

Software Engineering
Prof. Umesh Bellur
Dept. of Computer Science & Engineering
IIT Bombay

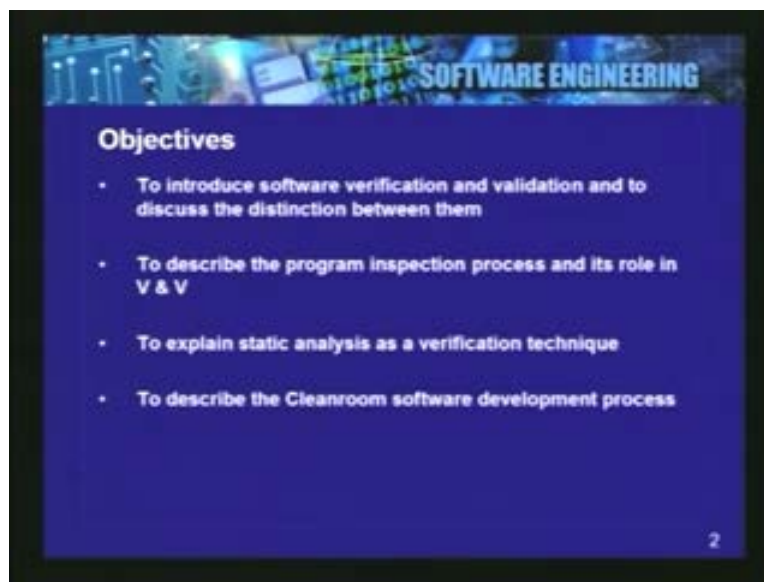
Verification & Validation, Inspection, Clean room development
Lecture # 22

Today we are going to be talking about the process of software verification and validation, and what these terms really mean over and above, what you normally do in the course of the software testing process.

We are trying to make sure in a way that the software does exactly what is meant to do and we look at the difference between what validation is and what verification is specifically and how the process differs from the process software testing.

We will also take a look at what is called the clean room software development process and this is a process whose philosophy is rooted in trying to ensure that the software can be gotten right in the first go, as suppose to having iterations where the software is released, bugs are found, it comes back, and the bugs are fixed and so on.

(Refer Slide Time 01:47 min)



What are the techniques that we can use to get software right in the first go, so that it does not have to be [c..not clear] and that is what we are going to conclude with in today's lecture.

The first question is what is really the difference between verification and validation and why do we want to take these two terms separately?

Verification essentially deals with the question "Are we building the product right".

(Refer Slide Time 02:14 min)



In the sense that, there is certain requirement specification that is being laid out, a requirement document that is being laid out, the documentation is being converted to a specification for the system and as long as that conversion is accurate you want to go back at the end of the cycle and check whether you met the specification of the software. And obviously if you are using automated techniques to generate the code from the specification then it will obviously match, but in the case where humans are involved in the process, we have to go back and make sure that we actually met the specification that we started out in the first place.

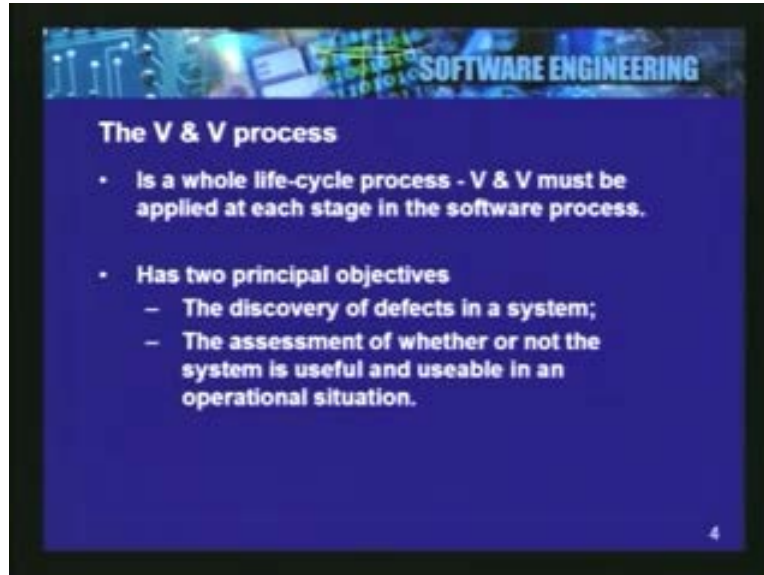
Now validation is the question of "Are we the building the right product?" That is, does it really help the user do what they want to do?

For example if you have a document editor of some kind, let us say it is a word processor and you do not have a spell check built in to that at all. This may be an issue which is a missed out requirement and these are the kind of question that you will try to answer with the validation process.

So you will go back to user scenarios in the first place and see whether whatever you ended up building at the end of the day, really meets the user scenario that were laid out to begin with.

This is also process that has to be applied across the lifecycle. So we cannot just do verification and validation at the end of the life cycle and hope that everything is going to have come out right at that point in time, because "what if did not?" - is the question.

(Refer Slide Time 03:51 min)



The cost of fixing something let us say that you missed out a requirement and let us say that you did not conform to the specification because you misunderstood it in some way, then the cost of finding that out in the end of the cycle is very high and the cost of fixing it is also high. The earlier you catch that problem and fix it will be cheaper for the software development process.

So it is a whole life cycle process in other words. It cannot just be done towards the end and it has to be applied towards every state within the process.

The principal objectives of verification & validation, much like testing for example, is the discovery of defects in the system is the first one; that is we find out all the bugs in the system and may not all be bugs that we want to fix and that is the second question that we should try to the answer. You should assess whether these are the bugs that affects for example usability of the system or that affects the user scenarios which the software is most likely to be used for.

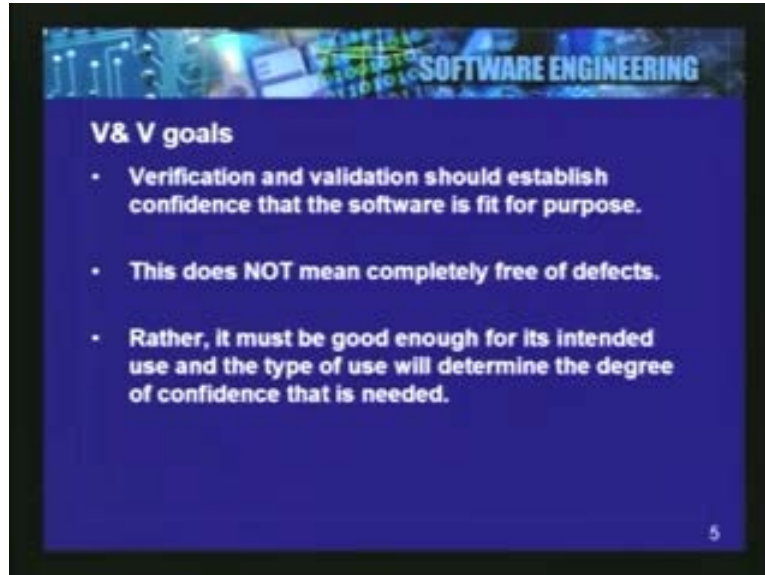
So, if you had a feature in there, for example that was only going to be used for say five percent of the time or it is some kind of a complex configuration feature and is going to be only used for five percent of the time, even if there are 1 or 2 bugs pertaining to the feature, then it may be okay to release the software at that point in time and that is the call we will have to make and this process helps us to make that call.

So what are the goals of verification and validation?

The basic goal is that it establishes the confidence that the software is fit for the purpose for which it is being built.

Going back to the first slide that we saw on this, what is trying to do is to ensure that we are building the product right, it is meeting the specifications or meeting the requirements that were laid out for the users and also it is going to try to ensure that this is the right product for the users in the first place, so that the software is fit for the purpose.

(Refer Slide Time 05:15 min)



The slide has a blue background with a header image showing circuitry and the text 'SOFTWARE ENGINEERING'. The title 'V&V goals' is in white. The content consists of three bullet points in white text.

