Diseño de Aplicaciones 2 - Tecnología

Testing

Repaso, Test Doubles

Repaso

DIFERENTES TIPOS DE PRUEBAS

- Pruebas unitarias: verifican una pequeña unidad de código, función o método.
- Pruebas de integración: verifican la integración entre varias unidades de código. La idea es probar a nivel de módulos.
- Pruebas funcionales: verifican las funcionalidades del sistema de acuerdo a los requerimientos.
- Hay más... performance, aceptación...

F.I.R.S.T

- Fast: las pruebas unitarias deben ser de ejecución rápida.
- Isolated / Independent: las pruebas deben ser auto-conenidas.
- Repeatable: repetibles, es decir que tienen que dar el mismo resultado cada vez que ejecuto (en las mismas condiciones).
- Self-validating: la prueba debe automáticamente determinar su éxito o fracaso.
- Timely: idealmente, escritas antes del código o cuanto antes.

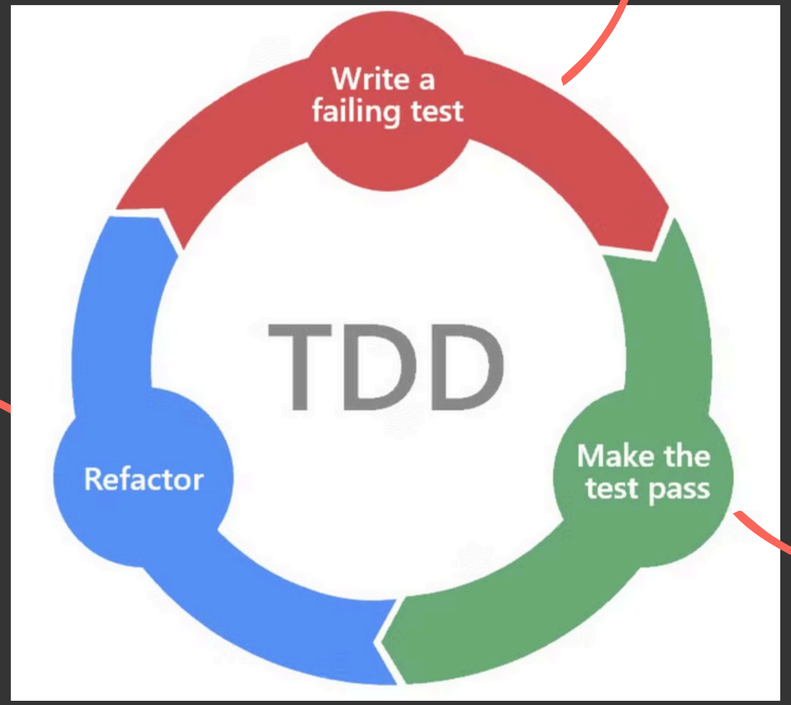
Arrange Act Assert

```
// Arrange
var calculator = new Calculator();
var number1 = 5;
var number2 = 3;

// Act
var result = calculator.Add(number1, number2);

// Assert
Assert.AreEqual(8, result);
```

TDD

TEST DRIVEN DEVELOPMENT

RED



GREEN

REFACTOR

TEST DOUBLES

- Objeto que reemplaza a otro a los efectos de realizar pruebas y simular su comportamiento

- Tipos:
  - Dummies
  - Fake
  - Stubs
  - Spies
  - Mocks

DUMMIES

- Objeto “vacío” que se usa para cumplir con una dependencia.
- No nos importa como se comporta, solamente lo usamos para cumplir con una necesidad que tiene otro método (el cuál queremos probar).
- Generalmente se utilizan para completar una lista de parámetros.

EJEMPLO 1

```
// Implementación en C# (MsTest)
public void SendEmail_ShouldSendEmail()
{
    // Arrange
    var dummyEmailService = new Mock<IEmailService>();

    // Lo uso para cumplir con una dependencia...
    var emailSender = new EmailSender(dummyEmailService.Object);
    var recipient = "test@example.com";

    // Act
    var result = emailSender.SendWelcomeEmail(recipient);

    // Assert
    // Acá vemos que buscamos probar otra cosa...
    Assert.IsTrue(result);
}
```

FAKE

- Son objetos con implementaciones reales que, por lo general, toman algún atajo o simplifican las cosas.
- Estos atajos los hacen inutilizables para producción.
- Nos da un comportamiento más real (no por completo), pero sin las complicaciones que nos puede dar una dependencia real.

