Software Testing

2024S

Summary

1. Bugs

What is a bug?

- Error – a human action that produces an incorrect result.
- Fault – a flaw in the program that if encountered during execution, produces a failure.
- Failure – deviation of the system from its expected result.

Failure reproduction:

- Reach – the input to the software must cause the faulty statement to be executed.
- Infection – the faulty statement must produce an incorrect result, causing an incorrect internal state for the system.
- Propagation – the incorrect internal state must propagate to the output so that the result of the fault is observable.

Defect reports:

- Title and description – high-level summary of the problem.
- Detailed steps and expected behavior.
- Priority – severity and frequency of the issue.
- Context – relevant configuration (OS, version...) and hardware.
- Pictures and examples.

Defects:

- delivered defects/kLoC = introduced defects - removed defects
- Typically, 60 defects/kLoC.

2. Test Processes

What is software testing?

- *Testing is the process of executing a program or system with the intent of finding errors.*
- Goals of software testing:
  - Finding defects.
  - Assess the quality.
  - Increase confidence.
  - Prevent defects.
- Non-goals:
  - Defect localization and removal (debugging).
  - Prove the absence of bugs.

Verification vs Validation:

- Verification – *Did we build the product right?*
  - Focus on current implementation vs specifications.
- Validation – *Did we build the right product?*
  - Focus on current implementation vs customer requirements.

Testing processes:

- Testing processes are embedded within the development process.
- Test Management Process:
  - Test strategy and planning.
  - Test monitoring and control.
  - Test completion.
- Dynamic Test Process:
  - Test design and implementation.
  - Test execution.
  - Test report.

Quality assurance:

- Ongoing activity across all phases of a development process.
- Static Quality Assurance – reviews and inspections; focus on early defect detection.
- Dynamic Quality Assurance:
  - Defect detection in executable code documents via testing – sequential process via e.g. V-Model.
  - Coupling testing and implementation in e.g. SCRUM (TDD).

Test-Driven Development (TDD):

- Testing in traditional sequential processes:
  - Long cycles in large engineering projects.
  - Finding bugs becomes difficult.
- Testing in agile processes:
  - Short cycles.
  - Test cases written before implementation.
- TDD Steps:
  - **Think** - select next feature to implement and specify test cases.
  - **Red** – all tests must fail.
  - **Green** – stepwise implementation and test execution until all tests pass.
  - **Refactor** – modification and optimization of classes without functional changes; test cases **may not** fail.

3. Test Design Techniques

The test design problem:

- Problem: it is impossible to test all combinations of inputs and paths, except for trivial cases.
- Solution: sampling approach – choose a few powerful tests that represent the rest.
- Black-box, White-box, and Experience-based techniques.

Test design techniques:

1. Specification-Based (Black-Box):
    - **Equivalence Partitioning:**
        - Divide the range of inputs into groups of equivalent tests to avoid unnecessary testing – 1 test per partition is sufficient.
        - **Equivalence Partition/Class** – *a portion of an input or output domain for which the behavior of the system is assumed to be the same, based on the specification.*
        - Applicable for systematic data (input and output) coverage at all levels of testing.
        - Limitations: weak selection of representative values (compare with boundary value analysis below).
    - **Boundary Value Analysis:**
        - Selects representative values from an equivalence partition (programs are more likely to fail at boundaries).
        - 3 values – boundary, outside (invalid), inside (valid); if no "real" boundary, only 2 values.
        - Applicable with equivalence partitioning at all levels of testing, high probability of defect detection.
        - Limitations: requires inputs that can be partitioned to an ordered set and identifiable boundaries; requires creative testers (e.g. boundaries of string?).
    - **Combinatorial Testing:**
        - Testing several variables in combination; pairwise testing often used to reduce the number of combinations.
        - In case of non-uniformity of input distribution, pairwise testing can fail (highly probable combinations get too little attention), e.g. options dialog of MS Word.
    - **State Transition Testing:**
        - State diagrams show the various states that a system can get into and the transitions that occur between the states.
        - To test, identify states and transitions, draw a transition tree, and derive test cases (all paths from root to leaves).
        - For each state, add an invalid transition as well.
        - A good test heuristic is to test all transitions at least once.

    - Applicable for systems that exhibit different behavior depending on their state (used in embedded systems and technical automation).
    - Limitations: the **state-explosion problem** (too many states).
2. Structure-Based (White-Box):
    - **Statement Testing/Coverage:**
        - Each statement is executed at least once.
        - Control-flow graphs are the foundation for control flow testing.
        - Shortcoming: branch logic, e.g. if-statements.
    - **Decision/Branch Testing/Coverage:**
        - Each branch must be executed at least once.
        - The entire expression is considered as a single predicate.
        - Decision coverage guarantees statement coverage, but not vice versa.
        - Limitations: ignores branches within Boolean expressions due to short-circuit operators (compiler optimizations).
    - **Condition Testing/Coverage:**
        - Like decision/branch, but every Boolean subexpression is evaluated to true and to false at least once.
        - Multiple Condition Coverage (MCC):
            - Requires a test for each possible combination (2^n test cases)
        - Modified Condition/Decision Coverage (MC/DC):
            - Requires testing only those conditions that independently affect the outcome.
    - **Data Flow Testing (Path Coverage):**
        - Def-Use – follow paths where a value is defined (Def) and used (Use).
    - **Mutation Testing/Coverage:**
        - Error seeding – artificial faults are introduced into the program (e.g. changing constants, modifying conditions).
        - Mutation testing:
            - Test suite is run on original program and all tests pass.
            - After each mutation, the test suite is run again. If at least one test case fails, mutant is killed, else it survives (i.e. change remains undetected).
3. Experience-Based:
    - **Error-guessing:**
        - Tests are derived from the tester's skill, intuition, and experience with similar technologies.
        - A structured approach is to enumerate a list of possible errors and design tests that attack those errors; lists can be built based on experience, or from available defect and failure data.
    - **Intuitive Testing:**
        - Useful to identify special tests not easily captured by formal techniques.
    - **Exploratory Testing:**
        - Interactively gaining experience about the product while testing and designing new tests based on that experience.
        - Organized in 60-90 min sessions; goals defined as *charters*.

