NEC
Empowered by Innovation

University of Waterloo

Software Testing and Quality Assurance

Theory and Practice

Chapter 3

Unit Testing

Outline of the Chapter

- Concept of Unit Testing
- Static Unit Testing
- Defect Prevention
- Dynamic Unit Testing
- Mutation Testing
- Debugging
- Unit Testing in eXtreme Programming
- Tools For Unit Testing

Concept of Unit Testing

- Static Unit Testing
  - Code is examined over all possible behaviors that might arise during run time
  - Code of each unit is validated against requirements of the unit by reviewing the code
- Dynamic Unit Testing
  - A program unit is actually executed and its outcomes are observed
  - One observe some representative program behavior, and reach conclusion about the quality of the system
- Static unit testing is not an alternative to dynamic unit testing
- Static and Dynamic analysis are complementary in nature
- In practice, partial dynamic unit testing is performed concurrently with static unit testing
- It is recommended that static unit testing be performed prior to the dynamic unit testing

Static Unit Testing

- In static unit testing code is reviewed by applying techniques:
  - **Inspection:** It is a step by step peer group review of a work product, with each step checked against pre-determined criteria
  - **Walkthrough:** It is review where the author leads the team through a manual or simulated executed of the product using pre-defined scenarios
- The idea here is to examine source code in detail in a systematic manner
- The objective of code review is to *review* the code, and *not* to evaluate the author of the code
- Code review must be planned and managed in a professional manner
- The key to the success of code is to divide and conquer
  - An examiner inspect small parts of the unit in isolation
    - nothing is overlooked
    - the correctness of all examined parts of the module implies the correctness of the whole module

Static Unit Testing (Code Review)

- Step 1: **Readiness**
  - **Criteria**
    - **Completeness**
    - **Minimal functionality**
    - **Readability**
    - **Complexity**
    - **Requirements and design documents**
  - **Roles**
    - **Moderator**
    - **Author**
    - **Presenter**
    - **Record keeper**
    - **Reviewers**
    - **Observer**
- Step 2: **Preparation**

