Unit Testing Patterns for Concurrent Code

Dror Helper

Senior Microsoft Specialist Architect
AWS

We live in a concurrent world!

The free lunch is over!

- Multi-core CPUs are the new standard
- New(er) language constructs
- New(ish) languages

Meanwhile in the unit testing “world”

```
[Test]
public void AddTest()
{
    var cut = new Calculator();
    var result = cut.Add(2, 3);

    Assert.AreEqual(5, result);
}
```

The dark art of concurrent code

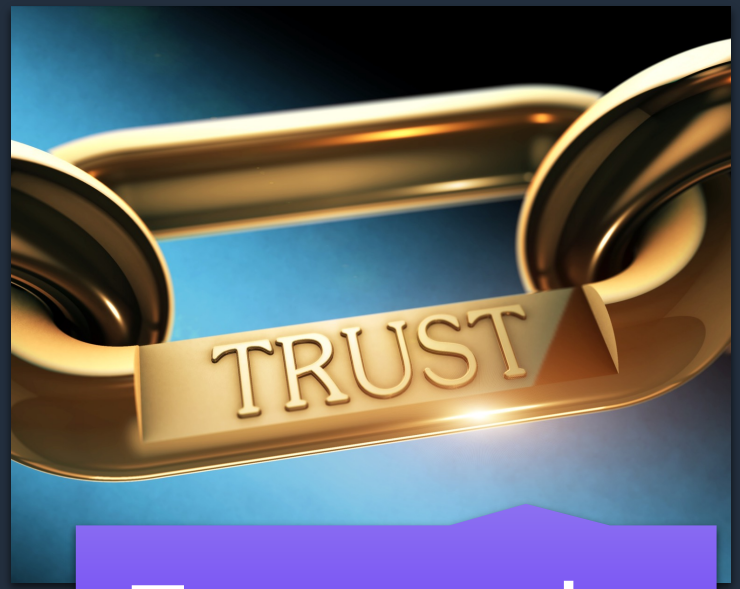
Several actions at the same time

Hard to follow code path

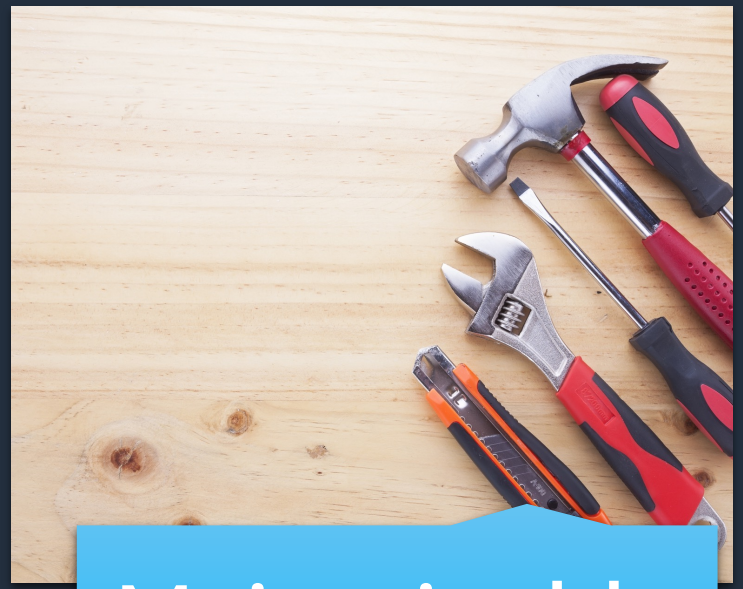
Nondeterministic execution

aws

A good unit test must be:



Trustworthy



Maintainable

Readable

Concurrency test smells

- Incosistant results
- Untraceable fail
- Long running tests
- Test freeze

How would we test this method

```
public void Start() {
    _worker = new Thread(() => {
        while (_isAlive) {
            Thread.Sleep(1000);
            var msg = _messageProvider.GetNextMessage();
            //Do stuff
            LastMessage = msg;
        }
    });
    _worker.Start();
```

