unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 2

Construcción de software

Testing y scrips

Índice

Testing y scripts

En esta segunda unidad, profundizaremos en la ingeniería del software con el objetivo de facilitar la creación de programas correctos y funcionales: no basta con saber programar sino que se debe poder hacer código legible, mantenible y libre de errores.

- Conceptos de testing:
  - Tipos de test: unitarios, integración, funcionales…
  - Técnicas de testing: mocks, stubs,doubles.
- Código más fácilmente testable:
  - Clean code.
  - Refactoring.
  - Inyección de dependencias.
- Scripts en DevOps:
  - Concepto de script.
  - Utilidades y aplicaciones.
  - Diferentes lenguajes.

Objetivos

Al terminar esta sección:

- Se conocerá en profundidad la forma más común de minimizar los errores durante el desarrollo: las pruebas de software.
- Se conocerán principios de código limpio y mantenible, que facilitan la creación de software y de sus pruebas.
- Se comprenderá la utilidad de conocer y dominar lenguajes de scripting para complementar nuestra tarea como DevOps.

Testing

La segunda parte de esta lección estará centrada en cómo dotar al código de validaciones (test) que garanticen que:

- Hace lo que tiene que hacer.
- Lo seguirá haciendo, a medida que evolucione.

El uso de **técnicas de testing** en el código de nuestras aplicaciones y sistemas, en las condiciones adecuadas (rápidos, consistentes y fiables) es lo que posibilita una de las características más deseables de toda la cultura DevOps: **la velocidad.**

La **velocidad de un equipo** no es exclusivamente la cantidad de nuevas funcionalidades que se pueden entregar en cada iteración o cada semana: mide también que se entreguen de forma **correcta y fiable**, reduciendo el mantenimiento de los entregables al reducir la cantidad de errores que se presentan después de darlos por listos.

Una manera de conseguir esto es añadir todo tipo de pruebas al código que se genera, de modo que se 'pruebe solo' sin necesidad de manualidades, tan solo 'pulsando un botón': ya sea un botón literal en el IDE que empleemos, cada vez que se guarda a disco, ejecutando un script, de forma automática al subir el código al repositorio…

Tener disponible feedback al instante de que hemos roto algo que estaba funcionando y saber cuándo lo hemos arreglado, es lo que se conoce como **ciclos rápidos de desarrollo** (*fast development* feedback*).*

Los ciclos rápidos de desarrollo garantizan que se tendrá feedback rápidamente de posibles errores, lo que permite **arreglarlos y seguir iterando,** hasta llegar a resolver el problema que nos habíamos propuesto.

¿Qué es un test?

Vale la pena hacerse está pregunta, pues puede no ser inmediatamente obvio qué tipo de pieza software es un test.

Como veremos a continuación, hay muchos y muy diferentes tipos de test, por lo que no existe una única respuesta a esa pregunta.

En general, se busca escribir los test en el mismo lenguaje en el que está la aplicación, lo que permite:

- Aprovechar el conocimiento y pericia del equipo en el lenguaje de la aplicación.
- Incluirlos de forma natural en el ciclo de vida, al usar herramientas como npm, gradle, make…

Dependiendo del lenguaje, habrá diferentes formas de probar: mediante dependencias externas, embebidas en el mismo framework o en el lenguaje, quizás exista una librería recomendada que mejore el caso base… Cada lenguaje es un mundo y, a menudo, lo fácil o difícil que sea escribir y ejecutar test es un motivo de peso para inclinar la balanza hacia una tecnología determinada.

Pero como veremos, no siempre será posible que coincidan: es usual utilizar Docker para levantar entornos de pruebas o servicios de los que depende nuestra aplicación y con ellos probar integraciones y simular entornos end-to-end.

También es un caso común que los diferentes proveedores de servicio cloud o plataforma de despliegue, como Kubernetes, integren test de validación de servicio de forma automática, cuya configuración dependerá de la herramienta que se use.

Vamos a tratar de ilustrar todo esto con ejemplos:

Probando SelectionSort

En la lección anterior revisamos el algoritmo de ordenación SelectionSort y para ilustrarlo y poder estudiarlo se dieron dos posibles implementaciones del mismo. Una manera obvia de probarlo puede ser crear un programa con un main, pasarle el input que queramos, a ver qué sale:

```
// Java version
class SelectionSort {
  public static int[] selectionSort(int[] lista) {
    ...
  }

  public static void main(String[] args) {
   int[] input = new int[]{7, 1, 3, 2, 5};
   int[] expectedOutput = new int[]{1, 2, 3, 5, 7};
    if (!Arrays.equals(selectionSort(input), expectedOutput)) {
       throw new RuntimeException("Error: El algoritmo no funciona");
    }
  }
}


# Python version
if __name__ == '__main__':
  assert selection_sort([7, 1, 3, 2, 5]) == [1, 2, 3, 5, 7])
```

Aquí, introducimos un concepto muy común en el testing, aunque se puede emplear en otros ámbitos también: las **aserciones**, pequeños checks que se hacen en tiempo de ejecución y cuya evaluación negativa provoca un error no recuperable.

Si bien son de mucha utilidad en test unitarios porque son suficientemente claros en su propósito, su uso en el código de aplicación (por ejemplo, para proteger una inserción en base de datos de valores nocivos) no cumple con estándares de código limpio y mantenible:

- No se deben usar para validar parámetros de usuario, es mucho más claro usar validaciones ad-hoc que arrojen errores más significativos que una *AssertionError* genérica.
- No deben contener lógica: en todo caso deben verificar *a posteriori* que algo se ha cumplido o proteger una sección crítica (para lo cual, sigue siendo mejor una validación adecuada).

Si bien, con esto podemos probar que nuestro código va funcionando, vemos que:

1. (Sobre todo en el caso de Java) tenemos una serie de manualidades que no van a hacer más que crecer, con lo que tendremos dos problemas: organizar el código de pruebas y solucionar el problema original.
2. Una vez 'terminemos' nuestra labor de implementar un algoritmo o una aplicación, ¿qué hacemos con todo ese código de prueba? ¿Lo podríamos aprovechar? Parece interesante que se pueda ejecutar cuando queramos, por si algún día tenemos que cambiar algo.
3. Sería también interesante que se pudiesen ejecutar **con un clic o un simple comando**, de modo que podamos incorporarlos a nuestro flujo de trabajo o al sistema de integración continua.

Por estas y muchas más razones es por lo que existen los *frameworks de testing.*

Frameworks de testing

Un framework (en castellano se podría traducir por **marco de trabajo**) se define como un conjunto de normas o principios que, cuando se siguen correctamente, nos llevan a conseguir un objetivo de una manera sistemática, repetible.

En el mundo de la ingeniería software nos habremos encontrado ya con multitud de frameworks:

- Spring es un framework para el diseño de aplicaciones web en la JVM.
- Flask es un framework Python para la creación de servicios web.
- Hibernate es un framework para estandarizar el acceso a bases de datos desde una aplicación Java.

