SOFTWARE TESTING

PROCESS

Test and analysis activities within a software process; The quality process; Planning and monitoring; Quality Goals; Dependability Properties; Planning and Monitoring; Quality Goals; Dependability properties; Analysis; Testing; Improving the process; Organizational factors; Integration and component based software testing; Overview; Integration testing strategies; Testing components and assemblies; System, Acceptance and Regression Testing; Overview; System Testing; Acceptance Testing; Usability; Regression testing; Regression test selection techniques; Test case prioritization and selective execution
8.1 Test and Analysis Activities within a Software Process

Learning Objectives

- Understand the role of quality is the development process
  - Build an overall picture of the quality process
- Identify the main characteristics of a quality process
  - Visibility
  - anticipation of activities
  - feedback

8.1.1 Overview

- Test and Analysis are complex activities that must be suitably planned and monitored
- A good quality process obeys some basic principles:
  - visibility
  - early activities
  - feedback
- Aims at
  - reducing occurrences of faults
  - assessing the product dependability before delivery
  - improving the process

8.2 Software Qualities and Process

- Qualities cannot be added after development
  - Quality results from a set of inter-dependent activities
  - Analysis and testing are crucial but far from sufficient.
- Testing is not a phase, but a lifestyle
  - Testing and analysis activities occur from early in requirements engineering through delivery and subsequent evolution.
  - Quality depends on every part of the software process
- An essential feature of software processes is that software test and analysis is thoroughly integrated and not an afterthought

8.3 The Quality Process
• Quality process: set of activities and responsibilities
  - focused primarily on ensuring adequate dependability
  - concerned with project schedule or with product usability
• The quality process provides a framework for
  - selecting and arranging activities
  - considering interactions and trade-offs with other important goals.

8.3.1 Interactions and tradeoffs

Example: high dependability vs. time to market

• Mass market products:
  - better to achieve a reasonably high degree of dependability on a tight schedule than to achieve ultra-high dependability on a much longer schedule

• Critical medical devices:
  - better to achieve ultra-high dependability on a much longer schedule than a reasonably high degree of dependability on a tight schedule

8.3.2 Properties of the Quality Process

• Completeness: Appropriate activities are planned to detect each important class of faults.
• Timeliness: Faults are detected at a point of high leverage (as early as possible)
• Cost-effectiveness: Activities are chosen depending on cost and effectiveness
  - cost must be considered over the whole development cycle and product life
  - the dominant factor is usually the cost of repeating an activity through many change cycles.

8.4 Planning and Monitoring

• The quality process
  - Balances several activities across the whole development process
  - Selects and arranges them to be as cost-effective as possible
  - Improves early visibility
• Quality goals can be achieved only through careful planning
• Planning is integral to the quality process
8.4.1 Process Visibility

- A process is visible to the extent that one can answer the question
  - How does our progress compare to our plan?
  - Example: Are we on schedule? How far ahead or behind?
- The quality process has not achieved adequate visibility if one cannot gain strong confidence in the quality of the software system before it reaches final testing
  - quality activities are usually placed as early as possible
    - design test cases at the earliest opportunity (not “just in time”)
    - uses analysis techniques on software artifacts produced before actual code.
  - motivates the use of “proxy” measures
    - Ex: the number of faults in design or code is not a true measure of reliability, but we may count faults discovered in design inspections as an early indicator of potential quality problems

8.5 Quality Goals

- Process qualities (visibility,....)
- Product qualities
  - internal qualities (maintainability,....)
  - external qualities
    - usefulness qualities:
      - usability, performance, security, portability, interoperability
    - dependability
      - correctness, reliability, safety, robustness

8.6 Dependability Properties

- Correctness:
  - A program is correct if it is consistent with its specification
    - seldom practical for non-trivial systems
- Reliability:
  - likelihood of correct function for some “unit” of behavior
    - relative to a specification and usage profile
    - statistical approximation to correctness (100% reliable = correct)
- Safety:
- preventing hazards

