Testing software orientado a objetos

Nancy Hitschfeld Kahler

Departamento de Ciencias de la Computación

Universidad de Chile

Contenido

- Validación de software
  - Métodos de testing
  - Testing y herencia
- Marco para probar software orientado a objetos
  - Testing del comportamiento de los objetos
  - Testing estático
- JUnit (Java)

Validación de software

- Calidad del sofware:
  - Correctitud: satisface requerimientos funcionales del usuario
  - Robustez: responde adecuadamente bajo circunstancias anormales

"*Validación es en la práctica generalmente restringido a aspectos de la funcionalidad y de la interfaz del usuario. Sin embargo, las propiedades estructurales también son importantes*"

Validación de software

- Criterios estructurales
  - Mantención: software fácil de adaptar
  - Reuso: posee componentes reusables
  - Compatibilidad: componentes compatibles con otras para construir nuevos sistemas(plug-in's)

"*Testing es el proceso de juzgar experimentalmente si un sistema y sus componentes satisfacen los requerimientos funcionales. Este proceso involucra en general la ejecución del programa y no métodos formales de prueba.*"

Cómo verificar criterios estructurales?

Métodos de Testing

- Testing involucra:
  - Casos de prueba: dado una cierta entrada esperamos un cierto resultado (input-output behavior)
  - Procedimientos de prueba: especifica los pasos a seguir para validar un programa usando los casos de prueba
    - Casos de prueba
    - Orden de ejecución
    - Cuándo parar de probar?

"*En el caso de software orientado a objetos se debe diseñar un proceso para probar cada uno de los métodos de un objeto con el objetivo de descubrir errores de implementación en los métodos o en el estado del objeto, o ambos*"

Métodos de testing

- *Una buena práctica de testing intenta minimizar a)* ***el esfuerzo de producir tests*** *(en términos de tiempo y costo), b)* ***el número de tests*** *y c)* ***el esfuerzo en ejecutarlos****, y al mismo tiempo maximizar* ***el número de errores detectados*** *y* ***la confiafibilidad del software*** *(en términos de el número de tests pasados exitosamente)*
- *Es imposible decidir con absoluta certeza si un software está completamente libre de errores*
- *Cuándo decidir parar de probar?*
  - Después de detectar N errores
  - Si la razón entre el número de errores y el tiempo dedicado a buscarlos es lo suficientemente pequeño
  - …

Métodos de testing

- Caja negra (Black-box testing): prueba funcional
  - Clases de equivalencia: elegir un representante de la clase
  - Extremos
- Caja Blanca (White-box testing): prueba estructural
  - Cubrir instrucciones: cada instrucción es ejecutada al menos una vez
  - Cubrir condiciones: cada condición debe hacerse verdadera y falsa

Métodos de testing

- Ejemplo 1: double sqrt(double x), x >=0
- Clases de equivalencia:
  - Números negativos]…,0[. Elegir, por ejemplo el número -2
  - Números positivos]0,1[. Elegir, por ejemplo el número 0.5
  - Números positivos]1,…[. Elegir, por ejemplo el número 2
  - Número 1
- Extremos: 0, 1
- Ejemplo 2: Definir casos de prueba para Black y White-box testing clases de Quicksort sobre un arreglo de tamaño N.

Ciclo de testing

Para cualquier metodología:

- Prueba de módulos
- Pruebas de integración
- Pruebas funcionales
- Pruebas del sistema
  - Usabilidad: es satisfactoria la interfaz al usuario?
  - Volumen: cómo responde a grandes volúmenes de datos?
  - Stress: reacciona bien con una red cargada?
  - Seguridad: provee un nivel suficiente de protección?
  - Desempeño: es adecuado en los casos comunes?
- Pruebas de aceptación

Testing y herencia

- Cuánto probar una clase que no se modifica? *Una de las afirmaciones más conocidas es que orientación a objetos permite que el código heredado puede ser reusado fácilmente y confiablemente. Esta afirmación sugiere que métodos heredados no deberían ser probados nuevamente aplicados sobre objetos de la clase derivada.*

-
```
class A{ // n>=0
  public  A(){ n=0;}
  public int value(){ return next(n);}
  public void strange(){ next(-3);}

  protected int next(int i){ n=n+i*i;}
  protected int n;
 }
```

```
class B extends A{
    public B(){}
    protected  int next(int i){
                return n= n+ (n+1)*i;
    }
```

Testing y herencia

- Aplicando los métodos de la clase A a instancias de ella: sus instancias tienen siempre un estado >=0

```
A a= new A();
int valor = a.value();
a.strange();
valor = a.value();
```

- Aplicando los métodos heredados de A sobre instancias de la clase B: éstas pueden tener un estado negativo

```
A a= new B();
int valor = a.value();
a.strange();  // n < 0
valor = a.value();
```

"*El uso de contratos en forma explícita permite detectar este tipo de errores fácilmente* "

Metodologías de diseño y programación

Testing y herencia

- Resumen: El código heredado debe ser testeado nuevamente sobre objetos definidos usando las clases derivadas porque:
  - Métodos de las subclases pueden modificar variables de instancia de una clase base
  - Métodos de la clase base pueden usar métodos re-definidos en las clases derivadas

Marco para probar software orientado a objetos

Niveles de testing:

- clases --- interacción entre métodos y variables de instancia
- clusters --- interacción entre grupos de clases
- sistemas --- probar todas las clases

