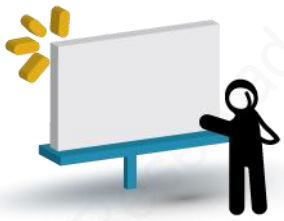
UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

Professional Testing Master

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Unidad 2: Técnicas de testing y diseño de casos de prueba (Test Cases)



Presentación:

En esta Segunda Unidad del curso, nos adentraremos en los detalles de la estrategia de testing, incluyendo los métodos de diseño de casos de prueba, y el testing en entornos especializados.

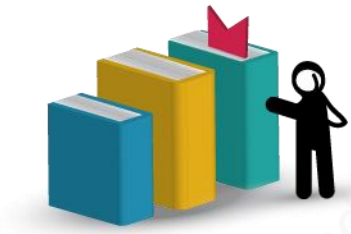
Objetivos:

Al terminar la Unidad los participantes:

Se habrán familiarizado con los diferentes tipos de cobertura de test.

Estarán en condiciones de diseñar casos de prueba efectivos usando pruebas de caja negra y caja blanca.

Conocerán los aspectos fundamentales a tener en cuenta al encarar el testing en entornos especializados.



Bloques temáticos:

1. Visión interna y externa del testing
2. Prueba de caja negra
3. Prueba de caja blanca
4. Testing especializado

Contenido

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

**** El MEC es el modelo de E-learning colaborativo de nuestro Centro.***

Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1 Visión interna y externa del testing

Cualquier producto sometido a ingeniería (y la mayoría de otras cosas) pueden probarse en una de dos formas: 1) al conocer la función específica que se le asignó, pueden llevarse a cabo pruebas que demuestren que cada función es completamente operativa mientras al mismo tiempo se buscan errores en cada función, 2) al conocer el funcionamiento interno de un producto, pueden realizarse pruebas para garantizar que “todas las piezas encajan”; es decir, que las operaciones internas se realizan de acuerdo con las especificaciones y que todos los componentes internos se revisaron de manera adecuada. El primer enfoque de pruebas considera una visión externa y se llama prueba de **caja negra**. El segundo requiere una visión interna y se denomina prueba de **caja blanca**. Ambos enfoques fueron introducidos en la Unidad 1. En ocasiones, en lugar de pruebas de caja negra y de caja blanca, se usan, respectivamente, los términos **prueba funcional** y **prueba estructural**.

Una prueba de caja negra examina algunos aspectos fundamentales de un sistema sin preocupación por la estructura lógica interna. La prueba de caja blanca del software se basa en el examen minucioso de los detalles de procedimiento. Las rutas lógicas a través del software y las colaboraciones entre componentes se ponen a prueba al revisar conjuntos específicos de condiciones y/o ciclos.

A primera vista, parecería que las pruebas de caja blanca muy extensas conducirían a “programas 100 por ciento correctos”. Lo único que se necesita es definir todas las rutas lógicas, desarrollar casos de prueba para revisarlas y evaluar resultados, es decir, generar casos de prueba para revisar de manera exhaustiva la lógica del programa. Por desgracia, las pruebas exhaustivas presentan ciertos problemas logísticos. Hasta para programas pequeños, el número de posibles rutas lógicas puede ser muy grande. Sin embargo, las pruebas de caja blanca no deben descartarse como imprácticas. Puede seleccionarse y revisarse un número limitado de rutas lógicas importantes. Puede probarse la validez de las estructuras de datos importantes.

2 Pruebas de caja negra

Como se mencionó en la Unidad 1, las pruebas de caja negra, también llamadas pruebas de comportamiento, se enfocan en los requerimientos funcionales del software; es decir, las técnicas de prueba de caja negra le permiten derivar conjuntos de valores de entrada que revisarán por completo todos los requerimientos funcionales para un programa. Las pruebas de caja negra no son una alternativa para las técnicas de caja blanca. En vez de ello, constituyen un enfoque complementario que es probable que descubra una clase de errores diferente que los métodos de caja blanca.

Las pruebas de caja negra intentan encontrar errores en las categorías siguientes: 1) funciones incorrectas o faltantes, 2) errores de interfaz, 3) errores en las estructuras de datos o en el acceso a bases de datos externas, 4) errores de comportamiento o rendimiento y 5) errores de inicialización y terminación.

A diferencia de las pruebas de caja blanca, que se realizan tempranamente en el proceso de pruebas, la prueba de caja negra tiende a aplicarse durante las últimas etapas de la prueba. Puesto que, expresamente, la prueba de caja negra no considera la estructura de control, la atención se enfoca en el dominio de la información. Las pruebas se diseñan para responder a las siguientes preguntas:

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueban el comportamiento y el rendimiento del sistema?
- ¿Qué clases de entrada harán buenos casos de prueba?
- ¿El sistema es particularmente sensible a ciertos valores de entrada?
- ¿Cómo se aíslan las fronteras de una clase de datos?
- ¿Qué tasas y volumen de datos puede tolerar el sistema?
- ¿Qué efecto tendrán sobre la operación del sistema algunas combinaciones específicas de datos?

Al aplicar las técnicas de caja negra, se deriva un conjunto de casos de prueba que satisfacen los siguientes criterios [Mye12]: 1) casos de prueba que reducen el número de casos de prueba adicionales para lograr pruebas razonables y 2) casos de prueba que dicen algo acerca de la presencia o ausencia de clases de errores, en lugar de un error asociado solamente con la prueba específica.

2.1.1 Partición de equivalencia

La partición de equivalencia es un método de caja negra que divide el dominio de entrada de un programa en clases de datos de las que pueden derivarse casos de prueba. Un caso de prueba ideal descubre por sí sólo una clase de errores (por ejemplo, procesamiento incorrecto de todos los datos de tipo texto) que de otro modo podrían requerir la ejecución de muchos casos de prueba antes de observar el error general.

El diseño de casos de prueba para la partición de equivalencia se basa en una evaluación de las clases de equivalencia para una condición de entrada. Si un conjunto de objetos puede vincularse mediante relaciones que son simétricas, transitivas y reflexivas, se presenta una clase de equivalencia [Bei95]. En palabras simples, piense en una partición de equivalencia como la clasificación de los posibles casos de prueba en distintos grupos (clases de equivalencia), cada uno de los cuales contiene los casos que son similares entre sí, es decir, prueban lo mismo o revelan el mismo tipo de error (bug). Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada. Por lo general, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición booleana. Las clases de equivalencia pueden definirse de acuerdo con los siguientes lineamientos:

1. Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos inválidas (inferior y superior al rango).
2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos inválidas (inferior y superior al valor).
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una inválida.
4. Si una condición de entrada es booleana, se define una clase válida y una inválida.

