

**Lecture – 16**  
**Integration Testing**

Welcome to this session. In the last session, we were discussing integration testing, and we had discussed about the basic error target in integration testing. And said that the error targets for integration testing are the interface errors, where one function calls another function, and there is a problem in the calling in the sense that the parameters are not proper. Even though as a unit the functions worked correctly, during calling there are problem. And integration testing, targets to detect such problems.

And then we had looked at the different types of integration testing. We said that the simplest integration testing is big bang testing. In big bang testing, we just integrate all the modules in all the modules or units in just one-step, but then we said that for a non-trivial program, nobody does a big bang testing, it would be too expensive to fix bugs in a big bang testing.

So, the testing process will become extremely expensive, but then what are the other alternatives. Alternatives are the bottom up testing, top down testing and a mixed or sandwich testing.

(Refer Slide Time: 02:53)

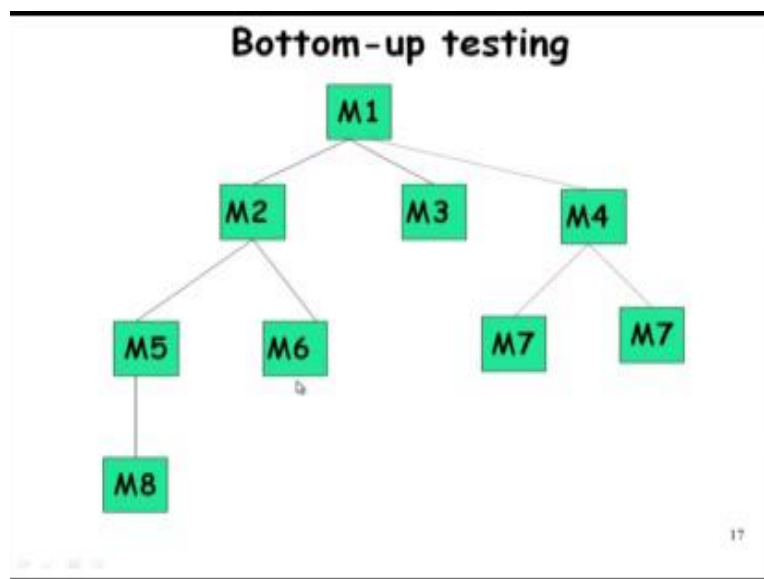
### **Bottom-up Integration Testing**

- Integrate and test the bottom level modules first.
- Disadvantages of bottom-up testing:
  - Drivers have to be written.
  - Test engineers cannot observe system level functions from a partly integrated system.

Let us look at the bottom-up testing. In bottom-up testing, we take up the unit or the module at the bottom most layer of the design. And then integrate it with one module, which is at the same level or there are no modules available in the same level, we integrate with the module in the higher level and so on until all the modules are integrated. So at a time we integrate only one module. And we need to be aware what are the disadvantages of this bottom-up testing, one is that we need to have drivers written as we do integration steps, since we are testing the bottom level modules we need to have some software written which will call these modules. So, for performing integration testing, the tester has to write these drivers.

And the second thing is that the tester cannot really run the system level test cases that **is** huge cases it cannot really run meaningful test cases, only some very simple test cases. For example, whether data input from a user interface is correctly received or whether the data written by to a user interface is correctly shown and so on that kind of very simple things need to be tested.

(Refer Slide Time: 03:57)

