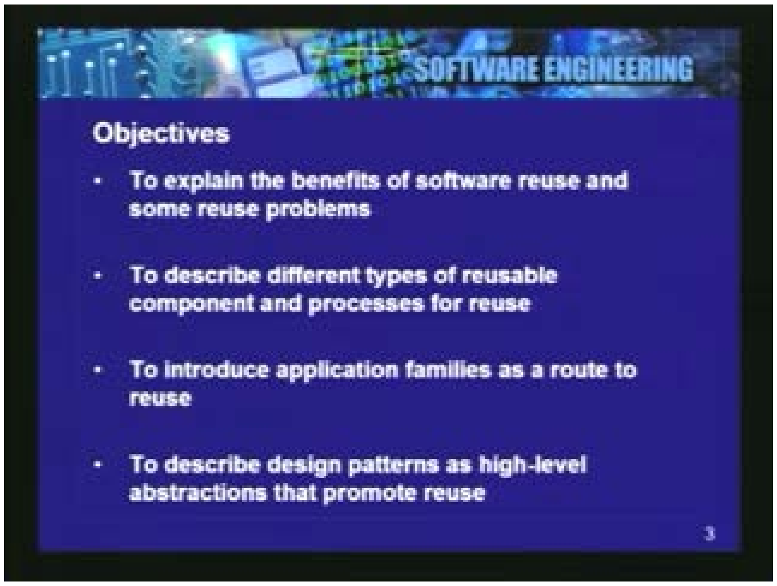
- Building software from reusable components.

So, design with reuse is a frame phrase that is often used and there are actually two phrases that are complimentary phrases to one another designed with reuse and designed for reuse. Designed with reuse essentially focuses on building software from reusable components. So, instead of trying to write software from scratch what I will do is take components that have code built into them that embeds certain algorithms within them that I know that I also want in this new project that I am trying to put together and I will simply reuse these components.

On the other hand, design for reuse is essentially the reverse mechanism. So when I am designing a component that I am going to use in my project I consciously make an effort to design it in a more generic in a more abstract manner so that it can be reused across multiple projects. So, in one view the software project that is currently being undertaken is a sink for component and it is a sink as far as the reuse process is concerned; the other case it is the source as for as the reuse process is concerned.

So what we like to look at today essentially are what are the benefits of reusing components; although it kind of seems obvious it is worthwhile discussing what exactly are we going to get out of this because it is not been adopted very widely if you take a look at software projects in the 70s and the 80s it is only in the mid to late 90s this notion of reuse really picked up and effects are being made to design with reuse.

(Refer Slide Time: 03:41 min)


SOFTWARE ENGINEERING

Objectives

- To explain the benefits of software reuse and some reuse problems
- To describe different types of reusable component and processes for reuse
- To introduce application families as a route to reuse
- To describe design patterns as high-level abstractions that promote reuse

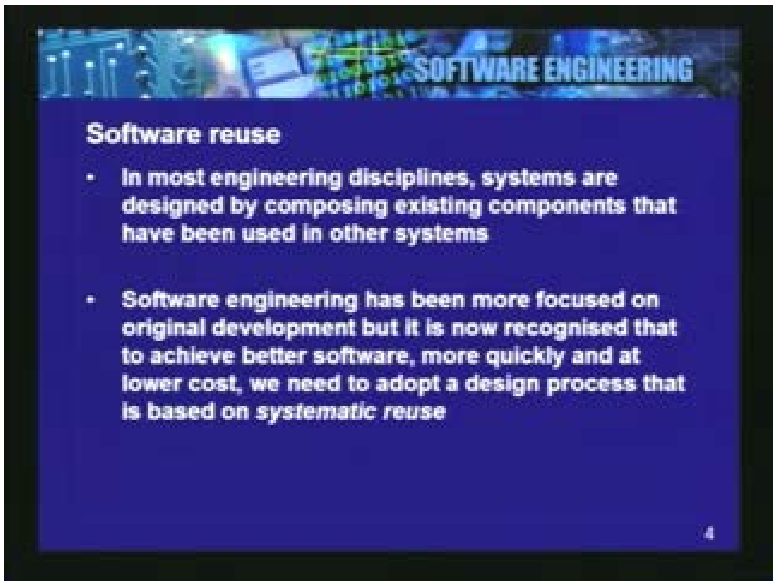
3

We will also take a look at some of the different reusable levels the levels of reuse that can occur. So, for example, you can go all the way from the notion of reusing a requirement specification that has already been written or all the way through to reusing the entire systems that have been built so reusing what are called as common off-the-shelf system or cot systems and the entire system can be reused in another instance.

We will also look at the intervening levels of reuse. so application framework for example is a way of reusing architecture, reusing design and reusing some code although some additional code would have to be written in this particular case; design patterns are a way of reusing designs only there is no code that typically comes with that, no clauses, no abstract clauses or anything of that sort and there can be component based reuse or functional reuse where an algorithm certain amount of code that have been written for a specific function can be reused; example of these could be math library, graphical user interface libraries and so on and we will take a look at the different levels of reuses as we go through this lecture.

So like I said earlier in most disciplines; in most engineering disciplines reuse is the way of life, you know; if you take a look at a typical design for a bridge you do not design the entire thing from scratch you basically understand the trusses that make up the bridge that are kind of prebuilt designs for all of these things. So if we know the type of bridge that you are going to build you are going to borrow designs from existing bridges which have already been done and there are parameterized.

(Refer Slide Time: 05:22)


SOFTWARE ENGINEERING

Software reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems
- Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on *systematic reuse*

4

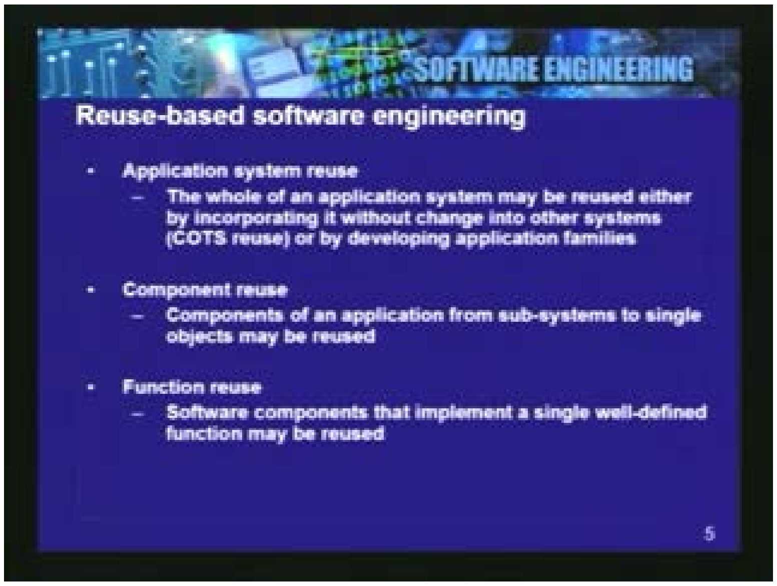
So, for example, if this bridge is to be 1 kilometer long instead of the previous bridge that was built which is going to be 2 kilometers long you can appropriately scale things down they have already been parameterized and therefore arriving at a design for the new bridge is not going to take as long because you are simply reusing bits and pieces of the design of the older bridge. Indeed this can extend to things beyond design; the entire parts of bridges may be prefabricated and kept when you are just using prefabricated components and more and more of that is seen when you are building mechanical devices, when you are building robots for example even when building computer hardware and you know the chips are pre-manufactured and whenever a computer is to be put together you do not start by making your own chip you buy a chip that is available that embeds certain capabilities; the CPU is a chip that embeds the processing capability, there are memory chips available of various kinds. So, depending on your needs you pick the set of appropriate components.

For example, if you need a 40 Gig hard drive or a 80 Gig hard drive you pick the appropriate component and you assemble everything together so you unify all of these and you may have to do some gluing work. In this case you might have to make your own motherboard which allows all these components to interact with each other appropriately using the standards that have been specified. But if you take a look at software engineering traditionally it is been more focused on building all the core all the way from scratch every time unless it is in the same company.

For example, if I were going to build a graphical user interface for an e-commerce website there have been hundreds of e-commerce websites that has been built; a lot of these user interface notions are already available as prebuilt components. But typically the tendency is to build the entire e-commerce website from scratch; you take a HTML editor and sit down and start building the e-commerce website and over the last five to ten years how the realization has definitely come in that this is an expensive process the first thing and it is prone to errors; every time you end up writing new codes you are likely to introduce errors so the way of reducing the error rate or increasing the robustness of the software that you are going to produce at the end of a project would be to try and reuse proven code try and reuse proven components that have been written earlier that have been tested thoroughly and have been used in previous production situations. So, if you are just able to pick that off and reuse as it is then you not only save time but you also provide error free code.