Cuando nos referimos al testing, un framework nos ayudará a escribir los test de una manera concreta, para que puedan ser después **descubiertos** y **ejecutados** de un modo sencillo.

JUnit

El framework de testing más potente y famoso para Java (y en general todos los lenguajes de la JVM) es JUnit 5.

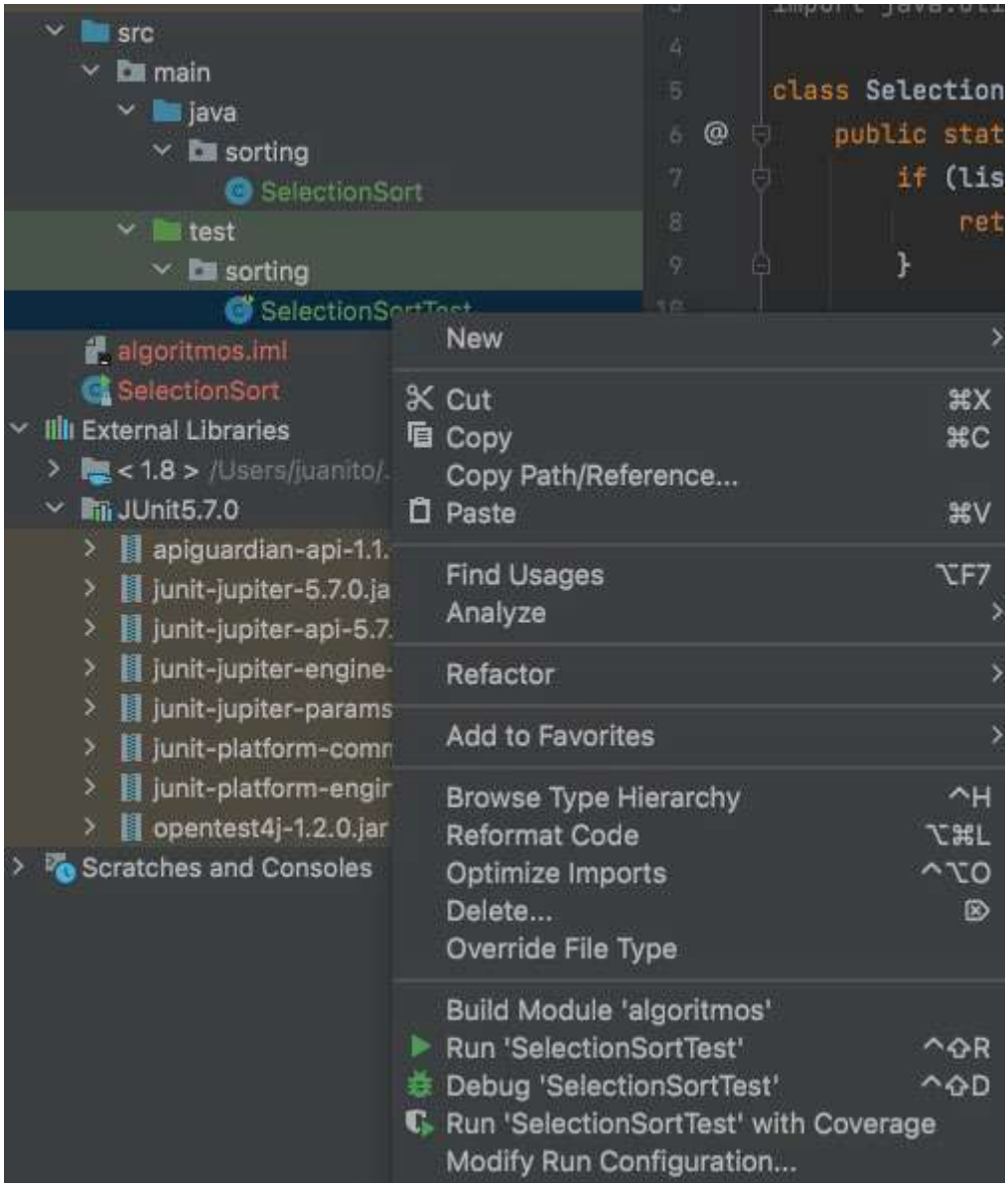
Un ejemplo ilustrará mejor por qué:

```
/*
 * Copyright 2015-2021 the original author or authors.
 *
 * All rights reserved. This program and the accompanying materials are
 * made available under the terms of the Eclipse Public License v2.0 which
 * accompanies this distribution and is available at
 *
 * http://www.eclipse.org/legal/epl-v20.html
 */

package com.example.project;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class CalculatorTests {

	@Test
	@DisplayName("1 + 1 = 2")
	void addsTwoNumbers() {
		Calculator calculator = new Calculator();
		assertEquals(2, calculator.add(1, 1), "1 + 1 should equal 2");
	}

	@ParameterizedTest(name = "{0} + {1} = {2}")
	@CsvSource({
			"0,    1,   1",
			"1,    2,   3",
			"49,  51, 100",
			"1,  100, 101"
	})
	void add(int first, int second, int expectedResult) {
		Calculator calculator = new Calculator();
		assertEquals(expectedResult, calculator.add(first, second),
				() -> first + " + " + second + " should equal " + expectedResult);
	}
}
```

Fuente: Github.com

JUnit usa **anotaciones** para marcar los métodos de test de muy diferentes maneras, de modo que sencillas piezas (los métodos son fáciles de entender) se pueden componer potentísimos test unitarios.

Dichos test residirán en una carpeta test dentro de 'src/main', por lo que pueden importar los paquetes de nuestra aplicación para probarlos:



Pie: IntelliJ IDEA tiene integración nativa con JUnit.

Para poder **ejecutar** los test, es conveniente leerse la documentación: si bien los IDEs más habituales (VSCode, Eclipse, IntelliJ…) tienen integración con JUnit y muchos otros frameworks, necesitaremos importar sus dependencias de alguna manera.

Para ello, la mejor manera es usar un gestor de dependencias como *Maven* o *Gradle*, de modo que lanzar los test sea algo tan sencillo como:

```
$ gradle test

# Es común asociar el testing al proceso de construcción:
# esto ejecutará los tests primero, y si pasan construirá el paquete
$ mvn package
# En ocasiones nos puede interesar construir sin pasar tests.
$ mvn package -DskipTests=true
```

Python unittest

Python viene con su propia librería de testing incluida, dentro del paquete unittest. Si consultamos su documentación oficial, veremos que está inspirada en JUnit, por lo que comparten:

- Sintaxis basada en anotaciones.
- Autodescubrimiento de test.
- Organización en test suites.

La ventaja es que, al venir incluida en el propio lenguaje, no es preciso instalar ninguna dependencia adicional y podemos añadir test desde el minuto 0 en nuestras aplicaciones.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Este script es ejecutable como cualquier otro script de Python, aunque podemos usar la potente capacidad de *Autodiscovery* para organizar mejor nuestro código, ubicando todos los ficheros de test en **módulos** dentro de una carpeta, comúnmente llamada *tests*.

Así, si el script anterior está en el fichero 'tests/test_strings.py', lo podemos ejecutar de las siguientes maneras:

```
$ python tests/test_string.py
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK

$  python -m unittest
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK

$  python -m unittest -v
test_isupper (tests.test_string.TestStringMethods) ... ok
test_split (tests.test_string.TestStringMethods) ... ok
test_upper (tests.test_string.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Las dos últimas usan autodiscovery, por lo que a medida que se añadan nuevos test dentro de esa carpeta y con la nomenclatura y formato esperados, se ejecutarán junto con este.

py.test

Por último, introducimos una librería de testing para Python que resulta muy sencilla de aprender y es 100% compatible con JUnit (por lo que podremos mezclar código de test sin problema) y añade una manera más sencilla y ligera de crear nuestros ficheros de test. Veamos un ejemplo de py.test:

```
from sort.sort import selection_sort


def test_selection_sort():
    assert selection_sort([]) == []
    assert selection_sort([1]) == [1]
    assert selection_sort([1, 2]) == [1, 2]
    assert selection_sort([3, 2, 1]) == [1, 2, 3]
    assert selection_sort([1, 4, 3, 2, 1]) == [1, 1, 2, 3, 4]
    assert selection_sort([1, 2, 3, 2, 1, 0]) == [0, 1, 1, 2, 2, 3]
    assert selection_sort(["abc", "def", "a"]) == ["a", "abc", "def"]
```

Ejecutar este test, esté donde esté, es tan sencillo como instalar py.test e invocarlo:

```
$ pip install pytest
<output de pip>
$ pytest
============================================================================= test session starts
=============================================================================
platform darwin -- Python 3.8.2, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /Users/juanito/Stuff/juan.arias/edix_devops/tema2/algoritmos/Python
collected 4 items

tests/test_sort.py .                                                                [ 25%]
tests/test_string.py ...                                                            [100%]

============================================================================== 4 passed in
0.07s ==============================================================================
```

¿La diferencia? Lo primero, la sintaxis, mucho menos pesada, libre de anotaciones, en donde simplemente se usan aserciones, a las que py.test añade una capa que nos ofrece detallada información de un posible error:

```
=================================================================================== FAILURES
===================================================================================
_____________________________________________________________________________
test_selection_sort
_____________________________________________________________________________

    def test_selection_sort():
        assert selection_sort([]) == []
        assert selection_sort([1]) == [1]
        assert selection_sort([1, 2]) == [1, 2]
>       assert selection_sort([1, 2, 0]) == [1, 2, 3]
E       assert [0, 1, 2] == [1, 2, 3]
E         At index 0 diff: 0 != 1
E         Use -v to get the full diff

tests/test_sort.py:8: AssertionError
========================================================================== short test summary
info ==========================================================================
FAILED tests/test_sort.py::test_selection_sort - assert [0, 1, 2] == [1, 2, 3]
```

Además, se puede comprobar que se han ejecutado ambos ficheros de test, el anterior y el nuevo, con un solo comando. Lo contrario **no es cierto**, ya que unittest **no es capaz de ejecutar** un fichero de py.test.

Test doubling

Está técnica sustituye **piezas reales** (conexión a una BD, llamada a un servicio remoto) por código fuente dentro de nuestros test. Este código fuente puede ser desde simples objetos que apenas cumplen una sencilla interfaz hasta simulaciones completas de la respuesta real de un servicio web.

Consulta

El mejor artículo para entender qué son los test doubles

Usar test doubles tiene una desventaja que **no se puede obviar**: al codificar nosotros mismos las respuestas de los servicios, caemos en el riesgo de que nuestros test no sean fieles a la realidad, o que queden obsoletos a lo largo del tiempo si el servicio simulado cambia su implementación.

Ejemplo:

```
class InfoUsuario:
    def __init__(self, db: DbConnection, id):
        self.__db = db
        self.__id = id

    def consulta_edad():
        return self.__db.get_datos(id)['edad']


def test_info_usuario_consulta_edad():

    # Esto no puedo hacerlo, pues sería una conexión de verdad
    db = DbConnection(usuario, password)
    assert InfoUsuario(db, 123456).consulta_edad() == 40    # fallo

    class MockDbConnection(DbConnection):
        def get_datos(self, id_usuario):
            return {
                'edad': 40
                'nombre': 'Elvis'
            }

    # Solo quiero probar InfoUsuario, no la base de datos
    mock_db = MockDbConnection(usuario, password)
    assert InfoUsuario(mock_db, 123456).consulta_edad() == 40    # exito
```

Los frameworks de testing que hemos estudiado también tienen sus propias funcionalidades de test doubling.

- Java: Mockito.
- Python nativo: unitest.mock.
- Pytest: Usa unittest.mock a través de pytest-mock.

Tipos de test

Vamos a ver con más detalle qué diferentes tipos de test existen. Es interesante conocerlos para poder 'ponerles nombre', ya que diferentes test deben figurar en diferentes partes de un ciclo de vida efectivo de una aplicación, **y esto es algo que como DevOps no podemos obviar**: nuestro objetivo debe ser siempre mejorar procesos y aumentar la velocidad.

Manuales

Los test manuales son los más frecuentes, el tipo de test que implica probar de primera mano que la funcionalidad es correcta: puede ser comprobar visualmente un componente web, validar unos resultados en la base de datos, o que una respuesta HTTP sea correcta.

¿El problema? Que este proceso es en el que más fácil resulta equivocarse (no en vano es un proceso humano) y, además, no es reproducible de forma automática, por lo que no es posible usar esos resultados en ningún proceso de integración continua.

Unitarios

Es el caso más común de prueba, aquel que usa el mismo lenguaje de programación de nuestra aplicación y el que está a un nivel más bajo (cercano a la lógica).

Su principal característica es que deben ser **rápidos**: una batería de test unitarios debe poder ejecutarse en pocos segundos, o no valdrían para darnos feedback.

Son muy útiles para probar, de manera rápida, que estamos por el buen camino de una implementación de un algoritmo o devolviendo la respuesta esperada en una llamada. También se usan para probar todos los *corner-cases* de una función de una vez, ahorrándonos tiempo de pruebas manuales con su ejecución **constante**.

Ejemplo:

Un ejemplo de un test unitario para un método cualquiera, escrito en pseudocódigo, podría ser:

```
void reverseString(String cadena) {
  // ...
  return result
}

void test_reverseString() {
  assert reverseString('') == ''
  assert reverseString('a') == 'a'
  assert reverseString('123') == '321'
  assert isThrown(InvalidInputException, () => reverseString(null))
}
```

Integración

Los test de integración nos aseguran que diferentes módulos de nuestra aplicación funcionan bien entre sí. También son test de integración aquellas pruebas que dependen de terceros como, por ejemplo, una base de datos o un servicio REST determinado.

Si para realizar esas pruebas tenemos que comunicarnos con versiones de prueba de esos servicios o levantar y configurar uno de ellos usando un contenedor, o alguna otra tecnología, estaríamos hablando de test que llevan mucho más tiempo que los unitarios, tanto de ejecución como de implementación.

Si es posible simular la respuesta de esos módulos de alguna manera, los test no difieren mucho de los test unitarios. Para ello, haremos uso, **si el diseño del código lo permite**, de técnicas de test doubling.

Funcionales

Muy similares en concepto a los test de integración, los test funcionales tienen un componente extra: se enfocan más a los requerimientos concretos del cliente, por lo que en lugar de probar con valores aleatorios o genéricos, deben demostrar que las piezas del sistema se comportan del modo esperado ante una entrada determinada.

La palabra clave es la interacción entre sistemas, así que un test funcional es más posible que dependa de servicios en lugar de usar doubles.

End-to-end

Un test E2E es un test complejo que pretende demostrar que nuestro software cumple las especificaciones para las que fue diseñado. Esto requerirá que el entorno de pruebas sea real o lo más parecido al real posible: copias de la base de datos de producción, servicios idénticos a los productivos, pero con otra configuración. Están muy relacionados con el concepto de **entornos de pruebas**: entornos con diferentes características para simular cómo se comportará nuestra aplicación cuando llegue a producción.

Rendimiento

Estos test sirven para demostrar que diferentes algoritmos, o incluso aplicaciones enteras, se ejecutan dentro de unos parámetros aceptables de velocidad, consumo de memoria, uso de disco y escalabilidad.

Una aplicación inmediata es para probar, por comparación, mejoras o refactorizaciones de la implementación. O decidir si el estado actual del software es aceptable para su puesta en marcha.

Las técnicas de pruebas de rendimiento son muy diversas y varían en función del lenguaje usado, del tipo de prueba que se quiera hacer, etc.

Ejemplo:

Un ejemplo de este tipo de test es el análisis de rendimiento del algoritmo recursivo de Fibonacci de la unidad anterior.

Smoke test

Una 'prueba de humo' es un tipo de test que debe ser **rápido** y ofrecer como salida si nuestra aplicación cumple uno o más requisitos para seguir con pruebas más exhaustivas.

Se puede considerar un subconjunto de los demás test, pues no obedece a un tipo concreto sino a un propósito: si los smoke test no funcionan, no se continuaría con pruebas más costosas en CICD como, por ejemplo, pruebas que levanten entornos virtuales efímeros en infraestructuras inmutables (creadas desde cero para garantizar que el estado entre pruebas o despliegues sea **idéntico**).

Ejemplo:

Como veremos cuando lleguemos a la sección dedicada a Kubernetes, cuando vamos a desplegar un servicio en esta plataforma existe un valor de configuración llamado '**readiness'.**

'Readiness' describe la ubicación del endpoint de nuestra aplicación que responde un *HTTP 200 OK* si, y solo si, el sistema está dando servicio correctamente. Internamente, este endpoint podría comprobar todos los parámetros de ejecución críticos de la aplicación, como por ejemplo:

- Que las conexiones a la base de datos se han establecido con éxito y que tenemos los permisos esperados.
- Que existe conectividad con servicios de terceros que necesitemos para operar, como un servicio de configuración (*etcd)* o una cola de mensajes (*RabbitMQ)*.

Si esta llamada **falla** y devuelve error, Kubernetes considera que el servicio está KO y hace *rollback* del despliegue.

Características de una batería de test (test suite)

Aislados

- Un test no debe modificar estados globales ni depender del estado de alguna variable que pueda ser modificada desde otro test.
- Un test no debe depender del orden de ejecución de la batería de test.

Autocontenidos

- Un test, o una batería de test concreta, debe encargarse de dar de alta y destruir todos los recursos que necesita para su correcta ejecución.

Repetibles

- Si no cambia el código fuente, el resultado de la ejecución de los test no debe cambiar jamás, independientemente del número de veces que se ejecute.

Informativos

- En caso de error, un test debe arrojar toda la información posible para su resolución inmediata.
    - Aquí hay que ponerse en los zapatos del compañero al que le toque arreglar nuestros test.

Confiables

- Los test deben darnos **garantías** de que si se cambia algo de lógica en el código fuente, algún test fallará por eso.
- Los test deben **generar confianza** en que todos los casos posibles por los que un error pueda aparecer en tiempo de ejecución están registrados y probados.

Rápidos

- Una batería de test debe ejecutarse de forma rápida. Tan rápido como sea posible. De otro modo, no se ejecutarán los test más que de vez en cuando.
- Si existen test lentos por definición, evitar pasarlos por defecto.

Código limpio (*clean code*)

Clean code, clean architectures, software craftsmanship… todos estos conceptos, acuñados por muy diferentes autores desde los tiempos en que se firmaba el manifiesto ágil, aluden a los mismos conceptos, una y otra vez:

- Considerar la creación de software como un arte, que se puede mejorar con la práctica y siguiendo una serie de preceptos.
- Mejorar la mantenibilidad de las aplicaciones mediante la mejora de la manera en que están escritas.
- Si el código es fácil de entender, es fácil de mantener.

El mundo de clean code es extenso, lleno de literatura interesante al respecto cuya lectura se recomienda encarecidamente:

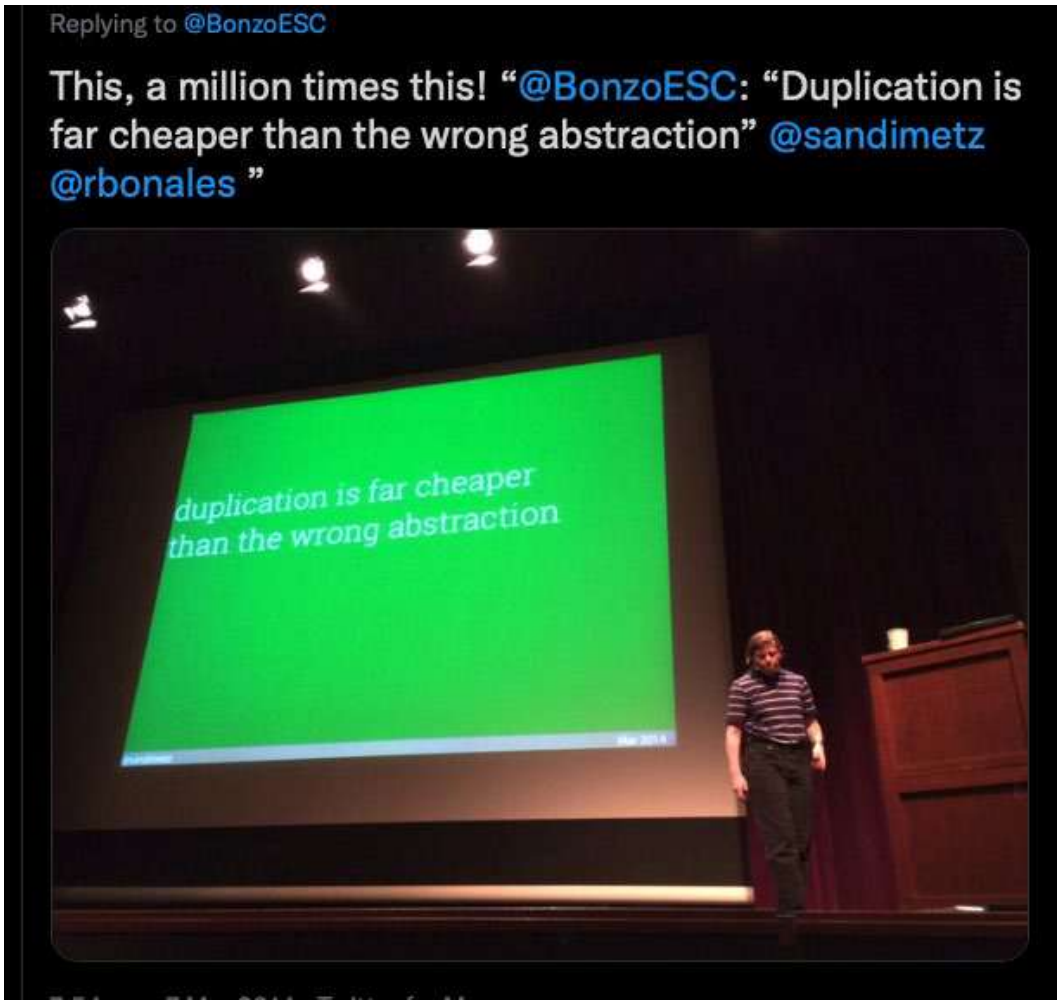
- *The pragmatic programmer* (Andrew Hunt & David Thomas).
- *Clean code* (Robert C. Martin) (ISBN 9780132350884).
- *Clean architectures* (Robert C. Martin) (ISBN 0134494164).
- *The software craftsman* (Sandro Mancuso) (ISBN 0134052501).
- *Software craftsmanship manifesto*.

Se puede sacar un factor común de todos estos libros y de toda la cantidad de información que existe al respecto. Se exponen aquí algunos de los principios que, aplicados correctamente, pueden ayudarnos a escribir código de mejor calidad, más mantenible y legible.

DRY - No te repitas / *Don't repeat yourself*

Se basa en la premisa de evitar duplicar código, ya que los fallos existentes habría que arreglarlos en varios sitios distintos o crearían efectos secundarios difíciles de prever y de corregir.

Si bien no parece en absoluto una mala práctica, y se debe seguir en la gran mayoría de los casos, hay que tener cuidado cuando se **desduplica** código fuente: al hacerlo estaremos tomando decisiones de diseño y creando abstracciones que quizá sean incorrectas, o no permitan que ese mismo código evolucione correctamente en el futuro:



Fuente: Twitter, blog de la autora.

KISS - Mantenlo simple / *Keep it simple, stupid*

La premisa, aplicable no solo a la ingeniería del software, es que normalmente las soluciones más simples funcionan mejor que las altamente sofisticadas. Se debe entender como el arte de **no añadir complejidad innecesaria**, de manera injustificada.

YAGNI - No lo vas a necesitar / *You ain't gonna need it*

La clave de este principio es el de no añadir nada a nuestro código que no sea estrictamente necesario (los famosos 'por si acaso'). La razón principal para no hacerlo es no tener que mantener código que no cumple ninguna necesidad de negocio, ya que nadie lo ha pedido.

Una variante de este principio, y derivado de los anteriores, podría ser **evitar optimizaciones prematuras**: es muy fácil caer en la tentación de hacer los mejores algoritmos posibles que se ejecuten a toda velocidad, pero -y especialmente al inicio de una tarea- lo más común es que **lo más prioritario** sea obtener un resultado **correcto** y solo entonces buscar conseguirlo de la mejor manera posible.

Principios SOLID

Se trata de un acrónimo con el que se conoce a cinco principios de diseño de programas en lenguajes orientados a objetos, acuñados por Robert C. Martin a principios del siglo XXI.

Si bien se puede decir que ha llovido mucho desde entonces, estos cinco principios no dejan de ser buenos consejos que vale la pena conocer:

Single responsibility principle

Un objeto solo debería tener una única responsabilidad.

***O**pen/closed principle*

Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación.

***L**iskov substitution principle*

Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Interface segregation principle

Muchas interfaces de cliente específicas son mejores que una interfaz de propósito general.

***D**ependency inversion principle*

Nuestro diseño debe depender de abstracciones, no depender de implementaciones concretas.

Inyección de dependencias

Una aplicación de la **I** y la **D** en S.O.L.I.D, la inyección de dependencias es una técnica de programación orientada a objetos en donde los métodos o las clases no crean instancias de clases para realizar sus tareas, sino que las reciben como parámetros.

Lo que se consigue con esto es que las clases dependen de una interfaz determinada y que mientras les pasemos un objeto que la cumpla, funcionarán correctamente.Veamos un ejemplo:

```
class Coche:
    def __init__(self, motor: Motor):
        self.motor = Motor

    def velocidad(self, pisada_de_acelerador):
        self.motor.velocidad(pisada_de_acelerador)


class Motor:
    def velocidad(self, pisada):
        pass

class MotorIbiza(Motor):
    def velocidad(self, pisada):
        return 50 * pisada

class MotorLamborghini(Motor):
    def velocidad(self, pisada):
        return 200 * pisada

ibiza = Coche(MotorIbiza())
lambo = Coche(MotorLamborghini())
```

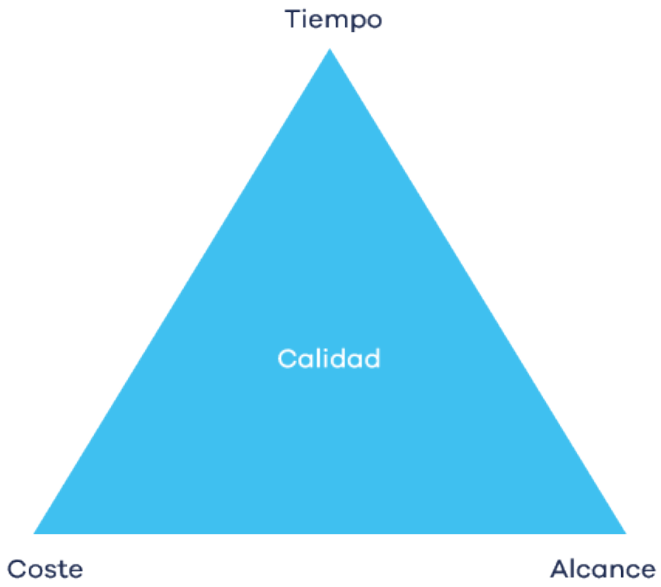
Esto tiene sus consecuencias en el mundo del testing, ya que facilita inmensamente la tarea del programador: es mucho más fácil modificar o crear una clase y pasársela a un método/clase, que modificar lo que hacen sus métodos.

De este modo, los test en código que usa inyección de dependencias mantienen aislado el código, pues se inyectan clases que cumplen el 'contrato', pero de los que controlamos el valor de retorno.