Let's test – take #1

```
[TestMethod]
public void ArrivingMessagePublishedTest()
{
    var fakeMessageProvider = A.Fake<IMessageProvider>();
    A.CallTo(() => fakeMessageProvider.GetNextMessage()).Returns("Hello!");

    var server = new Server(fakeMessageProvider);
    server.Start();

    Thread.Sleep(2000);

    Assert.AreEqual("Hello!", server.LastMessage);
}
```

Test smell – using *Sleep* in tests

- Time based – fail/pass incosistently
- Test runs for too long
- Hard to investigate failured


"In concurrent programming
if something can happen,
then sooner or later it will,
probably at the most
inconvenient moment"
Paul Butcher
Seven concurrency models in seven weeks

Let's test – take #2

```
[TestMethod]
public async Task ArrivingMessagePublishedTest() {
    var fakeMessageProvider = A.Fake<IMessageProvider>();
    A.CallTo(() => fakeMessageProvider.GetNextMessage()).Returns("Hello!
");

    var server = new Server(fakeMessageProvider);
    server.Start();

    await Task.Delay(2000);

    Assert.AreEqual("Hello!", server.LastMessage);
}
```

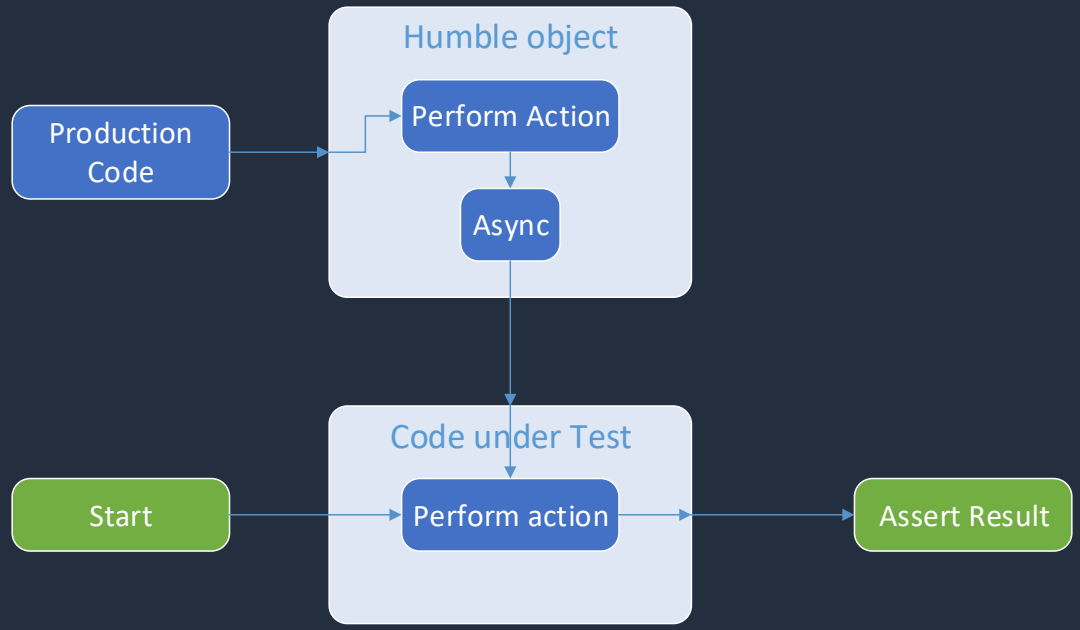
Solution:
avoid concurrent code !

Humble object pattern

Extract all the logic from the hard-to-test component

into a component that is testable via synchronous tests.

http://xunitpatterns.com/Humble%20Object.html



```
public void Start() {
    _worker = new Thread(() => {
        while (_isAlive) {
            Thread.Sleep(1000);

            var msg = _messageProvider.GetNextMessage();

            //Do stuff

            LastMessage = msg;
        }
    });

    _worker.Start();
}
```

```
public void Start() {
    _worker = new Thread(() => {
        while (_isAlive) {
            Thread.Sleep(1000);

            _messageHandler.HandleNextMessage();

        }
    });

    _worker.Start();
}
```

And finally – test!