We will look at some of the advantages of reuse as we go along. So reuse based software engineering can be done at different levels like I talked about. It can be a functional level that is, a single function can be reused. For example, in a mathematical library they may provide functions for calculating the factorial of a number, for calculating square roots of numbers and so on and so forth in the simplest level and all of these functions can be……

(Refer Slide Time: 08:39 min)


SOFTWARE ENGINEERING

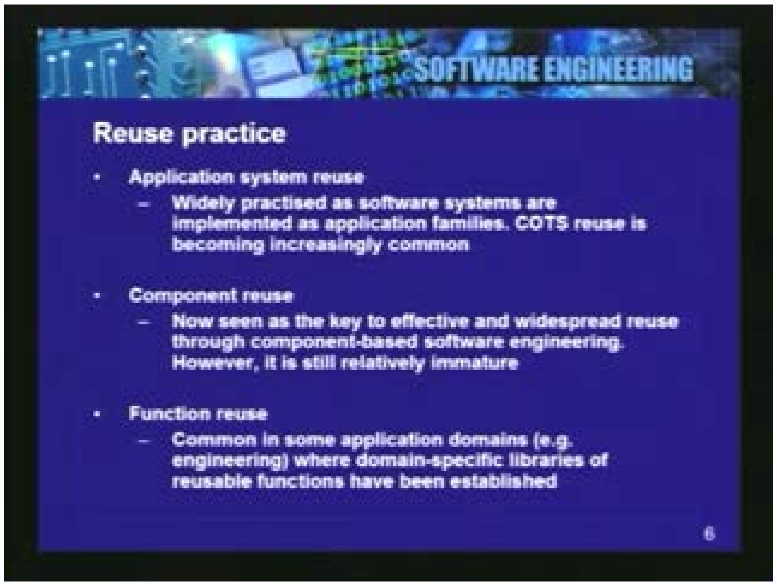
Reuse-based software engineering

- Application system reuse
  - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families
- Component reuse
  - Components of an application from sub-systems to single objects may be reused
- Function reuse
  - Software components that implement a single well-defined function may be reused

5

When you are writing a program that requires calculating the factorial you would not write your own code you will simply call this function of a library and this is a common form of reuse this has been done from quite a while but a higher level of abstraction for reuse may be that of a component. So the component embeds a certain set of related functionality into it and a good example of a component may be a text frame within a graphical user interface that is a good example of a component or it can be something much of a higher level; the component may embed a more series algorithm such as optimizing a certain function so optimizing a set of related functions and therefore you can reuse the entire component because you may need more than one interface of that component within your program and yet a higher level of reuse like we have seen before is that the entire application can be reused called cots reuse are common off-the-shelf applications that are taken as a whole and basically reused by adopting it to meet the business processes and the needs of your particular project.

(Refer Slide Time: 09:47 min)


SOFTWARE ENGINEERING

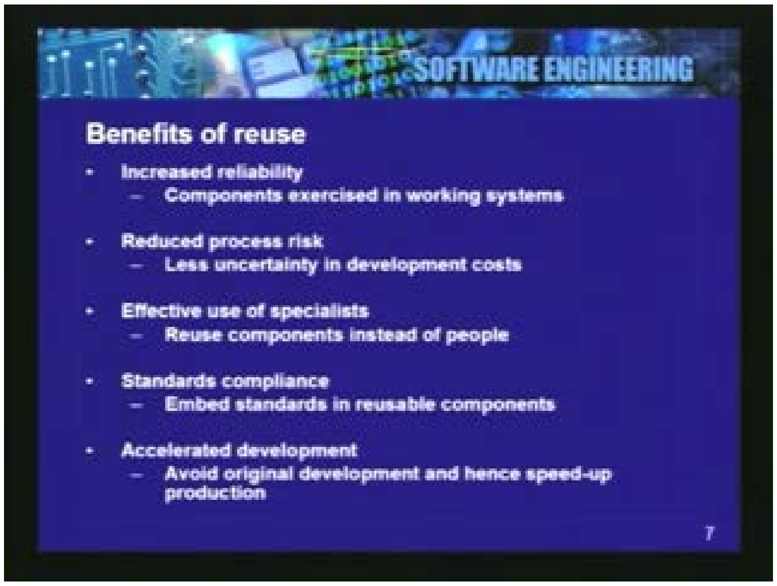
Reuse practice

- Application system reuse
  - Widely practised as software systems are implemented as application families. COTS reuse is becoming increasingly common
- Component reuse
  - Now seen as the key to effective and widespread reuse through component-based software engineering. However, it is still relatively immature
- Function reuse
  - Common in some application domains (e.g. engineering) where domain-specific libraries of reusable functions have been established

6

In practice basically common off-the-shelf reuse is becoming quite common. The functional reuse paradigm has existed from quite a long time. We have seen libraries that were written in C, we have seen libraries that were written for Fortran, there are common libraries available for engineering design problems that you can pick and use that have been available from the last two to three decades but component based reuse creating both creating components as well as creating systems out of components is a relatively new science, it is only becoming solid now; it is only over the past five years that component based software engineering as it is come to be known has become a science in its own right and people have developed ways of describing components which are quite abstract of gluing together components in a visual manner and so on that makes software engineering much simpler and much more reliable more importantly.

(Refer Slide Time: 10:52 min)



SOFTWARE ENGINEERING

Benefits of reuse

- Increased reliability
  - Components exercised in working systems
- Reduced process risk
  - Less uncertainty in development costs
- Effective use of specialists
  - Reuse components instead of people
- Standards compliance
  - Embed standards in reusable components
- Accelerated development
  - Avoid original development and hence speed-up production

7

So, just to summarize the different benefits that we can get use that we can get of reusing at various levels it does not have to be at any one level; the first one is obviously that of increased reliability that the component, the function, the system whatever it is that you are reusing has not only been thoroughly tested because that has been the specialization of the person who has written that component but it is also been used in other situations before in production situations and it has stood the test of actual systems. So, that increases reliability that has got out of reusing components and is something that is hard to beat.

Also, there is a reduced process risk and this kind of goes along with the accelerated development bullet that you see on this particular slide.

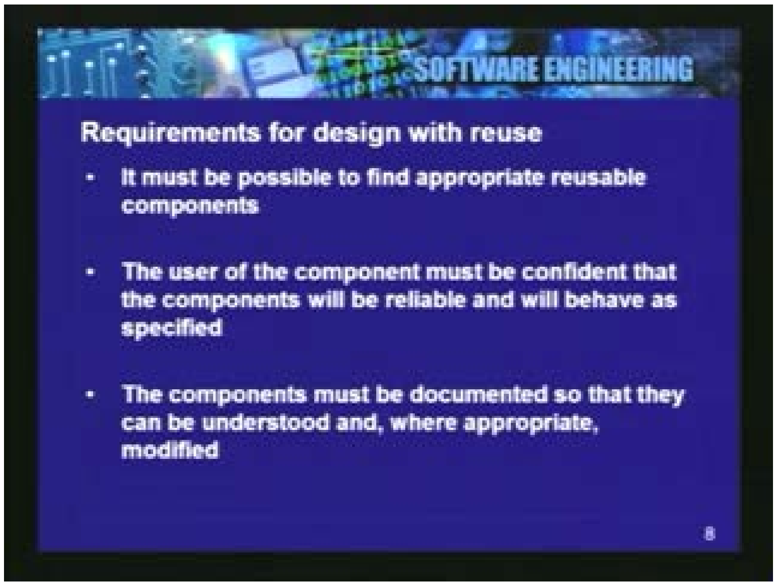
The process risk is that it takes away some of the uncertainty because you may not know for example how long it is going to take to develop a particular say user interfaces to develop a particular engine to do something. So, for example, it may be an ERP system that you are developing and you need a database design for the ERP system to store a set of to store a set of data. But if the database schema is already available and can be entirely reused from another ERP like project that was being done before then you will you know that at that amount of time you do not have to budge it into your process and it takes a risk of estimating the software development effort, it brings it down significantly and at the same time it accelerates development so that is also a process related benefit that you get out of reuse because you can proceed much faster; there the thing is that you do not have to do by hand that are already available, you may have to verify that indeed does what is it that you wanted to do. Those are some of the conditions as we can see to reusing components.

There can be compliance to standards because typically when a component is built to be reused in multiple situations it is built to some kind of a standard because that is the easiest way of ensuring that it is applicable across a wide variety of situations and across a wide set of systems and therefore when you reuse such components you automatically become compliant with the standards without having to make an additional effort and what you are essentially really bringing to the fore in case of reuse is that you are reusing that intellectual property that went into creating the component in the first place instead of having to bring the people who built that component and make them build it again.

