

Verificación del Software

## 4. Técnicas y herramientas

# Verificación del Software

Tema 1. Fundamentos de Calidad del Software

Tema 2. QA en el SDLC

Tema 3. Revisión y pruebas

**Tema 4. Técnicas y herramientas**

Tema 5. Gestión de las pruebas

# Técnicas y herramientas

- 4.1. Herramientas para el testing
- 4.2. Modelos orientados a pruebas: TDD, BDD, ATDD
- 4.3. Automatización de pruebas
- 4.4. CI/CD y DevOps

**Verificación del Software**

## **4.1 Herramientas para el testing**

# Herramientas para el testing (1)

Las herramientas pueden ayudar en el testing de varias formas:

- Apoyo directo a la ejecución:
  - Diseño de tests, preparación de datos, ejecución de pruebas, ...
- Investigación y evaluación:
  - Análisis dinámico, monitorización de red, ...
- Gestión del proceso, de los tests y los defectos
- Ayudas generales:
  - Documentación, interrogación, visualización de datos, ...

# Herramientas para el testing (2)

El uso de herramienta aporta una serie de **beneficios**:

- Mejoras de la **eficiencia**, por ejemplo automatizando tareas laboriosas y repetitivas (ejecución de paquetes de regresión) o apoyando actividades manuales
- Mejora de la **calidad** y la **fiabilidad** del testing, reduciendo la probabilidad de fallo en las pruebas manuales, o realizando actividades de análisis y comparación a gran escala
- Abriendo la posibilidad a **actividades que no pueden realizarse manualmente**, por ejemplo todo lo relacionado con pruebas de performance y escalado

Es muy importante no olvidar que:

- Hay herramientas que interfieren en el comportamiento del sistema,
- O afectan a su rendimiento,
- O introducen código adicional que distorsiona el análisis del software y las métricas de cobertura.

# Clasificación de herramientas

Hay muchas herramientas que pueden ayudar en el testing, por lo que hay que organizarlas de alguna forma: propósito, plataformas, uso, licencia, ... Una forma de organizar las herramientas es en función de las actividades a las que dan soporte, considerando que algunas pueden ofrecer apoyo en varias actividades.

Distinguiremos 6 tipos principales:

- Gestión de las pruebas
- Pruebas estáticas
- Diseño e implementación de las pruebas
- Ejecución
- Rendimiento
- Especiales

# Herramientas para gestión de pruebas (1)

Cubren un amplio espectro de funcionalidades:

- Gestión de pruebas y ciclo de vida
  - Soporte al desarrollo y monitorización de pruebas, incluyendo en muchos casos otros aspectos como la gestión de defectos
  - Ayudan a planificar, organizar, registrar y seguir tests cases y sus ejecuciones
- Gestión de requisitos
  - Algunas de propósito especial para organizarlos desde el punto de vista de Negocio o técnico
- Gestión de defectos
  - Para seguimiento, informes, trazabilidad
- Gestión de la configuración
  - Control de versionado de todos los componentes implicados en la construcción de productos software: código, configuraciones, casos de prueba
- Herramientas para Integración Continua (CI, *continuous integration*)
  - Builds automáticos, ejecución automática de análisis, de test unitarios o de integración



# Herramientas para gestión de pruebas (2)

En estas herramientas es muy importante proporcionar información y reporting de forma accesible y rápida, ya que deben dar apoyo a la gestión.

También es habitual que proporcionen interfaces con otras herramientas que les proporcionen datos para elaboración la información de gestión y seguimiento.

Muchas herramientas de gestión son extensibles para integrar todos los elementos del ciclo de pruebas bajo su paraguas. Veremos un ejemplo más adelante.

# Herramientas para pruebas estáticas

Este tipo de análisis sólo tiene sentido con herramientas, tal y como se mencionó en el módulo 3:

- Análisis del código fuente sin necesidad de ejecutarlo
- Identificar potenciales defectos, anomalías, indicadores de calidad, ...
- Algunos indicadores extraídos con estas herramientas, como la complejidad ciclomática, ayudan a anticipar la mantenibilidad del código a través de lo complicado que puede resultar su seguimiento

# Herramientas para diseño e implementación de pruebas

Son ayudas para la construcción de casos de prueba, por ejemplo usando lenguajes formales de especificación aplicando técnicas TDD y BDD, como veremos más adelante.

Tipos:

- Basadas en modelos:
  - Para generar casos de prueba a partir de la modelización del código o sistema
- Preparación de datasets
  - Extracción y preparación de datos para generar los que se usarán en la prueba
  - También pueden revisar la calidad del dato (completitud, formato, ...)

# Herramientas para ejecución

Se trata de herramientas que ayudan en el proceso de ejecución de testing.

A su vez, pueden ser de varios tipos:

- Herramientas de ejecución
  - Encargadas de lanzar las pruebas que se ejecutan automáticamente. Muchas de las usadas para CI caerían en esta categoría.
- Herramientas para analizar la cobertura de código
  - Usadas en análisis de caja blanca, permiten determinar las ramas e instrucciones que deben ejecutarse para ayudar a definir los tests.
- Herramientas para seguimiento de cobertura
  - En realidad, lo que hacen es dar soporte a la gestión haciendo seguimiento de pruebas que se han ejecutado frente al conjunto previsto. Muy importantes en procesos de regresión donde hay un gran número de tests que no siempre pueden lanzarse automáticamente.
- Otras herramientas para dar soporte a la ejecución
  - Por ejemplo, las que permite aislar determinados elementos para poder ser ejecutados individualmente. Son una ayuda a las pruebas pero también al desarrollo.

# Herramientas para performance

La forma de probar las capacidades de un sistema requieren simular condiciones especiales de carga, demanda, de red, ... Las herramientas usadas para ello son muy específicas y a veces requieren usar equipos especializados (*appliances*).

Pueden ser:

- Herramientas específicas de performance
  - Generan cargas con distintos perfiles y simulan comportamientos individuales o agregados de usuarios o de otros sistemas
- Herramientas de análisis
  - Estudian el comportamiento del sistema determinando magnitudes como tiempos de respuesta, recuperaciones ante caídas, puntos de saturación, ... y el consumo de recursos críticos como memoria, procesador o almacenamiento

# Herramientas especiales

Todas aquellas que no tienen cabida en alguna de las categorías anteriores:

- Herramientas para usabilidad
  - Revisión de enlaces, completitud e integridad (en el caso de Web)
  - Comportamiento de usuario (secuencia de navegación, atención, tasa de abandono, ...)
- Herramientas para accesibilidad
  - Análisis automatizado de determinados parámetros que determinan el grado de accesibilidad de una aplicación. Por ejemplo imágenes con textos alternativos, preparación para conversores de texto a voz, etc
- Herramientas para localización
  - Análisis de textos, restricciones legales, consideraciones culturales, ...
- Herramientas para seguridad
  - Detección automatizada de vulnerabilidades, inyección de elementos maliciosos

## 4.2 Modelos orientados a pruebas: TDD, BDD, ATDD

# Test-first programming

En el año 2000 Ken Beck publicó el libro “eXtreme Programming” que recoge un marco de trabajo ágil que llevaba ya varios años aplicándose.

Este marco de trabajo está construido desde y para el desarrollo software, al contrario de otros más populares y generales como Scrum o Kanban.

El principio “***Test first programming***” de XP significa que los tests se definen e implementan **ANTES** de programar el código.

Los tests son una red de seguridad que permite mejorar el código (*refactoring*) o corregirlo (*debugging*) sabiendo que los tests que confirman su funcionamiento están disponibles para verificar en todo momento que todo sigue siendo correcto.

La técnica que facilita este principio es el llamado TDD, *Test Driven Development* o Desarrollo Dirigido por Pruebas.

Para cada funcionalidad, se introduce el test y a continuación el código que lo valida.

## Prácticas XP:

- Sit together
- Whole team
- Informative workspace
- Energised work
- Pair programming
- Stories
- Weekly cycle
- Quarterly cycle
- Slack
- Ten-minute build
- Continuous integration
- **Test first programming**
- Incremental design



# Las tres reglas de TDD (1)

El proceso TDD se puede explicar con estas tres reglas:

1. **No escribas código hasta que no hayas escrito el test** que falle debido a la ausencia de ese código.
2. **No escribas más tests de los necesarios para producir el fallo.** Es decir, no escribas tests adicionales para otra funcionalidad.
3. **No escribas más código del necesario para pasar el test.** Es decir, no agregues funcionalidad que no ha pedido nadie y que no esté validada por tests.

# Las tres reglas de TDD (2)

Estas tres reglas definen un **proceso iterativo** en el que progresivamente se van introduciendo nuevas funcionalidades programando paso a paso los tests y el código que los satisface.

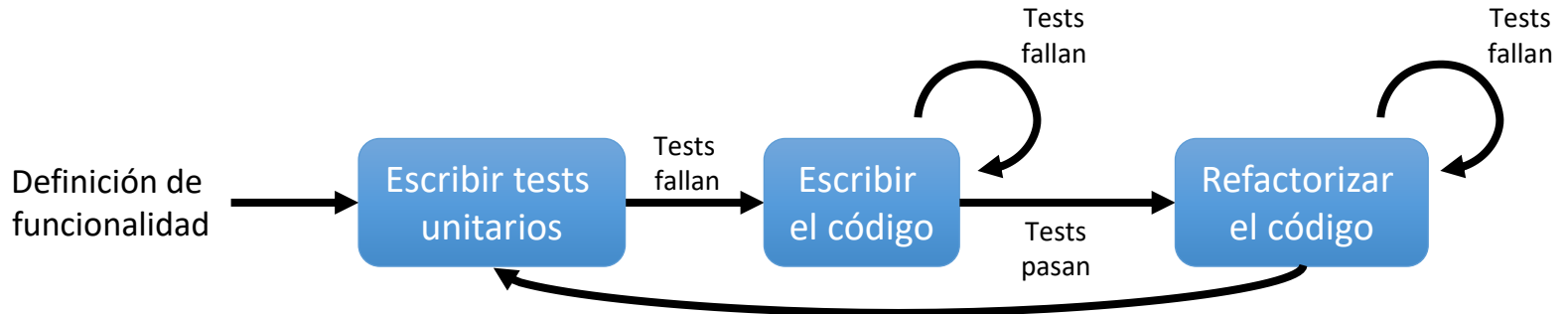
Cuando se trabaja así, una vez se ha conseguido un código que satisface un test, la próxima vez que falle (por introducir un nuevo test) las modificaciones necesarias serán pequeñas, reduciendo la complejidad y la probabilidad de error. Con TDD los tests son una red de protección para evitar introducir grandes errores, y al mismo tiempo un sistema que genera confianza frente a modificaciones por refactoring o mejoras.

Los tests son además una forma inequívoca de **documentar** el comportamiento del código.

# El ciclo TDD (1)

- Escribir un test automatizado
- Confirmar que falla (no hay código que lo cumpla)
- Escribir el código
  - Ejecutar el test e ir agregando/modificando código hasta que pase
- Refactorizar el código (defectos, mejoras)
  - Ejecutar el test e ir modificando código hasta que pase
- Seguir con el siguiente tests unitario

El proceso de refactorización puede hacerse más adelante en cualquier momento.



# TDD y la programación

TDD aporta un medio para ganar confianza en el código de forma que las modificaciones cuenten con una red de seguridad. Además, TDD impone cierto estilo de programación:

- **Definir el comportamiento** claramente y la forma de validarlo **antes** de escribir el código: es una forma de pensar poco convencional que requiere adaptación
- **Simplicidad**: el software debe mantener simple y comprensible desde el primer momento para seleccionar sólo aquello que va a ser necesario (*"Keep it simple, stupid"*, KISS).
- **Foco**: como consecuencia de lo anterior, hay que evitar el *goldplating* y agregar funcionalidades que no ha pedido nadie (*"You aren't gonna need it"*, YAGNI).
- **Trabajar en bloques pequeños**: no tiene sentido construir todas las pruebas y a continuación todo el código. Hay que ir paso a paso, ganando confianza y seguridad.
- **Acostumbrarse a fallar**: el fallo no es la excepción, si no algo que ocurre desde el principio. Hay que convivir naturalmente con él.

Hay que tener muy presente que TDD tiene limitaciones: no es una buena solución para problemas complejos, no permite validar el end-to-end.

Es una buena idea para las pruebas unitarias o de componentes, pero más allá no suele ser apropiada.

# Estructura de un test con TDD

TDD aporta un medio para ganar confianza en el código de forma que las modificaciones cuenten con una red de seguridad.

Para ello, los tests TDD deben seguir cierta estructura:

- **Precondiciones:**
  - establecer las condiciones necesarias para hacer la prueba (estados, datos, ...).
- **Ejecución:**
  - Lanzar el componente (función, método, bloque de código) para validar el comportamiento que se está analizando.
- **Validación:**
  - Asegurarse de los resultados obtenidos son los correctos.
- **Postcondiciones:**
  - Acciones necesarias para volver al estado inicial

# Buenas prácticas TDD (1)

Dado que TDD es una forma de integrar el testing en el código, el punto de partida es hacer un **buen diseño de tests**, para lo que son de ayuda las técnicas que hemos ido viendo hasta ahora. Por ello **desarrollar mentalidad de testing** es la mejor forma de abordarlo.

Los tests unitarios:

- Deben ser **independientes**
- **Simular** elementos externos
- Tener **alcance muy limitado**
- Y poder ejecutarse **aisladamente**.

# Buenas prácticas TDD (2)

TDD no se adapta a integraciones ni funciona bien con dependencias. Pero a veces es inevitable contar con elementos externos. Por ese motivo, TDD se apoya en componentes simulados como mocks o dummies, que pueden ser programados o contruidos con la ayuda de herramientas externas.

¡Cuidado! Un test **no es un test unitario** si:

- Habla con la base de datos
- Modifica ficheros
- Se comunica por la red
- No puede ejecutarse a la vez que otros tests unitarios
- Hay que cambiar el entorno para lanzarlo

# Cobertura de testing en TDD

Una forma de conocer el grado de seguridad que aporta TDD es conocer su **cobertura**. La cobertura aquí se refiere a las unidades de código (funciones, métodos, instrucciones) que cuentan con tests que las validan.

Idealmente, y si el código se ha construido desde el principio con TDD, la cobertura debe ser del 100%, sin embargo puede que no sea así por:

- Generar deuda técnica, dejando sin cubrir partes del código por distintas razones (falta de tiempo, esfuerzo asignado, prioridades, ...).
- Construir los tests después del código, por lo que se rompe el proceso paso a paso de crear tests y a continuación código.

La métrica de cobertura es un buen indicador para el **equipo** del grado de confianza en el código. Usarla como métrica de gestión o como objetivo conduce a desvirtuar el propósito (y con ello las ventajas) de TDD.



# Herramientas para TDD (1)

No es estrictamente necesario contar con herramientas para hacer TDD, pero simplifican mucho el trabajo. Los tests pueden crearse como funciones dentro del código que se prueba o –mejor- uno externo que haga llamada a los métodos, funciones, subrutinas pasando unos datos de entrada y contrastando con unos valores de salida que se han definido previamente con las técnicas que hemos visto hasta ahora.

La familia **xUnit** es una colección de frameworks creados originalmente por Ken Beck para TDD. La primera letra se refiere al lenguaje de programación (JUnit para Java, RUnit para R, PyUnit o unittest para Python, ...).

# Herramientas para TDD (2)

Cada elemento de la familia comparte una misma arquitectura con:

- **Test runner:** programa que ejecuta el test.
- **Test case:** clase de la que se heredan los tests.
- **Test fixtures:** que alberga las precondiciones.
- **Tests suite:** colección de tests que comparte el mismo contexto o fixture.
- **Test execution:** ejecución de los distintos tests, que arranca con un “setup” de condiciones y termina con un “teardown” que limpia el entorno para otros tests.
- **Test result formatter:** encargado de dar formato al resultado.
- **Assertion:** encargado de monitorizar el comportamiento del objeto probado.

Además, hay herramientas para mocks, simulación, gestión, etc

# Ejemplos TDD (1)

En Python es muy sencillo armar un ejemplo al contar con un módulo que nos ayuda a crear tests unitarios, `unittest`:

```
import unittest
from mycode import *

class MyFirstTests(unittest.TestCase):

    def test_hello(self): self.assertEqual(hello_world(),
        'hello world')
```

En paralelo creamos una función `hello_world()` sin que haga nada (sólo hemos creado el test):

```
def hello_world():    pass
```

# Ejemplos TDD (2)

El resultado de ejecutar el test es:

```
F
=====
FAIL: test_hello (__main__.MyFirstTests)
-----
Traceback (most recent call last):
  File "TDD1.py", line 6, in test_hello
    self.assertEqual(hello_world(), 'hello world')
AssertionError: None != 'hello world'

-----

Ran 1 test in 0.001s

FAILED (failures=1)
```

# Ejemplos TDD (3)

Si ahora cambiamos la definición de la función `hello_world`:

```
def hello_world():  
    return 'hello world'
```

El resultado de ejecutar el test será:

```
.  
-----  
Ran 1 test in 0.000s  
  
OK
```

# Ejercicio: partición equivalente

SOLUCIÓN

Busca las particiones para este supuesto

0%		No válido		2%		5%		10%	
$-\infty$	-0,01	0,00	24,99	25,00	49,99	50,00	99,99	100,00	$\infty$

Caso de prueba	Entrada	Salida esperada		Partición
	Gasto	Descuento	Total	
1	20,35	0	20,35	0%
2	44,44	0,89	43,55	2%
3	75,75	3,79	71,96	5%
4	1234,56	123,46	1111,10	10%
5	-123,45	Mensaje de error	Mensaje de error	No válido

# Ejemplos TDD (4)

Podemos hacer unos tests muy simples para validar la función que calcula el descuento:

```
def test_intereses(self)
    self.assertEqual(descuentos(20.35), 0)
    self.assertEqual(descuentos(44.44), 2)
    self.assertEqual(descuentos(75.75), 5)
    self.assertEqual(descuentos(1234.56), 10)
    self.assertEqual(descuentos(-123.45), -1)
```

Normalmente será mejor usar valores fijos, o si son calculados, usar una forma alternativa de cálculo.

Nunca debemos usar para calcular el valor esperado el mismo método de cálculo en la función real, porque entonces sólo validamos que las dos funciones son iguales, pero no que sean correctas.

# Ejemplo paso a paso TDD (1)

Conociendo los fundamentos, vamos ver cómo es realmente programar usando TDD, no sólo elaborar los tests cases para código que ya existe. Vamos a construir una función que suma números.

Fichero Tests:

```
def test_suma_numeros():  
    calculadora = CalculadoraSimple()  
  
    result = calculadora.add(4, 5)  
  
    assert result == 9
```

Fichero Código:

```
class CalculadoraSimple:  
    pass
```

La prueba falla.



# Ejemplo paso a paso TDD (2)

Agregamos código suficiente hasta que pase el test:

```
class CalculadoraSimple:  
    def add(self):  
        pass
```

Sigue fallando.

# Ejemplo paso a paso TDD (3)

Y ahora:

```
class CalculadoraSimple:  
    def add(self, a, b):  
        pass
```

Falla.

# Ejemplo paso a paso TDD (4)

Y finalmente:

```
class CalculadoraSimple:  
    def add(self, a, b):  
        return 9
```

**¡Funciona!**

¿Qué hemos hecho? Seguir el tercer principio: “No escribas más código del necesario para pasar el test”.

Es el punto que más puede chocar de aplicar TDD.

# Ejemplo paso a paso TDD (5)

Ahora que tenemos un test que pasa, es el momento de refactorizar: podemos hacer cambios con la red de seguridad que nos da el tener unos tests unitarios que funcionan. Ahora llegar el momento de mejorar el código generalizando:

```
class CalculadoraSimple:
    def add(self, a, b):
        return a+b
```

# Ejemplo paso a paso TDD (6)

Podemos seguir avanzando incorporando nuevas posibilidades, como un número variable de argumentos, pero antes tendremos que agregar nuevos tests. Por ejemplo:

```
numbers = range(100)
result = calculator.add(*numbers)
```

Y:

```
class CalculadoraSimple:
    def add(self, *args):
        return sum(args)
```

# Ejercicio: TDD

## Viaje en tren

Hay nuevas tarifas y condiciones para billetes de tren y te ha tocado programar la lógica que calcula los descuentos y la oferta de asientos.

Se establecen varios tramos (la edad es un valor entero):

- Menores de dos años viajan gratis
- Hasta 14 (menores de 15) hay un descuento del 30%
- Los mayores de 54 tienen un 30%
- Los mayores de 64 tienen un 60% de descuento.

Además hay una tarjeta de fidelización pero sólo para mayores de edad que aporta un 10% adicional de descuento en cada tramo.

Por otra parte, hay cuatro tipos de acomodaciones para los tres tipos de trenes (Mañana, Tarde y Noche):

- Asientos individuales
- Mesas, sólo para viajeros en grupos de 4
- Literas, sólo en trenes nocturnos
- Departamentos, sólo en para viajeros en grupos de 2 en trenes nocturnos

Construye la lógica aplicando TDD, es decir, construyendo los tests y el código.

# Ejercicio: TDD

SOLUCIÓN

## Viaje en tren (1)

En primer lugar hay que crear las clases de equivalencia, los valores límite y las tablas de decisión.

Sabemos que la edad es un entero, lógicamente positivo. Podemos excluir los caracteres no numéricos, los decimales, los negativos y los números muy altos (por ejemplo a partir de 150).

Lo que nos queda puede pintarse así:

Error	100%		30%		30%		0%		30%		60%		Error		
$-\infty$	-1	0	1	2	14	15	17	18	54	55	64	65	149	150	$+\infty$

Los descuentos se refieren al caso de que no haya tarjeta. Si la hay, hay que añadir un 10% adicional en los tramos **a partir de 18**. Por ese motivo hay un tramo extra entre 15 y 18 para el caso de viajeros sin tarjeta de descuento.

Eso quiere decir que tendremos que preguntar por la edad y si hay o no tarjeta.

# Ejercicio: TDD

SOLUCIÓN

## Viaje en tren (2)

Para los asientos disponibles hay que montar una tabla de decisión sobre el número de viajeros del grupo, y el horario del tren:

	1	2	3	4	5	6	7	8	9
<i>Condiciones</i>									
Tren Mañana	S	S	S	N	N	N	N	N	N
Tren Tarde	N	N	N	S	S	S	N	N	N
Tren Noche	N	N	N	N	N	N	S	S	S
Núm. Viajeros	*	2	4	*	2	4	*	2	4
<i>Acomodación</i>									
Asiento	S	S	S	S	S	S	S	S	S
Mesa	N	N	S	N	N	S	N	N	S
Litera	N	N	N	N	N	N	S	S	S
Departamento	N	N	N	N	N	N	N	S	N



# Ejercicio: TDD

SOLUCIÓN

## Viaje en tren (3)

La tabla de decisión se puede simplificar con facilidad:

	1	2	3	4
<i>Condiciones</i>				
Tren Mañana	~	~	N	N
Tren Tarde	~	~	N	N
Tren Noche	~	~	S	S
Núm. Viajeros	~	4	~	2
<i>Acomodación</i>				
Asiento	S	~	~	~
Mesa	~	S	~	~
Litera	~	~	S	~
Departamento	~	~	~	S

A partir de aquí se pueden construir los test **uno a uno** junto al código que los resuelve.

# BDD y ATDD

Se trata de dos conceptos muy similares. Se basan en la misma idea de *Test First* que se aplica en TDD. Es decir: escribir primero las pruebas, y luego el código que las satisface. En el caso de BDD y ATDD el objetivo son los requisitos, y el resultado el producto final.

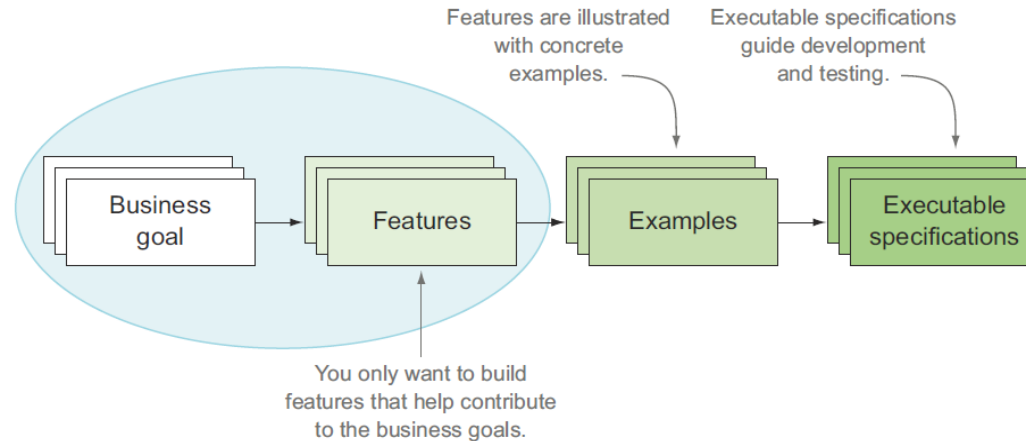
BDD o *Behavior Driven Development*, es un método que facilita construir requisitos de forma que sea posible elaborar la prueba que debería satisfacer el código para considerar que el requisito se ha cumplido. Esto se traduce en un lenguaje formal para construir unos requisitos *ejecutables*. Por lo demás, el sistema funciona de forma parecida a TDD, sólo que enfocada a comportamientos más sofisticados en el nivel de integración o sistema. La orientación de BDD también se puede usar como sustituto de TDD

ATDD o *Acceptance Test Driven Development* es una metodología de trabajo que implica a todo el equipo, normalmente atiende a un nivel superior (hasta pruebas de aceptación) y busca no sólo que se haga **correctamente**, sino además que se haga **lo correcto**.

En el fondo, las diferencias son pequeñas y hay autores que reconocen que en realidad son lo mismo, o ven a BDD una evolución de ATDD.

# Objetivos de BDD

BDD no es una forma de automatizar las pruebas, ni siquiera una forma de especificación formal de las funcionalidades. Es ante todo, una serie de técnicas que ayudan en el descubrimiento de las funcionalidades que hay que implementar, que luego se traduce en un sistema de “especificaciones ejecutables”:



Sin embargo, nosotros nos centraremos en las etapas finales. Pero hay que recordar que BDD es sobre todo una herramienta para definir más que para construir productos.

# Principios BDD

Como framework, BDD acerca la tecnología a Negocio y viceversa. De hecho, aunque se presenta como una herramienta para el testing, es habitual que sea gente con perfil de negocio (Product Manager, Program Manager, Product Owner) quien lo use.

BDD se basa en la aplicación de tres principios:

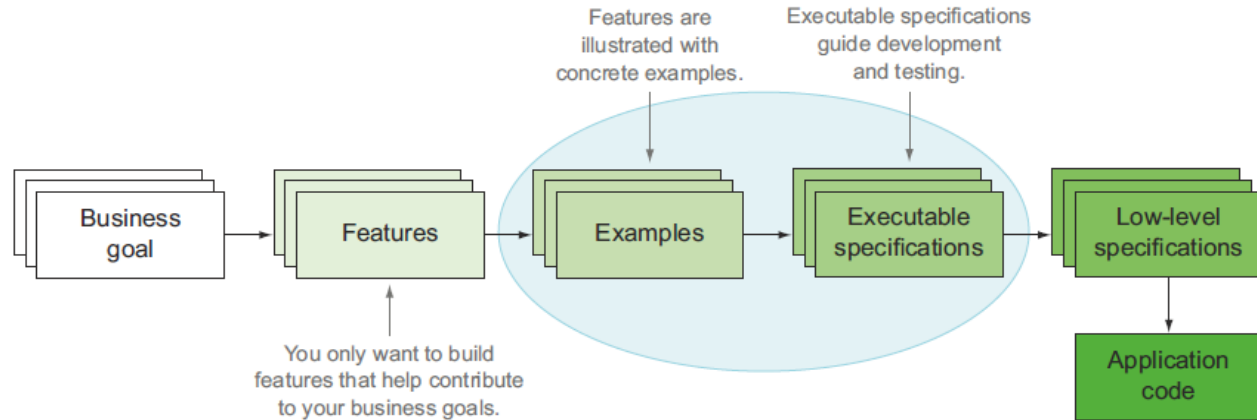
1. Negocio y tecnología tiene un modelo de referencia y lenguaje común.
2. Cualquier sistema debe ofrece un valor identificable y verificable al negocio.
3. El análisis, diseño y planificación preliminares tienen retornos decrecientes.

BDD ofrece ante todo un medio para reducir los problemas de comunicación por medio de un vocabulario reducido y común. Por ese motivo, BDD es una forma de construcción formal de especificaciones.

El uso de un lenguaje formal es lo que facilita la automatización a partir de BDD.

# Ejemplos y criterios de aceptación

A partir de una historia de usuario que define una funcionalidad, es preciso y aclarar y acotar su alcance para poder implementarla y validar esa implementación:



Esos criterios de aceptación emanan de las preguntas que hacemos para aclarar aspectos de la funcionalidad. Un mecanismo aún más potente es pensar en ejemplos de uso porque ayudan a:

- Entender mejor qué es lo queremos obtener desde el lado de cliente/negocio
- y extraer de ahí los criterios de aceptación

# Ejemplos de BDD (1)

Historia de Usuario

**Como** conductor

**Quiero** un sistema de control de velocidad

**Para** poder conducir más relajado sin riesgo de multas

Esta historia de usuario acepta múltiples criterios de aceptación. Por ejemplo

**Given** con el control de velocidad activado

**When** el conductor pisa el freno

**Then** el control de velocidad se desactiva

**Given** con el control de velocidad activado

**When** el conductor pisa el acelerador

**Then** el vehículo acelera hasta que deja de pisar el acelerador y vuelve a la velocidad fijada anteriormente.

# Ejemplos de BDD (2)

## Historia de Usuario

**Como** usuario del transporte público con tarjeta de transporte

**Quiero** poder pagar el viaje con mi tarjeta

**Para** despreocuparme de llevar cambio y hacer la operación más rápida

## Posibles criterios de aceptación:

**Given** la tarjeta es válida y tiene al menos un viaje cargado

**When** la tarjeta se acerca al lector

**Then** se descuenta un viaje y se muestra un mensaje de OK

**Given** la tarjeta es válida pero no viajes cargados

**When** la tarjeta se acerca al lector

**Then** se muestra alerta al viajero y al conductor de que no hay viajes y no se puede completar la operación

# Programación con BDD

Las definiciones BDD ayudan a construir comportamientos en código que se traducen en tests de niveles Integración y superiores. Al describir comportamientos, normalmente implican varios componentes dentro de un mismo sistema.

El estilo de programación con BDD es similar al que hemos visto en TDD: se construye un test, se desarrolla el código que permite pasarlo, se refactoriza y se pasa al siguiente tests.

Esos tests pueden construirse manualmente o con la ayuda de herramientas.

Dado que no siempre van a existir los componentes necesarios para validar el código, se suele acudir a la creación de mocks, bien manualmente, bien con la ayuda de herramientas (por ejemplo Mockito).



# Gherkin para BDD (1)

Gherkin es un lenguaje formal para la descripción de funcionalidades. Es un lenguaje “legible” de forma que además poder ser procesado por herramientas, permite ser usado como un medio de intercambio de información, especialmente entre Negocio y los equipos técnicos.

Gherkin se basa en el esquema que hemos visto, aunque agregando un grado adicional de formalidad:

In Gherkin  
use the  
Feature  
keyword  
to indicate  
a feature  
title.

1

Feature: Earning Frequent Flyer points from flights  
In order to encourage travellers to book with Flying High Airlines  
more frequently  
As the Flying High sales manager  
I want travellers to earn Frequent Flyer points when they fly with us

Scenario: Earning standard points from an Economy flight  
Given the flying distance between Sydney and Melbourne is 878 km  
And I am a standard Frequent Flyer member  
When I fly from Sydney to Melbourne  
Then I should earn 439 points

Scenario: Earning extra points in Business class  
Given the flying distance between Sydney and Melbourne is 878 km  
And I am a standard Frequent Flyer member  
When I fly from Sydney to Melbourne in Business class  
Then I should earn 878 points

2

A short  
description of  
the feature  
follows the title.

One or more  
scenarios  
follow.

# Gherkin para BDD (2)

Gherkin tiene versiones en los principales idiomas, pero usaremos el inglés como convenio. El lenguaje tiene cinco palabras reservadas para describir los escenarios:

- **Feature:** o nombre de la funcionalidad que vamos a describir. Se espera que tenga un título descriptivo, y es además donde se deja la definición de la historia de usuario: “**Como** [persona] **Quiero** [funcionalidad] para [valor o beneficio]”.
- **Scenario:** descripción del o de los escenarios que vamos a especificar y que luego se convertirán en escenarios de pruebas. Cada feature puede tener varios escenarios.
- **Given:** contexto en el que se desarrolla el escenario. Se traduce en prerequisites o acciones previas.
- **When:** acciones del escenario, lo que se traduce en el proceso de la prueba.
- **Then:** resultado esperado del escenario (y de la prueba).

And y But se usan también para enriquecer la descripción del escenario.

Como lenguaje formal, Gherkin puede ser interpretado por varias herramientas como Cucumber, JBehave, ... que lo convierte en un esqueleto del código de prueba. Luego tendremos que construir toda la lógica que del test y de la funcionalidad que lo satisface.

# Ejercicio: BDD

## Viaje en tren

Vamos a usar la notación Gherkin para construir una especificación formal de una funcionalidad. En este caso, la definición del ejercicio anterior para TDD:

Para los descuentos se establecen varios tramos (la edad es un valor entero):

- Menores de dos años viajan gratis
- Hasta 14 (menores de 15) hay un descuento del 30%
- Los mayores de 54 tienen un 30%
- Los mayores de 64 tienen un 60% de descuento.

Además hay una tarjeta de fidelización pero sólo para mayores de edad que aporta un 10% adicional de descuento en cada tramo.

Por otra parte, hay cuatro tipos de acomodaciones para los tres tipos de trenes (Mañana, Tarde y Noche):

- Asientos individuales
- Mesas, sólo para viajeros en grupos de 4
- Literas, sólo en trenes nocturnos
- Departamentos, sólo en para viajeros en grupos de 2 en trenes nocturnos

Hay que usar las palabras reservadas `Feature`, `Scenario`, `Given`, `When` y `Then`.

Recuerda que dentro de la descripción del escenario tienes que usar la notación de User Story (como, quiero, para).

# Ejercicio: BDD (1)

SOLUCIÓN

## Viaje en tren

Empezamos por los escenarios de los descuentos:

Feature: Descuentos en el billete de tren

Scenario: Descuentos por edad

Como cliente que quiere comprar un billete

Quiero tener descuentos

Para poder hacer viajes más económicos

Given Cuando voy a sacar un billete

When Si tengo menos de 2 años

Then Mi descuento es del 100%

Given Cuando voy a sacar un billete

When Si tengo entre 2 y 14 años

Then Mi descuento es del 30%

Given Cuando voy a sacar un billete

When Si tengo entre 55 y 64 años

Then Mi descuento es del 30%

# Ejercicio: BDD (2)

SOLUCIÓN

## Viaje en tren

Given Cuando voy a sacar un billete

When Si tengo 65 o más años

Then Mi descuento es del 60%

Scenario: Descuentos con tarjeta de fidelización

Given Cuando voy a sacar un billete

When Si tengo tarjeta de fidelización válida

Then Tengo un descuento adicional del 10%

Se supone que habrá otra Feature con un escenario donde se defina que una tarjeta válida requiere, entre otras cosas, que la edad debe ser al menos 18. Pero también se puede incluir:

Scenario: Descuentos con tarjeta de fidelización

Given Cuando voy a sacar un billete

When Si tengo tarjeta de fidelización válida y tengo más de 18 años

Then Tengo un descuento adicional del 10%

# Ejercicio: BDD (3)

SOLUCIÓN

## Viaje en tren

Segunda parte, la feature de tipo de asiento. También podría ser un escenario dentro de una feature más general. No hay una forma única de hacerlo.

Feature: Tipos de acomodación en el billete de tren

Scenario: Por horario de tren y número de viajeros

#Podría haber otros escenarios especiales (grupos, grandes, configuraciones, ...)

Como cliente que quiere comprar un billete

Quiero poder elegir el tipo de acomodación en función del horario y número de personas en mi grupo

Para tener el modo de viaje más adecuado a mis necesidades

Given Cuando voy a sacar un billete

When En cualquier horario y número de viajeros

Then Tengo el asiento individual como opción

Given Cuando voy a sacar un billete

When En cualquier horario y si pertenezco a un grupo de 4 viajeros

Then Tengo la mesa como opción

# Ejercicio: BDD (4)

SOLUCIÓN

## Viaje en tren

Given Cuando voy a sacar un billete

When En trenes de horario nocturno y cualquier de viajeros

Then Tengo la litera como opción

Given Cuando voy a sacar un billete

When En trenes de horario nocturno y grupo de 2 viajeros

Then Tengo el departamento como opción

# Ejercicio: BDD (5)

SOLUCIÓN

## Viaje en tren

Es posible usar tablas en los escenarios para simplificar su redacción:

Feature: Descuentos en el billete de tren

Scenario: Descuentos por edad

Como cliente que quiere comprar un billete

Quiero tener descuentos

Para poder hacer viajes más económicos

Given Cuando voy a sacar un billete

When Si introduzco mi edad

Then Se me ofrecen descuentos

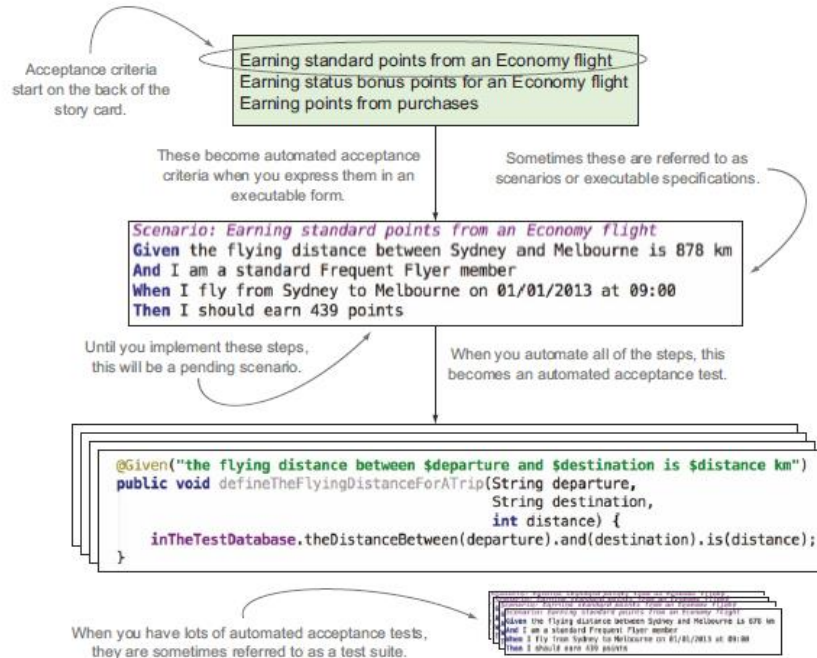
Examples:

Edad inicio	Edad fin	Descuento
0	1	100
2	14	30
55	64	30
65	150	60



# De Gherkin a código

Las distintas herramientas que procesan la notación Gherkin genera una serie de componentes que luego hay que rellenar. El modelo es similar al que conocemos de BDD.



Verificación del Software

## 4.3 Automatización de pruebas

# Beneficios de la automatización

Aunque sobre el papel pueda parecer que todos los tests deben ser automatizados, hay que examinar con detenimiento, lo que supone hacerlo.

En el lado positivo hay claros **beneficios**:

- Se **ahorra tiempo** reduciendo tareas repetitivas. Esto es especialmente en los llamados tests de regresión
- Se gana en **calidad**:
  - No hay posibles errores humanos. Una vez el test se ha automatizado correctamente, siempre se va a hacer igual, sin errores ni olvidos que den lugar a falsos positivos o a pasar por alto defectos
- Se gana en **consistencia**:
  - Todos los tests automatizados se ejecutan de la misma forma
- Se genera **información fiable**
  - Qué tests se han ejecutado, con qué resultado, bajo qué condiciones, ...

Además se pueden automatizar otras tareas relacionadas como la configuración del entorno, preparación de datos, establecimiento de pre y post condiciones.

Puede haber beneficio también en un modelo híbrido: manual y automatizado.

# Inconvenientes de la automatización

El principal inconveniente es el coste, que debe examinarse cuidadosamente antes de proceder a la automatización. Pero además, automatizar tests implica ciertos **riesgos**:

- **Mala estimación de esfuerzo y tiempo**, lo que puede comprometer la utilidad de las pruebas automatizadas (no tener suficiente retorno de inversión o no llegar a tiempo).
- **No contar con los costes de mantenimiento** y actualización de las pruebas automatizadas: deben evolucionar con el software, y además no pueden permanecer estáticas durante la fase de mantenimiento. Además, puede haber problemas con la responsabilidad sobre el código que automatiza las pruebas (si hay varios grupos implicados en la construcción, soporte y evolución).
- **No asegurar la trazabilidad y el control de versiones**, y acabar validando objetos o versiones incorrectas.
- **Interoperabilidad**: el testing automático tiene que comunicarse y sincronizarse con otros componentes.

Y no olvidar que **la automatización no puede hacer cualquier cosa**, las pruebas manuales son insustituibles en determinadas circunstancias.

# El coste de la automatización

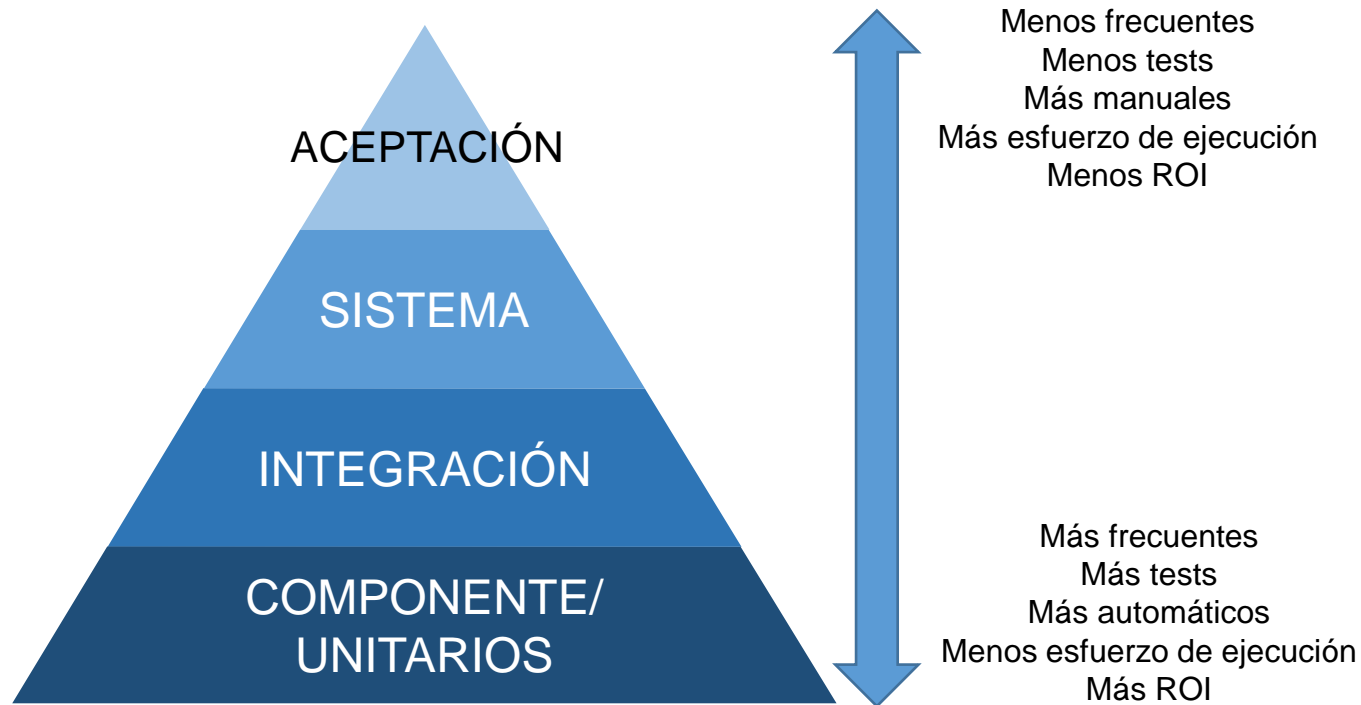
HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?  
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	1 DAY
	1 DAY					8 WEEKS	5 DAYS

@xkcd vía @madrillano

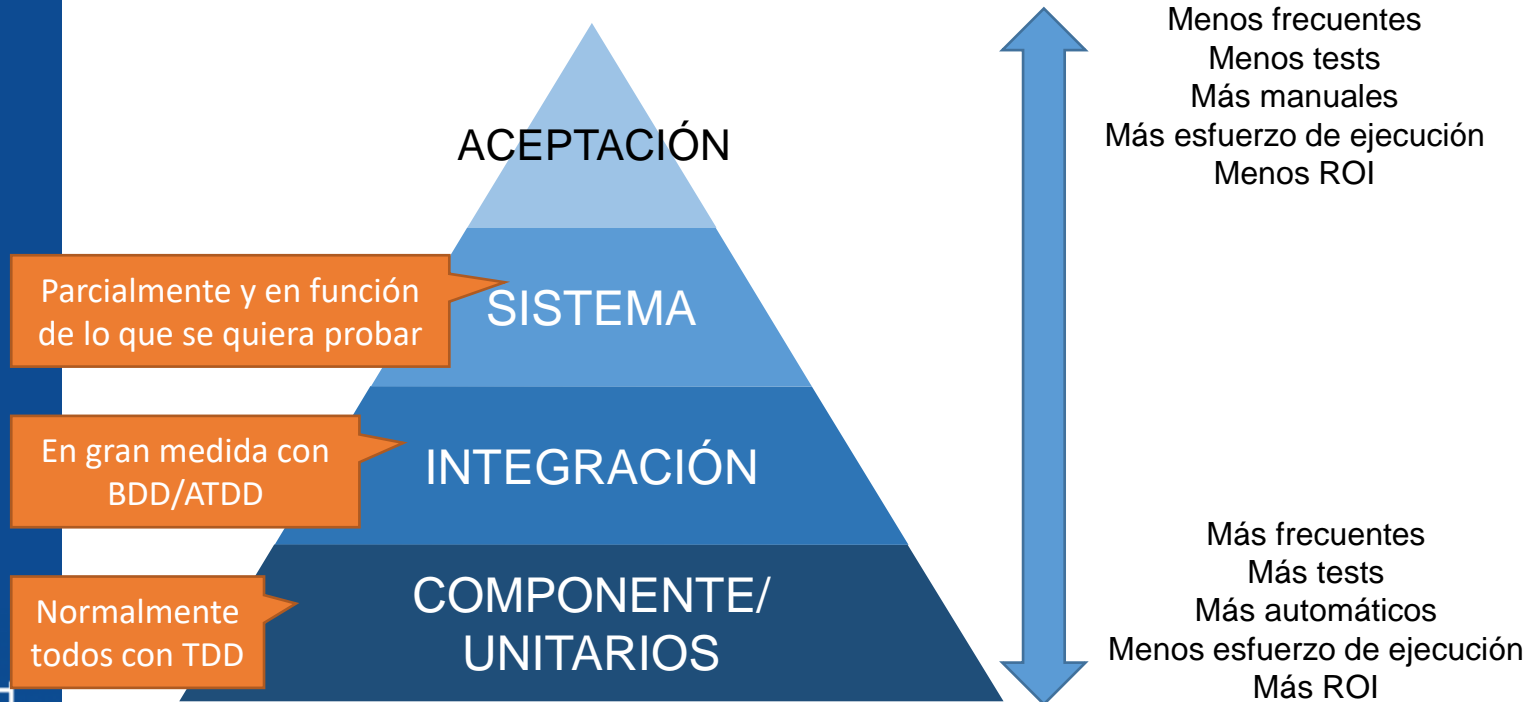
# ¿Qué pruebas automatizar? (1)

Recordemos la pirámide del testing:



# ¿Qué pruebas automatizar? (1)

Recordemos la pirámide del testing:



# ¿Qué pruebas automatizar? (2)

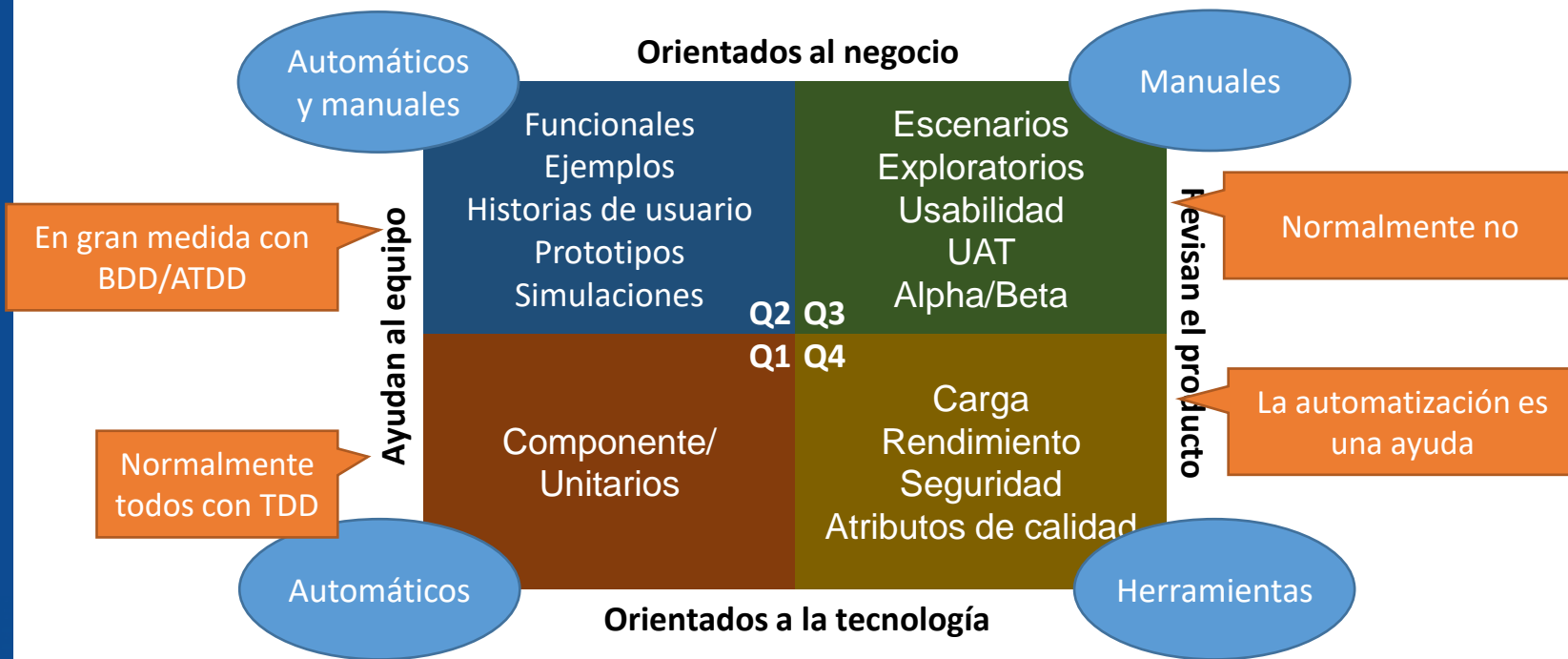
Recordemos el cuadrante del testing:





# ¿Qué pruebas automatizar? (2)

Recordemos el cuadrante del testing:



# ¿Qué pruebas automatizar? (3)

En resumen y por niveles:

- Las pruebas unitarias o de componente se pueden automatizar usando técnicas como TDD.
  - El esfuerzo de automatización sólo tiene sentido si el uso lo justifica. Por ejemplo aplicando XP u otros marcos ágiles con código sujeto a frecuentes cambios. En un modelo más lineal puede que no compense en esfuerzo.
- Pruebas de integración (BDD, ATDD).
  - Misma consideración que en el caso anterior: hay que analizar la posible volatilidad o estabilidad del código antes de invertir en la automatización.
- Pruebas de sistema.
  - No es infrecuente la automatización, pero al bajar la frecuencia de ejecución y aumentar el coste, el análisis de retorno de inversión es más importante.
- Pruebas de aceptación.
  - Por definición deberían ser manuales, pero en determinados casos pueden automatizarse determinadas partes.

En este apartado nos centraremos en medios para automatizar pruebas de integración y sistema, especialmente simulando escenarios complejos end-to-end.

# ¿Quién automatiza las pruebas?

- Unitarias:
  - Personas con perfil de desarrollo, aplicando las técnicas de testing que hemos visto hasta ahora.
- Integración:
  - Personas con perfil de desarrollo o testing (ambas tienen la capacidad de automatizar). Normalmente las personas especializadas en testing aportarán una visión más orientada a las pruebas y validación de escenarios.
- Sistema:
  - Normalmente personas con perfil de testing. La automatización en este caso es más ad-hoc y pensando en el end-to-end.

En ciertas organizaciones las personas especializadas en testing se encargan de crear escenarios, supervisar el testing automatizado y preocuparse de la correcta implementación de los procesos (QA). Hoy en día sigue siendo común la división entre personas especializadas en desarrollo y en calidad, aunque sus perfiles sean cada vez más parecidos.

# Automatizar el end-to-end (1)

Hay muchas plataformas, frameworks y aplicaciones que ayudan en la automatización de pruebas de integración y sistema, especialmente cuando hay que **simular la acción de un usuario**. Las pruebas de APIs, elementos de backoffice e infraestructura pueden llevarse a cabo con los mecanismos ya conocidos para TDD o BDD.

La mayoría de ellas se basa en la idea de **grabar y ejecutar**. Es decir: permiten monitorizar y almacenar las acciones de un usuario, y luego reproducirlas automáticamente. Si hay desviaciones entre los resultados esperados y los observados, se dispara un fallo. Este modelo tiene limitaciones, así que la gran mayoría tienen el complemento de APIs y conectores en varios lenguajes de programación que permiten modificar código generado automáticamente e introducir condiciones y acciones nuevas.

Para conseguir unas pruebas automatizadas efectivas, hay que recurrir a veces a introducir cambios en el software o usar elementos adicionales, que modificar el objeto que se está probando: esto puede falsear los resultados.

# Automatizar el end-to-end (2)

Ejemplos de sistemas para automatizar las pruebas del e2e:

- Selenium (web, aplicaciones móviles, rendimiento, escalado)
- Appium (aplicaciones nativas y móviles, especialmente iOS)
- Robot Framework (web, usa componentes de Selenium, se basa en Python)
- Serenity (para Java, orientada a BDD)
- TestProject.io (más orientada a testers que developers, facilita de reutilización)
- Sahi (web)
- Galen (web)
- WebDriverIO (tests escritos con Javascript)appi

# Automatizar el end-to-end (3)

Ejemplos de código para pruebas (de Robot Framework, que son especialmente simples):

```
*** Test Cases ***
```

```
TC1
```

```
    Open Browser https://www.TestURL.com/ chrome
```

```
    Maximize Browser Window
```

```
    Close Browser
```

Otro ejemplo:

```
Simple Smoke Test - Correct Answer
```

```
    [Tags] cloud
```

```
    Set Up And Open Android Application
```

```
    Input Name ${NAME}
```

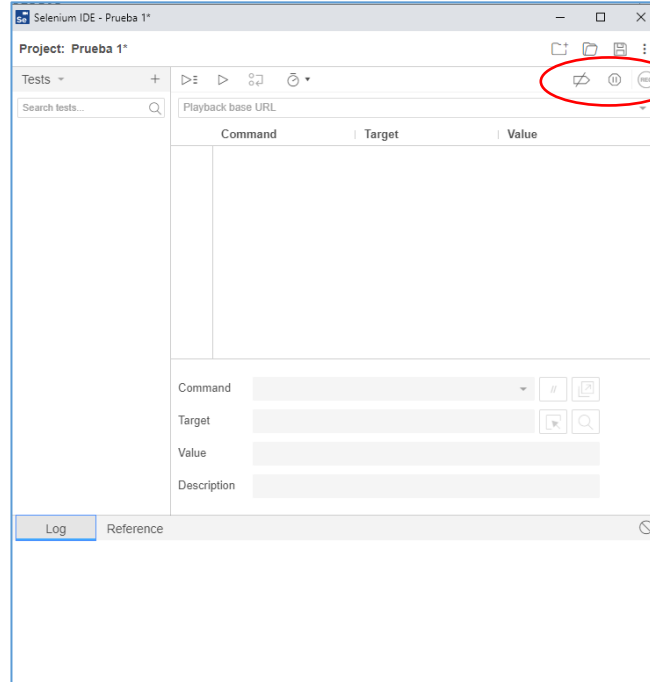
```
    Select Option Use Testdroid Cloud
```

```
    Submit Selection
```

```
    Validate Correct Answer
```

# Automatizar el end-to-end (4)

Vamos a ver cómo funciona Selenium en su función de grabar y reproducir. Selenium cuenta con componentes que se comportan como extensiones de Chrome o Firefox:



Comandos  
de grabación

# Automatizar el end-to-end (5)

Resultado de una grabación:

The screenshot displays the Selenium IDE interface with a project named 'Pruebas\*'. The URL bar shows 'https://www.comunidad.madrid/servicios/educacion/buscador-titulaciones-universitarias'. A table of recorded actions is visible, with a red circle highlighting the first seven rows. Below the table, the 'Log' tab is active, showing a list of events with a red circle highlighting the first six rows. The final status message at the bottom indicates the test was completed successfully.

Command	Target	Value
1 open	https://www.comunidad.m...	
2 set window size	1357x868	
3 run script	window.scrollTo(0,492)	
4 click	linkText=ACCEDER AL BUSCADOR	
5 click	id=dnn_ctr1403_ViewEITitulacion_ddlUniversidad	
6 select	id=dnn_ctr1403_ViewEITitulacion_ddlUniversidad	label=Centro Superior de Diseño y Arte Digital
7 click	id=dnn_ctr1403_ViewEITitulacion_ddlUniversidad	

Log	Reference	Time
13 click on id=dnn_ctr1403_ViewEITitulacion_btnBuscar OK		18:22:17
14 click on id=dnn_ctr1403_ViewEITitulacion_ddlRama OK		18:22:17
15 select on id=dnn_ctr1403_ViewEITitulacion_ddlRama with value label=Todas OK		18:22:18
16 click on id=dnn_ctr1403_ViewEITitulacion_ddlRama OK		18:22:18
17 click on id=dnn_ctr1403_ViewEITitulacion_btnBuscar OK		18:22:18
18 click on id=dnn_ctr1403_ViewEITitulacion_rptTitulaciones_ct009rptContenido OK		18:22:18

'Buscador de titulaciones' completed successfully 18:22:18

Acciones registradas

Log de la grabación



# Automatizar el end-to-end (6)

Código equivalente (Python):

```
def test_buscadordetitulaciones(self):
    self.driver.get("https://www.comunidad.madrid/servicios/educacion/buscador-titulaciones-
universitarias")
    self.driver.set_window_size(1357, 868)
    self.driver.execute_script("window.scrollTo(0,492)")
    self.driver.find_element(By.LINK_TEXT, "ACCEDE AL BUSCADOR").click()
    self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlUniversidad").click()
    dropdown = self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlUniversidad")
    dropdown.find_element(By.XPATH, "//option[. = 'Centro Superior de Diseño y Arte
Digital']").click()
    self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlUniversidad").click()
    self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlRama").click()
    dropdown = self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlRama")
    dropdown = self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlRama")
    dropdown.find_element(By.XPATH, "//option[. = 'Todas']").click()
    self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_ddlRama").click()
    self.driver.find_element(By.ID, "dnn_ctr1403_ViewEITitulacion_btnBuscar").click()
    self.driver.find_element(By.ID,
"dnn_ctr1403_ViewEITitulacion_rptTitulaciones_ctl00_hlContenido").click()
```

# Automatizar el end-to-end (7)

Código para pruebas creado manualmente:

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class PythonOrgSearch(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("http://www.python.org")
        self.assertIn("Python", driver.title)
        elem = driver.find_element_by_name("q")
        elem.send_keys("pycon")
        elem.send_keys(Keys.RETURN)
        assert "No results found." not in driver.page_source

    def tearDown(self):
        self.driver.close()

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

**Verificación del Software**

## **4.4 CI/CD y DevOps**

# Las pruebas en DevOps

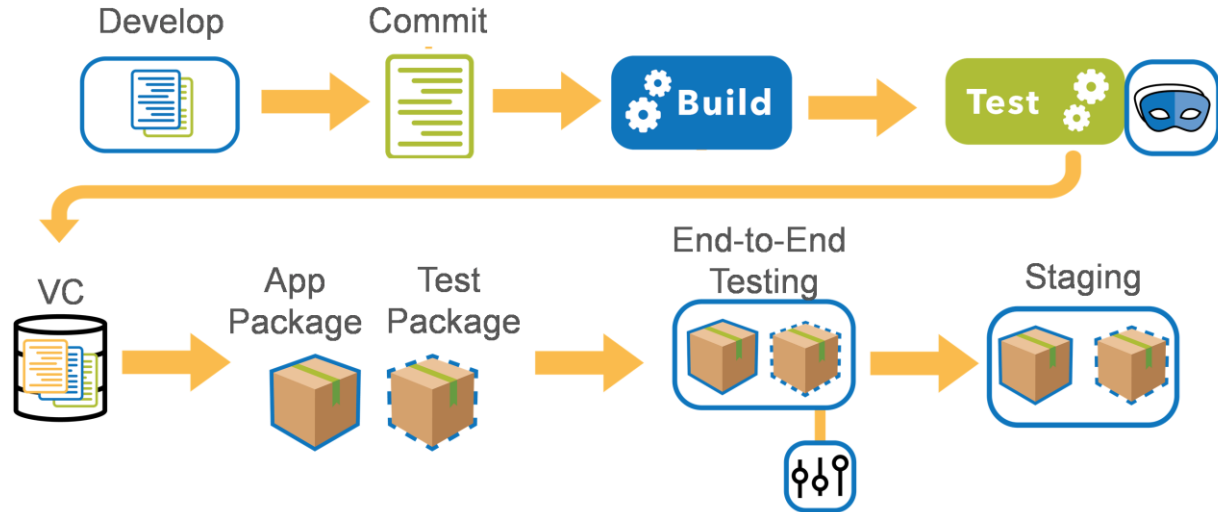
DevOps es sobre todo una filosofía, complementada con una serie de técnicas y prácticas. De sus principios, hay uno que nos afecta especialmente: “**Automate Everything You Can**“, y eso se traduce en acciones sobre dos prácticas principales de DevOps: Continuous Integration (CI) y Continuous Delivery (DC).

DevOps se basa en la construcción del código con marcos de trabajo ágiles, como Scrum con prácticas XP. Por ese motivo, muchas de las técnicas que hemos visto hasta ahora tienen cabida natural en DevOps.

Como el objetivo de DevOps es minimizar las fricciones entre Desarrollo y Operaciones y optimizar el flujo de entrega de valor, la automatización del testing se ha convertido en una de sus palancas y un elemento diferencial de DevOps.

# Continuous Integration

La integración continua dicta que el proceso de build se lanza con frecuencia, a ser posible tras cada commit y que los tests unitarios y el análisis estático de código se ejecuten como parte de la build. Los problemas se identifican rápidamente y resolver un *broken build* es la máxima prioridad de todo el equipo.



# Pruebas automáticas en CI/CD

Las pruebas automáticas, como parte del proceso de CI/CD, precisan que se cumplan determinadas condiciones:

- **Alineamiento de entornos**
- **Fiabilidad de datos**
- **Emulación de componentes externos**

# Qué pruebas automatizar en DevOps

Ya sabemos que no todas las pruebas se pueden automatizar. En la cadena de CI/CD podremos hacerlo con:

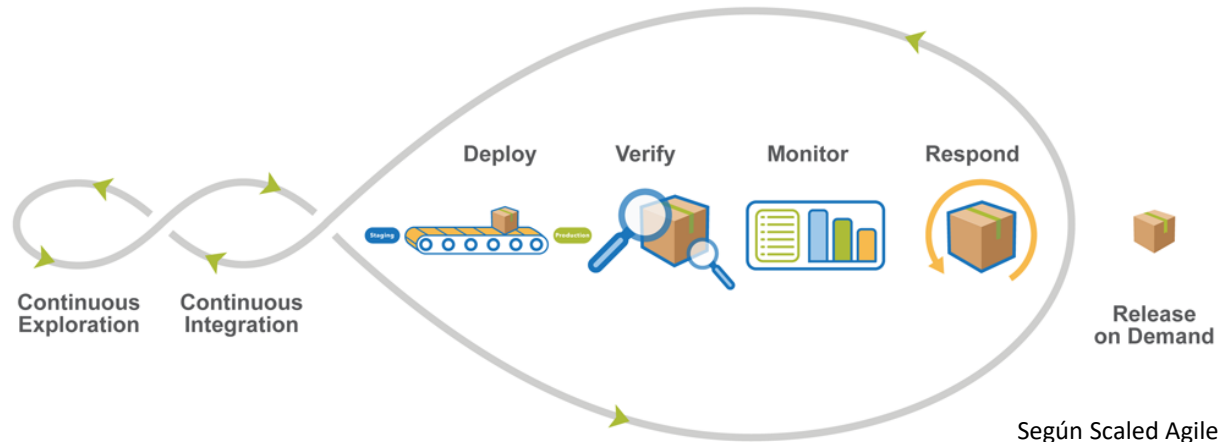
- Todos los tests unitarios, incluyendo el análisis estático del código.
- Buena parte de los de integración.
- Según la naturaleza del producto, una proporción variable de los de sistema.
- Determinados aspectos de los requisitos no funcionales (NFRs):
  - Rendimiento, consumo de recursos, escalabilidad
  - Seguridad
  - Accesibilidad y usabilidad

Por regla general, resolución de defectos, pequeñas mejoras, intervenciones puntuales ... permitirán un proceso altamente automatizado.

Nuevas funcionalidades, cambios sustanciales y en general intervenciones que modifiquen sustancialmente el producto o servicio, suelen tener una aproximación más conservadora que implica un testing con una componente más manual .

# Continuous Deployment

Es la tercera pata del proceso DevOps (junto a Continuous Exploration, y Continuous Integration) y se encarga de llevar el sw a su entornos destino. Incluso en el caso de que haya despliegue enteramente automatizado (no es frecuente) eso no implica que se tenga hacer una release también automatizada. Lo recomendable es dejar la llave de la activación en Negocio, en función de la naturaleza y el impacto de las funcionalidades desplegadas.



Según Scaled Agile Framework



# Estrategias de despliegue

Dependiendo del tipo de organización, la naturaleza del software y de los cambios, hay varias formas de abordarlo:

- **Big bang: TODO** el software se despliega y activa de una vez. Normalmente son procesos largos, complicados, con afectación al servicio, con altos riesgos y necesidad de estrategias de roll back en el caso de que algo salga mal
- **Blue/Green deployment:** similar al anterior, pero se cuenta con dos entornos completos. Se despliega en uno (Idle) y cuando se ha verificado que todo funciona correctamente, se hace el cambio a real (Live). No siempre es posible contar con dos entornos completos, y además puede no ser viable por el tipo de software (sincronización de BDs, ...).
- **Despliegue progresivo, release on demand:** las nuevas funcionalidades se van desplegando y conviven con las anteriores. La desactivación de unas y activación de otras se hace de manera controlada
- **Automatic deployment:** se despliega y activa inmediatamente. Sólo recomendable en determinadas circunstancias (cambios menores, bugs, ...)
- Otras estrategias: Alpha/Beta; Canary; Family & Friends; A/B testing, ...

# **Anexo. Información adicional**

# Selección de herramientas

En el mercado hay disponibles muchas herramientas y seleccionar las más apropiadas es un problema significativo para muchas organizaciones. En la selección influye:

- La naturaleza de la organización
- Las tecnologías que use
- El ciclo de vida que aplique (por ejemplo, si usa DevOps)
- La necesidad de especialización
- El tipo de licencia
- El soporte que requiera del proveedor
- Los requisitos de infraestructura, formación
- El coste, y sobre todo su relación con los beneficios esperados

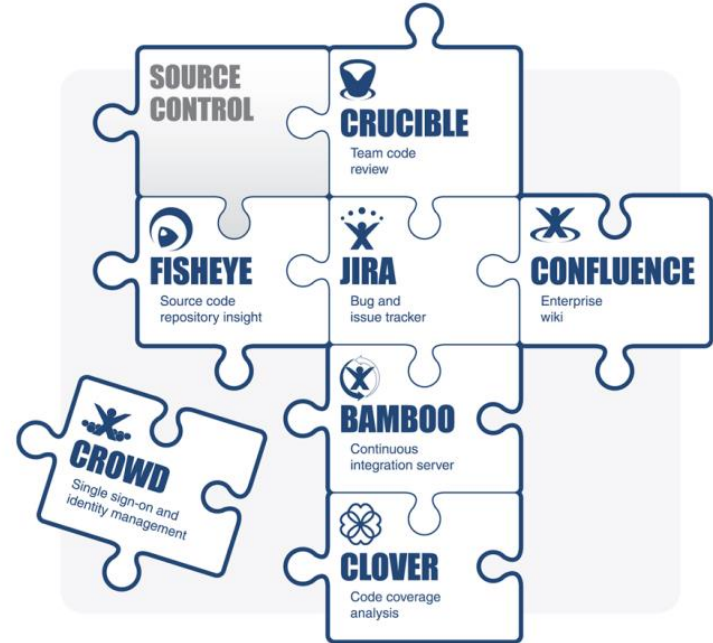
Estos procesos de selección son largos y tienen mucho impacto. Por ese motivo suelen llevar tiempo y el despliegue de las herramientas hacerse progresivamente.

# Herramienta todoterreno: JIRA (1)

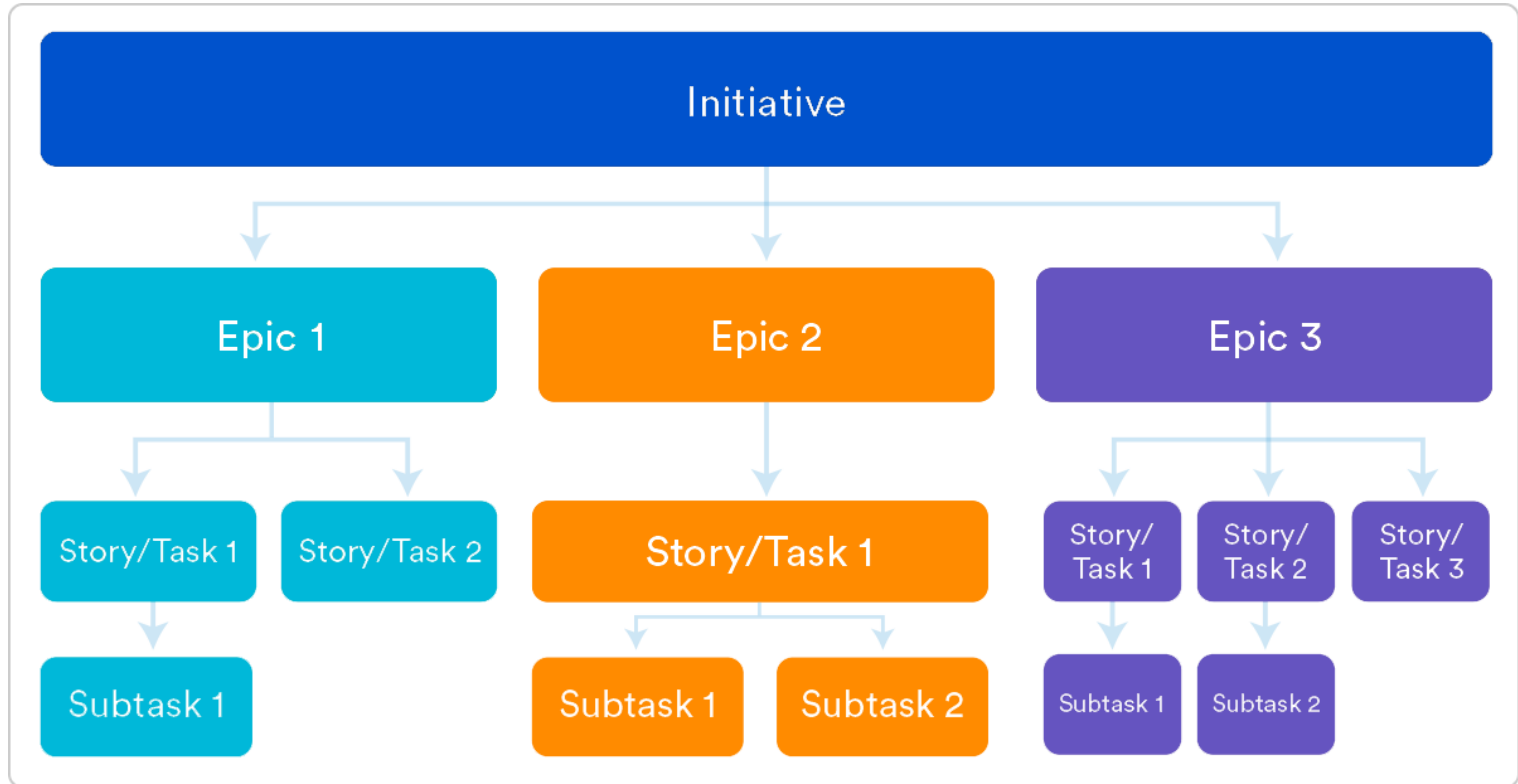
Aunque no es específicamente una herramienta para testing, se ha convertido en un estándar *de facto* que usan las empresas para todo tipo de tareas.

JIRA es una herramienta de tracking (seguimiento de tareas) de la compañía Atlassian Software con 3 potentes características:

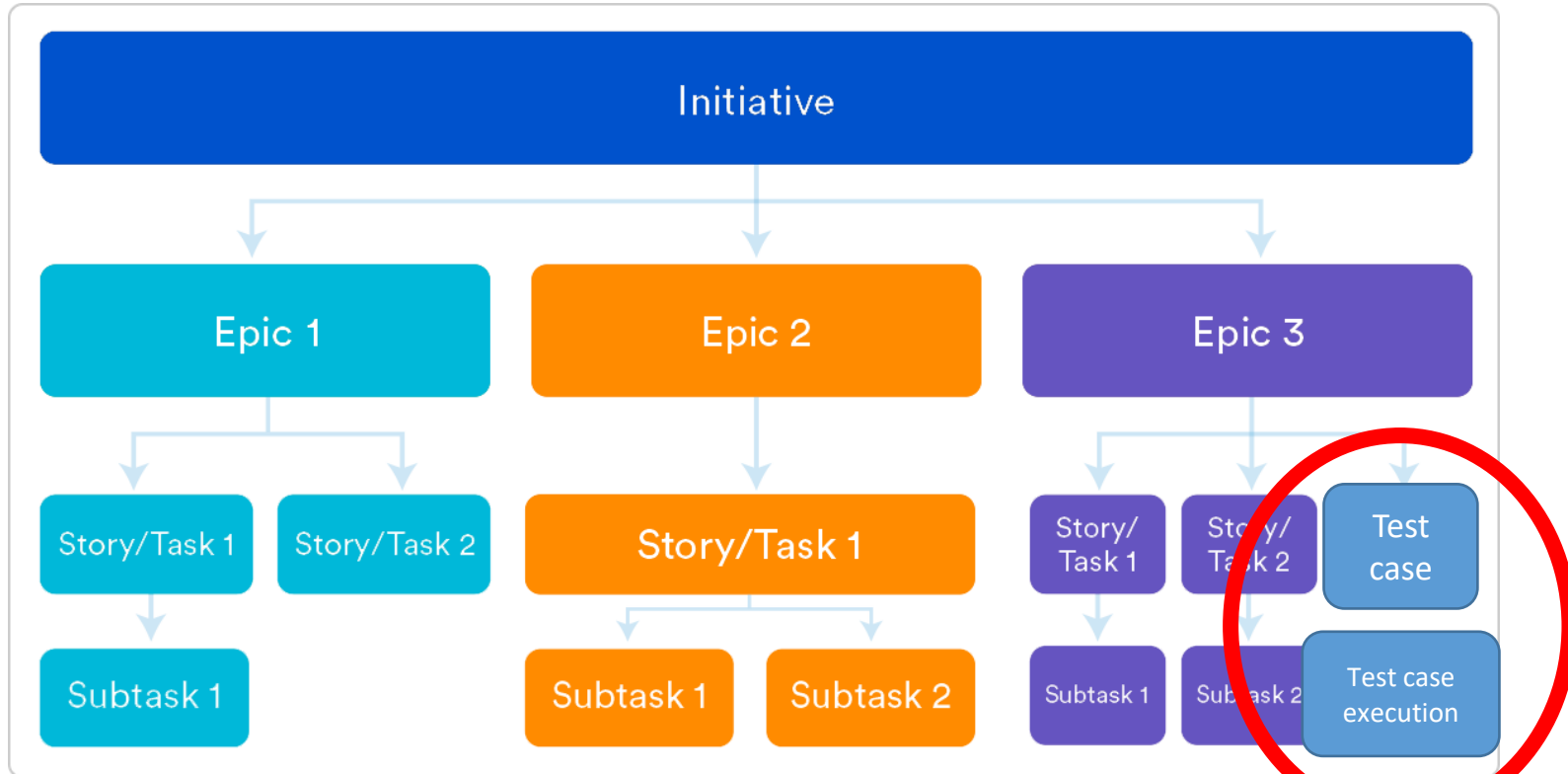
- Ligera
- Flexible
- Personalizable



# Herramienta todoterreno: JIRA (2)



# Herramienta todoterreno: JIRA (3)



# Herramienta todoterreno: JIRA (4)

Pantalla principal de proyecto en JIRA:

The screenshot displays the JIRA project main screen for 'SP Sprint 1'. The interface includes a top navigation bar with 'Jira Software', 'Your work', and various dropdown menus. A left sidebar lists project navigation options like 'Scrum Project', 'SP board', 'Backlog', 'Active sprints', 'Reports', 'Issues', 'Components', 'Code', 'Releases', 'Project pages', 'Add item', and 'Project settings'. The main content area shows the 'SP Sprint 1' board with three columns: 'TO DO', 'IN PROGRESS', and 'DONE'. Each column contains a task card with a title, a progress bar, and a status icon. The 'IN PROGRESS' column is highlighted in blue.

Projects / Scrum Project / SP board

## SP Sprint 1

9 days remaining Complete sprint

Only My Issues Recently Updated

TO DO	IN PROGRESS	DONE
<p>Primera tareas</p> <p>SP-1</p>	<p>Segunda tarea</p> <p>SP-2</p>	<p>Tercera tarea</p> <p>SP-3</p>

# Herramienta todoterreno: JIRA (5)

Pantalla principal de proyecto en JIRA:

The screenshot displays the JIRA project main screen. On the left, a sidebar contains navigation links: Scrum Project, SP board, Backlog, Active sprints, Reports, Issues, Components, Code, Releases, Project pages, Add item, and Project settings. The main area shows 'SP Sprint 1' with a search bar, filters for 'Only My Issues' and 'Recently Updated', and a Kanban board with three columns: TO DO, IN PROGRESS, and DONE. Each column contains a task card with a status icon, a title, and a sprint ID (SP-1, SP-2, SP-3). Annotations include blue callout boxes: 'Búsqueda' (Search) pointing to the search bar, 'Auxiliar' (Auxiliary) pointing to the top right utility icons, 'Vistas y elementos' (Views and elements) pointing to the left sidebar, and 'Tablero o pizarra' (Board or whiteboard) pointing to the Kanban board.

**Búsqueda**

**Auxiliar**

**Vistas y elementos**

**Tablero o pizarra**



# Herramienta todoterreno: JIRA (6)

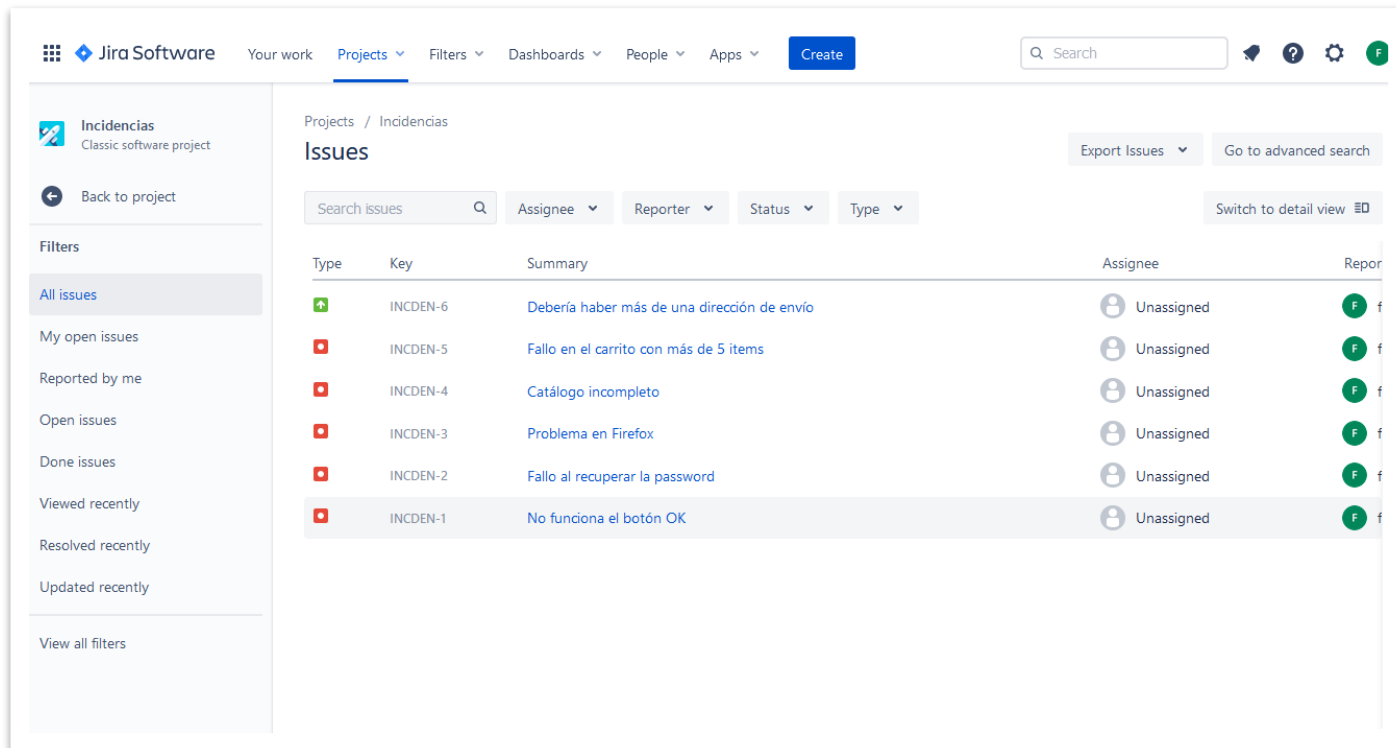
JIRA es altamente **configurable**.

Se puede:

- Enriquecer el workflow y agregar nuevos pasos y transiciones
- Agregar nuevos tipos de issues
- Cambiar los campos de cada elemento
- Modificar la visualización de elementos y tableros
- Crear etiquetas y realizar agrupaciones y tratamientos en función de ellas y de otros elementos
- Crear dashboards para visualizar información relevante
- Guardar y compartir configuraciones, búsquedas, tableros, ...
- Incorporar extensiones y plugins
- Programar scripts y automatismos

# Herramienta todoterreno: JIRA (7)

## Otras vistas de JIRA: Issues



The screenshot shows the JIRA interface for the 'Incidencias' project. The top navigation bar includes 'Jira Software', 'Your work', and various dropdown menus. The left sidebar shows the project name and a list of filters. The main area displays a table of issues with columns for Type, Key, Summary, Assignee, and Reporter. The issues are listed in descending order of their key.

**Filters**

- All issues
- My open issues
- Reported by me
- Open issues
- Done issues
- Viewed recently
- Resolved recently
- Updated recently
- View all filters

**Issues Table**

Type	Key	Summary	Assignee	Reporter
+	INCEN-6	Debería haber más de una dirección de envío	Unassigned	F f
+	INCEN-5	Fallo en el carrito con más de 5 items	Unassigned	F f
+	INCEN-4	Catálogo incompleto	Unassigned	F f
+	INCEN-3	Problema en Firefox	Unassigned	F f
+	INCEN-2	Fallo al recuperar la password	Unassigned	F f
+	INCEN-1	No funciona el botón OK	Unassigned	F f

# Programación y contabilidad

La Programación: produce enormes documentos usando una simbología técnica e incomprensible para la gente no familiarizada. Cada símbolo del documento debe ser correcto. Un solo error puede dar lugar a la pérdida de dinero y vidas.

La Contabilidad: produce enormes documentos usando una simbología técnica e incomprensible para la gente no familiarizada. Cada símbolo del documento debe ser correcto. Un solo error puede dar lugar a la pérdida de dinero e incluso vidas.

¿Cómo se consigue evitar los errores en Contabilidad? Con una técnica nacida hace mil años llamada “Partida Doble”:

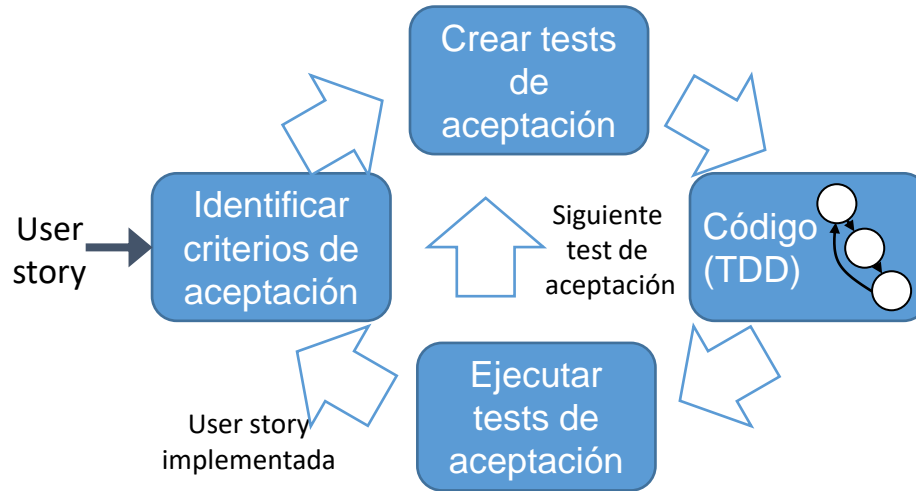
*El **sistema de partida doble** es el método o sistema de registro de las operaciones más usado en la contabilidad. Cada operación se registra dos veces, una en el debe y la otra en el haber, con el fin de establecer una conexión entre los diversos elementos patrimoniales. La anotación que involucra las dos partidas (debe y haber) se denomina asiento contable. Este sistema se asemeja a una **balanza en equilibrio**, ya que, dentro de un asiento contable, la suma de los conceptos del debe y del haber siempre tienen que ser iguales.*

El sistema contable se basa en introducir progresivamente y uno a uno cada asiento, en vez de hacerlo con todos en bloque. Incluso hay una obligación legal de hacerlo así.

# Ciclo de trabajo ATDD

El de BDD sería muy similar.

Aquí se muestra como un complemento de TDD y funciona de forma similar. Como se ve, trabaja a un nivel superior (US o Historia de Usuario). El punto de partida es la identificación de los criterios de aceptación que serán los que definen la prueba.



# Pruebas automáticas en CI/CD

Las pruebas automáticas, como parte del proceso de CI/CD, precisan que se cumplan determinadas condiciones:

- Alineamiento de entornos:
  - De nada sirve que los entornos sobre los que se pruebe no repliquen a los de producción.
  - La definición de entornos debe estar bajo control de versiones e incorporar mecanismos que faciliten su replicación.
  - Por ese motivo que se recurre habitualmente a la virtualización.
- Fiabilidad de datos
  - Los datos para las pruebas tienen su propia gestión para garantizar consistencia y fiabilidad de las pruebas.
  - Es necesario que los datos de prueba emulen los de producción cumpliendo las restricciones (generalmente legales) necesarias.
- Emulación de componentes externos
  - No siempre es posible contar con todos los sistemas y componentes necesarios para la prueba. Tener réplicas fieles, gestionadas adecuadamente (control de versiones) es básico para ganar confianza en las pruebas.

# Herramientas para DevOps

Hay multitud de herramientas para DevOps, que cubren todos sus posibles aspectos. Algunas son muy especializadas, otras muy generales, otras integran a su vez a otras herramientas. De hecho, al cubrir DevOps toda la cadena de procesos, desde la ideación hasta la operación, el catálogo es excesivamente amplio.

Si nos fijamos específicamente en las relacionadas con el testing, tendríamos (algunas cubren varias funciones):

- Herramientas propiamente dicho para automatizar y lanzar tests, medir performance, análisis de código, ... (JUnit, Selenium, JMeter, Sonarqube ...).
- Herramientas para gestión de la configuración (Git, Puppet, Ansible, Bitbucket, SolarWinds, ...).
- Integración continua –Build- (Jenkins, Bamboo, ...)
- Despliegue continuo (Spinnaker, Jenkins, Octopus, Bamboo, ...)
- Monitorización, registro (Nagios, Zabbix, SolarWinds, ...)