- Robustness
  - acceptable (degraded) behavior under extreme conditions

8.7 Analysis

- Analysis includes
  - manual inspection techniques
  - automated analyses
- can be applied at any development stage
- particularly well suited at the early stages of specifications an design

8.7.1 Inspection

- Can be applied to essentially any document
  - Requirements statements
  - Architectural and detailed design documents
  - Test plans and test cases
  - Program source code
- May also have secondary benefits
  - Spreading good practices
  - Instilling shared standards of quality.
- Takes a considerable amount of time
- Re-inspecting a changed component can be expensive
- Used primarily
  - Where other techniques are inapplicable
  - Where other techniques do not provide sufficient coverage

8.7.2 Automatic Static Analysis

- Are selected when available
  - substituting machine cycles for human effort makes them particularly cost-effective.
- More limited in applicability
- can be applied to some formal representations of requirements models
- not to natural language documents

8.8 Testing

- Executed late in development
- Start as early as possible
- Early test generation has several advantages
  - Tests generated independently from code, when the specifications are fresh in the mind of analysts
  - The generation of test cases may highlight inconsistencies and incompleteness of the corresponding specifications
  - tests may be used as compendium of the specifications by the programmers
8.9 Improving the Process

- Long lasting errors are common
- It is important to structure the process for
  - Identifying the most critical persistent faults
  - tracking them to frequent errors
  - adjusting the development and quality processes to eliminate errors
- Feedback mechanisms are the main ingredient of the quality process for identifying and removing errors

8.10 Organizational factors

- Different teams for development and quality?
  - separate development and quality teams is common in large organizations
  - indistinguishable roles is postulated by some methodologies (extreme programming)
- Different roles for development and quality?
  - test designer is a specific role in many organizations
  - mobility of people and roles by rotating engineers over development and testing tasks among different projects is a possible option

8.10.1 Example of Allocation of Responsibilities

- Allocating tasks and responsibilities is a complex job: we can allocate
  - Unit testing
    - to the development team (requires detailed knowledge of the code)
    - but the quality team may control the results (structural coverage)
  - Integration, system and acceptance testing
    - to the quality team
    - but the development team may produce scaffolding and oracles
  - Inspection and walk-through
    - to mixed teams
  - Regression testing
    - to quality and maintenance teams
  - Process improvement related activities
    - to external specialists interacting with all teams
8.11 Integration and Component-based Software Testing

8.11.1 Learning objectives

- Understand the purpose of integration testing
  - Distinguish typical integration faults from faults that should be eliminated in unit testing
  - Understand the nature of integration faults and how to prevent as well as detect them

- Understand strategies for ordering construction and testing
  - Approaches to incremental assembly and testing to reduce effort and control risk

- Understand special challenges and approaches for testing component-based systems

8.11.2 Overview

A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software. As important, a software testing strategy provides a road map for the software developer, the quality assurance organization, and the customer- a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote the creativity and customization that are necessary to adequately test all large software-based systems. At the same time, the strategy must be rigid enough to promote reasonable planning and management tracking as the project progresses.

Considering the process from a procedural point of view testing within the context of software engineering is a series of four steps that are implemented sequentially.
The steps are shown in Figure 8.1. Initially, tests focus on each module individually, assuring that it functions as a unit; hence the name **unit testing**. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection. Next, modules must be assembled or integrated to form the complete software package. **Integration testing** addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), sets of high-order test are conducted. Validation criteria (established during requirements analysis) must be tested. **Validation testing** provides final assurance that software meets all functional, behavioral, and performance requirements. Black-box testing techniques are used exclusively during validation.

**Figure 8.1: Software Testing Steps**
8.11.3 What is integration testing?

Integration testing is a systematic technique for constructing the program structure while conducting tests to uncover errors associated with interfacing. Once all the individual units have been tested there is a need to test how they were put together to ensure no data is lost across interface, one module does not have an adverse impact on another and a function is not performed correctly. Integration testing is a systematic approach that produces the program structure while at the same time producing tests to identify errors associated with interfacing.