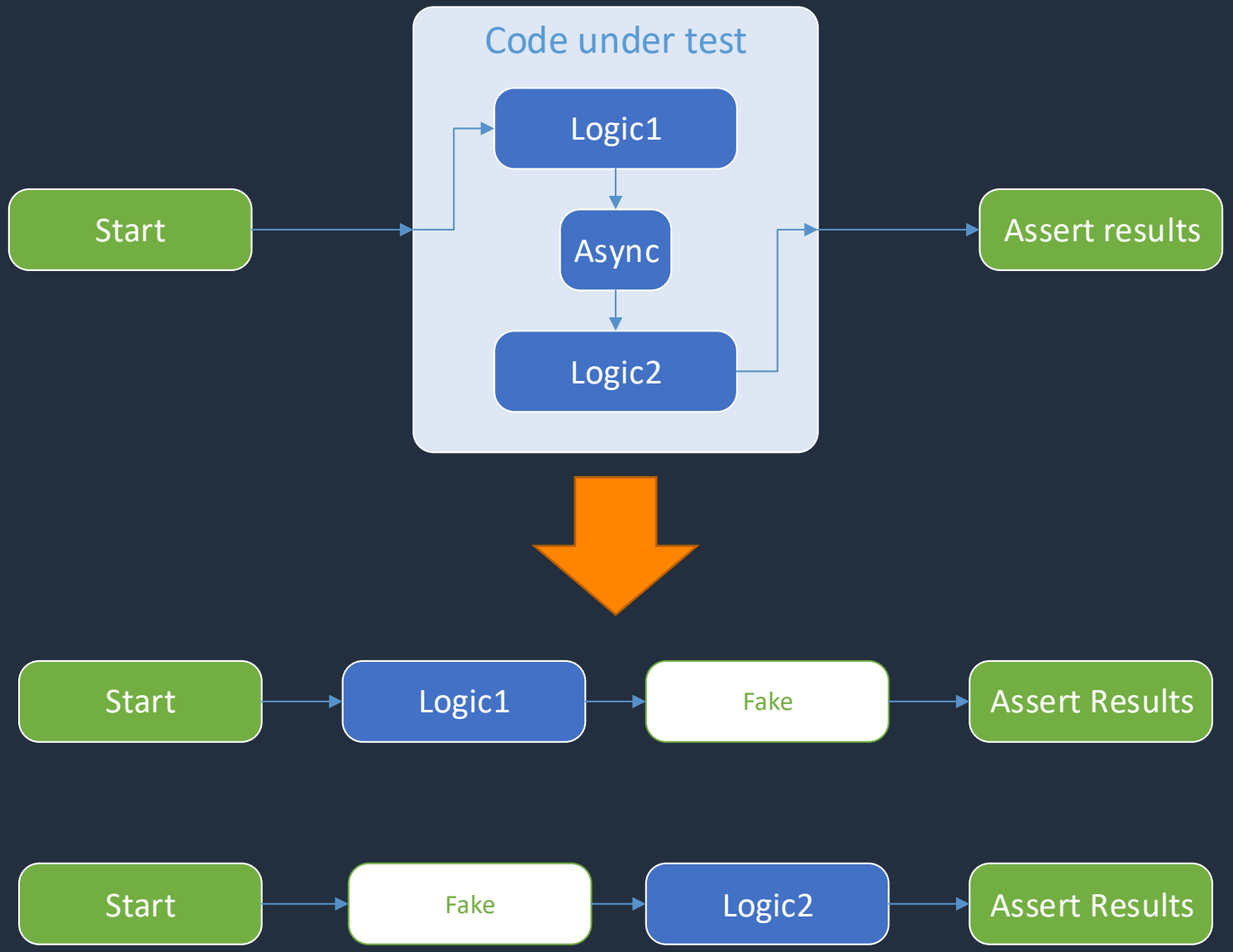
```
[Test]
public void ArrivingMessagePublishedTest()
{
    var fakeMessageProvider = A.Fake<IMessageProvider>();
    A.CallTo(() => fakeMessageProvider.GetNextMessage()).Returns("Hello!");

    var messageHandler = new MessageHandler(fakeMessageProvider);
    messageHandler.HandleNextMessage();

    Assert.AreEqual("Hello!", messageHandler.LastMessage);
}
```

