Software Testing (UE18CS400SB)
Unit 2

Aronya Baksy

September 2021

1 Unit Testing

- Testing of individual units/components of a software to validate that each unit performs as expected
- Done during implementation phase by developers, and involves white-box testing.
- Unit: Single function/method/procedure/module/class
- Reasons for unit testing:
  1. Reduces defect costs, saves time and money
  2. Increases confidence in code maintenance
  3. Enhances reusability, reliability, speed of development
  4. Makes debugging easier
- Unit testing is of 2 types, **manual** or **automated**.

1.1 How does it work?

- Developer may write test code within the application to test that unit (this test code is commented out during final deployment)
- Developer may also copy the unit under test to a separate environment and test it in isolation. This is more rigorous and reveals unnecessary dependencies between units.
- Developer uses unit test framework to develop automated test cases. Each test consists of critera to verify the correctness of the code. Failed test cases are logged and reported

1.2 Unit Test Techniques

- **Black-Box**: Test of user interface, input and output
- **White-Box**: Testing functional behaviour of the application
- **Grey-Box**: Used to execute test suites/methods/cases and perform risk analysis

1.3 Advantages

- Unit tests allow new team members to quickly understand the unit and the project API
- Ensures that units function normally even after refactoring (a.k.a regression testing)
- Units can be tested in parallel independent of the schedule of other units

1.4 Disadvantages

- 100% branch and condition coverage is not possible, hence all errors may not be caught
- Errors caused by integration of units cannot be caught by unit tests.

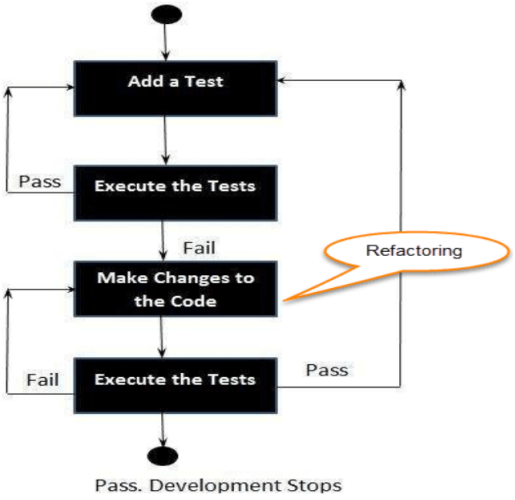

Figure 1: Test-Driven Development

1.5 Test-Driven Development

- Rules of TDD:
  1. Before writing code, write a failing test case
  2. Remove all duplicates

- Advantages of TDD:
  1. It promotes affirmative testing of the application and its specifications.
  2. Makes code simpler and clear.
  3. Reduces the documentation process at developers end.

1.6 Unit Testing Checklist

- Write unit tests for every unit
- Don't postpone unit tests, perform them immediately
- Ensure that code passes unit and integration tests before checking it in to the source control
- Use automated tools like JUnit and NUnit for unit testing
- Check test files and assets related to testing into the source control
- Use test driven development for maximum coverage
- Refactor the code only when thorough tests are available

2 White-Box Testing

- Testing based on knowledge of the software's internal workings
- Focus: strengthening quality of implementation proving design and usability
- Requires mapping between program code and actual functionality
- Why: early defect identification, confidence building, reduces complexity of further testing, addresses both functional and non-functional aspects

2.1 Objectives of White Box Testing

- Finding Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Defects due to common programming mistakes and common security holes
- Testing of each statement, object and function on an individual basis

3 Static Testing

- Analysis of code either by human or by tool (no executables involved)
- Aspects:
  1. Works according to requirements, does not miss any functionality
  2. Code is according to architecture developed eariler
  3. Error and exception handling
  4. Follows coding standards, professional best practices
- Humans can compare code more accurately with the specification, identify root causes, and special/rare conditions

3.1 Types of Static Testing

3.1.1 Desk Checking

- Informal check against specifications (no formalisms, no structure, no documentation maintained)
- Performed by the code author
- Suffices for programming errors, but may not work for incomplete/misunderstood requirements
- **Advantages**: Programmers know the code in and out hence suited to do testing, less logistics needed, reduced delays in defect detection and correction
- **Disadvantages**: Developers are narrow minded, don't like testing, and process is not scalable/reproducible

3.1.2 Code Walkthrough

- Group oriented, brings in multiple perspectives, each having their own role (Fagan Inspection)
- Highly formal, structured, participants have clear roles, diverse views
- **Roles**:
  - **Author**: writes code, fixes defects
  - **Moderator**: controls meetings
  - **Reviewers**: Read documentation before meeting, report defects
  - **Scribe**: Take notes during meetings, documents the minutes of the meeting
- Fagan inspection process: Planning, overview, individual prep, meeting, re-work, follow-up
- **Advantages**: Thorough inspection, multiple perspectives, real-world effective
- **Disadvantages**: Lots of logistics, time consuming, full code coverage is hard

3.2 Tool-driven Static Testing

- Static analysis tools uncover the following:
  1. Unreachable code
  2. Unused but declared variables, not-freed dynamically allocated memory
  3. Type errors related to unsafe type casting
  4. Non-portable code, code that does not adhere to standards
  5. Errors related to coding guidelines: naming conventions, indentation, documentation
- Cyclomatic complexity is a testing metric that determines the number of independent paths executed
- Cyclomatic complexity helps to improve code coverage, evaluate risks and ensures that all paths are covered
- Formula for cyclomatic complexity given a control flow graph G with E edges, N nodes, out of which P nodes are predicate nodes:

$$V(G) = E - N + 2 = P + 1$$

4 Structural Testing

- Entails running the executable against a set of test cases, then compare the results with the expected

