Projecte de Programació

Prueba de Programas

Prueba de Programas

La prueba de programas es el proceso de ejecución y evaluación, manual y/o automática, de un sistema informático con los siguientes objetivos:

- Eliminación de los errores de especificación, diseño e implementación

- Verificación de los requerimientos funcionales:
  - Qué funcionalidades faltan?
  - Es esto lo que el usuario quería?

- Verificación de los requerimientos no funcionales

Prueba de Programas

La prueba de programes es un proceso cíclico. Un error pasa por las siguientes fases:

1- Detección (síntomas)

2- Localización (causas)

3- Determinación de la fase en que se ha producido

4- Evaluación de la gravedad y del coste de corrección

5- Corrección

6- Comprobación (tests de regresión)

7- Volver a 1

Prueba de Programas

Detección de errores

- La EXPERIENCIA es FUNDAMENTAL
- Hay dos técnicas fundamentales:
  - Verificación formal (*program proving*)
  - Verificación experimental (*program testing*)

Prueba de Programas

Verificación formal

- Demostración *formal* de que el programa satisface la especificación

- Problemas:
  - Requiere conocimientos muy específicos, es difícil y caro
  - No garantiza la ausencia de errores de especificación
  - No siempre disponemos de la especificación formal de todos los componentes del sistema (librerías, etc.)

- Suele usarse típicamente en fragmentos críticos del sistema o en sistemas muy críticos

Prueba de Programas

Verificación experimental o program testing

- Ejecución del programa i comprobación de que funciona tal como se espera

- Más barato y sencillo que la verificación formal

- Problemas:
  - Las pruebas hay que hacerlas manualmente
  - Es imposible probar *todas* las posibles combinaciones de entradas y estados, pasar por todos los *if-else*, etc.
  - Cómo elegir un subconjunto suficientemente **representativo** de todos los estados?
  - No garantiza la ausencia total de errores, de ningún tipo

Prueba de Programas: Program Testing

Estrategias de prueba

Juegos de Prueba: Subconjunto "representativo" con el que probar

Un buen juego de pruebas es aquel que tiene una alta probabilidad de encontrar un error -> para escogerlo, se pueden seguir diversas estrategias

Lo más usado son los métodos de "caja": el software es visto como una *caja* con entradas y salidas
Tres estrategias básicas:

- Caja **blanca**
- Caja **negra**
- Caja **gris**

Prueba de Programas: Program Testing

Caja blanca

Quien hace las pruebas (programador/analista) conoce la estructura interna del código:

- Puede forzar la ejecución al menos una vez de todas las instrucciones: condicionales, casos límite en los bucles, etc.

- Puede definir cadenas estilo "variable-instrucciones donde aparece" (*dataflow*) -> hay que ir a buscar los casos **límite**, asegurándose de que los casos **representativos** funcionan

- Normalmente se usa sólo a nivel de pruebas de componentes específicos

Prueba de Programas: Program Testing

Caja negra

Quien hace las pruebas NO conoce la estructura interna del código, tiene acceso a su interficie:

- Se generan las entradas y las salidas que deberían corresponder a esas entradas

- Conviene definir una relación de equivalencia entre todos los posibles estados del sistema -> Hay que ir a buscar los casos **representativos** y vigilar los casos **límite**

- Normalmente se hacen después de les pruebas con caja blanca

Prueba de Programas: Program Testing

Caja gris

Mezcla de los dos estrategias anteriores. Quien hace las pruebas conoce la estructura interna del código:

- Las pruebas se hacen en modo caja negra: se generan las entradas y se comprueba que las salidas son las que corresponden

- Las entradas y salidas pueden ser más precisas, pues pueden obligar al código a pasar por donde interese

- Se pueden focalizar las entradas en los **puntos críticos**

Prueba de Programas: Program Testing

En OOP tendremos varios niveles de pruebas:

- Pruebas de *componentes*: pruebas de (los métodos de) las clases aisladamente

- Pruebas de *integración*: pruebas de les interacciones entre componentes. Probamos conjuntos de clases, posiblemente agrupados por casos de uso o funcionalidad

- Pruebas del *sistema*: pruebas del programa como un todo

- Pruebas de *integración del sistema*: el programa puede no estar solo, sino formar parte de un sistema más grande, en el que hayan otros programas

Prueba de Programas: Program Testing

Pruebas de componentes: ***Drivers*** y ***Stubs***

Las pruebas de componentes tienen como objetivo comprobar el funcionamiento correcto de una clase. Pero la mayoría de las clases no están aisladas

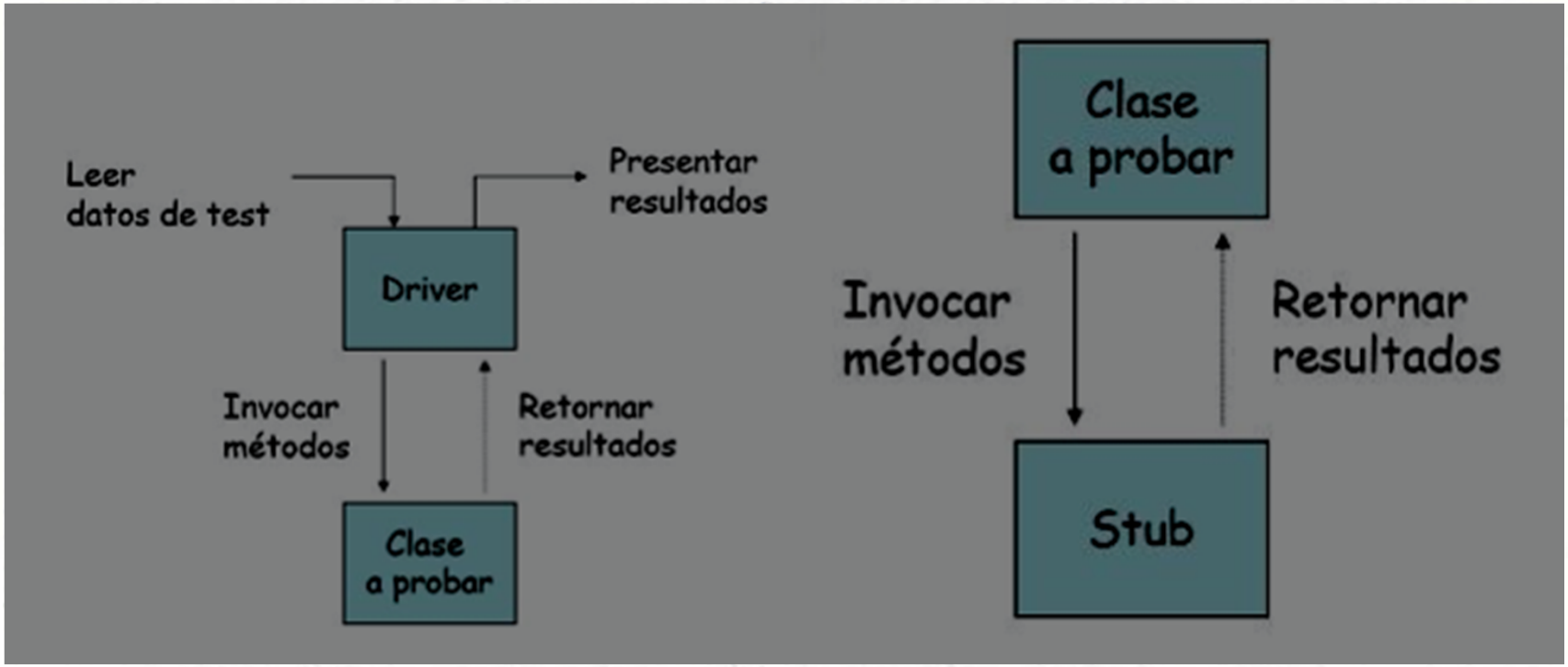
Supongamos que hemos implementado una clase y queremos probar sus métodos:

- Necesitamos una clase que invoque estos métodos: ***Driver***

- Si la clase que estamos probando usa otras clases, necesitamos probarla sin tener la implementación de estas otras clases: ***Stubs***

Prueba de Programas: Program Testing

Pruebas de componentes: ***Drivers*** y ***Stubs***



Prueba de Programas: Program Testing

Pruebas de componentes: ***Drivers*** y ***Stubs***

- Para cada clase a probar tenemos que tener un *driver*. Los *stubs* sólo han de estar si se necesitan

- En teoría: aunque tengamos la clase B probada, si estamos probando una clase A que usa la clase B, *hay que usar un stub para probar la clase A* (si no, no podremos asegurar completamente que el error esté en la clase A).

- En el caso de la prueba de clases abstractas, hay que implementar una subclase *stub* con la implementación de los métodos abstractos. Eso permite crear instancias del *stub* y poder probar los métodos de la clase abstracta que sí están implementados (vía un *driver* y eventualmente otros *stubs*)

Prueba de Programas: Program Testing

Pruebas de componentes: ***Drivers*** y ***Stubs***

Supongamos que queremos probar una clase ClaseA:

```
public class ClaseA {
   private String atrib1;
   private ClaseB atrib2;

   public ClaseA (String a1, ClaseB a2) {
      atrib1 = a1;
      atrib2 = a2; }
   public String getAtrib1() { return atrib1; }
   public ClaseB getAtrib2() { return atrib2; }
   public int getCalculoAtrib2() {
      int x = atrib2.calculoComplejo();
      int y = cálculo específico de ClaseA que usa x
      return y;
       }
    }
```

(ClaseB tiene un método calculoComplejo() que devuelve un int)

Prueba de Programas: Program Testing

Pruebas de componentes: ***Drivers*** y ***Stubs***

El driver tendría la siguiente especificación:

```
public class DriverClaseA {
  public void testConstructor();
  public void testGetAtrib1();
  public void testGetAtrib2();
  public void testGetCalculoAtrib2();
  public static void main (String [] args);
}
```

Es decir, un *testMétodo* por cada *Método*, que nos permita probar que el método hace lo que se espera de él -> recordar estrategias de prueba…

Prueba de Programas: Program Testing

Pruebas de componentes: ***Drivers*** y ***Stubs***

Para acabar nos haría falta un *stub* para la clase B, con la siguiente especificación:

```
public class ClaseB {
  public int calculoComplejo();
}
```

Es decir, se define un método por cada método de ClaseB que se usa en ClaseA. La implementación ha de ser trivial:

- retornar un valor fijo o random del tipo de retorno

```
return 42;
```

- escribir un mensaje de paso por el método

```
System.out.println("llamada método X con parám Y");
```

- …

Prueba de Programas: Program Testing

Pruebas de integración

Las pruebas de integración tienen por objetivo comprobar el funcionamiento correcto de un conjunto de clases consideradas como un grupo (normalmente porque están dentro del mismo caso de uso o funcionalidad)

NO es posible probar todas las clases de golpe (*big bang*): la integración hay que hacerla por grupos pequeños de clases e INCREMENTALMENTE

Los juegos de prueba de las funcionalidades principales en versión "realista" se pueden considerar pruebas de integración

Prueba de Programas: Program Testing

Pruebas de integración: Enfoques

- *Bottom-Up*
  - Las clases de nivel más bajo son las *primeras* que se prueban
  - El programa no se prueba hasta el final

- *Top-Down*
  - Las clases de nivel más bajo son las *últimas* que se prueban
  - Necesitamos tener el programa completo desde el principio

- Híbridos (*Sandwich Testing*)