Concurrency as part of the program flow

```
public class MessageManager {
  private IMesseageQueue _messeageQueue;

  public void CreateMessage(string msg)
  {
     // Here Be Code!


     _messeageQueue.Enqueue(message);
  }
}
```

```
public class MessageClient {
  private IMesseageQueue _messeageQueue;

  public string LastMessage { get; set; }

  private void OnMsg(object o, EventArgs e) {
     // Here Be Code!


     LastMessage = e.Message;
  }
}
```

Test before – test after



Testing flow – part 1

```
[Test]
public void AddNewMessageProcessedMessageInQueue() {
    var messeageQueue = new AsyncMesseageQueue();

    var manager = new MessageManager(messeageQueue);

    manager.CreateNewMessage("a new message");

   Assert.AreEqual(1, messeageQueue.Count);
}
```

Testing flow – part 2

```
[Test]
public void QueueRaisedNewMessageEventClientProcessEvent() {
    var messeageQueue = new AsyncMesseageQueue();

    var client = new MessageClient(messeageQueue);

    client.OnMessage(null, new MessageEventArgs("A new message"));

    Assert.AreEqual("A new message", client.LastMessage);
}
```

Avoid concurrency patterns

- ✓ The best possible solution
- ✓ No concurrency == no problems

- ❖ Do not test some of the code
- ❖ Not applicable in every scenario

How can we test this class?

```
public class ClassWithTimer {
    private Timer _timer;

    public ClassWithTimer(Timer timer) {
        _timer = timer;
        _timer.Elapsed += OnTimerElapsed;
        _timer.Start();
    }
    private void OnTimerElapsed(object sender, ElapsedEventArgs e) {
        SomethingImportantHappened = true;
    }
    public bool SomethingImportantHappened { get; private set; }
}
```

aws

This is NOT a good idea

```
[Test]
public void ThisIsABadTest() {
    var timer = new Timer(1);

    var cut = new ClassWithTimer(timer);

    Thread.Sleep(100);

    Assert.IsTrue(cut.SomethingImportantHappened);
}
```

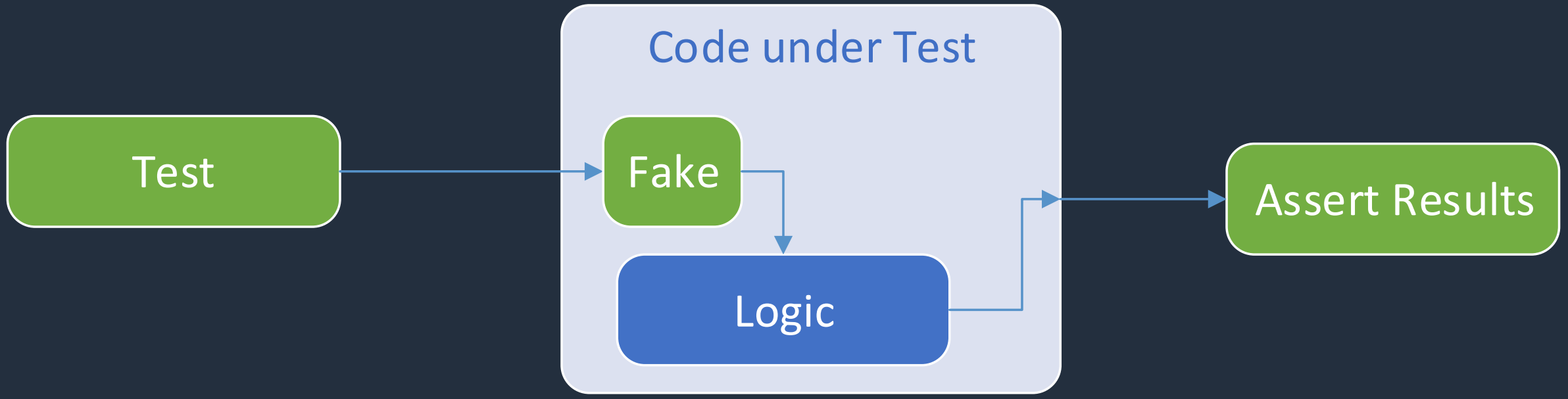
Set timeout/interval to 1

Also seen with a very small number

Need to wait for next tick/timeout

- Time based == fragile/incosistant
- Hard to investigate failures
- Usually comes with Thread.Sleep

Fake & Sync



Using Typemock Isolator to fake a timer

```
[Test, Isolated]
public void ThisIsAGoodTest() {
    var fakeTimer = Isolate.Fake.Instance<Timer>();

    var cut = new ClassWithTimer(fakeTimer);

    var fakeEventArgs = Isolate.Fake.Instance<ElapsedEventArgs>();
    Isolate.Invoke.Event(
        () => fakeTimer.Elapsed += null, this, fakeEventArgs);


    Assert.IsTrue(cut.SomethingImportantHappened);
}
```

Not every .NET class can be faked...

