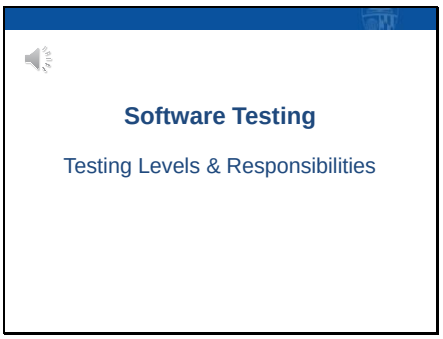
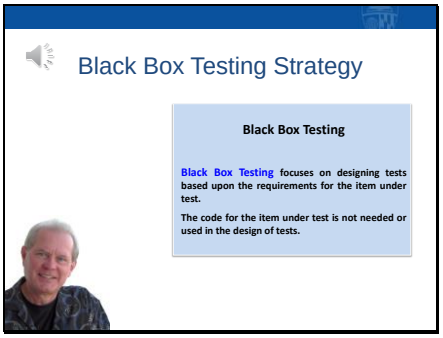
1



In this lecture we'll discuss different testing levels & responsibilities

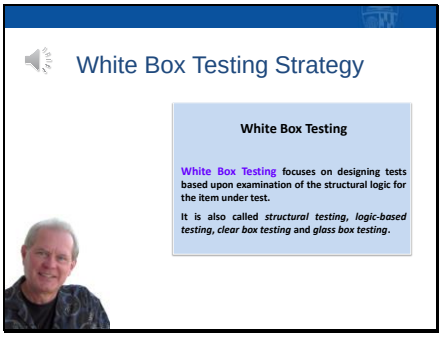
2



Let me start this lecture by introducing some of the basic test design strategies used in practice.

The first strategy is called black box testing…and in this strategy tests are designed from the requirements. You saw an example of black box testing in the triangle exercise in an earlier lecture.

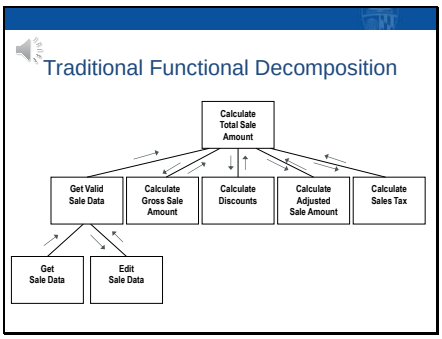
3



Another strategy for testing is commonly called white box testing or logic-based testing. In this approach the test designer uses the code as part of the test case design strategy.

Now, in practice, both approaches to testing must be used…the difference is which approach is best-suited to the various levels of testing and who is doing the testing.

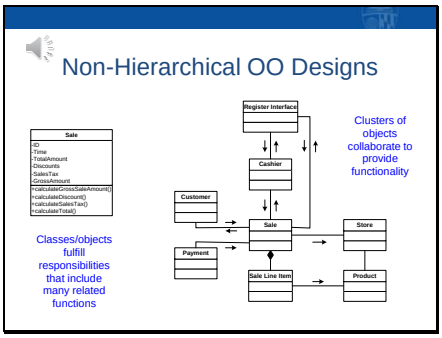
4



Before I discuss the different testing levels, I want to take a minute to compare some of the differences between traditional software design architecture and object-oriented design architecture…because the differences drive slightly different approaches to testing, at least at the lower levels of testing.

Traditional approaches to software design result in architectures that exhibit what we call functional decomposition. Components that perform higher levels of functionality…like calculating the total amount of a sale…are decomposed into simpler and simpler components that provide pieces of the higher level functionality, as this design diagram illustrates.

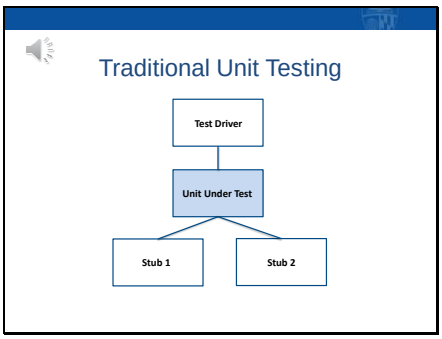
5



Object-oriented designs are generally not functionally decomposed, though functional decomposition techniques can certainly be used in the design of complex methods for a class. Object-oriented designs tend to be driven by class responsibilities and collaborations rather than by functional decomposition.