You are basically capturing that intellectual property, you are you are making effective use of the specialist knowledge that existed so it may be a domain specific component that you are reusing; so, for example, something that would build a business process flow, automate a business process flow, this could be a backend server side component that knows how to control workflow; this can be reused in several…. it can be embedded as the workflow engine and the workflow engine now becomes the component that can be used across any project that requires the workflow.

You do not build your own workflow engine any more just like you would not build your own database; you would not even think about building your own database today, you would always buy a database and you will just build your own schema and insert it into the database. So, just as we have taken infrastructure for granted almost today the question is can we raise that level of abstraction and take application components, frameworks and indeed entire applications for granted and then build on top of that instead of doing all that all over again.

(Refer Slide Time: 14:34 min)


SOFTWARE ENGINEERING

Requirements for design with reuse

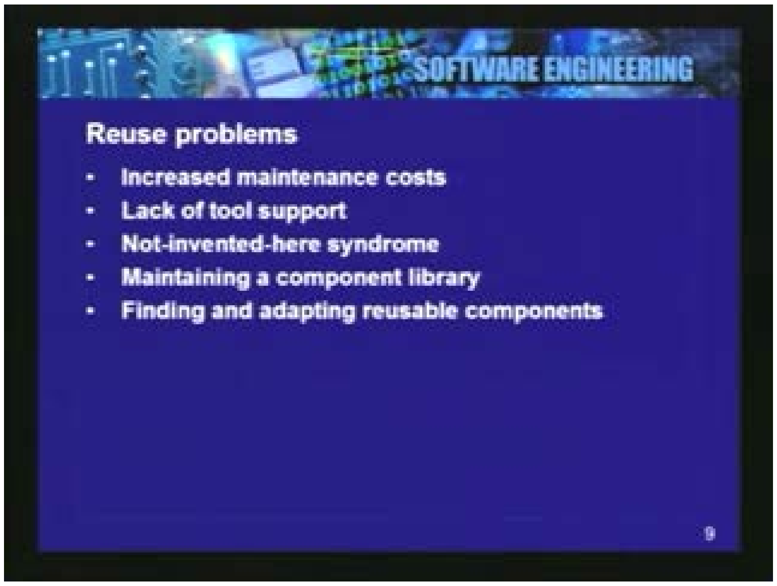
- It must be possible to find appropriate reusable components
- The user of the component must be confident that the components will be reliable and will behave as specified
- The components must be documented so that they can be understood and, where appropriate, modified

8

Now at the same time we have seen all the benefits it certainly comes with certain cause, there are some requirements for design with reuse. The first thing is it must be possible and easy to find the appropriate components to reuse. So there must be some kind of a component repository that has been set up, it must be easy to search that repository, the components must be very very well documented otherwise you may not be able find the components at all and even if you do find the likely component you may never realize whether it fully solves your problem. And there only two ways of going about doing this: one is you test the component thoroughly to figure out and you can kind of take it apart and figure out whether it meets your need or it comes with appropriate documentation and when it comes with the documentation then you do not have to spend as much time in trying to figure out whether it is appropriate for your need and that is the real benefit that you are going to save time at the end of the day and certainly you need to have the confidence that this component is complete, the component is solid in terms of in the terms of the capability that it provides; it has been thoroughly tested and that it will be reliable and it will behave exactly as has been specified and that confidence can come out of various means it can be for example certified by some kind of a Standard's Body that can say this component has been certified, it has been tested independently other than the persons who developed the component. The other way of gaining confidence is that it has been used in several situations production situations to build real systems before you are picking it up and using it in your particular project.

So what is the flip side of this?
So far we have seen all the things that are great with reuse and how you can save time and how you can save money and how you can build more reliable systems and so on but reuse comes with its own (…16:15) so why do people not adopt it is the question; if it is so great then why was not this something that was that had been adopted say twenty or thirty years ago. There can be increased cost of maintenance.

(Refer Slide Time: 16:43 min)


SOFTWARE ENGINEERING

Reuse problems

- Increased maintenance costs
- Lack of tool support
- Not-invented-here syndrome
- Maintaining a component library
- Finding and adapting reusable components

9

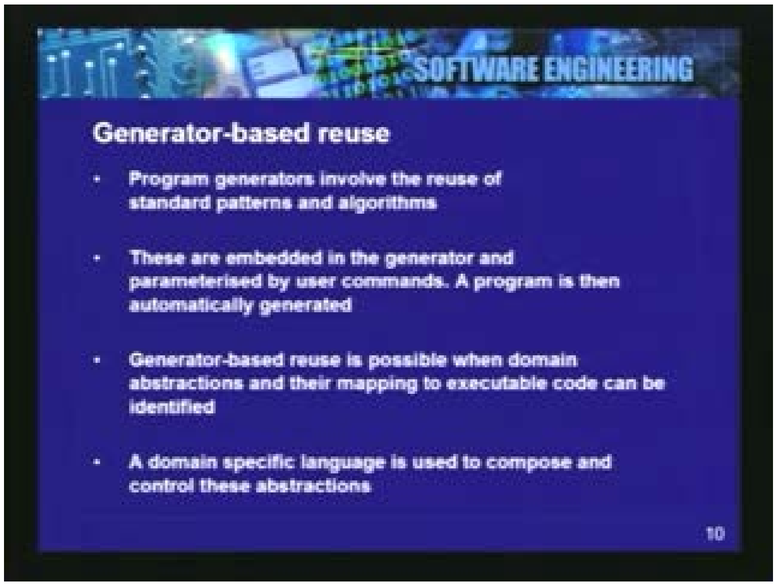
The one thing that you can point to is that if something goes wrong and now you suddenly end up losing confidence in the components that have been reused you might have to go, figure out whether you know this component is indeed the one that is responsible for the errors that are being caused, how do I fix this at this point in time because you will not have access to the source code of these components or the frameworks that you end up using, it is typically a black box kind of reuse that happens and there may be lack of tools support also. Most IDs today typically do not support the notion of visually gluing components together as opposed to writing codes; most IDs or Interactive Development environments are targeted towards allowing you to write code efficiently not necessarily targeted towards building systems by gluing together prebuilt components today. So the tools support is still not very good; lot of IDs today do have some level of reuse support built into them especially at the level of user interface components. But when you go into the backend server side part of the system you would not find much available in terms of tool support.

The third thing which may be one of the hardest things to deal with which is just because it is something that is hard to get your hands around is something called as not invented here syndrome or the nix syndrome. This basically deals with the programmer's attitude that they want to build everything as opposed to using something that somebody else has constructed for them it stems from several sources and we will not going in to that discussion here; this has kind of being well studied, several papers have been published in this area as well.

Then the cost of maintaining a component library there are there is no well-known place that you can go to buy software components; certainly applications are fairly well advertised but if I had a particular need for a component there is no universal component repository or a library that again goes and searches for and find one of these things. So it may be something that you have maintain at your own site and the cost of maintaining this kind of a library becomes quite expensive and the same thing is what this last bullet here is talking about is finding and adopting reusable component.

So adaptation is a different problem in the finding one. So you may find one but it may not it may not be something that suits your needs exactly and you may have to adapt to that component you may have to in object oriented terms you may have to redefine some of the interfaces of that component which means you have to understand how the component is put together internally that implies that it is no longer black box reuse that you are doing; you are not kind of taking the whole component as it is and plugging it into your system but you have to take it apart to understand what it is doing and then redefine some of the pieces of ex-functionality so that it becomes appropriate for reuse. At the same time you might be deriving some benefit out of the whole thing because the component exists and it has done some work for you already.

(Refer Slide Time: 19:46 min)


SOFTWARE ENGINEERING

Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterised by user commands. A program is then automatically generated
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified
- A domain specific language is used to compose and control these abstractions

10

So let us start taking a look at the different forms of reuse that can exist. Remember we said that you know there are different levels of reuse and this is related to that. The first one is that of generator-based reuse and these magnifies themselves in the form of program generators, code generators and so on.

The second one would be component based software engineering. We have already mentioned this. So you are reusing components; you are creating and reusing reusable components so that it is both sides of the equation designed for reuse and designed with reuse then there are frameworks that we can take a look at; application frameworks which are basically ways of reusing high level designs of architectures and some level of codes so there are abstract interfaces that are provided that have to be completely defined in your project but largely it is possible to reuse a lot of the design complexity that has been solved by somebody else.

Then there are what are called application families or product lines and these also end up reusing lot of the underlying infrastructure in the core ideas and finally we will take a look at fairly recent phenomenon called design patterns which basically allow you to reuse designs and just designs by themselves not necessarily code or not necessarily the entire systems.

