TEST DRIVEN DEVELOPMENT Y UNIT TESTING

TEST DRIVEN DEVELOPMENT

TEST DRIVEN DEVELOPMENT

- Práctica de programación que consiste en escribir primero las pruebas (generalmente unitarias), después escribir el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito.
- Todo el código debe ser probado y refactorizado continuamente.

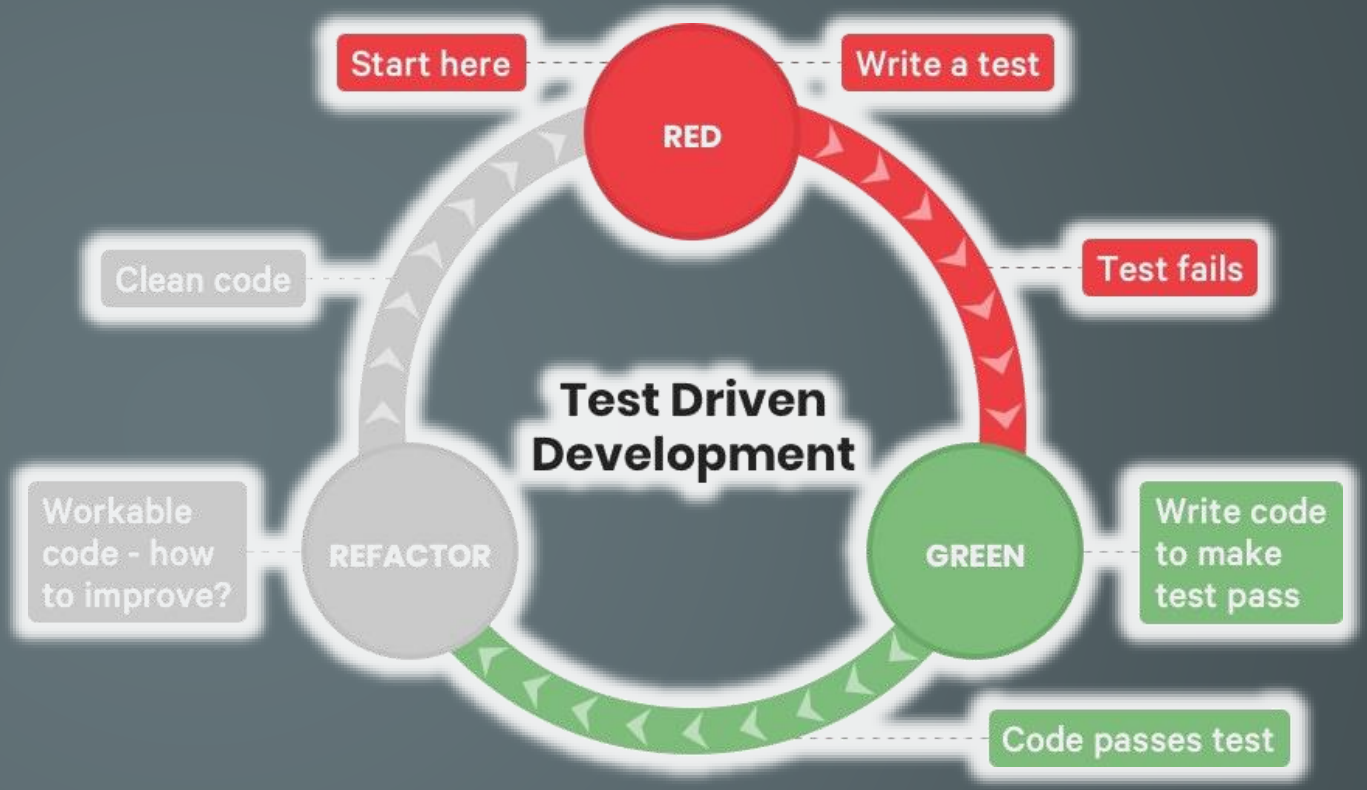
PRINCIPIOS DE TEST DRIVEN DEVELOPMENT

- Escribir las pruebas, luego el código.
- Escribir el código solo cuando la prueba falle.
- Escribir el código para que la prueba pase.

