Growing Up: Better Unit Test Design and Implementation

Our sample domain: hairdresser schedule

EDUMENT

Hannah the Hairdresser helps people to have awesome hair, working at it between 4 and 6 days of the week depending on the season.

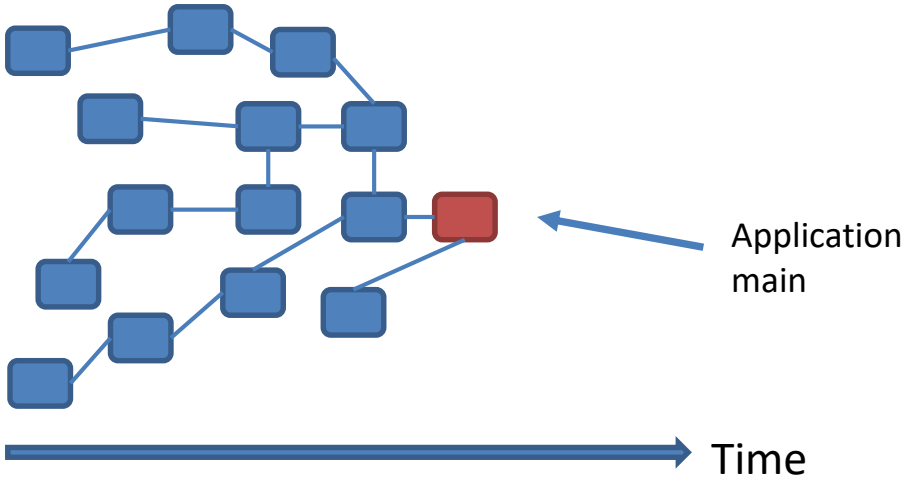
She has asked us to develop a scheduling tool which is supposed to make bookings for different time slots. Hannah wants to ensure she doesn't get double-booked, and also to ensure she doesn't schedule appointments on days when she is not planning to work.

We will **implement** the core business logic for this domain using TDD, and **learn more** about effective testing along the way!

Sketching in test code

When developing a new class, we may traditionally start by creating the class, stubbing in some methods, implementing them, and then trying to use the class in our system.

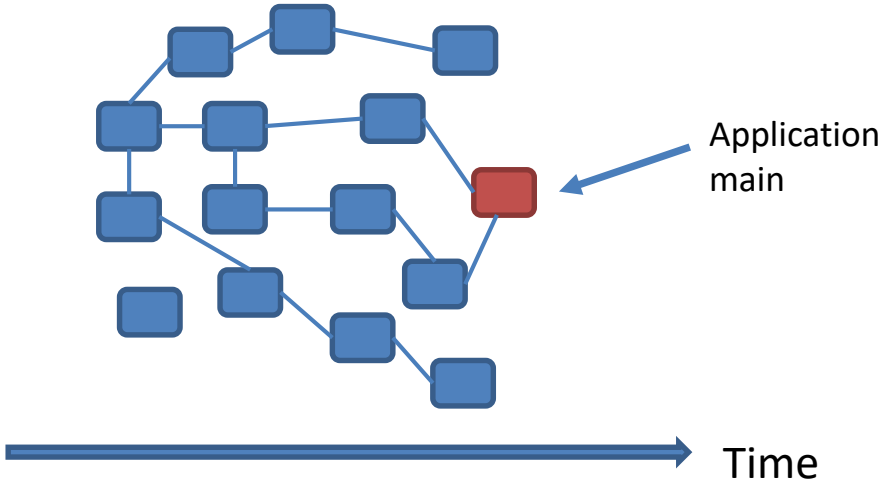
This is the **inside-out** way of working; we focus first on what goes on in the class, then consume it.



Sketching in test code

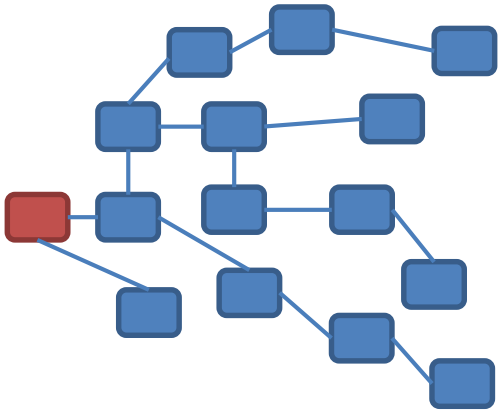
TDD instead pushes us to work **outside-in**. That is, we...

