Week 2 Total time of completion: 6.5 hr

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Start Time | End Time | Total Time | Particulars |
| 8/1/2020 | 11:14AM | 11:44AM | 30 | Week 2 video 1 and 2 |
| 12/01/2020 | 1:03 AM | 2:00 AM | 60 | Design |
| 13/01/2020 | 11:00 PM | 12:30 AM | 90 | Design completed. Implementation started |
| 15/01/2020 | 10:30PM | 11:00PM | 30 | Implementation continued. |
| 17/01/2020 | 10:50AM | 11:30PM | 30 | Implementation completed. |
| 17/01/2020 | 10:30PM | 1:00AM |  | Testing and verification complete. |

**Software Design Introduction:**

Software design is what we call the deliverable, design, the noun, and what we call the process to make that design.

Design, the verb, is the creative process of transforming the problem into a solution. In our case, transforming a requirement specification into a detailed description of the software that's code-ready.

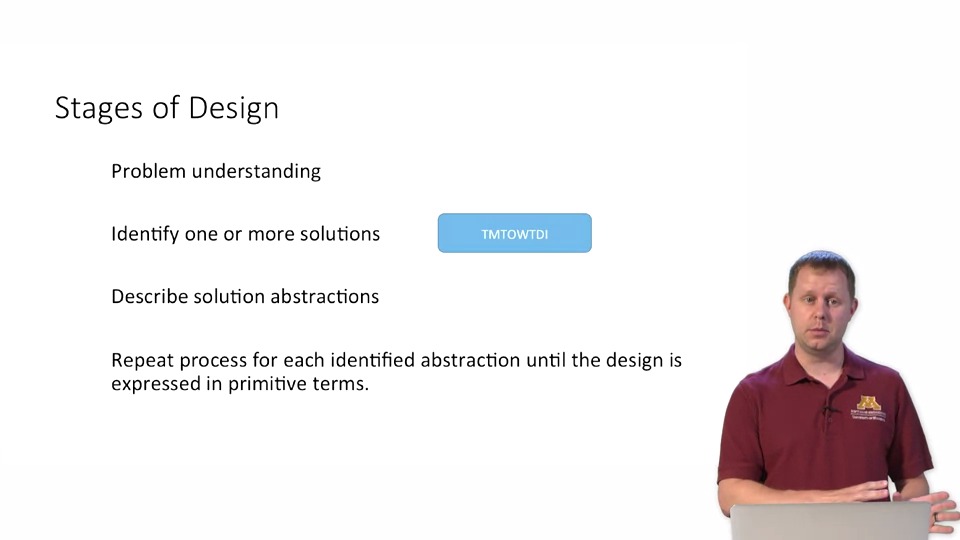
Architecture vs Design:

Architecture:

Large scale decision like

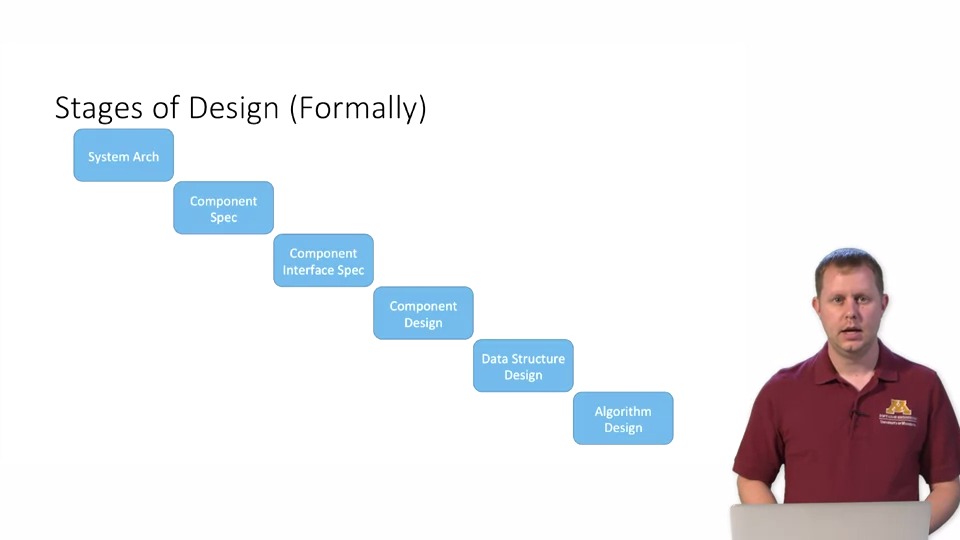
1. Buy or build software
2. Security
3. Appointing Resources
4. Fund related
5. Cost

Stages of Design:



TMTOWTDI: There’s more than one way to do it. There is almost always another way to reach the same singular goal, so consider multiple alternatives before deciding definitively which one to pursue.

Formal Stages of Design:



Architecture: Separate behaviour responsibility into components and, determine how those components will interact through interfaces.

Design: Design individual components in isolation.

**Software Design Modularity:**

Aspects of Modularity:

1. Coupling:

Defines how well does a module work together.

1. Cohesion:

Defines how well a module meets a single well defined goal.

1. Information Hiding
2. Information hiding describes our ability to abstract away information and knowledge in a way that allows us to complete complex work in parallel without having to know all the implementation details concerning how the task will be completed eventually.
3. Basically we know, what a module is doing and not how.
4. Data Encapsulation
5. data encapsulation refers to the idea that we can contain constructs and concepts within a module, allowing us to much more easily understand and manipulate the concept when we're looking at it in relative isolation.
6. It ensures protecting data from unauthorised access and maintaining integrity.
7. Only developer can modify the data.
8. And if in future we know that data is corrupted, it will happen only inside the module.
9. It makes software robust because later we can upgrade the module and

Coupling, cohesion are measures of how well modules work together and how well each individual module meets a certain single well-defined task and they tend to go together

Coupling

Primary Goals of Modularity:

1. Decomposability
2. Composability
3. Ease of Understanding

**Software Design: Coupling**

Coupling: Gives measure of how tightly coupled one module is to another.

Idea is when one module is changed, we hope/ensure that the change doesn’t affect other modules. It can be achieved only my low coupling. i.e. modules are not tightly bound.

Coupling levels:

1. Tight
2. Content coupling

Occur when module A relies directly on local data of module B.

1. Common coupling:

It occurs when both modules rely on common global data/variable.

1. External coupling:

It occurs when modules rely on external format e.g. protocol. It is highly tight coupling and can affect a large number of modules. Some abstractions are tried to create between module data and external format to minimise the impact.

1. Medium
2. Control coupling:

When one module control the logical flow of other module. Like what to do or how to do. Thus, changing first module affects the other module.

1. Data-structure coupling:

This occur when two modules rely on same composite data-structure.

1. Loose
2. Data coupling:

It occurs when modules share parameters i.e. elementary data like integer.

1. Message Coupling:

Loosest type of coupling. It occurs when components interact only through message or parameters.

1. No coupling:

Trivial case. This case doesn’t require our attention because any modules of no coupling aren’t of great importance or complexity.

**Software Design: Cohesion**

It tells how well within a module does components fit together.

Levels of cohesion:

1. Weak: when components are similar but not enough to be put together.
   1. Coincidental cohesion:

Two piece of code are coincidental if they are in same file. They are cohesive only because of their proximity.

* 1. Temporal cohesion:

Codes are connected because they are activated at same time.

* 1. Procedural cohesion:

Also a time based cohesion. Components are activated one after other

* 1. Logical cohesion:

It occurs when two components have similar function.

1. Medium:
   1. Communicational cohesion: It occur when two components work on same input and have same format of output.
   2. Sequential cohesion:

It is better form of procedural cohesion. The components are not just one after another, the part of first component provides input to the other component.

