Cree un conjunto de casos de prueba para este diagrama. Suponga que todas las pruebas deben comenzar en el estado inicial—mostrado con un círculo con borde doble—y deben terminar cuando las transiciones del sistema vuelvan al estado inicial. Trate de crear el número mínimo de casos de prueba y aún así lograr la cobertura completa de los estados y las transiciones de estados. ¿Terminó? Bien. Debería haber creado dos casos de prueba.

Teniendo en cuenta la regla acerca de los casos de prueba que tienen que empezar y terminar en el estado inicial, ¿Cuál es el número máximo de casos de prueba?

Es infinito. Si usted nos muestra cualquier conjunto de pruebas que piensa que es exhaustivo, podemos añadir otro ciclo del trabajo desde el trabajo de impresión a la espera de los usuarios y de vuelta al trabajo de impresión que es una nueva prueba. Esta propiedad, de siempre ser capaz de añadir una única variante más a una lista, es la definición misma del infinito contable.

Para cualquier sistema que puede describirse mediante un diagrama de transición de estados con esos bucles en el diagrama, puede crear un número infinito de pruebas.

Ahora cree una tabla de transición de estados.

¿Terminó? Bien. Su tabla de transición de estados debería tener 35 filas en la tabla. ¿Tiene 40? Si es así, probablemente contó dos veces el evento “error recuperable reportado”.

4.4 Técnicas Basadas en la Estructura

Objetivos del Aprendizaje

LO-4.4.1 Describir el concepto y el valor de cobertura de código. (K2)

LO-4.4.2 Explicar los conceptos de cobertura de sentencia y decisión, y dar las razones por qué estos conceptos pueden también ser utilizados en otros niveles de prueba distintos a las pruebas de componente (p.ej. en procedimientos comerciales en el nivel de sistema). (K2)

LO-4.4.3 Escribir casos de prueba de los flujos de control dados utilizando las técnicas de diseño de pruebas de sentencia y decisión. (K3)

LO-4.4.4 Evaluar la completitud de la cobertura de sentencia y decisión con respecto a los criterios de salida definidos. (K4)

En esta sección, Técnicas Basadas en la Estructura, cubrirá los siguientes conceptos clave:

- Niveles de cobertura de código.
- Obtención de la cobertura de sentencia y decisión con las pruebas.
- Utilización del flujo de control del programa para diseñar las pruebas.

Empecemos con los conceptos básicos de las pruebas basadas en la estructura o de caja blanca.

Recuerde que las pruebas basadas en la especificación, también conocidas como las pruebas de caja negra o como las pruebas de comportamiento, son las que son diseñadas desde la especificación del sistema. En las pruebas de comportamiento, nosotros ignoramos el funcionamiento interno del sistema y nos enfocamos acerca de cómo este se supone que se debe comportar.

Ahora, las pruebas basadas en la estructura, las cuáles son llamadas también pruebas estructurales o de caja blanca, se basan en la estructura interna del sistema o en un componente del sistema; p.ej., nosotros examinamos cómo el sistema funciona y qué hace. Específicamente, seleccionamos las entradas, las precondiciones, las condiciones, los eventos u otros estímulos basados en una parte de la estructura del sistema que estos ejercerán.

Por supuesto, cuando realicemos las pruebas estructurales—porque éstas son pruebas dinámicas—el sistema o componente presentará algún comportamiento. Tenemos que comparar ese comportamiento con algún resultado esperado, de lo contrario no estamos probando. No podemos derivar los resultados esperados para las pruebas estructurales por completo de la estructura, porque de lo contrario terminamos probando el compilador, el sistema operativo y otros elementos del entorno, más que el sistema propio. De este modo nosotros derivamos típicamente los resultados esperados para las pruebas estructurales en parte de la estructura, pero también de las especificaciones, las expectativas razonables, y así sucesivamente.

Como con las pruebas de comportamiento, cuando nosotros estamos observando los comportamientos mostrados por una prueba de estructura, nosotros podemos estar observando comportamientos funcionales— ¿Qué hace el sistema o componente? —O comportamientos no funcionales— ¿Cómo lo hace? —basado en la clasificación del ISO 9126.

Hay tres maneras para diseñar las pruebas estructurales.

