Writing Tests with the Unity Test Framework

Harald Reingruber & Peter Kofler
@Harald3DCV @codecopkofler

Harald Reingruber

- Working in Visual Computing industry for 10 years
- Experienced the benefits of having tests
- Thinks bad tests are better than no tests
- Experiments with mob-programming

⇒ https://www.meetup.com/Mob-Programming-on-Open-Source-Software/

Peter Kofler

- Professional Software Developer for 20+ years
- I help development teams with Quality and Productivity
- “Developing Quality Software Developers”
- I run Coding Dojo and Coderetreats

⇒ https://www.softwerkskammer.org/groups/wien

Agenda

- Why Write Tests?
- Unity Test Framework
- Test “Situations” we faced
- How to get started
- QA

What is your experience?

What is a test?

I write tests for most of my business logic.

I am using TDD day to day.

I am testing also the UI of my applications.

I am writing automated tests for ALL THE THINGS ;-)

Why Write Tests?

Why write automated tests in general?

Tests are about getting feedback and managing risk.

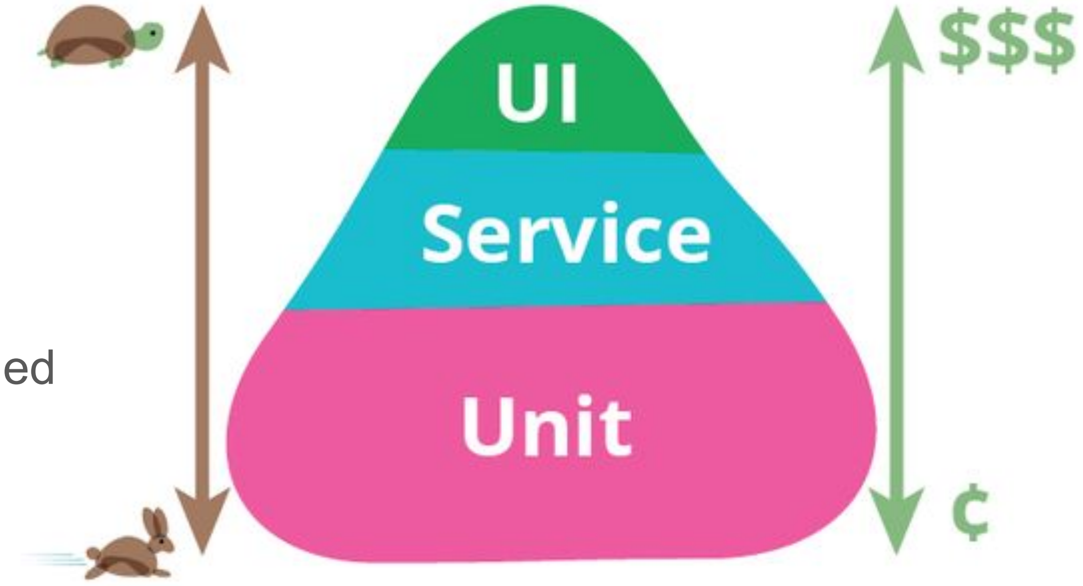
A big part is the (business) logic and is usually automatic testable.

Benefits of automated tests:

- Get feedback during development if your ideas work (TDD)
- Keep features working if code is changed (Regressions)
    - Allow refactoring
    - Return on Investment (ROI)
- Volume (thousands of cases) & scale testing to other platforms, devices, etc.
- Running tests 24/7 (“addicted to feedback”)

Unit- vs. Integrated Tests

- System tests are easier to write (and often the only ones possible) for user interfaces.
- UI tests are much slower, feedback takes longer, tests are run rarely.
- Integrated tests are not aligned with TDD process and have less benefits.



