Environmental Issues: Time, UIs, Databases, oh my

Keeping the outside at a distance

Following the **Dependency Inversion Principle** helped us to write tests that **controlled the dependencies** of the system under test. This made our tests repeatable and not dependent on the outside world.

There are various places where it takes some careful thought and attention to keep the outside world under our test's control.

In this section, we'll consider three of them in detail, taking a careful look at why it's important and how we can achieve the repeatability and environment independence that we desire.

Taking control of time

Working with **dates and times** is difficult. There are plenty of opportunities for boundary errors and bad assumptions. Therefore, its is valuable to be able to write good tests for such code.

There are some specific traps to look out for:

- Code that gets the current system time directly (for example, using `DateTime.Now`)
- Code that sleeps (for example, using `Thread.Sleep(...)`)
- Code that sets up timers and waits for them to be fired (for example, using `System.Timers`)

Can you pass in the date/time?

If you have code that is dependent on dates and times, the first question to ask is whether you can pass the date and time information in **as an argument**.

For example, if you have a system that is processing orders, you may require that the date and time the order was placed to be supplied to the system.

In yesterday's example of an appointment system, we also passed all dates and times in to the system as arguments. The code worked with dates and times, but in a **pure** way, and thus was **not dependent on the environment**.

If you must know the date/time...

EDUMENT

Sometimes, a notion of current time really is needed. Let's consider an additional requirement from Wing and a Prayer Airlines:

The rate at which we send SMS messages should be throttled. We should not send a message about the same flight more than once every 10 minutes.

A design that sees us passing the current time in with every call feels odd here. However, we'd still like to be able to control it from our tests.

Dependency Inject the time

EDUMENT

An easy way out of this is to turn to interfaces and DI again:

```
public interface IDateTime
{
    DateTime Now();
}
```

We can re-use this - and a standard implementation of it - throughout our system. We add it as a dependency to the **SUT**, and update our test's **SetUp** method.

```
[SetUp]
public void Setup()
{
    smsStub = Substitute.For<ISMSSender>();
    passStub = Substitute.For<IPassengerInfoDAL>();
    dateStub = Substitute.For<IDateTime>();
    sut = new Notifier(smsStub, passStub, dateStub);
}
```

A time-based test

EDUMENT

An easy way out of this is to turn to interfaces and DI again:

```
[Test]
public void DoNotSendMultipleMessagesWithinTenMinutes()
{
    //Arrange
    passStub.PassengersOnFlight("WP123").Returns(new List<Passenger>
    {
        new Passenger {Name = "Dave", Mobile = "123456"},
        new Passenger {Name = "Lena", Mobile = "987654"}});

    //Act
    var fakeTime = DateTime.Now;
    dateStub.Now().Returns(fakeTime);
    sut.FlightCancelled("WP123");

    dateStub.Now().Returns(fakeTime.AddMinutes(5));
    sut.FlightCancelled("WP123");

    //Assert
    smsStub.Received(1).Send(Arg.Any<string>(), Arg.Any<string>());
}
```

Implementation: first a refactor

EDUMENT

Sends messages is duplicated in three places in our code!

```
public void FlightCancelled(string flight)
{
    var message = "Unfortunately, your flight " + flight +
        " has been cancelled.";

    foreach (var pass in dal.PassengersOnFlight(flight))
    {
        smsSender.Send(pass.Mobile, message);
    }
}
```

Applying **Once And Only Once**, we factor it out:

```
private void SendMessage(string flight, string message)
{
    foreach (var pass in dal.PassengersOnFlight(flight))
        smsSender.Send(pass.Mobile, message);
}
```

Implementation: injected time

EDUMENT

After the refactor, the new requirement can be handled in a single place:

```
private Dictionary<string, DateTime> lastMessage;

private void SendMessage(string flight, string message)
{
    if (lastMessage.ContainsKey(flight))
    {
        var sinceLast = dateTime.Now().Subtract(lastMessage[flight]);
        if (sinceLast.TotalMinutes < 10)
            return;
    }

    foreach (var pass in dal.PassengersOnFlight(flight))
        smsSender.Send(pass.Mobile, message);

    lastMessage[flight] = dateTime.Now();
}
```

Sleeping testability

We know that our unit tests should execute quickly. So how do we cope with code that wants to sleep for half an hour?

We have various options:

- Handle it as we did with getting the current time: create an interface with a `Sleep` method, dependency inject and use it, then in the tests just log the amount of time slept for.
- Separate out the logic executed each time period and the sleeping, either to different methods or different classes, whichever feels most natural

Timers can be handled in a similar kind of way.

