Software Testing

Motivation
- People are not perfect
  - We make errors in design and code
- Goal of testing: given some code, uncover as many errors as possible
- Important and expensive activity
  - Not unusual to spend 30-40% of total project effort on testing
  - For critical systems (e.g., flight control): cost can be several times the cost of all other activities combined

A Way of Thinking
- Design and coding are creative
- Testing is destructive
  - The primary goal is to "break" the software
- Very often the same person does both coding and testing
  - Need "split personality": when you start testing, become paranoid and malicious
  - Surprisingly hard to do: people don't like finding out that they made mistakes

Testing Objectives
- Testing is a process of executing software with the intent of finding errors
- Good testing has a high probability of finding as-yet-undiscovered errors
- Successful testing discovers unknown errors
  - If did not find any errors, need to ask whether our testing approach is good

Basic Definitions
- Test case: specifies
  - Inputs and pre-test state of the software
  - Expected results (outputs and state)
- Black-box testing: ignores the internal logic of the software
  - Given this input, was the output correct?
- White-box testing: uses knowledge of the internal structure of the software
  - e.g., write tests to "cover" internal paths

Testing Approaches
- In this course: small sample of approaches for testing
- White-box testing
  - Control-flow-based testing
  - Data-flow-based testing
- Black-box testing
  - Equivalence partitioning
Control-flow-based Testing

- Traditional form of white-box testing
- **Step 1**: From the source code, create a graph describing the flow of control
  - Called the control flow graph
  - The graph is created (extracted from the source code) manually or automatically
- **Step 2**: Design test cases to cover certain elements of this graph
  - Nodes, edges, paths

Example of a Control Flow Graph

```plaintext
s := 0;
d := 0;
while (x < y) {
    x := x + 3;
y := y + 2;
    if (x + y < 100)
        s := s + x + y;
    else
        d := d + x - y;
}
```

Elements of a Control Flow Graph

- Three kinds of nodes:
  - **Statement nodes**: single-entry-single-exit sequences of statements
  - **Predicate (decision) nodes**: conditions for branching
  - **Auxiliary nodes**: (optional) for easier understanding (e.g. "merge points" for IF)
- Edges: possible flow of control

IF-THEN, IF-THEN-ELSE, SWITCH

```plaintext
if (c) then ...
else ...
switch (c)
    case 1: ...
    case 2: ...
endswitch
```

Example

```plaintext
switch (position)
    case CASHIER:
        if (empl_yrs > 5)
            bonus := 1;
        else
            bonus := 0.7;
        break;
    case MANAGER:
        bonus := 1.5;
        if (retiring_soon)
            bonus := 1.2 * bonus
        break;
    case ...
endswitch
```

Mapping for Loops

```plaintext
while (c) {
    ...
}
```

Note: other loops (e.g. FOR, DO-WHILE, ...) are mapped similarly

Mini-assignment: figure out how this is done
Statement Coverage

- Given the control flow graph, define a “coverage target” and write test cases to achieve it
- Traditional target: statement coverage
  - Test cases that cover all nodes
  - Code that has not been executed during testing is more likely to contain errors
  - Often this is the “low-probability” code

Example

- Suppose that we write and execute two test cases
- Test case #1: follows path 1-2-exit
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Problem: node 6 is never executed, so we don’t have 100% statement coverage

Branch Coverage

- Target: write test cases that cover all branches of predicate nodes
  - True and false branches of each IF
  - The two branches corresponding to the condition of a loop
  - All alternatives in a SWITCH
  - In modern languages, branch coverage implies statement coverage

Example

- Same example as before: two test cases
  - Path 1-2-exit
  - Path 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- Problem: the “false” branch of 4 is never taken - don’t have 100% branch coverage

Branch Coverage

- Statement coverage does not imply branch coverage
- Example: if (c) then s;
  - By executing only with c=true, we will achieve statement coverage, but not branch coverage
  - Motivation: experience shows that many errors occur in “decision making”
  - Plus, it subsumes statement coverage

Achieving Branch Coverage

- Branch coverage: a necessary minimum
  - Pick a set of start-to-end paths that cover all branches, and write test cases to execute these paths
- Basic strategy
  - Add a new path that covers at least one edge that is not covered by the current paths
  - Sometimes the set of paths chosen with this strategy is called the “basis set”
- Good example in Pressman, Sect. 17.4.3
Data-flow-based Testing

- Test connections between variable definitions ("write") and variable uses ("read")
- Variation of the control flow graph
  - A node represents a single statement, not a single-entry-single-exit chain of statements
- Set $\text{DEF}(n)$ contains variables that are defined at node $n$ (i.e., they are written)
- Set $\text{USE}(n)$: variables that are read

**Example**

```plaintext
s:=0;
x:=0;
while (x<y) {
    x:=x+3;
y:=y+2;
    if (x+y<10)
        s:=s+x+y;
    else
        s:=s+x-y;
}
```

**Assume $y$ is already initialized**

Reaching Definitions

A definition of $x$ at $n_1$ reaches $n_2$ if and only if there is a path between $n_1$ and $n_2$ that does not contain a definition of $x$.

