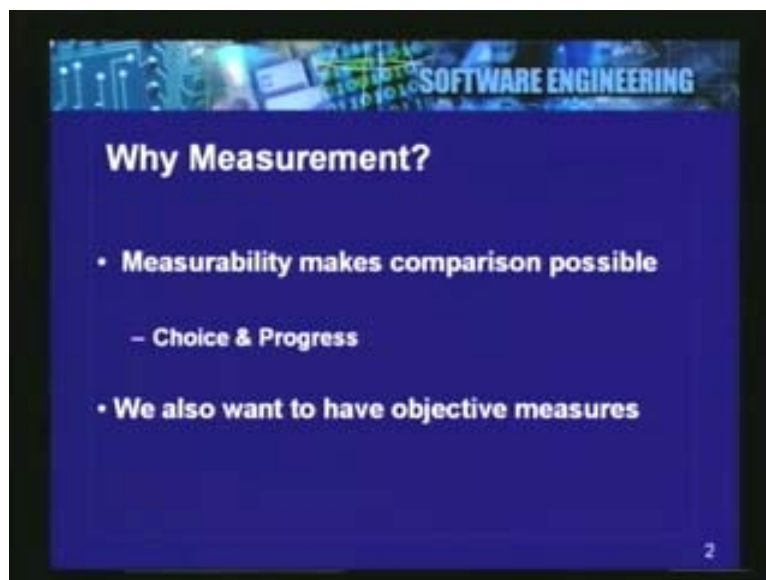


Software Engineering
Prof. Rushikesh K. Joshi
Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 21
Software Metrics & Quality

Some aspects of quality software development and what are the practices that we should use when measuring our software and developing towards quality. So, first question that we have to address is why do we need to measure? Why measurement?

(Refer Slide Time 01:16 min)



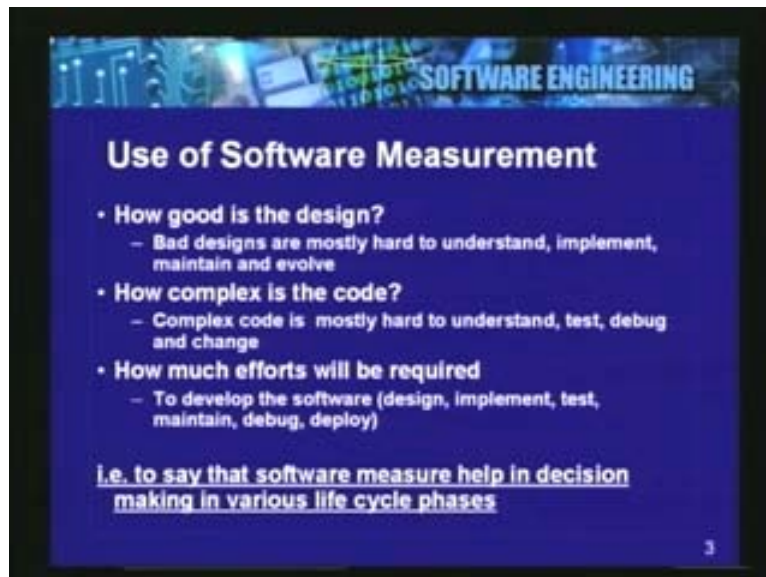
We can see that, if you want to compare two different things, you need to measure. You can say that a person is taller than the other or a person is heavier than the other. When you can measure, you can compare and when you can compare, you also have a choice that means you have some sense of quality and the quality is quantified in terms of the values after measurement. And once you measure you can decide. So choice is possible and then progress is possible. If you know that something is wrong or something is not as per the quality, for example your code may not be efficient. You have measured the efficiency. Now you can thrive to achieve higher efficiency or say utilization.

So once you are able to measure, you have some quantities with you i.e., the numbers with you. Now you know what is expected and in order to improve you can also objectively try to achieve better measures or higher quality. So measurement helps us distinguish different things, which means we are able to compare and we are able to capture quality in terms of some quantity or we are trying to quantify the qualities. And once we are able to do that, we will be able to exercise our choice and also we will be able to progress or we will be able to make corrective steps.

We can encourage quality once we have the numbers, based on of course the numbers. So, basically when we want to deal with software, we want to have these objectives measures so that different people can use it. Once you have an objective measure, everybody understands the meaning and there is only one way to calculate these numbers, then there is uniformity and practice. So we want to have these objective measures and we want to apply these measures to various tasks. If you are able to measure our software, we can compare, we can exercise choice and we can also improve based on the measures.

These are the important considerations which drive this practice of measurement in software engineering. So, what is the use of software measurement? This slide captures some of the points. How good is the design? If we measure we will be able to know whether our design is good or bad. So bad designs are mostly hard to understand, implement, maintain and evolve.

(Refer Slide Time 04:18 min)



Then your software might be correct, but the design of the software maybe very complex. So once I had tried one experiment for a small problem in my class, there were about 20 to 25 designs. If you have many designs for same problem, we have to choose one of them. So we should be able to measure, we should be able to tell objectively which design is good or which design is bad and that we should try to achieve.

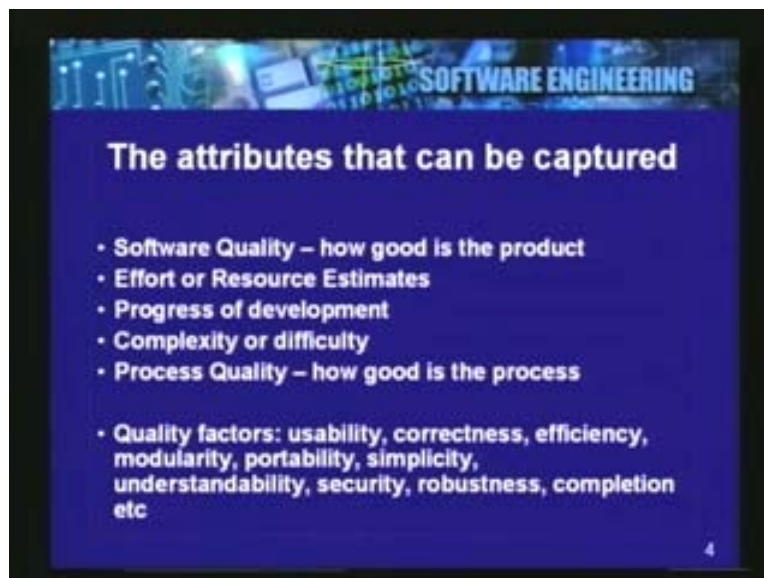
We should find out if there are any objective criteria with which we can measure a design. For example, we can say that, I am using ten classes to achieve this solution and somebody else may use three classes, and somebody else maybe using five classes; which one is better? You cannot say if I use lesser number of classes, I have better design or if I use too many classes, I have a better design.

There must be some criteria, first lead to measure that means to count. And come up with the measures and then compare them, to associate them with quality factors in software engineering. So one important point is about design, and then we also need to look at the code. Complex code is mostly hard to understand, test, debug and change. If the code is very complex, it will be hard to understand. It also deals with styles, the algorithm could be the same, but the code that is written by different people differs.

How much efforts will be required? This is another use of software measurement. If you have the measures of your software or if you can predict, for example the number of classes that are going to be required to implement your software. You can find out how many people will be required, how much time required that means you can estimate your man months, you can estimate your cost. To develop the software, you need to find out how much efforts are going to be required and this deal with your management aspect of software. In order to design, implement, test, maintain, debug, and deploy, how much effort will be required. So these measures can also be useful from that point of view.

That is to say that software measure helps in decision making in various life cycle phases. So this is the relying motivation that if you are able to measure your software product and also the process, it basically helps you in various kinds of decision making, in various phases of your software life cycle. What are the attributes that can be captured? That means what you can capture through these measures about software? Basically, we are trying to capture software quality - How good is the product. Efforts or resource estimates - We can capture them. Progress of development - How far you have progressed, how much is still remaining? That is the most important question which everyone wants to ask when you are developing with deadlines in front of you.