1. Strong:
   1. Object cohesion:

Every operation in a module is allowed to modify object attributes. Each part is designed for purpose within the object itself.

* 1. Functional cohesion:

Better than sequential cohesion. Here, each part of the component is necessary for the execution of a single well defined function or a behaviour.

Cohesion and Inheritance:

Inheritance weakens cohesion.

In inheritance, the child class doesn’t have all functionality. It has to visit parent/super class. However, inheritance has its own benefit over cohesion.

**Inheritance:**

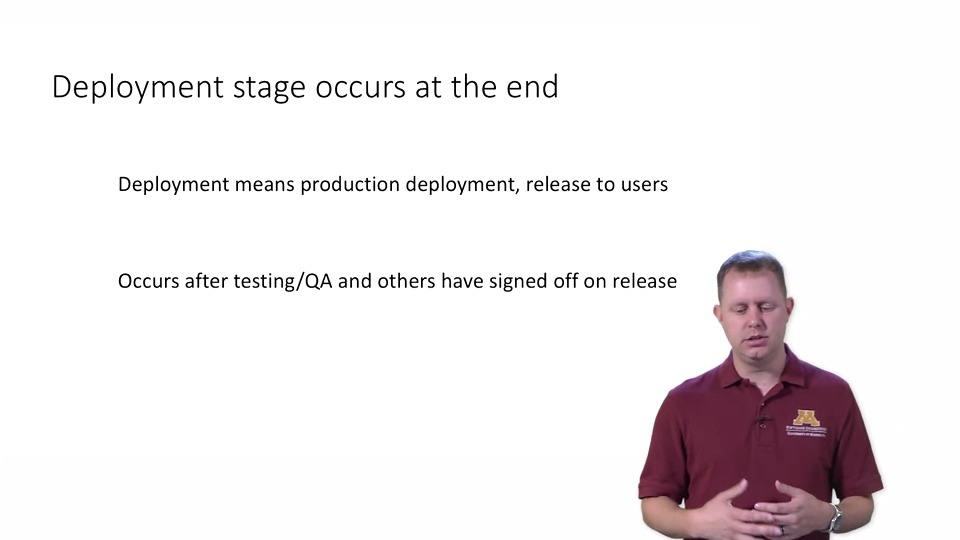
Let your comments describe the **why** and let the code describe the **how.**

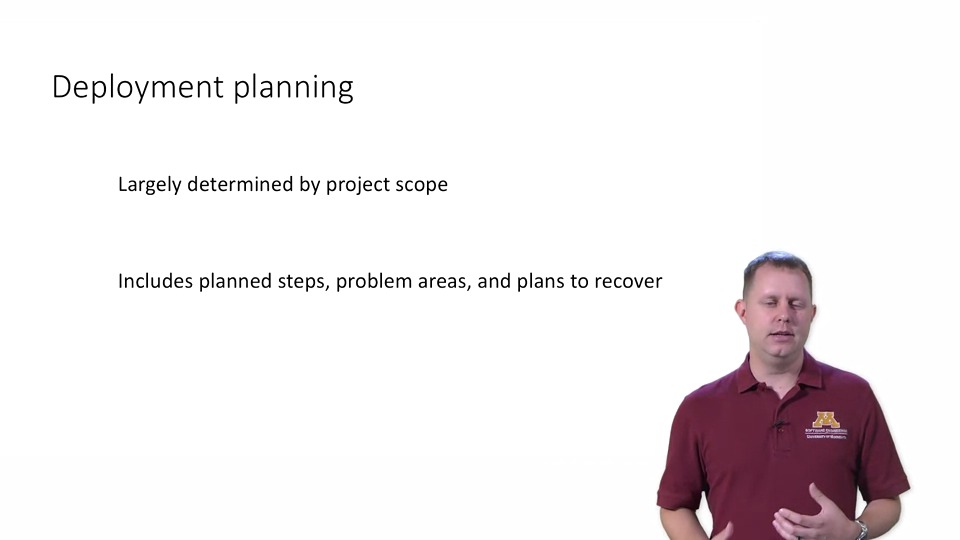
The Google style guide for C++ recommends that if a function is longer than 40 lines, you need to think about breaking it down.

So if you're using it twice, make it a method.

Optimise only when you are sure it is necessary.

Deployment:

****

****

Even more important than deployment, you have to make sure that you plan for recovery. One of the issues that we had in the business that I used to work in was that we were not allowed to even begin a production deployment unless our recovery plans had been approved by change management.

So there's an entire team of people that look over our plans for how we're going to install the software, but not only that, the plan set up in advance of how we're going to roll that back,

how we're going to get out of it if we somehow screw it up when we do

try to install it because if you're installing something on production, a production system that's supposed to be up 24/7, servicing millions of customers every hour,

you can't have the system entirely go down.

You need to have a plan to roll those changes

back to make sure that the system can get back into

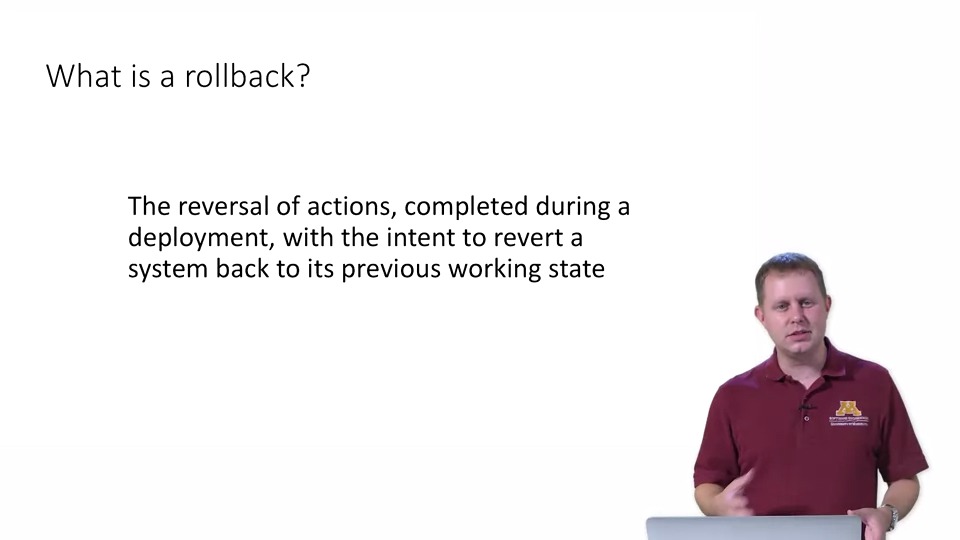
a working state if whatever you were planning to do doesn't end up working.

Deployment:

Rollback is what happens when that deployment doesn't go as you intended.