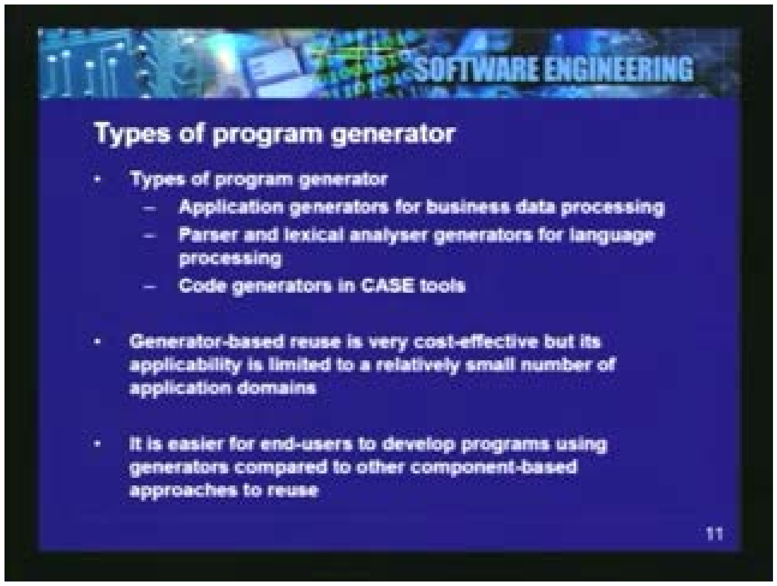
So what is generator-based reuse is the first thing. We find the notion of a code generator, a compiler for example is composed of a parser, a lexical analyzer, a parser and a code generator (….21:16) once it analyses whatever has been parse it is able to generate the appropriate piece of code.

Now if you extend this notion to being domain specific for example then we can generate entire program out of some kind of a specification and this in a way is the reuse of algorithms, the reuse of notions of designs as well because you can write out let us say a specification and if you take a look at financial systems as a vertical segment then suppose there is a program generator for financial system which very well understands some of the elements of the workflow some of the algorithms that are embedded within it so for example if an invoice has to be generated in a financial system tax has to be imposed on that invoice so there is an original amount to be billed and there is a taxable amount so that notion can be embedded within the program generator itself. It can also, for example, have tax tables that are built into it that it finds out what the tax is and then just plugs this into this particular program.

So the notion of reusing algorithms patterns that exist in certain vertical domains and embedding these within the program generator itself leads to what is called generator based reuse. This is possible certainly when there are domain abstractions and the mapping to code can be very clearly defined. Eventually what needs to come out of the program generator is executable code and when you can map these abstractions that you are feeding into the generator to the executable code pretty cleanly then you have a way of reusing lot of stuff that is specific to that particular domain.

There are different types of program generators, parses and lexical analyzers are very very common once that you end up seen lex and yak…. are the two common examples that you would end up using on building compilers so you would build a lexical analyzer or a parser; you would just write a grammar or you would write a piece of lex input and what the lexical analyzer now knows is it knows is how to recognize all those symbols that you have fed in as input. Same thing is the parser; once it reads the grammar it now knows how to parse anything that can be reconstructed out of the grammar. So the parser is being entirely reused the notions of building an abstracts, syntax tree out of the parser is an algorithm and the algorithmic reuse that is happening at this point in time.

(Refer Slide Time: 23:30 min)



SOFTWARE ENGINEERING

Types of program generator

- Types of program generator
  - Application generators for business data processing
  - Parser and lexical analyser generators for language processing
  - Code generators in CASE tools
- Generator-based reuse is very cost-effective but its applicability is limited to a relatively small number of application domains
- It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse

11

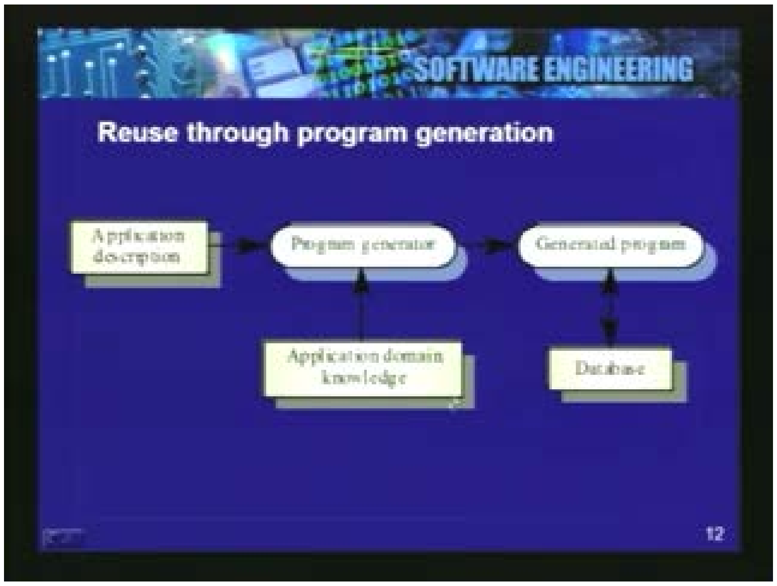
Also there are several cases where code is generated. So, if you take a look at distributed object technology which is a fairly new technology and core bugs is one of the standards that is available in the space there is something called an interface definition language in core bug.

An interface definition language allows you allows you to specify program interfaces in a high level specification language. So, for example, a bank account has two functions that are available on it: one is a withdraw function, one is a deposit function and the third function may be that of a querying function to find out what the available balance is at any given point of time.

Now you write out this high level specification without writing the code of what the withdrawer has to do and then it generates the backend code; at least the code framework can be generated. So you do not have to, for example, write the specification in C language, in Java language or anything like that; you write out in this high level English like language and then it translates it into a framework within a specific programming language say C or Java and at that point of time you will just fill in the details of the business logic and not worry about writing the code for the rest of the framework itself.

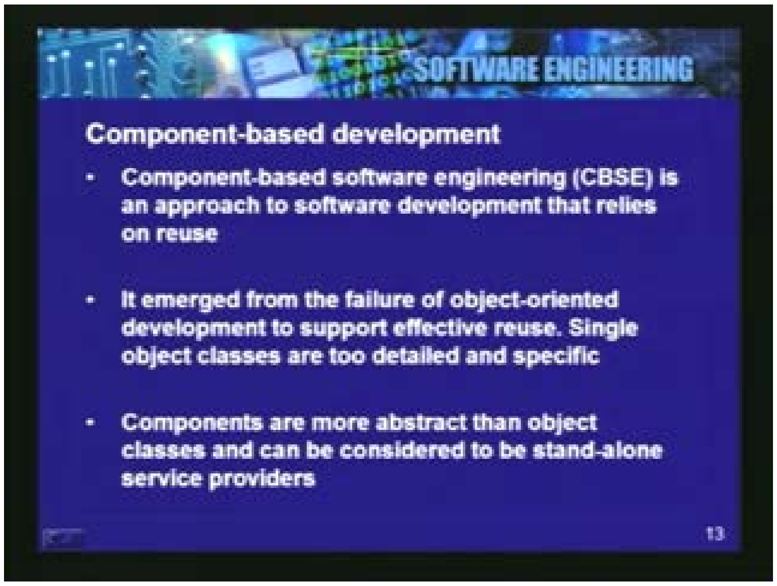
In that case, for example, the account has to be published so that somebody else can access it and all those functions are automatically generated by the code generator in this case. It is obviously a very very cost effective method of doing things but program generators are very specialized they are not generically usable everywhere so a few things that are fairly general as things like lexes and parsers but otherwise you take a look at code generators and case tools and things like that they are pretty pretty specialized and it is not very often used.

(Refer Slide Time: 25:43 min)



So the reuse through program generation the process looks like what is shown in the diagram here. You basically give the description of an application and the description of an application is fed into a program generator; program generator has some application domain knowledge that it picks up so this is the notion of reuse of algorithms; so, parser for example here knows what the notion of an abstract syntax tree is, it knows the algorithm to construct an abstract syntax given a particular parsing technique and so on. So through this you basically give a grammar in this case in the case of parser and then it basically generates a program in this particular case that can parse a particular language. So, for example, if you wanted to write a parser for Java you write down the Java grammar, you give it to a program generator and it would generate a compiler for Java in this particular case.

