ESTRUCTURAS DE DATOS Y ALGORITMOS

TECNÓLOGO EN INFORMÁTICA

VERIFICACIÓN, TEST Y DEBUGGING

1. INTRODUCCIÓN

Podemos decir que un programa funciona cuando:

- ✓ Hace lo que debe hacer. Esto es que cumpla con las especificaciones que teníamos descritas.
- ✓ No hace lo que no debe hacer.

Validar el correcto funcionamiento de un programa es un proceso muy complejo, aquí solamente nos referiremos a la realización de *pruebas* (test) para un subconjunto de *casos*. El numero de de casos de prueba dependerá de la complejidad del programa o su funcionalidad y generalmente no es razonable la realización de test para todas las posibles entradas y salidas (en algunos casos resultará imposible). La elección de dichos casos es una tarea delicada que nos permitirá aumentar la confianza en el programa que estamos desarrollando.

Los *casos de prueba* se pueden escribir en papel y ejecutarlos disciplinadamente de forma manual, estrategia que silo es viable para programas pequeños. En programación profesional conviene automatizar las pruebas de forma que se pueden realizar muchas veces, tanto durante el desarrollo (hasta estar satisfechos), como durante la aceptación (para dar el programa por correcto) y también si en el futuro hay que modificar el programa (para cerciorarnos de que no hemos roto que antes funcionaba).

A la hora de localizar un error es cuando se aprecia un buen diseño. En un programa claro puede ser más fácil detectar dónde y cuándo se pierde el control.

2. TEST

Podemos considerar las siguientes fases de prueba (no disjuntas entre sí en su aplicación):

- o Pruebas de Unidades: como caja negra y caja blanca.
- o Pruebas de Integración.

La *prueba de unidades* consiste en ir probando uno a uno los diferentes módulos que constituyen una aplicación.

Las *pruebas de integración* se centran en probar la coherencia semántica entre los diferentes módulos, tanto de semántica estática (se importan los módulos adecuados; se llama correctamente a los procedimientos proporcionados por cada módulo), como de semántica dinámica (un módulo recibe de otro lo que esperaba). Normalmente estas pruebas se van realizando por etapas, englobando progresivamente más y más módulos en cada prueba.

Las *pruebas de integración* se pueden empezar en cuanto tenemos unos pocos módulos, aunque no terminarán hasta disponer de la totalidad. En un diseño descendente (*top-down*) se empieza a probar por los módulos más generales; mientras que en un diseño ascendente se empieza a probar por los módulos de base.

El planteamiento *descendiente* tiene la ventaja de estar siempre pensando en términos de la funcionalidad global; pero tiene el inconveniente de que para cada prueba hay que hacer implementaciones "dummy" de los módulos inferiores que aún no están disponibles, es decir implementaciones simplificadas que respetan la interfaz pero no el comportamiento definitivo (por ejemplo una función que no compute el resultado correcto sino que retorne siempre un mismo valor constante). El planteamiento *ascendente* evita esta dificultad, pues vamos construyendo pirámides más y más altas a medida que avanzamos en la implementación.

2.1. Prueba de Unidades[1] [2]

2.1.1. Cajas Negras

El término "caja negra" (black box) es utilizado pues se analiza el comportamiento de los módulos desde el punto de vista entradas-salidas. Los casos de prueba surgen solamente del análisis de la especificación, sin importar los detalles de implementación. Sería ideal que éstos se realizaran antes de comenzar con la implementación del módulo para tratar de mantener la "independencia" de dicha implementación. Una ventaja de estos casos de prueba es su robustez frente a cambios en la implementación.

Al examinar las especificaciones de un módulo, se puede seguir con una técnica algebraica conocida como "clases de equivalencia". Esta técnica trata cada parámetro como un modelo algebraico donde unos datos son equivalentes a otros. Si logramos partir de un rango excesivamente amplio de posibles valores reales a un conjunto reducido de clases de equivalencia, entonces es suficiente probar un caso de cada clase, pues los demás datos de la misma clase son equivalentes.

El problema está pues en identificar clases de equivalencia, tarea para la que no existe una regla de aplicación universal; pero hay recetas para la mayor parte de los casos prácticos:

- Si un parámetro de entrada debe estar comprendido en un cierto rango, aparecen 5 clases de equivalencia: por debajo, borde inferior, adentro, borde superior y por encima del rango.
- Si una entrada requiere un valor concreto, aparecen 3 clase de equivalencia: por debajo, el valor y por encima.
- Si una entrada requiere un valor entre los de un conjunto, aparecen 2 clase de equivalencia: en el conjunto o fuera de él.
- Si una entrada es booleana, hay 2 clases: si o no.

Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

También nos encontraremos con una serie de valores singulares, que marcan diferencias de comportamiento del módulo. Estos valores son claros candidatos a marcar clases de equivalencia: por abajo y por arriba.

Una vez identificadas las clases de equivalencia significativas, se procede a tomar un valor de cada clase, que no esté justamente al límite de la clase. Este valor aleatorio, hará las veces de cualquier valor normal que se le pueda pasar en la ejecución real.

No hay que olvidarse tampoco de considerar los llamados "casos de borde", son los puntos de cambio de clases de equivalencia. Conviene probar 2 valores por frontera, uno justo por debajo y otro justo encima.

2.1.2. Cajas Blancas

Aquí lo que vamos a hacer es generar casos de prueba analizando la implementación del módulo, trataremos de que se recorran todos los posibles caminos del flujo de ejecución del programa.

---

[1] Liskov, Barbara; Guttang Jhon; "Abstraction and Specification in Program Development", MIT Press.
[2] Mañas, José A.; Apuntes sobre Reparacion de programas, Dept. de Ing. De Sistemas Telemáticos, Universidad Politécnica de Madrid.

Al igual que antes, en algunos casos, cubrir todos los casos (o caminos) posibles es imposible (o económicamente inviable).
Aquí también se introduce la idea de "cobertura" que se utiliza para indicar cuanto código (del escrito para el módulo) hemos cubierto (o recorrido).

2.1.2.1. Cobertura de segmentos

A veces también denominada "cobertura de sentencias". Por segmento se entiende una secuencia de sentencias sin puntos de decisión. Como el ordenador está obligado a ejecutarlas una tras otra, es lo mismo decir que se han ejecutado todas las sentencias o todos los segmentos.

2.1.2.2. Cobertura de ramas

La cobertura de segmentos es engañosa en presencia de segmentos opcionales. Por ejemplo:

```
if (condicion){
      segmento
};
```

Desde el punto de vista de cobertura de segmentos, basta ejecutar una vez, con éxito en la condición, para cubrir todas las sentencias posibles. Sin embargo, desde el punto de vista de la lógica del programa, también debe ser importante el caso de que la condición falle (si no lo fuera, sobra el `if`). Sin embargo, como en la rama `else` no hay sentencias, con 0 ejecuciones tenemos el 100%.
Para afrontar estos casos, se plantea un refinamiento de la cobertura de segmentos consistente en recorrer todas las posibles salidas de los puntos de decisión. Para el ejemplo de arriba, para conseguir una cobertura de ramas del 100% hay que ejecutar (al menos) 2 veces, una satisfaciendo la condición, y otra no.
Estos criterios se extienden a las construcciones que suponen elegir 1 de entre varias ramas. Por ejemplo, el `switch`. En este caso se deben diseñar pruebas que ejerciten todas las ramas del `switch`, incluyendo el `default`.
Nótese que si lográramos una cobertura de ramas del 100%, esto llevaría implícita una cobertura del 100% de los segmentos, pues todo segmento está en alguna rama.

2.1.2.3. Cobertura de decisiones

La cobertura de ramas resulta a su vez engañosa cuando las expresiones **booleanas** que usamos para decidir por qué rama ir son complejas. Por ejemplo:

```
if (condicion1 || condicion2){
      segmento
};
```

Las condiciones 1 y 2 pueden tomar 2 valores cada una, dando a lugar 4 posibles combinaciones. No obstante hay dos posibles ramas y bastan 2 pruebas para cubrirlas. Pero con este criterio podemos estar cerrando los ojos a otras combinaciones de las condiciones.
Consideremos sobre el caso anterior las siguientes pruebas:

i. Prueba 1: *condicion1* = **true** y *condicion2* = **false**
ii. Prueba 2: *condicion1* = **false** y *condicion2* = **true**
iii. Prueba 3: *condicion1* = **false** y *condicion2* = **false**
iv. Prueba 4: *condicion1* = **true** y *condicion2* = **true**

Bastan las pruebas *ii* y *iii* para tener cubiertas todas las ramas. Pero con ellos sólo hemos probado una posibilidad para la *condicion1*.