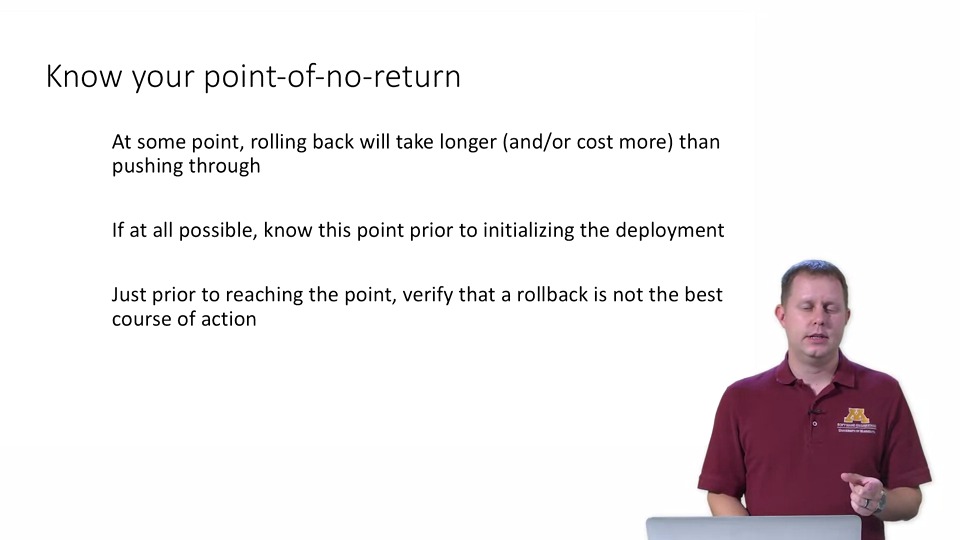
Why Roll-Back?

There's lots of reasons why you rollback, but most of them have something to do with things not going quite the way you expected them to.

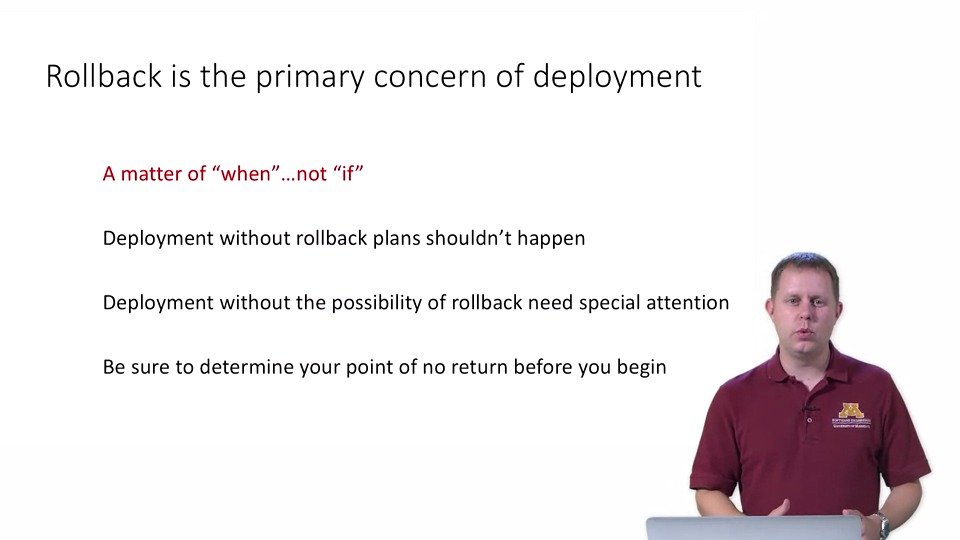


Point of no-return:

There's at some point rolling back is going to take longer than you have.



Summary of Roll-Back:



Deployment Cutover Strategy:

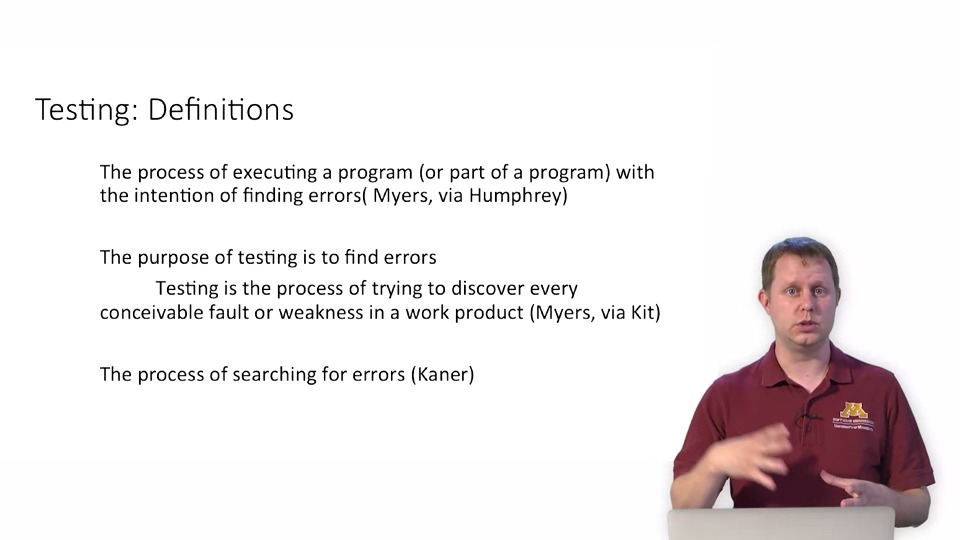
Maintaining backup or storage. If your current server goes down, then there should be a back-up server ready to continue the ongoing function and storing the transaction information.

Backup Strategy:

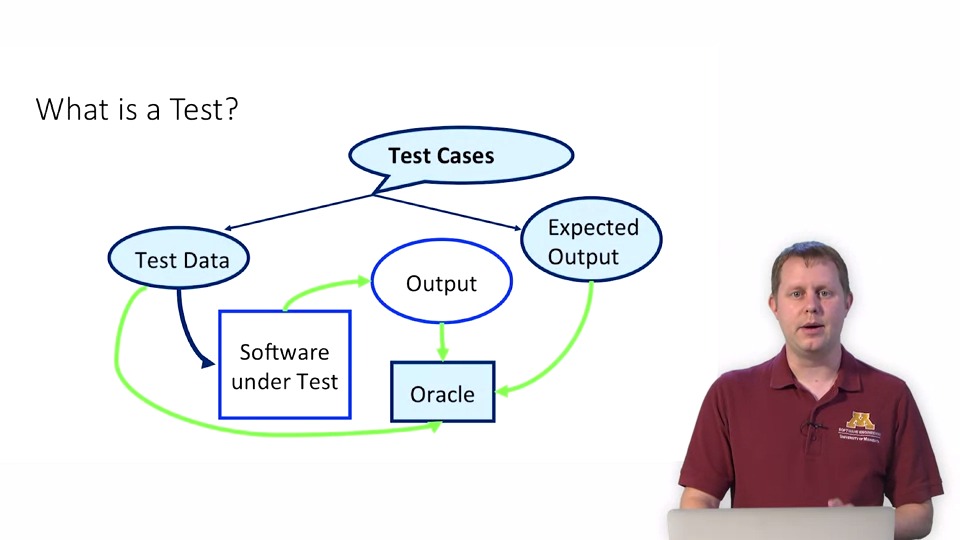
1. Cold storage/ Cold Backup: requires installation and full set-up
   1. Only an office is present.
   2. We have to assemble hardware, do the installation of software and other task.
   3. Usually takes 24 hours to go back online starting from making decision to software installation to verification to deployment.
2. Warm Standby: machine running but still require installation
   1. Hardware is already present and installation is or have to be done.
   2. All you got to do is to run the server, do verification and deploy.
   3. It usually takes 2-3 hours to get system started.
3. Hot Failover: server is running
   1. Everything is present.
   2. Hardware and software are already installed. System is running but doesn’t get any transaction.
   3. If failure occurs, we transfer the transaction to new system and then it starts to store data.
   4. It is different than node balancing. In node balancing, there are multiple servers present and all server share the load of transaction.
   5. But in hot failure, both systems are running and if main system fails, the second system gets all the traffic.
   6. It usually takes 30 minutes to get everything done. It includes the time to make decision and the time to make the switch.
   7. Hot failover are tested regularly and during the testing the switch is made to send all the transaction to the new system.

**Software Testing and Verification:**

Introduction:



Note that the first definition though, excludes the testing of documentation, which is definitely something that we do. Also, testing of documentation is done.