(Refer Slide Time: 26:30 min)


SOFTWARE ENGINEERING

Component-based development

- Component-based software engineering (CBSE) is an approach to software development that relies on reuse
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- Components are more abstract than object classes and can be considered to be stand-alone service providers

13

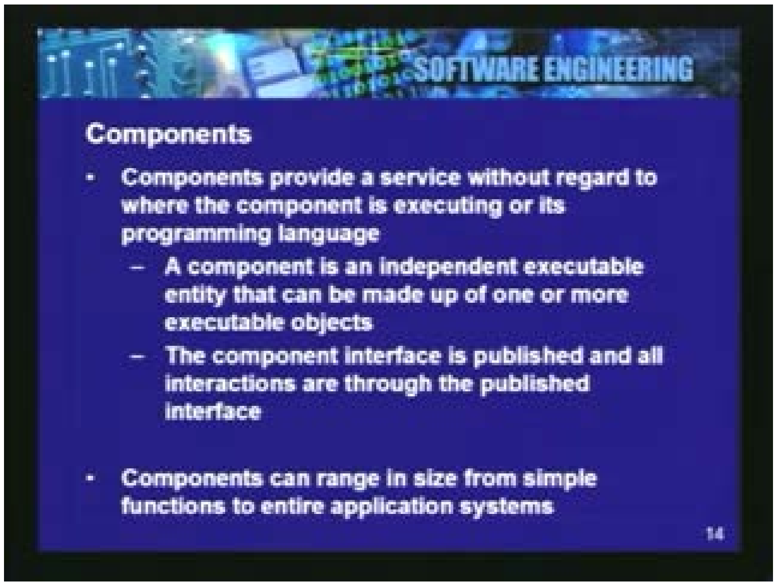
The second form of reuse that we are going to go into is what is called component based software development or reuse at the component level. This is an approach to software engineering that basically relies on reuse at its core.

Component based software engineering is the notion of reusing component and building systems by gluing together these components and often the level of abstraction has been raised to a point where the components are glued together visually in fact as opposed to even writing code and the visual gluing together of components generates code in the backend which will put all these components together appropriately. And components are slightly more abstract than objects.

It is actually, component based software engineering originated from the notion of object oriented design where individual classes and individual objects could be reused in the case of object oriented programming and design but what was needed was a slightly higher level of abstraction where an entire concept could be reused in the form of a component. So an example of a component may be an html page which has frames. There are certain parameters that can be fed into these components; I want frames in the left hand side or the right side, I want this many frames within this particular page, the colors of the different background colors of the frame and so on and so forth. But the component itself embeds multiple objects within it. It may embed the different frame objects, it may embed certain other notions that can be reused through the parameters that have been supplied to the user so it is a slightly higher level of abstraction than individual object; very much like an individual object except that it may combine multiple objects to perform a single set of functions; it is not a single function typically it is a single set of functions.

A calculator, for example, could be a component whereas the different objects within it could be things like a mathematical representation for data then there could be different functions on that representation for data and so on and all these things have been looped together in the notion of a calculator that could be a scientific calculator object there could be an arithmetic calculator object and these two come together within a calculator component for example.

(Refer Slide Time: 29:20 min)


SOFTWARE ENGINEERING

Components

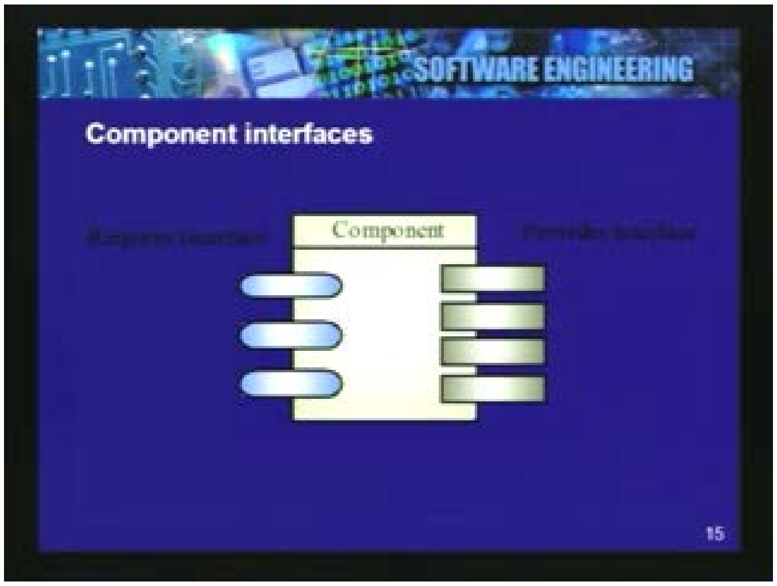
- Components provide a service without regard to where the component is executing or its programming language
  - A component is an independent executable entity that can be made up of one or more executable objects
  - The component interface is published and all interactions are through the published interface
- Components can range in size from simple functions to entire application systems

14

Therefore, the components essentially end up providing a service without regard to where the component is going to end up getting executed. That is, another aspect of component is that build and there can be many platforms on which this component will end up executing. So, for example, the Windows verses the Linux platform could be an operating system classification. So the component has and can be complied for either of these two platforms and when you install the component it automatically figures out as to what platform this is going to work on and generates the underlying code by itself and that is an that is what we mean by it provides a service at a level of abstraction that is independent of the platform on which it is going to run.

Then the interface, the component typically has two interfaces; the interfaces that it requires for it to run and the interfaces that it provides for somebody to reuse. We will take a look that in this diagrammatic representation.

(Refer Slide Time: 29:57 min)

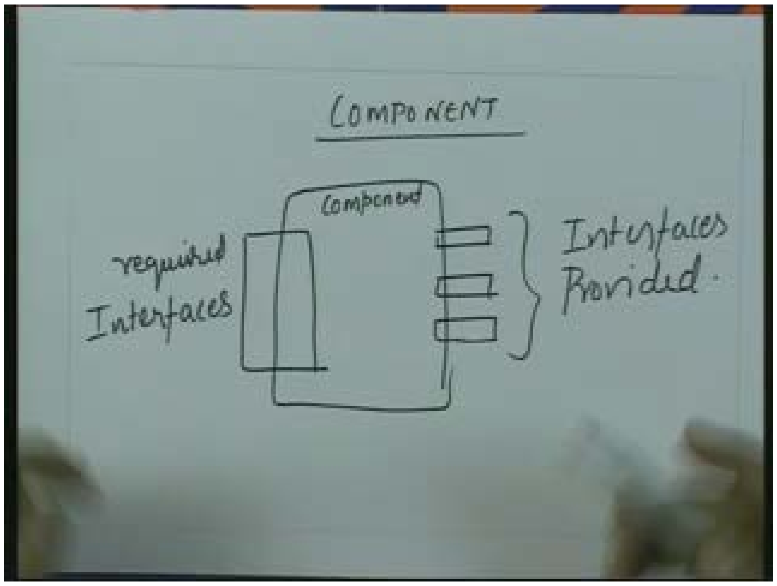

SOFTWARE ENGINEERING

Component interfaces

15

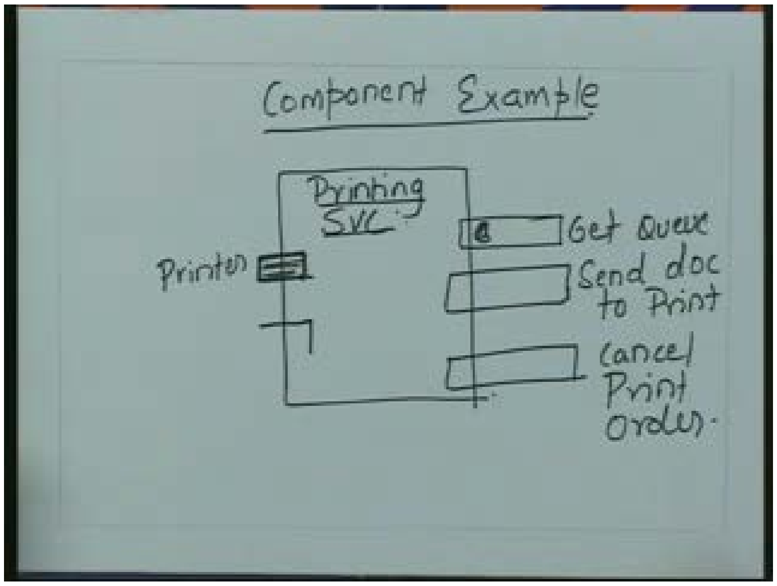
So the component in this case can be considered, let us say this black box is the component (Refer Slide Time: 30:10) it has two sets of interfaces; one is the set of required interfaces as they call it and one is the set of interfaces that it provides or provided interfaces.

(Refer Slide Time: 30:42 min)



So, we can extend this to a particular example and this example can be that of let us say a printing service component; a printing service component and the case of a printing service component it has a required interface which is that of the printer so this is a printing service which is a slightly higher level of abstraction and it requires the printer interface itself because it has to eventually send documents into the printer and it is going to provide interfaces which we will put out on the right side in this case where you can get the queue I will write it outside here so send document to print, one can be cancel print order and so on.

