TDD

CVDS

Principios STUPID

- **S**ingleton
- **T**ight Coupling
- **U**ntestability
- **P**remature Optimization
- **I**ndescriptive Naming
- **D**uplication (DRY)

Errores en el Software

- Hay errores que tan solo son defectos
  (Un error en una etiqueta)
- Hay errores costosos (‘+’ por ‘-’)
  (En un cajero automático)
- Hay errores que cuestan vidas humanas
  (Antimisil guerra del golfo)

Pruebas

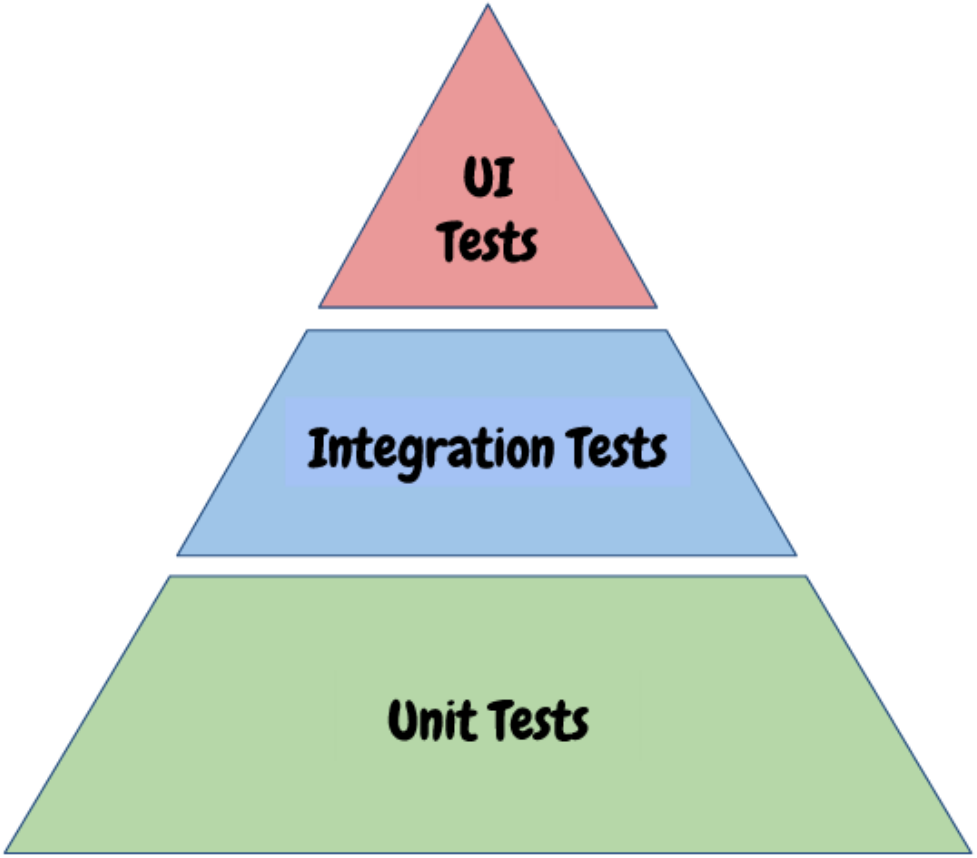
Ventajas

- Ayuda a encontrar algunos defectos de Software
- Entre más pruebas más confiabilidad en el software (?)
- Cuando se modifica el software, ayuda a encontrar errores que son introducidos por las modificaciones

Desventajas

- Cuantas pruebas se pueden hacer a mano? 1 - 100 - 1000 - 10000?
- Se hacen muy pocas pruebas
- No es muy agradable hacer pruebas
- No se prueban todos los escenarios posibles