Clusters of objects collaborate to provide a product's functionality…as illustrated in this diagram. Object-oriented designs are also comprised of layers that fulfill certain responsibilities. For example, a presentation layer would contain classes that are responsible for the user interface, an application or domain layer would contain classes like the ones in this diagram, a persistence layer would contain classes that are responsible for storing and retrieving the information stored in objects…and so forth.

6



The lowest level of testing is called unit testing. A unit is the smallest element that can be separately compiled and tested. Depending on the implementation language a unit could be a subroutine or a function.

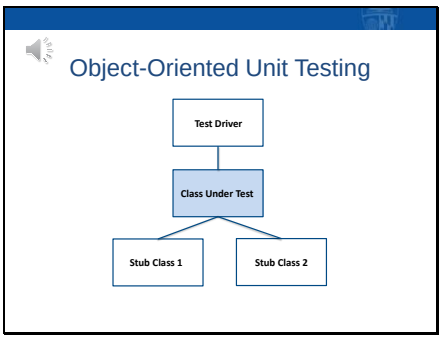
In traditional…or functional oriented…software development, testing units independently typically requires the use of test drivers and test stubs if different software engineers are tasked with developing different units.

A test driver serves as a main program that contains or accepts test case data that is fed to the unit under test.

A stub is software that in some way simulates the behavior of units that are called by the unit under test. For example, if the unit under test called two other units, one way to test it would be to develop stubs for the called units that maybe just printed a message that they were called or printed any data that were passed to them from the unit under test. When the actual code for the called units is available, they are integrated into the configuration and tested.

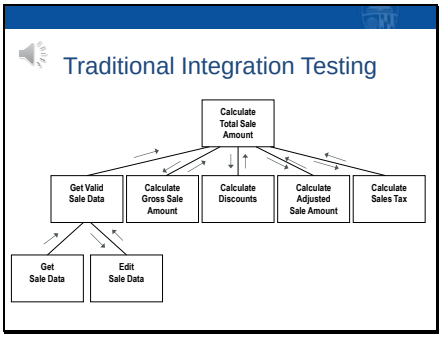
If units are scheduled to be developed bottom up…that is, from the bottom of the component hierarchy to the top, then the actual called units would be available and stubs may not be required.

7



Unit testing in object-oriented development projects works a little differently. Since the individual methods belong to a class, some or all of the methods are developed and then individual methods of the class are tested one at a time. A test driver simply creates a class instance and invokes one or more of its methods. If a method calls another method in the same class, either a stub is used or the actual method is used. If a method calls a method in another class, a stub may be used if the other class is not yet developed.

8



Integration testing involves testing product components that make up a particular hierarchy or collaborative relationship. In traditional integration testing there are various integration test strategies that are used in practice…such as top-down and bottom-up strategies.

Here's an example of a traditional functionally decomposed design. In top-down integration testing, we would start with the top-most unit and use it as the basic test driver and control module.

If we were to use a level-by-level, or breadth-first approach, we would integrate the first five components directly subordinate to the calculate total sale amount component, and use stubs for the get sale data and edit sale data components. We would then move down to the next level in the control hierarchy and integrate those two modules.

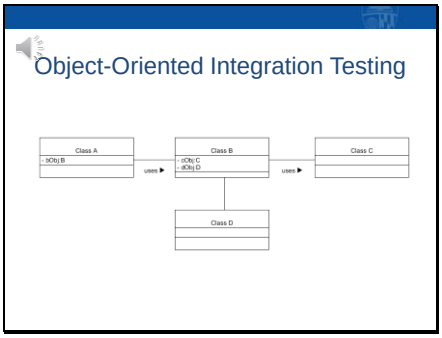
If we were to use a depth-first integration test approach, we would pick a control path, like the left-most one, and integrate it with the calculate total sale amount component, using stubs for the other first-level components. Then, we would integrate the first-level components one at a time.

In bottom-up testing, we start at the bottom of the hierarchy and work our way up. We'd typically start with a subgroup of components that work together to

provide a higher-order level of functionality. In this example, that would be the get sale data and edit sale data components. We'd then develop a test driver to harness and invoke those components. We'd then move up a level, replacing the driver with the actual get valid sale data component…and so forth.

Another integration test strategy, sometimes called the "big bang" strategy, is to take all the components, build the entire architecture, and test everything at once. That might be okay for very small products, but is not recommended for large ones.

9

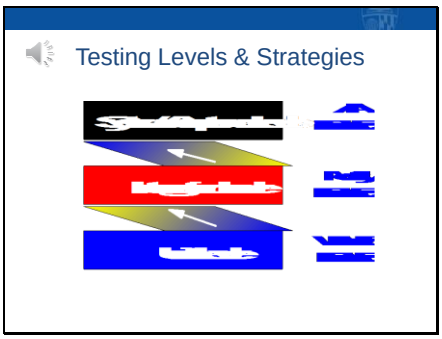


In object-oriented integration testing, there really isn't a functional-decomposition hierarchy, so we typically look at class collaborations, dependencies, and architectural layers.

In this example, Class A is dependent upon Class B, and Class B is dependent upon C and D. So, one integration strategy that would work here is to first develop Class C and Class D, then integrate them into Class B, and then integrate B into A. Test drivers would be used to create and run the tests at each of these increments.

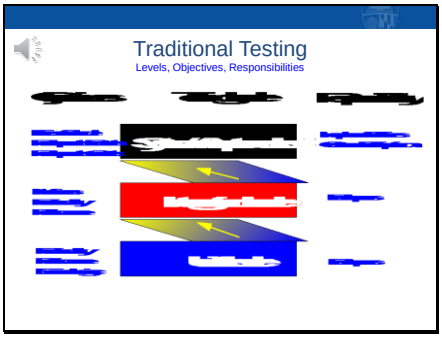
Object-oriented design architecture is layered, so maybe these four classes constitute the application layer of the architecture. We may then use a test driver to simulate the user-interface layer during the integration increments and while the actual GUI components are still under development.

10



Depending on the level of test that is being done, the testing strategies will be different. At the unit level, both black box and white box tests are used. As we go into integration testing, tests generally become mostly black box in nature, and at the system and acceptance levels...where the software product is tested in its entirety...the tests should be all black box.

11



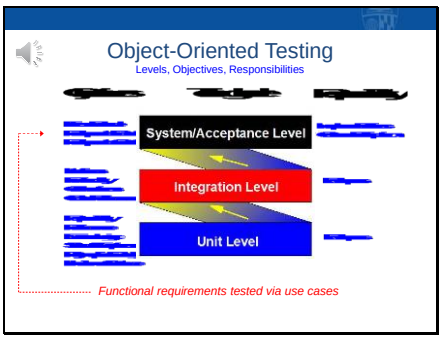
In terms of test responsibilities, developers typically perform the unit level and integration levels of test...though on very large scale products, an independent test team might be responsible for integrating major subsystem components. It's very common for an independent test team to perform the system level testing, and customers may be responsible for performing or at least participating in acceptance tests.

In terms of test objectives, for traditional testing the unit level is where it is important to exercise a high percentage of program logic, test low level functionality, and low level performance requirements.

Integration testing usually focuses on higher level functionality, component interfaces, and higher-level component performance.

And...at the system and acceptance levels the focus is on requirements completeness and correctness and overall fitness for use.

12



When it comes to object-oriented testing, the responsibilities are the same, and the objectives at the system and acceptance levels are the same...but there are additional types of testing that need to be done at the lowest levels, as indicated here.

There are different types of relationships that exist between classes in object-oriented software development, and those relationships require that more types of testing be done.

Also, object-oriented projects tend to be use case driven, so it is common for the use cases to serve as the basis for system and acceptance testing...at least for a product's functional requirements.