

CONTORNOS DE DESENVOLVEMENTO

TEMA 3: DISEÑO E REALIZAC. DE PROBAS



C.S. Desenvolvemento de Aplicacións Multimedia
Módulo: Contornos de Desenvolvemento - 2020-2021



XUNTA DE GALICIA
CONSELLERÍA DE EDUCACIÓN
E ORDENACIÓN UNIVERSITARIA



Unión Europea
Fondo Social Europeo

Índice

1.	Introdución ás probas de software.....	3
1.1	Introdución.....	3
1.2	Estándares e certificacións	4
1.3	Probas	6
1.4	Documentación do deseño das probas.....	15
2.	Depuración de código	17
2.1	Introdución.....	17
2.2	Sesión de depuración	18
2.3	Inspección e modificación de variables, expresións e métodos	23
2.4	Pila (call stack)	26
2.5	Aplicar cambios no código	27
2.6	Punto de interrupción.....	28
3.	Probas unitarias	36
3.1	Técnicas de deseño de casos de proba	36
3.2	Probas unitarias estruturais	37
3.3	Probas unitarias funcionais.....	41
3.4	Probas unitarias aleatorias	44
3.5	Enfoque recomendado para o deseño de casos.....	44
3.6	Junit.....	45
3.7	Proba do camiño básico	53

Traballo derivado por: Patricia González Pardo

Con Licenza Creative Commons BY-NC-SA (recoñecemento - non comercial - compartir igual) a partir dos documentos orixinais:

© Xunta de Galicia. Consellería de Cultura, Educación e Ordenación Universitaria.

Autores: María del Carmen Fernández Lameiro

Licenza Creative Commons BY-NC-SA (recoñecemento - non comercial - compartir igual).

Para ver unha copia desta licenza, visitar a ligazón <http://creativecommons.org/licenses/by-nc-sa/3.0/es/>.

1. Introducción ás probas de software

1.1 Introducción

Calidade do software

A calidade do software é o grado no que o software cumpre cos requisitos funcionais establecidos co cliente, cos estándares de desenvolvemento explicitamente documentados e coas características que se esperan de calquera software desenvolvido profesionalmente. Desta definición pode deducirse que:

- Se o software non cumpre cos requisitos iniciais non será de calidade.
- Os estándares de desenvolvemento guían na forma de elaborar o software polo que se o software non cumpre con eses estándares, non será de calidade.
- Se o software ten erros importantes, ou non é fácil de utilizar ou é difícil de manter, non será de calidade.

A determinación da calidade é moi complicada debido á natureza do software xa que non é un elemento tanxible como outros produtos industriais, pero debe estar libre de erros, poden facerse medicións sobre el e pódese validar o proceso de desenvolvemento do mesmo.

En todo proceso de desenvolvemento de software realízanse controis periódicos, normalmente coincidindo cos fitos do proxecto que deben permitir:

- Verificar que se está a construír correctamente o produto.
- Validar que se está a construír o produto correcto, é dicir, o que realmente quere o usuario.



Tarefa 3.1. Explica coas túas propias palabras ou con exemplos, os 7 principios das probas do software.

1.2 Estándares e certificacións

A normalización ou estandarización é o proceso de elaborar, difundir, aplicar e mellorar as regras que se utilizan en distintas actividades científicas, industriais ou económicas co fin de ordenalas e melloralas. Segundo a ISO (*International Organization for Standardization*), a normalización é a actividade que ten por obxecto establecer, ante problemas reais ou potenciais, disposicións destinadas a usos comúns e repetidos, co fin de obter un nivel de ordenamento óptimo nun contexto dado, que pode ser tecnolóxico, político ou económico. A normalización permite:

- Simplificar, é dicir, reducir os modelos para quedar só cos máis necesarios.
- Unificar para permitir o intercambio a nivel internacional.
- Especificar para evitar erros de identificación creando unha linguaxe clara e precisa.

A certificación é a acción levada a cabo por unha entidade recoñecida, independente das partes interesadas, mediante a que se manifesta que unha organización, produto, proceso ou servizo cumpre os requisitos mínimos establecidos nunhas normas ou especificacións técnicas. Leva á empresa a diferenciarse do resto e tomar vantaxe no mercado, ao demostrar que segue uns estándares de calidade. A petición dunha certificación ten carácter voluntario, carrega uns custos e ten que renovarse periodicamente.