8.12 Integration test strategies

There is often a tendency to attempt to construct the program using a “big bang” approaches. All modules are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

8.12.1 Top-Down integration

Top-down integration is an incremental approach to the production of program structure. Modules are integrated by moving downwards through the control hierarchy, starting with the main control module. Modules subordinate to the main control module are included into the structure in either a depth-first or breadth-first manner. Relating to the figure below depth-first integration would integrate the modules on a major control path of the structure. Selection of a major path is arbitrary and relies on application particular features. For instance, selecting the left-hand path, modules M1, M2, M5 would be integrated first. Next M8 or M6 would be integrated. Then the central and right-hand control paths are produced. Breath-first integration includes all modules directly subordinate at each level, moving across the structure horizontally. From the figure modules M2, M3 and M4 would be integrated first. The next control level, M5, M6 etc., follows.
The integration process is performed in a series of five stages:

1. The main control module is used as a test driver and stubs are substituted for all modules directly subordinate to the main control module.
2. Depending on the integration technique chosen, subordinate stubs are replaced one at a time with actual modules.
3. Tests are conducted as each module is integrated.
4. On the completion of each group of tests, another stub is replaced with the real module.
5. Regression testing may be performed to ensure that new errors have been introduced.

### 8.12.2 Bottom-up Integration

Bottom-up integration testing begins testing with the modules at the lowest level (atomic modules). As modules are integrated bottom up, processing required for modules subordinates to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level modules are combined into clusters that perform a particular software subfunction.
2. A driver is written to coordinate test cases input and output.
3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

8.12.3 **Sandwich testing** - a best compromise between two methods. It uses a top-down strategy for upper levels of the program structure, coupled with a bottom-up strategy for subordinate levels.

8.12.4 **Choosing a Strategy**

- Functional strategies require more planning
- Structural strategies (bottom up, top down, sandwich) are simpler
- But thread and critical modules testing provide better process visibility, especially in complex systems
- Possible to combine Top-down, bottom-up, or sandwich are reasonable for relatively small components and subsystems
- Combinations of thread and critical modules integration testing are often preferred for larger subsystems

8.12.5 **Comments on Integration Testing**

There has been much discussion on the advantages and disadvantages of bottom-up and top-down integration testing. Typically a disadvantage is one is an advantage of the other approach. The major disadvantage of top-down approaches is the need for stubs and the difficulties that are linked with them. Problems linked with stubs may be offset by the advantage of testing major control functions early. The major drawback of bottom-up integration is that the program does not exist until the last module is included.

8.12.6 **Testing Components and Assemblies**

A Component is a

- **Reusable** unit of deployment and composition
  - Deployed and integrated multiple times
  - Integrated by different teams (usually)
    - Component producer is distinct from component user
• Characterized by an *interface* or *contract*
  o Describes access points, parameters, and all functional and non-functional behavior and conditions for using the component
  o No other access (e.g., source code) is usually available
• Often larger grain than objects or packages
  o Example: A complete database system may be a component

8.12.7 Components — Related Concepts
• Framework
  • Skeleton or micro-architecture of an application
  • May be packaged and reused as a component, with “hooks” or “slots” in the interface contract
• Design patterns
  • Logical design fragments
  • Frameworks often implement patterns, but patterns are not frameworks. Frameworks are concrete, patterns are abstract
• Component-based system
  • A system composed primarily by assembling components, often “Commercial off-the-shelf” (COTS) components
  • Usually includes application-specific “glue code”

8.12.8 Component Interface Contracts
• Application programming interface (API) is distinct from implementation
  - Example: DOM interface for XML is distinct from many possible implementations, from different sources
• Interface includes *everything* that must be known to use the component
  - More than just method signatures, exceptions, etc
  - May include non-functional characteristics like performance, capacity, security
  - May include dependence on other components