(Refer Slide Time 07:46 min)



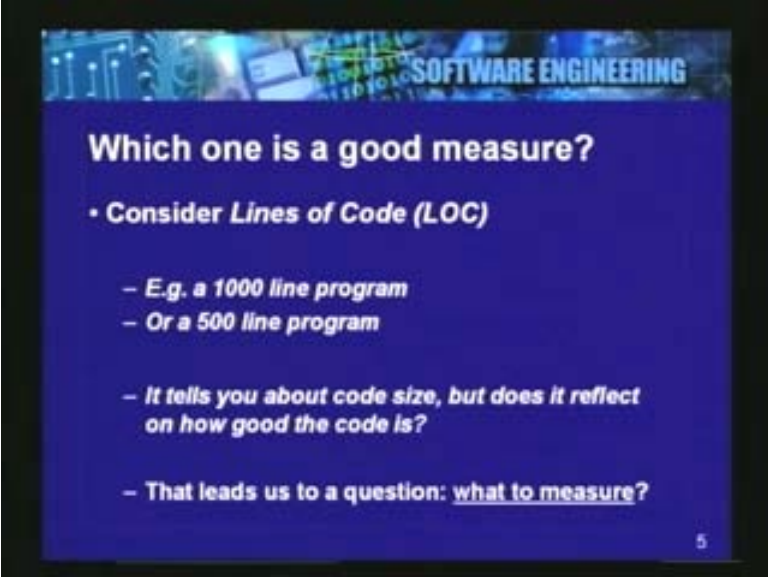
Complexity or difficulty - How complex is the task? How difficult is the task? Obviously you want measure this and you want to capture these attributes through your measures.

Process quality - How good is your software development process? You want to capture this if possible. Quality factors: Usability, correctness, efficiency, modularity, portability, simplicity, understandability, security, robustness, completion that means how much you have completed, how many **functions**, how many use cases have been over, is it 90% complete or 95% complete etc. You want to have these measures and based on these you have some assessment of the quality of your software. So measurement is quite important and these are the attributes which we would like to capture. Quality of your software process, the progress of your development, then the complexity and difficulty and also the efforts or your resource estimates. We would like to capture these through measures and correlate these numbers with some of these quality attributes.

In this lecture we are going to look at some of the basics of measurement and look at the motivation and various measures and some important properties associated with measurement so that we will have good introduction to measurement. And then we will also look at, how you can organize your process and how you can use some of the good practices which are known in the software community or which are well established as good practices.

First we will take a look at different aspects of measurement. Which one is good measure? That is the question which comes to our mind. Lines of code - Is it a very good measure? Will it capture everything that we just said? For example, consider you have a thousand line program and a five hundred line program. Can you say that thousand line program is more complex than the five hundred line program?

(Refer Slide Time 10:00 min)



SOFTWARE ENGINEERING

Which one is a good measure?

- Consider *Lines of Code (LOC)*
 - E.g. a 1000 line program
 - Or a 500 line program
 - It tells you about code size, but does it reflect on how good the code is?
 - That leads us to a question: what to measure?

5

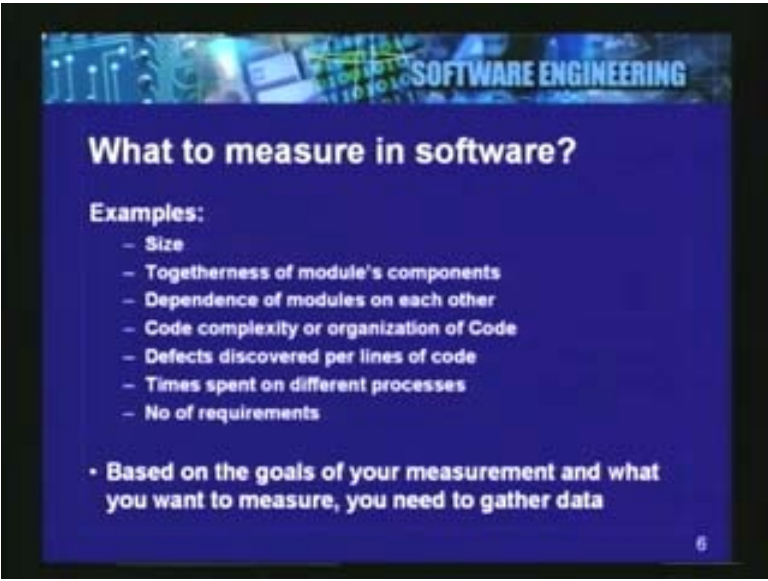
So how can we use this measure? This is the measure - LOC: Lines of Code. This definitely tells you about the code size in terms of number of lines, but does it reflect on how good the code is? How complex the code is? Is it high quality code?

Can you say that the thousand line program is a high quality code and a five hundred line program is a low quality code or can you say that thousand line program is more complex than the five hundred line program? You cannot make these statements but you can say that thousand line program is larger than five hundred line program in terms of number of lines. So LOC is a measure, it tells you about the length of the code, maybe you can associate this with time required to understand the code. Say if I have one million lines of code, I can have some estimates from that number and if I know that I have hundred line programs, I can have some estimates about the efforts required, the time required to understand this one.

So it is useful but we cannot capture everything. That means we need to associate the appropriateness of measure when we are using it in specific context. For example LOC can be appropriate in one context but not in another. This leads to question what to measure. You can of course measure lines of code and use it. So what to measure and the answer depends on what is your need. Where are you going to use this measure, for what purpose? Now there are plenty of measures available and you can go through the metric data bases, there is lot of material available, papers available.

So, many measures are available and you can use appropriate measures in a given context. The important point that we want to highlight is what to measure depends on what is your purpose and accordingly you can choose the right measure and apply to your context. So we will have some attributes mentioned here on the slide on what to measure in software. What can we measure? Can we measure length of the software? Can we measure the size? Yes, but the units are different. If you want to measure the size, you can look at the line of code for example.

(Refer Slide Time 12:51 min)



SOFTWARE ENGINEERING

What to measure in software?

Examples:

- Size
- Togetherness of module's components
- Dependence of modules on each other
- Code complexity or organization of Code
- Defects discovered per lines of code
- Times spent on different processes
- No of requirements

• Based on the goals of your measurement and what you want to measure, you need to gather data

6

There are different kinds of size measures. You can measure togetherness of module's components: We know that we want to achieve high cohesion within a module. We do not want to put unrelated things, unrelated functions or completely unrelated classes in one module. That property is the cohesion property. You want to have high cohesion that means the modules should be together. So we want to measure this togetherness of modules component.

And we also want to measure dependence of modules on each other. That means how much a module depends on another module. So you want to achieve a very good separation of our components or modules so that they are independent. Once we have this good modularization, we know that we can do parallel development, we can give different modules to different programmers, we can test them differently and then only we can make a build. So independence or separations of concerns is a very important principle in software engineering. You want to separate these concerns and you want to reflect them into different modules which are separated. We also want to measure the separation, how separated they are? Or they really coupled together or are they very highly coupled? There has to be some coupling because modules interact.

You do not simply have your software as completely independent modules otherwise it will be simply a collection library. You want your software that is executing and that has lots of collaborating components. So coupling will be there. But how heavy is the coupling? How modules are interdependent on each other? Is it a very spaghetti coupling or is it a very clean structured decomposition where you have a very good low coupling, but there is collaboration through low coupling and you have very highly cohesive module? Each module is together, the components in the module are together, but modules are not very tightly coupled to each other. This is what you will like to achieve. You want to achieve high cohesion, low coupling, at the same time you want to achieve separation of concerns and also collaboration.

So if we have these measures of coupling and cohesion with us, we will be able to tell whether given module is highly cohesive or low on cohesion and then reflect back on the programmer and ask the programmer or the designer to improve on the design or the code whichever is the cause of this bad coupling or bad cohesion. So we would like to measure the dependence of modules on each other.

We would like to measure code complexity or organization of code: How complex is your code, how good the organization is or how bad it is? Defects discovered per lines of codes: You want to find out how many defects have been discovered so far. Say, if you have 100 lines of code and if you have discovered 20 defects, so $20/100$ which means you have 1 defect per 5 lines of code and if you have the same number of defects for a larger program that means there the program is probably better or less error prone. This is a good measure, but you see defects which also have different severity. You may have smaller number of defects, but they could be very severe whereas, trivial or minor defects which are more in number maybe tolerable than the less number of severe defects, specifically in machine critical system.

