Dirty Tests

Practise improving brittle, complicated, incomprehensible automated tests

The case of the Time Dependent Test

The case of the Obscure Test

Background

- Transport Information System
- Assign Carrier to Transfer Shipment of Goods
- Tracks Shipment, Owner and Carrier
- Carriers change over time
- Test is about assigning (new) carrier

Some implementation is missing.

The case of the Argument Capturing Test

Background

Part of a system that generates invoices with discounts.
There is some argument capturing going on here.

Have a look at the tests and code

- What do you think about the tests?
- What makes the tests dirty?
- Pick one of the two exercises

Useful information

- Code in *~/dirtytests_cs/*
- Solution in *dirtytests_cs.sln*
- Each exercise has a README.md file in the Tests project
- Keep track of your steps and decisions (small commits!) - the path you follow is as interesting as the end result!

Refactor the tests to make them clean

- Make them express intent
- Think controlled development
- In Baby Steps

deliberate practice

It is not forbidden to change production code with changing the tests

Mid session interval

How are we doing?

Hints - Obscure Tests

- Fix low hanging fruit first: renames
- Split test class, to clean up setup code
- Duplicate (pieces of) setup code per test

Separate construction & wiring (object-under-test + structural collaborators)

```
[SetUp]
public void SetUp() {
  notificationPublisher = Substitute.For<NotificationPublisher> ();
  processRepository = Substitute.For<ProcessRepository> ();
  ...
  carrierProcessor =  new CarrierProcessor();
  carrierProcessor.notificationPublisher = notificationPublisher;
  carrierProcessor.processRepository = processRepository;
  ...
```

from state and behavior of collaborators

```
  processRepostitory.findBy....("blah").Returns(AProcessThatContains
}
```

Create builder like methods for object setup. So:

```
[SetUp]
public void initMocks(..., bool stateChangeAllowed) {
  assignmentCarrierTask = Substitute.For<AssignmentCarrierTask>();
  assignmentCarrierTask.isStateChangeAllowed(Arg.Any<AssignmentTaskS

  process = Substitutestitute.For<Process>();
  process.getTask<AssignmentTaskDefinition,AssignmentCarrierTaskignm
}
```

becomes

```
[SetUp]
public void initMocks(..., bool stateChangeAllowed) {
  assignmentCarrierTask = AnAssignmentCarrierTaskWithStateChangAllow
  var processRepository = AProcessRepositoryWith(AProcessWith(assign
}
```

More hints

- For each mock: is it a mock or a stub?
- Introduce builders to construct objects
- Replace 'unnecessary mocks' (objects you can just instantiate)
- Build your own matchers / NUnit constraints
- Extract objects to support your tests (these might even turn out to be missing concepts from the domain)

Hints - Argument Capturers

- Fix low hanging fruit first:
  - renames
  - common setup code
  - (extract constants) - discuss
- Split test class, to clean up setup code
- Duplicate (pieces of) setup code per test
- Apply wishful thinking to tests

Solution 1 - Argument Capturers

Split responsabilities

- Inserting a non-empty InvoiceCreatedEvent
- Building an InvoiceCreatedEvent
  - With or without discount
  - Several other fields

Solution 1 - Argument Capturers

Wishful thinking

```
[Test]
public void ExecuteTriggersInsertWithCorrectEvent()
{
    InvoiceCreatedEvent invoiceCreatedEvent =
                    _invoiceService.CreateEvent(_recipient,_invoiceW
    _invoiceService.Execute(_recipient, _invoiceWithoutDiscount);

    _invoiceDao.Received().Insert(Arg.Is(invoiceCreatedEvent));
}
```

Solution 1 - Argument Capturers

Baby steps

One test only looks at one concern - testExecuteNoEvent. Do that first.

```
public void testExecuteNoEvent() {
    _invoiceService.Execute(_recipient, new List<InvoiceLine>());
    _invoiceDao.Received(0).Insert(Arg.Any<InvoiceEvent>());
}
```

Solution 1 - Argument Capturers

Extracted setup

```
// connascence of value + duplicated in multiple tests
private readonly List<InvoiceLine> _invoiceWithoutDiscount = new []
    {
        new InvoiceLine("consulting", 15000.0), new InvoiceLine("tra
    }.ToList();

private InvoiceDao _invoiceDao; // duplicated
private InvoiceService _invoiceService; // duplicated
private string _recipient = "recipient"; // connascence of value

[SetUp]
public void Before()
{
    _invoiceDao = Substitute.For<InvoiceDao>(); // duplicated
    _invoiceService = new MyInvoiceService(); // duplicated
    _invoiceService.SetInvoiceDao(_invoiceDao); // duplicated
```

