Software Quality Control and Testing

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

March 9, 2023

Outline

Outline

Topics in Last Class

1. Overview of requirement engineering
2. Agile vs. traditional (plan & document)
3. An agile approach of requirement analysis
   - Behavior-Driven Development (BDD)
   - UI sketches and storyboards

Outline

What is Software Quality?

- Conforms to requirements – via validation.
  - Did we build the right thing? – Is this what the customer wants, and is the specification correct?
- Fit to use – via verification.
  - Did we build the thing right? – Did we meet the specification?

- Quality assurance refers to all activities designed to measure and improve quality in a product, including the whole process, training, and preparation of the team.
- Quality control usually refers to activities designed to verify the quality of the product, detect faults or defects, and ensure that the defects are fixed prior to release.

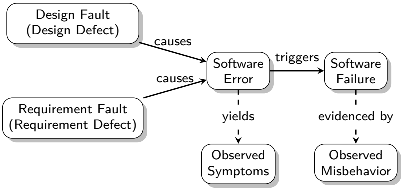
“Error-Detection” Techniques

- Testing: executing program in a controlled environment and “verifying/validating” output.
- Inspections and reviews.
- Static analysis detects “error-prone conditions.”
- Formal methods (proving software correct).
  - Limited use in safety-critical and security-critical applications

Faults and Failures

- Error: a mistake made by a programmer or software engineer that caused the fault, which in turn may cause a failure
- Fault (defect, bug): condition that may cause a failure in the system
- Failure (problem): inability of system to perform a function according to its spec due to some fault
- Fault or failure/problem severity (based on consequences)
- Fault or failure/problem priority (based on importance of developing a fix, which is in turn based on severity)

Faults and Failures



(a) Relations of software defects (faults), errors, and failures.

```
#include <stdlib.h>
enum { BUFFER_SIZE = 32 };

int f(void) {
    char *text_buffer = (char *)malloc(
        BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }

    return 0;
}
```

(b) A code snippet showing a memory allocation defect **seicertcstandard**

Figure: Relationship of defect, error, and failure and an example of defective code

Outline

Testing

- Activity performed for:
  - Evaluating product quality
  - Improving products by identifying defects and having them fixed prior to software release
- Dynamic (running-program) verification of program's behavior on a finite set of test cases selected from execution domain.
- Exhaustive testing infeasible
- Divide and conquer – perform different tests at different levels of the software Upper level doesn't redo tests of lower level

Limitation of Testing

"Testing can never show the ABSENCE of errors in software, only their PRESENCE" — by Edsger Dijkstra

- Testing can NOT prove product works 100%—even though we use testing to demonstrate that parts of the software works
- Exhaustive testing infeasible

Aspects of Testing

- Why test?
- What is tested?
- Who tests?
- How (are test cases designed)?

Why Test?

- Quality control measure (developers)
  - Evaluating product quality
  - Improving products by identifying defects and having them fixed prior to software release
- Acceptance (customers)
  - System or Acceptance Test: integrated program meets users' specifications
  - We have mentioned these before. In what context?
- Conformance (standards, laws, etc.)
- Configuration (user vs. developers.)
- Performance, stress, security, etc.

What is Tested (Type of Tests)

- User interface testing
- Integration/system testing: interfaces between units have consistent assumptions, communicate correctly
- Module or Functional Test: within individual units
- Unit testing : single method does what was expected

Progression of Testing

- Unit tests $\rightarrow$ Functional tests $\rightarrow$ Component tests $\rightarrow$ System/regression tests

Who Tests Software?

- Developers
- Testers/Requirement Analysts
- Users

In agile model, we don't usually have specialized testers, but some organizations still have such teams

Testing Methods

- Glass-box (aka white-box) testing – Tester understands the details of system to be tested. When, for instance, the developer is testing the code.
- Black-box testing – Tester does NOT use (or understand) the details of system to be tested.

How to Test?

How (test cases designed)?