Al aplicar los lineamientos para la derivación de clases de equivalencia, pueden desarrollarse y ejecutarse los casos de prueba para cada ítem de datos del dominio de entrada. Los casos de prueba se seleccionan de modo que se revise a la vez el número más grande de atributos de una clase de equivalencia, reduciendo la cantidad de casos de prueba (lo cual por supuesto también implica un riesgo de errores no descubiertos).

2.1.2 Análisis de valor límite

Un mayor número de errores ocurre en las fronteras del dominio de entrada y no en el "centro". Por esta razón es que el análisis de valor límite (BVA, del inglés boundary value analysis) se desarrolló como una técnica de prueba. El análisis de valor límite conduce a una selección de casos de prueba que revisan los valores de frontera.

El análisis de valor límite es una técnica de diseño de casos de prueba que complementan la partición de equivalencia. En lugar de seleccionar algún elemento de una clase de equivalencia, el BVA conduce a la selección de casos de prueba en los "bordes" de la clase. En lugar de enfocarse exclusivamente en las condiciones de entrada, el BVA también deriva casos de prueba a partir del dominio de salida [Mye12].

Los lineamientos para el BVA son similares en muchos aspectos a los proporcionados para la partición de equivalencia:

1. Si una condición de entrada especifica un rango acotado por valores a y b (donde a es el mínimo válido y b el máximo), los casos de prueba deben designarse con valores a, b, justo por encima de b y justo debajo de a.
2. Si una condición de entrada especifica un número discreto de valores, deben desarrollarse casos de prueba que revisen los números mínimo y máximo. También se prueban los valores justo arriba y abajo del mínimo y máximo.
3. Aplicar lineamientos 1 y 2 a condiciones de salida. Por ejemplo, suponga que como salida de un programa de nutrición se requiere una tabla de edad contra peso. Deben diseñarse casos de prueba para crear un reporte de salida que produzca el número máximo (y mínimo) permisible de filas de tabla.
4. Si las estructuras de datos internas tienen fronteras predefinidas (por ejemplo, una tabla que tenga un límite definido de 100 entradas), asegúrese de diseñar un caso de prueba para revisar la estructura de datos en su frontera.[1]

La mayoría de los testers e ingenieros de software realizan intuitivamente BVA en cierta medida. Al aplicar dichos lineamientos, la prueba de fronteras será más completa y, por tanto, tendrá una mayor probabilidad de detectar errores.

[1] Si se aplica esto último se estaría en el caso de pruebas de "caja gris", dado que se usa una mínima información sobre la estructura interna del programa.

3 Prueba de caja blanca

La prueba de caja blanca, en ocasiones llamada prueba de **caja de cristal**, es una filosofía de diseño de casos de prueba que usa la estructura de control descripta como parte del diseño a nivel de componentes para generar casos de prueba. Al usar los métodos de caja blanca, puede derivar casos de prueba que: 1) garanticen que todas las rutas independientes dentro de un módulo se revisaron al menos una vez, 2) revisen todas las decisiones lógicas en sus lados verdadero y falso, 3) ejecuten todos los ciclos en sus fronteras y dentro de sus fronteras operativas y 4) revisen estructuras de datos internas para garantizar su validez.

3.1 Prueba de ruta básica

La prueba de **ruta**, camino o trayectoria **básica** (Basis Path Testing) es una técnica de prueba de caja blanca. El método de ruta básica permite al diseñador de casos de prueba obtener una medida de complejidad lógica de un diseño de procedimiento y usar esta medida como guía para definir un conjunto básico de rutas de ejecución. Los casos de prueba del conjunto básico garantizan ejecutar toda instrucción al menos una vez durante la prueba.

3.1.1 Notación de grafo de flujo

Antes de considerar el método de ruta básica, debe introducirse una notación para la representación del flujo de ejecución, llamada **grafo de flujo**. Este grafo se construye a partir del **pseudocódigo** o del **diagrama de flujo**. El grafo de flujo muestra el flujo de control lógico y usa la notación ilustrada en la Figura 3-1.

Figura 3-1. Notación de grafo de flujo

Cada construcción estructurada o constructo estructurado (structured construct)[2] tiene su correspondiente símbolo de grafo de flujo, como se vio en la figura.

Para ilustrar el uso de un grafo de flujo, considere la representación del diseño de procedimiento en la Figura 3-3a. Aquí se usó un **diagrama de flujo** para mostrar la estructura de control del programa. La Figura 3-3b traduce el diagrama de flujo en el grafo de flujo correspondiente (suponiendo que el diagrama de flujo no contiene condiciones compuestas en los rombos de decisión). Con referencia a la Figura 3-3b, cada círculo, llamado nodo del grafo de flujo, representa una o más instrucciones. Una secuencia de cajas de proceso y un rombo de decisión pueden traducirse en un solo nodo. Las flechas en el grafo de flujo, llamadas **aristas** o enlaces, representan flujo de control y son análogas a las flechas en el diagrama de flujo. Una arista debe terminar en un nodo, incluso si el nodo no representa ninguna instrucción (por ejemplo, vea el símbolo de grafo de flujo para la construcción if-then-else[3]). Las áreas acotadas por aristas y nodos se llaman regiones. Cuando se cuentan las regiones, el área fuera del grafo también se cuenta como región.

[2] Las construcciones estructuradas [Pre19] son secuencia, condición y repetición. La secuencia implementa pasos de procesamiento que son esenciales en la especificación de cualquier algoritmo. La condición permite elegir un procesamiento según algún suceso lógico y la repetición permite la ejecución de ciclos.
[3] En castellano si-entonces-sino.