- Consider our **use cases**
- Pick one without dependencies
- **Write a test** demonstrating how we'd solve it with our system under test
- Once we're **comfortable with the API design**, implement code to pass the test

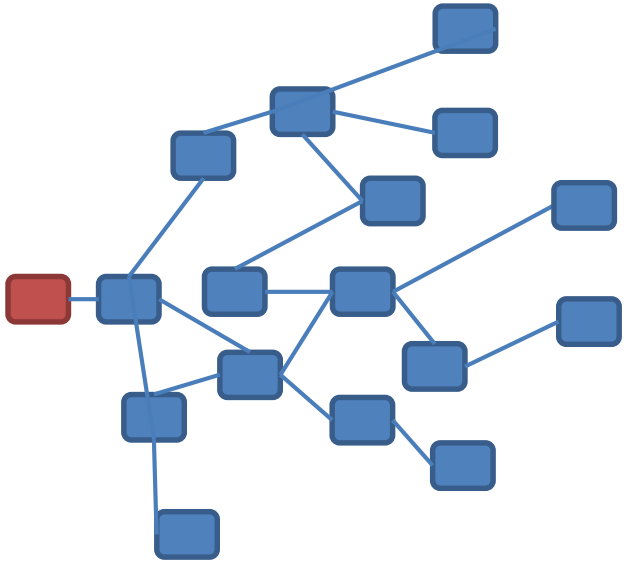


Sketching in test code

Designing the same system with or without TDD will give you two different architectures.



With TDD



Without TDD

Hopefully the version with TDD will be cleaner, have less bugs and will be more maintainable.

Considering a test for making a booking

EDUMENT

Without writing an implementation, we sketch out a first test.

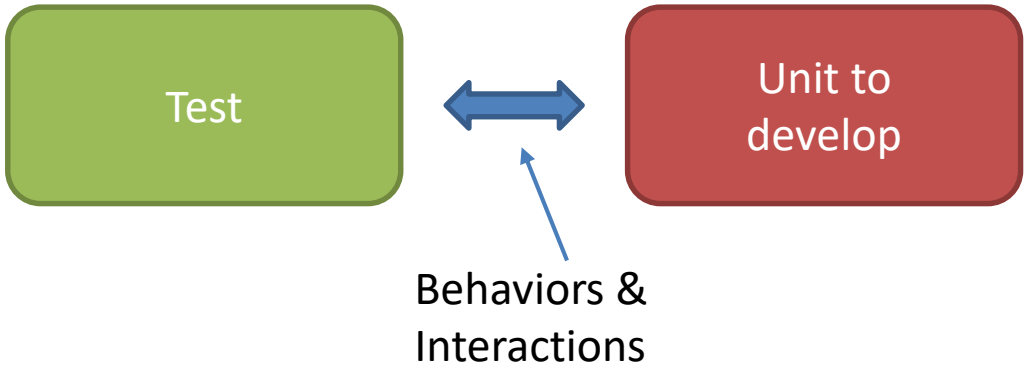
```
[TestFixture]
public class ScheduleTests
{
    [Test]
    public void CanMakeABookingOnAnOpenDay()
    {
        var sut = new Schedule();
        sut.OpenOn(new DateTime(2012, 10, 14));

        sut.MakeBooking("Dave",
            new DateTime(2012, 10, 14, 10, 0, 0),
            TimeSpan.FromMinutes(45));

        var bookings = sut.GetBookingsFor(new DateTime(2012, 10, 14));
        Assert.AreEqual(1, bookings.Count, "Have a single booking");
        Assert.AreEqual("Dave", bookings[0].Name);
    }
}
```

Try to work your **domain's language** into the tests. Take a little time to **try some alternative API designs** rather than settling on the first one you think of!

From behaviors to a model

Writing the tests gives us an opportunity to focus on **outside behaviors** and interactions.

Now we consider the behaviors we need to implement, and define a **model inside of the class** that will enable us to do so.



From behaviors to a model

EDUMENT

The model may involve...

- Private fields, which **encapsulate** the state of the model
- Other classes, if our class is some kind of aggregate

The code inside our class is responsible for **upholding invariants** (things that must always be true), such as "a booking can not be made on a day that is not open".