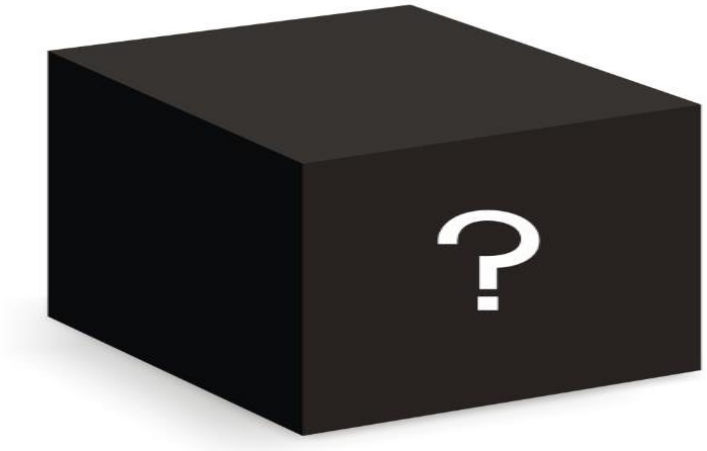
Test Automation Pyramid



Tipos de Pruebas

Caja Blanca

Caja Negra

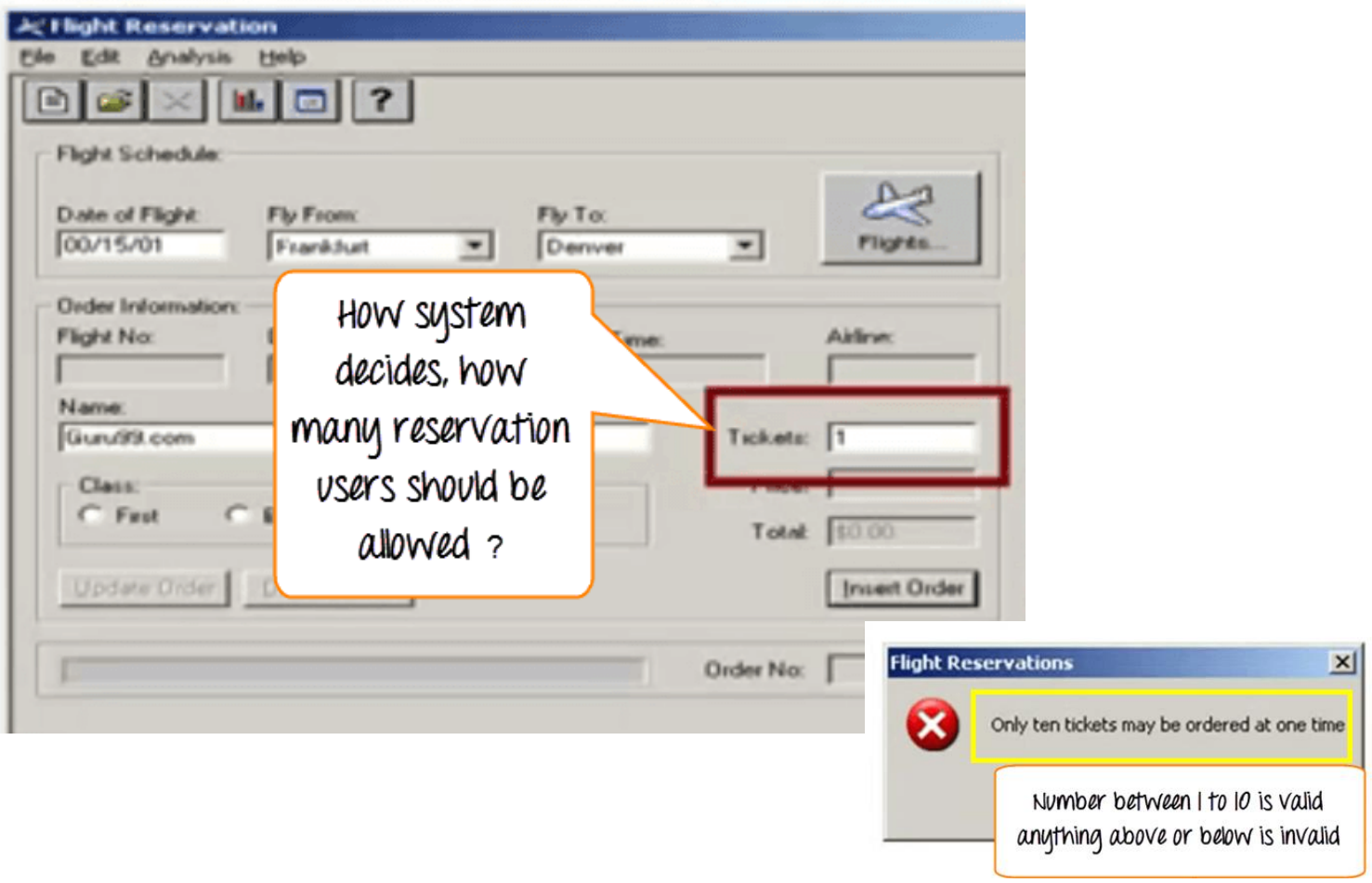


- Cómo comprobar que una función -sin conocer su implementación- funciona de acuerdo con la especificación?

Problema de la dimensionalidad

- Pruebas intuitivas?
- Pruebas exhaustivas?
- Pruebas aleatorias?

Reservación



Tomado de:
http://www.guru99.com

Reservación

- Backend:

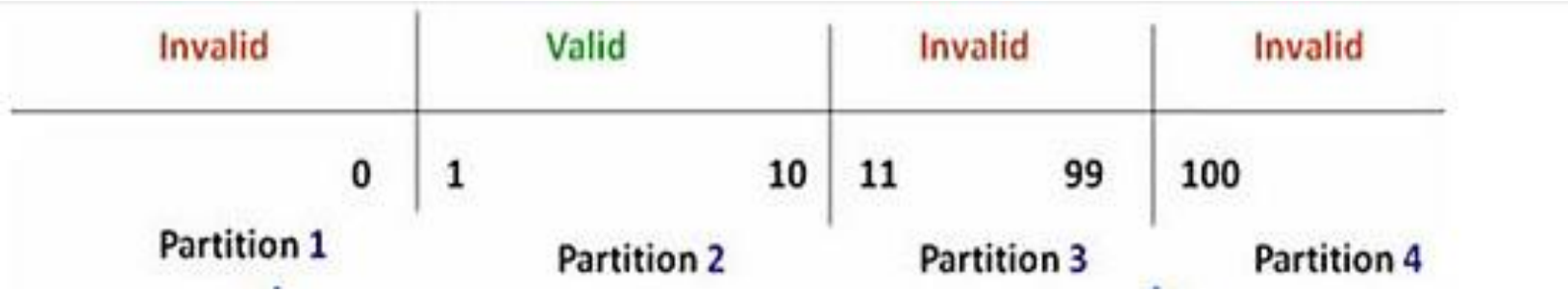
```
/**
* @obj registrar una reserva para el número de tiquetes indicado,
* siguiendo las restricciones puestas por la línea aérea respecto
* al número máximo de tiquetes permitidos por reserva.
*
* @pre hay tantos puestos disponbles en el vuelo como número de
*   tiquetes.
* @param numeroVuelo el número del vuelo
* @param numeroTiquetes el número de tiquetes que se quiere reser-
*   var en el vuelo indicado.
**/

public void realizarReserva(String numeroVuelo, int numeroTiquetes){
    ????
}
```

Diseño de casos de prueba

- Input: n
- Valid: 1 <= n <= 10
- Invalid: not (1 <= n <= 10) == (n < 1) or (n > 10)

- Las pruebas exhaustivas son costosas
- Idea: partir el dominio en clases de equivalencia



Pruebas utilizando Clases de Equivalencia

Hipótesis de las pruebas con clases de equivalencia:

- Si un valor/condición de una de partición pasa la prueba, es probable que los demás casos de la misma partición pasen.
- Si un valor/condición de una partición produce un fallo, es probable que los demás casos de la misma partición también fallen

¿Qué son Pruebas Unitarias ?

