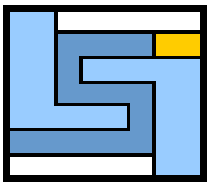
Diseño y Pruebas II



WIS TESTING REPORT

- ## Cover

Group: E7-06
Repository: https://github.com/plss12/Acme-Toolkits
Colleagues:

1. Cabezas Villalba, Juan Pablo (juacabvil@alum.us.es)
2. Martínez Jaén, Javier (javmarjae@alum.us.es)
3. Moreno Calderón, Álvaro (alvmorcal1@alum.us.es)
4. Navarro Rodríguez, Julio (julnavrod@alum.us.es)
5. Parejo Ramos, Salvador (salparram@alum.us.es)
6. Soto Santos, Pedro Luis (pedsotsan@alum.us.es)

Date: 27/02/2022

- **Revision table**

Version	Date	Description
1.0	28/02/2022	First Revision

- **Table of contents**

- Introduction

Las pruebas de un sistema o “testing” hacen referencia al conjunto de métodos y operaciones aplicados a dicho sistema con el fin de examinar la calidad del mismo y la cantidad de fallos que posee. Esto se hace con el fin de poder corregir errores difíciles de percibir a simple vista.

Normalmente, las pruebas consisten en ejecutar parte de nuestro código y comprobar que el sistema se comporta de manera esperada. Para ello, diseñaremos pruebas que nos permitan analizar el comportamiento del sistema a la hora de manejar datos, peticiones, excepciones, etc.

Sin embargo, las pruebas no nos aseguran que nuestro sistema esté libre de fallos, pero sí que nos permiten encontrar errores importantes que pueden causar una mala impresión a los usuarios o un mal desempeño del sistema una vez implantado.

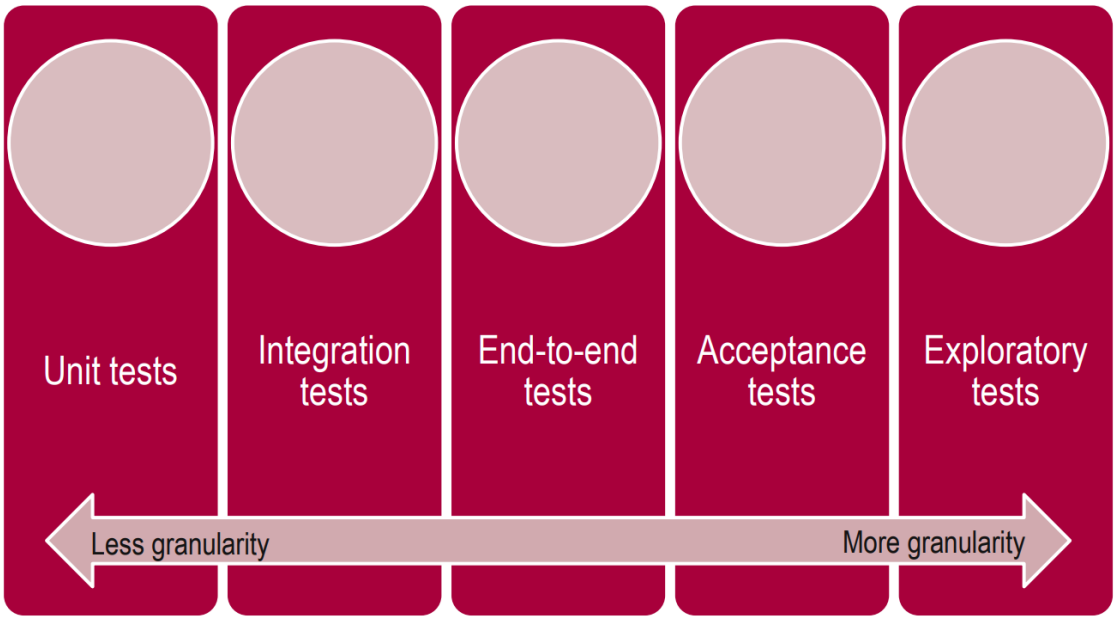
- Main concepts

1. **Verificación:** comprobar que el sistema cumple con los requisitos indicados en la fase de especificación de requisitos.
2. **Validación:** comprobar que el sistema cumple con lo que el cliente necesita.
3. **Subject Under Test (SUT):** parte del software a probar.
4. **Test Suite:** conjunto de pruebas a realizar sobre un sistema.
5. **Fixture:** conjunto de datos o parámetros previamente planeados que se van a emplear en una prueba.

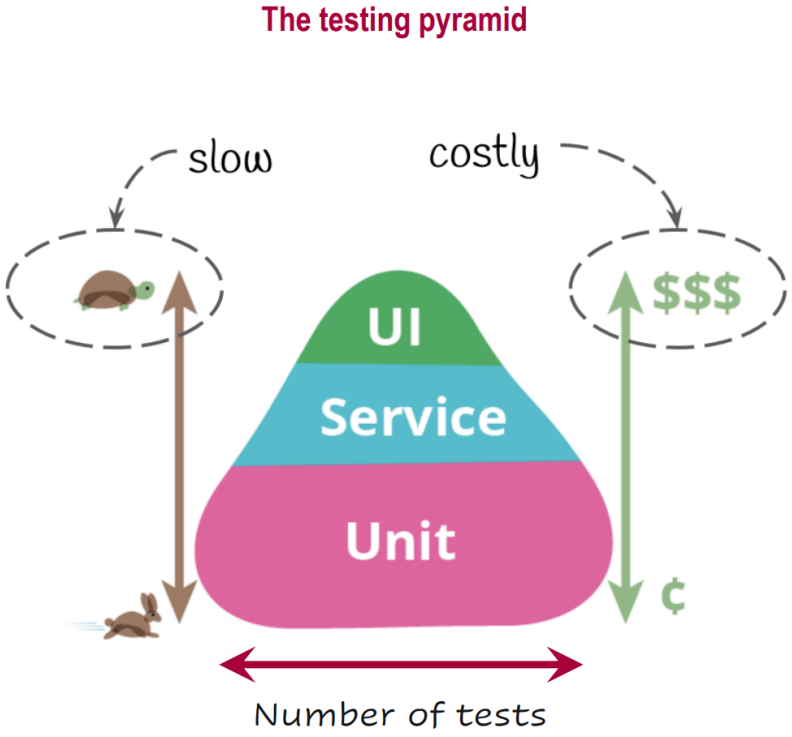
- Granularity

Existen diversos tipos de test según la cantidad de código y componentes del sistema que se encargan de probar. A esta cantidad se le denomina granularidad, y hasta

ahora, nos hemos centrado principalmente en las pruebas de menor granularidad: las pruebas unitarias o unit tests.



Esquema de granularidad. DP1, Tema 1



The Testing Pyramid. DP1, Tema 7

En la imagen anterior se describe el coste y el tiempo de ejecución de cada test según el elemento que se está testeando, así como el número de tests de cada elemento que se incluirá en el test suite. Podemos ver como los test unitarios son rápidos, baratos y numerosos ya que son la base de la pirámide, y por tanto son fundamentales para el test suite.

- ## Unit Tests

En las pruebas unitarias se examinará el comportamiento de las distintas unidades del sistema. No existe una definición concreta de unidad ya que para cada sistema, una unidad representará algo diferente. En nuestro caso, las unidades son los servicios de las entidades del modelo de dominio y los validadores.

El testing de una aplicación no puede ser exhaustivo, ya que sería imposible definir un test suite con todas las situaciones y datos posibles. Las pruebas unitarias deben ejecutarse rápidamente y su estructura de divide en tres partes:

1. Arrange: definir el conjunto de datos (fixture) que se va a utilizar en la prueba.
2. Act: ejecutar la prueba.
3. Assert: comparar los resultados obtenidos con los esperados.

Cada prueba se suele dividir en dos casos posibles: positivo y negativo.

En los casos positivos se introducen datos válidos y se espera que la unidad se comporte correctamente y no ocurra ningún error.

Por otra parte, en los casos negativos se introducen datos no válidos y se comprueba que se genera un error y que el sistema lo maneja correctamente.

Las pruebas unitarias tienen una serie de características principales conocidas como FIRST:

1. **Fast:** son rápidas de ejecutar.
2. **Independent:** no dependen de otras pruebas.
3. **Repeatable:** el comportamiento no depende de factores externos como la fecha del momento de ejecución.

4. **Self-checking:** la misma prueba es capaz de dictaminar su resultado.
5. **Timely:** las pruebas deben crearse o actualizarse al mismo tiempo que se crea o modifica el código.

- Covering Strategies

La cobertura de un test suite es la cantidad de código que es testeado. Como se dijo anteriormente, el testeo no puede ser exhaustivo, pero existen varias estrategias para que la cobertura sea lo más óptima posible. Estas estrategias se emplean pensando en la estructura de los métodos de las clases a probar, teniendo en cuenta:

1. **Condicionales:** si en el SUT existen uno o varios condicionales, lo recomendable es que existan casos de prueba en los que se observa el comportamiento del sistema para cada rama del condicional. Aplica para los if y los switch de los posibles métodos de ciertas clases.
2. **Bucles:** es recomendable que cada bucle se ejecute ninguna, una y varias veces para analizar el comportamiento en cada caso.
3. **Rangos:** para ciertos tipos primitivos en Java, como los int o los double, existe un rango de valores posibles que puede tomar una variable. La idea es introducir datos por debajo, dentro y por encima del rango permitido para comprobar el resultado.
4. **Colecciones:** para tipos de datos como listas o arrays, es bueno tener casos de pruebas en los que las colecciones tienen cero, uno y varios elementos.

- Best Practices

1. **Parametrizar pruebas:** permite realizar el mismo test para la misma unidad con valores distintos, sin necesidad de crear un test nuevo para cada caso de prueba.
2. **Fluent Assertions:** proporcionan mensajes de error muy detallados.
3. **Mantener la cohesión:** realizar un caso de prueba para cada escenario posible, en lugar de uno solo para varios escenarios. De esta manera queda un código mucho más legible e intuitivo.
4. **Principios DRY y DAMP:** es mejor reutilizar código mediante la creación de métodos con operaciones recurrentes que copiar y pegar dichas operaciones cada vez que lo necesitemos. Además, es buena idea comentar el código con frases cortas pero claras para que el test sea más inteligible.

- Test Doubles

Normalmente, los test no se realizan sobre los auténticos objetos del sistema, sino que se generan dobles sobre los que trabajamos. Los dobles nos permiten realizar todas las pruebas que queramos con un SUT determinado sin dañarlo a él o al resto del sistema.

Según la interacción de un SUT con otras unidades del sistema, un test puede ser sociable o solitario. En los test sociables se asume que los colaboradores (otras unidades) funcionan correctamente y no se utilizan dobles a no ser que los colaboradores requieran recursos externos. Por el contrario, en los solitarios siempre se generan dobles del SUT perfectamente aislados para evitar daños colaterales en caso de errores.

Existen diversos tipos de dobles, entre ellos:

1. **Stubs:** no incorporan ninguna lógica. Se usan cuando necesitamos que un colaborador devuelva un determinado valor para que el SUT pase a un estado en concreto. Para determinar si la prueba es superada, evaluamos el estado del SUT.
2. **Mocks:** se usan para analizar las interacciones entre objetos del sistema. Son útiles cuando no existe un cambio de estado de nuestro SUT. Comprobamos el comportamiento del SUT para decidir si pasa la prueba o no.
3. **Fakes:** implementaciones a pequeña escala de APIs aún no en producción. Son usados cuando la auténtica implementación no es posible, ya sea porque es muy lenta o requiera acceso a la red.

- Bibliography

- Tema 1 “Introduction to Software Design & Testing”.
- Tema 7 “Introduction to Information Systems Testing”.
- Tema 11 “Intermediate Information Systems Testing”.