Defining a model (1)

EDUMENT

A schedule consists of many bookings. Therefore, we'd like a way to represent bookings. We define a simple, immutable type for this.

```
public class Booking
{
    public string Name { get; private set; }
    public DateTime When { get; private set; }
    public TimeSpan Duration { get; private set; }

    public Booking(string name, DateTime when, TimeSpan duration)
    {
        this.Name = name;
        this.When = when;
        this.Duration = duration;
    }
}
```

Defining a model (2)

EDUMENT

We need to represent what days are open, and what bookings have been made so far on each of those days.

When choosing a data structure, it is important to spend a moment considering whether it can provide the behaviors we need, and whether it can do so reasonably efficiently.

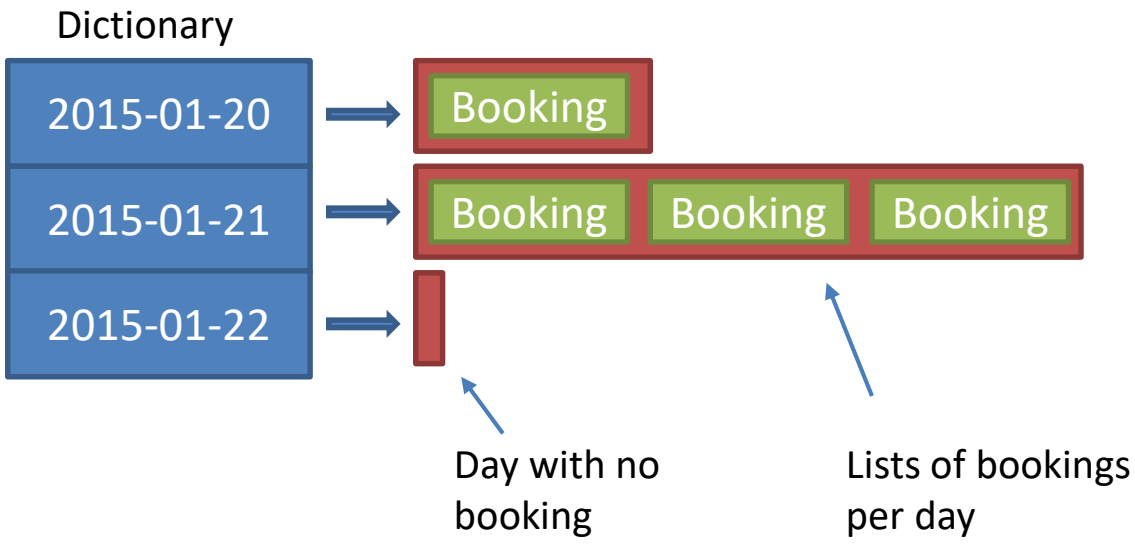
For now, we'll go with something simple:

```
private Dictionary<DateTime, List<Booking>> scheduleByDay =
    new Dictionary<DateTime, List<Booking>>();
```

Defining a model (2)

This means that the **ScheduleByDay** dictionary will contain the following:

```
private Dictionary<DateTime, List<Booking>> scheduleByDay =
    new Dictionary<DateTime, List<Booking>>();
```



When a day is marked as open, we place a new, empty list of bookings on that day. That list will contain all of the bookings that have been made so far on that day.

From compilation error to failing test

EDUMENT

After considering our model and tweaking the generated stub code, the three stub methods in our class look as follows:

```
public class Schedule
{
    private Dictionary<DateTime, List<Booking>> scheduleByDay =
        new Dictionary<DateTime, List<Booking>>();

    public void OpenOn(DateTime dateTime)
    {
        throw new NotImplementedException();
    }

    public void MakeBooking(string name, DateTime when, TimeSpan duration)
    {
        throw new NotImplementedException();
    }

    public List<Booking> GetBookingsFor(DateTime day)
    {
        throw new NotImplementedException();
    }
}
```

We now have **a compiling, failing test**; next, we'll make it pass.

Just enough to get a pass

EDUMENT

It's sensible to spend some time considering a sufficiently good model.