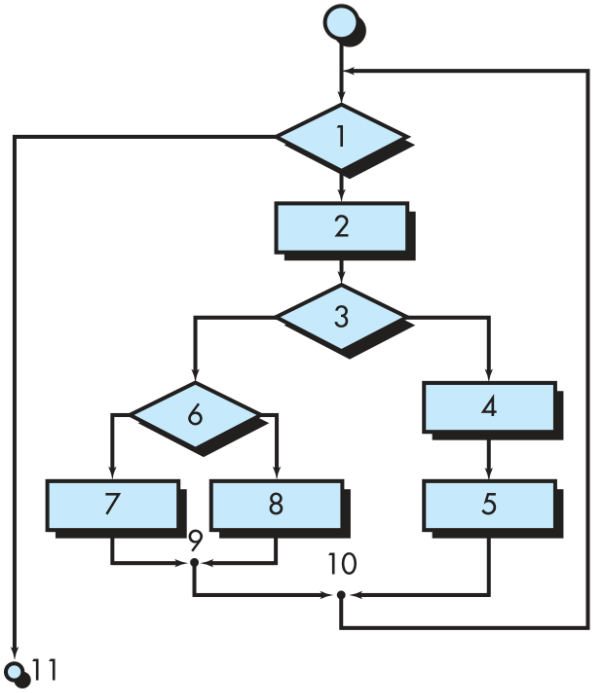


Figura 3-2 a) diagrama de flujo

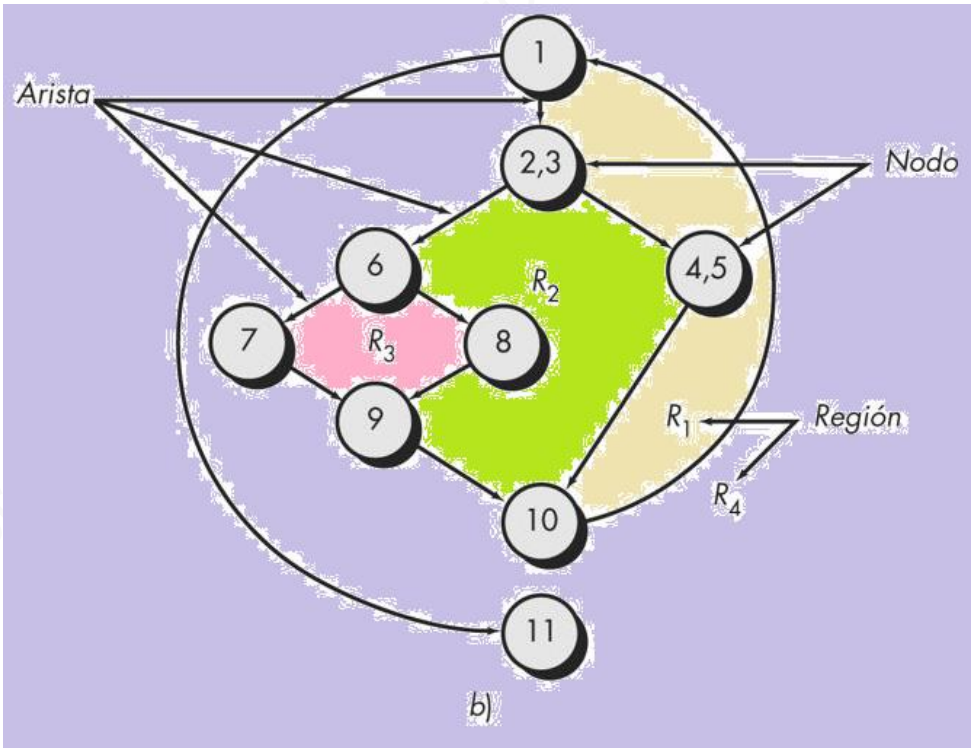


Figura 3-3 b) grafo de flujo

Cuando en un diseño de procedimiento se encuentran condiciones compuestas, la generación de un grafo de flujo se vuelve ligeramente más complicada. Una condición compuesta ocurre cuando una instrucción condicional incluye uno o más **operadores lógicos o booleanos** (OR, AND[4]). En la Figura 3-4, el fragmento en pseudocódigo se traduce en el grafo de flujo mostrado. Observe que se crea un nodo separado para cada una de las condiciones a y b en la instrucción IF a OR b. Cada nodo que contiene una condición se llama **nodo predicado** y se caracteriza por **dos aristas que salen de él**.

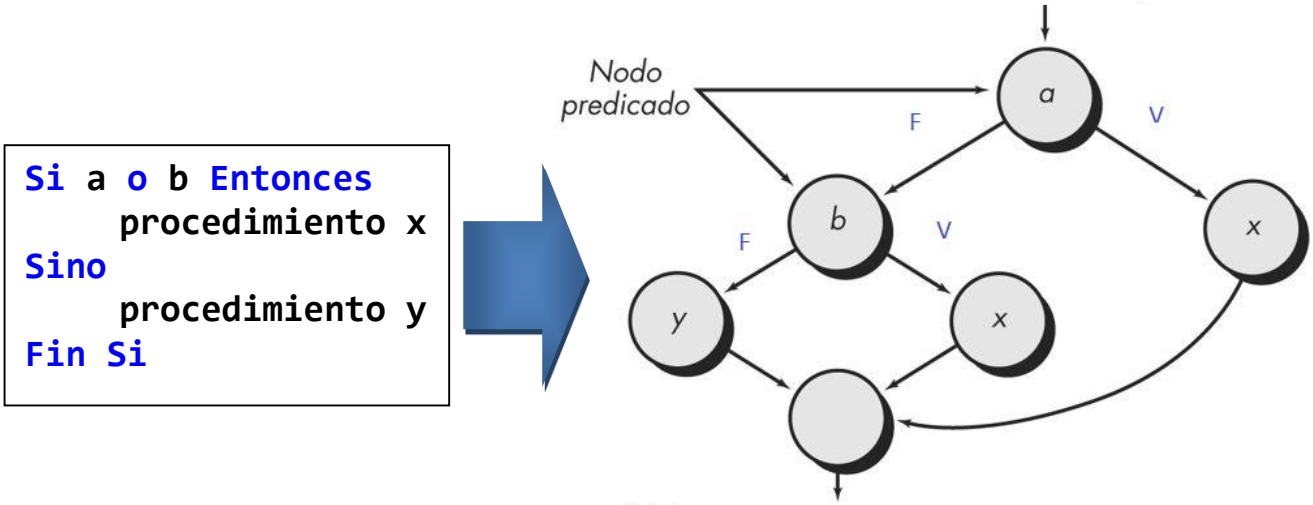


Figura 3-4. Lógica compuesta

3.1.2 Rutas de programa independientes

Una **ruta independiente** es cualquiera que introduce al menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición en el programa. Cuando se define para un grafo de flujo, una ruta independiente debe recorrer al menos una arista que no se haya recorrido antes de definir la ruta. Por ejemplo, un conjunto de rutas independientes para el grafo de flujo que se ilustra en la Figura 3-3b es:

ruta 1: 1-11

ruta 2: 1-2-3-4-5-10-1-11

ruta 3: 1-2-3-6-8-9-10-1-11

[4] En castellano: AND significa Y, OR significa O.

ruta 4: 1-2-3-6-7-9-10-1-11

Observe que cada nueva ruta introduce una nueva arista.

La ruta 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 no se considera como independiente porque simplemente es una combinación de rutas ya especificadas y no recorre ninguna arista nueva.

Las rutas de la 1 a la 4 constituyen un conjunto básico (basis set) para el grafo de flujo de la Figura 3-3. Es decir, si se pueden diseñar pruebas para forzar la ejecución de estas rutas, toda instrucción en el programa tendrá garantizada su ejecución al menos una vez. Debe señalarse que el conjunto básico no es único. De hecho, para un diseño de procedimiento, pueden derivarse conjuntos básicos diferentes.

