

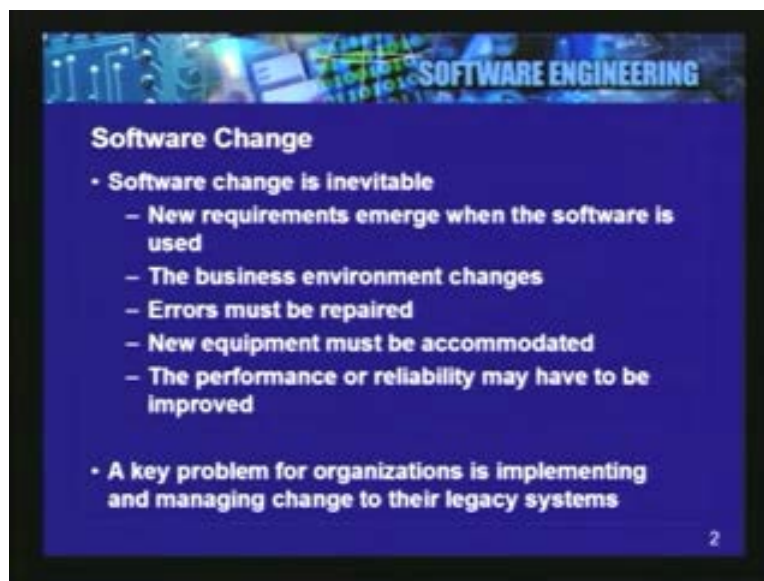
Software Engineering
Prof. Umesh Bellur
Department of Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 25
Software Evolution

So far in the course we have primarily looked at the process of how to create software and the different methods in the methodology that would be needed and useful to actually create a software product for a system starting from the requirements.

Now, once this software has been created it will be put into production and typically has a lifetime even beyond just the creation of software and the process that comes about after the software is created is that of software evolution or that of software change and that is what we are going to be looking at in today's talk.

Now there are various reasons as to why the software must be changed after it has been created. It could **even it could** happen as long as the software ends up living all the way from immediately after the software has been **trust** into production all the way through to the lifetime of the software system which could be up to several years.

(Refer Slide Time: 01:47 min)



Now it could happen that the requirements of the software change as the software is being constructed and there may not be time to fold in the new requirement that are coming in during the processes of creation of software because there are certain deadlines that people would have to meet in order for the software to be rolled out. Also, new requirements can come in; completely new requirements that would have to enhance the software after it has been pressed into service. So, for example, you could have a library system that has been built and one of the new requirements that would come in where the

notion of being able to rent articles for a certain fee or download articles for a certain fee and print them. So here the notion of accounting package could have to be added to an existing software system that was in service.

Other reasons could be that the business environment in which the software is living changes and what could be the result of such a change in business environment. An example of such a change could be that a business might end up buying out another business, the business processes within the original enterprise where the software existed could now change as the result of the buyout or the merger acquisition or the merger as it is called in business terms.

So, an example could be that here is a telecommunications company it is providing data communication service and it was providing it using high speed DSL technology so they were providing data communication service of 144 Kbps Kilobits per second or more but now they wish to also provide dial up service to compliment this high speed data communication service that they have and so they go and acquire another company. So, as a result of acquisition it could be that the original company which is providing the high speed service now wishes to change its business process to say that if a customer coming in wants to start out with dialup where high speed services are not available he can choose to sign up for that service.

Now the change in the business process typically reflects would have to be reflected back in the systems that are now running this service industry. So this is an example of a business environment changing as the result of which you have to change something. Also there could be errors in the software that has just been put out. So, for example, a button was mislabeled that could be a simple error in the software all the way to complex logic errors that could occur. Hopefully this would have been **readed** out during the process of software testing but no testing system is guaranteed to deliver software that is completely bug free as it is called in the software world and therefore these errors would have to be repaired and that is another reason for software having to change; an algorithm might have to change, some code might have to change, some interface might have to change, some UI might have to change as a result of this and so on.

Now it could also be that new equipment might have to be accommodated and a very classy example of this is the change from the mainframe error where we were building software to run on very large servers down to the scenario today which is lots of cheap servers that are networked; the smaller cheaper servers that are networked to one another creating what is called distributed system so this can cause a change because it is much cheaper to buy and maintain the smaller servers; the company might decide as a policy to move to this new hardware architecture as a result of which software would also have to change.

Certainly the performance or the reliability of the software may have to be improved because as things start out and software has rolled out initially it could be that it was built to be reliable say 95 percent of the time because that was enough to meet the business needs of the company at that point in time. But as the company grows to rely more and

more on the software the availability may have to be pushed up to the 99.9 percent or more which is what is being asked of today of software around service companies typically. So all these reasons exist for software changing and as the software changes different kinds of changes take place within the software environment so there needs to be some kind of a coordinated process or a strategy to deal with this change and a key problem that has always been is the process of managing change to the legacy systems within this environment.

So, software change control is nothing but a process just like there was a process for building of software in the first place building something from scratch the process for integration and so on there is a process for software change control and that is the process of managing changes to the software system itself and we should see what the process is during the course of this talk. So there are different strategies for managing change and this can also be due to the different reasons that may be causing the change within the software itself.

(Refer Slide Time: 6:57)



The first thing is that there is a software maintenance strategy that comes about to handle small changes to the code. Typically these kinds of changes do not change the underlying structure or the architecture of the software system itself; they are typically meant to fix problems or errors within system or accommodate very very small enhancements or new requirements within the system. So, an example of such a change could be that I now wish to add a history button which will give me the history of the last ten commands typed on this software system and this was not there before I wish to add something like this.

So a change such as this may not necessitate any structural changes to the architecture of the software. So, for example, if it was a client server system with two servers coordinating amongst themselves in a distributed fashion this does not have to change to

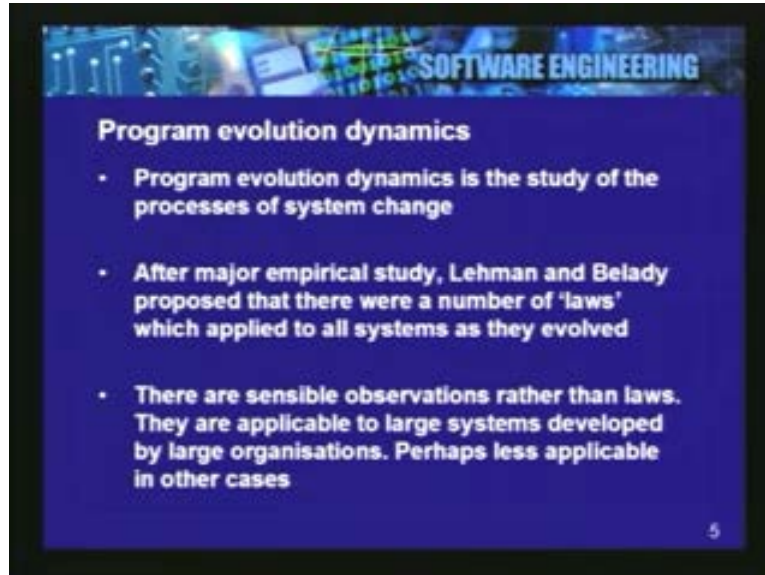
accommodate the new requirement that has just come in. this is the software maintenance strategy; so nothing very fundamental about the software changes and incremental changes or the code is typically done.

Another one is what is called architectural transformation and this strategy is one that will actually cause a significant change to the structure of the software itself and it typically is done because you are moving to a different hardware platform for example, the server **to main the server** or the mainframe to the distributed system was one such example for this. This architectural transformation could also be a result of, for example, you are changing the business processes that the software is automating is becoming much more complex as a result of which, for example, you did not have rules engine that was sitting within the software before but now the complications and the complexity within the software actually force you to embed a rules engine into the software so this an example of architectural transformation; you are changing something fairly fundamental and when you introduce a component such as the rules engine into the software that interact with many different modules within the software that interaction has to be very carefully controlled. So here is a major enhancement that is being done; it can be architectural in order to future proof the software as well in this particular case.

The third one is what is called software re-engineering and this is also an example of a fairly major architectural trend. So, moving from a two tier to a three tier system could be an example of such a change and this change could be because you want better maintainability with the code base that you have, better maintainability with the application or it could be because you want to effect a major performance enhancement in the software you want a scale from a hundred users to ten thousand users so you would have to re-engineer the software such that no new functionality is now being added to the software but you are still effecting a major change in the underlined structure of the software.