Marco para probar software orientado a objetos

- Testing de una clase: orden sugerido de testing de acuerdo a los tipos de métodos
  1. Creación
  2. Selectores: retornan el estado
  3. Predicados: retornan boolean
  4. Modificadores: modifican el estado
  5. Destrucción
  6. Iteradores

• *Necesidad de tener conocimiento de la estructura interna del objeto, es decir, de su estado y como cambia*
• *Necesidad de técnicas para hacer testing del comportamiento de los objetos*

Testing el comportamiento de los objetos

Entre las técnicas existentes están**:**

Transición de estados (Matriz de transición): técnica para probar una clase/objeto con pocos estados y métodos

- Clases de equivalencia: inputs que generan distintos estados del objetos
- Casos extremos: bordes (incluir parámetros de los métodos)

La matriz de transición ayuda a visualizar el efecto de cada llamada a un método sobre el estado. Los parámetros van en otra dimensión de la matriz

- ***Transiciones de identidad:*** técnica que consiste en identificar secuencias de llamadas a métodos que lleven a un mismo estado (axiomas dentro de la descripción den un TDA)

Testing el comportamiento de objetos

```
Class Counter{
  protected:
     int value;
     virtual int invariante(){
           return   0<=value;}
  public:
     Counter(int x){value=x;
           assert(invariante());}
     void reset(){ value=0;
           assert(invariante());}
     void incr(int i){
           assert( i>0);
           value+=i;
           assert(invariante());}
     void decr(int i){
           assert( i>0);
           value-=i;
           assert(invariante());}
     int value(){ return value;}
}
```

```
Class BoundedCounter: public Counter{
     protected:
          int max_value;
          virtual int invariante(){
            return   0<=value && value <= max_value;}
     public:
          BoundedCounter(int x, int max){
               value=x; max_value = max;
               assert(invariante());}
          boolean isDone(){ return value == max_value;}
}
```

Testing el comportamiento de objetos

Counter: número de estados = 2 (0 y >0)
BoundedCounter: número de estados = 3 (0,]0,max[, max)

- *Es importante considerar los parámetros de los métodos pues puede influir en el cambio del estado del objeto.*

- *Los errores pueden ser considerados como un estado más (producidos por un resultado erróneo o un estado ilegal).*

Como detectarlos:

- *Dentro del objeto:* contratos
- *Durante la interacción de muchos objetos:* protocolos de interacción

- *Cuáles casos probar?* detectar casos límite: inputs que producen un cambio de estado y inputs que mantienen el estado

Testing el comportamiento de objetos

Matrices de transición para las clases Counter y BoundedCounter

Counter	(1) value== 0	(2) value >0
incr(i)	2	2
decr(i)	error	error, 1, 2
value()	1	2

BoundedCounter	(1) value==0	(2) 0<value<max	(3) value==max
incr(i)	2,3,error	2,3,error	error
decr(i)	error	error, 1, 2	error, 1,2
value()	1	2	3

Testing el comportamiento de objetos

- Ejemplos de transiciones de identidad para la clase Counter:
  - Counter c(0); int n;
  - n=c.value(); c.incr(3); c.incr(4); c.decr(7); => n==c.value();
  - Counter c(10);
  - n=c.value(); c. decr(4); c.incr(4); => n==c.value();

- En general, probar tambien con secuencias para producir errores y notar que estos son detectados por el contrato:
  - Counter c(5); n=c.value(); c.incr(-5);
  - Counter c(5); n=c.value(); c.decr(6);

Testing el comportamiento de objetos

- Cómo detectar errores en la interacción de múltiples objetos?

  - contratos --- más orientados a una clase en si, pero las precondiciones ayudan en esta etapa tambien a detectar problemas en la interacción entre objetos
  - transiciones de identidad --- al involucrar multiples objetos

Prueba estática

- Leer cuidadosamente el programa
- Inspección de código (mejor en grupo)
- Walkthroughs. Simular un cálculo leyendo las partes del código relacionadas
- Probar formalmente que el programa es correcto

JUnit (Java)

- Permite testear clases automáticamente
- Al implementar una clase:
  - Diseñar sus casos de prueba para probar su correctitud
  - El disponer de un conjunto de casos de prueba es muy útil cuando se cambia la implementación: permite verificar si lo que funcionaba antes aún funciona
- Cómo hacerlo?
  - Crear para cada clase una clase asociada que contiene los casos de prueba
  - Cada caso de prueba se pone en un método cuyo nombre comienza con “test”
    - *Java genera automáticamente el esqueleto de las clases de prueba*

JUnit (Java)

Cómo se recomienda probar la siguiente clase?

```
public class Day{
    public Day(int aYear, int aMonth,
                    int aDate){
        julian = toJulian(aYear, aMonth,
                                aDate);
    public Day addDays(int n){
        ….}
    public int daysFrom(Day other){
        …}
    …
    private int julian; //julian day number
    // numero de días desde el 1 de
    // enero de 4713 antes de cristo

}
```

```
import junit.framework.*;
public class DayTest extends TestCase{
     public void testAdd(){
            Day d1 = new Day(1970,1,1);
            int n= 1000;
            Day d2 = d1.addDays(n);
            assert  d2.daysFrom(d1) == n;
    }
    public void testDaysBetween(){…}
}
```

Para compilar de necesita el archivo junit.jar

Javac –classpath .:junit.jar –ea junit.swingui.TestRunner DayTest