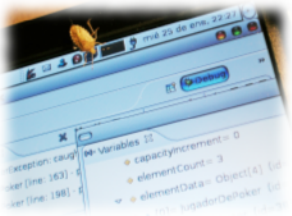
Tema 8: Introducción a las pruebas Software

BLOQUE III:
VERIFICACIÓN, VALIDACIÓN Y PRUEBAS DE LOS SISTEMAS SOFTWARE.

Segundo curso de Grado en Ingeniería Informática
Curso 2014-2015

Javier Sánchez Monedero
jsanchezm@uco.es
http://www.uco.es/users/i02samoj



Índice

Índice

1. Índice 2
2. Introducción 2
3. Tipos de pruebas 6
4. Técnicas de prueba 9
5. Desarrollo guiado por pruebas 17
6. Consideraciones prácticas 18
7. Conclusiones 20

1.

2. Introducción

Introducción

Advertencia

Advertencia
Estos apuntes no contienen todo el material necesario respecto al tema. Intentan dar una visión de conjunto y proporcionar los conocimientos básicos para que podáis consultar la bibliografía y manuales de referencia, así como material complementario colgado en Moodle y los ejercicios realizados en clase.

Introducción

La **prueba de software es un elemento crítico** para la garantía de la calidad del producto de programación y representa un último repaso de las especificaciones, del diseño y de la codificación[1].

Los costes asociados a los fallos software pueden llegar a ser muy altos, por lo que dependiendo del entorno las pruebas pueden llegar a ocupar el 40 % del esfuerzo de un proyecto.

Validación de requisitos

- La **validación de requisitos** es el proceso mediante el cual nos aseguramos de que los requisitos que hemos elegido para el producto que estamos desarrollando reflejan las expectativas de las partes interesadas.
- La **validación de requisitos** nos permite asegurar que los desarrolladores han entendido correctamente los requisitos del producto de software que hay que desarrollar [Miquel and Martos, 2012].

Si no lleváramos a cabo un proceso de validación de los requisitos, podríamos acabar construyendo un software que satisfaga a la perfección lo que se recogió como requisitos, lo que se entendió, pero que no satisfaga las necesidades reales de las partes interesadas.

Verificación de requisitos

La **verificación de los requisitos** es el proceso mediante el cual comprobamos que el sistema desarrollado cumple sus requisitos. La verificación de requisitos se realizar a través de distintas **pruebas del software**.

[1] Punto de vista de modelo en cascada

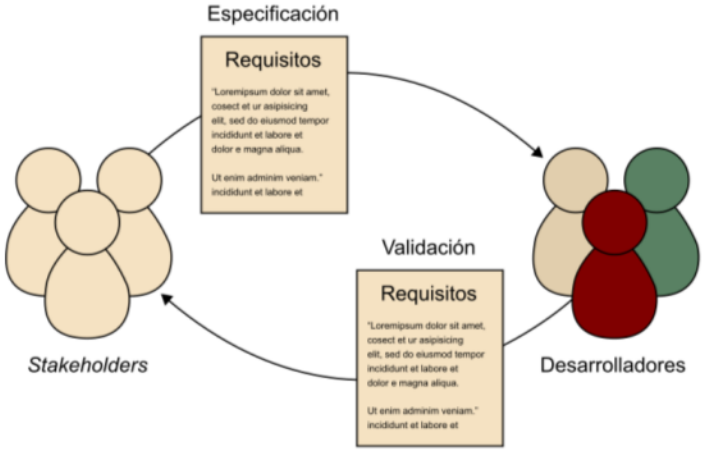


Figura 1: Proceso de validación de requisitos, fuente [Miquel and Martos, 2012].

Tipos de fallos

- **Algorítmico**: Fallo en la secuencia de procesamiento, a nivel de algoritmo. Es fácil de detectar ejecutando con datos prueba. Causas típicas incluyen: ramificar demasiado pronto o demasiado tarde, comprobar una condición incorrecta, no inicializar y no comprobar casos especiales.
- **Precisión**: Utilización de formulas incorrectas, errores de truncamiento o redondeo, o falta de precisión en los resultados.
- **Documentación**: La documentación no concuerda con el software.
- **Capacidad**: La respuesta del sistema es inaceptable cuando éste se carga.
- **Sincronización**: Error en la sincronización entre procesos.
- **Eficiencia**: El sistema no actúa a la velocidad requerida.
- **Recuperación**: El sistema no se comporta correctamente tras detectar un error.
- **Estándares y procedimientos**: El código no sigue las normas de codificación.

Las pruebas software

Pruebas del software

Conjunto de actividades en el que un sistema o componente es ejecutado bajo un conjunto de condiciones específicas, se registran los resultados y se hace una evaluación de alguno de los aspectos del sistema o componente, como por ejemplo su rendimiento, su usabilidad, su seguridad, etc.

