Automated Testing

Lessons From 20 Years of Practice

About This Talk

This talk was originally delivered as a 40-minute live presentation.

It was presented to students of Keyin Technical College’s Software Development program in 2023 as part of a series of industry tech talks.

About Me

Lifelong automated test enthusiast

Writing code test-first (on and off) since 2002

Test mentor and evangelist at 5 different companies (Celtx, Verafin, Bell, ATS, DPSI)

Implemented a best-in-class automated testing system at Celtx

Terminology: Test components

Test - code that runs other code and makes assertions about its behaviour

Assertion - code that checks a boolean condition and throws an exception when the condition is false

Validation - assertions that, taken together, indicate a system is running as expected

System Under Test - the unit or units being exercised by a test

Terminology: Test types

Unit test - common usage: Any test that focuses on one class or function

Unit test - my usage: A test that focuses on a single operation ***in isolation***

Integration test: A test that validates an operation that validates an operation comprised of multiple sub-operations

Acceptance test: A test that validates something obvious from outside the system

End to End test: A test that simulates user interaction and validates the same things a user would

Unit Tests

Why the difference?

- Anything can be a unit, but TDD practitioners reserve that word for the most granular level of code
- Unit tests, defined this way, are not “tests” in the usual sense - they do not perform **system validation**

If they’re not tests, why write them?

- Specify the behaviour of an operation without reference to other operations
- Document the expected usage and output of a unit of code

Unit Tests: Example

Test

```
def test_json_table_to_html():
        json = {
                “data”: [{
                        “Prop1”: “val1”,
                        “Prop2”: “val2”
                }]
        }
        Html = json_to_html_table(json)
        assert(html ==
        “<table>
                <tr><th>Prop1</th><th>Prop2</th></tr>
                <tr><td>val1</td><td>val2</td</tr>
        </table>”)
```

Unit

```
def json_to_html_table(json:object):
        table= “<table>CONTENT</table>”
        Hdr = “”
        Data = json[“data”]
        for key in Data.keys():
                Hdr = hdr + f’<th>{el}</th>’
        Hdr = f’<tr>{Hdr}</tr>
        Rows = Hdr
        for row in Data:
                Row = “”
                for key in Data.keys():
                        Row = row + f’<td>{row[key}]</td>’
                Row = f’<tr>{Row}</tr>
        Rows = Rows + Row
        Return table.replace(CONTENT, Rows)
```

Integration Tests

What are they?

- Combine a small number of units in a common operating configuration
- First level of validation - a small number of units working together
- Still isolate the system under test from other parts of the system

Why write them?

- Much easier to write than higher-level tests
- Validate useful outputs
- Provide a second level of contract enforcement

Acceptance Tests

What are they?

- Tests that validate an entire system
- Typically include caveats - not testing the GUI, for example, or not using a live server

Why write them?

- Provide near-user-level system validation
- Sometimes it is difficult to validate the system at the user level

End to End (E2E) Tests

What are they?

- Black box tests that validate more or less the way a user would
- Run against a system that is “production-like”

Why write them?

- Validates the system at the level that is most important
- Streamlines the process of regression testing full releases
- Frees up manual testing for exploratory testing

Test-Driven Development

What is it?

- Red, Green, Refactor
- Established by Kent Beck and other signatories to the Agile Manifesto

Why use it?

- Proves code is functioning as intended throughout development
- Helps ensure code is resilient to various kinds of change

Red-Green-Refactor: Red

Test

```
Def test_dothing():
        input = [“input”, “out”]
        expected_out = “output”
        assert(dothing(input) == expected_out)
```

Unit

```
Def dothing(input):
        return “”
```

Test compiles but assertion fails

Red-Green-Refactor: Green

Test

```
Def test_dothing():
        Input = [“input”, ”out”]
        Expected_out = “output”
        assert(dothing(input) == expected_out)
```

Unit

```
Def dothing(input):
        return “output”
```

Test passes but there is duplication of work

Red-Green-Refactor: Refactor

Test

```
Def test_dothing():
    Input = [“input”, ”out”]
    Expected_out = “output”
    assert(dothing(input) == expected_out)
```

Unit

```
Def dothing(input):
    return input[0].replace(“in”, input[1])
```

Test passes and duplication has been reduced

Demo:
TDD in Realtime

Further Reading

Common Objections

1. Tests Break too easily
    a. Tests break for 2 reasons:
        i. The behaviour has changed - the test SHOULD break*
        ii. The test was brittle - the test should change to become more resilient
            1. This often means the system under test changes in a similar way
2. Tests take too much time to write
    a. This indicates either
        i. The test is doing too much**
        ii. Code isn't being tested in other ways, meaning the risk of change is very high

***: This change could be handled with a new test and new component, preserving the old, unbroken version**

****: Reducing test complexity is a talk in itself, but Single Responsibility and One Assertion Per Test both try to tackle this issue**

Further Terminology

Fixture - A set of tests that assume a common starting point and commonly use shared code

Suite - A set of tests that collectively test a module or system

Coverage - A method of measuring how many possible paths within the system are exercised by automated tests

Exploratory Tests

What are they?

- Typically manual tests
- Aimed at testing edge conditions within the system

Characterization Tests

What are they?

- Tests against 3rd party code
- Exercise the API in well-defined ways that reflect actual intended usage

Why write them?

- Establish known conditions for valid use
- Find problematic cases and boundary conditions
- Detect functional changes when upgrading to a new version

Beyond TDD

Various folks have tried to extend TDD into whole-system frameworks

BDD (Behaviour-driven development)

- Starts at the highest level of interest (E2E, acceptance, integration, unit)
- Progressively refines the internal implementation using recursive BDD
- Typically uses English-like syntax

Visual Regression Testing

- E2E tests that validate using rendered output
- Often uses Machine Learning and Computer Vision
- Typically includes manual validation when mismatches are detected