I. E. S. MARE NOSTRUM
Beato Fco. Castelló Aleu s/n • ☎
965936520
03008 ALICANTE
www.iesmarenostrum.com
correo@iesmarenostrum.com

UNIDAD 5
DISEÑO Y REALIZACIÓN DE PRUEBAS

Contenido

1 INTRODUCCIÓN

Una de las últimas fases del ciclo de vida antes de entregar un programa para su explotación, es la fase de pruebas.

Una de las sorpresas con las que suelen encontrar los nuevos programadores es la enorme cantidad de tiempo y esfuerzo que requiere esta fase. Se estima que la mitad del esfuerzo de desarrollo de un programa (tanto en tiempo como en gastos) se va en esta fase. Si hablamos de programas que involucran vidas humanas (medicina, equipos nucleares, etc.) el costo de la fase de pruebas puede fácilmente superar el 80%.

Pese a su enorme impacto en el coste de desarrollo, es una fase que muchos programadores aún consideran clasificable como un arte y, por tanto, como difícilmente conceptualizable. Es muy difícil entrenar a los nuevos programadores, que aprenderán mucho más de su experiencia que de lo que les cuenten en los cursos de programación.

1. ¿Qué es probar?

Los fallos de software ocasionan graves pérdidas económicas; éstos son 100 a 1000 veces más costosos de encontrar y reparar después de la construcción. Entonces, los objetivos fundamentales de la fase de prueba son:

- Evitar plazos y presupuestos incumplidos, insatisfacción del usuario, escasa productividad y mala calidad en el software producido y finalmente la pérdida de clientes.

- Automatizar el proceso de pruebas (consigue reducciones de hasta un 75% en el costo de la fase de mantenimiento).

Como parte que es de un proceso industrial, la fase de pruebas añade valor al producto que se maneja: todos los programas tienen errores y la fase de pruebas los descubre; ese es el valor que añade. El objetivo específico de la fase de pruebas es encontrar cuantos más errores, mejor.

Es frecuente encontrarse con el error de afirmar que el objetivo de esta fase es convencerse de que el programa funciona bien. En realidad ese es el objetivo propio de las fases anteriores (¿quién va a pasar a la sección de pruebas un producto que sospecha que está mal?). Cumplido ese objetivo lo mejor posible, se pasa a pruebas.

Probar un programa es el "proceso de ejecutar un programa con el fin de encontrar errores"

2. Alcance (cobertura) de la prueba

La prueba ideal de un sistema sería exponerlo a todas las situaciones posibles, así encontraríamos hasta el último fallo. Indirectamente, garantizamos su respuesta ante cualquier caso que se le presente en la ejecución real. Esto es imposible desde todos los puntos de vista: humano, económico e incluso matemático.

Dado que todo es finito en programación (el número de líneas de código, el número de variables, el número de valores en un tipo, etc.) cabe pensar que el número de pruebas posibles es finito. Esto deja de ser cierto en cuanto entran en juego bucles, en los que es fácil introducir condiciones para un funcionamiento sin fin. Aún en el irrealista caso de que el número de posibilidades fuera finito, el

número de combinaciones posibles es tan enorme que se hace imposible su identificación y ejecución a todos los efectos prácticos.

Probar un programa es someterle a todas las posible variaciones de los datos de entrada, tanto si son válidos como si no lo son. Imagínese hacer esto con un compilador de cualquier lenguaje: ¡habría que escribir, compilar y ejecutar todos y cada uno de los programas que se pudieran escribir con dicho lenguaje!

Sobre esta premisa de imposibilidad de alcanzar la perfección, hay que buscar formas humanamente abordables y económicamente aceptables de encontrar errores.

3. Tipos de pruebas

Existen numerosas clasificaciones para las pruebas, dependiendo del criterio que se elija para realizar dicha clasificación.

A. En función del grado de conocimiento del código

- Pruebas de caja negra (Pruebas Funcionales): no conocemos la implementación del código, sólo la interfaz. Tan sólo podemos probar dando distintos valores a las entradas y salidas.
- Pruebas de caja blanca (Pruebas Estructurales): conocemos el código (la implementación de éste) que se va a ejecutar y podemos definir las pruebas que cubran todos los posibles caminos del código.

B. Según el grado de automatización

- Pruebas manuales: son las que se hacen normalmente al programar o las que ejecuta una persona con la documentación generada durante la codificación

(P. ej.- comprobar cómo se visualiza el contenido de una página web en dos navegadores diferentes).

- Pruebas automáticas: se usa un determinado software para sistematizar las pruebas y obtener los resultados de las mismas (P. ej.- verificar un método de ordenación).