So here are different causes and corresponding strategies as how you would end up dealing with it and these can all be done either in isolation and typically **that is** the recommended strategy is that you know to attempt one or more major changes together and you tend to keep these as separate as possible so that you can rollback if one of the changes does not really happen the way you wanted to go and you can roll back the change completely back out the change and continue the other changes which are happening in parallel with the software.

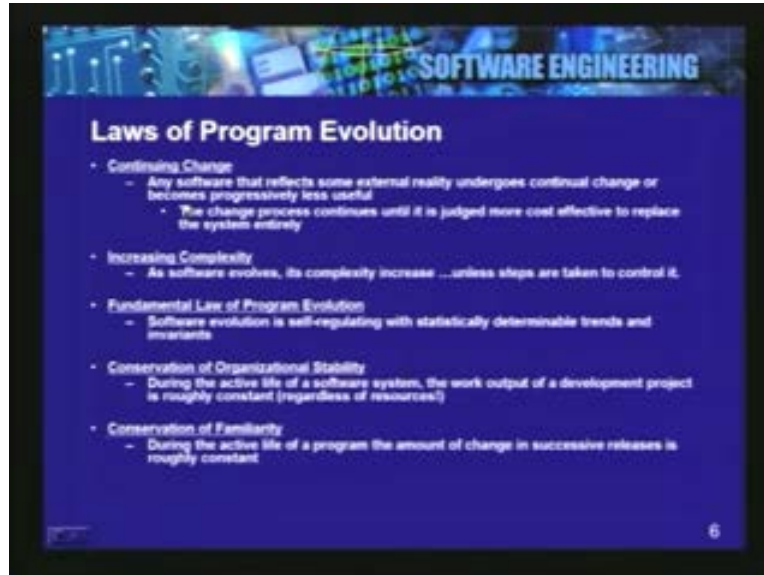
(Refer Slide Time: 10:44 min)



So the dynamics of what is called program evolution are the changes that happen in the software. The program evolution dynamics are simply the study of the change itself. for example, how many changes took place to this software system after it was deployed, what are the kinds of changes that were taking place; mainly bugs were getting [10:50.....doubt] by the major new enhancement that were getting done but there were structural changes that were being affected and so on. So, people have obviously studied these system and Lehman and Belady were a couple of engineers who studied major software projects and came up with what they called as set of laws or Lehman's laws which applied to all the systems that they studied as far as the evolution of the software went; that is, the period of the software life cycle after it had been initially put into service and more than laws I mean **these are not mathematical** there is no mathematical basis for lot of these these are empirical in nature so **they have** it is just an observation based on the studies that they have done and they are just sensible observation rather than laws and they have been studied basically in the context of very very large systems development so these are large enterprise system say ERP for a company like GE which is worldwide in nature, which has hundreds and thousands employees, which moves literally hundreds and thousands of products across the globe so the evolution of the systems that controls something like this is primarily the subject of Lehman's study.

We will just briefly take a look at the observations or the laws that they came up with they are called laws of program evolution.

(Refer Slide Time: 12:14 min)



The first one is that of what is called continuing change and what this is primarily trying to say is that any software that does not have to change after it has been deployed is really going to become useless very quickly in the sense that the environment in which the software runs is pretty dynamic always almost always in the context of large systems because things are changed, business rules are changing, architectures are changing, hardware platforms are changing, software platforms are evolving so you may move from let us say Windows 98 based system to a Windows XP system you may move from a Linux 2 system to a Linux 3 system and so on and so forth. The database versions change on which the systems run and so on. So there is constant change around the software and the software typically has to react.

So any software that does not require any kind of change is quickly going to become useless and this process of change will continue until it is judged that it is no longer cost feasible in order to keep affecting these changes or rather just build a new system instead of clean out trying to patch the whole thing. It is very similar to a car that you may end up owning. So, after a few years, after owning a car, if it is a brand new car that you bought typically things will start to break down; the windshield wipers need replacing, the carburetor needs cleaning and so on and so forth; the tyres need to be replaced once and all, the battery needs to be replaced and so on and there comes a time in the life of a vehicle that you own that you will judge that it is no longer feasible to just maintain this vehicle, the cost of maintenance is running so high that is probably far cheaper to get rid of it and go in for a replacement a new vehicle; it could happen to almost anything in your house, household appliances and so on; the same is to a software. So that is what the first law is really all about; it undergoes change progressively and if there is no change there is a period in which no change really occurs then it is kind of time to retire that piece of software and move on to the next evolution of the software, also, the laws of increasing complexity.

Now typically software when it is architected from scratch or when it is architected on a clean sheet of paper is build in a fairly modular clean way so that it is very maintainable at that stage but as the software involves and changes are made to it several patches are applied to it typically this process if not carefully managed will cause the complexity of the software to increase; the complexity you mean things like coupling, the cohesion, kind of starts breaking down, the degree of coupling between different modules may increase and these are some of the metrics that you have already seen in earlier lectures and so these metrics basically tend towards a situation which the software is not very healthy in nature and as the complexity is increased you have to have a really solid process to keep a check on it otherwise you are going to collapse into the first law where you will really have to..... a time comes to retire and move on to the next evolution of it.

The third law is called the fundamental law of program evolution and basically what it says is that software evolution itself is self-regulating in nature. So, if there are too many changes and the change is stopped because it is no longer possible to make the kinds of changes that are being made continually. Also, there are certain statistical trends that can be observed so there are average number of changes that takes place say every month for example or every three months once in a quarter and so on and there are certain observable variances around these kinds of averages or mean and those are fairly observable. So it is not a random process of change that is happening all the time and this is the observation of these two people over several different projects that they have studied.

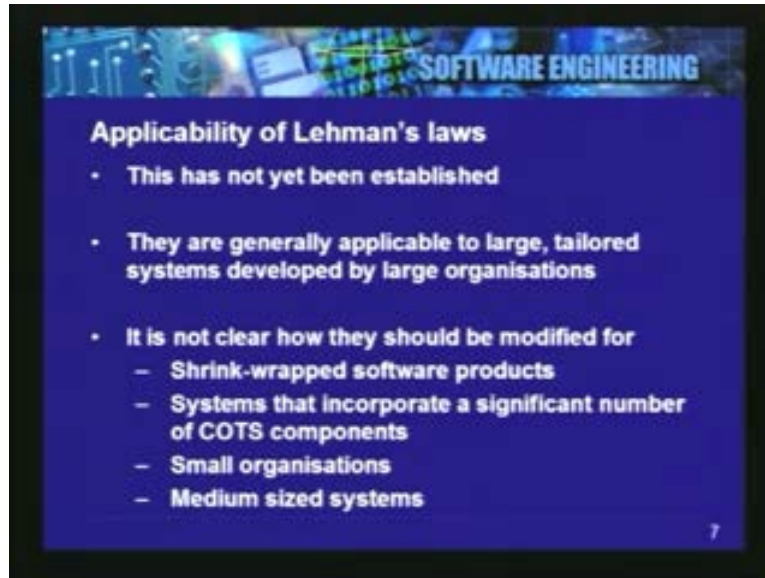
Also, the next law is one that Fredric Brooks has also stated in his software engineering book which is just throwing people; more people at a software project is not going to help it happen any faster. This is stated slightly differently here. What this states is that no matter how many people work on that the output is going to be constant of a development project so the more people you end up throwing at you probably just create more work for yourself and you are not going to end up producing anything more out of that same project.

And finally the law of conservation of familiarity is that if there **are if there** is a large amount of change that is taken place from one version of software to the next then the less familiar that you are going to end up becoming with the software because the software as you knew it as an architect or designer is now no longer going to be how it exists. So, during the lifetime of a program then the amount of change between successive releases has to be a constant and if that is you know widely varying then your degree of familiarity with the software goes down as a result of which the maintenance cost suddenly shoots up because you now had to spend a lot more time trying to figure out how this software is doing what it is supposed to be doing earlier.

So again like we stated before these are not really laws; these are general observations that has been made admittedly in the course of a fairly comprehensive empirical study that was done by these two people. so they are again they are generally applicable to very very large systems, they are applicable to systems that are developed by hand based on some specification you know these are not things like computer games and so on but

these are enterprise system that tend to run large companies like the financial systems of large companies and so on.

