Testing & Debugging

Need for Testing

- Software systems are inherently complex
  - » **Large systems 1 to 3 errors per 100 lines of code (LOC)**
- Extensive **verification** and **validiation** is required to build quality software
  - » **Verification**
    - > **Does the software meet its specifications**
  - » **Validation**
    - > **Does the software meet its requirements**
- Testing is used to determine whether there are faults in a software system

Need for Testing – 2

- The process of testing is to find the minimum number of test cases that can produce the maximum number of failures to achieve a desired level of confidence
- Cost of validation should not be under estimated
- The cost of testing is extremely high
  - » **Anything that we can do to reduce it is worthwhile**

Testing Not Enough

- Exhaustive testing is not usually possible
  - **Testing can determine the presence of faults, never their absence**

    Dikstra

- Test to give us a high level of confidence

Test Strategy

- Identify test criteria
  - » **What are the goals for comparing the system against its specification**
    - > **Reliability, completeness, robustness**
- Identify target components for testing
  - » **In an OO system the classes and class hierarchies**
- Generate test cases
  - » **Produce test cases that can identify faults in an implementation**
- Execute test cases agains target components
- Evaluation
  - » **If expected outputs are not produced, a bug report is isssued**

Test Plans

- Specify a set of test **input**
- For each input give the **expected output**
- Run the program and document the **actual output**
- Example – square root program
  - » **Input: 4.0**
  - » **Expected Output: 2.0**
  - » **Actual Output: 1.99999** **looks all right**

  - » **Input: -4**
  - » **Expected Output: Invalid input**
  - » **Actual Output: 1.99999** **Uh oh! problem here**

Black Box Testing – Data Coverage

- Testing based on input and output alone
  - » **Do not consider underlying implementation**
- Test cases are generated from the specification
  - » **Pre and post conditions**
- Specific kinds of black box testing
  - » **Random testing**
    - > **Generate random inputs**
    - > **Easy to generate cases**
    - > **good at detecting failues**
    - > **Must be able to easily generate expected output**
  - » **Partition testing**
    - > **See next slides**

Partition Testing

- Cannot try all possible inputs
  - » **Partition input into equivalence classes**
    - > **Every value in a class should behave similarly**
- Test cases
    - > **just before boundary**
    - > **just after a boundary**
    - > **on a boundary**
    - > **one from the middle of an equivalence class**
- Loops
  - » **Zero times through the body**
  - » **Once through the body**
  - » **Many times through the body**

Partition Testing – 2

- Example 1 – Absolute value – 2 equivalence classes
  - » **Values below zero and values above zero**
  - » **5 test cases: large negative, -1, 0, +1, large positive**
- Example 2 – Tax rates – 3 equivalence classes
  - > **0 .. $29,000 at 17%**
  - > **29,001 .. 35,000 at 26%**
  - > **35,001 ... at 29%**
  - » **13 Test cases**
    0 1 15,000 28,999 29,000 29,001 29,002 30,000
    34,999 35,000 35,001 35,002 50,000

Partition Testing – 3

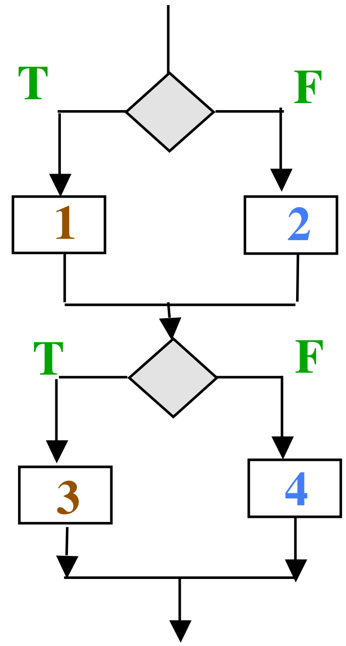
- Example 3 – Insert into a sorted list -- max 20 elements
- About 25 test cases
  - **Boundary conditions on the list**
    - **empty boundary – length 0, 1, 2**
    - **full boundary – length 19, 20 , 21**
    - **middle of range – length 10**
  - **Boundary conditions on inserting**
    - **just before & after first list element**
    - **just before & after last list element**
    - **into the middle of the list**
- Suppose an error occurs when adding to the upper end of a full list
  - **Devise additional test cases to test hypothesis**

White Box Testing

- Use internal properties of the system as test case generation criteria
- Generate cases based on
  - » **Statement blocks**
  - » **Paths**
- Identify unintentional infinite loops, illegal paths, unreachable program text (**dead code**)
- Need test cases for exceptions and interrupts

Statement Coverage

- Make sure every statement in the program is executed at least once
- Example
  - » if a < b then c = a+b ; d = a*b   /* 1 */
    else c = a*b ; d = a+b   /* 2 */
    if c < d then x = a+c ; y = b+d   /* 3 */
    else x = a*c ; y = b*d   /* 4 */