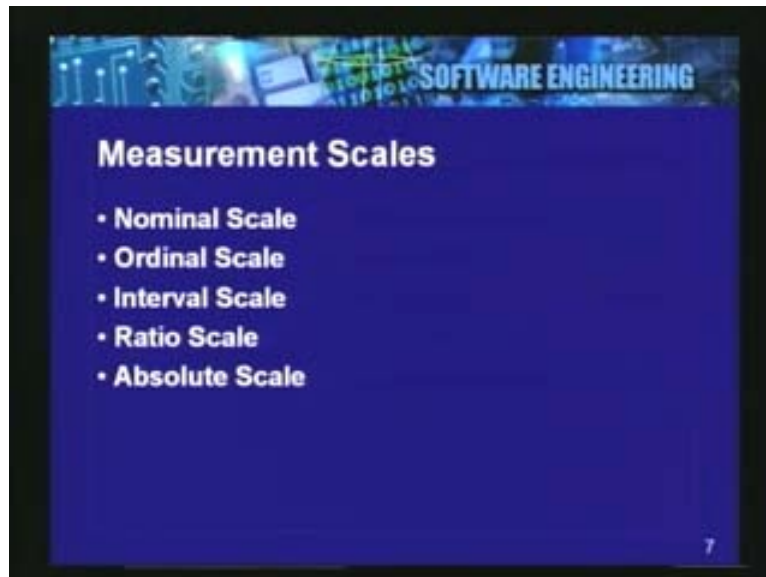
If there is one defect that **search** down the whole system it could cost very heavily. So a defect discovered per lines of code is of course a good measure, but it does not tell you about severity of your defects, which is something different. But however it is a measure and it can tell you whether the programmer is making many mistakes so on. During testing you have to measure this. So not only that you are measuring your software and your source code, but you are also measuring during your testing phase. You are measuring number of defects that are being discovered for a given module or a code and then you can relate it to people or the resources involved and try to improve. So, we can use these measures, we can feed back these measures in to the process and improve on the quality of the software.

Then time spent on different processes: You can measure the time to control the process well. The number of requirements for your software: We can measure based on number of requirements. If you have too many requirements, simple measure could be just number of use cases in your requirements documents. If you have too many use cases you have some measure on how much time it is going to take to carry out this project.

These are some examples. So based on the goals of your measurement and what you want to measure, you need to gather data. This data collection is important. So you need to gather the related data in order to measure. Sometimes the data can be collected automatically. You can write tools which can directly give you measures. For example if you are measuring your code, you can write tools and get these measures and during testing also you can count number of defects and so on. And few measures, the data could also be manually maintained in the form of tables and so on.

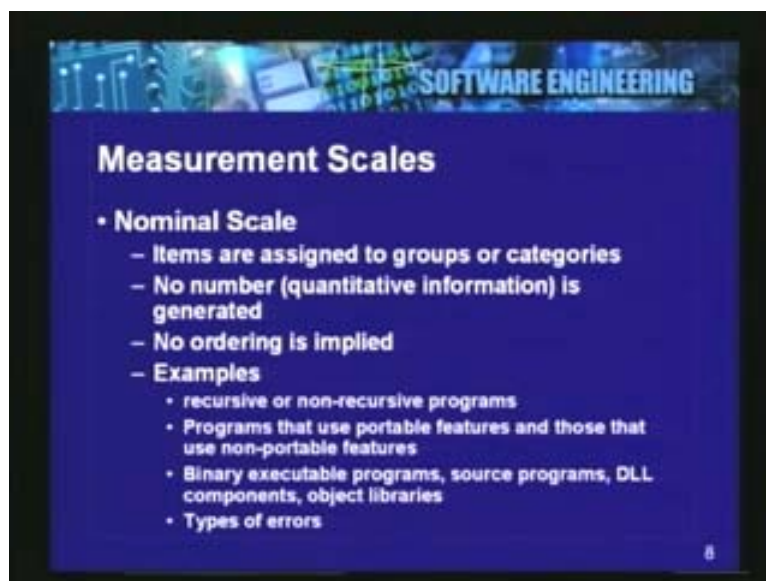
So it is important that in appropriate life cycle stages, you keep an accounting system and keep gathering the data so that you can collect these measures and then you can feed back into the quality of the software. That is the important point that, once you measure your software, you can use them to choose to improve the quality of software. Now we will look at some aspects of measurement. What are the different kinds of measurement? Can we classify them? One important concept to study in measurement is 'scale'.

(Refer Slide Time 19:08 min)



We could have measures on different scales. They are called nominal scale, ordinal scale, interval scale, ratio scale and absolute scale. Let us look at these scales, what they are and what kinds of measures we have and what can we do with them? The first one is the nominal scale. In the nominal scale items are assigned to groups or categories. No number is generated. For example you cannot have ordering, no ordering is implied. No quantitative information is generated, but you have groupings or categories. Examples are whether your programs are recursive or non-recursive programs or programs that use portable features and those that use non portable features.

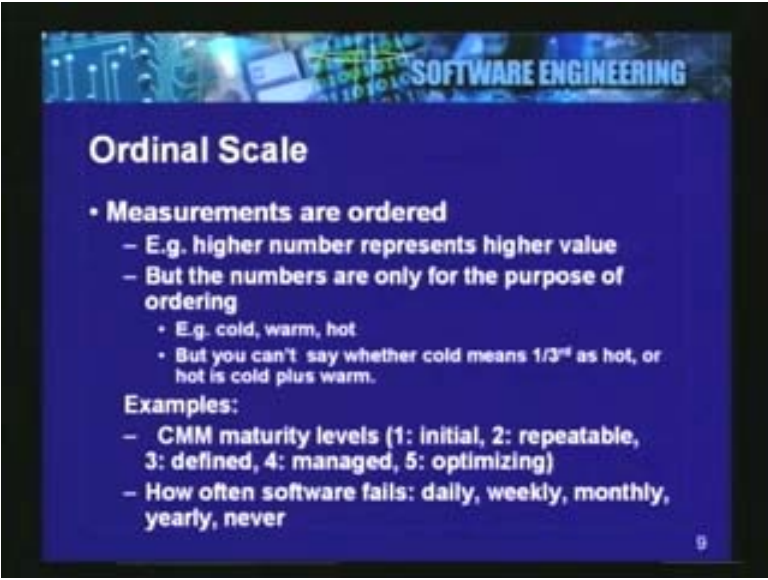
(Refer Slide Time 20:16 min)



Binary executable programs, source programs, DLL components, object libraries. So you can classify them, categorize your programs or functions or modules in these different classes. **Different types of errors**. In one way you are basically classifying. In this scale you get groupings or categories, but you cannot order them. So you can say that using this measure, you basically say this is different from the other and it is a qualitative difference, but there is no quantity associated with this. But we do use these measures and they are also very helpful.

Then, you have the ordinal scale on which you order your measurement. Measurements are ordered. Higher number and represent a higher value and the lower number represents the lower value. But the numbers are only for the purpose of ordering. Say, for example 'cold', 'warm', 'hot' - this is an ordinal scale. But you cannot say whether cold means one third as hot or hot is cold plus warm. So these numbers are basically for the purpose of ordering or in this case cold warm hot we can say cold is 1, warm is 2 and hot is 3. But you cannot say hot is cold plus warm or you cannot say cold means one third of hot.

(Refer Slide Time 21:13 min)



Ordinal Scale

- **Measurements are ordered**
 - E.g. higher number represents higher value
 - But the numbers are only for the purpose of ordering
 - E.g. cold, warm, hot
 - But you can't say whether cold means 1/3rd as hot, or hot is cold plus warm.

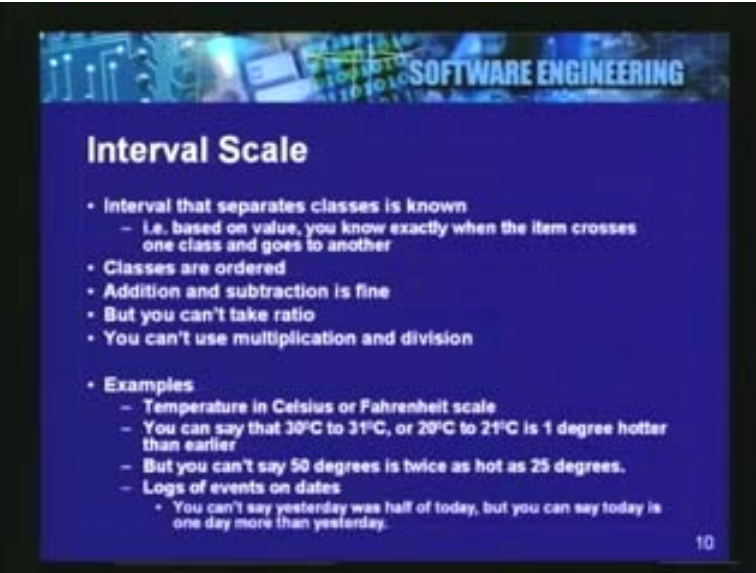
Examples:

- CMM maturity levels (1: initial, 2: repeatable, 3: defined, 4: managed, 5: optimizing)
- How often software fails: daily, weekly, monthly, yearly, never

9

For example you have CMM maturity levels: Initial, repeatable, defined, managed and optimizing. This gives you an ordinal scale. How often software fails: Daily, weekly, monthly, yearly or never. We have this ordinal scale. So ordinal scale is also useful and it is also often used in software engineering. Then you have the interval scale, where you have intervals that separate classes when the interval is known. That is based on value, you know exactly when the item crosses one class and goes to another. The classes are ordered; different classes or the categories are ordered.