C. En función de la fase del ciclo de vida en que se prueba

- Pruebas unitarias: se aplican a un componente del software. Podemos considerar como componente (elemento indivisible) a una función, una clase, una librería, etc. En nuestro caso, generalmente hablaremos de una clase como componente de software.
- Pruebas de integración: consiste en construir el sistema a partir de los distintos componentes y probarlo con todos integrados. Estas pruebas deben realizarse progresivamente. 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.
- Pruebas funcionales: sobre el sistema funcionando se comprueba que cumple con la especificación (normalmente a través de los casos de uso).
- Pruebas de rendimiento: los tres primeros tipos de pruebas de los que ya se ha hablado comprueban la eficacia del sistema. Las pruebas de rendimiento se basan en comprobar que el sistema puede soportar el volumen de carga definido en la especificación, es decir, hay que comprobar la eficiencia (P. ej.-

Se ha montado una página web sobre un servidor y hay que probar qué capacidad tiene, estado de aceptar peticiones).

- Pruebas de aceptación: son las únicas pruebas que son realizadas por los usuarios, todas las anteriores las lleva a cabo el equipo de desarrollo. Podemos distinguir entre dos pruebas:
- Pruebas alfa: las realiza el usuario en presencia de personal de desarrollo del proyecto haciendo uso de una máquina preparada para tal fin.
- Pruebas beta: las realiza el usuario después de que el equipo de desarrollo les entregue una versión casi definitiva del producto.

El cliente final decide qué pruebas va a aplicarle al producto antes de darlo por bueno y pagarlo. De nuevo, el objetivo del que prueba es encontrar los fallos lo antes posible, en todo caso antes de pagarlo y antes de poner el programa en producción.

4. Otros conceptos relacionados con las pruebas de software

- Completitud: nos da una idea del grado de fiabilidad de las pruebas y por consiguiente la fiabilidad del software. No es posible llegar al 100% puesto que nunca llegaremos a realizar todas las pruebas posibles al software, puesto que las pruebas tienen un coste (P. ej.- Bug del Excel).
- Depuración: ejecución controlada del software que nos permite corregir un error (P. ej.-Usar el debugger de nuestra máquina).

5. Otras pruebas

A. Regresión (regression testing)

Todos los sistemas sufren una evolución a lo largo de su vida activa. En cada nueva versión se supone que o bien se corrigen defectos, o se añaden nuevas funciones, o ambas cosas. En cualquier caso, una nueva versión exige una nueva pasada por las pruebas. Si éstas se han sistematizado en una fase anterior, ahora pueden volver a pasarse automáticamente, simplemente para comprobar que las modificaciones no provocan errores donde antes no los había.

Las pruebas de regresión son particularmente espectaculares cuando se trata de probar la interacción con un agente externo. Existen empresas que viven de comercializar productos que "graban" la ejecución de una prueba con operadores humanos para luego repetirla cuantas veces haga falta "reproduciendo la grabación". Y, obviamente, deben monitorizar la respuesta del sistema en ambos casos, compararla, y avisar de cualquier discrepancia significativa.

B. Recorridos (walkthroughs)

Quizás es una técnica más aplicada en control de calidad que en pruebas. Consiste en sentar alrededor de una mesa a los desarrolladores y a una serie de críticos, bajo las órdenes de un moderador que impida un recalentamiento de los ánimos. El método consiste en que los revisores se leen el programa línea a línea y piden explicaciones de todo lo que no está meridianamente claro. Puede que simplemente falte un comentario explicativo, o que detecten un error auténtico o que simplemente el código sea tan complejo de entender/explicar que más vale que se rehaga de forma más simple. Para un sistema complejo pueden hacer falta muchas sesiones.

Esta técnica es muy eficaz localizando errores de naturaleza local; pero falla estrepitosamente cuando el error deriva de la interacción entre dos partes alejadas del programa. Nótese que no se está ejecutando el programa, sólo mirándolo con lupa, y de esta forma sólo se ve en cada instante un trocito del listado.

C. Aleatorias (random testing)

Ciertos autores consideran injustificada una aproximación sistemática a las pruebas. Alegan que la probabilidad de descubrir un error es prácticamente la misma si se hacen una serie de pruebas aleatoriamente elegidas, que si se hacen siguiendo las instrucciones dictadas por criterios de cobertura (caja negra o blanca).

Como esto es muy cierto, probablemente sea muy razonable comenzar la fase de pruebas con una serie de casos elegidos al azar. Esto pondrá de manifiesto los errores más patentes. No obstante, pueden permanecer ocultos errores más sibilinos que sólo se muestran ante entradas muy precisas.

Si el programa es poco crítico (una aplicación personal, un juego, ...) puede que esto sea suficiente. Pero si se trata de una aplicación militar o con riesgo para vidas humanas, es de todo punto insuficiente.

D. Aguante (stress testing)

En ciertos sistemas es conveniente saber hasta dónde aguantan, bien por razones internas (¿hasta cuantos datos podrá procesar?), bien externas (¿es capaz de trabajar con un disco al 90%?, ¿aguanta una carga de la CPU del 90?, etc.)

