Testing

CPSC 315 – Programming Studio

Testing

- Testing helps find that errors exist
  - Debugging finds and fixes them
- Systematic attempt to break a program that is working
- Unlike all other parts of software development, whose goal is to avoid errors
- Can never prove absence of errors
- Testing alone does not improve quality

Types of Testing

- Unit testing
  - Testing of a single class, routine, program
  - Usually single programmer
  - Testing in isolation from system
- Component testing
  - Testing of a class, package, program
  - Usually small team of programmers
- Integration testing
  - Combined test of two or more classes, packages, components, or subsystems

Types of Testing (continued)

- Regression testing
  - Repetition of previously tested cases to find new errors introduced
- System testing
  - Executing software in final configuration, including integration with all other systems and hardware
  - Security, performance, resource loss, timing issues

Other Testing

- Usually by specialized test personnel
  - User tests
  - Performance tests
  - Configuration tests
  - Usability tests
  - Etc.
- We're interested in developer tests

Writing Test Cases First

- Helps identify errors more quickly
- Doesn't take any more effort than writing tests later
- Requires thinking about requirements and design before writing code
- Shows problems with requirements sooner (can't write code without good requirements)

Testing As You Write Code

- Boundary Testing
- Pre- and Post-conditions
- Assertions
- Defensive Programming
- Error Returns

- Waiting until later means you have to relearn code
  - Fixes will be less thorough and more fragile

Boundary Testing

- Most bugs occur at boundaries
  - If it works at and near boundaries, it likely works elsewhere
- Check loop and conditional bounds when written
  - Check that extreme cases are handled
    - e.g. Full array, Empty array, One element array
    - Usually should check value and +/- 1
- Mental test better than none at all

Preconditions and Postconditions

- Verify that routine starts with correct preconditions and produces correct postconditions
- Check to make sure preconditions met
  - Handle failures cleanly
- Verify that postconditions are met
  - No inconsistencies created
- Need to define pre-/postconditions clearly
- “Provable” software relies on this approach

Assertions

- Available in C/C++
  - assert.h
- Way of checking for pre-/postconditions
- Helps identify where problem occurs
  - Before the assertion
  - e.g. usually in calling routine, not callee
- Problem: causes abort
  - So, useful for testing for errors

Defensive Programming

- Add code to handle the “can’t happen” cases
- Program “protects” itself from bad data

Error Returns

- Good API and routine design includes error codes
- Need to be checked

Systematic Testing

- Test of complete code pieces
- Test incrementally
- Test simple parts first
- Know what output to expect
- Verify conservation properties
- Compare independent implementations
- Measure test coverage

Test Incrementally

- Don't wait until *everything* is finished before test
- Test components, not just system
- Test components individually before connecting them

Test Simple Parts First

- Test most basic, simplest features
- Finds the "easy" bugs (and usually most important) first

Know What Output To Expect

- Design test cases that you will know the answer to!
- Make hand-checks convenient
- Not always easy to do
  - e.g. compilers, numerical programs, graphics

Verify Conservation Properties

- Specific results may not be easily verifiable
  - Have to write the program to compute the answer to compare to
- But, often we have known output properties related to input
  - e.g. #Start + #Insert - #Delete = #Final
- Can verify these properties even without verifying larger result

Compare Independent Implementations

- Multiple implementations to compute same data should agree
- Useful for testing tricky code, e.g. to increase performance
  - Write a slow, brute-force routine
  - Compare the results to the new, “elegant” routine
- If two routines communicate (or are inverses), different people writing them helps find errors
  - Only errors will be from consistent misinterpretation of description

Measure Test Coverage

- What portion of code base is actually tested?
- Techniques to work toward this
  - Following slides
- Tend to work well on only small/moderate code pieces
- For large software, tools help judge coverage

Logic Coverage

- Or, Code Coverage
- Testing every branch, every path through the code
- Can grow (nearly) exponentially with number of choices/branches
- Only suitable for small to medium size codes

Structured Basis Testing

- Testing every line in a program
  - Ensure that every *statement* gets tested
  - Need to test each part of a logical statement
- Far fewer cases than logic coverage
  - But, also not as thorough
- Goal is to minimize total number of test cases
  - One test case can test several statements

Structured Basis Testing (continued)

- Start with base case where all Boolean conditions are true
  - Design test case for that situation
- Each branch, loop, case statement increases minimum number of test cases by 1
  - One more test case per variation, to test the code for that variation

Data Flow Testing

- Examines data rather than control
- Data in one of three states
  - Defined – Initialized but not used
  - Used – In computation or as argument
  - Killed – Undefined in some way
- Variables related to routines
  - Entered – Routine starts just before variable is acted upon
  - Exited – Routine ends immediately after variable is acted upon

Data Flow Testing (continued)

- First, check for any anomalous data sequences
  - Defined-defined
  - Defined-exited
  - Defined-killed
  - Entered-killed
  - Entered-used
  - Killed-killed
  - Killed-used
  - Used-defined
- Often can indicate a serious problem in code design
- After that check, write test cases

Data Flow Testing (continued)