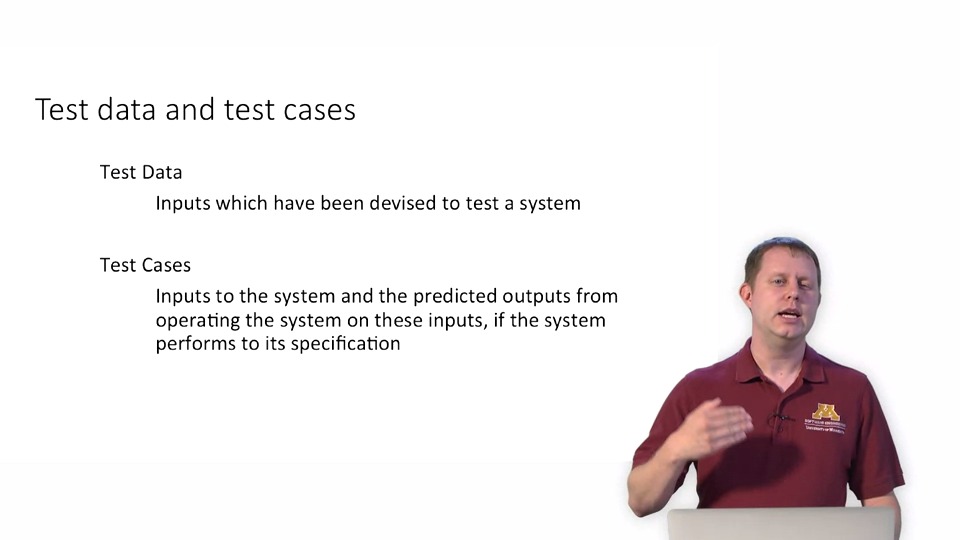


Has the software provided the correct result given the test data?

Oracle determines if the output given by software is correct or not.

Traditionally, oracle is developer or tester but now the oracles are automated.

**Test case and Test Data:**



1. Test data:

The test data is just the input to the program.

1. Test Case:

A test case has the data, the input and what you expect for each individual output. We're going to have automated oracles in the future, which means that you need both.

Test Failures/ Bug/ Error:

System failure occurs when the delivered service deviates from the specified service.

That means that something didn't happen the way it was supposed to.

The failure occurred because the system was erroneous.

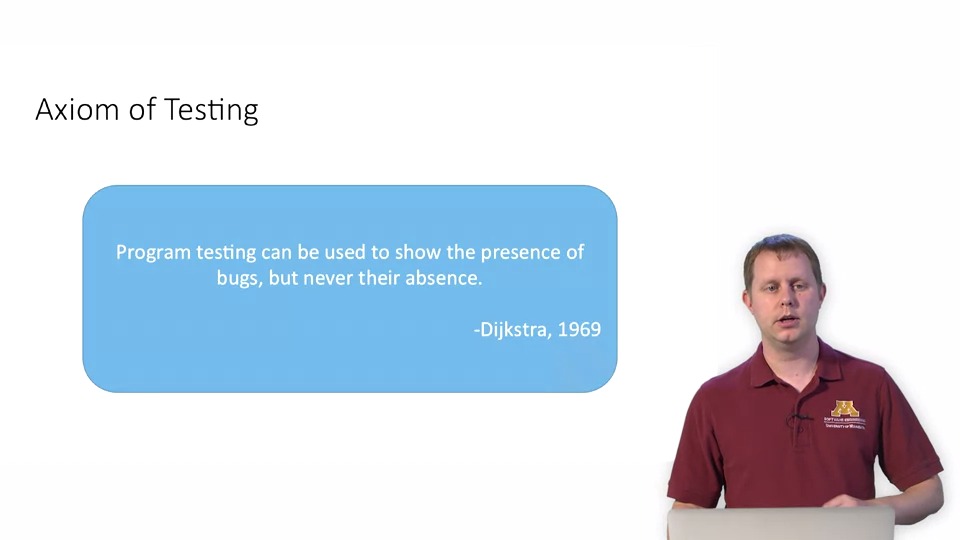
Error/fault:

The part of the system state which is liable to lead to a failure is called to be in error.

It's whatever is wrong that leads to delivery of a surface that doesn't comply with how it's specified to work.

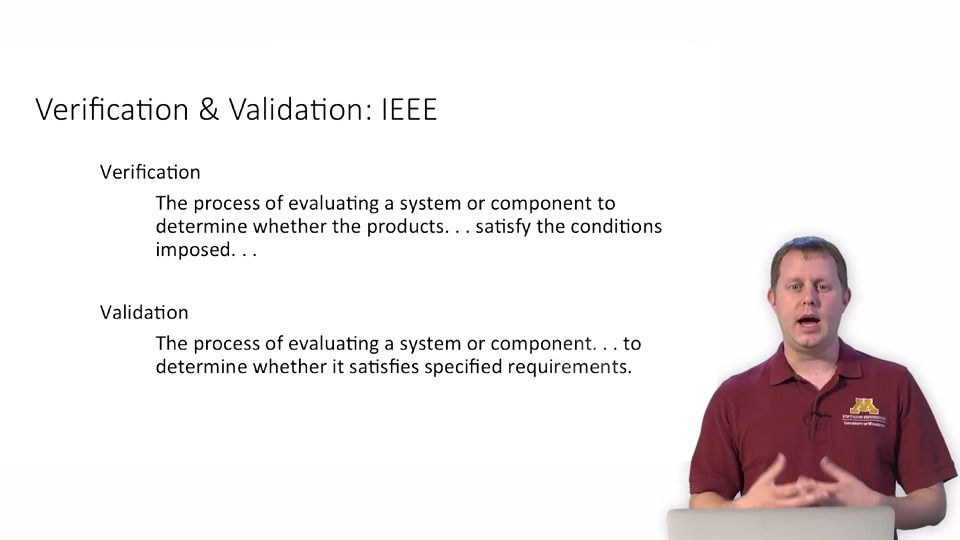
The cause and it's phenomenological sense is an error, is a fault.

A fault is the manifestation of an error.



**Verification and Validation Definition:**

By IEEE:

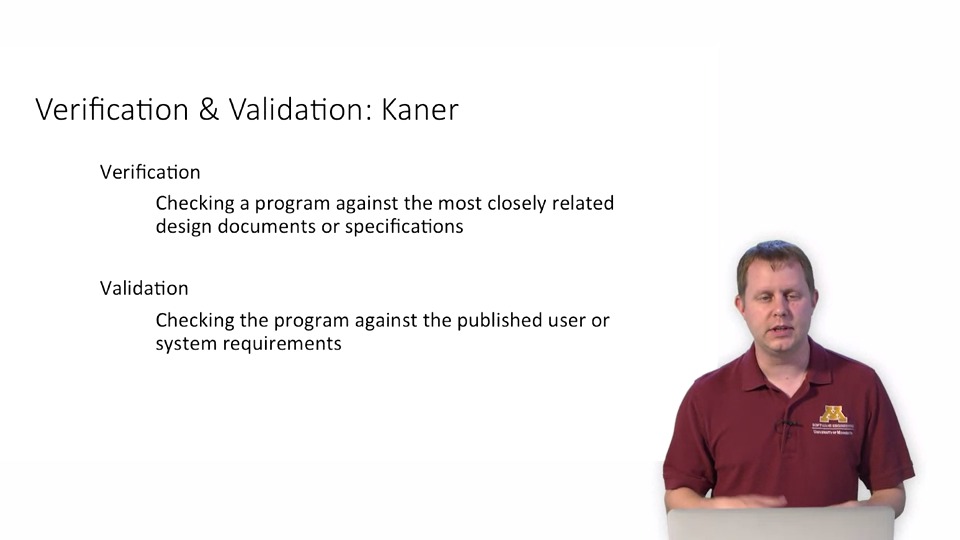


Satisfying Conditions Imposed: the conditions imposed on the system, by the developers as we try and transition from what the user wants, the requirements, into what the system does in order to meet those requirements.