E. Prestaciones (performance testing)

A veces es importante el tiempo de respuesta, u otros parámetros de gasto. Típicamente nos puede preocupar cuánto tiempo le lleva al sistema procesar tantos

datos, o cuánta memoria consume, o cuánto espacio en disco utiliza, o cuántos datos transfiere por un canal de comunicaciones, o ... Para todos estos parámetros suele ser importante conocer cómo evolucionan al variar la dimensión del problema (por ejemplo, al duplicarse el volumen de datos de entrada).

2 PROCEDIMIENTOS Y CASOS DE PRUEBA. PRUEBAS DE CÓDIGO. CUBRIMIENTO. VALORES LÍMITE. CLASES DE EQUIVALENCIA

1. Definición de caso de prueba

Un caso de prueba es el conjunto formado por unas entradas (de prueba), unas condiciones de ejecución y unos resultados esperados. Tiene como objetivo concreto realizar una prueba concreta del código.

Supongamos que queremos diseñar un caso de prueba para un procedimiento que nos valida el 'login' al sistema. Un posible caso de prueba podría ser el siguiente:

- ENTRADA: usuario "hacker" password "kaixo"
- CONDICIONES DE EJECUCIÓN: no existe en la tabla CUENTA(usuario,pass,intentos) la tupla <"hacker", "kaixo",x> pero sí una tupla <"hacker","hola",x>
- RESULTADO ESPERADO: no deja entrar y cambia la tupla a <"hacker","hola",x+1>

El objetivo del caso de prueba es comprobar que no se deja entrar a un usuario existente con un password equivocado.

El *Procedimiento de prueba* serían los pasos que hay que llevar a cabo para probar uno (o varios) casos de prueba. Para el caso anterior, un posible procedimiento de prueba podría ser el siguiente:

1) Ejecutar la clase Presentacion
2) Comprobar que en la BD "passwords.mdb"existe la tupla <"hacker","hola",x>
3) Escribir "hacker"en la interfaz gráfica (en el campo de texto etiquetado "Escribe nombre usuario")
4) Escribir "kaixo"en la interfaz gráfica (en el campo de texto "Escribe password")
5) Pulsar botón "Acceder al sistema"
6) Comprobar que no deja entrar al sistema y que en la BD la tupla ha cambiado a <"hacker","hola",x+1>

Un buen caso de prueba es aquel que tiene una probabilidad muy alta de descubrir un nuevo error. Un caso de prueba no debe ser redundante, y debe ser el mejor de un conjunto de pruebas de propósito similar.

No debe ser ni muy sencillo ni excesivamente complejo: es mejor realizar cada prueba de forma separada si se quiere probar diferentes casos.

Una prueba tiene éxito si descubre un error. Eso sí, no asegura la ausencia de defectos, sólo puede demostrar que existen errores en el software.

Para diseñar los casos de prueba, vamos a estudiar dos técnicas: pruebas de caja blanca y pruebas de caja negra.

2. Caja blanca (pruebas estructurales, caja transparente)

Las pruebas de caja blanca aseguran que el código interno del programa se ajusta a las especificaciones y que todos los componentes internos se han probado adecuadamente, intentando garantizar que todos los caminos de ejecución del programa quedan probados.

En estas pruebas estamos siempre observando el código, que las pruebas se dedican a ejecutar con ánimo de "probarlo todo". Esta noción de prueba total se formaliza en lo que se llama "cobertura" y no es sino una medida porcentual de ¿cuánto código hemos cubierto?

Hay diferentes posibilidades de definir la cobertura. Todas ellas intentan sobrevivir al hecho de que el número posible de ejecuciones de cualquier programa no trivial es (a todos los efectos prácticos) infinito. Pero si el 100% de cobertura es infinito, ningún conjunto real de pruebas pasaría de un infinitésimo de cobertura. Esto puede ser muy interesante para los matemáticos; pero no sirve para nada.

A. 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.

El número de sentencias de un programa es finito. Basta coger el código fuente e ir contando. Se puede diseñar un plan de pruebas que vaya ejercitando más y más sentencias, hasta que hayamos pasado por todas, o por una inmensa mayoría.

En la práctica, el proceso de pruebas termina antes de llegar al 100%, pues puede ser excesivamente laborioso y costoso provocar el paso por todas y cada una de las sentencias.

A la hora de decidir el punto de corte antes de llegar al 100% de cobertura hay que ser precavido y tomar en consideración algo más que el índice conseguido.

B. Cobertura de ramas

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

```
IF Condicion THEN
     EjecutaEsto;
```

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 CASE.

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. Esto es cierto salvo en programas triviales que carecen de condiciones (a cambio, basta 1 sola prueba para cubrirlo desde todos los puntos de vista). El criterio también debe refinarse en lenguajes que admiten excepciones (por ejemplo, Ada). En estos casos, hay que añadir pruebas para provocar la ejecución de todas y cada una de las excepciones que pueden dispararse.