8.12.9 Challenges in Testing Components
• The component builder’s challenge:
  - Impossible to know all the ways a component may be used
- Difficult to recognize and specify all potentially important properties and dependencies

- The component user’s challenge:
  - No visibility “inside” the component
  - Often difficult to judge suitability for a particular use and context

### 8.12.10 Testing a Component: Producer View

- First: Thorough unit and subsystem testing
  - Includes thorough functional testing based on application program interface (API)
  - Rule of thumb: Reusable component requires at least twice the effort in design, implementation, and testing as a subsystem constructed for a single use (often more)

- Second: Thorough acceptance testing
  - Based on scenarios of expected use
  - Includes stress and capacity testing
    - Find and document the limits of applicability

### 8.12.11 Testing a Component: User View

- Not primarily to find faults in the component
- Major question: Is the component suitable for *this* application?
  - Primary risk is not fitting the application context:
    - Unanticipated dependence or interactions with environment
    - Performance or capacity limits
    - Missing functionality, misunderstood API
  - Risk high when using component for first time
- Reducing risk: Trial integration early
  - Often worthwhile to build driver to test model scenarios, long before actual integration

### 8.13 System Testing

Software once validated must be combined with other system elements (e.g., hardware, people, and databases). *System testing* verifies the tall elements mesh properly and that overall system function/performance is achieved.
Ultimately, software is included with other system components and a set of system validation and integration tests are performed. Steps performed during software design and testing can greatly improve the probability of successful software integration in the larger system. System testing is a series of different tests whose main aim is to fully exercise the computer-based system. Although each test has a different role, all work should verify that all system elements have been properly integrated and form allocated functions. Below we consider various system tests for computer-based systems.

8.13.1 Security Testing

Any computer-based system that manages sensitive information or produces operations that can improperly harm individuals is a target for improper or illegal penetration. Security testing tries to verify that protection approaches built into a system will protect it from improper penetration. During security testing, the tester plays the role of the individual who wants to enter the system. The tester may try to get passwords through external clerical approaches; may attack the system with customized software, purposely produce errors and hope to find the key to system entry. The role of the designer is to make entry to the system more expensive than that which can be gained.

8.13.2 Stress Testing

Stress testing executes a system in the demands resources in abnormal quantity, frequently or volume. A variation of stress testing is an approach called sensitivity testing in some situation a very small range of data contained with the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.

8.14 Regression Testing

Each time a new model 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. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test, strategy regression testing is the re-execution of subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
Regression testing is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors.

How is regression test conducted?

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture playback tools.

Capture-playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite contains three different classes of test cases.

1. A representative sample of tests that will exercise all software functions.
2. Additional tests that focus on software functions that are likely to be affected by the change.
3. Tests that focus on software components that have been changed.

Note:

It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Selection of an integration strategy depends upon software characteristics and some time project schedule. In general, a combined approach that uses a top-down strategy for upper levels of the program structure, coupled with bottom-up strategy for subordinate levels may be best compromise.

Regression tests should follow on critical module function.

WHAT IS CRITICAL MODULE?

A critical module has one or more of the following characteristics.

- Addresses several software requirements
- Has a high level of control
- Is a complex or error-prone
- Has a definite performance requirement.
- Strategy:
Start with riskiest modules

Risk assessment is necessary first step

May include technical risks (is X feasible?), process risks (is schedule for X realistic?), other risks

May resemble thread or sandwich process in tactics for flexible build order

E.g., constructing parts of one module to test functionality in another

Key point is risk-oriented process

Integration testing as a risk-reduction activity, designed to deliver any bad news as early as possible

8.15 Summary

- Integration testing focuses on interactions
  - Must be built on foundation of thorough unit testing
  - Integration faults often traceable to incomplete or misunderstood interface specifications
    - Prefer prevention to detection, and make detection easier by imposing design constraints
- Strategies tied to project *build order*
  - Order construction, integration, and testing to reduce cost or risk
- Reusable components require special care
  - For component builder, and for component user