(Refer Slide Time: 32:08 min)

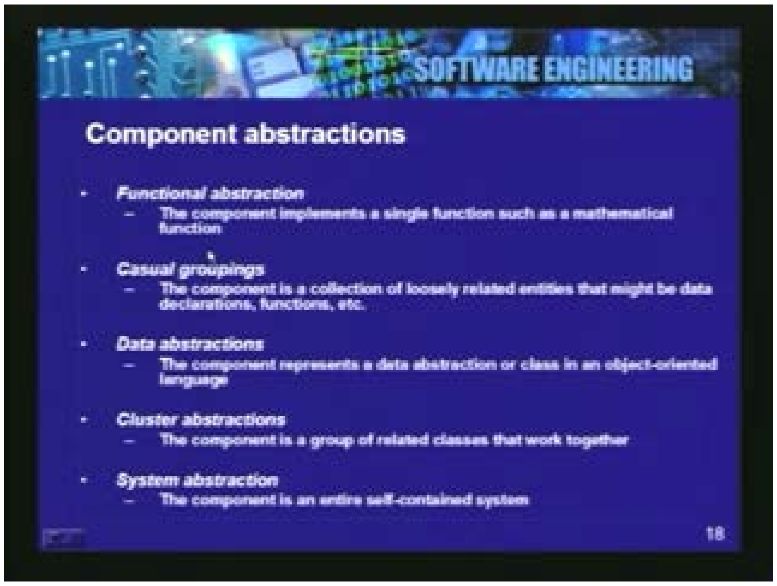


So here is an example of a printing service component which has a certain set of required interfaces as well as provides a set of interfaces that other people can use in order to accomplish and task. So the notion is that every time you want to have a service to print documents you do not need to build your own code to do this; you basically pick up this printing service document and embed it in the place that requires it and then use the interfaces that are been provided on the right side the provided interfaces and then you can just call them appropriately. You of course need to make sure that the required interface is also present.

So the analogy is clearly one of building hardware out of using prebuilt components such as chips. You do not build the gates for example within the hardware design today you have chips with three built pieces of functionality. For example, you may get a 4-bit adder, for example, you may get a piece of memory that stores certain 10 5 4 words of 8 bits each or a byte each and you can assemble a larger memory set by taking a bunch of these memory chips and putting it together; you can assemble a computer by putting together memory blocks such as this; even there there is a level of componentization.

Typically you get memory cards which contain let us say 4 mega bytes of memory or 8 mega bytes of memory and each card is in turn built up of smaller chips that contain may be a kilo byte or memory each and the entire memory card can be plugged in in an extensible way to create a computer that ranges anywhere from let us say 64 mega bytes of RAM all the way to Giga byte of RAM and you can extend that and this in turn uses the other components such as you know an arithmetic logic unit, may be a floating point processing unit, a CPU and so on and so forth, a disk driver and so on. So that is the analogy that you should be thinking of when you are thinking of component based software engineering; can I do the same thing that I do in hardware in building hardware to building software systems as well and that is what component based software engineering is all about.

(Refer Slide Time: 34:37 min)


SOFTWARE ENGINEERING

Component abstractions

- *Functional abstraction*
  - The component implements a single function such as a mathematical function
- *Casual groupings*
  - The component is a collection of loosely related entities that might be data declarations, functions, etc.
- *Data abstractions*
  - The component represents a data abstraction or class in an object-oriented language
- *Cluster abstractions*
  - The component is a group of related classes that work together
- *System abstraction*
  - The component is an entire self-contained system

18

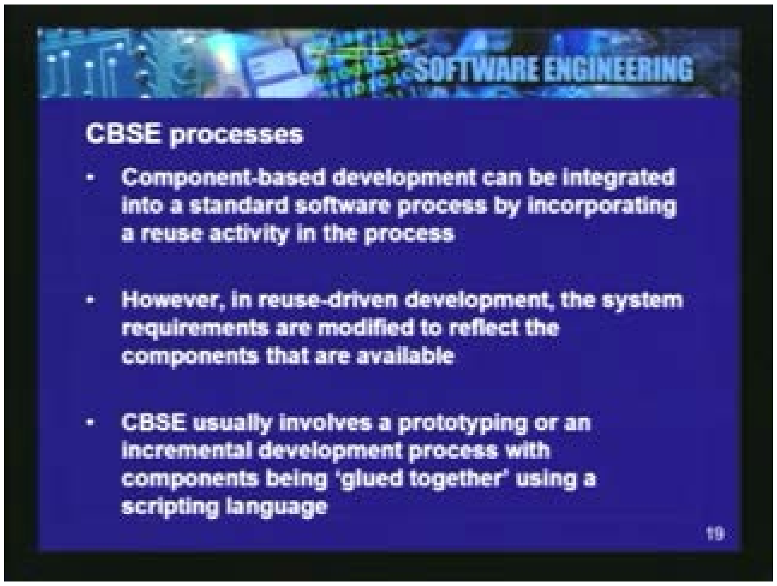
Components can provide different levels of abstractions as we have noted down in this slide. Typically most of them tend to be data abstractions or functional abstractions. so the component either implements a single function such as mathematical libraries that are available so there can be libraries of you know mathematical functions, there can be libraries of UI components, there can be libraries of you know communication components and so on or there can be data abstractions so things like data structures or lisps and stacks and sets and bags and hash tables and so on so these are data abstractions so that can exist as well. Rarely is the component of complete system abstraction that becomes cots type of reuse but for the sake of completeness we have included all the types of component abstractions that can exist on this slide.

The process is something that we will go into next. It can just be fitted into the existing software engineering process in the sense that you have a requirement stage and once the specification is done you typically have a design phase.

Now, here is where reuse really comes into the picture; you can either reuse it at its specification phase but more often than not reuse starts in the design phase and is most heavy during the implementation phases. So, given that what typically happens is you can say that you will extend the normal software engineering lifecycle to essentially have a reuse activity within the process. By this what we mean is that you have a design phase but as part of the design phase you first go look for components and if components are available then you just reuse them and then you design the rest of the system and go on; the same thing for implementation.

However, one should realize that this may not work very optimally. So, for example, you might go find a component whose specifications do not exactly match the design requirements that you have, whose specifications do not exactly match the implementation needs that you may have so it may be little bit more than what is required or it may be a little bit less than what is required.

(Refer Slide Time: 36:26 min)


SOFTWARE ENGINEERING

CBSE processes

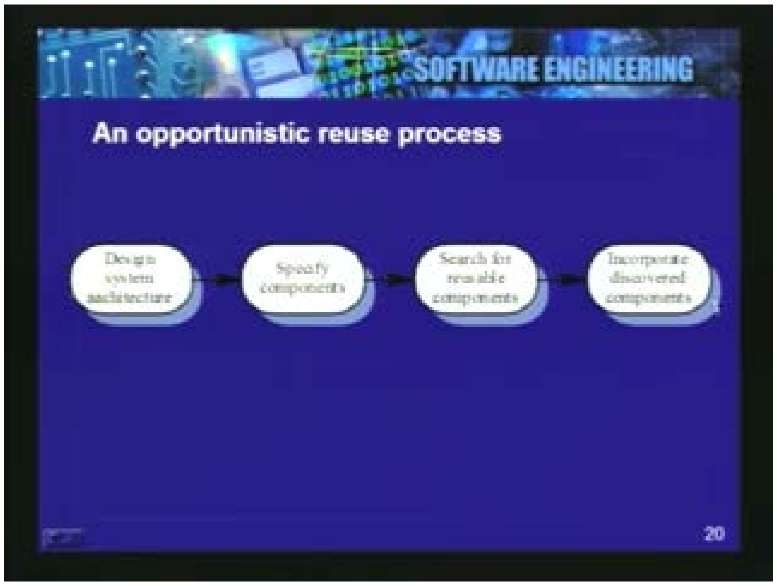
- Component-based development can be integrated into a standard software process by incorporating a reuse activity in the process
- However, in reuse-driven development, the system requirements are modified to reflect the components that are available
- CBSE usually involves a prototyping or an incremental development process with components being 'glued together' using a scripting language

19

In this case you have an option at this point: one is to put a feedback loop within the software engineering process and change these specifications of the system; not change the requirements exactly but change the specification of the system to meet what the components or the set of components that are available are giving you. So that is one way of going about doing it and something that is very important is obviously the prototyping effort in this case because you are getting many unknown what are essentially unknown quantities from different places and trying to put them together you should first have a prototyping phase or the prototyping phase becomes very important here where you are trying to first glue them together possibly using a scripting (37:08…..) quickly put them together in other words just to see whether there are incompatibilities that may exist within the different components etc etc and the components really perform according to specifications and so on.

Now, once that is done and you are kind of satisfied then you can kind of throw the prototype away and rebuild the system with the same components but now your glue is much stronger, your glue is may be a real programming language that has typing support and so on and so forth.

