JavaScript Part 4
JavaScript Testing

Samuel Cho, Ph.D.

NKU ASE/CS

Software Testing

What is not Software Testing?

- It is not Debugging; Debugging is an action to correct the issue found from the test.
- It is not Unittest; Unittest is a testing tool.
- It is not TDD (Test Driven Development); TDD is a testing process.
- It is nothing more than ...

What is Software Testing?

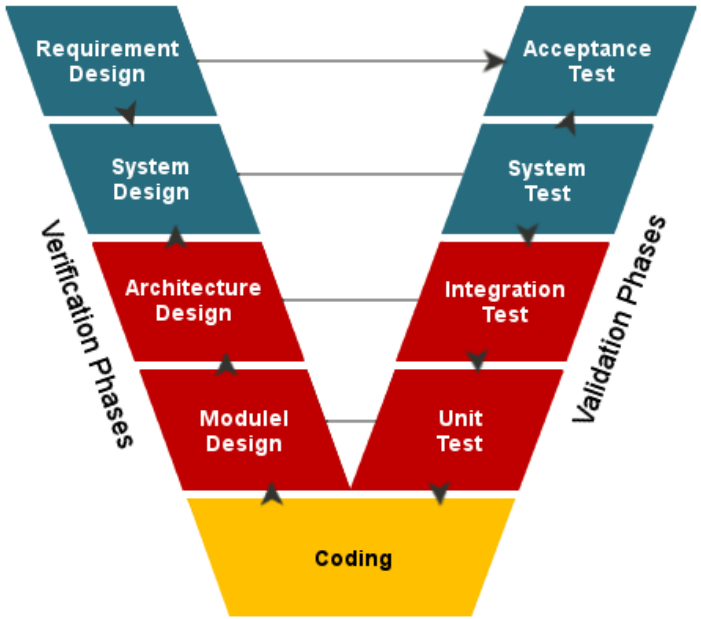
- Software testing is comparing 'What is' to What should be.'
- Software testing cannot prove the absence of bugs; it can only prove their existence.

- So, passing tests does not mean the absence of bugs.
- To prove the absence of bugs, we need a different mathematical approach: formal verification.
- However, we only focus on testing in this course.

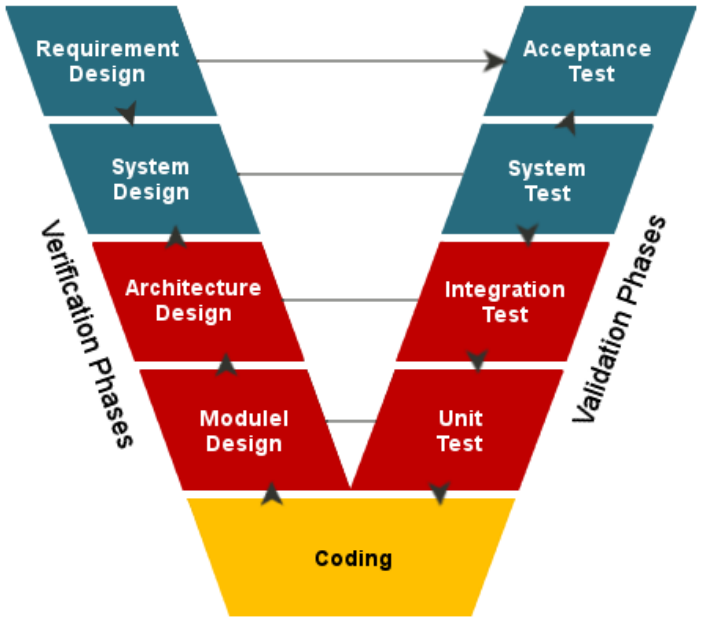
- To accomplish the goal, we should know 'what should be.' In other words, we should have "expected values or behaviors."
- We also know the 'what is.' In other words, we should execute some part of the code to get values or do something.
- We should compare them; a lot of cases in various circumstances.
- So, we need to automate the process.

What to Compare?

- In the V model (software process), we check if each design result shows expected values or behaves as expected.