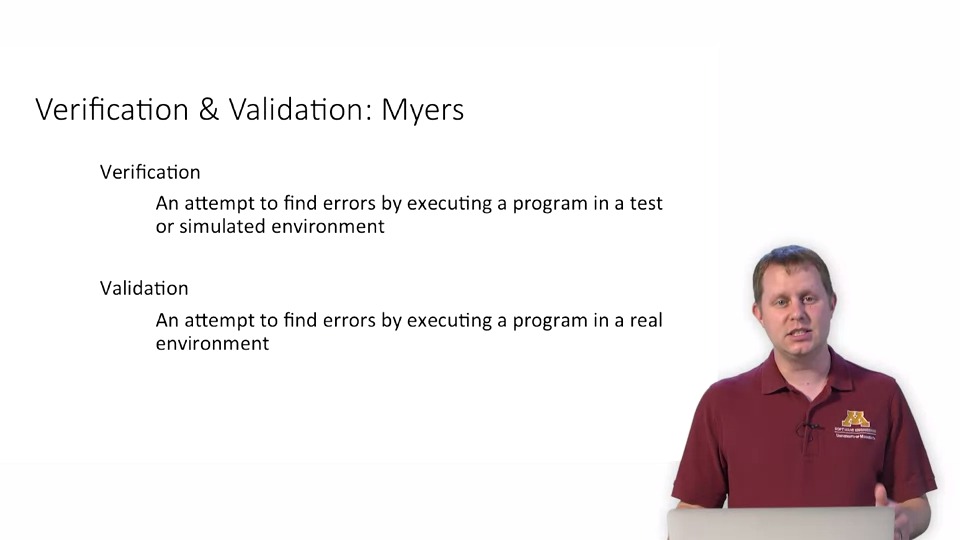
Specified Requirement:

it is specified as in told, explained by the users, it's what the users really want. Requirements are not necessarily a written-down document that we would really like it to be, but it's what the user wants.

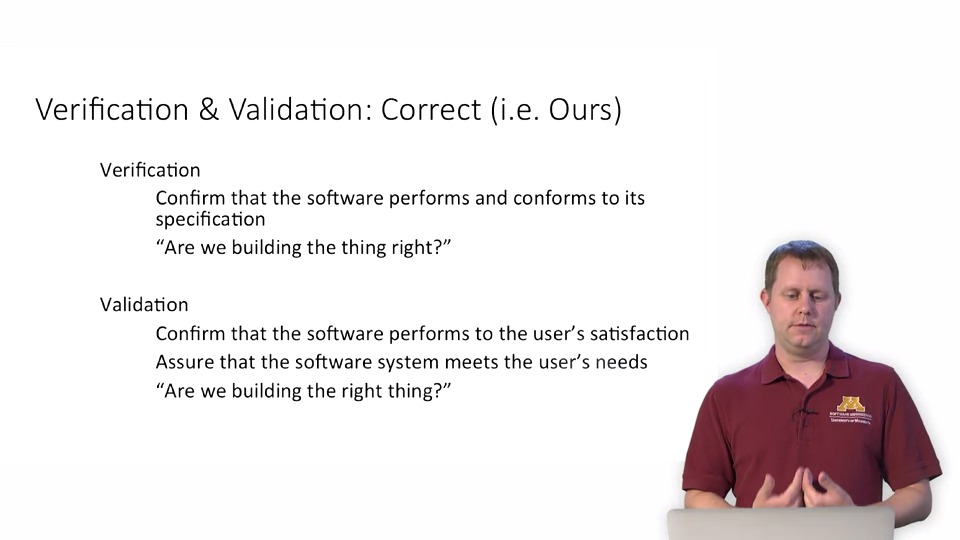
By Prof. Kaner:



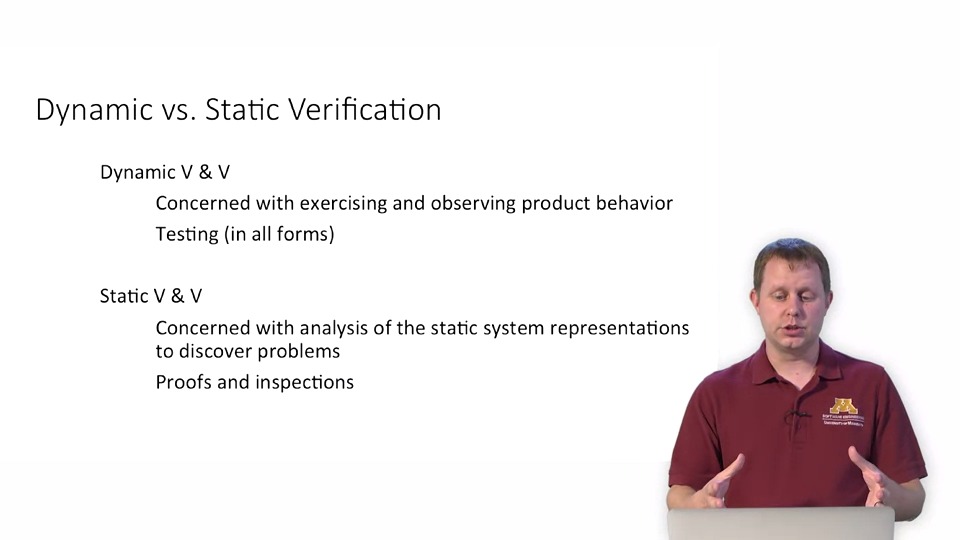
By Author Myers: (Specifically for dynamic testing )



By Instructor:



Dynamic V & V Definition:

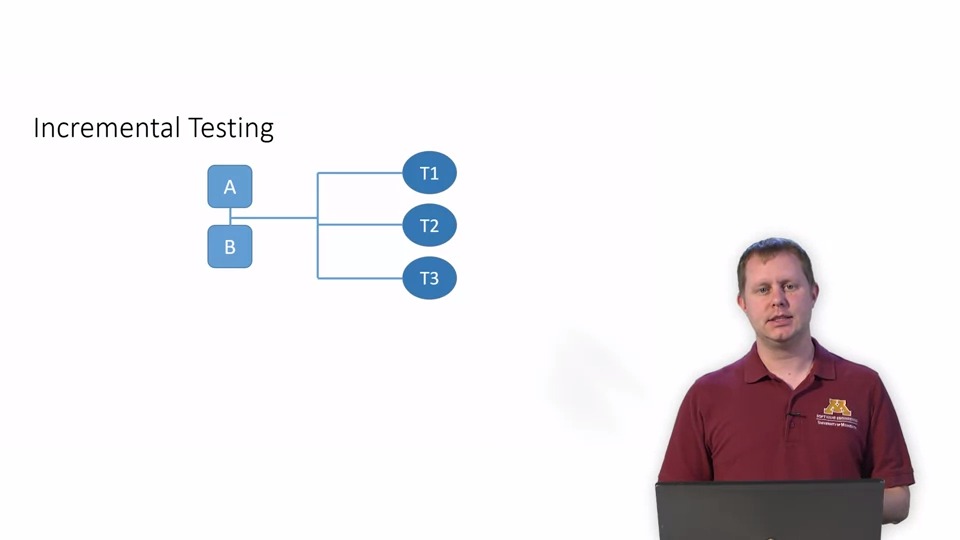


Dynamic is basically all forms of testing.

