Software Engineering II - Lecture 8 Study Notes

1. Review of Testing

Development Testing

- **Unit Testing**
  - Target: Individual program unit (class/method)
  - Focus: Testing the functionality of an object or a method
- **Integration Testing**
  - Target: Combination of units making up a component
  - Focus: Testing component interfaces (e.g., parameters, message passing)
- **System Testing**
  - Target: Entire system
  - Focus: Testing both functional and non-functional requirements (e.g., performance, usability)

Regression Testing

- Ensures application functions as expected after changes
- Goals:
  - Prevent code changes from breaking existing functionality
  - Find defects at an early stage

2. Test Coverage

- **Test Coverage**: Measures how much of the code is tested
- Applications for testing coverage include:
  - Checking the completeness of testing
  - Identifying areas of code not tested

3. Measurement Techniques

- **Test Suite Example**:
  - Test suite A: `P=5, Q=10`
  - Test suite B: `P=5, Q=10` and `P=51, Q=4`

Code Coverage Types

- **White Box Testing**:
  - **Covering Code**: How much of the code is executed by the test cases?
  - Goals: Target statement/branch/path coverage
- **Black Box Testing**:
  - **Covering Input Space**: Testing with a representative set of input values

Coverage Measurement

- **Instrumentation**:

    - **Compile-Time**: Adding logging statements like AspectJ
    - **Run-Time**: Monitoring with special VMs like VisualVM or Javassist

Java Bytecode Instrumentation

- Modifying bytecode at runtime using Instrumentation APIs (e.g., Javassist)

Control Flow Graph (CFG)

- Represents all possible execution paths in a program
- Components:
    - Entry/Exit points
    - Control nodes (branches)
    - Basic blocks (straight-line code with no branches)

4. Testing Strategies and Boundaries

Unit Testing Strategies

- **Equivalence Classes**: Group inputs with common characteristics; test one per class
- **Boundary Testing**: Test near the boundaries of input range
- **Negative Testing**: Test with incorrect/out-of-range inputs
- **Input Validation Testing**: Ensure erroneous input is appropriately handled

5. Fault Localization and Tools

Fault Localization

- Identifying buggy lines based on failing test cases
- **Coverage-Based Fault Localization**:
    - **Tarantula**: Ranks statements by suspiciousness based on test case success/failure

Code Coverage Tools

- **JaCoCo**: Free tool with support for branch, line, method coverage, and cyclomatic complexity
- **Coverage.py**: For Python programs
- **Cobertura** & **Emma**: Older tools, not actively maintained

6. Additional Resources

- Fuzz Testing - Synopsys
- The Fuzzing Book
- Mutation Testing - Guru99
- Mutation Testing - SoftwareTestingHelp

Class Notes

Comp 4350 Software Engineering II Lecture 8

Dr. Shaowei Wang

Review of testing

Development testing

- Unit testing
  - Target: particular program unit (class/method)
  - Focus: testing functionality of an object or a method
- Integrating testing
  - Target: combination of units that make up a component
  - Focus: interface (e.g., parameters, message passing) a component exposes to other components
- System testing
  - Target: the whole system
  - Focus: functional and non-functional requirements (e.g., performance, usability)

Regression testing

- A software testing practice ensuring an application functions as expected after code changes
- Goal: Ensure changes don't break existing functional and non-functional requirements
- Find defects in the early stage

Who tests the tests?

- How can we measure test cases?

Outline

- Test coverage
- Applications for testing coverage
- Mutation testing

Measure the test cases

```
Read P
Read Q
IF P+Q > 10 THEN
    Print “Large”
ENDIF
If P > 50 THEN
    // bug is here
    …
ENDIF

Test suite A
Test case 1 P= 5, Q=10
```

```
Test suite B
Test case 1 P =5, Q=10
Test case 2 P =51, Q=4
```

Covering code (for white box testing)

- How much code is covered by test cases?
- Objective: Cover all possible statements, branches, or paths (99%)

How to measure the coverage?

1. Run test case, collect execution information (e.g., executed statements)
2. Program analysis – count branches/paths in the program
3. Calculate coverage

```
Void Main(){
    …
    O.Func1();

    O.Func2();
    …
}
Void Main(){
    …
    O.Func1();
    log.info(“O.Func1 is executed”);
    O.Func2();
     log.info(“O.Func2 is executed”);
    …
}
```

- Instrumentation

Collecting execution info - instrumentation

- Instrument at compile time (e.g., logging, AspectJ)
- Instrument or monitor at runtime (e.g., Javassist, VisualVM)
- Java Bytecode Instrumentation

Control flow graph

- Representation of all paths traversed through a program during execution
- Entry, Exit, Control nodes, Basic blocks

Exercise: generate the test case input to cover all paths?

- Input: P=1, Q=1
- Input: P=60, Q=-60
- Input: P=60, Q=10

- Input: P=6, Q=6

The ugly truth about testing: Large number of possible paths

- Number of Paths = 2n
- If each test takes 10-3 seconds, with n=36, we need more time than has passed since the big bang!

How to measure coverage?

- In practice, measure statement (line) coverage

- # of executed statements / # total statements

- Input: P=60, Q=10

Exercise – calculate the statement coverage

```
Read P
Read Q
IF P+Q > 10 THEN
    Print “Large”
ELSE
    Print “Small”
ENDIF
If P > 50 THEN
    Print “P Large”
ENDIF

Input: P=60, Q=10
Print “Small”
```

Code coverage tools (for Java)

- JaCoCo – free code coverage tool
- Coverage.py – tool for measuring code coverage of Python programs
- Cobertura – used in earlier versions of Cassandra
- Emma – not actively maintained anymore

Know about JaCoCo - Features

- Free code coverage tool for Java
- Coverage analysis of branches, lines, methods, and cyclomatic complexity
- Several report formats (HTML, XML, CSV)
- Integration with various tools (Ant Tasks, Maven plug-in, Gradle)

Coverage.py

Overhead for instrumentation

- Affects performance

We will cover two types of coverage

- Covering code (white box)
- Covering input space (black box)

Covering input space (for black-box testing)

- Input domain for testing is infinite
- Testing is about choosing finite sets of values from the input domain

Unit testing strategies

- Equivalence classes: Identify groups of inputs with common characteristics
- Run one test per equivalence class

Examples of equivalence classes

- Valid input: month number (1-12)
- Valid input: ten strings representing a type of fuel
- File upload with different sizes: Small, Medium, Large, Oversize

Rule of thumb

- Boundary testing
- Negative testing
- Input validation testing

Application

- Application based on code coverage: Spectrum-based fault localization

Fault Localization

- Given a set of test cases, where are the buggy lines?

Fault Localization

- Given a set of test cases, where are the buggy lines?

```
int mid(x, y, z) {
}
```

Coverage-based Fault localization

- Intuition: Statements primarily executed by failed test cases are more likely to be faulty
- Solution: Ranking based on suspiciousness (Tarantula)

An Example of Using Tarantula

- Only running on 1/5 of the pass cases.
- Running on all the pass cases.

Other ranking formulas

- cef: Number of failing test cases executed by a program entity
- cnf: Number of failing test cases not executed by a program entity
- cep: Number of passing test cases executed by a program entity
- cnp: Number of passing test cases not executed by a program entity

Exercise

- Calculate ranking formulas

Materials

- What is Fuzzing?
- Fuzzing Book
- Mutation Testing
- Software Testing Help - Mutation Testing