EJEMPLO 2

```
public class FakeEmailRepository : IEmailRepository
{
    private List<string> _emails = new List<string>();

    public void SaveEmail(string recipient, string subject, string body)
    {
        _emails.Add(recipient);
        // Si bien cumplo con la definición del contrato, hay parámetros que
        // mi implementación fake no utiliza (subject, body).
    }

    public List<string> GetEmails()
    {
        return _emails;
    }
}
```

```
// Ahora sí, voy a testear...
public void SendEmail_ShouldSaveEmailInRepository()
{
    // Arrange
    var fakeEmailRepository = new FakeEmailRepository();
    var emailSender = new EmailSender(fakeEmailRepository);
    var recipient = "test@example.com";

    // Act
    emailSender.SendWelcomeEmail(recipient);

    // Assert
    Assert.AreEqual(1, fakeEmailRepository.GetEmails().Count);
    Assert.AreEqual(recipient, fakeEmailRepository.GetEmails()[0]);
}
```

STUBS

- Reemplazan a otro componente durante una prueba para obtener resultados consistentes.
- Implementación de un contrato que devuelve valores hardcoded.
- No contienen lógica adicional, como si lo pueden tener los fakes. En este caso, simplificamos aún más la lógica.

EJEMPLO 3

```
// Nuevamente, primeros implementamos...
public class StubEmailRepository : IEmailRepository
{
    public void SaveEmail(string recipient, string subject, string body)
    {
        // No hace nada, solo me importa cumplir con el contrato
    }

    public List<string> GetEmails()
    {
        // Devuelve una lista predefinida, a esto nos referimos con 'hardcoded'
        return new List<string> { "email1@example.com", "email2@example.com" };
    }
}
```

```
// Testeando... (en otro archivo)
public void GetAllSentEmails_ShouldReturnPredefinedEmails()
{
    // Arrange
    var stubEmailRepository = new StubEmailRepository();
    var emailSender = new EmailSender(stubEmailRepository);

    // Act
    var emails = emailSender.GetAllSentEmails();

    // Assert
    Assert.AreEqual(2, emails.Count);
    Assert.AreEqual("email1@example.com", emails[0]);
    Assert.AreEqual("email2@example.com", emails[1]);
}
```

SPIES

- Stubs que además guardan registro de la forma en que fueron llamados.
- Están pendientes si se da alguna condición o no.
- Tienen un uso muy particular.

EJEMPLO 4

```
public class SpyEmailRepository : IEmailRepository
{
    public void SaveEmail(string recipient, string subject, string body)
    {
        // Guardo registro del último mail recibido
        LastRecipient = recipient; // Se omiten las declaraciones
        LastSubject = subject;
        LastBody = body;
    }

    public List<string> GetEmails()
    {
        // No nos importa en este caso
    }
}
```

```
public void SaveEmail_ShouldRecordLastEmailDetails()
{
    // Arrange
    var spyEmailRepository = new SpyEmailRepository();
    var emailSender = new EmailSender(spyEmailRepository);

    var recipient = "test@example.com";
    var subject = "Test Subject";
    var body = "Test Body";

    // Act
    emailSender.SendEmail(recipient, subject, body);

    // Assert
    // Acá puedo notar el uso. Chequeo en base al registro que guarde en mi spy
    Assert.AreEqual(recipient, spyEmailRepository.LastRecipient);
    Assert.AreEqual(subject, spyEmailRepository.LastSubject);
    Assert.AreEqual(body, spyEmailRepository.LastBody);
}
```

MOCKS

- Son objetos preprogramados para cumplir con expectativas que tenemos sobre ellos.
- Pueden tirar excepciones si reciben una llamada que no esperaban.
- Se chequean durante la verificación para asegurar que tienen todas las llamadas que esperaban.

EJEMPLO 5

```
public void SendEmail_ShouldCallSaveEmailWithExpectedParameters()
{
    // Arrange
    var mockEmailRepository = new Mock<IEmailRepository>();
    var emailSender = new EmailSender(mockEmailRepository.Object);

    var recipient = "test@example.com";
    var subject = "Test Subject";
    var body = "Test Body";

    // ¡Acá esta la gran diferencia! A esto nos referimos con 'preprogamados'...
    mockEmailRepository
        // Le digo los parámetros exactos que espera...
        .Setup(repo => repo.SaveEmail(It.Is<string>(s => s == recipient),
                                      It.Is<string>(s => s == subject),
                                      It.Is<string>(s => s == body)))
        .Verifiable("No se verifico!"); // Mensaje para nosotros

    // Act
    emailSender.SendEmail(recipient, subject, body);

    // Assert
    // Ahora verifico si fue llamado con los parámetros que le dije que esperaba...
    mockEmailRepository.Verify(repo => repo.SaveEmail(recipient, subject, body), Ti
    mockEmailRepository.Verify();
}
```