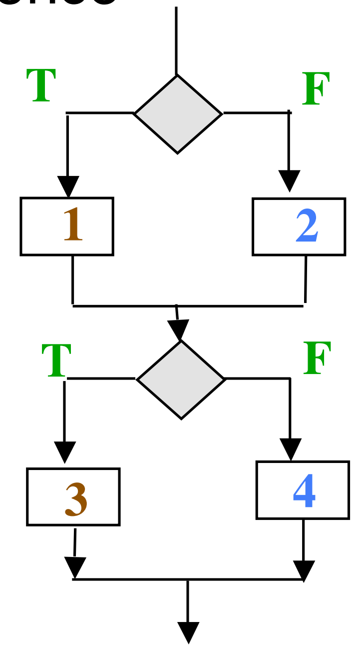
- Statement coverage – can do with 2 tests
  - » Execute 1 & 3 with a < b & a+b < a*b
    - > a = 2 ; b = 5
  - » Execute 2 & 4 with a >= b & a*b >= a+b
    - > a = 5 ; b = 2
- Loops – only 1 test required
  - » Execute body at least once

Statement Coverage – 2

- How do you know you have statement coverage?
- Instrument your program with an array of counters initialized to zero
- Increment a unique counter in each block of statements
- Run your test
- If all counters are non zero then you have achieved statement coverage

Path Coverage

- Every path in the program is executed at least once
- Example
  - » if a < b then c = a+b ; d = a*b /* 1 */
    else c = a*b ; d = a+b /* 2 */
    if c < d then x = a+c ; y = b+d /* 3 */
    else x = a*c ; y = b*d /* 4 */



- Path coverage – 4 tests
  - » Execute 1 & 3 with a < b & a+b < a*b
    a = 2 ; b = 5
  - » Execute 2 & 4 with a >= b & a*b >= a+b a = 5 ; b = 2

    Add for path
  - » Execute 1 & 4 with a < b & a+b >= a*b a = 0 ; b = 1
  - » Execute 2 & 3 with a >= b & a*b < a+b a = 1 ; b = 0

Path Coverage – 2

- Loops – 3 tests required
  - » **Execute body zero times, once in the path, many in the path**
    - > **once is not enough as frequently first time through is a special case**
- Path coverage usually requires exponential increase in tests as the number of choices and loops increases
  - » **due to multiplication**
    - > **two loops in sequence – 9 tests**
    - > **three loops in sequence – 27 tests**
    - > **ten if....then...else in sequence – 1024 tests**

Path Coverage – 3

- Convert an integer represented as a decimal string to a real number.
  - » **ASCII string "123.456" ==> 123.456 in binary**
- The EBNF fro the input
  - » **Input ::= +[Spaces] [+ , -] [IntegerPart] ['.' [DecimalPart]];**
  - » **IntegerPart ::= +[DecimalDigit];**
  - » **DecimalPart ::= +[DecimalDigit];**
  - » **Decimal Digit ::= ('0' , '1', '2' ,'3' ,'4' ,'5' ,'6' ,'7' ,'8' , '9');**

Path Coverage – 4

- The algorithm
  - » **1 Skip any leading spaces.**
  - » **2 Determines what the sign of the number is.**
  - » **3 Get the integer part of the number; determined by scanning either the end of the number or a decimal point.**
  - » **4 Continue building the integer representation of the input as if there was no decimal point, meanwhile counting the number of decimal digits.**
  - » **5 Compute the real number from the sign, integer representation and count of decimal digits.**

Path Coverage – 5

- 2 tests are sufficent for statement coverage
  - » **positive and negative real numbers.**
- 162 tests estimated for all paths.
  - » **3 cases first loop – step 1 skip lead spaces**
  - » **3 cases first if statement – step 2 determine sign**
  - » **3 cases second loop – step 3 get integer part**
  - » **2 cases second if statement – step 3 check decimal point**
  - » **3 cases third loop – step 4 get decimal part**
    - > **Not all cases are possible -- for example if there is no '.' (second if statement), then the third loop cannot not be executed one or many times, only zero times.**

Path Coverage – 6

- How to you know you have path coverage?
- As for statement coverage increment counters in each block of statements
- Compare the pattern of non zero counters with the expected statement blocks in each path
- Continue until every path pattern has been matched

Top Down Testing

- Test upper levels of program first
- Use stubs for lower levels
  - » **Stubs are dummy procedures that have "empty" implementations – do nothing**
    - > **For functions return a constant value**
  - » **Test calling sequences for procedures and simple cases for upper levels.**

Bottom Up Testing

- Use test drivers
  - » **Have complete implementation of subprograms**
  - » **Create a special main program to call the subprograms with a sequence of test cases**
  - » **Can be interactive, semi-interactive, or non-interactive**
    - > **See minimal output test program slides**

Mixed Top & Bottom Testing

- Use scaffolding
  - » **Have some stubs**
  - » **Have some special test drivers**
  - » **Like the scaffolding around a building while it is being built or repaired**
  - » **Not a part of the final product but necessary to complete the task**

Regressive Testing

- Rerun all previous tests whenever a change is made
  - **Changes can impact previously working programs**
  - **So retest everything**
  - **Also must test the changes**

Minimal Output Testing

- Reading and comparing expected with actual output is tedious and error prone
- Task should be automated – especially for regressive testing
- Build the expected output into the test driver and compare with the actual output
- Report only if **expected ≠ actual**
  - » **Output states which test failed, the expected and the actual outputs**
- Successful test runs only output the message
  - » **Tests passed**

Minimal Output Testing – 2

- Example checking a stack implementation

```
Stack list = new Stack();            verifyEmpty("1", list);
list.add("a");                       verifyL1("2", list, "[ a ]");
list.remove();                       verifyEmpty("3", list);
list.add("a");
list.add("b");               verifyL2("4", list, "[ b , a ]");

if (list.contains("c")) println("5 shouldn't contain c");
if (!list.contains("a")) println("6 should contain a");
if (!list.contains("b")) println("7 should contain b");

list.add("c");               verifyL3("8", list, "[ c , b , a ]");
list.remove();               verifyL2("9", list, "[ b , a ]");
list.remove();               verifyL1("10", list, "[ a ]");
list.remove();               verifyEmpty("11", list);
```

- Verify routines do the appropriate test and output messages

Debug Flags

- When debugging it is useful to turn on and off various features in a program
  - » **Especially test output**
- Create a Debug class that contains Boolean flags that can be set, reset and toggled
- Use the flags in if statements that surround interesting sections of your program
- For different test runs give different settings of true and false to the flags

Debug Flags – 2

- Example

```
if Debug.flag0 then
        Block1
fi
if Debug.flag1 then
        Block2
fi
if Debug.flag3 then
        Block3
        if Debug.flag4 then
                Block4
        fi
fi
```

- Depending upon the values of flag0 .. flag3 different combinations of **Block1 .. Block4** are executed

Assertions

- Assertions can be put into programs using if...then statements
  - » **Some languages such as Eiffel have them built in**
- The condition compares expected with actual values and prints a message if the assertion fails
- Combined with a debug flag you can turn assertion checking on and off depending on what you want to test

  if Debug.flag0 & expected ≠ actual then ... fi

Inspections & Walkthroughs

- Manual or computer aided comparisons of software development products
  - » **specifications, program text, analysis documents**
- Documents are paraphrased by the authors
- Walkthroughs are done by going through the execution paths of a program, comparing its outputs with those of the paraphrased documents
- Walkthrough team
  - » **Usually 4-6 people**
  - » **Elicit questions, facilitate discussion**
  - » **Interactive process**
  - » **Not done to evaluate people**

Inspections & Walkthroughs – 2

- Psychological side effects
  - **If a walkthrough is going to be performed, developers frequently write easy to read program text**
  - **This clarity can help make future maintenance easier**

OO Testing

- Using OO technology can have effects on three kinds of testing
  - » **Intra feature**
  - » **Inter feature testing**
  - » **Testing class hierarchies**
- Intra feature testing
  - » **Features can be tested much as procedures & functions are tested in imperative languages**
- Inter feature testing
  - » **Test an entire class against and abstract data type (specification)**
  - » **Some inter-feature testing methods specify correct sequences of feature calls, and use contracts to derive test cases**

Testing Class Hierarchies

- May have re-test inherited features
- For testing a hierarchy it is usually best to test from the top down
  - » **Start with base classes**
  - » **Test each feature in isolation**
  - » **Then test feature interactions**
- Build test histories
  - » **Associate test cases with features**
  - » **History can be inherited with the class, allowing for reuse of test cases**

Testing Classes

- Like unit (module) testing
  - » **Usually tested in isolation**
  - » **Surround class with stubs and drivers**
- Consider the class as the basic unit
- Exercise each feature in turn
  - » **Create test cases for each feature**
- Often a test driver is created
  - » **Simple menu that can be used to exercise each feature with inputs from a test file**
- Need to recompile when switching drivers

Integration Testing

- Testing with all the components assembled
- Focuses on testing the interfaces between components
- Usually want to do this in a piece by piece fashion
  - » **Avoid a big bang test**
  - » **Incremental testing and incremental integration is preferable**
    - > **Integrate after unit testing a component**
- Bottom up and top down approaches may be used