TEMA 4. Estrategias de prueba del Software (2/4)

Calidad del Software

Dr. José Luis Abellán Miguel

Grado en Ingeniería Informática

Dr. José Luis Abellán Miguel - Tlf: (+34) 968 278 657
Universidad Católica San Antonio de Murcia www.ucam.edu

Índice

Bibliografía

- Pressman, R. ***Ingeniería del Software: Un enfoque práctico***. 7ª edición. Madrid: McGraw Hill, 2010. ISBN: 9701054733 (disponible en la biblioteca UCAM) → **Capítulo 17**

Estrategia de Prueba para SW OO (1/3)

- **Prueba de Unidad**: Clases encapsuladas
  - Cada instancia de una clase empaqueta los atributos (datos) y las operaciones que manipulan estos datos.
  - Las operaciones (métodos) dentro de la clase son las unidades comprobables más pequeñas
    - Ya no es posible probar una sola operación en aislamiento (la visión convencional de la prueba de unidad) sino más bien como parte de una clase

      Jerarquía de clases: Superclase (Método X). X en subclases usa atributos de las subclases → El contexto en el que se usa X depende del contexto (superclase o subclases)

Estrategia de Prueba para SW OO (2/3)

- **Prueba de Integración**:
  - El software orientado a objeto no tiene una estructura de control jerárquico obvia:
    - Integración ascendente/descendente??
  - **Prueba basada en hebra**: integra el conjunto de clases requeridas para responder a una entrada o evento para el sistema
    - Cada hebra se prueba de manera individual
- **Prueba basada en uso**:
  1. Clases independientes
  2. Clases dependientes de las anteriores
  3. Hasta construcción del sistema

Estrategia de Prueba para SW OO (3/3)

- Uso de controladores
  - Interfaz de usuario
  - Pruebas de los métodos de las clases
- Uso de representantes o *stubs*
  - Clases que dependen de otras que todavía no se han implementado por completo
- Pruebas de grupo
  - Descubrir errores en las colaboraciones entre clases
    - Se partirá de los modelos Clase-Responsabilidad-Colaboración y diagrama de objeto-relación

Estrategia de Prueba para *Webapps* (1/2)

1. El modelo de contenido para la *webapp* se revisa para descubrir errores
2. El modelo de interfaz se revisa para garantizar que todos los casos de uso pueden adecuarse
3. El modelo de diseño para la *webapp* se revisa para descubrir errores de navegación
4. La interfaz de usuario se prueba para descubrir errores en los mecanismos de presentación y/o navegación
5. A cada componente funcional se le aplica una prueba de unidad
6. Se prueba la navegación a lo largo de toda la arquitectura

Estrategia de Prueba para *Webapps* (2/2)

7. La *webapp* se implementa en varias configuraciones ambientales diferentes y se prueba en su compatibilidad con cada configuración.
8. Las pruebas de seguridad se realizan con la intención de explotar vulnerabilidades en la *webapp* o dentro de su ambiente.
9. Se realizan pruebas de rendimiento.
10. La *webapp* se prueba mediante una población de usuarios finales controlada y monitorizada.
    - Los resultados de su interacción con el sistema se evalúan por errores de contenido y navegación, preocupaciones de facilidad de uso, preocupaciones de compatibilidad, así como confiabilidad y rendimiento de la *webapp*.

Pruebas de Validación (1/3)

- Comienzan tras la conclusión de las pruebas de integración
  - El software está completamente ensamblado como un paquete y los errores de interfaz se descubrieron y corrigieron
  - Desaparece la distinción entre software convencional, software orientado a objetos y *webapps*
  - Las pruebas se enfocan en las acciones visibles para el usuario y las salidas del sistema reconocibles por el usuario
    - Serie de pruebas que demuestran conformidad con los requerimientos establecidos por el cliente

Pruebas de Validación (2/3)

- Plan de prueba:
  - Clases de pruebas de validación que se van a realizar
  - Procedimiento para cada caso de prueba de validación para garantizar:
    - Satisfacen todos los requerimientos de funcionamiento
    - Se logran todas las características de comportamiento
    - Todo el contenido es preciso y se presenta de manera adecuada
    - Se logran todos los requerimientos de rendimiento
    - La documentación es correcta
    - Se satisfacen la facilidad de uso y otros requerimientos (e.g., transportabilidad, compatibilidad, recuperación de error, mantenimiento, etc.).

