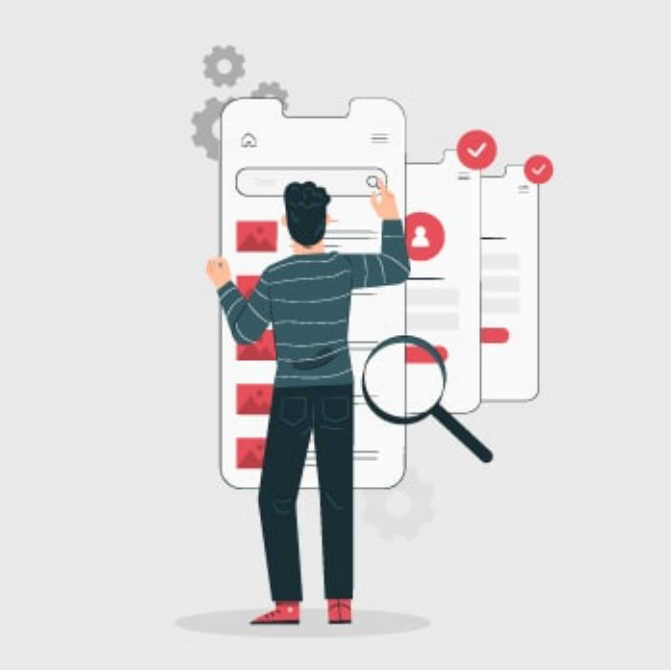
CSE 565 - Software Verification Validation and Testing
Assignment 1

Kedar Sai Nadh Reddy Kanchi
School of Computing and Augmented Intelligence
Arizona State University, Tempe
kkanchi@asu.edu

Introduction



Unit testing is a software testing method in computer programming that tests individual units of source code—sets of one or more computer program modules along with associated control data, usage procedures, and operating procedures—to determine whether they are fit for use or not.

Unit tests are typically small, isolated, and fast-running tests that cover specific functionality of the code. Software developers frequently write and run automated unit tests to ensure that a section of an application (referred to as the ”unit”) adheres to its design and behaves as expected. A single function or procedure is the most common type of unit in procedural programming, but it can also be an entire module. An entire interface, such as a class or a single method, is frequently referred to as a unit in object-oriented programming. Comprehensive tests for complex applications can be built by first writing tests for the smallest testable units, then for the compound behaviors between those. Generally these unit test cases are developed concurrently with production code by the developers themselves but are not built into the final software product. The relationship of unit tests to production code is shown in Figure 1.

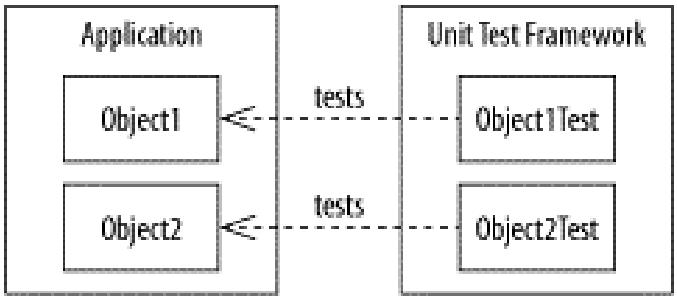


Figure 1: Production application and unit test framework

Manually performing tests and regression testing, as well as repeating the same actions over and over, is error-prone and time-consuming. Tooling helps to solve these issues. Unit testing frameworks help programmers write tests faster using a set of well-known APIs, run those tests automatically, and easily review the results.

Each test case should be run independently in order to identify any potential issues. Method stubs, mock objects, fakes, and test harnesses, among other tools, can be used to test a module in isolation. A software developer may incorporate criteria—or known good results—into the test during development to ensure the accuracy of the unit. Frameworks record tests that fail any criterion during the execution of a test case and report them in a summary.

Unit Testing Framework

A unit testing framework is a collection of tools and libraries that assist developers in writing and running automated tests for individual units of code. It includes a foundation for building tests as well as functionality to execute and report on the results of the tests. They are not only testing tools; they can also be used as development tools in the same way that preprocessors and debuggers are.

Unit testing framework libraries are used to code unit test snippets. Then the tests are executed from a separate unit testing tool or using IDE, and the test results are evaluated. Unit test frameworks can help with almost any stage of software development, such as software architecture and design, code implementation and debugging performance optimization, and quality assurance. Popular unit testing frameworks

include JUnit for Java, NUnit for .NET, and unittest for Python.

Scope of the Unit Testing Framework

The scope of a unit testing framework is to test small, isolated pieces of code, such as individual functions or methods, to ensure they work as intended.

Purpose of the Unit Testing Framework

The purpose of unit testing is to catch bugs and regressions early in the development process, allowing developers to identify and fix problems before they are integrated into the main code base.

Example of the usage of the Unit Testing Framework

An example of a unit testing framework is JUnit for Java. To use JUnit, a developer would write test cases, which are individual methods that exercise a specific piece of code and make assertions about its expected behaviour. These test cases are then executed by the JUnit framework, which reports any failures or errors.

An example of usage for a unit testing framework is to test the logic of a function that performs a specific calculation, such as a function that calculates the area of a circle. The developer would create a test case that calls the function with known input and checks that the output matches the expected value. If any of the tests fail, the developer would know that there is an issue with the code and would be able to fix it before committing the changes to the main code base.

Utilization of the Unit Testing Framework by a developer

A developer would utilize a unit testing framework by following these general steps:

- **Write test cases**: The developer writes test cases, which are individual methods that exercise specific units of code and make assertions about its expected behaviour. Typically, the code for these test cases is written in the same language as the code being tested.
- **Organize test cases**: The developer groups related test cases together in test suites by organizing the test cases into groups. Test runners are another tool the developer can use to automate the testing procedure.
- **Integrate the tests with the framework**: By running the tests as part of their local development process or as part of a continuous integration pipeline, the developer incorporates the test cases into their development workflow.
- **Run the tests**: The framework's command-line tool or IDE plugin would be used by the developer to run the tests. The test methods would be executed by the framework, which would also report any errors or failures.
- **Review and fix the test results**: After reviewing the test results, the developer would correct any ineffective tests. The developer will know the code is operating as intended once all tests succeed.

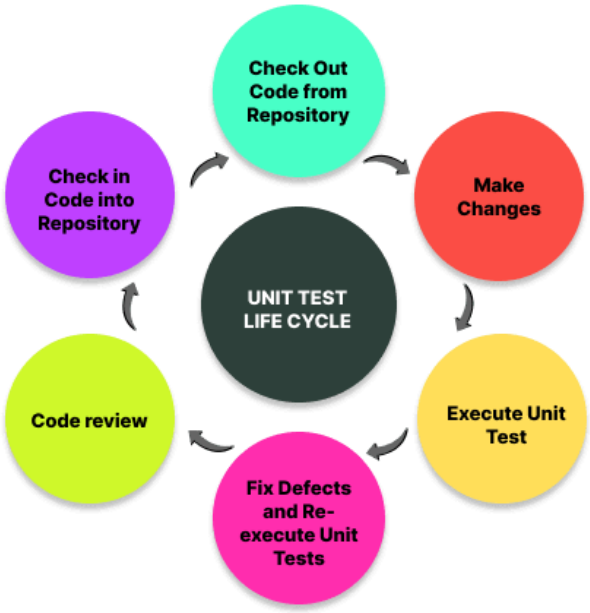


Figure 2: Unit testing life cycle

- **Repeat the process**: The developer repeats the above steps as they continue to develop the code, adding new test cases as necessary.
- **Continuously run the tests**: The developer would continuously keep re-running the test suite as they develop new features and fix bugs to ensure that the code is working as intended and catch issues early and that any new changes have not introduced regressions.

Benefits of using a Unit Testing Framework

A unit testing framework provides several benefits, including:

- **Early bug detection**: Unit tests are run automatically and frequently, which helps developers catch bugs and regressions early in the development process before they are integrated into the main code base. This can save time and resources in the long run by preventing the need for costly and time-consuming debugging later helping to ensure that the final product is of the highest quality.
- **Improved code quality**: Unit testing helps developers to create code that is more modular, more maintainable, and less prone to bugs. By writing test cases that test individual units of code, developers can ensure that each unit works as intended and that changes to the code do not introduce regressions.
- **Increased confidence in code changes**: By having a suite of automated tests that can be run quickly, it provides developers with a safety net that allows them to make changes to the code base with confidence. By running the test suite after making changes, developers can be sure that they have not introduced new bugs or regressions.

- **Improved documentation**: Unit tests serve as living documentation of the code's behaviour, making it easier for other developers to understand how the code works and how it should be used.
- **Faster development**: Writing test cases can save time in the long run by catching bugs early, reducing the need for debugging, and making it easier to make changes to the code base.
- **Improved collaboration**: When multiple developers are working on a project, unit tests can help to ensure that code changes made by one developer do not break code written by another.
- **Easy to maintain**: Unit tests are easy to maintain since they are automated and can be run by developers or by a continuous integration pipeline.
- **Facilitation of Continuous Integration and Continuous Deployment (CI/CD) workflows**: Unit tests can be integrated into CI/CD pipelines, by running the test suite automatically as part of the pipeline and preventing code changes that break the test suite from being deployed to production.

Overall, by making it easier to find bugs early on, enhancing the code's structure and maintainability, and boosting confidence in the code changes, unit testing frameworks help to ensure the quality and dependability of code.

Comparison Between different Unit Testing Frameworks

This section compares the JUnit and Jest Unit Testing Frameworks.

Common capabilities between JUnit and Jest

- **Test discovery**: Both JUnit and Jest support discovering and running tests automatically.
- **Test case classes**: Both frameworks provide a way to organize tests into test case classes.
- **Assertions**: Both frameworks provide a set of assertions for testing the expected behaviour of the code.
- **Test execution**: Both frameworks provide a way to execute the tests and report the results.
- **Test isolation**: Both frameworks provide a way to isolate the tests from one another, ensuring that the execution of one test does not affect the execution of other tests.
- **Grouping**: Both frameworks allow you to organize your tests into groups. You can declare as many test suites as you want in Jest. A describe block is used to group tests together. You can use JUnit to create a test suite that combines a few unit test cases and runs them all at once.

Differences between JUnit and Jest

- **Language**: JUnit is a unit-testing framework for Java while Jest is a unit-testing framework for JavaScript applications and React components.
- **Test runner**: JUnit does not support snapshot testing and does not offer a built-in test runner or test coverage reports. Jest, on the other hand, supports snapshot testing and has a built-in test runner as well as test coverage reports.
  - JUnit requires a build tool such as Maven or Gradle to run the tests, whereas Jest can run the tests directly from the command line.
- **Mocking and spying**: Jest provides built-in support for mocking and spying on functions while JUnit requires additional libraries to be used.
- **Generators**: You can use JUnit-quick check to generate test data but when using Jest you cannot generate test data to work with.
  - Data generators generate input data for test. The test is then run for each input data produced in this way.
- **Asynchronous testing**: Jest provides built-in support for testing asynchronous code while JUnit requires additional libraries to be used.
- **Performance**: Jest has a faster test execution because it runs the tests in parallel by default, JUnit does not provide parallel test execution by default.
- **Matchers**: Jest uses matchers to perform assertions, which makes it easy for developers to read and understand the test results. JUnit uses assert statements for performing assertions.
- **Community and support**: Jest has a larger community and more active development which makes it more up-to-date with the latest JavaScript features and trends, JUnit is widely used and supported in the Java community but it's not as active as Jest.

References

- Prof. James Collofello, CSE 565 Lecture Videos, 2023
- Prof. Ayca Tuzmen, CSE 565 Lectures, 2023
- Unit testing
- What is Unit Testing and Why Developer Should Learn It
- Unit Test Frameworks by Paul Hamill
- Why Is Unit Testing Important in Software Development?
- Jest Tutorial: Complete Guide to Jest Testing
- Jest Framework
- JUnit Tutorial — Testing Framework for Java
- What are the differences between Jest and JUnit?
- HeapSort Algorithm
- Jest Crash Course - Unit Testing in JavaScript

```javascript
// JavaScript program for implementation of Heap Sort

const heapsort_algorithm = {

    heapsort_code: function(arr) {
        var N = arr.length;

        // Build heap (rearrange array)
        for (var i = Math.floor(N / 2) - 1; i >= 0; i--)
            heapify(arr, N, i);

        // One by one extract an element from heap
        for (var i = N - 1; i > 0; i--) {
            // Move current root to end
            var temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
        return arr;
    }


}

// To heapify a subtree rooted with node i which is an index in arr[].
// n is size of heap
const heapify = (arr, N, i) => {

    var largest = i; // Initialize largest as root
    var l = 2 * i + 1; // left = 2*i + 1
    var r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        var swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
}

module.exports = heapsort_algorithm;

```

```
const heapsort = require("./heapsort");

test('heapsort test case 1', () => {
    expect(heapsort.heapsort_code([12, 11, 13, 5, 6, 7])).toEqual([ 5, 6, 7, 11, 12, 13 ]);
});

test('heapsort test case 2', () => {
    expect(heapsort.heapsort_code([4,1,3,9,7])).toEqual([ 1,3,4,7,9 ]);
});

test('heapsort test case 3', () => {
    expect(heapsort.heapsort_code([10,9,8,7,6,5,4,3,2,1])).toEqual([ 1, 2,3,4,5,6,7,8,9,10 ]);
});

test('heapsort test case 4', () => {
    expect(heapsort.heapsort_code([])).toEqual([]);
});

test('heapsort test case 5', () => {
    expect(heapsort.heapsort_code(["banana", "apple", "mango", "pineapple", "orange"])).toEqual(["apple", "banana", "mango", "orange", "pineapple"]);
});

test('heapsort test case 6', () => {
    expect(heapsort.heapsort_code(["CAB", "ACB", "BCA", "ABC", "CBA", "BAC"])).toEqual(["ABC", "ACB", "BAC", "BCA", "CAB", "CBA"]);
});

test('heapsort test case 7', () => {
    expect(heapsort.heapsort_code([5,3,3,1,5,3,2])).toEqual([1,2,3,3,3,5,5]);
});

test('heapsort test case 8', () => {

 expect(heapsort.heapsort_code(['5','3','3','1','5','3','2'])).toEqual(['1','2','3','3','3','5','5']);
});

test('heapsort test case 9', () => {
    expect(heapsort.heapsort_code(["M4dkbH6NjJ", "1XSRJpa60H", "hxWwGbcMJM", "PjMgcuKxsO", "AWVQ1I3pZs", "6I4vODbmk8", "LjeMBnEqr1", "lnJnShEBE6", "RJzCi0UcRE", "IOvB93CeP9"])).
    toEqual(["1XSRJpa60H", "6I4vODbmk8", "AWVQ1I3pZs", "IOvB93CeP9", "LjeMBnEqr1", "M4dkbH6NjJ", "PjMgcuKxsO", "RJzCi0UcRE", "hxWwGbcMJM", "lnJnShEBE6"]);
});

test('heapsort test case 10 - checking to fail', () => {
    expect(heapsort.heapsort_code([4,1,3,9,7])).toEqual([ 1,7,4,3,9 ]);
});
```

```
PS D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Veri
fication Validation and Testing\ASSIGNMENTS\ASSIGNMENT1\code> npx jest --coverage
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
 FAIL  ./heapsort.test.js
  √ heapsort test case 1 (4 ms)
  √ heapsort test case 2 (1 ms)
  √ heapsort test case 3 (1 ms)
  √ heapsort test case 4
  √ heapsort test case 5 (1 ms)
  √ heapsort test case 6 (1 ms)
  √ heapsort test case 7
  √ heapsort test case 8
  √ heapsort test case 9
  × heapsort test case 10 - checking to fail (4 ms)

  ● heapsort test case 10 - checking to fail

    expect(received).toEqual(expected) // deep equality

    - Expected  - 2
    + Received  + 2

      Array [
        1,
    -   7,
    -   4,
        3,
    +   4,
    +   7,
        9,
      ]

      39 |
      40 | test('heapsort test case 10 - checking to fail', () => {
    > 41 |     expect(heapsort.heapsort_code([4,1,3,9,7])).toEqual([ 1,7,4,3,9 ]);
         |                                                 ^
      42 | });

      at Object.toEqual (heapsort.test.js:41:49)

-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |     100 |      100 |     100 |     100 |
 heapsort.js |     100 |      100 |     100 |     100 |
-------------|---------|----------|---------|---------|-------------------
Test Suites: 1 failed, 1 total
Tests:       1 failed, 9 passed, 10 total
Snapshots:   0 total
Time:        1.958 s, estimated 2 s
Ran all test suites.
PS D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\ASSIGNMENT1\code>
```