ISO

A Organización Internacional de Normalización ou ISO (<http://www.iso.org>) naceu o 23 de febreiro de 1947 e é un organismo non governamental encargado de promover o desenvolvemento de normas internacionais de fabricación, comercio e comunicación para todas as ramas industriais. A súa función principal é a de buscar a estandarización de normas de produtos e seguridade para as empresas ou organizacións a nivel internacional. As normas que produce denomínanse normas ISO. A central está en Xenebra (Suíza) pero está integrada polos organismos de normalización nacionais de moitos países. O contido dos estándares está protexido por dereitos de copyright e para acceder a eles o público corrente debe comprar cada documento na súa páxina oficial.

A familia de normas ISO 9000 aborda distintos aspectos da xestión da calidade e contén algunha das normas máis coñecidas da ISO. As normas proporcionan orientación e ferramentas para as empresas e organizacións que queren asegurarse de que os seus produtos e servizos cumpran cos requirimentos do cliente e que a calidade se mellora constantemente. Esta familia inclúe as normas:

- ISO 9001: 2015 - Establece os requisitos dun sistema de xestión de calidade incluíndo unha forte orientación ao cliente, a motivación e implicación da alta dirección, o enfoque por procesos e a mellora continua.
- ISO 9000: 2015 - Cubre os conceptos e principios da xestión da calidade.
- ISO 9004: 2018 - Céntrase en como facer que un sistema de xestión de calidade sexa sostido, é dicir, proporciona recomendacións para melloralo.
- ISO 19011: 2018 - presenta unha guía sobre as auditorías internas e externas dos sistemas de xestión de calidade.

IEC

A Comisión Electrotécnica Internacional ou CEI (<http://www.iec.ch/>) ou IEC (*International Electrotechnical Commission*) é unha organización de normalización nos campos eléctrico, electrónico e tecnoloxías relacionadas. Foi fundada en 1904 durante o Congreso Eléctrico Internacional de San Luís (EEUU). Actualmente a súa sede está en Xenebra (Suíza) e está integrada polos organismos nacionais de normalización, nas áreas indicadas, dos países membros. En 1938, o organismo publicou o primeiro dicionario internacional (*International Electrotechnical Vocabulary*) co propósito de unificar a terminoloxía eléctrica, esforzo que se mantivo durante o transcurso do tempo, sendo o Vocabulario Electrotécnico Internacional un importante referente para as empresas do sector.

Numerosas normas desenvólvense conxuntamente coa ISO e chámanse normas ISO/IEC. Por exemplo, a ISO/IEC 90003:2014 proporciona orientación para as organizacións e empresas na aplicación da norma ISO 9001:2008 para a adquisición, subministro, desenvolvemento, operación e mantemento de software e servizos de apoio relacionados.

AENOR

A Asociación Española de Normalización e Certificación ou AENOR é a única entidade recoñecida en España para desenvolver tarefas de normalización e certificación e para representar a España nos organismos europeos (CEN, CENELEC e ETSI) e internacionais (ISO e IEC). É unha entidade privada sen fins lucrativos que se creou en 1986. No seu sitio oficial (<http://www.aenor.es>) ofrece información sobre o proceso de certificación, publicación de novas normas, xornadas....

As normas que adapta ou elabora, coñecidas como normas UNE, indican como debe ser un produto ou como debe funcionar un servizo para que sexa seguro e responda ao que o consumidor espera

del. Por exemplo, a ISO 9000 foi adoptada sen modificacións como norma europea (serie EN 29000) e como norma española (serie UNE 66-90). AENOR pon a disposición de todos un dos catálogos máis completos, con máis de 28.900 documentos normativos que conteñen solucións eficaces.

Os certificados AENOR son moi valorados, non só en España senón tamén no ámbito internacional, emitindo certificados en máis de 60 países. AENOR sitúase entre as 10 certificadoras máis importantes do mundo.