However, this does not mean we should go racing ahead implementing functionality we don't have tests for! Instead, we'll do **just enough to get the test to pass**.

```
public void OpenOn(DateTime dateTime)
{
    scheduleByDay.Add(dateTime.Date, new List<Booking>());
}

public void MakeBooking(string name, DateTime when, TimeSpan duration)
{
    scheduleByDay[when.Date].Add(new Booking(name, when, duration));
}

public List<Booking> GetBookingsFor(DateTime day)
{
    return scheduleByDay[day.Date];
}
```

Testing multiple bookings

EDUMENT

Here's our next test case: multiple booking on one day.

```
[Test]
public void MultipleBookingsRetrievedInTimeOrder()
{
    var sut = new Schedule();
    sut.OpenOn(new DateTime(2012, 10, 14));

    sut.MakeBooking("Dave",
                    new DateTime(2012, 10, 14, 10, 0, 0),
                    TimeSpan.FromMinutes(45));
    sut.MakeBooking("Tina",
                    new DateTime(2012, 10, 14, 9, 0, 0),
                    TimeSpan.FromMinutes(60));

    var bookings = sut.GetBookingsFor(new DateTime(2012, 10, 14));

    Assert.AreEqual(2, bookings.Count, "Have two bookings");
    Assert.AreEqual("Tina", bookings[0].Name);
    Assert.AreEqual("Dave", bookings[1].Name);
}
```

Factoring out boilerplate with Setup methods

Already, creating the SUT and adding an open day in these tests is getting repetitive.

We can improve things by moving the SUT to a field and writing a **SetUp** method, which is run before each test.

```
[TestFixture]
public class ScheduleTests
{
    private Schedule sut;

    [SetUp]
    public void Setup()
    {
        sut = new Schedule();
        sut.OpenOn(new DateTime(2012, 10, 14));
    }

    //.....
```

Factoring out boilerplate with Setup methods

The Setup method will be executed before each and every test in the class.

You should only have one setup method per class.

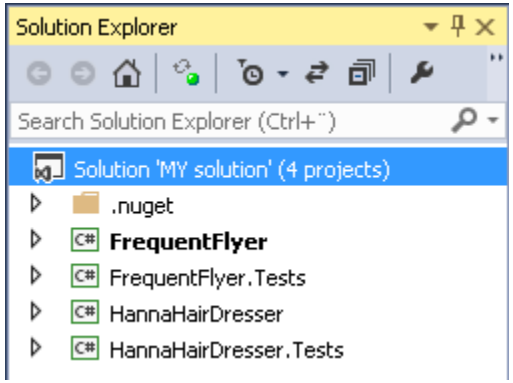
```
[TestFixture]
public class ScheduleTests
{
    private Schedule sut;

    [SetUp]
    public void Setup()
    {
        sut = new Schedule();
        sut.OpenOn(new DateTime(2012, 10, 14));
    }

    //.....
```

Test organization

EDUMENT

The anatomy of our solution and classes are as follow:

Solution structure



One separate test project per real project.

Class structure

```
[TestFixture]
public class ScheduleTests
{
    [SetUp]
    public void Setup()
    { }

    [Test]
    public void CanMakeABookingOnAnOpenDay()
    { }

    [Test]
    public void MultipleBookingsRetrievedInTimeOrder()
    { }
}
```

One optional setup method and one or many tests.

Passing the test

Adding multiple bookings in a day already works. The test fails because we do not return the day's bookings sorted in time order.

The test can be made to pass with a simple addition to the **GetBookingsFor** method.

```
public List<Booking> GetBookingsFor(DateTime day)
{
    return scheduleByDay[day.Date]
        .OrderBy(b => b.When)
        .ToList();
}
```

It's the least invasive change to the code, but additionally means we don't ever return the **List<Booking>** held inside of the class - removing the risk of outside modification.

Considering happy and sad paths

EDUMENT

All of our tests so far have been **happy path** tests.

They make sure that the system under test responds in the appropriate way when we perform a sequence of valid operations.

However, many requirements need **sad path** tests.

These ensure that the system under test indicates failure in an appropriate way when an invalid operation is performed.

Considering happy and sad paths

Sad path tests are primarily interested in two things:

- Does the system under test indicate a failure when an invalid operation is performed?
- Does it indicate the correct kind of failure?

"It's arguably even more important to test that the code doesn't do what it shouldn't do."