(Refer Slide Time: 17:53 min)

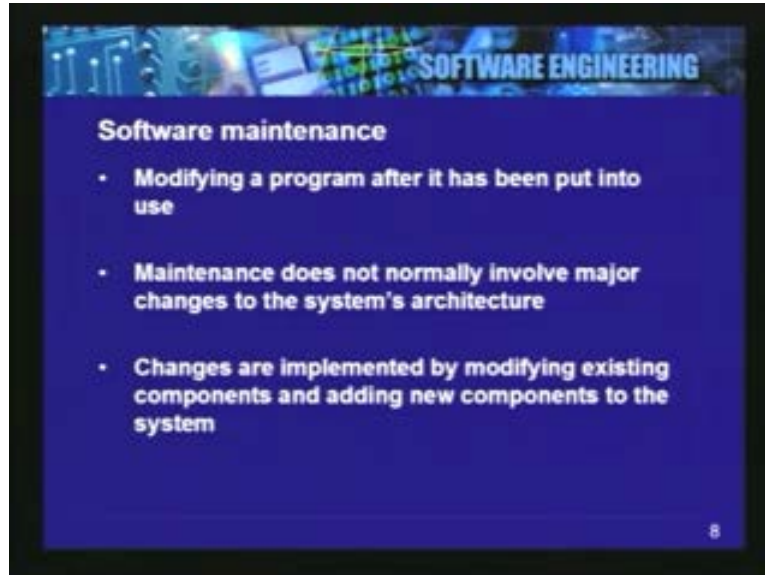


Also, it is not clear how these things end up applying to, for example, shrink wrap software product. shrink wrap products are those that you go out, buy in a store, come back and install and it works; things like virus scan, things like Norton utilities for Windows those are examples of shrink wrap products and systems that incorporate significant number of COTS component what it really means are those that built out of integrating a lot of pre-existing pre-built components rather than systems that has been developed from scratch without any large degree of reuse with systems built and used in very small organizations, medium size systems and so on.

So **to take** you have to take this study with a grain of salt but largely what they say is common sense and you see this for yourself as you get involved in the evolution maintenance of these systems.

So let us look into the first two categories of change or the two strategies that we discussed earlier. Remember, software maintenance was one of the categories and architectural transformation was the second one; the third one is the software reengineering which is something that we will not go into today or during **the course** this course but it is something you can study for yourself; there are several books that have been published in the notion of software reengineering, business process engineering and so on. So we will focus mainly on the software maintenance and architectural transformation, kinds of changes that get introduced.

(Refer Slide Time: 19:14 min)



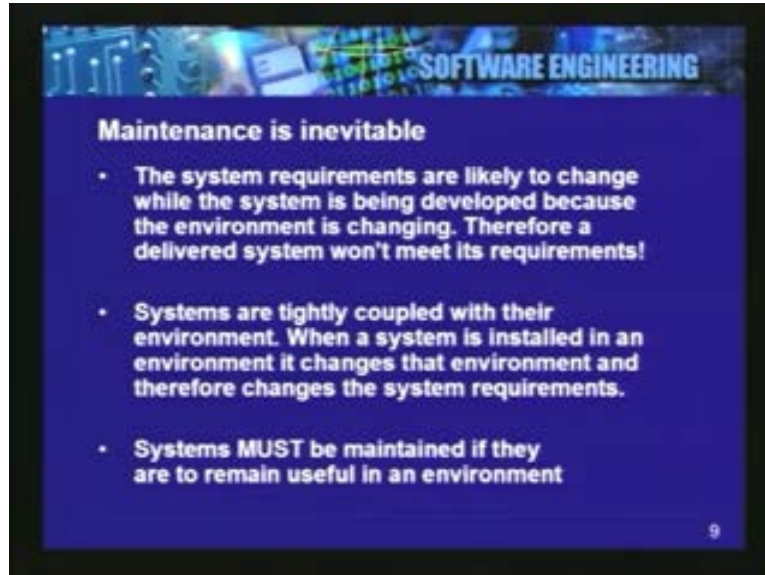
So what exactly is maintenance?

We said that maintenance was primarily, it had to do with small changes, they had to do with new requirements and so on. So, maintenance fundamentally is changing a program after it has been put into place; that is the basic definition of maintenance. And like we said before, it is constrained to smaller changes and there is no defining line or a rule to say you know anything more than thirty lines of code for example is a large change and anything less than thirty lines is a small change. So this is something that is best done by gut feel; if you feel that it is going to change the underlined design of a system and it is no longer a maintenance issue at that point; it is an evolution issue, it is a re-architecting kind of an issue and you have to treat it differently as well.

The testing cycles that you go through may be entirely different is one good example of how these two things are treated differently. And the changes in the case of maintenance are typically affected by modifying the existing code and components rather than adding new components or adding a lot of new code. So the example that I gave of adding the business process rules engine in order to make it easier to setup and change business rules in software is a major change; you are adding a new component which is a rules engine in this case and that does not come into the heading of maintenance.

So we will explore the process of maintenance a little more today. Maintenance is bound to happen for various reasons. One of the reasons is that **software is** the process of testing software is never perfect; it is almost impossible to test the entire gamut of input ranges that can be applied to a particular piece of software. So, for example, if there are twenty parameters and every parameter can range among hundreds and thousands of values it can end up taking it cannot be possible to test the software, every possible combination of input that it can come under.

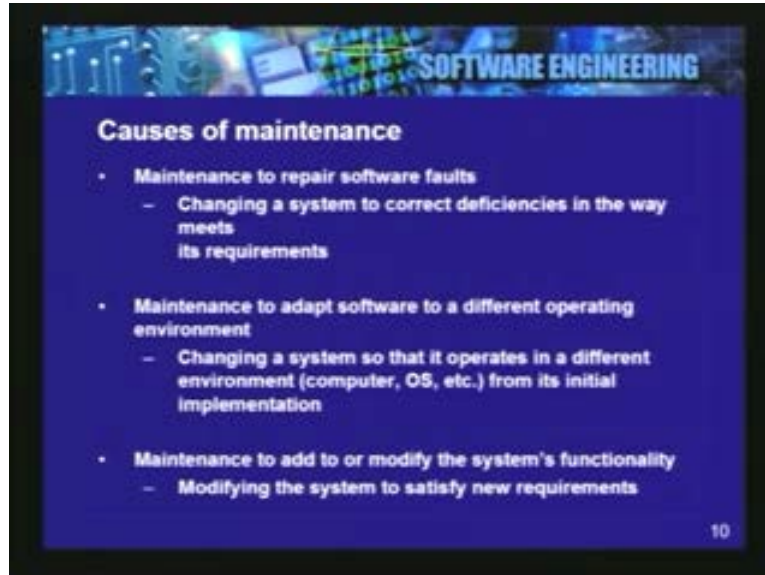
(Refer Slide Time: 21:25 min)



So testing is never perfect in software. Every single path in the software to exercise that would probably take years for any kind of medium to major system. So testing is done selectively as a result of which there could be bugs or errors that are left over when you compare to the original specification of the software because this can never be completely automated and it is done by humans and also there is a conscious engineering reason for limiting testing in particular areas. This could result in certain left over errors that would have to be fixed once the software is **put into** put into service. And this is a risk issue; you (go....22:00) the amount of risk that exist by doing a certain amount of testing to a particular module of the software and if the risk is something that you can live with you can go ahead with that limited testing.

The other thing that can happen is that there is evolution around the software. The software itself may be okay but things like environment that they end up getting coupled to..... so the software runs on the top of an operating system, applications runs on the top of an operating system and the operating system's versions change; newer versions come out with some advantages, the business wants to move to those new versions which means the software now has to adapt. It might be a matter as simple as just recompiling a software for a newer version of the operating system but then on the other than it could be that such systems could fall into the face of change and you would have to make the corresponding changes based on the application as well if it is not been architected in a perfect way to insulate you from these kinds of changes. So they must be maintained if they have to remain useful in the current environment and maintenance is therefore something that is inevitable.

