UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

PARTE 1 DE 3: CONTROL DE CALIDAD Y PRUEBAS

v1.0 19.11.20

ceedcv
CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Entornos de desarrollo (ED)

Sergio Badal
Carlos Espinosa
c.espinosamoreno@edu.gva.es
Extraído de los apuntes de:
Cristina Álvarez Villanueva; Fco. Javier Valero Garzón; M.ª Carmen Safont

UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

- **Pasos a seguir**

1)Lee la documentación (PDF)
2)Instala el software necesario (sigue los pasos)
3)Realiza los TESTS todas las veces que quieras
4)Acude al FORO DE LA UNIDAD
- Para cualquier duda sobre esta unidad

5)Acude al FORO DEL MÓDULO

- Para cualquier duda sobre el módulo

UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

- **¿Qué veremos en esta UNIDAD?**

– **SEMANA 1**

- PARTE 1 DE 3: **CONTROL DE CALIDAD Y PRUEBAS**
- PARTE 2 DE 3: **JUNIT CON ECLIPSE**

– PRÁCTICA NO **EVALUABLE**

– **SEMANA 2**

- PARTE 3 DE 3: **JUNIT CON NETBEANS**

– PRÁCTICA NO **EVALUABLE**

UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

4.1 CONTROL DE CALIDAD Y PRUEBAS

4.4.1 CALIDAD DEL SOFTWARE

4.4.2 PRINCIPIOS BÁSICOS DE LAS PRUEBAS

4.4.3 PRUEBAS DE CAJA NEGRA VS CAJA BLANCA

4.4.4 PRUEBAS UNITARIAS VS DE INTEGRACIÓN

4.4.1 CALIDAD DEL SOFTWARE

- Cuando se desarrolla software, resulta recomendable comprobar que el código que hemos escrito funciona correctamente.
  - Para ello, implementamos pruebas que verifican que nuestro programa genera los resultados que de el esperamos.

- Hasta aquí todo no hay nada nuevo en el horizonte.
  - **Hasta un niño de 8 años sabe que las cosas hay que probarlas antes de decir si te gustan o no.**

4.4.1 CALIDAD DEL SOFTWARE

Pues bien, **desarrollar software de calidad** es mucho más que “probar si funciona”.

Como adelanto, podríamos proponer esto:

1)Promover ciclos de vida ágiles [UD10 de ED; Abril]

2)Realizar un buen análisis y VALIDARLO [UD01 de ED; Octubre]

3)Realizar una buena codificación y VERIFICARLA

Generar “código limpio”

Seguir principios como: SOLID, KISS, CLEAN … (ver enlace)

4)Refactorizar si es necesario [UD06 de ED; Diciembre]

Aplicar “chapa y pintura” a nuestro código

5)Realizar pruebas manuales y automáticas [¡Esta UD de ED!]

De manera transversal a todo el ciclo de vida

6)Retroalimentación constante [UD10 de ED; Abril]

Con el cliente y con el equipo de desarrollo

4.4.1 CALIDAD DEL SOFTWARE

1. MARCO CONTEXTUAL DE LA REFACTORIZACIÓN

1.1. DEFINICIÓN

Programar código no únicamente es buscar funcionalidad, sino también limpieza y elegancia de escritura. Muchas veces se programa comprobando resultados, sin importar si el código está enmarañado. Esto tiene varias problemáticas: es más difícil hacer una actualización o corrección, pasarlo a otra persona implica que ha de perder tiempo intentado leerlo, y al cliente no se debe presentar con este aspecto.

Así, aunque debería irse programando ya con una estructura inicial y buscando la limpieza de código, una vez terminado siempre se pasa a la fase de **Refactorización**. En ella le damos "pintura y chapa" a nuestro código. Es decir, **cambiamos la estructura interna** de nuestro software para hacerlo más fácil de comprender, de modificar, más limpio y sin cambiar su comportamiento observable.

Por tanto, **refactorizar es *mejorar el código fuente sin cambiar el resultado*** *del programa* (Aldarias, 2012). Nuestro objetivo es mantener el código sencillo y bien estructurado.