Prueba de Programas: Program Testing

Pruebas de integración: Enfoque Híbrido

- Híbridos (*Sandwich Testing*)

  - Mezcla de *top-down* y *bottom-up*

  - Permite empezar a probar por las clases más críticas

  - Permite desarrollar el programa de forma más flexible

Es el más adecuado para las pruebas de integración de PROP

Prueba de Programas: Program Testing

Pruebas de sistema (fuera del alcance de PROP "relativamente")

- Partimos de programa completo, la especificación de requerimientos funcionales y operativos y la documentación de usuario

- Las pruebas de sistema (entorno real) tendrán en cuenta:
  - Seguridad
  - Recuperación (caída de sistema)
  - Potencia (condiciones extremas: volumen, frecuencia, picos de demanda, etc.)
  - Eficiencia (tiempo de respuesta, consumo de recursos, etc.)
  - Interacción con otro software
  - Usabilidad, etc.

Prueba de Programas: Program Testing

Objetivos del *program testing*:

- *Pruebas de programa*: validación del código implementado por los desarrolladores (a diferentes niveles, lo visto hasta ahora)

- *Pruebas de regresión*: verificar que, después de arreglar un error, no hemos generado otros errores o que partes del programa que se consideraban probadas no dejen de funcionar

- *Tests de aceptación*: validación del programa completo

Prueba de Programas: Program Testing

Pruebas de regresión: frecuentemente, añadir una funcionalidad o arreglar un error provoca que partes del programa que se consideraban probadas dejen de funcionar

Cómo gestionar estas comprobaciones?

- Ejecutar TODAS las baterías de pruebas ya superadas en etapas anteriores cada vez que se introduce un cambio

- Sería bueno tener una herramienta semi-automática para facilitar esta tarea ⇒ JUnit (en Java)

Prueba de Programas: Program Testing

Tests de aceptación: validación del programa completo

Un par de comentarios, ya que esto queda fuera de alcance de PROP:

- Realizado por el cliente o usuario final

- Pruebas de aceptación controladas:

  - *Alfa*: ejecución del programa en un entorno controlado por un equipo independiente (a veces el programador también está presente)

  - *Beta*: ejecución del programa en un entorno real por un número limitado de usuarios -> *feedback*

Prueba de Programas

Comentarios:
- Los *stubs* y los *drivers* no forman parte del programa: hay que retirarlos al final
- Se escribe MUCHO más código del que habrá en la versión final
- Pasar tiempo diseñándolos NO es tiempo perdido

Consejos:
- Prueba pronto: ahorrarás tiempo
- Escribe tests que prueben propiedades *bien definidas*
- Escribe tests que se entiendan
- Documenta los tests
- Usa herramientas (p.ej. JUnit)
- Escribe los tests *antes* que el código
- Escribe *mucho* código para hacer pruebas

Prueba de Programas

Consideraciones específicas para PROP

Primera entrega

1) Prueba de componentes:
- Hay que implementar un *driver* para TODA clase entregada
- Los *drivers* tienen que ser INTERACTIVOS:
  - No tiene que hacer falta recompilarlos para probar con nuevos datos
  - Entrada de datos por teclado o ficheros de texto
- Cuando sea necesario, implementar los *stubs* correspondientes

2) Pruebas de integración:
- Para probar las funcionalidades principales que se piden implementadas -> también vía un *driver*

3) JUnit: como mínimo para una de las clases (consultar con tutor!)

Ver el **formulario** de descripción de juegos de prueba en el documento de Normes dels Lliuraments

Prueba de Programas

Consideraciones específicas para PROP

Segunda entrega

1) Prueba de componentes:
- Interna del grupo, ya no se piden más *drivers* ni *stubs*

2) Pruebas de integración:
- Se piden las pruebas de integración del Proyecto Completo: Juegos de Prueba "realistas" (en lo posible con datos reales, algoritmos completos y eficientes) -> se solapan parcialmente con pruebas de sistema/validación