



UT5. Pruebas y depuración del software

ENTORNO DE DESARROLLO

Introducción

Introducción

Siendo realistas, es prácticamente imposible realizar pruebas exhaustivas a un programa. Ya lo decía Edsger Dijkstra en su momento, y es que las pruebas generalmente son demasiado costosas. Salvo que el programa sea tan importante como para realizarlas, lo que se hace es llegar a un punto intermedio en el cual se garantiza que no va a haber defectos importantes o muchos defectos y la aplicación está completamente operativa con un funcionamiento aceptable. El objetivo de las pruebas es convencer, tanto a los usuarios como a los propios desarrolladores, de que el software es lo suficientemente robusto como para poder trabajar con él de forma productiva. Cuando un software supera unas pruebas exhaustivas, las probabilidades de que ese software dé problemas en producción se atenúan y, por tanto, su fiabilidad aumenta.

"Las pruebas solo pueden demostrar la presencia de errores, no su ausencia".
Edsger Dijkstra.

Planificación de pruebas a lo largo del ciclo de desarrollo

Planificación de pruebas a lo largo del ciclo de desarrollo

La alta calidad no se puede lograr sólo mediante la prueba del código fuente. Las pruebas por sí solas no podrían garantizar, como ya hemos comentado, la ausencia de errores de un programa. Se dice que el número de errores ocultos en un programa es proporcional al número de los ya descubiertos (por tanto, se tiene menos confianza en programas que ya han sufrido correcciones). La mejor manera de minimizar el número de errores de un programa es encontrarlos y eliminarlos durante el análisis y el diseño, de modo que se introduzcan pocos errores en el código fuente.

Podemos considerar dos enfoques antitéticos para la realización de las pruebas de software, el testeo exploratorio y el guiado. En realidad, las pruebas casi siempre son una combinación de ambos, pero con una tendencia hacia uno de ellos.

El testeo exploratorio

Su principal característica es que el aprendizaje, el diseño y la ejecución de las pruebas se realizan de forma simultánea. Mientras se está probando el software, el testeador, también llamado tester, va aprendiendo a manejar el sistema y junto con su experiencia y creatividad, genera nuevas pruebas a ejecutar. Al realizar testeo exploratorio, algunos resultados pueden ser precedidos y esperados, otros puede que no.

El testeador configura, opera, observa y evalúa el producto y su comportamiento, investigando de forma crítica el resultado y reportando la información de lo que parecen ser defectos (que amenazan el valor del producto) o problemas (que amenazan la continuidad y calidad de las pruebas).

Estas pruebas son realmente útiles a la hora de probar aplicaciones ya desarrolladas, es decir, aquellas pruebas de software que no comienzan a la vez que el desarrollo: se identificarán los distintos módulos de la aplicación y se le dará libertad al testeador, que se pondrá en la piel de un usuario para probarlos.

El testeo guiado

También llamado testeo basado en procedimientos de prueba. En este enfoque los casos de prueba son diseñados con antelación, incluyendo tanto los pasos a realizar como el resultado esperado. Estas pruebas son más tarde ejecutadas por un testeador que compara el resultado actual con el esperado.

Podemos decir que, un procedimiento de prueba es la definición del objetivo que desea conseguirse con las pruebas, qué es lo que va a probarse y cómo.

El objetivo de las pruebas no siempre es detectar errores. Muchas veces lo que quiere conseguirse es que el sistema ofrezca un rendimiento determinado, que la interfaz tenga una apariencia y cumpla unas características determinadas, etc. Por lo tanto, la ausencia de errores en las pruebas nunca significa que el software las supere, pues hay muchos parámetros en juego.

El testeo guiado

Cuando se diseñan los procedimientos, se deciden las personas que hacen las pruebas y bajo qué parámetros van a realizarse. No siempre tienen que ser los programadores los que hacen las pruebas. No obstante, siempre tiene que haber personal externo al equipo de desarrollo, puesto que los propios programadores solo prueban las cosas que funcionan (si supieran dónde están los errores, los corregirían).

Hay que tener en cuenta que es imposible probar todo, la prueba exhaustiva no existe. Muchos errores del sistema saldrán en producción cuando el software ya esté implantado, pero siempre se intentará que sea el mínimo número de ellos.

El testeo guiado

En los planes de pruebas (es un documento que detalla en profundidad las pruebas que se vayan a realizar), generalmente, se cubren los siguientes aspectos:

1. Introducción. Breve introducción del sistema describiendo objetivos, estrategia, etc.
2. Módulos o partes del software por probar. Detallar cada una de estas partes o módulos.
3. Características del software por probar. Tanto individuales como conjuntos de ellas.
4. Características del software que no ha de probarse.
5. Enfoque de las pruebas. En el que se detallan, entre otros, las personas responsables, la planificación, la duración, etc.
6. Criterios de validez o invalidez del software. En este apartado, se registra cuando el software puede darse como válido o como inválido especificando claramente los criterios.
7. Proceso de pruebas. Se especificará el proceso y los procedimientos de las pruebas por ejecutar.
8. Requerimientos del entorno. Incluyendo niveles de seguridad, comunicaciones, necesidades hardware y software, herramientas, etc.
9. Homologación o aprobación del plan. Este plan deberá estar firmado por los interesados o sus responsables.

Las demás fases del proceso de pruebas, como puede entenderse, son el mero desarrollo del plan de pruebas anterior.

Casos de prueba

En la fase de pruebas, se diseñan y preparan los casos de prueba, que se crean con el objetivo de encontrar fallos. Por experiencia, no hay que probar los programas de forma redundante. Si se prueba un software y funciona, la mayoría de las veces no hace falta probar lo mismo. Hay que crear otro tipo de pruebas, no repetirlas. Lo que no implica que ante un cambio de software se ejecuten todas las pruebas otra vez, para verificar que no hay ningún problema con los cambios introducidos.

Que las pruebas tengan que hacerse en la fase de pruebas no implica que tengan que hacerse después de la fase de desarrollo. Conceptos como TDD están cambiando la concepción de cuando se inicia la fase de pruebas.

Hay que tener en cuenta que la prueba no debe ser muy sencilla ni muy compleja. Si es muy sencilla, no va a aportar nada y, si es muy compleja, quizá, sea difícil encontrar el origen de los errores.

Casos de prueba

Como se ha observado, las pruebas solo encuentran o tratan de encontrar aquellos errores que van buscando, luego, es muy importante realizar un buen diseño de las pruebas con buenos casos de prueba, puesto que se aumenta de esta manera la probabilidad de encontrar fallos.

| | | | |
|--|--|---|---|
| Nombre de Proyecto: Xebee | | ID Caso de Prueba: CP-001 | |
| Ambiente de Prueba: Access/Web | | ID Historia de Usuario: HU-001 | |
| Autor Caso de Prueba: Mariam Rodríguez | | | |
| Propósito | | | |
| Verificar que los tipos de documentos cargados desde Access como Invoice for attorney se puedan visualizar en la aplicación web en el Tab correspondiente. | | | |
| Descripción de las Acciones y/o condiciones para las Pruebas | | | |
| # | Acciones | Salida Esperada | Salida Obtenida |
| 01 | Agregar un documento a un request desde Access | El documento se carga como tipo solo para uso de Xebee | El documento se cargó con el tipo solo para uso de xebee |
| 02 | Cambiar el tipo de Documento a INVOICE FOR ATTORNEY desde Access | Actualizar el tipo de documento, “solo para uso interno” a Invoice for attorney | Se actualizó el tipo de documento, “solo para uso interno” a Invoice for attorney |
| 03 | Buscar request desde la Web | Debe existir en el listado de request. | Se encontró el request. |
| 04 | Consultar Tab Invoicing de la Web | Ver el documento que fue cargado desde Access de tipo Invoice for attorney | El documento estaba en la pestaña tab invoicing |
| 05 | Consultar Tabs (Inf. General/documentación de soporte) de la Web | El documento de tipo Invoice for attorney no debe verse en ninguno de estos tab | El documento no es visible desde estos tab. |
| Resultados Obtenidos | | | |
| Resultado: Aprobado | | | |
| Seguimiento: No Aplica | | Severidad: NO Aplica | |
| Evidencia: | | | |

Codificación y ejecución de las pruebas

Codificación y ejecución de las pruebas

Una vez diseñados los casos de prueba, hay que generar las condiciones necesarias para poder ejecutar dichos casos de prueba. Habrá que codificarlos en muchos casos generando set o conjuntos de datos. En estos set de datos, hay que incluir tanto datos válidos e inválidos como algunos datos fuera de rango o disparatados.

También habrá que preparar las máquinas sobre las que van a hacerse las pruebas instalando el software necesario, los usuarios de sistema, realizar carga del sistema, etc.

Una vez definidos los casos de prueba y establecido el entorno de las pruebas, es el momento de su ejecución. Irán ejecutándose los casos de prueba uno a uno y, cuando se detecte algún error, hay que aislarlo y anotar la acción que estaba probándose, el caso, el módulo, la fecha, la hora, los datos utilizados, etc. De esa manera, intentará documentarse lo más detalladamente posible el error. En el caso de que se produzcan errores aleatorios, también hay que registrarlos anotando este hecho.

Tipo de pruebas

Tipo de pruebas

Las pruebas con frecuencia se agrupan por su nivel de especificidad. Los principales niveles durante el proceso de desarrollo son las pruebas de unidad, de integración, y de sistema, que se distinguen por el destinatario de la prueba:

- Pruebas de unidad o pruebas unitarias.
- Pruebas de integración.
- Pruebas de sistema.

La prueba de unidad

También conocidas como pruebas de componentes, se refieren a pruebas para verificar el funcionamiento de una sección específica de código, por lo general en el nivel de función, o, en un entorno orientado a objetos, en el nivel de clase.

Este tipo de pruebas se escriben normalmente por los desarrolladores a medida que trabajan en el código, para asegurarse de que la función específica está funcionando como se esperaba; sobre una función se pueden realizar varias pruebas.

Las pruebas de unidad se utilizan para asegurar que los componentes básicos del software trabajan correctamente por separado. Es importante darse cuenta de que las pruebas unitarias no descubrirán todos los errores del código. Por definición, sólo prueban las unidades por sí solas; por lo tanto, no descubrirán errores de integración, problemas de rendimiento ni otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas que puede recibir la unidad de programa bajo estudio. Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras pruebas de software.

La prueba de unidad

Las pruebas de unidad proporcionan cinco ventajas básicas:

- ✓ **Fomentan el cambio:** Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
- ✓ **Simplifican la integración:** Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las pruebas de integración.
- ✓ **Documentan el código:** Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
- ✓ **Separación de la interfaz y la implementación:** Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
- ✓ **Los errores están más acotados y son más fáciles de localizar:** Dado que tenemos pruebas unitarias que pueden desenmascararlos.

Las pruebas de integración

Son pruebas del software que tratan de verificar las interfaces entre los componentes que fueron definidas al diseñar el software. Progresivamente grupos cada vez mayores de componentes de software ya probados se van integrando y probando hasta que el software funciona como un sistema.

Las pruebas de sistema

Prueban un sistema completamente integrado para verificar que cumple con sus requisitos. Estas pruebas deben garantizar que el software, además de trabajar como se esperaba, no destruye o corrompe parcialmente su entorno operativo ni causa que otros procesos dentro de ese entorno dejen de funcionar (por ejemplo, por consumir o bloquear excesivos recursos).

Técnicas de pruebas

Técnicas de prueba

Las técnicas de prueba se clasifican en dos grandes tipos:

- **Pruebas de caja blanca** (también conocidas como pruebas de caja de cristal o pruebas estructurales). Se analiza el interior del programa, es decir, la lógica y la estructura de su código fuente. Las pruebas de caja blanca se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. El testeador escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados. Al estar basadas en una implementación concreta, si ésta se modifica, por regla general las pruebas también deberán rediseñarse.
- **Pruebas de caja negra.** De una caja negra interesa conocer qué es lo que hace, pero no cómo lo hace. Por tanto, en las pruebas no se analiza el interior del programa, sino que conociendo la función específica para la que fue diseñado el producto, sólo interesa que funcione como es debido: se aplica una serie de entradas y se comprueba si las salidas son correctas.

Test-Driven Development

El Desarrollo de software guiado por pruebas (TDD — Test-Driven *Software* Development), adoptado por las metodologías ágiles, involucra dos prácticas: escribir las pruebas primero (Test First Development) y refactorizar. Generalmente utilizan pruebas unitarias: en primer lugar, se escribe una prueba y se verifica que la prueba falla; a continuación, se implementa el código que hace que la prueba se pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del TDD es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas; de este modo, la superación de las pruebas garantizará que el software cumple con los requisitos que se han establecido.

Para que el TDD funcione, el software que se programa tiene que ser lo suficientemente flexible como para permitir que sea probado automáticamente; frameworks como JUnit proveen de mecanismos para manejar y ejecutar conjuntos de pruebas automatizadas. Se podría argumentar que el TDD "desperdicia" la capacidad del programador escribiendo las pruebas unitarias; los seguidores de esta metodología responderían que asegurar la calidad del producto no es desperdicio alguno.