¿Cómo saber cuántas rutas buscar? El cálculo de la **complejidad ciclomática** proporciona la respuesta. La complejidad ciclomática (cyclomatic complexity) es una métrica o medición de software que proporciona una evaluación cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba de la ruta básica, el valor calculado por la complejidad ciclomática define el número de rutas independientes del conjunto básico de un programa y le brinda una cota (límite) superior para el número de pruebas que debe realizar a fin de asegurar que todas las instrucciones se ejecutaron al menos una vez.

La complejidad ciclomática tiene fundamentos en la teoría de grafos y proporciona una métrica de software extremadamente útil. Se puede calcular de tres formas:

1. El número de regiones del grafo de flujo.
2. E − N + 2
   donde E es el número de aristas del grafo de flujo y N el número de nodos.
3. P + 1
   donde P es el número de nodos predicado contenidos en el grafo de flujo.

En el grafo de flujo de la Figura 3-3b, la complejidad ciclomática puede calcularse usando cada uno de los algoritmos mencionados arriba:

1. El grafo de flujo tiene 4 regiones.
2. 11 aristas − 9 nodos + 2 = 4.
3. 3 nodos predicado + 1 = 4.

Por lo tanto, la complejidad ciclomática del grafo de flujo de la Figura 3-3b es 4.

Más importante, el valor proporciona una cota (límite) superior para el número de rutas independientes que forman el conjunto básico y, en consecuencia, una cota superior sobre el número de casos de prueba que deben diseñarse y ejecutarse para garantizar cobertura de todos las instrucciones del programa o fragmento.

3.1.3 Generación de casos de prueba

El método de prueba de ruta básica puede aplicarse a un diseño de procedimientos (pseudocódigo) o a un código fuente. En esta sección se presenta la prueba de ruta básica como una serie de pasos. El procedimiento **cuenta_multiplos**, que se muestra en pseudocódigo en la Figura 3-5, se usará como ejemplo para ilustrar cada paso del método de diseño de casos de prueba. Observe que, aunque es un algoritmo relativamente simple, contiene condiciones y un ciclo.

El algoritmo **cuenta _multiplos**, que podría ser parte de un software educativo, solicita al usuario el ingreso de números, que por simplicidad suponemos que son siempre enteros y positivos. Una vez ingresado un número, el algoritmo le permite ingresar otro, finalizando el ciclo cuando se ingrese un valor cero. Luego, el algoritmo informa la cantidad números que son múltiplo de 7 o bien de 8, por ejemplo 14 y 24.

```
1  Algoritmo cuenta_multiplos
2      cuenta ← 0
3      Escribir 'Ingresa el primer valor (cero para terminar):'
4  1   Leer x
5  2   Mientras x > 0
6          3 Si x es multiplo de 7 O x es multiplo de 8 Entonces 4
7              cuenta ← cuenta + 1 5
8          FinSi
9      6   Escribir 'Ingresa otro valor (cero para terminar):'
10         Leer x
11     FinMientras
12 7   Escribir 'La cantidad de múltiplos de 7 o de 8 es:', cuenta
13 FinAlgoritmo
```

Figura 3-5. Pseudocódigo con identificación de nodos.

Es posible aplicar los siguientes pasos para generar los casos de prueba:

1. Usando el diseño o el código como base, dibuje el grafo de flujo correspondiente. El grafo de flujo se crea usando los símbolos y reglas de construcción que se presentaron en la sección 3.1.1. Para el pseudocódigo de la Figura 3-5, el grafo de flujo se crea al numerar aquellas instrucciones que se asociarán a los nodos correspondientes del grafo. En la Figura 3-6 se muestra el grafo correspondiente.
2. Determine la complejidad ciclomática del grafo de flujo resultante. La complejidad ciclomática V(G) se determina al aplicar los algoritmos descriptos en la sección 3.1.2. Debe observarse que V(G) puede determinarse sin desarrollar un grafo de flujo, al contar todas las instrucciones condicionales en el pseudocódigo (para el algoritmo **cuenta_multiplos** son tres) y sumar 1. Para la Figura 3-6:

   V(G) = 4 regiones (cuadrados verdes)

   V(G) = 9 aristas − 7 nodos + 2 = 4

   V(G) = 3 nodos predicado + 1 = 4

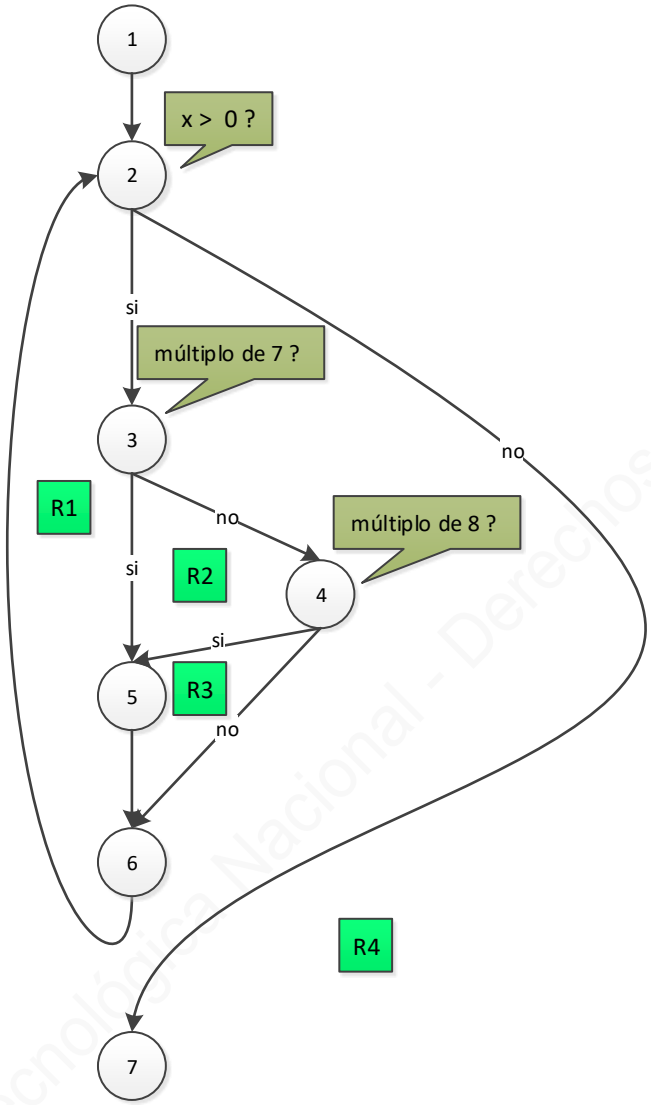


Figura 3-6. Grafo de flujo para el algoritmo Suma_y_maximo

3. Determine un conjunto básico de rutas independientes. El valor de V(G) proporciona la cota superior sobre el número de rutas independientes a través de la estructura de control del programa. Se esperaría especificar 4 rutas. Sin embargo, en este caso sólo es posible encontrar 3 rutas independientes. Esto se debe a que el algoritmo contiene un ciclo, lo que convierte en obligatoria a una de las aristas, la arista de salida del ciclo (que va del nodo 2 al nodo 7). Las 3 rutas independientes son:

   ruta 1: 1-2-3-5-6-2-7

   ruta 2: 1-2-3-4-5-6-2-7

ruta 3: 1-2-3-4-6-2-7

Vale la pena identificar nodos predicado como un auxiliar para la derivación de los casos de prueba. En este caso, los nodos 2, 3 y 4 son nodos predicado porque tienen cada uno 2 aristas salientes.