En el contexto de la verificación de requisitos, las pruebas se utilizan para someter a examen el conjunto del sistema y verificar que satisface los requisitos y que está libre de errores.

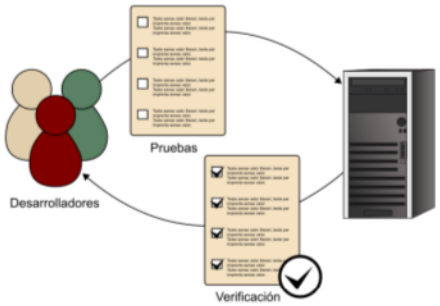


Figura 2: Proceso de verificación de requisitos, fuente [Miquel and Martos, 2012].

Conceptos relacionados

Algunas de las definiciones que se utilizan en esta fase del ciclo de vida son según el estándar IEEE, 1990:

- **Pruebas** (*test*): es una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto.
- **Caso de prueba** (*test case*): un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular.
- **Defecto** (*defect*, *fault*, «*bug*»): un defecto en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa. Se produce cuando una persona comete un error. Es una vista interna, lo ven los desarrolladores.
- **Fallo** (*failure*): La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados. Es un desvío respecto del comportamiento esperado del sistema, puede producirse en cualquier etapa. Es una vista externa, lo ven los usuarios. Es un resultado incorrecto.
- **Error**: Es una acción humana que conduce a un resultado incorrecto.

Objetivos de las pruebas

- La **prueba exhaustiva** del software es **impracticable** (no se pueden probar todas las posibilidades de su funcionamiento) ni siquiera en programas sencillos
- El objetivo de las pruebas es la detección de defectos en el software (**descubrir un error es el éxito de una prueba**)
- **Mito**:

    - Un defecto implica que somos malos profesionales y que debemos sentirnos culpables: todo el mundo comete errores
    - El descubrimiento de un defecto significa un éxito para la mejora de la calidad

Así, para la realización de las pruebas se debe seguir la siguiente **filosofía o intención de trabajo**:

- Cada caso de prueba debe **definir el resultado de salida esperado** que se comparará con el realmente obtenido.
- El **programador debe evitar probar sus propios programas**, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas. Además, es normal que las situaciones que olvidó considerar al crear el programa queden de nuevo olvidadas al crear los casos de prueba
- Se debe **inspeccionar** a conciencia **el resultado** de cada prueba, así, poder descubrir posibles síntomas de defectos
- Al generar casos de prueba, se deben incluir tanto **datos** de entrada **válidos** y esperados como **no válidos** e inesperados.
- Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):
    - Probar si el software no hace lo que debe hacer
    - Probar si el software hace lo que debe hacer, es decir, si provoca efectos secundarios adversos
- No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas (siempre hay defectos)
- La experiencia parece indicar que **donde hay un defecto hay otros**.
- Las pruebas son una **tarea tanto o más creativa que el desarrollo** de software.
- Aunque siempre se han considerado las pruebas como una tarea destructiva y rutinaria. Es interesante planificar y diseñar las pruebas de manera sistemática para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo.
- Es interesante planificar y diseñar las pruebas de manera sistemática para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo, de ahí que haya metodologías centradas en las pruebas (ver secciones finales).

3. Tipos de pruebas

Tipos de pruebas

Tipos de pruebas

El SWEBOK [Bourque and Fairley, 2014] clasifica las pruebas de software según:

- Ámbito (o nivel) de la prueba.
- Objetivo de la prueba.

Tipos de pruebas según ámbito

El SWEBOK [Bourque and Fairley, 2014][Capítulo 4] clasifica las pruebas del software en tres categorías (complementarias), **según el ámbito del sistema** que hay que probar:

1. **Pruebas unitarias** (*Unit Testing*): examinan un componente de manera aislada del resto de los componentes del sistema.
2. **Pruebas de integración** (*Integration Testing*): examinan un subconjunto del sistema para verificar la interacción entre los componentes.
3. **Pruebas de sistema** (*System Testing*): examinan todo el sistema (todos sus componentes a la vez).