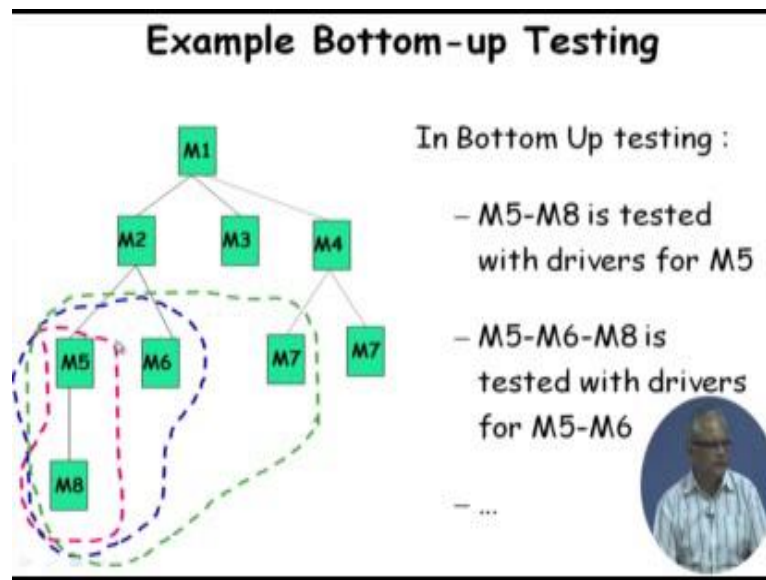


The way the bottom-up testing works can be seen from this diagram. We start with the bottom most module; and if there are no other module at that layer, we take the next level integrate, and then we integrate with another module in the same layer, three modules together.

And then after they work successfully they have been tested and any bugs fixed, then we take one more module in the same layer, again integrate and test it, fix any bugs that are present. And then

take one more and then the higher level and so on until all the modules are integrated. So, this is the essence of bottom-up testing, but then this is a incremental bottom-up testing. And if some of the modules are really trivial then we might even integrate two modules in one-step.

(Refer Slide Time: 05:17)




This is an example of bottom-up testing. So, first we integrate M 5 with M 8, and we write a driver for M 5, because M 8 is being called by M 5, but who would call M 5. Because M 2 was calling M 5, but we **have** not integrated M 2 and therefore, we need to write driver software which will call M 5, and through this driver software, we will test these two the M 5 and M 8. Now we integrate once M 5 is successfully integrated with M 8, we take up M 6 and integrate with M 5 and M 8. And then we need drivers for both M 5 and M 6, because some software must call M 5 and M 6, we have not yet integrated M 2.

And therefore, we need two drivers here. And then we integrate one more, and now we need three drivers. And then we check whether these are working correctly; if not, we fix the bugs.

(Refer Slide Time: 06:42)

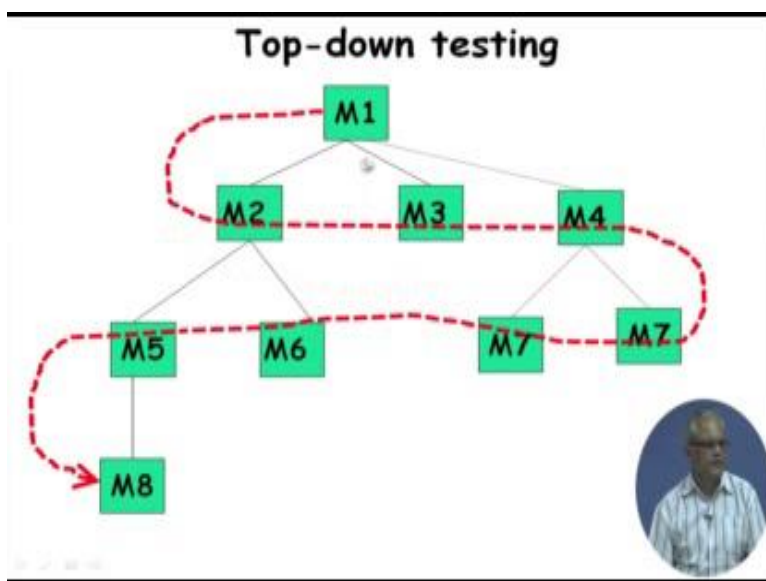
### Top-down Integration Testing

- Top-down integration testing starts with the top module:
  - and one or two subordinate modules
- After the top-level 'skeleton' has been tested:
  - Immediate subordinate modules of 'skeleton' are combined with it and



Now, let us look at the top-down integration testing. So here as the name implies start with the top most module. And then we take one lower level module or a subordinate module and integrate them. And if the subordinate module is very simple, almost trivial in module then we might even take two modules and one-step. And then once we have this top level module integration tested, we add the immediate subordinate module and so on.