Pruebas de regresión

Se denominan **pruebas de regresión** a las pruebas de un sistema o componente que pretenden verificar que las modificaciones realizadas sobre el mismo no han causado efectos no deseados, y que el sistema o componente sigue cumpliendo con sus requisitos especificados. Los diferentes casos de prueba de unidad y de integración que se van superando se pueden conservar para volver a ejecutarlos más adelante como parte de las pruebas de regresión. De hecho, se considera una buena práctica de programación que cuando se localiza un bug, se diseñe una prueba que lo ponga de manifiesto, y, tras corregirlo, dicho test se añada a la batería de pruebas de regresión. La práctica habitual en la programación extrema es que los test de regresión se ejecuten en cada uno de los pasos del ciclo de vida del desarrollo del software.

Técnicas comunes de pruebas de unidad

Técnicas comunes de pruebas de unidad

Para que una prueba unitaria sea buena debe cumplir los siguientes requisitos:

- **Automatizable:** No debería requerirse una intervención manual.
- **Completa:** Debe cubrir la mayor cantidad de código.
- **Repetible o reutilizable:** No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- **Independiente:** La ejecución de una prueba no debe afectar a la ejecución de otra.
- **Profesional:** Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Técnicas comunes de pruebas de unidad

Para la realización de las pruebas de unidad se utilizan diversas técnicas:

- **Las pruebas funcionales.** Son pruebas de caja negra. Las pruebas funcionales verifican una determinada acción o función del código. Consisten en ejercitar el código con valores nominales de entrada para los que se conocen los resultados esperados. Si la entrada de datos está limitada a un rango de valores se utiliza la técnica del análisis de los valores límite, que se basa en elegir entradas cercanas a los límites del rango pues en ellos muchas veces ocurren errores. Además, se prueban otras entradas especiales, como archivos vacíos, archivos de elementos idénticos, etc.
- **Las pruebas estructurales.** Son las pruebas de caja blanca. Ejercitan la lógica interna de un programa recorriendo rutas de ejecución particulares. Las actividades principales son: decidir cuáles rutas ejecutar, obtener los datos de prueba para ejercitar esas rutas, determinar el criterio de cobertura de la prueba que se usará, ejecutar los casos de prueba, y medir la cobertura de la prueba lograda cuando se ejercitaron esos casos.

Técnicas comunes de pruebas de unidad

Mediante las técnicas de prueba de caja blanca se pueden diseñar casos de prueba que:

- Ejerciten caminos independientes dentro de un módulo.
- Ejerciten decisiones lógicas en sus vertientes verdadera y falsa.
- Ejerciten bucles en sus límites y en sus rangos de operación.

Aunque estas pruebas son utilizables a varios niveles —unidad, integración y sistema—, habitualmente se aplican a las unidades de software.

Puede parecer mucho trabajo diseñar estos casos de prueba; sin embargo, las estadísticas muestran que un diseño de las pruebas estructurales cuidadoso, completo y sistemático encontrará tantos bugs como la realización de las pruebas diseñadas en ese proceso.

Partición en clases de equivalencia y el análisis de valores límite

Partición en clases de equivalencia y el análisis de valores límite

La partición en clases de equivalencia (ECP — Equivalence Class Partitioning), también llamada **prueba de dominios** es una técnica de pruebas de software, habitualmente de **caja negra**, que divide los datos de entrada y los resultados de salida de una unidad de software en particiones de datos equivalentes de las que se pueden derivar los casos de prueba.

Los datos de entrada y los resultados de salida de un programa normalmente se pueden agrupar en varias clases diferentes que tienen características comunes; los programas normalmente se comportan de una forma similar para todos los miembros de una clase. Debido a este comportamiento equivalente, estas clases se denominan a menudo particiones de equivalencia o dominios (Bezier, 1990). Cada partición de equivalencia es un conjunto de datos que deberían ser procesados de forma equivalente. Las particiones de equivalencia de salida son resultados del programa que tienen características comunes, por lo que pueden considerarse como una clase diferente.

Partición en clases de equivalencia y el análisis de valores límite

En general, una entrada tiene ciertos rangos que son válidos y otros que no lo son. Datos no válidos aquí no quiere decir que los datos sean incorrectos; significa que estos datos se encuentran fuera de la partición específica. Por ejemplo, supóngase una función que toma un parámetro “mes”: el rango válido para el mes es de 1 a 12, representando enero a diciembre; este rango válido se denomina partición. En este ejemplo hay dos particiones adicionales de rangos no válidos: la primera partición no válida sería ≤ 0 y la segunda partición no válida sería ≥ 13 .

La técnica ECP afirma que sólo se necesita un caso de prueba de cada partición para evaluar el comportamiento del programa para la partición relacionada. En otras palabras, es suficiente seleccionar un caso de prueba de cada partición para comprobar el comportamiento del programa. Utilizar más o incluso todos los casos de prueba de una división no encontrará nuevos fallos en el programa, pues los valores dentro de una partición se consideran “equivalentes”. Así, el número de casos de prueba se puede reducir considerablemente.

Partición en clases de equivalencia y el análisis de valores límite

ECP no es un método independiente para determinar los casos de prueba. Tiene que ser complementado con el **análisis de valores límite** (BVA — Boundary-value analysis).

BVA es una técnica de pruebas de software por la que los valores en los bordes mínimo y máximo de cada partición de equivalencia se han de incluir en los casos de prueba, dado que estas fronteras son lugares comunes para errores del software. En el ejemplo anterior el número 0 es el valor máximo en la primera partición y el número 1 es el mínimo en la segunda partición; ambos son valores límite. Se deben crear casos de prueba para ambos lados de cada frontera, lo que da lugar a dos casos por cada límite. Cuando un valor límite cae dentro de una partición no válida, el caso de prueba será diseñado para asegurar que el componente software maneja ese valor de una manera controlada.

Partición en clases de equivalencia y el análisis de valores límite

ECP y BVA proporcionan una estrategia para escribir casos de prueba, tanto de caja negra como de caja blanca. Cada vez que surge cualquier tipo de rango o límite en un algoritmo, se debería estar alerta respecto a las particiones. Nótese que en el caso de pruebas de caja blanca, y con existencia de bucles (tales como los bucles while), hay que aplicar ECP para ejecutar el bucle en el centro de su rango operativo, y utilizar BVA en los límites del mismo.

Supongamos que una persona quiere comprar una televisión; puede o no tener suficiente dinero. Habríamos de asegurarnos de que nuestros casos de prueba incluyen lo siguiente:

- La propiedad cuesta 100 €, tiene 200 € (partición "tiene suficiente dinero")
- La propiedad cuesta 100 €, tiene 50 € (partición "no tienen suficiente dinero")
- La propiedad cuesta 100 €, tiene 100 € (valor límite)
- La propiedad cuesta 100 €, tiene 99 € (valor límite)

Partición en clases de equivalencia y el análisis de valores límite

En la práctica es poco probable tener una unidad de software que reciba únicamente una entrada; en general hay más entradas. Cuando se tiene más de una variable de entrada hay que identificar las clases de equivalencia de cada variable; luego se deberán seleccionar valores de cada partición para todas las variables y combinarlos formando los casos de prueba. Este procedimiento puede automatizarse, de modo que se pueda reducir el trabajo, ya que el número de casos puede crecer enormemente. Supongamos que queremos probar un método que muestra el mayor de dos números enteros, cada uno en el rango de 0 a 1000. Si ambos enteros son iguales, el programa muestra su valor.

| Entrada 1 | Entrada 2 | Salida |
|-----------|-----------|--------|
| 0 | 0 | 0 |
| 0 | 550 | 550 |
| 0 | 1000 | 1000 |
| 450 | 0 | 450 |
| 450 | 550 | 550 |
| 450 | 1000 | 1000 |
| 1000 | 0 | 1000 |
| 1000 | 550 | 1000 |
| 1000 | 1000 | 1000 |

Cada entrada válida toma valores de una partición que abarca de 0 a 1000; elegiremos un valor centrado (ECP) y un valor en cada extremo (BVA). Ahora que hemos seleccionado los valores representativos de cada entrada, tenemos que considerar qué combinaciones de valores debemos utilizar: utilizaremos todas las combinaciones de los valores representativos. Así que los casos de prueba (considerando sólo las particiones de rangos válidos de las entradas) serán los mostrados en la tabla de la izquierda.

Prueba de Caminos Básicos

PCB

Prueba de Caminos Básicos

La **Prueba de Caminos Básicos** proporciona el mínimo número de casos de prueba que necesita ser escrito. La Prueba de Caminos Básicos es un medio para asegurar que todos los caminos independientes a través de un módulo de código han sido probados. Un camino independiente es cualquier camino a través del código que introduce al menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición. Se debería escribir un caso de prueba para asegurarse de que cada uno de los caminos independientes se prueba al menos una vez. El método, diseñado por Watson y McCabe en 1996, opera en tres etapas:

1. Convertir el código fuente a un grafo de control.
2. Calcular la complejidad ciclomática y obtener la base de caminos.
3. Preparar un caso de prueba para cada camino básico.

PCB. Grafo de control

La mayoría de las pruebas estructurales parten de convertir el código de la unidad en un grafo dirigido, llamado grafo de control, donde las instrucciones individuales, o secuencias lineales de ellas, forman nodos y los arcos representan el paso de control de la ejecución. El grafo debe cumplir:

- Existe un solo nodo inicial.
- Existe un solo nodo final.
- Cada nodo representa una secuencia de instrucciones, que puede ser vacía.
- La relación entre los nodos es una relación de “el control pasa a”.
- Un nodo puede tener uno o dos sucesores (en el caso de case, más de dos).
- Un nodo puede tener uno o muchos antecesores.
- Un nodo tendrá dos sucesores si existen dos caminos posibles dependiendo de una condición.

PCB. Grafo de control

La ejecución de cualquier algoritmo se realiza en forma secuencial, instrucción por instrucción, a menos que se realice un cambio de dirección, debido a la evaluación de una condición o a un fin de bucle. Para muchas técnicas las secuencias de instrucciones no resultan interesantes, pero sí los cambios de dirección. Un grafo de control representa un algoritmo de un modo en el que se observan con claridad los diferentes puntos donde existen cambios de dirección, permitiendo así analizar los diversos caminos que llevan del inicio al fin del algoritmo. El grafo contendrá dos tipos de elementos:

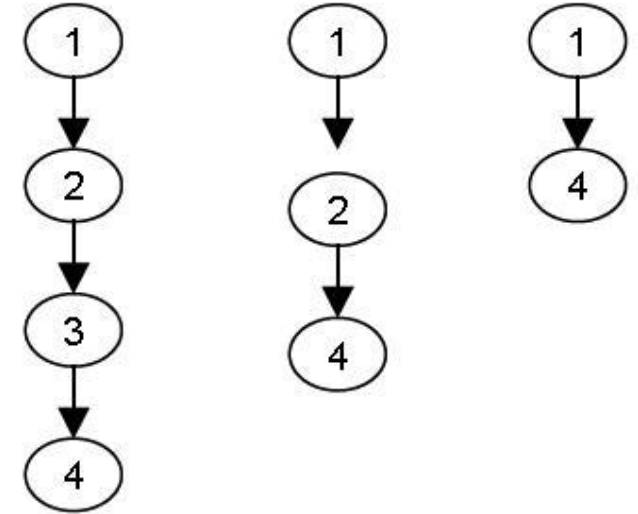
- Nodos que representan una o más instrucciones.
- Arcos que representan cambios de dirección.

PCB. Grafo de control

Al ir formando el grafo es de utilidad numerar las instrucciones y utilizar esos números como etiquetas de los nodos, para así simplificar el trabajo. Además, se recomienda utilizar un nodo inicial y otro terminal únicos, ya que facilita la identificación de los caminos.