Para afrontar esta problemática se define un criterio de cobertura de decisión que divide las expresiones **booleanas** complejas en sus componentes e intenta cubrir todos los posibles valores de cada uno de ellos.
Nótese que no basta con cubrir cada una de las condiciones componentes, sino que además hay que cuidar de sus posibles combinaciones de forma que se logre probar todas y cada una de las ramas. Así, en el ejemplo anterior no basta con ejecutar las pruebas *i* y *ii*, pues aún cuando cubrimos perfectamente cada posibilidad de cada condición por separado, lo que no hemos logrado es recorrer las dos posibles ramas de la decisión combinada. Para ello es necesario añadir la prueba *iii*.
El conjunto mínimo de pruebas para cubrir todos los aspectos es el formado por las pruebas *iii* y *iv*. Aún así, nótese que no hemos probado todo lo posible. Por ejemplo, si en el programa nos equivocamos y ponemos **&&** (and) donde queríamos poner **||** (or) –o viceversa-, este conjunto de pruebas no lo detecta. Solo queremos decir que la cobertura es un criterio útil y práctico; pero no es prueba exhaustiva.

Una dificultad añadida surge cuando las condiciones no son mutuamente excluyentes. Por ejemplo, puede ocurrir que *condicion1* implique *condicion2*. Esto se da cuando si *condicion1* es **true**, *condicion2* también lo es, pero si *condicion1* es **false**, *condicion2* puede ser **true** o **false**.

Aparte de las razones apuntadas en los casos anteriores que imposibilitan alcanzar el 100% de cobertura, en este caso puede haber condiciones que nunca/siempre se evalúen a **true** o **false**. Estos casos suelen indicar la presencia de algún error: o bien la sentencia `if` es totalmente innecesaria o bien existe un error en la condición. Correspondientemente, en el caso del `switch` su expresión asociada puede no evaluarse a ciertos valores.

2.1.2.4. Cobertura de bucles

Los bucles no son más que segmentos controlados por decisiones. Así, la cobertura de ramas cubre plenamente la esencia de los bucles. Pero eso es simplemente la teoría, pues la práctica descubre que los bucles son una fuente común de errores. Un bucle se ejecuta un cierto número de veces; pero ese número de veces debe ser muy preciso, lo más normal es cometer el error y ejecutarlo una vez de menos o una vez de más con consecuencias indeseables.

Para un bucle de tipo *while* hay que pasar 3 pruebas

- 0 ejecuciones
- 1 ejecución
- Más de 1 ejecución (pueden ser 2)

Para un bucle de tipo *do-while* hay que pasar 2 pruebas

- 1 ejecución
- Más de 1 ejecución (pueden ser 2)

Los bucles ***for*** basta pues con ejecutarlos 1 vez. No obstante, conviene no engañarse con los bucles ***for*** y examinar su contenido pues si dentro de él hay bifurcaciones hay que recorrerlo tantas veces como sea necesario para que se alcancen todas las condiciones o secciones de código.

2.1.3. Probando programas recursivos

Aquí conviene que se incluyan casos de prueba que provoquen que el programa ejecute solamente su paso base y otro caso que ejecute al menos un paso recursivo. Por supuesto en cada una de estas pruebas hay que agregar tantos casos como sean necesarios para lograr cubrir la mayor cantidad de segmentos o ramas de cada uno de los pasos del programa recursivo.

2.1.4. Probando tipos abstractos de datos

Hasta ahora consideramos para la generación de los casos de prueba a todo el programa por igual. Ahora vamos a trabajar con las funciones agrupadas en categorías como *Constructoras, Predicados y Selectoras*, y vamos a tener en cuenta la interacción entre categorías para analizar primeramente las básicas y luego las que dependen de ellas. Por ello, en general, deberemos probar las *Constructoras* primero y luego continuar con los *Predicados y Selectoras.*

2.2. Pruebas de Integración

Durante las pruebas de unidades nos concentramos en un módulo y su correcto funcionamiento, independientemente del resto. Ahora debemos probar la interacción entre los módulos y que todo siga funcionando correctamente.

Estas pruebas se pueden planear desde un punto de vista *estructural o funcional.*

Las *pruebas estructurales de integración* son similares a las pruebas de caja blanca; pero trabajan a un nivel conceptual superior. En lugar de referirnos a sentencias del lenguaje, nos referiremos a llamadas entre módulos. Se trata pues de identificar todos los posibles esquemas de llamadas y ejercitarlos para lograr una buena cobertura de segmentos o ramas.

Las *pruebas funcionales de integración* son similares a las pruebas de caja negra. Aquí trataremos de encontrar fallos en la respuesta de un módulo cuando su operación depende de los servicios prestados por otro(s) módulo(s). Según nos vamos acercando al sistema total, estas pruebas se van basando más y más sobre las especificaciones.

