Handling Duplicate Code

Dror Helper

@dhelper blog.drorhelper.com

Overview

- Code duplication in automated tests
- DRY vs. DAMP
- Using test fixtures
- Sections
- Data driven tests
- BDD-style test cases

Unit Tests vs. Integration Tests

Unit Tests

Focused

Isolated

Fast

Integration Tests

Extensive

Depends on Environment

Usually slower

Duplication in unit tests

- Initializing test subject
- Common operations

Duplication in integration tests

- Creating environment
- Common operations
- Cleanup

DRY

Don’t Repeat Yourself

Avoid Duplication

Increases maintainability

Isolate change

vs

DAMP

Descriptive And Meaningful Phrases

Promote code readability

Reduce the time it takes to read and understand the code

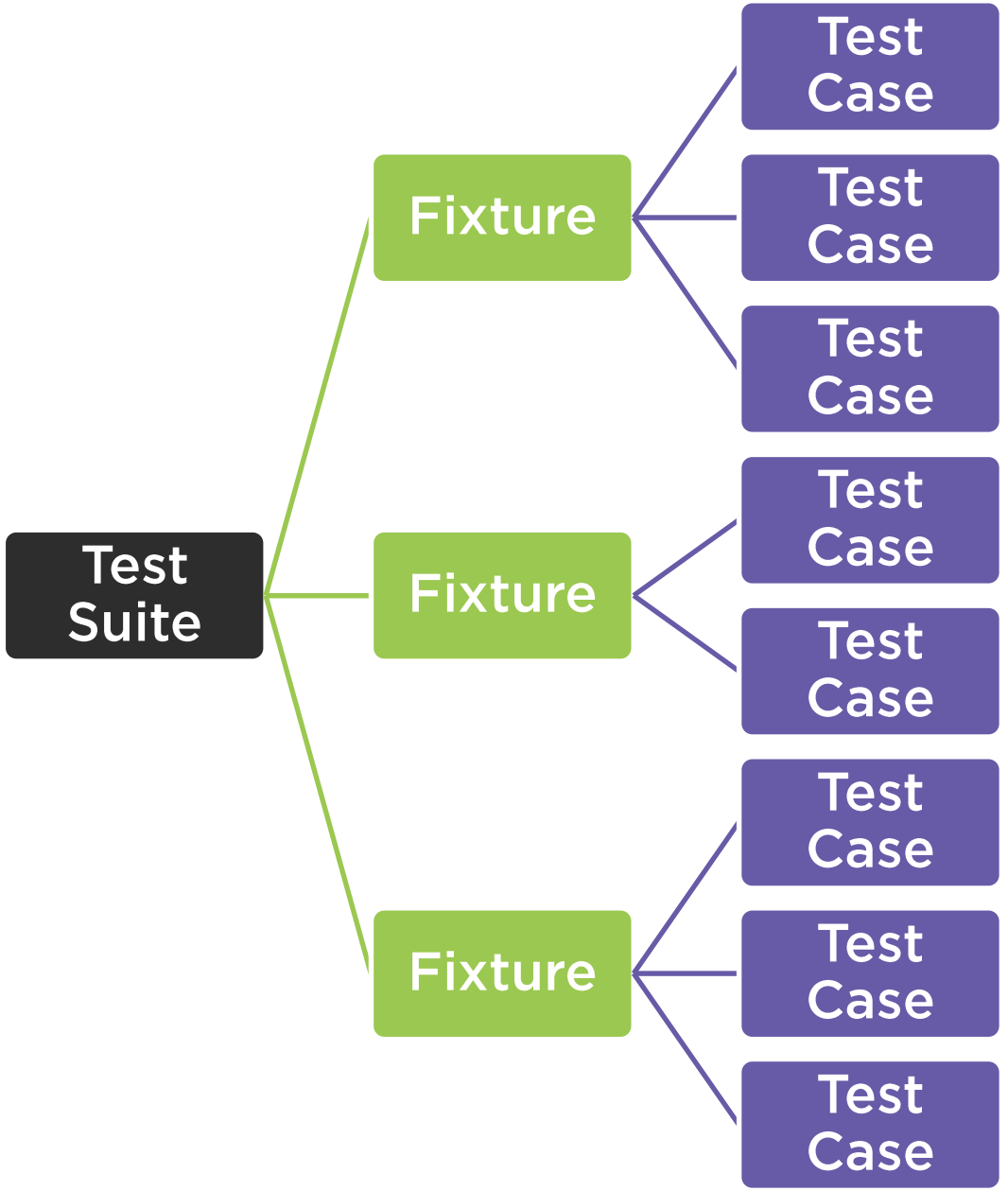
DAMP > DRY

* In unit tests

Test Suite == exe/dll

Fixture == class

Test Case == method



Creating Test Fixtures

```
class MyFixture {
    MyFixture()
    { // Common setup code }
    ~MyFixture()
    { // Common teardown code }
};

TEST_CASE_METHOD(MyFixture, "Test1") {
    ...
}
```

Using Test Fixtures

Reduce code duplication

- Preconditions
- Cleanup after each test
- Provide common operation
- Common state/environment

Group code in logical units

- Group related tests
- Use inheritance for utility code

Hide uninteresting code

- Usually in Integration tests

Demo

Using test fixtures

The Problem with Using Fixtures

Divided test logic

Hard to read and understand

Difficult to fix failures

Hides some of the test's logic

Setup/Teardown for all tests

Increased complexity

Violate SRP

Shared logic between tests

When to Use Test Fixtures

Integration tests

Create hierarchy

Introducing SECTIONS

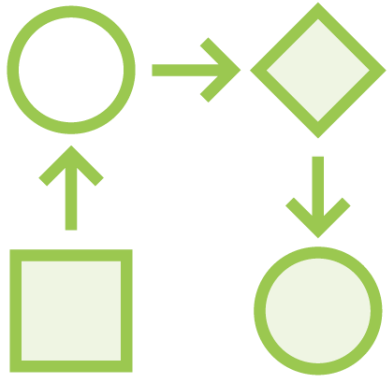
```
TEST_CASE(“This is a test case”) {
    // Initialization

    SECTION(“Test section 1"){
        // Test code
    }
    SECTION(“Test section 2"){
        // Test code
    }
```

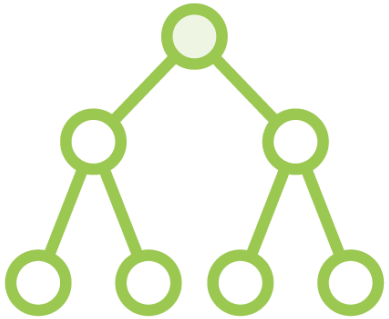
The Benefits of Using SECTIONS

Each section executed independently

Enable reuse



Easy to read and maintain



Supports nested sections

Demo

Using SECTIONS

```
TEST_CASE("Basic Generators") {
    auto n = GENERATE(1, 2, 3);
    auto c = GENERATE('a', 'b');
```

Data Generators

Run the same workflow with different values

Tests runs == once for each combination

Provided Generators

Fundamental generators

- SingleValueGenerator<T>
- FixedValuesGenerator<T>

Modify generators

- FilterGenerator<T, Predicate>
- TakeGenerator<T>
- RepeatGenerator<T>
- MapGenerator<T, U, Func>
- ChunkGenerator<T>

Specific Purpose

- RandomIntegerGenerator<Integral>
- RandomFloatGenerator<Float>
- RangeGenerator<T>
- IteratorGenerator<T>

Using Provided Generators

```
GENERATE(1, 2, 3);

GENERATE(values({1, 2, 3}));

GENERATE(as<std::string>{}, "a", "b", "c");


GENERATE(repeat(2, range({ 1, 100 })));

GENERATE(take(10, random(1, 100)));


GENERATE(filter([](int val) { return val % 2 == 0; }, range(1, 100)));

GENERATE(map([](int val) { return std::to_string(val); }, range(1, 100)));
```

Creating Custom Generators

```
class MyGenerator : public IGenerator<int> {
    int const& get() const override { ... }
    bool next() override { ... }
}

GeneratorWrapper<int> myGen(a1, a2, ...) {
    return GeneratorWrapper<int>(
        unique_ptr<IGenerator<int>>(new MyGenerator(a1, a2, ...)));
}
```

Demo

Using built-in generators

Creating your own generator

Type Parametrized Test Cases

```
TEMPLATE_TEST_CASE("template test", "[Tag1]", int, std::string, ...)
{
    std::vector<TestType> v( 5 );
    ...
```

```
-------------------------------------------------------------------------------
template test - int
-------------------------------------------------------------------------------
C:\Users\drorh\source\repos\Catch2Playground\Source1.cpp(17)
...............................................................................

C:\Users\drorh\source\repos\Catch2Playground\Source1.cpp(21): FAILED:
  REQUIRE( v.size() == 0 )
with expansion:
  5 == 0

-------------------------------------------------------------------------------
template test - std::string
-------------------------------------------------------------------------------
C:\Users\drorh\source\repos\Catch2Playground\Source1.cpp(17)
...............................................................................

C:\Users\drorh\source\repos\Catch2Playground\Source1.cpp(21): FAILED:
  REQUIRE( v.size() == 0 )
with expansion:
  5 == 0

===============================================================================
test cases: 2 | 2 failed
assertions: 2 | 2 failed
```

Additional Type Parameterized Test Cases

```
TEMPLATE_PRODUCT_TEST_CASE("name", "[tag1]...",
                           (std::vector, std::list), (int, float)) { ... }
```

```
using MyTypes = std::tuple<int, char, float>;

TEMPLATE_LIST_TEST_CASE("name", "[tag1]...", MyTypes) { ... }
```

```
TEMPLATE_TEST_CASE_SIG("name", "[tag1]...", signature, type1...)
```

“The deeper I got into TDD, the more I felt that my own journey had been less of a wax-on, wax-off process of gradual mastery than a series of blind alleys…”

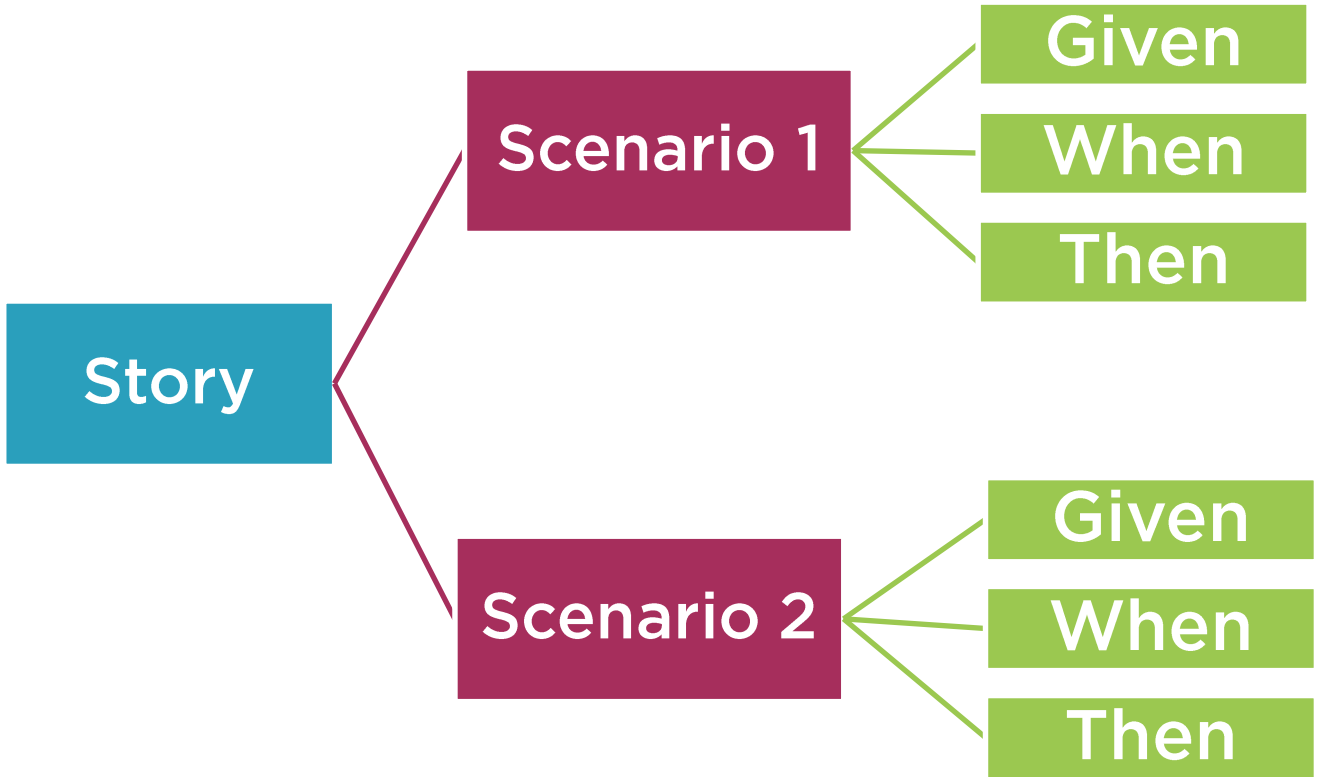
“I decided it must be possible to present TDD in a way that gets straight to the good stuff and avoids all the pitfalls.”

Dan North – introducing BDD

Given the initial state

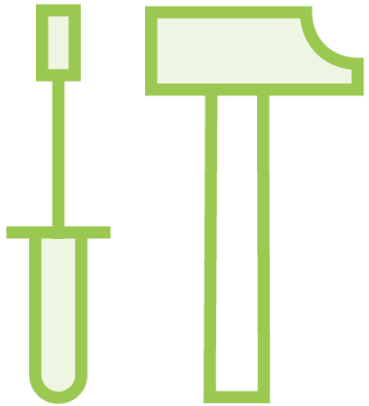
When event occurs

Then desired outcome



Behavior **D**riven **D**evelopment

Use tests to describe behavior of the system



Given

When

Then

Create executable specification

Clear confusion

```
SCENARIO("First roll is strike", “[Strike][Bowling]”) {
    GIVEN("Bowled strike on first turn") {
        Game game;
        game.Roll(10);

        WHEN("All rest rolls are gutter balls") {
            RollSeveral(game, 18, 0);
            THEN("Total score is 10") {
                REQUIRE(game.Score() == 10);
            }
        }

        WHEN("Next two rolls are not spare or strike") ...
```

Demo

BDD style tests using CATCH

Summary

- Unit tests vs. integration tests
- DRY vs. DAMP
- When to use test fixtures
- How to use sections
- Creating Data Driven Tests
- BDD with Catch2