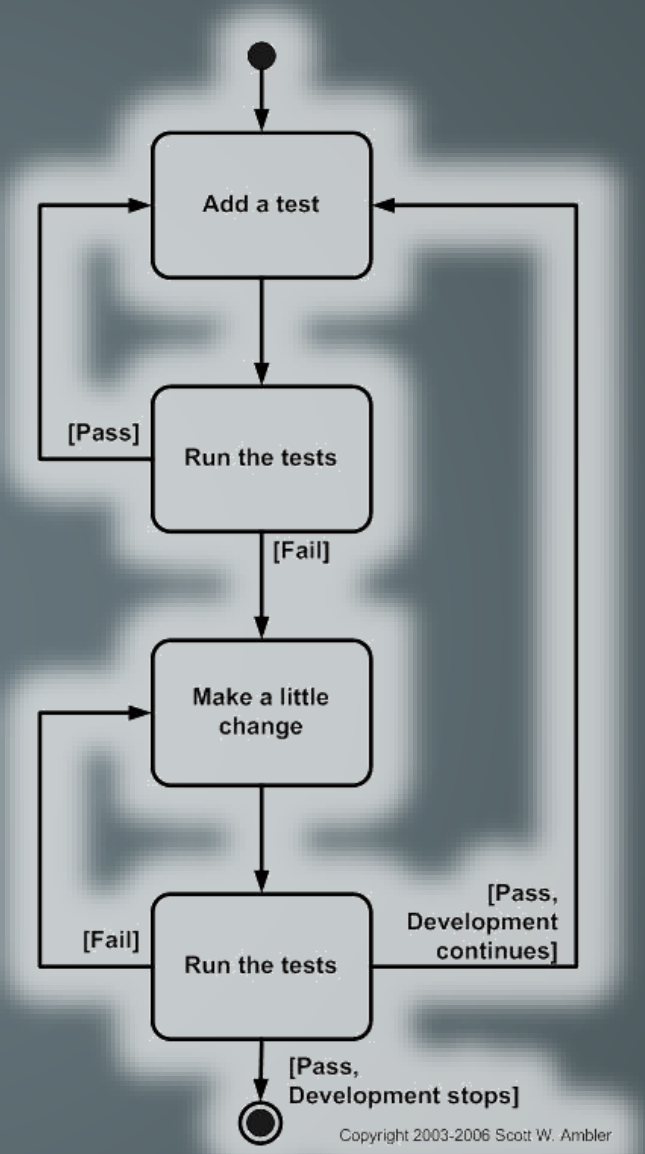
BENEFICIOS DE TEST DRIVEN DEVELOPMENT

- Confianza en los cambios de código.
- Código de mayor calidad.
- Detección temprana de errores.

TEST DRIVEN DEVELOPMENT



TEST DRIVEN DEVELOPMENT



UNIT TESTING

UNIT TEST

- Puede definirse como un mecanismo de comprobación del funcionamiento de las unidades de menor tamaño de un programa o aplicación en específico.
- Las pruebas unitarias de software evitan la escalada de errores en el código al identificarlas de manera temprana.

AAA

- Arrange
- Act
- Assert

ARRANGE

- Consiste en el código requerido para preparar una prueba en específico.
- Inicializa los objetos y establece los valores de los datos que vamos a utilizar en el Test que lo contiene.
- Se crean mocks en caso de usar alguno.

ACT

- Realiza la llamada al método a probar con los parámetros preparados para tal fin.

ASSERT

- Comprueba que el método de pruebas ejecutado se comporta tal y como teníamos previsto que lo hiciera.

AAA

```
[Fact]
public void Person_WithConstructorValues_ShouldBeNameAsExpected()
{
    // Arrange
    var person = new Person("Jorge", 18);
    // Act
    var personName = person.GetName();
    // Assert
    personName.Should().Equals("Jorge");
}
```