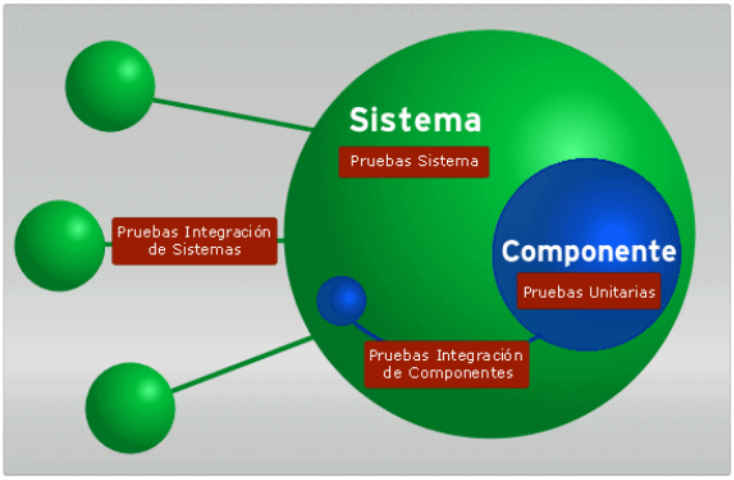


Figura 3: Tipos de pruebas según ámbito, fuente[2]

Los tres tipos de pruebas son complementarios desde el punto de vista de los requisitos. Así, por ejemplo, podemos hacer pruebas unitarias que verifiquen parte de un requisito funcional o pruebas de sistema que verifiquen la

funcionalidad completa o de requisitos no funcionales como, por ejemplo, los requisitos de rendimiento.

Pruebas de componentes

- Las **pruebas de componentes** o **pruebas de unidad** son el proceso de probar los componentes individuales del sistema [Sommerville, 2005].
- Los desarrolladores de los componentes son los propios que suelen ser responsables de estas pruebas.
- Componentes que se pueden probar:
  - Funciones individuales o métodos dentro de un objeto.
  - Clases de objetos que tienen varios atributos y métodos.
  - Componentes compuestos de diferentes objetos o funciones que suelen tener una interfaz definida para acceder a su funcionalidad.

Respecto a los objetos, las pruebas de clases deberían incluir:

- Pruebas aisladas de todas las operaciones.
- Asignación y consulta de todos los atributos.
- Ejecutar el objeto en todos sus posibles estados. Deben simularse los eventos que afectan al estado del objeto.

Tipos de pruebas según objetivo

Otra clasificación que hace el SWEBOK es en función del **objetivo** de la prueba. El total enumerado en el libro:

- Pruebas de aceptación (*Acceptance / Qualification Testing*)
- Pruebas de instalación (*Installation Testing*)
- Pruebas de versiones alfa y beta[3] (*Alpha and Beta Testing*)
- Pruebas de fiabilidad (*Reliability Achievement and Evaluation*)
- Pruebas de regresión (*Regression Testing*)
- Pruebas de rendimiento (*Performance Testing*)
- Pruebas de seguridad (*Security Testing*)
- Pruebas de estrés (*Stress Testing*)

[3] Puedes ver una diferencia entre versiones alfa y beta en http://www.centercode.com/blog/2011/01/alpha-vs-beta-testing/

- Pruebas *Back-to-Back* (*Back-to-Back Testing*), comparación de versiones del mismo software.
- Pruebas de recuperación (*Recovery Testing*)
- Pruebas de interfaz (*Interface Testing*)
- Pruebas de configuración (*Configuration Testing*)
- Pruebas de usabilidad (*Usability and Human Computer Interaction Testing*)

Las pruebas más relacionadas con la verificación de requisitos serían:

- **Pruebas de aceptación**: determinan si el comportamiento del sistema se adecua a los requisitos de los clientes (y, por lo tanto, si estos aceptarán o no el sistema desarrollado).
- **Pruebas de conformidad** (también llamadas funcionales o de corrección): verifican que el software cumpla la especificación.
- **Pruebas de rendimiento** (también *pruebas de estrés* o *pruebas de carga*): verifican que el sistema cumpla los requisitos de rendimiento como la capacidad, el volumen de los datos o los tiempos de respuesta.
- **Pruebas de estrés**: ejercitan el sistema al límite de su capacidad (e incluso un poco más allá) para verificar su comportamiento en estas condiciones.
- **Pruebas de regresión**: tras introducir modificaciones en el sistema, ejecutan de nuevo un conjunto de pruebas que el sistema ya superaba satisfactoriamente para verificar que las modificaciones no han causado efectos no deseados y que el sistema continúa superándolas.

La diferencia entre las pruebas de conformidad y las de aceptación: las pruebas de conformidad verifican los requisitos, las pruebas de aceptación también tienen en cuenta errores de validación.

El Plan de pruebas

Plan de pruebas
El Plan de pruebas documenta el alcance, enfoque, recursos necesarios, etc. de las diferentes pruebas del software.

El estándar IEEE-829 describe el contenido y la estructura recomendados para el plan de pruebas de un proyecto de desarrollo de software. Según este están- dar, un plan de pruebas debe contener, entre otros elementos:

- **Identificador** del Plan de pruebas.
- **Referencias** a otros documentos, como por ejemplo la especificación de requisitos, la documentación de diseño del sistema o la documentación del método que se ha seguido para desarrollar el sistema.