-- Mike Bland

http://martinfowler.com/articles/testing-culture.html

Testing exceptions

Testing and exceptions

EDUMENT

In C#, we typically indicate failure using **exceptions**.

NUnit provides a way to assert that a given operation throws the expected type of exception.

```
Assert.Throws<Exception>(() =>
        /* ...operation that should throw... */);
```

The operation that we expect to throw an exception is wrapped up in a lambda expression.

Note that the **exception type** must be specified. This is so we can test that the operation **failed for the correct reason**, not because of an implementation bug.

Testing we can't place overlapping bookings

The following test places a booking at 10:00, then tries to place another for 60 minutes at 9:30. This should result in an **OverlappingBookingException**.

Once again, we can get Visual Studio to stub it for us, giving us a new failing test.

```
[Test]
public void CannotPlaceOverlappingBookings()
{
    sut.MakeBooking("Dave",
                    new DateTime(2012, 10, 14, 10, 0, 0),
                    TimeSpan.FromMinutes(45));

    Assert.Throws<OverlappingBookingException>(() =>
                    sut.MakeBooking("Tina",
                    new DateTime(2012, 10, 14, 9, 30, 0),
                    TimeSpan.FromMinutes(60)));
}
```

Making the test pass

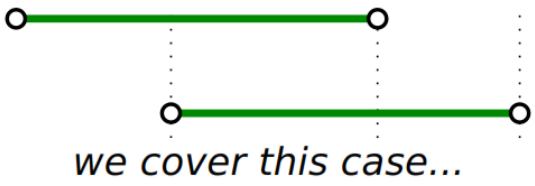
We use a **Linq query** to find any overlapping bookings. If it finds any, we throw the expected exception type, making the test pass.

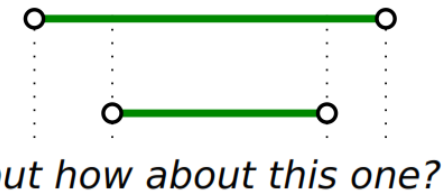
```
public void MakeBooking(string name, DateTime when, TimeSpan duration)
        {
            var dayBookings = scheduleByDay[when.Date];

            var whenEnd = when.Add(duration);

            var overlap = from booking in dayBookings
                          let bookingEnd = booking.When.Add(booking.Duration)
                          where (when > booking.When && when < bookingEnd) ||
                                (whenEnd > booking.When && whenEnd < bookingEnd)
                          select booking;

            if (overlap.Any())
                throw new OverlappingBookingException();

            dayBookings.Add(new Booking(name, when, duration));
        }
```

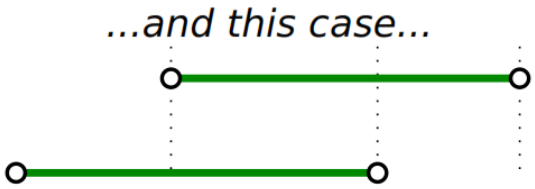
That nagging feeling

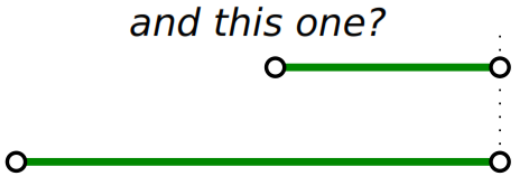
EDUMENT

You may have noticed that the overlap collision code doesn't consider all possible types of overlap. TDD is fairly cool with this.









If you get that nagging feeling that your code fails to take some use case into account, **put it on a to-do list** for later.

This minimizes distraction from the current cycle, and you have a handy item to come back to later and turn into a failing test.

Errors give you next steps

EDUMENT

Creating our own exception types to indicate what went wrong helps us to **get the language of failure into our tests**, as well as ensuring that things fail for the right reasons.

Knowing in what way the operation failed is important for providing the consumer with an idea of what the **appropriate next steps** might be. Thus, writing tests that check for the correct failure mode also helps us to build a consumer-friendly class.

While our exception types are **domain specific**, HTTP's error codes are domain agnostic example of errors giving next steps.

4xx means "don't submit this request again without correcting it", while **5xx** means "sorry, your request may well be OK, but we screwed up processing it".

Tell, Don't Ask

Tell, Don't Ask: Avoiding state inspection

EDUMENT

Imagine we need to implement a feature that suggests the next available time whenever an attempt to book fails due to an overlap.