MOCK

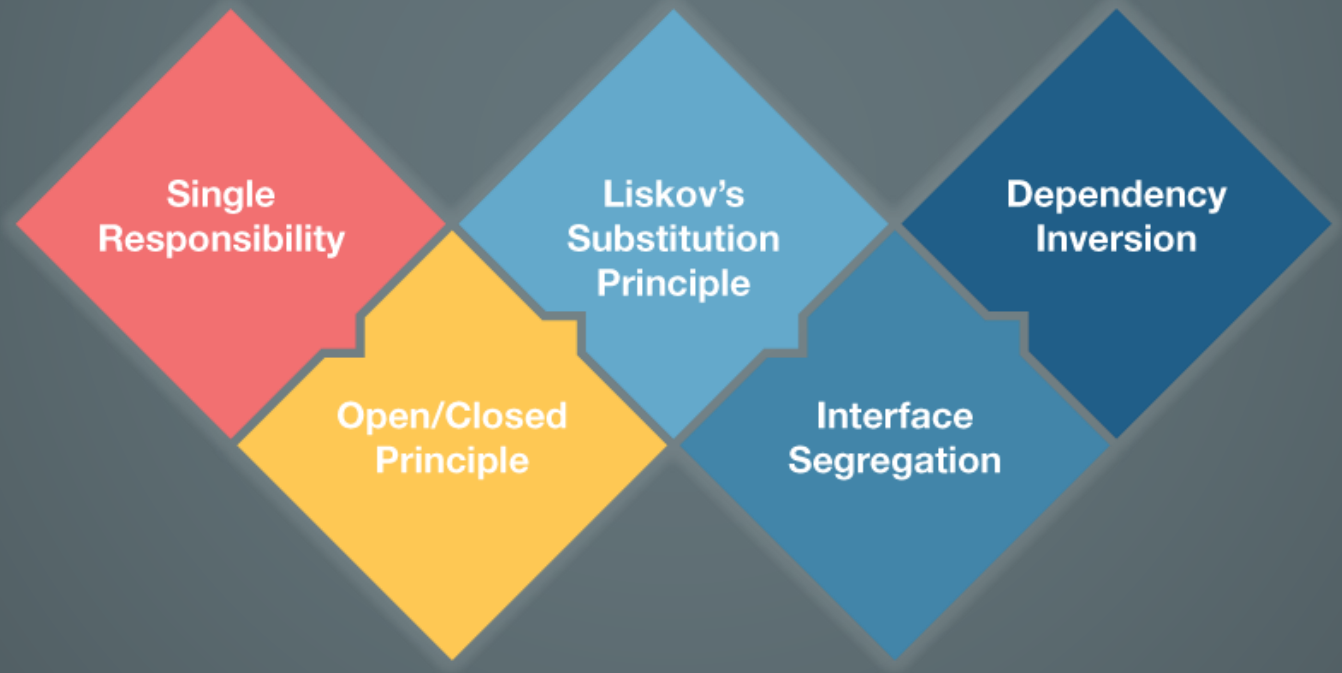
- Son objetos preprogramados con expectativas que conforman la especificación de lo que se espera que reciban las llamadas.
- Objetos que se usan para probar que se realizan correctamente llamadas a otros métodos, por ejemplo, a una web API, por lo que se utilizan para verificar el comportamiento de los objetos.

PRINCIPIOS SOLID

PRINCIPIOS SOLID

- Conjunto de principios aplicables a la Programación Orientada a Objetos que plantean que los diseños deben ser más comprensibles, flexibles, escalables y resistentes a los cambios.
- Los Principios Solid indican cómo organizar funciones y estructuras de datos en componentes y cómo dichos componentes deben estar interconectados.

S.O.L.I.D.


Single Responsibility
Liskov's Substitution Principle
Dependency Inversion
Open/Closed Principle
Interface Segregation

SINGLE RESPONSIBILITY

- Establece que una clase, componente o microservicio debe ser responsable de una sola cosa (desacoplamiento). Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

SINGLE RESPONSIBILITY

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }

    void guardarCocheDB(Coche coche){ ... }
}
```

SINGLE RESPONSIBILITY

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}

class CocheDB{
    void guardarCocheDB(Coche coche){ ... }
    void eliminarCocheDB(Coche coche){ ... }
}
```

OPEN/CLOSED PRINCIPLE

- Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.

OPEN/CLOSED PRINCIPLE

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}
```

OPEN/CLOSED PRINCIPLE

```
public static void main(String[] args) {
    Coche[] arrayCoches = {
            new Coche("Renault"),
            new Coche("Audi")
    };
    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
    }
}
```

OPEN/CLOSED PRINCIPLE

```
Coche[] arrayCoches = {
    new Coche("Renault"),
    new Coche("Audi"),
    new Coche("Mercedes")
};
```

OPEN/CLOSED PRINCIPLE

```
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
        if(coche.marca.equals("Mercedes")) System.out.println(27000);
    }
}
```

OPEN/CLOSED PRINCIPLE

```
abstract class Coche {
    // ...
    abstract int precioMedioCoche();
}

class Renault extends Coche {
    @Override
    int precioMedioCoche() { return 18000; }
}

class Audi extends Coche {
    @Override
    int precioMedioCoche() { return 25000; }
}

class Mercedes extends Coche {
    @Override
    int precioMedioCoche() { return 27000; }
}

public static void main(String[] args) {

    Coche[] arrayCoches = {
            new Renault(),
            new Audi(),
            new Mercedes()
    };

    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        System.out.println(coche.precioMedioCoche());
    }
}
```

LISKOV'S SUBSTITUTION PRINCIPLE

- Establece que cada subclase o clase derivada debe ser sustituible por su clase base o clase principal.
- Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

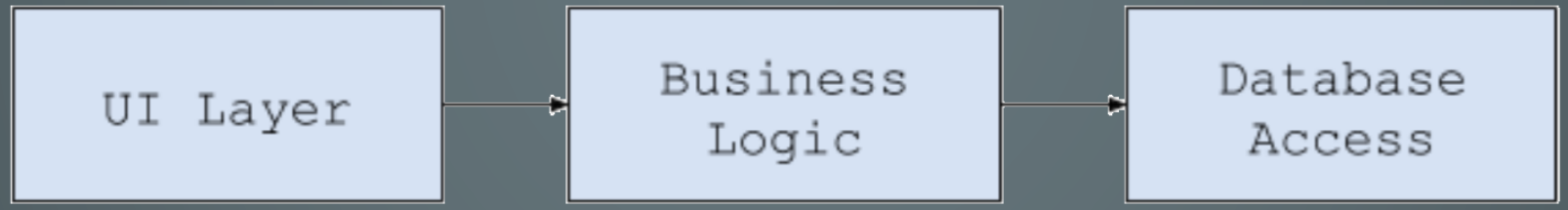
INTERFACE SEGREGATION

- Establece que ninguna clase debería depender de métodos que no usa.
- Todos los métodos definidos en una interfaz deben ser definidos en las clases que la implementen.

DEPENDENCY INVERSION

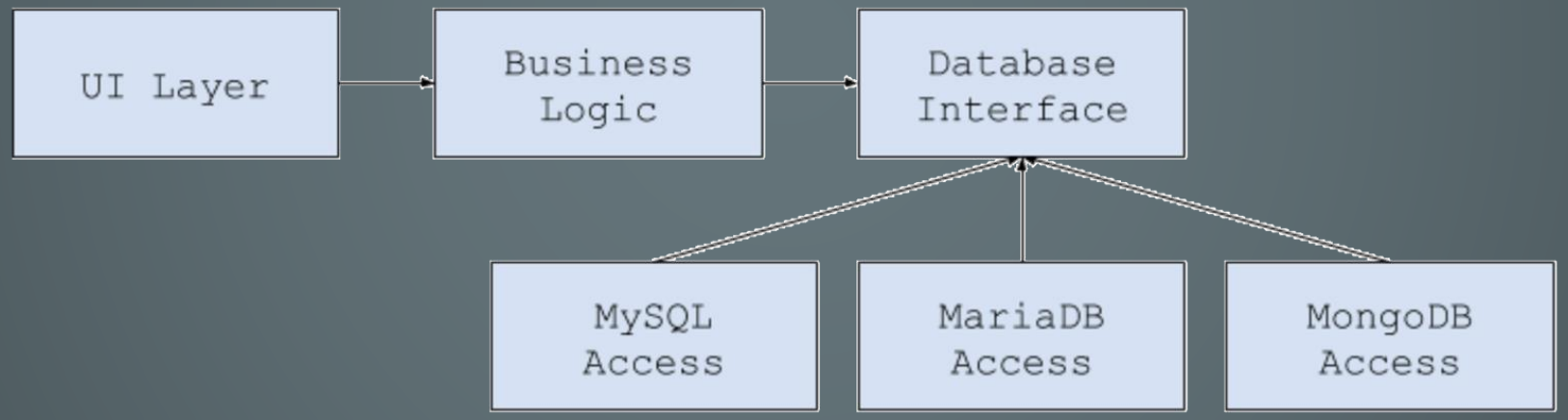
- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles (implementaciones concretas). Las detalles deberían depender de abstracciones.

DEPENDENCY INVERSION



DEPENDENCY INVERSION

DEPENDENCY INVERSION



Single Resposibility Principle

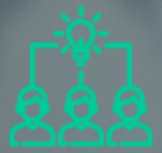
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle

A software module (it can be a class or method) should be open for extension but closed for modification.

Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.



Interface Segregation Principle

Clients should not be forced to depend upon the interfaces that they do not use.

Dependency Inversion Principle

Program to an interface, not to an implementation.

EJEMPLO PRÁCTICO

LINKS DE UTILIDAD

- Principios SOLID: https://www.enmilocalfunciona.io/principios-solid/
- xUnit Assertions: https://textbooks.cs.ksu.edu/cis400/1-object-orientation/04-testing/05-xunit-assertions/
- Documentación official de Moq: https://github.com/Moq/moq4/wiki/Quickstart