- **Elementos de prueba**, como cuál es el software, con las versiones exactas que probaremos, o qué subsistemas del software probaremos, y características que vamos a examinar (desde el punto de vista del usuario), asociadas a un nivel de riesgo.
- **Riesgos** identificados en el software, como cuáles son las áreas en las que es más probable que tengamos problemas y, por lo tanto, que debemos examinar de forma más exhaustiva.
- **Características que no se probarán**, indicando el motivo por el que no se hará dicha comprobación (bajo riesgo, no forman parte de la entrega actual, etc.).
- **Estrategia** de pruebas:
  - ¿Qué herramientas se utilizarán?
  - ¿Qué métricas se recogerán? ¿En qué nivel se recogerá cada métrica?
  - ¿Cómo se gestionará la configuración? ¿Cuántas configuraciones diferentes se gestionarán?
  - ¿Qué software y hardware se utilizarán?
  - ¿Se harán pruebas de regresión? ¿Hasta qué punto?
- **Criterios de aceptación de la prueba** (para cada prueba): ¿cómo se decidirá que la prueba se ha completado con éxito?
- **Criterios de cancelación** de la prueba (para cada prueba): ¿cómo se decidirá si hay que suspender o cancelar una prueba?
- **Entregables**: ¿qué se entregará como resultado de la ejecución de la prueba? (el documento del plan de pruebas, los casos de prueba, los registros (*logs*) de error, el informe de problemas, etc.).
- **Responsables** de las pruebas y aprobaciones: ¿quién es el responsable de la ejecución de las pruebas? ¿Quién está autorizado a dar el proceso por finalizado y a aceptar el resultado de las pruebas?
- **Calendario**.

Organización de las pruebas

4. Técnicas de prueba

Técnicas de prueba

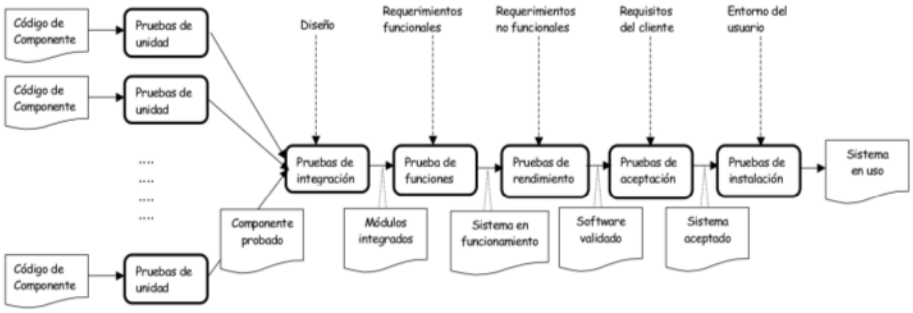


Figura 4: Organización de las pruebas software.

- El diseño de casos de prueba está condicionado por la **imposibilidad de probar exhaustivamente el software**, el objetivo de dichas técnicas es conseguir una confianza aceptable en que se detectaran los defectos existentes sin necesidad de consumir una cantidad excesiva de recursos.
- La idea fundamental para el diseño de casos de prueba consiste en elegir algunas de ellas que, por sus características, se consideren **representativas del resto**.
- Por tanto podremos asumir que si no se detectan defectos en el software al ejecutar dichos casos podemos tener un **cierto nivel de confianza** en que el programa no tiene defectos.
- La dificultad estará en **saber elegir los casos de prueba adecuados** en cada situación.

Enfoques de diseño de pruebas I

Existen varias aproximaciones para diseñar casos de prueba:

- **Pruebas basadas en requisitos**: los casos se diseñan para probar los requisitos del sistema. Se utiliza principalmente en la etapa de pruebas del sistema, ya que los requisitos a menudo se implementan por varios componentes. Para cada requisito se establece un caso de prueba para comprobar que se satisface.
- **Pruebas de particiones** o **funcionales** o de **caja negra**: se identifican **particiones** de entrada y salida y se diseñan pruebas para que el sistema ejecute todas las entradas y genere todas las salidas. Las **particiones** son grupos de datos con características comunes.
- **Pruebas estructurales** o de **caja blanca**: se utiliza el conocimiento de la estructura interna del programa para diseñar pruebas que ejecutan todas las partes del programa. Cuando probamos un programa, cada sentencia debería ejecutarse al menos una vez.