La más simple es analizar los flujos de control en el código y utilizar ese análisis para medir la cobertura de las pruebas existentes y para adicionar las pruebas adicionales para alcanzar un nivel deseado de cobertura. (Hablaremos más acerca de la cobertura de código más adelante). Estos tipos de pruebas estructurales son cubiertos en el programa de estudios básico.

La siguiente, de alguna manera más complicada de diseñar las pruebas estructurales es la de analizar los flujos de datos, utilizando el código y las estructuras de datos. Usted luego

utiliza ese análisis para medir la cobertura de las pruebas existentes y para adicionar pruebas adicionales para alcanzar un nivel deseado de cobertura del flujo de datos. Estos tipos de pruebas estructurales **no** son cubiertas en el programa de estudios nivel básico.

La final, la manera más complicada de diseñar las pruebas estructurales es el de analizar las interfaces, las clases, los flujos de llamada y otros por el estilo observando las interfaces de programación de la aplicaciones (APIs); el hardware, el software y los documentos de diseño y los diagramas de un sistema de redes; las tablas de base de datos, las restricciones de integridad y los procedimientos memorizados ("stored procedures"). Como antes, usted puede analizar ese análisis para medir la cobertura de las pruebas existentes y para adicionar pruebas adicionales para alcanzar un nivel deseado de cobertura del diseño. Este tipo de diseño de pruebas estructurales es comúnmente utilizado durante las pruebas de integración y de sistema. Sin embargo, estos tipos de pruebas estructurales **no** son cubiertos en el programa de estudios básico.

Permítanos resaltar de nuevo un uso importante del diseño de pruebas estructural: Es una mejor práctica para medir el grado de cobertura estructural de las pruebas basadas en la especificación y la experiencia, de tal manera que usted pueda buscar partes importantes de la estructura del sistema, las cuales no están siendo probadas.

Si no tiene una base en programación, podría considerar esta sección un poco difícil. Porque comprendiendo cómo los sistemas tienden a fallar y por qué, es una habilidad clave cuando se está haciendo el análisis de los riesgos de calidad, nosotros lo animaríamos a que se auto enseñe un lenguaje de programación si intenta que su profesión sea la de pruebas de software.

Entonces, ¿Qué significa la cobertura de flujos de control en el código-más frecuentemente llamada cobertura de código-? ¿Qué niveles de cobertura de código existen?

Hay un número de variantes aquí, pero las más comunes son las siguientes:

La cobertura de sentencia, la cuál es el porcentaje de las sentencias ejecutadas mínimamente una vez por las pruebas.

La cobertura de rama o de decisión, la cuál es el porcentaje de ramas o decisiones ejecutadas mínimamente una vez por las pruebas. De nuevo, si usted dice que ha alcanzado "cobertura de rama" eso significa usualmente de que usted ha probado el 100% de las ramas. La cobertura de rama y decisión significan la misma cosa. La cobertura de rama se refiere a una manera gráfica de visualizar el flujo de control, mientras que la cobertura de decisión se enfoca en la condición, si simple o compuesta, que determina cuál rama será tomada.

Glosario del ISTQB

Cobertura de decisión: El porcentaje de resultados de decisión que hayan sido ejercidos

por un juego de pruebas. 100% de cobertura de decisión implica ambos 100% de cobertura de rama y 100% cobertura de sentencia.

La ramificación ocurre cuando el programa hace una decisión acerca de que si una situación particular de dos o más situaciones posibles existen, y luego procede a fluir el control del programa en una manera apropiada para manejar la situación. Ya que una manera de manejar la situación es a través de la secuencia de sentencias, el 100% de cobertura de rama significa que alcanzaremos el 100% de cobertura de sentencia.

La cobertura de condición, la cuál es el porcentaje de condiciones simples, que han sido evaluadas por una de las pruebas mínimamente. Una condición simple es algo como "edad >= 18" o "nombre==”Robert””. El 100% de la cobertura de condición requiere que cada condición única en cada sentencia de decisión sea probada como verdadera y como falsa.

Ahora, cuando no sólo contamos con condiciones simples en un programa, pero con condiciones compuestas como "(edad >= 0) && (edad 18)", entonces podemos hablar acerca de la cobertura de multicondición. La cobertura de multicondición (más conocida como la cobertura completa de condición múltiple) es el porcentaje de las condiciones de todos los resultados de cada condición única dentro de una sentencia que ha sido evaluada por mínimamente una de las pruebas.

