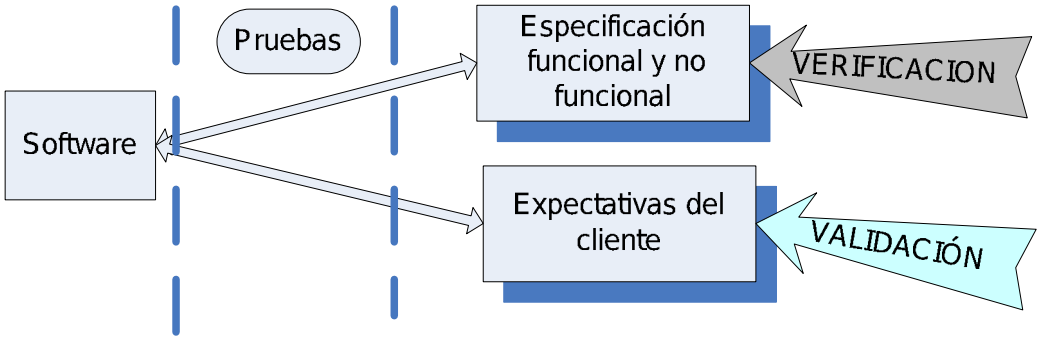
2. Verificación y validación del software

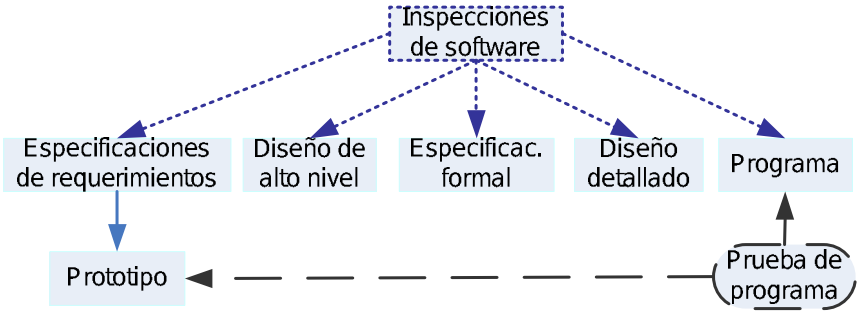
Durante y después del proceso de implementación, el programa que se está desarrollando debe ser comprobado para asegurar que satisface su especificación y entrega la funcionalidad esperada por el cliente. La **verificación** implica comprobar que el software está de acuerdo con su especificación. Debería comprobarse que satisface sus requerimientos funcionales y no funcionales. ¿Estamos construyendo el producto correctamente?
La **validación** pretende asegurar que el sistema software satisface las expectativas del cliente, es decir, demostrar que el software hace lo que el cliente quiere que haga. ¿Estamos construyendo el producto correcto?



Dentro del proceso de V&V, existen dos aproximaciones para el análisis y comprobación de los sistemas.

1. **La revisión del software.** Tal como vimos en la sección dedicada a Técnicas para evitar defectos, la revisión del software analiza y comprueba las representaciones del sistema tales como el documento de requerimientos, los diagramas de diseño y el código fuente del programa. Las revisiones pueden usarse en todas las etapas del proceso y consisten en una técnica de V&V estática, ya que no se necesitan ejecutar el software en una computadora.
2. **Las pruebas del software.** Implican ejecutar una implementación del software con datos de prueba. Se examinan las salidas del software y su entorno operacional para comprobar que funciona tal y como se requiere. Las pruebas son una técnica dinámica de verificación y validación.

La siguiente figura muestra que las revisiones del software y las pruebas son actividades complementarias en el proceso del software. Las flechas indican las etapas en el proceso en las que pueden utilizarse dichas técnicas.



Consideraciones:

a. Las revisiones del software y las pruebas son actividades complementarias.
b. Las revisiones de los requerimientos y del diseño son las principales técnicas utilizadas para la detección de errores en el diseño y la especificación.

c. Un sistema puede probarse cuando está disponible un prototipo o una versión ejecutable del programa.
d. Las técnicas estáticas sólo pueden comprobar la correspondencia entre un programa y su especificación (verificación); no pueden demostrar que el software es operacionalmente útil.
e. Tampoco se pueden utilizar técnicas estáticas para comprobar las propiedades emergentes del software tales como su rendimiento y fiabilidad.

Una ventaja del desarrollo incremental es que una versión probable del sistema está disponible en etapas tempranas del proceso de desarrollo. Las funcionalidades pueden probarse a medida que se van añadiendo al sistema, por lo que no tiene que realizarse una implementación completa antes de que comiencen las pruebas.

Aunque el uso de revisiones del software no es generalizado, la prueba de programa siempre será la principal técnica de verificación y validación. Las pruebas implican ejecutar el programa utilizando datos similares a los datos reales procesados por el programa. Los defectos en los programas se descubren examinando las salidas del programa y buscando las anomalías. Existen dos tipos distintos de pruebas que pueden utilizarse en diferentes etapas del proceso del software.

1. **Las pruebas de validación.** Intentan demostrar que el software es el que el cliente quiere –que satisface sus requerimientos-. Como parte de la prueba de validación, se pueden utilizar pruebas estadísticas para probar el rendimiento y la fiabilidad de los programas, y para comprobar cómo trabaja en ciertas condiciones operacionales. Para las pruebas de validación, una prueba con éxito es aquella en la que el sistema funciona correctamente.
2. **Las pruebas de defectos.** Intenta revelar defectos en el sistema en lugar de simular su uso operacional. El objetivo de las pruebas de defectos es hallar inconsistencias entre un programa y su especificación. Para las pruebas de defectos, una prueba con éxito es aquella que muestra un defecto que hace que el sistema funciona incorrectamente. Ahora ¿qué es una buena prueba de defectos?
    - *Una buena prueba tiene una elevada probabilidad de encontrar un error.* Alcanzar este objetivo requiere que la persona que aplica la prueba comprenda el software y trate de desarrollar una imagen mental de la manera en que puede fallar. Lo ideal es probar los tipos de fallas. Por ejemplo, un tipo de falla posible en una interfaz grafica de usuario es la incapacidad de reconocer la posición correcta del mouse; por tanto, se diseñaría un conjunto de pruebas para probarlo tratando de evidenciar un error en el reconocimiento de su posición.
    - *Una buena prueba no es redundante.* El tiempo y los recursos destinados a las pruebas son limitados. No hay razón para realizar una prueba que tenga el mismo propósito que otra. Cada prueba debe tener un propósito diferente (aunque las diferencias sean sutiles).
    - *Una buena prueba debe ser "la mejor de su clase"*. En un grupo de pruebas con un objetivo similar y recursos limitados podría optarse por la ejecución de un solo subconjunto de ellas. En este caso, debe usarse la prueba que tenga la mayor probabilidad de descubrir un tipo completo de errores.
    - *Una buena prueba no debe ser ni muy simple ni demasiado compleja.* Aunque a veces es posible combinar una serie de pruebas en un caso de prueba, los posibles efectos colaterales asociados con este enfoque podrían enmascarar errores. En general, cada prueba debe ejecutarse por separado.

Sin embargo, durante las pruebas de validación, se encontrarán defectos en el sistema. Durante las pruebas de defectos, alguno de los test mostrará que el programa satisface sus requerimientos.

A medida que se descubren defectos en el programa que se está probando, tiene que cambiarse este para corregir tales defectos. Normalmente, los procesos de V&V y depuración se intercalan. Sin embargo, las pruebas (o más generalmente la V&V) y la depuración tienen diferentes objetivos:

a. Los procesos de V&V intentas establecer la existencia de defectos.

b. La depuración es un proceso que localiza y corrige estos defectos.

2.1. Inspecciones de software

Existen tres ventajas fundamentales de la inspección sobre las pruebas:

1. *Durante las pruebas, los errores pueden enmascarar (ocultar) otros errores*. Cuando se descubre un error, nunca se puede estar seguro de si otras anomalías de salida son debido a un nuevo error o son efectos laterales del error original. Debido a que la inspección es un proceso estático, no hay que preocuparse por las interacciones entre errores. Por lo tanto, una única sesión de inspección puede descubrir muchos errores en un sistema.
2. *Pueden inspeccionarse versiones incompletas de un sistema sin costos adicionales*. Si un programa está incompleto, entonces se necesita desarrollar software de soporte especializado para las pruebas a fin de probar aquellas partes que están disponibles. Esto, obviamente, añade costos al desarrollo del sistema.
3. *Además de buscar los defectos en el programa, una inspección también puede considerar atributos de calidad más amplios de un programa* tales como grado de cumplimiento con los estándares, portabilidad y mantenibilidad. Puede buscarse ineficiencias, algoritmos no adecuados y estilos de programación que podrían hacer que el sistema fuese difícil de mantener y actualizar.

Las revisiones y las pruebas tienen cada una sus ventajas e inconvenientes y deberían utilizarse conjuntamente en el proceso de verificación y validación. Expertos en la industria sugieren que uno de los usos más efectivos de las revisiones es la revisión de los casos de prueba para un sistema. Las revisiones pueden descubrir problemas con estos test y ayudar a diseñar formas más efectivas para probar el sistema. Se puede empezar la V&V del sistema con inspecciones en etapas tempranas del proceso de desarrollo, pero una vez que se entrega un sistema, se necesita comprobar sus propiedades emergentes y que la funcionalidad del sistema es la que el propietario realmente quiere.

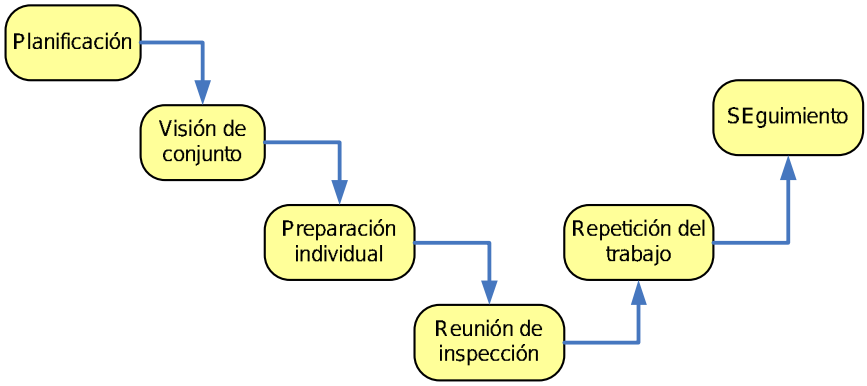
2.1.1. El proceso de inspección de programa

Las inspecciones de programas son revisiones cuyo objetivo es la detección de defectos en el programa. Los defectos pueden ser errores lógicos, anomalías en el código, o el incumplimiento de los estándares del proyecto o de la organización. Por otra parte, otros tipos de revisiones pueden estar más relacionadas con la agenda, los costos, el progreso frente a hitos definidos o la evaluación de si es probable que el software cumpla los objetivos fijados por la organización.