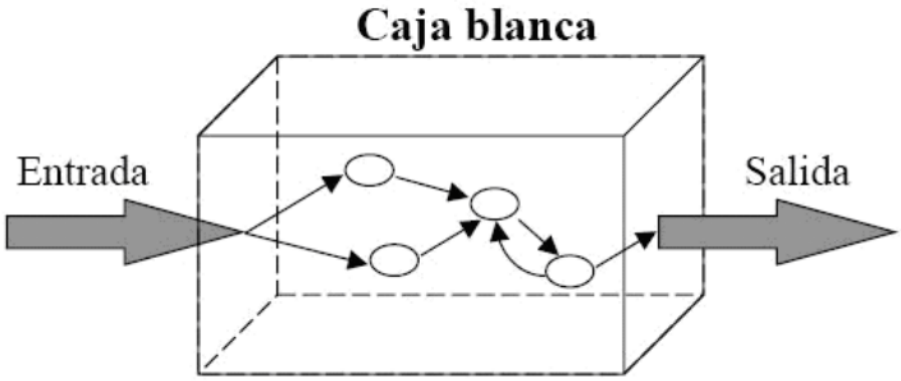


Figura 5: Pruebas caja blanca

Figura 6: Pruebas caja negra

- El **enfoque aleatorio** consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba. Por ejemplo, la técnicas de *siembra de fallos* consiste en insertar fallos intencionalmente, y realizar pruebas en busca de errores:

$$fallos\_reales = fallos\_intencionales \frac{fallos\_reales\_detectados}{fallos\_intencionales\_detectados}$$

Pruebas basadas en requisitos

- Un principio general de la ingeniería de requisitos es que **los requisitos deberían poder probarse**.
- Se considera cada requisito y se deriva un conjunto de pruebas para validarlos (**prueba de validación**).

Ejemplo de requisitos

1. El usuario será capaz de buscar en un conjunto inicial de bases de datos o bien seleccionar un subconjunto de éstas.
2. El sistema proporcionará vistas apropiadas para que el usuario pueda leerlos documentos almacenados.
3. Cada petición debería contener un único identificador (ORDER_ID) que el usuario deberá ser capaz de copiar en el área de peticiones de almacenamiento permanente.

Ejemplo de pruebas de requisitos
Posibles pruebas para el primero de estos requisitos, suponiendo que se ha probado una función de búsqueda:

- Iniciar búsquedas de usuario para elementos de los que se conoce que están presentes y para elementos que se sabe que no están presentes, en las que el conjunto de bases de datos incluye una base de datos.
- Iniciar búsquedas de usuario para elementos de los que se sabe que están presentes y para elementos de los que se sabe que no están presentes, en las que el conjunto de bases datos incluye dos base de datos.
- Iniciar búsquedas de usuario para elementos de los que se sabe que están presentes y elementos de los que se sabe que no están presentes, en las que el conjunto de bases datos incluye más de dos bases de datos.
- Seleccionar una base de datos del conjunto de bases de datos e iniciar búsquedas de usuarios para elementos que se sabe que están presentes y para elementos de los que se sabe que no están presentes.
- Seleccionar más de una base de datos del conjunto de bases de datos e iniciar búsqueda de usuario para elementos de los que se sabe que están presentes y para elementos de los que se sabe que no están presentes.

Pruebas de particiones

- Los datos de entrada y los resultados de salida de un programa o componente normalmente se pueden agrupar en varias clases diferentes con características comunes.
- Estas clases con comportamiento equivalente se denominan **particiones de equivalencia** o **clases de equivalencia** o dominio.
- Ejemplo: todos los números negativos, todas las cadenas con menos de 30 caracteres, todos los eventos asociados a una opción de un menú, etc.
- En el diseño de pruebas, se tratará de **identificar todas las particiones** para un sistema o componente de forma que las entradas y salidas pertenezcan a estas particiones.
- Una buena práctica para la selección de los casos de prueba es elegir estos en los **límites de las particiones** (valores atípicos) junto con los casos de prueba cercanos al **punto medio de la partición** (valores típicos). A menudo los fallos se producen en los valores atípicos.

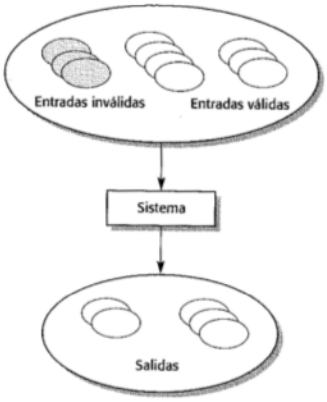


Figura 7: Particiones de equivalencia, fuente [Sommerville, 2005]

- Los valores típicos o atípicos se obtienen de la especificación, la documentación o la propia experiencia (Ejemplo fecha *40 de enero*).
- Habitualmente se unen las clases válidas y se tratan individualmente cada una de las clases no válidas.

Guías		
Dominio	Clases	Comprobación
[a, b]	{x < a}, [a, b], {x > b}	Rango
a	{x < a}, {a}, {x > a}	Longitud
true, false	{true}, {false}	Presencia/Ausencia
x∈S	S, ¬S	Tipo

Figura 8: Guías para las clases de equivalencia, fuente Dpt. Lenguajes y Sistemas Informáticos ETSII. Univ. de Granada [2005].