4.1 Types of Structural Testing

4.1.1 Unit Functional Testing

- Initial quick checks by developer, removes "obvious" errors
- Done at programmer level, uses stubs and harnesses to replace those modules that are not yet implemented
- Unit testing tools like JUnit are used

4.1.2 Coverage Testing

- Map code to required functionality and cover as much code as possible with test cases
- Find out percentage of code covered by "instrumentation", identify the most executed critical routines in the code
- Instrumentation: profiling and logging activites that help in measuring software performance and bugs.
- Coverage = percentage of statements executed out of the total statements in the program.
- Types: statement, path, condition, function coverage
- Test for sequential instructions: generate test cases that cover the entire block of statements, test for asynchronous exceptions and multiple entry points (if present)
- Test for conditional branch: atleast 1 test case per possible execution path
- Test for loop statements: Test boundary conditions (0 loop exec, 1 loop exec, Max loop exec, Max-1 loop exec)
- Path Coverage: provides better representation than sequential coverage (e.g.: in an "if-then-else" statement having 2 test cases, 1 for the True and 1 for False part, each case has only 50% code coverage)
- Condition coverage: Refinement of path coverage, makes sure all constituent boolean expressions are covered by testing, protects against compiler optimizations
- Function coverage: more logical, easy to trace against the RTM, easy to prioritize, easier to achieve full coverage

4.1.3 Complexity Testing

- Number of independent paths in the control flow of the program gives some upper bound on number of tests needed for full coverage
- Such information is derived from cyclomatic complexity testing

5 Integration Testing

- Testing of components/units after being integrated at any level. Once integrated, the appropriate testing technique is used
- Integration testing uncovers interfacing problems between components, and functional problems
- Errors caught in integration test:
  1. Interface mismatch: parameters/args, return type, return value, other semantics mismatch
  2. Missing interfaces
  3. Protocol mismatch across interfaces
  4. Error handling across interfaces
- Integration errors may or may not manifest themselves in the form of error messages and runtime errors
- **Internal interface**: Communication across modules, used only by devs, not exposed to customer
- **External interface**: Used by third-party developers to be used by other systems/solutions, need to understand usage and purpose of providing

- Steps in integration test:
    1. Create test cases, test data and test plan
    2. test environment setup acc. to test plan
    3. Execute test cases and report results
    4. Repeat above 2 steps till required (till test adequacy criteria is reached)
    5. Repeat till all components have been integrated, use integration tools/scripts where appropriate

5.1 Top-Down Integration

- Approach: integrate high-level components first then low level.
- If some unit/component not available, "stub" required.
- Advantages: Test high level logic and data flow early, early demo of prototypes, coverage improves without changes to test code
- Disadvantages: Need for stubs, hard to observe data flow, low-level functions are developed later as they don't need testing up front which is bad practice, poor support for early release

5.2 Bottom-Up Integration

- Integrate lower level modules first, then integrate higher and higher level modules
- Stubs are required to replace unavailable modules
- Advantages: low-level utilities are tested earlier, less need for stubs, easy to observe data flow
- Disadvantages: Need for drivers, high-level logic and data flow are tested late, demo is created late, Many loose integration segments to be managed, poor support for early release

5.3 Bi-Directional Integration

- A combination of top-down and bottom up testing, using both stubs (downstream connection) and drivers (upstream connection)
- Top layers testing uses top-down testing (stubs). Bottom layers are tested using bottom-up testing (drivers)

5.4 Big-Bang System Integration

- Approach in which all software components (modules) are combined at once and make a complicated system.
- This combination of different modules is then tested as a single entity.
- Ideal when the interfaces are stable and have fewer defects.

5.5 Scenario Testing

- Uses scenarios i.e. speculative stories to help the tester work through a complicated problem or test system.
- Scenario characteristics: Story, Motivating, Credible, Complex, Easy to evaluate
- Possible Strategies to create good scenarios:
    1. Evaluate user actions and objectives, list system events
    2. Hacker mindset: try to break the system
    3. Study complaints with past systems/competitor systems
- Risks of scenario testing:
    1. Complex involving many features.
    2. Not designed for coverage of the program or for test coverage.
    3. Complicated for unstable products

5.6 Choice of integration method

- Clear requirement and design: top down
- Dynamic requirement/design/architecture: bottom up
- Stable design, changing architecture: bi-directional
- Limited, low-impact changes to architecture: system
- Combination of above: select one with analysis

S

6 System Testing

- Tests a completely integrated system to verify that its compliant with its specified requirements, in the context of a FRS and a SRS
- Why system test? Provides independent and third-party perspective, holistic and realistic tests, functional and non-funcitonal tests, build confidence and analyze risks, ensure specs are complied with
- Types of non-functional system testing:
    1. Performance testing: execution time in comparision with competitors/different versions
    2. Scalability Testing
    3. Reliability Testing
    4. Stress Testing
    5. Interoperability testing
    6. L10n and I18n

6.1 Functional Testing

- Uses black-box tests to validate the software system against the functional requirements spec (maybe manual or automated)
- Checks User Interface, APIs, Database, Security, Client/Server communication and other functionality of the Application Under Test

6.2 Non-Functional Testing

- Check non-functional aspects (performance, usability, reliability, etc) of a software application. This affects the client interaction
- Test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.