Porque probando todas las combinaciones de las condiciones, también significa la prueba de todas las condiciones posibles, el 100% de la cobertura de condición múltiple implica el 100% de cobertura de condición.

Ahora, la cobertura de condición y decisión modificada (MC/DC) es una variante leve de la cobertura de multicondición. En la cobertura de condición y decisión modificada nos fijamos en el porcentaje de las combinaciones de las condiciones que pueden influenciar la decisión que ha sido probada. Por ejemplo, considere una condición compuesta como "(edad >= 0) && (edad 18)". Si la primera condición es verdadera y el valor de la edad es menor que cero, entonces la condición compuesta completa será evaluada como falsa. No hay ninguna necesidad de tratar de evaluar la segunda condición, porque esta no influenciará la decisión. Muchos compiladores, incluyendo todos los compiladores de C++, cesarían la evaluación de una condición compuesta tan pronto como la decisión total sea conocida. Porque la prueba de todas las combinaciones de las condiciones es una medida más estricta de la cobertura, que las pruebas de sólo esas combinaciones de condiciones que puedan afectar la decisión, tome en cuenta que el 100% de la cobertura de condición múltiple implica el 100% de la cobertura de condición y decisión modificada.

Finalmente, hay una cobertura de bucle. Ésta no es una métrica formalizada de la cobertura estructural en el ISTQB nivel básico. Sin embargo, informalmente, podemos decir que hemos alcanzado el 100% de la cobertura del bucle cuando tengamos un conjunto de pruebas que obligan al programa a tomar todos los caminos del bucle cero, uno y muchas veces.

Nosotros mencionamos arriba que el 100% de la cobertura de rama significa que hemos alcanzado el 100% de la cobertura de sentencia. ¿Piensa usted de que el 100% de la cobertura de sentencia significa de que alcanzaremos el 100% de cobertura de rama? La respuesta es “no”. Si usted ha escrito programas de computación antes, sabrá de que no cada sentencia if tiene una sentencia correspondiente else y que ningún constructo switch/case tiene un bloque por defecto de sentencias. En tales casos, el else implícito o la acción por defecto es “hacer nada”, la cual representa una decisión tomada. Por su puesto, si haciendo nada es la cosa correcta para hacer, es algo que también tenemos que probar.

¿Cómo se relaciona esto a las técnicas de diseño de pruebas basadas en la especificación que hemos abordado en la sección previa? Básicamente, la cobertura de rama o decisión corresponde a la cobertura de particiones de equivalencia. Las ramas dicen cómo el programa maneja las diferentes situaciones con fragmentos de código diferentes, que son creados y esas situaciones diferentes son las particiones de equivalencia.

Si probamos en los valores límite, probamos la implementación correcta de alguna de las condiciones. Sin embargo, porque el código podría implementar las combinaciones de condiciones, no significa que hemos logrado la cobertura de condición simple o condición múltiple. De este modo, la analogía entre las pruebas basadas en la especificación y la estructura se rompe en este punto.

```
 1 #include <stdio.h>
 2 main()
 3 {
 4   int i, n, f;
 5   printf("n = ");
 6   scanf("%d", &n);
 7   if (n < 0) {
 8     printf("Invalid: %d\n", n);
 9     n = -1;
10   } else {
11     f = 1;
12     for (i = 1; i <= n; i++) {
13       f *= i;
14     }
15     printf("%d! = %d\n", n, f);
16   }
17   return n;
18 }
```

Figura 4.19: Ejemplo acerca de la Cobertura de Código

Veamos un ejemplo. El programa simple en C en esta página calcula un factorial. Un factorial es el producto de un entero dado y todos los enteros más pequeños, los cuales son mayor que cero. En otras palabras, el factorial de 3 es 3*2*1 o 6. El factorial de 0 está definido como uno.

En este programa, la entrada es el número del cual se calcula el factorial. Está guardado en la variable n.

De este modo, para alcanzar la cobertura de sentencia, ¿Qué valores de pruebas

necesitamos para n? Por favor calcúlelos ahora.

¿Listo? Bueno. Usted debería ver que si usted escoge un valor de n menor que 0 y otro valor de n mayor que 0, ejecutará cada sentencia al menos una vez.

Ahora, ¿Alcanzaría eso la cobertura de rama?

No, necesitamos también n==0 para comprobar que el bucle no sea ejecutado.