Def-Use Pairs

- A def-use (DU) pair for variable $x$ is a pair of nodes $(n_1,n_2)$ such that
  - $x$ is in $\text{DEF}(n_1)$
  - the definition of $x$ at $n_1$ reaches $n_2$
  - $x$ is in $\text{USE}(n_2)$

- i.e., the value that is assigned to $x$ at $n_1$ is used at $n_2$
- Since the definition reaches $n_2$, the value is not "killed" along some path $n_1...n_2$

Data-flow-based Testing

- Identify all DU pairs and construct test cases that cover these pairs
- Variations with different "strength"

- **All-DU-paths**: for each DU pair $(n_1,n_2)$ for $x$, exercise all possible paths $n_1...n_2$ that are clear of definitions of $x$

- **All-uses**: for each DU pair $(n_1,n_2)$ for $x$, exercise at least one path $n_1...n_2$ that is clear of definitions of $x$
### Data-flow-based Testing

- **All-definitions**: for each definition, cover at least one DU pair for that definition
  - i.e., if x is defined at n1, execute at least one path n1...n2 such that x is in USE(n2) and the path is clear of definitions of x
- **Motivation**: see the effects of using the values produced by computations
  - Focuses on the data, while control-flow-based testing focuses on the control

### Black-box Testing

- Unlike white-box testing: don’t use any knowledge about the internals of the code
- Test cases are designed based on specifications
  - Example: search for a value in an array
    - Postcondition: return value is the index of some occurrence of the value, or -1 if the value does not occur in the array

### Equivalence Partitioning

- Consider input/output domains and partition them into equivalence classes
  - For different values from the same class, the software should behave equivalently
  - Test values from each class
    - Example for input range 2..5: “less than 2”, “between 2 and 5”, and “greater than 5”
  - Testing with values from different classes is more likely to find errors than testing with values from the same class

### Equivalence Classes

- **Examples**
  - Input x in range [a..b]: three classes “x<a”, “a<=x<=b”, “b<x”
  - boolean: classes “true” and “false”
  - Some classes may represent invalid input
  - Choosing test values
    - Choose a typical value in the middle of the class(es) that represent valid input
    - Choose values at the boundaries of classes
      - e.g. for [a..b], use a-1, a, a+1, b-1, b, b+1

### Example

- Spec says that the code accepts between 4 and 24 inputs; each is a 3-digit integer
  - One partition: number of inputs
    - Classes “x<4”, “4<=x<=24”, “24<x”
    - Chosen values: 3,4,5,14,23,24,25
  - Another partition: integer values
    - Classes: “x<100”, “100<=x<=999”, “999<x”
    - Chosen values: 99,100,101,500,998,999,1000

### Another Example

- Similarly for the output: exercise boundary values
  - Spec: the output is between 3 and 6 integers, each in the range 1000-2500
  - Try to design inputs that produce
    - 3 outputs with value 1000
    - 3 outputs with value 2500
    - 6 outputs with value 1000
    - 6 outputs with value 2500
**Example: Searching**

- Search for a value in an array
  - Return: index of some occurrence of the value, or -1 if the value does not occur
- One partition: size of the array
  - Programmer errors are often made for size 1: a separate equivalence class
  - Classes: "empty array", "array with one element", "array with many elements"
- Another partition: location of the value
  - "first element", "last element", "middle element", "not found"

<table>
<thead>
<tr>
<th>Array</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>[7]</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>[7]</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>[1,6,4,7,2]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[1,6,4,7,2]</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>[1,6,4,7,2]</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>[1,6,4,7,2]</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Testing Strategies**

- Many issues beyond what we discussed
  - Who does the testing?
  - Which techniques should we use and when?
- No universal strategies
  - Principles that have been useful in practice
    - e.g. the notions of unit testing and integration testing

**Scope and Focus**

- **Unit testing**: scope = individual component
  - Focus: component correctness
  - White-box and black-box techniques
- **Integration testing**: scope = set of interacting components
  - Focus: correctness of component interactions
  - Mostly black-box, some white-box
- **System testing**: scope = entire system
  - Focus: overall system correctness
  - Only black-box techniques