This could be implemented by the UI querying the `Schedule` object for the day's bookings, then doing the calculation. However, this constitutes a **logic leak**.

```
DateTime SuggestedDate = xxx; //Date to book, from the user

//Ask, get the current bookings
var bookings = sut.GetBookingsFor(SuggestedDate);

//Do some calculations to find next available slot
DateTime NextAvailableTime = . . . ca

 //Tell what to do
 sut.MakeBooking("Joe", SuggestedDate, TimeSpan.FromMinutes(20));
```

Tell, Don't Ask: Avoiding state inspection

Better would be to **include the suggestion in the exception**. This **keeps domain logic in the domain**, rather than having the UI query to obtain it.

This principle is known as **tell, don't ask**. We design our objects to **tell**

their knowledge, rather than **ask** them about their state.

A test for suggesting the next available time

EDUMENT

First, a test. To make it compile, we stub **SuggestedTime**.

```
[Test]
public void OverlapSuggestsCorrectNextTime()
{
    sut.MakeBooking("Dave",
        new DateTime(2012, 10, 14, 10, 0, 0),
        TimeSpan.FromMinutes(45));

    var e = Assert.Throws<OverlappingBookingException>(() =>
                    sut.MakeBooking("Tina",
                    new DateTime(2012, 10, 14, 9, 30, 0),
                    TimeSpan.FromMinutes(60)));

    Assert.AreEqual(
            new DateTime(2012, 10, 14, 10, 45, 0),
            e.SuggestedTime,
            "Correct suggested time");
}
```

First, a refactor

Looking at the code we have and thinking about the feature we need to add, we realize that factoring out the overlap check will help.

```
public void MakeBooking(string name, DateTime when, TimeSpan duration)
{
    if (OverlapsExisting(when, duration))
        throw new OverlappingBookingException();

    scheduleByDay[when.Date].Add(new Booking(name, when, duration));
}

private bool OverlapsExisting(DateTime when, TimeSpan duration)
{
    // Code for checking overlap moved here
}
```

After this refactor, we can use our tests to ensure nothing broke!

Suggesting a time that would work

EDUMENT

We take all bookings after the point we tried to place our one. We then go through them, calculating their end time and seeing if a booking placed at that time would overlap with any after it.

```
private DateTime? SuggestTimeFor(DateTime tried, TimeSpan duration)
{
    var after = scheduleByDay[tried.Date].Where(b => b.When >= tried);

    foreach (var booking in after)
    {
        var finishedBy = booking.When.Add(booking.Duration);

        if (!OverlapsExisting(finishedBy, duration))
            return finishedBy;
    }

    return null;
}
```

Integrating our time suggestion

Finally, we incorporate the suggestion into the exception by passing it to the constructor.

```
if (OverlapsExisting(when, duration))
    throw new OverlappingBookingException(SuggestTimeFor(when, duration));
```

For completeness, here is **OverlappingBookingException:**

```
public class OverlappingBookingException : Exception
    {
        public DateTime? SuggestedTime { get; private set; }

        public OverlappingBookingException(DateTime? suggestedTime)
        {
            this.SuggestedTime = suggestedTime;
        }
    }
```

Testable design and good OO design are related

TDD makes us constantly consider how we will test the classes we implement.

The desire to control the information our classes depend on directs our designs to follow the **dependency inversion principle**.

Wanting to write tests that rely on what the class tells, rather than (often fragile) state inspection, helps us focus on placing **responsibility** in the right places, giving **cohesive** classes.

Don't resist TDD because it makes you design differently. Most likely, it will make you design better!

Testable design and good OO design: TDD book quote

EDUMENT

Kent Beck confirms this in the **TDD by Example** book.

"Isolating tests encourages you to compose solutions out of many highly cohesive, loosely coupled objects.

I always heard this was a good idea, and I was happy when I achieved it, but I never knew exactly how to achieve high cohesion and loose coupling regularly until I started writing isolated tests."

-- Kent Beck

The Law of Demeter

The Law of Demeter

The **Law of Demeter** can be useful in assessing how well we're following the Tell, Don't Ask principle. It suggests we should only call methods...

- *On objects that are in instance fields*
- *On objects that were passed as parameters*
- *On objects created in the current method*

Of note, it suggests that we **avoid calling methods on objects returned to us** as the result of some other call.