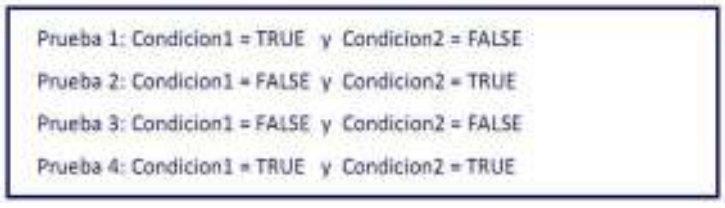
C. Cobertura de condición/decisión

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

```
IF Condicion1 OR Condicion2 THEN
     HazEsto;
```

Las condiciones 1 y 2 pueden tomar 2 valores cada una, dando lugar a 4 posibles combinaciones. No obstante sólo 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:



Bastan las pruebas 2 y 3 para tener cubiertas todas las ramas. Pero con ellos sólo hemos probado una posibilidad para la Condición1.

Para afrontar esta problemática se define un criterio de cobertura de condición/decisión que trocea las expresiones booleanas complejas en sus componentes e intenta cubrir todos los posibles valores de cada uno de ellos.

D. 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 inagotable de errores, todos triviales, algunos mortales. Un bucle se ejecuta un cierto número de veces; pero ese número de veces debe ser muy preciso, y lo más normal es que ejecutarlo una vez de menos o una vez de más tenga consecuencias indeseables. Y, sin embargo, es extremadamente fácil equivocarse y redactar un bucle que se ejecuta 1 vez de más o de menos.

Para un bucle de tipo WHILE hay que pasar 3 pruebas:

1. 0 ejecuciones
2. 1 ejecución
3. más de 1 ejecución

Para un bucle de tipo DO-WHILE hay que pasar 2 pruebas:

1. 1 ejecución
2. más de 1 ejecución

Los bucles FOR, en cambio, son muy seguros, pues en su cabecera está definido el número de veces que se va a ejecutar. Ni una más, ni una menos, y el compilador se encarga de garantizarlo. Basta pues con ejecutarlos 1 vez.

No obstante, conviene no engañarse con los bucles FOR y examinar su contenido. Si dentro del bucle se altera la variable de control, o el valor de alguna variable que se utilice en el cálculo del incremento o del límite de iteración, entonces eso es un bucle FOR con trampa.

También tiene "trampa" si contiene sentencias del tipo EXIT (que algunos lenguajes denominan BREAK) o del tipo RETURN. Todas ellas provocan terminaciones anticipadas del bucle.

E. Y en la práctica ¿qué hago?

En la práctica de cada día, se suele procura alcanzar una cobertura cercana al 100% de segmentos. Es muy recomendable (aunque cuesta más) conseguir una buena cobertura de ramas. En cambio, no suele hacer falta ir a por una cobertura de decisiones atomizadas.

¿Qué es una buena cobertura? Pues depende de lo crítico que sea el programa. Hay que valorar el riesgo (o coste) que implica un fallo si éste se descubre durante la aplicación del programa. Para la mayor parte del software que se produce en Occidente, el riesgo es simplemente de imagen (si un juego fallece a mitad, queda muy feo; pero no se muere nadie). En estas circunstancias, coberturas del 60-80% son admisibles.

La cobertura requerida suele ir creciendo con el ámbito previsto de distribución. Si un programa se distribuye y falla en algo grave puede ser necesario redistribuirlo de nuevo y urgentemente. Si hay millones de clientes dispersos por varios países, el coste puede ser brutal. En estos casos hay que exprimir la fase de pruebas para que encuentre prácticamente todos los errores sin pasar nada por alto. Esto se traduce al final en buscar coberturas más altas.

Es aún más delicado cuando entramos en aplicaciones que involucran vidas humanas (aplicaciones sanitarias, centrales nucleares, etc) Cuando un fallo se traduce en una muerte, la cobertura que se busca se acerca al 99% y además se presta atención a las decisiones atómicas.

También se suele perseguir coberturas muy elevadas (por encima del 90%) en las aplicaciones militares. Esto se debe a que normalmente van a ser utilizadas en condiciones muy adversas donde el tiempo es inestimable. Si un programa fallece, puede no haber una segunda oportunidad de arrancarlo de nuevo.

La ejecución de pruebas de caja blanca puede llevarse a cabo con un depurador (que permite le ejecución paso a paso), un listado del módulo y un rotulador para ir marcando por dónde vamos pasando. Esta tarea es muy tediosa, pero puede ser automatizada. Hay compiladores que a la hora de generar código máquina dejan incrustado en el código suficiente código como para poder dejar un fichero (tras la ejecución) con el número de veces que se ha ejecutado cada sentencia, rama, bucle, etc.