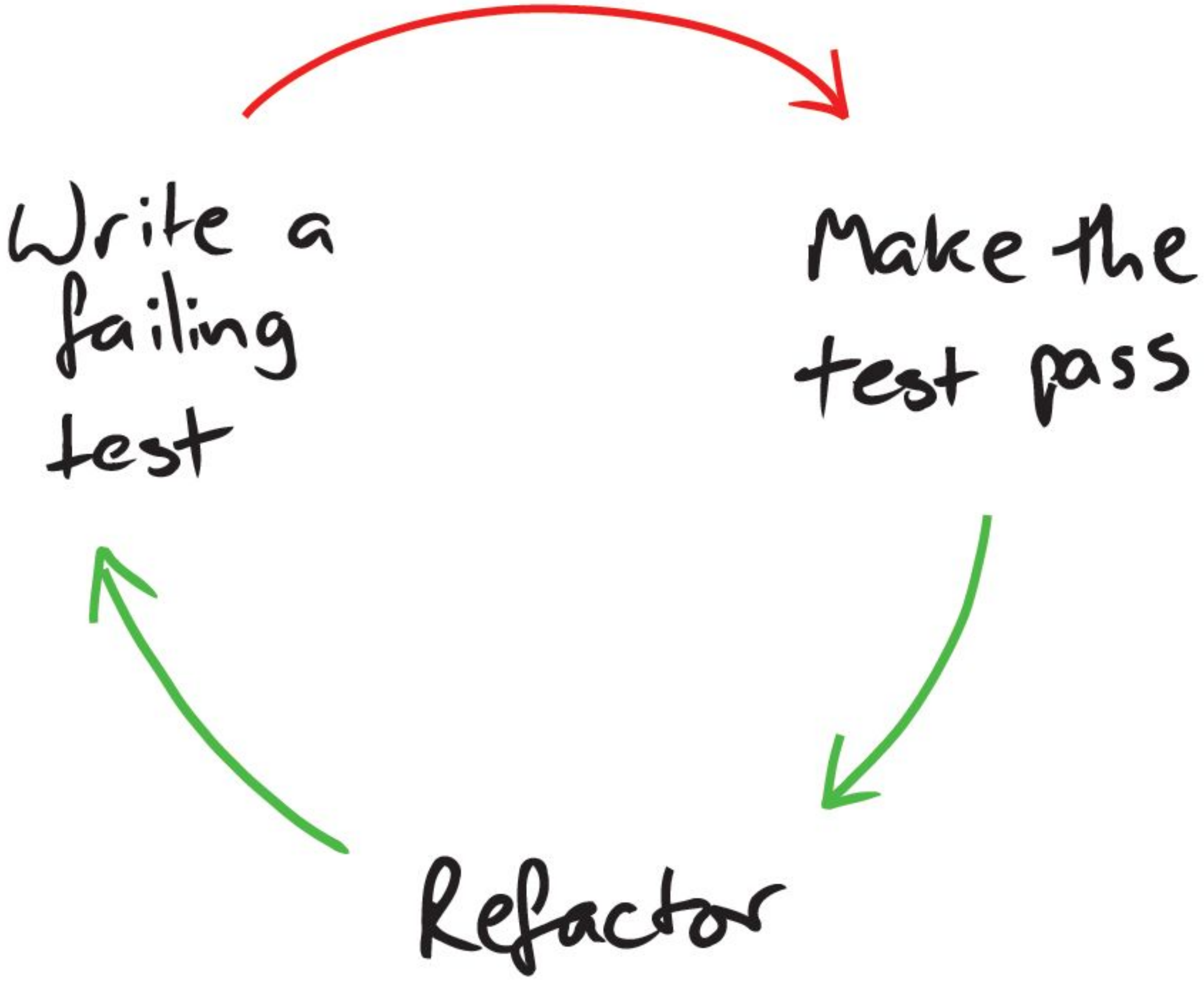
Why write the test first?

When using Test Driven Development you write your test
before you write the production code.

Pros of TDD:

- Faster feedback of your ideas
- We develop inherently testable code
- We design the production code interface from the viewpoint of calling code
  - encourages modular design with few dependencies

Cons: Hard to learn. Doesn’t work well in all cases.