Pruebas de Validación (3/3)

- Tipos de pruebas de validación:
  - **Prueba alfa**:
    - Se lleva a cabo en el sitio del desarrollador por un grupo representativo de usuarios finales en un ambiente controlado
    - El desarrollador está presente
  - **Prueba beta**:
    - Se realiza en uno o más sitios del usuario final
    - El desarrollador no está presente

      El usuario final detecta defectos durante la prueba beta y los reporta al desarrollador periódicamente.
    - Prueba de aceptación: es un tipo de prueba beta en la que se contrata a un cliente(s) para probar el SW antes de liberarlo

      Puede ser muy formal y abarcar muchos días (semanas)

Pruebas de Sistema (1/4)

- Serie de pruebas cuyo propósito principal es ejercitar por completo el sistema basado en computadora
  - Cada prueba tenga un propósito diferente, todo él funciona para verificar que los elementos del sistema se hayan integrado de manera adecuada y que se realicen las funciones asignadas
- Problema clásico: el "*dedo acusador*"
  - Se descubre un defecto y los desarrolladores de diferentes elementos del sistema se culpan unos a otros por el problema

Pruebas de Sistema (2/4)

- **Pruebas de recuperación**
  - Forzar al software a fallar en varias formas y verificar que la recuperación se realice de manera adecuada.
    - Si la recuperación es automática: se evalúa el reinicio, los mecanismos de puntos de verificación, la recuperación de datos y la reanudación para correcciones
    - Si la recuperación requiere intervención humana: se evalúa el tiempo medio de reparación (TMR) para determinar si está dentro de límites aceptables

- **Pruebas de seguridad**
  - Verificar que los mecanismos de protección que se construyen en un sistema en realidad lo protegerán de cualquier penetración impropia

  ¡Cualquier cosa vale!
    - Adquirir contraseñas, puede atacar el sistema con software a la medida diseñado para romper cualquier defensa que se haya construido, prueba por denegación de servicio, etc.
  - **Objetivo**: costo por vulnerar la seguridad sea mayor que el costo de la información que se protege

Pruebas de Sistema (3/4)

❑ Pruebas de esfuerzo

- Ejecuta un sistema en forma que demanda recursos en cantidad, frecuencia o volumen anormales
  - Diseñarse pruebas especiales que generen diez interrupciones por segundo, cuando una o dos es la tasa promedio
  - Aumentarse las tasas de entrada de datos en un orden de magnitud para determinar cómo responderán las funciones de entrada
  - Ejecutarse casos de prueba que requieran memoria máxima y otros recursos
  - Ejemplo de herramientas para pruebas de esfuerzo sobre aplicaciones *con acceso a Internet* (*webpagetest* , *loadimpact*).

La persona que realiza la prueba intenta romper el programa

Pruebas de Sistema (4/4)

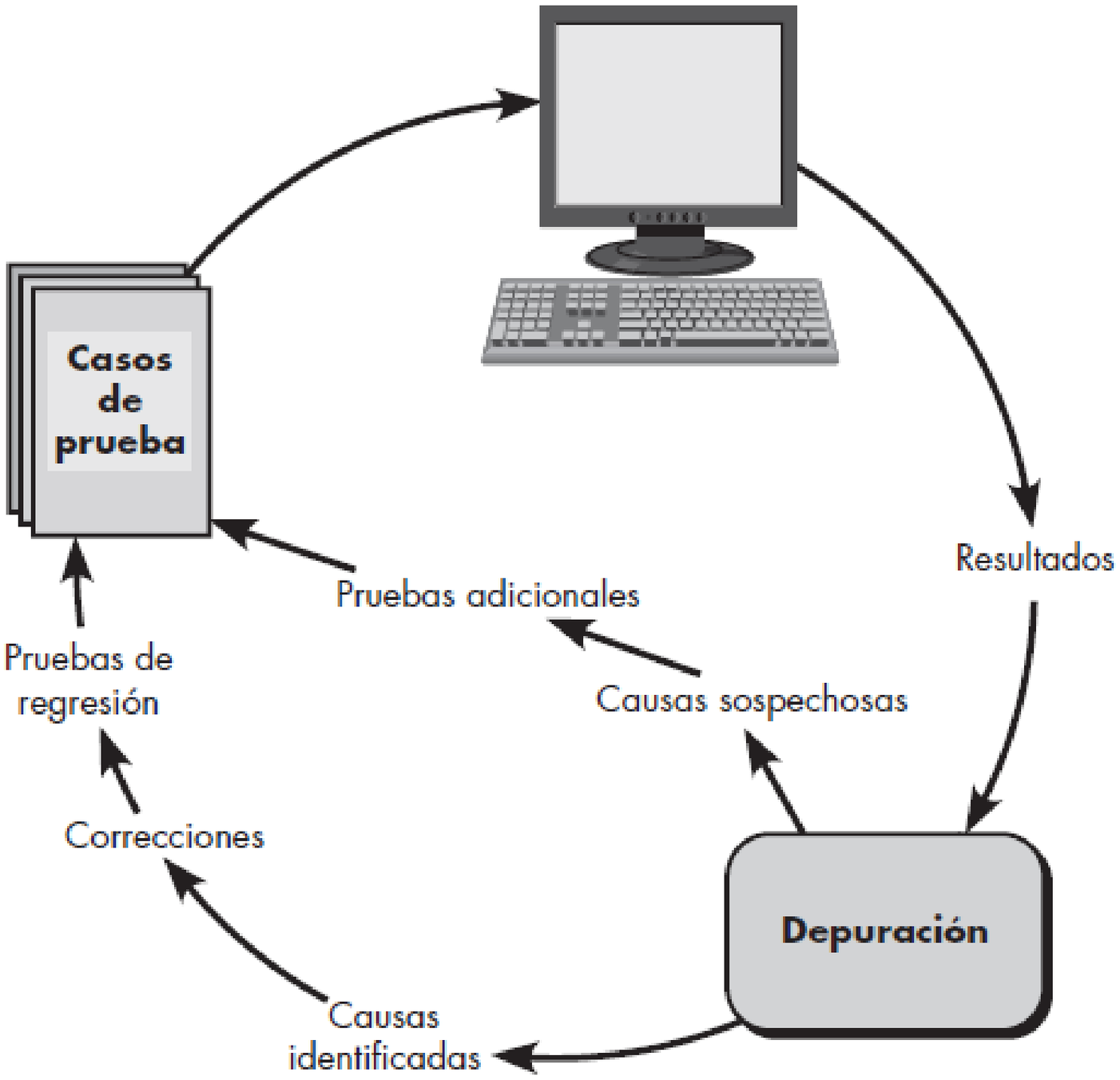
- **Pruebas de rendimiento**
  - Sistemas en tiempo real y sistemas embebidos además de satisfacer los requerimientos del cliente tienen que soportar unos mínimos de rendimiento
    - Es necesario medir la utilización de los recursos (por ejemplo, ciclos del procesador) en forma meticulosa
- **Pruebas de despliegue o configuración**
  - El software debe ejecutarse en varias plataformas y bajo más de un entorno de sistema operativo
    - Ejercita el software en cada entorno en el que debe operar
    - Examina procedimientos de instalación y el software de instalación especializado que usarán los clientes, así como toda la documentación que se usará para introducir el software a los usuarios finales

El Proceso de Depuración (1/4)

- La *depuración* no es una prueba pero ocurre como consecuencia de las pruebas exitosas
  - Un caso de prueba exitoso descubre un error → la depuración es el proceso que da como resultado la eliminación del error
- Es un proceso complejo: la manifestación externa del error (síntoma) y su causa interna pueden no tener relación obvia

a) La causa se encontrará y corregirá
b) La causa no se encontrará → la persona que realiza la depuración puede sospechar una causa, diseñar un caso de prueba para auxiliarse en la validación de dicha suposición y trabajar hacia la corrección del error en forma iterativa.

El Proceso de Depuración (2/4)



El Proceso de Depuración (3/4)

- El problema de la complejidad de la depuración
  - El síntoma y la causa pueden ser remotos
  - El síntoma puede desaparecer (temporalmente) cuando se corrige otro error
  - El síntoma en realidad puede no ser causado por errores (por ejemplo, imprecisiones de redondeo)
  - El síntoma puede ser resultado de problemas de temporización más que de problemas de procesamiento.
  - Puede ser difícil reproducir con precisión las condiciones de entrada (por ejemplo, una aplicación en tiempo real en la que el orden de la entrada esté indeterminado)
  - …

El Proceso de Depuración (4/4)

Estrategias de depuración (1/3)

- La depuración tiene un objetivo dominante: encontrar y corregir la causa de un error o defecto de software
  - Combinación de evaluación sistemática, intuición y suerte

"Establezca un límite, por decir, dos horas, en la cantidad de tiempo que empleará al intentar depurar un problema por cuenta propia. Después de eso, ***¡pida ayuda!"***

- Se han propuesto tres estrategias de depuración:
  - Fuerza bruta
  - *Backtracking*
  - Eliminación de causas

El Proceso de Depuración (4/4)

Estrategias de depuración (2/3)

- Fuerza bruta:
  - Probablemente es el método más común y menos eficiente para aislar la causa de un error de software
  - Se aplican cuando todo lo demás falla: “deje que el ordenador encuentre el error”
  - Reporte masivo de información de la ejecución del sistema: copias de la memoria (*dumps*), generación de trazas de memoria, consumo energético, tráfico de red, etc.
  - El análisis automático/humano de esta información podría conducir a entender dónde está la causa del problema

El Proceso de Depuración (4/4)

Estrategias de depuración (3/3)

- *Backtracking*:
  - Enfoque de depuración bastante común que puede usarse exitosamente en programas pequeños
  - Comenzar en el sitio donde se descubrió un síntoma, el código fuente se rastrea hacia atrás (de manera manual) hasta que se encuentra la causa → El número de rutas potenciales hacia atrás puede volverse inmanejable
- Eliminación de la causa:
  - Los datos relacionados con la ocurrencia del error se organizan para aislar las causas potenciales
  - Se hace una lista de las posibles causas y se realizan pruebas para eliminar cada una
    - Si una hipótesis de causa particular se muestra prometedora, los datos se refinan con la intención de aislar el error

El Proceso de Depuración (4/4)

La Corrección del Error

- Una vez encontrado el error debe corregirse, ¡¡pero la corrección de un error puede introducir otros errores!!
- Preguntas simples que han de plantearse previamente a la corrección del error
  - *¿La causa del error se reproduce en otra parte del programa?*
    - Reutilizar patrones de lógica de corrección de errores
  - *¿Qué "siguiente error" puede introducirse con la corrección que está a punto de realizar?*
    - *Software fuertemente acoplado puede sufrir más errores*
  - *¿Qué debió hacerse para evitar este error desde el principio?*
    - *Entender bien la causa para no volver a repetirlo*

◙ Ejercicio

❑ La depuración de código en Java con Eclipse

- Se trata de entender cómo utilizar Eclipse como herramienta de depuración
  - ¿Cómo se define un *breakpoint y para qué sirve?*
  - *¿Cómo se inspecciona el contenido de variables?*
  - *Conocer las opciones básicas de depuración:*

    Resume:

    Terminate:

    Step into:

    Step over:

    Step return:
- Para realizar esta tarea, se podrá utilizar el paquete de software de ejemplo: gest-tienda disponible en el campus virtual