Antes que comience una inspección del programa, es esencial que:

a. Se tenga una especificación precisa del código a inspeccionar. Es imposible inspeccionar un componente a un nivel de detalle requerido para detectar defectos sin una especificación completa.
b. Los miembros del equipo de inspección estén familiarizados con los estándares de la organización.
c. Se haya distribuido una versión compatible y actualizada del código a todos los miembros del equipo. No existe ninguna razón para inspeccionar código que esté "casi completo" incluso si un retraso provoca desfases en la agenda.

Las actividades dl proceso de inspección se muestran en la siguiente figura.



La inspección de programa es un proceso formal realizado por un equipo de al menos cuatro personas. Los miembros del equipo analizan sistemáticamente el código y señalan posibles defectos. El **moderador** del equipo de inspección es el responsable de la planificación de la inspección. Esto implica seleccionar un equipo de inspección, organizar una sala de reuniones y asegurar que el material a inspeccionar y sus especificaciones estén completas. El programa a inspeccionar se presenta al **equipo de inspección** durante la etapa de revisión general cuando el **autor del código** describe lo que el programa debería hacer. A continuación se procede a un período de preparación individual. Cada miembro del equipo de inspección estudia la especificación y el programa y busca defectos en el código.

La inspección en sí misma debería ser bastante corta (no más de dos horas) y debería centrarse en la detección de defectos, cumplimiento de los estándares y programación de baja calidad. El equipo de inspección no debería sugerir cómo deben repararse estos defectos ni recomendar cambios en otros componentes.

A continuación de la inspección, el autor del programa debería realizar los cambios para corregir los problemas identificados. En la etapa siguiente, el moderador debería decidir si se requiere una inspección de código. Puede decidir que no se requiere una reinspección completa y que los defectos han sido reparados con éxito. El programa entonces es aprobado por el moderador para su entrega.

Durante una inspección a menudo se utiliza una lista de comprobación de errores de programación comunes para centrar el análisis. Esta lista de comprobación puede basarse en ejemplos de lista de comprobación de libros o en conocimiento de los defectos que son comunes en un dominio de aplicación particular. Se necesitan diferentes listas de comprobación para distintos lenguajes de programación debido a que cada lenguaje tiene sus propios errores característicos.

En la siguiente tabla se muestran posibles comprobaciones que podrían realizarse durante el proceso de inspección. No obstante, cada organización debería desarrollar su propia **lista de comprobación** de inspecciones basadas en estándares y prácticas locales. Las listas de comprobación deberían ser actualizadas regularmente a medida que se encuentran nuevos tipos de defectos.

Clase de defecto	***Comprobación de inspección***
Defectos de datos	¿SE inicializan todas las variables antes de que se utilicen sus valores? ¿Tienen nombre todas las constantes? ¿El límite superior de los vectores es igual al tamaño del vector o al tamaño 21? Si se utilizan cadenas de caracteres ¿tienen un delimitador explícitamente asignado? ¿Existe alguna posibilidad de que el búfer se desborde?
Defectos de control	Para cada sentencia condicional, ¿es correcta la condición? ¿Se garantiza que termina cada bucle? ¿Están puestas correctamente entre llaves las sentencias compuestas? En la sentencia case, ¿Se tienen en cuenta todos los posibles casos? Si se requiere una sentencia break después de cada caso en las sentencias case ¿se ha incluido?
Defectos de	¿Se utilizan todas las variables de entrada?

entrada/salida	¿SE les asigna un valor a todas las variables de salida? ¿Pueden provocar corrupciones de datos las entradas no esperadas?
Defectos de interfaz	¿Las llamadas a funciones y a métodos tienen el número correcto de parámetros? ¿Concuerdan los tipos de parámetros reales y formales? ¿Están en el orden correcto los parámetros? Si los componentes acceden a memoria compartida, ¿tienen el mismo modelo de estructura de la memoria compartida?
Defectos de gestión de almacenamiento	Si una estructura enlazada se modifica, ¿se reasignan correctamente todos los enlaces? Si se utiliza almacenamiento dinámico, ¿se asigna correctamente el espacio de memoria? ¿Se desasigna explícitamente el espacio de memoria cuando ya no se necesita?
Defecto de manejo de excepciones	¿Se tienen en cuenta todas las condiciones de error posible?

2.2. Aspectos estratégicos de las pruebas

2.2.1. Directrices para una prueba exitosa

Antes de implementar una estrategia de prueba del software es aconsejable atender las siguientes directrices:

- ***Especificar los requisitos del producto de manera cuantificable mucho antes de que empiecen las pruebas.*** Aunque el objetivo primordial de la prueba es encontrar errores, una buena estrategia de prueba también evalúa otras características de la calidad, como las opciones de llevarla a otra plataforma, además de la facilidad de mantenimiento y uso. Esto debe especificarse de manera tal que permita medirlo para que los resultados de la prueba no resulten ambiguos.
- ***Establecer específicamente los objetivos de la prueba.*** Los objetivos específicos de la prueba se deben establecer en términos cuantificables. Por ejemplo, dentro del plan de prueba deben establecerse la efectividad y la cobertura de la prueba, el tiempo medio de falla, el costo de encontrar y corregir defectos, la densidad o la frecuencia de las fallas restantes, y las horas de trabajo por pruebas de regresión.
- ***Comprender cuáles son los usuarios del software y desarrollar un perfil para cada categoría de usuario.*** Los casos de uso que describan los escenarios de interacción para cada clase de usuario reducen el esfuerzo general de prueba, ya que concentran la prueba en la utilización real del producto.
- ***Desarrollar un plan de prueba que destaque la "prueba de ciclo rápido".*** Es recomendable que un equipo de ingeniería de software "aprenda a probar en ciclos rápidos (2% del esfuerzo del proyecto) los incrementos en el mejoramiento de la funcionalidad, la calidad, o ambas, de manera que sean útiles para el cliente y se pueda probar en el campo". La retroalimentación que generan estas pruebas de ciclo rápido se utiliza para controlar los grados de calidad y las correspondientes estrategias de pruebas.
- ***Construir un software "robusto" diseñado para probarse a sí mismo.*** El software debe diseñarse de manera tal que use técnicas antierror. Es decir, el software debe tener la capacidad de diagnosticar cierta clase de error. Además, el diseño debe incluir pruebas automatizadas o de regresión.
- ***Usar revisiones efectivas como filtro previo a la prueba.*** Las revisiones técnicas formales llegan a ser tan efectivas como las pruebas para descubrir errores. Por lo tanto, las revisiones reducen la cantidad de esfuerzo de prueba que se requiere para producir software de alta calidad.
- ***Realizar revisiones para evaluar la estrategia de prueba y los propios casos de prueba***. Las revisiones técnicas formales posibilitan descubrir inconsistencias, omisiones y errores

evidentes en el enfoque de la prueba. Esto ahorra tiempo y también mejora la calidad del producto.

- ***Desarrollar un enfoque de mejora continua para el proceso de prueba.*** ES necesario medir la estrategia de prueba. Las medidas reunidas durante la prueba deben usarse como parte de un enfoque estadístico de control del proceso para la prueba del software.

2.2.2. La visión de los objetos de la prueba

Al probar un componente, un grupo de componentes, un subsistema o un sistema, la propia visión del objeto de la prueba (es decir del componente, grupo de componentes, subsistema o sistema) puede afectar la forma en que se lleve a cabo la prueba.
Si el objeto de prueba se ve desde afuera, como una caja cerrada o una caja negra, cuyo contenido se desconoce, las pruebas consisten en alimentar la caja negra con entradas y anotar cuales son las salidas que se producen. En este caso la meta de la prueba es asegurar que se ha ingresado toda clase de entrada y que la salida observada en cada caso se corresponde con la salida esperada.
Por otra parte, el objeto de la prueba puede verse en cambio como una caja abierta (también denominada caja de cristal, o caja blanca, o caja transparente); luego, se puede utilizar la estructura del objeto de la prueba para probar de diferentes maneras. Por ejemplo, se pueden inventar casos de prueba que ejecuten todas las instrucciones o todos los caminos de control que existen en el interior del componente (o de los componentes), para asegurar que el objeto de la prueba está trabajando correctamente. Sin embargo, puede resultar poco práctico adoptar este tipo de enfoque.
Por ejemplo, un componente con bucles y ramificaciones numerosos puede tener demasiados caminos para comprobar. Aun con una estructura lógica bastante sencilla, un componente con iteración o recursión sustancial es difícil de probar en profundidad.
Cuando se decide como probar, no es necesario seleccionar exclusivamente pruebas de caja abierta o de caja cerrada. Se puede considerar que caja cerrada es uno de los extremos de un continuum de prueba y que caja abierta constituye el otro extremo. Una filosofía de prueba puede quedar en cualquier parte entre ambos extremos. En general, la selección de una filosofía depende de muchos factores, incluyendo:

- El número de caminos lógicos posibles
- La naturaleza de los datos de entrada
- La cantidad de cómputo involucrado
- La complejidad de los algoritmos

2.2.2.1 Prueba de caja negra

Las pruebas de caja negra o pruebas de comportamiento, se enfocan en el comportamiento de entrada/salida del componente. **No manejan los aspectos internos del componente ni el comportamiento o estructura de los componentes**. Es decir, **las pruebas de caja** negra permiten al ingeniero de software derivar conjuntos de condiciones de entrada que **ejercitarán por completo todos los requisitos funcionales de un programa**. Estas pruebas representan un enfoque complementario al de caja blanca y tratan de encontrar errores en las siguientes categorías:

1. Funciones incorrectas o faltantes.
2. Errores de interfaz.
3. Errores en estructuras de datos o en acceso a base de datos externas.
4. Errores de comportamiento o desempeño.
5. Errores de inicialización y término.

Las pruebas de caja negra centran su atención en el dominio de la información y se diseñan para responder las siguientes preguntas:

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueban el comportamiento y el desempeño del sistema?
- ¿Cuáles clases de entrada serán buenos casos de prueba?
- ¿El sistema es particularmente sensible a ciertos valores de entrada?
- ¿Cuáles tasas de datos y cuál volumen tolera el sistema?
- ¿Qué efecto tienen combinaciones específicas de datos sobre la operación del sistema?

