CONTORNOS DE DESENVOLVEMENTO

C.S. Desarrollo de aplicaciones multiplataforma/curso web 20...-20...

Módulo: Desarrollo del esquema

Soy Fernando Wirtz. La Coruña

Wirtz I.E.S.

IES Fernando Wirtz Suárez. A Coruña.
Módulo: Contornos de Desenvolvemento
Profesor: Fernando Rodríguez Diéguez

XUNTA DE GALICIA
CONSELLERÍA DE EDUCACIÓN
E ORDENACIÓN UNIVERSITARIA

Unión Europea
Fondo Social Europeo

Catál
ogo

Materiales complementarios y vídeos:

https://wirtzIDE.blogspot.com/

Fernando Rodríguez Dieges
rdf@fernandowirtz.com

Versión 2022-07-29

1. Introducción a las pruebas de software

1.1 Introducción

Calidad del software

La calidad del software se refiere al grado en que el software cumple con los requisitos funcionales establecidos con el cliente, los estándares de desarrollo claramente documentados y la funcionalidad esperada de cualquier software desarrollado profesionalmente. De esta definición se deduce que:

- No habrá calidad si el software no cumple con los requisitos iniciales.
- Los estándares de desarrollo guían cómo se desarrolla el software, por lo que si el software no cumple con estos estándares no habrá calidad.
- Si el software tiene errores importantes, o no es fácil de usar, o es difícil de mantener, no es calidad.

Debido a la naturaleza del software, la determinación de la calidad es muy complicada, ya que no es un elemento tangible como otros productos industriales, pero debe ser libre de errores, se pueda medir y se pueda verificar su proceso de desarrollo.

Las comprobaciones periódicas se llevan a cabo a lo largo del desarrollo del software, generalmente coincidiendo con los hitos del proyecto que deben permitir:

Verifique que el producto esté construido correctamente.

- Verificar que se está construyendo el producto correcto, es decir, lo que realmente quiere el usuario.

> Misión 3.1. Utilice tus propias palabras o ejemplos para ilustrar los 7 principios de pruebas de software.

1.2 Normas y certificaciones

La estandarización o estandarización es el proceso de elaboración, difusión, aplicación y perfeccionamiento de las reglas utilizadas en diversas actividades científicas, industriales o económicas para ordenarlas y perfeccionarlas. Según la definición de la ISO (Organización Internacional de Normalización), la normalización se refiere a la actividad que tiene por objeto establecer disposiciones destinadas a ser comunes y reutilizadas, frente a problemas reales o potenciales, con el fin de obtener un nivel óptimo de orden en un contexto dado, que puede ser técnico, político o económico. A normalización permite:

- Simplificar, es decir, reducir los tipos de aeronaves, manteniendo sólo los más necesarios.
- Unificar para permitir o intercambio a nivel internacional.
- Especificar mediante la creación de un lenguaje claro y preciso para evitar errores de identificación.

La certificación es la acción realizada por una entidad reconocida, independiente del interesado, mediante la cual se demuestra que una organización, producto, proceso o servicio cumple con los requisitos mínimos establecidos en una norma o especificación técnica. Permite a la empresa diferenciarse de otros lugares y tener una ventaja en el mercado, al tiempo que demuestra que sigue ciertos estándares de calidad. La solicitud de certificación es voluntaria, implica ciertos costos y requiere renovaciones periódicas.

ISO

La Organización Internacional de Normalización o ISO (http://www.iso.org) nació el 23 de febrero de 1947 como una organización no gubernamental que promueve el desarrollo de normas internacionales de fabricación, comercio y comunicación para todas las ramas de la industria. Su función principal es buscar la estandarización internacional de productos y normas de seguridad para empresas u organizaciones. Los estándares que produce son conocidos como estándares ISO. Con sede en Ginebra, Suiza, pero formada por la integración de organismos nacionales de normalización de varios países. Los contenidos de los estándares están protegidos por derechos de autor y para que el público en general pueda acceder a ellos, debe comprar cada documento en su sitio web oficial.
La serie de normas ISO 9000 cubre todos los aspectos de la gestión de la calidad e incluye algunos de los estándares ISO más conocidos. Las normas proporcionan orientación y herramientas para las empresas y organizaciones que desean garantizar que sus productos y servicios satisfagan los requisitos de los clientes y que su calidad mejora constantemente. Este hogar incluye estándar:

ISO 9001:2015-especifica los requisitos para un sistema de gestión de calidad, incluyendo una fuerte orientación al cliente, motivación y participación de la alta dirección, un enfoque de procesos y la mejora continua.

- ISO 9004:2018 – Centrarse en cómo hacer sostenible el sistema de gestión de la calidad, es decir, ofrecer propuestas de mejora.

ISO 19011:2018-Propone directrices para auditorías internas y externas de sistemas de gestión de calidad.

IEC

La Comisión Electrotécnica Internacional o IEC (http:// www. IEC. ch/) o IEC (International Electrotechnical Commission) es un organismo de normalización en el campo de la tecnología eléctrica, electrónica y conexas. Fue fundada en 1904 durante el Congreso Internacional de Electricidad en St. Louis (Estados Unidos). Actualmente tiene su sede en Ginebra, Suiza, y está compuesta por organismos nacionales de normalización en áreas designadas por los Estados miembros. En 1938, la institución publicó el primer diccionario internacional (International Electrotechnical Vocabulary) con el objetivo de unificar la terminología eléctrica, un esfuerzo que se mantuvo a lo largo del tiempo, haciendo del International Electrotechnical Vocabulary una referencia importante para las empresas del sector.

Muchos estándares se desarrollan conjuntamente con la ISO y se conocen como estándares ISO/ IEC. Por ejemplo, la norma ISO/IEC 90003:2014 proporciona orientación para las organizaciones y empresas en la aplicación de la norma ISO 9001:2008 en la adquisición, suministro, desarrollo, operación y mantenimiento de software y servicios de soporte relacionados.

Enor

La Asociación Española de Normalización y Certificación o AENOR es el único organismo acreditado en España para desarrollar tareas de normalización y certificación y representa a España en organizaciones europeas (CEN, CENELEC y ETSI) e internacionales (ISO e IEC). Es una organización privada sin fines de lucro fundada en 1986. En su página web oficial (http:// www. aenor. es) ofrece información sobre el proceso de certificación, la publicación de nuevas normas, la reunión....

Las normas a las que se adapta o desarrolla, las normas UNE, indican cómo debe ser el producto o cómo debe funcionar el servicio para que sea seguro y responda a las expectativas de los consumidores. Por ejemplo, la norma ISO 9000 se adoptó sin modificaciones como norma europea (serie EN 29000) y norma española (serie UNE 66-90). AENOR ofrece uno de los catálogos más completos, con más de 28.900 documentos normativos para soluciones efectivas.

Los certificados AENOR son muy valorados, no solo en España, sino también a nivel internacional, con certificados expedidos en más de 60 países. AINO Corporation se encuentra entre los diez principales organismos de certificación del mundo.

IEEE

IEEE (español lido i-e-cubo, latinoamericano i-triplo- e) corresponde al Instituto de Ingenieros Eléctricos y Electrónicos, en español Instituto de Ingenieros Eléctricos y Electrónicos, es una asociación técnica profesional global dedicada a la estandarización y otros trabajos. Nació en 1884 , pero adoptó el nombre IEEE en 1963 y actualmente es la mayor asociación internacional sin fines de lucro de profesionales de las nuevas tecnologías como ingenieros eléctricos, electrónicos, informáticos, informáticos, biomédicos, telecomunicaciones, mecatrónicos, etc. Según el propio IEEE, su trabajo es promover la innovación, el desarrollo e integración, el intercambio y la aplicación de avances en tecnologías de la información, electrónica y ciencias en general, en beneficio de la humanidad y de los mismos profesionales. Colaboran con la ISO y la IEC en temas comunes. El sitio web oficial es http://www.ieee.org. La División de Lengua Española del IEEE fue reconocida en abril de 1968 en el ámbito del Distrito 8, con su sitio web oficial: http:// www. ieespain. org/

Algunos estándares relacionados con el software son:

IEEE 730. Programa de garantía de calidad del software.
IEEE 829. Documentación de pruebas de software.

IEEE 982.1, 982.2. Dicionario estándar de medidas para producir software fiable.
IEEE 1008. Pruebas unitarias de software.
IEEE 1012. Verificación y validación de software.
IEEE 1028. Revisión de software.
IEEE 1044. Clasificación estándar de anomalías de software.
IEEE 1061. Estándares para métodos de medición de la calidad del software.
IEEE 1228. Programa de seguridad del software.

1.3 Pruebas

No es práctico probar completamente el software porque no se pueden probar todas las posibilidades Por su funcionamiento, incluso en programas pequeños y simples. El objetivo de la prueba es Detecta defectos, errores o [1]errores en el software, por lo que si se encuentra un defecto. Se trata de una actividad ex post, que detecta problemas en el software, en lugar de prevenirlos.

El proceso de prueba comienza con la generación de un plan de prueba basado en la documentación sobre el proyecto y la documentación sobre el software bajo prueba. A partir del plan anterior, refine las pruebas específicas y ejecute con los casos de prueba para obtener los resultados. Los resultados pueden indicar la presencia de errores, en cuyo caso los errores deberán registrarse e informarse para corregirlos y volver a ejecutar la prueba.
Los planes de prueba no deben desarrollarse suponiendo que no hay defectos; Tienes que asumir que siempre están ahí. Las pruebas deben planificarse y diseñarse sistemáticamente para poder detectar el máximo número y variedad de defectos con el mínimo gasto de tiempo y esfuerzo. Son una tarea tan creativa o más creativa que el desarrollo de software y deben evitar las pruebas no documentadas o bien diseñadas, como las que se realizan en tiempo de ejecución

. Las pruebas deben centrarse en dos objetivos: comprobar si el software no está haciendo lo que debería hacer y comprobar si el software está haciendo lo que no debería hacer, es decir, si está causando efectos secundarios indeseables; Olvidarse de este último gol es común.

Tipo de prueba

Existen diferentes formas de clasificar las pruebas que no se excluyen mutuamente:

- Funcional o no funcional. El primero está diseñado para verificar cierto requisito funcional del software, mientras que el segundo no. Ejemplos de pruebas no funcionales: aquellas que permiten indicar el esfuerzo requerido para aprender a manejar dicho software, analizar código y localizar fallas, apoyar cambios y facilitar pruebas relacionadas con dichos cambios, o migrar software a otro entorno.
- Pruebas manuales y automáticas. El primero se llevará a cabo de acuerdo con un plan detallado y estructurado, pero sin procesos automatizados para ejecutarlo. Este último tiene un proceso automático que se puede repetir convenientemente varias veces si lo desea.

[1] bug se traduce como error y se utiliza en informática para indicar una vulnerabilidad o

Pruebas dinámicas y estáticas. El primero se ejecuta mientras el software se ejecuta, el segundo no.

- Pruebas de caja negra y caja blanca: Las pruebas de caja negra son pruebas que utilizan la interfaz externa de una aplicación para realizar pruebas sin preocuparse por su implementación. La clave aquí es verificar aquellos Resultado Datos Según la entrada que recibe, la aplicación se ejecuta como se espera. La prueba de caja blanca (White Box Testing) es la prueba de una aplicación desde dentro, siguiendo el camino del flujo del programa basado en su lógica y código.
- Realizar pruebas tipo caja negra sin conocer la estructura del sistema o el funcionamiento interno. Al realizar este tipo de prueba, se conocen las entradas apropiadas que la aplicación debe recibir, y sus salidas correspondientes, pero no se conoce por qué proceso la aplicación obtiene estos resultados.
- Por el contrario, las pruebas de caja blanca, que analizan y prueban directamente el código de la aplicación.

Hay algunas acciones que son realizadas en el código por ciertas herramientas para detectar defectos, y estas acciones no son rigurosamente probadas, como:

- Validación de código: permite asegurarse de que el código cumple con un determinado estándar del lenguaje, por ejemplo, si una variable está inicializada o no.
- Depuración (Debugging): permite detectar código que produce resultados erróneos en tiempo de ejecución. El entorno de desarrollo tiene herramientas de depuración automática que permiten que el código se ejecute de una manera controlada por el programador y permite que el programador ejecute línea por línea, o establezca puntos de interrupción para que la ejecución se detenga en ese punto; En tiempo de ejecución, el programador puede examinar una variable, expresión o pila y modificarla para ver el comportamiento en tiempo de ejecución.

Análisis de líneas de código: permite detectar, por ejemplo: código duplicado en varios sitios, comprobar código documentado, lo que facilitará la tarea de generar documentos automáticamente (Javadoc en Java), encontrar líneas de código comentadas que ocupen espacio incluso si no están en ejecución y pueden eliminarse mediante un sistema de control de versiones.

Pruebas unitarias

Son pruebas funcionales realizadas por técnicos que permiten detectar problemas en módulos individuales y básicos, tales como clases o métodos de clases. Su trabajo se centra en la implementación de la lógica del módulo siguiendo la estructura del código (tecnología de caja blanca) y la implementación de las funciones que el módulo debe realizar satisfaciendo las entradas y salidas (tecnología de caja negra).

El porcentaje de código probado a través de una prueba unitaria se denomina cobertura Software.

El entorno de desarrollo proporciona herramientas para diseñar y ejecutar pruebas unitarias. Las herramientas disponibles en el mercado incluyen la familia XUnit: JUnit para Java, CppUnit para C ++, PHPUnit para PHP y NUnit para NUnit. Red y mono.

Las pruebas unitarias son el tema central desde las 3 en punto.

Misión 3.2. Busque herramientas para depurar código y realice pruebas unitarias en diferentes lenguajes en un entorno de desarrollo.

Pruebas de integración

Son pruebas funcionales realizadas por técnicos que permiten detectar problemas entre módulos relacionados, previamente probados individualmente a través de pruebas unitarias. Deben tener en cuenta los mecanismos de ensamblaje de módulos fijados en la estructura del programa, es decir, las interfaces entre componentes en la arquitectura del software. Las pruebas unitarias y las pruebas de integración a menudo se superponen y se mezclan en el tiempo.

Dependiendo del orden de integración, existen diferentes tipos de pruebas de integración. El orden de integración elegido afecta varios factores, tales como: cómo preparar el caso de uso, las herramientas requeridas, el orden de codificación y prueba de los módulos, el costo de depuración y el costo de preparar el caso de uso. Los tipos básicos de integración son los siguientes:

- Integración incremental: agrupa el próximo módulo a probar con un conjunto de módulos ya probados. Hay dos tipos principales:
  - Arriba: comienza con el módulo de folla.
  - Down: comienza con el módulo raíz.
- Integración no incremental: cada módulo se prueba por separado, luego se integra todo de una vez y se prueba todo el programa. También se conoce como el Big Bang, ya que el número de módulos crece instantáneamente.

Integración incremental ascendente

En la integración incremental ascendente, comience con una combinación de módulos de nivel inferior. La integración ascendente se caracteriza por:

- Los módulos pueden ser tratados individualmente o los módulos de bajo nivel que realizan algunas funciones específicas pueden ser combinados en grupos para reducir el número de pasos de integración.
- Escribir un módulo de unidad o unidad para cada grupo, este módulo es un módulo de escritura que permite simular el módulo de llamada, ingresar datos de prueba a través de parámetros de entrada, recoger resultados a través de parámetros de salida.

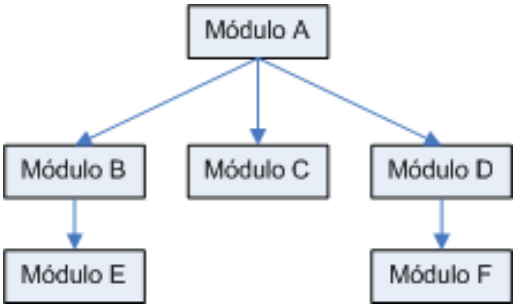
  Cada grupo se sometió a prueba con su propio propulsor.
- Se eliminan los módulos de accionamiento de cada grupo y se sustituyen por módulos de nivel superior en la jerarquía.

La construcción de los módulos de propulsión generalmente no es muy complicada. Esta conveniencia condujo a la aparición de herramientas automatizadas capaces de realizar tareas de propulsores. Por lo general, se compone de:

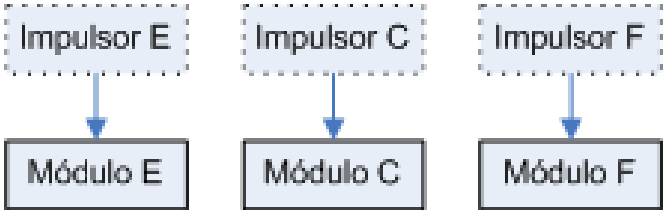
- Las instrucciones obtienen los datos requeridos, pasándolos al módulo a prueba.

  La llamada al módulo a probar.
- Instrucciones necesarias para mostrar los resultados devueltos por el módulo a prueba.

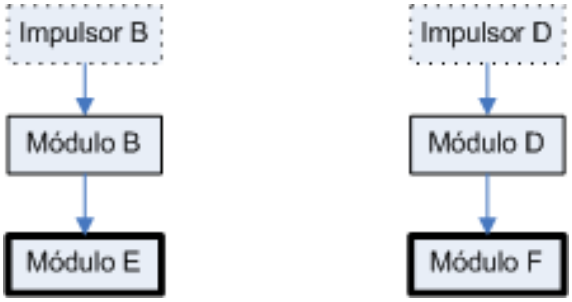
Por ejemplo, si hay los siguientes módulos, los pasos de integración serán 6.



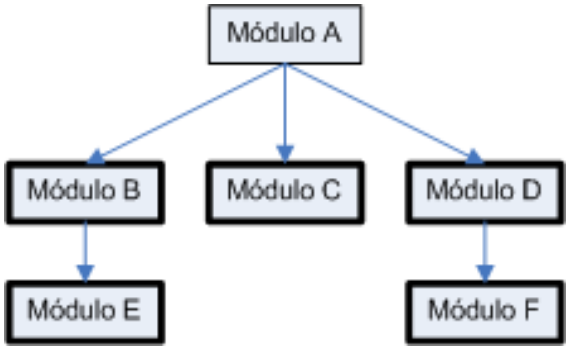
Pasos 1, 2 y 3: Realice pruebas unitarias E, C y F. Los impulsores se indican con líneas discontinuas en la siguiente figura.



Pasos 4 y 5: una parte realiza simultáneamente pruebas de unidad B-E de integración B y A (o interfaz), y la otra parte realiza simultáneamente pruebas de unidad D y pruebas de interfaz D-F. Los módulos que han sido probados en los pasos anteriores se representan con bordes gruesos en la siguiente figura.



Paso 6: El módulo A se incrusta y se prueba todo el programa, lo que no requiere ningún controlador, ya que todos los códigos de gestión de entrada y salida del programa están presentes.



Integración incremental descendente

La integración incremental de abajo hacia arriba comienza con el módulo maestro (nivel superior o módulo raíz) e integra gradualmente los módulos esclavos. No existe una regla universal para determinar qué módulos subordinados deben incorporarse primero. Se recomienda que los componentes clave y los módulos de entrada/salida se incorporen lo antes posible. Existen dos secuencias básicas para la integración posterior:

- Profundidad primero: Completa la ramificación del árbol modular. En el ejemplo anterior , la secuencia de módulos sería A-B-E-C-D-F.
- Amplitud primero: se están completando los niveles horizontales de la jerarquía modular . En el ejemplo anterior, la secuencia de módulos sería A-B-C-D-E-F.

La integración descendente se caracteriza por:

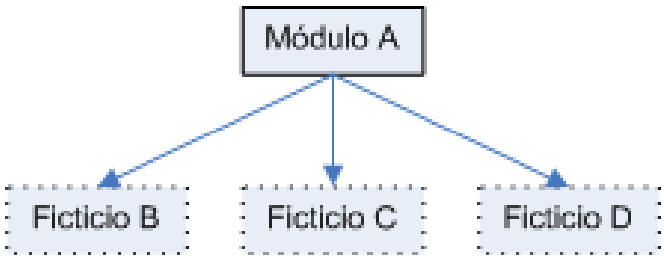
- El módulo raíz primero prueba y escribe módulos virtuales (stubs) que simulan la presencia de subordinados ausentes, que son llamados por el módulo raíz.
- Una vez probado el módulo raíz, sustituir uno de los módulos subordinados ficticios por el correspondiente según el orden elegido y fusionar los módulos subordinados ficticios para recoger las llamadas a este último.
- Se repite el procedimiento detallado para el módulo raíz, es decir, cada vez que se añade un nuevo módulo se realiza la prueba correspondiente y al final de cada prueba se reemplaza una ficticia por su equivalente real. Se recomienda repetir algunos casos de prueba realizados previamente para asegurarse de que no se introduzcan nuevos defectos.