V&V goals

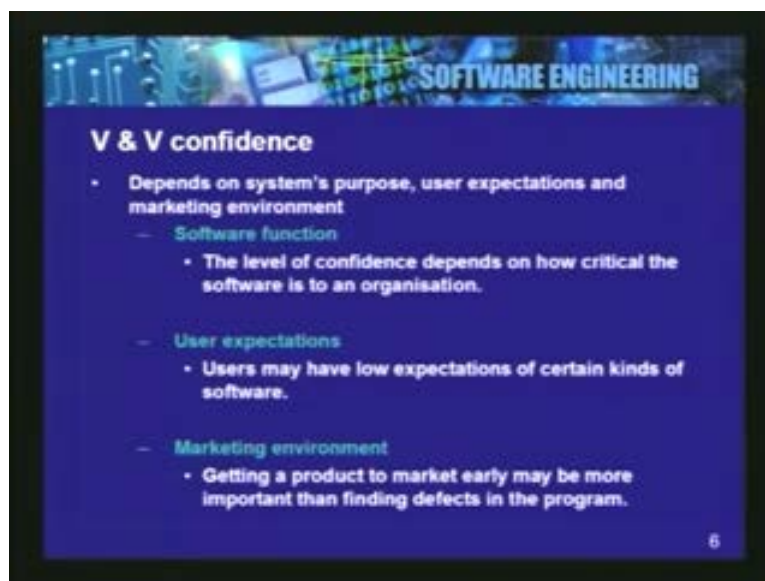
- Verification and validation should establish confidence that the software is fit for purpose.
- This does NOT mean completely free of defects.
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

5

Obviously it does not mean that the software is completely “defect free”. What it means is that, this can actually accomplish the task that it is set out to do. It must be good enough for the intended usage of the product and it will determine the degree of confidence that is needed as far as this particular software goes.

Depending on the systems’ purpose in this case, user expectations, marketing environment etc, what are the factors that influence the confidence level of a particular piece of software?

(Refer Slide Time 06:18 min)



The slide has a blue background with a header image showing circuitry and the text 'SOFTWARE ENGINEERING'. The title 'V & V confidence' is in white. The content consists of a main bullet point followed by three sub-bullets, each with its own sub-bullets.

V & V confidence

- Depends on system's purpose, user expectations and marketing environment
 - Software function
 - The level of confidence depends on how critical the software is to an organisation.
 - User expectations
 - Users may have low expectations of certain kinds of software.
 - Marketing environment
 - Getting a product to market early may be more important than finding defects in the program.

6

It is worth looking at that, because those are the factors that we have to end up attacking within the verification and validation process.

The first factor is that of the software function. What is it this software itself does? How critical is the software to the organization? What is the software function within the entire organization that this is meant to serve?

What is the expectation of the user is a second thing that would have to be considered. The user may have very low expectations of certain kinds of software or they may have very high expectation with the degree of automation that the piece of software is going to bring to them.

For example, if you are going to put out a word processor kind of software, it is not going to write documents on its own. It is unreasonable to expect something like that to happen. Whereas in enterprise automation software, not desktop software, basically the kind of ERP systems for example that exists within enterprise, the expectation is that it will largely automate the business process of enterprise and it will take the human touch away from most of this process and it become much more efficient to the result.

Then there maybe market conditions, the market conditions may drive the fact that the software product has to go to market early.

It may be at some point in time, if you are aiming for a “zero defects” kind of software, you may not be there in a **year**. But market condition maybe dictating that, unless you release the software in a particular window of opportunity, then there is no point in releasing the software at all. Because a competitor has probably beaten you to the market with pretty much exactly the same piece of software with somebody else is building or the need for software like this may not exist outside that window.

So it is either driven by competition or it is driven by the need for such a piece of software. It may become obsoleted because the fact that some other environmental conditions change and the software is no longer needed in the first place.

So getting to market early may be a concern and this is also something that is going to end up driving the confidence building process too: What is the level of confidence you need to have and what is the amount of risk that the organization is willing to undertake in order to release the software product into the market?

There can be two kinds of at the highest level **[dec..not clear]** of verification methods. The static and dynamic verification is that what we call them.

(Refer Slide Time 09:34 min)



Static verification is essentially something that is done with the code base and not with the actual product. For example you may be using the code base, you may be using the documentation, you may be using the design diagrams, and you may be using the specification and so on, but you are not actually playing with the built product, you are not testing the product in other words in any kind of dynamic sense.

Largely it has to do with software inspections. Software inspections can be supplemented by code analysis and so on. And all of these belong to set of methods which are static in nature. So code analysis through programs like 'lint' for example for C programs and there may be equivalent program for java as well. So software inspection is manual inspection of the code, the document and so on.

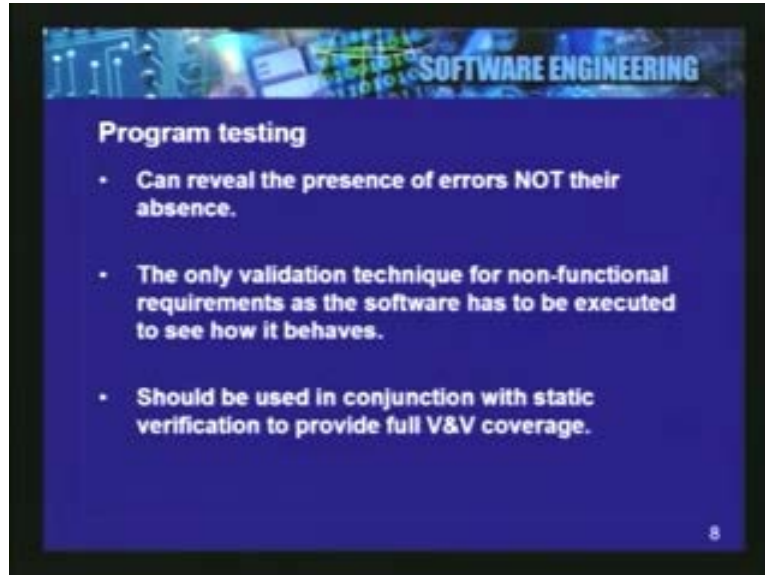
Then there is software testing. Testing is a dynamic verification process. In testing you actually write some test cases, drive the software through the condition that should be expected to encounter, and depending on how the software behaves, you determine whether the software is functioning properly and it meets the specification or not and this is completely dynamic. Basically it is actually executed.

Whereas in the static method it is not executed and purely some set of inspections either an automated inspection or manual inspection is done to determine whether the software is being built right.

Program testing can typically reveal the presence of errors; it cannot reveal the absence of errors. So how do you make sure that errors do not exist?

Certainly when you test the program, it can never be completely exhaustive. There can be hundreds and thousands of test cases for a program that can take say 10 inputs which can take in a variety of values and impossible to test every single set of values.

(Refer Slide Time 10:41 min)



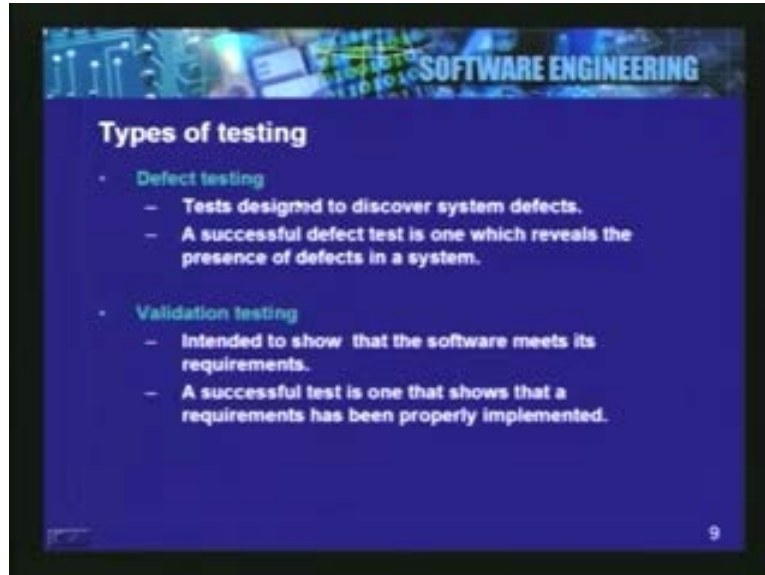
So the tests reveal certain errors that may exist. For example you may do a validation test, give it a wrong kind of input; it is a date and date is always expected in the day; the month and the year format and you feed the date in a different format and see whether it throws up an exception. That is one thing that you have to test, but certainly the other thing that you have to test is that variety of correct format that it accepts is also something that it is going to go through. So is there an absence of errors under certain conditions is something that you have to test for as well and program testing does not reveal the absence of errors, it only reveals presence of an error.

And as non functional requirements are concerned, the only validation technique is execution. It is dynamic in nature. Because performance for example, you can never predict although there are predictive or analytical techniques for predicting performance largely something like reliability or performance has to be determined only by running the software under certain conditions: How does it behave with hundred users in the system executing concurrently? Is it reliable if it ran for thirty days under this kind of load conditions? Etc.