4. Prepare casos de prueba que fuercen la ejecución de cada ruta en el conjunto básico. Los datos deben elegirse de modo que las condiciones en los nodos predicado encaminen el flujo de ejecución por cada ruta. Cada caso de prueba se ejecuta y se compara con los resultados esperados. Una vez completados todos los casos de prueba, el examinador (tester) puede estar seguro de que todas las instrucciones del programa se ejecutaron al menos una vez.

   Por ejemplo, en este caso los 3 casos de prueba quedan:

Caso Nº	Ruta	Datos de entrada	Resultado esperado
1	1-2-3-5-6-2-7	14 y luego 0	1
2	1-2-3-4-5-6-2-7	16 y luego 0	1
3	1-2-3-4-6-2-7	9 y luego 0	0

Por último, los grafos de flujo se pueden representar como matrices, lo que permitiría el procesamiento automatizado. Aquellos lectores interesados pueden recurrir a [Bei90] para obtener una descripción de esta representación.

3.2 Prueba de la estructura de control

La técnica de prueba de ruta básica descripta en la sección 3.1 es una de varias técnicas para probar la estructura de control. Aunque la prueba de ruta básica es simple y enormemente efectiva, no es suficiente en sí misma. En esta sección se estudian un par de variantes de la prueba de la estructura de control (control structure testing). Esta prueba más amplia cubre y mejora la calidad de la prueba de caja blanca.

3.2.1 Prueba de condición múltiple

La **prueba de condición múltiple** (multiple condition coverage) es un método de diseño de casos de prueba que revisa las condiciones lógicas contenidas en un programa o

módulo. Una condición simple es una variable booleana o una expresión relacional, posiblemente precedida de un operador NOT[5]. Una expresión relacional toma la forma

E1 <operador relacional> E2

donde E1 y E2 son expresiones aritméticas y <operador relacional> es uno de los siguientes: <, ≤, =, ≠, > o ≥. Una condición compuesta se integra con dos o más condiciones simples, operadores booleanos y paréntesis. Se supone que los operadores booleanos permitidos en una condición compuesta incluyen OR, AND y NOT.

Si una condición es incorrecta, entonces al menos un componente de la condición es incorrecto. Por lo tanto, los tipos de errores en una condición incluyen errores de operador booleano (operadores booleanos incorrectos/faltantes/sobrantes), de variable booleana, de paréntesis booleanos, de operador relacional y de expresión aritmética. El método de prueba de condición múltiple se enfoca en la prueba de cada condición del programa para asegurar que no contiene errores.

Por ejemplo, sea el siguiente fragmento de pseudocódigo:

```
1  Escribir 'La temperatura es: ', temperatura
2  Escribir 'La fecha es: ', fecha
3  Si temperatura > 30 Y fecha == '01/01' Entonces
4      Escribir 'Feliz año nuevo!!'
5  FinSi
```

El objetivo del programa es informar la fecha y la temperatura, y adicionalmente mostrar un saludo en caso de ser primero de enero. Sin embargo, se puede apreciar que, por un error en la lógica del programa, el mensaje “Feliz año nuevo!!” no siempre se muestra para el primero de enero, sino que depende de la temperatura.

En este programa de ejemplo, para tener cobertura completa de las condiciones, es necesario tener los cuatro casos de prueba se muestran en la Tabla 3-1. Estos casos aseguran que cada posibilidad en la instrucción *Si* está cubierta.

[5] NOT significa NO (negación). Por ejemplo, si la afirmación x es verdadera, entonces NOT x es falsa.

temperatura	*fecha*	*Nro de línea ejecutada*	*Saludo esperado*	*Saludo en pantalla*	*Resultado del caso de prueba*
29	02-01	1,2,3,5	No	No	OK
29	01-01	1,2,3,5	Si	No	Fallido
31	02-01	1,2,3,5	No	No	OK
31	01-01	1,2,3,4,5	Si	Si	OK

Tabla 3-1. Casos de prueba para lograr cobertura completa de la condiciones

Si hubiéramos estado interesados solamente en cubrir las rutas básicas, el segundo caso de prueba sería redundante y se podría haber probado el fragmento con sólo tres casos de prueba (queda como ejercicio para el estudiante mostrarlo realizando el grafo de flujo y calculando la complejidad ciclomática).

Esto significa que con los tres casos del método de ruta básica **no se descubre el error en el programa** dado que los tres arrojan el resultado esperado. Sin embargo, el caso adicional que requiere el método de condición múltiple produce una falla y por lo tanto descubre el error.

3.2.2 Prueba de ciclo

Los ciclos o bucles (loops) son la piedra angular de la gran mayoría de los algoritmos implementados en el software. La Figura 3-7 muestra la estructura de un ciclo típico, llamado **while** (mientras). Se puede ver que primero se verifica una condición. En caso de cumplirse, se ejecutan las instrucciones correspondientes, que se volverán a repetir mientras la condición siga siendo verdadera.

Y aun así, con frecuencia se les pone poca atención mientras se realizan las pruebas de software.

La prueba de ciclo (loop testing) [Bei90] es una técnica de prueba de caja blanca que se enfoca exclusivamente en la validez de las construcciones de tipo ciclo.

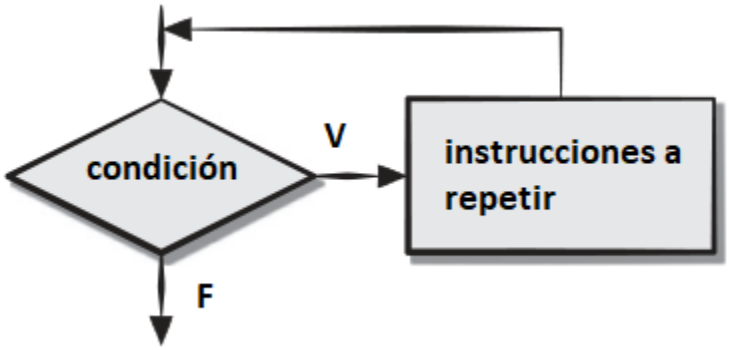


Figura 3-7. Estructura de un ciclo típico

El siguiente conjunto de pruebas puede aplicarse a los ciclos, donde **n** es el máximo número de pasadas permisibles a través del ciclo.

