Luz Verde

Implementando Unit Testing

_undefined conf x Women Who Code BA

Bonus: Tengo dos gatos y un tatuaje de ellos en cada brazo

About Me

- Trabajo como desarrolladora desde 2015
- Fui .NET developer en software factories
- *Casi* técnica en diseño y programación de videojuegos

Unit testing

¿Qué es y para qué sirve?

Unit testing

1.
Escribir código que prueba otro código

2.
Instanciar pequeñas porciones de la aplicación

3.
Verificar comportamiento de forma independiente

4.
Cada prueba reporta su resultado de forma individual

5.
Si todas las pruebas pasan, el código *posiblemente* funcione

6.
Si una sóla prueba falla, **es seguro** que hay una falla en el código

Unit testing

Verifica una pequeña parte del comportamiento, aislada del resto de la implementación y del ambiente

Integration Testing

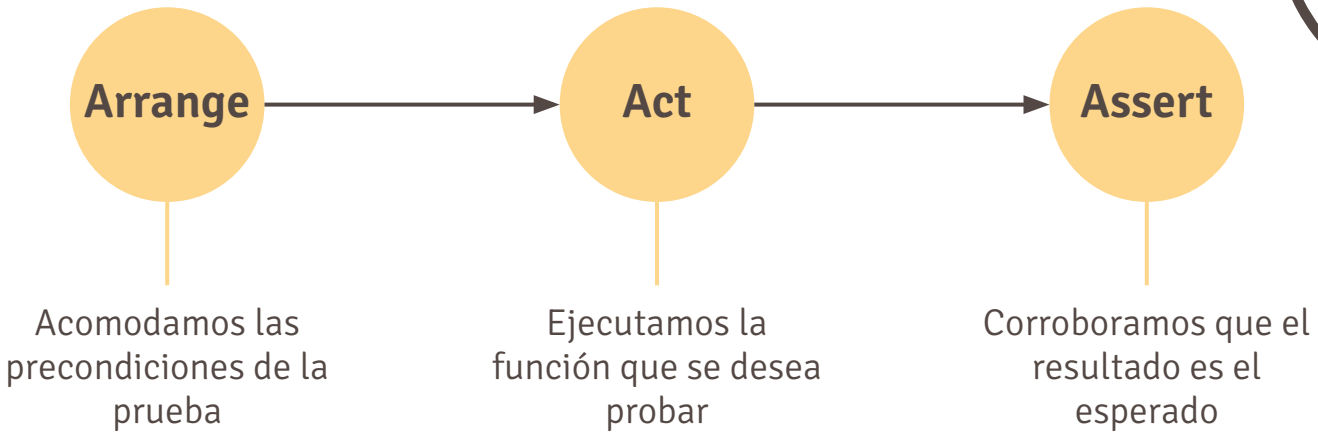
Cubre interacciones entre distintos componentes, en un ambiente similar a la vida real

Arrange, Act, Assert

Un patrón de implementación

AAA Pattern

Antes de comenzar a hacer unit testing, debemos definir cuál va a ser nuestro sistema a probar (SUT). Normalmente esto se traduce en la clase que queremos probar.



Inyección de dependencias

El primer paso para poder realizar unit tests, que sean independientes entre sí y que no estén acoplados a una implementación específica (por ejemplo una base de datos o la hora de un servidor) es tener la capacidad de **aislar el código** que se desea probar.

Para poder lograr esto, utilizaremos el principio de **Inversión del Control (IoC)**. En esta técnica, el código recibe el flujo de control desde el exterior, en lugar de crearlo por sí mismo.

Así, utilizando **interfaces** podemos saber qué comportamientos esperar de una clase, y qué dependencias necesita. Las clases dependientes conocen a cuáles tendrán acceso, pero recibirán la implementación concreta en tiempo de ejecución.

Inyección de dependencias

Pero también hay que cuidar la **Inversión de Dependencias.**

1. Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones
2. Las abstracciones no deben depender de los detalles, sino que a la inversa.

Las interfaces son nuestras **abstracciones**. Los detalles, es decir, la implementación dependen de ellas. Y de varias implementaciones posibles, **podemos elegir cuál usar** (inyectar), sin que esto afecte a las capas de niveles superiores.

SOLID

- Single Responsibility Principle
- Dependency inversion

¡Manos a la obra!

Un ejemplo en .NET

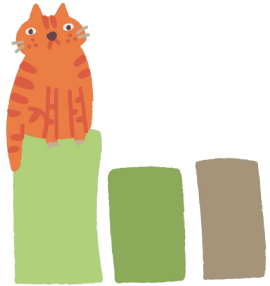
Code Coverage

Conceptos y recomendaciones

Code Coverage

Es una **métrica** que ayuda a saber qué porcentaje del código fue cubierto por la suite de tests.

La mayoría de las herramientas de code coverage, permiten ver qué partes del código fueron cubiertas por los unit tests y cuáles no.



No hay números mágicos sobre el coverage, todos los proyectos son diferentes. Normalmente **se apunta a tener un coverage del 80%**. Lograr un porcentaje más alto normalmente no trae demasiados beneficios respecto al costo.

En ningún caso, tener un porcentaje alto de coverage **no implica** directamente la **calidad** de la suite de tests.

Referencias

- Demo project: https://github.com/kawzar/cat-shelf
- Adaptive Code via C#: Agile coding with design patterns and SOLID principles (Gary McLean Hall)
- Unit Tests, How to Write Testable Code and Why it Matters

¡Gracias!

hmacarena@gmail.com

@Kawzar__

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.

Cat illustrations from Icons8