# Dobles de prueba

autentia

## ¿Qué es?

A veces resulta difícil probar el "system under test" SUT porque depende de otros componentes que no pueden utilizarse en el entorno de la prueba, o puede interesarnos no utilizar las implementaciones reales.
En estos casos sustituimos los componentes reales por otros que proporcionan un comportamiento simulado del componente real y respetan el mismo contrato. Estos componentes simulados se denominan dobles de prueba, o “mocks” de forma generalizada.
El uso de dobles de prueba es muy utilizado en el ámbito de metodologías ágiles, dentro de XP como parte de las prácticas de testing, TDD y automatización de pruebas.

### ¿Cómo?

- Para la prueba que queremos hacer, se **sustituye la dependencia real** a usar por el SUT por nuestro doble de prueba.
- En la primera fase del test, fase de "setup", donde se configura el **contexto de la prueba** necesario para que el doble muestre el comportamiento esperado, así como todo lo necesario para poder observar el resultado real.
- Hay que tener en cuenta que nuestro doble de prueba sólo debe dar una implementación de la funcionalidad usada y esperada por el SUT en dicha prueba y no todo el contrato de la dependencia real.
- Podemos tener diferentes dobles de prueba de la misma dependencia para distintas pruebas.

### ¿Cuándo?

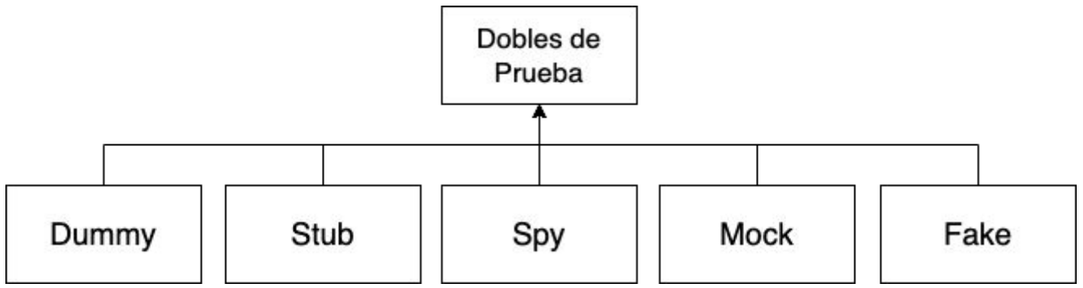
- Necesitamos **aislar** el SUT de sus dependencias para probarlo de forma independiente.
- Cuando tenemos **dependencias no disponibles**, permite desarrollar funcionalidades a pesar de no tener implementaciones reales de éstas.
- Cuando el comportamiento del sistema sometido a prueba (SUT) incluye **acciones que no pueden observarse** a través de la API pública del SUT, pero que son vistas o experimentadas por otros sistemas o componentes de la aplicación.
- Forzar respuestas de las dependencias para **probar distintos caminos** (ej. situaciones de error).
- Cumplir con los principios **FIRST** de las pruebas.

# Dobles de prueba: Tipos

autentia

## ¿Qué es?

Para realizar pruebas sobre el “system under test” SUT nos puede interesar sustituir los componentes de los que depende por **implementaciones específicas para la prueba** que proporcionan un comportamiento simulado del componente sustituido. A estas implementaciones simuladas se las conoce como dobles de prueba. Dependiendo de la necesidad existen varios tipos de dobles de prueba.




### Frameworks

Existe un gran número de frameworks que implementan dobles. Entre ellos podemos destacar:

- **Mockito**: Framework de código abierto bajo licencia MIT que permite la creación de dobles de pruebas para la automatización de unitarias, TDD y BDD.
- **EasyMock**: Framework de código abierto bajo licencia Apache que para la creación de dobles de pruebas en la automatización de unitarias, TDD y BDD.
- **Mock Service Worker**: es un framework que intercepta las peticiones del frontend y simula las respuestas del backend haciendo uso de la API de Service Workers.

### Tipos de dobles

- **Dummy**: Satisface dependencias no utilizadas en el test o rellenar parámetros. Cubren la necesidad de proporcionar una implementación vacía devolviendo null o lanzando una excepción
- **Stub**: Satisface dependencias utilizadas que requieran de un determinado comportamiento o datos. Proporciona respuestas predefinidas programadas para la prueba.
- **Spy**: Su objetivo principal es proporcionar información sobre las llamadas recibidas para luego poder inspeccionarlas. De ahí su nombre “espía”.
- **Mock**: Puede comportarse como un Stub o un Spy, satisfaciendo dependencias con un determinado comportamiento, pero la diferencia está en que es el propio Mock el que tiene conocimiento sobre el comportamiento esperado y es el responsable de su verificación.
- **Fake**: Satisface dependencias utilizadas que requieran de un determinado comportamiento real. Al implementar métodos con cierta lógica su complejidad es muy variada.

# Dobles de prueba: Dummy

autentia

## ¿En qué consiste?

Dummy es el tipo de doble más simple que existe. Su función principal es la de satisfacer **dependencias no utilizadas** en el test o **rellenar parámetros**. Proporcionar una implementación vacía devolviendo **null** o lanzando una **excepción**.