F. Limitaciones

Lograr una buena cobertura con pruebas de caja blanca es un objetivo deseable; pero no suficiente a todos los efectos. Un programa puede estar perfecto en todos sus términos, y sin embargo no servir a la función que se pretende.

Por ejemplo, un Rolls-Royce es un coche que sin duda pasaría las pruebas más exigentes sobre los últimos detalles de su mecánica o su carrocería. Sin embargo, si el cliente desea un todo-terreno, difícilmente va a comprárselo.

Otro ejemplo, si escribimos una rutina para ordenar datos por orden ascendente, pero el cliente los necesita en orden decreciente; no hay prueba de caja blanca capaz de detectar el error.

Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace; pero no de que haga lo que necesitamos.

G. Ejemplo

Supongamos el siguiente código para comprobar si un número es o no primo:

```
static void Main(string[] args)
{
            int j = 2, s = 0, n;

            if (textBox1.Text == "")
                textBox2.Text = "Introduce un número";
            else {
                try
                {
                        n = int.Parse(textBox1.Text);

                        while (j < n)
                        {
                            if (n % j == 0)
                                s = s + 1;

                            j = j + 1;
                        }


                        if (s == 0)
                            textBox2.Text = n + " Es un número primo";
                        else
                            textBox2.Text = n + " No es un número primo";
                }
                catch
                {

                    MessageBox.Show("Has introducido un carácter");
                }
             }
}
```

NOTA: try{} y catch{} son funciones para el manejo de excepciones –errores-. Si dentro de try{} se produjera algún error, automáticamente se ejecutaría el contenido del catch.

Un posible caso de prueba sería el siguiente:

Entrada	Salida	COMENTARIO
""	"Introduce un número"	Se cumple la condición del primer if.
a b	Mensaje Box	Se produce una excepción y se ejecuta el contenido del catch.
1	Primo	Recorro 0 veces el bucle; no se cumple la condición del 1º if; se cumple la condición del 3º if.
3	Primo	Recorro 1 vez el buble; no se cumple la condición del 1º if; no se cumple la condición del 2º if; se cumple la condición del 3º if
4	No primo	Recorro 2 veces el buble; no se cumple la condición del 1º if; se cumple la condición del 2º if.; no se cumple la condición del 3º if
23	Primo	Recorro n veces el buble; no se cumple la condición del 1º if; no se cumple la condición del 2º if; se cumple la condición del 3º if

Con todas estas entradas se prueban todas las coberturas de bucles y de condición, ademas de todos los posibles caminos.

3. Caja negra (pruebas de caja opaca, pruebas funcionales, pruebas de entrada/salida, pruebas inducidas por los datos)

A. Introducción

Las pruebas de caja negra se centran en lo que se espera de un módulo, es decir, intentan encontrar casos en que el módulo no se atiene a su especificación. Por ello se denominan pruebas funcionales, y el probador se limita a suministrarle datos como entrada y estudiar la salida, sin preocuparse de lo que pueda estar haciendo el módulo por dentro.

Las pruebas de caja negra se apoyan en la especificación de requisitos del módulo. De hecho, se habla de "cobertura de especificación" para dar una medida del número de requisitos que se han probado. Es fácil obtener coberturas del 100% en módulos internos, aunque puede ser más laborioso en módulos con interfaz al exterior. En cualquier caso, es muy recomendable conseguir una alta cobertura en esta línea.

El problema con las pruebas de caja negra no suele estar en el número de funciones proporcionadas por el módulo (que siempre es un número muy limitado en diseños razonables); sino en los datos que se le pasan a estas funciones. El conjunto de datos posibles suele ser muy amplio (por ejemplo, un entero).

A la vista de los requisitos de un módulo, se sigue 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 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 3 clases de equivalencia: por debajo, en y por encima del rango.
- si una entrada requiere un valor concreto, aparecen 3 clases de equivalencia: por debajo, en y por encima del valor.

- si una entrada requiere un valor de entre los de un conjunto, aparecen 2 clases 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.

B. Ejemplos

Supongamos que utilizamos un valor entero para identificar el día del mes. Los valores posibles están en el rango [1..31]. Así, hay 3 clases de equivalencia:

1. números menores que 1
2. números entre 1 y 31
3. números mayores que 31

Otro posible ejemplo: elaborar una batería de pruebas para un programa que analiza la validez de una palabra clave. Una clave es válida cuando cumple los siguientes requisitos:

- Está formada por más de 7 y menos de 13 caracteres.
- Los caracteres permitidos son:
  - Las letras a – z, A – Z.
  - Los dígitos 0 – 9
  - El carácter especial % .
- Contiene al menos dos letras.
- Contiene al menos un carácter que no es letra.

Clase de equivalencia:

- Palabra clave

1. 7<=NumCaracteres<=13
2. NumCaracteres>13
3. NumCaracteres<7
4. La palabra contiene dos letras.
5. La palabra contiene menos de dos letras.
6. La palabra contiene más de dos letras.
7. La palabra contiene un caracter no letra.
8. La palabra contiene menos de un caracter no letra.
9. La palabra contiene más de un un caracter no letra.

