

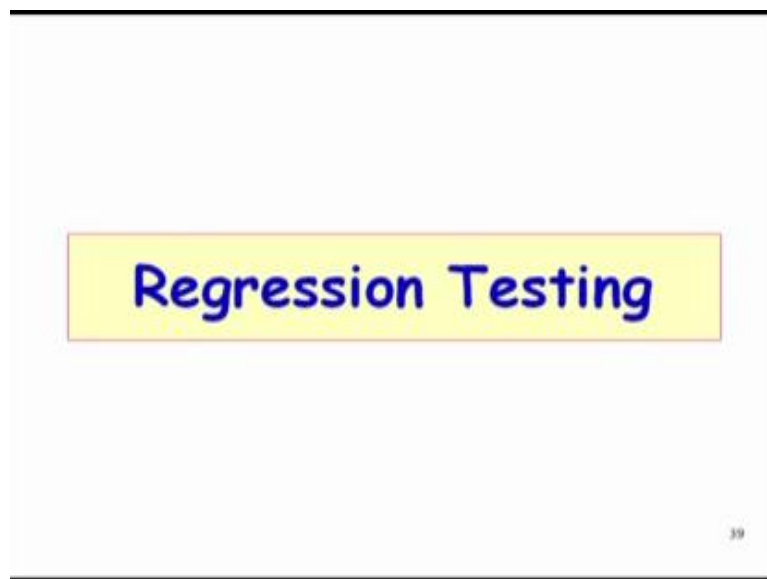
**Software Testing**  
**Prof. Rajib Mall**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 18**  
**Regression Testing**

Welcome to this session. So far, we had looked at some very basic concepts of testing and then, later we looked at black box testing techniques, how to design black box test cases and different types of black box testing.

And later, we looked at white box testing, the coverage based and fault based testing techniques; and we also looked at integration and system testing. Today, we look at regression testing. Regression testing is a different dimension of testing. Actually, it is applicable for all types of testing, unit, integration and system testing. Let us see, what is involved in regression testing.

(Refer Slide Time: 01:08)



(Refer Slide Time: 01:16)

### Recap: How Many Errors are Still Remaining?

- Make a few arbitrary changes to the program:
  - Artificial errors are seeded into the program.
  - Check how many of the seeded errors are detected during testing.

40

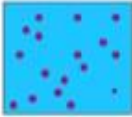
Before we start looking at regression testing, let us recall it little bit, about what we were discussing in the last session. In the last session, if you remember we were looking at, how does a manager estimate the number of errors that are still left in the code, after all the things that he could do, including careful development, adequate testing and so on.

In spite of all that, some bugs will be still left behind and a manager naturally would like to know, how many bugs are there in the software. Is it hundreds? Is it thousands or is it tens of thousands of bugs there? So, we looked at the error seeding technique, as a possible technique to estimate the number of bugs in the software. If you remember, we said that the manager before the testing starts typically the system testing makes several arbitrary changes to the program. The changes may be in the form of changing **an** arithmetic operator.

For example, plus to minus or may be interchanging 2 statements in the code, may be changing the type of a variable from int to float, may be in expression, changing a variable b into some other variable d or something; these are typical programming errors. So, the manager seeds the code with this many bugs and he keeps a note of that, how many bugs he has seeded in the code. And then, the testing starts. The testers do not know what the seeded bugs are, and then as they run the test cases. The failures are detected and then, they are debugged and if the manager finds out that some of the seeded bugs have surfaced have been detected he takes a note of that.

(Refer Slide Time: 03:50)

### Error Seeding



- Let:
  - $N$  be the total number of errors in the system
  - $n$  of these errors be found by testing.
  - $S$  be the total number of seeded errors,
  - $s$  of the seeded errors be found during testing.

41

So, this explains the expression that we are going to derive, to estimate the number of bugs that are remaining in the software. If we imagine this blue rectangle as the program and the red dots as the bugs that have remained in the code after all the careful design, coding, testing, not testing, design and coding; these are the bugs that are remaining. Now, the manager to get an estimate of how many bugs will remain when the software is shipped to the customer, seeds a number of bugs, represented by these violet dots.

Let the red dots, which were the bugs that have remained in the software originally, this be capital  $N$  and the manager seeds  $S$  number of bugs, capital  $S$  number of bugs. Before testing starts, a system testing; and then, as the system testing is taken up, bugs get detected.

(Refer Slide Time: 05:40)

### Error Seeding

- $n/N = s/S$
- $N = S n/s$
- remaining defects:  
$$N - n = n ((S - s)/s)$$

42

And let, small  $s$  number of the seeded bugs get detected and small  $n$  number of original errors in the code be detected. And then, with the fast approximation, we can write  $n / N$  is equal to  $s / S$ , but to write this expression, we have made a implicit assumption. What is the implicit assumption here, in this developing this expression? The implicit assumption is that, the rate at which the original bugs are getting detected is the same as the rate at which the seeded bugs are getting detected; or in other words, the seeded bugs are not too easy type of bug, too obvious bugs, which get detected almost immediately after the testing starts; or these are also neither the type of bugs that are too insidious or too complex, not to be detected by the test and they remain in the code.

So, the implicit assumption we can say is that, the type of the bugs that remain in the code matches with the type of the bugs that are seeded; not only the type of the bugs, but also their frequency. So, if the arithmetic expression interchange is a type of bug that remains, then and their occurrence is about 5 percent then, the seeded bugs would also be arithmetic expression interchange be 5 percent.


And, not only that, the location at which the, in the code, at which the error occurs, the specific components, that is also important; because, some components are used very heavily and some components are used very less. So, not only the type of the bugs, the frequency, but their location in the code should approximately match, for this expression to hold. Now, let us assume that, somehow, the manager has a way to know the type of bugs, their frequency, and their location and so on and he seeds the bug accordingly.

We can simplify this expression in that case and find the bugs that were remaining before testing; capital N is capital  $S * n / s$ ; but then, small n of them have got detected during testing. So, the bugs that remain after testing is  $N - n$ ; or we can substitute this in this expression and write  $n * (S - s) / s$ ; this is the number of bugs that would remain in the code.

(Refer Slide Time: 09:22)

### Quiz 1: Solution

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors=  
 **$50 (100-90)/90 = 6$**




Now, let us work out a small quiz, based on this concept. Assume that, a manager, before system testing, introduced 100 bugs into the code, unknown to the testers and developers. He just went there and made changes and introduced 100 bugs. And then, the system testing started and during system testing, 90 of the seeded bugs were found out; but, in the system testing process, 50 other bugs were also found, which were not seeded by the manager. So, find an error estimate for the code. So, that means, that how many of the original bugs would still remain in the code after system testing.

So, how do we go about doing this? We can use the expression that we derived,  $n / N = s / S$ ; and from there, and we derived,  $n * (S - s) / s$ . Then, the problem becomes very simple; we need to just substitute the values appropriately into the expression. So, that is  $50 * (100 - 90) / 90$ , which is approximately 6. So, for this case, approximately 6 errors are estimated to be there in the code, assuming that, the number of sorry, the type frequency and the location of the seeded bugs roughly match with that of the original bugs.

(Refer Slide Time: 11:26)

### Error Seeding: An Issue

- The kinds of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.



But, as we have been saying that, the implicit assumption in the expression  $n / N$  is equal to  $s / S$  is that, the type, the frequency and location of the seeded bugs, roughly match to that of the bugs that are existing in the code. But then, the manager does not know what are the unknown bugs there. Otherwise, no testing, nothing will be necessary; he does not know that. So, how does he go about seeding bugs which roughly match with the type, frequency and location of the bugs?

Normally, what is done is, a working solution, where the manager looks at the data, the error data for related software; similar software; he finds out, what were the types of bug that were remaining, were reported by the customer; what was their frequency and what were their locations. and the manager seeds according to that. So, this is the workable solution to making that expression valid. There is one more quiz here.

(Refer Slide Time: 12:15)

### Error Seeding: An Issue

- The kinds of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.
- **Working solution:**
  - **Estimate by analyzing historical data from similar projects**

45

(Refer Slide Time: 12:59)

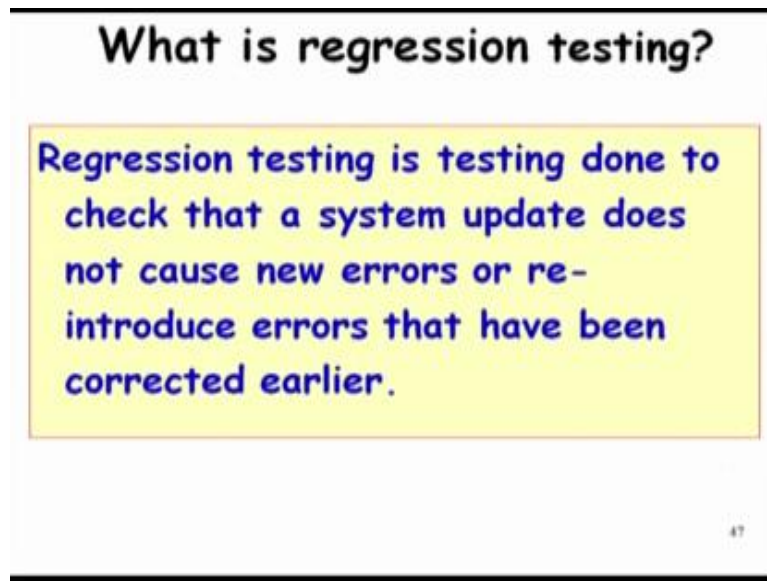
### Quiz 2

- Before system testing 100 errors were seeded by the manager.
- During system testing 60 of these were detected.
- 150 other errors were also detected
- **How many unknown errors are expected to remain after system testing?**

46

Before the system's testing started, the manager introduced 100 bugs into the software. Now, during system testing, 60 of these seeded bugs were detected. 150 other bugs were also detected. So, what would be the manager's estimate of the number of bugs that are still remaining in the software? Please work out this problem. If you have understood the expression then you should be able to do it; a simple problem, substitution the appropriate numbers into the expression.

(Refer Slide Time: 13:54)

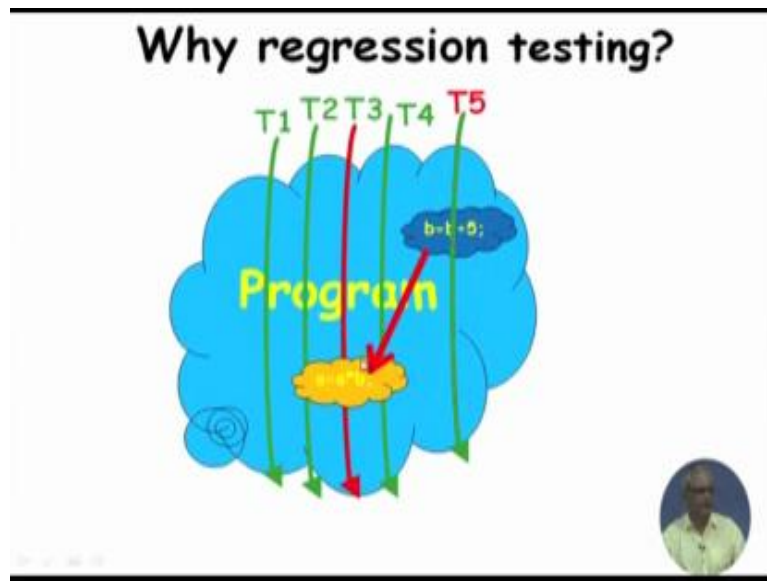


Now, let us look at regression testing. The regression testing is done to check that a system update does not cause new errors or reintroduce, errors that have been corrected earlier. So, this is the definition of regression testing. What it really means is that, when you have software, you make numerous changes to it; that is the normal thing; that has the development undergoes lot of changes happen to the software. For example, bugs may get detected and these are fixed; that is a change; or may be, some new feature was added or may be, some feature was modified and so on.

So, each time there is a change to the software, there is a large part of the software that has not changed. So, is it ok that if we just test the changed part of the software and do not test the unchanged part? No, we cannot do that because any change small change will induce bugs in the unchanged part of the software. Why is that, because of the data sharing and data dependency the changes may cause some variables to assume different values, which are used by the unchanged part of the code. And then, those start to fail; the unchanged part of the code may be, had passed all the tests, was found to be working correctly,

But then, a small change somewhere just two lines of change that may show up as many failures in the other part of the software, because it changed a variable value and that value was used by the other parts; and can show up as failures. So, that is the idea here that we need to carry out regression testing to the unchanged part of the code, when anything changes. **Or,** in other words, we have to rerun the test cases which had already passed for some part of the code and even if that part has not changed, we have to rerun those test cases again and that is called as regression testing.

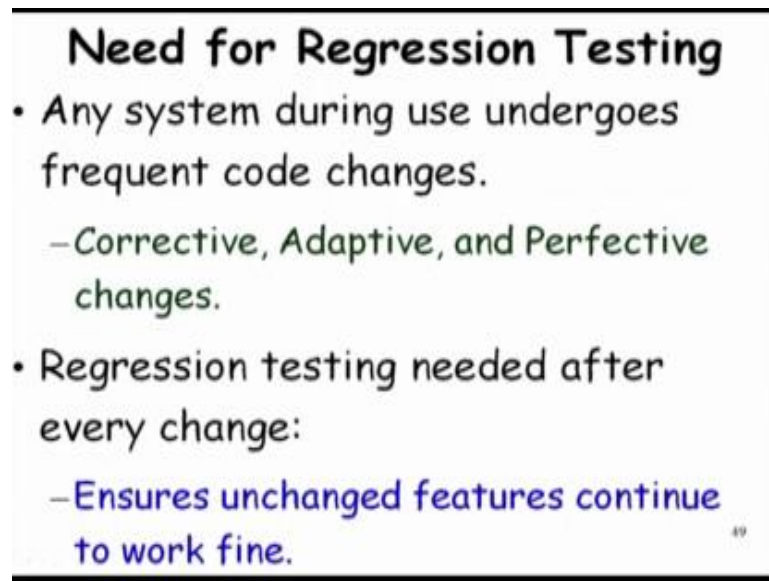
(Refer Slide Time: 16:39)



Now, let us try to understand the idea behind regression testing with a small animated example. Let us assume that, this blue cloud is a program and we tested the program with a number of test cases; and they all passed. This is just an example to illustrate what exactly we mean by regression testing. So, T 1 passed; we have shown it by green; T 2 also passed, shown it by green; T 3 passed; test T 3, that is green and test T 4 also passed, that is green; but, when we ran T 5, it failed, shown by a red. So, T 5 was observed to be a failure and once a test case fails, the developers debug the code, try to locate what is the source of the problem and then, modify the code to correct the error.

So, in this case also, the developer got to work and then found that they could eliminate the problem by introducing this statement  $b = b + 5$ . and as they changed the code, the test case which was failing, actually passed. So, T 5 is also green; the test case passed. But, when we run the other test cases, the parts that are not changed actually, the test, other test cases, they run; they execute statements that have not changed. But still, T 3 failed; why? Because, it was using the variable b;  $a = a * b$ . So, T 3 was using the variable b.

And now, we have changed the value b and therefore, T 3 which was working correctly, has started fail, even though none of the code of T 3 was changed. So, then, the meaning is that, the fix was not proper. It could, should have been something else, like  $b = b * a$  or something here; and  $b = b + 5$  makes this test case pass, but the regression test cases fail.



**Need for Regression Testing**

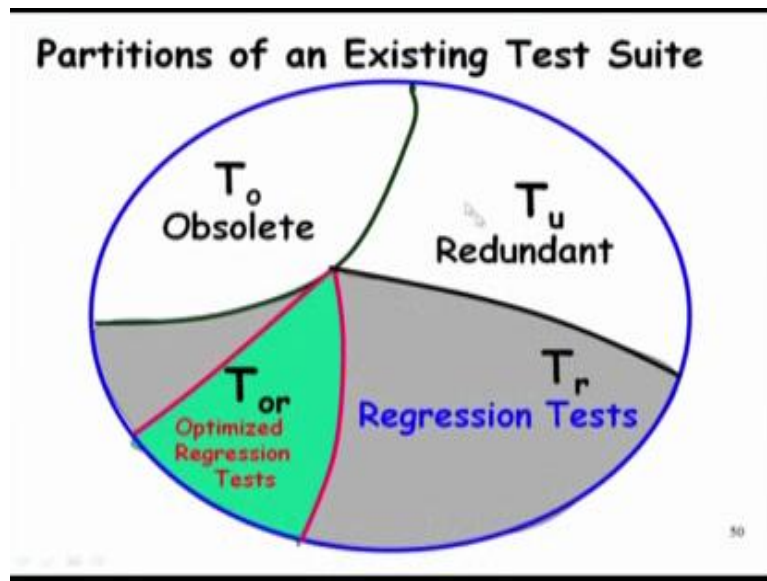
- Any system during use undergoes frequent code changes.
  - Corrective, Adaptive, and Perfective changes.
- Regression testing needed after every change:
  - Ensures unchanged features continue to work fine.