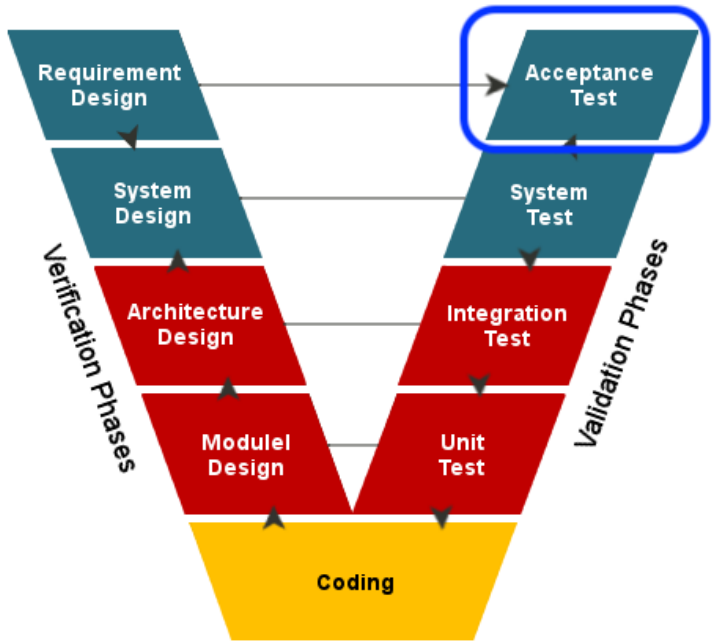


- Notice that we verify each design phase and validate the design by comparing values/behaviors.



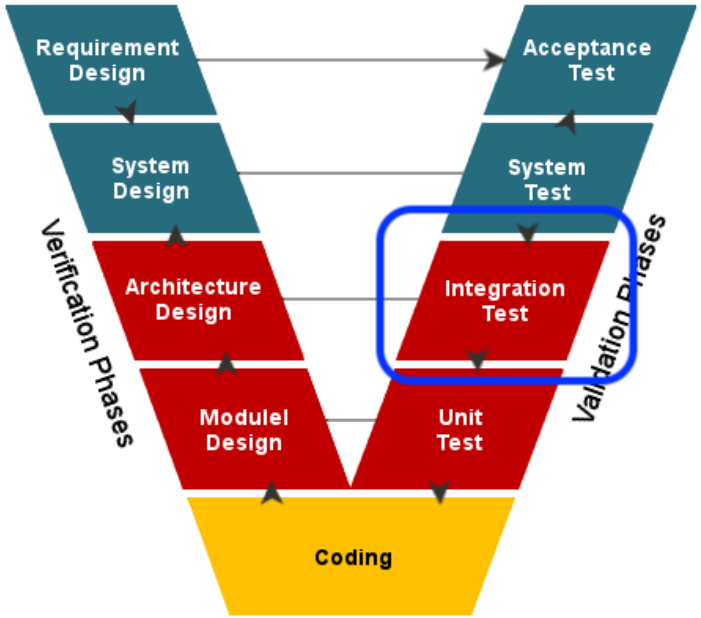
Acceptance Tests

- We identify users' requirements.
- Acceptance tests compare if users' requirements are implemented.



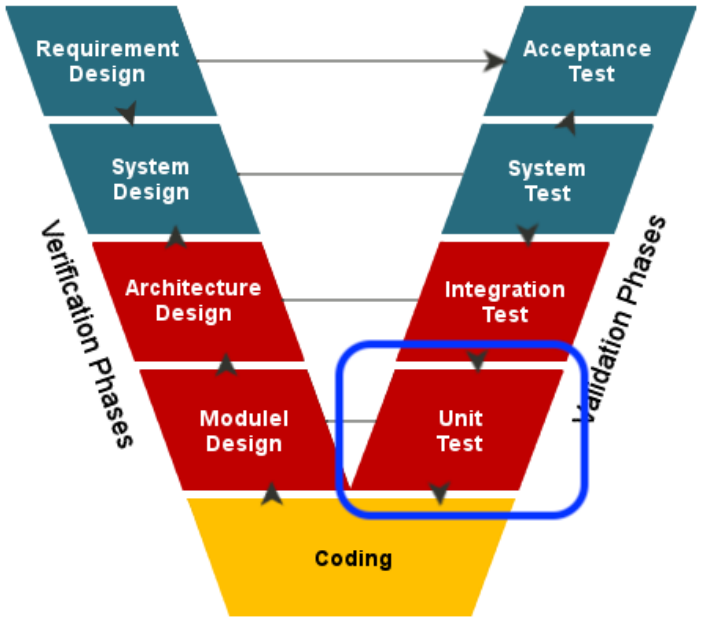
Integration Tests

- From users' requirements, we architect a system with multiple modules.
- Integration tests compare the values from the interaction between modules.



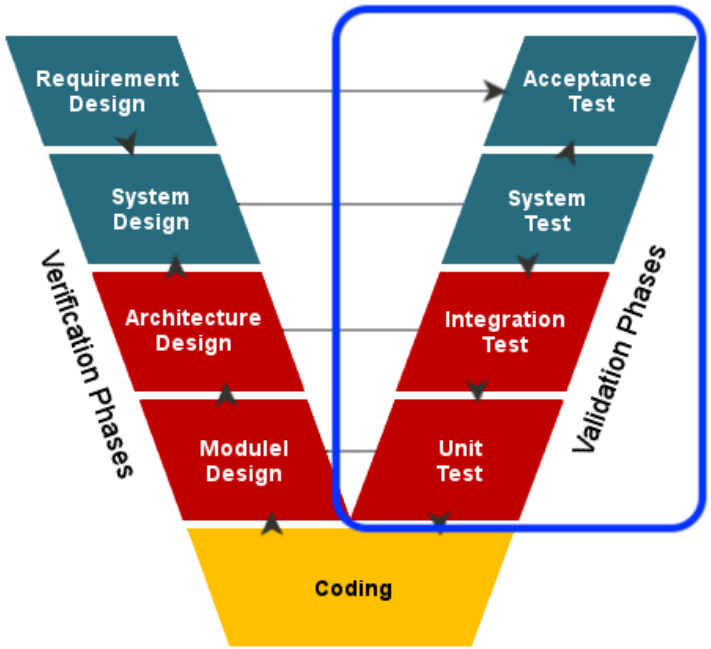
Unit Tests

- We design modules and their interfaces.
- Unit tests compare the values from the module interfaces.



Regression Tests

- We keep refactoring the code.
- Regression tests check if any changes do not impact the existing code base.



Jest

Jest

- Jest (https://jestjs.io) is one of JavaScript Test Frameworks.
- We can automate JavaScript testing with Jest.

Installation with Node.js

- We execute the command ‘npm install jest’.
- Then the package.json has the Jest library as a dependency.
- When we need to install Jest as a command line tool, we can use ‘npm install -g jest’.

Code #

```
"dependencies": {
  "jest": "^27.5.1"
},
```

Run Jest as a Command Line Tool

- We can execute the jest command to find all the tests and run them in the current directory.

Command > Results

```
> jest
PASS  ./sum.test.js
adds 1 + 2 to equal 3 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.2 s, estimated 1 s
Ran all test suites.
```

jest.config.js File

- To execute jest for any directory, we should make the jest.config.js in the root of the project.
- This is an example of the file.

Code #

```
1 module.exports = {
2     testEnvironment: 'node', // or 'jsdom' for browser
      ↪ environment
3     testMatch: [...],
4     moduleFileExtensions: ['js', ...]
5   };
```

- We can specify the test to run (line 1).
- We can use a pattern to execute tests.
- We can execute all the javascript files (js) in the simple directory (line 2).

Command > Results

```
jest -i simple.tests.js
jest -i simple/*.js
```

Use NPM to run Jest

- We can add a "script" section in the ‘package.json’ file.
- In the section, we can choose what command is executed when we give command “test.”

Code #package.json

```
1 {
2   "scripts": {
3     "test": "jest"
4   },
5   ...
6 }
```

- We can run ‘npm test’ to invoke jest.
- We can change the command to execute by changing the "test" section.

Command > npm test

```
> npm test
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.253 s
```

Unit Tests

Step 1: Make a Module (src/sum.js)

Use the tests/unit project as an example.

- Module is a basic unit that we design to implement features.
- In this example, the arith module has one API to export.

Code #arith.js

```
1 function add(a, b) {
2   return a + b;
3 }
4 module.exports = add;
```

Step 2: Make a Unit Test (tests/arith.test.js)

- We make a unit test (arith.test.js) that checks the APIs of the arith module.
- In Jest, we use the test() function that compares the expected value (3) with the return value from the API (add(1,2)).

- This code shows the unit test code.
- We need to use the require() function to use the APIs (line 1).

Code #

```
1 const add = require('../src/arith');
2
3 test('adds 1 + 2 to equal 3', () => {
4   expect(add(1, 2)).toBe(3);
5 });
```

Step 3: Set Up Jest

- Make a package.json and run ‘npm install jest.‘ to install the Jest library.
- Notice that in the ‘script’ section, we choose ‘jest’ as the command to be invoked.
- If we make the ‘jest.config.js,’ we can control the jest’s behavior.

Code #

```
1 {
2   "scripts": {"test": "jest"},
3   "dependencies": { ... }
4 }
```

Run Jest

- We can use ‘npm test’ in the command line.
- We can also execute the ‘jest‘ to run all the tests.
- Jest automatically calls all the test files and shows the results.

Run Single Jest Test

- When we want to run single test, we use 'jest -i tests/arith.test.js.'
- We can change the scripts section to use 'npm test.'

Code #

```
1 "scripts": {
2   "test": "jest -i sum.test.js"
3 },
```

When We Find Bugs

- When we find a bug, we should (a) fix the bug and (b) update unit tests to catch the missed bug.

Integration Tests

Two Modules

Use tests/integration for this example.

- We create two modules, arith and logic, in the src directory.
- We also make two unit tests in the tests directory.

Integrating the Two Modules

- We need to check if the two modules are working as expected, so we need integration tests.
- In this example, we make the check1() function for the integration tests.

Code #

```
1 function check1(x, y) {
2   var r1 = arith.add(x, y);
3   var r2 = arith.add(x, y*10)
4   var r3 = logic.all_more_than_5(r1, r2);
5   return r3;
6 }
```

- We can use the same Jest routine that we used for unit tests for the integration tests.

Code #

```
1 test('integration test 1', () => {
2   expect(check1(5, 1)).toBe(true);
3 });
```

npm test

- When we run the 'npm test,' Jest runs all three tests in the tests directory.
- We can run only the integration test using 'jest -i tests/arith_logic.test.js.'

Command > Results

```
 PASS  tests/logic.test.js
 PASS  tests/arith_logic.test.js
 PASS  tests/arith.test.js

Test Suites: 3 passed, 3 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.185 s, estimated 1 s
Ran all test suites.
```

Acceptance Tests

Requirements and Acceptance Tests

- Acceptance Tests are checking we implemented all the requirements.
- Some of the requirements can be tested automatically using tools, including Jest.
- Some of the requirements need to be tested manually.

- Testing is expensive, so it is hard and time-consuming to test 100% test coverage.
- However, we should make sure all the requirements are fully tested (manually or automatically) before shipping the software product to the users.

Regression Test

Regression

- Regression tests are to make sure no existing code is broken with any changes.
- As a result, regression tests are executed whenever we make changes to existing code.

Regression Tests Execution

- So, for the regression tests, we run existing tests.
- When the tests are large or time-consuming, we need to design the regression tests to execute only the necessary tests that are selectively executed through change impact analysis (CIA).

- CIA is a complex process, so we run all the tests for the regression tests.
- It is too much of a burden for manual tests, so making automatic tests, not manual, is important in software testing.

Regression Test with Exec()

- When we run jest, it executes all the test files.
- However, we may need to control the test execution depending on the situation.
- In this case, we can use the 'child_process' module.

- In this example, we execute a program (cat tests/logic.test.js | wc -l) and run code depending on the results.
- Notice that when we have an error (error !== null) we can run necessary code.

Code #

```
1 var exec = require('child_process').exec;
2 exec('cat tests/logic.test.js | wc -l',
3     function (error, stdout, stderr) {
4         console.log('stdout: ' + stdout);
5         console.log('stderr: ' + stderr);
6         if (error !== null) {
7             console.log('exec error: ' + error);
8         }
9     });
```

Implementation Example of Regression Tests

Use the tests/regression as an example.

- When we need to manage the execution of jest, we can make a test application.
- In this example, we execute each jest test one by one, check the results, and run the necessary code.

Code

```
1  var exec = require('child_process').exec;
2
3  const add = require('../src/arith').add;
4  const all_more_than_5 = require('../src/logic').
       ↪ all_more_than_5;
5
6  exec('jest -i tests/arith.test.js',
7      function (error, stdout, stderr) {
8        if (error != null) {
9          console.error('exec error' + error)
10       }
11       console.log('Unit Test: arith.test.js OK')
12     });
13 exec('jest -i tests/logic.test.js', ...)
14 exec('jest -i tests/arith_logic.test.js', ...)
```