3. TÉCNICAS BÁSICAS DE DEPURACIÓN

Luego que detectamos un comportamiento incorrecto, tenemos que determinar el punto exacto del error. En este momento es cuando se aprecia una buena estructuración del programa.
Para facilitar dicha tarea se pueden utilizar técnicas llamadas "programación defensiva" y que son:

- Aserciones
- Trazas de valores
- Trazas de flujo de ejecución

La utilización de trazas y aserciones, en conjunto con la utilización de programas especialmente diseñados para depurar código (llamados *Debuggers*), nos van a facilitar la tediosa tarea de la localización y reparación del fallo.

3.1. Aserciones

La técnica de las aserciones (o afirmaciones) consiste en sembrar el código de chequeos de condiciones que suponemos deben cumplirse, de forma que si algo no es normal, lo detectemos lo antes posible, por sí mismo, en vez de tener que estar trazando marcha atrás las causas que han llevado a un fallo. Por ejemplo, puede controlarse que un puntero no valga NULL al inicio de una función que asume el puntero inicializado como precondición.

Cuando una variable toma un valor inesperado (erróneo en definitiva), el código de aserción despliega un mensaje indicando el problema y se detiene la ejecución del problema. De este modo, las consecuencias se pueden apreciar inmediatamente evitando que el programa arrastre el error durante un tiempo antes que se aprecie un fallo de ejecución o entregue un resultado final equivocado.

Las aserciones suponen un trabajo extra para el programador, trabajo que se compensa durante el desarrollo y las pruebas; pero que parece redundante cuando el programa ha sido exhaustivamente probado y simplemente queremos que produzca resultados a la mayor velocidad posible. En otras palabras, queremos eliminar las aserciones cuando entremos en producción. No obstante, si teniendo un programa en producción se detecta una situación de fallo, conviene reactivar las aserciones para poder afinar rápidamente dónde está el origen del problema.

En resumen, las aserciones deberían poder activarse y desactivarse con comodidad para afrontar con el mismo código las situaciones de depuración de errores y de producción.

3.2. Trazas de valores

Son los listados que se generan cuando ejecutamos el código y nos van informando los valores que van tomando las variables. Por supuesto que acá también el programador tiene que escribirlas y determinar qué variables quiere examinar. Hay que tener cuidado con la información a imprimir, pues de tratarse de una variable de tipo compuesto (*struct*), no la podemos imprimir simplemente con una instrucción (por ej. *printf*) podríamos armarnos una subrutina que se encargue de ello.

3.3. Trazas de flujos de ejecución

Al igual que las anteriores, éstas producen listados con marcas que nos indican por donde o que secciones del código se han ejecutado. También aquí es conveniente determinar qué secciones del código son relevantes pues no resulta muy razonable dejar trazas para cada línea de programa.
Es muy útil utilizar la identación en las marcas que escribimos pues facilitarán la lectura de los listados.

Para las trazas es conveniente poder fijar un nivel de detalle, desde cero o sin trazas hasta un nivel de máximos detalles.

4. TEST SUITE

Para realizar las pruebas lo mejor es realizar una serie de programas auxiliares (que llamaremos *Test Suite*) los cuales nos ayudaran a realizar estas pruebas en forma automática, esto es sin intervención del usuario.

4.1. Programas con I/O de dispositivos estándar

Para los programas que interactúan con el usuario podemos realizar scripts o batch (.bat) para testearlos. Recordemos que la entrada estándar (la capturada por defecto por los procedimientos *scanf* y variantes –de stdio.h- o *cin* –de iostream-) se puede redirigir desde el DOS al invocar un programa utilizando el carácter "<" seguido del nombre del archivo que contiene las entradas (recuerde que para que este archivo funcione con los ejecutables generados por el *compilador g++* debe estar en formato UNIX y no DOS o MAC [3]).

También podemos redirigir la salida estándar o a pantalla (la generada por los procedimientos *printf* y variantes –de stdio.h- o *cout* –de iostream-) a un archivo, aquí al invocar el ejecutable debemos seguirlo del carácter ">" y el nombre del archivo de destino, si el archivo ya existía

[3] La diferencia principal entre éstos formatos es como trabajan el fin de línea. Para DOS cada línea de un texto termina con los códigos 0Dh y 0Ah, el formato MAC solamente utiliza el 0Dh y el formato UNIX sólo 0Ah. Por ejemplo utilizar el editor de textos **Notepad++** que se encuentra en: http://notepad-plus.sourceforge.net/ .

borra su contenido. Si en lugar del "**>**" utilizamos "**>>**" lo que se hace es concatenar al final del archivo.