**Unit Testing**

- Scope: one component from the design
  - Often corresponds to the notion of "compilation unit" from the prog. language
- Responsibility of the developer
- Both white-box and black-box techniques
- May be necessary to create stubs: "fake" code that replaces called modules
  - If not yet implemented, or not yet tested
**Basic Strategy for Unit Testing**

- Create black-box tests
  - Based on the specification of the unit
- Evaluate the tests using white-box techniques (test adequacy criteria)
  - How well did the tests cover statements, branches, paths, DU-pairs, etc.?
  - Many possible criteria; at the very least need 100% branch coverage
- Create more tests for the inadequacies: e.g. to increase coverage of DU-pairs

**Test-First Programming**

- Modern practices: the importance of unit testing during development
- Example: test-first programming
  - Before you start writing any code, first write the tests for this code
  - Write a little test code, write the corresponding unit code, make sure it passes the tests, and then repeat
  - E.g., used in Extreme Programming: any program feature without a corresponding test simply doesn’t exist

**Advantages of Test-First Programming**

- Developers don’t “skip” unit testing
- Satisfying for the programmer: feeling of accomplishment when the tests pass
- Helps clarify interface and behavior before programming
  - To write tests for something, first you need to understand it well
- Software evolution
  - After changing existing code, rerun the tests to gain confidence (regression testing)

**System Testing**

- Goal: find whether the program does what the customer expects to see
  - Black-box
  - From requirements analysis, there should be validation criteria
    - How are the developers and the customers going to agree that the software is OK?
  - Many issues: functionality, performance, usability, portability, etc.

**System Testing (cont)**

- For multiple customers: two phases
  - **Alpha testing**: at the vendor’s site by a few customers
  - **Beta testing**: distributed to many end-users
    - Users run it in their own environment
    - Sometimes done by thousands of users

**System Testing (cont)**

- Initial system testing is done by the software producer
- Eventually need testing by the customers
  - Customers are great testers
- If the software is built for a single customer: series of acceptance tests
  - Deploy in the customer environment and have end-users run it
Stress Testing

- Form of system testing: behavior under very heavy load
  - e.g. normally there are 1-2 interrupts per second; what if there are 10 interrupts?
  - e.g. what if data sets that are an order of magnitude larger than normal?
  - e.g. what if our server gets 10 times more client requests than normally expected?

- Find how the system deals with overload
  - Reason 1: determine failure behavior
    - If the load goes above the intended, how "gracefully" does the system fail?
  - Reason 2: expose bugs that only occur under heavy loads
    - Especially for OS, middleware, servers, etc.
    - e.g. memory leaks, incorrect resource allocation and scheduling, race conditions

Regression Testing

- Rerun old tests to see if anything was "broken" by a change
  - Changes: bug fixes, module integration, maintenance enhancements, etc.
- Need test automation tools
  - Load tests, execute them, check correctness
  - Everything has to be completely automatic
- Could happen at any time: during initial development or after deployment

Testing of Object-Oriented Software

- Widespread popularity in the last decade
- Initial hopes: it would be easier to test OO software than procedural software
  - Soon became clear that this is not true
- Some of the older testing techniques are still useful
- New testing techniques are designed specifically for OO software

One Difference: Unit Testing

- Traditional view of "unit": a procedure
- In OO: a method is similar to a procedure
  - But a method is part of a class, and is tightly coupled with other methods and fields in the class
- The smallest testable unit is a class
  - It doesn’t make sense to test a method as a separate entity
- Unit testing in OO = class testing
Class Testing

- Traditional black-box and white-box techniques still apply
  - E.g. testing with boundary values
  - Inside each method: at least 100% branch coverage; also, DU-pairs inside a method
- Extension: DU pairs that cross method boundaries
  - Example: inside method m1, field f is assigned a value; inside method m2, this value is read

Example: Inter-method DU Pairs

```java
class A {
    private int index;
    public void m1() {
        index = ...;
        m2();
    }
    private void m2() {
        x = index; ...
    }
    public void m3() {
        z = index; ...
    }
}
test 1: call m1, which calls m2 and reads the value of index

test 2: call m1, and then call m3
```

Possible Test Suite

```java
public class MainDriver {
    public static void main(String[] args) {
        A a = new A();
        ...
        a.m1();
        a.m3();
        ...
    }
}
```

Note: need to ensure that the actual execution exercises definition-free paths for the two DU pairs

Polymorphism

- A variable may refer to objects of different classes at different points of time
  - e.g., in Java, if D is a subclass of C, a variable of type C may refer to instances of C or to instances of D
- An operation may be defined in several classes and may have different implementations in each class
  - Method overriding