En principio cada instrucción se representará como un nodo. Ahora bien, existen dos tipos de instrucciones: las que tienen dos o más sucesores posibles (como el if, la condición del for y el case), y las que tienen un sucesor único (como las asignaciones y las instrucciones de lectura).

```
1. Read(x,y);  
2. A:=x+cte1;  
3. B:=A/cte2;  
4. C:=A * B;
```

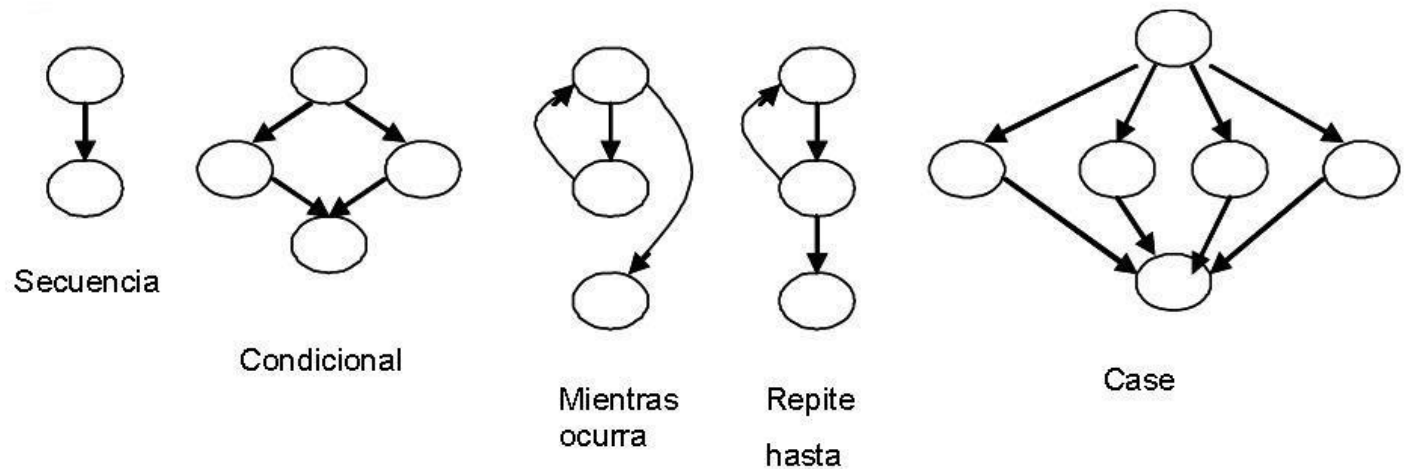


Las instrucciones con sucesor único, es decir las secuenciales, pueden representarse una sola en cada nodo o varias de ellas en uno solo, ya que si se ejecuta la primera se ejecutarán las siguientes una tras otra. En cambio, las instrucciones con sucesor múltiple no pueden unirse a una posterior, ya que hay varias alternativas; sin embargo, sí pueden unirse a un grupo secuencial anterior a ellas.

PCB. Grafo de control

Esto permitirá reducir los elementos a considerar: como se dijo antes, las secuencias de instrucciones donde no hay alternativas pueden considerarse como un solo nodo; así pues, se dejará un solo nodo por cada secuencia de este tipo, que es el primer nodo que no sea secuencial y que sea posterior a todos los elementos de la secuencia. Cuando no se llegue a un nodo condicional, pero sí al fin del algoritmo o a un arco que regresa o cambia de dirección, se dejará el último nodo de la secuencia.

La construcción del grafo de control puede emprenderse de dos maneras, ambas sistemáticas: descendente y ascendente. Las dos se auxilian de las estructuras típicas de la programación estructurada, aunque el grafo de control puede construirse también en algoritmos no estructurados.



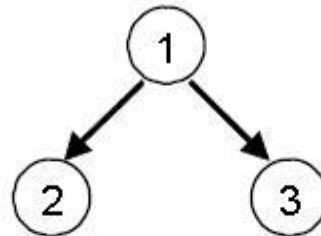
PCB. Grafo de control

- Al seguir la estrategia descendente el programa será inicialmente un único nodo, que se detallará cada vez más hasta donde sea necesario. A cada paso se expande un nodo en varios que lo forman, de acuerdo con las estructuras típicas mostradas anteriormente. Se comienza con los nodos que forman las estructuras iterativas o condicionales principales; luego se expanden éstas.
- La estrategia ascendente comienza, en cambio, por las instrucciones individuales, que se van agrupando formando las estructuras típicas ya presentadas. Usualmente se comienza por instrucciones que están dentro de las iteraciones o condiciones más profundamente anidadas y se avanza hacia el exterior. Si se identifica un bucle o una construcción condicional, se reemplaza por la estructura correspondiente; en las iteraciones siempre existirá un nodo dentro de la iteración.

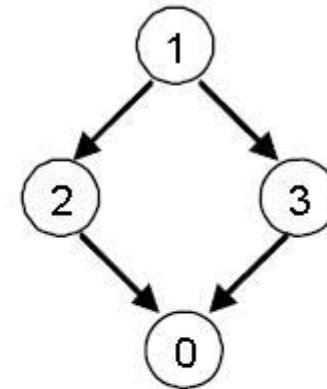
PCB. Grafo de control

A veces resulta confuso tener muchos arcos que llegan a un sólo nodo y se prefiere agregar nodos redundantes que sirven para unir, pero no representan ninguna operación. Tales nodos se etiquetan con un 0 y se llaman conectores. Por ejemplo, en las instrucciones condicionales pueden cerrarse los dos caminos alternativos en un solo punto de terminación.

```
1  if (x<20)
2  then a=1;
3  else a=-1;
```



a) Nodo condicional
y sus dos sucesores



b) Nodo condicional,
con nodo extra para
cerrar los caminos

PCB. Grafo de control

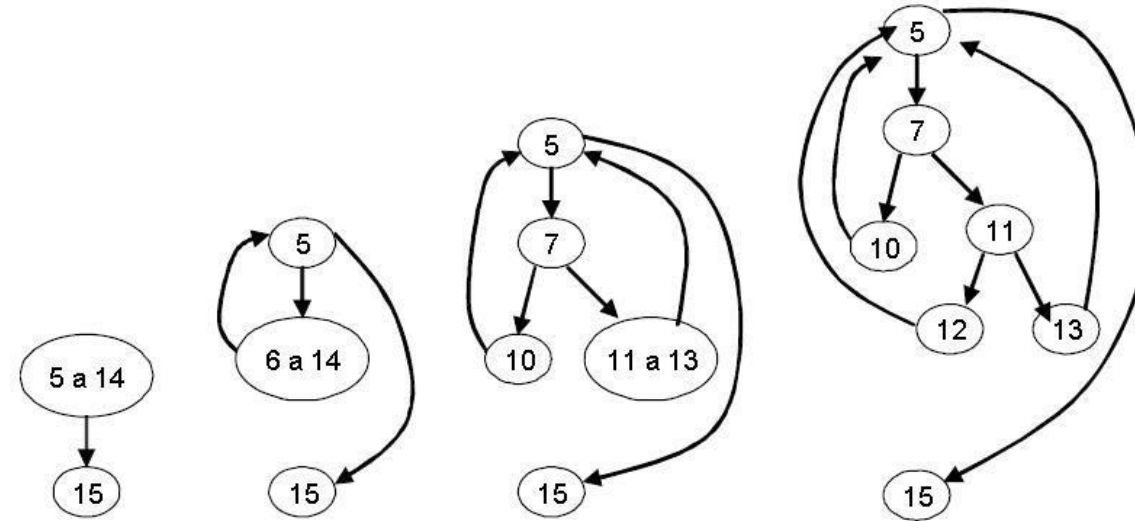
Veamos el siguiente ejemplo: Corresponde a una búsqueda de una clave en una lista ordenada de valores previamente almacenados. La salida será el índice de la lista donde se encontró el valor o -1 si no se encontró. La lista va del índice indMin hasta el indMax.

```
01 Lee clave
02 x1 = indMin
03 x2 = indMax
04 resp = -1
05 mientras ( x1 < x2 - 1 )
06     ix = ( x1 + x2 ) / 2
07     si ( Lista[ix] == clave ) entonces
08         resp = ix
09         x1 = ix
10         x2 = ix
11     de otro modo si ( Lista[ix] < clave ) entonces
12         x1 = ix
13     de otro modo x2 = ix
14 fin del mientras
15 regresa resp
```

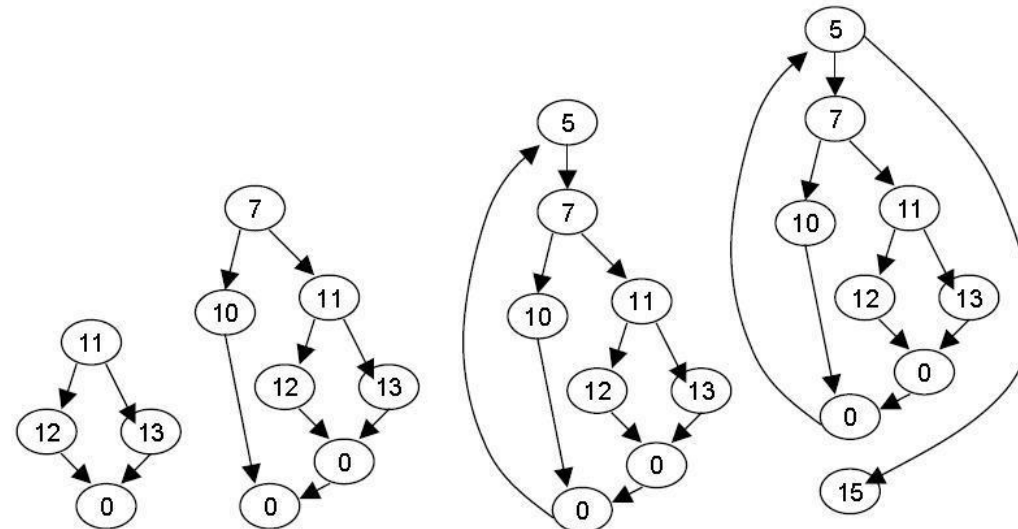
Del 1 al 5 están agrupados porque tiene que pasar siempre en todos
Luego va del 5 al 14
y por último 15

PCB. Grafo de control

- Diseño descendente



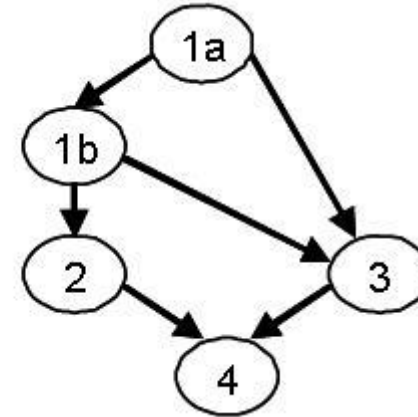
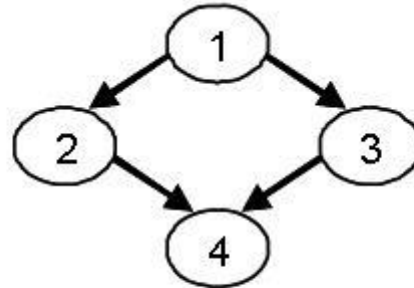
- Diseño ascendente



PCB. Grafo de control

Con lo visto hasta aquí, es posible que el grafo aún esconda parte de la complejidad. El problema reside en las condiciones con predicados compuestos que puedan existir, tanto en las instrucciones condicionales como en las iteraciones. La razón es la siguiente: una condición compuesta, como por ejemplo $a > 2$ AND $b < 0.1$, en ejecución se descompone en varios pasos, evaluándose primero la condición $a > 2$ y después, si resulta verdadera, la otra condición ($b < 0.1$); en caso de evaluarse a falso la primera condición, la segunda no se comprobaría.

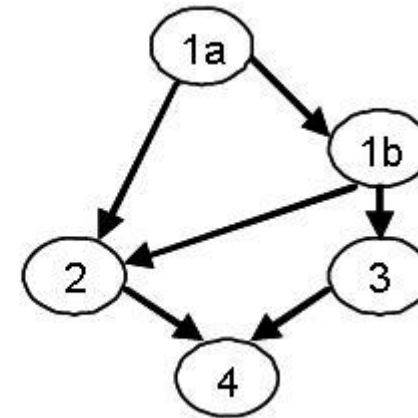
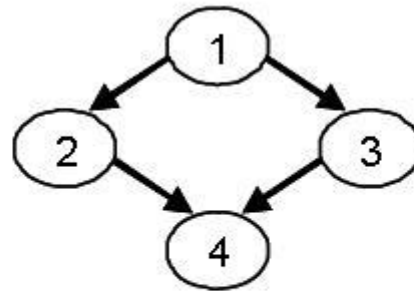
```
1. if (a<3.47 and zx =0)
2. then sd:=2
3. else sd:=1;
4. print (sd);
```



PCB. Grafo de control

Algo similar ocurre con el operador lógico OR. Por todo ello, las condiciones deben separarse en un nodo para cada predicado; se acostumbra a representar el “entonces” a la izquierda y el “de otro modo” a la derecha:

```
1. if (a<3.47 or zx =0)
2. then sd:=2
3. else sd:=1;
4. print (sd);
```



PCB. Complejidad ciclomática

El software puede ser más o menos complejo, dependiendo de la naturaleza del problema y de la implementación particular. El software más complejo origina más problemas: falla con mayor frecuencia, es más lento de desarrollar, contiene más defectos y resulta más difícil de probar.

La complejidad debida a la naturaleza del problema que el software pretende resolver no se podrá reducir; sin embargo, la complejidad asociada con una implementación particular sí se puede analizar y reducir.

Una manera de medir la complejidad del software es el uso de la métrica conocida como “complejidad ciclomática” (o “número ciclomático”), propuesta por McCabe en 1976.

La complejidad ciclomática se calcula a partir del grafo de control del software bajo análisis y mide, a grandes rasgos, el número de caminos independientes que pueden ocurrir en la ejecución, entre el inicio y fin del software.