Si probamos con n menos que 0, n igual a 0 y n mayor que cero, ¿Alcanzaría eso cobertura de condición? Sí, porque no tenemos condiciones compuestas.

Finalmente, ¿Qué hay de la cobertura de bucle? Para la cobertura de bucle del 100%, necesitamos de cubrir n==1 y n==el número máximo de veces en todo el bucle.

De este modo, ¿Cómo podemos utilizar la cobertura de código para diseñar las pruebas? ¿Qué bueno es este material de las pruebas estructurales para nosotros como probadores?

En debates acerca de otras pruebas y con nuestra propia experiencia personal, nos hemos dado cuenta de que, por sí mismas, las técnicas de caja negra pueden dejar de cubrir hasta un 75% o más de las sentencias.

Ya sea si esta gran cantidad de código no probado representa un problema o no, depende de lo que no es cubierto, así como si los programadores probaron o no este código en las pruebas de unidad.

De este modo, como probadores independientes, podemos utilizar las herramientas de cobertura de código para instrumentalizar un programa. Esto nos permite monitorear la cobertura del código durante la ejecución. (Hablaremos más al respecto en el capítulo 5). Una vez que hayamos encontrado los vacíos en nuestra cobertura del código, decidiríamos de adicionar más casos de prueba para alcanzar niveles más altos de cobertura.

Al principio de este libro hablamos del análisis estático del código. Mencionamos que algunas herramientas de análisis estático pueden localizar segmentos del código altamente complejos. Veamos esa medida de complejidad, la cual tiene algunas implicaciones interesantes del diseño de pruebas.

La complejidad ciclomática de McCabe mide la complejidad del flujo de control. Específicamente, la métrica trabaja como sigue. Cada función en un programa, incluyendo la función principal, comienza con un contador de complejidad de uno. Contamos la complejidad función por función. Cada vez que hay una rama o bucle en una función, el contador de complejidad aumenta en uno. En otras palabras, cuando Tom McCabe diseñó esta métrica, la diseñó para sostener su teoría de la complejidad de la programación, la cual dice que la programación es una tarea compleja a causa de las decisiones que deben ser creadas en el código para dirigir los flujos de control.

Por supuesto, la mejor manera de medir la complejidad ciclomática es utilizando una herramienta. La herramienta creará un grafo dirigido o diagrama de flujo desde el código. Los nodos (o las burbujas) representan los puntos de entrada, los puntos de salida y las decisiones. Las aristas (o las flechas) representan secuencias de cero o más sentencias sin ramas. Por supuesto, si tienen que crear el grafo y calcular la complejidad manualmente, usted puede, pero no es muy divertido para los programas del mundo real.

Porque no es muy divertido y las herramientas pueden ser costosas (sin embargo algunas son libres), ¿Por qué ocuparse con este concepto de la complejidad ciclomática? Bien, éste tiene algunas implicaciones útiles de las pruebas.

Por un lado, si McCabe tenía razón de que la complejidad de la tarea de programación se incrementa con el número de decisiones, los módulos que marcan alto en la complejidad ciclomática de McCabe, serán inherentemente defectuosos y propensos a la regresión. Hay diferentes opiniones acerca de si McCabe **tenía** razón.

Hemos leído un estudio de algunas personas en IBM, que examinaron una de sus aplicaciones. Para esta aplicación, ellos no encontraron correlación entre las varias métricas de complejidad para los varios módulos y las densidades de defectos para esos mismos módulos. Ahora, eso puede significar que la complejidad no influencia la defectuosidad. Sin embargo, eso podría también significar que las métricas de complejidad que ellos examinaron no se adecuan a la captura de todos los elementos de complejidad. ¿Tal vez lo que hace a un programa verdaderamente complejo—y así propenso a los defectos—es algo que la mayoría de las métricas de complejidad no miden?

Personalmente, tenemos alguna experiencia trabajando con código altamente complejo—acerca lo cual queremos decir las dos cosas, intuitivamente complejo así como también complejo en cuanto a la métrica ciclomática de McCabe. Hemos trabajado con tal código en ambos tipos de aplicaciones tanto comerciales como científicas. Hemos encontrado ciertamente situaciones, donde la complejidad del código no condujo a más defectos. De ese modo, diríamos que aunque no pudimos generar ningunas conclusiones finales acerca de que si McCabe tenía razón basados en la evidencia empírica hasta ahora, es todavía una regla de dedo útil para el análisis de los riesgos de calidad para afirmar que esperamos una alta probabilidad de defectos en esas partes del sistema las cuales son altamente complejas.