(Refer Slide Time: 07:25)

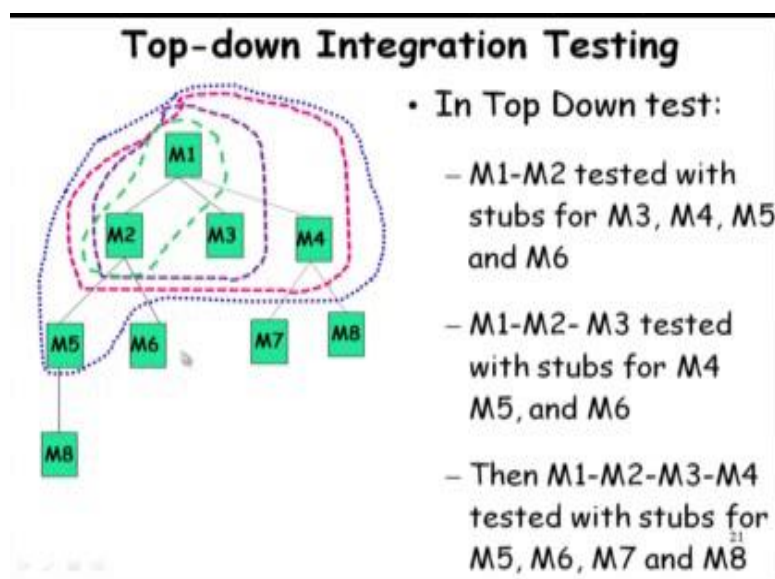


If we want to show a diagrammatically that the order in which the modules are integrated, we start with the top most and then integrate a subordinate module M 1 with M 2, and then integrate M 1, M 2 and M 3 together. But just remember that here we will have to write stubs. Just like in the bottom up testing, we need drivers, in top-down testing, we need stubs. Because observe here M 2 is calling M 5 and M 6, but we were not integrated M 5 and M 6 we need to write small toy software which will act like M 5 and M 6 very simplistic software not really providing all the functionality of M 5 and M 6.

But at least some dummy functionality may be through a table look up we have stored what are the results that M 5 computes the form of a table, and when that parameter is given we just look up the table and **return** that value that is the work of the stub here. So, when we integrate M 1 with M 2, we need two stubs.

And then integrate M 1 with M 2 and M 3, and then we integrate M 1, M 2, M 3 with M 4 and test and here see we need four stubs. And then once we have tested it, and made it to work correctly, then we integrate with M 7 here this module here, and then we would need actually three stubs because this module is there, and therefore, we do not need a stub for this we need a stub for only this one, this one and this one. So, we need three stubs. And then next we take up this one this module here, and therefore, we would need only two stubs and so on until we have the full set of modules integrated.

(Refer Slide Time: 09:55)



So, first we integrate M 1 with M 2. So we need stub for with M 1 and M 2, we need stub for M 3, M 4, M 5 and M 6. So, in this situation when we integrate M 1 with M 2, we need four stubs, because M 1 also needs to call M 3 and M 4, and M 2 needs to call M 5 and M 6. So, we need four stubs. And then the next step we integrate M 1, M 2, M 3 and we need three stubs M 4 M 5 and M 6. In the next step, we integrate M 1, M 2, M 3 and M 4, and here we need four stubs, stubs for M 5, M 6, M 7 and M 8 and so on. And then we next take up with this module and then the next this one and so on. So this is the top-down integration testing.

One advantage of the top-down integration testing is that the test cases are actually meaningful **use** cases, because we are testing ( ) the top-level functionality here, and also we reuse the same test cases as we integrate more and more module. So, earlier remember that we are testing the bottom level modules we are needing different types of test cases to check whether these are working, but here in the same test case we just keep on running with while adding new modules and we check whether those are working, so that is advantageous for top-down testing.

(Refer Slide Time: 11:56)