4. Test Automation

Motivation:

- Automated and frequent test case execution.
- Reduce boring and repetitive activities.
- Reproducibility of tests.
- Automated generation of large data samples.
- Nevertheless: high additional effort.

Traceability:

- *Identifying and verifying the history across development processes*.
- Traceability classes:
    - **Traceability over time** – relationship between versions/releases.
    - **Traceability across development phases (horizontal)** – relationship between artifacts across phases (e.g. requirements -> implementation -> test cases).
    - **Traceability across artifacts (vertical)** – relationship between architecture levels (e.g. system -> subsystem -> component -> class).
- Can be used to see which code has to be checked when requirements change, as well as which requirements are affected by failing code.

Automation:

- Test automation – the use of software to perform test activities.
- Test execution automation – the use of software to control the execution of tests.

Scenarios:

- Regression testing – (re-)testing a previously tested program following modification to ensure that no defects have been introduced in the unchanged area of the software.
- Smoke testing – a subset of all planned/defined test cases ascertaining that the most crucial/main functions of the system work; no finer details are tested.
- Load testing – performance testing conducted to evaluate the behavior of a component with increasing load, as well as to determine what load can be handled.

GUI-Testing:

- The process of testing a GUI OR testing a system via its GUI.
- Testing types:
    - **Manual testing:**
        - Easy, cheap, flexible.
        - No auto regression testing or results log.
    - **Capture-Replay:**
        - Inputs during manual testing are recorded and generate test scripts that can be executed later (e.g. Selenium IDE).
        - Easy, flexible, auto regression testing.
        - Expensive first execution, fragile tests break easily.
        - Problems:
            - Interactions are recorded for the lowest-level UI elements (e.g. clicking a button label ≠ clicking a button).
            - Intended user interaction cannot always be reconstructed.
            - Some events may not be captured, e.g. touch gestures.
            - Some UI elements may not be recognized, e.g. cells in a grid.
            - Timing is relevant – C/R requires delays and synchronization.
            - Requires a working system – no TDD!
    - **Programmatic** (e.g. Selenium API)**:**
        - Test scripts are source code, auto regression testing.
        - Requires maintenance of test scripts.

Test Maintenance:

- **Corrective maintenance** – reactive modification of tests to correct problems (e.g. test fails but system is correct).
- **Adaptive maintenance** – modifications of the test to keep up with changes in the system.
- **Perfective maintenance** – modifications to improve quality of tests.
- **Preventive maintenance** – modifications to detect and correct problems in the tests.

Page Object Pattern:

- Pattern for web testing.
- Page Object:
    - Abstraction over low-level detail.
    - API to access the elements of a Web page.
    - Better maintainability.
    - More readable and understandable tests
    - Less code duplications
    - Separation between UI code and test code.

Behavior-Driven Development:

- BDD:
    - Concrete examples of system behavior.
    - Requirements are turned into automated tests that guide the developer, verify and document the feature.
- Specification by example:
    - Using examples to define the specification (the behavior of the system).
- Acceptance-Test-Driven Development (ATDD):
    - Using acceptance tests to define the specification (often a synonym for Specification by example).

5. Automating Test Automation

Random Testing:

- A black-box testing technique where test cases are randomly selected to match an operational profile.
- **Monkey testing:**
  - Random testing applied to GUI testing.
  - Random selection from a large range of inputs – e.g. randomly pushing buttons.
- **Fuzzy testing:**
  - Testing by providing invalid, unexpected, or random input data.
  - Commonly used for security testing.
- **Feedback-directed Random Testing** – e.g. Randoop.
- Problems:
  - Generation problem:
    - How to sufficiently cover a large input space (e.g. guessing the correct user/password with random inputs)?
    - Randomly generated data follows a homogenous distribution.
  - Test minimization problem:
    - Large number of redundant test cases.
    - Generated tests have much noise (failing test cases may have a large number of irrelevant steps – difficult to debug failures).
    - False positives.
  - Oracle problem:
    - Easy to generate random data, but hard to determine if the outcome passes or fails.

Test Oracle:

- *A source to determine expected results for comparison with the actual result of the execution in order to decide pass or fail of the test.*
- Differential testing – comparing the output of 2 different systems.
- Delta testing – comparing the output of 2 versions of the same system.
- Assertions vs Junit Asserts.

Model-Based Testing:

- *A model is an abstraction of the system it describes.*
- *Model-based testing is defined as automatable derivation of concrete test cases from abstract formal models, and their execution.*
- The output of the model (expected) is compared to the output of the system (actual).

- Techniques:
  - Random generation (e.g. random walk).
  - Markov chains.
  - Graph search algorithms.
  - Model checking – only paths that reach a certain state or transition.
- Test generation:
  - On-line – test cases are generated and executed at once; testing non-deterministic systems.
  - Off-line – test cases are generated from a model; can be stored and maintained like conventional test cases.
- Pros:
  - Early bug detection (modeling is early in the development).
  - Better evolution support (models are easier to update than individual tests).
  - Reduced costs (auto-generated test cases).
- Cons:
  - High effort shifted from testing to modeling.
  - Advanced modeling skills required of the testers.
  - Modeling is an abstraction – important details may be missed.
  - Difficult to define an appropriate oracle.