So software or program testing which is the dynamic technique not something that we are going to go in here because you have already seen dynamic techniques and how to do OO testing etc. But it is obviously used in conjunction with the static verification techniques to provide complete coverage or complete validation.

So testing can also be of multiple type, defect testing or basically tests that are designed to discover system defects as indicated here. And successful defect test is one which reveals the presence of defects in the system.

(Refer Slide Time 12:11 min)



For example, if your test was checking to see whether you are accepting only the correct format of the input and the test actually showed that you are taking in some format that are not the correct format as well, then there are two things that can be tested there.

The first thing is that you give it a correct format in and it actually accepts and does not give out an error.

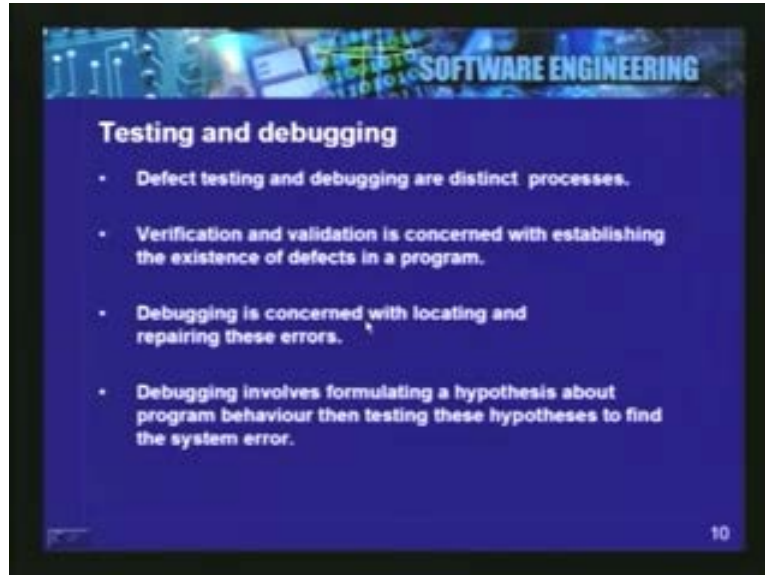
The second thing is, you give it a wrong format and ensures that it generates an error. So a successful defect test is one which actually reveals the presence of a defect or presence of a bug in the software.

The validation testing is basically intended to show that the software meets its requirements. If there was no requirement for example to validate the format of the input, then there is no point in having a test that checks the format of the input. Yet the system may kind be unusable really if you did not validate the format because different people would be entering these dates in different formats.

So here, a successful test is one that shows that requirements have been properly implemented and the system is usable and that is something that needs to be added to validation testing. Because in validation, “Are we building the right product”, is what we end up asking. So here it is not just that it reveals the presence of defects, but it is a test that shows and that it conforms to the requirements and the requirements are complete in nature.

Now debugging and testing are also slightly different processes, again very dynamic in nature. Here, you are putting the software through the phases and trying to determine whether it is meeting certain requirements.

(Refer Slide Time 14:22 min)



But, what is the difference between debugging and testing. They are distinct processes. Testing is meant to indicate that there is a defect in the system and debugging is a process of determining localizing the defect and removing the defect.

So while verification validation is concerned primarily with establishing the existence of defects, debugging is concerned with removing the errors, locating and repairing these errors.

Typically you end up forming a hypothesis. The way you go about debugging is; the system for example, is failing with this set of inputs. Let us say that you gave it a value greater than 10 for a particular integer value that it is expecting to take and the system fails after some point in time.

So you form a hypothesis saying, here is how the value is processed, here is what was done with the value. You kind of know the control flow of the program and how the inputs are getting used in the control flow. And based on that you form a hypothesis and then you test these hypotheses to figure out whether there is system error.

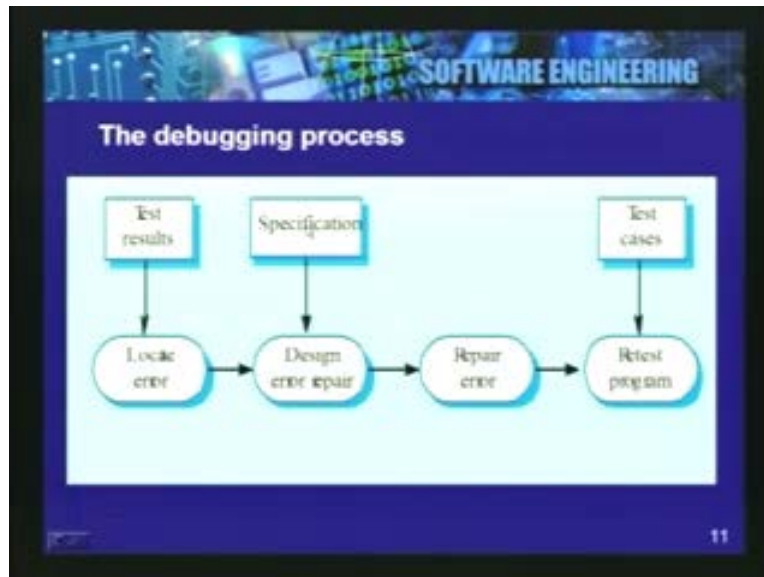
So you might sort localizing these. There maybe four modules in the system, you test the first module for example, because you feel that the validation of the input is not being done right and it is passing a piece of input that it should not be passing down to the rest of the program. If that is the case, you kind of just test that module by testing whether it is doing the validation. That is how the hypotheses do if the validation is not being done right.

So you form a set of such hypothesis and that is how debugging ends up getting done.

Here is the diagrammatic view of the debugging process. You have a set of test results to start with typically. You have to locate where the error is. The test results also typically indicate the software fail, not just that the software fail but where it is failed, typically the line number in the program is something that is indicated as part of the test result. You take that and then you take the specification along with that and figure out where you

ended up deviating from the specification: What is the specification and why is the test failing this particular specification?

(Refer Slide Time 15:55 min)



So you then design the error repair around that and you repair the error and then you have to retest the program with the same set of test cases that caused into fail in the first place.

An important fact here is that set of test cases have been generated, the program failed in those test cases, you fixed the failure and now it may or may not be just sufficient to rerun the original set of test cases in the first place. You may need to add certain tests, because you made certain changes to the structure of the program itself. Some of the tests may now become invalid because of the changes to the structure. So you have to be careful about that. But certainly the cases which caused the system to fail in the first place have to pass and then if any new additional test cases are added those have to run as well before you are done with the debugging process.

So having seen the difference between debugging and testing, these are all the dynamic methods of verification/validation that we talked about. Remember the static methods which focused on inspection, either manual or automated inspection, then there is dynamic method focused on testing. So we kind of looked at different methods of the testing, the difference between the testing and debugging process at a high level.

And the question is what else should be done? What is the overall process of verification and validation? What is the kind of planning that goes into making this process happen? First thing is obviously planning is required in order to get the most out of the testing and inspection process.

(Refer Slide Time 17:37 min)



Since you are also doing manual inspections of the code base of designs, discussing the design document before you move on, there is some degree of optimality that can be achieved by focusing your attention on specific pieces of the system.

Say, if there is a very critical part of the system, maybe only you want to do a formal specification and go down that route. Or may be there is a piece of the system that is being thoroughly inspected and you have a high degree of confidence in that part of the system may be the testing does not have to be that exhaustive in that particular module of the system and so on.

So, the planning is something that has to be done and it should start very early in the development process. In fact the V&V development process, the verification/validation development process emphasizes that you start writing test cases as soon as the requirements are written, as soon as the specifications are written.

So, the testing process does happen in parallel with the development process. We should take a look at the process itself in a little while.

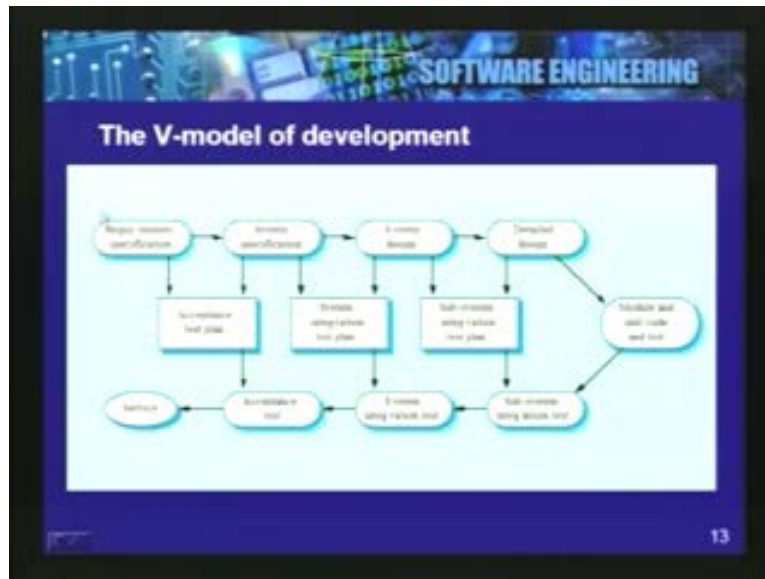
And certainly there should be a balance between the static verification methods and the testing methods or the dynamic verification methods, just like we talked about little while ago.