1. Saltar por completo el ciclo.
2. Sólo una pasada a través del ciclo.
3. Dos pasadas a través del ciclo.
4. m pasadas a través del ciclo, donde m < n es el valor típico de pasadas.
5. n − 1, n, n + 1 pasadas a través del ciclo. Por supuesto n + 1 no debe ser posible pero es lo que se debe intentar.

4 Testing especializado

En ocasiones, se requieren lineamientos y enfoques de prueba singulares cuando se consideran entornos, arquitecturas y aplicaciones especializados. Aunque las técnicas de prueba estudiadas anteriormente en esta Unidad y las siguientes, con frecuencia pueden adaptarse a situaciones especializadas, vale la pena considerar individualmente sus necesidades únicas.

4.1 Pruebas de interfaces gráficas de usuario

Las interfaces gráficas de usuario (GUI, por sus siglas en inglés de Graphical User Interface) presentan interesantes desafíos de prueba. Puesto que los componentes reutilizables ahora son parte común en los entornos de desarrollo GUI, la creación de la interfaz para el usuario se ha vuelto menos consumidora de tiempo y más precisa. Pero, al mismo tiempo, la complejidad de las GUI ha crecido, lo que conduce a más dificultad en el diseño y ejecución de los casos de prueba.

Debido a que muchas GUI modernas tienen la misma apariencia y ambiente, puede derivarse una serie de pruebas estándar. Es posible usar **diagramas de estados** para derivar una serie de pruebas que verifiquen los objetos de datos y programa específicos que sean relevantes para la GUI. Esta técnica de prueba es un caso particular de la llamada **prueba basada en modelo (PBM)** y es una técnica de prueba de caja negra que usa la información contenida en el modelo de requerimientos o diseño como la base para la generación de casos de prueba. La técnica de prueba basada en modelo usa diagramas de estado UML[6], un elemento del modelo de comportamiento [Pre19], como la base para el diseño de los casos de prueba.

La PBM ayuda a descubrir errores en el comportamiento del software y, como consecuencia, es extremadamente útil cuando se prueban aplicaciones orientadas a eventos, como es el caso de una GUI.

En la Figura 4-1 se muestra como ejemplo un diagrama de estado simplificado para la GUI de un editor de texto. Los recuadros color crema representan los 3 **estados** posibles de la GUI, en tanto que las aristas (llamadas **transiciones**) son las acciones que realiza el usuario que llevan a la GUI de un estado a otro (o al mismo).

La idea de la técnica de prueba es usar la información del diagrama para generar los casos necesarios para forzar al software a realizar la transición de estado a estado ejercitando todas las posibles transiciones.

Como puede existir un gran número de permutaciones asociadas con las operaciones de la GUI, la prueba de GUI puede abordarse usando herramientas automatizadas como por

[6] El Lenguaje de Modelado Unificado (UML) es un lenguaje gráfico estándar para escribir especificaciones y diseños de software [Boo05]. En [Pre19] se puede encontrar un apéndice con una introducción a UML.

ejemplo Katalon Studio, AutoIt o Selenium, entre otras[7]. Algunas de estas herramientas se tratarán más adelante en el curso.

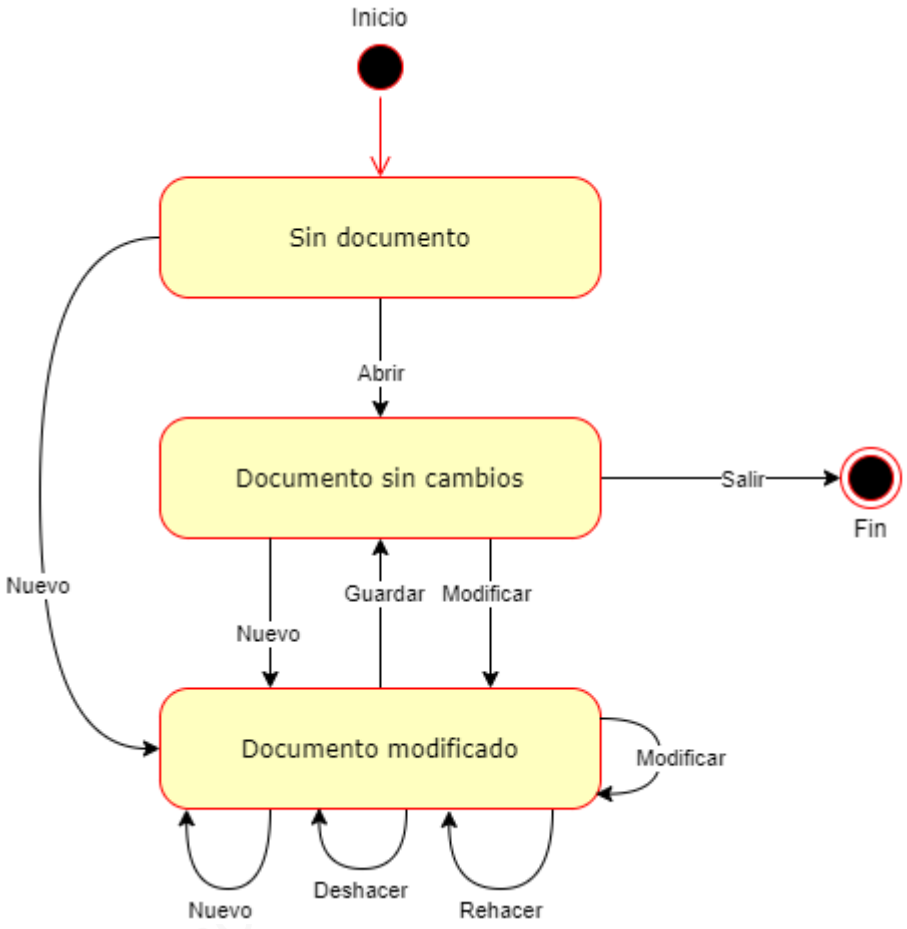


Figura 4-1. Ejemplo de diagrama de estado para la GUI de un editor de texto

[7] Una breve reseña de algunas de estas herramientas puede encontrarse en el sitio *https://www.softwaretestinghelp.com/best-gui-testing-tools/*

4.2 Prueba de arquitecturas cliente-servidor

La naturaleza distribuida de los entornos cliente-servidor, los conflictos de rendimiento asociados con el procesamiento de transacciones, la potencial presencia de algunas plataformas de hardware diferentes, las complejidades de la comunicación en red, la necesidad de atender a múltiples clientes desde una base de datos centralizada (o en algunos casos, distribuida) y los requerimientos de coordinación impuestos al servidor se combinan para realizar las pruebas de las arquitecturas cliente-servidor, y el software que reside dentro de ellas es considerablemente más difícil que las aplicaciones independientes. De hecho, estudios recientes en la industria indican un aumento significativo en el tiempo y costo de las pruebas cuando se desarrollan los entornos cliente-servidor.

