




Pruebas en sistemas orientados a objetos

Sistemas de Programas
Universidad Simón Bolívar



Agenda

2

- Introducción 
- Qué es probar software?
- Por qué necesitamos probar el software?
- Terminología de Pruebas
- Black box vs. white box testing
- Estrategia de Pruebas



Una definición

3

“Prueba de Software es la ejecución del código usando combinaciones de entradas, en un determinado estado, **para revelar defectos.**”

“Prueba de Software [...] es **el diseño e implantación de un software especial**: uno que ejercita otro software con **la intención de hallar defectos.**”

Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (1999)



En qué consisten las pruebas?

4

- Determinar **qué partes** del sistema desea probar
- Definir **valores de entrada** que aporten información significativa
- **Correr** el software con los valores de entrada
- **Comparar** los **resultados** producidos con los esperados
- (Medir características de ejecución: tiempo, memoria usada, etc)



Más profundidad sobre el tema

5

*"Prueba de Programas puede ser muy efectiva para mostrar la presencia de defectos, pero es absolutamente inadecuada **para demostrar la ausencia de defectos.**"*

Edsger Dijkstra, *Structured Programming* (1972)

A qué ayudan las pruebas: encontrar defectos

A qué no ayudan las pruebas: demostrar la ausencia de defectos



Pruebas no es ...

6

- Pruebas \neq **corrección**
 - Cuando se descubre un defecto, corrección es el proceso de eliminar el defecto
- Pruebas \neq **prueba formal** de programas
 - Las pruebas formales de correctitud son pruebas matemáticas de la equivalencia entre la especificación y el programa



Probar o no probar?

- **Usuarios aceptan defectos** como un hecho
- **Pero:**
 - Software con defectos **mata** gente – hay ejemplos en el mundo médico
 - Software con defectos es muy **costoso**
 - e.g. DS-1 Orion 3 Galileo Titan 4B (1999) – costo: \$1.6 billion (May 2002 NIST)
 - Mismo NIST: \$60 billion/year costo de defectos en el software (USA)
 - Software defectuoso causa **pérdida de datos**
 - e.g. cualquiera de los casos anteriores

...para listar sólo algunas consecuencias



- **Falla** – inhabilidad manifiesta sistema para realizar una función necesaria
 - Evidenciado por:
 - Salida incorrecta
 - Terminación anormal
 - Limitaciones de tiempo o espacio incumplidas
- **Defecto** – código incorrecto o faltante
 - Ejecución puede resultar en una falla
- **Error** – acción humana que produce un defecto
- **Problema** – error, falla, defecto



Alcance de las pruebas

- **Prueba Unitaria** – alcance: típicamente un ejecutable pequeño
- **Prueba de Integración** – alcance: un sistema o subsistema completo de componentes de software y hardware
 - Ejercita las interfaces entre componentes para demostrar que son operables en conjunto
- **Prueba de Sistema** – alcance: una aplicación completa integrada
 - Focalizada en características que están presentes sólo al nivel de todo el sistema
 - Categorías:
 - Funcional
 - Rendimiento
 - Estrés o carga



Intención (1)

10

- **Prueba dirigida a defectos** – intención: revelar defectos a través de fallas
 - Pruebas unitarias e integración
- **Prueba dirigida a Cumplimiento** – intención: demostrar que está conforme con las capacidades requeridas
 - Prueba de sistema
- **Prueba de aceptación** – intención: permitir a un usuario/cliente decidir si acepta un producto de software



Intención (2)

11

- **Prueba de Regresión** – Volver a probar un programa previamente probado, después de algunas modificaciones para asegurarse que no se hayan introducido o aparecido defectos debido a los cambios realizados
- **Pruebas de Mutación** – Introducir defectos a propósito en el software para determinar la calidad de las pruebas



Componentes de una prueba

12

- **Caso de Prueba** – especifica:
 - El estado y ambiente del programa antes de ejecutar la prueba
 - Las entradas a la prueba
 - El resultado esperado
- **Resultados esperados** – qué debe producir el programa:
 - Valores devueltos
 - Mensajes
 - Excepciones
 - Estado resultante del programa y el ambiente
- **Oráculo** – produce los resultados esperados del caso de prueba
 - Puede decidir si se satisfizo la evaluación



Pruebas por Partición

- Partición – divide el espacio de la entrada en grupos para los que se espera tengan la propiedad de que cualquier valor en el grupo producirá una falla si existe un problema en el código relacionado con la partición
- Ejemplos de prueba de partición:
 - Clase de Equivalencia – un conjunto de valores de entrada tal que si cualquier valor del conjunto es procesado correctamente (incorrectamente) entonces cualquier otro valor del conjunto sera procesado correctamente (incorrectamente)
 - Análisis de valores de borde
 - Pruebas con valores especiales



Ejecución de Pruebas

14

- **Grupo de Pruebas (Test suite)** – colección de casos de prueba
- **Administrador de Prueba (Test driver)** – clase o programa utilitario que aplica casos de prueba al programa
- **Función Parcial (Stub)** – Implantación parcial, temporal de un componente
 - Puede servir para alojar un componente incompleto o código para soportar las pruebas
- **Arnés/Armadura de Prueba (Test harness)** – un sistema de test drivers y otras herramientas para soportar la ejecución de las pruebas



Caja Oscura vs Caja Clara testing (1)

15

Prueba Caja Oscura	Prueba Caja Clara
No conoce los detalles internos del programa	Conoce la estructura interna del programa
Se conoce como pruebas basadas en responsabilidades y pruebas funcionales	Se conoce como <i>pruebas basadas en implantación</i> o <i>pruebas estructurales</i>
Objetivo: probar que tan bien el programa esta conforme a requerimientos (Cubre todos los requerimientos)	Objetivo: probar que todos los caminos del código están correctos (Cubre todo el código)



Caja Oscura vs Caja Clara testing (2)

16

Prueba Caja Oscura	Prueba Caja Clara
Sólo conoce la especificación	Requiere análisis del código fuente para diseñar los casos de prueba
Usado típicamente en integración y prueba de sistema	Usado en pruebas unitarias
Puede ser realizado por los usuarios	Típicamente hecho por los programadores



Cubrimiento del código

17

- **Cubrimiento de código** – cuánto código es ejercitado con las pruebas
- **Análisis de cubrimiento de código** = el proceso de:
 - Encontrar secciones de código no ejercitadas por casos de prueba
 - Crear casos adicionales para aumentar el cubrimiento
 - Calculando una medida del cubrimiento (que mide la calidad del test suite)
- Un **analizador de cubrimiento de código** automatiza este proceso



Medidas básicas de cubrimiento de código

18

- **Cubrimiento de instrucciones** – reporta si encontró cada instrucción ejecutable
 - Desventaja: insensitivo a ciertas estructuras de control
- **Cubrimiento de Decisiones** – reporta si las expresiones condicionales en estructuras de control evaluaron a cierto y falso
 - Conocido como **cubrimiento de ramas**
- **Cubrimiento de Condiciones** – reporta si cada sub-expresión booleana (separada por logical-and o logical-or) evalúa a cierto y falso
- **Cubrimiento de Caminos** – reporta si cada uno de los caminos posibles de cada función ha sido probados
 - Camino = secuencia única de ramas entre la entrada a la función, hasta el punto de salida



Cómo planeamos y estructuramos las pruebas de un programa de mediana envergadura?

- **Quién prueba?**
 - Desarrolladores / equipos especiales de prueba / clientes
 - Es difícil probar su propio código
- **Qué niveles de prueba necesitamos?**
 - Unidad, integración, sistema, aceptación, regresión
- **Cómo lo hacemos en la práctica?**
 - Pruebas manuales
 - Herramientas de prueba
 - Pruebas automáticas



Desarrollo guiado por pruebas

20

- Metodología de desarrollo de software
- Una práctica central de programación extrema (XP)
- **Escribir prueba, escribir código, rehacer**
- Más explícitamente:
 1. Escribir una pequeña prueba.
 2. Escribir el código suficiente que demuestre éxito de la prueba.
 3. Mejorar el código.
 4. Repetir.
- Estas pruebas son **pruebas unitarias + pruebas de aceptación**
- Siempre usarlas junto con xunit

- Nombre genérico de cualquier plataforma automática para pruebas unitarias
 - **Plataforma automática de pruebas** – provee todos los mecanismos necesarios para correr pruebas de forma que sólo lógica específica de prueba es la que se escribe
- Implantada en todos los principales lenguajes de programación:
 - JUnit – Java
 - cppunit – C++
 - SUnit – Smalltalk (el primero)
 - PyUnit – Python
 - vbUnit – Visual Basic



Utilizar los Casos de Uso y posiblemente los Contratos para elaborar los casos de prueba

- Numerar/nombrar el caso de prueba;
- Indicar el estado del sistema antes de la ejecución
- Entradas consideradas
- Salidas esperadas

Para cada escenario de uso, elaborar casos de prueba que ejerciten:

- el curso normal del caso de uso;
- los cursos opcionales;
- considerar las excepciones que puedan ocurrir



Exigencia Débil:

- Ejecutar al menos una vez cada acción condicionada de un curso normal
- Ejecutar cero, una y más veces una acción iterable de un curso normal
- Ejecutar al menos una vez cada opción
- Ejecutar al menos una vez cada excepción

Exigencia Media:

- Además de las anteriores, realizar pruebas de frontera para cada parámetro de cada evento



Referencias (1)

- **Pruebas:**
 - OO testing "bible":
Robert V. Binder: *Testing Object-Oriented Systems: Models, Patterns, and Tools*, 1999.
 - Writing unit tests with JUnit:
Erich Gamma and Kent Beck: *Test Infected: Programmers Love Writing Tests*
<http://junit.sourceforge.net/doc/testinfected/testing.htm>
 - Code coverage:
<http://www.bullseye.com/coverage.html>
 - Mutation testing:
Jezequel, J. M., Deveaux, D. and Le Traon, Y.
Reliable Objects: a Lightweight Approach Applied to Java. In IEEE Software, 18, 4, (July/August 2001) 76-83



Referencias (2)

25

- **Herramientas:**

- JUnit: <http://www.junit.org/index.htm>

- Gobo Eiffel Test:

- <http://www.gobosoft.com/eiffel/gobo/getest/>

- TestStudio:

- http://se.inf.ethz.ch/people/leitner/test_studio/