And finally test planning is all about defining standards for the testing process. For example what is the test environment going to be like, what degree of scalability testing are you going to end up doing?

Say, if this system is going to take hundred thousand users at any given point in time and you are not going to be able to test it for hundred thousand users. So you have to have some means of building a confidence that when tested for hundred users, you can somehow extrapolate those results to ensure that the system is going to work with same level of performance with hundred thousand users.

Is this system going to scale? - is the question that is going to be asked. And so in test planning you try to build confidence that the test that you are doing are indeed going to let you know that the system is going to scale or not.

(Refer Slide Time 19:49 min)



Here is again a diagram of what is the verification model of development.

You start off with the requirements specification in this case. With requirement specification you can start writing the acceptance test plan because the requirement specification is done by the customer and the customer is ultimately the one who is going to accept the system.

So the requirement specification will lead to the acceptance test plan, which in turn leads to actually doing the acceptance test when the software is ready to be acceptance tested.

Then the system specification is written, system specification can also go towards the acceptance test plan, because in some cases the user maybe involved the system specification, but more often that it will take the right side arrow (Refer Slide Time: 20:33) of this particular box coming down into a system integration test plan.

Because once you specify the system as: Here are the four different modules, here is the specification of each one of the modules of the system, that is what the functional specification looks like. In that case you have to have some kind of an integration test plan that is built around this specification and design.

And then the detail design or the system design and specification together go into creating the system integration test plan. Because the design is also responsible for modularizing the software, it is responsible for breaking it up to say that, we are going to use these components, we are going to build these modules and the modules themselves or the components themselves maybe individual units of work that are put together by different people.

Even though they may be tested individually, there has to be some test that covers the cohesiveness of these different modules. Once they come together, is it clearly fulfilling

the specification even though they work very well at unit level? That is what system design at specifications end up driving the integration test plan.

And finally the detail design may typically ends up driving the unit test plan, which is the last box on the right side of the diagram. But it may also contribute to some subsystem integration test plan. So, if you hierarchically breakdown the system into subsystems of lower and lower complexity, then there maybe some subsystem integration, but at a high level you may not end up seeing the subsystem integration test plan box. What you see is acceptance test plans box, a system integration test plan, and finally the unit test plan and pieces of it.

So, the detailed design will drive the unit test plan, and as soon as the code is done the unit test around on the code for each module; this is done on an individual module level, and once the code has been unit tested or the components have been unit tested, then subsystem integration and system integration testing is done using the appropriate plan; either subsystem integration test plan or a system integration test plan.

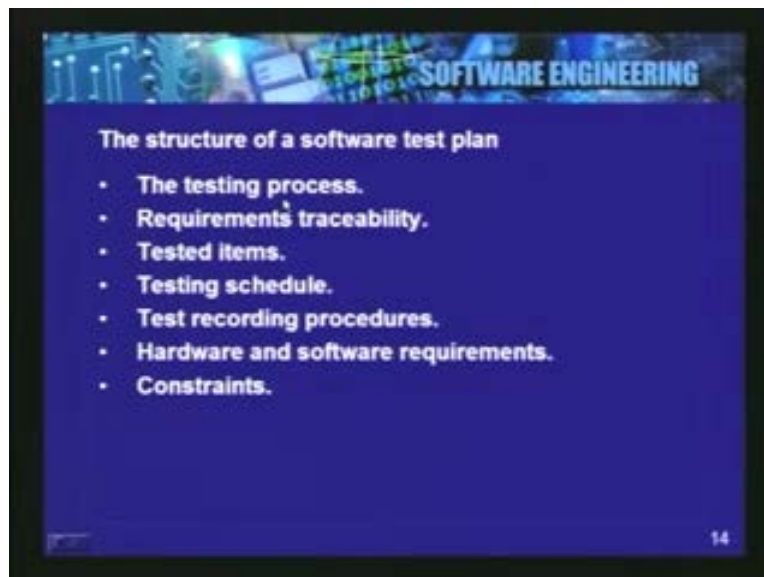
And once this is done, it then goes into acceptance testing using acceptance test plan that you have already created before.

So it is kind of like a V-process, start from the requirement specification and goes back to the product or the service that is being delivered to the customer and what are the different stages of testing as it is integrated with the development process itself is what this is trying to show you.

Now, this is something that you might have seen during the software testing part of it, but here the structure of a software test plan.

The software test plan primarily focuses on what is the testing process going to be.

(Refer Slide Time 23:23 min)



For example, how are we going to test software?

It also focuses on things like: What is the test environment going to look like? Are you going to have a full scale test environment which exactly duplicates what the production is going to be like? Are you going to have a scaled down version of this? If there is a scaled down version, then is there a process to extrapolate the results to get at what this software is going to behave like in the fully scaled up version under production conditions?

Then it is typically going to have to test every single requirement, so there needs to be some kind of requirement traceability. Let us say as; here is the test that tests the requirement 1.2, here is the test that tests the requirement 1.3 and so on.

Because, at the end of the day, you have to have confidence at every user requirement has been tested and works.

So, what are the different items that are being tested? What is the testing schedule? - is one thing that it puts together.

If there are 450 tests that need to be done because there are 450 requirements and each test kind of breaks down into 10 other sub tests, then what is the schedule?

All these are going to take time and going to take resources. Now, what is the plan going to look like? What are the recording procedures? It is not just sufficient to do the tests, but you have to record the results from a test in a very systematic fashion, so that you can go back and analyze the results at a later point in time. So, what is the recording procedure going to look like, what exactly in fact the template of the test results is going to look like, is very important to specify upfront.

And then what are the hardware and software requirements for testing? Do I need just one server? Am I going to simulate the system or I need a bank of servers, because I wanted to run the real system on here? Is there a software test harness that is required etc? Or there any constraint from the testing for example, you cannot test a particular requirement because it is testing let us say the degree of user friendliness. It is a very subjective kind of a requirement and maybe it is something that cannot be tested objectively.

So, we just went through the testing process. Let us come down to this software inspection again.

We have seen in testing remember, we broke up the entire verification/validation into static and dynamic process. Static process dealt with inspections and dynamic process dealt with testing. And we have gone through the dynamic process.

Coming back to the static process, which is inspection either by manual or automated and it is a very key feature of how this is done.

So software inspection essentially involves people examining the source representation.

By source representation we mean, the code, it can mean design documents or it can mean specification which is used to generate the code, or it means an architectural document etc.

(Refer Slide Time 25:51 min)



The main goal of this is to try and improve the design, improve the code, improve the architecture, improve the ways specifications are done, and so on.

So the aim is to discover anomalies or defects within the source representations. It is typically done before implementation so that you can catch things upfront. And earlier, you do it within the process, the more beneficial this is likely to prove because, a bug or software defect or an error that is caught up early in the cycle is going to be much cheaper to fix than we have already seen.

It can be applied to any representation like we have already seen before and it has been shown to be an effective technique for discovering program errors. It cannot necessarily mean the coding errors; there could be a fundamental design flaw in the system for example that shows a security breach occurring in the system.

Design flaw could be that you are not doing any kind of encryption between two different processes within the design. This could essentially result in a potential security breach in one process translating to a security breach in the second process because, there is no encryption going on between these two. Or the data flowing from one process to another is subject to snooping by a third party and can potentially corrupt the data, attack the system and so on.

So, there can be a design flaw or there can be a coding flaw. For example, somebody can spot a fact that the loop is never going to terminate because one of the conditions that they can think of which can occur in the program is going to cause it loop forever.

So there can be different kinds of flaw as we should take a look at shortly. Again this can be an incremental process, it can be a process that take place several times because you could end up finding defects in the first part of the design that you are discussing or the code that you stop inspection at that point and you go back, fix these, comeback and continues the inspection. Or you know typically what may happen and this is to unit testing one defect may mask several other defects.

(Refer Slide Time 28:20 min)



So, in test when the first defect is found, the test stops at that point of time so that you can go back and fix these errors. Just like several executions of test run maybe required, several inspections can be required to actually catch all the defects.

The second point that is important here is, the reuse domain and programming knowledge that exists and reviewers are likely to have seen the types of errors that are commonly committed. For example, one of the things that are commonly done in coding is that, a variable is not initialized before it is used and so this is something that people know to watch out for when they are doing an inspection. As result of which they are likely to catch it pretty quickly.

Other thing may be coding standards that are set up by the organization. Typically a coding standard may say that there is a certain naming nomenclature that has to be used or a nomenclature that has to be used throughout the program.

So, for example module/method are named by 'interCap' notations could be a particular coding standard and violation of that can be easily be caught by people because they have been doing this for a long time and they know exactly what to look for in this particular case.

So once again inspections and testing are complementary process. They are not something that is mutually exclusive, so both of them have to be done. Inspections are typically applied at a higher level of design. It can be applied to code, in fact in some cases it is applied, but it is not trying to catch code errors such as the infinite loop of a program etc. Typically, it is meant to catch design errors that of a higher level just like the security design error that we just saw a little while ago. And both of them obviously have to be used during entire process.

(Refer Slide Time 30:25 min)



So inspections are something that can check conformance with a specification. Actually the third bullet in the above screen is not completely true in the sense that, inspections can understand the true intent of the customer when they specified the requirement. Because there are humans involved who are sometimes doing these inspections, whereas testing is typically automated. It is an automated process and you have written the test and it is hard to understand. So in testing, you are going exactly as per the specification and you may not be able to understand the true intent the customer when they specify the requirement.

So, in that case again inspections can check the conformance with the specification and also it can go beyond that. It can be used in a situation where in there is a human in the team basically an architect, there is a business analyst, system analyst who understands what the customer meant when they wrote down a particular requirement. In that case they can bring that knowledge to bear on the inspection process and make that to be that much more useful.

The disadvantages of inspection are, obviously it cannot test non functional requirements. So, it cannot test user friendliness, it cannot test performance, it cannot test reliability and so on. This is because they are not actually playing with the software at that point in time. They are not executing the software, they are not executing the code in order to test for a particular property. Whereas in testing you can catch them and there can be test for non functional requirements as well.

A test for example that will measure the end to end response time of a particular request and if it exceeds the threshold, you know that you have broken a non functional requirement very clearly. But that cannot be done during the inspection process. However there can be hints that the particular piece of program code or a particular design is going to be too resource intensive.

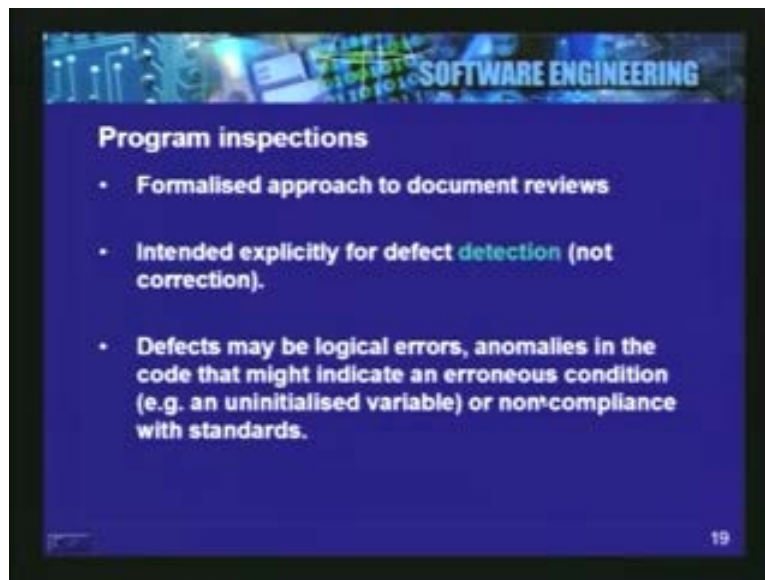
In other words, say you can make out that an algorithm that is going to be used in encryption is far too CPU intensive, so given a fact that there could be a hundreds of user banking on the system simultaneously, then that is the not the right algorithm to be used.

That kind of a decision making can be done with inspection process much earlier on in the life cycle.

So program inspections are basically formalized approach. They are intended to explicitly check for defect detection and certainly it is not going to help correct it, but all those suggestions may come out during the inspection process as on how to correct the particular defect.

And typically we are looking for high level logical errors, we looking for design flaws, we looking for architectural suggestions, the choice of particular component, and the choice of particular data structure on a certain situation and so on.

(Refer Slide Time 32:50 min)

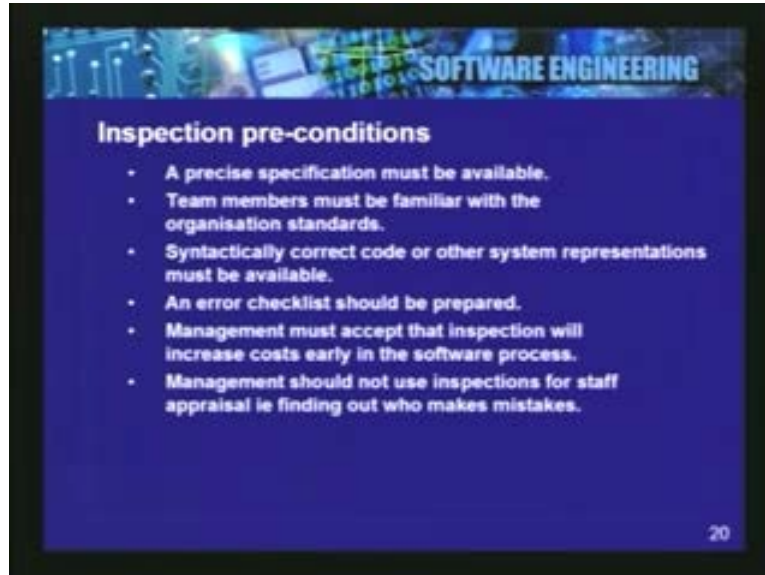


But we are not looking for very small code anomalies such as this is an un-initialized variable and so on. There can be automated tools which do some of those things as we shall see in the next part of talk.

Continuing on, what are some of the preconditions for an inspection to take place?

We shall take a look at the precondition, then we shall take a look at the inspection process; what are the kinds of errors that you are going to likely look for in an inspection and so on.

(Refer Slide Time 33:33 min)



The preconditions are that; a precise specifications must be available because you are always going to refer back to what is it that this module is supposed to be doing, what is it that this software is supposed to be doing, what is it that this function is supposed to be doing and so on.

So a precise specification must be available.

Then whatever standards that is it that you are checking against. Suppose you are checking a coding standard, suppose you are checking a design standard that the users' design pattern is appropriately being met etc. Those guidelines must be published appropriately and must be published in advance and must be pretty clear. And all the team members involved in the inspection must be familiar with these guidelines.

Also syntactically correct representation of the source must be available. There is no point in going through a document which is full of spelling mistakes for example. There is no point in going through a code which is not going to compile. Since the compiler can automatically check this using machine, there is no point in human sitting down and trying to figure out whether this code is going to a compile or not. That is not what an inspection is meant to do at all. Inspection is meant to catch logical errors in the program. So it should have passed the compiler test already.

The error check list must be available well in advance.

What are the kind of errors we looking for; that categorization should exist.

And we shall take a look at an example categorization. Obviously there has to be management buyoff in the whole process.

A. Because this is going to increase cost and no question about it, because it is a team of designers who are sitting there and not actually designing, but reviewing the design of somebody else. So this is going to cost them, but the cost that is born early in the process

likely to save a lot more cost down the line. And that is a fairly proven fact through empirical evidence, something that needs buy-in from the management.

The last thing which is probably the most important is that this should not be used by management as a blame pointing exercise. So in other words this cannot be used to point fingers at a programmer or a designer and say “see you are doing such a terrible job, you are going to be fired the next year or whatever”.