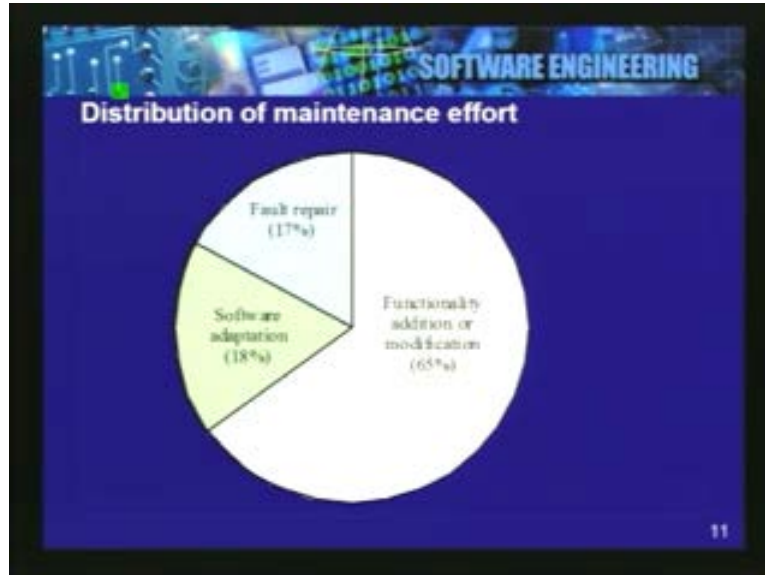
(Refer Slide Time: 23:23 min)



So there are different reasons as to why maintenance can come about and we just saw couple of them. So maintenance to repair software falls as the first one that we actually discussed. Maintenance to adapt the software to a different set of operating environment is the second one which we also discussed the third one which we really go into is the notion of small features and enhancements that get added to the system.

The old example that we used was to add a feature that would give you the history of the last ten commands which we typed up. Now if this were an order processing system may be the user wanted an additional field or a label on one of the screens that showed a certain detail. So, for example, the zip code might have been missing and you have to add the zip code or the second phone numbers, the cell phone number of the customer might have not been there and this is the new thing they want to add because lot of people end up carrying cell phones today so the addition of a small field may not result in a major change in software; certainly no new components now are required in order to add a single field to a screen in the UI especially when the data is already present; now if the data would not be present that would cause a slightly larger change. So every change has to be evaluated with respect to what is there and what is the change actually happening to what is there. So the maintenance to add or modify new features is also another cause of maintenance that can come in.

(Refer Slide Time: 25:05 min)

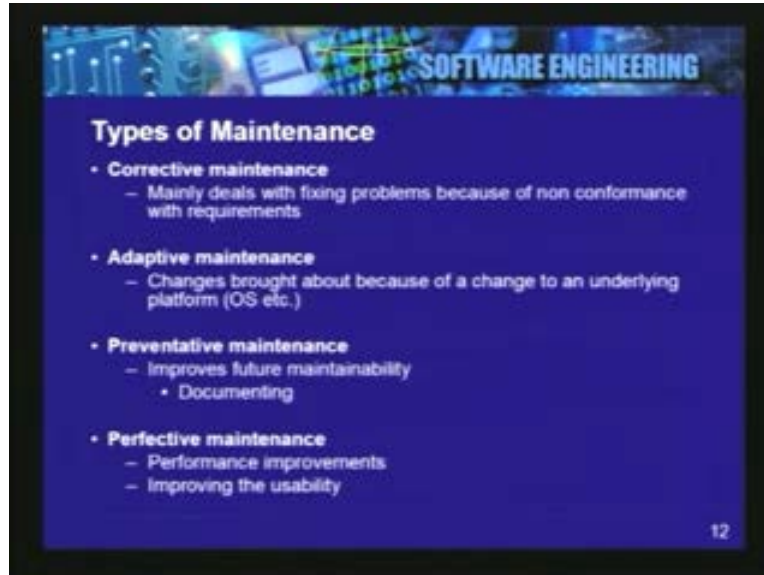


And if you take a look at the breakup of the maintenance effort in any software organization it largely breaks down like this. So, after three categories the fault repair which you would expect would be a fairly major category actually the smallest. So, less than twenty percent of the change that takes place in a software system is to repair faults that exist in the system. then software adaptation that is the environment changing, the operating system's version changing we underline infrastructure on which the software depends; say the database for example changing and the software changing to meet those environmental changes is also about 20 percent it is not very large and the multitude of new features the small changes that you would come in the form of new requirements or added requirements of the software is really the largest set of changes so close to 65 or 70 percent of the changes end up falling into this category.

The study of this is useful from the perspective of realizing how you want to model your process and whether your testing procedures are adequate or not. So for example, the fault repair section was 85 percent instead of 70 percent in this case you could argue that your testing procedure is completely inadequate and you had to go back and look at them as opposed to **changing the** working on the change management process by itself.

So another perspective on the same kind of issues we have been discussing is we can look on maintenance as being of four different types. One is corrective maintenance and corrective maintenance is simply the fixing of bugs; adaptive maintenance which is changes brought about because of the change in the environment; there is couple of things which we kind of did not look at and we kind of dumped it under the heading of new requirements and we can certainly add that here as well but that would be a fifth category if you work in types of maintenance.

(Refer Slide Time: 26:40 min)



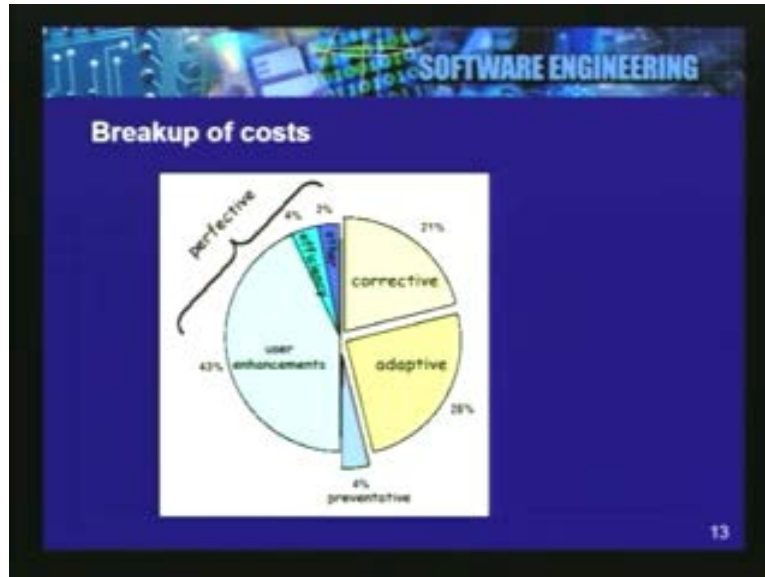
But preventive maintenance is really which I am going to performance tune; I am going to fix it so that it kind of gets future proof; an analogy you can go back with respect to owning an automobile or car is that you typically end up changing the oil and the oil filter on the car every 3000 kilometers that is preventative right; you do not want to run into problems later so you are going to do this once every 3000 kilometers. very similar kinds of changes that you can make with the software and example of a very simple change is to document the code as you go along; it is often not done when it needs to be done which is during the development process and preventative maintenance simply says you go back, go in reverse, **engineer** it to document the architecture, to document the design so that anybody coming in after that would have to fix the bug, would have to add a module here and to make changes to the module will immediately know what structure the software has and they will not spend as much time in hunting down the different aspects of the software itself.

And lastly the last category is what is called perfective maintenance. Perfective maintenance can be things like improving performance. So you might have met the performance requirements of the goal of let us say response time of less than five seconds for a particular function of logging into a system but you may want to improve on that even more even though it is not a requirement you may want to improve on that because you know that the software is going to have to scale from a hundred users today to ten thousand users in a couple of months so you may make changes to the software to say I am going to perfect it; I am going to make it take as smaller amount of time as it possibly can; this is the theoretical limit which we can go and I can bring it down to the theoretical limit and so on.

Improving the usability is a never ending process; it is an excellent example of a perfective change **because** it is never very easy and it has no metric to say the system is perfective in terms of usability; **it is always the** it is a very subjective topic on the metric

and you can always keep improving on it as you go long. So, if you take a look at the break up of costs with respect to the types of maintenance that we just discussed it is also an interesting study to do and what this shows is that the preventative maintenance is very very small which means it is usually not done and the breakup of cost is really based on the empirical studies that have been done it is not any kind of formula that exist, it is just based on the empirical studies that have been done across a number of software systems from the last twenty five to thirty years.