Solution 1 - Argument Capturers

Extract method

Refactor towards *subclass for test*

```
public void Execute(String recipient, List<InvoiceLine> invoiceLines
{
    if (invoiceLines.Count != 0)
    {
        var newEvent = CreateEvent(recipient, invoiceLines);
        invoiceDao.Insert(newEvent);
    }
    else
    {
        log.Info("No invoice lines");
    }
}
```

One level of abstraction per method.

Responsabilities of creation and insertion better separated.

Solution 1 - Argument Capturers

Subclass for test

Remember value set in test, so production can use the same, and we can check it with `Arg.Is(value)`.

```
class MyInvoiceService : InvoiceService
{
    private InvoiceCreatedEvent memoized;

    public override InvoiceCreatedEvent CreateEvent(string recipient
    {
        return memoized ?? (memoized = base.CreateEvent(recipient, i
    }
}
```

Lazy initialization, we have connascence of identity in tests, but allows us to postpone implementing equality operator.

Solution 1 - Argument Capturers

Now refactor test for the other responsability

```
[Test]
public void GrandTotalIsSumOfInvoiceAmountsWhenSubTotalBelowDiscount
    InvoiceCreatedEvent e = _invoiceService.CreateEvent(_recipient,

    // Still too many asserts
    Assert.NotNull(e.Id); // Can be made more specific
    Assert.NotNull(e.CreatedAt); // Another extraction is waiting fo
    Assert.That(e.AmountDue, Is.EqualTo(20000.0));
    Assert.That(e.Services.Count, Is.EqualTo(2));
    Assert.That(e.Amounts.Count, Is.EqualTo(2));
}
```

Solution 1 - Argument Capturers

Conclusion

- Mock with behaviour gone
- Two responsabilities separated
- Tiny change to implementation
- A few lines of support code in test
- Still more work to be done, but current refactorings make it visible

Concluding remarks

Enjoy, but use mocks responsibly

mocks where invented to for test driving responsibilities

- Keep interactions simple
- Think "tell don't ask"
- Do not use mocks when inappropriate

Example inappropriate use

Not this:

```
@Test
public void changingTransportOrganisationMovesTheItems() {
  transportOrganisation = mock(TransportOrganisation.class);
  when(transportOrganisation.getOrganisationReferenceNumber())
      .thenReturn(transportOrgId);
  when(transportOrganisation.getOrganisationType())
      .thenReturn(OrganisationType.CARRIER);
}
```

Using real object with builders

```
@Test
public void changingTransportOrganisationMovesTheItems() {
  transportOrganisation = build(aTransportOrganisation()
          .withReferenceNumber(transportOrgId)
          .withType(OrganisationType.CARRIER));
}
```

```
public class TransportOrganisationBuilder implements Builder<Transpo
  private TransportOrganisation build = new TransportOrganisation();
  public TransportOrganisationBuilder withReferenceNumber(String tra
    build.setOrganisationReferenceNumber(transportOrgId);
    return this;
  }
  public TransportOrganisationBuilder withType(OrganisationType orga
    build.setOrganisationType(organisationType);
    return this;
  }
  @Override
  public TransportOrganisation build() { return build; }
}
```

And use helpers for builders and examples

```
public class DomainBuilders {
    public static TransportOrganisationBuilder aTransportOrganisatio
      return new TransportOrganisationBuilder();
    }
    //....
}
```

```
public class DomainExamples {
    public static TransportOrganisationBuilder aValidTransportOrgani
      return DomainBuilders.aTransportOrganisation()
              .withReferenceNumber("ORN123123")
              .withType(OrganisationType.CARRIER);
    }
    // ...
}
```

Test case class follows common setup

Not this:

```
public class CarrierTest {
  @Before
  public void setUp() {
    carrier = new Carrier();
  }
  public void getsAssignedWhenNominatedAndNominationIsConfirmed() {
    carrier.setState(CarrierState.NOMINATED);
    // ...
  }
  public void getsDeclinedWhenNominatedAndNominationIsRejected() {
    carrier.setState(CarrierState.NOMINATED);
    // ...
  }
  public void getsNominatedForATransportWhenRequested() {
    carrier.setState(CarrierState.ASSIGNED);
    // ...
```