(Refer Slide Time 22:18 min)



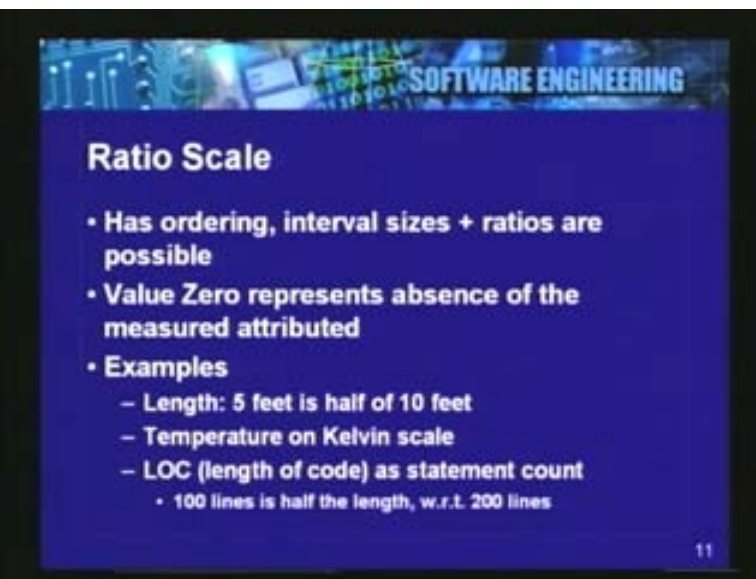
Interval Scale

- Interval that separates classes is known
 - I.e. based on value, you know exactly when the item crosses one class and goes to another
- Classes are ordered
- Addition and subtraction is fine
- But you can't take ratio
- You can't use multiplication and division
- Examples
 - Temperature in Celsius or Fahrenheit scale
 - You can say that 30°C to 31°C, or 20°C to 21°C is 1 degree hotter than earlier
 - But you can't say 50 degrees is twice as hot as 25 degrees.
 - Logs of events on dates
 - You can't say yesterday was half of today, but you can say today is one day more than yesterday.

10

Addition and subtraction is fine, but you cannot take ratio, you cannot use multiplication and division. For example temperature in Celsius or Fahrenheit scale: You can say that 30 degree centigrade to 31 degree centigrade or 20 to 21 degree centigrade is 1 degree hotter than earlier. But you cannot say 50 degrees is twice as hot as 25 degrees. Example in software engineering is logs of events on different dates. You cannot say yesterday was half of today, but you can say that today is one day more than yesterday. So you have these intervals and they are also ordered. Addition subtraction is not fine in the ordinal scale, whereas it is possible in the interval scale.

(Refer Slide Time 23:28 min)



Ratio Scale

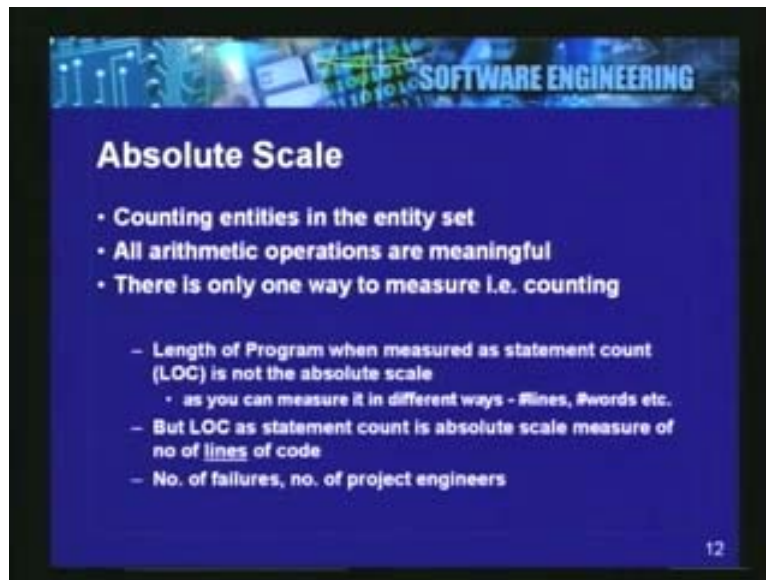
- Has ordering, interval sizes + ratios are possible
- Value Zero represents absence of the measured attributed
- Examples
 - Length: 5 feet is half of 10 feet
 - Temperature on Kelvin scale
 - LOC (length of code) as statement count
 - 100 lines is half the length, w.r.t. 200 lines

11

Then we have the ratio scale. Ratio scale has ordering, interval sizes and ratios are also possible. So value 0 is available and represents absence of the measured attribute. Examples: Length 5 feet is half of 10 feet; Temperature on Kelvin scale; LOC (length of code) as statement count: you can say 100 lines is half the length with respect to 200 lines.

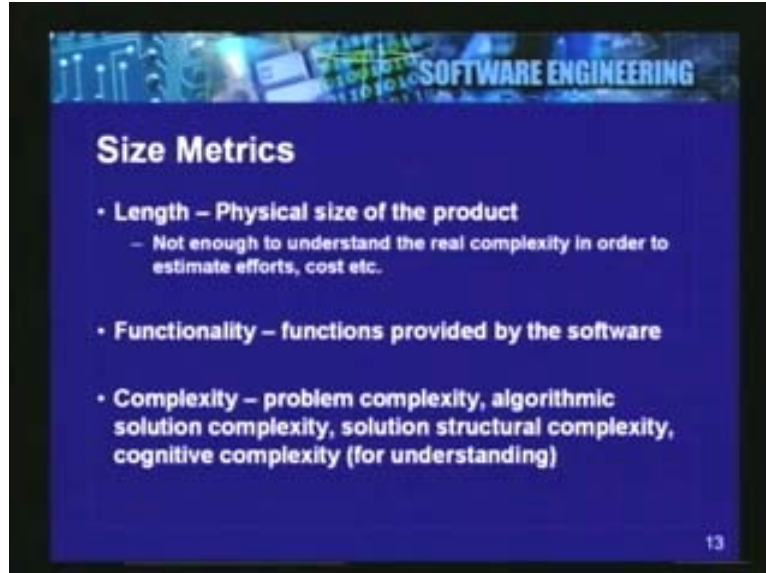
Then you have the absolute scale: Counting entities in the entity set, where you considerably count the entities. All arithmetic operations are meaningful. And there is only one way to measure that is by counting. So length of program when measured as statement count, LOC is not the absolute scale as you can measure it in different ways. For example you can have number of lines or number of words etc. But LOC as statement count is absolute scale measure of number of lines of code.

(Refer Slide Time 23:57 min)



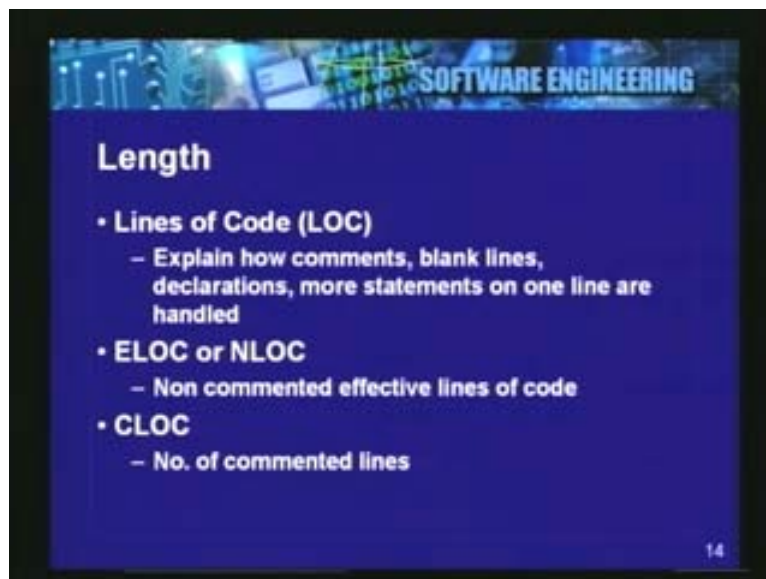
So, if you are saying that I am simply measuring the number of lines of code then the LOC is absolute count. Number of failures, number of project engineers - these are absolute scale measure. Now, let us look at some of the metrics which we can use in the software life cycle. For example, you have length gives you physical size of the product. It is not enough to understand the real complexity in order to estimate efforts, cost etc. So the length or number of lines of code is not enough to gauge the real complexity, but it is a useful measure. Then the functionality; Size can be also found out in terms of functionality, how much functionality is present in the software, how loaded is the software in terms of functions or the functionality that it provides.

(Refer Slide Time 25:29 min)



These are also some size metrics. And then the complexity metrics mainly provides the measures of the problem complexity, algorithmic solution complexity, structural complexity, or cognitive complexity that is an understandability of your software, how easily you can understand your software. These are various aspects of complexity. These are basically various size metrics. You are trying to measure your software size in terms of these aspects: The length of the software, or the functionality of the software, the size in terms of the functionality provided or the complexity of your problem domain and the solution. Let us look at length. It is a simple set of metrics.

(Refer Slide Time: 26:06 min)

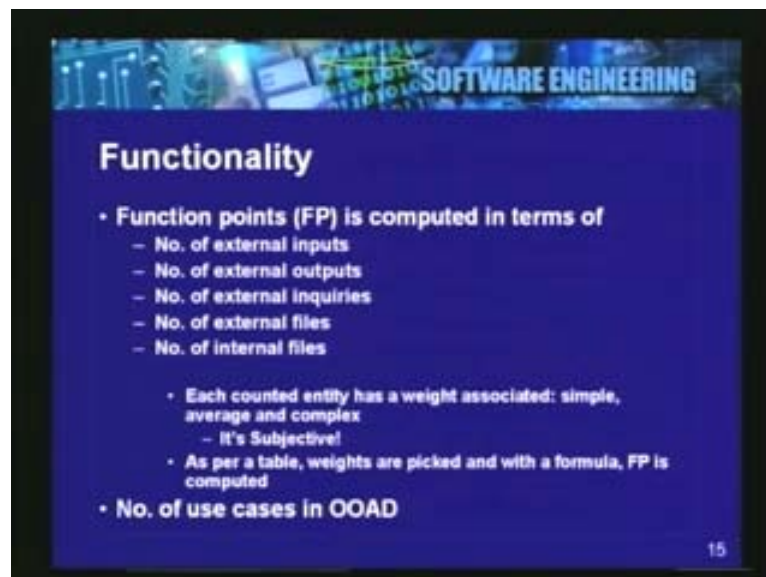


You have lines of code(LOC), but when you say that you have 1000 lines of code, you have to also specify what have you done with comments, blank lines, data declarations or if you had say more statements on one line of your print out how did you count them. When you measure LOC you have to explain these so that they can be compared. You have to have standard about measurement, so that in a company when you report these figures everybody will know what you mean. And then follow this same practice again and again so that different entities measured at different times can be compared.

So when you report 'lines of code', you have to also explain how comments, blank line, declarations and many statements on one printable line are accounted. You have another measure called ELOC or NLOC which mainly counts 'Non-commented effective lines of code'. You can say LOC is 28 means, 20 are the non-commented effective lines of code. Or CLOC which gives you number of commented lines in the programs. These are various length related measures.

Functionality: There is an important measure available in software engineering called function points (FP), which is computed in terms of number of external inputs, number of external outputs, number of external inquires, number of external files, number of internal files, **you can change files' interfaces** and so on.

(Refer Slide Time 28:02 min)



But these different factors are taken in to account and each counted entity has a weight associated: simple, average and complex. Say for example, you have various external inputs and some of them may be simple, some of them may be average and some of them complex. This kind of measurement could be subjective. One person may say that something is complex and another person might put it as average. So there is some kind of subjectivity associated here. But you give these weights and then as per a table these weights of these various numbers are picked and then with the formula you compute your FP or your 'function points' measure.

So this is one way to measure your functionality because we are talking about inputs, outputs, files used, or the different number of entities used, number of external inquiries into your software. So you are trying to measure the functionality in some sense. Then another good measure is a simple number of use cases in OOAD. Complexity: We know that we have the big 'O' notation and we have various kinds complexities and you can compare them.

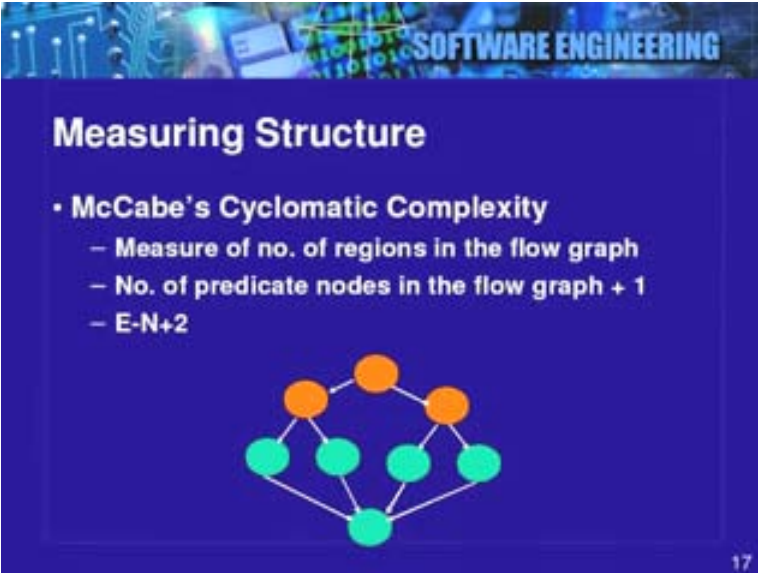
(Refer Slide Time 29:26 min)



Suppose you can say that, I have a function and it is a constant time algorithm. So irrespective of your input size it has same number of steps that have to be executed. You may have a logarithmic, linear, quadratic and exponentially complex algorithm. So once you measure these you know how complex your algorithm is and how the timings are going to vary based on input sizes. So you basically measure both time and space complexity, you may also want to say the space required so that you can use your resources effectively.

Now we would like to measure structure. We just talked about measuring size, various aspects of size for software that is, it could be a length, in terms of functionality, in terms of complexity, and you could also want to measure the structure of your code or your software. One interesting measure is McCabe's Cyclomatic complexity or it basically measures number of regions in the flow graph or number of predicate nodes in the flow graph: plus 1 or E minus N plus 2


(Refer Slide Time 30:07 min)



SOFTWARE ENGINEERING

Measuring Structure

- **McCabe's Cyclomatic Complexity**
 - Measure of no. of regions in the flow graph
 - No. of predicate nodes in the flow graph + 1
 - $E - N + 2$

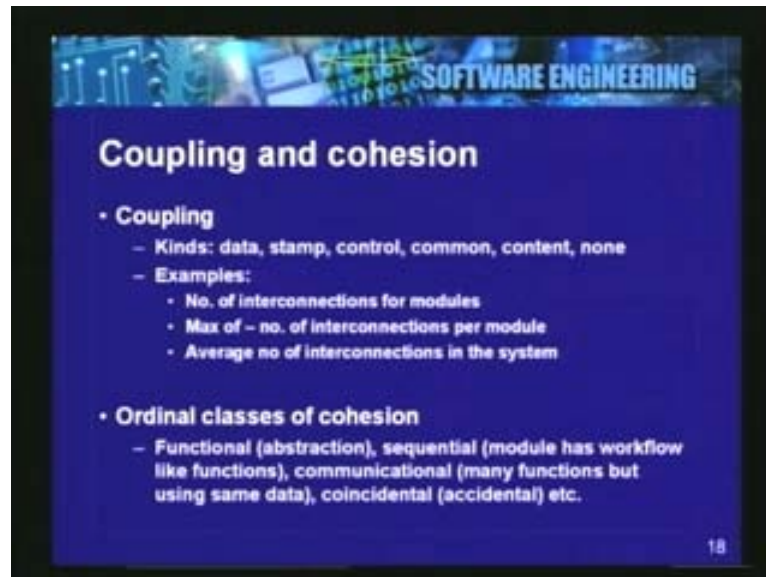


17

In this figure we have these arrangements. You have predicate nodes or decision nodes and you have four regions or these also represent the number of independent parts that you can use in your testing phase and so on. We have seen similar pictures in lectures on testing. This is one good measure that talks about McCabe's Cyclomatic complexity. But you can see that a program can be very large and still have same McCabe's Cyclomatic complexity as a small program.

It is possible in the sense, for example it is useful in testing but it may not really tell you whether the program is really complex or you may have a very big program, but it does tell you something about your decision nodes or predicate nodes or the structure of your program. And you can use this in testing because you can use this number of independent paths and we have the coverage criteria derived from McCabe's complexity. Now we will see how coupling and cohesion could be measured.

(Refer Slide Time 31:58 min)



There are different kinds of coupling as you see on the slide. We have data coupling, stamp coupling, control coupling and so on. Through these various kinds of couplings, modules are interrelated or interconnected with each other. So you would like to measure this number of interconnections of different kinds of coupling or maximum of the number of interconnection per module. Say for example at most what the heaviest coupling is in the entire system or average number of interconnections per module in the system. So you can measure coupling through these number of coupling connections and you can also look at cohesion. There is one ordinal scale for cohesion. So you have the functional, sequential, communicational, coincidental and different kinds of cohesions.

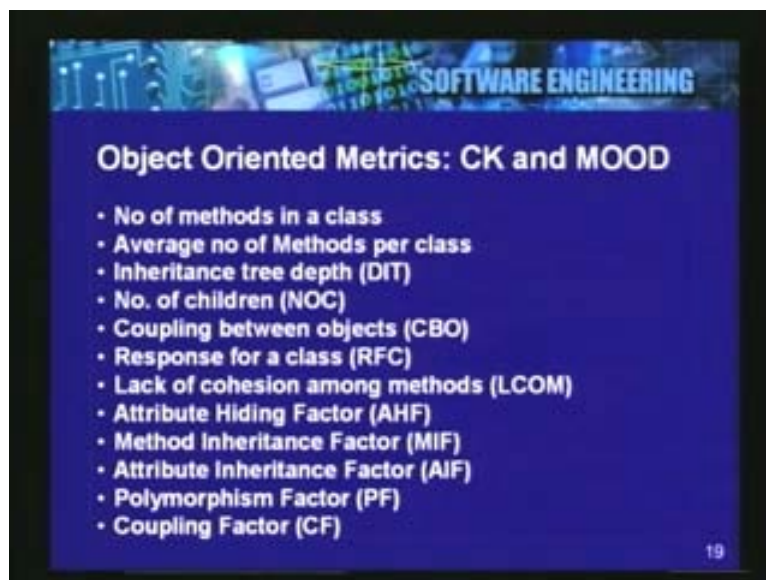
You can say functionally cohesive is the best cohesion. We are putting all the components together, so that is the best cohesion. Coincidental cohesion is just 'they are together', but it is not really cohesion. We have put some of the classes. There are various classes in which you can define the cohesion and you can say one is better than the other. So the best is the cohesion where, which is due to abstraction. So you are putting components together because they are achieving some related functionality and abstraction of the module is one and that is the best cohesion possible. So there is one such scale available.

But you can also measure cohesion in terms of your code for example you can try to find out whether a class can be partitioned into two, by partitioning the classes functions into two. Partition such that, each set of functions uses different set of variables. If it is partition able in that way, then the class is not cohesive or you can find out whether class is highly cohesive or not so cohesive. So there is measure available in object oriented software metrics suite for measuring lack of cohesion.

Now, we are going to have a slide on the object oriented metrics. The object oriented community has come up with quite a few metrics which can measure your object oriented design and object oriented software or object oriented code.

CK that is 'Chidamber and Kemerer' suite and MOOD metrics suite are the two important suites which can be looked at, but there are many other metrics available. For example, you can measure number of methods in a class, average number of methods per class or depth of inheritance tree, how deep is your inheritance tree. If it is too deep then it is considered to be very complex in terms of say the behavior, different polymorphism possible or the tracing required to understand which method is going to be invoked through the inheritance tree.

(Refer Slide Time 34:53 min)



Number of children, coupling between objects, response for a class or RFC, lack of cohesion among methods that is LCOM, attributes hiding factor, method inheritance factor, attribute inheritance factor, polymorphism factor, and coupling factor. These are some various metrics available in the object oriented domain as well and the main references for these are the CK and MOOD papers. You can pick them up and they are also covered in many text books.

Some of these are quite simple and they can be enabled through tools or also by manual accounting. These measures are useful. As we can see that, we are trying to measure various attributes, which connects to quality of the software and then we want to feed back into the quality of the software based on the measures. For example if you find that class is not cohesive, you may want to redesign your system or if you find that there is very heavy coupling between two classes, you may want to reconsider the design or you may want to move or you may want to apply some refactorings on your design and readjust your collaborating classes or the hierarchy, so that these figures improve.

If you have some quantification of your design and code, you can use that in order to redesign the system or re-factor the system. So this is a very interesting approach to software evaluation and refactoring. You can do it based on the metrics as well and if you have the set of tools available and if you have the expertise available that can make it possible to interpret these measures.

So interpretation of measures is also a very important area. In a given set of measures, how to interpret about the quality of software or the corrective steps that are required in order to improve on the quality. So it is an important area and also an area of research. Many interesting metrics are available and one can go through them and keys to apply the right metrics in the right context and also interpret these metrics, so that you can feed back into the quality software. Now after having looked at measurement and some of the metrics and their importance in software engineering, we will now move on to some guidelines for coding and development.

(Refer Slide Time: 37:56 min)

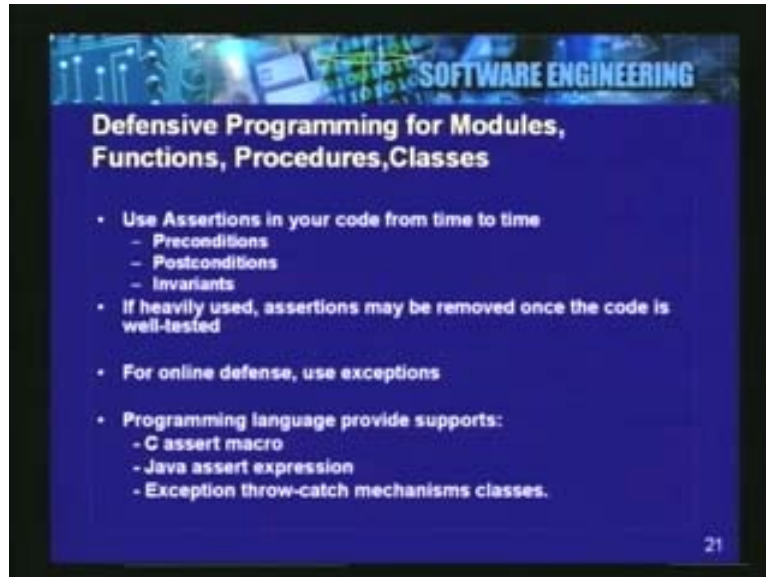


We will move on to the quality rated guidelines so that if you practice these, they will lead to improvement and quality of your software. We might have seen these in our earlier lectures, but I am putting it down again to emphasize some of these aspects which can be practiced even in the small program or when you program in your data structure assignments or in your any class assignments, when you code in groups, or when you code against the course projects, you can try to use some of the techniques and they come in a long way so that it affects the quality of your software and they are scalable and they can be used in the large as well.

For example use assertions in your code from time to time. This is called defensive programming style and can be applied for modules, functions, procedures, classes and in different kinds of modules. Why defensive, because these can act as defense against errors which you can make.

If you use assertions in your code, then the assertions can capture various errors. The assertions can be of different kinds they could be preconditions, they could be used as post conditions and invariants.

(Refer Slide Time 38:53 min)



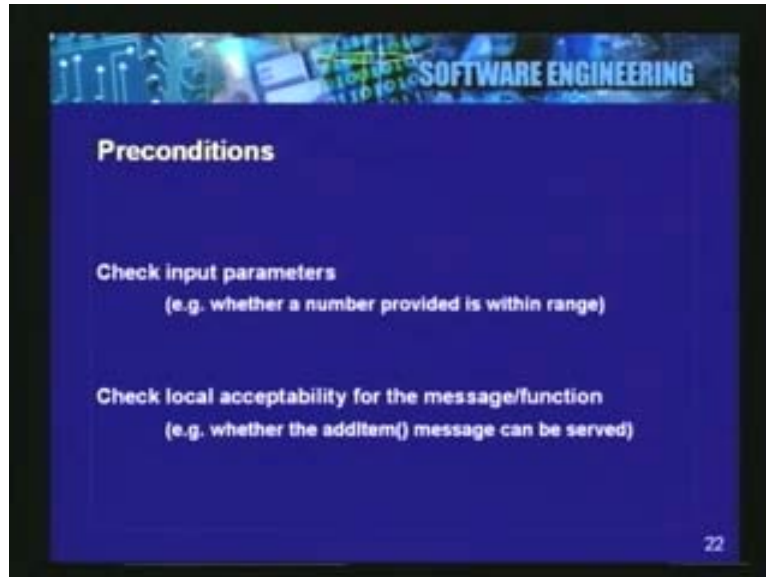
If you are using assertions very heavily and there could be practices that recommend very heavy use of assertions depends on the context in which you are working or in the project which you are working, they could be removed once the code is well tested or some of them could be removed once the code is well tested. For online defense typical technique used is that of exceptions. So the software should be able to catch an error or an exceptional condition and handle it. That gives you handle to gracefully terminate your software or to gracefully handle conditions.

We can use assertions when we are developing the code, and then possibly remove them when the code is well tested and when we want to deploy it. But you can use exceptions for online defense. For example, if you know that your software is not handling certain situations and if they arise you can create exceptions throw them and catch them. So it is very important and if you know that you cannot handle a situation, you have to terminate software gracefully or there has to be a path to handle these kinds of failures. It should not be an abrupt termination, which means the software does not know that there is a fault and they can cause very severe failures.