That is, avoid designs that lead to chains like:

```
schedule.GetDay(d).GetBookings()
```

"You can play with your own toys, toys given to you (but don't take them apart), and toys you made yourself"

http://c2.com/cgi/wiki?LawOfDemeter

Applying the LoD: focus on the SUT

EDUMENT

In our tests, we should **only act on the System Under Test**. It's highly preferable if the arrange and assert parts of our test also only work directly against the SUT.

So does this violate the Law of Demeter?

```
var bookings = sut.GetBookingsFor(new DateTime(2012, 10, 14));
Assert.AreEqual(1, bookings.Count, "Have a single booking");
Assert.AreEqual("Dave", bookings[0].Name);
```

Technically, perhaps. But really, **bookings** is **just a List** and a **Booking** is **just a data vessel**. They exist to expose information!

Everything may be an object in C#, but some objects are more object-y than others.

Test method granularity

EDUMENT

How big should a unit test be?

Trying to answer this with a target number of lines of code is unhelpful. Getting the granularity right is better achieved by following these rules:

- The **arrange** part of the test does just what's needed for the act part.
- The **act** part of the test should be a **single method call**.
- The **assert** part of the test should focus on checking the consequences of the act.
- Each unit test does a single arrange/act/assert.

Remember - tests are methods! You can **factor out repetitive arrangement or assertion** and place it in private methods!