IEEE

IEEE (lido i-e-cubo en España e i-triplo-e en latinoamérica) corresponde ás siglas de *Institute of Electrical and Electronics Engineers*, en español Instituto de Enxeñeiros Eléctricos e Electrónicos, e é unha asociación técnico profesional mundial dedicada á estandarización, entre outras cousas. Naceu en 1884 pero adoptou o nome de IEEE en 1963 e na actualidade é a maior asociación internacional sen ánimo de lucro formada por profesionais das novas tecnoloxías, como enxeñeiros eléctricos, enxeñeiros en electrónica, científicos da computación, enxeñeiros en informática, enxeñeiros en biomédica, enxeñeiros en telecomunicación e enxeñeiros en Mecatrónica. Segundo o mesmo IEEE, o seu traballo é promover a creatividade, o desenvolvemento e a integración, compartir e aplicar os avances nas tecnoloxías da información, electrónica e ciencias en xeral para beneficio da humanidade e dos mesmos profesionais. Colaboran con ISO e IEC en temas comúns. O sitio oficial é: <http://www.ieee.org>. A Sección Española do IEEE é recoñecida dentro da Rexión 8 en abril de 1968 e o seu sitio oficial é: <http://www.ieeespain.org/>.

Algunhas normas relacionadas co software son:

- IEEE 730. Plans de aseguramento da calidade de software.
- IEEE 829. Documentación de probas do software.
- IEEE 982.1, 982.2. Dicionario estándar de medidas para producir software fiable.
- IEEE 1008. Probas unitarias de software.
- IEEE 1012. Verificación e validación de software.
- IEEE 1028. Revisións de software.
- IEEE 1044. Clasificación estándar para anomalías do software.
- IEEE 1061. Estándar para unha metodoloxía de métricas de calidade do software.
- IEEE 1228. Plans de seguridade do software.

1.3 Probas

A proba exhaustiva do software é impracticable xa que non se poden probar todas as posibilidades do seu funcionamento, mesmo en programas pequenos e sinxelos. O obxectivo das probas é a detección de defectos, bugs¹ ou erros do software, polo que unha proba é un éxito se descobre un defecto. Trátase dunha actividade a posteriori, de detección de problemas no software e non de prevención.

O proceso de proba comeza coa xeración dun plan de probas en base á documentación sobre o proxecto e á documentación sobre o software a probar. A partir do devandito plan, detállanse as probas específicas, execútanse cos casos de proba e obtéñense os resultados. Os resultados poden

¹ A tradución de bug é bicho e en informática utilízase para indicar un burato ou defecto no software ou hardware. A explicación máis difundida de esta utilización é que o primeiro fallo informático debeuse a unha polilla atrapada nun repetidor do computador Mark II.

indicar a existencia de erros e nese caso haberá que documentar e notificar os erros co obxectivo de que sexan arranxados e se volvan a executar as probas.

Non se deben de facer plans de proba pensando que non hai defectos; hai que asumir que sempre os hai. As probas deben de planificarse e deseñarse de forma sistemática para poder detectar o máximo número e variedade de defectos co mínimo consumo de tempo e esforzo. Son unha tarefa tanto ou máis creativa que o desenvolvemento de software e débense evitar as probas non documentadas nin deseñadas con coidado como por exemplo as que se realizan sobre a marcha.

As probas deben centrarse en dous obxectivos: probar se o software non fai o que debe, e probar se o software fai o que non debe, é dicir, se provoca efectos secundarios adversos; é habitual esquecer este último obxectivo.

Tipos de probas

Existen distintas maneiras de clasificar as probas non excluíntes entre si:

- Funcionais ou non funcionais. As primeiras están destinadas a comprobar algún requisito funcional do software e as segundas non. Exemplos de probas non funcionais: as que permiten indicar o esforzo requirido para aprender a manexar ese software, para analizar o código e localizar un fallo, para soportar cambios e facilitar as probas asociadas a eses cambios, ou para migrar o software a outro entorno.
- Manuais e probas automáticas. As primeiras realizáranse seguindo un plan detallado e estruturado pero sen dispoñer dun proceso automático para executalas. As segundas dispoñen dun proceso automático que pode repetirse cantas veces se queira de forma cómoda.
- Dinámicas e probas estáticas. As primeiras realízanse mentres o software se está executando e as segundas non.
- Probas de Caixa negra e de Caixa Branca: As probas de **caixa negra** (Black Box Testing) son aquelas nas que a aplicación é probada empregando a súa interface externa, sen preocuparnos da implementación da mesma. Aquí o fundamental é comprobar que os resultados da execución da aplicación, son los esperados, en función das entradas que recibe. Nas probas de caixa branca (White Box Testing) se proba a aplicación desde dentro, baseándose na súa lóxica e os camiños que segue o fluxo do programa polo código.

Unha proba de tipo **Caixa Negra** lévase a cabo sen ter que coñecer nin a estrutura, nin o funcionamento interno do sistema. Cando se realiza este tipo de probas, so se coñecen as entradas adecuadas que deberá recibir a aplicación, así como las saídas que lles correspondan, pero non se coñece o proceso mediante o que a aplicación obtén eses resultados.