So exceptions are very important and which should be used. Programming languages do provide supports for handling exceptions or assertions. For example, we have the C assert macro, javas assert expressions, throw-catch mechanism classes for exceptions and throw-catch mechanism of the exception handling for example you have it in java and so on.

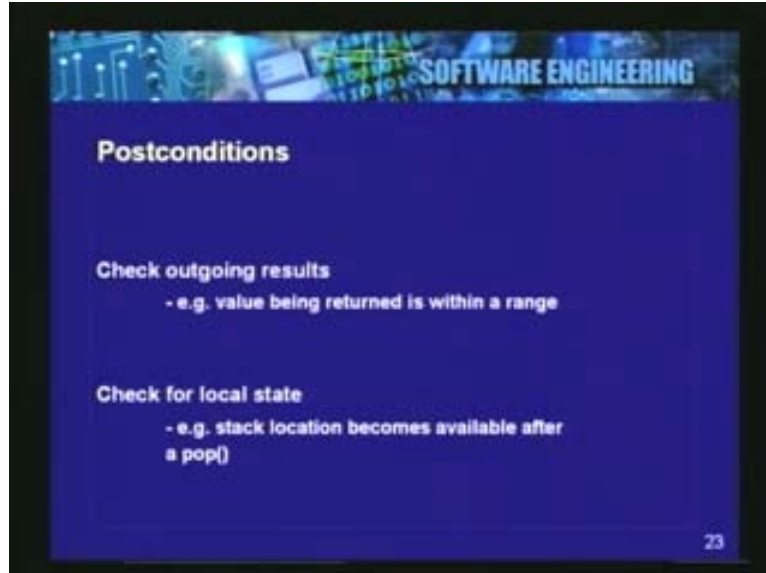
So you can use assertions and exceptions in your code frequently. Try to use them in your smaller assignments so that you can improve on your developmental practices. Now, we will look at what are these preconditions, what can we do with preconditions. We can check your input parameters for example, whether a number provided is within the range.

(Refer Slide Time 41:46 min)



We can check local acceptability for the message or function. For example, whether the function 'add item ()' message can be served? Is there capacity in the buffer so that an item can be added? You can check these at the input itself as soon as function is invoked. These are preconditions. And then you can write your actual business logic so that, the business logic does not have to worry about these preconditions. They will be guaranteed when you come to business logic. So you can separate this part in the precondition code. And in post conditions you can check outgoing results whether value being returned is within a range that is guaranteed.

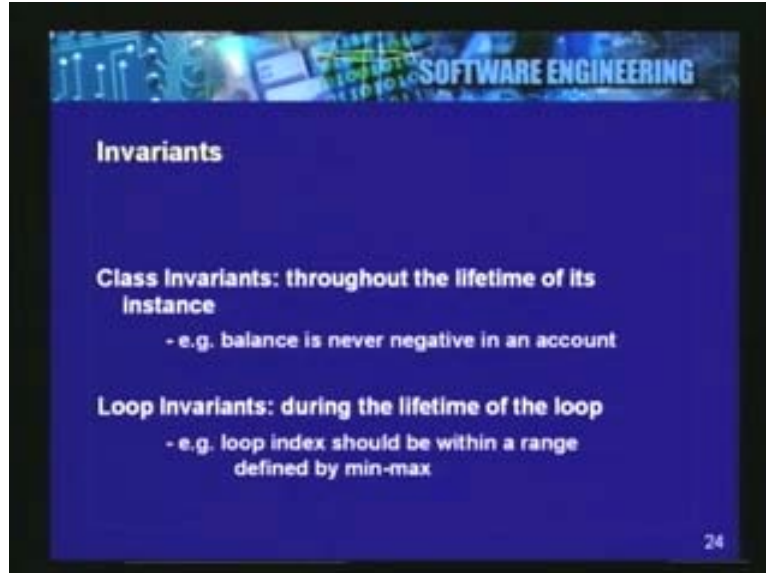
(Refer Slide Time 42:23 min)



And you can check for the local state that is, for example after executing your business logic just before returning you call whether your local state is consistent. For example stack location becomes available after a successful 'pop ()'. So you can check these in your post conditions.

And then you have invariants. There are different kinds of invariants. You can have class invariants; throughout the lifetime of the instance, the invariant **remains too**. For example balance is never negative in an account and if it goes negative, then obviously there is a problem and if that is the case, you can raise an exception or you can raise an alarm. The program language has to support these so that you can specify them and monitors can be inserted in the code.

(Refer Slide Time 43:11 min)



Loop invariants can be specified and are valid during the lifetime of a loop. Loop index for example should be within a range defined by min-max of your loop index. Now we will talk about organizing your modules. How do you organize your modules? How do you decompose them nicely so that you follow the separation of concerns and that reflects into quality of your module organization?

- Write related functions in one file.
- Write header file for this module.
- User programs that uses this module is in a different file.
- User program includes the header file.
- User module and function module can be separately compiled to object code.
- And then you can link these two together to produce an executable.

(Refer Slide Time 43:45 min)

SOFTWARE ENGINEERING

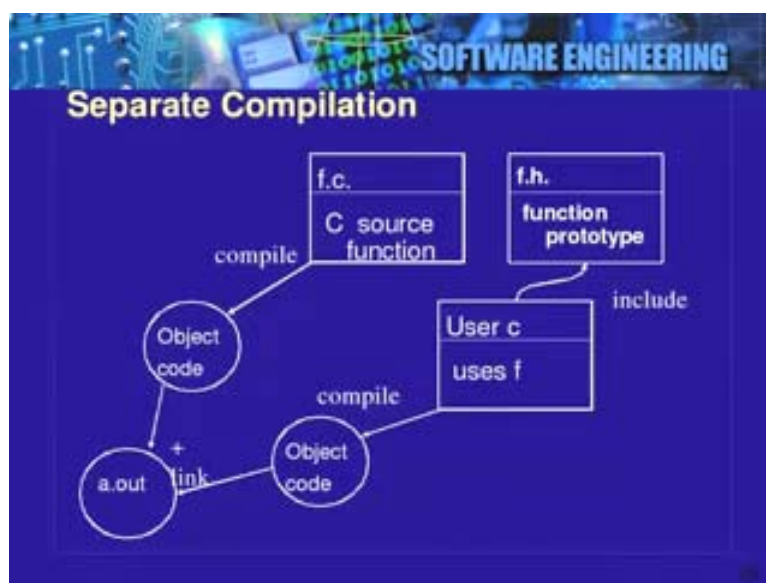
Organizing Modules (C-style)

- Write Related Functions in one file
- Write a header file for this module
- User program that uses this module is in a different file
- User program includes the header file
- User Module and Function Module can be separately developed, compiled to object code
- Link the two object codes to produce an executable

25

This is a good style and the same thing is used in large scale development. When you actually separate the concerns, when you have many modules and you have the module signatures or the header files which specify the interfaces and basically you are separating the interfaces from implementation. So a module that is using another module can be still developed if you have the interface for the other module available with you. This is what basically we are trying to capture and this figure captures this process.

(Refer Slide Time 44:38 min)



You have this function, you have the header file of the interface defined in this file and you have the user file that means you have one module here which actually is going to need this module, but you do not need to wait for this module to be developed (Video is not clear to understand which path and which module is being pointed out in this portion of the lecture 44:58). We can see that this path is different and this path is completely different. So you can take these two independent paths, you can have two programmers taking these paths simultaneously so that you can speed up the development process. But the module which depends on this module has to include this function prototype and this module uses this function prototype. This conforms to this prototype, so this has to be done before you can start developing this module or this module.

That means you can take this independent paths and you can **exploit (explore?)** these independent paths. Once you develop this function code, you can compile it to object code, make it ready and once you develop the user code, compile it to object code, make it ready and then you can link when both are available and you can deploy your software or 'a. out' for example in your simple Unix environment.