- Mocking tool limitation (example: inheritance based)
- Programming language attributes
- Special cases (example: MSCorlib)

Solution – wrap the unfakeable

Problem – requires code changes

Step 1 – create an interface

```
public interface ITimer
{
    event EventHandler<EventArgs> Elapsed;
    void Start();
    void Stop();
}
```

Step 2 – create a wrapper class

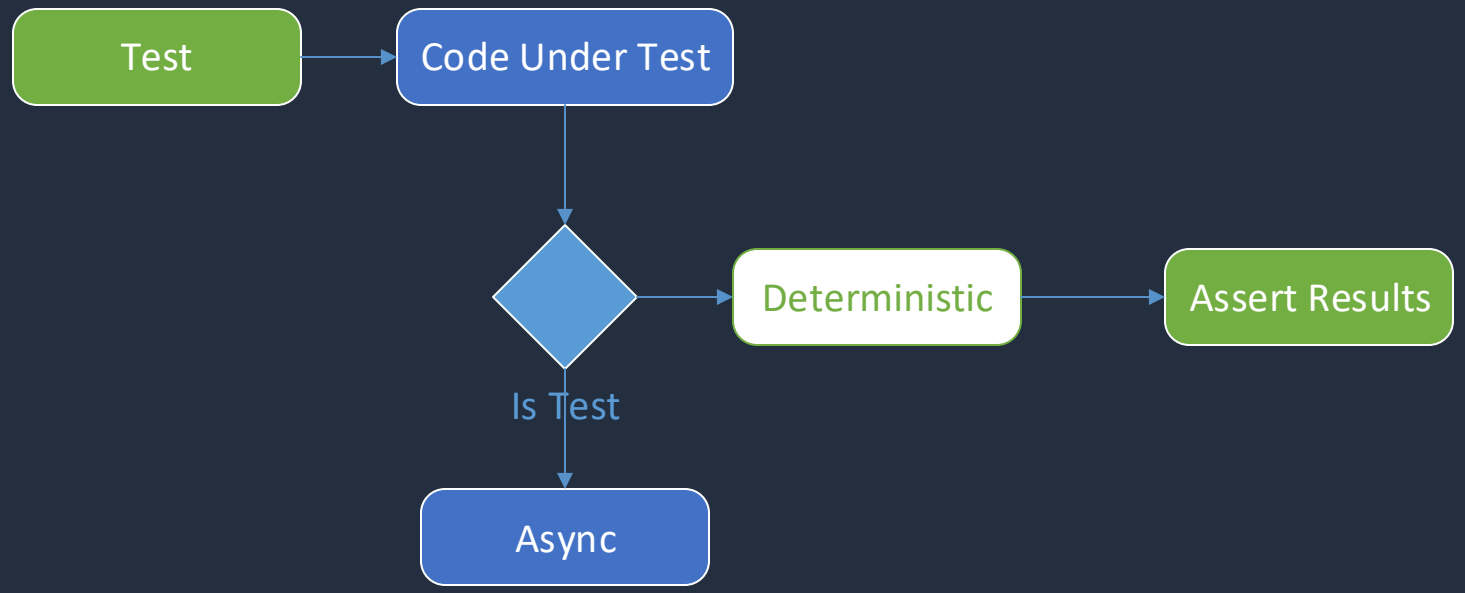
```
internal class MyTimer : ITimer
{
    private readonly Timer _timer;

    public event EventHandler<EventArgs> Elapsed;
    public MyTimer(double interval)
    {
        _timer = new Timer(interval);
        _timer.Elapsed += OnTimerElapsed;
    }
    ...
}
```