Ejemplo de particiones de equivalencia

Supongamos un programa que acepta como entrada de 4 a 8 valores de entrada que son cinco dígitos mayores de 10.000. La Figura 10 muestra las particiones de posibles valores de prueba.

Pruebas estructurales

- El **diseño de casos de prueba** tiene que estar basado en la **elección de caminos importantes** que ofrezcan una seguridad aceptable de que se descubren defectos (un programa de 50 líneas con 25 sentencias if en serie da lugar a 33,5 millones de secuencias posibles), para lo que se usan los criterios de **cobertura lógica**, por eso estos test también suelen llamarse **test de cobertura**.

Guías	
Dominio	Casos de prueba
[a, b]	a, b, pred(a), succ(b)
S= a, b, c, d, e, f	Min(S), Max(S), pred(Min(S)), succ(Max(S))
Estructuras de datos	Casos de prueba
Tamaño N	Alcanzar tamaños N y N+1

Figura 9: Guías para el análisis de valores límite, fuente Dpt. Lenguajes y Sistemas Informáticos ETSII. Univ. de Granada [2005].

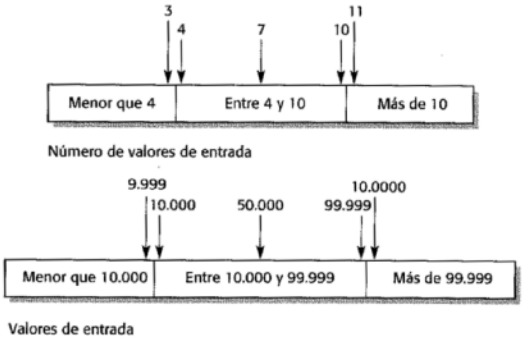


Figura 10: Ejemplo de particiones de equivalencia, fuente [Sommerville, 2005]

- Pruebas de caja blanca:
  - **Prueba de instrucciones**. Cada instrucción se ejecuta al menos una vez.
  - **Prueba de decisiones**. Cada decisión se evalúa al menos en una vez dando cada posible resultado.
  - **Prueba de bifurcaciones**. Cada camino en cada bifurcación es seguido al menos una vez.
  - **Prueba de caminos**. Cada camino distinto en el código se sigue al menos una vez.
  - **Prueba de cadenas definición-uso**.

Pruebas de caminos base

- Las pruebas de caminos base son una estrategia de pruebas estructurales cuyo objetivo es probar cada **camino de ejecución independiente** en un componente o programa.
- Si se recorre al menos una vez cada camino independiente, entonces todas las sentencias del componente se han ejecutado al menos una vez y todas las sentencias condicionales pruebas sus casos verdadero y falso.

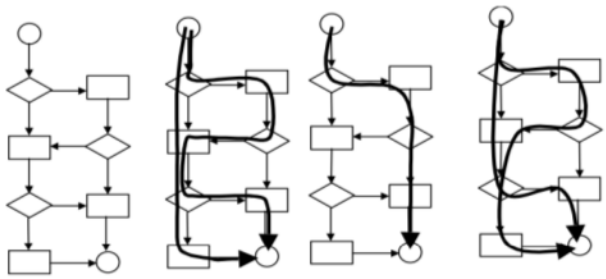


Figura 11: Prueba de caminos, fuente [Dpt. Lenguajes y Sistemas Informáticos ETSII. Univ. de Granada, 2005].

- Como el número de caminos es proporcional al tamaño del módulo, estas pruebas se utilizan sobre todo a nivel de componente.
- Estas pruebas no prueban todas las combinaciones de caminos. Por ejemplo, en componentes con bucles el número de combinaciones de caminos tiende a infinito.

Pruebas de caminos base: grafo

- El punto de partida de las pruebas es un **grafo de flujo del programa** que representa todos los caminos.
- Los nodos representan decisiones y aristas el flujo de control.
- Cada rama en una sentencia condicional (*if-else* o *case*) se muestra como un camino independiente.
- Una flecha que vuelve a un nodo representa un bucle.

Complejidad ciclomática

El número de caminos independientes en un programa se puede encontrar calculando la **complejidad ciclomática** (propuesta por McCabe en 1976).

La **complejidad ciclomática** (en inglés, *Cyclomatic Complexity*) es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Es una de las métricas de software de mayor aceptación, ya que ha sido concebida para ser independiente del lenguaje[4].