Al aplicar técnicas de caja negra se derivan un conjunto de casos de prueba que satisfacen los siguientes criterios:

1. Casos de prueba que reducen, mediante una cuenta mayor que uno, el número de casos de prueba adicionales que deben diseñarse para alcanzar una prueba razonable.
2. Casos de prueba que indican algo acerca de la presencia o ausencia de clases de errores, en lugar de un error asociado sólo con la prueba específica a la mano.

Los métodos de pruebas de caja negra son tan apropiados para los sistemas OO como los sistemas que se desarrollan con métodos convencionales de ingeniería de software. Los casos de uso proporcionan información útil para el diseño de pruebas de caja negra.

2.2.2.2. Pruebas de caja blanca

La prueba de caja blanca, es un método de diseño que usa la estructura de control descrita como parte del diseño a nivel de componente para derivar casos de prueba. Es decir, ***se enfoca en la estructura interna de nuestro programa.***

Al emplear los métodos de caja blanca, el ingeniero de software podrá derivar casos de prueba que:

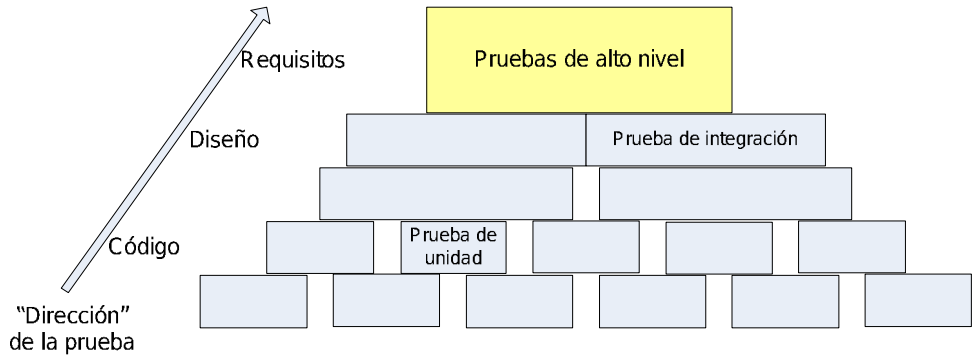
a. Garanticen que todas las rutas independientes dentro del módulo se han ejercitado por lo menos una vez.
b. Ejerciten los lados verdadero y falso de todas las decisiones lógicas.
c. Ejecuten todos los bucles en sus límites y dentro de sus límites operacionales.
d. Ejerciten estructuras de datos internos para asegurar su validez

2.3. Métodos de prueba para arquitecturas convencionales del software

Si se considera el proceso desde un punto de vista procedimental, en realidad la prueba dentro del contexto de la ingeniería del software consiste en una serie de cuatro pasos que se implementan de manera secuencial.

Al principio, la prueba se concentra en cada componente individual, asegurando que funciona de manera apropiada como unidad (por ello se le denomina **prueba de unidad**). La prueba de unidad emplea en forma recurrente las técnicas de pruebas que recorren caminos específicos en una estructura de control del componente, lo que asegura una cobertura completa y una detección máxima de errores. En seguida deben ensamblarse o integrarse los componentes para formar el paquete de software completo. La **prueba de integración** atiende todos los aspectos asociados con el doble problema de verificación y construcción del programa. Las técnicas de diseño de casos de prueba que se concentran en entradas y salidas son más dominantes durante la integración, aunque podrían usarse técnicas que recorren rutas específicas del programa para asegurar la cobertura de los principales caminos de control. Después de que se ha integrado (construido) el software se aplica un conjunto de pruebas de alto nivel. Se debe evaluar los criterios de validación establecidos durante el análisis de requisitos. La **prueba de validación** proporciona un aseguramiento final de que el software cumple con todos los requisitos funcionales, de comportamiento y desempeño.

El último paso de la prueba de alto nivel queda fuera de los límites de la ingeniería del software, pero dentro de un contexto más amplio de la ingeniería de los sistemas de computo. El software, una vez validado, debe combinarse con otros elementos (por ejemplo, hardware, personas, base de datos). La **prueba del sistema** verifica que todos los elementos encajan apropiadamente y que se logre la función y desempeño generales del sistema.



2.3.1. Prueba de Unidad

La prueba de unidad se concentra en el esfuerzo de verificación de la unidad más pequeña del diseño de software: el componente o módulo del software. Las pruebas de unidad se concentran en la lógica del procedimiento interno y en las estructuras de datos dentro de los límites de un componente.
Hay tres motivaciones tras el enfoque en los componentes: Primero, la prueba unitaria reduce la complejidad de las actividades de pruebas generales, permitiéndonos enfocarnos en unidades más pequeñas del sistema. Segundo, la prueba unitaria facilita resaltar y corregir defectos, ya que están involucrados pocos componentes. Tercero, la prueba unitaria permite el paralelismo en las actividades de prueba; esto es, cada componente puede probarse en forma independiente de los demás.
Como parte de la prueba de unidad, se prueban: la interfaz, estructuras de datos locales, condiciones límites, rutas independientes, y rutas de manejo de error.

La **interfaz del módulo** se prueba para asegurar que la información fluye apropiadamente dentro y fuera de la unidad de programa sujeta a prueba. Si los datos no entran ni salen apropiadamente, todas las demás pruebas resultarán inútiles.

Se examinan las **estructuras de datos locales** para asegurar que los datos temporales mantienen la integridad durante todos los pasos de la ejecución de un algoritmo. Y, debe evaluarse (si es posible) el impacto local sobre los datos globales.

Se recorren todos los **caminos** independientes (caminos de base) en toda la estructura para asegurar que todas las instrucciones de un módulo se hayan ejecutado por lo menos una vez. Se deben diseñar casos de prueba para descubrir errores debido a cálculos incorrectos, comparaciones erróneas o flujos de control inapropiados. Entre los errores más comunes en los cálculos se encuentran los siguientes:

1. Aplicación incorrecta o mal entendida de la precedencia aritmética.
2. Operaciones de modo mezcladas.
3. Inicialización incorrecta.
4. Falta de precisión.
5. Representación simbólica incorrecta de una expresión.

La comparación y el flujo de control están estrechamente acoplados entre sí (es decir, el flujo cambia con frecuencia después de una comparación). Los casos de prueba deben descubrir errores como:

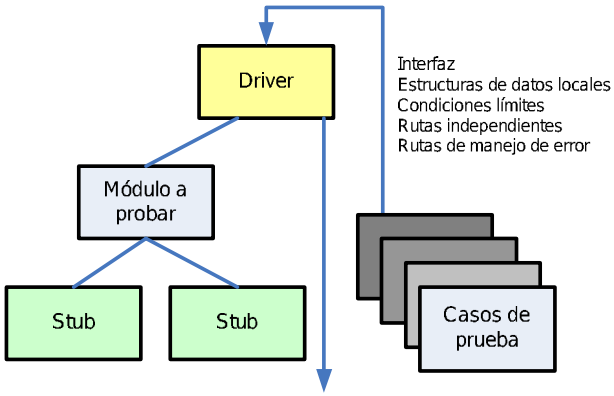
1. Comparaciones entre diferentes tipos de datos.
2. Operadores lógicos o precedencia de estos aplicados incorrectamente.
3. Expectativa de igualdad cuando los errores de precisión hacen que sea probable.
4. Comparación incorrecta de variables.
5. Terminación inapropiada o inexistente de bucles.
6. Falla en la salida cuando se encuentra una iteración divergente.
7. Variables de bucles modificadas de manera inapropiada.

Se prueban las **condiciones límites** para asegurar que el módulo opera apropiadamente en los límites establecidos para restringir el procesamiento. Con frecuencia el software falla en sus límites. Es decir, a

menudo los errores ocurren cuando se procesa el enésimo elemento de una matriz de n dimensiones, cuando se evoca la i-ésima repetición de un bucle con i pasos, cuando se encuentra el valor máximo o mínimo permisible.

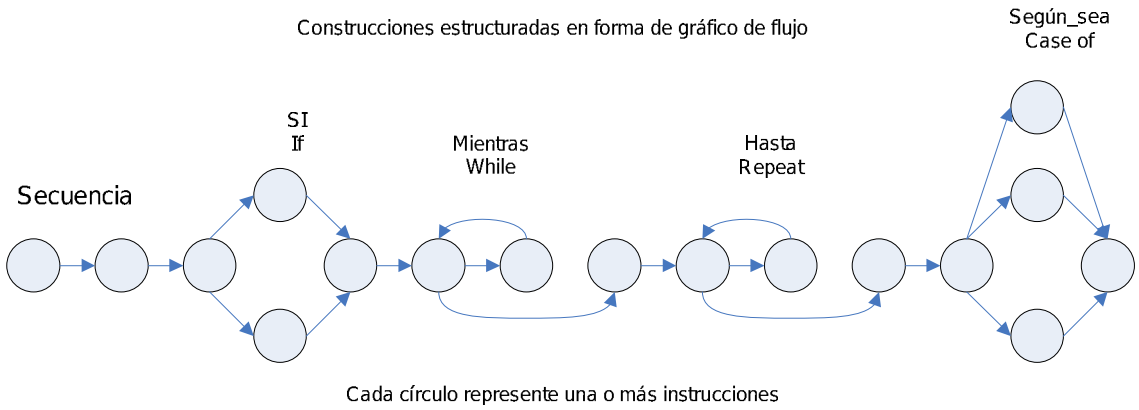
Y, por último, se prueban todos los **caminos de manejo de errores**. Un buen diseño impone que se prevean las condiciones de errores y que se configuren rutas de manejo de errores para modificar la ruta o terminar limpiamente el procesamiento cuando ocurra un error. Este enfoque se llama antierror. Entre los posibles errores que deben probarse cuando se evalúe el manejo de errores se encuentran los siguientes:
1. La descripción del error es ininteligible.
2. El error indicado no corresponde al encontrado.
3. La condición de error causa la intervención del sistema operativo antes de que se dispare el manejo de errores.
4. El procesamiento de la condición de excepción es incorrecta.
5. La descripción de error no proporciona información suficiente para ayudar a localizar la causa del error.

Debido a que un componente no es un programa independiente, para cada prueba de unidad puede ser necesario desarrollar software mock o un stub, o ambos. El mock simula la parte del sistema que llama al componente a aprobar. Un mock pasa al componente las entradas de prueba identificadas en el análisis del caso de prueba y muestra los resultados.
El stub de prueba simula a los componentes que son llamados por el componente a probar (módulos subordinados). El stub usa la interfaz del módulo subordinado, realiza una mínima manipulación de datos, proporciona verificación de la entrada y devuelve el control al componente de prueba.
Mocks y stub representan una sobre carga de trabajo. Es decir, resulta necesario escribir ambos tipos de software (sin que suela aplicarse un diseño formal), pero no se entregan con el software final. Si se les mantiene en un nivel simple, la sobrecarga real es relativamente pequeña. Por desgracia, no es posible aplicar adecuadamente una prueba de unidad a muchos componentes con un "simple" software de sobrecarga. En muchos casos es posible posponer la prueba completa hasta el paso de prueba de integración (donde también se utilizan mocks o stubs).

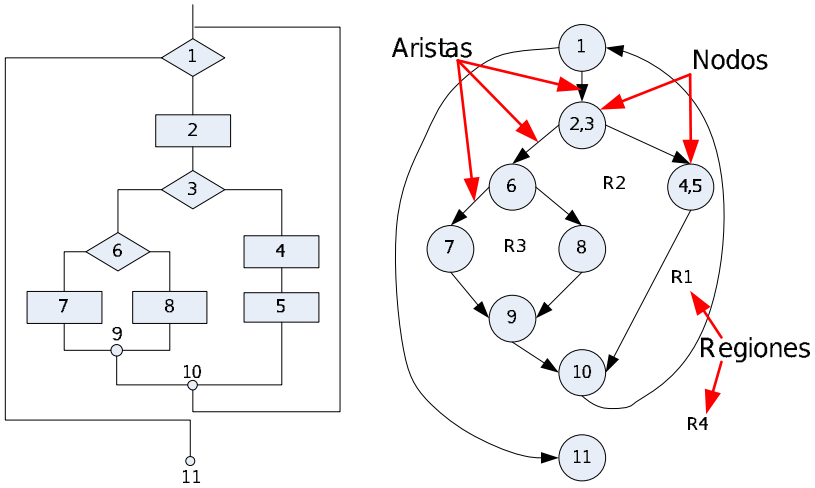


2.3.1.1 Prueba de ruta básica

Es una prueba de caja abierta, también conocida como prueba de ruta o de camino, y supone que ejercitando todas las rutas posibles del código, al menos una vez, la mayoría de los defectos producirá fallas. La identificación de las rutas requiere conocimiento del código fuente y las estructuras de datos.
El **punto inicial** de la prueba es la construcción de una **gráfica de flujo**, la que describe un flujo de control lógico empleando la notación que a continuación se muestra:



Cada construcción estructurada tiene su símbolo correspondiente en la gráfica de flujo. El uso de una gráfica de flujo se dibuja considerando la representación del diseño procedimental. Observe la siguiente figura, donde se correlaciona el diagrama de flujo con la gráfica de flujo:



Los círculos, llamados nodos de gráfica, representan bloques ejecutables (una o más instrucciones procedimentales). Cada nodo que contiene una condición es un nodo predicado y se caracteriza porque de él emanan dos o más aristas.
Las flechas en la gráfica de flujo, llamadas aristas o enlaces, representan flujos de control y son análogos a las flechas de los diagramas de flujo. Una arista debe terminar en un nodo, aunque el nodo no represente ninguna instrucción procedimental. Las áreas que limitan aristas y nodos se denominan regiones. Cuando se cuentan las regiones se incluyen las áreas ubicadas fuera de la gráfica.
La prueba de ruta consiste en el diseño de casos de prueba de forma tal que cada transición de la gráfica de flujo se recorra al menos una vez. Esto se logra examinando la condición asociada con cada punto de ramificación y seleccionando una entrada para la rama verdadero y otra para la rama falso.
El **segundo paso** es **determinar el número de pruebas necesarias** para probar todos los caminos. ¿Cómo se sabe cuántas rutas buscar? Usando la teoría de grafos puede mostrarse que la cantidad mínima de pruebas necesarias para cubrir todas las aristas es igual a la cantidad de rutas independientes que hay a lo largo de la gráfica. Esto se define como la ***complejidad ciclomática CC*** de la gráfica de flujo.
La complejidad ciclomática es una métrica que resulta útil a la hora de predecir cuáles módulos tienen más probabilidad de contener errores. SE emplea para la planeación de pruebas además del diseño de casos de prueba. La CC proporciona el límite superior del número necesario de casos de prueba para garantizar que cada instrucción del programa se haya ejecutado por lo menos una vez.
La complejidad ciclomática se puede calcular utilizando una de las siguientes formas:

- CC = número de regiones de la gráfica. CC = 4 regiones

- CC = E – N +2, donde E es el número de aristas, y N, es el número de nodos de la gráfica de flujo. CC = 11 aristas – 9 nodos + 2 = 4
- CC = P + 1, donde P es el número de nodos predicados incluidos en la gráfica. CC = 3+1 = 4.

El **tercer paso** es calcular **un conjunto básico de rutas linealmente independientes**. Considerando el ejemplo anterior encontramos las siguientes rutas independientes:
Ruta1: 1,11
Ruta2: 1, 2, 3, 4, 5, 10,1, 11.
Ruta3: 1, 2, 3, 6, 8, 9, 10, 1, 11.
Ruta4: 1, 2, 3, 6, 7, 9, 10, 1, 11.
Tenga en cuenta que cada nuevo camino ingresa una nueva arista. El camino 1, 2, 3, 4, 5, 10, 1, 2, 3, 6, 8, 9, 10, 1, 11 no se considera una ruta independiente porque se trata simplemente de una combinación de rutas ya especificadas y no recorre ninguna arista nueva.
El **cuarto paso** es **preparar los casos de prueba** que forzarán la ejecución de ruta en el conjunto básico. Es necesario seleccionar los datos de manera tal que se establezcan apropiadamente las condiciones de los nodos predicados, a medida que se prueba cada ruta. Cada caso de prueba se ejecuta y compara con los resultados esperados. Una vez completados todos los casos, la persona que aplica las pruebas puede estar segura de que todas las instrucciones del programa se han ejecutado por lo menos una vez.
La prueba de ruta básica fue desarrollada para los lenguajes imperativos. Los lenguajes orientados a objetos introducen varias dificultades cuando se usa la prueba de ruta:

- ***Polimorfismo***: El polimorfismo permite que los mensajes se unan a diferentes métodos basados en la clase del destino. Aunque esto permite que los desarrolladores reutilicen código a lo largo de una mayor cantidad de clases, también introduce más casos a probar. Se debe identificar y probar todas las uniones posibles.
- ***Métodos más cortos***. Los métodos en los lenguajes OO tienden a ser más cortos que los procedimientos y funciones de los lenguajes imperativos. Esto disminuye la probabilidad de defectos de flujos de control, que pueden descubrirse usando la técnica de prueba de ruta. En vez de ello, los algoritmos se implementan a lo largo de varios métodos, los cuales están distribuidos a lo largo de una mayor cantidad de objetos y necesitan probarse juntos.

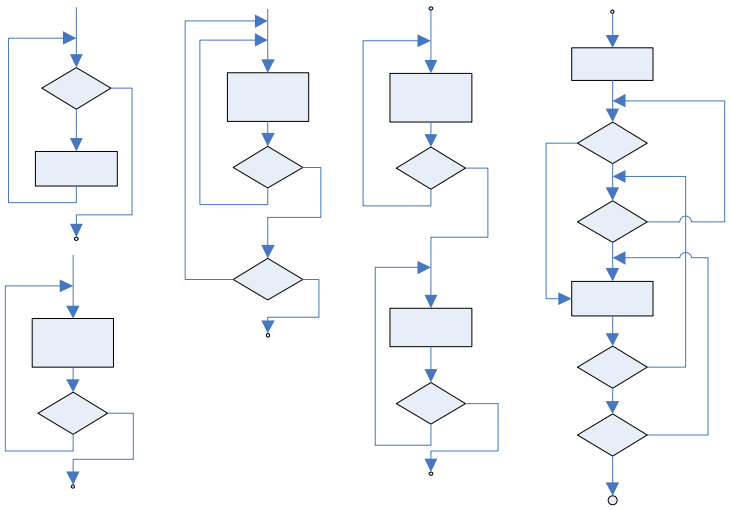
En general, la prueba de ruta y los métodos de caja blanca sólo pueden detectar defectos que se encuentran al ejercitar una ruta de programa. Los métodos de prueba de caja blanca no pueden detectar omisiones. La prueba de ruta también está fuertemente basada en la estructura de control del programa: los defectos asociados con la violación de invariantes de estructuras de datos, como el acceso a un arreglo más allá de sus límites, no se tratan de manera explícita. Sin embargo, ningún método de prueba que no sea la prueba exhaustiva puede garantizar el descubrimiento de todos los defectos.

2.3.1.2. Prueba de condición

También es una prueba de caja abierta y ejercita las condiciones lógicas contenidas en un módulo del programa. Una condición simple es una variable booleana o una expresión relacional, tal vez precedida con un operador NOT. Una expresión relacional toma la forma:
E_1 < operador relacional < E_2
Donde E_1 y E_2 son expresiones aritméticas y <operador relacional> es uno de los siguientes: <, ≤, =, ≠ (distinto), > o ≥. Una condición compuesta la integran dos o más condiciones simples, operadores booleanos y paréntesis. Se supone que entre los operadores booleanos permitidos en una condición compuesta se incluyen OR, AND y NOT. Una condición sin expresiones relacionales se considera una expresión booleanas. Por lo tanto, todos los posibles tipos de elementos en una condición incluyen un operador booleano, una variable booleana, un par de paréntesis (que encierran una condición booleana simple o compuesta), un operador relacional o una expresión aritmética.
Si una condición es incorrecta, entonces por lo menos un componente de la condición es incorrecto. Por lo tanto, entre los tipos de errores en una condición se incluyen los presentes en el operador booleano (operadores booleanos incorrectos / faltantes / adicionales), en la variable booleana, en los paréntesis booleanos, en los operadores relacionales y en la expresión aritmética.
El método de prueba de condición se concentra en la prueba de cada condición del programa para asegurar que no contiene errores.

2.3.1.3. Prueba de bucle

Es una técnica que se concentra exclusivamente en la validez de la construcción de bucles. Es posible definir cuatro diferentes clases de bucles: simple, anidados, concatenados y no estructurados:



Bucles simples. El siguiente conjunto de pruebas se aplica a bucles, donde n es el número máximo de pasos que permite el bucle.

1. Omitir por completo el bucle.
2. Sólo un paso por el bucle.
3. Dos pasos por el bucle.
4. m pasos por el bucle, donde m < n.
5. n = 1, n, n+1. pasos por el bucle.