(Refer Slide Time: 37:45 min)



An opportunistic reuse process

Design system architecture → Specify components → Search for reusable components → Incorporate discovered components

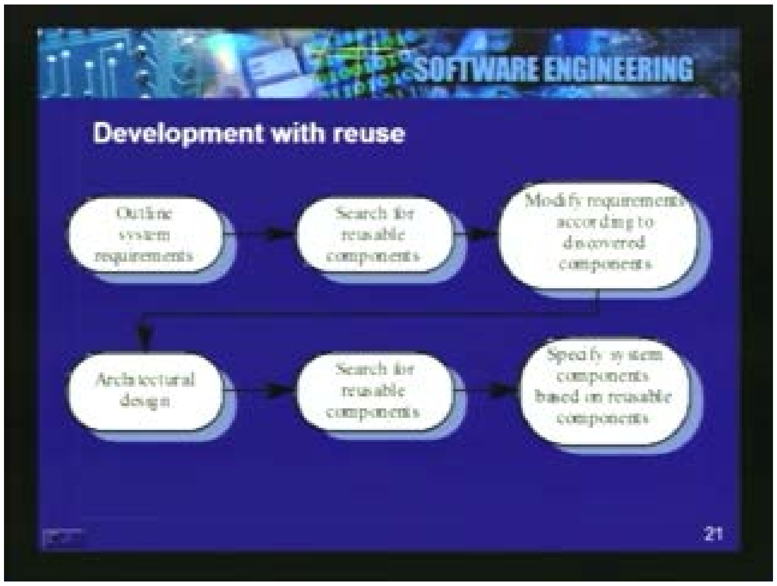
20

So an opportunistic reuse process like the first thing that you showed by simply extending the software engineering process would be to design the system architecture to specify all the components that would be required. The architecture itself is specified in the form of a set of components that are required and now for every component that is required you go out and search for components that may be reusable somewhere and if they are reusable you simply incorporate all the discovered components and then this entire flow can then be incorporated by saying the rest of the components might just now have to be built and that is the regular software engineering process at this point.

However, the development with reuse; remember what we said that the components that you may end up finding may not exactly be a match for what is that you are looking for. So once you outline the system specification you should really read requirements as specification in this particular case; once you outline the system's specifications you search for the reusable components and depending on what you find you may end up modifying the system specification according to what has been discovered and in this case….

For example, suppose you are looking for a lisp component, your system specification says you need a lisp or singly linked lisp in this case what you really find is only a doubly linked lisp is available as a reusable component so you basically modify the specification of the system at this point to say I am going to use a doubly linked lisp because it really provides all the functionalities that could be required and in this case is an example of the component's requirements are a superset of what is really needed within the design that you have created.

(Refer Slide Time: 39:19 min)

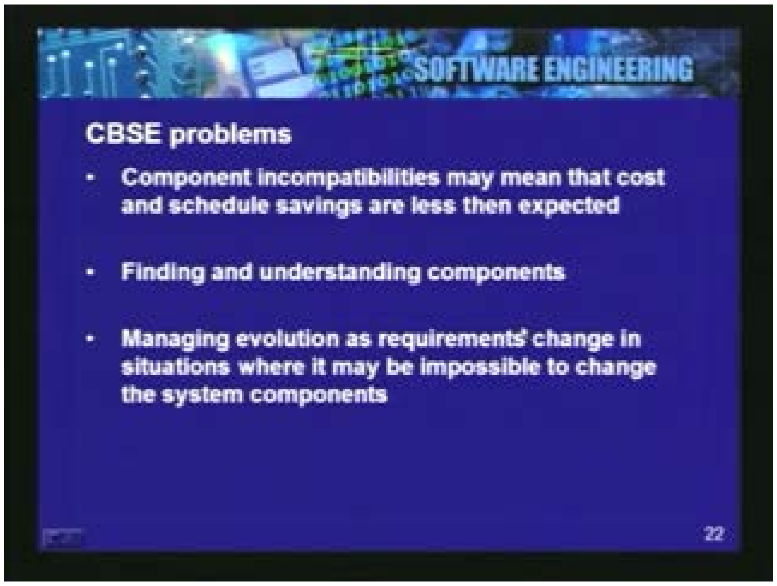

SOFTWARE ENGINEERING

Development with reuse

Outline system requirements
Search for reusable components
Modify requirements according to discovered components
Architectural design
Search for reusable components
Specify system components based on reusable components
21

There can be a situations where it is a subset and the component may now have to be extended a little bit as well and then you do the architectural design at that point in time using the set of components that have been discovered and now you search for other reusable components and so on and then you specify all the system components based on the reusable component. So this is the development with reuse scenario at this point in time and the last half of this process is not…… the search for reusable components here is different than the search for reusable components here (Refer Slide Time: 40:03) why; basically because the first part is referring to design with reuse and the second part is referring to design for reuse. So, one of the components that you may be building might look very very similar to a generic component that can be created out of this. So, what you should do is to then extend the effort to make this piece slightly generate if the project can absorb the cost because there is certain cost to be paid when you try to make something more generate and then you specify the system components that are based on reusable components at that point in time.

(Refer Slide Time: 40:39)



SOFTWARE ENGINEERING

CBSE problems

- Component incompatibilities may mean that cost and schedule savings are less then expected
- Finding and understanding components
- Managing evolution as requirements' change in situations where it may be impossible to change the system components

22

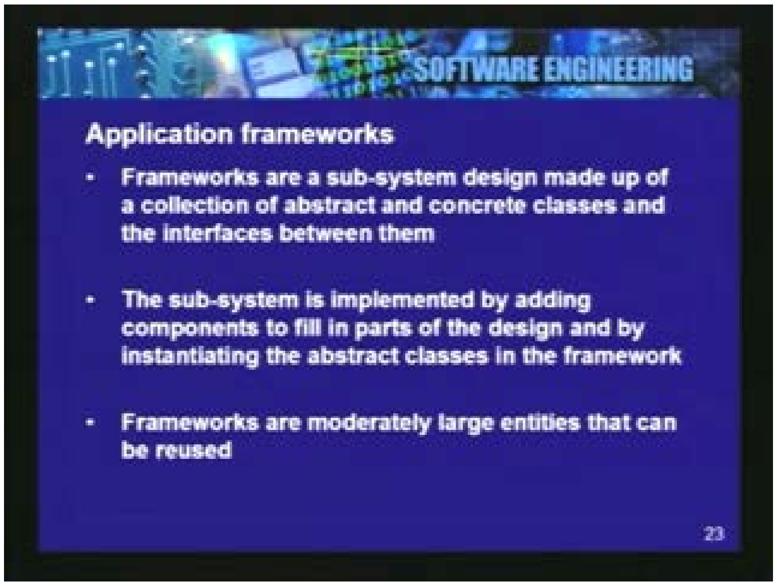
So, some of the problems that component based software engineering face today are that there can be component incompatibilities; you may get two components from two different sources and there may be standards in compatibility that may exist; even though they both provide the kind of functionality that you need one may be talking as slightly different protocol than the other one and the glue that you may have to write now to bring these things together may prove quite expensive; it may be in fact in some cases more expensive than writing the original piece of software from scratch so that is one thing that has to be taken into account.

The other thing is the old problem of reuse that we already saw which is finding and understanding these components. So, is there a standard means of describing a component; there is no component that exists today for example, for describing what a component does even though there are designs there are defacto standards such as UML that is available there are no component based software engineering standards for saying here is a description of a component that can be processed by a machine for example while doing a search.

The third thing is managing evolution of systems. If your system is largely built out of pre-existing components and now the system specifications change the system requirements change, the specifications change now you may end up in a bind because the existing components may not fit the new requirements that you have, the slightly extended requirement that you have and if no component is now available to match those needs you have to write that piece of code from scratch, write that part of the system from scratch and that can prove quite expensive so in some cases you would find out that evolution of a system to support a slightly incremental requirement is much more expensive than the original system that was built and that does not look very good.

So what are the again so those were the advantages and disadvantages of component based software engineering; fairly well-known methodology that is being adopted today increasingly; a lot of the IDs end up supporting that if you are using them on any kind of operating platform. The next level of reuse that we like to take a look at is the application frameworks.

(Refer Slide Time: 43:05 min)


SOFTWARE ENGINEERING

Application frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework
- Frameworks are moderately large entities that can be reused

23

What is an application framework is the first thing that you have to ask yourself. And an application framework is essentially a half implemented design to put it informal. What it provides…. It is a subsystem design and it is made up of a collection of concrete components that it embeds and some abstract components which have to be implemented when you use the framework. An example of such a framework is the interviews graphical user interface framework that is available from Stanford University and here is where they provide a collection not just of individual components by themselves that can be reused but there is interwiring between the components as well.