**Software Testing Strategy:**

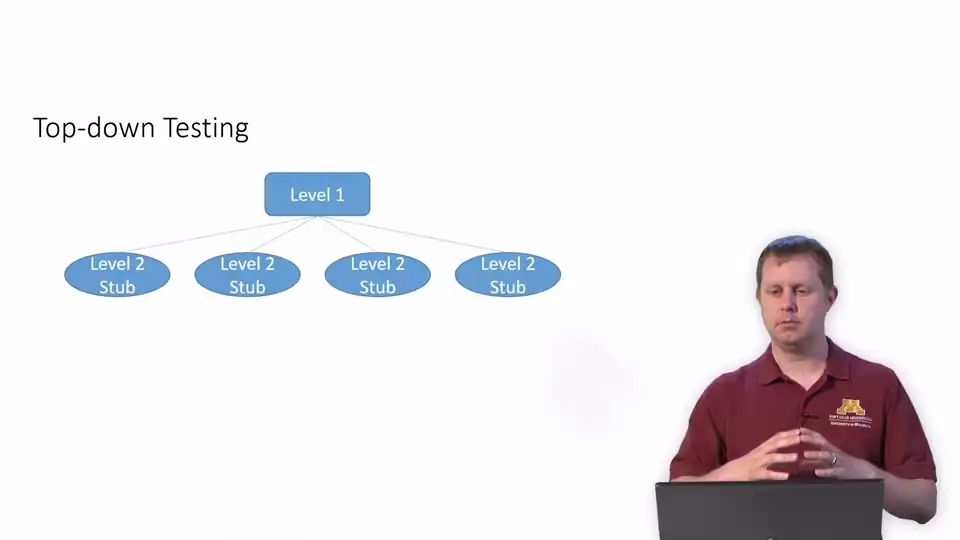
Types:

1. Incremental Testing:



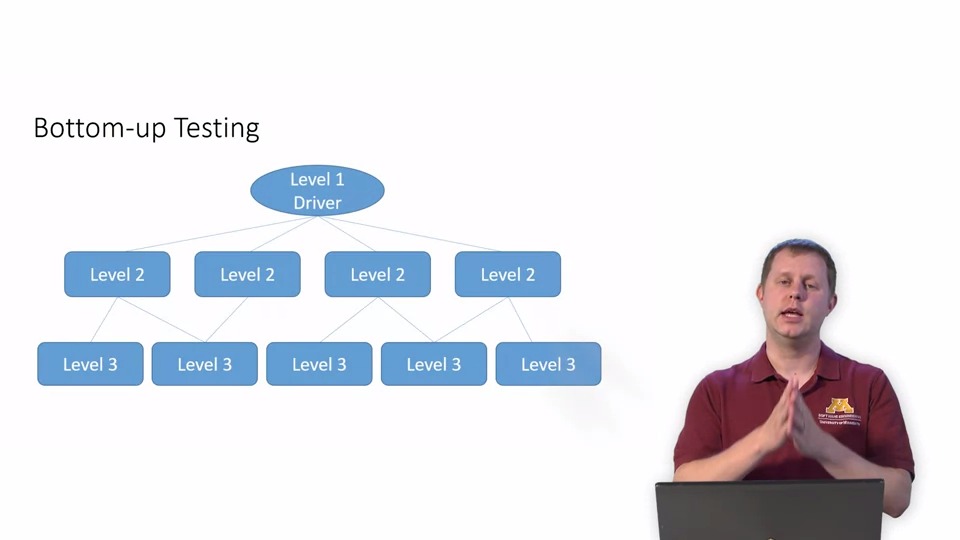
* 1. When you are done testing module set say A and B, you don’t get rid of its tests, you keep them.
  2. You take next module say C and test it.
  3. Then you combine module set A, B with module C and also their test.
  4. By that, we know how modules behave in isolation and do they change when combined with others.

1. Top-Down Testing:



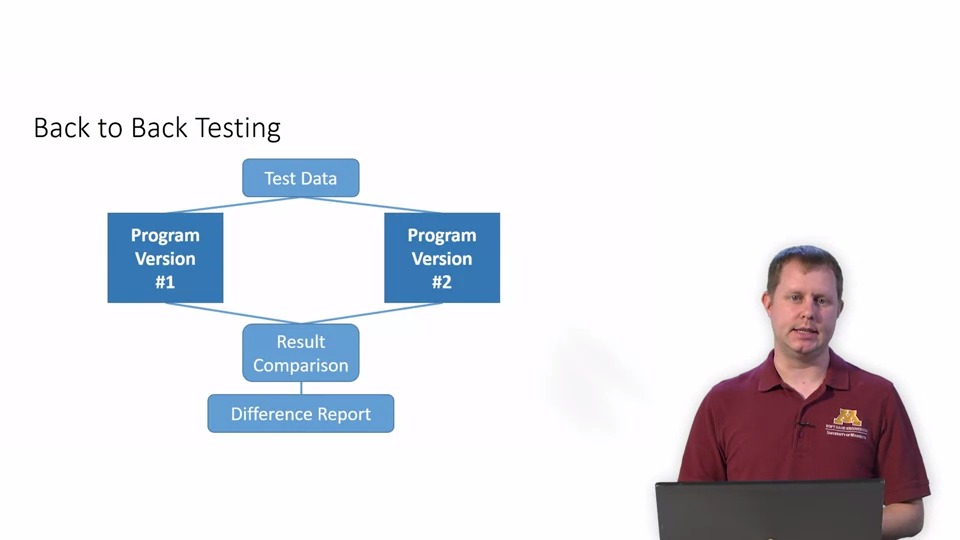
* 1. When you're developing Top-down you have to develop something to stand in for the elements at lower levels that you haven't created yet. These are called stubs.
  2. A Stub is something that is a few lines of code that when called it essentially just returns a hard coded value that stands in for a real return value.
  3. An upper level element say Level 1 rely on lower level 2 element for some value. These lower level could be an object that upper level element will instantiate.
  4. The same kind of thing can be done with what's called a Mock.
  5. A Mock is something where you don't actually hard code something, you just say, was this method called, yes, you move on.

1. Bottom-Up Testing:



* 1. When you have the lower lying implementation's complete but you don't have the larger picture integration execution driver hence, Drivers.
  2. Drivers i.e. higher level element say level 2 call the lower level element say level 3 to see if they are operational.

1. Back to Back Testing:

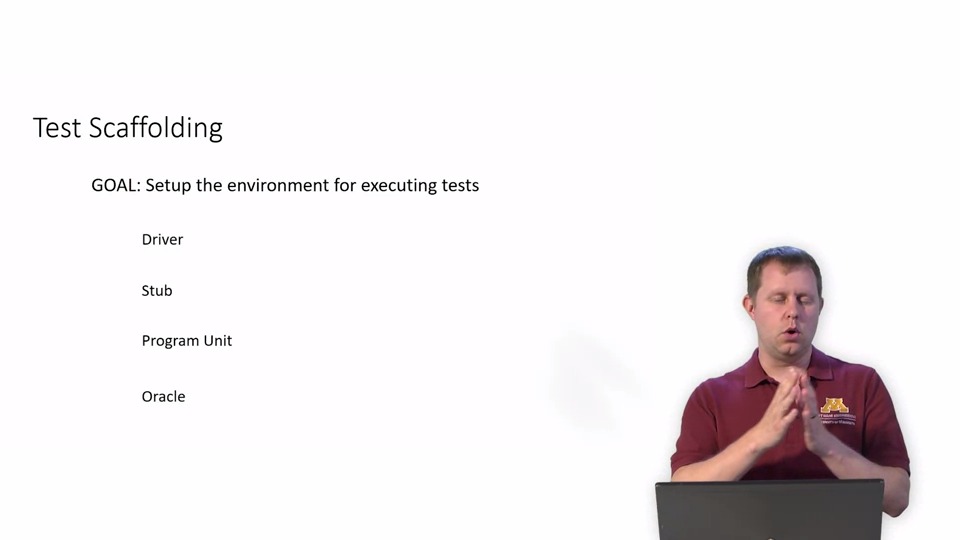


* 1. Here, we make use of earlier iterations of a program as an effective automated Oracle.
  2. This is particularly useful for expanding test data without necessarily including expected output or if you don't already have automated tests from before.
  3. Program has worked before and we have its output. Later the new version of program is available and the testing method compares both the new and old version. We do direct comparison of the output and it should be the same.
  4. Alternatively, we can change if output is different if we want it to be different.