Write a failing test
Make the test pass
Refactor

What about writing tests for user interfaces?

Graphics and user interface work is a lot of “fiddling” with pixels and coordinates. You must “massage” the code into place. It needs a very rapid cycle of edit-and-run. (Uncle Bob)

Most people agree that we are not writing automated tests for

- styles
- colours
- pixel perfect positions
- maybe hard coded text displayed

What about testing in the 3D world?

Is it even possible?

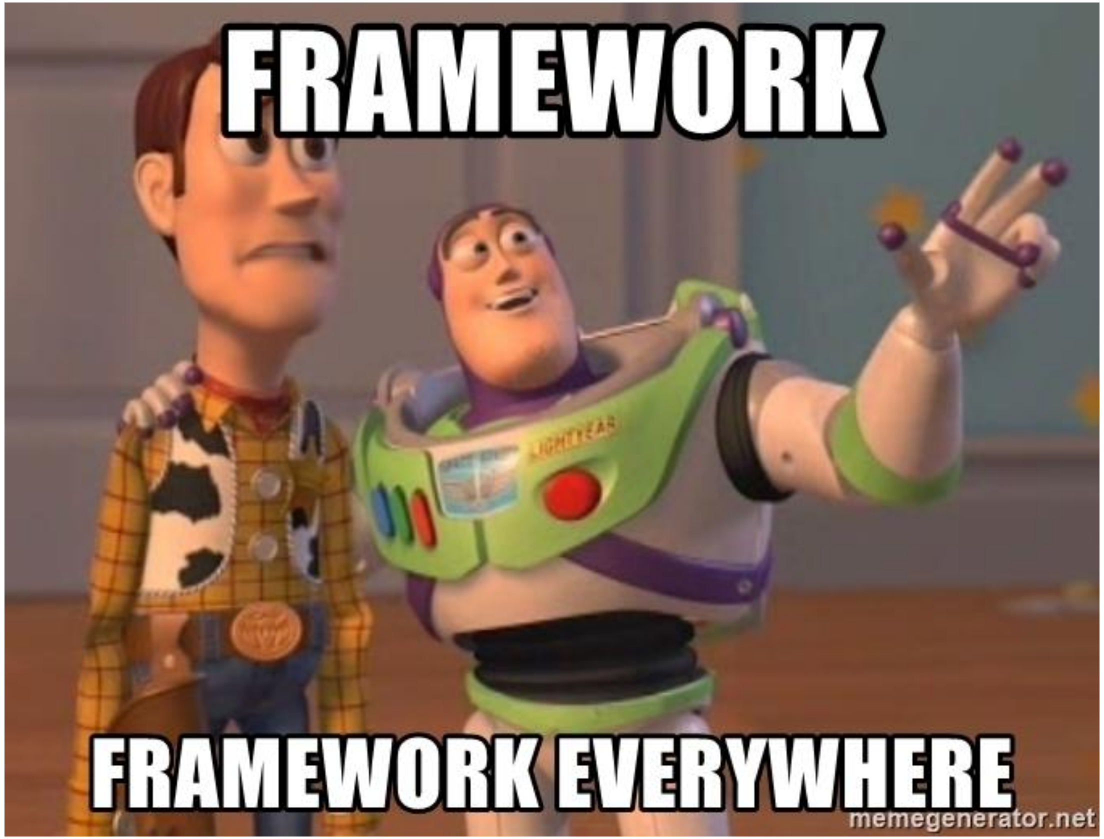
We would like to have the benefits of automated tests.

What about TDD?

We tried a sample project, approx. 30 hours pair programming.

This talk is about what we learned...

Btw: We don’t want to test the graphics engine or the GPU driver.



