Software Quality Assurance

REVIEWING

- Types
  - **Buddy Check**
    - an informal verification technique
      - •two members work together to review the same project
  - **Circulation**
    - The project is circulated to each reviewer who adds their own comments
      - •easiest type of review to be completed when the reviewers are separated
  - **Walkthrough**
    - The project is examined by a group of peers for the purpose of finding defects
      - •usually a group of 3-4 developers
      - •majority of the program testing is conducted by people other than the author
  - **Technical**
    - Formal team evaluation of a life-cycle project
      - •led by the trained moderator, who is not the author
      - •documented and uses a defect detection process
      - •examines the suitability of the project for its intended use; compares the specifications to the standards
      - •a report is prepared with the list of issues that needs to be addressed
  - **Code Inspection**
    - A team logically steps through a project to find errors
      - •team usually made up of four people: moderator, programmer, designer, test specialist
      - •moderator duties include:
        - odistributing materials for, and scheduling, the session
        - oleading the session
        - orecording all errors found
        - oensuring that the errors are later corrected
      - •essential to have pre-meeting preparation
      - •errors are detected, but not fixed; however, the post inspection is used to ensure timely and prompt corrective action
      - •Inspection Report is prepared and shared with author

Testing

- Types
  - **Unit**
    - Process of testing the individual components, subsystems, hardware, and software
  - **Integration/Incremental**
    - Tests component interfaces and confirm requirements
      - Top-Down
      - Bottom-Up
  - **Systems**
    - – The entire system can be tested against the requirement specifications
      - Facility – ensures that the functionality in the objectives is implemented
      - Volume – subject the program to abnormally large volumes of data to process
      - Stress – subject the program to abnormally lard loads, generally concurrent processing
      - Usability – determines how well the end user can complete specified requirements
      - Security – tries to subvert the program's security measures
      - Performance – determines whether the program meets response/throughout requirements
      - Storage – ensures the program can correctly manage its storage needs
      - Configuration – checks the program performs adequately on recommended configurations
      - Compatibility/Conversion – checks if new versions are compatible with old versions
      - Installation – ensures the installation methods work on all supported platforms
      - Reliability – determines whether the program meets reliability specifications
      - Recovery – tests whether the system's recovery facilities work as designed
      - Serviceability/Maintenance – determines whether the application correctly yields data on events requiring technical support
      - Documentation – validates the accuracy of all user documentation
      - Procedure – determines the accuracy of special procedures required to maintain program
  - **Functional**
    - Process of attempting to find discrepancies between behavior and requirements; testing the end-to-end functionality of the system as a whole
      - Black Box
        - Tests Behavior
        - Code not known
        - Involved testing from user perspective
      - Gray Box
        - Little bit of everything
      - White Box
        - Internal workings is known to tester
        - Involves testing structure validation
        - Main focus on security flaws
  - **Acceptance**
    - The process of comparing the program to its initial requirements
  - **Regression**
    - Execution of tests to check that modifications do not break working code
- Principles
  - 1.A necessary part of a test case is a definition of the expected output or result.
  - 2. A programmer should avoid attempting to test his or her own program.
  - 3.A programming organization should not test its own programs.
  - 4. Any testing process should include a thorough inspection of the results of each test.
  - 5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
  - 6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
  - 7. Avoid throwaway test cases unless the program is truly a throwaway program.
  - 8. Do not plan a testing effort under the tacit assumption that no errors will be found.
  - 9. The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
  - 10. Testing is an extremely creative and intellectually challenging task.
- Verification
  - Compare input of System Design Phase to Program Design Phase
- Design Techniques
  - **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.
  - **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.

DEBUGGING

- Types
  - **Brute-force**
    - Most common debugging scheme and is the most mentally taxing
      - •Automated Tools – sets breakpoints that causes suspension in the program so the user can examine the current state
      - •Storage Dump – shows the program state at only one instant
      - •Scatter Print Statements – requires user to make the changes to the program, which can lead to masking errors
  - **Induction**
    - Moving from the particulars of a situation to the whole (i.e., follow the clues)
  - **Deduction**
    - Uses the process of elimination and refinement to arrive at a conclusion
  - **Backtracking**
    - Work through the incorrect results, moving backwards, until you find the logic error
- Principles
  - Error-Locating
    - •Think
    - •If you reach an impasse, sleep on it
    - •If you reach an impasse, describe the problem to someone else
    - •Use debugging tools only as a second resort
    - •Avoid experimentation—Use it only as a last resort
  - Error-Repairing
    - •Where there is one bug, there is likely to be another
    - •Fix the error, not the symptom
    - •The probability of the fix being correct is not 100%
    - •The probability of the fix being correct drops as the size of the program increase
    - •An error correction could create a new error
    - •The process of error repair should put on temporarily back into the design phase
    - •Change the source code, not the object code
  - Error Analysis
    - •Where was the error made?
    - •Who made the error?
    - •What was done incorrectly?
    - •How could the error have been prevented?
    - •Why wasn't the error detected earlier?
    - •How could the error have been detected earlier?
- Lifecycle
  - 1.Identify the bug.
  - 2.Report and document features.
  - 3.Triage all reports by defining each.
  - 4.Communicate details.
  - 5.Fix the bug during a sprint.