### Ejemplo

En el ejemplo la clase DummyGreetingsProvider es un Dummy que contiene el método getHelloWorld(). Este método no espera ser llamado porque en el caso de que lo haga lanza una excepción.

```
class Service {
    public Service(GreetingsProvider greetingsProvider){...}
    public boolean isAService() {
        return true;
    }
}
class DummyGreetingsProvider implements GreetingsProvider {
    @Override
    public String greetings(String name) {
        throw new RuntimeException("Not expected to be called");
    }
}
class ServiceTest{
    @Test
    public void dummyTest() {
        GreetingsProvider dummy = new DummyGreetingsProvider();
        Service service = new Service(dummy);
        assertTrue(service.isAService());
    }
}
```

# ¿En qué consiste?

Es un tipo de doble que  tiene como función satisfacer **dependencias utilizadas** que requieran de un **determinado comportamiento**. La implementación es fija **siempre devuelven los mismos valores**, estando éstos predefinidos en función de la entrada.

## Ejemplo

En el ejemplo, la clase StubGreetingsProvider es un Stub que contiene el método getHelloWorld(), este método devolverá siempre el string “Hello World” cada vez que sea llamado.

```
class Service {
    public Service(GreetingsProvider greetingsProvider){...}
    public String sayHello() {
        return greetingsProvider.greetings();
    }
}
class StubGreetingsProvider implements GreetingsProvider {
    @Override
    public String greetings() {
        return "Hello World";
    }
}
class ServiceTest{
    @Test
    public void stubTest() {
        GreetingsProvider stub = new StubGreetingsProvider();
        Service service = new Service(stub);
        assertEquals("Hello World", service.sayHello());
    }
}
```

# Dobles de prueba: Spy

autentia

## ¿En qué consiste?

Satisface dependencias utilizadas que requieran de un determinado comportamiento, al igual que los Stub, pudiendo proporcionar una implementación que devuelva resultados predefinidos. Aunque su principal objetivo es **proporcionar información sobre las llamadas que recibe**, como el número de veces que se le ha llamado o los argumentos de la llamada. Hay que tener cuidado con este doble ya que acopla con un mayor grado el test con una implementación concreta del SUT.

### Ejemplo

En este ejemplo, hay una clase Service con un método doSomething() y este depende de una implementación de Logger.

SpyLogger proporciona una implementación de Logger para la prueba, aportando el método getCount(), este método devolverá un entero con el número de veces que se ha llamado al método Logger().

```
class SpyLogger implements Logger{
    private int count = 0;
    public void log(String message) {
        count++;
    }
    public int getCount() {
        return count;
    }
}
class Service {
    public Service(Logger logger){...}
    public void doSomething() {
        this.logger.log("message");
    }
}
class ExampleTest{
    @Test
    void spyTest() {
        SpyLogger spyLogger = new SpyLogger();
        Service service = new Service(spyLogger);
        service.doSomething();
        assertEquals(1, spyLogger.getCount());
    }
}
```

# ¿En qué consiste?

Este tipo de doble  tiene como característica principal que es él mismo es que **es el responsable de conocer el comportamiento esperado y de la verificación de este comportamiento.** Además, al igual que el Stub y el Spy, puede dar una implementación que satisface las dependencias y proporciona resultados predefinidos..

## Ejemplo

En este ejemplo, hay una clase Service con un método doSomething() y este depende de una implementación de Logger.

MockLogger proporciona una implementación de Logger para la prueba, llevando internamente un contador de las veces y el mensaje con el que ha sido llamado. Además proporciona el método veify() para comprobar que se ha llamado el número de veces y mensaje esperado.

```
class MockLogger implements Logger{
    private int count = 0;
    private String message;
    public void log(String message) {
        this.message = message;
        count++;
    }
    public void verify() {
        assertEquals(1, this.count);
        assertEquals("message", message);
    }
}
class Service {
    public Service(Logger logger){...}
    public void doSomething() {
        this.logger.log("message");
    }
}
class ExampleTest{
    @Test
    void mockTest() {
        MockLogger mockLogger = new MockLogger();
        Service service = new Service(mockLogger);
        service.doSomething();
        mockLogger.verify();
    }
}
```

# ¿En qué consiste?

Se utilizan para satisfacer **dependencias utilizadas** que requieran de un **determinado comportamiento real**, pudiendo además entrenarse con una **implementación dinámica** en función de los argumentos recibidos. Implementa la misma interfaz que una dependencia real del SUT, pero este toma atajos para ser más simple que el objeto que está “imitando”. Al implementar métodos con cierta lógica su complejidad es muy variada.

## Ejemplo

En el ejemplo, la clase FakeGreetingsProvider es un Fake que contiene el método getHelloWorld(String), este método devolverá un saludo personalizado en función del nombre recibido..

```
class Service {
    public Service(GreetingsProvider greetingsProvider){...}
    public String sayHello(String name) {
        return greetingsProvider.greetings(name);
    }
}
class FakeGreetingsProvider implements GreetingsProvider {
    @Override
    public String greetings(String name) {
        return "Hello "+name;
    }
}
class ServiceTest{
    @Test
    public void fakeTest() {
        GreetingsProvider fake = new FakeGreetingsProvider();
        Service service = new Service(fake);
        assertEquals("Hello John", service.sayHello("John"));
    }
}
```