En general, la prueba del software cliente-servidor ocurre en tres niveles diferentes: 1) las aplicaciones cliente individuales se prueban en un modo “desconectado”; no se considera la operación del servidor ni la red subyacente. 2) El software cliente y las aplicaciones servidor asociadas se prueban en conjunto, pero las operaciones de red no se revisan de manera explícita. 3) Se prueba la arquitectura cliente-servidor completa, incluidos la operación de red y el rendimiento.

Aunque en cada uno de estos niveles de detalle se realizan muchos tipos de pruebas diferentes, para las aplicaciones cliente-servidor se encuentran comúnmente los siguientes abordajes de prueba:

- Pruebas funcionales de aplicación. La funcionalidad de las aplicaciones cliente se prueba usando los métodos analizados anteriormente en esta Unidad y en las siguientes. En esencia, la aplicación se prueba en forma independiente con la intención de descubrir errores en su operación.
- Pruebas de servidor. Se prueban las funciones de coordinación y gestión de datos del servidor. También se considera el rendimiento del servidor (tiempo de respuesta global y cantidad de datos transmitidos).
- Pruebas de base de datos. Se prueban la precisión y la integridad de los datos almacenados por el servidor. Se examinan las transacciones colocadas por las aplicaciones cliente para asegurar que los datos se almacenen, actualicen y recuperen de manera adecuada. También se prueba la forma de archivar.

- Pruebas de transacción. Se crea una serie de pruebas para garantizar que cada clase de transacciones se procese de acuerdo con los requerimientos. Las pruebas se enfocan en comprobar que el procesamiento sea correcto y también en los conflictos de rendimiento (por ejemplo, tiempos de procesamiento de transacción y volumen de transacción).
- Pruebas de comunicación de red. Estas pruebas verifican que la comunicación entre los nodos de la red ocurre de manera correcta y que el mensaje que pasa, las transacciones y el tráfico de red relacionado ocurren sin errores. Como parte de estas pruebas, también pueden realizarse pruebas de seguridad de red.

Para lograr estos abordajes de prueba, Musa [Mus93] recomienda el desarrollo de perfiles operativos derivados de escenarios de uso cliente-servidor. Un perfil operativo indica cómo interactúan con el sistema cliente-servidor diferentes tipos de usuarios. Es decir, los perfiles proporcionan un “patrón de uso” que puede aplicarse cuando las pruebas se diseñan y ejecutan. Por ejemplo, para un tipo particular de usuario, ¿qué porcentaje de transacciones serán consultas?, ¿cuántas serán actualizaciones?, ¿cuántos serán pedidos?

Para desarrollar el perfil operativo, es necesario derivar un conjunto de escenarios que sean similares a los casos de uso. Cada escenario aborda quién, dónde, qué y por qué. Es decir: quién es el usuario, dónde (en la arquitectura cliente-servidor física) ocurre la interacción del sistema, cuál es la transacción y por qué ocurre. Los escenarios pueden derivarse usando técnicas de respuesta a requerimientos o a través de análisis menos formales con los usuarios finales. Sin embargo, el resultado debe ser el mismo. Cada escenario debe proporcionar un indicio de las funciones del sistema que se requerirán para atender a un usuario particular, el orden en el que se requieren dichas funciones, la temporización y la respuesta que se espera, así como la frecuencia con la que se usa cada función. Luego, estos datos se combinan (para todos los usuarios) a fin de crear el perfil operativo. En general, el esfuerzo de prueba y el número de casos de prueba por ejecutar se asignan a cada escenario de uso con base en la frecuencia de uso y en lo crítico de las funciones realizadas.

4.3 Prueba de documentación y ayuda

El término prueba de software invoca imágenes de gran número de casos de prueba preparados para revisar los programas y los datos que manipulan. Es importante notar que las pruebas también deben extenderse al tercer elemento de la configuración del software: la documentación.

Los errores en la documentación pueden ser tan devastadores para la aceptación del programa como los errores en los datos o en el código fuente. Nada es más frustrante que seguir con exactitud una guía de usuario o un centro de ayuda en línea y obtener resultados o comportamientos que no coinciden con los predichos por la documentación. Por esta razón, las pruebas de documentación deben ser parte significativa de todo plan de prueba de software.

La prueba de documentación puede abordarse en dos fases. La primera, la revisión técnica [Mye12], examina el documento en su claridad editorial. La segunda, prueba en vivo, usa la documentación en conjunto con el programa real.

Sorprendentemente, una prueba en vivo para la documentación puede abordarse usando técnicas que son análogas a muchos de los métodos de prueba de caja negra estudiados anteriormente. La prueba basada en gráfico [Pre19] puede usarse para describir el uso del programa; la partición de equivalencia y el análisis del valor límite pueden usarse para definir varias clases de entrada e interacciones asociadas. La PBM puede usarse para garantizar que el comportamiento documentado y el comportamiento real coinciden. Entonces, el uso del programa puede rastrearse a través de la documentación.

Las siguientes preguntas deben responderse durante las pruebas de documentación y/o ayuda:

- ¿La documentación describe con precisión cómo lograr cada modo de uso?
- ¿La descripción de cada secuencia de interacción es precisa?
- ¿Los ejemplos son precisos?
- ¿La terminología, descripciones de menú y respuestas del sistema son consistentes con el programa real?
- ¿Es relativamente fácil localizar guías dentro de la documentación?
- ¿La solución de problemas puede lograrse con facilidad usando la documentación?

- ¿La tabla de contenido y el índice del documento son consistentes, precisos y completos?
- ¿El diseño del documento (plantilla, fuentes, sangrías, gráficos) contribuye a comprender y asimilar rápidamente la información?
- ¿Todos los mensajes de error del software que se muestran al usuario se describen con más detalle en el documento? ¿Las acciones a tomar como consecuencia de un mensaje de error se delinean con claridad?
- Si se usan enlaces de hipertexto, ¿son precisos y completos?
- Si se usa hipertexto, ¿el diseño de navegación es apropiado para la información requerida?

La única forma viable para responder estas preguntas es hacer que un tercero independiente (por ejemplo, usuarios seleccionados) pruebe la documentación en el contexto del uso del programa. Todas las discrepancias se anotan y las áreas de ambigüedad o debilidad en el documento se marcan para su potencial reescritura.