Y por último también se puede redirigir la salida estándar de error (la generada por el método *cerr* y la macro *assert*), que sea realiza de forma análoga al anterior pero hay que colocar el digito "2" antes del carácter "**>**".

Resumiendo:

Mi_programa < *Arch_entrada* > *Arch_salida* 2>*Arch_Err*

Veamos un ejemplo:

```
// prod1.cc
// Realiza el producto de dos numeros ingresados por teclado
// Para salir el numero b debe ser 0
// El numero B debe ser menor o igual a 10

#include <iostream>
#include <assert.h>
using namespace std;

int main() {

      int a, b;
      bool salir = false;

      while (!salir) {
            cout << "Ingrese numero A: ";
            cin >> a;
            cout << "Ingrese numero B: ";
            cin >> b;

            if (b>10) {
                  cerr << "Numero B muy grande!" <<endl;
                  assert(b<= 10);
            }

            cout << "El producto es: " << a*b << endl<< endl;
      }

      if (b=0)
            salir = true;

      return 0;
}
```

Luego podemos genera el siguiente archivo de datos de prueba, que llamaremos *caso1.txt*:

```
2
3
5
5
7
12
```

Entonces si lo ejecutamos de la siguiente forma:

```
prod1 < caso1.txt > sal1.txt 2> err1.txt
```

veremos que los archivos generados contienen la sigiuente información:

Archivo *sal1.txt*	**Archivo** *err1.txt*

`Ingrese numero A:`
`Ingrese numero B:`
`El producto es: 6`
 	
`Ingrese numero A:`
`Ingrese numero B:`
`El producto es: 25`
 	
`Ingrese numero A:`
`Ingrese numero B:`
	`Numero B muy grande!`
	`assertion "!(b>10)" failed:`
	`file "../main.cc", line 18`

Tenga presente que no es necesario que se realicen todas la redirecciones a la vez, podríamos por ejemplo solo redirigir la salida de error para lo cual solamente habría que poner el "2>".

A los efectos de producir test automáticos tenemos la dificultad de tener que comparar la salida obtenida con una que este generada por otro programa que sí está verificado (situación poco real en la vida profesional) o con una salida generada "a mano". Es claro que también se pueden realizar programas que interpreten dicha salida y "evalúen" si los resultados son correctos o no.

Las técnicas de redirección de las salidas (de pantalla y/o error) también nos serán útiles cuando realicemos las pruebas de los módulos mediante por ejemplo Trazas y Aserciones.

4.2. Pruebas de módulos

Aquí las pruebas conviene realizarlas escribiendo programas auxiliares que evalúen distintos aspectos de los módulos a testear. Aplicando lo mostrado en las secciones anteriores podemos determinar los casos de prueba, luego escribimos los programas (o el programa) que los evalúe. En caso de falla comienza la búsqueda del error para la cual nos podemos apoyar en la utilización de Trazas y Aserciones.

Para la escritura de trazas debemos tener en cuenta que nos tiene que proporcionar información relevante al problema o error a buscar, puede ser muy útil redirigir las salidas y luego inspeccionar los archivos cómodamente desde cualquier procesador de textos.

Hay que tener presente los tipos de los datos que queremos imprimir y utilizar la función adecuada (o realizar la conversión correspondiente). Para el caso de los TADs, y al utilizar trazas de valores, puede resultar muy conveniente contar (en el modulo que lo implementa) con una función que despliegue su contenido pues en el modulo si se conoce exactamente la implementación.

Es importante que al momento de dar por finalizado el programa (esto es "lo probamos y quedamos satisfechos con su comportamiento") se eliminen todo tipo de impresión de resultados (trazas y aserciones) que interfieran con el funcionamiento pedido en las especificaciones, y en este punto hay que tener sumo cuidado al borrar (o comentar) el código extra, conviene luego de esta operación correr nuevamente los casos de prueba para verificar que todo quedó funcionando.

Por último otro tipo de pruebas que podemos realizar son las llamadas *pruebas de carga*, la idea principal es generar mediante bucles iterativos una gran cantidad de casos que impliquen la creación y liberación de porciones de memoria. Estas pruebas son buenas para los módulos que trabajan con asignación dinámica de memoria pues nos permiten detectar posibles errores en la liberación de memoria.