Note: In real practice, top-down or bottom-up is hard to implement. The structure of software may not necessarily follow that pattern.

**Test Scaffolding:**

The goal is setting up an environment for executing your tests.



Components:

1. Driver:

The Driver initializes non-local variables, initializes parameters and activates units under test.

1. Stub:

Stubs will use templates of modules not actual working modules usually, that's why it's a Stub.

1. Program Unit:

Templates of the modules used by the unit and templates of any other entity, or data structure that is used within the unit, that is the Program Unit.

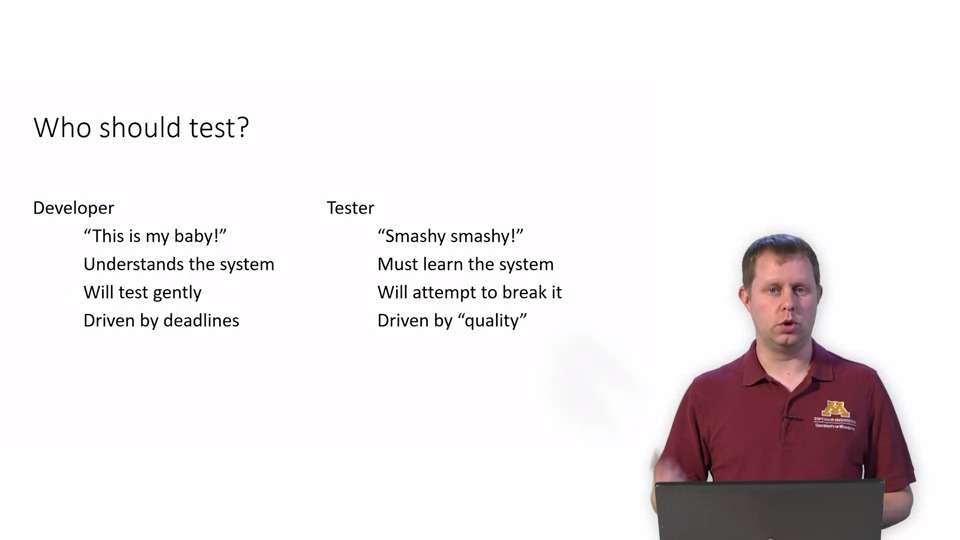
1. Oracle:
   1. The Oracle then, is at the end which verifies the correspondence between produced and expected results.
   2. Oracle can be human developer or automated.
   3. Automated Oracle could be:
      1. Star Unit
      2. J Unit
      3. PI Unit
   4. It checks that our driver, stub and program unit has worked properly.

**Trade-off:**

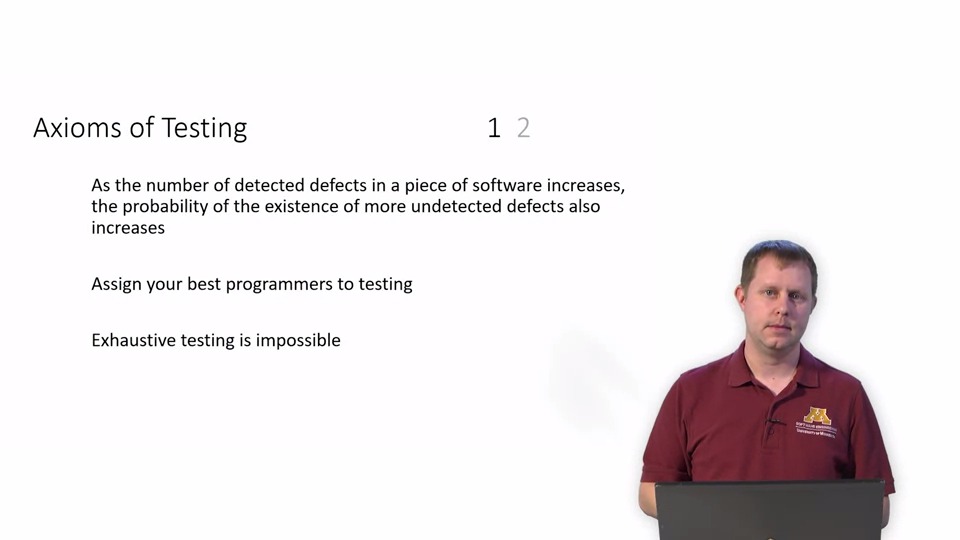


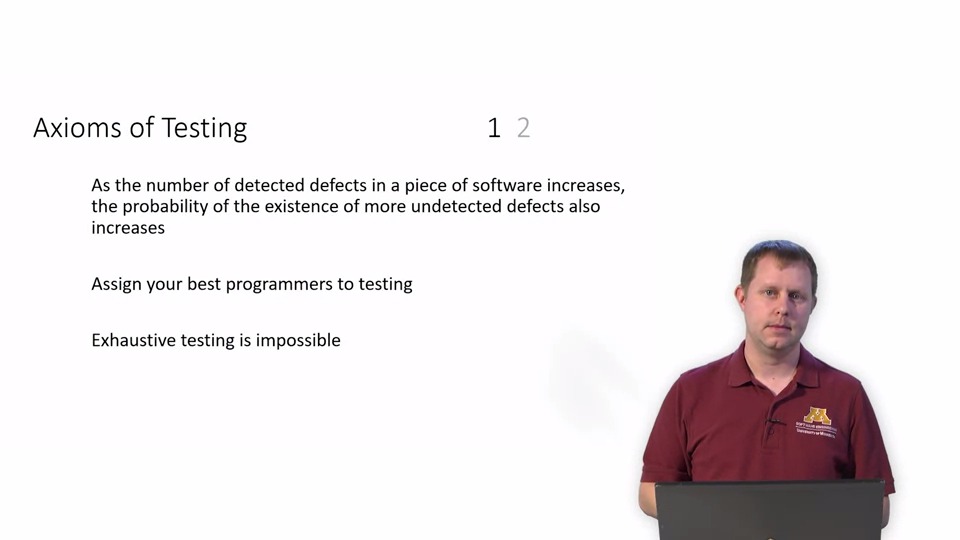
1. One way:
   1. You can build very sophisticated, well-designed Drivers and Stubs, very, very high effort in developing those drivers and Stubs.
   2. But you get a lot of lower effort in text execution and regression by the nature of having produced these very sophisticated Drivers and Stubs. On the other side, we have the poorly designed Stubs
2. On the other hand:
   1. We have the poorly designed Stubs.
   2. Those are poorly designed Drivers and Stubs.
   3. It doesn't take very long at all in development but there are really isn't a whole lot of reuse you can have.