Bucles anidados. Si fuese a extender el enfoque de prueba de los bucles simples a los anidados, el número de pruebas posibles crecería vertiginosamente a medida que aumente el nivel de anidamiento. Esto generaría un número poco práctico de pruebas. Por ello se sugiere:

1. Iniciar en el bucle más interno. Asignar a todos los bucles los valores mínimos.
2. Aplicar pruebas de bucle simple al más interno mientras se mantienen los externos en los valores mínimos del parámetro de iteración (como el contador de bucles). Agregar otras pruebas para los valores fuera de rango o excluidos.
3. Trabajar hacia fuera, conduciendo pruebas para el siguiente bucle, pero manteniendo todos los demás bucles externos en valores mínimos y otros bucles anidados en valores “típicos”.
4. Seguir mientras no se hayan probado todos los bucles.

Bucles concatenados. Los bucles concatenados se prueban empleando el enfoque definido para los bucles simples, si cada uno de los bucles es independiente. Sin embargo, si dos bucles están concatenados y el contador del bucle 1 se emplea como valor inicial para el bucle 2, entonces los bucles no son independientes. Cuando los bucles no lo son, entonces se recomienda el enfoque aplicado a los bucles anidados.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles debe diseñarse nuevamente para reflejar el uso de las construcciones de programación estructurada.

2.3.1.4. Partición equivalente

Es una técnica de caja negra y también se la llama como prueba de dominios o de equivalencia. Esta prueba minimiza la cantidad de casos de prueba. Las entradas posibles se reparten en clases de equivalencia y se selecciona un caso de prueba para cada clase. La suposición de la prueba de equivalencia es que los sistemas se comportan, por lo general, en forma similar para todos los miembros de una clase. Para probar el comportamiento asociado con una clase de equivalencia sólo necesitamos probar a un miembro de la clase. La prueba de equivalencia consiste en ***dos pasos: identificación de las clases de equivalencia y selección de las entradas de prueba***. SE usan los siguientes criterios para determinar las clases de equivalencia.

- *Cobertura*. Cada entrada posible pertenece a una de las clases de equivalencia.
- *Interconexión*. Ninguna entrada pertenece a más de una clase de equivalencia.
- *Representación*. Si la ejecución muestra un error cuando se usa como entrada un miembro particular de una clase de equivalencia, el mismo puede detectarse usando como entrada a cualquier otro miembro de la clase.

*Para cada clase de equivalencia se seleccionan, al menos, dos partes de datos: una **entrada típica**, que ejercita el caso común, y una **entrada inválida** que ejercita las capacidades del componente para el manejo de excepciones*. Después de que se han identificado todas las clases de equivalencia tiene que identificarse una entrada de prueba para cada clase que cubra la clase de equivalencia. Si hay una posibilidad de que no todos los elementos de la clase de equivalencia estén cubiertos por la entrada de prueba, la clase de equivalencia debe dividirse en clases de equivalencia más pequeñas y deben identificarse entradas de prueba para cada una de las nuevas clases.

2.3.1.5 Análisis de valores límite

Esta prueba, también llamada prueba de frontera, es un caso especial de la prueba de equivalencia y se enfoca en las condiciones de frontera de las clases de equivalencia. En vez de seleccionar cualquier elemento de la clase de equivalencia, la prueba de frontera requiere que los elementos se seleccionen de los "extremos" de la clase de equivalencia. La suposición que hay tras la prueba es que los desarrolladores a menudo olvidan los casos especiales en la frontera de las clases de equivalencia (por ejemplo, 0, cadenas vacías, año 2000).

Una desventaja de las pruebas de la clase de equivalencia y de frontera es que estas técnicas no exploran combinaciones de datos de entrada de prueba. En muchos casos, un programa falla debido a que una combinación de determinados valores causa el error.

2.3.2. Pruebas de Integración

Una vez que se han eliminado los defectos de cada componente y los casos de prueba no revelan ningún defecto nuevo, los componentes están listos para su integración en subsistemas más grandes. El objetivo de las pruebas de integración es descubrir errores asociados con la interfaz entre componentes.
Las pruebas de integración detectan defectos que no se han descubierto durante las pruebas unitarias enfocándose en pequeños grupos de componentes. Dos o más componentes se integran y prueban, y después de que las pruebas ya no detectan ningún nuevo defecto se añaden componentes adicionales al grupo. Este procedimiento permite la prueba de partes cada vez más complejas del sistema, manteniendo relativamente pequeña la ubicación de los defectos potenciales.

2.3.2.1. Prueba ascendente

Las **prueba de abajo hacia arriba o ascendente** prueba primero de manera individual a cada componente de la capa inferior y luego los integra con componentes de la siguiente capa superior. Si dos componentes se prueban juntos a esto le llamamos prueba doble. A la prueba de tres componentes juntos le llamamos prueba triple y a la de cuatro, prueba cuádruple. Esto se repite hasta que se combinan todos los componentes de todas las capas. Se usan mockss de pruebas para simular los componentes de las capas superiores que todavía no se han integrado. Observe que no se necesitan stubs de prueba durante el proceso.

Una prueba ascendente se implementa mediante los siguientes pasos:

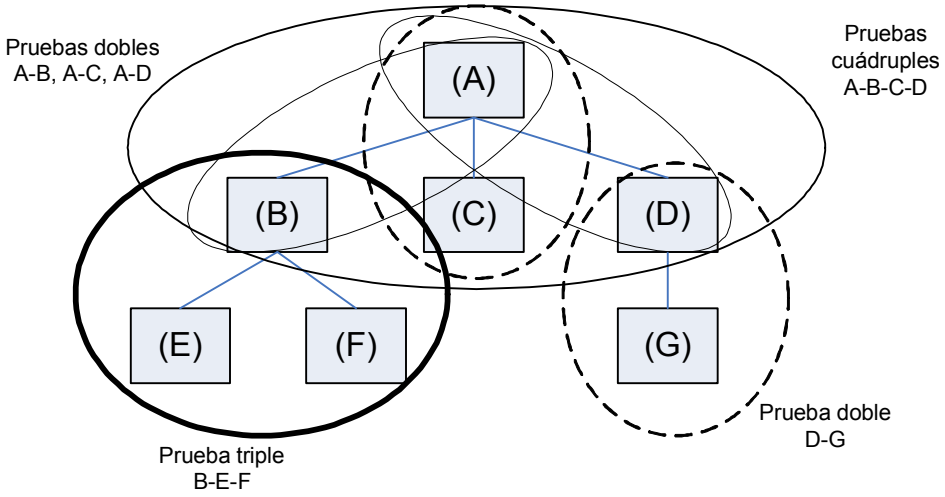
1. Se combinan los módulos de bajo nivel en grupos (también llamados construcciones) que realicen una subfunción específica del software.
2. SE escribe un mock (un programa de control para pruebas) con el fin de coordinar la entrada y salida de los casos de prueba.
3. SE prueba el grupo.
4. SE eliminan los mocks y se combinan los grupos ascendiendo por la estructura del programa.

2.3.2.2. Prueba descendente

La estrategia de **pruebas de arriba hacia abajo o descendente** prueba primero en forma unitaria los componentes de la capa superior y luego integra los componentes de la siguiente capa hacia abajo. Cuando se han probado juntos todos los componentes de la nueva capa se selecciona la siguiente capa. De nuevo, las pruebas añaden de manera incremental un componente tras otro. Esto se repite hasta que se han combinado o involucrado todas las capas en la prueba. Los stubs de pruebas se usan para simular a los componentes de las capas inferiores que todavía no se han integrado. Observe que no se necesitan mocks de prueba durante la prueba descendente.
El proceso de integración se realiza en una serie de cinco pasos:

1. El módulo de control principal se utiliza como mock de prueba, y se sustituyen los stubs en todos los componentes directamente subordinados al módulo de control principal.
2. SE van reemplazando los stubs subordinados, uno por uno, con los componentes reales.
3. SE aplican pruebas cada vez que se integran un nuevo módulo.
4. Al completar cada conjunto de pruebas, se reemplaza otro stub con el módulo real.
5. Se aplica la prueba de regresión para asegurarse de que no se han introducido nuevos errores.

La ventaja de las pruebas ascendente es que pueden localizarse con más facilidad los defectos de interfaz: cuando los desarrolladores sustituyen un mock de prueba con un componente de nivel superior tienen un modelo claro de la manera en que funciona el componente de nivel inferior y las suposiciones incrustadas en su interfaz. Si el componente de nivel superior viola las suposiciones del nivel inferior, es más probable que los desarrolladores lo encuentren con rapidez. La desventaja de las pruebas ascendentes es que prueba al último los subsistemas más importantes, a saber, los componentes de la interfaz de usuario. Primero, los defectos que se encuentran en la capa superior a menudo pueden conducir a cambios en la descomposición en subsistemas o en las interfaces de subsistemas de las capas inferiores, invalidando las pruebas anteriores. Segundo, las pruebas de la capa superior pueden derivarse del modelo de requerimientos y, por lo tanto, son más efectivas para la localización de defectos que son visibles ante el usuario.
La ventaja de la prueba descendente es que comienzan con los componentes de la interfaz de usuario. El mismo conjunto de pruebas derivadas de los requerimientos puede utilizarse en las pruebas de conjunto de subsistemas cada vez más complejos. La desventaja de estas pruebas es que el desarrollo de stubs de prueba es consumidor de tiempo y propenso a errores. Por lo general, se requiere una gran cantidad de stubs para probar sistemas no triviales, en especial cuando el nivel más bajo de la descomposición del sistema implementa muchos métodos.
La siguiente figura ilustra una combinación posible de subsistemas que puede usarse durante las pruebas de integración. Usando una estrategia ascendente se prueban primero de manera unitaria los subsistemas E, F y G, luego se ejecuta la prueba triple B- E- F y la prueba doble D-G, y así en forma sucesiva. Usando una estrategia descendente se prueba en forma unitaria el subsistema A, luego se ejecutan las pruebas dobles A-B, A-C y A-D, luego la prueba cuádruple A-B-C-D y así en forma sucesiva. Ambas estrategias cubren la misma cantidad de dependencias de subsistemas pero las ejercitan en orden diferente.



2.3.2.3. Prueba de regresión

Si las pruebas de cualquier tipo tienen éxito, el resultado es el descubrimiento de errores, y estos deben corregirse. Cada vez que el software se corrige también cambia algún aspecto de la configuración del software (el programa, su documentación o los datos de soporte). La prueba de regresión es una actividad que ayuda a asegurar que los cambios (debido a la prueba u otras razones) no introduzcan comportamiento indeseable o errores adicionales.
El conjunto de pruebas de regresión contienen tres clases de casos de prueba:

- Una muestra representativa de pruebas que ejercitarán todas las funciones del software.
- Pruebas adicionales que se concentren en las funciones del software que probablemente el cambio afectaría.
- Pruebas que se concentran en los componentes del software que han cambiado.