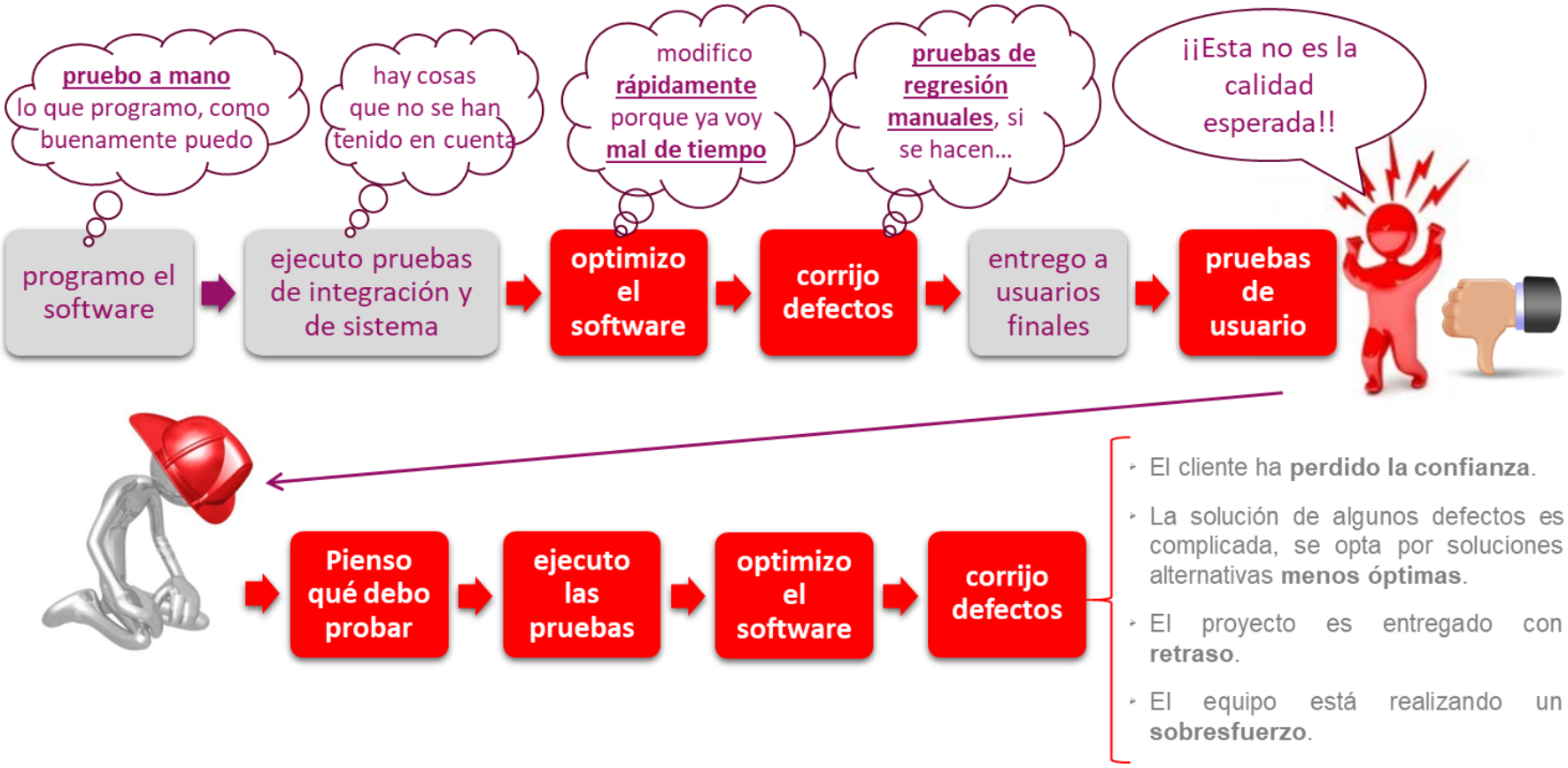
*Una **unidad** es simplemente una porción de código o una funcionalidad que realiza una acción específica, de la cual podemos probar los resultados.*

En el caso de lenguajes como Java y .Net, la unidad es el *método*. Entonces:

*Una **prueba unitaria** es una prueba en forma de código para asegurar que la porción de código o funcionalidad hace lo que debería hacer, independientemente del resto de la aplicación.*

"El objetivo de las **pruebas unitarias** es probar el comportamiento de cada uno de los métodos de la clase de manera aislada."

¿Qué ocurre normalmente?