- Salida
  1. True
  2. False

Durante la lectura de los requisitos del sistema, nos encontraremos con una serie de valores singulares, que marcan diferencias de comportamiento. Estos valores son claros candidatos a marcar clases de equivalencia: por abajo y por arriba.

Una vez identificadas las clases de equivalencia significativas en nuestro módulo, se procede a coger 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.

La experiencia muestra que un buen número de errores aparecen en torno a los puntos de cambio de clase de equivalencia. Hay una serie de valores denominados "frontera" o valores límite que conviene probar, además de los elegidos en el párrafo anterior (clase de equivalencia). Usualmente se necesitan 2 valores por frontera, uno justo abajo y otro justo encima:

- Si la entrada se encuentra en el rango a..b entonces hay que probar con los valores : a -1, a, a + 1, b - 1, b y b + 1
- Si la entrada es un conjunto de valores entonces hay que probar con los valores : max-1, max, max+1, min-1, min y min+1

C. Limitaciones

Lograr una buena cobertura con pruebas de caja negra es un objetivo deseable; pero no suficiente a todos los efectos. Un programa puede pasar con holgura millones de pruebas y sin embargo tener defectos internos que surgen en el momento más inoportuno (Murphy no olvida).

Por ejemplo, un PC que contenga el virus Viernes-13 puede estar pasando pruebas de caja negra durante años y años. Sólo falla si es viernes y es día 13; pero ¿a quién se le iba a ocurrir hacer esa prueba?

Las pruebas de caja negra nos convencen de que un programa hace lo que queremos; pero no de que haga (además) otras cosas menos aceptables.

D. Caso Práctico

Los ejemplos de pruebas de programas suelen irse a uno de dos extremos: o son triviales y no se aprende nada, o son tan enormes que resultan tediosos. El ejemplo elegido para esta sección pretende ser comedido, a costa de no contemplar más que un reducido espectro de casos.

Nos dan para probar una función

```
Funcion Busca (C: CHAR; V: ARRAY OF CHAR): BOOLEAN;
```

A este procedimiento se le proporciona un carácter C y un array V de caracteres. Se admitirá cualquier carácter de 8 bits de los representables en un PC. El ARRAY podrá tener entre 0 y 10 caracteres. El procedimiento devuelve TRUE si C está en V, y FALSE si no.

Con estas explicaciones identificamos las siguientes clases de equivalencia

- C: CHAR
    1. Cualquier carácter.
- V: ARRAY OF CHAR
    1. El ARRAY vacio.
    2. Un ARRAY entre 1 y 10 elementos, ambos inclusive.
    3. Un ARRAY con más de 10 elementos.
- resultado: BOOLEAN
    1. TRUE
    2. FALSE

Por último, cabe considerar combinaciones significativas de datos de entrada: que C sea el primero o el último del ARRAY.

1. Pruebas de caja negra: valores límite.

Carácter	vector	Salida	Prueba
k	" "	False	1
k	k	True	2
k	kl	True	3
k	jk	True	4
k	j	False	5
k	aaaaaaaaaa	False	6
k	aaaaaaaaa	False	7
k	aaaaaaaaaaa	Vector > 10	8

Los valores límite para el vector:

- Valor min=1, probamos el vector con min-1 elemento, min elementos y min+1 elemento.
- Valor max=10, probamos el vector con max-1 elementos, max elementos y max+1 elementos.
- Cuando probemos el vector con min elementos, probamos que el carácter exista y que no exista.
- Cuando probemos el vector con min+1 elementos, probamos que el carácter esté en la primera posición y en la última. Para max-1 no haría falta hacerlo, porque estaría comprobando lo mismo que con min+1.
- Cuando probemos el vector con max+1 elementos, el resultado que nos dará el programa será un mensaje de error, ya que el vector como máximo tendrá 10 elementos.

2. Pruebas de caja negra: valores normales

Carácter	vector	Salida	Prueba
k	abc	False	9
k	jkl	True	10
1	jkl	Carácter no es una letra	11
K	L	False	12

Para pasar a caja blanca necesitamos conocer el código interno:

```
1    public bool Buscar(char c, char[] vector)
       {
3     bool esta = false;
4     int a = 0, z, m;

5     if(vector.Length>10)
```

```
            {
6                   MessageBox.Show("El vector tienen una dimensión demadiado grande");
            }
7           else
            {

8              if((c<'a' || c>'z') && (c<'A' || c>'Z'))
               {
9                MessageBox.Show("El carácter no es una letra");
               }
10             else
               {
11                   z = vector.Length - 1;

12                   while (a <= z && !esta)
                     {
13                       m = (a + z) / 2;

14                       if (vector[m] == c)
15                          esta = true;
16                       else
                         {
17                          if (vector[m] < c)
18                            a = m + 1;
19                          else
20                            z = m - 1;
                         }
                     }

               }
            }
21      return esta;
        }
```

Es laborioso; pero si nos molestamos en ejecutar todas las pruebas anteriores marcando por dónde vamos pasando sobre el código, nos encontraremos con que hemos ejecutado todas las sentencias con excepción de la rama de la línea 20. Para atacar este caso necesitamos un caso de prueba adicional de caja blanca.