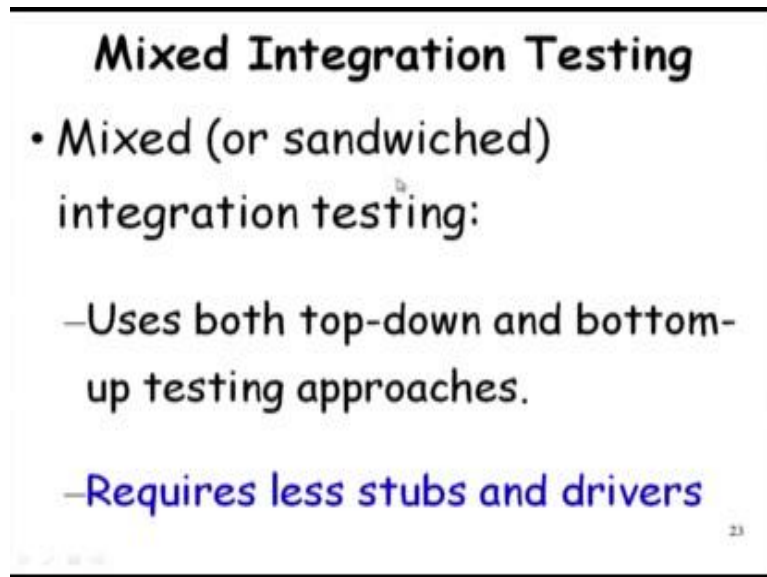
### Top-Down Integration

- **Advantages of top down integration testing:**
  - Test cases designed to test the integration of some module are reused after integrating other modules at lower level.
  - Advantageous if major flaws occur toward the top of the program.
- **Disadvantages of top down integration testing:**
  - It may not be possible to observe meaningful system functions because of an absence of lower level modules which handle I/O.
  - Stub design become increasingly difficult when stubs lie far away from the level of the module.<sup>22</sup>

So, the test cases are reused after integration. And also top-down integration testing is advantageous if the bugs are in the top-level module, because if the bugs are more at the bottom level module, then we would have by the time integrated many modules together and it would be difficult to localize the error.

Whereas if the bugs are at the low level modules, then a bottom-up integration testing would be possibly more advantageous. The disadvantage is that since the low-level modules are I/O and we do not integrate it in the beginning, so observing meaningful system functions may not be possible to start with and also stop design becomes a problem. Because if for a non-trivial program, when we are testing the top-level modules, writing stubs for those **way** down would be difficult.

(Refer Slide Time: 13:28)



**Mixed Integration Testing**

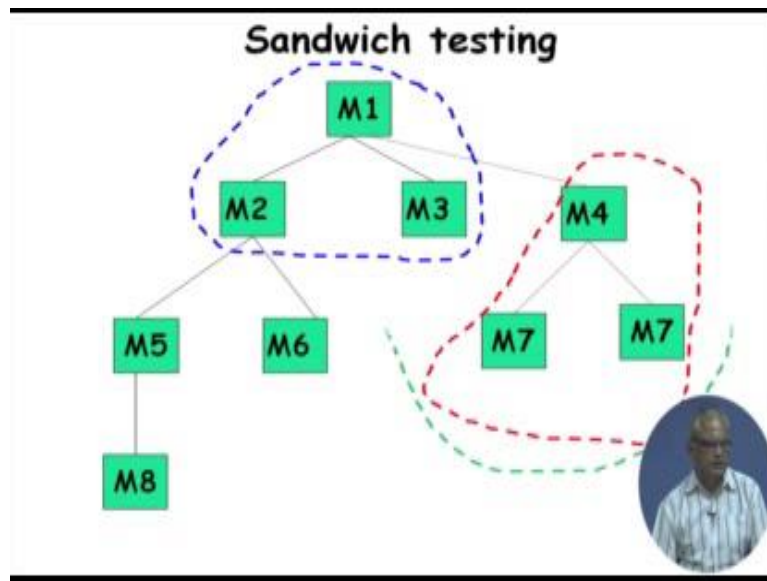
- Mixed (or sandwiched) integration testing:
  - Uses both top-down and bottom-up testing approaches.
  - Requires less stubs and drivers

23

The other alternative is a mixed integration testing also called as a sandwiched integration testing. So, here it is more flexible in the sense that we have the liberty to use both top-down and bottom-up approaches as it suites us. And we would typically write, we will typically integrate such that we need less number of stubs and drivers to be written and therefore, mixed integration testing is quite popular.