Presentation patterns for testable UIs

Some user interfaces are trivial:

grab some data and show it, or take the contents of a few text boxes and send it off to some business logic.

In these cases, our main concern will be to isolate the business logic in the backend of our system from the frontend code. We may decide the UI itself has no significant logic worth testing.

Presentation patterns for testable UIs

EDUMENT

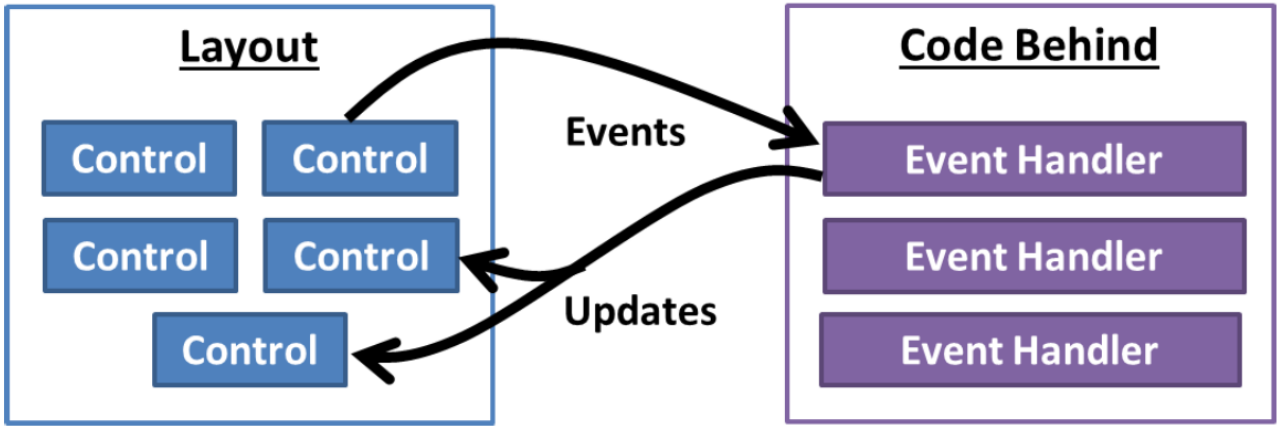
However, UIs that offer a **rich user experience** often have interesting **presentation logic** that is valuable to test. Our product's UI may be a large part of its business value!

There are a number of **presentation design patterns** that can help us to isolate presentation logic into a plain object, decoupling it from the UI framework and making it testable.

Why do we need presentation patterns?

EDUMENT

In many UI frameworks, you end up writing event handlers & using controls that are supplied by the UI framework itself.