Example of Polymorphism

- Example: class A with subclasses B and C
  - class A { ... void m() (...) ... }
  - class B extends A { ... }
  - class C extends A { ... void m() (...) ... }
- Suppose inside class X there is call a.m(), where variable a is of type A
  - Could potentially send message m() to an instance of A, instance of B, or instance of C
  - The invoked method could be A.m or C.m

Testing of Polymorphism

- During class testing of X: "drive" call site a.m() through all possible bindings
- All-receiver-classes: execute with at least one receiver of class A, at least one receiver of class B, and at least one receiver of class C
- All-invoked-methods: need to execute with receivers that cover A.m and C.m
  - i.e. (A or B receiver) and (C receiver)
State-based Testing

- **Natural representation with finite-state machines**
  - States correspond to certain values of the attributes
  - Transitions correspond to methods
- **FSM can be used as basis for testing**
  - e.g. “drive” the class through all transitions, and verify the response and the resulting state

Example: Stack

- **States**
  - Initial: before creation
  - Empty: number of elements = 0
  - Holding: number of elements >0, but less than the max capacity
  - Full: number elements = max
  - Final: after destruction
- **Transitions: starting state, ending state, action that triggers the transition, and possibly some response to the action**

Examples of Transitions

- **Initial -> Empty**: action = “create”
  - e.g. “s = new Stack()” in Java
- **Empty -> Holding**: action = “add”
- **Empty -> Full**: action = “add”
  - if max_capacity = 1
- **Empty -> Final**: action = “destroy”
  - e.g. destructor call in C++, garbage collection in Java
- **Holding -> Empty**: action = “delete”

Finite State Machine for a Stack

FSM-based Testing

- Each **valid transition** should be tested
  - Verify the resulting state using a state inspector that has access to the internals of the class
- Each **invalid transition** should be tested to ensure that it is rejected and the state does not change
  - e.g. Full -> Full is not allowed: we should call add on a full stack

Inheritance

- People thought that inheritance will reduce the need for testing
  - Claim 1: “If we have a well-tested superclass, we can reuse its code (in subclasses, through inheritance) without retesting inherited code”
  - Claim 2: “A good-quality test suite used for a superclass will also be good for a subclass”
- Both claims are wrong
Problems with Inheritance

- Incorrect initialization of superclass attributes by the subclass
- Missing overriding methods
  - Typical example: equals and clone
- Direct access to superclass fields from the subclass code
  - Can create subtle side effects that break unsuspecting superclass methods
- A subclass violates an invariant from the superclass, or creates an invalid state

Testing of Inheritance

- Principle: inherited methods should be retested in the context of a subclass
- Example 1: if we change some method m is a superclass, we need to retest m inside all subclasses that inherit it
- Example 2: if we add or change a subclass, we need to retest all methods inherited from a superclass in the context of the new/changed subclass

Example

class A {
    int x; // invariant: x > 100
    void m() { // correctness depends on
        // the invariant ... } ... }
class B extends A {
    void m2() { x = 1; ... } ... }

- If m2 has a bug and breaks the invariant, m is incorrect in the context of B, even though it is correct in A
- Therefore m should be retested on B objects

Another Example

class A {
    void m() { ... m2(); ... }
    void m2 { ... } ... }
class B extends A {
    void m2() { ... } ... }

- If inside B we override a method from A, this indirectly affects other methods inherited from A
  - e.g. m now calls B.m2, not A.m2: so, we cannot be sure that m is correct anymore and we need to retest it inside B

Testing of Inheritance (cont)

- Test cases for a method m defined in class X are not necessarily good for retesting m in subclasses of X
  - e.g. if m calls m2 in A, and then some subclass overrides m2, we have a completely new interaction
  - Still, it is essential to run all superclass tests on a subclass
    - Goal: check behavioral conformance of the subclass w.r.t. the superclass (LSP)

Testing of Interacting Classes

- Until now we only talked about testing of individual classes
- Class testing is not sufficient
  - OO design: several classes collaborate to implement the desired functionality
- A variety of methods for interaction testing
  - We will only consider testing based on UML interaction diagrams
UML Interaction Diagrams for Testing

- UML interaction diagrams: sequences of messages among a set of objects
  - There may be several diagrams showing different variations of the interaction
- Basic idea: run tests that cover all diagrams, and all messages and conditions inside each diagram
  - If a diagram does not have conditions and iteration, it contains only one path

Coverage Requirements

- Run enough tests to cover all messages and conditions
  - test with 0 loop iterations and >=1 iterations
  - test with hasStock=true and hasStock=false
  - test with needsReorder=true and needsReorder=false
- To cover each one: pick a particular path in the diagram and “drive” the objects through that path