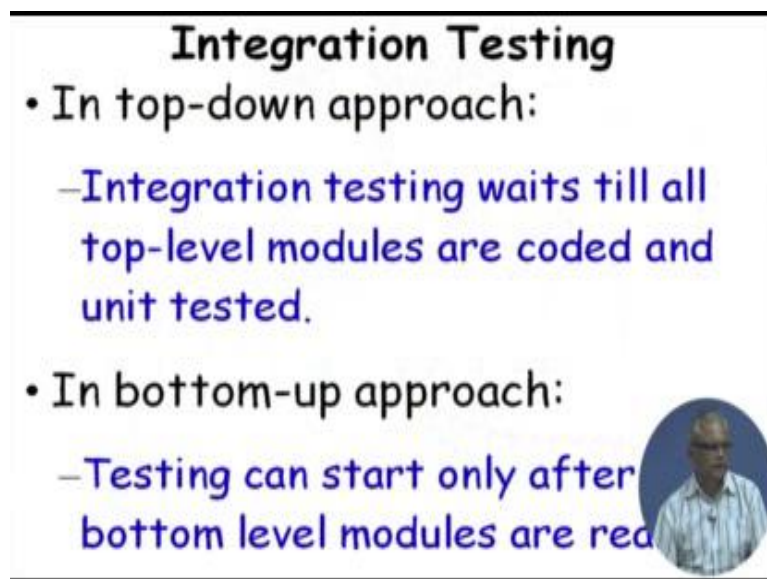


(Refer Slide Time: 14:00)



Let us look at how is this **sandwiched** testing or mixed integration testing would work. So, we might integrate these three and then we might integrate **these** three, so some at the bottom-level some at the top-level and then we integrate them together. So, the number of stubs and drivers require to be written can be reduced substantially through a **sandwiched** integration testing technique.

(Refer Slide Time: 14:37)



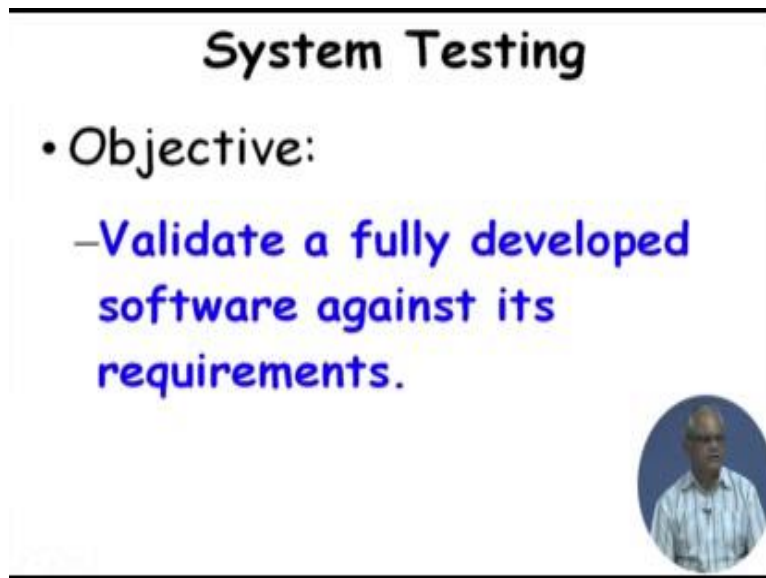
So, another point that we need to observe is that if we are doing a top-down approach, we cannot really start testing integration testing until the top most module is ready. So, in a purely top-down



approach, our testing has to wait until the top-level modules are ready, even though the bottom levels are available we can do a pure top-down testing.


And similarly, in a bottom-up approach, our integration testing cannot start until all the bottom-level modules are ready, so that is another advantage for mixed integration or **sandwiched** integration testing that even if some of the modules are available you can start testing.

(Refer Slide Time: 15:39)



**System Testing**

- Objective:
  - **Validate a fully developed software against its requirements.**




Now, let us look at system testing. We had at the beginning said that system testing is a validation technique; the other types of testing are actually verification techniques. So, here in system testing, we validate a fully integrated, a fully developed system against **its** requirements. So, the test cases are written by looking at the requirements document and then they are tested on the fully working software to validate whether the software that has been developed is according to what was required for this software.

(Refer Slide Time: 16:33)

## System Testing

- There are three main types of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing




There are actually three main types of system testing; one is alpha testing, the other is beta testing, and the third is acceptance testing. All three are system testing, but the names of the testing is depending on who does the testing, we call it as alpha testing, beta testing or acceptance testing. If the system testing is done by the developing organization itself, the development team or the developing organization, we call it as the alpha testing.

And if the testing is done by a friendly organization, who are just asked to test and report if there are any problems that is called as the beta testing. Whereas acceptance testing is the system testing, where the customer does the system testing to decide whether to accept or reject the software. So, these are all these three are system testing and depending on who carries out the testing, we call it as alpha testing, beta testing or acceptance testing.

(Refer Slide Time: 18:00)

## Alpha Testing

- System testing carried out by the test team within the developing organization.
  - Test cases are designed based on the SRS document




In alpha testing, the testing is carried out within the developing organization, but then the test cases are largely designed based on the SRS document, because this is a system testing.

(Refer Slide Time: 18:13)

## Beta Testing

- System testing performed by a select group of friendly customers.




In beta testing, it is tested by a select group of friendly customers; so before the software is delivered to the customers it is given to a set of friendly organizations just to test whether there are any problems.

(Refer Slide Time: 18:36)

---

## Acceptance Testing

- System testing performed by the customer himself:
  - To determine whether the system should be accepted or rejected.



---

And then the final is the acceptance testing, where the customer is given the software and the customer carries out the system testing and decide whether to accept the software or reject it. And remember that in all the three types of system testing alpha testing, beta testing and acceptance testing, the test cases are designed based on the requirements document. Normally, do not have to look through the source code to carry out system testing.

(Refer Slide Time: 19:17)

---

## System Testing

- Two types:
  - **Functionality:** Black-box test cases are designed to test the system functionality against the requirements.
  - **Performance:** Tests are designed against the non-functional requirements documented in the SRS document.

31

---

Now, the next question that comes is, if we have to do the system testing, what sort of testing we need to do. There are two major categories of tests; one is called as the functionality test and other is called as the performance test. So all these tests are both functionality and performance tests are designed based on the requirements document. And the functionality tests are done based on the functional requirements, so we find out what are the high-level functions there or the huge cases and what are the scenarios and then we do the functionality test, which are basically black-box test cases, because we do not have access to the source code during system testing.

And we had seen several black-box test techniques – equivalence, partitioning, boundary value, robustness testing and so on. So, these are the different functionality tests that are done during system testing. And once the functionality test does the software successfully passes then we say that the software is functionally correct. And then we need to carry out the performance tests.

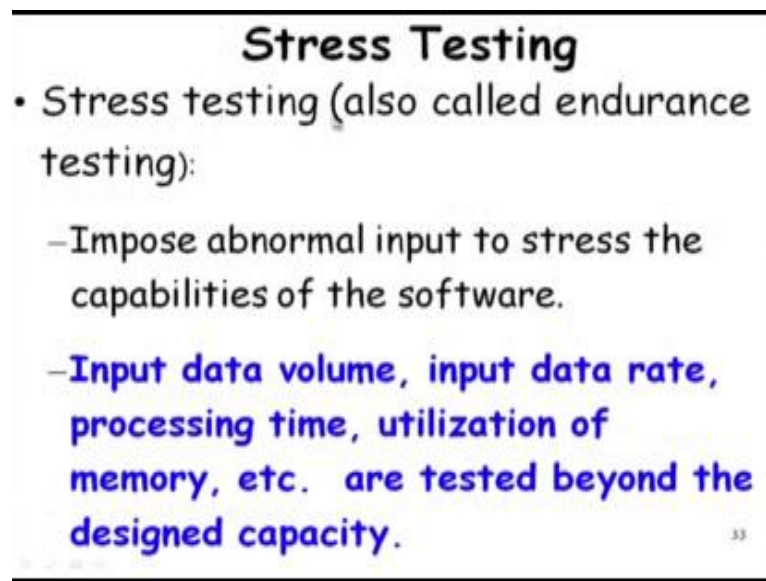
The performance tests are designed based on the non-functional requirements in a requirements document. So as you might already know that in a requirements document, there are two important sections; one dealing with various functional requirements, and the other dealing with the non-functional requirements. The non-functional requirements are those which are not really functions, but then the document items such as what should be the reliability, what should happen when there is a power failure, what should be the through put how many data items can be input and results can be observed, what is the response time and so on. So, those are the performance aspects.