Testing with *ITimer*

```
[Test]
public void ThisIsAGoodTestWithFakeItEasy()
{
    var fakeTimer = A.Fake<ITimer>();

    var cut = new ClassWithMyTimer(fakeTimer);

    fakeTimer.Elapsed += Raise.With(EventArgs.Empty);

    Assert.IsTrue(cut.SomethingImportantHappened);
}
```

“How can we test that an asynchronous operation never happened ”

Run in sync



Another day – another class to test

```
public void Start() {
    _cancellationTokenSource = new CancellationTokenSource();
    Task.Run(() => {
    var message = _messageBus.GetNextMessage();

    if(message == null)
        return;

    // Do work

        if (OnNewMessage != null) {
            OnNewMessage(this, EventArgs.Empty);
        }

    }, _cancellationTokenSource.Token);
```

Running code in *test mode*

- ✓ Dependency injection
- ✓ Preprocessor directives
- ✓ Pass delegate
- ✓ Other

Run in single thread patterns

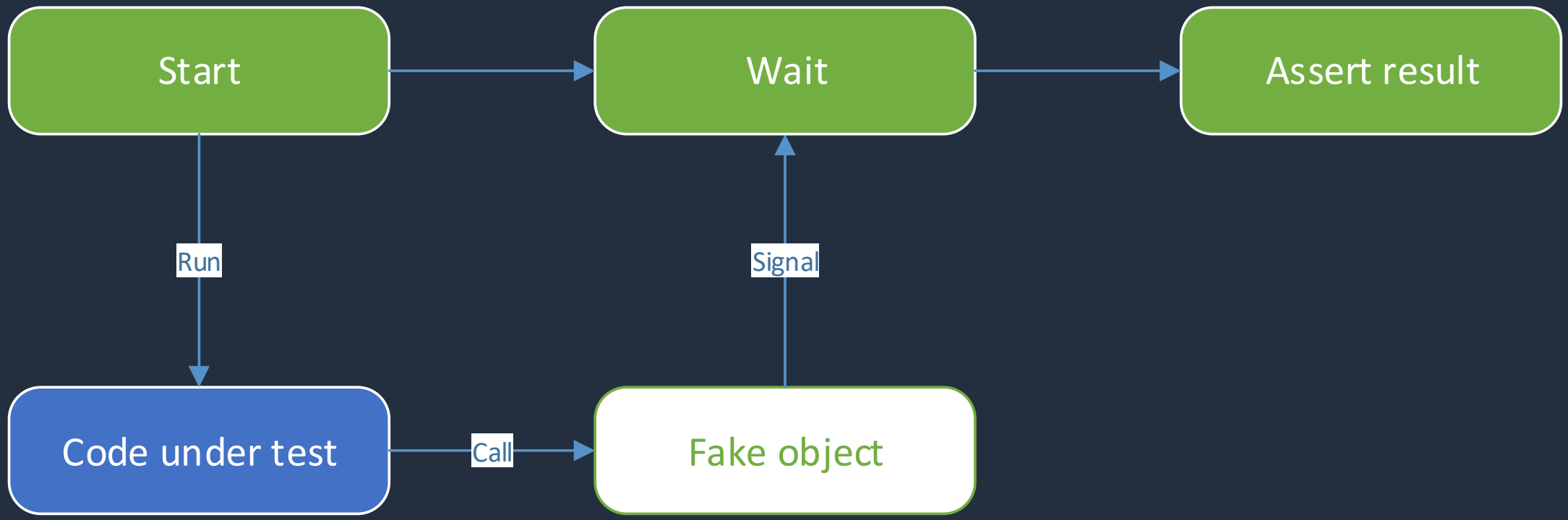
Fake & Sync

Async code → sync test

Synchronized run patterns

When you have to run a concurrent test in a predictable way

The Signalled pattern



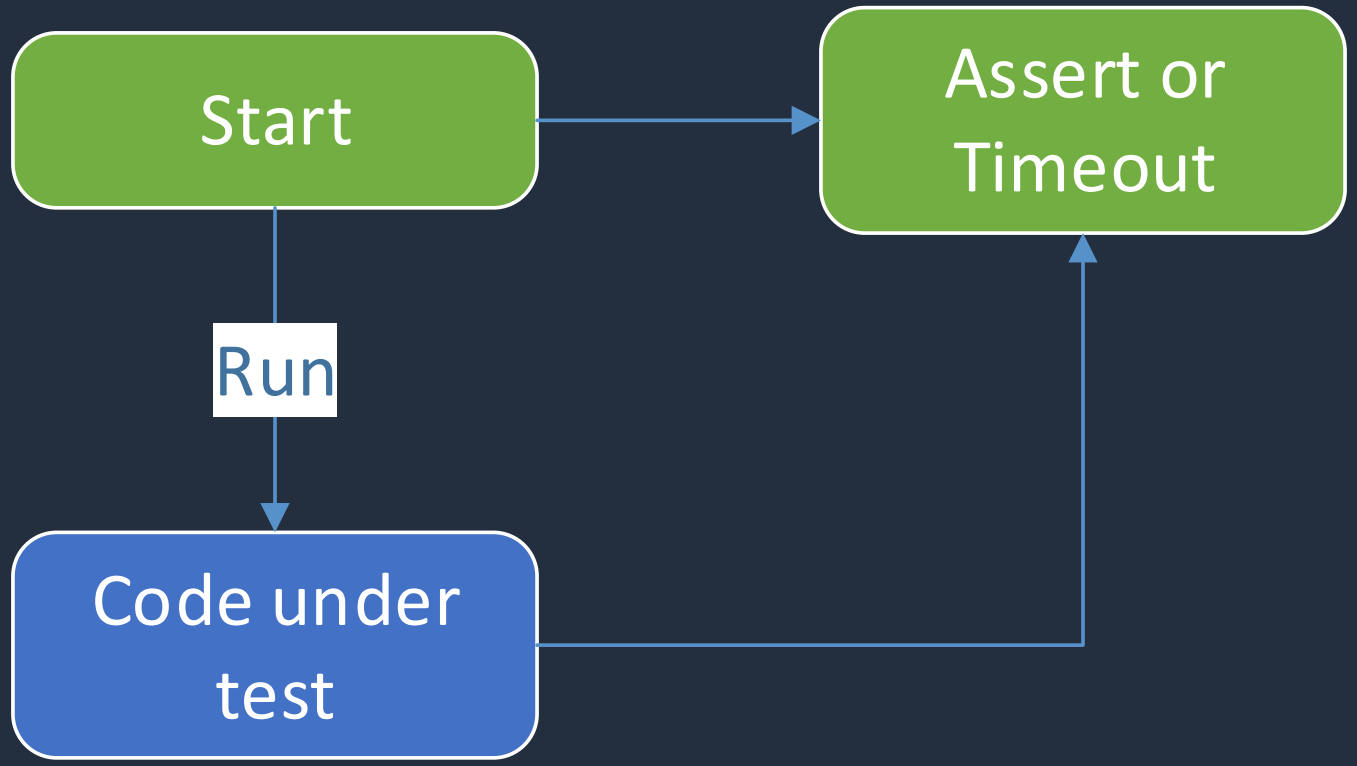
Using the Signalled pattern