1.2. ¿POR QUÉ Y CUÁNDO REFACTORIZAR?

Conforme se modifica, el software pierde su estructura. Así, refactorizamos por varios motivos:

- Para **eliminar código duplicado** simplificar su mantenimiento.
- Para hacerlo **más fácil de entender** (por ejemplo la legibilidad del código facilita su mantenimiento)
- Para **encontrar errores** (por ejemplo al reorganizar un programa, se pueden apreciar con mayor facilidad las suposiciones que hayamos podido hacer)
- Para **programar más rápido**: al mejorar el diseño del código, mejorar su legibilidad y reducir los errores que se cometen al programar, se mejora la productividad de los programadores.

4.4.1 CALIDAD DEL SOFTWARE

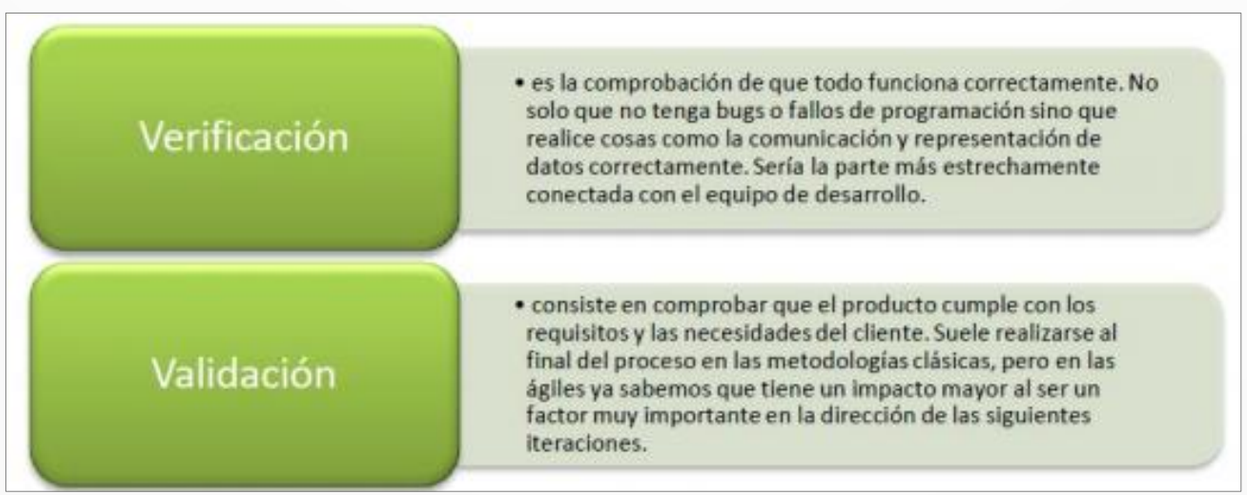
- Tan importante es VALIDAR como VERIFICAR y es crucial tener clara la diferencia entre ambos conceptos.

–¿Qué **VALIDAMOS**?

- El producto/resultado de la fase de análisis: Los requisitos del sistema

–¿Qué **VERIFICAMOS**?

- El producto/resultado de la fase de implementación: El código/sistema



4.4.1 CALIDAD DEL SOFTWARE

- Un factor fundamental para comprender todo el resto del tema es saber que las pruebas se deben diseñar después de la fase de implementación pero, también, **antes de empezar a programar nada** o, al menos, **tras crear los primeros módulos o la estructura del programa.**
  - En la frontera entre diseño e implementación

- Para ello se escriben **casos de pruebas unitarios** compuestos por varios datos pero principalmente con los factores, datos o pasos a seguir para realizar la prueba y el resultado esperado. Tras realizar la prueba se registra el resultado real.
  - Si una prueba se ejecuta con resultado insatisfactorio, debemos buscar el germen del error en la primera fase que aparezca (por ejemplo análisis o diseño) y empezar a solucionarlo desde ahí.