La otra implicación útil de esta métrica para las pruebas es cuando se evalúa un conjunto de pruebas de unidad para una función. Usted puede extender la técnica de las gráficas del flujo de control de McCabe para generar lo que él llama los caminos a través del grafo. El número de caminos básicos es igual al número de pruebas básicas requeridas para cubrir el grafo. Retornaremos a este concepto en breve, porque éste le ayudará a visualizar los caminos básicos y las pruebas básicas como lo hablamos.

Entonces examinemos más de cerca la complejidad ciclomática de McCabe…

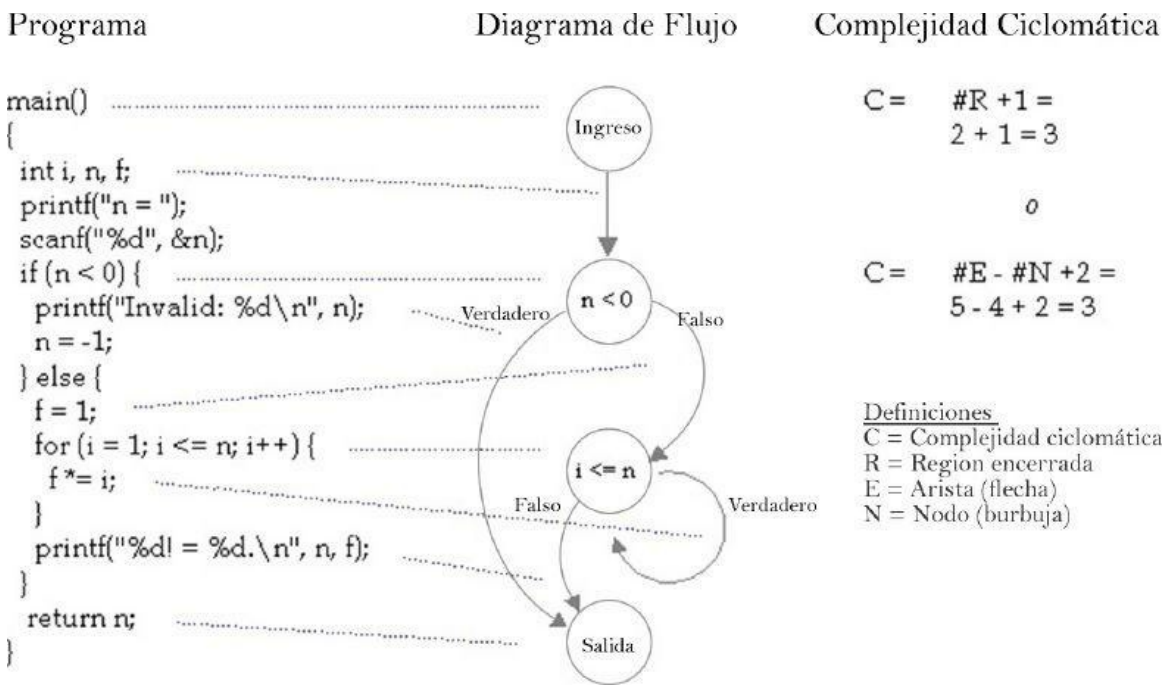

Figura 4.20: La Complejidad Ciclomática para el Factorial

Esta figura muestra el programa que calcula el factorial en el lado izquierdo. Las líneas punteadas del código al diagrama del flujo de McCabe en el centro de esta figura muestra cómo las secuencias sin ramas de cero o más sentencias llegan a ser las aristas (o las flechas), y cómo los constructos con las ramas y los bucles llegan a ser los nodos (o las burbujas).

En la parte derecha de la figura, usted puede observar dos métodos que calculan la métrica de la complejidad ciclomática de McCabe. El más simple es tal vez el cálculo de la "región cerrada". Las dos regiones cerradas, representadas por R en la ecuación superior, se encuentran en el diagrama de abajo y mayormente en la izquierda del nodo "n0" y abajo en la derecha del nodo "l=n".

El otro método de cálculo involucra el conteo de las aristas (o flechas) y los nodos (o burbujas).