```
public void DiffcultCalcAsync(int a, int b)
{
    Task.Run(() =>
    {
        Result = a + b;

        _otherClass.DoSomething(Result);
    });
}
```

Using the Signalled pattern – cont.

```
[Test]
public void TestUsingSignal() {
    var waitHandle = new ManualResetEventSlim(false);

    var fakeOtherClass = A.Fake<IOtherClass>();
    A.CallTo(() => fakeOtherClass.DoSomething(A<int>._)).Invokes(waitHandle.Set);


    var cut = new ClassWithAsyncOperation(fakeOtherClass);

    cut.DiffcultCalcAsync(2, 3);

    var wasCalled = waitHandle.Wait(10000);

    Assert.IsTrue(wasCalled, "OtherClass.DoSomething was never called");
    Assert.AreEqual(5, cut.Result);
}
```

Busy Assert



Synchronized test patterns

- Harder to investigate failures
- Cannot Test that a call was not made

- Test runs for a long time – if fails
- Use if other paterns are not applicable

Concurrent unit testing patterns

Avoid Concurrency	• Humble object • Test before – test after
Run in single thread	• Fake & Sync • Async in production - sync in test
Synchronize test	• The Signaled pattern • Busy assertion

Thank you!

Dror Helper
dhelper@amazon.com
@dhelper