Complejidad ciclomática y métricas software

Al programar se debe tratar de mantener baja la complejidad ciclomática de las funciones. Valores altos de complejidad ciclomática (máximo 10 y excepcionalmente hasta 15[5]) nos indican que el código debe refactorizarse para reducirla, por ejemplo aplicando principios de diseño SOLID. Puedes ver otras métricas

[4] http://es.wikipedia.org/wiki/M%C3%A9trica_del_software
[5] http://en.wikipedia.org/wiki/Cyclomatic_complexity

de software en Wikipedia[6]. Eclipse tiene extensiones para extraer métricas del proyecto software.

Cálculo de la CC y pruebas

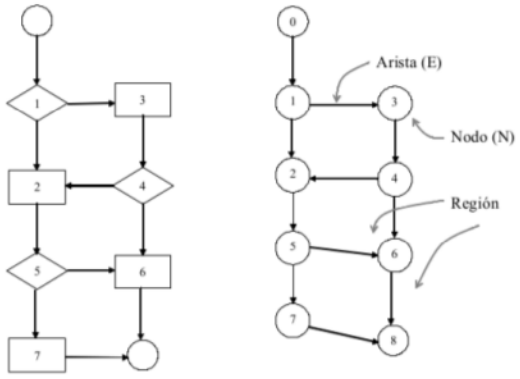


Figura 12: Ejemplo de grafo para cálculo de complejidad ciclomática.

1. Construir el grafo de flujo.
2. Determinar la complejidad ciclomática (tres formas):
   - V(G) = Número de regiones
   - V(G) = Número de condiciones + 1
   - V(G) = E-N+2
3. Obtener un conjunto de caminos base independientes.
4. Elaborar casos de prueba que fuercen el paso por cada uno de los caminos base.

Pruebas de estructuras de datos

- Cuando se implementan o utilizan **estructuras de datos** como árboles, grafos, listas, etc. se suelen probar operaciones de inserción, búsqueda, modificación y borrado.
- Estas operaciones en inglés se suelen recoger bajo el acrónimo **CRUD** (*Create*, *Read*, *Update*, *Delete*).

En las pruebas de estas estructuras se realizarán operaciones CRUD partiendo de y/o tratando de generar las siguientes **situaciones**:

[6] http://en.wikipedia.org/wiki/Software_metric

- Estructura con ningún elemento almacenado.
- Estructura con uno y dos elementos almacenados.
- Estructura con varios elementos almacenados.
- Si se ha definido un límite máximo de elementos, se prueba a llegar y rebasar este límite[7].

Obviamente, dependiendo del tipo de estructura estas pruebas se adaptarán. Por ejemplo un árbol que debe mantener un orden siempre que se realice una operación CRUD y esto afectará al diseño de pruebas.

5. Desarrollo guiado por pruebas

Desarrollo guiado por pruebas

Introducción al desarrollo guiado por pruebas

- La metodología en cascada sólo incluye las pruebas como parte final del proceso de desarrollo software.
- Las metodologías iterativas e incrementales incluyen las pruebas de software en cada iteración.
- Es habitual que por motivos de tiempo/coste las pruebas sean una cuestión secundaria, e incluso se obvien.

El **desarrollo guiado por las pruebas** es una manera de desarrollar software que intenta solucionar este problema aplicando dos reglas muy sencillas:

- Hay que escribir las pruebas antes de implementar el sistema, por tanto sirven para especificar el sistema.
- Solo se tiene que implementar la funcionalidad necesaria para superar las pruebas.

Inicialmente el desarrollo guiado por pruebas se denominó *test-driven development* (TDD) al referirse sólo a pruebas unitarias. Al generalizarse dio lugar al **desarrollo guiado por el comportamiento** (*behavior driven development*, BDD).

El BBD es muy popular hoy en día sobre todo en el desarrollo de sistemas de información donde se tiene claro cuál será el comportamiento del usuario y qué respuesta se le quiere dar con la aplicación.

[7] A menudo el límite de memoria lo establece el sistema operativo y/o la máquina virtual en el caso lenguajes interpretados

6. Consideraciones prácticas

Consideraciones prácticas

Herramientas

- **Analizadores estáticos**: Analizadores de código, generadores de grafos de estructura, analizadores de datos.
- **Analizadores dinámicos**: Registran el comportamiento del sistema, generando informes útiles para evaluar la eficiencia y el comportamiento en general.
- **Ejecución de pruebas**: Captura y *playback*, generadores de casos prueba.

Existe software específico que ayuda a realizar distintos tipos de pruebas:

- Pruebas de unidad: JUnit, Google Test[8]...
- Pruebas de interfaz de usuario: listado en Wikipedia[9].
- Herramientas para desarollo guiado por comportamiento en PHP: Behat[10]

Automatización de pruebas

- La ejecución de las pruebas de software es un proceso repetitivo (sobre todo en el caso de las pruebas de regresión) y, en muchas ocasiones, monótono.
- Es probable que la persona encargada de la prueba cometa errores durante la ejecución del guión de pruebas o durante la verificación de los resultados.
- La automatización de las pruebas del software es fundamental para asegurar la consistencia y la **reproductividad** de los resultados.
- Las pruebas automatizadas **pueden ejecutarse con más frecuencia** y, por lo tanto, nos ayudan a **detectar problemas antes** y a obtener un software de **más calidad**.