Correspondence within lifecycle

- Development
  - Requirements → Acceptance Test
  - Objectives → System Test
  - External Specification → Function Test
  - System Design, Program Structure Design → Integration Test
  - Module Interface Specifications → Module Test
  - Code → Installation Test

Methodologies

- Software Development Life Cycle (SDLC)
  - 1.Planning and Requirements Analysis
  - 2.Defining Requirements
  - 3.Designing the Product Architecture
  - 4.Building or Developing the Product
  - 5.Testing the Product
  - 6.Deployment into the Market
- Waterfall Model
  - Pros
    - -simple to use
    - -easy to manage due to the rigidity of the model
    - -phases are processed and completed one at a time
    - -works well for smaller projects
    - -clearly defined stages
    - -understood milestones
    - -easy to arrange tasks
    - -process and results are well documented
  - Cons
    - -no working software is produced until late
    - -not a good model for complex and object-oriented projects
    - - a poor model for long and ongoing projects
    - -not suitable for projects when requirements are subject to change
    - -difficult to measure progress within stages
- Iterative Model
  - Pros
    - -working functionality can be developed early in the life cycle
    - -results are obtained early and periodically
    - -parallel development can be planned
    - -progress can be measured
    - -less costly to change the scope/requirements
    - -testing and debugging during smaller iteration is easy
    - -easier to manage risk
    - -risk analysis is better
    - -supports changing requirements
    - -with each increment, operational product is delivered
  - Cons
    - -more resources are required
    - -more management attention is required
    - -design issues may arise because not all requirements are gathered in the beginning
    - -not suitable for smaller projects
    - -end of project may not be known
    - -highly skilled resources are required for risk analysis
    - -project progress is highly dependent upon the risk analysis phase
- Spiral Model
  - Pros
    - -changing requirements can be accommodated
    - -allows extensive use of prototypes
    - -requirements can be captured accurately
    - -users see the system early
    - -development can be divided into smaller parts, allowing for better risk management
  - Cons
    - -management is more complex
    - -end of the project may not be known early
    - -note suitable for small or low risk projects
    - -process is complex
    - -spiral may go on indefinitely
    - -large number of intermediate stages requires excessive documentation
- V-Model
  - Pros
    - -highly-disciplined model and phases are completed one at a time
    - -works well for smaller projects
    - -simple and easy to use
    - -easy to manage due to rigidity of the model
  - Cons
    - -high risk and uncertainty
    - -not good for complex and object-oriented projects
    - -poor model for long and ongoing projects
    - -not suitable for projects when requirements are subject to change
    - -difficult to change functionality once application is in testing phase
- Agile Model
  - Pros
    - -very realistic approach to software development
    - -promotes teamwork and cross training
    - -functionality can be developed rapidly and demonstrated
    - -resource requirements are minimum
    - -suitable for fixed or changing requirements
    - -minimal rules, documentation easily employed
    - -little or no planning required
    - -easy to manage
    - -gives flexibility to developers
  - Cons
    - -not suitable for handling complex dependencies
    - -an agile leader is a must
    - -depends heavily on customer interaction
    - -minimum documentation generated
    - -transfer of technology to new team members may be challenging

Frameworks

- **Quality Assurance**
  - is the systematic process used to determine whether a product meets specifications.
- **Capability Maturity Model Integration (CMMI)**
  - a process level improvement training and appraisal program. It can be used as a guide to improve process involvement through identifying five maturity levels.
    - Levels
      - Level 1: Initial
        - Unpredictable, poorly controlled, and reactive
      - Level 2: Managed
        - Characterized for projects and is often reactive
      - Level 3: Defined
        - Characterized for the organization and is proactive
      - Level 4: Quantitatively Managed
        - Measured and controlled
      - Level 5: Optimizing
        - Focuses on process improvements