La codificación de un módulo ficticio subordinado es más complicada que la creación de un impulsor. Los ficticios deben simular el control de las llamadas de recolección y hacer algo con los parámetros que se les pasan. La ficción tiene varios niveles de complejidad:

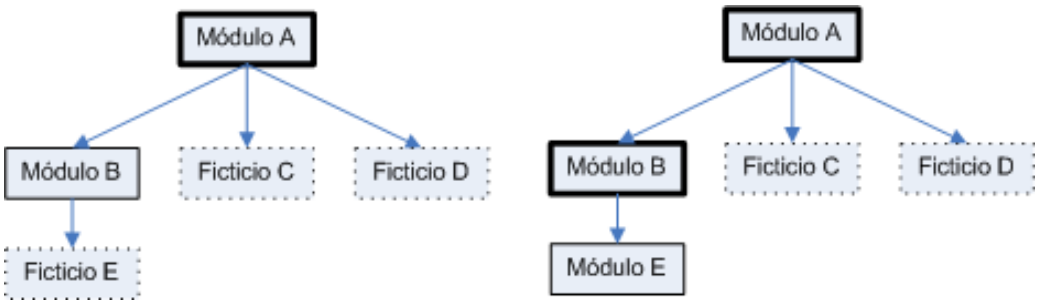
- Un módulo que muestra solo un mensaje de seguimiento. Por ejemplo, printf ("módulo ejecutado 1");
- Módulos que muestran los parámetros que se les pasan. Por ejemplo, recibir el parámetro x y mostrarlo usando printf("%d",x);
- Devuelve un módulo que no depende del valor del argumento pasado como entrada (siempre devuelve el mismo valor, o un valor aleatorio, etc.). Por ejemplo, return(5); independentemente dos parámetros que reciba.
- Un módulo que devuelve un valor de salida más o menos correspondiente a dicha entrada, dependiendo de los parámetros pasados. Por ejemplo, utilice el valor de retorno de una entrada buscada en una matriz bidimensional, return(table[x][y]); Cuando obtienes x e y.

Por ejemplo, adoptar el mismo diseño de módulos que la integración de arriba hacia abajo, con prioridad de pedidos en profundidad, requiere 6 pasos.

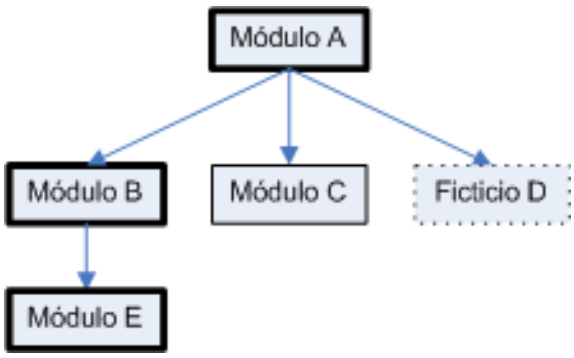
Primera etapa: realizar pruebas unitarias de A. Los ficticios se indican con líneas discontinuas en la siguiente figura.



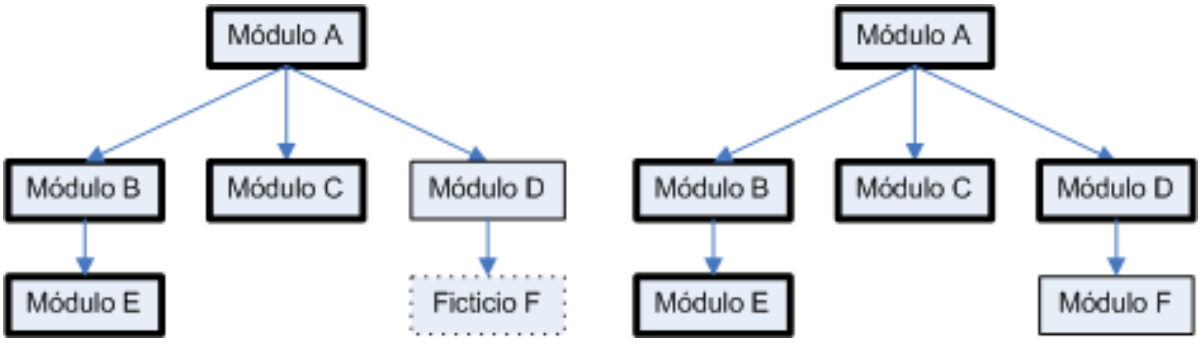
Pasos 2 y 3: Realice simultáneamente pruebas de unidad de integración (o interfaz) A-B, seguidas de pruebas de unidad E y pruebas de interfaz B-E simultáneamente. Los módulos probados en los pasos anteriores se representan con bordes gruesos en la siguiente figura.



Paso 4: Las pruebas del módulo C y la interfaz A-C se realizan simultáneamente.



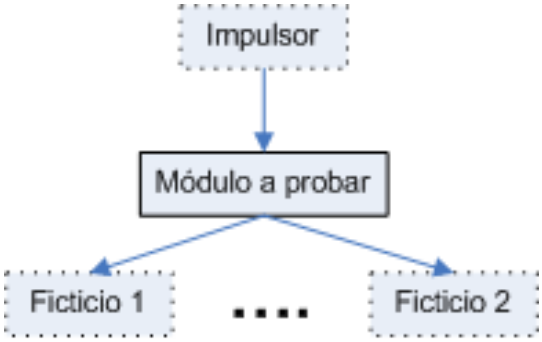
Pasos 5 y 6: Las pruebas del módulo D y la interfaz A-D se realizan simultáneamente, seguidas de las pruebas del módulo F y la interfaz D-F, y el resto se integran.



Integración no incremental

La integración no incremental es el único caso en el que las pruebas unitarias y la integración están completamente separadas. La integración no incremental se caracteriza por:

- Cada módulo require para ser probado:
  - Desde el módulo de accionamiento, que transfiere los datos de prueba al módulo y muestra los resultados de tales casos de prueba.
  - Os módulos ficticios necesarios para simular a función de cada módulo subordinado ao módulo que imos probar.



- Después de probar cada módulo por separado, ensamble todos de una sola vez para formar un programa completo.

Comparación entre los diferentes tipos de integración

La comparación de las ventajas de la integración no incremental e incremental se refleja en la siguiente tabla:

Ventajas de la integración no incremental	Ventajas de la integración incremental
Menos tiempo de máquina requerido para las pruebas, ya que la combinación de módulos solo requiere una prueba. Hay más oportunidades para probar módulos en paralelo. Requiere menos trabajo porque se escriben menos controladores y módulos ficticios.	Detección temprana de defectos y errores en las interfaces, ya que las pruebas de conexión entre módulos comienzan temprano. La depuración es mucho más fácil, ya que desde el síntoma de un defecto detectado en el paso de integración, hay que atribuirlo con mayor probabilidad al último módulo Incrustado. Revisar el programa con más detalle, revisando cada interfaz mientras avanza.

Las características de integración incremental upstream y downstream están relativamente rezagadas Reflejado en el siguiente cuadro:

Integración incremental ascendente	Integración incremental descendente
Es ventajoso cuando existen defectos en los niveles inferiores del programa. Las entradas son más fáciles de crear. El programa, como entidad, no existe antes de incluir el último módulo. Se requiere un módulo de accionamiento. Os módulos impulsores son fáciles de crear. Facilitar la observación de los resultados de las pruebas.	Es ventajoso cuando hay fallos en los niveles superiores del programa. Antes de incorporar E/S, es difícil representar instancias y las entradas pueden ser difíciles de crear. Después de la función de E/S incorporada, facilita la representación del caso. Antes de incorporar el último módulo, revise la estructura anterior del programa. Necesidad de subordinados ficticios. Subordinados ficticios no son fáciles de crear. Dificultad para observar los resultados. Inducción de retraso en la finalización de parte de la prueba Módulo.

La elección de la estrategia de integración depende de las características del software y, a veces, de la planificación del proyecto. Algunas sugerencias podrían ser:

Identificación y prueba de canto antes de aquellos módulos considerados críticos o problemáticos.

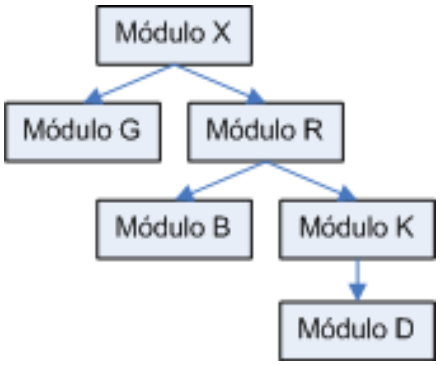
- En general, el mejor compromiso puede ser un enfoque combinado (a veces denominado sándwich) que incluya:

• Adoptar un enfoque de integración ascendente para las capas superiores de la estructura del programa.

• Utilice el orden ascendente a los inferiores al mismo tiempo.

• Continúe hasta que ambos métodos estén en algún nivel intermedio.

> Misión 3.3. Detalle los pasos para la integración incremental de arriba hacia abajo del siguiente conjunto de módulos.



Pruebas o implementación del sistema

Son pruebas no funcionales realizadas por técnicos que permiten verificar que el sistema está completo, cumple con los requisitos funcionales y técnicos establecidos y está correctamente conectado con otros sistemas. Además, se realizan otras pruebas tales como:

- Prueba de carga: Este es el tipo de prueba de rendimiento más sencillo. Las pruebas de carga generalmente se realizan para observar el comportamiento de la aplicación bajo la cantidad esperada de solicitudes. Esta carga puede ser el número esperado de usuarios simultáneos que utilizan la aplicación y realizan un número específico de transacciones durante la duración de la carga. Esta prueba muestra el tiempo de respuesta de todas las transacciones importantes de la aplicación. Si también se monitorea la base de datos, el servidor de aplicaciones, etc., entonces esta prueba puede mostrar el cuello de botella en la aplicación.
- Pruebas de sobrecarga o estrés: Permite evaluar el comportamiento del sistema cuando está sometido a situaciones extremas, como solicitudes de demasiada solicitud, uso de memoria máxima, trabajo en situaciones de memoria baja, tener varios usuarios realizando la misma operación simultáneamente, uso de una cantidad máxima de datos.

Prueba de estabilidad: Esta prueba se utiliza típicamente para determinar si una aplicación puede soportar la carga sostenida esperada. Por lo general, esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación.

- Prueba de pico: como su nombre indica, intenta observar el comportamiento del sistema a través de los cambios en el número de usuarios, incluyendo cuándo están inactivos y cuándo hay cambios

Dramáticamente en su equipaje. Este tipo de prueba se recomienda utilizar un software automatizado que permita variar el número de usuarios mientras que el administrador registra los valores a monitorizar.

- **Prueba de compatibilidad: permite probar el sistema en diferentes entornos, medios o sistemas operativos para ver si hay defectos en apariencia o funcionalidad.**
- **Pruebas de seguridad y acceso a datos: Permite probar cómo responde el sistema a ataques externos, tales como intrusiones no autorizadas o el disfraz de código malicioso en la entrada de datos.**
- **Pruebas de recuperación y tolerancia a fallas: permite anomalías externas, tales como fallas eléctricas, de equipos, de software externo o de comunicaciones, y vea que el sistema se recupera sin pérdida de datos y fallas de integridad, y el tiempo necesario para realizar esto.**

Pruebas de verificación o aceptación

Son pruebas no funcionales realizadas por personas no técnicas para verificar que el producto final cumple con los requisitos originales del software y se basan en acciones visibles para el usuario y la salida del software que el usuario puede reconocer.
Pueden existir pruebas alfa y/o beta:

Las pruebas alfa a menudo se asocian con el software de contrato. Usted ejecuta el cliente en un entorno controlado bajo la supervisión de la compañía que desarrolla el software. Los desarrolladores de software serán conscientes de posibles errores y problemas de uso.

- Las pruebas beta a menudo se asocian con software de interés general, como suites de oficina, pero también se pueden utilizar para software contratado. Permiten a los usuarios confiar en su entorno de trabajo real, sin el control directo de la empresa de software. El usuario reportará los resultados de la prueba. A veces, estas versiones se lanzan al público en general, pero bajo esta etiqueta, para que todos sepan que puede haber errores.

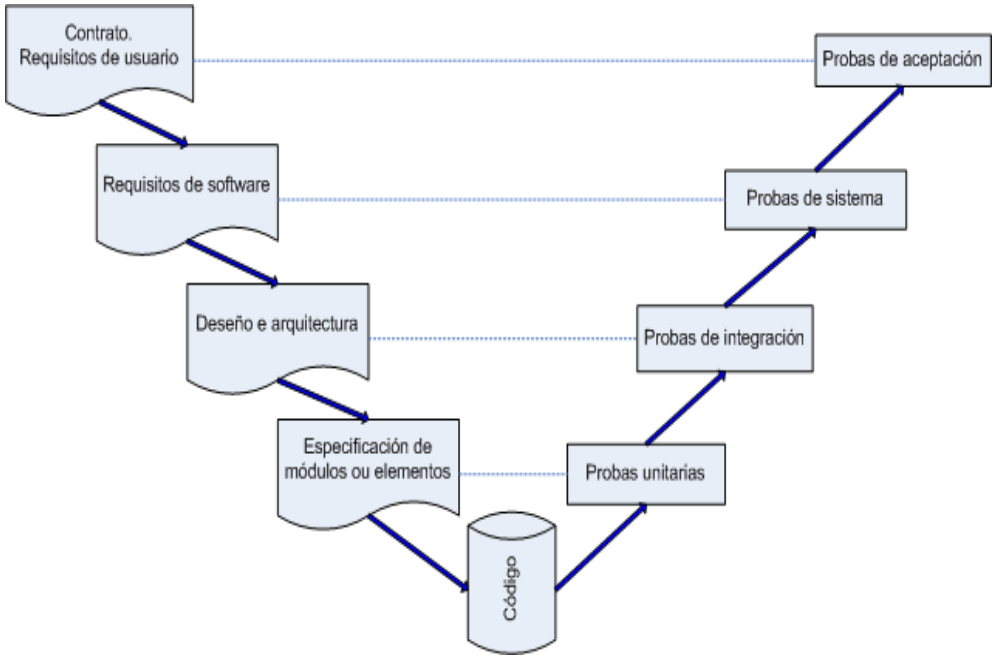
Pruebas de regresión

Estas son las pruebas realizadas por los técnicos para evitar efectos secundarios. Se aplican cada vez que se realiza un cambio en el software para verificar que otros módulos o partes del software no presenten comportamientos o errores no deseados.

Se proporciona un caso especial de cambios en el software cuando se agregan nuevos módulos a pruebas de integración incremental. Puede ocurrir una situación en la que un módulo que alguna vez funcionó bien deje de funcionar porque se ha unido a otro módulo. En este caso, las pruebas de regresión incluirán un subconjunto significativo de las pruebas realizadas para la reaplicación del módulo ya integrado.

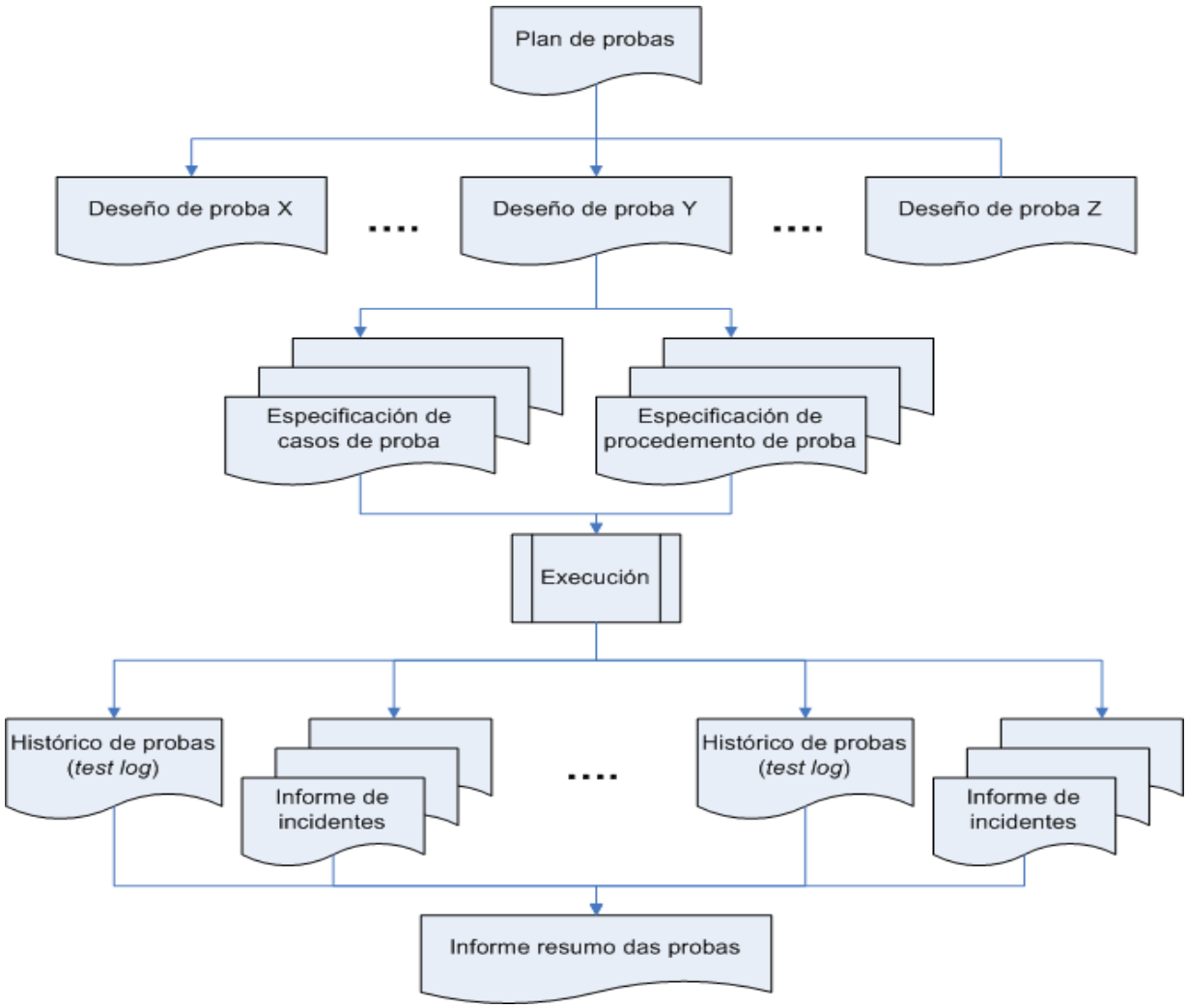
Estrategia de aplicación de prueba en el ciclo de vida clásico

La estrategia de aplicación y el plan de prueba están diseñados para integrar el diseño de casos de prueba en una serie de pasos bien coordinados, creando diferentes niveles de pruebas, con diferentes objetivos. En general, las estrategias de prueba suelen seguir los pasos que se muestran en la siguiente figura a lo largo del ciclo de vida del software.



1.4 Documentación de diseño de pruebas

Los documentos de prueba son necesarios para organizarlos bien y garantizar su reutilización. De acuerdo con la norma IEEE 829, la documentación relacionada con la prueba está asociada con las diferentes fases de la prueba, como se muestra en la siguiente figura.



El primer paso es un plan general (plan de prueba) para el trabajo de prueba, incluido el alcance del plan, los recursos a utilizar, el plan de prueba y las actividades a realizar. El segundo paso es el diseño de la prueba, que se deriva de la ampliación y refinamiento del plan de prueba y especifica las características de las diferentes pruebas. A partir de este diseño, se especifica cada caso de prueba mencionado brevemente en el diseño de prueba y se especifica cómo se lleva a cabo y desarrolla la ejecución de tales casos de uso (procedimientos de prueba) en detalle. Tanto la especificación del caso de prueba como la especificación del programa deben ser los documentos básicos para realizar la prueba. El último paso es un informe que contiene un resumen de las pruebas, que refleja los resultados de las actividades de prueba y evalúa el software frente a estos resultados.

La documentación de la ejecución de la prueba es esencial para detectar y corregir defectos de manera efectiva y dejar registros de los resultados de la ejecución de la prueba. Los documentos generados por cada ejecución son:

- Historial de pruebas: registra los hechos relevantes que ocurren durante la ejecución de pruebas.
- Reporte de incidentes: Registrar cada incidente ocurrido durante la prueba que requiera más investigación.

2. Depuración de código

2.1 Introducción

Las operaciones de depuración se utilizan para verificar el código de la aplicación en tiempo de ejecución y encontrar soluciones a los errores detectados. Permite que las líneas de código se ejecuten hasta el momento en que el estado puede ser comprobado para encontrar problemas.
El ejemplo de depuración para esta actividad utilizará un proyecto de Java llamado Statistics, que contiene paquetes llamados pruebas para las clases Statistics.java y Main.java. Ejecutar el proyecto le permite escribir dos números enteros positivos (el primero es mayor o igual al segundo) y ver las estadísticas:

¦ Factorial de cada número: producto de enteros desde 2 hasta ese número.

¦ Combinación de dos números: resultado de dividir el factorial del primer número entre el factorial del segundo y el factorial de la diferencia entre los dos números.

¦ Variación no repetida de dos números: resultado de multiplicar la combinación por el factorial del segundo número.

¦ Variación en la que se repiten dos dígitos: el resultado de elevar el primer dígito al segundo. El

código de Main.java es:

```
Paquete probas;
Importar java.util.Scanner;

Clase pública Main {
    public static void main(String [] args) {
System.out.print("\ncálculos estadísticos\n ");
Teclado del escáner = nuevo escáner (System.in);
        Error booleano;
        int m, n;
        De {
            Intenta {
Error = falso;
System.out.println("Teclee m (> = 0): ");
m=Integer.parseInt(teclado.next line ());
System.out.println("Teclee n (> = 0 y < = m): ");
n=Integer.parseInt(teclado.next line ());
Estadística es = nueva estadística (m, n);
Doble factor n = es. factorial (n);
System.out.printf("permutación(%d) = %f\n",n, factN);
              Doble factM = es. factorial(m);
System.out.printf("Permutaciones(%d)=%f\n",m,factM); System.out.printf("variations(%d,%d)=%f\n",
m,n,es.variations()); System.out.printf("combinations(%d,%d)=%f\n",m,n,es.combinations ());
System.out.printf("variaciones con repet.(%d,%d)=%f\n",m,n,es.variac_repet ());

            } catch (excepción e) {
System.out.println(e.get class () + "->" + e.get message ()); // Muestra el error error = true;

            }
        } while (error);
    }
}
```

El código de statistics.java es:

```
Paquete probas;

Estadísticas de clase pública {

    Privado internacional;
    Privado int n;

    public Estadisticos (int m, int n) lanza una
        excepción {if(m < 0 n < 0 m < n){
            Se lanza una nueva excepción ("error." +
                    "Los argumentos tienen que ser >=0 y el primero >= que el segundo");
        }
        Este. n=n;
        this.m=m;
    }
    /* Cálculo Factorial o permutaciones de x */
    public double factorial(int x) lanza una excepción {
        Doble resultado = 1;
        Pa  (int i=2; i<=x; i++) {
        ra  Resultado *=i;
        }
        El efecto de retorno;
    }
    /* Cálculo Combinaciones de m elementos tomados de n en n*/
    public double combinations () lanza una excepción {
        Doble combinación = factorial (m)/(factorial (n) * factorial (m-n));
        Combinación de retorno;
    }
    /* Cálculo Variaciones de m elementos tomados de n en n */
    public double variations () lanza una excepción {
        double vari = combinaciones()*factorial(n);
        Devuelve una variable;
    }
    /* Cálculo Variaciones con repetición de m elementos tomados de n en n */
    public double variac_repet() throws Exception {
        Doble varirepe=Matth.pow(m,n);
        Volver a varirepe
    }
}
```

2.2 Sesión de depuración

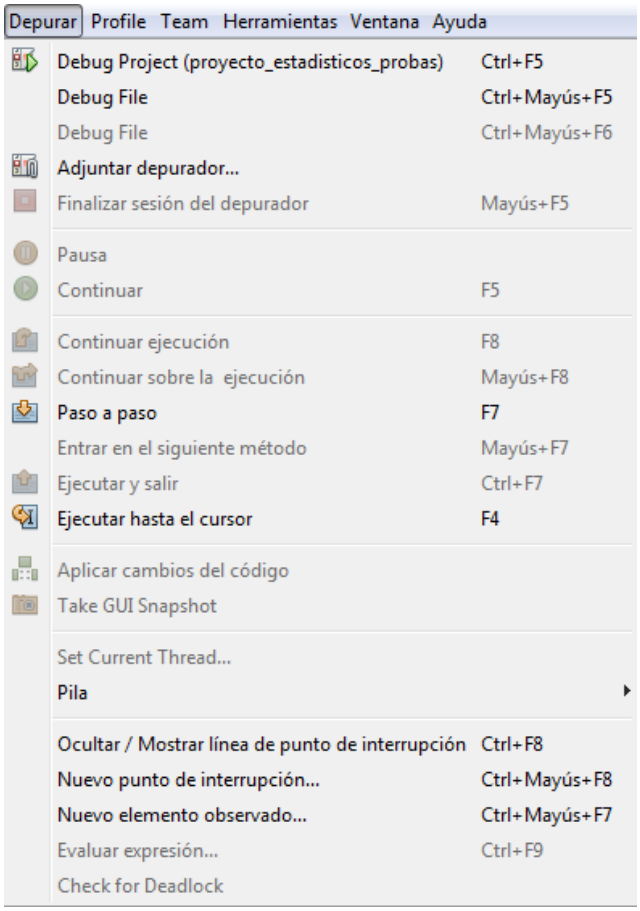
Para iniciar una sesión de depuración, el archivo o proyecto a depurar debe tener el método principal, es decir, debe poder ejecutarlo.

NetBeans le permite iniciar sesión en la depuración de un proyecto o archivo y elegir cómo comenzar la depuración de diferentes maneras, la forma más fácil es seleccionarlo en la ventana Proyecto, seleccionarlo y hacer clic derecho en él y luego seleccionar Depurar.

También podemos depurar el archivo fuente que se está editando en ese momento acudiendo al menú principal y seleccionando Debug > Debug File o seleccionando cualquier archivo de la ventana Proyectos, seleccionando el archivo, haciendo clic derecho y seleccionando Debug File.

En todos los casos anteriores, la sesión de depuración comienza en la clase principal del archivo seleccionado o del proyecto seleccionado y continúa hasta que el programa finaliza normalmente, Encontré algunos errores o algunos puntos de interrupción. Entonces la mejor opción para empezar es La depuración será colocar el cursor en la línea anterior del programa y presionar F4 o Debug > Run to *Cursor*También podemos crear un punto de interrupción, como veremos más adelante.

En el menú Debug/Debug, puede ver que también puede iniciar una sesión de depuración para la ejecución paso a paso, incluso en la línea donde se encuentra el cursor.



Corre al cursor

La opción Ejecutar al cursor permite que el programa se ejecute hasta la posición del cursor en el archivo que se está editando y detenga el programa hasta que se le indique en la depuración que realice la siguiente acción. El archivo editado debe ser llamado desde la clase principal del proyecto principal. Para realizar esta depuración, debe colocar el cursor en el código fuente, presionar F4 o seleccionar desde el menú principal Debug > Ejecutar el cursor o desde la barra de herramientas de depuración.

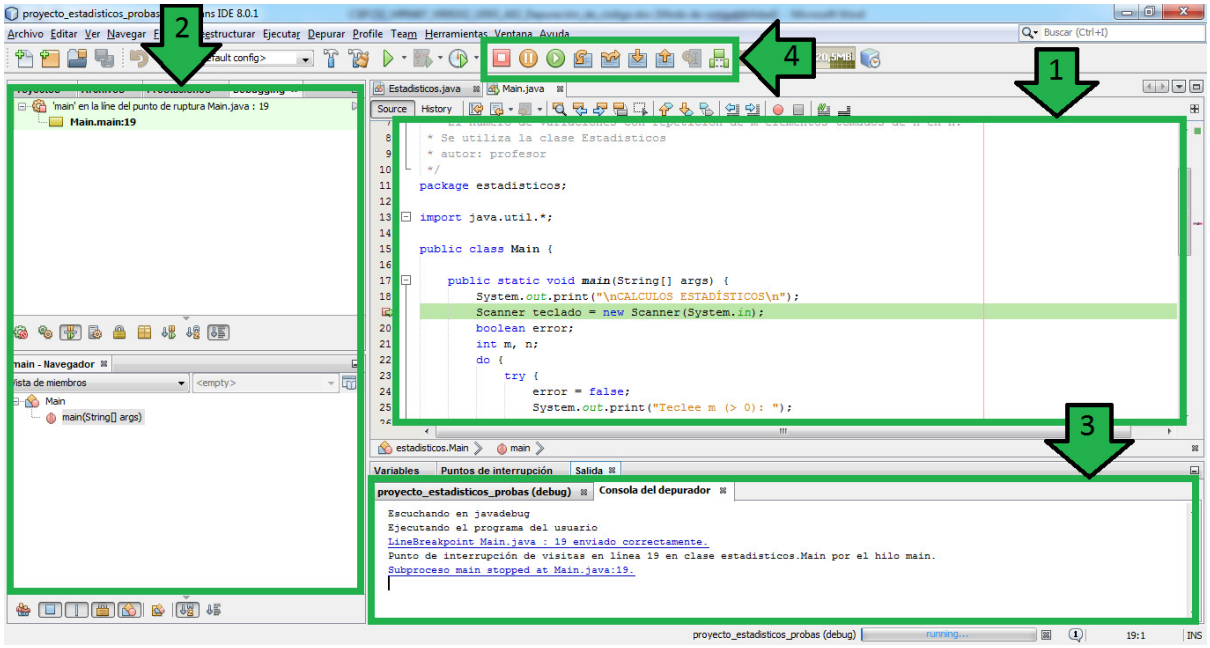
Ejecución paso a paso

La opción paso a paso le permite ejecutar el programa línea a línea y pausar la ejecución hasta que se le indique en la depuración qué hacer a continuación. Para iniciar una sesión de depuración paso a paso, seleccione "Depuración- > Paso a paso" en el menú principal, o pulse F7 en la barra de herramientas de depuración o pulse para detener la ejecución en la línea de la primera instrucción ejecutable. Cada vez que desea correr paso a paso a la siguiente línea, debe presionar F7 nuevamente. Si la línea de código está compuesta por múltiples métodos, se muestra Seleccionar con borde negro, qué métodos vas a realizar paso a paso, puedes seleccionar uno de los otros métodos con la tecla Tab. Puede usar las teclas [Enter] o [F7] para realizar el método seleccionado paso a paso.

```
24      /* Cálculo Combinaciones de m elementos tomados de n en n*/
25      public double combinaciones() throws Exception {
            double combi = factorial(m)/(factorial(n)*factorial(m-n));
27          return combi;
28      }
```

Pantalla de depuración

La pantalla de depuración aparece después de iniciar la depuración y tiene múltiples áreas. Por ejemplo, después de entrar en una sesión de depuración paso a paso, aparece el área como se muestra en la siguiente imagen.



Área 1: El área donde se está depurando el código fuente. La siguiente línea a ejecutar durante la depuración se marca con un color de fondo verde con una flecha verde en el margen izquierdo.

Zona 2: Ventana de depuración, que contiene información sobre el proceso en ejecución. Muestra un menú de iconos en la parte inferior de esta ventana para que pueda cambiar la vista de la información:

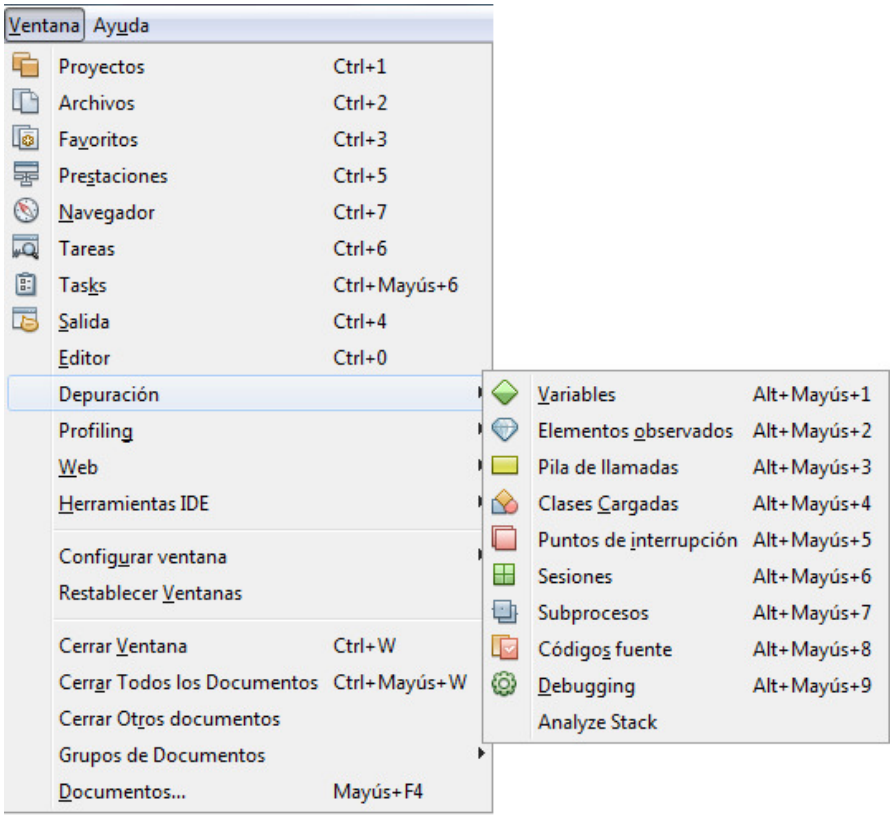
Área 3: Área de ventana:

- La ventana de salida o ventana de salida se subdivide en:
  - *Consola del depurador con información sobre o proceso de depuración.*
  - *Estadística (puesta en marcha). En este proyecto específico con entrada y salida de consola, la entrada de datos se hará aquí y verá la salida resultante.*

- Ventana de variables donde se puede ver y cambiar información sobre variables y expresiones locales.
- La ventana “Puntos de interrupción”, donde se puede ver y cambiar la información sobre los puntos de interrupción.

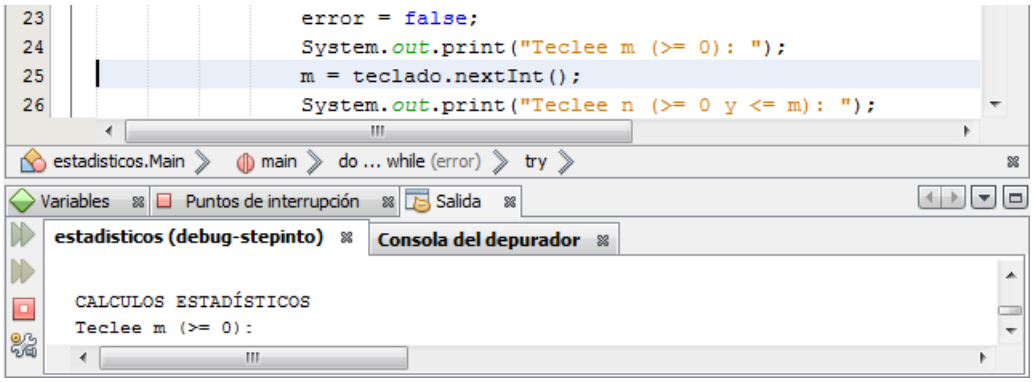
Área 4: Barra de herramientas de depuración para indicar las siguientes acciones a realizar en la depuración. Algunas de estas características también aparecen en el menú principal de "depuración". Puede seleccionar la barra de herramientas que desea ver desde el menú superior de Ver > Barra de herramientas.

Si no ve ninguna de las ventanas anteriores, puede visitar Ventana del menú principal-> Opciones de depuración y seleccionar la ventana que desea ver.



Durante la depuración, puede ocurrir que la ejecución de una línea de código requiere entrada del teclado, y luego:

- La línea de código que contiene la entrada cambia de fondo verde a azul en la ventana de edición.
- El proceso de depuración se detiene hasta que se escriban los datos en la ventana de salida estadística.



Explicación rápida del significado de cada icono en la barra de herramientas de depuración (área 4):

Iconos	Descripción	Detalle
	Salir de la sesión del depurador (mayúsculas + F5)	Complete la puesta en servicio inmediatamente.
	Suspensión	Haga una ligera pausa durante la depuración.
	Continuar (F5)	Reanudar el funcionamiento normal del programa después de una pausa.
	Continuación de la aplicación (F8)	Ejecutar una instrucción. Si contiene una llamada a un método, se ejecutará sin detenerse en cada instrucción.
	Continúa ejecutando (mayúsculas + F8)	Continúe con la aplicación. Es un refinamiento F8 de una expresión con una llamada a un método. En una sesión de depuración paso a paso, cada vez que se utiliza este comando en una expresión con una llamada a un método, se detiene la depuración antes de realizar la llamada al método actual, y se puede ver el historial de los valores de retorno de los métodos inmediatamente anteriores y los valores de parámetro del método actual en la ventana de variables locales. Puede ser útil cuando el valor de retorno de un método no se guarda en una variable y, por lo tanto, no se puede verificar en el momento de la depuración.
	Paso a paso (F7)	Corre paso a paso. Le permite ingresar al método paso a paso para ejecutar la línea actual. Si hay más de una manera de hacerlo, puedes: Utilice la tecla de movimiento del cursor o la tecla de pestaña para seleccionar el método que desea depurar paso a paso y confirme con F7. Funcionamiento normal (F7) En esta versión de NetBeans, de forma predeterminada, Step Into (F7) entra en la ejecución paso a paso de los métodos de la API de Java. Si desea que estos métodos no se abran y se ejecutan, necesita presionar F8 en lugar de F7.
	Ejecución y salida (Ctrl+F7)	En el caso de la depuración interna del método, Ctrl + F7 termina la sesión de depuración del método y vuelve a llamar al método actual. No caso de utilizarse no método principal, finaliza a sesión de depuración.
	Ejecutar al cursor F4	Ejecute en el cursor y espere a que la instrucción continúe la depuración.
	Aplicar cambios de código	Aplicar cambios al código.
		Presione para activar la recolección de basura.

Misión 3.4. Puesta en marcha básica.

La tarea es realizar la siguiente depuración del proyecto Statistics Java:

- 1) Iniciar la depuración paso a paso del método Main, escribiendo los valores de m=5 y n=2 sin ejecutar línea por línea ninguno de los métodos llamados.
- 2). Repita el paso anterior, pero solo ejecuta el método factorial (5). Complete la depuración y vea el resultado final.
- 3). Repita los pasos anteriores, pero ahora solo ejecute el método variables (). Complete la depuración y vea el resultado final.
- 4) Coloque el cursor en la línea 21 de Main.java y comience a depurar hasta que el cursor se encuentre (escriba los valores m=15 y n=3). Complete esta depuración y vea el resultado final.

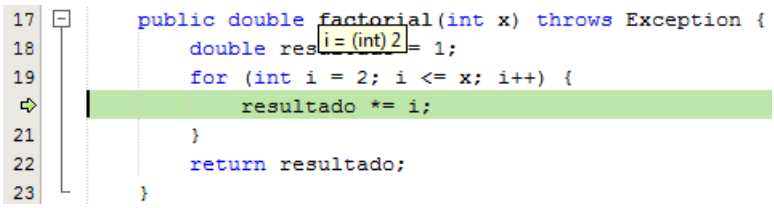
Cabe recordar que para abortar una sesión de depuración antes de que finalice normalmente, es necesario presionar Enter en la barra de depuración, mientras que para ejecutar un método normalmente antes de que finalice la ejecución gradual, es necesario presionar Enter en la barra de depuración.

Prueba las diferencias entre F7, F8 y May+F8 en la depuración.

2.3 Comprobación y modificación de variables, expresiones y métodos

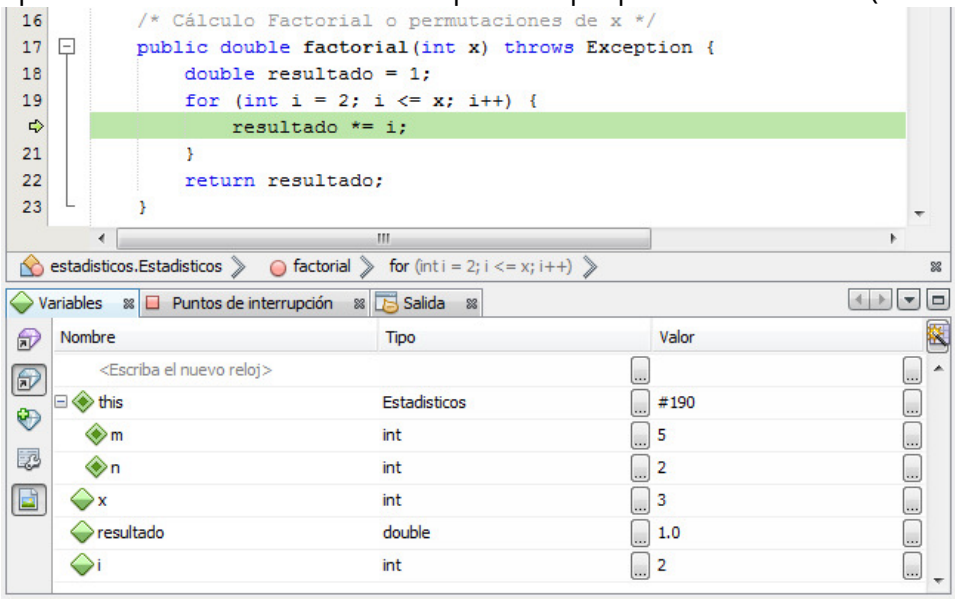
En fuente

Durante la depuración, puede ver el tipo y el valor de una variable colocando el cursor sobre la variable en el código fuente:

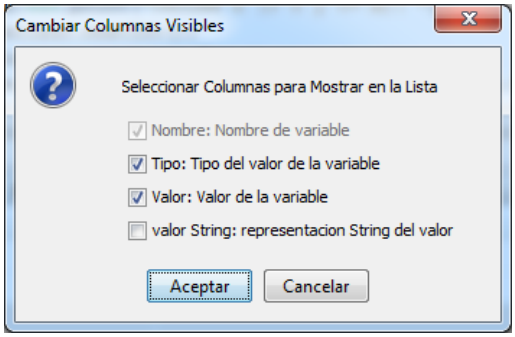


En una ventana variable

Durante la sesión de depuración, puede ver información sobre las variables locales en la ventana Variables, tanto para las variables locales como para las propiedades de clase (si existen).



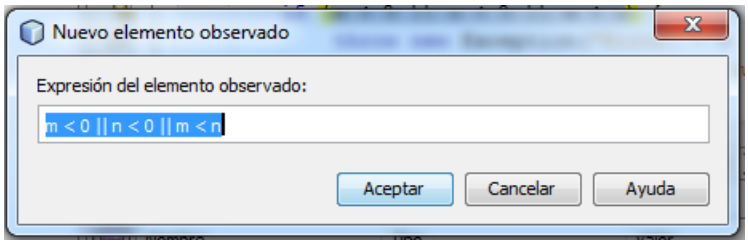
Los iconos ubicados a la derecha te permiten modificar la información que ves.



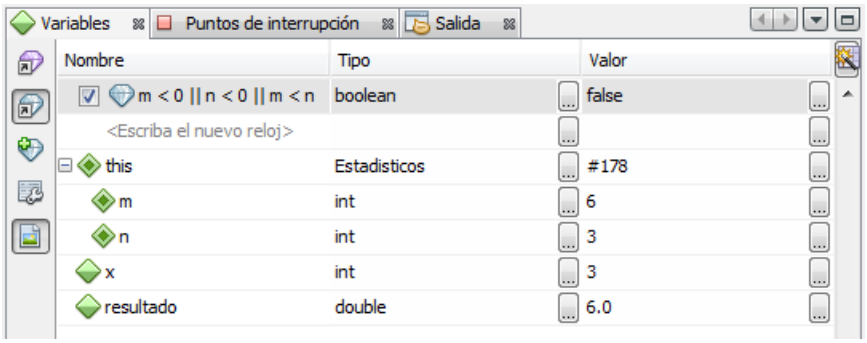
Además de las variables locales, también se pueden agregar expresiones para observaciones. Esto puede hacerse de varias maneras:

- Seleccione la expresión en el archivo fuente editado, haga clic con el botón derecho y seleccione el nuevo elemento de observación, o pulse Ctrl+Case+F7.
- Seleccionar no menú principal Depurar->Nuevo elemento observado.
- Haga clic en el icono en la ventana Variables.
- Escriba el nuevo elemento en la línea de la ventana "Variables" de < Enter new watch >.

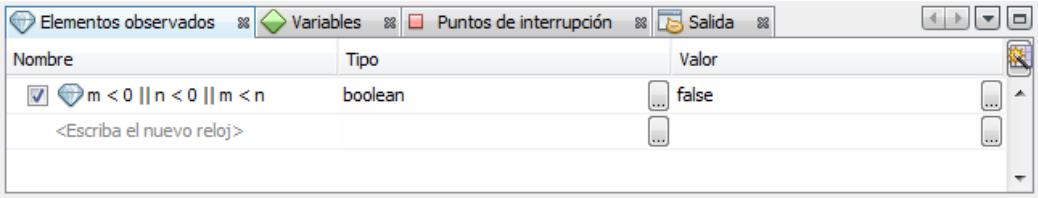
En cualquiera de los tres primeros casos, aparecerá una ventana en la que necesitará definir la expresión que desea observar.



La expresión se muestra como un reloj en la ventana Agregar variable.



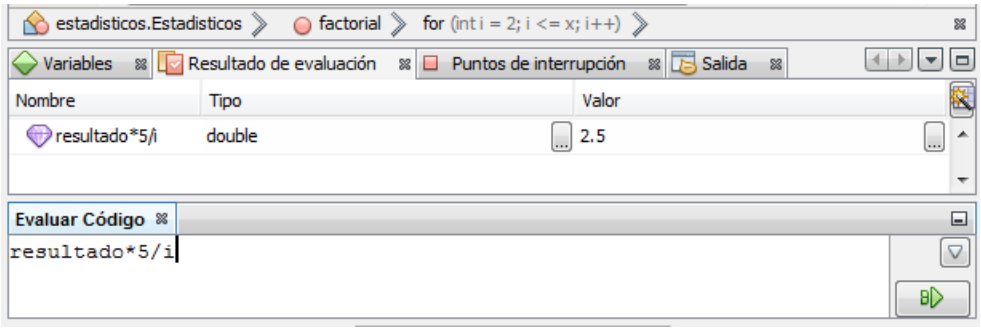
Utilice el interruptor para mover observaciones entre la ventana Variables y la ventana Observaciones.



En la ventana Variables, también puede hacer clic derecho en el nombre de una variable o expresión y realizar cambios, por ejemplo: eliminar una variable o expresión de la ventana, eliminar todos, ver valores en otro formato o editarlos.

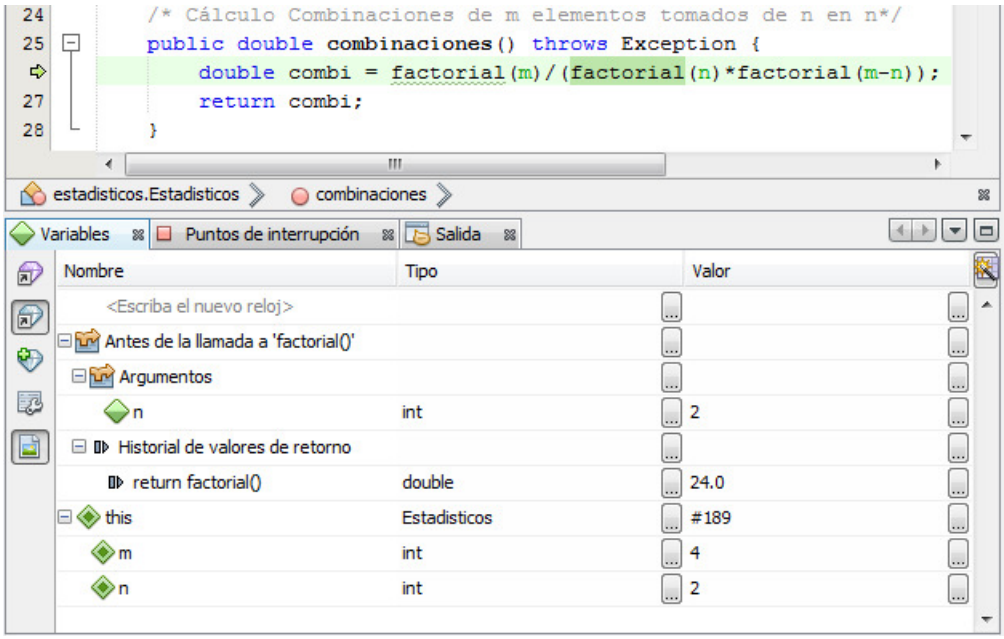
Co menú *Depuración->Evaluar expresión*

Durante la sesión de depuración, el resultado de la expresión se puede ver desde el menú principal seleccionando Depurar- > Evaluar la expresión o pulsando Ctrl+F9; Abre la ventana " Código de evaluación" donde puede escribir la expresión y evaluar el resultado en ese momento pulsando el botón. Si la expresión es imposible en el contexto actual, es posible que no se vean los resultados. El resultado es similar a lo siguiente:



Coa opción de depuración *Continuar sobre la ejecución*

Los valores de retorno del método anterior (historial de valores de retorno) y los valores de parámetros del método siguiente (antes de llamar...) pueden ser visualizados mediante la letra mayúscula +F8 (continuar con ejecución) en una sesión de depuración o haciendo clic en el icono. Una expresión con una llamada de método desde la barra de depuración.



Modificación

En las ventanas Expresiones de evaluación, Variables y Observaciones, puede modificar el valor de la variable y usar ese valor para continuar el proceso de depuración. Para hacerlo, haga clic en el icono junto al valor actual de la variable, o en el icono cuando se usa el valor actual, escriba el nuevo valor, presione Aceptar y continúe con la depuración.

Misión 3.5. La depuración se realiza examinando variables, expresiones y métodos y modificando los valores.

La tarea es realizar la siguiente depuración del proyecto Statistics Java:

Parte 1) Depurar para poder ver el valor de retorno del método factorial en la línea 26 de statistics.java.

```
Doble combinación = factorial (m)/(factorial (n) * factorial (m-n));
```

Parte 2) Depuración para poder comprobar el valor de la variable local i del método factorial(5) durante la sesión de depuración.

Parte 3) de depuración para que el valor de la variable local i del método factorial (5) pueda ser modificado en el momento de la depuración y ver los cambios realizados en la variable local.

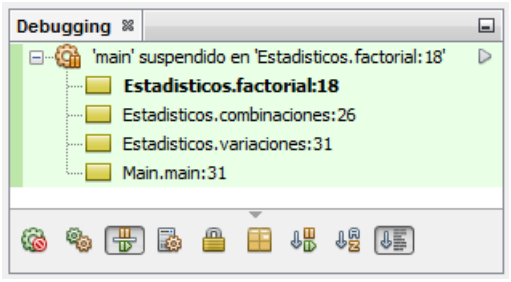
Parte 4) Depuración para que el valor de la expresión m-n pueda verse en la sesión de depuración.

2.4 Pila (pila de llamadas)

El uso de una pila de llamadas es particularmente útil cuando el tiempo de ejecución utiliza múltiples subprocesos o subprocesos. Cuando se inicia una sesión de depuración, la ventana de depuración se abre automáticamente, que muestra información sobre los subprocesos existentes y la pila de llamadas para cada subproceso suspendido o suspendido; Sólo uno de ellos es la corrección actual.

Puesta en servicio de Venta

La ventana de depuración tiene los siguientes aspectos:



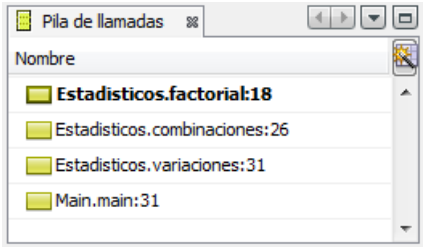
Esta imagen muestra una sesión de depuración en la que la ejecución de main () se interrumpe por una llamada al método variaciones () en la línea 31; La ejecución de variaciones () se interrumpe llamando al método combinations () en la línea 31; La ejecución de composiciones () se interrumpe en la línea 26 mediante una llamada al método factorial (), que es el método actual interrumpido en la línea 18.
La última llamada hecha es la que se considera la actual y se muestra en negrita en la pila. Al mirar las variables locales, verá la variable que se está llamando actualmente. Si el archivo fuente está disponible , puede hacer clic derecho en la llamada y seleccionar Ir a fuente para ver el código fuente de la llamada. En el costado de cada proceso aparecerá un icono con la información del proceso:

Icons	Description
	Indicates a thread that is running
	Indicates a thread suspended by hitting a breakpoint
	Indicates a suspended thread
	Indicates a thread group where all threads are running
	Indicates a thread group where all threads are suspended
	Indicates a thread group with running and suspended threads
	Indicates a call stack frame
	Indicates a call stack frame group

Puede cambiar la vista de esta información utilizando el menú de iconos en la parte inferior:

Button	Description
	Show thread groups
	Shows or hides the controls for suspending and resuming threads.
	Show system threads
	Show suspended and current threads only
	Show monitors
	Show qualified names
	Sort by suspended/resumed state
	Sort by name
	Sort by default

Ver la pila



También puede ver la información de la pila de llamadas al menú principal y seleccionar Ventana-> Depuración-> Pila de llamadas o pulsando Alt + Caps + F3 y se abrirá la ventana de la pila de llamadas en el área de ventanas. La información de cada llamada está marcada con un icono y una descripción de la llamada.

Misión 3.6. Depurar inspeccionando a pila.

La tarea es depurar el proyecto de estadísticas de Java hasta que se ve en la ventana 3 de la pila de llamadas. Intenta también ver las 4 llamadas en la pila.

2.5 Aplicación de cambios de código

Se pueden hacer ciertas modificaciones en el código durante la depuración sin tener que reiniciar el programa. Para reparar el código, debe hacerlo en la ventana de edición, ingresando al menú principal y seleccionando Depuración-> Aplicar cambios de código, o haciendo clic en la barra de herramientas de depuración, se compilará y se reparará el código fuente. (Nota: Debe desactivar la opción "Compilar al guardar" en las propiedades del proyecto, de lo contrario el botón aparecerá como desactivado)

Consideraciones generales:

- Si se produce un error en el momento de la compilación, no se realiza ningún cambio, el error debe ser corregido.
- Si no hay errores, el código objeto generado será el mismo que el que se está ejecutando en la depuración y:
  - Si el cambio de código se realiza dentro de un método que se está depurando actualmente, la pila de llamadas se modifica eliminando la llamada al método para permitir que el módulo se vuelva a llamar, pero
  - La pila no se modifica si se realiza un cambio después de llamar a un método Para reutilizar el código modificado, debe eliminar la llamada de la pila Contienen estos códigos.
- Se excluyen las siguientes modificaciones:
  - Cambiar el modificador de un campo, método o clase.
  - Añadir o eliminar métodos o campos.
  - Cambiar la jerarquía de clases.
  - Cambiar las clases que no se cargan en la máquina virtual.

Misión 3.7. (Opcional) Realiza cambios en el código mientras depuras.

La tarea es modificar el código estático del proyecto Java durante la depuración. La modificación a realizar podría ser para evitar calcular factoriales cuando m o n son mayores que 170, ya que esto provocaría un desbordamiento y devolvería un valor de Infinity. Los siguientes pasos son:
-Iniciar una sesión de depuración, detenerse antes de que el usuario ingrese un número
-Modificar el constructor Statisticos.java para añadir la condición m > 170 al if para que ocurra la excepción, mientras que también se modificó el texto de advertencia.
- Premer o botón “Aplicar cambios en el código”
-Continuar la ejecución introduciendo un valor mayor que 170 para m.
-Compruebe si las modificaciones en el código se han hecho.

2.6 Puntos de interrupción

Definición

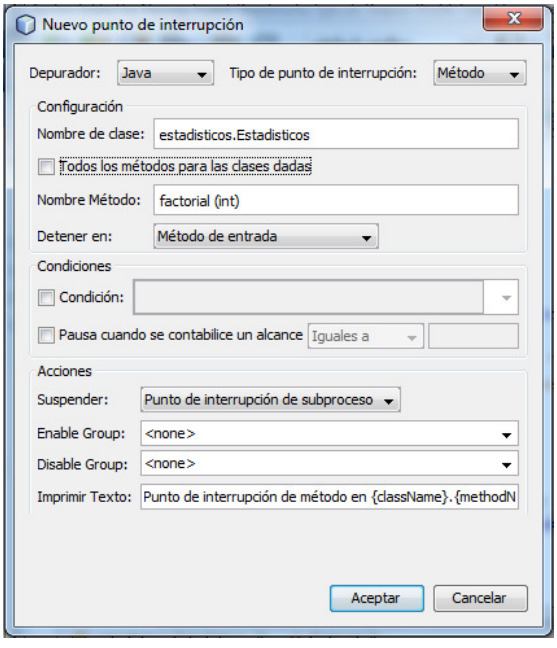
Un punto de interrupción o punto de interrupción o punto de interrupción es una etiqueta en el código fuente que indica al depurador que se detenga en ese punto y espere a que la instrucción continúe ejecutando, durante el cual se puede realizar una operación de comprobación en una variable, expresión o código. Los puntos de interrupción de Java se definen globalmente y afectan a todos los proyectos que contienen el código fuente del punto de interrupción. Por ejemplo, si Statistics. java tiene un punto de interrupción en el método factorial (), la sesión de depuración se detendrá en ese método cada vez que depuras un proyecto que contiene esa clase. NetBeans permite varios tipos de puntos de interrupción:

- Línea: te hace parar cuando llegas a esa línea.
- Clases: para que se detenga al cargar una clase.
- Excepción: hace que se detenga cuando se detecta una excepción, independientemente de si el programa la maneja o no.
- Campos: Se detiene cuando se accede y/o se modifica un campo de una clase.
- Método: se detiene al entrar y/o salir de un método.
- Thread: se detiene cuando se inicia y/o termina un hilo o hilo.

Establecer un punto de interrupción

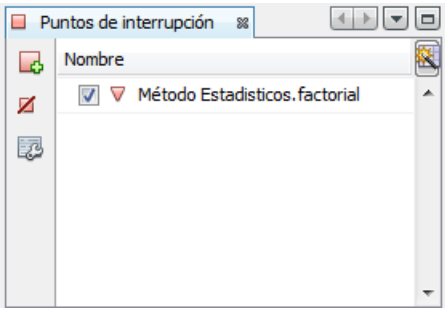
Para establecer un punto de interrupción, seleccione el elemento de código en el que desea establecer el punto de interrupción y seleccione en el menú Depuración- > Nuevo punto de interrupción, o presione Crtl + Case + F8. Aparecerá el cuadro de diálogo Nuevo punto de interrupción, que contiene la información predeterminada relacionada con el elemento seleccionado y debe realizar los ajustes apropiados en él. El IDE indica los puntos de interrupción establecidos por un icono en el borde izquierdo del código fuente, mientras que los puntos de interrupción de línea se indican además colocando la línea en un fondo morado.

La siguiente imagen muestra un ejemplo de un punto de interrupción del método para depurar una interrupción cuando se ingresa el método factorial de la clase Statistics:



Los puntos de interrupción del método marcados con iconos aparecerán en el código fuente, como se muestra en la siguiente imagen:

```
16      /* Cálculo Factorial o permutaciones de x */
17      public double factorial(int x) throws Exception {
18          double resultado = 1;
19          for (int i = 2; i <= x; i++) {
20              resultado *= i;
21          }
22          return resultado;
23      }
```



NetBeans verifica la validez de dos durante la sesión de depuración Puntos de interrupción y si se encuentra un mensaje inválido a través Icono "break" en el código fuente, y en Consola del depurador. También puedes ver el punto de ventana *Interrupción e información sobre el punto de interrupción. Si a* La ventana no está abierta se puede abrir desde la ventana del menú principal- *>Depuración->Puntos de interrupción* O presionar Alt+F5.

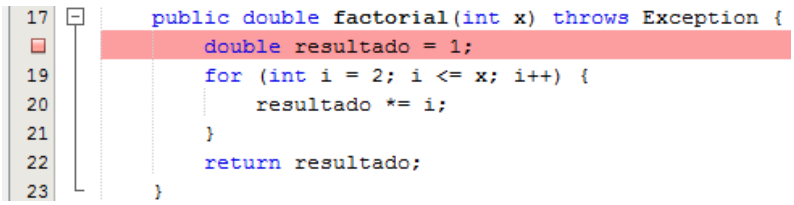
Posibles iconos para marcar un punto de interrupción son:

Annotation	Description
	Breakpoint
	Disabled breakpoint
	Invalid breakpoint
	Multiple breakpoints
	Method or field breakpoint
	Disabled method or field breakpoint
	Invalid method or field breakpoint
	Conditional breakpoint
	Disabled conditional breakpoint
	Invalid conditional breakpoint
	Program counter
	Program counter and one breakpoint
	Program counter and multiple breakpoints
	The call site or place in the source code from which the current call on the call stack was made
	Suspended threads
	Thread suspended by hitting a breakpoint

Los puntos de interrupción de línea son los más utilizados, por lo que hay varias maneras de establecerlos:

- La forma más sencilla es hacer clic en el borde izquierdo de la ventana de edición hasta la altura de la fila donde desea colocar el punto de interrupción.
- Coloque el cursor sobre la línea donde se encuentra la instrucción donde se desea colocar el punto, haga clic derecho y seleccione la opción “Ocultar/Mostrar línea de punto de ruptura” o pulse las teclas Ctrl+F8.
- Coloque el cursor sobre la línea donde se encuentra la instrucción donde se desea colocar el punto y, en el menú principal, seleccione Depuración-> Ocultar/Mostrar línea de punto de interrupción o pulse las teclas Ctrl+F8.

En el código fuente, los puntos de interrupción marcados aparecen como se muestra en la siguiente imagen.



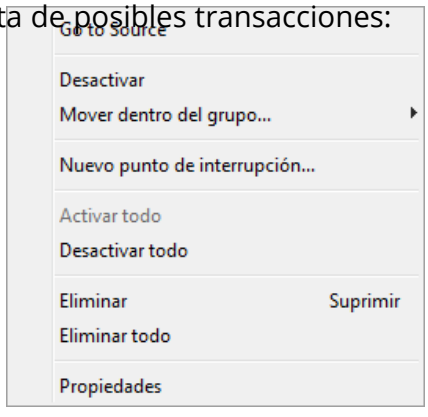
Corre hasta el punto de ruptura

Para poder ejecutar el proyecto principal hasta el punto de interrupción, debe establecer el punto de interrupción y depurar el proyecto, por ejemplo, seleccionando la opción "Debug- > Debug Project" en el menú principal, que ejecutará el programa principal hasta el primer punto de interrupción, o excepción, o si no hay punto de interrupción, hasta el final. Desde el punto de interrupción, puede continuar la depuración con las opciones que ya ha visto.

Modificar un punto de interrupción

Debido a que se establece un punto de interrupción, se puede deshabilitar cuando está habilitado (registrado pero no utilizado), habilitar cuando está deshabilitado (registrado pero no utilizado), eliminar (eliminado) o modificar. Todas estas operaciones se llevan a cabo desde la ventana de punto de interrupción, son:

- Una forma rápida de habilitar o deshabilitar un punto es marcar o desmarcar el cuadro de texto correspondiente en la ventana anterior.
- Una forma rápida de eliminar un punto de interrupción es colocar el cursor sobre el nombre del punto de interrupción en la ventana anterior y presionar la tecla Supr.
- Todas las operaciones de modificación del punto de interrupción se pueden hacer colocando el ratón sobre el nombre del punto de interrupción en la ventana anterior y haciendo clic derecho. A continuación se presenta una lista de posibles transacciones:
  - En el caso de estar en un punto de interrupción activo, la opción Cerrar aparece en la ventana Opciones; Si está desactivado, aparece
    En su lugar, hay una opción de activación.
  - Opciones para desactivar todo, habilitar todo o eliminar Todo tendrá un impacto en todos. Puntos



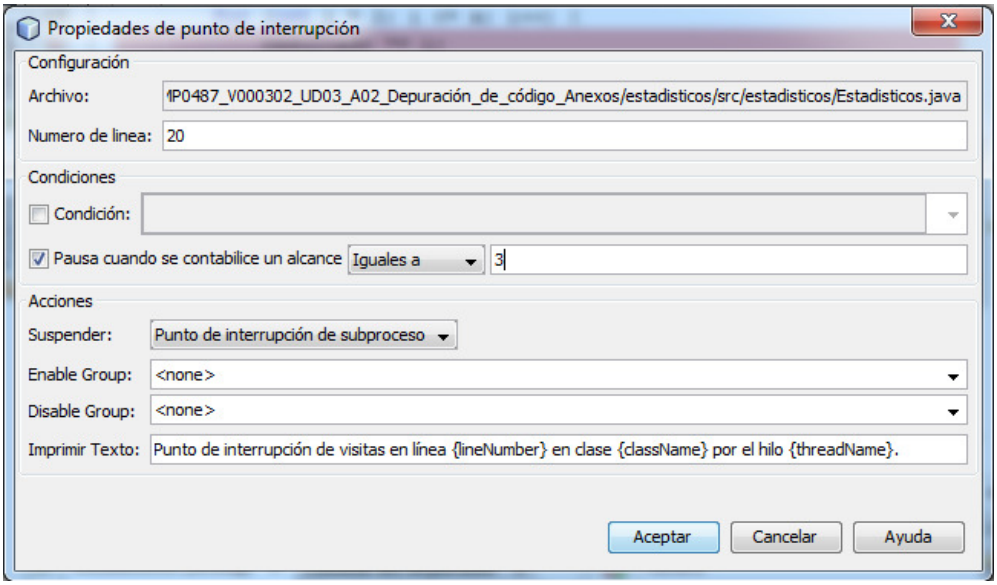
- La opción Propiedades muestra la ventana Propiedades del punto de interrupción, donde puede ajustar la configuración del punto de interrupción.
- También se puede crear un nuevo punto de interrupción.

Poner las condiciones en el punto de equilibrio

Puede establecer condiciones para que un punto de interrupción detenga la depuración una vez; Algunos son comunes a todos los puntos de interrupción, mientras que otros dependen de si son hilos, clases o excepciones.

Condiciones válidas para todos los puntos de interrupción

Todos los puntos de interrupción pueden suspender la depuración de acuerdo con la frecuencia establecida, seleccione la casilla de verificación "Pausa" al calcular el rango, seleccione una condición (igual, mayor que, igual) de la lista desplegable y establezca un valor numérico para la condición en la ventana de propiedades del punto.
La siguiente figura muestra cómo se activa el punto de interrupción de línea en la línea 20 de statistics.java cuando se pasa por tercera vez en la sesión de depuración.

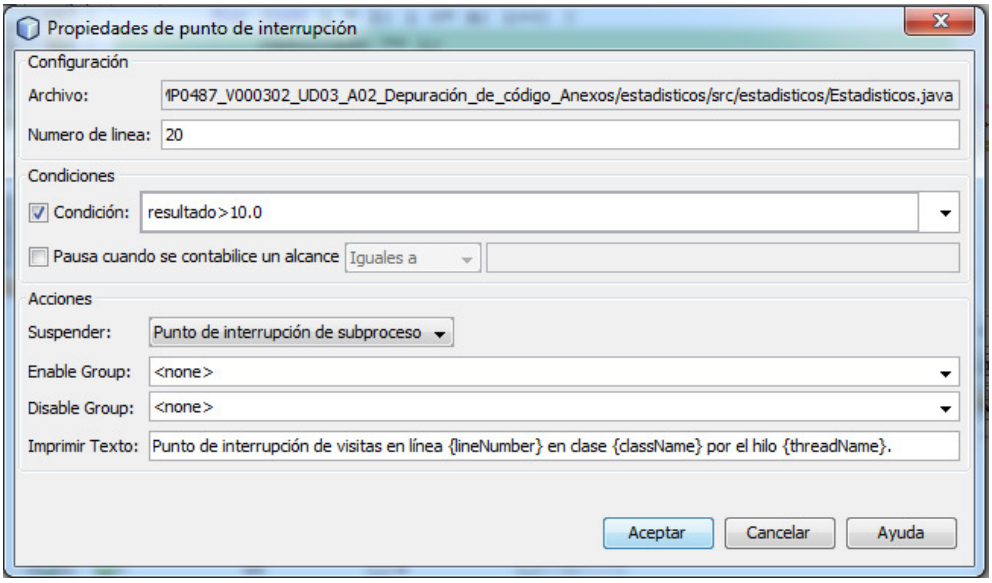


Condiciones válidas en todos los puntos excepto el tipo de rosca

Los puntos de interrupción de tipo no subproceso tienen la posibilidad de suspender la depuración cuando una condición es verdadera. Esta condición se establece en la ventana Propiedades del punto de interrupción seleccionando Condición y escribiendo Condición. Las condiciones deben seguir las reglas de sintaxis de Java y pueden incluir variables y métodos utilizados en el contexto actual, con las siguientes excepciones:

- Se ignoran las importaciones. Debe utilizar el nombre completo, por ejemplo obj instanceof java.lang.String
- No se puede acceder directamente a métodos y variables de clases externas. Debe utilizar this.name o this$1

La siguiente figura muestra las condiciones bajo las cuales el punto de interrupción de línea en la línea 20 de Statistics.java se activa durante una sesión de depuración cuando el valor de la variable Resultados es mayor que 10.



Las condiciones específicas son puntos extra y excepciones para la clase

Se pueden dar las siguientes condiciones específicas:

- Los puntos de interrupción de clase también permiten la exclusión de una clase como condición.
- Los puntos de interrupción excepcionales permiten como condiciones los filtros de clases que se van a incluir o excluir.

Misión 3.8. Depurar utilizando puntos de interrupción.

La tarea es realizar la siguiente depuración usando líneas, métodos, líneas condicionales y puntos de interrupción del contador:

Parte 1) Defina un punto de interrupción en la línea 21 de Main.java. Ejecute una depuración y vea la salida del resultado. Completar la puesta en servicio. Deshabilite este punto de interrupción.

Parte 2) Definir un punto de interrupción que pare a depuración cada vez que se sae do método factorial() de Estadisticos.java. Ejecute la depuración, vea cómo cambia la pila en la ventana de pila de llamadas cada vez que salga del método y vea los valores de los elementos x y result en la ventana Observaciones. Eliminar ese punto de interrupción.

Parte 3) Establecer un punto de interrupción para detener la ejecución en la línea 22 para los valores de m=4 y n=2 Statistics. java.

Parte 4) Eliminar tódolos puntos de interrupción.

	Step Over (F8) Ejecuta una línea de código. Si la instrucción es una llamada a un método, ejecuta el método sin entrar dentro del código del método.
	Step Into (F7) Ejecuta una línea de código. Si la instrucción es una llamada a un método, salta al método y continúa la ejecución por la primera línea del método.
	Step Out (Ctrl + F7) Ejecuta una línea de código. Si la línea de código actual se encuentra dentro de un método, se ejecutarán todas las instrucciones que queden del método y se vuelve a la instrucción desde la que se llamó al método.
	Run to Cursor (F4) Se ejecuta el programa hasta la instrucción donde se encuentra el cursor.
	Continue (F5) La ejecución del programa continúa hasta el siguiente breakpoint. Si no existe un breakpoint se ejecuta hasta el final.
	Termina la sesión del depurador (mayúsculas + F5). Termina la depuración del programa.

Misión 3.8b. Encuentra errores de depuración

Crear un proyecto Java con la clase main en Netbeans, el código es el siguiente:

```
Teclado del escáner = nuevo escáner (System.in);
int pos = 0, contPosiciones = 0;

System.out.print("Introducir una cadena: ");
String texto=teclado.next line ();
System.out.print("Introduce un caracter: ");
char caracter=teclado.next (). charAt(0);

 De {
pos=text.indexOf(carácter, pos);
if (pos != -1) contPosiciones++; //si la encuentra, la cuenta
pos++; //pasa a la siguiente posición
} while(Posición!=-1); //termina cuando no la encuentre más

System.out.print("El caracter aparece " + contPosiciones + " veces");
```

El programa está diseñado para contar el número de veces que un carácter ingresado aparece en una cadena de entrada, aparentemente bien hecho, pero está "envolvido". Depurar hasta que se encuentre el error.

.

3. Pruebas unitarias

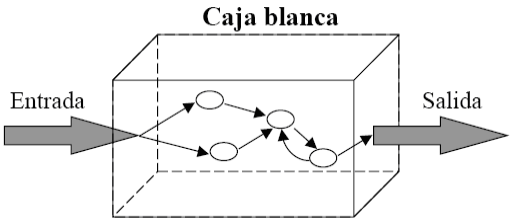
3.1 Habilidades de diseño de casos de prueba

El diseño de los casos de prueba está limitado por la incapacidad de probar completamente el software. Por ejemplo, si queremos probar todos los valores sumables en un programa en el que se suman dos números enteros de dos dígitos (de 0 a 99), deberíamos probar 10.000 combinaciones diferentes (variaciones repetidas de 100 elementos, de 2 por 2 = 100 por 2), y también debemos probar todas las posibilidades de error al ingresar datos (como escribir una letra en lugar de un número).

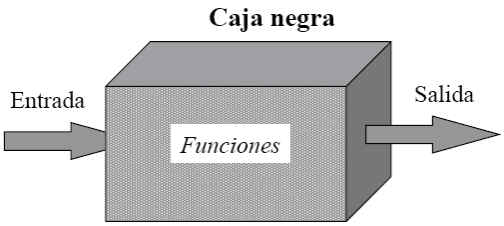
Las técnicas de diseño de casos de prueba pretenden obtener un nivel de confianza aceptable de que los defectos existentes serán detectados, ya que la seguridad completa solo se puede obtener a partir de pruebas completas, lo que no es factible sin consumir recursos excesivos. Toda la disciplina del juicio debe equilibrar la disponibilidad de recursos y la confianza que genera el caso para detectar las deficiencias existentes.

Como no es posible realizar pruebas exhaustivas, la idea básica para diseñar casos de prueba es elegir algunos de ellos, debido a sus características, que se consideran representativos de otros. La dificultad de este tipo de pensamiento radica en saber cómo seleccionar los casos que deben ser ejecutados, ya que la selección puramente aleatoria no proporciona mucha confianza para detectar los errores que existen. Hay tres formas principales de diseñar situaciones que no se excluyen mutuamente que se pueden combinar para lograr una detección de defectos más eficiente:

- Método estructural o de caja blanca también conocido como método de caja de cristal: se centran en la ejecución del programa para seleccionar casos[2] de prueba.



- Método funcional o de caja negra: consiste en estudiar la especificación de la función y sus entradas y salidas.[3]



- Método estocástico: utiliza modelos, generalmente estadísticos, que representan las posibles entradas del programa, creando así casos de prueba.

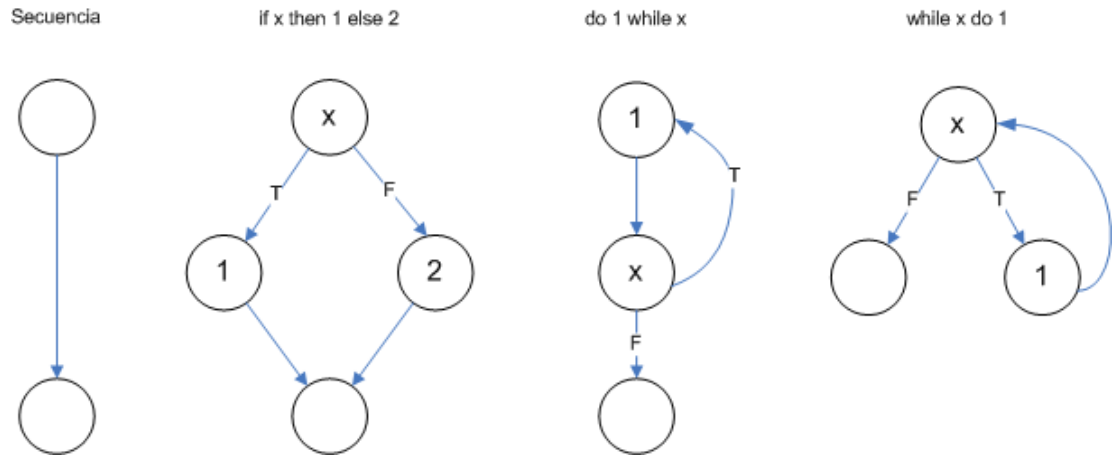
[2] Imaxe extraída de http://osl.uca.es/wikihaskell/index.php/Pruebas

[3] Imaxe extraída de http://osl.uca.es/wikihaskell/index.php/Pruebas

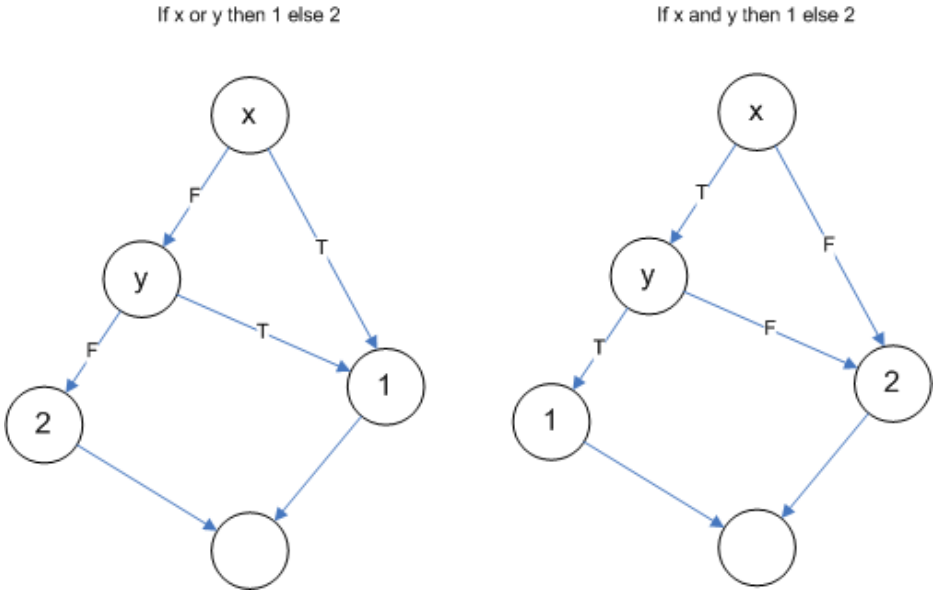
3.2 Pruebas de unidades estructurales

Para utilizar técnicas estructuradas, realizaremos un diagrama de flujo del programa o función a probar. Esto no es estrictamente necesario, pero dibujarlos ayuda a entender el funcionamiento de las técnicas de prueba de caja blanca. Los grafos que se utilizarán serán fuertemente conectados, es decir, siempre habrá un camino entre el par arbitrario de nodos seleccionado, y para remediar que el primer y último nodos estén conectados directamente, se añadirá un arco ficticio en lugar de uno.

Diagrama de flujo básico:



La imagen muestra múltiples condiciones:



Criterio de cobertura lógica de Miles

El diseño del caso debe basarse en la selección de rutas importantes que proporcionen una seguridad aceptable cuando se detectan defectos. Se utilizan los denominados criterios de cobertura lógica. Los criterios de cobertura lógica de Myers se muestran en orden ascendente, por lo que el coste económico es:

- Cobertura de oraciones: generar los casos de prueba necesarios para ejecutar cada oración al menos una vez.

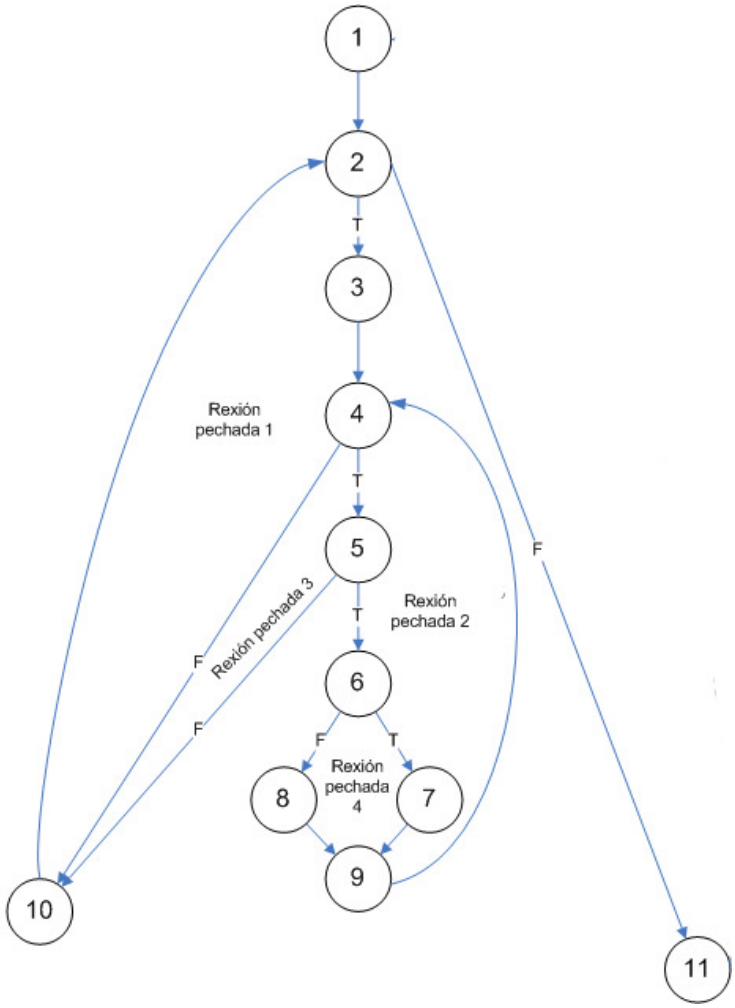
- Cobertura de decisiones: generar casos de manera que cada decisión tenga al menos una vez un resultado verdadero y al menos una vez un resultado falso.
- Cobertura condicional: generar los casos de prueba requeridos para que cada condición de cada decisión adopte al menos una vez un valor verdadero y al menos una vez un valor falso. Por ejemplo, decidir: si ((a==3)(b==2)) tiene dos condiciones: (a==3) y (b==2).

▪ Criterios de Decisión/Condición: Se requiere que las condiciones cubran los criterios y se cumplan obligatoriamente los criterios de decisión.

- Criterio de condiciones múltiples: Cuando se considera que la evaluación condicional de cada decisión no se realiza simultáneamente, se puede considerar descomponer cada decisión de condiciones múltiples en varias decisiones de condición única.
- Criterio de cobertura de ruta: cada ruta posible del programa debe ejecutarse al menos una vez. Una ruta se define como una secuencia de declaraciones en serie desde la declaración inicial hasta la declaración final de un programa. El número de rutas, incluso en un pequeño programa, puede ser poco práctico para las pruebas. Para reducir el número de rutas de prueba, se puede usar la complejidad de anillo de McCabe.

Ejemplo de diagrama de flujo fuertemente conectado de una pieza de pseudocódigo:

```
Abrir archivo ventas;                                              1
Ler archivo ventas;
while (haxa rexistro){ 2
        totalnacional=0;
        totalforaneo=0;                                            3
        while (haxa rexistro) 4 e (mesmo produto)5 {
                if (nacional){ 6
                        totalnacional=totalnacional+ventanacional; 7
                }
                else{
                        totalforaneo=totalforaneo+ventaforaneo;    8
                }
                Ler archivo ventas;                                9
        }
        Imprimir liña de listado;                                  10
}
Pechar archivo ventas;                                             11
```



Las pistas para dibujar los gráficos, puedes utilizarlos como un patrón de lista de verificación, es decir, a medida que completes cada ejercicio, puedes comprobar si todos están completos.

1. Un programa tendrá un punto de entrada y un punto de salida.
2. Cada nodo incondicional tendrá un borde de salida (excepto los nodos finales que no tienen)
3. Cada nodo condicional (if, for, while, etc.) tendrá dos bordes de salida, uno con " verdadero" y el otro con "falso".
4. Un nodo de inicio de bucle (for, while, etc.) tendrá al menos dos bordes de entrada (borde de entrada "normal" y borde de retorno de bucle)
5. Una excepción (instrucción de arrojo) o'return' tendrá un borde de salida que apunta al nodo final del programa.
6. Tanto la estructura condicional como la estructura de bucle terminan con un nodo, que será la siguiente instrucción, o un nodo ficticio si no hay más instrucciones (será el atajo "}")

Complejidad del círculo de McCabe

La complejidad del bucle es una métrica que nos dice la cantidad de rutas independientes que tiene un diagrama de flujo. McCabe define un buen criterio de prueba como probar que el número de rutas independientes es igual a la métrica. Un camino independiente se refiere a cualquier camino que introduce al menos un nuevo conjunto de juicios o condiciones de proceso con respecto a un camino existente. En lo que respecta al diagrama de flujo, una trayectoria independiente está constituida por al menos un borde que no ha pasado en la trayectoria definida previamente. Al determinar las diferentes rutas, debe tenerse en cuenta que cada nueva ruta debe tener un número mínimo de nuevos juicios o nuevas condiciones en relación con la ruta existente. Pruébalo de esta manera, el proceso de depuración es más simple.

La complejidad de McCabe V(G) se puede calcular de las tres maneras siguientes, a partir de un gráfico de flujo fuertemente conectado G:

- V(G)=A-N+2
  - A: Número de arcos o lados
  - n: número de nodos
- V(G)=r+1
  - Respuesta: Número de áreas cerradas del gráfico.
- V(G)=c+1
  - c: número de nodos de condición

Por ejemplo, la complejidad de bucle de McCabe para el diagrama de flujo del bloque de pseudocódigo anterior es de 5, es decir, es el valor resultante de cualquiera de las tres fórmulas anteriores:

- V(G) = 14 – 11+2=5
- V(G) = 5
- V(G) = 4+1

Una vez que se calcula la complejidad del bucle, tenemos que elegir tantos caminos independientes como los valores de complejidad. Para seleccionar estos caminos, primero definimos un camino inicial y luego creamos nuevos caminos cambiando el camino inicial lo menos posible. Para ejecutar una ruta, es posible que deba conectarla con otras rutas.

Para el ejemplo anterior tendremos 5 trayectorias representadas por el número de nodos del gráfico recorrido y usando puntos de pausa después de recorrer el nodo de control:

- 1-2-11
- 1-2-3-4-10-2-...
- 1-2-3-4-5-10-2-...
- 1-2-3-4-5-6-7-9-4-...
- 1-2-3-4-5-6-8-9-4-...

Misión 3.9. Dibujar diagramas de flujo, calcular la complejidad del bucle de McCabe y obtener rutas están disponibles con la herramienta en línea draw.io.

La tarea es dibujar un diagrama de flujo, calcular la complejidad de McCabe y detallar las rutas de los siguientes métodos:

Part1) Proyect_División El método calcularDivision del proyecto.

Parte 2) Método factorial del proyecto Projecto_Factorial.

Parte 3) Método de búsqueda de proyectos project_arrays.

Parte 4) Método para obtener el acrónimo del proyecto projecto_acronimos.

- El método calcularDivision recibe un dividendo y un divisor de tipo float y devuelve el resultado de un divisor del mismo tipo float si el divisor no es 0, en cuyo caso se lanza una excepción.

```
package proyecto_division;
División pública {
    public float calcularDivision(float dividendo, float divisor) lanza una
excepción {
        if (divisor ==0) {
            throw (new Exception("Error. El divisor no puede ser 0."));
        }
        Resultado de coma flotante = dividendo/divisor;
        El efecto de retorno;
    }
}
```

Para los propósitos del gráfico, "lanzar una nueva excepción" es como imprimir en la pantalla y ir directamente al final del programa.

- El método factorial recibe un número n de tipo byte y devuelve su factorial de tipo float, a menos que sea negativo, en cuyo caso lanza una excepción. El factorial de un número n es el producto de todos los números menores que él a 2. Los casos especiales de factoriales son factorial(0)=1 y factorial(1)=1.

```
package proyecto_factorial;
Factorial de clase común {
    El factorial común de coma flotante (byte n) lanza una excepción {
        If (n < 0) {
            throw new Exception("Error. El número tiene que ser >=0");
}
        Resultado de coma flotante = 1
        ; for(int i=2;i<=n;i++) {
Resultado *=i;}

        Devuelve los
resultados;}
}
```

- O método busca recibe un carácter c e un array de caracteres v de 10 elementos como máximo ordenados de forma ascendente. Devuelve un valor booleano verdadero o falso dependiendo de si el carácter se encuentra o no en la matriz. La búsqueda es dicotómica, es decir, la primera búsqueda considera toda la matriz, pero más adelante solo considera la comparación de un segmento del, c obtenido calculando el índice de la mitad del segmento. Un elemento se almacena en el índice; Si coincide, se termina la búsqueda y se encuentran los caracteres en la matriz; Si c es pequeño, el siguiente segmento es la primera mitad actual; Si c es más grande, entonces el siguiente párrafo será la segunda mitad del actual. Si el proceso termina con un segmento vacío y el carácter no se encuentra, entonces no existe.

```
package proyecto_arrays;
Clase pública OperationsArrays {
    Búsqueda booleana pública (char c, char [] v) {
        int a, z, m;
A = 0;
z = v. longitud -1;
        Resultado booleano = falso;
        while(a<=z&&result==false){
m = (A+z)/2;
            If (v[m]==c){
Resultado=verdadero;
            }
            else
            {
                If (v[m]<c){
A = m+1;
                }
                De lo contrario {
z=m-1;
                }
            }
        }
        El efecto de retorno;
    }}
```

- El método geterAcronimo recibe una cadena y devuelve una que contiene el acrónimo correspondiente. Un acrónimo consiste en el primer carácter de cada palabra seguido de un punto, cuando dicho carácter es diferente del espacio.

```
package proyecto_acronimos;
Abreviaturas de clase pública {
    Cadena pública obtenerAcronimo (Cadena cadena
) {Cadena resultado= "";
        Character Character;
        int n=cadena. longitud ();
        for(int i=0; i<n; i ++) {
Carácter=cadena.charAt(i);
            IF (character! = '') {
                If (i==0) {
resultado=resultado+caracter+'.'; }

                De lo contrario {
                    if(charAt(i-1)== '') {resultado=
resultado+carácter+'.';}

                }
            }
        }
        Devuelve los
resultados;}
}
```

3.3 Pruebas unitarias funcionales

¿Tomarías una prueba de caja blanca para pensar en probar un fragmento de código? La respuesta es no y prueba con el siguiente código:

```
Si ((x+y+z)/3==x)
print("X, Y, Z son iguales");
else print("X, Y, Z no son iguales");
```

Hay dos trayectorias posibles para los valores x=5, y=5, z=5 y x=4, y=3 y z=5, lo que confirma la validez del código, pero el código falla para los valores x=2, y=5 y z=3. De esto se deduce que se necesitan otros tipos de pruebas, tales como pruebas funcionales, para complementar las pruebas estructurales.

Las pruebas funcionales o pruebas de caja negra se centran en estudiar las especificaciones del software, analizando las funciones, entradas y salidas que pretende implementar. Las pruebas funcionales exhaustivas también a menudo son poco prácticas debido a la existencia de diversas técnicas de diseño de carcasa de caja negra.

Clases de equivalencia

Cada caso equivalente deberá abarcar el número máximo de entradas. Se debe tratar un rango de valores de entrada dividido en un número finito de clases de equivalencia que cumplen las siguientes condiciones: La prueba de los valores representativos de una clase permite suponer " razonablemente" que los resultados obtenidos (con o sin defectos) serán los mismos que los obtenidos mediante la prueba de los valores de cualquier otra clase. La forma de diseñar casos de uso es identificar clases de equivalencia y crear casos de prueba correspondientes.

Echemos un vistazo a algunas reglas que nos ayudan a identificar clases de equivalencia, teniendo en cuenta las limitaciones de los datos que pueden entrar en el programa:

Al especificar un rango de valores para los datos de entrada, por ejemplo "El número estará entre 1 y 49", se creará una clase válida: 1 < = número < = 49 y dos clases inválidas: número < 1 y número > 49.

- La especificación de varios valores para los datos de entrada, por ejemplo “código de 2 a 4 caracteres”, creará una clase válida: 2 < = número de caracteres del código < = 4, y dos clases inválidas: menos de 2 caracteres y más de 4 caracteres.
- En el caso de “Debe ser” o de tipo booleano, como “El primer carácter debe ser una letra ”, se reconocerá una clase válida: es una letra y otra clase inválida: no es una letra.

- Al especificar un conjunto de valores de soporte que el programa trata de manera diferente, se creará una clase para cada valor válido y otro no válido. Por ejemplo, si tenemos tres tipos de bienes raíces: apartamentos, villas y locales comerciales, haremos una clase de equivalencia para cada valor, mientras que una clase de invalidez representa cualquier otro caso, como una plaza de garaje.
- En cualquier caso, si existe la sospecha de que algunos elementos de una clase no son tratados de la misma manera que otros elementos de esa clase, deben ser divididos en clases de equivalencia menores.

Para crear un caso de prueba, siga estos pasos:

- Asignar un valor único a cada clase de equivalencia.

- Escribir casos de prueba que cubran todas las clases de equivalencia válidas que no estaban incluidas en casos de prueba anteriores.
- Escribir un caso de prueba para cada clase inválida hasta cubrir todas las clases inválidas. Esto se hace porque si juntamos varias clases de equivalencia inválidas, puede detectar un error en una de ellas, lo que hará que el resto ya no se revise.

Por ejemplo: una aplicación bancaria donde el operador proporciona un código de área (3 dígitos que no comienzan con 0 o 1), un nombre para identificar la operación (6 caracteres) y una orden que activará una serie de funciones bancarias ("cheque", "depósito", "pago de facturas" o "retiro de fondos").
Todas las clases numeradas son:

Entrada	Clases válidas	Clases inválidas
Código Área	(1) 200 < = código < = 999	(2) Código < 200 (3) Código > 999
Operación de reconocimiento de nombres	(4) 6 caracteres	(5) menos de 6 caracteres (6) más de 6 caracteres
Orden	7) "Cheques" 8) "Depósito de seguridad" (9) "Pago por factura" (10) "Retiro de fondos"	11) "Moneda"

Caso de prueba, suponiendo que el orden de entrada de datos es: orden de nombre de código es el siguiente:

- Casos válidos:

Código	Apellidos	Orden	Clases
200	Nómina	Cheques	(1) (4) (7)
200	Nómina	Depósitos	(1) (4) (8)
200	Nómina	Pago de facturas	(1) (4) (9)
200	Nómina	Retiro de fondos	(1) (4) (10)

- Casos no válidos:

Código	Apellidos	Orden	Clases
180	Nómina	Cheques	(2)
1032	Nómina	Cheques	(3)
200	Nomin	Cheques	(5)
200	Nóminas	Cheques	(6)
200	Nómina	Moneda	(11)

Análisis de límites (AVL)

Se ha encontrado empíricamente que los casos de prueba que exploran las condiciones de contorno del programa dan mejores resultados para la detección de defectos. Podemos definir condiciones de contorno para las entradas, por ejemplo, situadas directamente encima, debajo y en el borde de una clase de equivalencia, y dentro del rango de valores permitidos para tales tipos de entradas.

Podemos establecer condiciones de contorno para las salidas, por ejemplo, en las posibles salidas ocurren situaciones que conduzcan a valores de contorno. Para la aplicación de AVL, es necesario usar ingenio para considerar todos los aspectos, a veces sutiles sutilezas. Algunas reglas para generar casos de prueba:

- Si se especifica un rango de valores para una entrada, se debe generar una mayúscula válida para los puntos finales de ese rango y una mayúscula inválida para los casos justo más allá del punto final. Por ejemplo, clase de equivalencia: "-1.0 < = valor < = 1.0", caso válido: -1.0 y 1.0, caso inválido: -1.01 y 1.01, en el caso de soportar 2 decimales.
- Si se especifican varios valores para una entrada, se debe escribir mayúsculas y minúsculas para el máximo, el mínimo, el máximo más uno y el mínimo menos uno. Por ejemplo: "El archivo de entrada tendrá de 1 a 250 registros", caso válido: 1 y 250 registros, caso inválido: 251 y 0 registros.
- Al especificar un rango de valores para la salida, se intentará escribir la mayúscula y minúscula para manejar restricciones en la salida. Por ejemplo: "El programa puede mostrar de 1 a 4 listas", caso válido: 1 y 4 listas, caso inválido: 0 y 5 listas.

Si desea especificar múltiples valores para la salida, debe intentar escribir una instancia para manejar las restricciones en la salida. Por ejemplo: "El descuento máximo será 50% y el mínimo 6%", Casos válidos: 50% y 60%, Casos no válidos: 5,99% de descuento, 50,01% si el descuento es real.

- Si la entrada o salida es un conjunto ordenado (por ejemplo, una tabla, un archivo secuencial...), el caso debe centrarse en el primer y último elemento.

Conjetura errónea:

La idea básica de esta técnica es hacer una lista de errores que un programador puede cometer o situaciones en las que es propenso a cometer ciertos errores, y generar casos de prueba a partir de esta lista. Esta técnica también se conoce como generación de casos especiales (o numéricos) porque no se obtienen de acuerdo con otros métodos sino a través de la intuición o la experiencia . Algunos valores numéricos a considerar en casos especiales pueden ser los siguientes:

- El valor 0 es propenso a generar errores en la salida y la entrada.
- En el caso de ingresar un número desigual de valores, como las listas, se recomienda centrarse en los casos en los que no se ingresa ningún valor y un único valor. También puede ser interesante que todos los valores sean iguales.
- Se sugiere que el programador puede haber malinterpretado algo de la especificación.
- También le interesa imaginar las acciones que realiza el usuario al introducir una entrada, aunque quiera romper el programa. Si el valor de entrada está fuera del rango de umbral permitido por este tipo de variable, puede verificar el comportamiento del programa. Por ejemplo, si la variable de entrada es de tipo int, debe comprobar qué sucede si el valor de entrada está fuera del rango de valores permitido por int, o incluso si tiene decimales, o es una letra. Lo que se podría verificar es que en la entrada no se disfrazarían códigos de peligro. Por ejemplo, compruebe las entradas de la base de datos para posibles inyecciones de código.

Completar las pruebas de caja blanca y caja negra en situaciones de ciclo. Buscar un bucle se ejecuta 0, 1 o más veces. Si se conoce el número máximo de iteraciones de un bucle (n), el bucle debe ejecutarse 0, 1, n-1 y n veces. Si hay bucles anidados, los casos de prueba crecerán exponencialmente, por lo que se recomienda comenzar con un bucle de prueba

Mantenga más interior en el exterior con iteraciones mínimas y cree casos de prueba fuera del anidado.

Misión 3.10. Definir clases de equivalencia, realizar análisis de valores límite y conjeturas de error. Esta tarea consiste en definir clases de equivalencia, realizar análisis de valores límites y realizar suposiciones de error para:

Parte1) Proyect_División El método calcularDivision del proyecto

. Parte 2) Método factorial del proyecto Projecto_Factorial.

Parte 3) Método de búsqueda de proyectos project_arrays.

Parte 4) Método para obtener el acrónimo del proyecto projecto_acronimos.

3.4 Pruebas unitarias aleatorias

En una prueba aleatoria, las entradas habituales del programa se simulan mediante la creación de datos, de manera repetitiva (se prueban muchas combinaciones), introduciéndolas en el orden y la frecuencia que pueden ocurrir en la práctica diaria. A menudo se utiliza una herramienta llamada Generador de casos de prueba automatizado para esto.

3.5 Método recomendado de diseño de casos

Las diferentes técnicas utilizadas para diseñar casos de prueba representan diferentes enfoques. El enfoque sugerido consiste en combinar estas técnicas para alcanzar un nivel de prueba " razonable". Por ejemplo:

Desarrollar casos de prueba de caja negra para entradas y salidas clases de equivalencia Válido (CEV) e inválido (CEI), rellenar la co Análisis de valores límite (AVL) e coa Conjetura errónea (CE).

- Diseñar casos de prueba de caja blanca basados en El camino a la prueba de carretera Básic os (PCB) para completar los casos de prueba de caja negra. En muchos casos, estas pruebas coincidirán con las clases de equivalencia de los puntos anteriores.

La mejor manera de dar forma a un caso de prueba sería una tabla con las siguientes columnas:

- Número de caso de proba (simplemente un número secuencial, 1,2,3...para logo identificalo en JUnit).

C) Tipos de juicios (a partir de lo anterior, incluso un caso puede abarcar varios tipos).

- Los valores de entrada correspondientes a las pruebas requeridas.

Valor de salida esperado.

Tarea 3.11 Preparación de casos de prueba. La tarea es diseñar casos de prueba para: Parte 1) Método calcularDivision del proyecto Project_Division.

Parte 2) Método factorial del proyecto Projecto_Factorial.

Parte 3) Método de búsqueda de proyectos project_arrays.

Parte 4) Método para obtener el acrónimo del proyecto projecto_acronimos.

3.6 Junit

JUnit es un marco o colección de clases Java que permite a los programadores automatizar la construcción y ejecución de casos de prueba para métodos de clase Java. Los casos de prueba se reflejan en programas Java que se archivan y se pueden volver a ejecutar tantas veces como sea necesario. Estos casos de prueba le permiten evaluar si el comportamiento de la clase es como se espera, es decir, a partir de algunos valores de entrada, evaluar si los resultados resultantes son como se espera. Escrito por Erich Gamma y Kent Beck, JUnit es un producto de código abierto distribuido bajo la Licencia Pública General -v 1.0. La página oficial de JUnit es: https://junit.org/junit5/

IDE como NetBeans y Eclipse tienen complementos que utilizan JUnit, lo que permite al programador centrarse en las pruebas y los resultados esperados, mientras que deja al IDE responsable de crear clases que permiten las pruebas.

JUnit5 tiene una estructura diferente de sus versiones anteriores. Ahora ya no es una sola biblioteca, sino una colección de tres subproyectos: JUnit Platform, JUnit Jupiter y JUnit Vintage.

- La plataforma JUnit es la base que nos permite lanzar frameworks de prueba en la JVM, además de esto se encarga de proporcionarnos la capacidad de lanzar la plataforma desde la línea de comandos y plugins para Gradle y Maven.
- JUnit Jupiter es la herramienta que más utilizamos cuando programamos. Vamos a usar el nuevo modelo de programación para escribir nuevas pruebas de JUnit 5.

JUnit Vintage se encarga de probar la compatibilidad de Junit 3 y 4.

En este documento, JUnit5 se utilizará para el IDE NetBeans 11, y las pruebas en esta sección se realizarán en el proyecto sencillo projecto_division que consiste en las clases Division.java y Main.java (ver archivo adjunto).

Generar pruebas en JUnit

Hay tres tipos de pruebas que se pueden generar en NetBeans: métodos para probar clases, conjuntos de pruebas para probar un conjunto de clases en un paquete o un caso de prueba JUnit vacío.

Para crear una prueba JUnit, puede seleccionar Nuevo archivo en la opción Archivo del menú principal, seleccionando la categoría de pruebas de unidad y el tipo de archivo de las siguientes tres posibilidades:

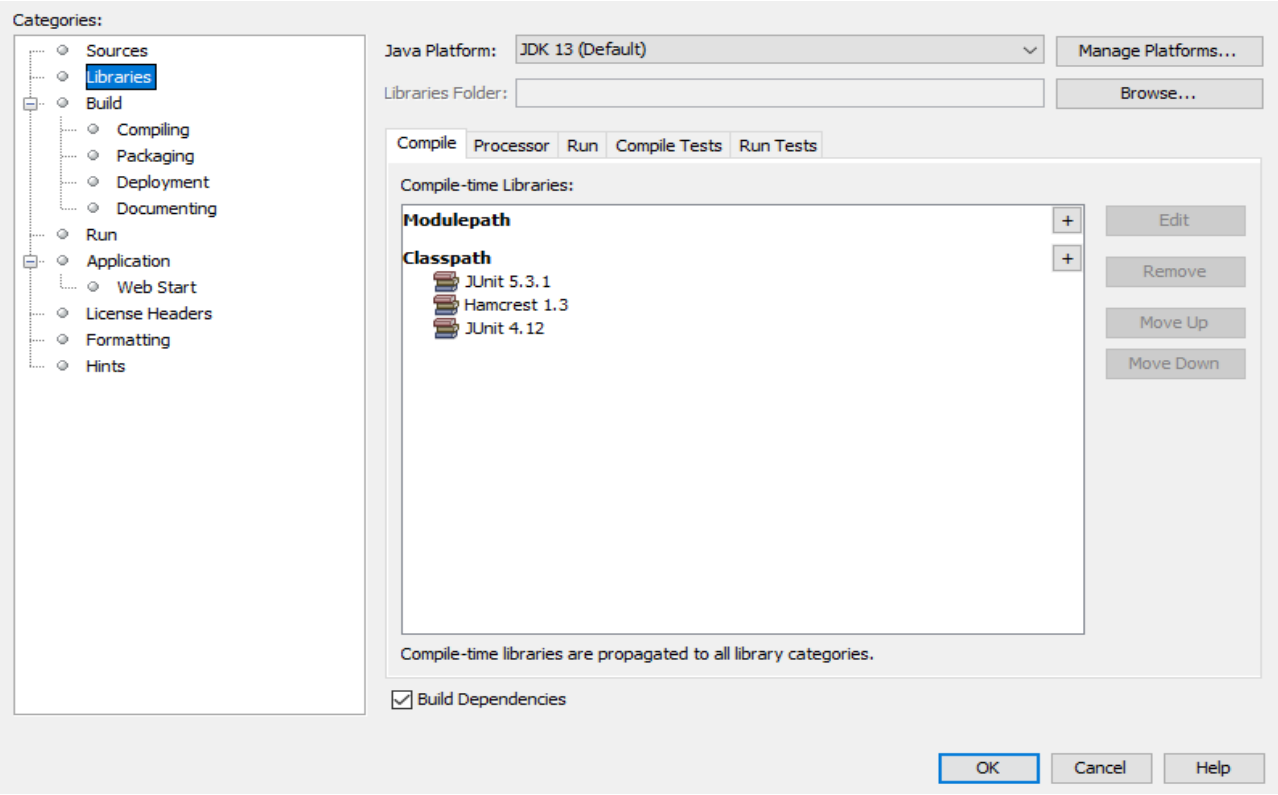
Crear una prueba JUnit para un caso de prueba vacío

Para las pruebas de clases existentes, crea un caso de prueba para los métodos de la clase

- *Crear una suite de pruebas para los paquetes Java seleccionados (esta opción ya no es compatible en JUnit5, por lo que hay que usar el módulo junit5.vintage e incrustar las dependencias).*

Dependiendo del tipo de prueba que elija, tendrá que completar la siguiente ventana emergente de diferentes maneras.

Importantes proyectos "Ant": En el caso de proyectos "Ant", el primer paso es ir a las propiedades del proyecto (hacer clic derecho en el nombre del proyecto en la ventana del proyecto) y añadir 3 como se muestra en la siguiente imagen en la opción lateral "Biblioteca", pero haremos proyectos Marvin.



Para proyectos de Maven, debemos incluir lo siguiente en el archivo de configuración pom.xml:

```
< Construir >
    < Plugin >
        < Plugin >
            < artifactId>maven-surefire-plugin</artifactId
            ><version>2.22.2</version >
        < /plugin
    ></plugins >
< /build >
```

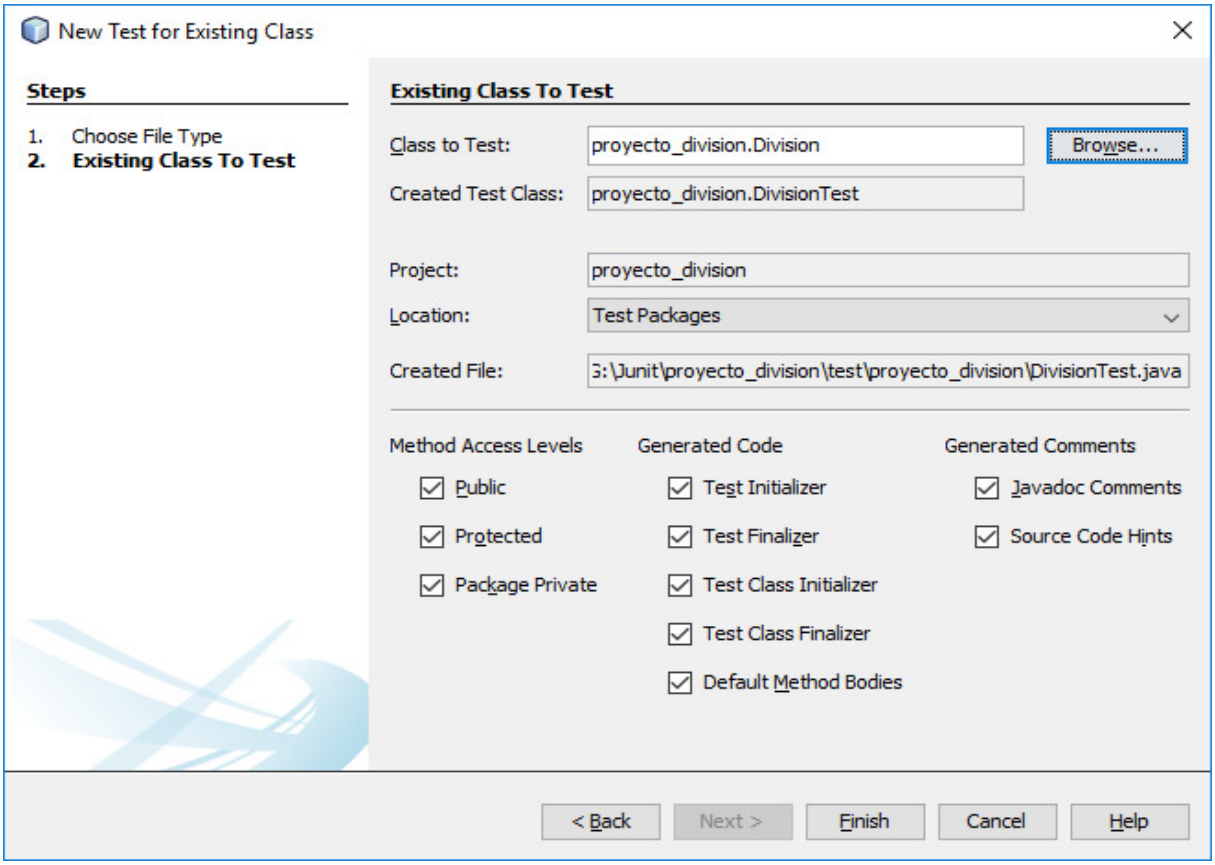
Para una clase

Seleccione desde el menú principal *Archivo-> Nuevo archivo-> Pruebas unitarias-> Prueba clases existentes* , aparecerá la ventana *Clase existente a la prueba Debe indicar: el nombre de la clase de prueba (se recomienda mantener el nombre predeterminado: el nombre de la clase a probar seguido de Prueba), la ubicación donde guardar la prueba (se recomienda mantener el valor predeterminado) y las características de generación de código se dividen en tres secciones:*

La sección de nivel de acceso del método se usa para indicar el nivel de acceso al método de prueba.

- La sección de código de generación indica si la prueba es antes de comenzar la prueba (inicializador de prueba), después de finalizar la prueba (finalizador de prueba) o código de muestra de clavo de prueba (cuerpo de método predeterminado).
- La sección de Comentarios Generados se utiliza para indicar que las pruebas lleven anotaciones y comentarios Javadoc para sugerir cómo implementar métodos de prueba (consejos de código fuente).

Deje todo seleccionado y presione Finalizar.



El código generado por las pruebas de NetBeans por defecto para el método calcularDivision () solo considera un caso de prueba (0/0=0). Puede modificar este método y agregar nuevos casos de prueba o agregar otros métodos de prueba. NetBeans recomienda eliminar las últimas dos líneas de código del comentario de prueba. Excepto el comentario de prueba y la línea de importación correspondiente al comentario de eliminación, las líneas en el comentario pueden ser eliminadas o comentadas, de lo contrario no se utilizarán.

```
package proyecto_division;

Importar org.junit.jupiter.api.AfterEach
Importar org.junit.jupiter.api.AfterAll;
Importar org.junit.jupiter.api.BeforeEach;
Importar org.junit.jupiter.api.BeforeAll;
Importar org.junit.jupiter.api. Pruebas;
Importar org.junit.jupiter.api.Assertions.*
estático; Prueba de partición de clase pública {

    Prueba de división pública () {

    }

@BeforeAll
    public static void setUpClass () {

    }

@ Después de todo
    public static void tearDownClass () {

    }

@BeforeEach
    Público void setUp () {

    }
```

```
@AfterEach
   public void tearDown () {

    }

@prueba
    public void testCalcularDivision () lanza una excepción {
System.out.println ("División cálculo");
        Dividendo flotante = 0.0F;
        Divisor de coma flotante = 0,0f;
División instance=nueva División ();
        Expresión de coma flotante = 0,0F;
        Resultado de coma flotante = instance.Calcular Division (
dividendo, divisor); assertEquals(expResult, resultado, 0.0);
        //TODO Mira el código de prueba generado y elimina el llamada por defecto falla.
Falla ("El caso de prueba es un prototipo.");
    }
}
```

Estructura de prueba

Los comentarios y métodos que aparecen por defecto son:

- La anotación @BeforeAll marca el método setUpClass () como ejecutado antes del inicio de la prueba de clase, es decir, solo una vez y antes de que el método de prueba comience a ejecutarse. Por ejemplo, se puede utilizar para crear una conexión a una base de datos. Es un método estático.
- La anotación @AfterAll marca el método tearDownClass () a ejecutar al final de la prueba de clase. Por ejemplo, se puede utilizar para cerrar una conexión previamente realizada a la base de datos. Es un método estático.
- La anotación @BeforeEach marca al método setUp () como ejecutado antes de cada método de prueba, es decir, cuántas veces se ejecutará mientras exista un método de prueba. Se utiliza para inicializar recursos, variables de clase o propiedades que son iguales para todas las pruebas.
- La anotación @AfterEach marca el método tearDown () para ser ejecutado inmediatamente después de la ejecución de cada método de prueba.

La anotación @Test marca cada método de prueba. A menudo habrá varios, cada uno con un nombre diferente. Por defecto, no tienen que ejecutarse en el orden en que los escribimos.

Podemos cambiar los nombres predeterminados de todos los métodos, importantes y anteriores comentarios, para que sepamos cuándo se ejecutarán.

Método assertEquals. Este método declara que el primer parámetro (resultado esperado) es igual al segundo parámetro (resultado obtenido). Si ambos parámetros son verdaderos, entonces puede haber un tercer parámetro, llamado el valor delta, que es un número real igual a la diferencia máxima absoluta entre el valor esperado y el valor real, por lo que la afirmación tiene éxito.

```
Math.abs (expectativa-adquirido) < delta
```

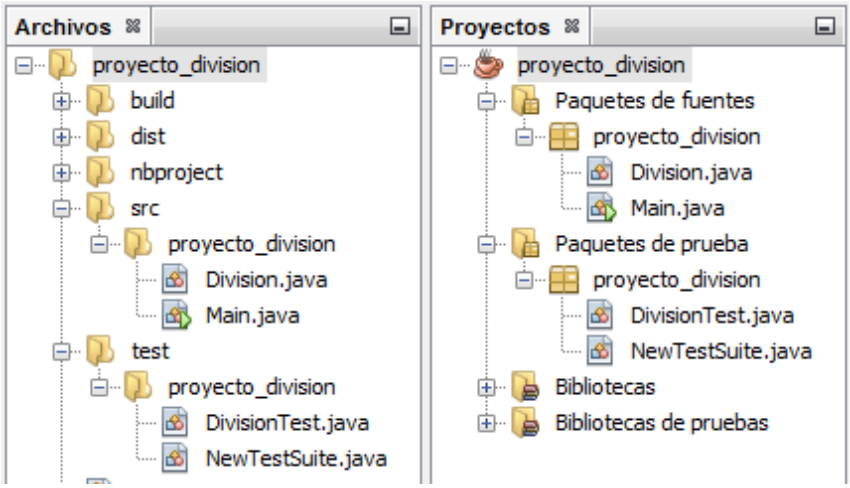
En la prueba del método calcularDivision () utilizado en el ejemplo, la prueba puede modificarse para verificar que el resultado de la división entre 1 y 3 es 0.33 y el valor delta es 1E-2. Si asumimos que el método calcularDivision(1.3) devuelve 0.333, los valores esperados de 0.34 y 0.33 serán comparables a los valores reales, mientras que los valores de 0.32 no, ya que abs(0.333-0.33)<0.01, abs(0.333-0.34)<0.01 y abs(0.333-0.32)>0.01. El código de prueba puede ser:

```
@prueba
```

```
    public void testCalcularDivision () lanza una excepción {
System.out.println ("Caso: 1/3 = 0,33, valor delta 1E-2");
División instance=nueva División ();
        Resultado de coma flotante = instancia. Calcula la división (1F, 3F);
assertEquals(0.33, resultado, 1E-2);
    }
}
```

Carpeta de prueba

En las ventanas Archivos y Proyectos, puede ver la estructura lógica y física de la carpeta en la que se almacena las pruebas JUnit. Puede agregar carpetas de prueba adicionales en el cuadro de diálogo Propiedades del proyecto, pero tenga en cuenta que el archivo de prueba y la fuente no pueden estar en la misma carpeta.



Ejecutar la prueba

Pasos para ejecutar la prueba completa del proyecto:

- Seleccione cualquier nodo o archivo del proyecto en la ventana Proyecto o en la ventana Archivo y en el menú principal seleccione Ejecutar-> Proyecto de prueba (nombre_do_ proyecto) o pulse Alt-F6.
- El IDE ejecuta todos los métodos de prueba del proyecto. Si desea ejecutar un subconjunto de las pruebas del proyecto, o ejecutar las pruebas en un orden específico , debe crear una suite de pruebas que especifique las pruebas que se ejecutarán.

Para ejecutar la prueba de la clase, puede seleccionar uno Hay dos posibilidades:

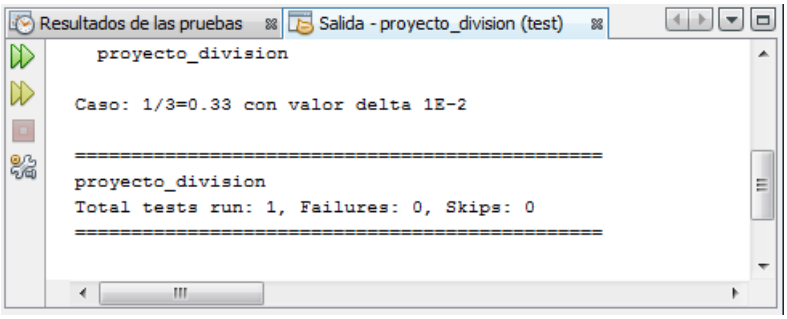
- Seleccione la clase en la ventana Proyectos o en la ventana Archivos, haga clic con el botón derecho y seleccione Archivo de prueba o pulse Ctrl-F6.
- Selecciónase a proba da clase e elíxese no menú principal Ejecutar -> Ejecutar archivo ou prémese Mayús-F6.

Pasos para ejecutar un caso de prueba:

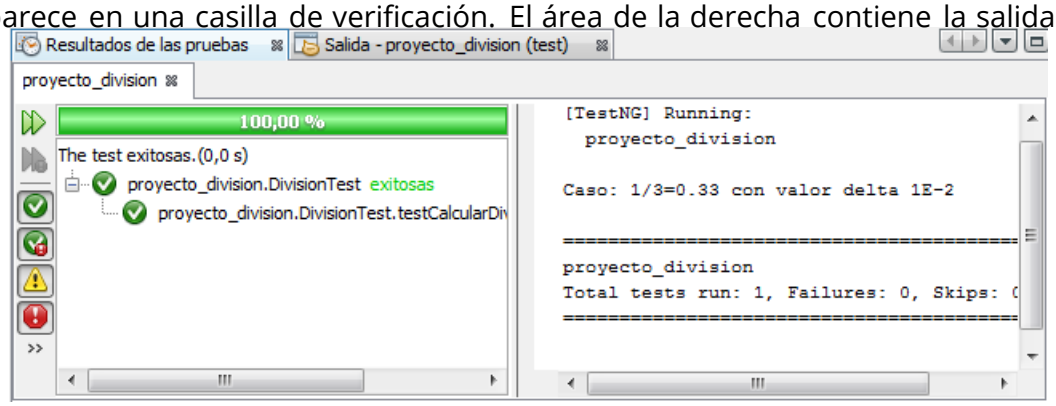
- Ejecutar una prueba que contiene este caso de prueba.
- Haga clic derecho en el método en la ventana de resultados y seleccione Ejecutar nuevamente.

Ventana de salida y resultados

La ventana de salida refleja los detalles del proceso de ejecución de prueba en formato de texto.



La ventana de resultados tiene dos áreas. El área de la izquierda contiene un resumen de los casos de prueba aprobados y no aprobados, así como una descripción de ellos en formato gráfico. Cuando pasa el mouse sobre la descripción de un método de prueba, la salida correspondiente al método aparece en una casilla de verificación. El área de la derecha contiene la salida de texto.



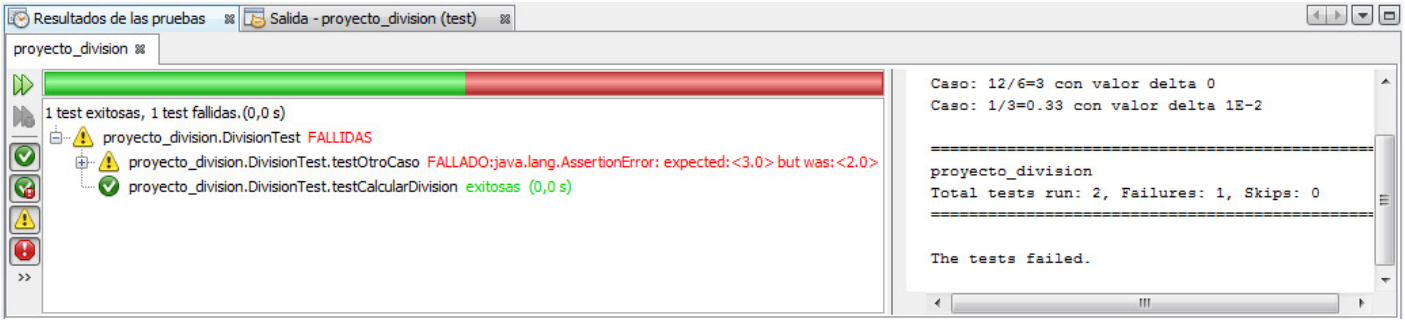
Algunos de los iconos que se pueden utilizar en la ventana de resultados son:

volver a executar a proba
volver a executar as probas non superadas
mostrar en detalle as probas superadas
mostrar en detalle as probas non superadas
mostrar en detalle os erros das probas non superadas
moverse polos métodos non superados cara arriba ou cara abaixo

Estas ventanas cambian si no pasan ninguna de las pruebas. Por ejemplo, cuando se ejecuta un método de prueba, espera un resultado de 3 para 12/6 casos de prueba, como se muestra a continuación:

```
@prueba
    public void testOtroCaso () lanza una excepción {
System.out.println("Caso: 12/6 = 3 con un valor incremental de 0 ");
        Dividendo flotante = 12.0F;
        Divisor de coma flotante = 6.0F;
División instance=nueva División ();
        Expresión de coma flotante = 3F;
        Resultado de coma flotante = instance.Calcular Division (dividendo,
divisor); assertEquals(expResult,result,0.00);
   }
```

Los detalles de los casos de prueba fallidos se mostrarán en la ventana de resultados.



Afirmación

La clase Assert le permite comprobar si la salida del método que estás probando coincide con el valor esperado. La sintaxis completa de los métodos de aserción (aserciones) que se pueden utilizar para pruebas se puede encontrar en:
https://junit.org/junit5/docs/5.3.0/api/org/junit/jupiter/api/Assertions.html

Una breve descripción de ellos es:

void assertEquals(value, actual value) es un método sobrecargado para cada tipo en java que le permite comprobar si una llamada al método devuelve el valor esperado. En el caso del valor real, tiene un tercer argumento para indicar que el valor delta o el número real es igual a la diferencia absoluta máxima entre el valor esperado y el valor actual, por lo que la afirmación tiene éxito.

- *void fail () indica cuánto tiempo se espera que el programa falle. Se utiliza cuando la prueba indica que hay un error o se espera que el método que se está probando llame a una excepción.*
- *void assertTrue (boolean) Si la expresión booleana es verdadera, la prueba es exitosa.*
- *void assertFalse (boolean) Si la expresión booleana es false, la prueba es exitosa.*
- *void assertNull (Object) Si el objeto está vacío, la prueba es exitosa.*
- *void assertNotNull (Object) Si el objeto no es nulo, la prueba es exitosa.*
- *void assertSame (Object, Object) Si los dos objetos son idénticos, la prueba es exitosa.*
- *void assertNotSame (Objeto, Objeto) Si dos objetos no son iguales, la prueba es exitosa.*

Notas

Las anotaciones proporcionan información sobre un programa que puede ser utilizada por el compilador (por ejemplo, @Override para informarle que un método está siendo reescrito, @ Deprecated para informarle que está siendo abandonado, @SupressWarnings para informarle que no debe ser advertido o advertido), por herramientas de software que pueden manejarlos (por ejemplo, código generado o archivos xml) o en tiempo de ejecución. Las anotaciones se pueden aplicar a declaraciones de clase, campos, métodos y otros elementos de un programa.

La nota más importante en una prueba es @Ignore, que se utiliza para deshabilitar una prueba y colocarla antes de @test, así como @test.

@test server La nota califica una prueba. Puede tener dos parámetros opcionales, esperado y timeout. El primero define la clase de excepción que se espera que la prueba arroje, y el segundo define el tiempo de ejecución máximo en milisegundos requerido para que la prueba sea aprobada.

Ejemplo con tiempo de espera: las siguientes pruebas no serán aprobadas debido a que la prueba se ejecuta durante más de 100 ms:

```
@ Test (timeout=100)
public void infinity () {
        while (true
);}
```

Si vamos a comprobar si ha ocurrido una excepción, assertEquals no funciona. Tenemos que usar otra estructura: assertThrows. (Nota: El proyecto Maven utiliza Ant para generar errores de compilación)

```
@prueba
public void testDivException () {
División instance=nueva División ();
assertThrows (Exception.class, new Executable () {@ Override

            public void execute () Throwable {instance
                . calcularDivisión(5,0);
            }});
}
```

Ollo: Para que la declaración anterior funcione, necesitamos hacer la siguiente importación: importar org.junit.jupiter.api.function.Executable;

La recomendación oficial es usar funciones Lambda, pero todavía no las conocemos. La sintaxis debería ser así:

```
@prueba
public void testDivException () {
División instance=nueva División ();
assertThrows(Exception.class, ()-> instance.calcularDivisión(5,0));}
```

Más información se puede ver en el video https://youtu.be/ZOGz_1XtTKc. Los elementos que aparecen en este video se encuentran en zip con el resto de elementos de este tema.

Misión 3.12. Generar y ejecutar pruebas y registrar eventos en JUnit

La tarea es generar pruebas en JUnit, ejecutarlas y registrar eventos, para métodos: Parte 1) el método calcularDivision del proyecto project_division.

Parte 2) ProyectO_factorial Método factorial del proyecto.

Parte 3) project_arrays Método de búsqueda del proyecto.

Parte 4) Método de adquisición del proyecto projecto_acronimo.

Se recomienda usar un método de prueba para cada caso de prueba en las tres primeras tareas, ya que es más fácil ver los resultados de cada tarea. Varios casos de prueba deben agruparse en la última tarea En el mismo método de prueba.

Incluye algo de código en el método con las anotaciones @BeforeAll y @ Por ejemplo un BeforeEach, que es un mensaje de consola.

3.7 JaCoCo y cobertura de pruebas

JaCoCo es una herramienta para probar la cobertura del código, es decir, si nuestras pruebas cubren todo el código y si todas las líneas de código se ejecutan, al menos una vez.

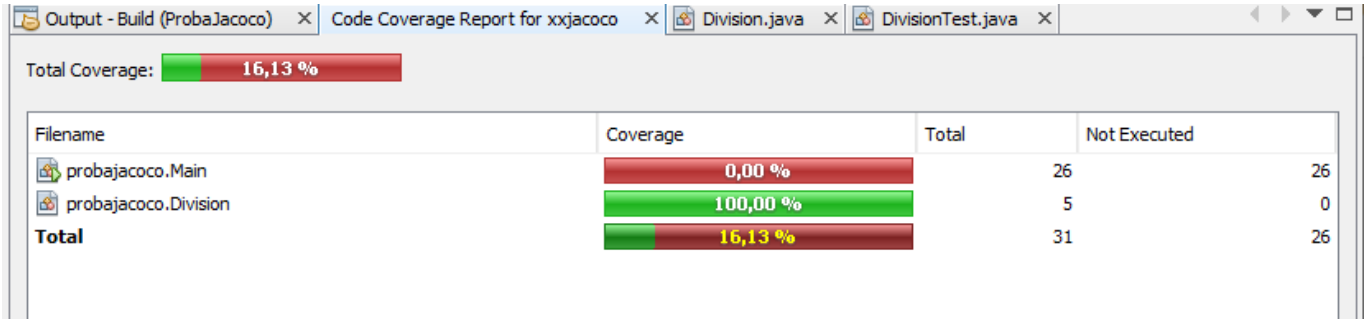
En Netbeans, para los proyectos Ant necesitas instalar un plugin, pero para los proyectos Maven ya viene con la instalación predeterminada, por lo que solo tenemos que agregarlo al archivo de configuración de nuestro proyecto: pom. xml. Añadiré la siguiente oración:

```
< Construir >
    < Plugin >
      < Plugin >
        < groupId > org.jacoco < /groupId >
        < artifactId > jacoco-maven-plugin</artifactId >
        < Versión>0.8.5</version >
        < Ejecución >
            < Ejecución >
              < id>Antes de la prueba unitaria</id >
              < Objetivo >
                 < goal>Preparar el agente</goal >
              < /objetivos >
              < Estructura >
                 < destFile>${project.build.directory}/informes de cobertura/jacoco.exec</
                 destFile><propertyName>surefireArgLine</propertyName >
              < /configuración >
            < /ejecución >
            < Ejecución >
              < id>Después de la prueba unitaria</id >
              < etapa > prueba < / etapa >
              < Objetivo >
                 < goal>Informes</goal >
              < /objetivos >
              < Estructura >
                < dataFile>${project.build.directory}/informes de cobertura/jacoco.exec</dataFile>
                <outputDirectory>${project.reporting.outputDirectory}/jacoco</outputDirectory >
              < /configuración >
                < /ejecución >
        < /ejecuciones >
       < /plugin >
       < Plugin >
             < groupId>org.apache.maven.plugins</groupId >
                < artifactId > maven-surefire-plugin</artifactId >
                < Versión2.22.2sion >
                < Estructura >
                    < argLine>${surefireArgLine}</argLine >
                < /configuración >
            < /plugin >
        < /plugins >

    < /build >
```

- Después de modificar el pom.xml, limpiamos y compilamos haciendo clic derecho en nuestro proyecto en la ventana del proyecto.
- Haga clic derecho en el proyecto de nuevo > Cobertura de código > Mostrar informe
- En la ventana que aparece, haga clic en Ejecutar todas las pruebas.

En la siguiente figura vemos la prueba del método calcularDivision de la clase División en ejecución.



En nuestro caso, hemos hecho dos pruebas, una para probar una división válida y la otra para probar una excepción que ocurre cuando se divide por cero.

```
@prueba
    public void testCalcularDivision () lanza una excepción {
        Dividendo flotante = 10.0F;
        Divisor de coma flotante = 5.0F;
División instance=nueva División ();
        Expresión de coma flotante = 2.0F;
        Resultado de coma flotante = instance.Calcular Division (
dividendo, divisor); assertEquals(expResult,result,0.001f);

    }

@prueba
    public void testDivException () {
División instance=nueva División ();
assertThrows (Exception.class, new Executable () {
@Override
            public void execute () Throwable {
Instance. Calcular Division(5,0);
            }
        });

    }
```

Prueba del camino básico

La prueba de ruta básica es una prueba de "caja blanca" que verifica el código de nuestro sistema de una manera que verifica que todo está bien, es decir, debe verificarse que todas las instrucciones del programa se ejecutan al menos una vez.

Los pasos para desarrollar una prueba de ruta básica son:

1.- Dibujo de un diagrama de flujo

2.- Complejidad del círculo computacional

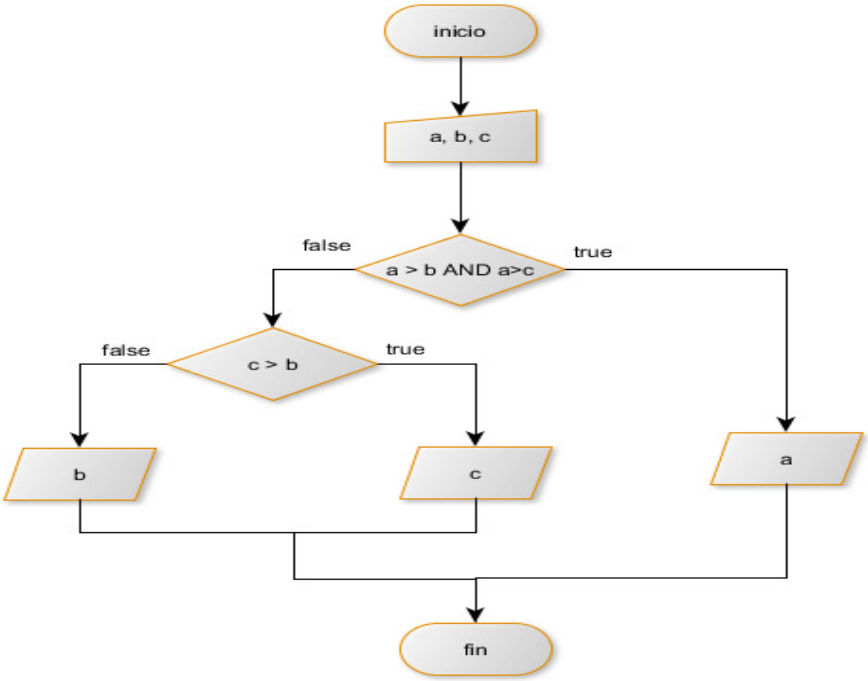
3.- Determinar el conjunto básico de rutas independientes

4.- Implementación de pruebas en JUnit

5.- Verificación de la cobertura de las pruebas con JaCoCo

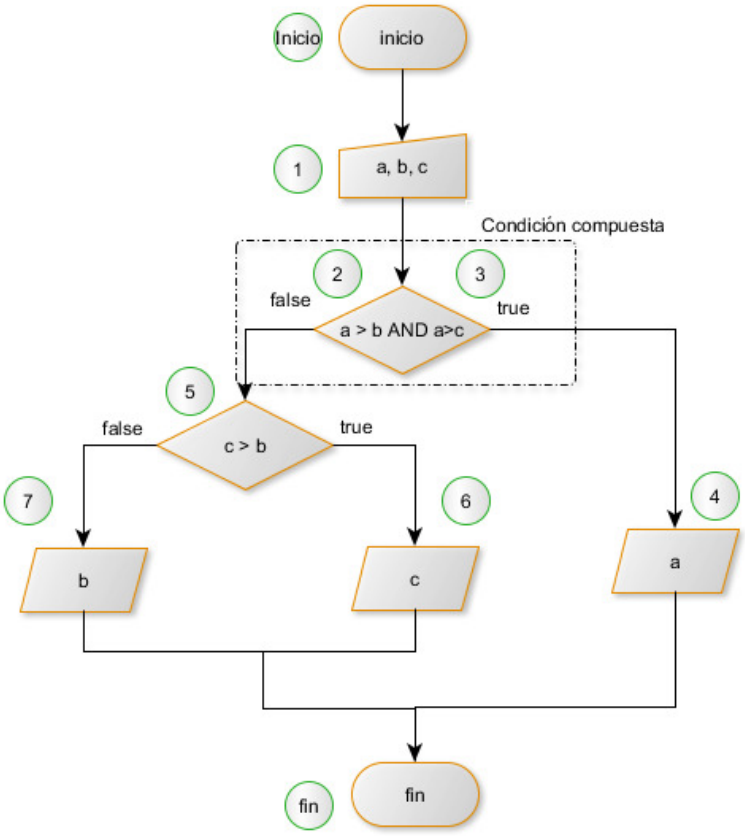
Veamos un ejemplo:

El siguiente diagrama de flujo corresponde al algoritmo que determina el mayor número de 3 valores dados.

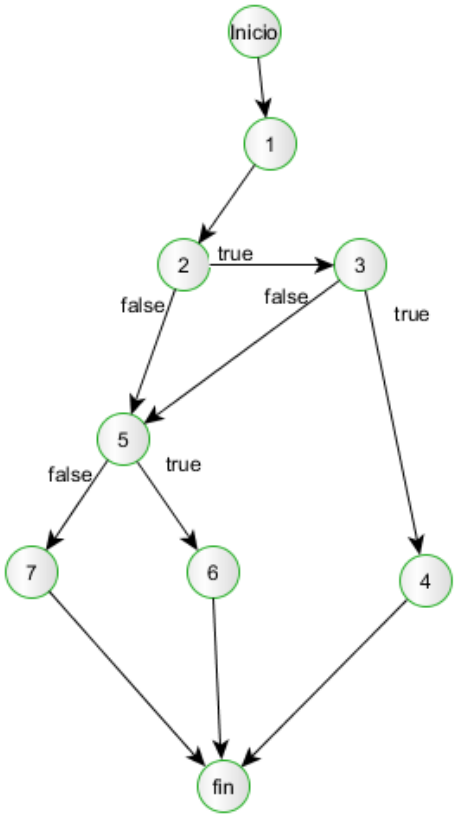


Paso 1: Dibuja el diagrama de flujo

Examinamos los nodos que componen el diagrama de flujo, así como las rutas que el programa puede tomar durante su ejecución. Si tenemos una condición compuesta, como en nuestro caso (a >b y a>c), tenemos que descomponerla creando un nodo para cada condición.



A continuación dibujamos un diagrama de flujo:



Paso 2: Complejidad del círculo

La complejidad del bucle mide el número de rutas independientes:

V(G) = borde – nodo + 2, en nuestro caso: 11 – 9 + 2 = 4

Entonces tenemos que realizar 4 pruebas para asegurarnos de que cada instrucción se ejecuta al menos una vez.

Paso 3: Camino independiente

Estamos formando caminos independientes, desde el más largo hasta el más corto, con el viento soplando nuestro diagrama de flujo.

CAMINO	ENTRADA	PRUEBA	SALIDA
1,2,3,5,6,F	a>b=TRUE, a>c=FALSE, b>c=TRUE	a=5 b=3 c=7	c
1,2,3,4,F	a>b=TRUE, a>c=TRUE	a=5 b=3 c=4	a
1,2,5,7,F	a>b=FALSE, b>c=FALSE	a=5 b=7 c=6	b
1,2,5,6,F	a>b=FALSE, b>c=TRUE	a=5 b=7 c=9	c

Paso 4: Prueba de JUnit

Implementamos estos casos de prueba en una clase JUnit en NetBeans.

Paso 5: Análisis de cobertura con JaCoCo

Para la utilidad JaCoCo de NetBeans (Maven), verificamos que hemos cubierto el 100% del código.

> Misión 3.13. Realice una prueba en una clase que solo tiene los siguientes métodos, probando con la cadena "abcabca" y la letra "a". Use JaCoCo para el análisis de cobertura, junto con pruebas precisas, para obtener un método de cobertura completa.

```
      Estático int contarLetras (cadena cadena, char letra) lanza una excepción
         {int contPos=0, conVeces=0, longCadena=0;
longCadena = cadena.length();
         if(longCadena > 0) {
            De {
               if (cadena.charAt(contPos)== letra) conVeces++;
contPos++;
             } while(contPos<longCadena);
Retorno a la convección;
         }
else throw new Exception("Parámetros incorrectos");
      }
```