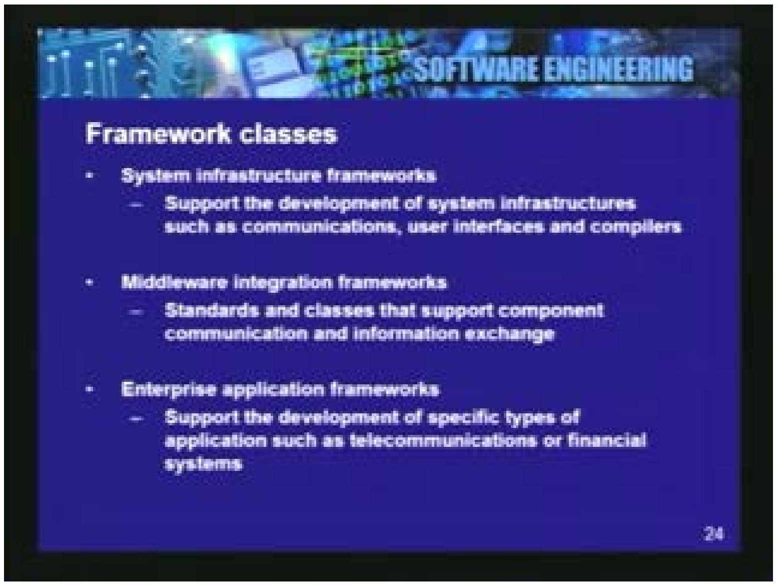
For example, it might provide the MVC pattern that has been partially implemented, it might provide other design patterns that has been partially implemented and there is some specialization there is some configuration that would have to be done if this framework were to be really useful. So you cannot just use the framework as is; if you use the framework as is it becomes a common off-the-shelf system that you are reusing at that point in time.

So what distinguishes the framework from a complete system is the fact that one certain abstract classes might have to be instantiated; you actually have to write some code that provides some concrete functionality for some abstract classes within the system. The second thing is it would have to be configured according to the system that were being used. A good example of this on the business side would be let us say there is an inventory management framework. What this means is that it is not an inventory management system; I just cannot take it and instantiate it and start using it to management inventory, I need to describe to it, the first thing I need to describe to it is the platform on which it is going to run, the second thing is every inventory management system embeds within it business process of how inventory is managed within the system.

So, for example, if the business process may be that of an item comes in it has to be approved by somebody then it has to be entered into the inventory system at that point in time and then it is tracked as to the movement of the particular item from warehouse to a particular store and so on and so forth. So that is the business process that is the control flow and unless that flow is described to this framework the framework is useless the framework cannot be used as is and when the flow is described to the framework what the framework does is it instantiates that particular flow; it makes it concrete, it makes it usable and that point of framework becomes the system that can be used directly.

There are fairly large entities, the level of granularity in a framework is quite high so it might act as an entire system for example, it could be a framework of an inventory management system or an ERP system within the industry or it can be a framework for GUI designs such as MVC designs and so on so it is fairly large. However, there has to be some work that is to be done by the person who is reusing that framework. So there are different classes of frameworks that are available. There are system infrastructure frameworks, there are middleware frameworks, there are application frameworks and so on depending on the level in the stack that you add.

(Refer Slide Time: 46:26 min)



SOFTWARE ENGINEERING

Framework classes

- System infrastructure frameworks
  - Support the development of system infrastructures such as communications, user interfaces and compilers
- Middleware integration frameworks
  - Standards and classes that support component communication and information exchange
- Enterprise application frameworks
  - Support the development of specific types of application such as telecommunications or financial systems

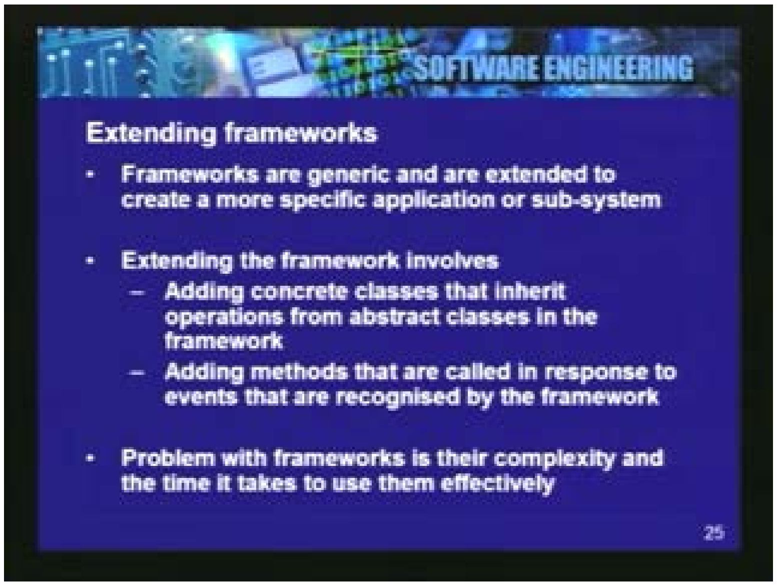
24

System infrastructure frameworks basically are things like communication frameworks so there may be a socket based communication framework, there may be a multicast framework that is available that supports multicast based on IP or something else and things like that. So there can be user interfaces frameworks and so on.

Now, the middleware integration frameworks are typically those that support a certain standard such as code bar for example it may support a J2E standard, it may support a dot net standard and it supports communication between components here. The framework for example, may go beyond simply a piece of middleware such as an ORB by saying I am going to provide reliability to this piece of middleware by replicating the object in multiple places. So it is a framework for object replication in other words. That is the notion of middleware at the I mean that's the notion of frameworks in the middleware level.

And finally the frameworks at the application level are those that are developed that support the development of specific applications. So inventory management was an application that we just talked about, there can be frameworks that do different types of vertical domains as well so a framework that does account management for example so something that have to be fed into that would be the account management, business process, the tax tables for the account management so it needs a lot of things just like a component has certain required interfaces and has certain interfaces that it ends up providing and the framework also has certain requirements the things that it ends up depending on as well as it has a set of services that it ends up providing.

(Refer Slide Time: 48:26 min)


SOFTWARE ENGINEERING

Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system
- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework
  - Adding methods that are called in response to events that are recognised by the framework
- Problem with frameworks is their complexity and the time it takes to use them effectively

25

So, frameworks are meant to be slightly generate because they can be instantiated in different conditions and different circumstances with different requirements so the framework can provide an entire family of application if you will, each one characterized by its own configuration, by its own business process, by its own business rules and so on. That is another example of the variability of a framework. For an application level framework there could be a set of business rules that are different within every component.

So, for example, the discount rules that are to be given when doing invoicing one company may say that we do not give any discount at all that is the rule, the second company that ends up instantiating the framework may give a business rule stating that if the invoice is over a 100,000 rupees then a discount of 5 percent is going to be applied to this.

The third company may say that any customer who has had business with us over the past three years to the tune of say 10 million rupees is going to get a 10 percent discount on any further purchases. So there can be different business rules that have to be provided to instantiate the framework as well.

Typically extending the framework would imply that you either do some development in which case you are adding certain concrete classes to the framework that actually implement some functionality that are provided as abstract interfaces within the system or you add certain methods that are called in response to events that are recognized by the framework. So these are called callback methods and every time the framework recognizes certain events; so suppose your total business volume has crossed a certain threshold now it is impossible for a framework to figure out what you want to do when you cross the threshold may be you want to set up an alert, you want to send an e-mail to a bunch of people saying that his business volume has crossed the threshold, may be you want to set up a press release etc etc. Now the framework writer who is writing a generic framework has no idea so what he does is he says that I am going to generate an event of type business volume threshold exceeded and when this event occurs then I am going to call one of these functions which is described in an abstract way so there is no implementation for this function, he has just called one of those functions and it is upto you as a framework instantiator to implement that particular function. So you have to write the code to receive that event and appropriately act on that event.

Another example could be that your average execution time for a particular servlet that was executed in the web [….5052] is suddenly spiked up by 50 percent and the framework an application management framework may give you the event and say it is up to do what you want with it; you can raise an alarm, you can try to take auto corrective action and so on and so forth.

The typical problems that exist with frameworks is the complexity. So the framework is not like a single component which can be easily understood. The printing service component was fairly simple in nature it had small set of interfaces typically two to three interfaces are what makes up a component; it may contain like four or five objects that build up that particular component. So it is not very large in size and it can be easily understood and it is also easy to discard and replace that is the other advantage of the component which is that if you find that the evolution of the system does not support that component then you can replace it with another one fairly easily. But a framework is a much larger commitment because of the size; it is almost an entire subsystem all the way to being an entire system by itself. Once it is instantiated it could form the bulk of your system. in such cases it is obviously not easy to replace the framework therefore evolution of the system may become much harder, also understanding the complexity that goes into creating one of these frameworks and therefore specializing it to meet your needs to configuring the framework for building these concrete classes that have to be added to the framework for dealing with events that the framework may end up throwing out are all very very difficult to do in certain cases. Obviously it has its own advantages. It is saving you a lot of development time, a lot of development effort, this has been again well tested so it is a level of reuse that is much higher than that of components but it also comes with a certain baggage that you will have to deal with. So, going forward what we are going to take a look at is the remaining levels of reusability (………52:49) we are going to take a look at design patterns and levels of design reuse and what are the benefits and disadvantages of doing reuse at those levels.