(Refer Slide Time: 29:00 min)



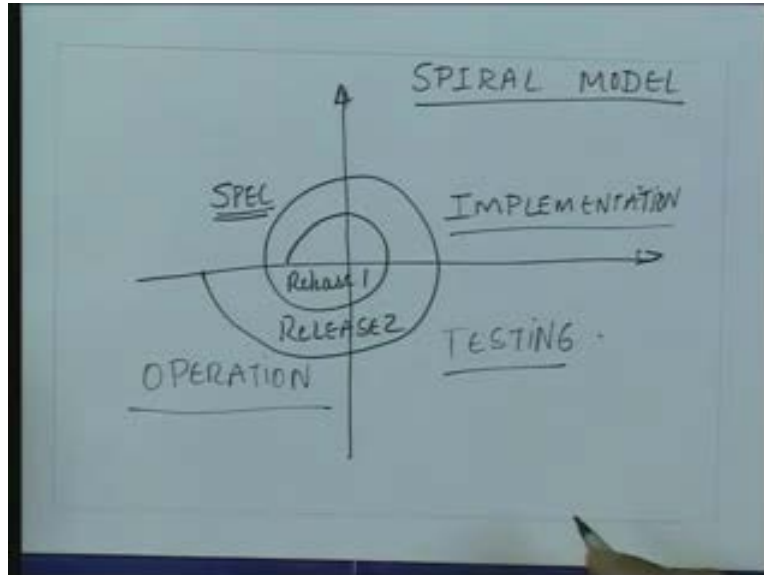
So, preventative maintenance is really very very small in this case; adaptive maintenance is about 25 percent which is something that deals with the 17 percent figure that we saw in the earlier slide, corrective maintenance is also about 20 percent which is again very much in line with what we saw earlier, user's enhancement being about 50 percent; again it is very very similar and user enhancements can be considered efficiency improvement, performance improvement and so on can be considered to be perfective in nature where the rest of them can be considered to be preventative, adaptive and so on.

(Refer Slide Time: 30:07 min)



Now, maintenance models can be very similar to the development models that we have talked about. So, if you take a look at these spiral models that existed we can have a spiral model of maintenance as well. You are already familiar with the development spiral model so you can have something very similar to that. In different phases what you are basically doing is this is the specification phase (Refer Slide Time: 30:32) and this specification phase when applied to maintenance is really that you are collecting a set of requirements and then you are deciding that it is going to go into a particular release so this can be for example, the first spiral may be released 1; the patch level one of the software, the second spiral may be released 2 and so on and so forth and you can have many spirals as you want so the first case may be simply the specification of the changes that happen; the second phase would be the implementation of the changes; the third phase would be testing or validation of the changes that occur and the fourth phase would be putting it into operation.

(Refer Slide Time: 31:23 min)



So we can have something, a process a changed process that is very similar to the spiral model of the development that you have seen except that it has been applied in a maintenance mode in this particular case.

One of the important things to go back and look at is the notion of maintenance cost. What is it that ends up costing when you **when you** go to maintain a piece of software over a period of time and one thing that we notice very quickly is that maintenance cost are almost two to hundred times more than the development cost of the software itself.

(Refer Slide Time: 32:00 min)

The slide has a dark blue background with a header area showing a circuit board and the text 'SOFTWARE ENGINEERING'. Below the header, the title 'Maintenance costs' is written in white. A list of four bullet points follows, each preceded by a white dot. The text is white on a dark blue background. The slide number '15' is in the bottom right corner.

- Usually greater than development costs (2* to 100* depending on the application)
- Affected by both technical and non-technical factors
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.)

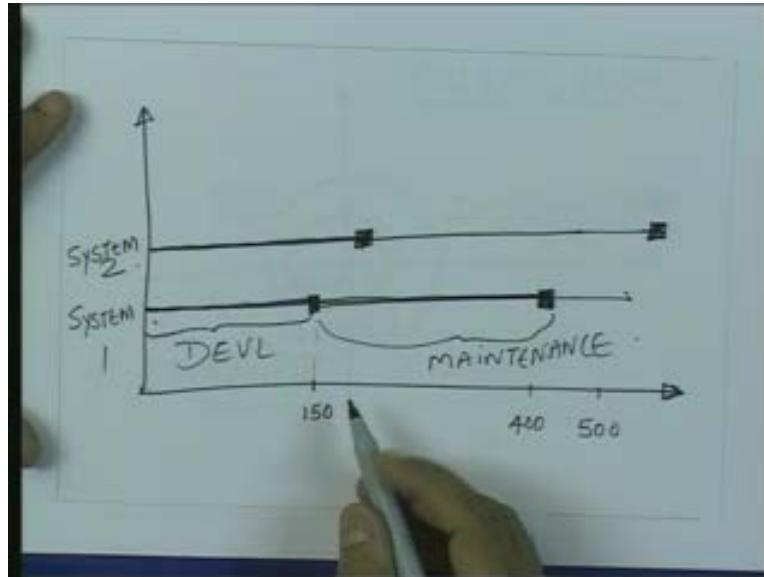
Again, based on more empirical study, there is not any formula for this but having studying lots of applications over the past couple decades people have realized that software maintenance cost accounts for quite a large part of the total life cycle cost of the software. It can be affected by lots of factors including social factors such as maintenance as usually considered, a low quality task to do so they do not get very bright people to come and do this and so on and so forth. It is affected by both technical and non-technical factors in other words and the cost keep tending to increase.

Remember the laws of complexity, Lehman's laws of complexity; the complexity of the software increases, the familiarity of the software goes down and the cost tends to increase as a result of both these factors. So ageing software have very very high support cost and that is when it is time to replace the software. So there is some line you end up drawing within your organization to say that on a per year basis this is the maintenance process **causes** this I am going to replace the system with a brand new system which we are architecting in this pointy in time.

Coming back to the cause issue that we were just been discussing let us take a look at an example of two systems that we have studied empirically and I have kind of drawn a graph here that shows what the relative cause of development in maintenance were for these two systems; what is graph is showing is in hundreds of thousands of dollars on the x axis what are the cause and here it is just showing two different systems over a period of time; system one basically took about 150,000 dollars to be developed these are the examples of real systems that we have looked at and over a period of the next three years it took about additional 250,000 dollars to maintain so the investing in the building of the system **would it turned out** as it turned out was not such a big deal as investing in the maintenance of the system.

Another example that we also studied is the second system here is the large ERP system which we took above 175,000 dollars to build and then you know the maintenance cost were anywhere between 350 to 400,000 dollars over the next three years which is well over two and half times the original cost of building the software.

(Refer Slide Time: 34:07 min)



So you see why it is important to pay attention to the **to the** notion of maintenance **as** a lot of attention has been paid to development; development methodologies exist, development tools exist, IDEs exist and so on and so forth, CM systems exist but not as much importance traditionally was given to maintenance until these problems were realized through extensive empirical studies and that is why we study this as part of the mainstream course on software engineering today.

So what are some of the causes? I mean why do these things come in is the question that we are going to ask ourselves naturally and the first thing is the stability of the team itself. Like I said there are some social factors that come into play in this case and the social factors essentially are that it is not very good to go into a maintenance job as opposed to a development job so the staff involved typically the turnover rate in the maintenance position is very high people tend to come in and if they get a development opportunity they tend to leave or migrate out of the group so the stability of the team plays a very very important role in driving up maintenance cost. New people are coming in and they are taking a lot more time to understand the software as a result of which making a change and as we see later a large percentage of time and money that goes towards making the change is spent on understanding the software itself. So, making the actual change is a much smaller portion of the cost within this so the stability of the team is very very important in this case.

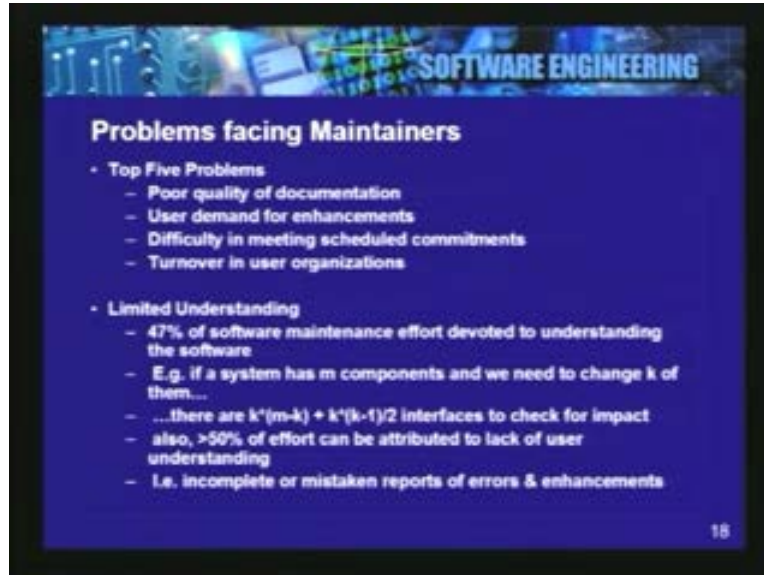
(Refer Slide Time: 35:17 min)



Also there is a 'throw it over the wall' kind of attitude. Developers typically have no contractual responsibility for maintaining the software so they build it and then they say we have done with development let somebody else take care at this point in time and that attitude would have to change as well. So the skills of the various tasks involved clearly, the complexity of the system that exist, if it is a very large system with high degree of coupling very many modules then obviously maintenance costs are going to be high something like this. If the modularization itself is not done right then the cost of maintenance is going to be very high, if the architecture of the system is not right the cost and maintenance are going to be high. So, whatever is done during development is going to have a significant impact on the cost of maintenance itself.