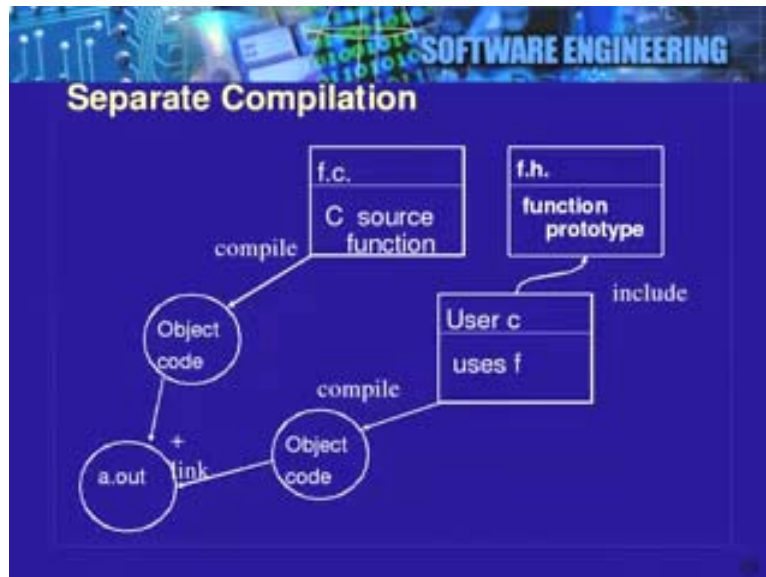
Then you make into practice to use make files. You can use 'make' utility to organize your program, compilation and building processes. If you use sophisticated tools, this can be specified in the integrated development environment. The make files or the build files can be specified through the environment itself, so that it organizes the build on its own and you do not need to key them in one by one.

(Refer Slide Time 45:57 min)



However for your small assignments, do try to write to your own 'make' files, so dependencies can be captured. For example, the dependencies in this process figure:

(Refer Slide Time: 46:27 min)



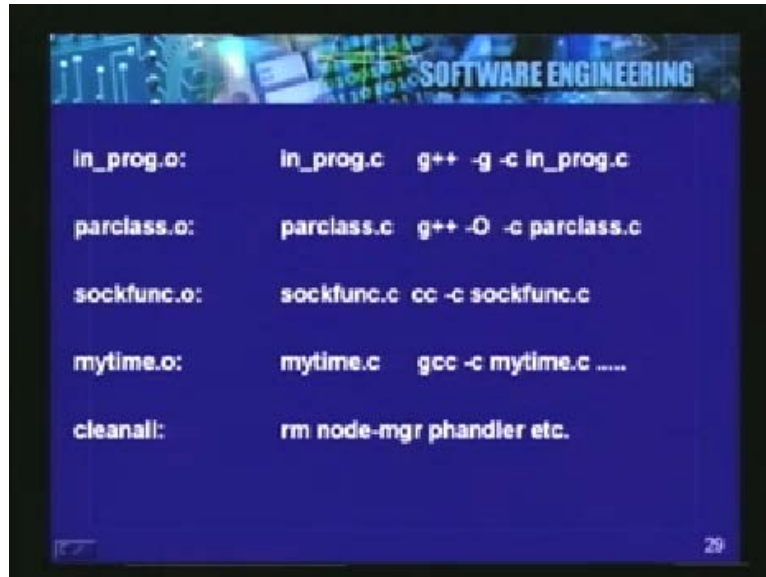
In order to produce 'a. out', you need this and if you have already compiled this, then if this user class changes or if the user file changes, you do not need to recompile this. So if you want to capture these dependencies somewhere and when you simply execute the 'make' file, when you want to build the software, you are building only those units which need to be build because the changes have been done to the files which are being used by the module that you are building.

(Refer Slide Time 47:09 min)



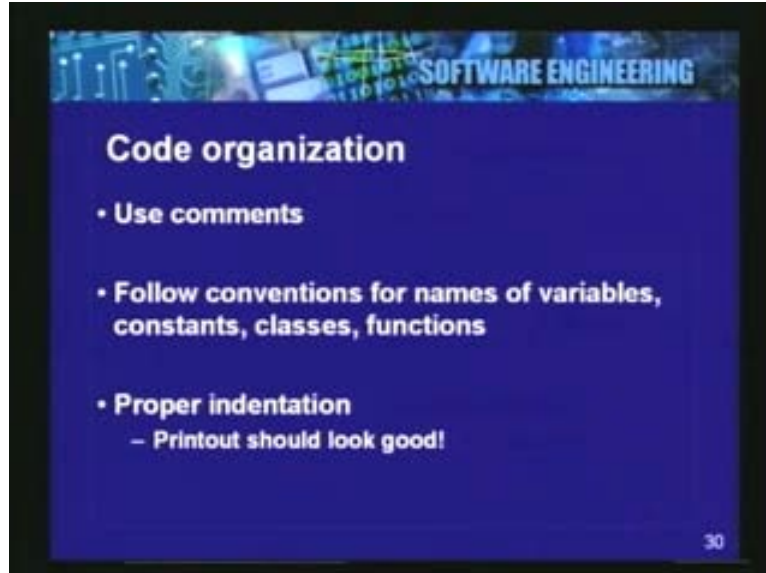
I have an example 'make' file here. You can say these when you want to make something, when you want to make software. This 'whole' represents the entire software and this is your symbol which you are using. These are your actual units; for example you have the 'in_prog', 'phandler', these are some executable files you want to generate. When you want to build the whole software, you need to build these and in order to build in program you need to build this **dot o's**. How can they be built? They can be built by running this command: g plus plus or whatever.

(Refer Slide Time 47:44 min)



This is your sample make file and you can do the 'man' page of make file and find out the format in internet or you can download examples from internet or send me an e-mail if you want have these samples. Then how do you organize your code? You use comments; follow conventions for names of variables, constants, classes and functions. For example class names could start with capitals, variable names need not start with capitals, and then constants can be written with all capitals, interfaces can start with a capital I and then the interface name and so on.

(Refer Slide Time 48:00 min)



Once you follow these conventions, stick to them so that you follow them uniformly and your code becomes easier to understand and it can also be understood by someone else. Once the conventions are known it is easy to follow the code. Then use proper indentation, the print out should look good and this is very important. There are many good guidelines for different languages. You can look at some of these and try to follow good indentation for your code so that the code should look good. When you take a print out it should be good so that it will be easy to follow the code.

Then, when you develop, try to develop your test cases first. When the software is going to be handed over to the customer, how will the customer test? Imagine yourself in the role of the customer and try to list the tests that you are going to plan or the test that you are going to carry out on the software. So write test cases first and that gives a very good view of what is required or what you are going to do with software. At any point of time, let your code be executable.

(Refer Slide Time 49:41 min)



When you start writing your programming assignment, you can just start empty 'main ()' if it is a C file and so executable file is ready, but it does not do anything. But however, you can now add functionalities one by one and then it develops one by one. So you are not writing the entire code in one big bang and then once you compile it you might get hundreds of errors. In order to avoid that situation, if you incrementally develop it and if you keep it executable or runnable from time to time, it may run but it may run partially. So that is not a problem, but getting plenty of errors in one big bang makes it difficult for you to go through the errors and fix those bugs. Some of them could be syntactic and some of them could be logical as well.

Hence the architecture blows up step by step or at any time the software is runnable but with varying functionality coverage. This is the practice which you can put in when you do small scale programming and it comes a long way also and it helps you in your large scale software development. The other good practices are listed here.

(Refer Slide Time 51:04 min)



Use tools which are available. For example 'make' file is available or integrated development environments are available. So use them. There are various kinds of tools and many case tools are available. So whatever is available to you in a given set up, basically in your software engineering lab where you carry out the development process. Try to use tools such as debuggers. You should try to use them for debugging your code.

Then produce good documentation for your [not clear 51:41]. Documentation should not be taken as documentation for documentation sake. It is for the purpose of firstly understanding your own code and then somebody else should be able to understand it, modify it and evolve it. So a good documentation is important. When you analyze your software, if you are following OOAD or you are going to produce class diagrams, interaction diagram and so on, that gives you very good documentation so that you can develop it well, you can test it well, you can understand your own code, somebody else can understand, evolve it, maintain it, debug it and so on.

Pair programming is when you have this practice is where when the code is very complex you can do it in pairs. Sometimes two pairs put together is a good practice. Basically, something that you are not able to handle or not able to see, when another person can see it quickly. So usually it is an effective practice to carry out projects quicker. Then if you have hard bugs, take a printout show it your friend or you can relax for sometime and comeback to the bug. Sometimes a bug can bug you for huge amount of time, for example for few days. It may bug you when you are writing your assignments and you may not be able to find the bug easily. Because you are so much accustomed to your code that you do not even see a bug which could also be trivial. So one good way to fix the bugs is say, if you are not able to fix the bug in say half an hour or one hour you can take a print out of that portion of the code and try to read the print out, maybe you will spot the bug. Or if you still not able to spot the bug, then you can show the code to your friend.

You can apply some of these techniques and these are some good practices which are known to good software practitioners. So you follow some of them, meet your local experts in coding or development or local software engineers so that you can put into practice some of these good guide lines which are available to you. So this is about the factors that can go into quality software development in long run. We looked at measurements and different aspects of measurement and we also looked at some good practices which you can apply so that your software quality improves.