SOFTWARE TESTING

CO3015 / CO5252

CH5. UNIT TESTING

Content

- Unit testing
- Test case design
- Incremental testing
- Top-down vs. Bottom-up testing
- Test automation

Unit testing

Module testing (or unit testing) is a process of testing the individual subprograms, subroutines, classes, or procedures in a program. More specifically, rather than initially testing the program as a whole, testing is first focused on the smaller building blocks of the program. The motivations for doing this are threefold. First, module testing is a way of managing the combined elements of testing, since attention is focused initially on smaller units of the program. Second, module testing eases the task of debugging (the process of pinpointing and correcting a discovered error), since, when an error is found, it is known to exist in a particular module. Finally, module testing introduces parallelism into the program testing process by presenting us with the opportunity to test multiple modules simultaneously.

Unit testing

- 1. The manner in which test cases are designed.
- 2. The order in which modules should be tested and integrated.
- 3. Advice about performing the tests.

Test case design

- Using white-box and/or black-box testing techniques

The Strategy

The test-case design methodologies discussed in this chapter can be combined into an overall strategy. The reason for combining them should be obvious by now: Each contributes a particular set of useful test cases, but none of them by itself contributes a thorough set of test cases. A reasonable strategy is as follows:

1. If the specification contains combinations of input conditions, start with cause-effect graphing.
2. In any event, use boundary value analysis. Remember that this is an analysis of input and output boundaries. The boundary value analysis yields a set of supplemental test conditions, but as noted in the section on cause-effect graphing, many or all of these can be incorporated into the cause-effect tests.

Test case design

- Using white-box and/or black-box testing techniques

3. Identify the valid and invalid equivalence classes for the input and output, and supplement the test cases identified above, if necessary.
4. Use the error-guessing technique to add additional test cases.
5. Examine the program's logic with regard to the set of test cases. Use the decision coverage, condition coverage, decision/condition coverage, or multiple-condition coverage criterion (the last being the most complete). If the coverage criterion has not been met by the test cases identified in the prior four steps, and if meeting the criterion is not impossible (i.e., certain combinations of conditions may be impossible to create because of the nature of the program), add sufficient test cases to cause the criterion to be satisfied.

Test case design

- Using

Summary

Once you have agreed that aggressive software testing is a worthy addition to your development efforts, the next step is to design test cases that will exercise your application sufficiently to produce satisfactory test results. In most cases, consider a combination of black-box and white-box methodologies to ensure that you have designed rigorous program testing.

Test case design techniques discussed in this chapter include:

- *Logic coverage.* Tests that exercise all decision point outcomes at least once, and ensure that all statements or entry points are executed at least once.
- *Equivalence partitioning.* Defines condition or error classes to help reduce the number of finite tests. Assumes that a test of a representative value within a class also tests all values or conditions within that class.
- *Boundary value analysis.* Tests each edge condition of an equivalence class; also considers output equivalence classes as well as input classes.
- *Cause-effect graphing.* Produces Boolean graphical representations of potential test case results to aid in selecting efficient and complete test cases.

Test case design

- Using white-box and/or black-box testing techniques

- *Error guessing*. Produces test cases based on intuitive and expert knowledge of test team members to define potential software errors to facilitate efficient test case design.

Extensive, in-depth testing is not easy; nor will the most extensive test case design assure that every error will be uncovered. That said, developers willing to go beyond cursory testing, who will dedicate sufficient time to test case design, analyze carefully the test results, and act decisively on the findings, will be rewarded with functional, reliable software that is reasonably error free.

Test case design

https://reqtest.com/testing-blog/test-case-design-techniques/

- Techniques:
  - Specification-Based techniques
  - Structure-Based techniques
  - Experience-Based techniques
- Specification-Based techniques = Black-box techniques
  - Boundary Value Analysis
  - Equivalence Partitioning
  - Decision Table Testing
  - State Transition Diagrams
  - Use Case Testing
- Structure-Based techniques = White-box techniques
  - Statement Testing & Coverage
  - Decision Testing Coverage
  - Condition Testing
  - Multiple Condition Testing
  - All Path Testing
- Experience-Based techniques
  - Error Guessing
  - Exploratory Testing

Incremental testing

https://www.tutorialspoint.com/software_testing_dictionary/incremental_testing.htm

- After unit testing is completed, developer performs integration testing. It is the process of verifying the interfaces and interaction between modules. While integrating, there are lots of techniques used by developers and one of them is the incremental approach.

- In Incremental integration testing, the developers integrate the modules one by one using stubs or drivers to uncover the defects. This approach is known as incremental integration testing. To the contrary, big bang is one other integration testing technique, where all the modules are integrated in one shot.

Incremental testing

https://www.tutorialspoint.com/software_testing_dictionary/incremental_testing.htm

- Incremental Testing Methodologies
  - **Top down Integration** - This type of integration testing takes place from top to bottom. Unavailable Components or systems are substituted by stubs
  - **Bottom Up Integration** - This type of integration testing takes place from bottom to top. Unavailable Components or systems are substituted by Drivers
  - **Functional incremental** - The Integration and testing takes place on the basis of the functions or functionalities as per the functional specification document.

Incremental testing

https://www.tutorialspoint.com/software_testing_dictionary/incremental_testing.htm

- Incremental Testing - Features
  - Each Module provides a definitive role to play in the project/product structure
  - Each Module has clearly defined dependencies some of which can be known only at the runtime.
  - The incremental integration testing's greater advantage is that the defects are found early in a smaller assembly when it is relatively easy to detect the root cause of the same.
  - A disadvantage is that it can be time-consuming since stubs and drivers have to be developed for performing these tests.

Top-down vs. Bottom-up testing

Chapter 13

Integration Testing

Tc

TABLE 5.3 Comparison of Top-Down and Bottom-Up Testing

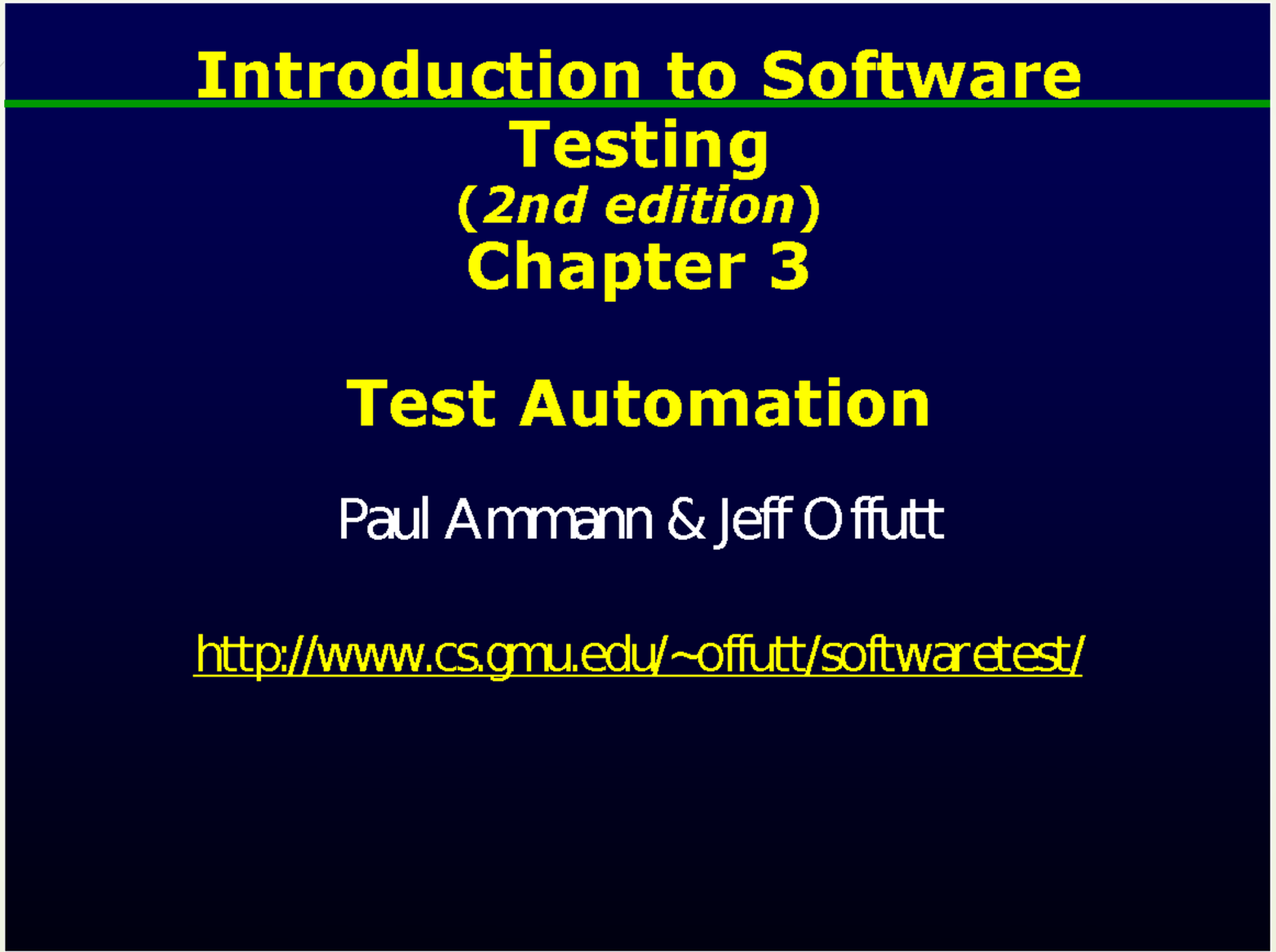
Top-Down Testing

Advantages	Disadvantages
1. Advantageous when major flaws occur toward the top of the program.	1. Stub modules must be produced.
2. Once the I/O functions are added, representation of cases is easier.	2. Stub modules are often more complicated than they first appear to be.
3. Early skeletal program allows demonstrations and boosts morale.	3. Before the I/O functions are added, the representation of test cases in stubs can be difficult.
	4. Test conditions may be impossible, or very difficult, to create.
	5. Observation of test output is more difficult.
	6. Leads to the conclusion that design and testing can be overlapped.
	7. Defers the completion of testing certain modules.

Bottom-Up Testing

Advantages	Disadvantages
1. Advantageous when major flaws occur toward the bottom of the program.	1. Driver modules must be produced.
2. Test conditions are easier to create.	2. The program as an entity does not exist until the last module is added.
3. Observation of test results is easier.	

Test automation



Web automation testing tool

https://www.guru99.com/top-20-web-testing-tools.html

- https://www.selenium.dev/
  - Selenium Tutorial for Beginners: Learn WebDriver in 7 Days
    - https://www.guru99.com/selenium-tutorial.html

Summary

- Unit testing
- Test case design
  - Specification-Based techniques: Black-box
  - Structure-Based techniques: White-box
  - Experience-Based techniques
- Incremental testing: Integration testing
  - Top-down vs. Bottom-up testing
- Automation testing
  - JUnit
  - Web automation testing: Selenium