VENTAJAS

- ✓ Aseguran la calidad del código entregado
- ✓ Fomentan el cambio y la refactorización
- ✓ Reducen los problemas y tiempos dedicados a la integración
- ✓ Documentan el código
- ✓ Fomentan la mejora del diseño de forma emergente

RETOS

- ✓ Nos ayudan a detectar defectos más no garantiza la ausencia de ellos
- ✓ No permiten detectar defectos con un enfoque sistémico
- ✓ Son código por lo cual están sujetas a buenas prácticas de codificación
- ✓ Se requieren mantener y evolucionar continuamente.

"Las pruebas unitarias son aquello que nos permite mantener el código flexible, mantenible y reusable."

(Robert Martin, co-autor manifiesto ágil)

Principio F.I.R.S.T.

Rápida
Fast

Independiente
Isolated

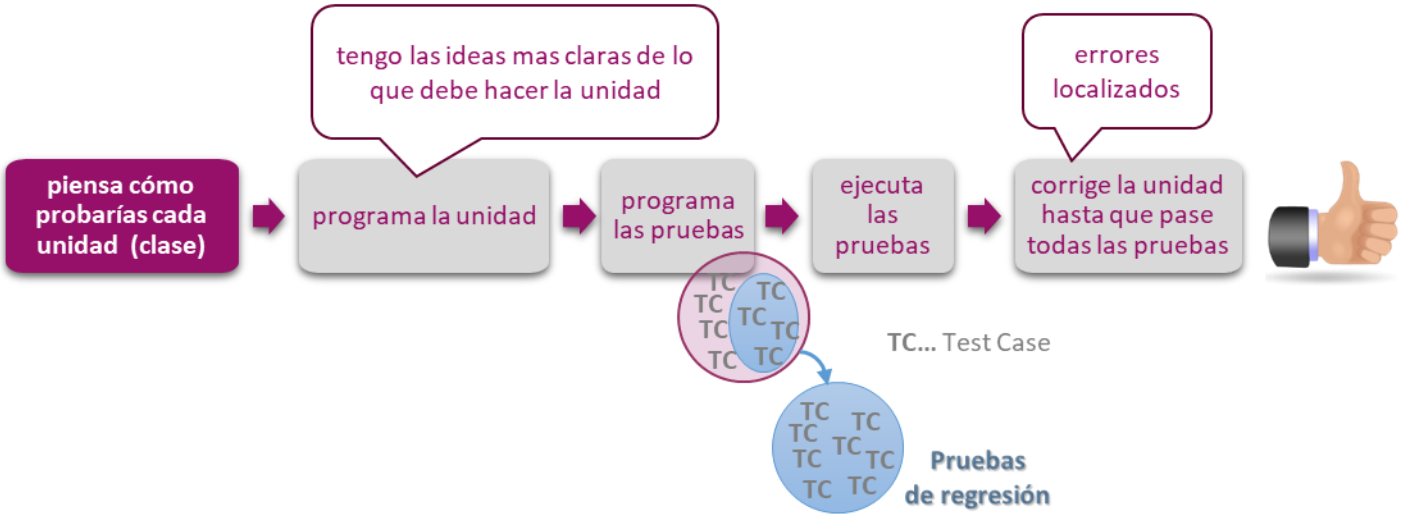
Repetible
Repeatable

Auto-Validación o Validación Automática (Confiable)
Self-checking

Oportuna y Completa (Exhaustiva)
Timely & Thorough

¿Cómo abordar estas pruebas?

Enfoque convencional



*Como parte del proceso de **integración continua**, utiliza las **pruebas unitarias** como parte de las **pruebas de regresión,** que se ejecuten **automáticamente** cada vez que se produzca un cambio.*