3. Prueba de caja blanca:

Carácter	vector	Salida	Prueba
k	1	False	13

Con el conjunto de pruebas que llevamos hemos logrado una cobertura al 100% de segmentos y de condiciones. Respecto del bucle, la prueba 1 lo ejecuta 0 veces, y las demás pruebas 1 o más veces.

3 PLANIFICACIÓN DE PRUEBAS. PRUEBAS UNITARIAS. PRUEBAS DE INTEGRACIÓN. PRUEBAS DEL SISTEMA. PRUEBAS DE ACEPTACIÓN. AUTOMATIZACIÓN DE PRUEBAS

1. Plan de pruebas

Un plan de pruebas está constituido por un conjunto de pruebas. Cada prueba debe:

- dejar claro qué tipo de propiedades se quieren probar (corrección, robustez, fiabilidad, amigabilidad, ...)
- dejar claro cómo se mide el resultado
- especificar en qué consiste la prueba (hasta el último detalle de cómo se ejecuta)
- definir cual es el resultado que se espera (identificación, tolerancia, ...) ¿Cómo se decide que el resultado es acorde con lo esperado?

Respecto al orden de pruebas, una práctica frecuente es la siguiente:

1. Pasar pruebas de caja negra analizando valores límite. Recuerde que hay que analizar condiciones límite de entrada y de salida.
2. Identificar clases de equivalencia de datos (entrada y salida) y añadir más pruebas de caja negra para contemplar valores normales (en las clases de equivalencia en que estos sean diferentes de los valores límite; es decir, en rangos amplios de valores).
3. Añadir pruebas basadas en "presunción de error". A partir de la experiencia y el sentido común, se aventuran situaciones que parecen proclives a padecer

defectos, y se buscan errores en esos puntos. Son pruebas del tipo "¡Me lo temía!"

4. Medir la cobertura de caja blanca que se ha logrado con las fases previas y añadir más pruebas de caja blanca hasta lograr la cobertura deseada. Normalmente se busca una buena cobertura de ramas (revise los comentarios expuestos al hablar de caja blanca).

2. Estrategias de aplicación de pruebas

Las pruebas comienzan a nivel de módulo. Una vez terminadas, progresan hacia la integración del sistema completo y su instalación. Culminan cuando el cliente acepta el producto y se pasa a su explotación inmediata.

Para ello se deberá elaborar un documento con el plan de pruebas. El objetivo del documento es señalar el enfoque, los recursos y el esquema de actividades de prueba, así como los elementos a probar, las características, las actividades de prueba, el personal responsable y los riesgos asociados.

Un posible esquema del documento del plan de pruebas podría ser el siguiente:

1. Identificador único del documento
2. Introducción y resumen de elementos y características a probar
3. Elementos software a probar
4. Características a probar
5. Características que no se probarán
6. Enfoque general de la prueba
7. Criterios de paso/fallo para cada elemento
8. Criterios de suspensión y requisitos de reanudación
9. Documentos a entregar

10. Actividades de preparación y ejecución de pruebas
11. Necesidades de entorno
12. Responsabilidades en la organización y realización de las pruebas
13. Necesidades de personal y formación
14. Esquema de tiempos
15. Riesgos asumidos por el plan y planes de contingencias
16. Aprobaciones y firmas con nombre y puesto desempeñado

Junto a este documento**,** se pueden también encontrar los siguientes:

- Documento de especificación del diseño de pruebas: especifica los refinamientos necesarios sobre el enfoque general reflejado en el plan e identifica las características que se deben probar con este diseño de pruebas
- Documento de especificación de caso de prueba: define uno de los casos de prueba identificado por una especificación del diseño de las pruebas.
- Documento de especificación de procedimientos de prueba: especifica los pasos para la ejecución de un conjunto de casos de prueba o, más generalmente, los pasos utilizados para analizar un elemento software con el propósito de evaluar un conjunto de características del mismo

A. Prueba de unidad

Hablamos de una unidad de prueba para referirnos a uno o más módulos que cumplen las siguientes condiciones:

- Todos son del mismo programa
- Al menos uno de ellos no ha sido probado
- El conjunto de módulos es el objeto de un proceso de prueba

Se trata de las pruebas formales que permiten declarar que un módulo está listo y terminado (no las informales que se realizan mientras se desarrollan los módulos)

Especialmente orientado a la prueba de la caja blanca aunque se completa también con la prueba de la caja negra. Se prueban los caminos de control importantes para descubrir errores dentro del límite del módulo. Puede realizarse paralelamente en varios módulos.