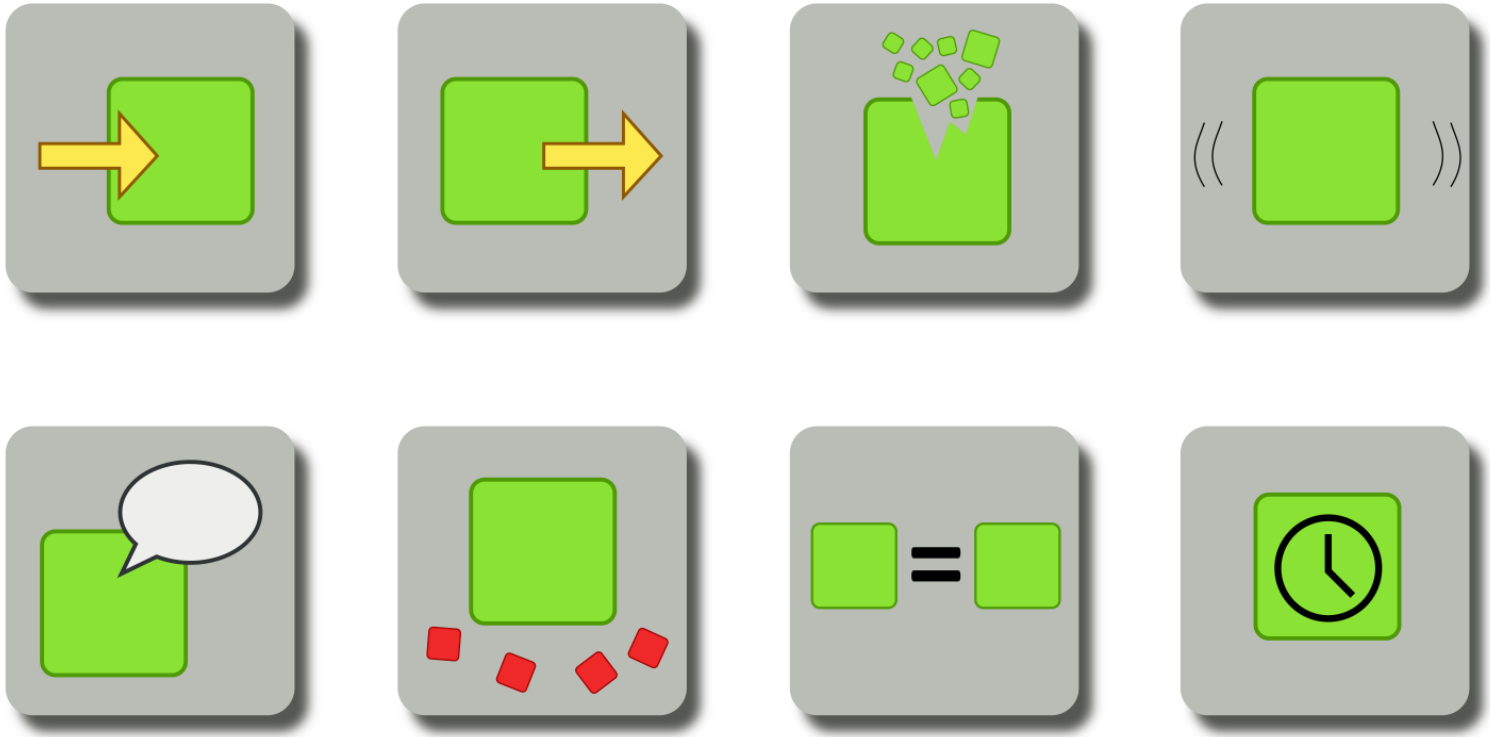
Anti-patterns to avoid

Anti-patterns to avoid

EDUMENT

Let's look at a few mistakes other people have made when learning to write things with tests.

With luck, it will help us avoid known pitfalls and get more out of testing!



Anti-patterns to avoid

EDUMENT

Environment leaking into a test - files or databases or environment

variables needing to be set in a certain way for the test to succeed, leading to fragile, non-portable tests.

```
[SetUp]
public void Setup()
{
    // only works on Jeff's computer :(
    ClearDataBase();
}
```

As we learn more about the nature of unit tests, we will find that environment leaking is more of a symptom than the original problem. Good testable units have injected dependencies.

Anti-patterns to avoid

A test leaking out into environment - leading to ordering dependencies, and some tests passing of failing depending on what ran before them.

```
[Test]
public void Test1()
{
    customer.name = "Torben";
    // ... more testing ...
}

[Test]
public void Test2()
{
    // assumes customer.name is "Torben"
}
```

Make your tests isolated and independent!

Anti-patterns to avoid

Intimate knowledge of internals - the test code needs to violate encapsulation in order to get enough details to assert against.

```
[Test]
public void Test()
{
    Assert.IsTrue(sut.privateField == 42); // yuck!
}
```

Magic numbers in code is a code smell!

Anti-patterns to avoid

Excessive expectations - expecting output to be **exactly** of a certain form, or values to come in a certain order, when such details don't matter. **Assert only what matters**.

```
[Test]
public void Test()
{
    var data = new Dictionary<string, int>
    {
        {"foo", 1},
        {"bar", 2},
    };

    var ordered = data.ToList();

    Assert.IsTrue(ordered[0].Key == "foo" && ordered[0].Value == 1);
    Assert.IsTrue(ordered[1].Key == "bar" && ordered[1].Value == 2);
}
```

Anti-patterns to avoid

Tests which talk -the test, when run, clutters up the console with

irrelevant output, even when it passes.

```
[Test]
public void Test()
{
    // ...
    Console.WriteLine("Lots of debug output");
    sut.method(); // This method also outputs noise
    // ...
}
```

Anti-patterns to avoid

Testing for an error, but not which one - leading to actual failures going undetected, because "an error" was what was expected.

```
[Test]
    public void Test()
    {
        Assert.Throws<Exception>(
            () => /* do dangerous stuff */);
    }
```

Be specific about expected error conditions.

Anti-patterns to avoid

EDUMENT

Tautology - the test always trivially passes, and tests nothing. Was probably written after the implementation code and never observed to fail.

```
[Test]
public void Test()
{
    sut.frobnitz = 3;
    Assert.AreEqual(3, sut.frobnitz, "Check it's 3");
    // Oddly enough, turns out it's always 3
}
```

The test is always true and gives us no value

Anti-patterns to avoid

Sllllllooooooouuuuuuwww - the test relies on a database, the file system, or an external service, and makes your unit test suite take longer than an eye-blink.

```
[Test]
 public void Test()
 {
     account.Deposit(100);
     Thread.Sleep(1000 * 60 * 60 * 24 * 365 * 10); // now wait ten years
     Assert.AreEqual(100 * Math.Pow(interest, 10), account.Balance);
 }
```

The tests should always be as fast as possible, **every ms counts**!

Exercise #2

A chance to write more sophisticated tests.

Our domain this time is travel agencies. You'll implement a system for scheduling day-long tours.

We'll consider both happy and sad paths in our tests.