Separated by setup

```
public class NominatedCarrierTest {
  @Before
  public void setUp() {
    carrier = new Carrier();
    carrier.setState(CarrierState.NOMINATED);
  }
  public void getsAssignedWhenNominationIsConfirmed() {}
  public void getsDeclinedWhenNominationIsRejected() {}
}
```

```
public class AssignedCarrierTest {
  @Before
  public void setUp() {
    carrier = new Carrier();
    carrier.setState(CarrierState.ASSIGNED);
  }
  public void getsNominatedForATransportWhenRequested() {}
  public void getsDeclinedFromATransportWhenFinedThreeTimes() {}
}
```

Only one (conceptual) assert per test

Not this:

```
@Test
public void deliversDrinkAndReturnsChange() {
  machine.configure(Choice.sprite, Can.sprite, 1);
  machine.insertCredits(2);
  assertEquals(Can.sprite, machine.deliver(Choice.sprite));
  assertEquals(2, machine.get_change());
  assertEquals(0, machine.get_change());
}
```

But this:

```
@Test
public void deliversDrink() {
  machine.configure(Choice.sprite, Can.sprite, 1);
  machine.insertCredits(2);
  assertEquals(Can.sprite, machine.deliver(Choice.sprite));
}
@Test
public void returnsChangeWhenPaidTooMuch() {
  machine.configure(Choice.sprite, Can.sprite, 1);
  machine.insertCredits(2);
  machine.deliver(Choice.sprite);
  assertEquals(2, machine.get_change());
}
@Test
public void returnsNoChangeWhenOutOfCredits() { /* ... */ }
```

Tests in test case should tell a story

Name of the test case class and methods should tell a story

```
public class NominatedCarrierTest {
  public void getsAssignedWhenNominationIsConfirmed() {}
  public void getsDeclinedWhenNominationIsRejected() {}
}
```

```
public class AssignedCarrierTest {
  public void getsNominatedForATransportWhenRequested() {}
  public void getsDeclinedFromATransportWhenFinedThreeTimes() {}
}
```

In rspec:

```
describe(Carrier) do
  context("when nominated") do
    it "gets assigned when nomination is confirmed" {}
    it "gets declined when nomination is rejected" {}
  end
  context("when assigned") do
    it "gets nominated for a transport when requested" {}
    it "gets declined from a transport when fined three times" {}
  end
end
```

Test name expresses action and expectation

```
public class NominatedCarrierTest {
  public void getsAssignedWhenNominationIsConfirmed() {}
  public void getsDeclinedWhenNominationIsRejected() {}
}
```

- NominatedCarrier
  - getsAssignedWhenNominationIsConfirmed
  - getsDeclinedWhenNominationIsRejected

Given when then

```
public class VendingMachineTest {
  @Test
  public void deliversCanOfChoice() {
    Bin bin = new Bin();                                    // given
    VendingMachine machine = build(aVendingMachine          // ...
        .deliveringInto(bin)                                // ...
        .withChoice(Choice.Cola).delivering(Can.Coke));     // ...
    machine.deliver(Choice.Cola);                           // when
    assertThat(bin.contents(), is(asList(Can.Coke)));       // then
  }
}
```

Repeat actions, not assertions

Not this:

```
@Before
public void setUp() {
  machine.configure(Choice.sprite, Can.sprite, 1);
}
@Test
public void deliversDrinkWhenPaidFor() {
  machine.insertCredits(1);
  assertEquals(Can.sprite, machine.deliver(Choice.sprite));
}
@Test
public void returnsChange() {
  machine.insertCredits(2);
  assertEquals(Can.sprite, machine.deliver(Choice.sprite));
  assertEquals(2, machine.get_change());
  assertEquals(0, machine.get_change());
}
```

But this:

```
@Before
public void setUp() {
  machine.configure(Choice.sprite, Can.sprite, 1);
}
@Test
public void deliversDrinkWhenPaidFor() {
  machine.insertCredits(1);
  assertEquals(Can.sprite, machine.deliver(Choice.sprite));
}
@Test
public void returnsChange() {
  machine.insertCredits(2);
  machine.deliver(Choice.sprite);
  assertEquals(2, machine.get_change());
}
```

Prevention

- Write more tests
  - -> more good examples
  - -> tend to have smaller scope
  - -> tend to be simpler
  - -> tend to be easier to read
- Write them first
- Write tests more often (make it a habit)