4.4 Prueba para sistemas de tiempo real

La naturaleza asíncrona, dependiente del tiempo de muchas aplicaciones de tiempo real, agrega un nuevo y potencialmente difícil elemento a la mezcla de pruebas: el tiempo. El diseñador de casos de prueba no sólo debe considerar los casos de prueba convencionales, sino también la manipulación de eventos (es decir, el procesamiento de interrupciones), la temporización de los datos y el paralelismo de las tareas (procesos) que manejan los datos. En muchas situaciones, probar los datos proporcionados cuando un sistema de tiempo real está en un estado dará como resultado un procesamiento adecuado, mientras que los mismos datos proporcionados cuando el sistema está en un estado diferente pueden conducir a error.

Por ejemplo, el software de tiempo real que controla una nueva fotocopiadora acepta interrupciones del operador (es decir, el operador de la máquina presiona teclas de control como RESET ú OSCURECER) sin error cuando la máquina saca copias (en el estado “copiando”). Estas mismas interrupciones del operador, si se ingresan cuando la máquina está en estado “atascado”, hacen que se borre la visualización del código de diagnóstico que indica la ubicación del atasco (esto es un error).

Además, la íntima relación que existe entre el software de tiempo real y su entorno de hardware también puede causar problemas en las pruebas. Las pruebas del software deben considerar el impacto de los fallos de hardware en el procesamiento del software. Tales fallos pueden ser extremadamente difíciles de simular de manera realista.

Los métodos exhaustivos de diseño de casos de prueba para sistemas de tiempo real continúan evolucionando. Sin embargo, puede proponerse una estrategia global de cuatro pasos:

- **Prueba de tareas**. El primer paso en la prueba del software en tiempo real es probar cada tarea de manera independiente. Es decir, las pruebas convencionales se diseñan para cada tarea y se ejecutan independientemente durante dichas pruebas. La prueba de tareas descubre errores en lógica y funcionalidad, mas no en temporización ni comportamiento.
- **Prueba de comportamiento**. Con modelos de sistema creados con herramientas automatizadas, es posible simular el comportamiento de un sistema de tiempo real y examinar su comportamiento como consecuencia de eventos externos. Estas actividades de análisis pueden servir de base para el diseño de los casos de prueba que se realizan cuando se construye el software. Al usar una técnica similar a la partición de equivalencia (sección 2.1.1), los eventos (por ejemplo, interrupciones, señales de control) se categorizan para las pruebas. Por ejemplo, los eventos para la fotocopiadora pueden ser interrupciones del usuario (ej. resetear contador), interrupciones mecánicas (ej. atasco de papel), interrupciones del sistema (ej. tóner bajo) y modos de fallo (ej. sobrecalentamiento del rodillo). Cada uno se prueba de manera individual y el comportamiento del sistema ejecutable se examina para detectar los errores que ocurren como consecuencia del procesamiento asociado con dichos eventos. El comportamiento del modelo del sistema (desarrollado durante la actividad de análisis) y el software ejecutable pueden compararse para asegurar que actúan correctamente. Una vez que se prueba cada clase de eventos, éstos se presentan al sistema en orden aleatorio y con frecuencia aleatoria. El comportamiento del software se examina para detectar errores de comportamiento.
- **Prueba intertarea**. Una vez aislados los errores en las tareas individuales y en el comportamiento del sistema, las pruebas se orientan a los errores relacionados con el tiempo. Las tareas asíncronas que se sabe que se comunican mutuamente se prueban con diferentes tasas de datos y carga de procesamiento para determinar si ocurrirán errores de sincronización entre tareas. Además, las tareas que se comunican vía cola de mensaje o almacenamiento de datos se prueban para descubrir errores en el tamaño de estas áreas de almacenamiento.
- **Prueba de sistema**. Al integrar software y hardware, se lleva a cabo un amplio rango de pruebas del sistema con la intención de descubrir errores en la interfaz software-hardware. La mayoría de los sistemas en tiempo real procesan interrupciones. Por

tanto, probar la manipulación de estos booleanos es esencial. Al usar el diagrama de estado, el tester desarrolla una lista de las posibles interrupciones y del procesamiento que ocurre como consecuencia de ellas. Entonces se diseñan pruebas para evaluar las siguientes características del sistema:

- ¿Las prioridades de interrupción se asignan y manejan de manera adecuada?
- ¿El procesamiento para cada interrupción se maneja de manera correcta?
- ¿El rendimiento o performance (por ejemplo, tiempo de procesamiento) de cada procedimiento de manejo de interrupción se apega a los requerimientos?
- ¿Un alto volumen de interrupciones que llegan en momentos críticos crea problemas en el funcionamiento o en el rendimiento?

Además, las áreas de datos globales (memoria compartida, etc.) que se usan para transferir información como parte del procesamiento de interrupciones deben probarse a fin de evaluar el potencial para la generación de efectos colaterales.

Bibliografía utilizada y sugerida

Libros y otros manuscritos

- [Bei90] Beizer, Boris. Software Testing Techniques. Second Edition. 1990.
- [Bei95] Beizer, Boris. Black-Box Testing, Wiley, 1995.
- [Boe81] Boehm, B. Software Engineering Economics. 1981.
- [Boo05] Booch, G., Rumbaugh, J. & Jacobsen, I. The Unified Modeling Language User Guide. Second Edition. 2005.
- [Cop04] Copeland, Lee. A Practitioner's Guide to Software Test Design. 2004.
- [Eve07] Everett, Gerald & McLeod, Raymond. Software Testing - Testing Across The Entire Software Development LifeCycle. 2007.
- [Far08] Farrell-Vinay, Peter. Manage Software Testing. 2008
- [Mus93] Musa, J. Operational Profiles in Software Reliability Engineering. 1993
- [Mye12] Myers, Glenford. The Art of Software Testing. Third Edition. 2012.
- [Pat05] Patton, Ron. Software Testing. Second Edition. 2005.
- [Pre19] Pressman, Roger y Maxim, B. Software Engineering: A Practitioner's Approach. Ninth Edition. 2019.

Lo que vimos:

En esta Unidad nos adentramos en los detalles relacionados con el diseño de casos de prueba, usando tanto técnicas de caja blanca como de caja negra.

Finalizamos la Unidad describiendo las particularidades del testing en entornos especializados, como sistemas cliente-servidor y sistemas de tiempo real; y de aspectos específicos, como las interfaces de usuario y la documentación y ayuda en línea.

Lo que viene:

En las próximas Unidades nos enfocaremos en el testing de tipos particulares de proyectos, y luego completaremos el conocimiento con conceptos, métodos y herramientas complementarias.