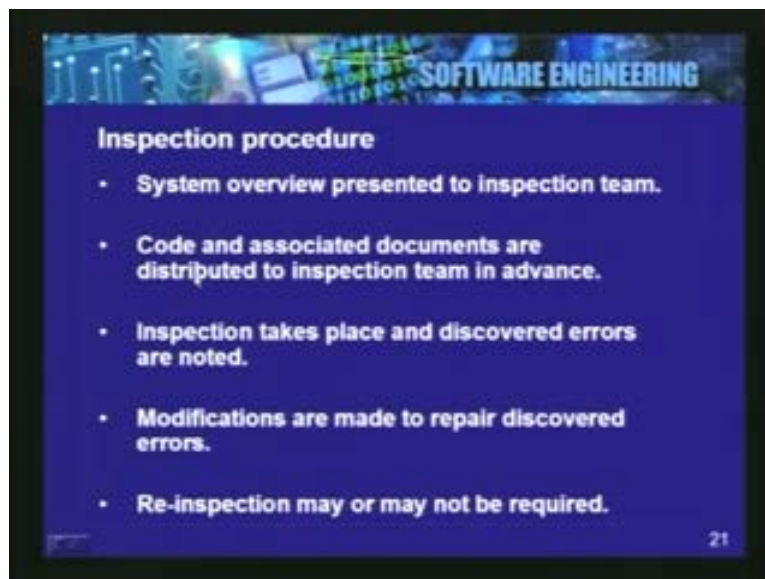
It is not meant for that. It is meant to fix the software. It is not an appraisal tool in other words and we have to be very careful about this, because often this is what ends up happening within software development.

So what is the procedure itself looks like. Procedure is fairly straight forward. The system overview is first presented to the inspection team. Not the developer, but somebody else would present the overview and we will take a look at the roles in the next slide.

So, somebody presents the modules that is being reviewed, what is the overall goal of the module, what is the specification of the module like, what is the module design look like that is now been reviewed and so on.

And then, all the documents should have been presented well in advance. Actually this is not something that takes place after the system overview.

(Refer Slide Time 36:18 min)



The code and associated documents should be distributed to the team well in advance, so that they could study this and come forward with their opinions already.

Then the process inspection itself takes place. So there is a meeting in which all these people get together, the design is presented formally by a reader and then reviewed by all the team members.

Modifications are made to repair all the discovered errors and re-inspection maybe done in that point.

Now, what are the roles of the people involved in an inspection process?

The author is the first role. This is the person who produces the source, produce the design or produce the code.

The inspector is a designated role which says that this is the person responsible for determining defects, determining whether this design passes the standards that they have set for the organization.

(Refer Slide Time 37:06 min)



Reader is like a third party who presents this to the inspection team, who presents the design, who presents the code etc to the inspection team. This could be the same person as of the author. Typically these are roles, remember not the people.

Scribe minutes the meeting.

Moderator essentially makes sure that this is an impartial evaluation process and somebody is not out there to get the author.

And process improvement person usually from quality assurance who is sitting there to try and figure out whether the process of inspection, the process of development, the process of verification can be improved, can be made more efficient and can be made less costly and so on.

This is typically the roles. So what is the check list of errors that one must have, that one is checking for. We are going to go through instead of example let us actually go through the different categories of inspection check list that we can have.

First category is what is called data faults.

Data faults are typically those that can be checked by a compiler. It may not be something that is done through manual inspection.

Remember, these categories can be applied both to the manual inspection process as well as to an automated static verification process.

So data faults are the variable initialized. If it is not, then it could be likely result in some arbitrary error later.

Have all the constants been properly named? Are you using hard coded values or are you using a constant? This is a stylistic feature or a code style feature. Instead of using hard coded values, one must always use a name constant. That is something that the compiler is not going to catch. It has to be caught by an inspection team.

Are the array bounds being met appropriately? For example some programming languages always start array numbering from 1, other programming language like C starts from 0. So is the array bound is “array size – 1” or array size itself?

Are the delimiters existent for all the character strings that are being used in the program, because if it is printed it may not print it appropriately? This is something that testing might be able to catch or testing may not be able to catch this as well. So this is important for manual inspection.

So these are the kinds of data faults that one can look for. Some of these such as variables being used before they are declared and variable not being initialized etc can be caught through a compiler or by a static analyzer. The rest of them typically have to be done manually.

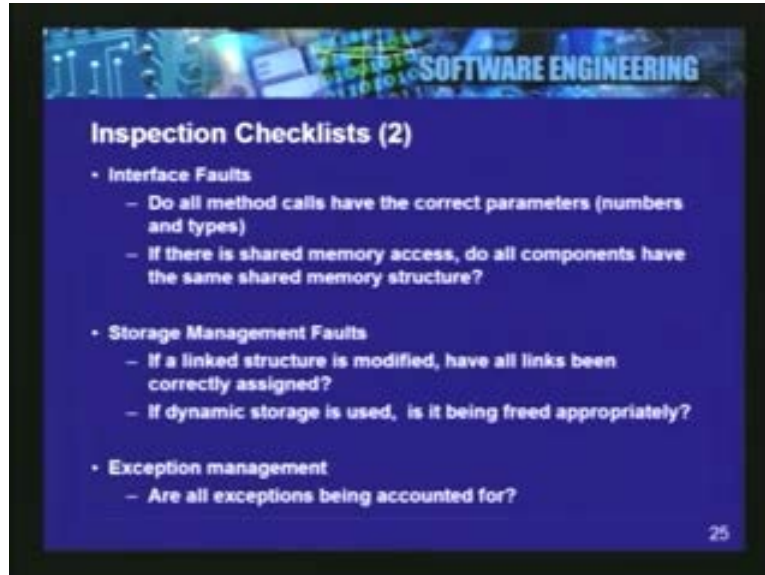
Control faults are another category of faults that you can look for. Control faults are for the things like conditional logic: Is the condition correct? Is each loop certain to terminate? Is there a possibility of an infinite loop under different conditions that this loop can be entered into?

So in ‘switch’ statement, are all the possible cases are accounted for? Is there a default case for a ‘switch’ statement that is provided?

Again that is something that can possibly be checked by compiler. A more intelligent compiler can check for some of these things.

Third category is interface faults.

(Refer Slide Time 40:28 min)



In interface faults you primarily check some simple things such as the number of parameters, the formal types of the parameters matching the normal type parameters and so on.

If there is shared memory access which is part of the interface then, do all the components have the same shared memory structure? Do they have the same view of that chunk of shared memory? If not, then they could be talking at cross purposes with each other.

Storage management faults are typically had to do with managing memory which is the fairly scarce resource in most programs. If a linked structure is modified for example, have you appropriately released the memory? If you are using dynamic storage for something like this, are you releasing the memory? In the case of the linked list, are you patching up the pointers appropriately so that the linked list is still unbroken?

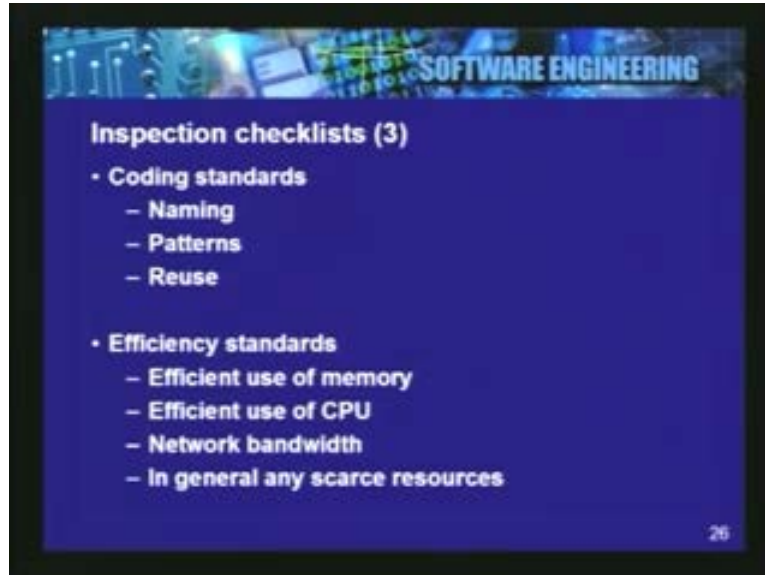
That is the kind of storage management faults that you can look for.

Exception managements: If a function declares three or four exceptions as part of its specification, then when you call that function are you making sure that you handle every one of those exceptions that can arise from that? That is the question you have to ask yourself. Basically, are all exceptions being accounted for?

The next one is coding standards: Are you using the appropriate nomenclature? Are you using interCap naming for the functions, small letter naming for the variables or whatever standard that happens to be within that organization?

At the design level, are you using the patters that are mandated that you use within the design? For example, is the observer pattern being used? Is the singleton pattern, is the factory method being used to create new instances that have to be created and so on?

(Refer Slide Time 42:01 min)



Then there can be efficiency standard that you are looking for.