- El objetivo final es que el cliente esté satisfecho con el producto que ha adquirido, pero ello implica pruebas a muchos niveles, desde **pruebas unitarias de pequeñas partes de código** (por ejemplo un método o una clase) hasta pruebas del producto final desde el punto de vista del usuario, pasando por **pruebas de integración de diferentes componentes.**

UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

4.1 CONTROL DE CALIDAD Y PRUEBAS

4.4.1 CALIDAD DEL SOFTWARE

4.4.2 PRINCIPIOS BÁSICOS DE LAS PRUEBAS

4.4.3 PRUEBAS DE CAJA NEGRA VS CAJA BLANCA

4.4.4 PRUEBAS UNITARIAS VS DE INTEGRACIÓN

4.4.2 PRINCIPIOS BÁSICOS DE LAS PR

- Cualquiera que sea la técnica o **conjunto de técnicas que utilicemos para asegurar la calidad de nuestro software**, existen un conjunto de principios que debemos tener siempre presentes.
- A continuación enumeramos los principales:

1) Es imperativo disponer de unos requisitos que detallen el sistema

2) Los procesos de calidad deben ser integrados en las primeras fases del proyecto

3) Quien desarrolle un sistema no debe ser quien prueba su funcionalidad

UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

4.1 CONTROL DE CALIDAD Y PRUEBAS

4.4.1 CALIDAD DEL SOFTWARE

4.4.2 PRINCIPIOS BÁSICOS DE LAS PRUEBAS

4.4.3 PRUEBAS DE CAJA NEGRA VS CAJA BLANCA

4.4.4 PRUEBAS UNITARIAS VS DE INTEGRACIÓN

4.4.3 PRUEBAS DE CAJA NEGRA VS CA

•Caja blanca:

–La Prueba de la caja blanca o white-box testing, es un conjunto de técnicas que tienen como objetivo validar la lógica de la aplicación. Las pruebas se centran en verificar l estructura interna del sistema sin tener en cuenta los requisitos del mismo.

–Uno se centra en el código del módulo, por ejemplo un método y se prueba en función a él.

–Como norma general se intenta que cada instrucción se ejecute al menos una vez y que cada decisión se evalúe al menos una vez a cierto y otra a falso.

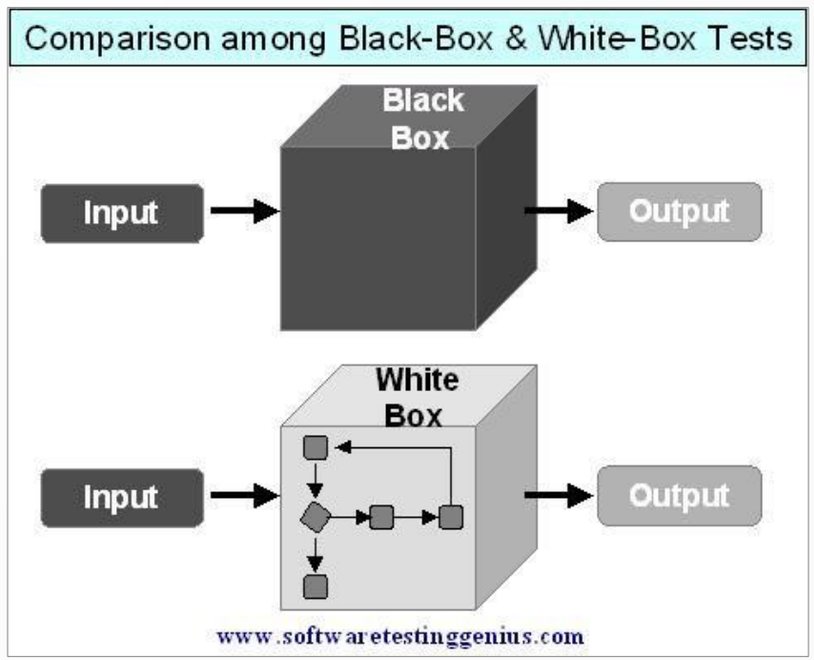
–Así, algunas herramientas usadas:

•Control de flujo:

–Realización de un diagrama de flujo del algoritmo a probar, centrándose en que se recorran todos los posibles caminos al menos una vez, implicando ejecutar todas las instrucciones. Para facilitar el diseño de estas pruebas vemos cada bifurcación, cada camino, cubrir todas las instrucciones y cubrir todas las decisiones a T/F.

•Flujo de datos:

–Seguir el flujo de los datos durante todo el proceso. La DEPURACIÓN es útil aquí.



4.4.3 PRUEBAS DE CAJA NEGRA VS CA

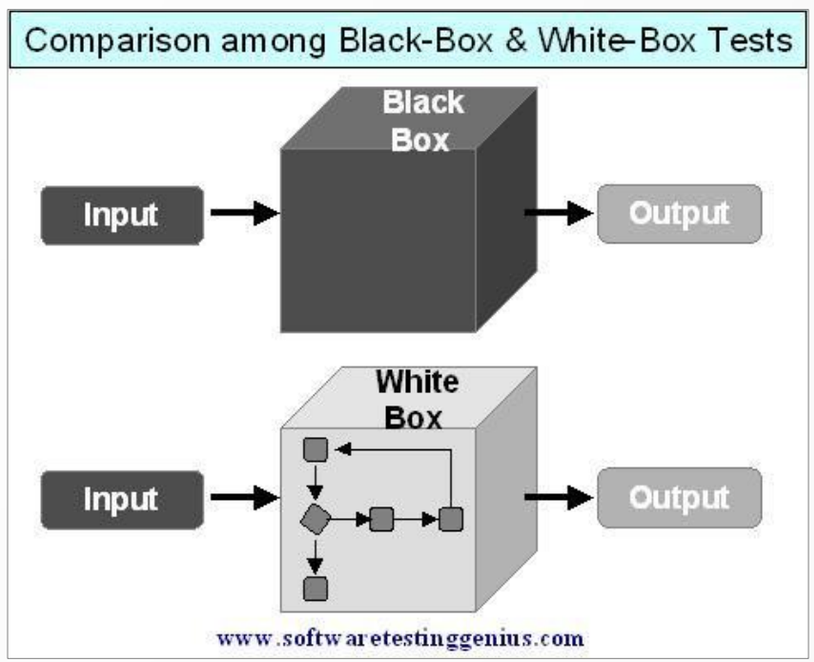
.Caja negra:

–La prueba de la caja negra o black-box testing se basa en comprobar la funcionalidad de los componentes de una aplicación. Determinados datos de entrada a una aplicación o componente deben producir unos resultados determinados.

–Este tipo de prueba está dirigida a comprobar los resultados de un componente de software, no a validar como internamente ha sido estructurado.

–Se llama black-box (caja negra) porque el proceso de pruebas asume que se desconoce la estructura interna del sistema, solo se comprueba que los datos de salida producidos por una entrada determinada son correctos.

–Hoy en día, las **pruebas unitarias** son la técnica black-boxing más extendida



4.4.3 PRUEBAS DE CAJA NEGRA VS CA

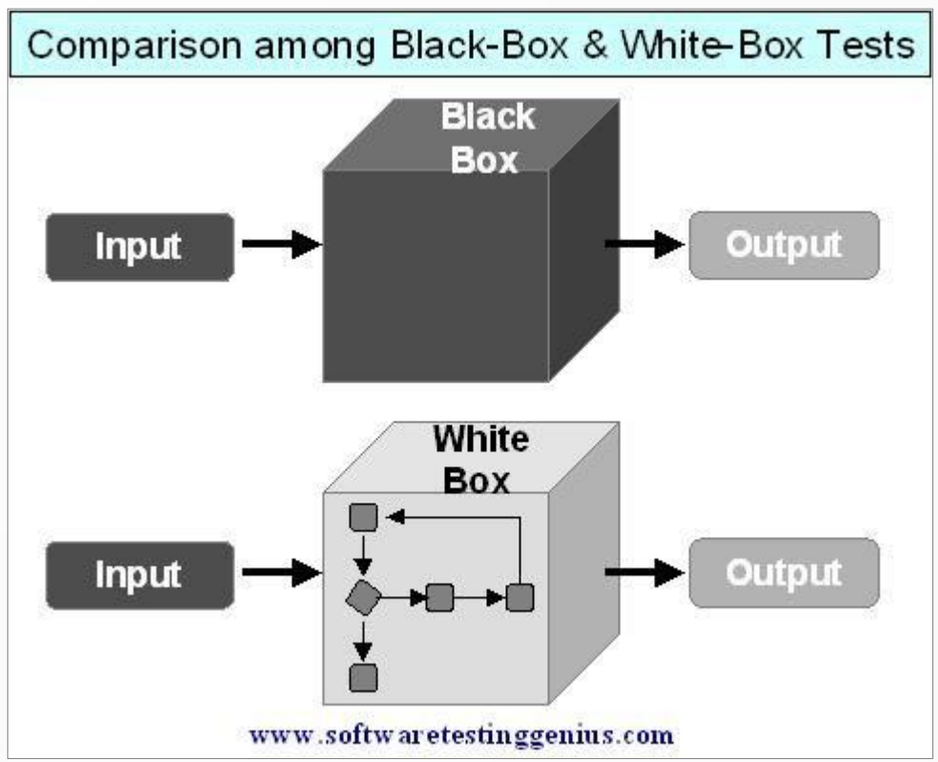
Caja blanca

- se centra en el **código del componente**, por ejemplo un método y se prueba en función a él. (también puede usarse en pruebas de integración: se deben haber probado loscomponentes que se incluyan y se usarán como si fueran instrucciones sueltas)
- Como norma general se intenta que cada **instrucción** se ejecute al menos una vez y que cada decisión se evalúe al menos una vez a **cierto** y otra a **falso**.
- Las **técnicas** usadas en caja blanca son **específicas** (control de flujo, flujo de datos...)

Caja negra

- se trata el elemento a probar como un **todo unitario** en el que solo nos preocupa su **función**: queremos comprobar que la realice correctamente.
- se estudian los **valores** de **entrada** y los valores de **salida** esperados. Se ejecutan las pruebas y se comparan los resultados con los esperados. No nos preocupamos para nada de cómo desempeña su función. Este aspecto es interesante también a la hora de diseñar un módulo ya que nos servirá para determinar la interfaz del mismo.
- Los casos que se prueban no se eligen al azar (al menos no siempre) sino que para ello se utilizan un conjunto de **técnicas comunes** a todas las programaciones (p.ej. si realizamos un método que ejecute una búsqueda binaria en un array las pruebas de caja negra se podrían diseñar incluso antes de implementar el método mientras que para las de caja negra tendríamos en cuenta cómo lo hemos resuelto cada uno.)

Fuente: adaptado de Cuesta (2014: 52)



UD 04.DISEÑO Y REALIZACIÓN DE PRUEBAS

4.1 CONTROL DE CALIDAD Y PRUEBAS

4.4.1 CALIDAD DEL SOFTWARE

4.4.2 PRINCIPIOS BÁSICOS DE LAS PRUEBAS

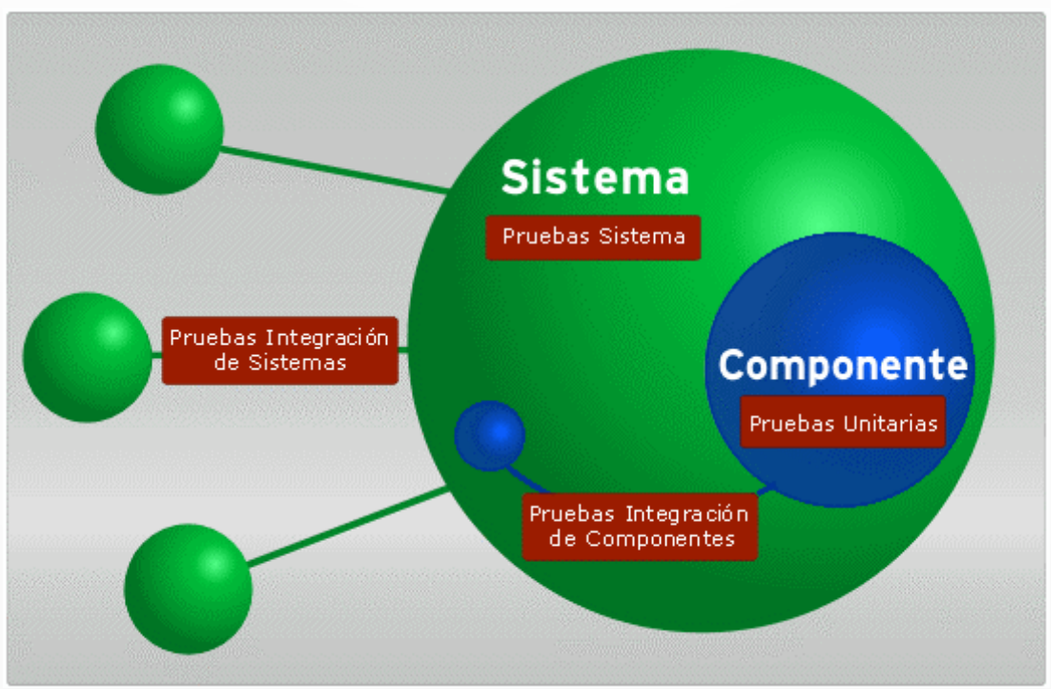
4.4.3 PRUEBAS DE CAJA NEGRA VS CAJA BLANCA

4.4.4 PRUEBAS UNITARIAS VS DE INTEGRACIÓN

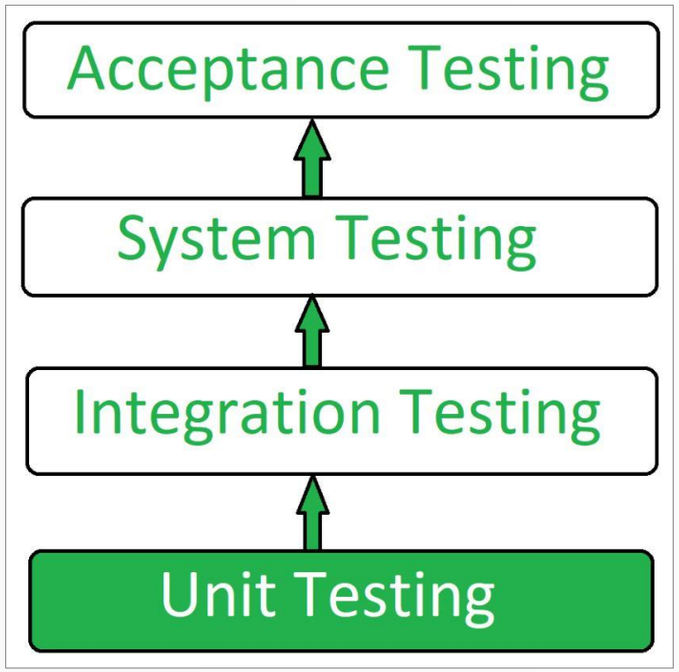
4.4.4 PRUEBAS UNITARIAS VS DE INTE

- Una **prueba unitaria** es una forma de probar el correcto funcionamiento de un módulo de código.
  - Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

- Las **pruebas de integración** sirven para asegurar el correcto funcionamiento del sistema o subsistema en cuestión y, por tanto, la interacción de sus componentes internos.



4.4.4 PRUEBAS UNITARIAS VS DE INTE



- La técnica de las **pruebas unitarias (unit testing)** se basa en la comprobación sistemática de clases o rutinas de un programa utilizando unos datos de entrada y comprobando que los resultados generados son los esperados.
- Una unidad de prueba bien diseñada debe cumplir los siguientes requisitos:

– **Debe ejecutarse sin atención del usuario (desatendida):**

- Una unidad de pruebas debe poder ser ejecutada sin ninguna intervención del usuario: ni en la introducción de los datos ni en la comprobación de los resultados que tiene como objetivo determinar si la prueba se ha ejecutado correctamente.

– **Debe ser universal:**

- Una unidad de pruebas no puede asumir configuraciones particulares o basar la comprobación de resultados en datos que pueden variar de una configuración a otra.

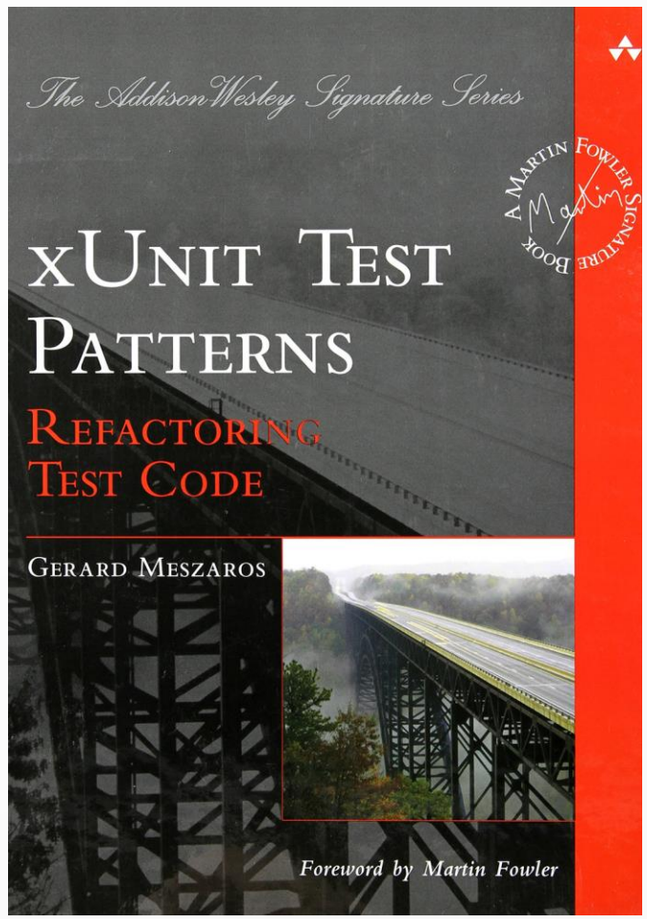
- Debe ser posible ejecutar la prueba en cualquier sistema que tenga el software que es objeto de la prueba.

– **Debe ser atómica:**

- Una unidad de prueba debe ser atómica y tener como objetivo comprobar la funcionalidad concreta de un componente, rutina o clase.

4.4.4 PRUEBAS UNITARIAS VS DE INTE

- Dos de los entornos de comprobación más populares en entornos de software libre con **JUnit y NUnit**.
- Ambos entornos proporcionan la infraestructura necesaria para poder integrar las unidades de comprobación como parte de nuestro proceso de pruebas.
  - **JUnit** está pensado para aplicaciones en entorno Java
  - **NUnit** para aplicaciones desarrolladas en entorno .Net.
- Es habitual el uso del término **xUnit** para referirse al sistema de comprobación independientemente del lenguaje o entorno que utilizamos.



4.4.4 PRUEBAS UNITARIAS VS DE INTE

- Existen muchas herramientas que pueden ayudarnos en el diseño y ejecución de pruebas unitarias.
- **Nosotros veremos JUnit**. Es un entorno basado en Java que utiliza anotaciones para identificar métodos de prueba.
- Asume que todos los métodos de prueba se pueden ejecutar en un orden arbitrario, así que unos tests no deberían depender de otros. Se utiliza alguno de los métodos de JUnit para comprobar si el resultado de la ejecución coincide con el resultado esperado.
- En las prácticas de este tema se detalla cómo trabajar con esta herramienta.