49

So, throughout the life cycle of software, lots of changes occur to the code. Roughly, these changes can be classified into corrective changes, adaptive changes and perfective changes. Corrective changes are those changes, which are made to the code to fix some bugs. Adaptive changes are made to the code to make it to work with some other libraries or some other software or hardware, to adapt the software to work with something else. and perfective changes are those, where we add new features to make the software more perfect or we improve the performance of the software by changing some parts of the code and so on.

So, these are 3 main types of changes; and each time we change the code, not only we have to test the changed part of the code as usual, but also we need to carry out regression testing on the unchanged part of the software. The regression testing is needed after every change on the unchanged part of the software, to ensure that the unchanged features continue to work fine.

(Refer Slide Time: 21:26)



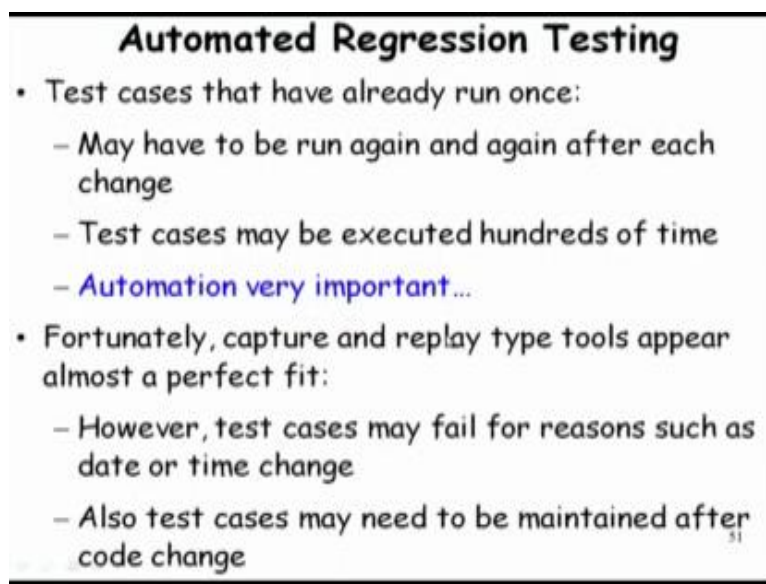
If we had a set of test cases which were used so far and let us say, the set is  $T$ , then, we can roughly partition this test cases into obsolete, redundant and the regression tests. Some of the original test cases become obsolete; they cannot be used anymore. They are invalid. They have to be thrown. What is the reason? Because, these test cases were executing the changed part of the code and possibly, the input values they were giving was inadequate. They were inputting 2 integer values, but, what is needed actually, 2 integer value and a float value and so on.

So, some of the test cases cannot be used anymore, after the change. These are called as the obsolete or invalid test cases. Some test cases are redundant. As we change some part of the code, then, some part of the unchanged code gets affected and they have to be regression tested. But, some part is not affected at all. There is a no sharing of variable, no dependency with the variables that are set in the changed part and therefore, we can be 100 percent sure that, these are truly independent part of the code with respect to the changed part and we need not test them. So, those are called as the redundant test cases.

Even though they are valid test cases, we do not need to run them, because they have zero possibility of detecting any bugs. They are only executing the independent part of the software. And then, the remaining part is the regression test cases. The regression test cases are those, which **execute** those parts of the unchanged program, unchanged part of the program, which have some dependency with the changed part; say, are some variable dependence some variable values that are assigned in the changed part and so on.