And as we have seen in one of the Lehman's law as the program ages and more and more changes are made to it the complexity just explodes at some point in time and making any further changes is going to prove very very costly. So the problems that face maintainers typically are again as a result of poorly followed processes during development largely that is one thing, so poor documentation; **people just** developers just do not take the time and the effort to document what is it that they are coding; is the design documented? Is the architecture documented; is the code documented; are there proper user manuals for this; does it give all the exceptional conditions that can arise in a software and so on.

(Refer Slide Time: 37:42 min)



The second thing is user demand. Once the software is put in front of a user they immediately start asking for changes because they realize..... oh! this could have been done like this, I wanted this particular feature instead of this one that you provided. So maintenance typically get inundated with request from users the moment that it kind of hits the users' screens. So therefore there is difficulty in meeting schedule commitments so lot of commitments would have to be made in that point in time; new version of the software have to be developed; the software is going to evolve from perspective of feature side itself but then there are all these other minor changes that had to be made as well and therefore both of these have to be put into some kind of a schedule that can be worked.

Also turn over and user organizations we already talked about. So here is a small example basically a software has m components and we need to change the key of the components then the number of interfaces that we would have to check for impact are significant as it can be seen here: so k star m minus k is really the number of interfaces that would have to be changed and each interface would end up interacting with the others as well. So lot of the effort can essentially be attributed to lack of user understanding. For any one person to understand this complexity is typically impossible and when there are multiple people and there is turn over in the organization all of these are compound factors. So this is also something that we have seen earlier as what are the different approaches to maintenance.

The first approach in terms of (.....39:10) that you throw it over the wall and say it is not my head ache somebody is going to maintain it and then this has the obvious disadvantages that the people who developed it are the people who are most familiar with the system design, architecture and the code yet if these are the people who are not involved in maintenance it is going to take a lot more cost, time and effort in order for somebody else to come up to the same degree of familiarity with the code and system. So

that notion of machine orientation needs to exist which typically does not which is that the development team itself makes a long term commitment to look after the maintenance of the software; some part of development team certainly needs to live on and provide the **quantinity** that is required from development to maintenance and there are different models that exist; there is the quick fix model in which I am just going to change some code so I am not going to go through an exhaustive design process in this case. So, if I were to add a feature it might require a design review, change review, test plans have to be looked at and so on. But if it is a very very small bug that has to be fixed it just might be a quick fix change to code; this is the first approach to maintenance that we can take and the disadvantage of this obviously is that it rapidly degrades the structure of the software if you keep changing the code, if you keep adding a few lines here a few lines there you do not document it you do not write new test cases for it and pretty much pretty soon you are going to be at a situation where this is non-maintainable.

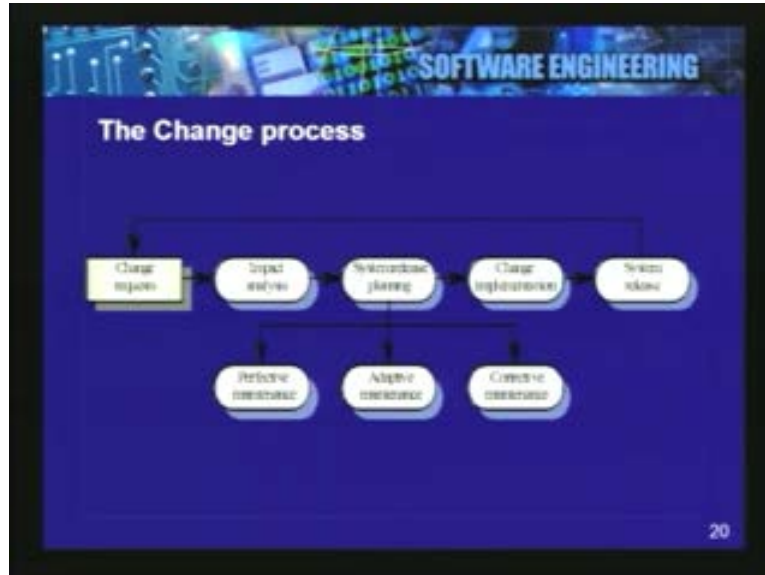
(Refer Slide Time: 40:12 min)



Then the other one is the iterative enhancement model. you make very very small changes, you see the effect of these changes by running through all the tests and then determine if there are no side effects then you incorporate the change and then you go to the next small change and so on and so forth. The full reuse model basically is trying to say that you start as if you were starting with requirements for a new system and you reuse as much as possible from the old system that exist so it is almost as if every iteration o the software were a brand new system that you are building except that you have reused everything from the old system. But this is typically not very practical to do. If you have to make a quick fix and you have to turn around to fix in the matter of twenty four hours which is often the case something that cannot be done as a full reuse model obviously.

So let us take a look at the process of change itself. I am just going to kind of write out the process here and we will follow it as we go long.

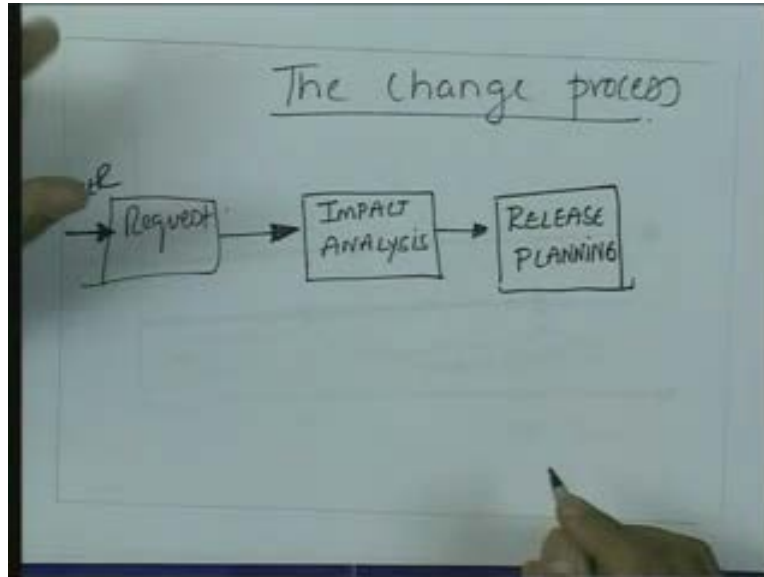
(Refer Slide Time: 41:42)



The change process typically starts with what is called a change request. This is the first step in the change process. The user is the one that submits the change request; it can actually come from the developer of the system as well who feels that something needs to change. The change request can be of many types: it can be a user directed change request, it can be a change request driven by the engineers of the system itself and so on. The moment the change request comes in the next step is to do an analysis of the impact that the change is going to have.

So you might want to go back and say is this going to cause a structural change to the software; is this nearly going to cause a change in few lines of code; do the new test cases have to be written for this, do the existing test covers the change that I am making and so on and so forth. So that is the impact analysis so that will come back with, for example, a category of change that would have to occur: major, minor, medium level change etc etc; it could also give back an effort level analysis of how much time would be required; what is the modules that would change so that is the impact analysis output. After this it has to be scheduled so you would have to schedule this for a particular release of the software so next step is what is called release planning.

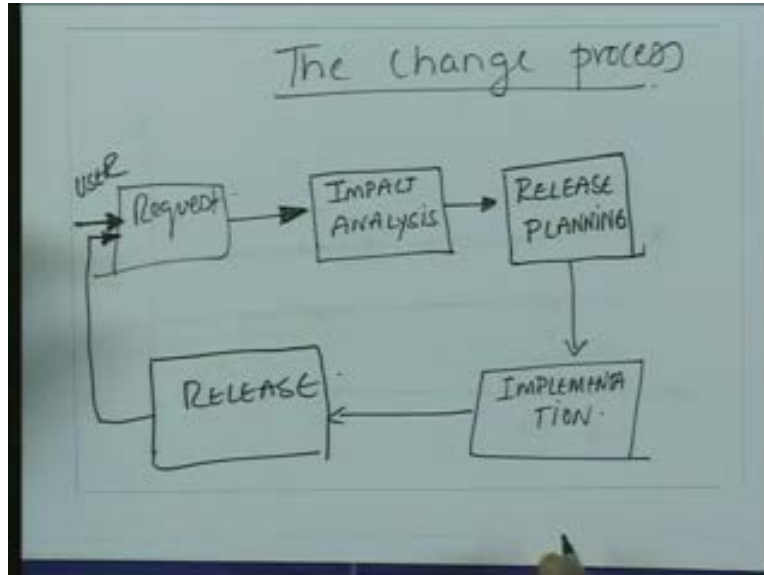
(Refer Slide Time: 43:21)



Every change that is made to the software introduced via change request has to be schedule for some particular release. every release can be tagged with a particular number, for example, you release 3.0 of the system today which is a major release typically and 3.1 is a minor release that contain bunch of features, 3.15 may be of a patch release that basically fixes some errors in the software and these errors or changes may have come in from change request or may have come in from the user who reported errors or errors that were discovered during additional testing and so on.

So release planning is the next step that is done after impact analysis and as soon as release planning is done then the implementation of the change can take place and the implementation of the change does not necessarily mean that which is going to be straightaway start writing the code; it could well be that a design review have to be held to be saying that this module is changing or we are appropriately considering the effect on the different interfaces; how are we going to effect this change, for example, the notion of the business rules engine that I talked about; fairly a major change can also come in through a change request but this is going to necessitate a design review to take place for example that is something you would have to think about and finally once you implement it then you will release it; it is the release process itself and the process of release itself can introduce new changes so this could be a cycle.

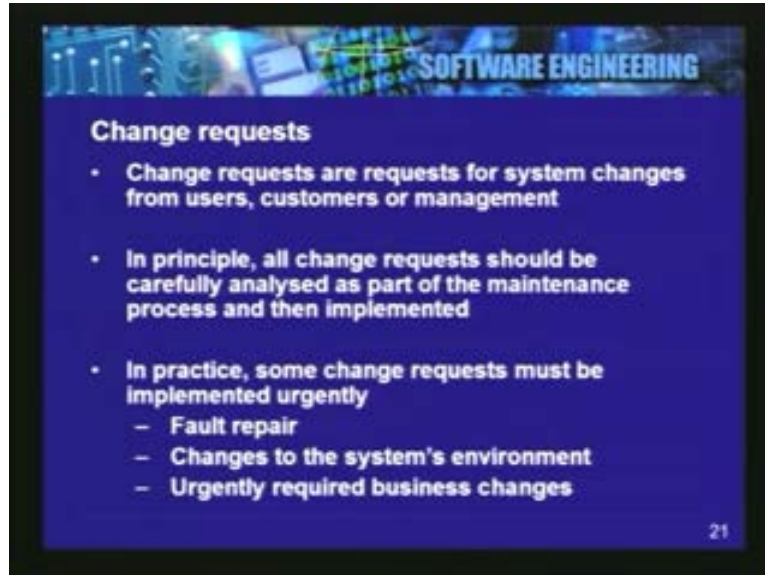
(Refer Slide Time: 45:02 min)



So changes can come in either side from an external source as has been set here as the user or it can come in from the internal sources as well and this can be applied to all the three categories of change; especially the release planning and the impact analysis modules are very different for perfective maintenance verses adaptive maintenance verses corrective maintenance and so on **corrective maintenance and so on** so that is something we have to be careful about. But typically a bunch of related corrective maintenance features of bug fix is going through a single path release; you do not introduce new features into a path release; you might introduce a minor release where a bunch of users enhancements that have been requested which are essentially new feature developments but very very small features are introduced and you never mix that along with the change to an environment. So, for example, if we are changing the operating system version you do not want to mix that along with the bunch of enhancement that are being released. You want to try and keep these things as separate patches as we possibly can so that each thing can be individually applied without it affecting the other changes in the system.

So, change request which is the first step in the process is simply a request for system changes and this can have a particular template or a format, for example, who is the user submitting the change request, was it a bug in the system, is that a new feature to be added in the system; if it is a bug in the system then to reproduce the exact conditions under which the error occurred; so, for example, the user was on this particular screen he was trying to enter this kind of data and the system crashed; that could be an example of an error report that is coming in or it could be a change request for a new feature; I already told about the changes in fields, I wanted to change the control flow of a piece of software in some way and so on.

(Refer Slide Time: 46:59 min)



In principle every single change is very carefully analyzed; all of these are entered into a bug tracking system so there are system tools that are available and every change request goes into the system and there is typically a change each control board that does the impact analysis of this and decides as to which change can go into which release and this change control board as it is called is composed of some developers, some maintainers and some of the management staff who sit down together and figure out how to schedule in the changes in an appropriate way to be released.

So, in practice what happens is some change request have to be fixed urgently; the turnaround time has to be twenty four hours or less in case it is a major bug so the system is crashing and giving a particular input is not an acceptable situation and in such situations you would always almost have to turn around the request very very quickly but some things can be scheduled for a particular release and done in an appropriate manner with requisite amount of testing and so on.

We will not go too much into the change implementation; the implementation procedure is very similar to that of the development procedure so you take a look at the requirements, analyze the requirements, you come up with design changes that has to take place, you do the development, do the testing and so on and all these can be done iteratively just like we showed in this spiral model where very very incremental or small changes are done to the software so as to not disturb the stability of the software through a large change or multiple small changes should not be done together; you do one change, you test it, you do another change test it and so on and so forth.

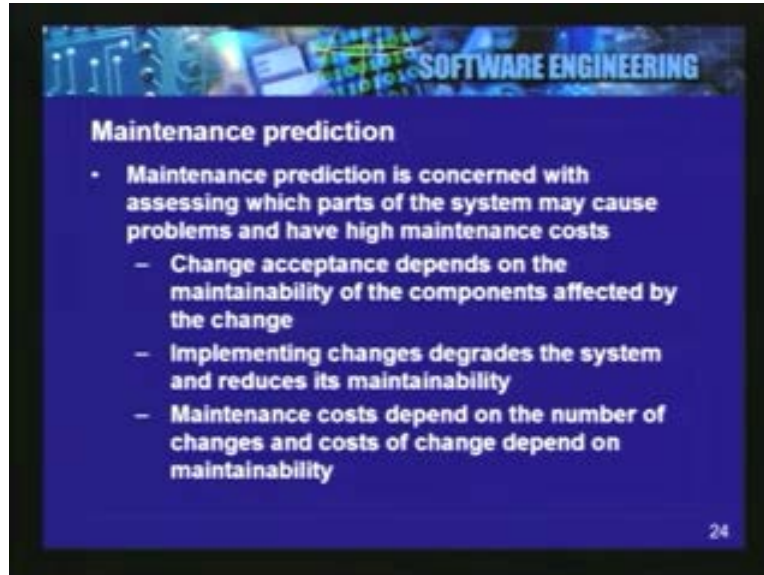
(Refer Slide Time: 48:16 min)



Of course, emergency repair cannot be dealt with in that way; you have to make the changes instantly and it would be changes across one or two more modules it does not have to be that a single module has to be affected. The crash could be for example because it was not doing adequate testing of the input range of parameters that were being given. So, for example, it only accepted integer values but somebody was trying to give it a float value or a floating point value so those are minor things that typically affect only one module but there could be slightly more major things in which the **protocol itself is incapable of the** communication protocol is incapable of handling a kind of data that have been sent across and so on and this may institute a change in multiple modules at the same time so here you would have to analyze the code in the case of emergency repair you directly go into that you do not do a requirement analysis phase but you directly go into the code because it has to be turned around very very quickly.

The next subject that we will look into briefly is that of maintenance prediction and maintenance prediction is basically concerned with trying to come up based on the history of the software that existed so far the history of the user behavior and so on to come up with a prediction of.....

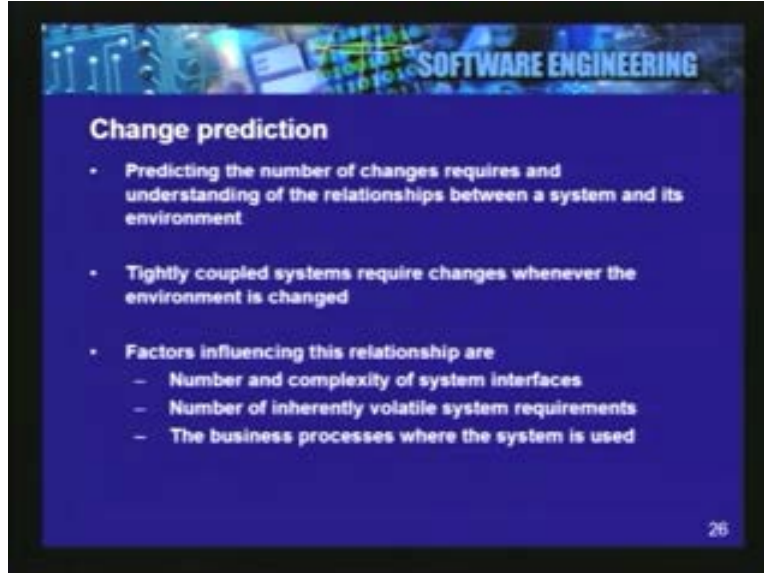
(Refer Slide Time: 49:46 min)



Now what are the areas in which the changes are likely to take place; so maybe we should staff up those area appropriately; what kind of changes are there; mainly bug fixes that are happening there or mainly new feature requests are coming in so what are the kind of changes; so you try to predict that as well and this will give you a good handle on what the lifetime of the software is to be. So, for example, if your maintenance prediction cycle is showing a very steep curve at say one and half years after the software has been put into production then it shows that you have a year and half to develop a new piece of software because essentially you are going to put it out of service at that one and a half year mark so it is a very important field and change prediction also essentially has to do with the notion of what are the kinds of changes that are coming in predicting, the number changes that are required, understanding the relationship of the system's environment and so on.

When there are very very tightly coupled systems so the system is tightly coupled with the operating system for example every OS change would require the change in the system to take place. So those kinds of analysis can be done how tightly can how many system **cause** for example you are using; you have a layer in between the operating system and the software application itself; it insulates you from certain changes in the system **called** interfaces and so on and so forth and change prediction and maintenance prediction is often dependent on **you know what is the** what is the size of the system, what are the modularization of the system like, what are the different interfaces, how many interfaces are there, what is the complexity of the interface and so on and so forth.

(Refer Slide Time: 51:00 min)



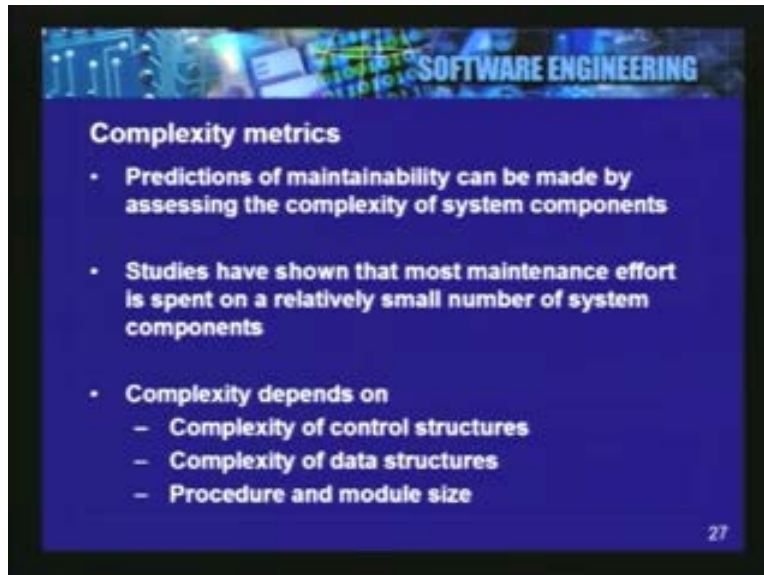
The slide is titled "SOFTWARE ENGINEERING" at the top. Below the title, the main heading is "Change prediction". The content is a bulleted list:

- Predicting the number of changes requires and understanding of the relationships between a system and its environment
- Tightly coupled systems require changes whenever the environment is changed
- Factors influencing this relationship are
 - Number and complexity of system interfaces
 - Number of inherently volatile system requirements
 - The business processes where the system is used

The slide number "26" is in the bottom right corner.

Several different complexity metrics and the predictions can be made looking at the complexity metrics itself.

(Refer Slide Time: 51:38 min)



The slide is titled "SOFTWARE ENGINEERING" at the top. Below the title, the main heading is "Complexity metrics". The content is a bulleted list:

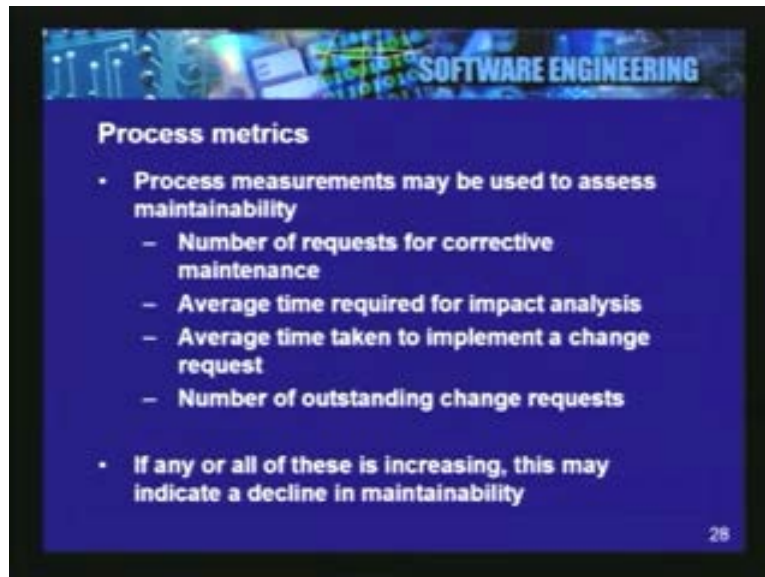
- Predictions of maintainability can be made by assessing the complexity of system components
- Studies have shown that most maintenance effort is spent on a relatively small number of system components
- Complexity depends on
 - Complexity of control structures
 - Complexity of data structures
 - Procedure and module size

The slide number "27" is in the bottom right corner.

Lots of studies have shown that this is usual primarily because of the fact that changes tend to get localized. Typically one development team may not be good as the rest of them, they may not have followed the same practices and the quality of the code that comes out of this development team may not be on par so as a result of which a lot of bugs keep getting formed in the same system and if that system is been maintained by the same team then it is likely that the number of bugs is going to continue to remain high so

these kinds of features also play into prediction itself and certainly the process metrics of something to keep track of; you do not want to lose track of the fact that how many of the historical perspective of whether your change process is working correctly.

(Refer Slide Time: 52:33 min)



So, for example, there is a particular bug that comes up repeatedly then your changed process does not work so that is what it means. It means that somewhere along the path testing was not done; somewhere along the path the impact analysis would not have adequately done and so on and so forth and **if there is** basically it is the average number of changes that are taking place; some of the important metrics are given out on the slide; so number of requirements for corrective maintenance; the average time that is taken for impact analysis; how much time you take to come back and save and how long it will take to fix this so the average time to fix it itself; then is the change reoccurring; does the same change reoccur often; number of outstanding change requests at any given point at time all of these should kind of be dropping off; if any of these times are increasing then it shows that there is a decline in the maintainability of the code so it is getting harder it is getting more expensive to maintain the code and it may not be appropriate to do this at this point in time.

So there are things that you would have to keep a watch out for in order to figure out as to when is the right time to kind of replace software if you will just like you do for household appliance that we talked about before. So in summary what we see is that software maintenance and evolution is a very very important aspect of the overall life cycle of the software, it is not just development that has to be given importance the process of software evolution, the tools that are required, so cm tools may be required, cm processes configuration management processes may be required, the change management process is required; it is important to keep a continuity between the development staff who developed the original system and the people who end up maintaining it; it is important to keep good level of documentation and so on. So all of

these contribute towards the maintainability of the software at the same time you have to be able to predict based on the historical data as to how much of maintenance you are going to be able to withstand on a particular system so that it can evolve to the next generation, it can be replaced with a newer system which can kind of start the whole cycle from the beginning.

In summary again this is a very very important aspect of the lifecycle of the software and is very important to study.