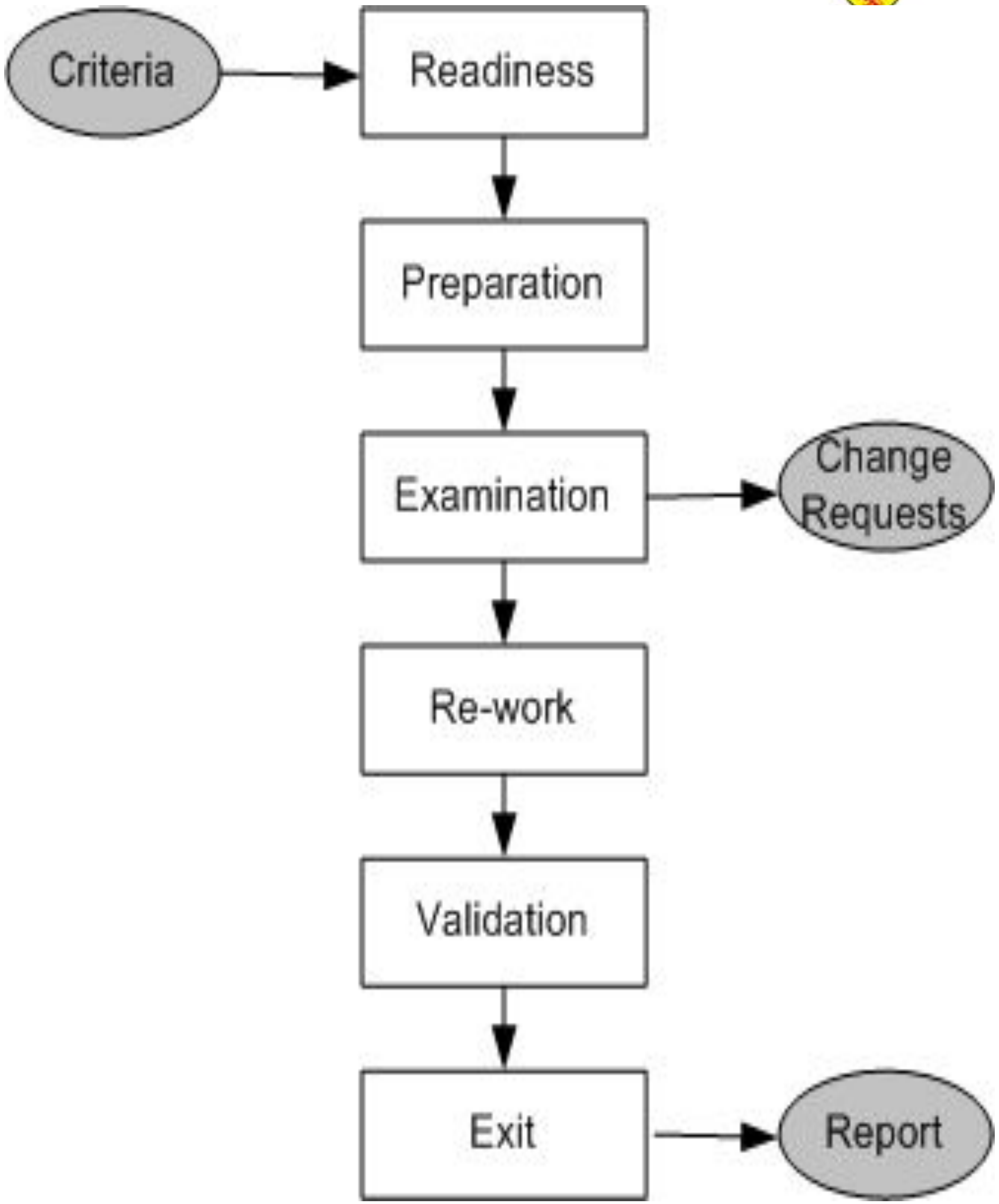

Figure 3.1: Steps in the code review process

Static Unit Testing (Code Review)

A **Change Request (CR)** includes the following details:
- Give a brief description of the issue
- Assign a priority level (major or minor) to a **CR**
- Assign a person to follow it up
- Set a deadline for addressing a **CR**

- Step 3: **Examination**
  - The author makes a presentation
  - The presenter reads the code
  - The record keeper documents the CR
  - Moderator ensures the review is on track
- Step 4: **Re-work**
  - Make the list of all the CRs
  - Make a list of improvements
  - Record the minutes meeting
  - Author works on the CRs to fix the issue
- Step 5: **Validation**
  - CRs are independently validated
- Step 6: **Exit**
  - A summary report of the meeting minutes is

Static Unit Testing (Code Review)

The following metrics can be collected from a code review:

- The number of lines of code (LOC) reviewed per hour
- The number of CRs generated per thousand lines of code (KLOC)
- The number of CRs generated per hour
- The total number of hours spend on code review process

Static Unit Testing (Code Review)

- The code review methodology can be applicable to review other documents
- Five different types of system documents are generated by engineering department
  - Requirement
  - Functional Specification
  - High-level Design
  - Low-level Design
  - code
- In addition installation, user, and trouble shooting guides are developed by technical documentation group

Hierarchy of System Documents
Requirement: High-level marketing or product proposal.
Functional Specification: Software Engineering response to the marketing proposal.
High-Level Design: Overall system architecture.
Low-Level Design: Detailed specification of the modules within the architecture.
Programming: Coding of the modules.

Table 3.1: System documents

Defect Prevention

- Build instrumentation code into the code
- Use standard control to detect possible occurrences of error conditions
- Ensure that code exists for all return values
- Ensure that counter data fields and buffer overflow/underflow are appropriately handled
- Provide error messages and help texts from a common source
- Validate input data
- Use assertions to detect impossible conditions
- Leave assertions in the code.
- Fully document the assertions that appears to be unclear
- After every major computation reverse-compute the input(s) from the results in the code itself
- Include a loop counter within each loop

Dynamic Unit Testing

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as *test driver*
  - A *test driver* is a program that invokes the unit under test (UUT)
  - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called *stubs*
  - It is a dummy program
- The *test driver* and the *stubs* are together called *scaffolding*
- The low-level design document provides guidance for selection of input test data

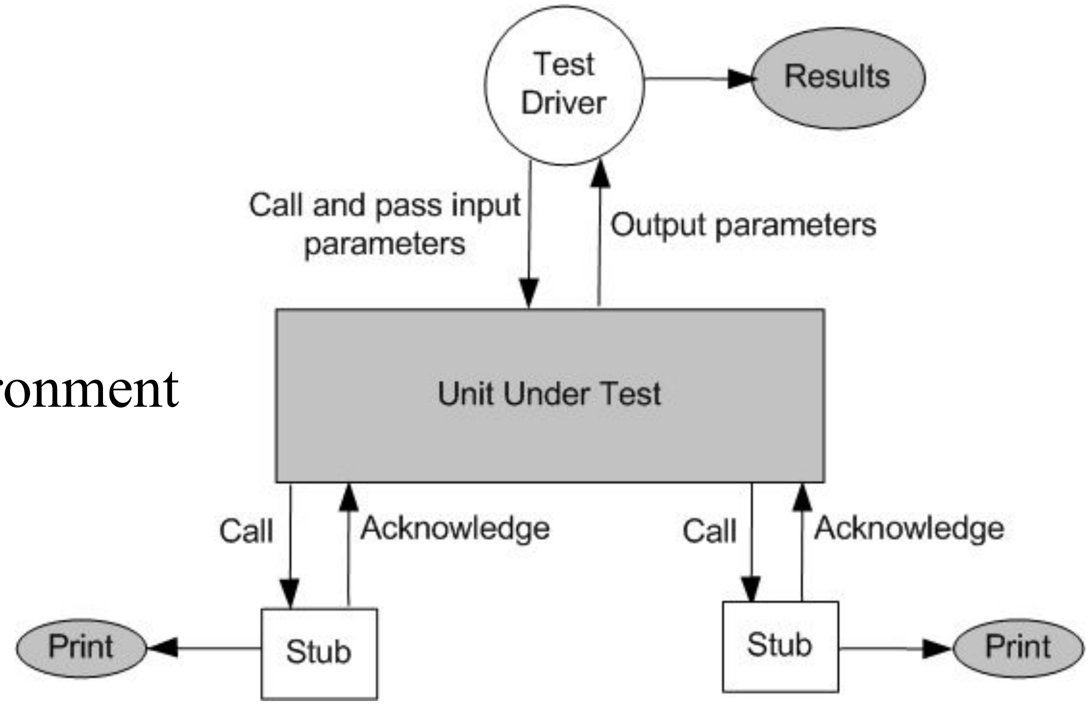

Figure 3.2: Dynamic unit test environment

Dynamic Unit Testing

Selection of test data is broadly based on the following techniques:

- Control flow testing
  - Draw a control flow graph (CFG) from a program unit
  - Select a few control flow testing criteria
  - Identify a path in the CFG to satisfy the selection criteria
  - Derive the path predicate expression from the selection paths
  - By solving the path predicate expression for a path, one can generate the data
- Data flow testing
  - Draw a data flow graph (DFG) from a program unit and then follow the procedure described in control flow testing.
- Domain testing
  - Domain errors are defined and then test data are selected to catch those faults
- Functional program testing
  - Input/output domains are defined to compute the input values that will cause the unit to produce expected output values

Mutation Testing

- Modify a program by introducing a single small change to the code
- A modified program is called *mutant*
- A mutant is said to be *killed* when the execution of test case cause it to fail. The mutant is considered to be *dead*
- A mutant is an *equivalent* tot the given program if it always produce the same output as the original program
- A mutant is called *killable* or *stubborn*, if the existing set of test cases is insufficient to kill it
- A mutation *score* for a set of test cases is the percentage of non-equivalent mutants *killed* by the test suite
- The test suite is said to be *mutation-adequate* if its mutation score is 100%

Mutation testing

- Test Case 1:
  input: 1 2 3
  output: Value of the rank is 3
- Test Case 2:
  input: 1 2 1
  output: Values of the rank is 2
- Test Case 3:
  input: 3 1 2
  output: Value of the rank is 1

Consider the following program P

- main(argc,argv)
- int argc, r, i;
- char *argv[];
- { r = 1;
- for i = 2 to 3 do
- if (atoi(argv[i]) > atoi(argv[r])) r = i;
- printf("Value of the rank is %d \n", r);
- exit(0); }

Mutant 1: Change line 5 to for i = 1 to 3 do
Mutant 2: Change line 6 to if (i > atoi(argv[r])) r = i;
Mutant 3: Change line 6 to if (atoi(argv[i]) >= atoi(argv[r])) r = i;
Mutant 4: Change line 6 to if (atoi(argv[r]) > atoi(argv[r])) r = i;
Execute modified programs against the test suite, you will get the results:
Mutants 1 & 3: Programs will pass the test suite, i.e., mutants 1 & 3 are not *killable*
Mutant 2: Program will fail test cases 2
Mutant 1: Program will fail test case 1 and test cases 2
Mutation score is 50%, assuming mutants 1 & 3 non-equivalent

Mutation testing

- The score is found to be low because we assumed mutants 1 & 3 are nonequivalent
- We need to show that mutants 1 and 3 are equivalent mutants or those are killable
- To show that those are killable, we need to add new test cases to kill these two mutants
- First, let us analyze mutant 1 in order to derive a “killer” test. The difference between P and mutant 1 is the starting point
- Mutant 1 starts with i = 1, whereas P starts with i = 2. There is no impact on the result r. Therefore, we conclude that mutant 1 is an equivalent mutant
- Second, if we add a fourth test case as follows:

Test Case 4:

input: 2 2 1

- Program P will produce the output “Value of the rank is 1” and mutant 3 will produce the output “Value of the rank is 2”
- Thus, this test data kills mutant 3, which give us a mutation score 100%

Mutation Testing

Mutation testing makes two major assumptions:

- Competent Programmer hypothesis
  - Programmers are generally competent and they do not create *random* programs
- Coupling effects
  - Complex faults are coupled to simple faults in such a way that a test suite detecting simple faults in a program will detect most of the complex faults

Debugging

- The process of determining the cause of a failure is known as *debugging*
- It is a time consuming and error-prone process
- Debugging involves a combination of systematic evaluation, intuition and a little bit of luck
- The purpose is to isolate and determine its specific cause, given a symptom of a problem
- There are three approaches to *debugging*
  - Brute force
  - Cause elimination
    - Induction
    - Deduction
  - Backtracking

Unit Testing in eXtreme Programming

1. Pick a requirement, i.e., a story
2. Write a test case that will verify a small part of the story and assign a fail verdict to it
3. Write the code that implement particular part of the story to pass the test
4. Execute all test
5. Rework on the code, and test the code until all tests pass
6. Repeat step 2 to step 5 until the story is fully implemented

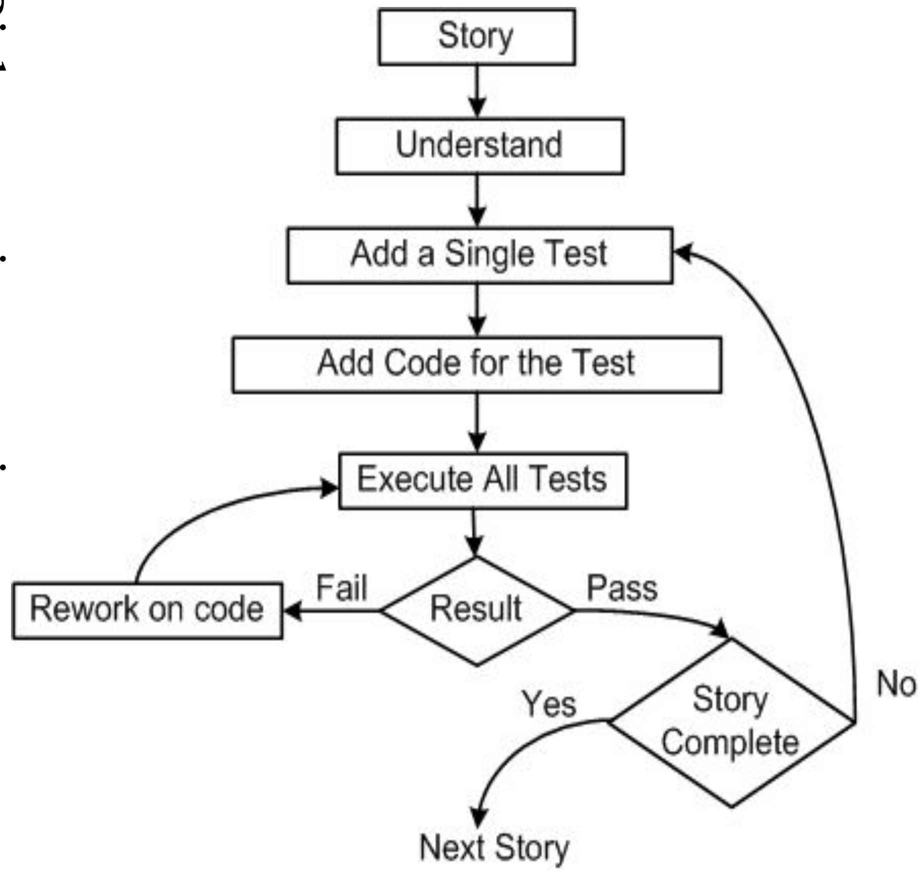

Figure 3.3: *Test-first* process in XP

Unit Testing in eXtreme Programming

Three laws of Test Driven development (TDD)

- One may not write production code unless the first failing unit test is written
- One may not write more of a unit test than is sufficient to fail
- One may not write more production code than is sufficient to make the failing unit test pass

Pair programming:

- In XP code is being developed by two programmers working side by side
- One person develops the code tactically and the other one inspects it methodically by keeping in mind the story they are implementing

JUnit – A Framework for Unit Testing

- JUnit: It is a framework for performing unit testing of Java programs.
  - Other frameworks: NUnit (C#), CPPUnit (C++), fUnit (Fortran)
- Intuitive steps to test a method in Java (Ex. Move() method of PlanetClass)
  - **Create** an object instance of PlanetClass. Call it Mars.
  - **Select** values of all input parameters of Move().
  - **Compute** the expected value to be returned by Move(). Let it be y.
  - **Execute** method Move() on Mars with the selected input values.
    - Let Move() return a value called z.
  - **Compare** the actual output (z) returned by Move() with the expected value (y).
    - If (z == y), Move() ***passes*** the test; otherwise it ***fails***. □ **Report** the result.
- JUnit makes writing of test cases easier. □ Next slide …

JUnit – A Framework for Unit Testing

NEC Empowered by Innovation

University of Waterloo

- JUnit provides a basic class called TestCase.
- The tester
  - *Extends* the TestCase class for each test case. 10 extensions for 10 test cases.
  - Alternatively, extend TestCase to have 10 methods for 10 test cases.
- The TestCase class provides methods to make *assertions*.
  - assertTrue(Boolean condition)
  - assertFalse(Boolean condition)
  - assertEquals(Object expected, Object actual)
  - assertEquals(int expected, int actual)
  - assertEquals(double expected, double actual, double tolerance)
  - assertSame(Object expected, Object actual)
  - assertNull(Object testobject)
  - …
- The tester can have her own assertions.

JUnit – A Framework for Unit Testing

- Each assertion accepts an optional *first* parameter of type String; if the assertion **fails**, the string is displayed. □ Help for the tester…
- The assertEquals() method displays a message upon failure.
  - junit.framework.AssertionFailedError: expected: <x> but was: <y>
- Note that only failed tests are reported.
- The following shows how assertTrue() works.

```
static public void assertTrue(Boolean condition) {
        if (!condition)
            throw new AssertionFailedError();
}
```

Figure 3.5: The assertTrue() assertion throws an exception

JUnit – A Framework for Unit Testing


```
import TestMe; // TestMe is the class whose methods are going
    to be tested.
import junit.framework.*; // This contains the TestCase class.

public class MyTestSuite  extends  TestCase { // Create a
    subclass of TestCase

    public void MyTest1() { // This method is the first test case
           TestMe object1 = new TestMe( ... ); // Create an
    instance of TestMe  with desired params
           int x = object1.Method1(...); //  invoke Method1 on
    object1
           assertEquals(365, x); // 365 and x are expected and
    actual values, respectively.
    }

    public void MyTest2() { // This method is the second test
    case
           TestMe object2 = new TestMe( ... ); // Create another
    instance of
                                // TestMe with desired parameters
           double y = object2.Method2(...); //  invoke Method2 on
    object2
           assertEquals(2.99, y, 0.0001d); // 2.99 is the expected
    value;
                              // y is the actual value;
                              // 0.0001 is tolerance level
    }
}
```

Figure 3.5: An example test suite

Tools For Unit Testing

- Code auditor
  - This tool is used to check the quality of the software to ensure that it meets some minimum coding standard
- Bound checker
  - This tool can check for accidental writes into the instruction areas of memory, or to other memory location outside the data storage area of the application
- Documenters
  - These tools read the source code and automatically generate descriptions and caller/callee tree diagram or data model from the source code
- Interactive debuggers
  - These tools assist software developers in implementing different debugging techniques

    Examples: Breakpoint and Omniscient debuggers
- In-circuit emulators
  - It provides a high-speed Ethernet connection between a host debugger and a target microprocessor, enabling developers to perform source-level debugging

Tools for Unit Testing

- Memory leak detectors
  - These tools test the allocation of memory to an application which request for memory and fail to de-allocate memory
- Static code (path) analyzer
  - These tool identify paths to test based on the structure of code such as McCabe's cyclomatic complexity measure

Cyclomatic complexity

McCabe's complexity measure is based on the cyclomatic complexity of a program graph for a module. The metric can be computed by using the formula: $v = e - n + 2$,
where:
v = cyclomatic complexity of the graph,
e = number of edges (program flow between nodes),
n = number of nodes (sequential group of program statements).
If a strongly connected graph is constructed (one in which there is an edge between the exit node and the entry node) the calculation is $v = e - n + 1$.
Example: A program graph, illustrated below is used to depict control flow. Each circled node represents a sequence of program statements, and the flow of control is represented by directed edges. For this graph the cyclomatic complexity is $v = 9 - 8 + 2 = 3$.

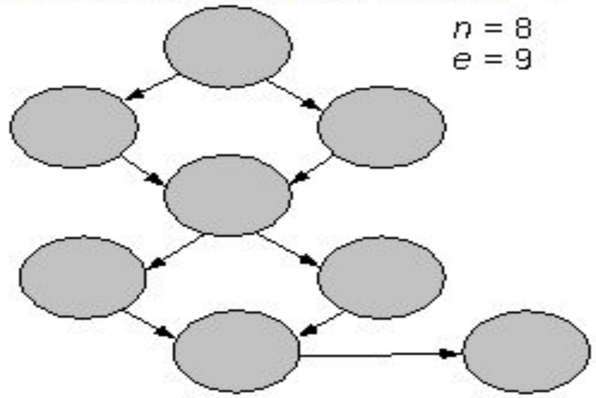

Table 3.3: McCabe complexity measure

Tools for Unit Testing

- Software inspection support
  – Tools can help schedule group inspection
- Test coverage analyzer
  – These tools measure internal test coverage, often expressed in terms of control structure of the test object, and report the coverage metric
- Test data generator
  – These tools assist programmers in selecting test data that cause program to behave in a desired manner
- Test harness
  – This class of tools support the execution of dynamic unit tests
- Performance monitors
  – The timing characteristics of the software components be monitored and evaluate by these tools
- Network analyzers
  – These tools have the ability to analyze the traffic and identify problem areas

Tools for Unit Testing

- Simulators and emulators
  - These tools are used to replace the real software and hardware that are not currently available. Both the kinds of tools are used for training, safety, and economy purpose
- Traffic generators
  - These produces streams of transactions or data packets.
- Version control
  - A version control system provides functionalities to store a sequence of revisions of the software and associated information files under development