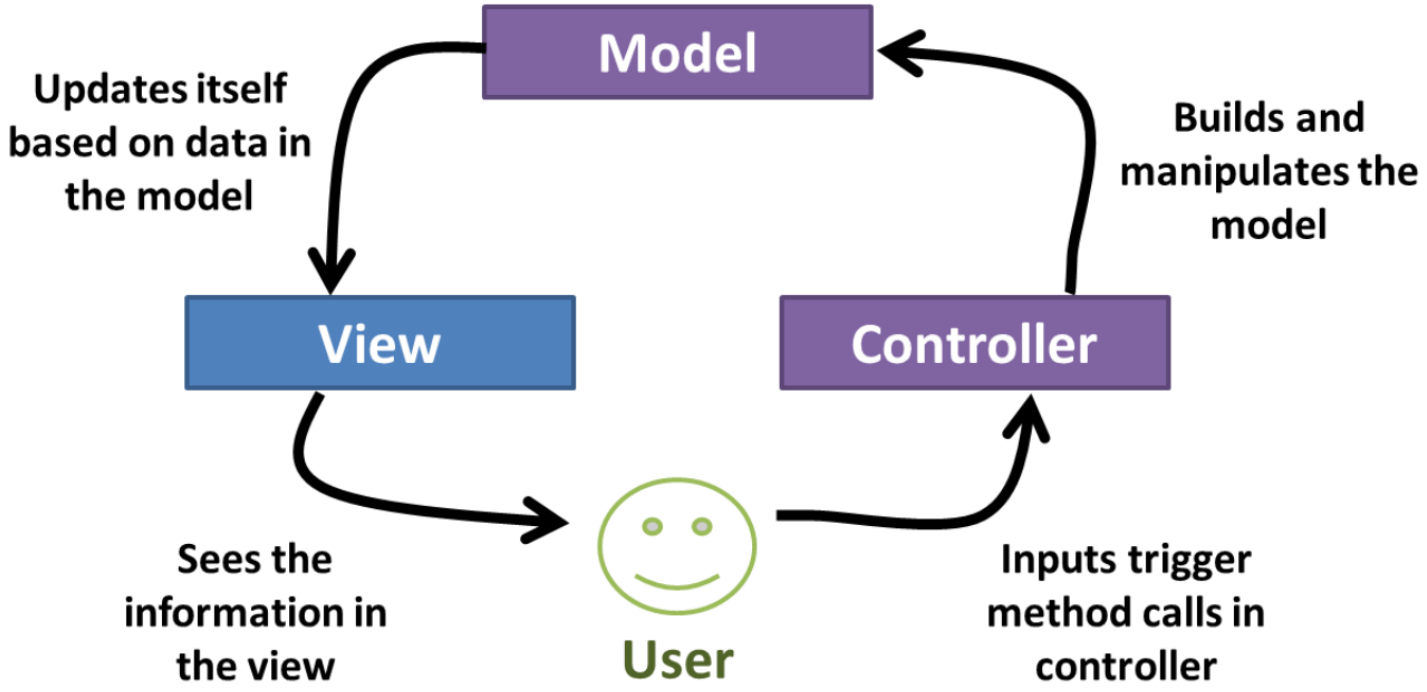


These objects are typically hard (and may be impossible) to replace with test doubles, and are not realistic to create for real in our automated tests.

This is a problem with WinForms, WebForms & WPF.

ASP.NET MVC

The Model View Controller pattern collects presentation logic into a controller class. This is a normal class, ideal for testing.



The view is typically written in a markup or template language. The nature of the model varies.

Concrete Example: Beer Review in ASP.NET MV

EDUMENT

Our sample application is a web site for reviewing beer. It is developed using Microsoft's ASP.NET MVC framework.

Views are written in template languages (the most common one being known as Razor). They should not contain any decision-making logic. They take information from the **model** and present it.

The **model** can be anything we like. A simple **DTO**, a domain or business object, or perhaps something created specially for holding the information we will include in the view.

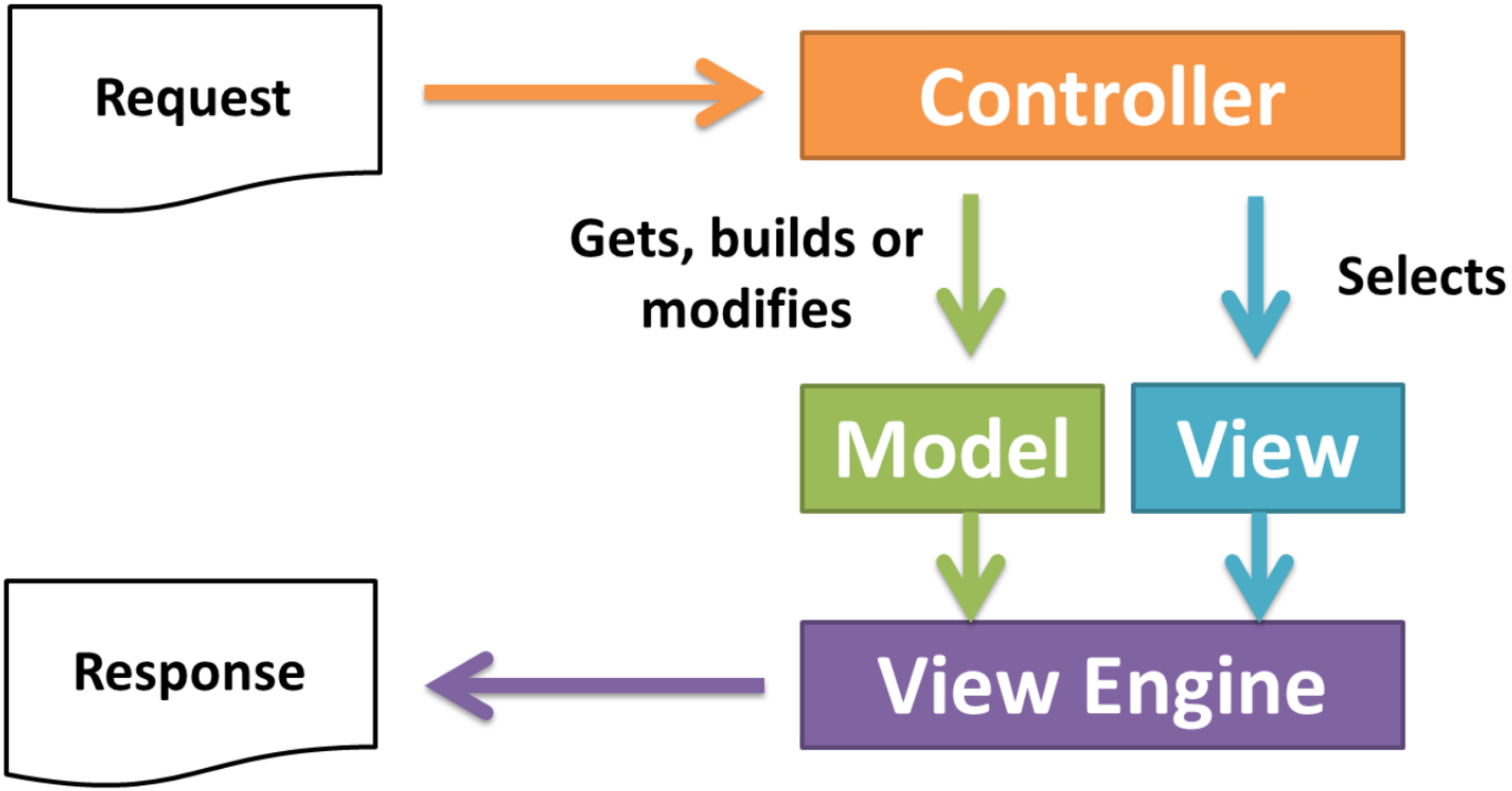
The **controller** is a class. It should inherit from a **`Controller`** base class