- Write test cases to examine all defined-used paths
- Usually requires
  - More cases than structured basis testing
  - Fewer cases than logic coverage

Example

```
if (cond1) {
    x = a;
} else {
    x = b;
}
if (cond2) {
    y = x+1;
} else {
    y = x+2;
}
if (cond3) {
    z = c;
} else {
    z = d;
}
```

- Logic Coverage / Code Coverage
  Conditions: T T T
  Conditions: T T F
  Conditions: T F T
  Conditions: T F F
  Conditions: F T T
  Conditions: F T F
  Conditions: F F T
  Conditions: F F F
- Tests all possible paths

Example

```
if (cond1) {
    x = a;
} else {
    x = b;
}
if (cond2) {
    y = x+1;
} else {
    y = x+2;
}
if (cond3) {
    z = c;
} else {
    z = d;
}
```

- Structured Basis Testing
  Conditions: T T T
  Conditions: F F F
- Tests all lines of code

Example

```
if (cond1) {
    x = a;
} else {
    x = b;
}
if (cond2) {
    y = x+1;
} else {
    y = x+2;
}
if (cond3) {
    z = c;
} else {
    z = d;
}
```

- Data Flow Testing

1. Conditions: T T T

  Conditions: T F F
  Conditions: F T ?
  Conditions: F F ?

- Tests all defined-used paths
  - Note: cond3 is independent of first two

Example

```
if (cond1) {
    x = a;
} else {
    x = b;
}
if (cond2) {
    y = x+1;
} else {
    y = x+2;
}
if (cond3) {
    z = c;
} else {
    z = d;
}
```

- Data Flow Testing
  Conditions: T T T
2. Conditions: T F F
  Conditions: F T ?
  Conditions: F F ?
- Tests all defined-used paths
  - Note: cond3 is independent of first two

Example

```
if (cond1) {
    x = a;
} else {
    x = b;
}
if (cond2) {
    y = x+1;
} else {
    y = x+2;
}
if (cond3) {
    z = c;
} else {
    z = d;
}
```

- Data Flow Testing
  Conditions: T T T
  Conditions: T F F
3. Conditions: F T ?
  Conditions: F F ?
- Tests all defined-used paths
  - Note: cond3 is independent of first two

Example

```
if (cond1) {
    x = a;
} else {
    x = b;
}
if (cond2) {
    y = x+1;
} else {
    y = x+2;
}
if (cond3) {
    z = c;
} else {
    z = d;
}
```

- Data Flow Testing
  Conditions: T T T
  Conditions: T F F
  Conditions: F T ?
4. Conditions: F F ?
- Tests all defined-used paths
  - Note: cond3 is independent of first two

Test Case Design (If you don't know the code)

- Boundary analysis still applies
- Equivalence partitioning
  - Don't create multiple tests to do the same thing
- Bad data
  - Too much/little
  - Wrong kind/size
  - Uninitialized
- Good data
  - Minimum/maximum normal configuration
  - "Middle of the Road" data
  - Compatibility with old data

Test Automation

- Should do lots of tests, and by-hand is not usually appropriate
- Scripts can automatically run test cases, report on errors in output
  - But, we need to be able to analyze output automatically…
  - Can't always simulate good input (e.g. interactive programs)
- People cannot be expected to remain sharp over many tests
- Automation reduces workload on programmer, remains available in the future

Regression Testing

- Goal: Find anything that got broken by "fixing" something else
- Save test cases, and correct results
- With any modifications, run new code against all old test cases
- Add new test cases as appropriate

Test Support Tools

- Test Scaffold
  - Framework to provide just enough support and interface to test
  - Stub Routines and Test Harness
- Test Data Generators
- System Perturber

Stub Routines

- Dummy object/routine that doesn't provide full functionality, but pretends to do something when called
  - Return control immediately
  - Burn cycles to simulate time spent
  - Print diagnostic messages
  - Return standard answer
  - Get input interactively rather than computed
  - Could be "working" but slow or less accurate

Test Harness

- Calls the routine being tested
  - Fixed set of inputs
  - Interactive inputs to test
  - Command line arguments
  - File-based input
  - Predefined input set
- Can run multiple iterations

Test Data Generators

- Can generate far more data than by hand
- Can test far wider range of inputs
- Can detect major errors/crashes easily
- Need to know answer to test correctness
  - Useful for “inverse” processes – e.g. encrypt/decrypt
- Should weight toward realistic cases

System Perturbers

- Modify system so as to avoid problems that are difficult to test otherwise
  - Reinitialize memory to something other than 0
    - Find problems not caught because memory is “usually” null
  - Rearrange memory locations
    - Find problems where out-of-range queries go to a consistent place in other tests
  - Memory bounds checking
  - Memory/system failure simulation

Other Testing Tools

- Diff tools
  - Compare output files for differences
- Coverage monitors
  - Determine which parts of code tested
- Data recorder/loggers
  - Log events to files, save state information
- Error databases
  - Keep track of what’s been found, and rates of errors
- Symbolic debuggers
  - Will discuss debugging later, but useful for tests