Tipos de pruebas automatizadas

Las pruebas que se pueden automatizar son de todo tipo, por ejemplo:

- **Pruebas de aceptación**: se puede automatizar la prueba de los requisitos funcionales del sistema.

[8] SegúnWikipediatambiénsepuedenhacerpruebasdeintegraciónyaceptación.
[9] http://en.wikipedia.org/wiki/List_of_GUI_testing_tools
[10] http://docs.behat.org

- **Pruebas de carga/rendimiento**: por ejemplo, en entornos web es bastante común hacer este tipo de pruebas, y existen herramientas específicas para este entorno como JMeter de Apache[11].

- **Datos de prueba**: los datos de prueba pueden ser datos reales o datos sintéticos (sobre todo en las pruebas de carga por volumen o el uso de casos límite).

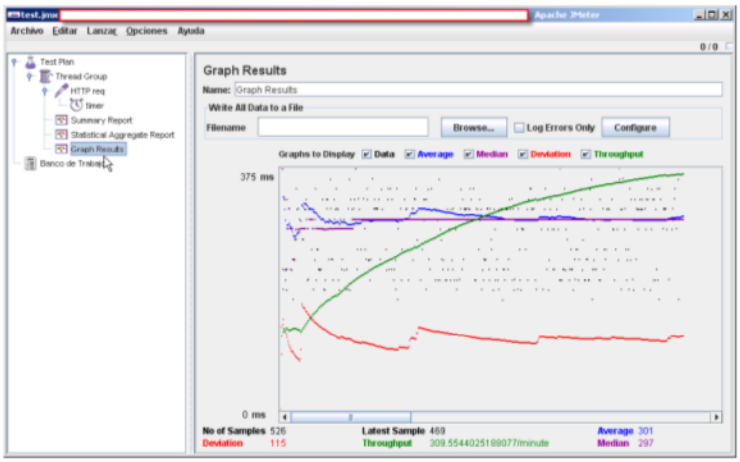


Figura 13: Captura de pantalla de la aplicación JMeter.

Integración continua

- La **integración continua** es un modelo informático que consiste en hacer integraciones automáticas de un proyecto lo más a menudo posible para así poder detectar fallos cuanto antes. Entendemos por integración la compilación y ejecución de pruebas de todo un proyecto [Wikipedia, 2014].

- El proceso puede realizarse periódicamente o al enviar cambios a un sistema de control de versiones.

- Aunque la integración continua suele asociarse a metodologías ágiles, existen muchos productos software que la integran por una simple razón de productividad y eficiencia.

[11] http://jmeter.apache.org/

7. Conclusiones

Conclusiones

Conclusiones

- La cuestión de la prueba de software es bastante compleja y este tema sólo presenta una introducción.
- Se han presentado algunos tipos de prueba haciendo énfasis en los más cercanos al curso, pero quedan muchos otros tipos que describir como las pruebas de integración o las pruebas de usabilidad (entre muchas).

8. Bibliografía

Fuentes

El contenido de estos apuntes se basa a su vez en otros apuntes [Irene T., 2013] y [Dpt. Lenguajes y Sistemas Informáticos ETSII. Univ. de Granada, 2005], y en las referencias citadas, fundamentalmente el Capítulo 23 del libro de Ingeniería del Software de Ian Sommerville [Sommerville, 2005].

Bibliografía y enlaces

Referencias

P. Bourque and R. Fairley, editors. *Guide to the Software Engineering Body of Knowledge, Version 3.0,*. IEEE Computer Society, 2014. URL www.swebok.org.

Dpt. Lenguajes y Sistemas Informáticos ETSII. Univ. de Granada. Tema 6: Prueba del Software. Ingeniería del Software III. 4o Curso Ingeniería Informática. 2005.

L. R. Irene T. Apuntes de la Asignatura Ingeniería del Software. Tema 8: Introducción a la Verificación, Validación del Software. 2013.

J. Pradel i Miquel and J. A. R. Martos. *Validación y verificación de requisitos. Asignatura Ingeniería de requisitos.* UOC, 2012.

I. Sommerville. *Ingeniería del Software*. Pearson Education. Addison-Wesley, 2005.

Wikipedia. Integración continua — wikipedia, la enciclopedia libre, 2014. URL http://es.wikipedia.org/w/index.php?title=Integraci%C3%B3n_continua&oldid=78139215. [Internet; descargado 18-diciembre-2014].