So, out of these regression tests, which may be too many, may be thousands of regression tests for large software can be identified and may be running regression test for each **change**, thousand numbers may be difficult. We can find out an optimum regression test, optimized regression test, out of the regression tests, which are almost as good as running the full set of regression tests. But during this session it will be out of scope to identify, how to get these optimized regression test cases, what are the algorithms, techniques, etcetera, to determine the optimized regression test cases, but we will look at how to identify the regression tests.

(Refer Slide Time: 25:01)



**Automated Regression Testing**

- Test cases that have already run once:
  - May have to be run again and again after each change
  - Test cases may be executed hundreds of time
  - Automation very important...
- Fortunately, capture and replay type tools appear almost a perfect fit:
  - However, test cases may fail for reasons such as date or time change
  - Also test cases may need to be maintained after code change

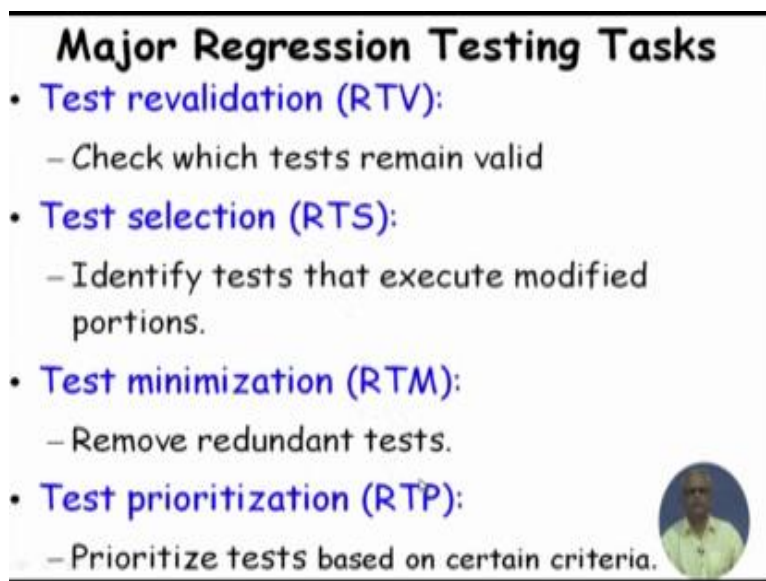
But, one thing that needs to be kept in mind that during regression test cases, the test cases which were run once they are run again and again after each change to the code. So, it may so happen that, a test case which was originally executed once, designed and executed, may have to be executed hundreds or thousands of times. and when we run a regression test hundreds or thousands of time during the lifetime of the software, after each change occurs once then, it becomes monotonous error prone. And therefore automation is very important.

Fortunately, the capture and replay type of tools that we had discussed, appear to be a perfect fit here. Because, the capture and replay type of automation tools that capture the input, when first time the test is run and they have also captured the output that was considered to be a pass. And then, whenever needed we can just press the switch and it will, the tool will rerun all the test cases which were recorded and check the results with respect to the recorded result and then, flag if any failed test cases are there. Appears very good, but then, there are some catches.

Let us be aware of the catches and how these are overcome. One of the catches is that, in the capture and replay type of tool, maintenance of test cases is a big problem in the sense that let us say, a test case which was run now with the changed code, it needs additional parameters to be given. In this case, we have to discard the entire test case and rerecord it. There is no way we can maintain the test case. And also, another thing is that, the test cases fail for some reasons like, it may be comparing with date.

So, the test case was run on some date and the date value was used for that date and it passed on that date for that input value but, may be on some other date or time; it may change, because it was just recorded as it is. If we have some scripting based testing, we might overcome this problem of maintaining; you can change the script little bit and that will maintain it. Similarly, we can use the date wherever it is needed and so on.

(Refer Slide Time: 28:17)



**Major Regression Testing Tasks**

- **Test revalidation (RTV):**
  - Check which tests remain valid
- **Test selection (RTS):**
  - Identify tests that execute modified portions.
- **Test minimization (RTM):**
  - Remove redundant tests.
- **Test prioritization (RTP):**
  - Prioritize tests based on certain criteria.

So, what are the different regression testing tasks? One is test validation, test selection, test minimization and test prioritization. So, we are just in the end of this session. We will stop here and we will continue from this point, in the next session.