(Refer Slide Time: 21:57)

Performance Tests	
• Stress tests	• Load test
• Volume tests	• Recovery tests
• Configuration tests	• Maintenance tests
• Compatibility tests	• Documentation tests
• Security tests	• Usability tests
	• Environmental tests

So, depending on the non-functional requirements that are there in the SRS document, we can have a large number of performance test cases. We can have stress tests, volume tests, configuration tests, compatibility tests, security tests, load test, recovery tests, maintenance tests, documentation tests, usability tests and environmental tests. Now let us see what are these different types of performance tests that are performed. What do you mean by stress test, how is it different from a load test, how is it different from a volume test, what do we do during a recovery test and so on.

(Refer Slide Time: 22:51)



**Stress Testing**

- Stress testing (also called endurance testing):
  - Impose abnormal input to stress the capabilities of the software.
  - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

33

The stress testing is also called as endurance testing. So, here we test the software with situations that are not normally specified in the SRS document. If the SRS document specifies that we could the software could be used by 5 users satisfactorily, we check what happens when there are 6 users, does the software crash or does it gracefully degrade. If it gracefully degrades with 6 users then it is ok, it passes the stress test; but if it crashes then it is not correct.

Similarly, if the job size is specified something, and if we give slightly larger job size, will the software perform very discontinuously, it will almost hang or will it gracefully handle this. Utilization of memory, if we give a very memory intensive task, what happens, does it suddenly hang, does it crash or does it gracefully degrade performance little bit. Remember that here we give, we create conditions in these testing which is beyond what is specified in the SRS document.

(Refer Slide Time: 24:30)

### **Stress Testing**

- Stress testing usually involves an element of time or size,
  - Such as the number of records transferred per unit time,
  - The maximum number of users active at any time, input data size, etc.
- Therefore stress testing may not be applicable to many types of systems. <sup>34</sup>

So, here most of the tests involve an element of time or size, how fast we give inputs or what is the size of the input, what is the number of records transferred per unit time, maximum number of users active at any time, input data size etcetera. And if our non-functional requirement specification does not tell anything about this then of course, the stress testing would not be applicable. If it does not tell about how many users, it should support, what is the rate at which the data should be transferred, what should be the job size, data size and so on, then stress testing would not be applicable.

(Refer Slide Time: 25:24)

### **Stress Testing Examples**

- If an operating system is supposed to support 15 multiprogrammed jobs,
  - The system is stressed by attempting to run 15 or more jobs simultaneously.
- A real-time system might be tested
  - To determine the effect of simultaneous arrival of several high-priority interrupts.

<sup>35</sup>



So, this is an example of ( ) stress testing. So, let say an operating system can handle at most 15 multiprogrammed jobs. And now if 16 jobs are given, what would happen to the operating system, would it just hang, or would it say that 16 jobs are not possible to handle, or would it just **queue** up 1 job and slowly degraded performance it will handle the 16 job, so that is what we observe here. And for a real-time system, we check what is the performance, time performance.

**So,** if there are high priority interrupts, simultaneous high priority interrupts occurring what happens, is the deadline missed. Normally, a large number of high priority jobs may not arrive; the SRS document may say that only 3 arrive, but what happens if 4 arrive, would the system gracefully handle that or would it crash or would it behave abnormally, so that is what we check here.

(Refer Slide Time: 26:55)

### Load testing

- Load testing
  - Determines whether the performance of the system under different loads acceptable?
  - **Example:** For a web-based application, what is the performance of the system under some specified hits?
- Tool:
  - JMeter:  
<http://jakarta.apache.org/jmeter/>

36

Another type of performance testing is the load testing. In load testing, we check whether the performance of the system under different specified loads is acceptable so that is one point in which the stress testing and load testing differ is that. In load testing, we check the system performance under specified conditions that have been specified in the requirements document. But in stress testing, we test the software with conditions that are not specified in the requirements document or those which beyond those which have been specified in the requirement document, we test the software that is the stress testing.

In load testing, we just check whether a different specified load the software performs satisfactorily.

For example, in a web based application, we might check if the requirements document says that the system should have a response time of less than one second, when 100 simultaneous clicks or hits occur on the website. So, in load testing, we will do that actually. We will have 100 simultaneous hits generated on the website or may be 90 hits generated and we check what how does the system perform. There are several tools available for doing load testing; one very popular tool open source tool is the J meter, which can be easily installed and tested available from the apache website.

We will stop at this point, and continue in the next session.

Thank you.