Patrón AAA – Arrange, Act, Assert

C#

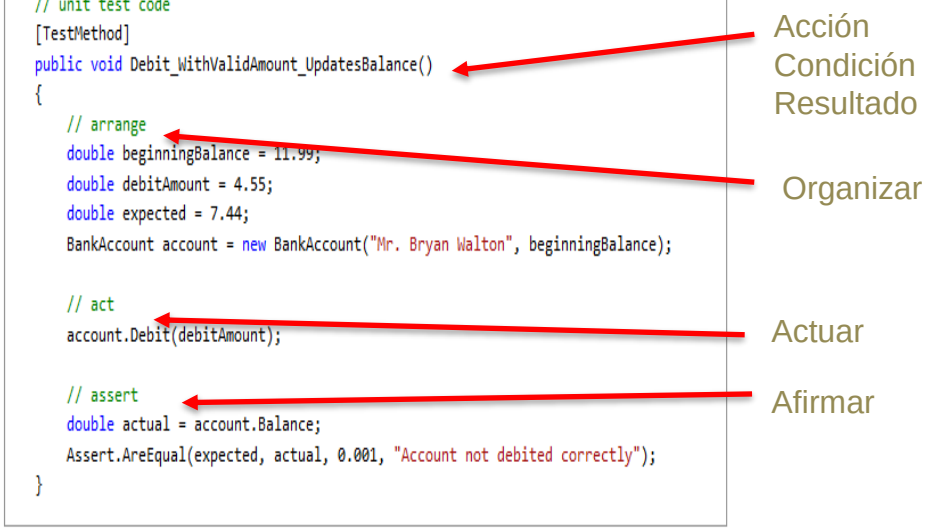
```
// method under test
public void Debit(double amount)
{
    if(amount > m_balance)
    {
        throw new ArgumentOutOfRangeException("amount");
    }
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }
    m_balance += amount;
}
```

C#

```
// unit test code
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
    // arrange
    double beginningBalance = 11.99;
    double debitAmount = 4.55;
    double expected = 7.44;
    BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

    // act
    account.Debit(debitAmount);

    // assert
    double actual = account.Balance;
    Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
}
```



Nombramiento de las Pruebas Unitarias

1. **test[Feature being tested]**
testIsNotAnAdultIfAgeLessThan18
2. **Feature to be tested**
IsNotAnAdultIfAgeLessThan18
3. **[MethodName]_[StateUnderTest]_[ExpectedBehavior]**
isAdult_AgeLessThan18_False
4. **[MethodName]_[ExpectedBehavior]_[StateUnderTest]**
isAdult_False_AgeLessThan18

Nombramiento de las Pruebas Unitarias

5. **Should_[ExpectedBehavior]_When_[StateUnderTest]**

Should_ThrowException_When_AgeLessThan18

6. **When_[StateUnderTest]_Expect_[ExpectedBehavior]**

When_AgeLessThan18_Expect_isAdultAsFalse

7. **Given_Preconditions_When_StateUnderTest_Then_ExpectedBehavior**

Given_UserIsAuthenticated_When_InvalidAccountNumberIsUsedToWithdrawMoney_Then_TransactionsWillFail

Cobertura de las Pruebas Unitarias

```
0 referencias
public class Operations
{
    /// <summary> Sums the specified num1 + num2.
    0 referencias | 0 excepciones
    public decimal Sum(decimal num1, decimal num2)
    {
        return num1 + num2;
    }
    /// <summary> Subtractions the specified num1 - num2.
    0 referencias | 0 excepciones
    public decimal Subtraction(decimal num1, decimal num2)
    {
        return num1 - num2;
    }
    /// <summary> Multiplications the specified num1 * num2.
    0 referencias | 0 excepciones
    public decimal Multiplication(decimal num1, decimal num2)
    {
        return num1 * num2;
    }
    /// <summary> Divisions the specified num1/num2.
    0 referencias | 0 excepciones
    public decimal Division(decimal num1, decimal num2)
    {
        if (num2 == 0)
        {
            throw new DivideByZeroException();
        }
        return num1 / num2;
    }
}
```

```
/// </summary>
[TestClass]
0 referencias
public class UnitTestOperations
{
    /// <summary> Tests the sum.
    [TestMethod]
    0 referencias | 0 excepciones
    public void TestSum()
    {
        //Arrange
        decimal num1 = 5; decimal num2 = 7; decimal actualResult; decimal expectedResult = 12;
        Operations oper = new Operations();
        //Act
        actualResult = oper.Sum(num1, num2);
        //Assert
        Assert.AreEqual(expectedResult, actualResult);
    }
    /// <summary> Tests the subtraction.
    [TestMethod]
    0 referencias | 0 excepciones
    public void TestSubtraction()
    {
        //Arrange
        decimal num1 = 7; decimal num2 = 5; decimal expectedResult = 2; decimal actualResult;
        Operations oper = new Operations();
        //Act
        actualResult = oper.Subtraction(num1, num2);
        //Assert
        Assert.AreEqual(expectedResult, actualResult);
    }

}
```

¿Cuál es el cubrimiento en esta prueba?