A medida que avanza la prueba de integración, la cantidad de pruebas de regresión llega a volverse muy grande. Por lo tanto, el conjunto de pruebas de regresión debe diseñarse para que sólo incluya las que atienden una o más clases de errores en cada una de las funciones principales del programa. Resulta poco práctico e ineficiente repetir cada prueba para todas las funciones del programa después de que se ha presentado un cambio.

2.3.2.4. Prueba de humo

Es un procedimiento de prueba diseñado para cubrir una amplia porción de la funcionalidad del sistema. Las pruebas así diseñadas dan prioridad al porcentaje del total del sistema que se pone a prueba, sacrificando de ser necesario la finura de estas. Las pruebas de humo son útiles a la hora de determinar si un sistema va cumpliendo lo requerido, así como para verificar una vez ya en producción, que luego de una instalación nueva o de la recuperación de una falla catastrófica el sistema se ha devuelto a su pleno funcionamiento.
La prueba de humo es un enfoque de prueba de integración que suele utilizarse mientras se desarrollan productos de software. Está diseñado como mecanismo para marcar el ritmo en proyectos en los cuales el

tiempo es crítico, lo que permite que el equipo de software evalúe su proyecto con frecuencia. Abarca las siguientes actividades:

1. Los componentes de software traducidos a código se integran en una "construcción", la que incluye archivos de datos, librerías, los módulos reutilizables y los componentes de ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para exponer errores que impedirán que la construcción realice apropiadamente su función. El objetivo es descubrir errores "paralizantes" que tengan la mayor probabilidad de retrasar el proyecto de software.
3. La construcción se integra con otras construcciones y diariamente se aplica una prueba de humo a todo el producto (en su forma actual). El enfoque de la integración puede ser descendente o ascendente. La aplicación diaria de una prueba de humo da a los jefes de proyecto o participantes una evaluación realista del avance de las pruebas de integración.

La prueba de humo debe ejercitar todo el sistema de principio a fin. No debe ser exhaustiva, pero debe tener la capacidad de exponer problemas importantes. La prueba de humo debe ser tan completa que si la construcción la aprueba, puede suponerse que ésta es lo suficientemente estable como para probarla de manera más completa.

La prueba de humo proporciona varios beneficios cuando se aplica en proyectos de ingeniería de software complejos y que dependen en forma crítica del tiempo:

- *Se minimiza el riesgo de la integración:* Por ser aplicadas diariamente se descubren errores paralizantes, por lo que reduce la probabilidad de impacto en el calendario del proyecto.
- *Se mejora la calidad del producto final:* Como el enfoque es de integración, es probable que estas pruebas descubran errores funcionales, arquitectónicos y al nivel de componentes, esto mejorara la calidad.
- *Se simplifican el diagnóstico y la corrección de errores:* Es probable que los errores no descubiertos en la prueba de humo estén asociados con "nuevos componentes de software".
- *El progreso es más fácil de evaluar:* Cada día que pasa se integra más software y se demuestra que funciona.

El término "prueba de humo" se originó en la industria del hardware. El término se deriva de esta práctica: después de modificar o reparar un producto o un componente de hardware, el equipo se encendía sin más. Si no salía humo, el componente pasaba la prueba.

Pongamos como ejemplo un sistema de un banco. Es algo grande y con muchas partes. Supongamos que ya corrimos todas las pruebas por separado, regresiones, y de integración. Ahora nos queda dar la decisión oficial de si testing aprueba la salida del sistema. Para asegurarnos que nada se rompió corremos la prueba de humo. La prueba para este ejemplo tendrá pruebas concretas, por ejemplo si analizamos la parte del sistema que maneja las cuentas las pruebas serían muy concretas: crear cuenta corriente en dólares, crear cuenta corriente en pesos, crear caja de ahorro en dólares, crear caja de ahorro en pesos, modificar cada una y borrar cada una. Siempre con valores válidos y casos limpios. De esta forma hacemos un checkeo superficial antes de tomar una decisión.

Módulo Crítico

Hemos analizado hasta aquí diferentes estrategias para probar de manera integral nuestro software. Ahora tengamos en cuenta que a medida que se realiza la prueba de integración, el responsable debe identificar módulos críticos. Ya que los módulos críticos deben probarse lo antes posible y además, las pruebas de regresión se deben concentrar en el funcionamiento de estos módulos.

Pero, ¿qué es un módulo crítico? ES un módulo que tiene una o más de las siguientes características:

1. Atiende varios requisitos del software.
2. Tiene un alto grado de control (se encuentra en un lugar relativamente alto de la estructura del programa).
3. Es complejo o propenso a errores, o.
4. Tiene requisitos de desempeño bien definidos.

2.4. Métodos de pruebas orientados a Objetos

La arquitectura del software orientado a objetos genera una serie de subsistemas separados en capas que encapsulan las clases que colaboran entre si. Cada uno de estos elementos del sistema (subsistemas y clases) realiza funciones que ayudan a cumplir con los requisitos del sistema. Es necesario probar un sistema orientado a objetos en diferentes niveles para descubrir errores que podrían ocurrir a medida que las clases colaboran entre sí y los subsistemas se comunican entre las capas de la arquitectura.
En el aspecto estratégico, la prueba orientada a objetos es similar a la de los sistemas convencionales. Pero es diferente en el aspecto táctico. Debido a que los modelos de análisis y diseño orientados a objetos tienen una estructura y un contenido similares al programa orientado a objetos resultante, la "prueba" podría empezar con la revisión de estos modelos Una vez que se ha generado el código, la prueba orientada a objetos real empieza por lo "pequeño", con una serie de pruebas diseñadas para ejercitar las operaciones de clase y examinar si existen errores a medida que una clase colabora con otra. Cuando las clases se integran para formar un subsistema, se aplica la prueba basada en el uso (se definen escenarios), junto con los enfoques basados en faltas, para ejercitar plenamente las clases que colaboran entre si. Por ultimo, se emplean los casos de uso para descubrir errores al nivel de validación del software.
El diseño convencional de casos de prueba lo determina el concepto entrada-proceso-salida del software o el detalle algorítmico de módulos individuales. La prueba orientada a objetos se concentra en el diseño de secuencias apropiadas de operación para ejercitar los estados de una clase.

2.4.1. Prueba de Unidad en el contexto OO

Al pensar en el software OO cambia el concepto de unidad. La encapsulación orienta la definición de clases. Esto significa que cada clase e instancia de una clase (objeto) empaqueta atributos (datos) y las operaciones (funciones) que manipulan estos datos. Una clase encapsulada suele ser el eje de las pruebas de unidad. Sin embargo, las unidades de prueba más pequeñas son las operaciones dentro de las clases. Debido a que una clase puede contener varias operaciones diferentes y a que una operación determinada puede existir como parte de varias clases diferentes, deben cambiar las tácticas aplicadas para la prueba de unidad. No es posible probar una sola operación de manera aislada (concepto convencional de la prueba de unidad) sino como parte de una clase. Por ejemplo, considérese una jerarquía de clase en que se define una operación X para la superclase y que heredan varias subclases. Cada una de estas usa la operación X, pero se aplica dentro del contexto de los atributos privados y las operaciones que se han definido para subclase. Dado que el contexto en que se emplea la operación X varía en formas sutiles, es necesario probar la operación en el contexto de cada una de las subclases. Esto significa que si se prueba la operación X de manera independiente (enfoque convencional) se podrá usar de manera deficiente en el contexto orientado a objeto.
La prueba de clases para el software OO es el equivalente a la prueba de unidad para el software convencional. A diferencia de ésta, que tiende a concentrarse en el detalle del algoritmo de un módulo y en los datos que fluyen por la interfaz del módulo, la prueba de clase para el software OO se orienta mediante las operaciones que encapsula la clase y en el comportamiento de estado de la clase (representado por los valores de sus atributos), para determinar si existen errores.

Los métodos de prueba de caja blanca descritos en secciones anteriores pueden aplicarse a las operaciones definidas para una clase. Las técnicas de prueba de la ruta básica o de bucle ayudan a asegurar que se han probado todas las instrucciones de una operación Sin embargo, la estructura concisa de muchas operaciones de clase hace que algunos argumenten que el esfuerzo aplicado a la prueba de caja blanca podría redirigirse mejor para probar un nivel de clase.
Los métodos de prueba de caja negra son tan apropiados para los sistemas orientados a objetos como los sistemas que se desarrollan con métodos convencionales de ingeniería de software. Como ya se indicó, los casos de uso proporcionan información útil para el diseño de pruebas de caja negra y basadas en el estado.

2.4.2. Prueba de Integración en el contexto OO

Debido a que el software OO no tiene una estrategia obvia de control jerárquico, tienen poco significado estrategias descendente y ascendente tradicionales. Además, integrar las operaciones una por una en una clase (el enfoque convencional de la integración incremental) suele resultar imposible debido a las "interacciones directas e indirectas de los componentes que integran la clase".
Hay dos estrategias diferentes para la prueba de integración de los sistemas OO.
La primera, la prueba basada en subprocesos, integra el conjunto de clases requerido para responder a una entrada o un evento del sistema. Cada subproceso se integra y se prueba individualmente. La prueba de regresión se aplica para asegurar que no se presentan efectos colaterales.
El segundo enfoque de integración, la prueba basada en el uso, empieza la construcción del sistema con la prueba de estas clases (llamadas clases independientes) que usan muy pocas clases de servidor (o ninguna). Después de que se prueban las clases independientes, se prueba la siguiente capa de clases, llamadas clases dependientes, que usan las clases independientes. Esta secuencia de capas de prueba de clases dependientes continúa hasta que se construye todo el sistema.
El uso de mocks y stubs también cambia cuando se aplican pruebas de integración en Sistemas OO. Con los mocks se prueban operaciones al nivel más bajo y grupos completos de clases. Un mock también se utiliza para reemplazar la interfaz de usuario, de modo que puedan aplicarse las pruebas funcionales del sistema antes de la implementación de la interfaz. Los stubs se usan en situaciones en que la colaboración entre clases es necesaria, pero en las cuales aún no se han implementado por completo una o más de las clases que colaboran.
La prueba de grupo es un paso en la prueba de integración del software OO. Aquí, un grupo de clases que colaboran entre sí se ejercitan al diseñar casos de prueba que tratan de descubrir errores en las colaboraciones.

2.4.3. Casos de prueba y jerarquía de clase

La herencia no obvia la necesidad de una prueba completa de todas las clases derivadas. En realidad, llega a complicar el proceso de prueba. Por ejemplo. Una clase **Base** contiene las operaciones **heredado()** y **redefinido()**. Una clase **Derivado** define **redefinido()** para que sirva en un contexto local. Entonces:
¿Debe probarse de **Derivado::heredado()**?
¿Es necesario derivar nuevas pruebas exclusivas para **Derivado::redefinido()**?

Sobre el primer interrogante diremos que: Si **Derivado::heredado()** llama a **Derivado::redefinido** y dado que este último ha cambiado su comportamiento, es posible que el primero maneje erróneamente el nuevo comportamiento. Por lo tanto, se necesitan nuevas pruebas aunque no hayan cambiado el diseño ni el código. Por otra parte, si **Derivado::heredado()** no llama a **redefinido()** ni a ningún código que lo llame indirectamente, no es necesario probar de nuevo ese código en la clase derivada.
En cuanto a la segunda pregunta, **Base::redefinido()** y **Derivado::redefinido** son operaciones distintas con diferentes especificaciones e implementaciones. Pero, es posible que sus operaciones sean similares, por lo tanto habrá un conjunto de requisitos de pruebas que se aplicarán a ambas clases. Sin embargo, será necesario derivar nuevas pruebas exclusivamente para los requisitos de **Derivado::redefinido()** que no se satisfagan con las pruebas de **Base::redefinido().**

2.4.5. Métodos de Prueba a nivel de clase

Anteriormente hablamos que la prueba del software empieza por lo "pequeño" y lentamente avanza hacia lo "grande". Se prueba en el pequeño entorno de una sola clase y los métodos que están encapsulados en la clase. La prueba aleatoria y la partición son métodos que se emplean para ejercitar una clase durante una prueba OO.

2.4.5.1. Prueba aleatoria para clases orientadas a objeto

Es una prueba que se emplea para ejercitar una sola clase y consiste en generar al azar secuencias de operaciones diferentes de la clase. Por ejemplo, imagínese una aplicación bancaria en una clase **Cuenta** tiene las siguientes operaciones: *abrir(), configurar(),depositar(), retirar(), saldar(), sumar(),*

límitecrédito() y cerrar(). Cada una de estas operaciones se aplica a **Cuenta**, pero hay ciertas restricciones (por ejemplo, la cuenta debe abrirse antes de aplicar otras operaciones, y debe cerrarse después de que se han completado todas las operaciones) relacionadas con la naturaleza del problema. Aun con estas restricciones, hay muchas permutas en las operaciones. El historial de comportamiento mínimo de una instancia de **Cuenta** incluye las siguientes operaciones:
Abrir-configurar-depositar-retirar-cerrar
Esto representa la secuencia de prueba mínima para **Cuenta**. Sin embargo, podría presentarse una amplia variedad de comportamientos distintos dentro de esta secuencia.
Abrir-configurar-depositar-[depositar| retirar| saldar| sumar| límitecrédito]- retirar-cerrar

2.4.5.2. Pruebas de partición a nivel de clase

La prueba de partición reduce el número de casos de prueba requeridos para ejercitar una clase de manera muy parecida a la partición equivalente para el software convencional. La entrada y la salida se ordenan en categorías y se diseñan casos de prueba para ejercitar cada categoría ¿Cómo se derivan las categorías de partición?

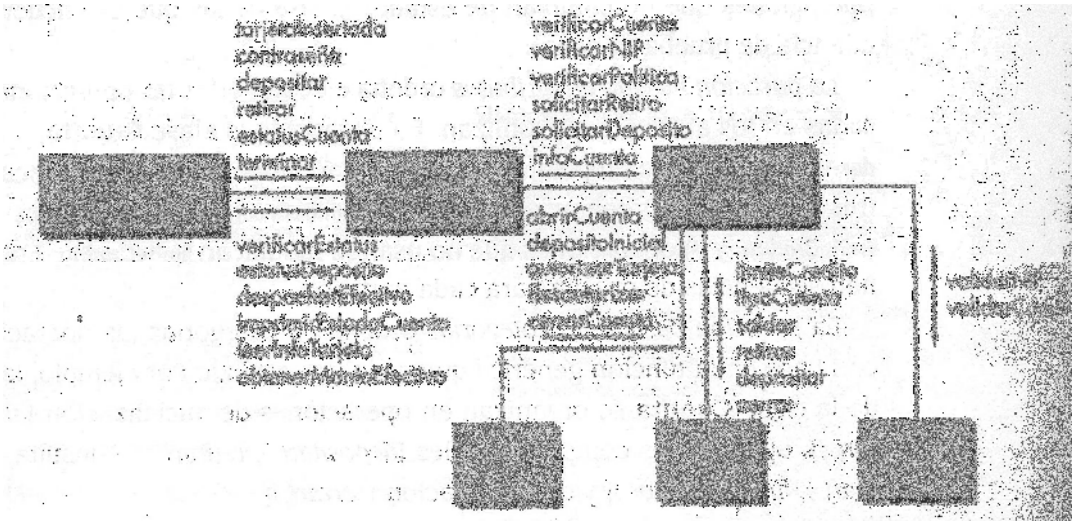
La **partición basada en el estado** ordena en categorías las operaciones de clase a partir de su capacidad para cambiar el estado de la clase. Si se piensa una vez más en la clase **Cuenta**, las operaciones de estado incluyen *depositar()* y *retirar()*, mientras que las que no son de estado incluyen *saldar()*, *sumar()*, y *limitecredito()*. Las pruebas están diseñadas de manera que ejercitan por separado las operaciones que cambian de estado y las que no lo hacen. Por ejemplo:
Caso de prueba 1: abrir-configurar-depositar-depositar-retirar-retirar-cerrar.
Caso de prueba 2: abrir-configurar-depositar-sumar-limitecredito-retirar-cerrar.
El caso de prueba 1 cambia de estado, mientras que el caso de prueba 2, ejercita operaciones que no cambian de estado (aparte de las que se encuentran en la secuencia de prueba mínima).

La **partición basada en atributos** ordena en categoría las operaciones de clase basadas en los atributos que utilizan. En el caso de la clase **Cuenta**, los atributos *saldar* y *limitecredito* se emplean para definir particiones. Las operaciones se dividen en tres particiones: 1) operaciones que usan *limitecredito*, 2) operaciones que modifican *limitecredito*, y 3) operaciones que no usan ni modifican *limitecredito*. Entonces se diseñan secuencias de pruebas para cada partición.

La **partición basada en categorías** ordena en categorías las operaciones de clase con base en la función genérica que cada una realiza. Por ejemplo, las operaciones de clase **Cuenta** se organizan en operaciones de *inicialización [abrir(), configurar()]*, *operaciones computacionales [depositar(), retirar()]*, *consultas [saldar(), sumar(), limitecredito()]* y *de terminación [cerrar()]*.

2.4.6. Diseño de casos de prueba de Interclase

El diseño de casos de prueba se vuelve más complicado cuando empieza la integración del sistema OO. En esta etapa debe empezar la prueba de colaboración entre clases. Como en la prueba de clases individuales, la prueba de colaboración entre clases se lleva a cabo al aplicar métodos aleatorios y de partición, además de pruebas basada en el escenario y de comportamiento. Para ilustrar la "generación de pruebas de casos de prueba interclase", se expande el ejemplo del sistema bancario.



NIP: Número de Identificación Personal.

La dirección de las flechas en la figura indica la dirección de los mensajes. Y las etiquetas individuales indican las operaciones que se invocan como consecuencia de la colaboración indicada por los mensajes. Como en la prueba de clases individuales, la prueba de colaboración entre clase se lleva acabo al aplicar métodos aleatorios y de partición, además de pruebas basadas en escenarios y de comportamiento.

Prueba basada en escenarios

La prueba basada en escenarios permite descubrir dos tipos importantes de errores: 1) especificaciones incorrectas y 2) interacciones entre subsistemas.
La prueba basada en escenarios se concentra en lo que hace el usuario, no el producto. Esto significa que se deben capturar las tareas (mediante casos de uso) que el usuario tiene que realizar y luego aplicarlas, junto con sus variantes, como pruebas.
Los escenarios descubren errores de interacción. La prueba basada en escenarios tienden a ejercitar varios subsistemas en una sola prueba (los usuarios no se limitan al uso de un subsistema a la vez).
Téngase en cuanta, que aunque la prueba basada en escenarios tenga méritos, se obtendrán mejores resultados por tiempo invertido si se revisan casos de uso cuando se desarrollan como parte del modelo de análisis.

2.4.6.1. Prueba de clase múltiple

Esta es una prueba que está orientada a probar la integración de una clase con otra. SE sugieren los siguientes pasos para generar varios casos de prueba aleatorios de clases múltiples:

1. En cada clase cliente use la lista de operaciones de clase para generar una serie de secuencias de pruebas aleatorias. Las operaciones enviarán mensajes a otras clases del servidor.
2. En cada mensaje generado, determínese la clase colaboradora y la operación correspondiente en el objeto servidor.
3. En cada operación del objeto servidor (invocada por los mensajes enviados desde el objeto cliente) determínese los mensajes que transmite.
4. En cada uno de los mensajes, determínese el siguiente nivel de operaciones invocadas e incorpórelos en la secuencia de prueba.

Como ejemplo, considérese una secuencia de operaciones para la clase **Banco** relacionada con una clase **CajeroAutomatico**:

verificarCuenta*verificarNIP*[[verificarPolitica*solicitudRetiro] | solicitarDeposito | solicitarInfoCuenta]^{n}

Un caso de prueba aleatoria para la clase **Banco** sería:
Caso de prueba r3 = **verificarCuenta * verificarNIP * solicitarDeposito**

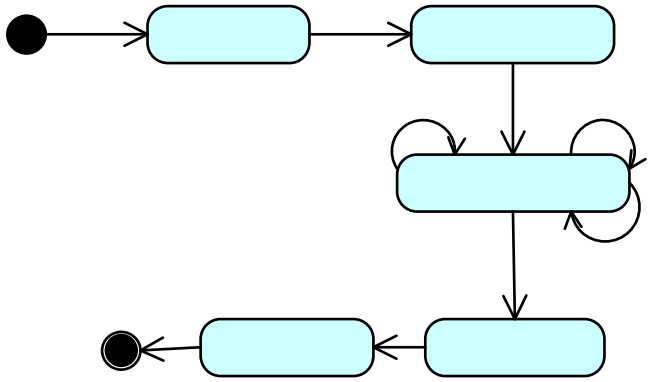
Considerar a los colaboradores que participan en la prueba requiere tomar en cuenta los mensajes asociados con cada una de las operaciones indicadas en el caso de prueba r3. **Banco** debe colaborar con **infoValidacion** para ejecutar *verificarCuenta()* y *verificarNIP()*. **Banco** debe colaborar con **Cuenta** para ejecutar *solicitarDeposito()*. Por lo tanto, se tiene un nuevo caso de prueba que ejercita estas colaboraciones:

Caso de prueba r4 = **verificarCuentaBanco [validarCuentaValidación] verificarNIPBanco*[validarNIPInfovalidacion]* solicitarDEposito*[depositarcuenta]**

El enfoque para la prueba de partición de clases múltiples es similar al enfoque empleado para la de clases individuales. SE somete a partición a una sola clase. Sin embargo, se expande la secuencia de pruebas para incluir las operaciones invocadas mediante mensajes a las clases que colaboran. Un enfoque alterno lleva a cabo la partición de las pruebas con base en las interfaces de una clase determinada. Si por ejemplo, la clase **Banco** recibe mensajes de las clases **CajeroAutomático** y **Cajero**, los métodos dentro de **Banco** se prueban al particionarlas entre las que sirven a **CajeroAutomatico** y las que sirven a **Cajero**. La partición basada en estado se usa para refinar aún más las particiones.

2.4.6.2. Pruebas derivadas de modelos de comportamiento

El diagrama de estado de una clase ayuda a derivar la secuencia de pruebas que revisa el comportamiento dinámico de la clase (y las clases que colaboran con ellas). Observe el diagrama de estado de la clase **Cuenta**:



Las transiciones iniciales recorren los estados *vaciar cuenta* y *configurar cuenta*. Un retiro final y un cierre de la cuenta causan que la clase **Cuenta** haga transiciones a los estados *Cuanta Inactiva* y *Cuenta Muerta*, respectivamente.
Las pruebas que se diseñen deben cubrir todos los estados. Es decir, las secuencias de operación deben lograr que la clase **Cuenta** haga una transición a todos los estados permisibles.

Caso de prueba 1: **abrir-configurarcuenta-depositar(inicial)-retirar(final)-cerrar**

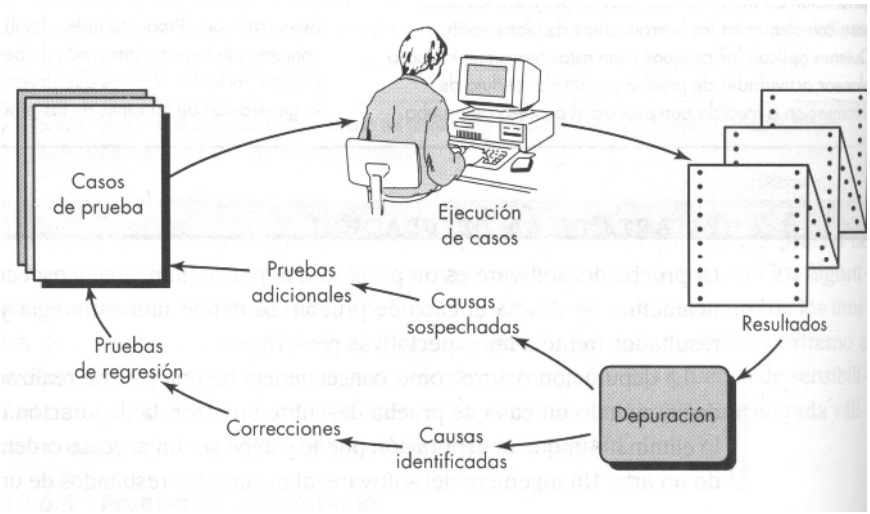
Siendo esta la secuencia mínima de prueba, a la que se debe agregar:
Caso de prueba 2: **abrir-configurarCuenta-depositar(inicial)-depositar-saldar-acreditar-retirar(final)-cerrar**
Caso de prueba 3: **abrir-configurarCuenta-depositar(inicial)-depositar-retirar-infocuenta-retirar(final)-cerrar**

Es posible derivar aún más casos de prueba para asegurar que todos los comportamientos de la clase se hayan ejercitado adecuadamente. En situaciones en que el comportamiento de la clase da como resultado la colaboración con una o más clases, se utilizan varios diagramas de estado para registrar el flujo del comportamiento del sistema.

El modelo de estado puede recorrerse de una manera "primero-en-anchura". En este contexto, primero-en-anchura indica que un caso de prueba solo ejercita una transición. Cuando debe probarse una nueva transición sólo se utilizan transiciones probadas previamente.
Imagínese que el objeto **TarjetaCredito** es parte del sistema bancario. El estado inicial de **TarjetaCredito** es *indefinido* (es decir, no se ha proporcionado un número de tarjeta de crédito). Tras leer la tarjeta durante una venta, el objeto toma un estado *definido;* es decir, se definen los atributos **número, tarjeta y fecha** vencimiento, junto con los identificadores específicos del banco. La tarjeta de crédito es *remitida* cuando se le envía para autorización, y es *aprobada* cuando se recibe la autorización. La transición de **TarjetaCredito** de un estado a otro se prueba derivando casos de prueba que causen la transición. Un método primero-en-anchura para este tipo de prueba no ejercitaría *remitida* antes de *indefinida* o *definida*. En este caso, usaría transiciones que no se han probado y, por tanto, violaría el criterio primero-en-anchura.

2.7. El arte de la depuración

La depuración no es una prueba pero siempre ocurre como consecuencia de una. El proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados y se encuentra una falla de correspondencia entre el desempeño esperado y el real. En muchos casos, los datos que no corresponden son síntomas de una causa que aún no aparece. La depuración trata de relacionar el síntoma con la causa, lo que conduce a corregir el error.



La depuración siempre arroja dos resultados:
1) se encuentra y se corrige la causa, o
2) no se localiza la causa.

En este último caso, la persona encargada de la depuración debe sospechar la causa, diseñar uno o más casos de prueba que ayuden a convalidar esa sospecha y avanzar hacia la corrección del error de manera iterativa.
En ocasiones la depuración se vuelve un tanto difícil dado que:

a. El síntoma y la causa pueden estar separados geográficamente. Es decir, aquel aparece en una parte del programa mientras esta se ubica en un sitio distante. Los componentes con un fuerte acoplamiento exacerban esta situación.

b. Es posible que el síntoma desaparezca (temporalmente) al corregir otro error.
c. Es probable que el síntoma no lo cause algún error (como en el caso de inexactitudes al redondeo de cifras).
d. El síntoma podría deberse a un error humano difícil de localizar.
e. El síntoma podría deberse a problemas de tiempo y no de procesamiento.
f. Tal vez sea difícil reproducir con exactitud las condiciones de entrada (por ejemplo, una aplicación en tiempo real en que no está definido el orden de entrada).
g. El síntoma podría presentarse intermitentemente. Esto suele ser común en sistemas empotrados que acoplan el hardware y el software de manera inextricable.
h. Probablemente el síntoma se debe a causas distribuidas entre varias tareas que se ejecutan en diferentes procesadores.

2.7.1. Estrategias de la depuración

En general, se han propuesto tres estrategias de depuración:

1. **Fuerza bruta.** Los métodos de depuración por la fuerza bruta se aplican cuando todo lo demás falla. Al aplicar esta filosofía, se hacen descargas de memoria, se invocan señales de tiempo de ejecución y se carga el programa con instrucciones de salida. En algún lugar del pantano de información que se produce se espera encontrar una pista que pueda conducir a la causa del error. Aunque la gran cantidad de información producida conduzca al éxito, lo más frecuente es que haga desperdiciar tiempo y esfuerzo.
2. **Rastreo hacia atrás.** Se utiliza con éxito en pequeños programas. Empezando en el sitio donde se ha descubierto un síntoma, se recorre hacia atrás el código fuente (manualmente) hasta hallar el sitio de la causa. Por desgracia, a medida que aumenta el número de líneas de código, la cantidad de caminos hacia atrás se vuelve tan grande que resulta inmanejable.
3. **Eliminación de causas.** Este enfoque lo determina la inducción o deducción e introduce el concepto de partición binaria. Los datos relacionados con el error se organizan para aislar las causas posibles. Se elabora una "hipótesis de la causa" y se aprovechan los datos ya mencionados para probar o desechar la hipótesis. Como opción se elabora una lista de todas las posibles causas y se aplican pruebas para eliminar cada una de ellas. Si las pruebas iniciales indican que determinada hipótesis de causa es prometedora, se refinan los datos para tratar de aislar el error.

Además, considere la posibilidad de solicitar ayuda a otra persona durante la revisión de las fallas y defectos. Un punto de vista fresco, despejado de horas de frustración, puede hacer maravillas. Sin mencionar que el uso de herramientas automáticas de depuración pueden ser un gran aliado.

2.7.1.1. Corrección del error

Cuando se encuentra un error debe corregirse. Pero, el corregir un error puede introducir otros y, por lo tanto, causar más daños que solucionar el problema. A fin de evitar esta situación es conveniente realizar las siguientes preguntas:

- ¿La causa del error se repite en otra parte del programa? En muchas situaciones un error se produce en un programa debido aun patrón erróneo de lógica que podría repetirse en cualquier lugar. La consideración explícita del patrón lógico puede llevar al descubrimiento de otros errores.
- ¿Cuál es el "siguiente error" que podría introducirse con la corrección que está a punto de realizarse? Antes de la corrección se debe evaluar el código fuente (o mejor aún, el diseño) para evaluar el acoplamiento entre las estructuras lógicas y de datos. Si la corrección se realiza en una sección del programa con un acoplamiento elevado, debe tenerse mucho cuidado cuando se haga cualquier cambio.
- ¿Qué debió hacerse para evitar este error desde un principio? Esta pregunta es el primer paso hacia el establecimiento de un enfoque estadístico de aseguramiento de la calidad del software. Si se corrige el proceso junto con el producto, eliminaría el error del programa actual y de todos los programas futuros.

Por otra parte, pueden usarse varias técnicas para manejar el agregado de nuevos errores a partir de las correcciones:

- Un seguimiento del problema incluye la documentación de cada falla, error o defecto detectado, su corrección, la fundamentación de los cambios y las revisiones de los componentes involucrados en el cambio. Junto con la administración de la configuración, el seguimiento de problemas permite que los desarrolladores estrechen la búsqueda de nuevos defectos.
- Las pruebas de regresión.