An Introduction to Systematic Software Testing

Robert France
CSU
Why do we need to systematically test software?

- Poor quality products can
  - Inconvenience direct and indirect users
  - Result in severe financial costs
  - Kill/maim users
  - Entangle society
  - Damage reputations

- Quality puts
  - Large amounts of money at stake
  - Corporate success/failure at stake
  - Developer’s personal future at stake
  - Lives at stake

- Systematic software testing is a quality control activity
Quality Control (QC)

• QC is the set of techniques and practices used by developers to help ensure software quality

• QC Activities
  – Validation & Verification
    • Analysis and simulation of design specifications or models
    • Program testing
    • Software reviews, walkthroughs, inspections
  – Process Improvement
  – Change Control
    • Software configuration management
Validation

• Validation is concerned with establishing that the software satisfies the specified and unspecified customer goals.

• We ask:

  “Did we build the right thing?”
Verification

• Verification is concerned with establishing that the software meets specified goals.
• We ask:
  
  ```Did we build the thing right?```
Testing

• Run (exercise) the program on sample inputs.
• Check the validity of the output.
  – If output is valid then test provides some evidence of validity
• Testing is part of either verification or validation.
Who Should Conduct Testing?

• Should the developer do the testing?
• Should we use an independent testing team?
Testing Programs

• Testing can show the presence of errors, but never their absence
• Testing should be considered as only one means of identifying errors and should be integrated with other V&V techniques
• Testing should be based on sound and systematic techniques
Limitations of Ad-hoc Testing

Suppose a programmer set z to 2 instead of 22 when x equals y in the code below:

```plaintext
read(x); read(y);
if x = y then z := 2;
else z := 0;
endif;
write(z);
```

Using a random number generator to generate values for x and y in the tests will not likely produce equal values for x and y.
Testing Terminology

Testing terminology is not standardized.

- **Failure:** invalid behavior
  - A failure occurs when a program executing on input data produces incorrect output.

- **Defect, error:** cause of a failure
  - Some texts make a distinction between errors and defects (e.g., see course text), where an error is a behavior by the programmer (e.g., a bad decision) that leads to a defect in software.

- **Fault:** an incorrect intermediate state that may be entered during program execution
  - A failure occurs only if a fault happens during execution
  - A fault occurs only if an error exists in the program
Testing Terminology (2)

• Analysis: a search for failures by experimentation (e.g., testing) or static analysis.
• Defect removal (debugging): search for errors and their repair.
• Test case: a set of values for the input variables of a program
• Test set: a finite set of test cases
• A program P is correct for a test set T if it produces the correct output for each test set in T. T is said to be successful for P.
Levels of Program Testing

Testing-in-the-small

- *Unit testing*: testing a procedure, function, or class.

Testing-in-the-large

- *Integration testing*: testing connections between units and components.
- *System testing*: test entire system.
- *Acceptance testing*: testing to decide whether to purchase the software.
Levels of Program Testing (2)

- **Alpha testing**: system testing by a user group within the developing organization.
- **Beta testing**: system testing by select customers.
- **Regression testing**: retesting after a software modification.
Dynamic Fault Classification

- Logic faults: omission or commission.
- Overload: data fields are too small.
- Timing: events are not synchronized.
- Performance: response is too slow.
- Environment: error caused by a change in the external environment.
Test Oracles

- Determine whether a test completed with or without errors.
- Often a person who monitors output.
  - Not a reliable method.
- Automatic oracles check output using another program.
  - Requires a formal specification of input/output relationship
Key Testing Questions

- How do we know that we have selected an “adequate” set of test cases?
- How can we determine the extent that selected test cases “cover” the software input domain?
  - Did we cover all the inputs that would “typically” be input to the software in the tests?
  - Did we cover all the “representative” inputs indicating different usages (including “bad” uses) of the software in the tests?
A Pragmatic Testing Strategy

1. Divide domain $D$ into sub-domains $D_1, D_2, \ldots, D_n$; each sub-domain represents some aspect of the program.

2. Select at least one test case from each $D_i$. 
An Example

• Calculating the factorial of a number:
  – If n<0 then output error message
  – If 0<=n<20 then compute n!
  – If 20<=n<200 then approximate n!
  – If n > 200 output error message

• Possible domains are suggested by the problem statement:
  \{n | n<0\}, \{n | 0<=n<20\}, \{n | 20<=n<200\}, \{n | n > 200\}

• One can reasonably expect that if a program behaves for one value in a subdomain it will work for other values.
Test Adequacy Criteria

Question: How do we identify the sub-domains?
Answer: Use test adequacy criteria

• A test criterion attempts to group elements in D into subdomains D1, …, Dn, such that the elements of a given subdomain are expected to behave in exactly the same manner.
  – For each subdomain we can choose a single test case as representative of the input values in the subdomain

• If D = D1 U D2 U … U Dn we say that the test criterion satisfies the complete coverage principle
Another Example

• Suppose we use a criterion that results in the following sub-domains for a program:
  D1={d | d mod 2=0},
  D2={d | d<=0},
  D3={d | d mod 2 not= 0}

• Note that D1 and D2 have common elements, thus we can choose two elements (instead of 3) and satisfy the complete-coverage principle.
  – The test set \{x=48; x=-37\}
Unit Testing

• **White box testing**: using knowledge of the structure of a program to produce test cases.

• **Black box testing**: testing a program using test cases that are generated without knowledge of the design and implementation.

• Both approaches are complementary
  – White box testing tests what a program does
  – Black box testing tests what a program is supposed to do
White box testing criteria

- **Statement coverage criterion**: Select a test set $T$ such that executing program $P$ for each $d$ in $T$ results in each elementary statement of $P$ being executed at least once.

- **Edge-coverage criterion**: Select a test set $T$ such that executing $P$ for each $d$ in $T$ results in each edge of $P$’s control graph being traversed at least once.

- **Condition-coverage criterion**: Select a test set $T$ such that executing $P$ for each $d$ in $T$ results in each edge of $P$’s control graph being traversed at least once and all possible values of the constituents of compound conditions being exercised at least once.

- **Path-coverage criterion**: Select a test set $T$ such that executing $P$ for each $d$ in $T$ results in all paths leading from the initial to the final node of $P$’s control graph being traversed.
Statement coverage example

1. read(x);
2. read(y);
3. if x > 0 then
4. write(“1”);
5. else
6. write(“2”);
7. end if;
8. If y > 0 then
9. write(“3”);
10. else
11. write(“4”);
12. end if;

Input domains for statement coverage
D1: \{x>0\}
D2: \{x<=0\}
D3: \{y>0\}
D4: \{y<=0\}

How did we get these domains?
Ans: from the branch conditions.
Statement coverage weakness

1. if x < 0
   then
2. x := -x;
3. end if;
4. z := x;

Input domains for statement coverage
D1: \{x<0\}

Weakness: does not cover the case when x >= 0.

A test set that satisfies the edge-coverage criterion will cover the case when x >= 0.
Control Flow Graph examples

Simple statement (e.g., \( x := y + 1 \))

\[ x := y + 1 \]
\[ y := z - y \]

Sequence of statements (\( x := y + 1; y := z - y; \))

\[ x := y \]
\[ y := z - y \]
\[ x := y + 1 \]
\[ y := x \]
\[ x := y \]

If \( C \) then \( x := y \) else \( y := x \)

While \( C \) do \( x := y \)

While \( C \) do

While \( C \) do

While \( C \) do
Condition coverage vs. edge coverage criterion

found:= false; counter:= 1;
while (not found) and counter < num_items loop
  if table(counter) = desired_elem then
    found := true;
  end if;
  counter := counter + 1;
end loop;
If found then
  write(“element found”);
else
  write(“element does not exist”);
end if;
Edge criterion test set weakness

A test set for the program on previous slide:
- A table with no items
- A table with three items, the second being the desired element.

The above satisfies the edge coverage criterion but fails to uncover the error in the condition of the while loop (< instead of <=)

The coverage criterion can be used to uncover this error.
The following test set satisfies the edge criterion:
\{<x=0,z=1>,<x=1,z=3>\}

It does not uncover the division by 0 error.

A test set that uncovers the error is given below:
\{<x=0,z=3>,<x=1,z=1>\}
White-box testing summary

- Tests what a program does
- Can catch only errors of “commission”; cannot catch errors of omission
  - Black box testing can be used to catch errors of omission.
- It is not always possible to select test sets that satisfy criterion
  - E.g., unreachable statements in code makes it impossible to satisfy statement coverage criterion
Testing Limitations

• If our testing results in:
  – 100% statement coverage,
  – 100% branch coverage,
  – 100% condition coverage,

The program may still have hidden faults. Why?
Black box testing example

Program specification:
The program receives an invoice as input (detailed description of invoice structure that follows is excluded here). The invoice must be inserted into an invoice file that is sorted by date. The invoice must be inserted in the appropriate position: If other invoices exist in then file with the same date, then the invoice should be inserted after the last one. Also, some consistency checks must be performed: the program should verify whether the customer is already in a corresponding file of customers, whether the customer’s data in the two files match, ...

Test set
• Invoice whose date is the current date
• Invoice whose date is before the current date
  – Invoice whose date is the same as that of some existing invoice
  – Invoice whose date does not exist in invoice file
• Incorrect invoices that can be used to check different types of inconsistencies
Testing boundary conditions

Some programming errors are on the boundary of input domains/partitions used for testing.

```plaintext
if x > y then
dosomething;
else
dosomethingelse
end if;
```

Input domains
D1:{x>y}
D2:{x <= y}

Easy to miss the case x=y when selecting from D2.

Rule of thumb: test using values at the boundaries of the input domains.
Black-Box Class Testing

• Black-Box testing: test a “component” taking an external view.
  – Use the specification to derive test cases.
  – No access to source code.

• Black-Box class testing.
  – Generate tests by analyzing the class interface.
  – Don’t look at method bodies.
Black-Box Class Testing (2)

• Look at the class in isolation, and in conjunction with other associated classes.
• Test each class method, and test sequences of messages that class objects should respond to.
• May need stubs and/or test drivers.
Black-Box Class Testing (3)

- Group class objects into categories.
- Test each method for each category.
- A *test plan* documents all of the tests to be performed.
Testing Java Class *String*: Find *String* Objects to Test

- Find an ordering of the class objects: 
  "", "a", "b", ..., "aa", "ab", ..., "zz...zz"
- Use the ordering to select test objects.
Use the Ordering to Select Test Cases

- Find objects at extremes and next to extremes:
  - Minimum size: "", "a"
  - Long strings: "zz...zz", "zz...zy"
- Middle length Strings.
- Different types of Strings:
  - numbers
  - control characters: "^D^C"
  - symbols: "&$@+->"
- Invalid strings: a null String variable. For C++, you can set a String variable to an integer.
Ex: Testing Java Class Stack

Stack method interface:

– boolean empty()
– Object peek()
– Object pop()
– void push(Object element)
– int search(Object element)

_Boo Hiss!!_ This is non-stack type of operation.

– Object Stack(): The default constructor.
Classify the Operations

• Constructors/Destructors:
  – Stack()
  – ~Stack() in C++

• State changing operations:
  – pop()
  – push(Object e)

• Non-state changing operations:
  – empty()
  – peek()
Test Each Type of Operation

- Constructors: test each constructor with all orderings of parameter boundary values.
- Destructors: test with each constructor.
- State changing operations: try to change the object state from every "`state" to every other `"state".
- Non-state changing operations: test on stacks in each `"state".
“State”

• Really a group of related states.
• Example stack states:
  – Empty stacks,
  – Mid-size stacks,
  – Just under the maximum size stack,
  – Large or full stacks.
Testing *Multiplicity*

- Create several stacks and test, alternating between them.
- This will determine whether each stack object has an independent state (independent instance variables).
The Test Oracle

• How do you know if an item is successfully pushed onto a stack?
  Examine the behavior of the resulting stack after the push operation is performed.
Class Testing Plan Structure

• Class name.
• For each public method:
  – Method name.
  – For each test case for the method:
    • A test ID.
    • Test strategy: black-box (BB) or white-box (WB); Test of valid or invalid input?
    • Test description.
    • Verification: what are the expected outputs? How do you identify success or failure?
Class Stack BB Test Plan

Constructor method Stack() tests:

• Test: Stack.Stack1
  – Strategy: Black Box, Valid.
  – Description: Create a Stack.
  – Verification: A stack object is created; a non-null Stack reference is returned.

• Test: Stack.Stack2
  – Strategy: Black Box, Valid.
  – Description: Create many Stacks.
  – Verification: Many stack objects are created; non-null, not equal Stack references are returned.
Method push(Object e) tests:

- **Test: Stack.push1**
  - Strategy: Black Box, Valid
  - Description: Push one item onto a Stack.
  - Verification: The item is on the top of the stack and can be popped off.

- **Test: Stack.push2**
  - Strategy: Black Box, Valid
  - Description: Push many items onto a Stack.
  - Verification: The items can be popped off in reverse order.
Method push(Object e) tests:

• Test: Stack.push3
  – Strategy: Black Box, Valid
  – Description: Push many items onto many Stacks.
  – Verification: The correct items can be popped off each stack in reverse order.

• Test: Stack.push4
  – Strategy: Black Box, Valid
  – Description: Push a null item onto a Stack.
  – Verification: The null object can be popped off.
Method push(Object e) tests:

• **Test: Stack.push5**
  – Strategy: Black Box, Invalid
  – Description: Push many items onto a null Stack.
  – Verification: An exception is raised.

• **Test: Stack.push6**
  – Strategy: Black Box, Invalid
  – Description: Push a null item onto a non-Stack object.
  – Verification: It won’t compile in Java; An exception is raised in C++.
Method pop() tests:

• Test: Stack.pop1
  – Strategy: Black Box, Valid
  – Description: Pop 1 item from a Stack.
  – Verification: The item can be popped off.

• Test: Stack.pop2
  – Strategy: Black Box, Valid
  – Description: Pop many items from a Stack.
  – Verification: The items can be popped off.
Method pop() tests:

• **Test: Stack.pop3**
  – Strategy: Black Box, Valid
  – Description: Pop 1 item from many Stacks.
  – Verification: The item can be popped off in reverse order.

• **Test: Stack.pop4**
  – Strategy: Black Box, Valid
  – Description: Pop a null item from a Stack.
  – Verification: The item can be popped off.
Method pop() tests:

- **Test: Stack.pop5**
  - Strategy: Black Box, Invalid
  - Description: Pop a null Stack.
  - Verification: An exception is raised.

- **Test: Stack.pop6**
  - Strategy: Black Box, Invalid
  - Description: Pop a non-Stack object.
  - Verification: Will not compile. Exception in C++
Class Stack BB Test Plan (8)

Method pop() tests:
• Test: Stack.pop7
  – Strategy: Black Box, Invalid
  – Description: Pop an empty Stack.
  – Verification: An exception is raised.

Method empty() tests:
• Test: Stack.empty1
  – Strategy: Black Box Valid
  – Description: Test a newly created Stack.
  – Verification: Returns true.
Method empty() tests:

- **Test: Stack.empty2**
  - Strategy: Black Box Valid
  - Description: Test a Stack with a history of 1 push and 1 pop.
  - Verification: Returns true.

- **Test: Stack.empty3**
  - Strategy: Black Box, Valid.
  - Description: Test a Stack with many pushes, and an equal number of pops.
  - Verification: Returns true.
Method empty() tests:

- **Test: Stack.empty4**
  - Strategy: Black Box Valid
  - Description: Test after a push-pop-push-pop sequence.
  - Verification: Returns true.

- **Test: Stack.empty5**
  - Strategy: Black Box, Valid.
  - Description: Test after 1 push.
  - Verification: Returns false.
Method empty() tests:

• Test: Stack.empty6
  – Strategy: Black Box Valid
  – Description: Test after many pushes.
  – Verification: Returns false.

• Test: Stack.empty7
  – Strategy: Black Box, Invalid.
  – Description: Test a null Stack.
  – Verification: Exception.
Method empty() tests:
• Test: Stack.empty8
  – Strategy: Black Box Invalid
  – Description: Test a non-Stack.
  – Verification: Won’t compile in Java. Exception in C++.
Test Drivers

• Must be able to:
  – Build the test cases.
  – Log testing results.
  – Make success or failure observable.

• Can be
  – Hard-coded.
  – Reads tests from a file.
  – Interactive.

• Each test should run independently.
Example Class Test Driver

public class StackTest {
    public static void main (String[] args) throws IOException{
        /** We run the tests **/
        push1();
        push5(); // No crash after invalid test!
        push2();
        push3();
        push5();
    }
}
public static void push1() {
    System.out.println("\n"
        + "Test StackPush1. Test Type: BB, Valid \n"
        + " Description: Push 1 item onto a stack. \n"
        + " Verification: Item is there and can be  
        + " popped off.");
try{
    Stack s = new Stack();
    String i1 = "Item 1";
    s.push(i1);
    System.out.println("Testing Results: \n"
                     + " Item i1: " + i1 +"\n"
                     + " After s.push(i1); The top is: " + s.peek()
                     + "\n"
                     + " Is i1==s.pop()? " + (i1==s.pop()));
}
catch (Exception e){
    System.out.println("Testing Results: Error. \n" + e);
}
public static void push2() {
    System.out.println("\n" +
        "Test Stack.push.2, Test Type: BB, Valid \n" +
        "  Description: Push many items onto a stack. \n" +
        "  Verification: Items can be popped off in reverse order.");
Now Run Test Stack.push.2

```
try{
    Stack s = new Stack();
    String i1 = "Item 1";
    String i2 = "Item 2";
    String i3 = "Item 3";
    String i4 = "Item 4";
    String i5 = "Item 5";
    s.push(i1);
    s.push(i2);
    s.push(i3);
    s.push(i4);
    s.push(i5);
```
Completing Stack.test.2

System.out.println("Testing Results: \\
  + " After pushing items 1 through 5 onto Stack s, \\
  + " we try to pop them off:");
for (int i=1; i<6; i++)
  System.out.println("  \\
  + " + i + ". s.pop(): " +
       (s.pop()));
} // end try
Catch Stack.test.2 Failures

catch (Exception e){
    System.out.println("Testing Results:");
    + " Error. \n    " + e);
}
}
public static void push3(){
    System.out.println("\n" +
    "Test Stack.push.3, Test Type: BB, Valid \n" +
    " Description: Push many items onto many stacks. \n" +
    " Verification: Items can be popped off in reverse order.");
Run Test Stack.push.3

```java
try{
    Stack s1 = new Stack();
    Stack s2 = new Stack();
    Stack s3 = new Stack();
    String i1 = "Item 1";
    String i2 = "Item 2";
    String i3 = "Item 3";
    String i4 = "Item 4";
    String i5 = "Item 5";
    String i6 = "Item 6";
    s1.push(i1);
    s2.push(i2);
    s3.push(i3);
    s3.push(i4);
    s2.push(i5);
    s1.push(i6);
}
```
System.out.println("Testing Results:\n" + " After pushing items 1 through 6 onto Stack s1, s2, s3, \n" + " we try to pop them off:");
System.out.println(" s1 has items pushed in order 1, 6");
for (int i=1; i<3; i++)
    System.out.println("   " + i + ". s1.pop(): " + (s1.pop()));

System.out.println(" s2 has items pushed in order 2, 5");
for (int i=1; i<3; i++)
    System.out.println("   " + i + ". s2.pop(): " + (s2.pop()));
Complete Stack.push.3

System.out.println(" s3 has items pushed in order 3, 4");
for (int i=1; i<3; i++)
    System.out.println("  " + i + ". s3.pop(): " + (s3.pop()));
}
catch (Exception e){
    System.out.println("Testing Results: Error. \n  " + e);
}
}
public static void push5(){
    System.out.println( "\n" +
        "Test Stack.push.5, Test Type: BB, Invalid \n" +
        "  Description:  Try to push onto a stack variable
        which \n" +
        "    is not instantiated (a null stack).\n" +
        "  Verification: Exception is raised." );
try{
    Stack s = null;
    String theItem = "The Item";
    s.push(theItem);
    System.out.println("Testing Results: 
        + " Item theItem: " + theItem +"\n"
        + " After s.push(i1); The top is: " + s.peek()
        + "\n" + " Is theItem==s.pop()? " +
           (theItem==s.pop()));
}
Now We Really Catch an Exception

catch (Exception e){
    System.out.println("Testing Results Error. \n   "+ e);
}