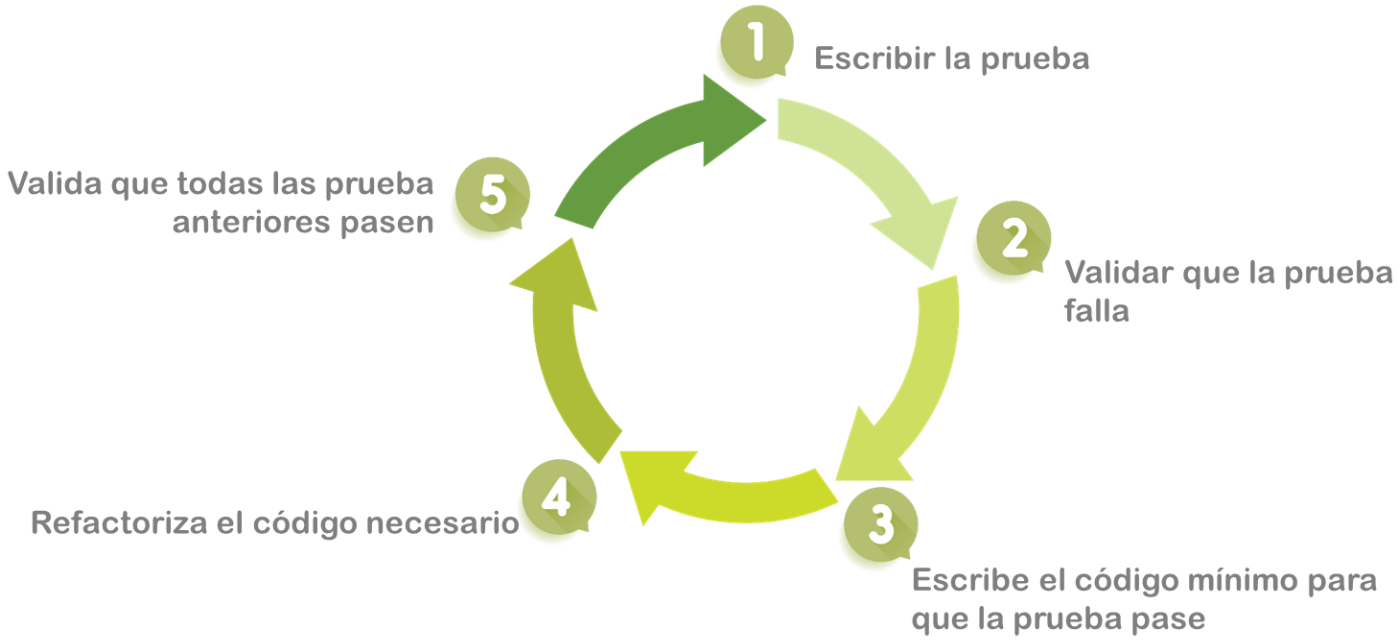
Errores de Principiante

- El nombre del test no es suficientemente descriptivo
- Desconocimiento del SUT o no se sabe que es lo que hace
- Un mismo método de test está haciendo múltiples afirmaciones
- Los test unitarios no están separados de los de integración
- Se nos olvida refactorizar
- El código de los tests debe estar tan limpio como el del SUT
- No eliminamos código muerto

Antipatrones de las Pruebas Unitarias

- **El mentiroso**
- **SetUp Excesivo**
- **Test Dios**
- **El que siempre funciona**
- **El libre Albedrío**
- **Test con comentarios**

TDD - Test Driven Development



Para el laboratorio

- Pruebas de frontera y clases de equivalencia
  - Design Techniques for Enhancing Unit Tests - DZone
- Revisar patrones de comportamiento
  - Comando
- Revisar patrones estructurales
  - Adaptador
  - Fachada