SOFTWARE ENGINEERING (CA725)

Dr. Ghanshyam S. Bopche
Assistant Professor
Dept. of Computer Applications

November 11, 2021

SOFTWARE ENGINEERING (CA725)

Syllabus

- **Unit-I**: Introductory concepts, The evolving role of software, Its characteristics, components and applications, A layered technology, The software process, Software process models, Software Development Life cycle, Software process and project metrics, Measures, Metrics and Indicators, Ethics for software engineers.
- **Unit-II**: Software Project Planning, Project planning objectives, Project estimation, Decomposition techniques, Empirical estimation models, System Engineering, Risk management, Software contract management, Procurement Management.
- **Unit-III**: Analysis and Design, Design concept and Principles, Methods for traditional, Real time of object oriented systems, Comparisons, Metrics, Quality assurance.

SOFTWARE ENGINEERING (CA725) (cont.)

- **Unit-IV**: Testing fundamentals, Test case design, White box testing, Basis path testing, Control structure testing, Black box testing, Strategies: Unit testing, integration testing, Validation Testing, System testing, Art of debugging, Metrics, Testing tools
- **Unit-V**: Formal Methods, Clean-room Software Engineering, Software reuse, Re-engineering, Reverse Engineering, Standards for industry.
- **References**:

1. Rajib Mall, "Fundamentals of Software Engineering", 4th Edition, PHI, 2014.
2. Roger S. Pressman, "Software Engineering-A practitioner's approach", 7 th Edition, McGraw Hill, 2010.
3. Ian Sommerville, "Software engineering", 10th Edition, Pearson education Asia, 2016.

SOFTWARE ENGINEERING (CA725) (cont.)

4. Pankaj Jalote, “An Integrated Approach to Software Engineering”, Springer Verlag, 1997.
5. James F Peters, Witold Pedrycz, “Software Engineering – An Engineering Approach”, John Wiley and Sons, 2000.
6. Ali Behforooz, Frederick J Hudson, “Software Engineering Fundamentals”, Oxford University Press, 2009.
7. Bob Emery , “Fundamentals of Contract and Commercial Management”,Van Haren Publishing, Zaltbommel, 2013

Software Testing

- A process of executing software with the intension to distinguish between actual results and test results.
- Primary objective: to identify and fix bugs as early as possible.
  - If the bugs migrate then they will have major impact on the software quality.
- Testing is always not exhaustive (not complete).
- When do we say "testing is successful"?
  - If defect free product is achieved i.e., when the maximum number of bugs are pro-actively identified and removed.
- Debugging: the process of diagnosing and correcting the uncovered errors.

Software Testing (cont.)

- Testing is a set of activities that can be planned in advance and conducted systematically.
- Software testing techniques: used for test case preparation.
- Software testing strategies: talks about the level of testing.

Generic characteristics of Software Testing

- Testing externally satisfies all the activities of development.
- Testing can be properly assisted with test cases.
  - The test cases can be prepared by using software testing techniques.
- Testing is better done by testers when compared with practitioners.
  - Practitioners perform at macroscopic level, whereas testers performs at microscopic level.
- Testing and debugging are two different processes.
  - Testing identifies bug.
  - Debugging corrects identified bugs.

A strategic approach to Software Testing

- A software testing strategy provides a template for software testing.
  - A set of steps into which you can place specific test case design technique and testing methods.

A strategic approach to Software Testing (cont.)

- **Generic characteristics of software testing strategy**
  - Effective technical reviews: many errors will be eliminated before testing commences.
  - Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
  - Different testing techniques are appropriate for different software engineering approaches and at different points in time.
  - Testing is conducted by the developer of the software and (for large projects) an independent test group (ITG).
  - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification vs Validation

- Verification refers to the set of tasks that ensure that software correctly implements a specific function.
    - "Are we building the product right?"
- Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
    - "Are we building the right product?"

Verification vs Validation (cont.)

- Verification and validation includes a wide array of SQA activities as follows:
  - Technical reviews,
  - Quality and configuration audits,
  - Performance monitoring,
  - Simulation,
  - Feasibility study,
  - Documentation review,
  - Database review,
  - Algorithm analysis,
  - Development testing,
  - Usability testing,
  - Qualification testing,
  - Acceptance testing, and
  - Installation testing.

Software Testing Strategy

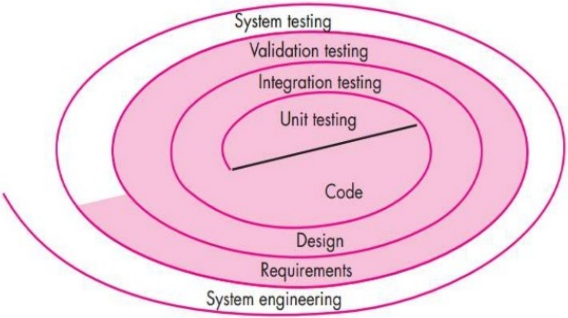

Figure: Testing strategy

Software Testing Strategy (cont.)

- System engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.
- Moving inward along the spiral, engineer come to design and finally coding.

Software Testing Strategy (cont.)

- **Levels of Testing**
  - Unit Testing
    - Testing of low-level design specifications.
    - Concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.
  - Integration Testing
    - Testing of high-level design specifications.
    - Focus is on design and the construction of software architecture.
  - Validation Testing
    - Requirements established as part of requirements modeling are validated against the software that has been constructed.
  - System Testing
    - The software and other system elements are tested as a whole.

Software Testing Steps

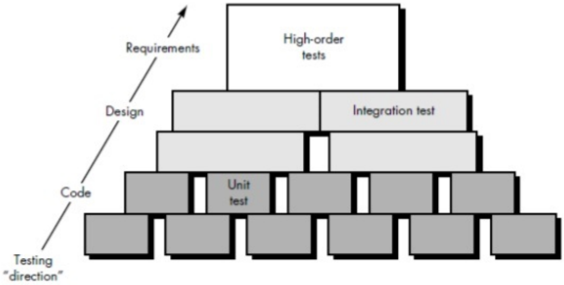

Figure: Software testing steps

- Unit testing
  - Focus on each component individually, ensuring that it function properly as a unit.
  - Makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

Software Testing Steps (cont.)

- Integration Testing
  - Components must be assembled or integrated to form the complete software package.
  - Addresses the issues associated with the dual problems of verification and program construction.
  - Test case design techniques that focus on inputs and outputs are more prevalent during integration testing, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.

Software Testing Steps (cont.)

- Higher-order Tests
  - A set of higher-order tests are conducted after the software has been integrated (constructed).
  - Validation criteria (established during requirement analysis) must be evaluated.
  - Validation testing provides final assurance that software meets all informal, functional, behavioral, and performance requirements.
  - Software, once validated, must be combined with other system elements (e.g., hardware, people, databases).
  - System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

Strategic Issues

1. Specify product requirements in a quantifiable manner long before testing commences.
   - A good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability.
2. State testing objectives effectively.
   - Test effectiveness, test coverage, mean-time-to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours should be stated within the test plan.
3. Understand the users of the software and develop a profile for each user category.
   - Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

Strategic Issues (cont.)

4. Develop a testing plan that emphasize "rapid cycle testing."
5. Build "robust" software that is designed to test itself.
   - Software should be designed in a manner that uses antibugging techniques.
   - Software should be capable of diagnosing certain classes of errors.
   - The design should accommodate automated testing and regression testing.
6. Use effective technical reviews as a filter prior to testing.
7. Conduct technical reviews to assess the test strategy and test cases themselves.
8. Develop a continuous improvement approach for the testing process.

Unit Testing

- Focuses verification effort on the smallest unit of software design - software component or module.
- Focuses on the internal processing logic and data structures within the boundaries of a component.
- The guide: the component-level design description.
- Test coverage: important control paths to uncover errors within the boundary of the module.
- The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.
- Unit testing can be conducted in parallel for multiple components.

Unit Testing (cont.)

Unit-test Considerations

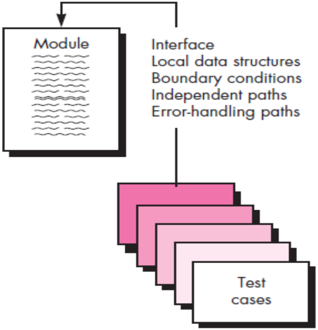

Figure: Unit test

Unit Testing (cont.)

1. Module interface testing
   - Ensure that information properly flows into and out of the program unit under test.
2. Local data structures
   - Examined to ensure component efficiency (space requirement, and performance requirement).
   - Examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithms execution.
3. Independent paths
   - All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
   - Selective testing of execution paths
     - Test cases should be designed to uncover errors due to erroneous computation, incorrect comparisons, or improper control flow.

Unit Testing (cont.)

4. Boundary Conditions
   - Software often fails at its boundaries.
   - **Examples**: errors often occur when
     - the n^{th} element of an n dimensional array is processed,
     - when the i^{th} repetition of a loop with i passes is invoked,
     - when the maximum or minimum allowable value is encountered.
   - Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
   - Test cases that exercise data structure, data flow, and data values just below, at, and just above maxima and minima are likely to uncover errors.

Unit Testing (cont.)

5. Error Handling
   - Error handling paths are tested.
   - A good design predict/expect error conditions and establishes error handling paths to reroute or cleanly terminate processing when error does occur.
   - Potential errors that could be tested when error handling is evaluated are:
     - error description in unintelligible,
     - error noted does not correspond to error encountered,
     - error condition causes system intervention prior to error handling,
     - exception-handling processing is incorrect, or
     - error description does not provide enough information to assist in the location of the cause of the error.

Unit Testing (cont.)

Unit-test Procedures

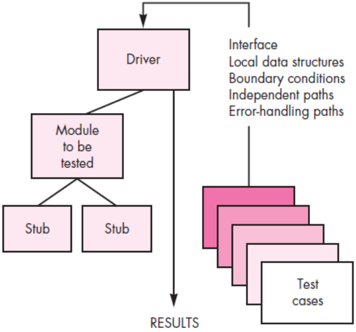

Figure: Unit test

- The design of unit test can occur before coding begins or after source code has been generated.
- A review of design information provides guidance for establishing test cases that are likely to uncover errors.

Unit Testing (cont.)

- Each test case should be coupled with expected result.
- Development of driver and/or stub software for each unit test.
- **Driver**
  - A "main program" that accepts test data, passes such data to the component (to be tested), and prints relevant results.
- **Stub**
  - serve to replace modules that are subordinate (invoked by) the component to be tested.
  - A "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and return control to the module undergoing testing.

Unit Testing (cont.)

- Drivers and stubs represent testing "overhead."
- Unit testing is simplified when a component with high cohesion is designed.
  - When only one function is addressed by a component, the number of test cases is reduces and errors can be more easily predicted and uncovered.

Integration Testing

- "If they (here, components) all work individually, why do you doubt that they will work when we put them together?"
- **Problem**: "putting components together" - interfacing.
  - Data can be lost across an interface,
  - One component can have inadvertent, adverse effect on another,
  - Sub-functions, when combined, may not produce the desired major function,
  - Individual acceptable imprecision may be magnified to unacceptable level,
  - Global data structures can present problems.
  - ...

Integration Testing (cont.)

- **Integration Testing**: a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
- Objective: take unit-tested components and build a program structure that has been dictated by design.
- **Integration approaches**
  - Non-incremental integration approach (Big bang approach)
  - Incremental integration approach
    - Top down integration approach (New trend)
    - Bottom up integration approach (Classical approach)

Integration Testing (cont.)

Non-incremental integration approach (Big bang approach)

- All components are combined in advance.
- The entire program is tested as a whole.
- Correction to encountered errors is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- Once identified errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Integration Testing (cont.)

Incremental integration approach

- The program is constructed and integrated in small increments.
- Errors are easier to isolate and correct.
- Interfaces are more likely to be tested completely.
- Systematic test approach may be applied.

Integration Testing (cont.)

Top-down integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Module subordinates to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

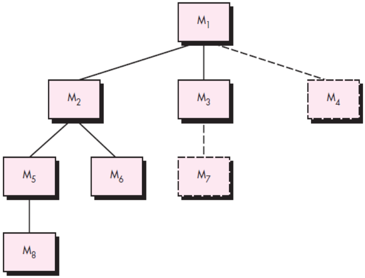

Figure: Top-down integration

Integration Testing (cont.)

- **The integration process**
  1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
  2. Depending on the integration approach (i.e., depth-first or breadth-first), subordinate stubs are replaced one at a time with actual components.
  3. Tests are conducted as each component is integrated.
  4. On completion of each set of tests, another stub is replaced with a real component.
  5. Regression testing may be conducted to ensure that new errors have not been introduced.
- The process continues form step 2 until entire program structure is built.

Integration Testing (cont.)

Bottom-up integration

- Begins construction and testing with atomic modules (i.e., components at the lowest level in the program structure).
- **Advantage**: need of stubs is eliminated
  - The functionality provided by components subordinate to a given level is always available.

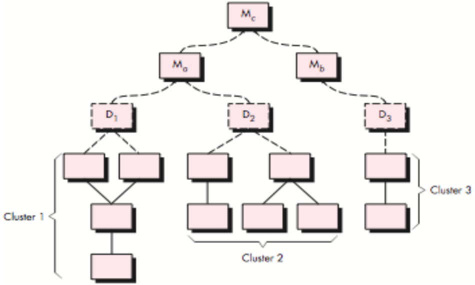

Figure: Bottom-up integration

Integration Testing (cont.)

- **Steps to implement bottom-up integration strategy**:
  1. Low-level components are combined into clusters (sometimes called build) that perform a specific software sub-function.
  2. A driver (a control program for testing) is written to coordinate test case input and output.
  3. A cluster is tested.
  4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration Testing (cont.)

Regression Testing

- Each time a new module is added as a part of integration testing, the software changes.
  - New data flow paths are established,
  - New I/O may occur, and
  - New control logic is invoked.
- The changes may cause problems with functions that previously worked flawlessly.

Integration Testing (cont.)

- **Regression testing**: the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

Validation Testing

- Focuses on user-friendly actions and user-recognizable output from the system.
- Begins at the height of integration testing (when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected).
- Validation succeeds when software functions in a manner that can be reasonably expected by the customer.
- **Basis for validation testing approach** - Software Requirements Specification (SRS) document.
    - SRS describes all user-visible attributes of the software and contains a validation criteria section.

Validation Testing (cont.)

Validation-Test Criteria

- Software validation: achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted.
- A test procedure defines specific test cases that are designed to ensure that
  - all functional requirements are satisfied,
  - all behavioral characteristics are achieved,
  - all content is accurate and properly presented,
  - all performance requirements are attained,
  - documentation is correct, and
  - usability and other requirements are met (e.g., portability, compatibility, error recovery, maintainability).

Validation Testing (cont.)

- After each validation test case has been conducted, one of two possible conditions exists:
  1. The function or performance characteristics conforms to specification and is accepted or
  2. A deviation from specification is uncovered and a deficiency list is created.

Validation Testing (cont.)

Configuration Review

- **Intent**: to ensure that all elements of the software configuration
  - have been properly developed,
  - are cataloged, and
  - have the necessary detail to bolster the support activities.

Validation Testing (cont.)

Alpha and Beta Testing

- **Series of acceptance tests**
  - When a custom software is built for one customer.
  - Enable customer to validate all requirements.
  - Conducted by the end user rather than software engineers.
- **Alpha Test**
  - Conducted at the developers site (in a controlled environment) by a representative group of end users.
  - The software is used in a natural setting with the developers "looking over the shoulder" of the users and recording errors and usage problems.

Validation Testing (cont.)

- **Beta Test**
  - Conducted at one or more end-users site.
  - No involvement of developers.
  - The customer records all problems (real or imagined) that are encountered during the beta testing and reports these to the developer at regular intervals.
  - Developer can make modifications (to the software) and then prepare for release of the software to the entire customer base.
- **Customer acceptance testing** (a variation of Beta testing)
  - Performed when custom software is delivered to a customer under contract.
  - The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer.

System Testing

- **Purpose**: exercising of complete computer-based system.
- A series of different tests were conducted.
- Software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted.

System Testing (cont.)

- **Types of system tests**
  1. **Recovery Testing**
     - A system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
     - If the recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanism, data recovery, and restart are evaluated for correctness.
     - If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

System Testing (cont.)

2. **Security Testing**
   - Attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
   - Given enough time and resources, good security testing will ultimately penetrate a system.
   - **The role of the system designer**: to make penetration cost more than the value of the information that will be obtained.

System Testing (cont.)

3. **Stress Testing**
   - Stress tests are designed to confront programs with abnormal situations.
   - Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
   - The tester attempts to break the program.
   - Sensitivity Testing (a variation of stress testing): attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

System Testing (cont.)

4. **Performance Testing**
   - Designed to test the run-time performance of software within the context of an integrated system.
   - Occurs throughout all steps in the testing process.
   - Often coupled with stress testing and usually require both hardware and software instrumentation.
5. **Deployment Testing**
   - Also called as configuration testing.
   - Exercises the software in each environment in which it is to operate.
   - Examines all installation procedures and specialized installation softwares (e.g., "installers") that will be used by customers,, and all documentation that will be used to introduce software to end users.

Debugging

- Debugging occurs as a consequence of successful testing.
- When a test case uncovers an error, debugging is a process that results in the removal of the error.
- Sometimes, the external manifestation of the error and its internal cause may have no obvious relationship to one another.
- Debugging is an Art.

The Debugging Process

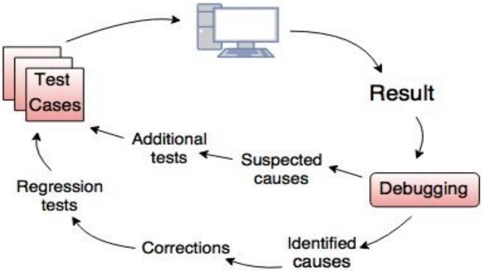

Figure: The debugging process

- The debugging process begins with the execution of a test case.
- Results are assessed and a lack of correspondence between expected and actual performance is encountered.
- In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden.

The Debugging Process (cont.)

- The debugging process attempts to match symptoms with cause, thereby leading to error correction.
- Outcomes of debugging process
    1. The cause will be found and corrected or
    2. The cause will not be found.
        - The person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Why is Debugging so Difficult?

- The symptom and the cause may be geographically remote.
  - Symptom may appear in one part of a program, while the cause may actually be located at a site that is far.
  - Highly coupled components exacerbate this situation.
- The symptoms may disappear (temporarily) when another error is corrected.
- The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
- The symptom may be caused by human error that is not easily traced.
- The symptom may be a result of timing problems, rather than processing problems.

Why is Debugging so Difficult? (cont.)

- It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
- The symptom may be intermittent.
  - Particularly common in embedded systems that couple hardware and software inextricably.
- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Debugging Strategies

1. **Brute force**
   - Most common and least efficient method for isolating the cause of a software error.
   - Apply only when all else fails.
   - Memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements.
   - In the mass of information that is produced, programmer find a clue that can lead to the cause of an error.
   - Frequently lead to wasted effort and time.

Debugging Strategies (cont.)

2. **Backtracking**
   - Common debugging approach that can be used successfully in small programs.
   - Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found.
   - As the number of source lines increases, the number of potential backward paths may become unmanageably large.

Debugging Strategies (cont.)

3. **Cause elimination**
   - Manifested by induction or deduction and introduces the concept of binary partitioning.
   - Data related to the error occurrence are organized to isolate potential causes.
   - A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis.
   - Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each.
   - If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Correcting the Error

Questions to be answered before making the "correction" that removes the cause of bug:

- Is the cause of the bug reproduced in another part of the program?
  - In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
  - Explicit consideration of the logical pattern may result in the discovery of the others.
- What "next bug" might be introduced by the fix I am about to make?
  - Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
  - If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.

Correcting the Error (cont.)

- What would we have done to prevent this bug in the first place?
  - A first step towards establishing a statistical software quality assurance approach.
  - If developer correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.