```
def test_un_coche_acelera():
    class MotorPrueba(Motor):
        def velocidad(self, pisada):
            return 42

    # No necesito saber como funciona el coche, solo que necesita un motor
    def coche_pruebas = Coche(MotorPrueba())
    assert coche_pruebas.acelera(100) == 42
```

Sobre la calidad del código

Al hacer tanto énfasis en la calidad del código, algo que requiere **tiempo,** se puede pensar que se compromete la integridad del famoso **triángulo de gestión de proyectos:**



La verdad es que siempre, en todo proyecto, es difícil justificar un aumento de tiempo de desarrollo en pos de perseguir una mayor calidad, ya que necesariamente implicaría:

- Reducir el alcance de un entregable, o de la aplicación, para mantener el coste original.
- Aumentar el coste para poder mantener el compromiso adquirido.

Si bien esto no se puede obviar, hay que combatir la **falacia** de que para conseguir mayor calidad en el código hace falta más tiempo, ya que precisamente la creación de código limpio, mantenible, con arquitecturas probadas y pensadas para generar software que pueda escalar en el tiempo con los requisitos nos ayudará a crear aplicaciones que **ahorran tiempo** en el futuro, ya que:

- Permiten que futuras ampliaciones/modificaciones se incluyan de forma más sencilla.
- Permiten añadir nuevos desarrolladores y que empiecen a contribuir mucho antes que en monolitos intragables.
- Tienen una base de pruebas sólida y completa, que ayuda en el mantenimiento productivo de la aplicación y futuras **refactorizaciones**.

Refactorización

La refactorización es el término empleado para nombrar al conjunto de técnicas que permiten, de forma iterativa y disciplinada, alterar la estructura interna o la arquitectura de una aplicación sin modificar en absoluto su comportamiento hacia el exterior.

La importancia de estas técnicas es enorme, pues no pocas veces nos encontraremos ante una pieza de software que, sin duda, creemos que hace algo de una manera cuestionable. Un ejemplo sencillo, que seguro que todos hemos podido ver en más de una ocasión:

```
public void calcula(String[] cds) {
  int d = 0;
  for (int i=0 ; i<cds.size(); i++) {
    String c = cds[i];
    if (c != null) {
      if (c.size() < 17 && c != 'test') {
        if (c[2] == 'K' && GLOBAL_RANDOM_FLAG == 42) {
          continue;
        } else {
          c = 'ok';
          if (c < 17) {
            d++;
          }
        }
      }
    }
  }
  return d;
}
```

Este (mal) ejemplo muestra algo que incumple casi todo lo que hemos comentado hasta ahora sobre código limpio y mantenible.

En general, si tenemos un buen conocimiento del dominio de la aplicación donde aparezca algo parecido a eso, lo normal es sentir la tentación de cambiarlo por algo que haga 'lo mismo, pero mejor'*:* esto es precisamente el proceso de refactorización.

Si bien existen multitud de técnicas para abordar el proceso de forma eficiente, lo más importante a tener en mente a la hora de refactorizar una parte de un software es:

- **Deben existir test**, o al menos alguna manera de garantizar que mantenemos **exactamente** la misma funcionalidad. Sin esto, nunca podremos decir que lo hemos conseguido.
- Debemos **acotar en el tiempo** el proceso: si una refactorización lleva semanas o meses no es un refactor, es una reconstrucción.
- Debemos realizar el proceso de forma iterativa, aportando valor en **pequeños lotes**.
- Imprescindible usar **control de versiones**, para almacenar cada una de esas deltas de funcionalidad y poder volver atrás en caso de tener necesidad.
- En ningún momento, el sistema debe **perder funcionalidad**, razón de más para contar con test que nos permitan, con un botón, saber que seguimos dando el mismo servicio.

Test-driven development (TDD)

TDD es un patrón de diseño en el que, a grandes rasgos, los test se codifican **antes** que el código fuente que los haría pasar: merece la pena dedicar tiempo a repasar este patrón y compararlo con las alternativas de desarrollo usuales.

- Si una condición o requisito de negocio no está reflejado en un test, **no es necesaria.**
- El código nunca debería implementar nada que no tenga un test que lo verifique. Dicho de otro modo, **primero se debe escribir un test, después el código que lo hace pasar (test first).**

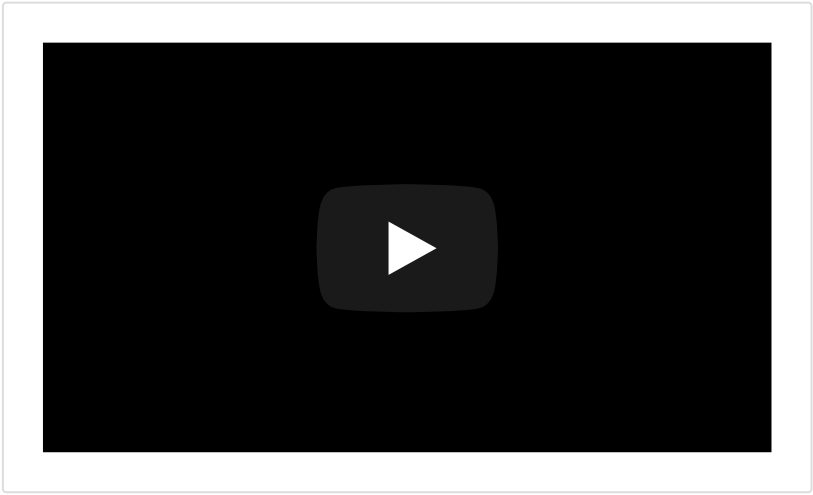
El flujo de trabajo debería ser algo similar a esto:

1. Crear un test con una condición que fallará, pues no existe el método al que hace referencia.
2. Añadir el citado método, vacío.
3. Una vez el test no falle porque el método ya existe, añadir una de las condiciones que sabemos que debe cumplir.
4. Modificar el código para que haga lo **mínimo imprescindible** para cumplir la nueva condición.
5. Repetir desde el paso 3 hasta que todas las funcionalidades estén presentes.
6. En este punto, **es seguro refactorizar**, pues sabemos que la funcionalidad está garantizada por la suite de test.

Esta manera de trabajar suele ser algo contraintuitiva para quien no está acostumbrada a usarla. Incluso para quien tiene mucha experiencia puede suponer un cambio muy radical, ya que puede parecer que no se enfoca a la resolución de problemas y es una simple técnica para resolver Katas o problemas sencillos.

Se ha demostrado que esto no es así, aunque la mejor manera de comprobarlo es probar, probar y probar, y usarlo solo si se ajusta a nuestra manera de trabajar.

Videotutorial de TDD



Scripting

No podemos cerrar una lección sobre programación y DevOps sin hablar sobre los scripts de shell: programas que facilitan, mediante el tratamiento automático de complejas líneas de parámetros o el encadenamiento de varios pasos en uno solo, tareas comunes o tediosas:

- Facilitar construcciones o despliegues 'con un botón'.
- Crear empaquetados complejos.
- Encadenar comandos con parámetros complicados.

Un script de shell, llamado así porque normalmente se ejecuta desde la línea de comandos, no tiene un lenguaje predeterminado: si bien existen lenguajes **específicos** para esto, como Bash o PowerShell, también se pueden usar lenguajes de programación para hacer (casi) el mismo trabajo, como por ejemplo:

- Python.
- Perl.
- Node.js.
- Golang.

En está sección, nos centraremos en el lenguaje de la shell.

Qué es la shell

Dicho de un modo muy sencillo, la shell es un macroprocesador que ejecuta comandos. Esta definición indica que hay una funcionalidad donde textos y símbolos se combinan para crear expresiones más grandes.

La shell es a la vez un intérprete de comandos y un lenguaje de programación. En su rol de intérprete de comandos, la shell ofrece al usuario una rica interfaz de utilidades o herramientas de GNU. Las características del lenguaje de programación, por su parte, permiten que estas herramientas se combinen.

Bash

Hay múltiples shells disponibles en entornos Linux y Unix.

Nosotros usaremos **Bash** ya que está disponible la práctica totalidad de distribuciones. En caso de no estarlo, es relativamente sencillo traducir scripts de bash en scripts de shell Bourne.

Antes de nada, recordad que cualquier duda, en cualquier momento, puede ser resuelta consultando el manual oficial de Bash.

Historia de Bash

La shell Bourne original data de 1979, cuando se empezó a distribuir en entornos UNIX. Todavía se encuentra en '/bin/sh' en muchas distribuciones y, de hecho, ha evolucionado solo en contadas ocasiones. Otra shell de la época, la shell C (csh) ofrecía funcionalidades útiles para uso interactivo, como control de procesos e historial de comandos.

Era habitual entre los administradores el uso de *sh* para programación de scripts y de *csh* para sesiones interactivas. Fue en 1983 cuando se presentó la shell Korn, o *ksh*, que se mantuvo compatible con *sh*, pero con funcionalidades interactivas de *csh*. Finalmente, Bash, de Bourne again shell, apareció en 1989 como un clon de *sh* escrito desde cero, incorporando muchas funcionalidades de *ksh* desde entonces.

Características de la shell

La shell puede funcionar en modo **interactivo** y en modo **no interactivo**.

- En **modo interactivo**, la shell acepta entrada desde el teclado, ya sea el teclado local o un teclado remoto en una sesión SSH. El sistema operativo arranca una shell en el momento en que un usuario inicia una nueva sesión, ya sea una sesión local o por SSH. Por tanto, una shell no es sino un proceso más.
- En **modo no interactivo**, la shell lee un archivo (es decir, un script) y ejecuta los comandos contenidos en él, línea por línea. Estos scripts pueden convertirse en comandos en sí mismos. Estos nuevos comandos tienen el mismo status que comandos del sistema en directorios como '/bin', permitiendo que usuarios o grupos puedan construir sus propios entornos y automatizar sus tareas comunes.

En cuanto a los comandos GNU, la shell permite su ejecución de modo síncrono y asíncrono. En el primer caso, la shell acepta un comando, lo ejecuta y espera a que este termine antes de aceptar el siguiente comando. Por su parte, los comandos asíncronos continúan ejecutándose en paralelo con la shell mientras lee y ejecuta comandos adicionales.

Las shell suelen incluir un pequeño conjunto de comandos que implementan funcionalidades que son casi imposibles de obtener por una vía diferente, por ejemplo, **cd, break, continue y exec**. Estos comandos no pueden ser implementados por fuera de la shell porque manipulan la shell directamente.

Otros comandos como '**history', 'getopts', 'kill'** o **'pwd'** pueden ser implementados por separado, pero, aún así es más conveniente utilizarlos como comandos 'built-in'. Algunos de estos comandos están orientados específicamente al uso interactivo, más que a aumentar el lenguaje de programación. Por ejemplo, la edición de línea de comando, el ya mencionado 'history', alias y los comandos de control de trabajo.

Funciones

Las funciones no son más que pequeñas subrutinas dentro de un script de shell. Son una forma de agrupar comandos para su ejecución posterior usando un solo nombre para referenciar varias sentencias. Se ejecutan como un comando regular u ordinario. Las funciones de shell se ejecutan en el contexto de shell actual: no se crea ningún proceso nuevo para interpretarlos.

Parámetros

Un parámetro es una entidad que almacena valores. Puede ser un nombre, un número o un carácter especial. Una variable es un parámetro denotado por un nombre. Un parámetro se establece si se le ha asignado un valor.

Comandos habituales

Estos comandos no son específicos de Bash, vienen por defecto (quizá con algún pequeño cambio) en cualquier shell.

Comandos de directorio:

```
# Muestra el camino completo del directorio actual.
$ pwd
/etc/apache2/extra

# Cambia de directorio
$ cd /etc/apache2
$ cd ~ # Lleva a la carpeta home del usuario.
$ cd - # Lleva a la última ruta.
$ cd ..# Cambia al directorio padre del directorio actual.
```

Listado de ficheros:

```
# Listado de ficheros.
$ ls
extra                   magic                   other
httpd.conf              mime.types              users
httpd.conf.pre-update   original

# Lista ficheros, carpetas e información
$ ls -al
total 128
drwxr-xr-x  10 root  wheel   320 Aug 27  2018 .
drwxr-xr-x  90 root  wheel  2880 Apr 18 17:22 ..
drwxr-xr-x  14 root  wheel   448 Apr  4  2018 extra
-rw-r--r--   1 root  wheel  21150 Aug 27  2018 httpd.conf
-rw-r--r--   1 root  wheel  21150 Apr  4  2018 httpd.conf.pre-update
-rw-r--r--   1 root  wheel  13077 Apr  4  2018 magic
-rw-r--r--   1 root  wheel  61118 Apr  4  2018 mime.types
drwxr-xr-x   4 root  wheel    128 Apr  4  2018 original
drwxr-xr-x   3 root  wheel     96 Apr  4  2018 other
drwxr-xr-x   3 root  wheel     96 Aug 27  2018 users