Unity Test Framework

Based on NUnit

Similar to other xUnit test frameworks.

https://docs.nunit.org

Based on NUnit

Basic test method

```
[Test]
public void HelloWorldTest()
{
    // Act
    var message = new Message();

    // Assert
    Assert.That(message.Text, Is.EqualTo("Hello World"));
}
```

Based on NUnit

Arrange - Act - Assert

```
[Test]
public void HelloWorldTest()
{
    // Arrange
    Environment.PrepareStuff();

    // Act
    var message = new Message();

    // Assert
    Assert.That(message.Text, Is.EqualTo("Hello World"));
}
```

Based on NUnit

Fixtures

```
[SetUp]
public void Init(){ /* ... */ }

[TearDown]
public void Cleanup(){ /* ... */ }

[Test]
public void HelloWorldTest()
{
    // Act
    var message = new Message();

    // Assert
    Assert.That(message.Text, Is.EqualTo("Hello World"));
}
```

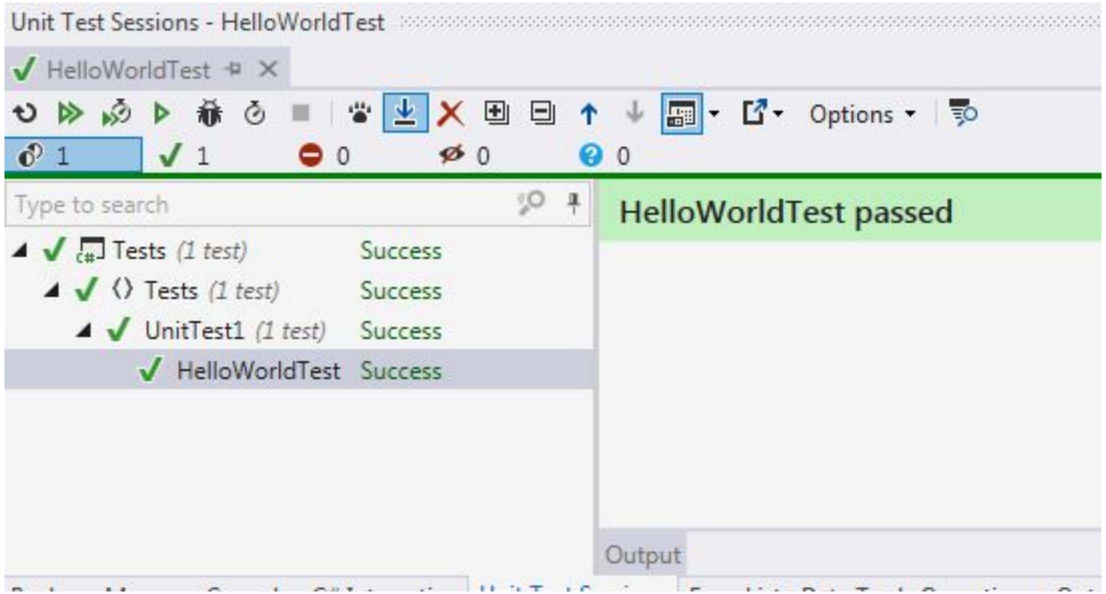
Based on NUnit

One time Fixtures

```
[OneTimeSetUp]
public void Init(){ /* ... */ }

[OneTimeTearDown]
public void Cleanup(){ /* ... */ }

[Test]
public void HelloWorldTest()
{
    // Act
    var message = new Message();

    // Assert
    Assert.That(message.Text, Is.EqualTo("Hello World"));
}
```

Based on NUnit

Test Runner



Unit Test Sessions - HelloWorldTest

HelloWorldTest

```
1    ✓ 1    0    0    0
```

Type to search

```
◢ ✓ Tests (1 test)          Success
  ◢ ✓ Tests (1 test)        Success
    ◢ ✓ UnitTest1 (1 test)  Success
        ✓ HelloWorldTest    Success
```

HelloWorldTest passed

Unity Test Framework

- Install via the Unity Package Manager
- Play-mode tests
  ⇒ load scenes and assert run-time behaviour
- Test builds
  ⇒ run tests on different platforms (mobile, consoles, WebAssembly)

https://docs.unity3d.com/Packages/com.unity.test-framework@1.1

Play-mode vs. Edit-mode Test

Play-mode Tests

- run **standalone** in a Player or inside the Editor

Edit-mode Tests

- only run in the Unity Editor
- and **have access to the Editor code** in addition to the game code
  ⇒ with Edit Mode tests it is possible to test any of your Editor extensions

Play-mode vs. Unit Test

Prefer unit tests, i.e. NUnit `Test` attribute.

Use unit tests instead of the `UnityTest` attribute, unless you need to **skip a frame** or **wait for a certain amount of time** (that is unless you need something of the Unity engine).

In Play Mode, the `UnityTest` attribute runs as a coroutine.

Test Scenarios
aka “Situations” we faced

Avoid (slow) Integration Tests

It’s tempting to often test with a Integration Test, but it comes at a cost.

Pattern:

If it’s unclear if a Unit Test is possible,
start with an Integration Test first
and try to refactor to a unit test later.

Scene Loading

Zenject (Dependency Injection framework) offers a `SceneTestFixture`

```
public class TestSceneStartup : SceneTestFixture
{
    [UnityTest]
    public IEnumerator TestSpaceFighter()
    {
        yield return LoadScene("SpaceFighter");

        // Wait a few seconds to ensure the scene starts correctly
        yield return new WaitForSeconds(2.0f);
    }
}
```

Query Object by ID

We used an `Identifier` script to query the GameObject from the test. This way, failing tests are avoided when the objects are renamed.

```
public static class Find
{
    public static GameObject SingleObjectById(string objectId)
    {
        var gameObjects = Object.FindObjectsOfType<Identifier>()
            .Where(identifier => identifier.id == objectId)
            .Select(identifier => identifier.gameObject)
            .ToList();
        Assert.That(gameObjects.Count, Is.EqualTo(1), $"Object {objectId} not found");
        return gameObjects[0];
    }
}
```

Linear search only works for scenes with not many objects.

Slow tests

- Make (real time) tests faster ⇒ tweak `Time.timeScale`

- Make your tests faster than real-time,
  but make sure to not execute “unnecessary” things first.

- Watch out for stability…
  Maybe have a nightly-build with real-time scale as well?

Waiting for something to be ready

- We created helper coroutines for better readability.

```
yield return Given.Scene("TestEnvironmentScene");
```

- **Recommendation:** Keep the actual test method clean and readable, future maintainers (or future you) will thank you 🙏

Input Testing

The (new) Input System allows to simulate user input.

```
[Test]
public void TestGamepad()
{
    var gamepad = InputSystem.AddDevice<Gamepad>();

    Press(gamepad.buttonSouth);
    Set(gamepad.leftStick, new Vector2(0.123f, 0.234f));
}
```

https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Testing.html

Comparing decimal numbers

allowed error

```
[Test]
public void VerifyThat_TwoFloatsAreEqual()
{
    var comparer = new FloatEqualityComparer(10e-6f);
    var actual = -0.00009f;
    var expected = 0.00009f;

    Assert.That(actual, Is.EqualTo(expected).Using(comparer));
}
```

Comparing Vectors

```
[Test]
public void VerifyThat_TwoVector3ObjectsAreEqual()
{
    // Custom error 10e-6f
    var actual = new Vector3(10e-8f, 10e-8f, 10e-8f);
    var expected = new Vector3(0f, 0f, 0f);
    var comparer = new Vector3EqualityComparer(10e-6f);

    Assert.That(actual, Is.EqualTo(expected).Using(comparer));
}
```

allowed error

Parameterized Test

Reduce duplication and increase test coverage.

```
[Test]
public void MyTest([Values(1, 2, 3)] int x, [Values("A", "B")] string s)
{
    /* ... */
}
```

⇩

```
MyTest(1, "A")
MyTest(1, "B")
MyTest(2, "A")
MyTest(2, "B")
MyTest(3, "A")
MyTest(3, "B")
```

Parameterized Test

Recommendation:
Parameters should only test one “dimension”.

Don’t merge different “requirements” because it’s easy to do. It will make analyzing a failing test more difficult.

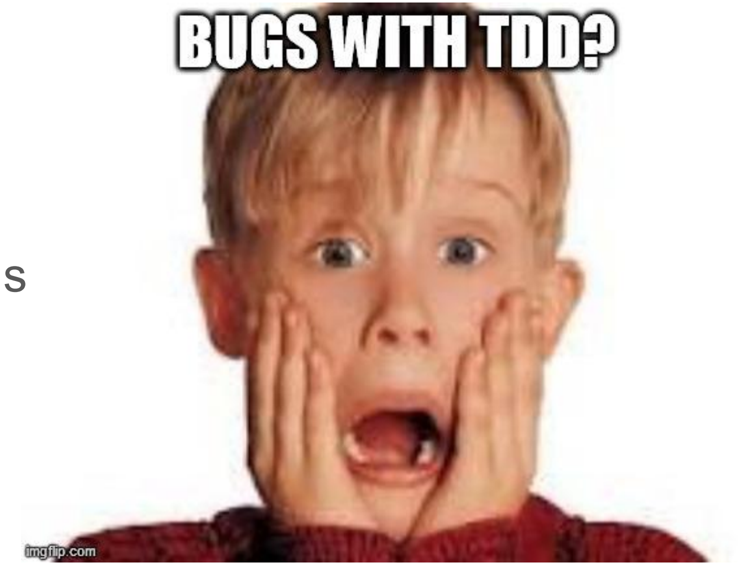
Balance test expressiveness vs code reuse.

Bugs we still saw

Remember the “fiddling” of colours and coordinates?

- We focused on the dynamic coordinates, states etc.
- and ignored the constant values in our tests
- The constant value was wrong ;-)
- E.g.
  - The object did move but had wrong height

Needed to run the application and see these bugs



How to get started

How to start automated testing in existing projects?

In the beginning much time is spent on

- Learning to write tests
- Arguing with team members
- Making the code testable
- Building test infrastructure

Expect it to be slow, painful, excruciating ;-)

Start slow, time-box a few hours a week. Do not expect to get anywhere soon.

How to start testing in your team?

Communicate the benefits of automated tests

- High complexity code needs tests
- High risk code needs tests
- Discuss trade-offs, avoid dogma

Educate yourself and your team on testing techniques

- Go to Coding Dojos and Coderetreats :)
- Try to find good and bad testing patterns, communicate them in the team
- Get help from outside, e.g. **Technical** Agile Coaching

Where to start testing in an existing code base?

Add tests when changing code

- Create a test for every bug.
- Focus on new developments and refactor and test code when it is changed
- Don’t test “production tested” code which doesn’t need to be changed.

Focus on areas and bring them under test

- Focus on high risk code to mitigate business risk.
- Focus on code which is often changed to avoid regressions.

Maybe start with code which is easy to test while you are learning.

Conclusion

Automated tests pay off in the future. They are a tool to manage risk.

Testing 3D components in Unity is possible.

Some of the 3D stuff was easy to test.

Not all things (on the UI) can be tested nor need to be tested.

Without experience it is hard to tell where tests are most beneficial.

You need to get started now!

Thank you!
Questions?

https://github.com/haraldreingruber/tron-tdd-3d-ui-kata

Feel free to contribute PRs ;-)

Writing Tests with the Unity Test Framework

Harald Reingruber & Peter Kofler
@Harald3DCV @codecopkofler

Image References

- Gorilla (CC) https://flickr.com/photos/cdevers/2855409766/
- Testing Pyramid https://martinfowler.com/bliki/TestPyramid.html
- TDD by Nathaniel Pryce, http://www.doc.ic.ac.uk/~np2/teaching/
- Framework https://memegenerator.net/instance/44668395/
- Scenario (CC) https://www.flickr.com/photos/21748859@N02/5729950342/
- Start (CC) https://www.flickr.com/photos/npobre/2601582256/
- Questions (CC) https://www.flickr.com/photos/oberazzi/318947873/