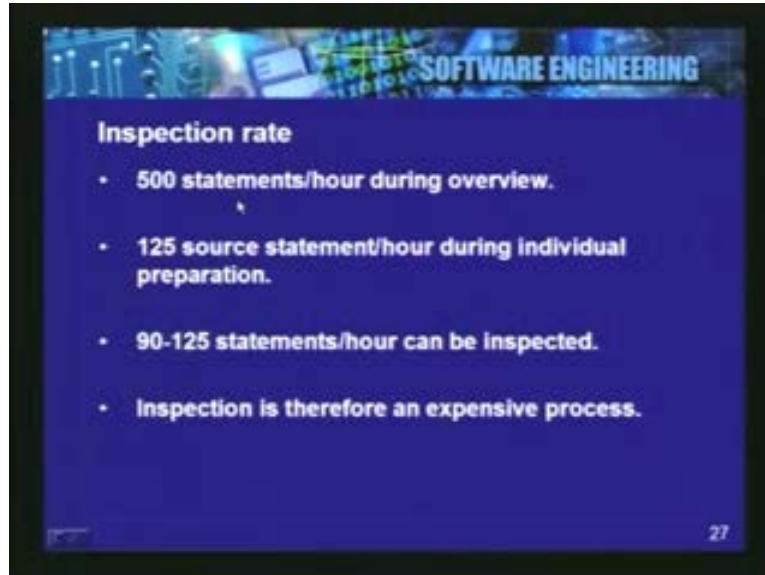
Are you making efficient use of memory? Are you making efficient use of the CPU? Is there a better algorithm that can be used which uses lower amount of CPU for example? Are you using scarce resources such as network bandwidth, IO bandwidth, memory, CPU etc appropriately? Are there better methods of doing this?

These are some things that can also be checked for and be part of an inspection check list.

The inspection rate: Here are some recommended values.

- 500 statements an hour during overview.
- 125 source statements an hour during individual preparation.

(Refer Slide Time 42:33 min)



- 90-125 statements during the actual inspection process.

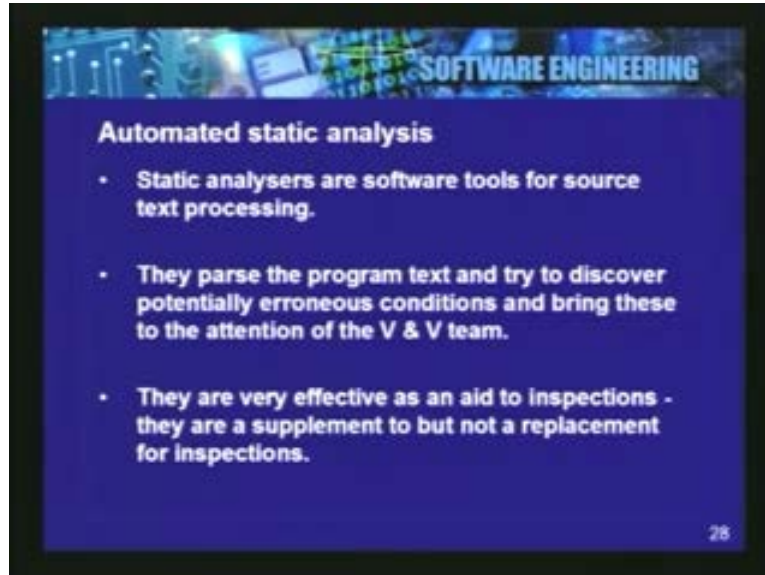
What this is meant to show is that, this is a fairly expensive process. A program can consist of 1000 lines of code. Obviously you cannot inspect every single line of code, because it is very expensive to sit there and inspect every line of code. Just that, the formal methods cannot be applied to a large program in its entirety.

So you have to pick the critical portions of the program, the machine critical portions for example, which can cause a lot of modules to fail if that one module fails, then go ahead with inspection for just those critical modules. It is an expensive process and therefore the buy-in of management is very important.

Moving ahead to static analysis; static analyzers are basically program such as 'lint'. The software tools that can do source code checking. It can check source text.

In the case of designs, for example it can check whether UML diagram is complete. Or for example the relationship values may not be correct, it is 'one to one' versus 'one to many' and so on.

(Refer Slide Time 43:38 min)



Automated static analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

28

So they basically parse the program text, they build up an abstract syntax notation just as a compiler would do and then you apply certain rules that has been set up by the user to do some checking against. We will take a look at some of these examples.

There are very effective as an aid to inspections. They do not replace inspections completely. There are certain things that cannot be checked with static program verifiers or program checkers. For example, stylistic guidelines are something that they cannot check. People have to be involved in the process.

(Refer Slide Time 44:34 min)



Static Analysis Checks

<u>Fault Class</u>	<u>Static Analysis Check</u>
Data Faults	Used before initialization Declared but never used Assigned twice but never used in between Array bound violations
Control faults	Unconditional branches into loops Unreachable code Infinite loops
Interface Faults	Parameter Mismatches Non usage of results Uncalled functions/methods

29

Static analysis checks: Data faults.

They can check ‘using before initialization’: very easy check to perform automatically. So, variable is used before it has been initialized.

Variable is declared, but never used in the program. This can be pointed out by compiler or a static program checker to say that, “you are declaring this variable, but not using it anywhere, so why don’t you get rid of it as it is just occupying the memory”.

It is assigned twice. Say, $X = 3$ and $X = 4$; the first assignment is effectively useless. If before it is used anywhere, if another assignment is being made, then the first assignment can be completely bypassed. So that is a check that can be done.

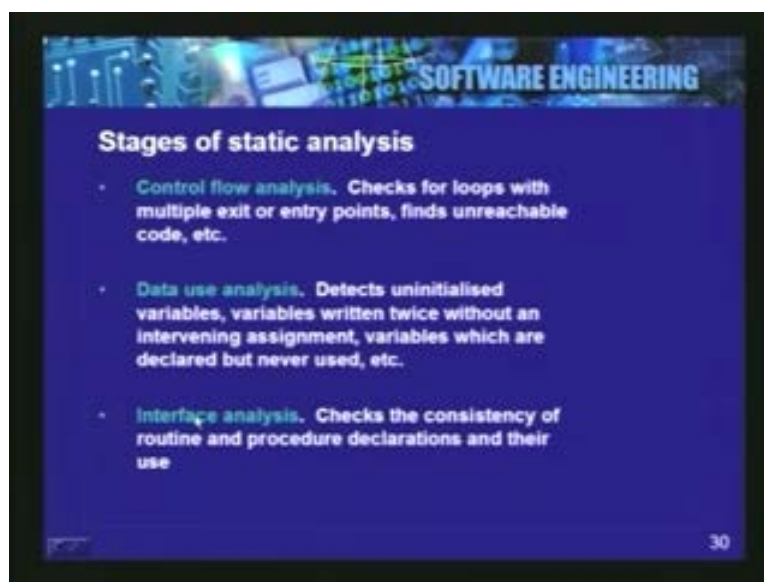
Array bound violations. In some cases this can be done statically, in some cases you cannot do this, because if it is completely dynamic in nature then static program checker is not going to help.

Similarly, we have control faults; interface faults which are the same category of faults that we saw in the inspection check list can to some extent be automated. Whatever can be automated through the use of the software tool falls under the static analysis checks.

Then, simple things like in interface fault; when there is a mismatch in the number of parameters, let us say, this function ‘foo’ has been specified as taking in 4 input parameters and when you are calling it, you are only giving 3 and there is no default value for the fourth one that ‘foo’ accepts. It is an obvious error. And there can be type mismatch also. For example, the function ‘foo’ is expecting an integer value for the first parameter, what you are passing it is a floating point value which cannot be cast down to an integer value. Type mismatches can be easily caught as well.

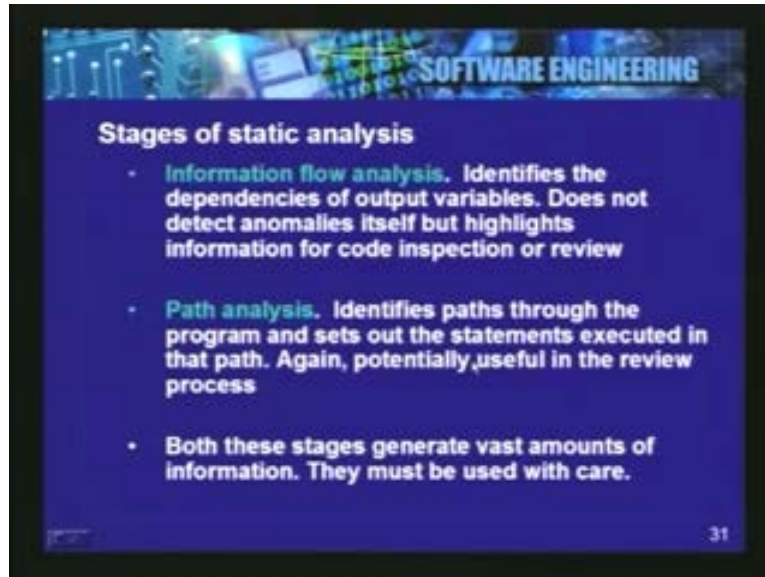
Just like you have unused variable, you can have unused function. Say, you are declaring a function and implementing the function never called anywhere within your program, do you really want to keep this function or move it to be part of some library which makes more efficient use of the space?