ASP.NET MVC Request Processing

EDUMENT

Requests are processed by **action methods** on the controller, which take the user input as **parameters**. They produce a **model** and select a **view**,

the pair forming the method's **return value**.



Controllers and DI

Typically, a web application depends on some kind of database or backend web service. If we want to test our controllers in isolation, we need to take control over these. We already know how to do that: **dependency inversion**!

```
public class BeerController : Controller
{
    private IBeerDb DB;

    public BeerController(IBeerDb DB)
    {
        this.DB = DB;
    }

    // Action methods come here
}
```

Test Setup

In controller tests, the controller is the System Under Test. We create it in the **SetUp** method, create a stub object for the database, and inject it into the controller's constructer. Nothing unfamiliar here!

```
[TestFixture]
 public class BeerTest
 {
     private IBeerDb dbStub;
     private BeerController sut;

     [SetUp]
     public void Setup()
     {
         dbStub = Substitute.For<IBeerDB>();
         sut = new BeerController(dbStub);
     }
 }
```

Testing the expected model is provided

We call the action method, passing parameters (which are obtained from the URL when processing a real web request). We then assert on the object produced.

```
[Test]
public void DetailsReturnsPopulatedModel()
{
    var testModel = new BeerDetailsModel
    {
        Name = "Test Beer",
        Description = "Tasty!"
    };

    dbStub.GetBeerDetails(0).Returns(testModel);

    var result = sut.Details(0) as ViewResult;

    Assert.IsNotNull(result, "Got a ViewResult");
    var model = result.Model as BeerDetailsModel;
    Assert.IsNotNull(model, "Got correct type of model");
}
```

Testing for a 404 response

Here, we check that a `null` result from the database lookup is turned by the controller into a `HttpNotFoundResult`, rather than giving an exception or pretending everything was fine.

```
[Test]
public void InvalidIdToDetailsReturns404()
{
    dbStub.Stub(x => x.GetBeerDetails(0)).Return(null);

    var result = sut.Details(0) as HttpNotFoundResult;

    Assert.IsNotNull(result,
       "Get HTTP not found result when ID is invalid");
}
```

Testing Submitted Data

Once again, the use of parameters for input makes testing easy.

We create an instance of the model and populate it as we wish in the arrange phase of our test, then pass it to the action method in the act phase.

```
[Test]
public void SubmittingNewReviewRedirectsAfterwards()
{
    var model = new ReviewModel {
        Rating = 4, Comments = "Awesome beer!"
    };

    var result = sut.SubmitReview(1, model) as RedirectToRouteResult;

    Assert.IsNotNull(result, "New review submission redirects");
}
```

MVC: Testable, with discipline

MVC makes it relatively straightforward to write tests for our controllers.

This depends on correct and careful application of the pattern. Clearly, if one was to write controllers that did not follow the dependency inversion principle, or implemented business logic inside the view, the solution would not be testable.

MVC is **very popular on the web**, with almost all languages having (often many) MVC frameworks available. It fits well with the stateless nature of HTTP.