SOLIDIFY

Welcome to

DevSquare

- Why another meetup?
  - Lack of hands on focus
  - More code focus. Lately there has been a big focus on infrastructure.
  - Encourage discussion

- What is the purpose?
  - It is an area (“square”) where we can meet and have open discussions

SOLIDIFY

www.solidify.se

SOLIDIFY

Microsoft Partner
Microsoft
Gold Cloud Platform
Gold DevOps
Gold Application Development

Professional Scrum Trainer
Scrum.org

Microsoft Azure

LaunchDarkly

Octopus Deploy

MODERN Requirements

Visual Studio

GitHub

SCALED AGILE PARTNER BRONZE

Meetup

MVP
Microsoft® Most Valuable Professional

How to test your tests?

- Unit testing background
- Metrics for unit testing
- Mutation testing
- Demo, Stryker .NET

Introduction

- Unit testing is a concept that has existed for a while and has proven to provide the following advantages:
  - Better code structure
  - Easy to debug and recreate bugs
  - Serves as documentation
  - Reduces the number of errors in your application
  - Can also speed up the development process

- To achieve most of the benefits of unit tests, you need to ensure that you can trust your tests and that they are of high quality

How do you measure the quality of your test suite?

Unit test metrics

- Do you trust your test suite?
- If you are refactoring your code are you confident that you will not break something?

Why do you trust your tests?

- “We trust the test suite because we do TDD”
  - You are not allowed to write any production code unless it is to make a failing unit test pass.
  - You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
  - You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Why do you trust your tests?

- “We have high code coverage. All our code is tested!”

Code Coverage

- It is a metric on the amount of code that is being tested
- It is a metric that shows which code is not tested
- Unfortunately, this metric can easily be faked

How many tests are needed to test the following function?

```
private const decimal DiscountPercentage = 0.1m;

2 references | Mihai Borz, 23 hours ago | 1 author, 1 change
public decimal ApplyDiscount(decimal total)
{
    var totalWithDiscount = total;

    if (total >= 500)
    {
        totalWithDiscount = totalWithDiscount - (totalWithDiscount * DiscountPercentage);
    }

    return totalWithDiscount;
}
```

Code coverage

```
[Test]
0 references | Mihai Borz, 23 hours ago | 1 author, 1 change
public void When_the_amount_is_greater_than_500_Then_the_discount_is_applied()
{
    const decimal totalAmount = 600m;

    var sut = new Discount();

    var result = sut.ApplyDiscount(totalAmount);

    Assert.IsTrue(result < totalAmount);
}
```

```
    private const decimal DiscountPercentage = 0.1m;

2 references | 2/4 passing | Mihai Borz, 23 hours ago | 1 author, 1 change
    public decimal ApplyDiscount(decimal total)
    {
        var totalWithDiscount = total;

        if (total >= 500)
        {
            totalWithDiscount = totalWithDiscount - (totalWithDiscount * DiscountPercentage);
        }

        return totalWithDiscount;
    }
```

Unit tests, quality summary

- A different metric is needed
- The metric should show:
  - That we can trust our suite when refactoring
  - The quality of our tests
- Mutation testing to the rescue!

Intro to mutation testing

- Mutation testing is not a new concept
- Originally proposed by Richard Lipton in 1971
- First tool (PIMS) was built in 1980 by Timothy Budd for Fortran IV
- Mutation testing is an CPU intensive process and has not been very popular
- With current computing power, it's worth to take a look at again

Mutation testing hypotheses

- Competent programmer
  - Programmers tend to make simple mistakes.

- Coupling effect
  - Tests that detect simple, small errors, should be capable of detecting more complex ones, that derives from a combination of other errors.

Why bother with mutation testing?

- Gives a metric for the quality of your test suite
- You do not have to write them yourself
  - Automated and no human factor errors
  - Works methodically on the entire source code
- Identifies weak tests
- Identifies poorly tested pieces of code

How it works

- A tool will introduce changes to your application code
- A “mutant” is a change in code, mimicking programmer error
- One or several mutants are inserted at a time
- “Killing” a mutant means a test failed for that mutation
- Mutants are introduced by a “mutation operator”
- A mutation tool is a set of of mutation operators

Mutation score

- Mutation score 0..100
  - (Killed Mutants / Total number of Mutants) * 100
- It reflects how effective a test suite is at detecting logic changes
- Low mutation score => there can be many changes to you logic that will **not** be detected by your test suite.
- High mutation score => there can be many changes to you logic that will be detected by your test suite.

Common mutation operators

- Arithmetic operators `(+, -, *, /)`
- Equality operators `(<, >, <=, >=, ==, !=)`
- Logical operators `(|| => &&)`
- Boolean literals `(true => false, bool => !bool)`
- Assignment statements `(+=, -=, *=, /=, etc.)`
- String literals `(“foo” => “bar”)`
- Bitwise operators `(<<, >>, &, ^, ~)`
- ... and more, depending on the tool

Mutation score vs Code coverage

Does a high mutation score imply a high code coverage?

Mutation testing tools

- Muter (Swift)
- PiTest (Java)
- Stryker (.NET, JS, Scala)
- VisualMutator (.NET)
- NinjaTurtles (.NET)
- ... and many others

Stryker .NET

- Provided as a dotnet tool easily integrated into your CI/CD pipeline
- Works on a single test project
- Finds project reference to tested project
- Outputs several different report formats
- Straightforward to use, fairly configurable
- Drawbacks:
  - If your test project tests several projects, you need to specify one project a time.
  - Currently, no support for custom mutation operators.

Internals, Stryker .NET

- Each project file is parsed into a syntax tree object, using the Roslyn API.
- Stryker mutates the entire tree recursively, starting with the root node.
- Each mutation operator (“mutator”) is run on every node it applies to in the tree, possibly creating several mutations per node.
- The mutated syntax tree root is then compiled into an assembly, and the original tested assembly is overwritten.
- Each mutation is tested in order, where only tests covering the mutated line(s) are run by default.