- Intuition
- Specification based (black box)
- Code based (white box)
- Existing cases (regression)

White-box vs. Black-box

- Goal of testing – to "break" it
- Testing goal runs counter to the goals of software development activities
- Hard for a developer to get in the proper mindset

Argument for white-box testing

- Knowing what's inside an interface (or class) will enable you to test it more thoroughly

Argument against white-box testing:

- You'll have the same blind spots in testing the class that you had in writing it

Example Testing Methods

- Equivalence Class Partitioning
- Boundary Value Analysis
- Path Analysis
- Combinations of Conditions

Data Partition Testing

Aim to test using examples different groups on inputs, e.g., Equivalence Class Partitioning

- Divide the input into several groups, deemed “equivalent” for purposes of finding errors.
- Pick one “representative” for each class used for testing.
- Equivalence classes determined by req./design specifications and some intuition

Table: Example. Pick “larger” of two integers and ...

Class	Representative
Low	-5
0 - 12	6
13 - 19	15
20 - 25	30
36 - 120	60
High	160

Lessen duplication and complete coverage

Boundary Value Analysis

- A black-box technique
- Experiences show that “boundaries” are error-prone.
- Do equivalence-class partitioning; add test cases for boundaries (at boundary, outside, inside).
    - Reduced cases: consider boundary as falling between numbers.
    - If boundary is at 12: normal: 11, 12, 13; reduced: 12, 13 (boundary 12 and 13)
- Large number of cases ($\sim$3 per boundary).
- Good for “ordinal values. ”

Path Analysis/Control Flow Testing

- A white-box technique
- Two tasks
  1. Aim to test different flows through code (high path coverage)
     - Happy path and sad path
  2. Analyze number of paths in program.
  3. Decide which ones to test.
- Decreasing coverage:
  - Logical paths
  - Independent paths
  - Branch coverage
  - Statement coverage

Combinations of Conditions

- For functions of several related variables.
- To fully test, we need all possible combinations (of equivalence classes).
- How to reduce testing:
  - Coverage analysis.
  - Assess "important" (e.g., main functionalities) cases.
  - Test all pairs of relations (but not all combinations).

Guideline testing

Use previous experiences on the types of errors that typically occur

Outline

Unit/Functional Testing

- Testing individual methods or classes
- Usually done by the programmer.
- Test each unit as it is developed (small chunks).
- Keep test cases/results around (Use JUnit or ...).
- Allows for regression testing.
- Facilitates refactoring.
- Tests become documentation !!

Assertion

- A boolean expression that should evaluate to true if the program is in a correct state. If it evaluates to false it throws an exception

```
x = 1;
assert(x > 0);
x++;
assert(x > 0);
```

Unit/Functional Testing: Testing Methods

- Tests are calls to methods with different input parameters
- Assert expectations on the return-values or side effects of method calls
- Aim for high coverage
  - Almost always white box, and
  - Almost always performed by developer

Unit/Functional Testing: Example – Step 0

Developer wrote the code.

```
private int clickClearCount;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    clickClearCount = 0;
}

public void onClickCounter(View v) {
  clickClearCount++;
}
```

Let's test whether the `clickClearCount` variable has the right value after a few clicks.

Unit/Functional Testing: Example – Step 1

Sometimes we need to add methods to aid testing, e.g.,

```
public int getClickClearCount()
{
  return clickClearCount;
}
```

For this example, need a way to access the private variable so it's value can be tested

Unit/Functional Testing: Example – Step 2

Now write the unit test ...

```
@Test
public void testButtonCounter()
{
   MainActivity mainAct = new MainActivity();
   mainAct.onClickCounter( null );
   mainAct.onClickCounter( null );
   mainAct.onClickCounter( null );
   assertEquals(mainAct.getClickCount(),3);
}
```

However, we need a Button object to pass to onClick

- get one via a `findViewById()` call
- or get a fake one (talk about this soon)

Test-Driven Development

- Write unit test cases BEFORE the code!
- Test cases "are"/"become" requirements.
- Forces development in small steps.
    1. Write test case and code.
    2. Verify (it fails or runs).
    3. Modify code so it succeeds.
    4. Rerun test case, previous tests.
    5. Refactor until (success and satisfaction).
- We will discuss more about this ...