Ahora, esto es simplemente suficiente para un programa pequeño y simple como éste. Para funciones más grandes, dibujando el gráfico y haciendo el cálculo de éste es un verdadero lío. De este modo, una simple regla de dedo es: Cuente los constructos de rama y bucle y adicione 1. Las sentencias "if" y los constructos "for", "while" y "do/while cuentan como uno. Para los constructos switch/case, cada bloque case cuenta como uno. En los constructos "if" y "ladder if", el "else" no se cuenta. Para los constructos switch/case, el bloque "por defecto" no se cuenta. Ésta es una regla heurística, pero parece que siempre funciona.

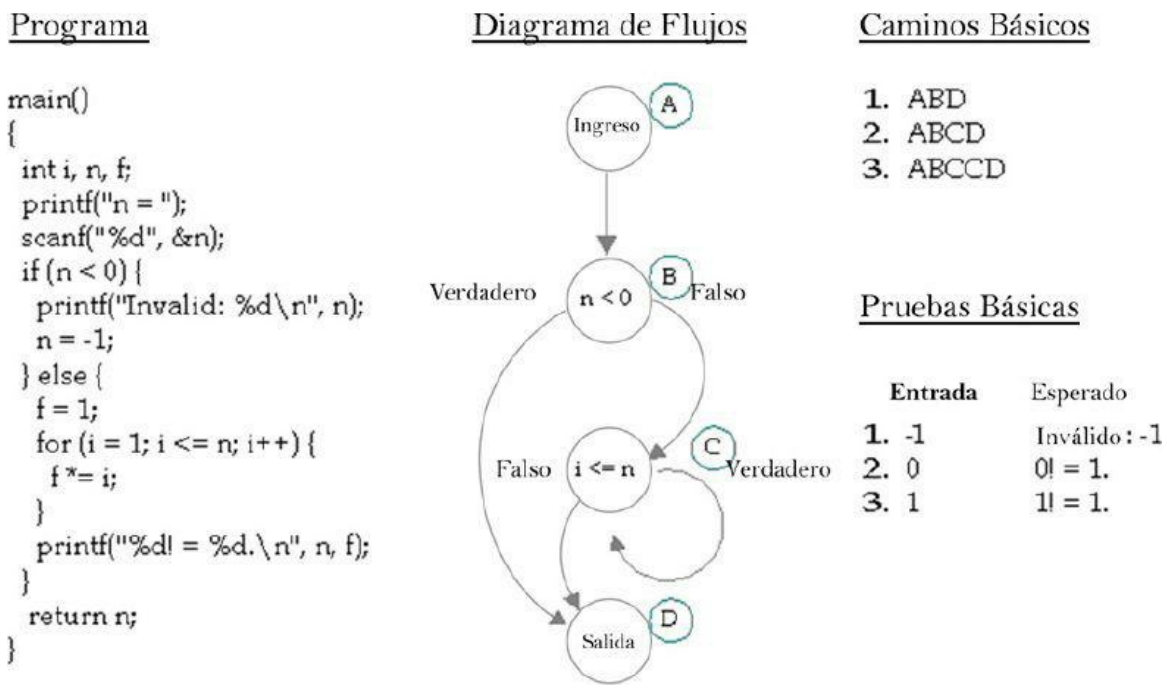

Figura 4.21: Caminos y Pruebas Básicas

Cuando introdujimos la complejidad de McCabe en páginas anteriores, mencionamos los caminos básicos y las pruebas básicas. Esta figura muestra los caminos básicos y las pruebas básicas en el lado derecho.

El número de caminos básicos es igual a la complejidad ciclomática. usted construye los caminos básicos comenzando con un camino cualquiera a través del diagrama, luego adicionando otro camino que cubre el mínimo número de aristas no cubiertas previamente, repitiendo este proceso hasta que todas las aristas hayan sido cubiertas por lo menos una vez.

Las pruebas básicas son las entradas y los resultados esperados asociados con cada camino básico. Usualmente, las pruebas básicas coincidirán con las pruebas necesarias para alcanzar la cobertura de rama. Esto tiene sentido, porque la complejidad se incrementa en cualquier momento en que más de una arista sale de un nodo en un diagrama de flujo de McCabe. En un diagrama de flujo de McCabe, una situación donde "más de una arista de un nodo" representa un constructo de ramas o de bucles.