(Refer Slide Time 46:36 min)



The stages of static analysis are, there can be control flow analysis, and data use analysis and interface analysis. And corresponding to the kinds of faults that it is checking for, all of these can be used.

(Refer Slide Time 46:43 min)

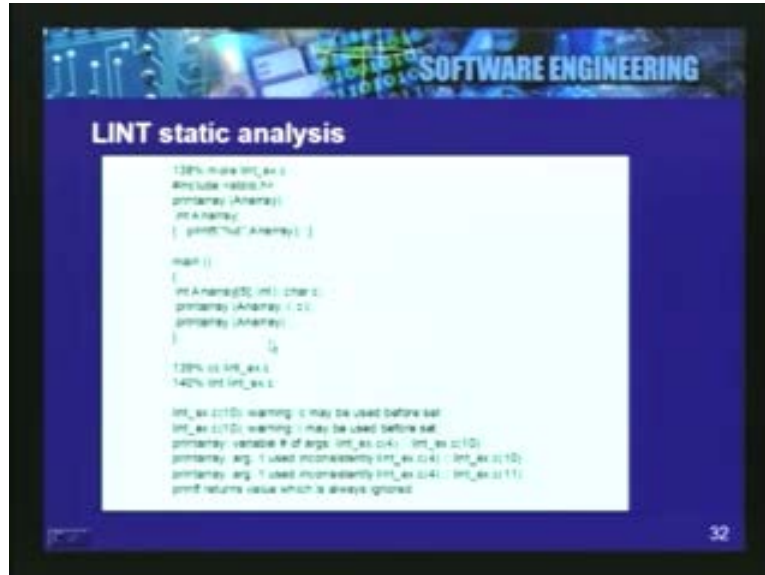


There can also be more complex static analyzers. There can be complex tools that do things like path analysis. Things like, what is the complexity of the program? What is the path of the longest number of hops that the program might end up taking?

So, it can look at entire program as it supposed to do individual module and simple rules. It is very useful, potentially depending on how efficiently it can do this.

Information flow analysis identifies the dependencies of output variable. For example, output variable is coming back from a function that you have called, but you have really never used that return value at all. That is an example here as well.

(Refer Slide Time 47:39 min)

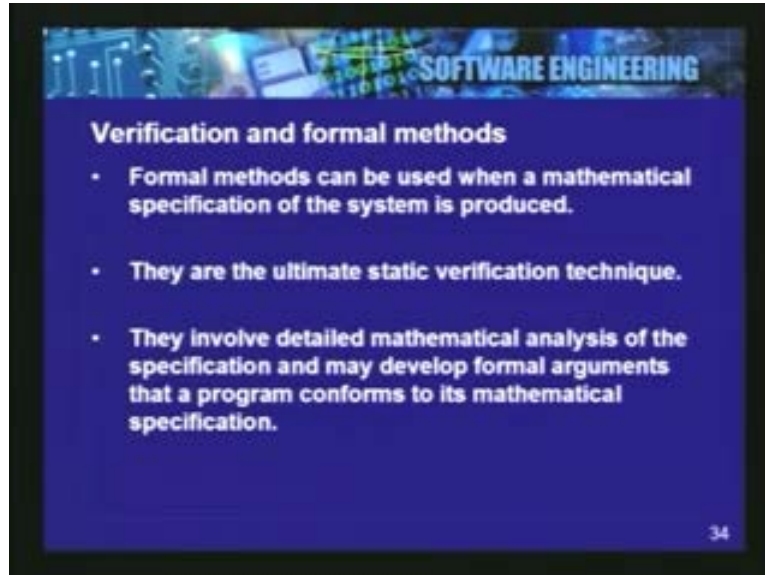


Here is an example. We are not going to go into a great detail about tool called LINT. You all would have heard a tool called LINT. It does static analysis on C programs and it can point out some simple things like a variable is used before being initialized and it can point out that a variable has been declared but never used and so on. But this is something that you can try out for yourself. 'Lint' is meant for C, but the 'lint' is available for other languages as well. It is very programming language dependent. Write a small C program, and then run 'lint' on it and see the kinds of error messages that it ends up giving you.

The next step that we will go into as far as the verification/validation process is concerned, is formal methods. This is something that we have seen before as part of this course as how do we do formal methods, user abstract data types and so on.

This is the ultimate static verification technique because if you are formally specifying the software and then the code generating it because it is executable specifications.

(Refer Slide Time 48:35 min)

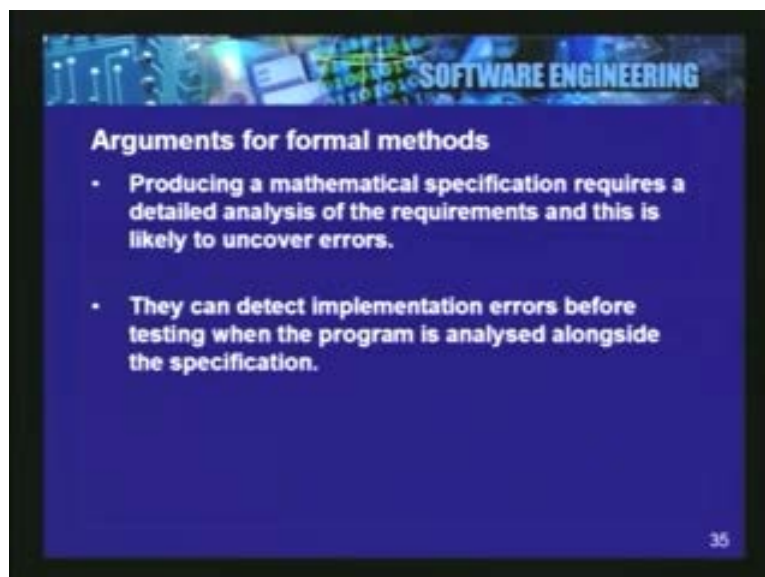


Remember the term 'executable specifications'.

Then, you can always verify that the code that is being generated is meeting the specifications in some way. The mathematical model can be built of the code and the mathematical model of the specification as well and see whether that these two things match.

The argument for formal methods is something that we have seen earlier already. It produces a specification, the thing that it can really catch errors very early on. In fact, it avoids the use of errors completely.

(Refer Slide Time 49:10 min)



Because specification can be run through verifiers that basically checks the completeness of specification, the checks for consistency and so on.

Even before implementation is done, you can do some analysis on this program. So obviously it is very useful but also very expensive. Even the tools that exist are not very prone to usage by regular programmers. It requires specialized knowledge and training in these tools and languages. It is very expensive and it might be possible to reach a level of confidence in verification and validation without the use of formal methods as well, in a much cheaper way through inspections and static analysis techniques and so on. So, one has to be very careful about how you end up using this.

There can be formal methods which is an extreme end of the scale type of process. There can be static analysis which is kind of a middle of the road process which is automated. So you do not have to depend on the vagaries of human nature.

Then, there can be inspection processes which do depend on the expertise of the humans. The added advantage is that, you can get the experience of these people who have been through this many times before. And therefore they may be able to catch things that a program may not be able to catch.

So that is what the scale looks like. At one end, the formal method is very accurate and can really save cost because it can catch defects very early on but very expensive, the other end of the scale is the inspections which are done by humans and which can capture types of errors and in between is the static analysis part of it.

We would not go into clean room development at this point of time.

In the next lecture we will take a look at this process called clean room software development, whose philosophy mainly is that if it can be done right for the first time then that is the way we should go.

We should not try to develop software fast with some errors and then come back and fix it rather we should take the additional amount of time in order to do it appropriately. The word clean room is borrowed from the chip development or the chip manufacturing process in which clean room development is done.

So you start from scratch always and you do not incorporate things from outside and so on. We will go into the detail of the process in the next lecture.

This is it about verification/validation. It is a very important process that has to be appropriately put into the various phases of the software development life cycle. It cannot all be done in the end and it cannot all be done in the beginning. It is something that has appropriate steps during the development life cycle itself and then care has to be taken to see that this is part of the planning for the software project.