PCB. Complejidad ciclomática

A partir de un grafo de control G se calcula la complejidad ciclomática $v(G)$ de tres maneras que resultan equivalentes:

- Si a es el número de arcos y n es el número de nodos; entonces $v(G) = a - n + 2$
- Si nps es el número de nodos con predicado simple (es decir, nodos de los que parten dos caminos); entonces $v(G) = 1 + nps$

Nótese que se dice “predicados simples”, es decir, se supone que ya se expandieron los predicados múltiples.

- Si r es el número de regiones rodeadas completamente por arcos del grafo; entonces $v(G) = r + 1$

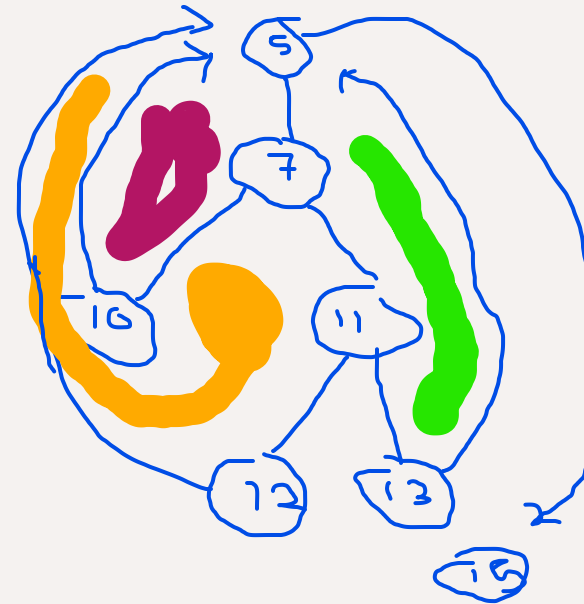
Existe una interpretación muy útil de la complejidad ciclomática: $v(G)$ es la cardinalidad de la base del conjunto de caminos posibles en el grafo, es decir, $v(G)$ determina cuántos son los caminos básicos de un grafo de control.

Actividad 1. Para el código anterior calcule la complejidad ciclomática

```

01 Lee clave
02 x1 = indMin
03 x2 = indMax
04 resp = -1
05 mientras ( x1 < x2 - 1 )
06     ix = ( x1 + x2 ) / 2
07     si ( Lista[ix] == clave ) entonces
08         resp = ix
09         x1 = ix
10         x2 = ix
11     de otro modo si ( Lista[ix] < clave ) entonces
12         x1 = ix
13     de otro modo x2 = ix
14 fin del mientras
15 regresa resp

```



Método 1
Arcos son las líneas
y nodos son los círculos.

Método 2

Del que salen dos ramificaciones

Método 3

Regiones cerradas dentro
(marcadas con otro color)

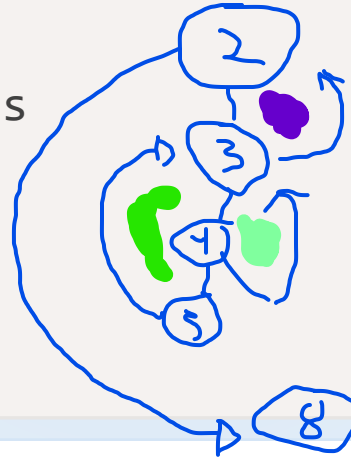
Primer método (arcos y nodos) || $V(g) = 9 - 7 + 2 = 4$

Segundo método (predicado simple) || $V(g)=3+1=4$

Tercer método (regiones cerradas) || $V(g)=3+1=4$

Actividad 2. Para el siguiente código dibuja el grafo y calcule la complejidad ciclomática

```
1 numAdh = 0
2 Para i de 0 hasta nt1-1
3     Para j de 0 hasta nt2-1
4         Si tok1[i]=tok2[j] entonces
5             numAdh = numAdh +1
6 total = tok1 + tok2 - numAdh
7 cohesión = numAdh / total
8 regresa cohesión
```



$$V(g)=7-5+2=4$$

$$V(g)=3+1=4$$

$$V(g)=3+1=4$$

PCB. Caminos básicos

En un grafo de control puede haber un número pequeño de caminos, pero también pueden existir un número infinito de ellos (un grafo que incluya un ciclo puede generar este último caso). Para tratar el problema de un número grande o infinito de caminos McCabe propuso un método de prueba que consiste en probar un conjunto de caminos que formen una base para todos los posibles caminos en el grafo (es decir, a partir de los caminos base se podrán generar algebraicamente todos los demás caminos).

En un grafo existen muchos posibles caminos; los que interesan para pruebas de software son caminos que vayan del nodo inicial al nodo final del grafo (en la práctica puede existir un grafo con varios nodos terminales, pero se les puede reunir en un nodo adicional sin ninguna función).

Si la complejidad ciclomática de un módulo es $v(G)$, se sabe que habrá $v(G)$ caminos básicos y que bastará probar éstos para tener la seguridad de haber ejercitado todas las proposiciones y todas las condiciones del módulo que se está probando.

Sin embargo, el obtener la base de caminos no siempre es tarea fácil y surge la tentación de usar cualquier grupo de caminos; de hacerse así, se corre el riesgo de dejar algún aspecto sin probar. Para asegurarse de no cometer errores como esos, conviene seguir un método.

Método simplificado. Este método tiende a recorrer primero los caminos de excepción, los errores, las salidas de emergencia. Comienza seleccionando el camino más corto de principio a fin y luego va buscando segmentos no recorridos hasta completar el número de caminos necesarios. El procedimiento es como sigue:

1. Asegurarse que haya sólo un nodo inicial y sólo un nodo final
2. Seleccionar el camino más corto entre inicio y fin y agregarlo a la base
3. Fijar numCaminos en 1
4. Mientras existan nodos de decisión con salidas no utilizadas y $\text{numCaminos} < v(G)$,
 - 4.1. Seguir un camino básico hasta uno de tales nodos
 - 4.2. Seguir la salida no utilizada y buscar regresar al camino básico tan pronto sea posible
 - 4.3. Agregar el camino a la base e incrementar numCaminos en 1

Método general. Estos caminos omiten, en primera instancia, las salidas de error y las excepciones y recorren al menos una vez cada ciclo. En este método se elige el primer camino como uno que tenga sentido funcional, es decir, que represente la operación normal del software. A partir de ese camino inicial, se sigue la misma idea del método simplificado, es decir, ir variando una parte cada vez hasta completar los caminos necesarios.

El procedimiento se puede describir como sigue:

1. Asegurarse que haya sólo un nodo inicial y sólo un nodo final
2. Seleccionar un camino funcional, que no sea salida de error, que sea el más importante a ojos del probador y agregarlo a la base
3. Fijar numCaminos en 1
4. Mientras existan nodos de decisión con salidas no utilizadas y $\text{numCaminos} < v(G)$,
 - 4.1. Seguir un camino básico hasta uno de tales nodos
 - 4.2. Seguir la salida no utilizada y buscar regresar al camino básico tan pronto sea posible
 - 4.3. Agregar el camino a la base e incrementar numCaminos en 1

PCB. Caminos básicos

Veamos el siguiente ejemplo:

```
01 i=1 j=1
02 while not eof( a )
03     lee arrayA[i]; i = i + 1
04 while not eof( b )
05     lee arrayB[j]; j = j + 1
06 regresa
```

Caminos básicos (método simplificado)

1-2-4-6 Primera excepcion

1-2-3-2-4-6 segundo con la primera entrando y el segundo excepción

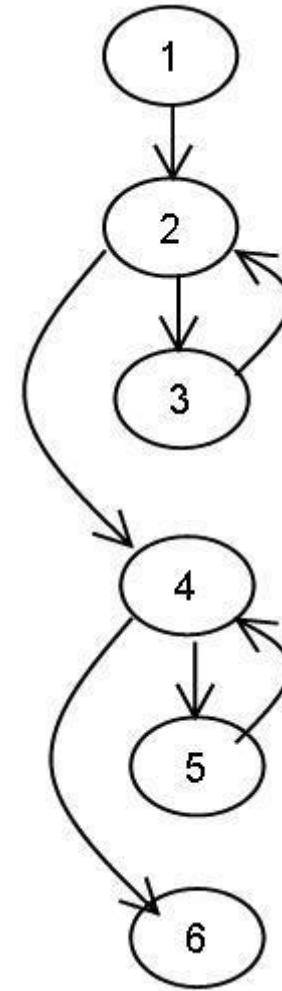
1-2-4-5-4-6 tercera con excepcion y entrando

Caminos básicos (método general)

1-2-3-2-4-5-4-6 Camino normal

1-2-4-5-4-6 Camino excepcion, camino normal

1-2-3-2-4-6 camino normal, camino excepcion



PCB. Generación de casos de prueba

Los caminos básicos identificados en el grafo de control se usan de dos formas durante las pruebas:

- Como guía para verificar que todos los caminos han sido recorridos al menos una vez al ejecutar casos de prueba generados con otro método: Mediante una herramienta de trazado de la ejecución de un programa se irán marcando los caminos recorridos; si todos los caminos básicos están cubiertos por los casos de prueba disponibles, no será necesario crear más.
- Para generar casos de prueba nuevos: Los caminos básicos identificados pueden emplearse para completar los casos de prueba para caminos no recorridos o bien para generar los casos de prueba desde un inicio. Esta forma tiene sus dificultades, ya que a veces existen caminos para los cuales no es posible generar un caso de prueba; generalmente esto indica un defecto en el diseño del programa.

PCB. Generación de casos de prueba

El proceso para generar los casos de prueba parte de que los elementos importantes para definir los caminos se encuentran en las condiciones, ya sea las que correspondan a un if, un while, un case, un for o un repeat. Así pues, el proceso será como sigue:

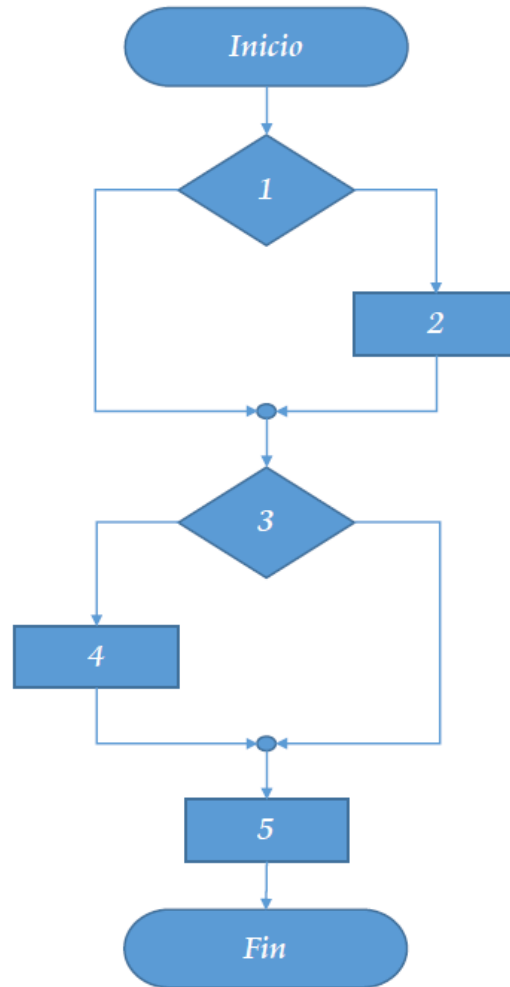
1. Seleccione un camino de los que identificó.
2. Observe la primera condición e identifique las variables que intervienen.
 - 2.1. Si las variables corresponden a parámetros o se leen directamente, entonces esas variables y los valores contra los que se comparan definen los valores de entrada de los casos de prueba.
 - 2.2. Si las variables son internas, habrá que retroceder hasta encontrar las variables externas (parámetros o leídas) de las que dependen y aplicar la misma idea.
3. Si en el camino hay otras condiciones deberá repetirse el proceso, agregando más variables a la parte de entrada del caso de prueba.
4. Cuando no haya más condiciones, agregue la parte de salidas esperadas del caso de prueba, de acuerdo a las especificaciones del programa.
5. Repita el proceso para los otros caminos.

Pruebas de cobertura

Pruebas de cobertura

La prueba de caminos básicos está relacionada con otras pruebas de caja blanca que utilizan la misma estructura del grafo de control. La adecuación de los casos de prueba a menudo se mide con una métrica llamada Cobertura (o cubrimiento). La cobertura es una medida del alcance del conjunto de casos de prueba. El significado de los criterios de cobertura es:

- **Cobertura de proposiciones:** se busca un conjunto de casos de prueba que ejecute cada instrucción por lo menos una vez.
- **Cobertura de ramificaciones:** se busca un conjunto de casos de prueba que ejecute cada instrucción de las ramificaciones en cada dirección por lo menos una vez.
- **Cobertura de condiciones:** Cuando existen condiciones compuestas en el software (es decir expresiones lógicas formadas a partir de expresiones booleanas simples relacionadas mediante operadores AND y OR), se busca un conjunto de casos de prueba que evalúe cada condición simple tanto a resultado verdadero como a resultado falso.
- **Cobertura de caminos (o rutas lógicas):** se basa en que el orden en que se ejecutan las ramas durante una prueba (la ruta seguida) es un factor importante para determinar el resultado de la prueba. Por tanto, se busca un conjunto de casos de prueba que siga cada una de las rutas posibles. Obviamente incluye la cobertura de un conjunto de caminos básicos.



Ejemplos de cobertura:

De proposiciones:

Inicio → 1 → 2 → 3 → 4 → 5 → Fin

De ramificaciones:

Inicio → 1 → 3 → 5 → Fin

Inicio → 1 → 2 → 3 → 4 → 5 → Fin

De caminos:

Inicio → 1 → 3 → 4 → 5 → Fin

Inicio → 1 → 3 → 5 → Fin

Inicio → 1 → 2 → 3 → 4 → 5 → Fin

Inicio → 1 → 2 → 3 → 5 → Fin

Pruebas de cobertura

Se debe determinar un criterio de terminación de las pruebas de cobertura porque las unidades de software contienen demasiadas rutas para permitir una prueba exhaustiva (además, aun cuando fuera posible probar con éxito todas las rutas la corrección no estaría garantizada).

Una regla empírica utilizada a menudo para la terminación de las pruebas de unidad es llegar al 85 ó 90% de la cobertura de ramificación. Por lo común, las pruebas de caja negra basadas en los requisitos funcionales y en la intuición del programador lograrán del 60 al 70% de cobertura de las proposiciones.

Por tanto, bastará con agregar casos de prueba para pruebas de estructura que completen ese 85-90% de cobertura de ramificación. Por supuesto para ello se requiere una herramienta que determine las rutas recorridas por los casos de prueba individuales y por el conjunto acumulado de casos de prueba.

Las pruebas de integración

Las pruebas de integración

Las **pruebas de integración** son pruebas en las que módulos individuales de software son combinados y probados como un grupo. Este tipo de pruebas deberán ejecutarse una vez se haya asegurado el funcionamiento correcto de cada componente implicado por separado, es decir, una vez se hayan ejecutado sin errores las pruebas unitarias de estos componentes. Preceden a las pruebas del sistema.

Aunque desde un punto de vista puramente académico resulta sencillo distinguir las pruebas unitarias de las pruebas de integración, en los proyectos reales la frontera entre estos tipos de pruebas resulta, en muchos casos, difusa, ya que son pocas las ocasiones en que se puede probar una clase o Sujeto en Pruebas (SUT – Subject Under Test) de forma totalmente aislada. En la mayoría de los casos, el SUT debe recibir por parámetros instancias de otras clases, debe obtener información de configuración de un fichero o de una base de datos, o incluso su lógica puede depender de otros componentes del sistema de información. Se podría pensar que aquellos componentes que dependen de otros para su funcionamiento no podrán probarse de manera unitaria.

Las pruebas de integración

Sin embargo, para evitar estos problemas se ha diseñado una serie de artefactos específicos para pruebas que permiten desarrollar y ejecutar pruebas unitarias haciendo creer al SUT que está trabajando en un entorno real:

- **Stubs:** Son componentes auxiliares que proporcionan respuestas predefinidas a las llamadas recibidas durante la prueba, y que generalmente no proporcionarán ningún tipo de respuesta a nada que no se haya programado en la misma. Básicamente, imitan el comportamiento de componentes aún no implementados (implementan su interfaz).
- **Objetos Mock:** Son objetos que están pre-programados con expectativas que, en conjunto, forman una especificación de las llamadas que se espera recibir. El concepto es similar al de los stubs, pero aplicado a la orientación a objetos y con sutiles diferencias.

Las pruebas de integración

En definitiva, y formalmente hablando, si es preciso realizar pruebas unitarias de SUTs que dependen de otros, tendremos que usar algún artefacto específico para pruebas, ya que en otro caso se estarán aplicando pruebas de integración.

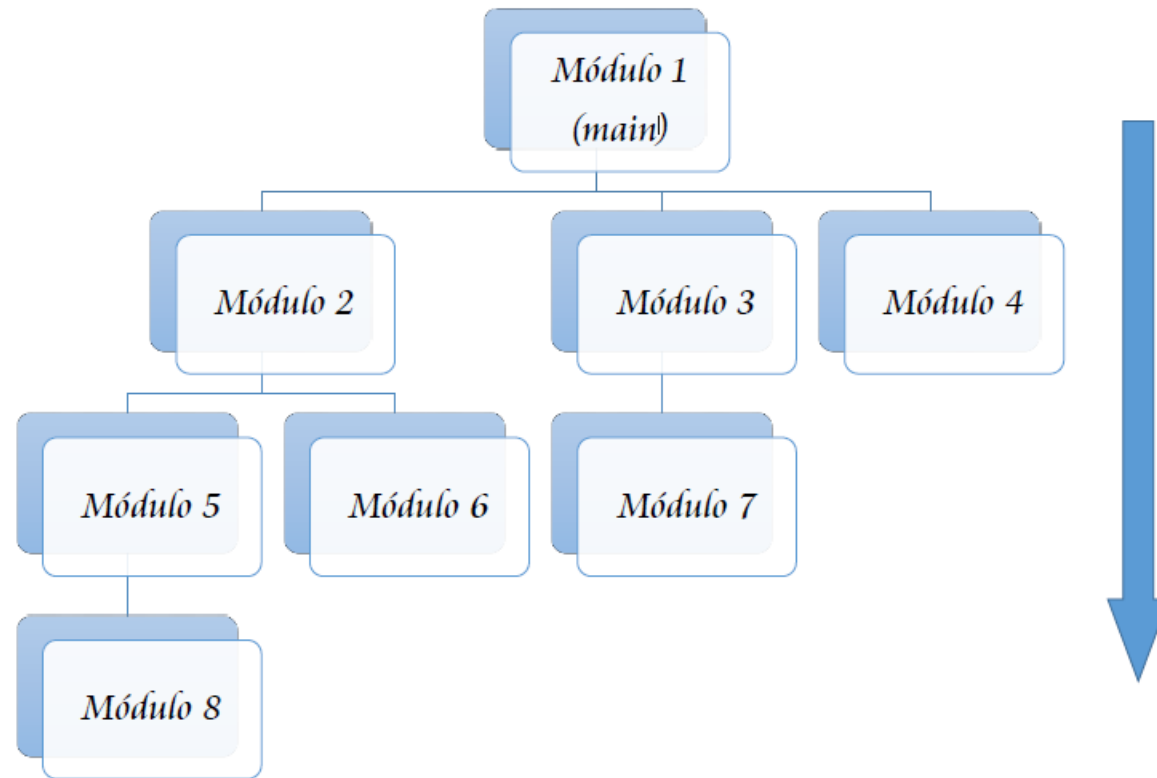
Hay diferentes estrategias para realizar las pruebas de integración:

- Pruebas no incrementales ("Big bang approach"). Se integran todos los componentes y entonces se prueba el sistema como un todo. No es una técnica recomendada pues dificulta el aislamiento de errores.
- Pruebas incrementales. El programa es construido y probado en pequeños incrementos. Esto facilita las pruebas sistemáticas y el aislamiento de errores: la depuración es más fácil, ya que si se detectan los síntomas de un defecto en un paso de la integración hay que atribuirlo muy probablemente al último módulo incorporado.

Las pruebas de integración

A su vez, las pruebas incrementales, pueden ser:

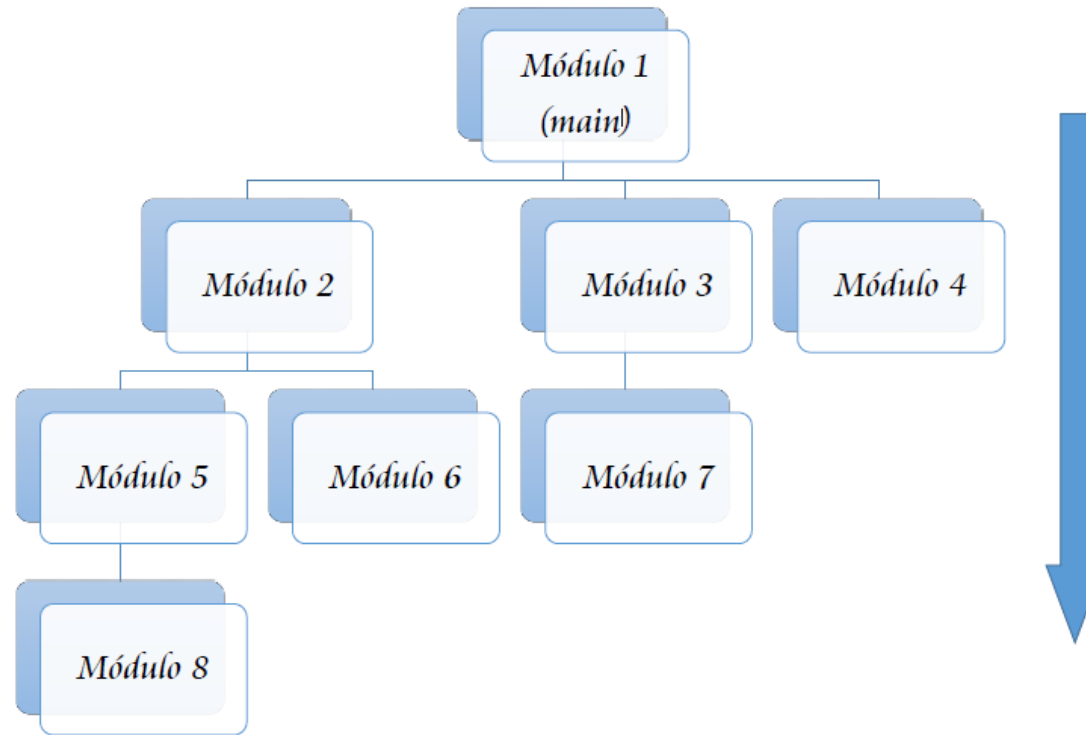
- **Descendentes (“Top-down”)**: La integración se realiza partiendo del programa principal y moviéndonos hacia abajo por la jerarquía de control. Suelen requerir de stubs u objetos Mock.



Las pruebas de integración

A su vez, las pruebas incrementales, pueden ser:

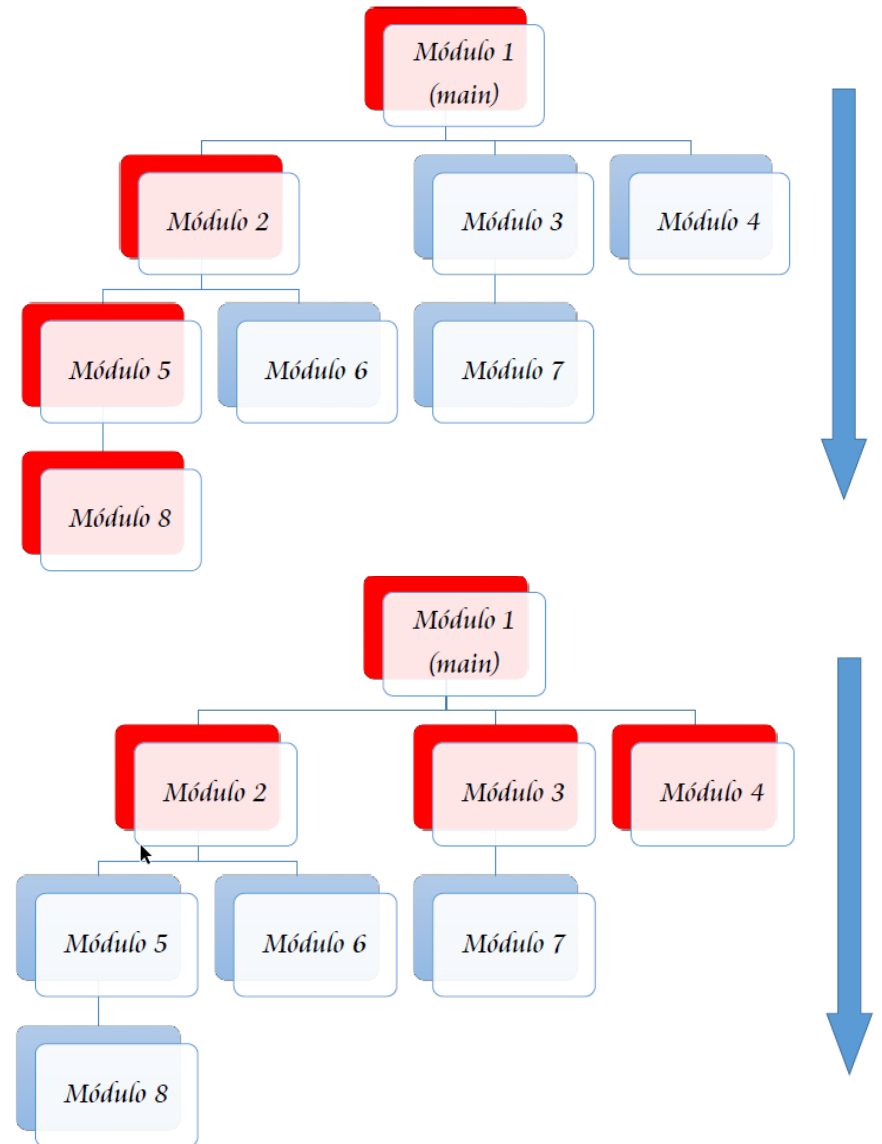
- **Descendentes (“Top-down”)**: La integración se realiza partiendo del programa principal y moviéndonos hacia abajo por la jerarquía de control. Suelen requerir de stubs u objetos Mock.



Hay a su vez dos modos de descenso, en profundidad o en anchura

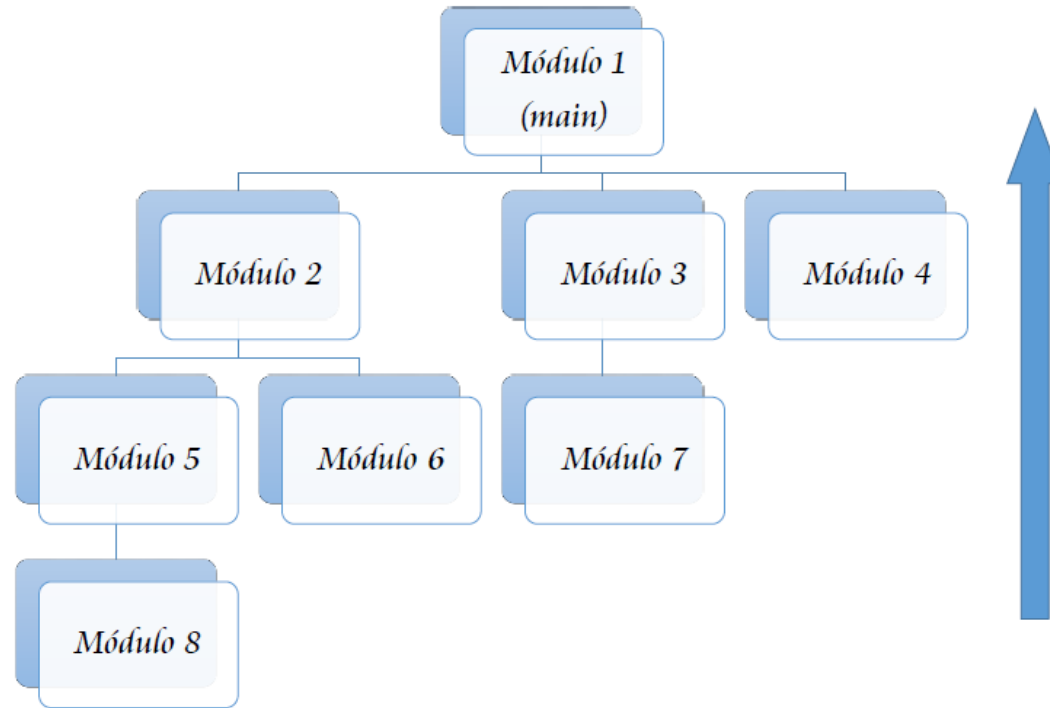
Las pruebas de integración

- **En profundidad ("Depth-first"):** La integración se realiza por ramas. Cada rama suele implementar una funcionalidad específica, por lo permite la prueba de funcionalidades completas.
- **En anchura ("Breadth-first"):** La integración se realiza por niveles moviéndose en horizontal por la estructura de control.



Las pruebas de integración

- **Ascendentes ("Bottom-up"):** La integración se realiza partiendo de módulos atómicos situados en los nodos finales de la jerarquía de control. No suele necesitar de stubs, pero precisa de controladores de prueba o impulsores ("drivers") que simulan la llamada a los módulos, introduciendo los datos de prueba y recogiendo los resultados.



Habitualmente se realiza una integración incremental sándwich ("sandwich testing"), en la que se combinan las estrategias ascendente y descendente.

Técnicas comunes de pruebas de sistema

Técnicas comunes de pruebas de sistema

Para realizar las pruebas de sistema se utilizan diferentes técnicas:

- **Pruebas funcionales.**
- **Pruebas de rendimiento:** Determinan el tiempo de ejecución empleado en las distintas partes del programa, la eficiencia del software y su tiempo de respuesta.
- **Pruebas de resistencia (o de tensión):** Su objetivo es tratar de colapsar el programa. Su forma de hacerlo es muy variada: introduciendo gran cantidad de datos de entrada, limitando los recursos de la aplicación, realizando muchas solicitudes simultáneas, etc.
- **Pruebas destructivas:** Intentan hacer que el software falle. Verifican que el software funcione correctamente incluso cuando recibe entradas no válidas o inesperadas, estableciendo así la solidez de las rutinas de validación de entradas y de la gestión de errores.
- **Pruebas de recuperación:** Se comprueba cómo reacciona el sistema ante un fallo general, y las posibilidades que tiene de solucionar el error. La recuperación del sistema puede realizarla automáticamente el programa, o mediante la intervención del usuario.

Técnicas comunes de pruebas de sistema

- **Pruebas de seguridad:** Se comprueba que tanto los datos como el propio sistema están completamente protegidos frente a todo tipo de agresiones externas e internas. Por ejemplo, se intentará entrar indebidamente en el sistema, bloquear el programa, eliminar los datos vitales, etc. Es decir, el probador actuará como un hacker.
- **Pruebas de usabilidad:** Se comprueba si la interfaz de usuario es fácil de usar y comprender. Tienen que ver principalmente con el uso de la aplicación.
- **Pruebas de accesibilidad:** Se comprueba el cumplimiento de estándares de accesibilidad, tales como la Iniciativa de Accesibilidad Web (WAI) del World Wide Web Consortium (W3C).
- **Pruebas de compatibilidad:** Se comprueba el correcto funcionamiento de la aplicación en diferentes entornos (navegadores web, sistemas operativos o dispositivos), pues el software puede presentar errores dependiendo de dónde se ejecute; p. e., botones o enlaces pueden dejar de funcionar, interfaces graficas pueden descuadrarse, etc.

Técnicas comunes de pruebas de sistema

Para las pruebas de sistema se pueden usar herramientas tales como:

- **Analizadores de cobertura:** fija la extensión de la cobertura obtenida en las pruebas de aceptación.
- **Analizadores de tiempos:** informa del tiempo empleado en varias regiones del código fuente. Normalmente se cumple el principio de Pareto: el programa estará el 80% del tiempo de ejecución en el 20% del código; este 20% deberá tener pues un buen rendimiento.
- **Verificadores de estándares de codificación,** que inspeccionarán el código para ver si se aparta del cumplimiento de estándares.

Técnicas comunes de pruebas de sistema

A veces se usan criterios de terminación de las pruebas basadas en consideraciones diferentes de la cobertura; por ejemplo, cuando se alcanza una tasa de descubrimiento de errores determinada (baja), o cuando se haya descubierto y corregido un número predeterminado de errores (por ejemplo, cuando el 95% de los errores estimados se encuentra y elimina). Las técnicas para estimar errores incluyen modelos predictivos (basados en la estadística), reglas empíricas (basadas en la experiencia previa), trazos de tendencias (gráficos de errores localizados por unidad de tiempo contra tiempo) y siembra de errores (se introducen intencionalmente errores en el código fuente; al cabo de un tiempo la proporción entre el número de errores sembrados descubierto por las pruebas y los errores sembrados no descubiertos ha de ser parecida a la proporción entre el número de errores no sembrados descubiertos y los que quedan por descubrir).

Técnicas comunes de pruebas de sistema

En el proceso de desarrollo del software las pruebas de sistema finalizan con las siguientes pruebas:

- **Pruebas de aceptación.** Consisten en pruebas de sistema que pretenden demostrar que el producto software construido satisface las necesidades establecidas en el documento de requisitos. Suelen realizarlas los grupos independientes y los clientes.
- **Pruebas Alfa (α) y Beta (β).** Son pruebas de caja negra, en las que el que utiliza el programa es el cliente.
 - **Pruebas α .** Un cliente usa el programa en la empresa donde se está programando. El encargado del proyecto revisa su uso, y registra sugerencias, errores y mejoras posibles para el proyecto. Es una prueba en un entorno controlado.
 - **Pruebas β .** Uno o varios clientes usan la aplicación en el lugar de trabajo para el que se ha diseñado. Los clientes son los que van anotando los fallos, sugerencias, mejoras, para que el equipo de desarrollo los solucione. Es una prueba en un entorno no controlado.

Documentación de las pruebas

Documentación de las pruebas

El Plan de Pruebas prescribe varias clases de actividades que se realizarán para demostrar que el producto de programación cumple con sus requisitos; describe los objetivos de las pruebas, los criterios de aprobación, el plan de integración (estrategia, calendario, personal responsable), las herramientas particulares y las técnicas a utilizar, así como los casos reales y los resultados esperados.

El estándar IEEE 829-2008, para la documentación de las pruebas del software, especifica un conjunto de documentos para su uso en las pruebas de software. El estándar especifica el formato de estos documentos, pero no estipula si todos ellos deben ser producidos, ni tampoco incluye ningún criterio sobre el contenido adecuado de los mismos.

Documentación de las pruebas

Los documentos relacionados con el diseño de las pruebas son:

- **Plan de pruebas maestro:** Proporciona un documento global de planificación y gestión de las pruebas (ya sea dentro de un proyecto o en varios proyectos).
- **Plan de pruebas:** Señala 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.
- **Especificación del diseño de pruebas:** Detalla los casos de prueba y los resultados esperados, así como los criterios de aprobación de pruebas.
- **Especificación de caso de prueba:** Especifica los datos de prueba a usar en la ejecución de los casos de prueba identificados en la especificación del diseño de las pruebas.
- **Especificación de procedimiento de prueba:** Detalla cómo ejecutar cada prueba, incluyendo condiciones previas de configuración y los pasos que se deben seguir.

Documentación de las pruebas

Los documentos más significativos relacionados con la ejecución de las pruebas son:

- **Histórico de pruebas:** Provee un registro cronológico de los datos pertinentes sobre la ejecución de las pruebas; por ejemplo, grabación de los casos de prueba realizados, quién los ejecutó, en qué orden, si cada test se superó o falló, etc.
- **Informe de anomalía:** Documenta cualquier incidente ocurrido en las pruebas y que requiera de una investigación. Este documento se denomina deliberadamente informe de anomalía, y no informe de fallo; la razón es que una discrepancia entre los resultados esperados y los reales puede ocurrir por razones distintas de un fallo en el sistema.
- **Informe resumen de pruebas:** Resume los resultados de las actividades de prueba y aporta evaluaciones y recomendaciones basadas en dichos resultados una vez finalizada la ejecución de las pruebas.
- **Informe de pruebas maestro:** Resume los resultados de las actividades de prueba y aporta evaluaciones. Este documento final se utiliza para indicar si el sistema de software bajo prueba ha cumplido los criterios de aceptación definidos por los participantes del proyecto.

Herramienta de testing

JUNIT

JUnit

JUnit es un framework que permite realizar test repetibles (pruebas de regresión), es decir, que puede diseñarse un test para un programa o clase concreta y ejecutarlo tantas veces como sea necesario.



La ventaja es que puede (o mejor, debe) ejecutarse el test cada vez que se modifique o cambie algo del código y verificar si el programa sigue funcionando correctamente tras los cambios.

Para usar JUnit será necesario descargar sus librerías. Podemos hacer uso de Maven para ello. La última versión de JUnit es JUnit5 (Jupiter) para hacer uso de esta versión desde un proyecto Maven, habrá que añadir al pom.xml

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.5.2</version>
  <scope>test</scope>
</dependency>
```



Para poder ejecutar los test con Maven hará falta usar el Maven-surefire-plugin como mínimo a la versión 2.22.0 y la versión de java mínima soportada será la 8.

```
<!-- Este POM no está completo, solo se muestra los ficheros de propiedades y la configuración necesaria para añadir JUnit -->
<properties>
  <java.version>11</java.version>
  <maven.compiler.target>11</maven.compiler.target>
  <maven.compiler.source>11</maven.compiler.source>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.5.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.5.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin><!-- Need at least 2.22.0 to support JUnit 5 -->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M3</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
    </plugin>
  </plugins>
</build>
```


JUnit sirve para crear test unitarios. Los tests unitarios prueban las funcionalidades implementadas en el SUT (System Under Test). Si somos desarrolladores Java, para nosotros el SUT será la clase Java.

Los tests unitarios deben cumplir las siguientes características:

- Principio **FIRST**
 - ✓ **Fast**: Rápida ejecución.
 - ✓ **Isolated**: Independiente de otros test.
 - ✓ **Repeatable**: Se puede repetir en el tiempo.
 - ✓ **Self-Validating**: Cada test debe poder validar si es correcto o no a sí mismo.
 - ✓ **Timely**: ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta hacer primero los test y después la implementación (TDD: Test-driven development), pero es lo suyo para centrarnos en lo que realmente se desea implementar.

- Además, podemos añadir estos tres puntos más:
 - Sólo pruebas los métodos públicos de cada clase.
 - No se debe hacer uso de las dependencias de la clase a probar. Esto quizás es discutible porque en algunos casos donde las dependencias son clases de utilidades y se puede ser menos estricto. Se recomienda siempre aplicar el sentido común.
 - Un test no debe implementar ninguna lógica de negocio (nada de if...else...for...etc)
- Los tests unitarios tienen la siguiente estructura:
 - Preparación de datos de entrada.
 - Ejecución del test.
 - Comprobación del test (assert). No debería haber más de 1 assert en cada test.

Un test Junit nunca jamás se debe utilizar if, while, for... solo puede haber sentencias una detrás de otra

JUnit es un framework Java para implementar test en Java. Se basa en anotaciones:

- **@Test**: indica que el método que la contiene es un test

```
@Test
void testDevuelveTrue() {
    System.out.println("Llamando a testDevuelveTrue");
}
```

- **@RepeatedTest**: Indica que el siguiente método será llamado las veces que la etiqueta recibe como parámetro. En el caso del ejemplo 2.

```
@RepeatedTest(2)
void testRepiteTest() {
    System.out.println("Llamando a testRepiteTest");
}
```

- **@BeforeEach**: Indica que el siguiente método es llamado antes de ejecutar cada una de las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest.

```
@BeforeEach
void MetodoBeforeEach() {
    System.out.println("Llamando a MetodoBeforeEach");
}
```

- **@AfterEach**: Indica que el siguiente método es llamado después de ejecutar cada una de las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest

```
@AfterEach
void MetodoAfterEach() {
    System.out.println("Llamando a MetodoAfterEach");
}
```

- **@BeforeAll**: Indica que el siguiente método es llamado una sola vez en toda la ejecución del programa. A continuación, son llamadas las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest

```
@BeforeAll
static void MetodoBeforeAll() {
    System.out.println("Llamando a MetodoBeforeAll");
}
```

- **@AfterAll**: Indica que el siguiente método es llamado una sola vez en toda la ejecución del programa. Antes habrán sido llamadas todas las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest.

```
@AfterAll
static void MetodoAfterAll() {
    System.out.println("Llamando a MetodoAfterAll");
}
```

- **@Disabled**: evita la ejecución del test. No es muy recomendable su uso porque puede ocultar test fallidos. Si dudamos si el test debe estar o no, quizás borrarlo es la mejor de las decisiones.

```
@Test
@Disabled
void testDesactivado() {
    System.out.println("Desactivada la llamada a testDesactivado");
}
```

tambien funciona con repited test, pero este comando no se puede tener para siempre hay que arreglar el código o si ya no se necesita el test se borra.

- **@ParameterizedTest** y **@ValueSource**. Permite pasar al método de prueba parámetros a utilizar en la ejecución. Se lanza una vez por parámetro pasado.

```
@ParameterizedTest
@ValueSource(strings = {"HOLA", "ADIOS"})
void testParameterizedTest(String sMiInput) {
    System.out.println("Llamando a testParameterizedTest " +
        sMiInput );
}
```

- **@ParameterizedTest** y **@CsvSource**. @ValueSource sólo permite el paso de un parámetro al método de prueba. Si necesitamos pasar varios se utiliza la etiqueta CsvSource. Se lanza una vez por cada tupla de parámetros pasada.

```
@ParameterizedTest
@CsvSource({"HOLA,1", "ADIOS,2"})
void testParameterizedIntTest(String a, int b) {
    System.out.println("Llamando a testParameterizedIntTest : "
        + a + " --- " + b);
}
```

Las condiciones de aceptación del test se implementan con los asserts. Se encuentran en la clase `Assertions` y los más comunes son los siguientes:

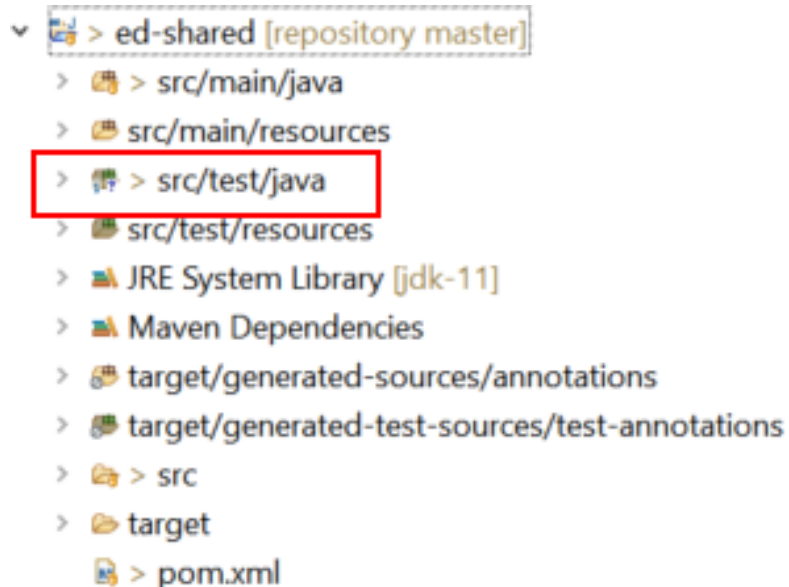
- `assertTrue/assertFalse` (condición a testear): Comprueba que la condición es cierta o falsa.
- `assertEquals/assertNotEquals` (valor esperado, valor obtenido). Es importante el orden de los valores esperado y obtenido.
- `assertNull/assertNotNull` (object): Comprueba que el objeto obtenido es nulo o no.
- `assertSame/assertNotSame(object1, object2)`: Comprueba si dos objetos son iguales o no.

JUnit

En JUnit además se ha añadido una nueva anotación que sirve para sobrescribir el nombre del test en la suite de JUnit `@DisplayName("")`.

Hay muchas más anotaciones y posibilidades, pero para una iniciación en JUnit con estas nos basta.

Por tanto, para empezar, creando una clase de test, deberemos crear una estructura para añadir los test, como estamos usando Maven, los proyectos por defecto con nave te generan un paquete para incorporar los tests.



Una vez creado el paquete donde se incorporarán todos los test, estos deberán estar en la misma ruta que la clase, es decir si tenemos una clase que queremos probar que se encuentra en src/main/java/ed/ut3/example/Calculadora.java el test debe ser creado en la ruta src/test/java/ed/ut3/example/CalculadoraTest.java, nótese que el único cambio ha sido test por java. Una vez hecho esto crearemos una clase de test, con el mismo nombre que la clase a probar, pero acabada en Test.

```
package ed.ut3.example;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

public class CalculadoraTest {

    Calculadora calculadora = new Calculadora();

    @DisplayName("Test suma correcto")
    @Test
    void suma_ok_test() {
        int suma = calculadora.suma(2,3);
        Assertions.assertEquals(5, suma);
    }
}
```


La forma de verificar que se lanza una excepción ha cambiado con JUnit5, y ahora se usa una lambda para su correcta verificación.

```
@DisplayName("Test divide por cero")
```

```
@Test
```

```
void divide_by_zero_test() {
```

```
    Exception e = Assertions.assertThrows(ArithmeticException.class, () ->
        calculadora.division(2, 0));
```

```
    Assertions.assertEquals("Division by zero", e.getMessage());
```

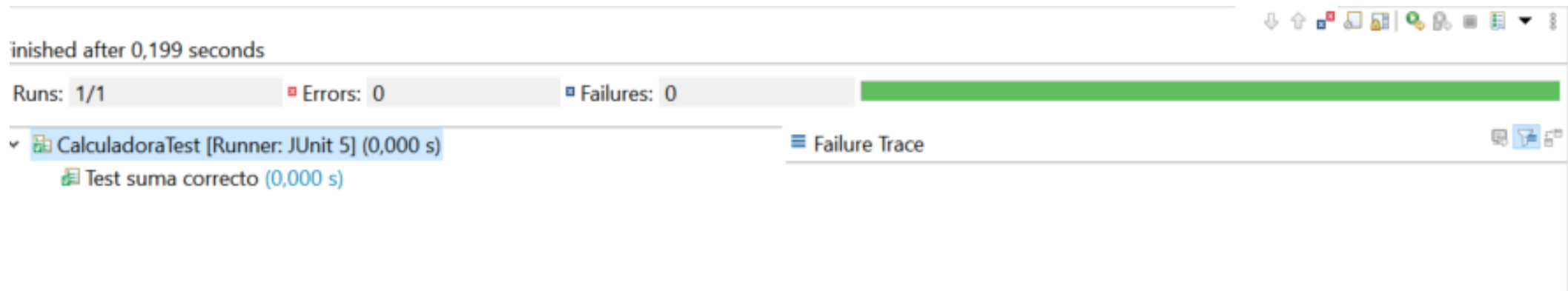
```
}
```

Programación funcional, llamada a la excepción, va siempre detrás de ()->

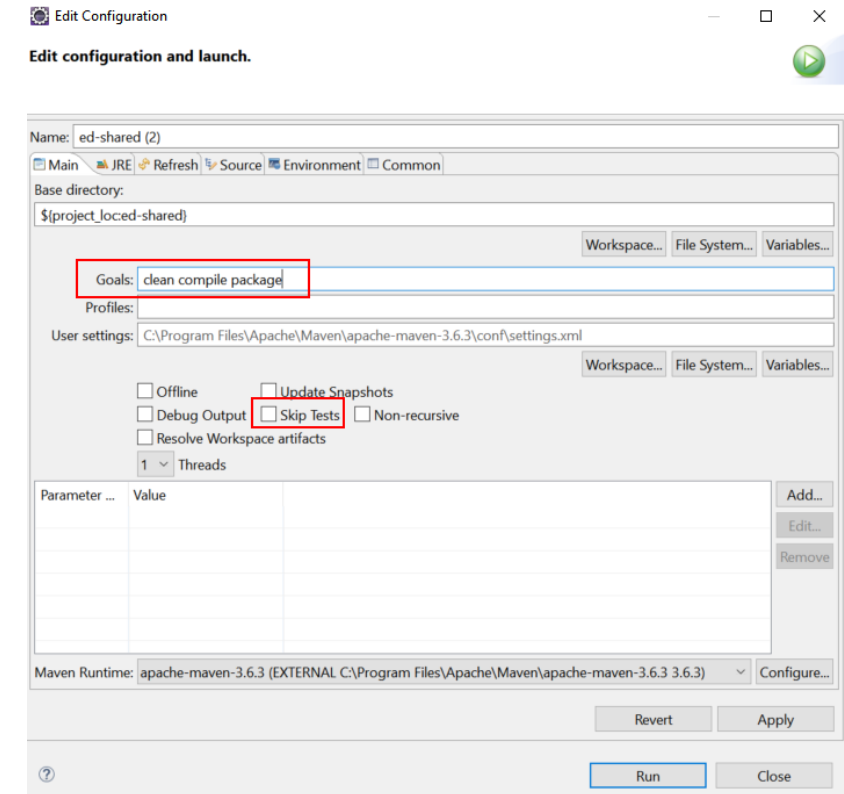
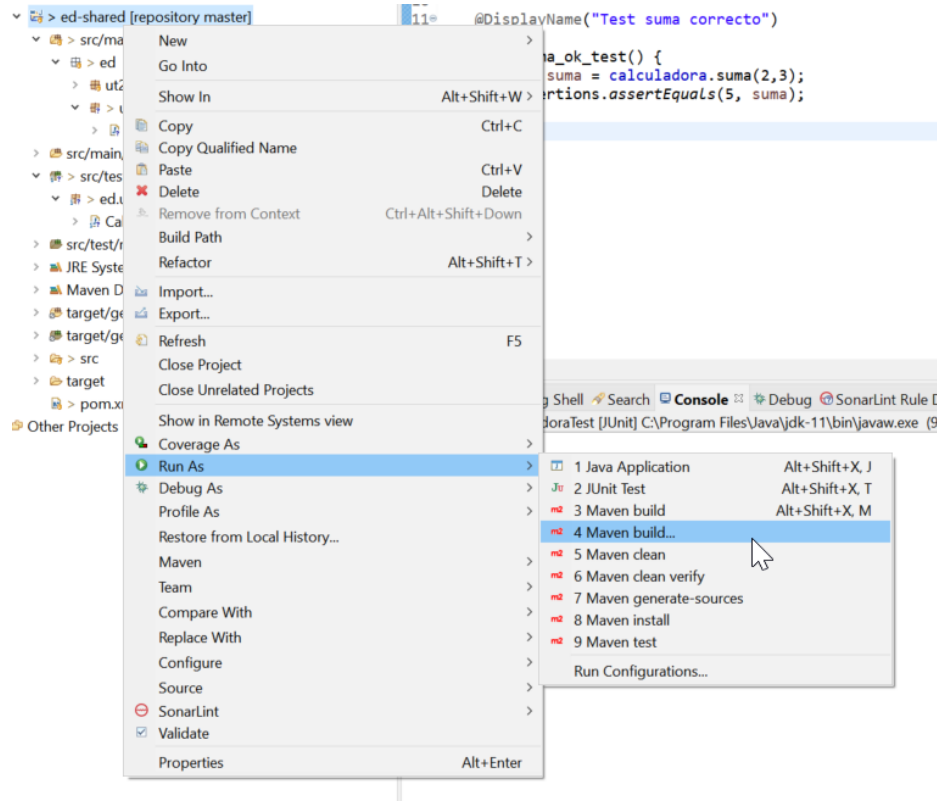
ESTO ES PARA CUANDO HAY UNA DIVISION ENTRE 0

JUnit

Una vez creado el test, para ejecutarlo podemos usar el IDE eclipse, para ello pulsamos con el botón derecho en la clase y con el ratón pulsamos en Run As > Junit Test. Si queremos hacer un debug, pulsaremos en Debug as > Junit Test



Por otro lado, como estamos con un proyecto Maven, podemos ejecutar todos los tests con Maven, simplemente ejecutando un clean compile package



Obsérvese que Skip Tests está desmarcado. Si quisiéramos saltarnos el paso de los tests, deberíamos marcar esta opción.

Actividad 3. Crea un método que reciba por parámetro un valor entero.

- Si es 0, devuelve null,
- Si es 1, devuelve "Hola"
- Si es 2, devuelve "Mundo"
- Si es 3, devuelve "Hola Mundo"

Crea los test JUnit

Actividad 4. Crea un proyecto de una calculadora con Maven y JUnit. La calculadora deberá hacer las operaciones de sumar, restar, multiplicar y dividir. Si se intenta dividir por cero deberá dar un ArithmeticException.

Cobertura de código

La **cobertura de código (coverage)** es una medida porcentual en las pruebas de software que mide el grado en que el código fuente de un programa ha sido comprobado. Es comúnmente utilizada en pruebas de caja blanca, como las pruebas unitarias, en las que sí se tiene acceso al código y estructura del software que se está testeando.

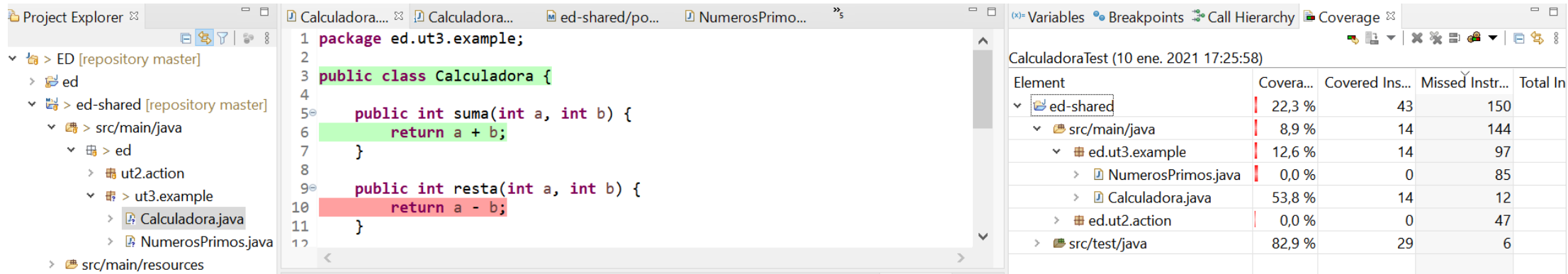
Hay herramientas que te indican la cobertura de código, nosotros usaremos la integrada en el propio IDE Eclipse para verificar la cobertura de código.

Con Maven tenemos algunos plugin para obtener la cobertura de código y prácticamente con cualquier IDE tienes herramientas que te indican la cobertura, nosotros usaremos la integrada en el propio IDE Eclipse para verificar la cobertura de código.

Cobertura de código

Para ello haremos lo siguiente, en la clase de Test pulsaremos con el botón derecho y seleccionaremos la opción Coverage As > Junit Test. También podemos pulsar en Run > Coverage As > Junit Test.

Con esto podremos ver el porcentaje de cobertura de código que tenemos y visualmente que código está testeado (verde) y que código no está testeado (rojo)



CalculadoraTest (10 ene. 2021 17:25:58)

| Element | Covera... | Covered Ins... | Missed Instr... | Total In |
|--------------------|-----------|----------------|-----------------|----------|
| ed-shared | 22,3 % | 43 | 150 | |
| src/main/java | 8,9 % | 14 | 144 | |
| ed.ut3.example | 12,6 % | 14 | 97 | |
| NumerosPrimos.java | 0,0 % | 0 | 85 | |
| Calculadora.java | 53,8 % | 14 | 12 | |
| ed.ut2.action | 0,0 % | 0 | 47 | |
| src/test/java | 82,9 % | 29 | 6 | |

La **cobertura de código** nos da un índice de que código está probado, pero no nos dice si las pruebas realizadas son las correctas o no, para ello habría que profundizar más en temas como mutation tests y diseño de pruebas.

Actividad 5. Crea un método que reciba por parámetro un valor entero perteneciente a un año y te diga si ese año es bisiesto.

- Es bisiesto si es divisible entre 4.
- Pero no es bisiesto si es divisible entre 100.
- Pero sí es bisiesto si es divisible entre 400.

Crea los test JUnit.

Actividad 6. Crea un programa que reciba dos parámetros enteros y devuelva todos los números primos entre esos dos valores. Un número primo es un número que sólo es divisible por sí mismo y por uno. Por definición ni el 0 ni el 1 es un número primo. Crea los test Junit para el programa.

Calidad del software

Calidad del software

La calidad es un tema que, desde hace años, tiene una importancia en el mundo de la comercialización de productos. El mercado actual es muy competitivo y la calidad es uno de los aspectos diferenciales que hace que un producto triunfe o fracase. Basta citar a Blackberry, empresa que, en medio de la vorágine de un mercado tan competitivo como el de la telefonía móvil, cometió varios errores tanto estratégicos como técnicos que la relegaron a un plano poco significativo.

Dadas sus características, garantizar la calidad del software es un proceso mucho más difícil que el de otro producto, dado que un proceso industrial es más fácil de testear que el proceso de desarrollo de software. El software tiene que estar libre de defectos y de errores y también tiene que adecuarse a los parámetros con los cuales se ha diseñado y desarrollado.

Las aplicaciones informáticas están presentes en multitud de ámbitos. Existe software en dispositivos que nadie puede imaginarse (lavadoras, televisiones, aires acondicionados, etc.).

Calidad del software

En los años noventa, se vivió una crisis del software. Fueron en esos años en los que la calidad y el proceso de desarrollo no tenían mucha importancia en los que se vivieron las consecuencias de desarrollar un software con poca profesionalidad en muchos casos. Algunas características de esta crisis fueron:

- Calidad insuficiente del producto final. Muchos de los errores tenían su base en un análisis pobre con una poca comunicación con el cliente.
- Estimaciones de duración de proyectos y asignación de recursos inexactas. Con el problema que ello conlleva.
- Escasez de personal cualificado en un mercado laboral de alta demanda. Algunos programas no estaban desarrollados bajo el paradigma de la programación estructurada. No tenían una estructura racional ni lógica, con lo cual los errores se multiplicaban y el mantenimiento era un suplicio.
- Tendencia al crecimiento del volumen y complejidad de los productos. En algunos casos, dichos desarrollos complejos estaban poco probados, pobremente documentados, etc.

Calidad del software

Con el tiempo, se ha constatado que la calidad no se mide solamente por unos parámetros de funcionamiento, sino que hay otros aspectos que son importantes, como el soporte, es decir, el respaldo organizacional que tiene un producto como la formación, la asistencia a problemas inesperados y el mantenimiento permanente y efectivo.

Para valorar dicha calidad, se lleva a cabo la evaluación y el rendimiento de las aplicaciones. Las mediciones de rendimiento de un software pueden estar orientadas hacia el usuario (tiempos de respuesta) u orientadas hacia el sistema (uso de la CPU). Son medidas típicas del rendimiento diferentes variables de tiempo (tiempo de retorno, tiempo de respuesta y tiempo de reacción), la capacidad de ejecución, la carga de trabajo, la utilización, etc.

Calidad del software

Para evaluar el software, es necesario contar con criterios adecuados que permitan analizar el software desde diferentes puntos de vista.

Las **pruebas de carga** se realizan sobre el sistema simulando una serie de peticiones esperadas o un número de usuarios esperado trabajando de forma concurrente, realizando un número de transacciones determinado. En estas pruebas, se evalúan los tiempos de respuesta de las transacciones. Generalmente, se realizan varios tipos de carga (baja, media y alta) para evaluar el impacto y poder graficar el rendimiento del sistema.

Otro tipo de pruebas bastante útiles son las **pruebas de estrés** en las que la carga va elevándose más y más para ver cómo de sólida es la aplicación y cómo se maneja ante un número de usuarios y transacciones extremos.

También existen otros tipos de pruebas como las **pruebas de estabilidad** donde se somete de forma continuada al sistema a una carga determinada o bien pruebas de picos donde el volumen de carga va cambiando.

Calidad del software

Se definen los criterios de calidad (o factores de calidad) de un software al principio de un proyecto y dichos criterios siguen teniéndose en cuenta durante toda su vida. No puede existir ningún criterio o factor de calidad que no pueda medirse. Algunos criterios de calidad pueden ser los siguientes:

- Número de errores por un número determinado de líneas de código.
- Número de tiempo que la aplicación estará dando servicio.
- Número medio de revisiones realizadas a una función o módulo de programa.

Generalmente, para evaluar los criterios de calidad, se realizan RTF o revisiones técnicas formales.

Calidad del software

Los criterios o factores de calidad, como no podía ser de otra forma, se establecen mediante métricas o medidas. Véanse algunas de las métricas de calidad más utilizadas:

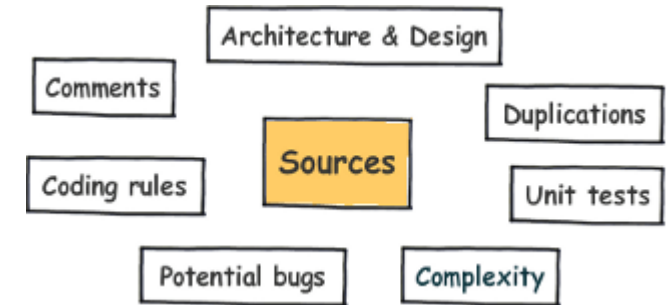
1. Tolerancia a errores. Mide los efectos que tiene un error sobre el software en conjunto. El objetivo es que no haya errores, pero, si los hay, que sus efectos sean limitados.
2. Facilidad de expansión. Mide la facilidad con la que pueden añadirse nuevas funcionalidades a un software concreto. Cuanto más fácil sea de ampliar, mejor.
3. Independencia de plataforma del hardware. Es sabido que un programa en Java es de los más independientes que existen. Cuanto mayor sea el número de plataformas donde pueda ejecutarse un software, mejor.
4. Modularidad. Número de componentes independientes de un programa.
5. Estandarización de los datos. Se evalúa si se utilizan estructuras de datos estándar a lo largo de un programa.

Calidad del código

Calidad del código

Define mediante métricas y reglas cómo de bien escrito está el código fuente que hemos creado. Vamos a diferenciar siete grandes puntos que serán los ejes de la calidad del código software.

1. Tener líneas de código duplicado ("Duplicated code").
2. No respetar los estándares de codificación y las mejores prácticas establecidas ("Coding standards").
3. Tener una cobertura baja (o nula) de pruebas unitarias, especialmente en partes complejas del programa ("Unit tests").
4. Tener componentes complejos y/o una mala distribución de la complejidad entre los componentes ("Complex code").
5. Dejar fallos potenciales sin analizar ("Potential bugs").
6. Falta de comentarios en el código fuente, especialmente en las APIs públicas ("Comments").
7. Tener el temido diseño spaghetti, con multitud de dependencias cíclicas ("Design and architecture").



Estos siete ejes son los mismos que utiliza la herramienta Sonar para evaluar el código.

SonarQube (conocido anteriormente como Sonar) es una plataforma para evaluar código fuente. Es software libre y usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.



Funciones

- Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, errores potenciales, comentarios y diseño del software.
- Aunque pensado para Java, acepta otros lenguajes mediante extensiones.
- Se integra con Maven, Ant y herramientas de integración continua como Atlassian Bamboo, Jenkins y Hudson.

Calidad del código

Sonar permite además realizar un análisis comparativo de las métricas en el tiempo, lo cual permite disponer de una línea de tiempo y comparar la salud del proyecto de un sprint a otro, hablando en términos de metodologías ágiles, iteraciones y entrega continua.

Para establecer un nivel de calidad mínimo en nuestros proyectos podemos definir el cumplimiento de, al menos, un umbral de dichas métricas.