Pola contra, a proba de **Caixa Branca**, vai analizar e probar directamente o código da aplicación.

Existen operacións que realizan certas ferramentas sobre o código para detectar defectos e que non son estritamente probas, como por exemplo:

- Validación de código: Permite garantir que o código cumpre algún estándar da linguaxe, por exemplo, que as variables estean inicializadas.
- Depuración (Debugging): Permite detectar o código que dá resultados erróneos na execución. Os contornos de desenvolvemento dispoñen de ferramentas automáticas de depuración que permiten executar o código de maneira controlada polo programador e

permitindo que o programador realice a execución liña a liña ou defina puntos de interrupción para que a execución se pare nese punto; cando se para a execución, o programador pode inspeccionar variables, expresións ou a pila e realizar modificacións sobre elas para ver como se comporta a execución.

- Análise de liñas de código: Permite detectar por exemplo: código repetido en varios sitios, revisar o código que está documentado e que facilitará a tarefa de xeración automática de documentación (Javadoc en Java), encontrar liñas de código comentado que ocupan espazo aínda que no se executen e que poden eliminarse utilizando un sistema de control de versións.

Probas unitarias

Son probas funcionais realizadas por persoal técnico que permiten detectar problemas nun módulo individual e elemental como por exemplo, un método dunha clase ou unha clase. Centran a súa actividade en exercitar a lóxica do módulo seguindo a estrutura do código (técnica de caixa branca) e as funcións que debe realizar o módulo atendendo ás entradas e saídas (técnica de caixa negra).

A porcentaxe de código probado mediante probas unitarias denomínase **cobertura do software**.

Os contornos de desenvolvemento dispoñen de ferramentas para deseñar e executar probas unitarias. Dentro das ferramentas dispoñibles no mercado destaca a familia XUnit: JUnit para Java, CppUnit para C++, PHPUnit para PHP e NUnit para .Net e Mono.

As probas unitarias son o tema central do punto 3 desde tema.



Tarefa 3.2. Buscar ferramentas para a depuración de código e para realizar probas unitarias no contorno de desenvolvemento para distintas linguaxes.

Probas de integración

Son probas funcionais realizadas por persoal técnico que permiten detectar problemas entre módulos relacionados e probados anteriormente de forma individual mediante probas unitarias. Deben ter en conta os mecanismos de ensamblaxe de módulos fixados na estrutura do programa, é dicir, as interfaces entre compoñentes da arquitectura do software. As probas de unidade e de integración solápanse e mestúranse no tempo normalmente.

Existen distintos tipos de probas de integración en función da orde de integración. A orde de integración elixida afecta a diversos factores, como: a forma de preparar casos, as ferramentas necesarias, a orde de codificar e probar os módulos, o custo da depuración, e o custo de preparación de casos. Os tipos fundamentais de integración son os seguintes:

- Integración incremental: Combínase o seguinte módulo que se debe probar co conxunto de módulos que xa foron probados. Existen dous tipos fundamentais:
 - Ascendente: Comézase polos módulos folla.
 - Descendente: Comézase polo módulo raíz.
- Integración non incremental: Próbese cada módulo por separado e logo intégranse todos dunha vez e próbese o programa completo. Denomínase tamén *Big-Bang* porque o número de módulos crece instantaneamente.

Integración incremental ascendente

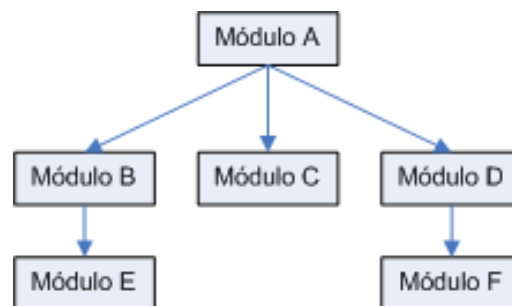
Na integración incremental ascendente empézase combinando os módulos de máis baixo nivel. As características da integración ascendente son:

- Pódese traballar con módulos de forma individual ou combinar os módulos de baixo nivel en grupos que realizan algunha función específica co obxectivo de reducir o número de pasos de integración.
- Escríbese para cada grupo un módulo impulsor ou condutor que é un módulo escrito para permitir simular a chamada aos módulos, introducir os datos de proba a través dos parámetros de entrada e recoller os resultados a través dos parámetros de saída.
- Próbese cada grupo empregando o seu impulsor.
- Elimínanse os módulos impulsores de cada grupo e substitúense polos módulos do nivel superior da xerarquía.

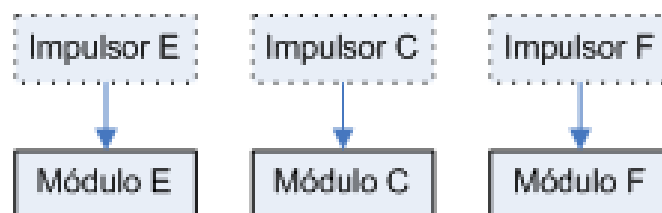
A construción de módulos impulsores non adoita ser moi complexa. Esta facilidade levou á creación de ferramentas automáticas capaces de realizar os labores dun impulsor. Normalmente están formados por:

- Instrucións para obter os datos necesarios para pasarlle ao módulo a probar.
- A chamada ao módulo a probar.
- Instrucións necesarias para mostrar os resultados devoltos polo módulo a probar.

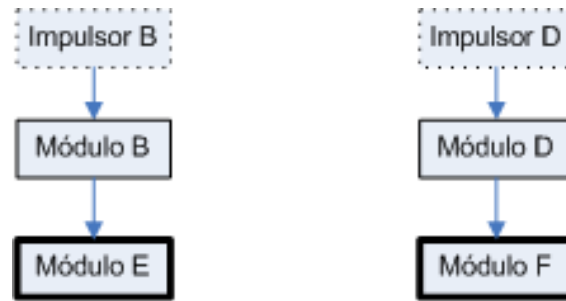
Por exemplo, as etapas de integración serían 6 no caso de contar cos módulos seguintes.



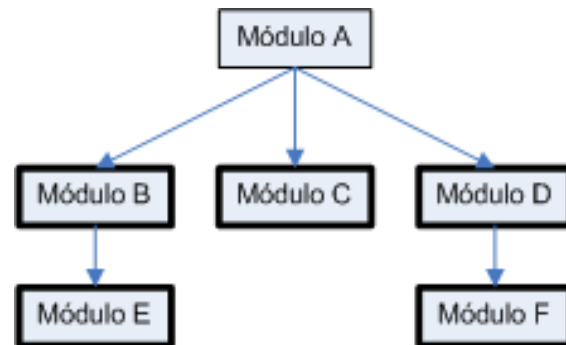
Etapas 1,2 e 3: Realízanse as probas unitarias de E, C e F. Os impulsores represéntanse cun borde punteado na seguinte gráfica.



Etapas 4 e 5: Realízanse por unha parte e de forma simultánea, a proba de unidade de B e a de integración (ou de interface) B-E, e por outra e de forma simultánea, a proba da unidade D e a interface D-F. Os módulos xa probados en etapas anteriores represéntanse cun borde grosso na seguinte gráfica.



Etapa 6: Incorporase o módulo A e próbase o programa completo, que non require impulsores, xa que todo o código de xestión da entrada e saída do programa está presente.



Integración incremental descendente

A integración incremental descendente comeza co módulo principal (o de maior nivel ou módulo raíz) e vai incorporando módulos subordinados progresivamente. Non existe unha regra xeral para determinar os módulos subordinados que convén incorporar primeiro. Como recomendación débense incorporar canto antes as partes críticas e os módulos de Entrada/Saída. Existen dúas ordes fundamentais de integración descendente:

- Primeiro en profundidade: vanse completando ramas da árbore modular. No exemplo anterior, a secuencia de módulos sería A-B-E-C-D-F.
- Primeiro en anchura: vanse completando niveis horizontais de xerarquía modular. No exemplo anterior, a secuencia de módulos sería A-B-C-D-E-F.

As características da integración descendente son:

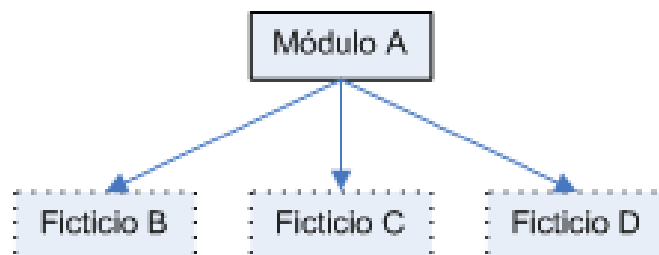
- O módulo raíz é o primeiro en probarse e escríbense módulos ficticios (*stubs*), para simular a presenza dos subordinados ausentes, que serán chamados polo módulo raíz.
- Unha vez probado o módulo raíz substitúese un dos subordinados ficticios polo módulo correspondente segundo a orde elixida, e incorpóranse ficticios para recoller as chamadas do último incorporado.
- Repítese o proceso detallado para o módulo raíz, é dicir, execútanse as correspondentes probas cada vez que se incorpora un módulo novo e ao terminar cada proba, substitúese un ficticio polo seu correspondente real. Convén repetir algúns casos de proba de execucións anteriores para asegurarse de que non se introduciu ningún defecto novo.

A codificación de módulos ficticios subordinados é máis complicada que a creación de impulsores. Os ficticios deben simular que recollen o control da chamada e que fan algo cos parámetros que se lles pasan. Existen diversos niveis de sofisticación dos ficticios:

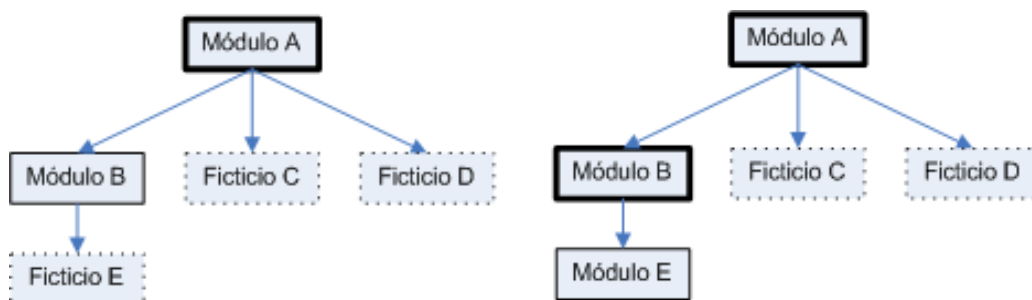
- Módulos que só mostran unha mensaxe de traza. Por exemplo, `printf("Executouse o módulo 1")`;
- Módulos que mostran os parámetros que se lles pasan. Por exemplo, recibe o parámetro x e o mostra con `printf("%d", x)`;
- Módulos que devolven un valor que non depende dos parámetros que se pasen como entrada (sempre devolve o mesmo valor, ou un valor aleatorio, etc.). Por exemplo, `return(5)`; independentemente dos parámetros que reciba.
- Módulos que, en función dos parámetros pasados, devolven un valor de saída que máis ou menos corresponda a dita entrada. Por exemplo, devolver un valor utilizando as entradas para buscar nun array bidimensional, `return(Táboa[x][y])`; cando recibe x e y.

Por exemplo, utilizando o mesmo deseño de módulos que para a integración ascendente, e utilizando a orde primeiro en profundidade, serían necesarias 6 etapas.

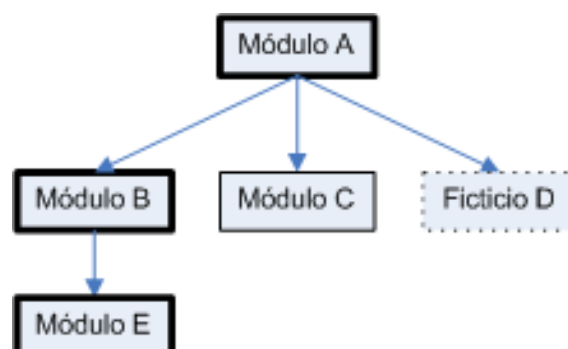
Etape 1: Realízase a proba unitaria de A. Os ficticios represéntanse cun borde punteado na seguinte gráfica.



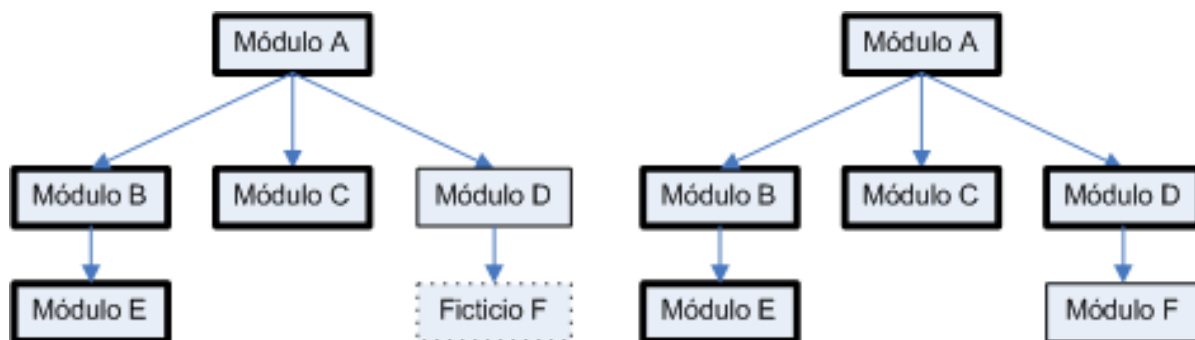
Etapas 2 e 3: Realízanse simultaneamente a proba de unidade de B e a de integración (ou de interface) A-B, e a continuación realízase simultaneamente a proba da unidade E e a interface B-E. Os módulos probados en etapas anteriores represéntanse cun borde grosso na seguinte gráfica.



Etape 4: Realízase simultaneamente a proba do módulo C e a interface A-C.



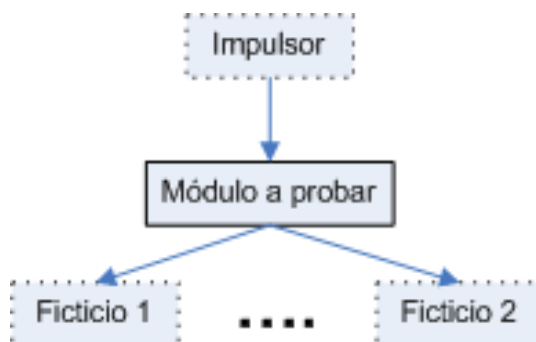
Etapas 5 e 6: Realízase simultaneamente a proba do módulo D e a interface A-D e a continuación a proba do módulo F e a interface D-F, quedando toda a integración probada.



Integración non incremental

A integración non incremental é o único caso no que as probas de unidade e integración están totalmente separadas. As características da integración non incremental son:

- Cada módulo require para ser probado:
 - Dun módulo impulsor que transmite os datos de proba ao módulo e mostra os resultados dos devanditos casos de proba.
 - Os módulos ficticios necesarios para simular a función de cada módulo subordinado ao módulo que imos probar.



- Despois de probar cada módulo por separado, se ensamblan todos eles dunha soa vez para formar o programa completo.

Comparación entre os distintos tipos de integración

A comparación entre as vantaxes da integración non incremental e a incremental queda reflectida na seguinte táboa:

Vantaxes da integración non incremental	Vantaxes da integración incremental
<ul style="list-style-type: none"> ▪ Require menos tempo de máquina para as probas, xa que se proba dunha soa vez a combinación dos módulos. ▪ Existen máis oportunidades de probar módulos en paralelo. ▪ Require menos traballo, xa que se escriben menos módulos impulsores e ficticios. 	<ul style="list-style-type: none"> ▪ Os defectos e erros nas interfaces detéctanse antes, xa que se empeza antes a probar as unións entre os módulos. ▪ A depuración é moito máis fácil, xa que de detectar os síntomas dun defecto nun paso da integración, hai que atribuílo moi probablemente ao último módulo incorporado. ▪ Examínase con maior detalle o programa, ao ir comprobando cada interface aos poucos.

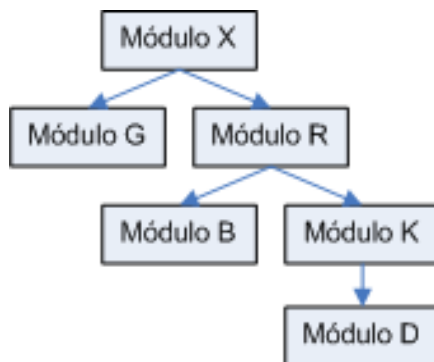
A comparación entre as características da integración incremental ascendente e descendente queda reflectida na seguinte táboa:

Integración incremental ascendente	Integración incremental descendente
<ul style="list-style-type: none"> ▪ É vantaxosa cando hai defectos en niveis inferiores do programa. ▪ As entradas son máis fáciles de crear. ▪ O programa, como entidade, non existe até incorporar o último módulo. ▪ Requírense módulos impulsores. Os módulos impulsores son fáciles de crear. ