Raising the Value of Your Unit Tests

Richard Taylor

Director of Engineering

SentryOne

@rightincode – http://www.rightincode.com

Who is Richard Taylor?

SentryOne

Monitor • Diagnose • Optimize

Full Stack Software Engineer	Director of Engineering
Lives in Huntersville, NC	Huntersville, NC
Organizer of CLT Xamarin Developers	http://www.sentryone.com
Lead Organizer of Modern Devs CLT	@sentryone
http://www.rightincode.com	
@rightincode	

Microsoft® MVP Most Valuable Professional

Goals of this talk

- Define unit testing
- Explore styles of unit tests
- Identify weaknesses in unit tests
- How to increase the value of unit tests

What is Unit Testing?

- Unit testing is the process of independently testing the smallest testable part of your code (unit of work) for proper function
  - Class (its public interface), individual method
- The target of a unit test is the system under test (SUT)
- A unit test is code (code to test code)

What is Unit Testing? – cont.

- While unit testing can be done manually, it is mostly automated
  - Local build on your development machine
  - CI/CD pipeline
- Unit testing should be isolated
  - testing more complicated scenarios is the responsibility of integration testing
- Unit testing is not Test Driven Development (TDD) but a component of TDD

Why Write Unit Tests?

- To increase confidence that code changes do not break existing functionality
- To find errors early
- To provide some documentation
- To help facilitate better software design via refactoring

Styles of Unit Tests

- Output Verification
  - Provide SUT with known input(s) and test for expected output
  - Highest value
- State Verification
  - Provide SUT with known inputs(s) or use public interface and test the state (single or multiple data points) for expected values
  - High value

Styles of Unit Tests – cont.

- Collaboration Verification
  - Provide known input(s) or use public interface and test how it interacts with collaborators
  - Least value compared to output/state verification but offers some value
  - Typically brittle and difficult to maintain

What is a valuable test?

- Has a high chance of catching a regression bug
- Has a low chance of producing a false positive
- Provides fast feedback

What can cause unit test to lose value?

- Test names that do not effectively describe the test
- Complicated unit test code
- Testing more that a single unit of work
- Brittle to system under test (SUT) code changes
- Difficult to maintain
- Unreliable
- Slow

How to make your unit test more effective/valuable?

- Clear, simple, and readable
- High value
- Flexible

Effective Unit Tests – Clear, Simple, and Readable

- Make test names consistent and meaningful
  - Utilize a naming convention
    - i.e. three part naming convention
    - UnitOfWork_InitialCondition_ExpectedResult
      - Easy to scan/search
      - Groups together tests for the same unit of work
      - Provides some insight into the business rules

Effective Unit Tests – Clear, Simple, and Readable

- Test suite should be organized
  - DRY – Don't Repeat Yourself
  - DAMP – Descriptive and Meaningful Phrases
  - Follow a distinct pattern in your test
    - Setup/arrange
    - Action
    - Assert

Effective Unit Tests – Clear, Simple, and Readable

- Focus on high precision
  - Test one expectation per test
  - Multiple assertions on a object is okay but be careful
  - Test should point to a precise location of a problem

Effective Unit Tests – High Value

- Focus on testable code
  - Use dependency injection to provide dependencies
  - Avoid using "new"; it creates dependencies
  - Avoid global state
  - Be careful with static methods

Effective Unit Tests – High Value

- Focus on testable code
  - Use seams with legacy code
    - "New" the dependency but provide the ability to override it and use that ability to unit test
  - Favor composition over inheritance
    - Dependency Injection
  - Apply SOLID principles

Effective Unit Tests – High Value

- Test the code that has high risk
  - Complex workflows
  - Calculations
  - Minimize "What if" scenarios
- Cover all business rules
- Cover happy and non-happy paths
- Avoid testing things the compiler would catch (types, etc.) and text

Effective Unit Tests – Flexible

- Maximum of one mock per test
- Fewer than 10% of your of your test with mocks
- Do not test private methods
  - Should be covered by testing other public methods
- Test by scenarios rather than method

Resources

- Repo: (code)
  - https://github.com/rightincode/Xamarin-Forms-ToDo
    - Branch: Testability
- Unit Testing
  - https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters
  - Pluralsight
    - Building a Pragmatic Unit Test Suite
    - Writing Highly Maintainable Unit Tests
    - Advance Unit Testing
  - Working Effectively with Legacy Code - Michael Feathers
  - Beyond Legacy Code - David Bernstein
- Testing Frameworks (.NET)
  - Unit testing C# with MSTest and .NET Core
  - NUnit
  - XUnit