B. Prueba integración

Los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto. Ha de tener en cuenta las interfaces entre componentes de la arquitectura del software.

Implican una progresión ordenada de pruebas que van desde los componentes o módulos y que culminan en el sistema completo. El orden de integración elegido afecta a diversos factores, como los siguientes:

- La forma de preparar casos
- Las herramientas necesarias
- El orden de codificar y probar los módulos
- El coste de la depuración y preparación de casos

Los tipos fundamentales de integración son:

- Integración incremental: Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados
- Ascendente: Se comienza por los módulos hoja
- Descendente: Se comienza por el módulo raíz

- Integración no incremental: Se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo

Habitualmente, las pruebas de unidad y de integración se solapan y mezclan en el tiempo. Al añadir un nuevo módulo, el software cambia y se establecen nuevos caminos de flujo de datos, nueva E/S y nueva lógica de control. Puede haber problemas con funciones que antes iban bien. Se ejecutarán un conjunto de pruebas que se han realizado anteriormente para asegurarse que los cambios no han dado lugar a cambios colaterales (Prueba de regresión)

C. Prueba validación

El software totalmente ensamblado se prueba como un conjunto para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimientos, seguridad, etc.

Se lleva a cabo cuando se ha terminado la prueba de integración: el software está ensamblado y se han eliminado todos los errores de la interfaz.

La validación se consigue cuando el software funciona según las expectativas del usuario. Generalmente son pruebas de caja negra.

D. Prueba sistema

El software ya validado se integra con el resto del sistema (Ej.: elementos mecánicos, interfaces electrónicas..) para probar su funcionamiento conjunto. Se efectúan además sobre procesos que afectan a la seguridad en el acceso a los datos, rendimiento bajo condiciones de grandes cargas de trabajo, tolerancia a fallos y recuperación del sistema. Se ejecutan y se observa su rendimiento en condiciones

límite y de sobrecarga. Estas pruebas sirven para verificar que se han integrado adecuadamente todos los elementos del sistema y que se realizan las funciones apropiadas

E. Prueba aceptación

Por último, el producto final se pasa a la prueba de aceptación para que el usuario compruebe en su propio entorno de explotación si lo acepta como está o no.

Se considera que es la fase final del proceso para crear una confianza en que el producto es apropiado para su uso.

3. Automatización de pruebas

La automatización de pruebas consiste en el uso de software especial (casi siempre separado del software que se prueba) para controlar la ejecución de pruebas y la comparación entre los resultados obtenidos y los resultados esperados. La automatización de pruebas permite incluir pruebas repetitivas y necesarias dentro de un proceso formal de pruebas ya existente o bien añadir pruebas cuya ejecución manual resultaría difícil.

Algunas pruebas de software tales como las pruebas de regresión intensivas de bajo nivel pueden ser laboriosas y consumir mucho tiempo para su ejecución si se realizan manualmente. Adicionalmente, una aproximación manual puede no ser efectiva para encontrar ciertos tipos de defectos, mientras que las pruebas automatizadas ofrecen una alternativa que lo permite. Una vez que una prueba ha sido automatizada, ésta puede ejecutarse repetitiva y rápidamente en particular con productos de software que tienen ciclos de mantenimiento largo, ya que incluso cambios relativamente menores en la vida de una aplicación pueden inducir fallos

en funcionalidades que anteriormente operaban de manera correcta. Existen dos aproximaciones a las pruebas automatizadas:

- Pruebas manejadas por el código: Se prueban las interfaces públicas de las clases, módulos o bibliotecas con una variedad amplia de argumentos de entrada y se valida que los resultados obtenidos sean los esperados. Este tipo de pruebas se realiza a través de herramientas específicas de los IDEs, como pueden ser JUnit (Java) o NUnit (Visual Studio), que permiten la ejecución de pruebas unitarias para determinar cuándo varias secciones del código se comportan como es esperado en circunstancias específicas. Los casos de prueba describen las pruebas que han de ejecutarse sobre el programa para verificar que éste se ejecuta tal y como se espera.
- Pruebas de Interfaz de Usuario: Un marco de pruebas genera un conjunto de eventos de la interfaz de usuario, tales como teclear, hacer click con el ratón e interactuar de otras formas con el software y se observan los cambios resultantes en la interfaz de usuario, validando que el comportamiento observable del programa sea el correcto.

La elección misma entre automatización y ejecución manual de pruebas, los componentes cuya prueba será automatizada, las herramientas de automatización y otros elementos son críticos en el éxito de las pruebas, y por lo general deben provenir de una elección conjunta de los equipos de desarrollo, control de calidad y administración. Un ejemplo de mala elección para automatizar, sería escoger componentes cuyas características son inestables o su proceso de desarrollo implica cambios continuos.