# Lista de forma recursiva.
$ ls -aR

# Enumera todos los archivos terminados en .htm
$ ls *.htm

#  Muestra un listado de los ficheros en ese directorio.
$ ls -al dir/subdir/
```

Crear, modificar y borrar ficheros y carpetas:

```
# Crea el fichero index.htm sin contenido.
$ touch /home/usr/html/index.htm:

# Borra file.txt
$ rm file.txt

# Borra el directorio de forma recursiva y sin confirmación.
$ rm -rf dir/

# Copia de ficheros
$ cp fichero_a fichero_b

# Cambio de nombre de fichero
$ mv fichero_a fichero_c

# Crea un directorio con el nombre carpeta.
$ mkdir carpeta

# Crea una estructura de directorios
$ mkdir -p carpeta/main/src/java carpeta/main/test/

# Borra el directorio 'carpeta'
$ rmdir carpeta
```

Archivos comprimidos:

```
# Comprime el directorio y su contenido en el fichero arch.zip.
$ zip arch.zip /home/usr/public/di

# Descomprime arch.zip.
$ unzip arch.zip

#  Visualiza el contenido de arch.zip.
$ unzip -v arch.zip

# Comprime directorio actual en package.tgz
$ tar cvzf package.tgz .

# Descomprime el fichero package.tgz.
$ tar xvzf package.tgz
```

Visualización de ficheros:

```
# Muestra el contenido del fichero en modo solo lectura; permite búsquedas, ir al principio y al final del fichero, pasar de página, etc.
$ less fichero.log

#  Vuelca el contenido del fichero en pantalla.
$ cat fichero.log

#  Muestras las últimas diez líneas del
$ tail fichero.log fichero.

# Muestra las últimas 5 líneas del fichero.
$ tail -5 fichero.log

#  Muestra las últimas 10 líneas del fichero y se engancha al fichero, mostrando las líneas nuevas que aparezcan en el fichero. Es muy útil para mostrar un fichero de log según se actualiza.
$ tail -f fichero.log

# Muestras las diez primeras líneas del fichero.
$ head fichero.log
```

Otros comandos:

```
# Copia todos los contenidos de un directorio a otro manteniendo sus permisos.
$ cp -a /home/usr/origen/* /home/usr/destino/

# Visualiza el espacio total ocupado por la carpeta actual.
$ du -sh

# Muestra el espacio ocupado de cada fichero.
$ du -sh *

# Espacio de disco en todos los volúmenes
$ df
$ df -h  # igual, pero valores “human readable”

# Muestra el nombre del usuario
$ whoami
```

Operadores de control:

Son tokens que tienen una función de control. Los más relevantes son:

```
# Comentarios

# ;
# Primero descarga el fichero, y luego lo procesa con un script
$ scp server:fichero.csv . ; ./procesa.sh -i fichero.csv

# |
# Vuelca salida grande a less para poder paginar resultado
$ ls -aR | less

# Comandos para contar las lineas de dos ficheros
$ cat fichero1.txt fichero2.txt | wc -l

# >
# Vuelca salida a fichero
$ ls -alR > resultado.txt

# &
# Envía tarea al segundo plano y continua
$ curl http://some.server/fichero_grande.pdf -o fichero.pdf &
$ curl http://another.server/otro_fichero_grande.pdf -o fichero2.pdf &

# devuelve última tarea al primer plano o la vuelve a enviar
$ fg
$ bg

# $?
# Muestra salida del último comando
$ mkdir /tmp/opt/random/dir # fallo!!
$ echo $?
1

# || y &&
# Si VAR1 tiene el valor 'val1', hago algo (dos sintaxis posibles)
$ test "$VAR1" = "val1" && apt install apache2
$ [ "$VAR1" = "val1" ] && apt install apache2

# Si no existe el fichero,lo crea
$ [ -f /tmp/datos.txt ] || touch ultimos_datos_vacio.txt
```

Cómo ejecutar un script

Cuando estamos en Linux, para que un fichero cualquiera se pueda ejecutar, podemos optar por varios métodos:

1. Llamarlo usando el ejecutable adecuado, normalmente asociado al lenguaje:

```
$ bash compila.sh
$ python release.py 1.0.0
```

2. Ejecución directa, para lo que deberemos:
    a. Marcar el fichero como ejecutable.

```
$ chmod +x release.py
```

    b. Especificar dentro del fichero qué intérprete debe usarse mediante un *shebang* en la primera línea del mismo, que indica o bien la ruta al binario en cuestión.

```
#!/bin/perl -v
```

O bien el nombre del binario que está presente en el entorno de shell actual, lo cual resulta **muy útil** cuando no sabemos o no tenemos por qué saber, la ruta o versión del mismo (por ejemplo, un 'virtualenv' de Python).

```
#!/usr/bin/env python3
```

Buenas prácticas

Si bien los posibles usos de este tipo de scripts son inabarcables, así como la manera de escribirlos, sí que se puede intentar dar una serie de reglas comunes para su creación y mantenimiento:

- Deben formar parte del repositorio del código al que dan soporte. Aunque no es raro tener repositorios dedicados únicamente a *tooling* especializado, el mantener scripts cerca del código que los necesita permite que el equipo los revise y entienda, y que se fomente su uso.
- Si se considera que el tamaño y alcance del script lo justifica, debe tratarse como una pieza de software más:
    - Documentación suficiente.
    - Código modular y claro.
    - Incluso algún test que pruebe la lógica más complicada, como algún bucle con *corner cases* oscuros.

- Si, por el otro lado, conseguimos condensar en unas pocas líneas de complicadísimo código shell la funcionalidad adecuada, es perfectamente válido también: **menos es más**, y no perdamos de vista que estos *scripts* se usan, fundamentalmente, como soporte de tareas más importantes.
- Si vamos a usar variables de entorno (lo cual es muy buena práctica), debemos dejarlo claro en alguna parte como, por ejemplo, en la cabecera del script, después del *shebang*.
    - De hecho, es muy usual que un script se pueda utilizar tanto en local como para despliegues dentro de un contenedor o en un sistema como Kubernetes, donde es muy común usar y abusar de variables de entorno para la configuración.
- Si nuestro script admite parámetros de entrada, hay que tener en cuenta que las shells **guardan histórico de ejecuciones**. Esto está muy bien cuando tenemos que ejecutar regularmente scripts con muchos o muy complicados parámetros, pero presenta una **grave vulnerabilidad** si alguno de esos parámetros es sensible, como credenciales de sistemas. Las alternativas en estos casos:
    - Leer las credenciales de ficheros de sistema, que sí son seguros pasar como parámetro, pero pueden tener restricciones de acceso.
    - Pasar dichas credenciales vía variables de entorno (recordemos, una buena práctica).
    - Leer las credenciales de algún sistema de configuración (aunque esto pueda presentar problemas, a su vez, de seguridad).
    - Delegarlo al usuario vía introducción interactiva de parámetros (lo que no es muy buena práctica e impide el uso del script en ejecuciones desatendidas).

LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA