

CptS 322- Software Engineering Principles I

Software Testing

Instructor: Sakire Arslan Ay
Fall 2021



World Class. Face to Face.

Outline

- Testing practice
- Granularity levels of testing
 - Unit testing : Black-box and white-box testing
 - Integration, functional, system, acceptance, regression testing
- Test driven development
- Testing Flask applications
 - Unittest
 - Pytest

Software has bugs!

- On average, 1-5 errors per 1K in mature software
 - >10 bugs for 1K lines of code in prototype software
- For complex software 100% correctness is not (economically) feasible
 - We must verify software as much as possible

Approaches to Software Verification

- **Testing:** exercising software to try to discover failures
 - Purpose: reveal failures by running the program
 - Limits: exercise only a small subset of the domain
(=>risk of inadequate test case set)
- **Inspection/review/walkthrough:** manual review of program text to detect faults
 - Code review, pair programming
 - Limits: informal, uneven

Approaches to Software Verification (Cont.)

- **Static verification:** identify (specific) problems by a program that inspects source code, that is, considering all execution paths statically
 - What are pros/cons ?
- **Formal proof:** proving, starting from program source, that the program implements the program specification
 - Limits: complexity, high cost

Today, quality assurance is mostly done through testing

- Commonly, 10% of engineering staff are testers, and the rest spends 50% of their time writing testing and testing infrastructure
- Serious organizations treat testers as first-class engineers
 - Example title: Software Engineer in Test
- Many organizations require developers to write most of the tests
 - The test infrastructure and the code should be designed and developed together
 - Test-driven development

Granularity Levels of Testing

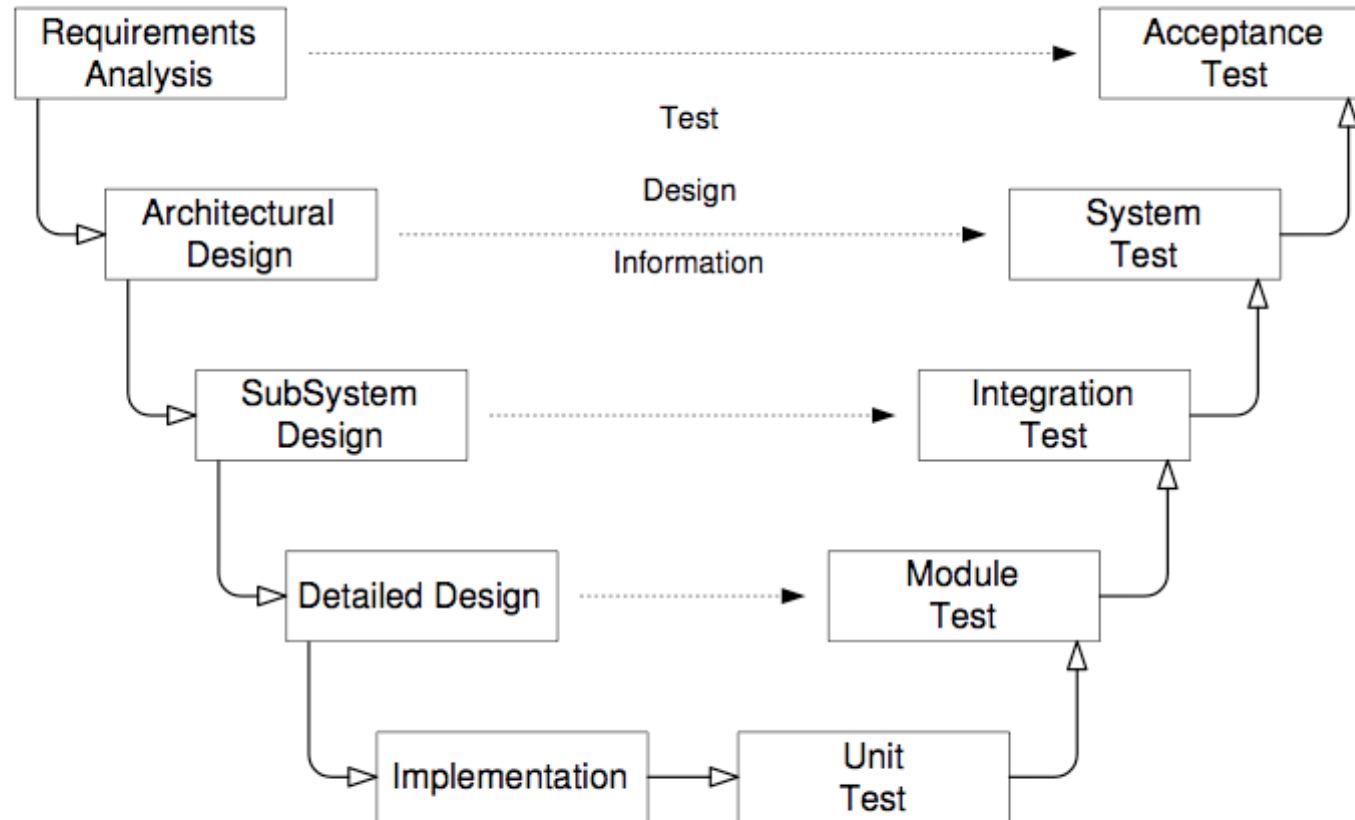



Figure 1.2: Testing levels and software activities – the “V Model”.

Granularity Levels of Testing

- Unit testing: verification of single units
 - Typically single methods, or a few closely related methods
- Functional and Integration testing:
 - verification of the interactions among different modules
- System testing:
 - testing of the system as a whole
- Acceptance testing:
 - verification of the software against the user requirements
- Regression testing:
 - testing of new versions of the software against old tests;
 - can have regression testing at all levels

Unit Testing

Two rules of unit testing

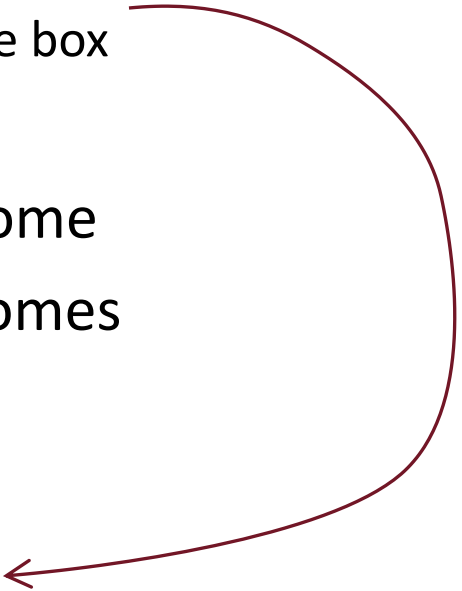
- Do it **early** and do it **often**
 - Catch bugs quickly, before they have a chance to hide
 - **Automate** the process if you can
- Be **systematic**
 - If you thrash about arbitrarily, the bugs  will hide in the corner until you're gone

Steps of a Unit Test

Four basic steps of a test

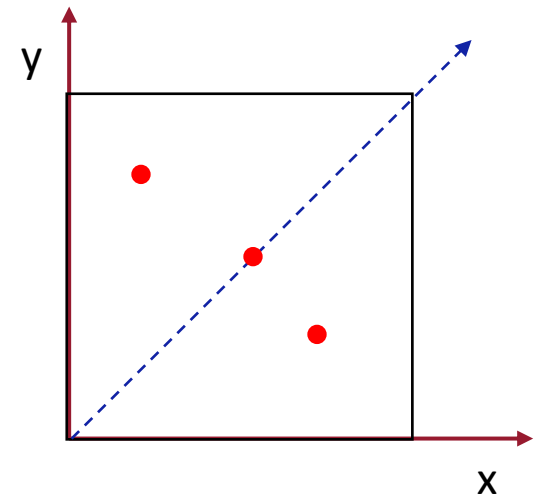
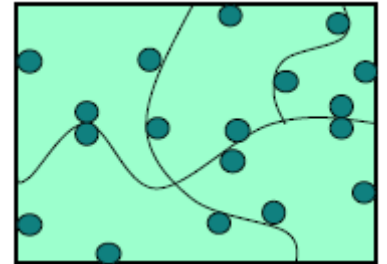
1. Choose input data
 - without looking at the implementation: black box
 - with knowledge of the implementation: white box
2. Define the expected outcome
3. Run on the input to get the actual outcome
4. Compare the actual and expected outcomes

This is not a trivial task!
Need a set of test cases that is
small enough to run quickly, yet
large enough to cover [all]
interesting program
behaviors.



Unit Testing – Test Cases

- Choosing inputs: two key ideas
 - Partition the input space
 - Identify subdomains with the same behavior
 - Pick one input from each subdomain
 - Boundary values
 - Pick inputs at the edges of the subdomains.
 - Effective at finding corner case bugs:
 - off-by-one bugs,
 - Overflow errors in arithmetic
 - Object aliasing
 - Empty cases (0 elements, null, ...)



Boundary Testing

- Partition the input space

- Example-1:

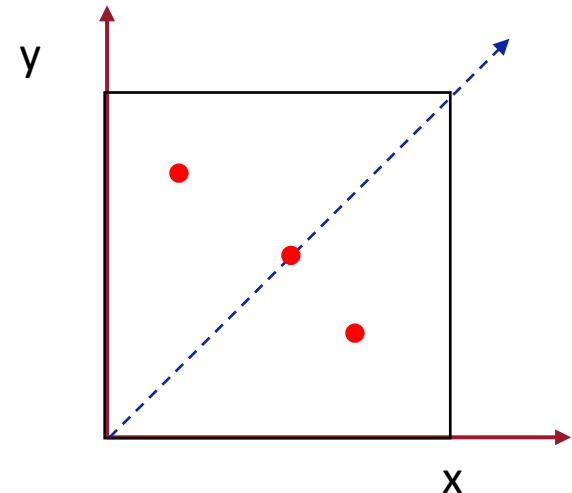
```
// returns the maximum of x, y  
public static int max(int x, int y) { ... }
```

- Partition into:

$x < y$, $x = y$, $x > y$

- Pick an input from each class

(1, 2), (1, 1), (2, 1)



- How would you partition the input space for set intersection?

Boundary Testing

➤ What are good boundary values for objects?

- Partition the input space

- Example-2:

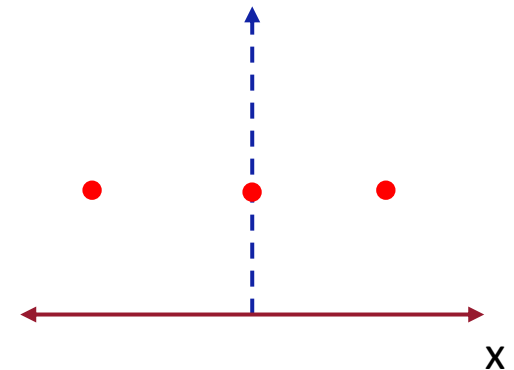
```
// returns |x|  
public static int abs(int x) { ... }
```

- Partition into:

- $x < 0$, $x > 0$, $x = 0$ (boundary)

- Other boundary values

- Integer.MAX_VALUE
 - Integer.MIN_VALUE



Boundary Testing

- Boundary cases for objects:
 - Null
 - Max size (if applicable)
 - Same object passed as multiple arguments (aliasing)
- Example: list of integers
 - Basic operations: create, append, remove
 - Boundary point: [] (can't apply remove)
 - Boundary point: MAX_LIST_SIZE (can't apply append)

Boundary Testing: Duplicates & Aliases

```
// modifies: src, dest
// effects: removes all elements of src and
           appends them in reverse order to
           the end of dest
<E> void appendList(List<E> src, List<E> dest) {
    while (src.size()>0) {
        E elt = src.remove(src.size()-1);
        dest.add(elt);
    }
}
```

- What happens if *src* and *dest* refer to the same object?
 - This is aliasing
 - It's easy to forget!
 - Watch out for shared references in inputs

Unit Test: Black box testing

- Explores alternate paths through the **specification**.
 - Module under test is a black box: interface visible, internals hidden

```
// returns: a > b ⇒ returns a
//          a < b ⇒ returns b
//          a = b ⇒ returns a
public static int max(int x, int y) {...}
```

- According to the specification, there are 3 paths, so there are 3 subdomains
 - (1, 2) => 2
 - (2, 1) => 2
 - (0, 0) => 0



Advantages/Disadvantages of Black Box Testing

- Advantages

- Process is not influenced by component being tested
 - Assumptions embodied in code not propagated to test data.
- Robust with respect to changes in implementation
 - Test data need not be changed when code is changed
- Tests can be written before code
- Independent testers can test the system
 - Testers need not be familiar with code

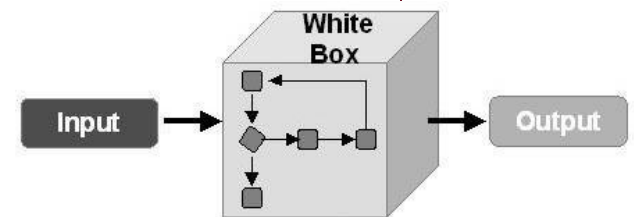
- Disadvantages

- It will miss bugs in the implementation that are not covered by the specification
 - Control-flow details
 - Performance optimizations
 - Alternate algorithms for different cases

Unit Test: White box testing

- Explores alternate paths through the **implementation**.
 - Module under test is a clear box: internals visible

```
boolean[] primeTable = new boolean[CACHE_SIZE];  
/*assume primeTable is initialized. */  
  
/*if x is prime -> true  
  else -> false*/  
boolean isPrime(int x) {  
    if (x>CACHE_SIZE) {  
        for (int i=2; i<x/2; i++) {  
            if (x%i==0) return false;  
        }  
        return true;  
    } else {  
        return primeTable[x];  
    }  
}
```



- Important transition at around $x = \text{CACHE_SIZE}$

Advantages/Disadvantages of White Box Testing

- Advantages
 - Finds an important class of boundaries.
 - Yields useful test cases.
 - In `isPrime` example, need to check numbers on each side of `CACHE_SIZE`
 - `CACHE_SIZE-1`, `CACHE_SIZE`, `CACHE_SIZE+1`
- Disadvantages
 - Tests may have same bugs as implementation
 - Buggy code tricks you into complacency once you look at it

xUNIT – Family of Test Frameworks

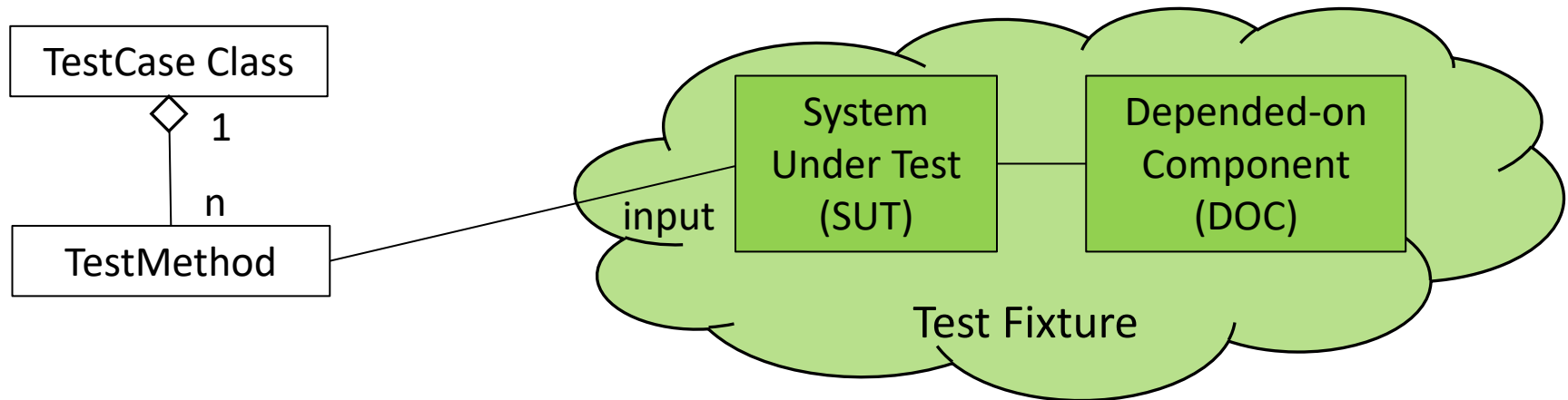
- Created by Kent Beck – 1989
 - Original was SUnit (Smalltalk)
- Once JUnit appeared, the framework got more popular
 - Wikipedia lists 70 ports to various languages
- Reference: Meszaros, “xUNIT Test Patterns” (i.e., UTP)

xUNIT – Family of Test Frameworks

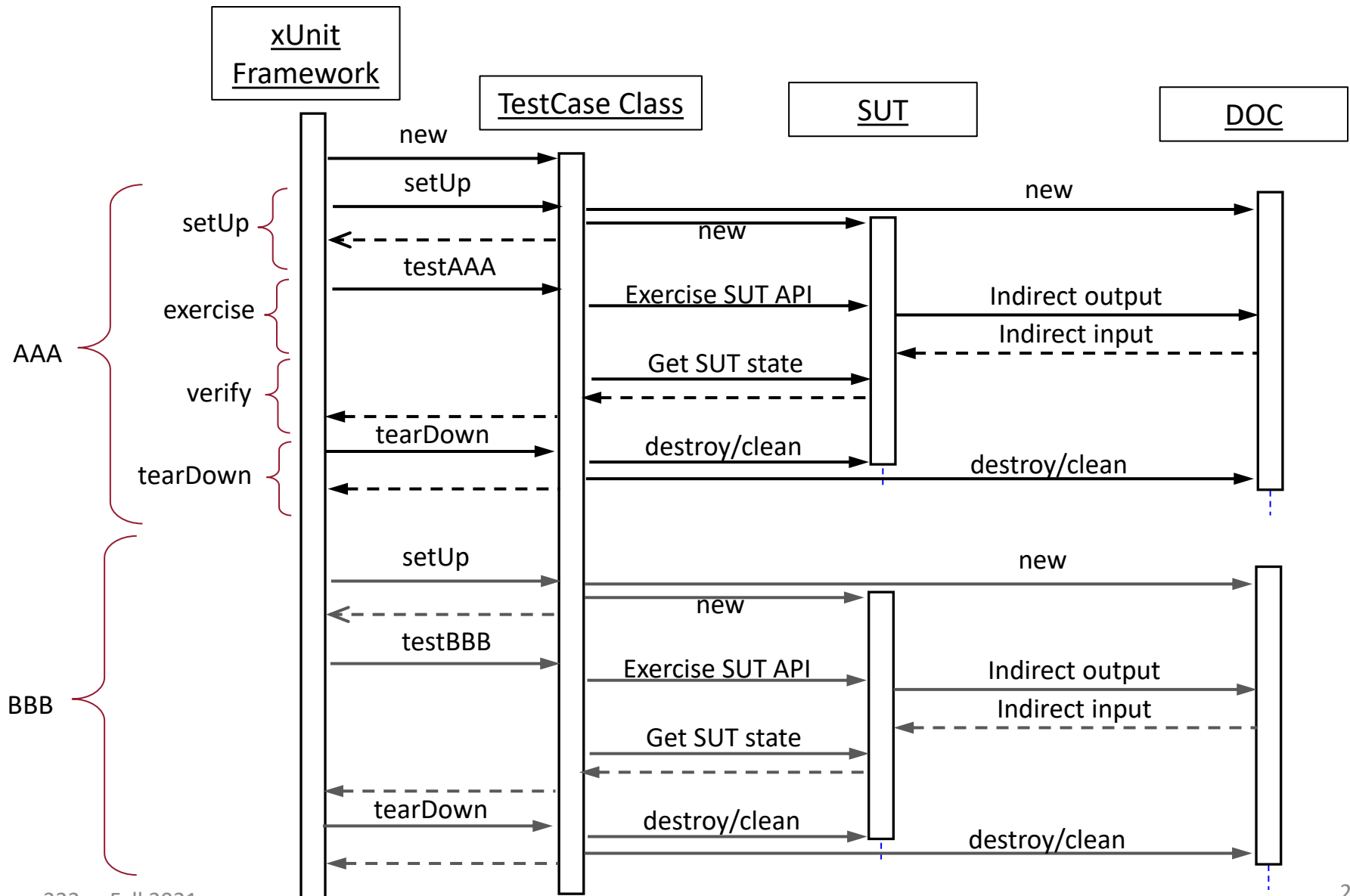
- Some examples of Test Automation Framework for xUnit:
 - JUnit (Java),
 - SUnit (Smalltalk),
 - CppUnit (C++),
 - NUnit or xUnit.Net (all .Net languages),
 - runit (Ruby),
 - PyUnit or unittest (Python)

xUNIT – Family of Test Frameworks

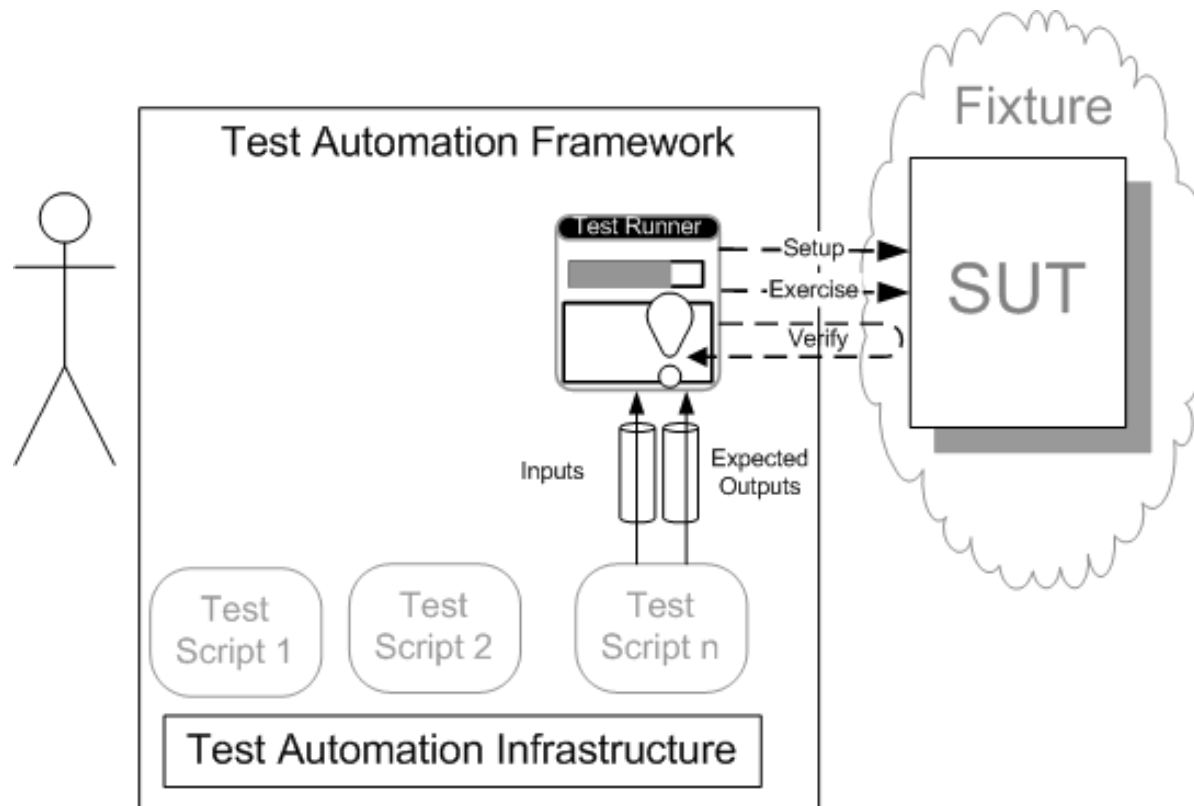
- **System under test:** The module/component/system/method we are testing
- **Depended-on Component:** Other code (not being tested) that SUT needs (also called Collaborator). E.g., database
- **Test Fixture:** SUT + DOC + Input data
- **Test Method:** The actual code of the test
- **Test Case:** A collection of tests with common purpose and setup



xUnit Sequence Diagram



xUnit Framework



xUNIT Cross-Language Terminology

- Typically tests are written as methods in classes inheriting from a built-in TestCase class
 - Such classes override “setUp” and “tearDown”
- How does xUNIT find the test methods ?
 - All methods whose name starts with “test”
 - Or, using metadata attributes
 - e.g., in Java

```
public void testSize() { ... }
```

```
@Test
```

```
public void myTest() { ... }
```

TestCase class

Name test classes similar to SUT class

```
public class VectorTest extends TestCase {  
    protected Vector fEmpty;  
    protected Vector fFull;  
    // public VectorTest();  
    @Override  
    protected void setUp() {  
        fEmpty= new Vector();  
        fFull= new Vector();  
        fFull.addElement(new Integer(1));  
        fFull.addElement(new Integer(2));  
        fFull.addElement(new Integer(3));  
    }  
    ...  
}
```

Some state to refer to the SUT instance

Typically use implicit constructor

Setup the test fixture (an instance of SUT)

Pick values that are recognizable in traces/debugging/assertions

```
public void testSize() {  
    int size= fFull.size();  
    for (int i= 0; i < 100; i++)  
        fFull.addElement(new Integer(i));  
    assertTrue(fFull.size() == 100+size);  
}
```

Some more setup
and the “exercise
code”

This is the verify
stage

- What do you see if the assertion fails ?

Assertion failed: myTest.java:150 (expected true but was false)

- A much better assertion is:

```
assertEquals(100+size, fFull.size())
```

- If this fails you see:

Assertion failed: myTest.java:150 (expected 103 but was 102)

- An even better assertion

```
assertEquals("new length", 100 + size, fFull.size())
```

- If this fails you see:

Assertion failed: myTest.java:150 (new length expected 103 but was 102)

More About Assertions

- Always put messages in assertions
- You may find it useful to use fuzzy-equality assertions:

```
assertAlmostEqual(int expected, int found, int gap)
```

- Checks “ $\text{expected} - \text{gap} \leq \text{found} \leq \text{expected} + \text{gap}$ ”

- Feel free to invent your own assertions:

```
assertStringContains(string needle, string haystack)
```

- This allows you to emit more informative errors, e.g.,
when compared to:

```
assertTrue(haystack.contains(needle))
```

At least one test per API method.

```
public void testClone() {  
    Vector clone= fFull.clone();  
    assertTrue(clone.size() == fFull.size());  
    assertTrue(clone.contains(new Integer(1)));  
}
```

```
public void testCloneEmpty() {  
    Vector clone= (new Vector());  
    assertTrue(clone.size() == 0);  
}
```

```
public void testContains() {  
    assertTrue(fFull.contains(new Integer(1)));  
    assertFalse(fEmpty.contains(new  
        Integer(1)));  
}
```

May want to test
corner cases
separately

Some developers
want to test only
one thing per test.
Some break this
rule.

OK to change the fixture.
Each test will re-setup.

```
public void testRemoveAll() {  
    fFull.removeAllElements();  
    fEmpty.removeAllElements();  
    assertTrue(fFull.isEmpty());  
    assertTrue(fEmpty.isEmpty());  
}
```

Test the error cases also

```
public void testElementAt() {  
    Integer i= (Integer)fFull.elementAt(0);  
    assertTrue(i.intValue() == 1);  
    try {  
        int j=(Integer)fFull.elementAt(fFull.size());  
        fail("Should raise an ArrayIndexOutOfBoundsException");  
    } catch (ArrayIndexOutOfBoundsException e) {  
        return;  
    }  
}
```

Be specific which
exception you
catch

Test Failures

- A test fails if
 - An assertion fails, or
 - An uncaught exception is thrown
- xUNIT runs the test method in a try ... catch

```
setUp ()  
try {  
    testAAA();  
    return true;  
} catch (Exception e) {  
    return false;  
}
```

Test stops on first assertion failure

- Continue with next test
- If you put all your assertions in one test you will not know how many of them fail

Test Tear-Down

- Consider a test that uses a shared fixture
 - E.g., external resources, global variables, files, databases
 - E.g., need to “acquire” and “release” each resource before a test
 - We do not want tests to interfere with each other

Bad

```
void setUp() {  
    Resource f = acquire();  
    Resource b = acquire();  
}  
void testAAA() {  
    use f and b  
}
```

Better

```
void setUp() {  
    Resource f = acquire();  
    Resource b = acquire();  
}  
void testAAA() {  
    try {  
        use f and b  
    } finally {  
        release(f);  
        release(b);  
    }  
}
```

Even Better

```
void setUp() {  
    Resource f =acquire();  
    Resource b =acquire();  
}  
void testAAA() {  
    use f and b  
}  
void tearDown () {  
    release(f);  
    release(b);  
}
```

Test Tear-Down Techniques

- Be extremely careful with tear down
 - If tear down is not complete, a test failure can pollute subsequent tests

Best

```
void setUp() {
    Resource f = acquire();
    Resource b = acquire();
}
void testAAA() {
    use f and b
}
void tearDown () {
    try {release(f);} catch(){ }
    try {release(b);} catch(){ }
}
```

Even better than Best

```
void setUp() {
    Vector<File> resources = new Vector();
    files.add(acquire());
    files.add(acquire());
}
void testAAA() {
    use resources
}
void tearDown () {
    for(f in resources) {
        try { release(f); } catch() { }
    }
}
```

Test Running

- There is often a way to select a subset of tests to run
- There are GUI and command-line runners:

```
public static Test suite() {  
    return new TestSuite(VectorTest.class);  
}  
public static void main (String[] args) {  
    junit.textui.TestRunner.run (suite());  
}
```

Test Automation

- Tests should be runnable by script
 - Otherwise they won't be run often
 - And will decay
 - And soon you will even forget how to set them up and run them
 - And then they are forgotten
- Tests should verify their own results without human intervention

Goals of Test Automation

- Automated tests should be repeatable
 - Same results on your machine and on somebody else's machine
- Automated tests should be robust
 - A failure should point to a bug (in the SUT or the test)
- Tests should be fast
 - Want timely feedback