¿Para qué sirve esta exquisita información? Bien, suponga que estuviese hablando con un programador acerca de sus pruebas de unidad. Usted pregunta cuántas entradas utilizaron ellos. Si ellos le dicen un número menor que la métrica de complejidad ciclomática de McCabe, para el código que ellos están probando, es una apuesta segura de que ellos no alcanzaron la cobertura de rama. Eso implica, como fue mencionado más antes, de que ellos no cubren las particiones de equivalencia.

4.4.1 Ejercicios

Ejercicio 1

Abajo encontrará un programa simple escrito en C que acepta una cadena con caracteres hexadecimales (entre otros caracteres no deseados). Éste ignora los otros caracteres y convierte los caracteres hexadecimales en una representación numérica.[28]

Si prueba con las cadenas de entrada: “059”, “ace” y “ACD” ¿Qué niveles de cobertura usted alcanzaría?

¿Qué cadenas de entrada podría usted añadir para alcanzar las coberturas de sentencia y rama?

¿Serían aquellas pruebas suficientes para probar este programa?

Argumente.

```
main()
/* Convertir los digitos hexadecimales en un número */
{
        int c;
        unsigned long int hexnum, nhex;

        hexnum = nhex = 0;

        while ((c = getchar()) != EOF) {
                switch (c) {
                        case '0': case '1': case '2': case '3': case '4':
                        case '5': case '6': case '7': case '8': case '9':
                                /* Convertir un digito decimal */
                                nhex++;
                                hexnum *= 0x10;
                                hexnum += (c - '0');
                                break;
                                case 'a': case 'b': case 'c':
                                case 'd': case 'e': case 'f':
                        /* Convertir un digito decimal de letra minúscula */
                                nhex++;
                                hexnum *= 0x10;
                                hexnum += (c - 'a' + 0xa);
                                break;
                        case 'A': case 'B': case 'C':
                        case 'D': case 'E': case 'F':
                        /* Convertir un digito decimal de letra mayúscula */
                                nhex++;
                                hexnum *= 0x10;
                                hexnum += (c - 'A' + 0xA);
                                break;
                        default:
                                /* Saltar cualquier carácter no hexadecimal */
                                break;
                }
        }
        printf("Tengo %d digitos hexadecimales: %x\n", nhex, hexnum);
        return 0;
}
```

Figura 4.22: Programa del Conversor Hexadecimal

Solución del Ejercicio 1

Las cadenas “059”, “ace” y “ACD” no alcanzan ningún nivel especifico de cobertura del cual hablamos en el libro. Necesitaríamos añadir “xyz” o alguna otra cadena que contenga dígitos no hexadecimales para alcanzar la cobertura de sentencia. Necesitaría probar la cadena nula para obtener la cobertura de rama (en cuanto a no ejecutar el bucle). Para tratar de ejecutar el bucle rigurosamente, también querríamos probar una cadena muy larga.

Mientras que no es necesario alcanzar ninguno de los niveles de cobertura estructural que abordamos, usted podría considerar cadenas cortas mezcladas. Por ejemplo, “6dpF” prueba cada uno de los cuatro bloques “case” en una sola prueba, y comprobar problemas que podrían ocurrir en esas situaciones.

4.5 Técnicas Basadas en la Experiencia

Objetivos del Aprendizaje

LO-4.5.1 Recordar las causas para escribir casos de pruebas basados en la intuición, experiencia y conocimiento acerca de los defectos comunes. (K1)

LO-4.5.2 Comparar las técnicas basadas en la experiencia con las técnicas de pruebas basadas en la experiencia. (K2)

En esta sección, Técnicas Basadas en la Experiencia, cubrirá los siguientes conceptos clave:

- Razones para escribir casos de prueba basados en la intuición, la experiencia y el conocimiento.
- Comparación de las técnicas basadas en la experiencia con las técnicas basadas en la especificación.

Comencemos con los conceptos básicos de las pruebas basadas en la experiencia.

En este punto, entenderá el diseño de pruebas de comportamiento o el diseño de pruebas basadas en la especificación. En las pruebas de comportamiento, ignoramos el funcionamiento interno del sistema y nos enfocamos en cómo se supone que éste se comporte. También entenderá el diseño de pruebas estructurales, donde se observa el funcionamiento interno del sistema y como el sistema hace lo que hace.

Cuando experimentamos las pruebas basadas en la experiencia, ponemos a un lado la distinción entre pruebas estructurales y de comportamiento, y utilizamos todo lo que