Some Testing Concepts

- Popular metric of testing
- Amount of code or execution paths covered by tests
- Several variants of this metric exist

Common Test Coverage Levels

- S0 (method coverage) – Is every method executed at least once by the test suite?
- S1 (call coverage or entry/exit coverage) – Has each method been called from every place it can be called?
- C0 (statement coverage) – Is every statement of the source code executed at least once by the test suite?
- C1 (branch coverage) – Has each branch been taken in each direction at least once?
- C2 (path coverage) – Has every possible route through the code been executed?

Sample Code to Test

```
public class MyClass {
    public void foo(boolean x, boolean y, boolean z) {
       if (x)
           if (y && z) bar(0);
           else
               bar(1);
   }
  public boolean bar(x) {
      return x;
   }
}
```

Examples of Test Coverage

- Satisfying S0 requiring calling foo and bar at least once each in the tests
- Satisfying S1 requiring calling bar from both line 4 and line 6 in the test suites
- Counting both branches of a conditional as a single statement, satisfying C0 requiring calling foo at least once with x true, and at least once with y false
- Satisfying C1 requiring calling foo at least once with x true, and with x false, and with `y && z` true and false.
- Satisfying C2 requiring calling foo with all 8 combinations of values of x, y, and z

Modified Condition/Decision Coverage (MCDC)

Combines a subset of the above levels

- Each point of entry and exit in the program have been invoked at least once
- Every decision in the code has taken all possible outcomes at least once
- Each condition in a decision has been shown to independently affect that decision's outcome

Achieving Test Coverage

- 100% of C0 coverage is not unreasonable.
- Achieving C1 coverage requires careful construction of tests.
- C2 is the most difficult of all, and the additional value of 100

Types of test execution approaches

- Regression Testing: automatically rerun old tests, so changes don't break what used to work
- Continuous Integration Testing: continuous regression testing vs. later phases

Outline

Inspections and Reviews

- Review: any process involving human testers reading and understanding a document and then analyzing it with the purpose of detecting errors
- Walkthrough: author explaining document to team of people
- Software inspection: detailed reviews of work in progress, following Fagan's method

Software Inspections

Steps:

1. Planning
2. Overview
3. Preparation
4. Inspection
5. Rework
6. Follow-up

Software Inspections

- Focused on finding defects
- Output: list of defects
- Team of:
  - 3–6 people
  - Author included
  - People working on related efforts
  - Moderator, reader, scribe

Inspections vs. Testing

Inspections

- Partially cost-effective.
- Can be applied to intermediate artifacts.
- Catch defects early.
- Helps disseminate knowledge about project and best practices.

Testing

- Finds errors cheaper, but correcting them is expensive.
- Can only be applied to code.
- Catch defects late (after implementation).
- Necessary to gauge quality.

Outline

Formal Methods

- Mathematical techniques used to prove that a program works.
- Used for requirements/design/algorithm specification.
- Prove that implementation conforms to spec.
- Pre and post conditions
- Problems:
  - Require math training.
  - Not applicable to all programs.
  - Only verification, not validation.
  - Not applicable to all aspects of program (e.g., UI and maintainability).

Outline

Static Analysis

- Examination of static structures of design/code for detecting error-prone conditions (cohesion — coupling).
- Automatic program tools are more useful.
- Can be applied to:
  - Intermediate documents (but in formal model)
  - Source code
  - Executable files
- Output needs to be checked by programmer.

Outline

Summary

Verification & Validation

Testing

Unit testing

Strategies in writing tests

How about acceptance tests for our user stories?

Outline

"Engineering Software as a Service" by Armando Fox and David Patterson (2nd Edition)
"Essentials of Software Engineering" by Frank Tsui, Orlando Karam, and Barbara Bernal(4th Edition) (Section 7.3.5)