**Why should we Test?**

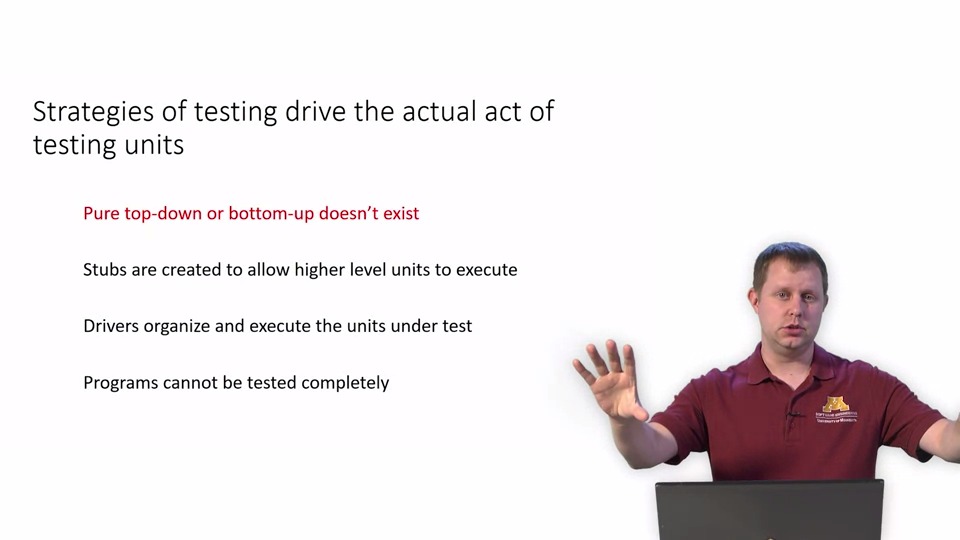


**Axioms of Testing:**

****

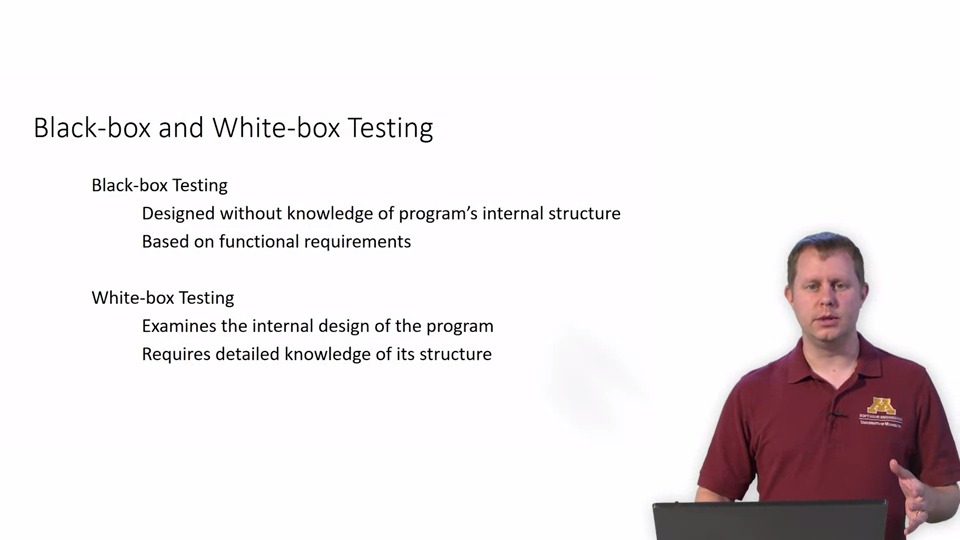
****

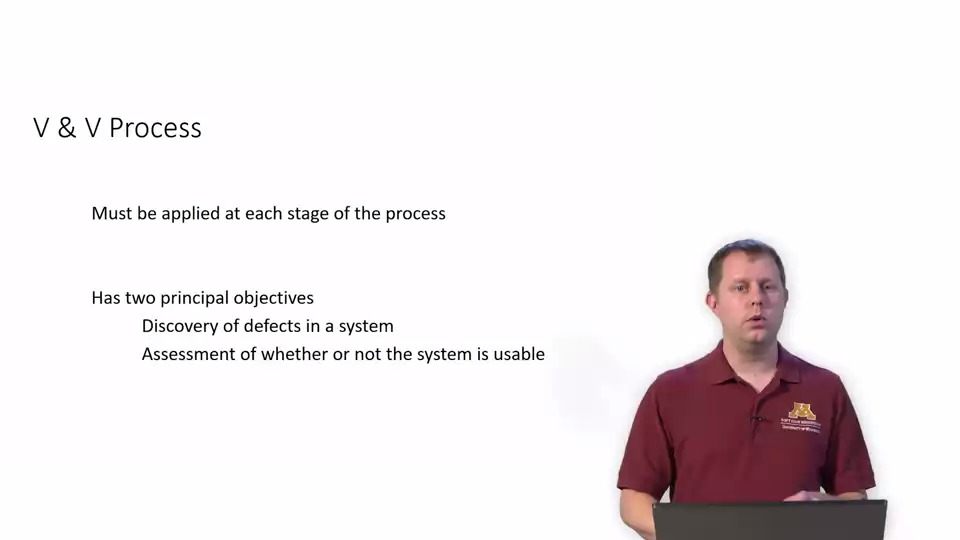
**Summary:**

****

**Software Testing Perspective:**

**Black box and White box Testing:**

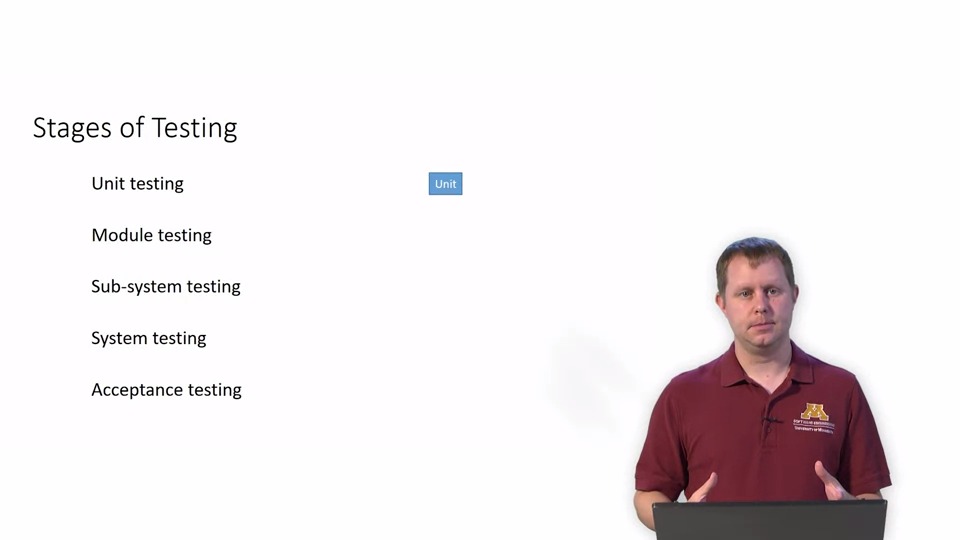
**** V & V Process:



We want to provide quality in our product, but it's not only about find and fix for behaviour. It's also about measuring or evaluating the system in other vectors of quality. e.g.

1. how well does that work when the CPU has that 90% load?
2. If there are a lot of users all on the system at the same time, does the database connection usage reach its limit? What happens if a server shuts down unexpectedly?
3. And how easy is it to hack into the system and corrupt data.

Stages of Testing:



Two or more unit make a module. Two or more module make a sub-system and so on.

Lastly, Acceptance testing or alpha testing or beta testing:

1. Testing by users.
2. Worst type of testing.

**Summary:**

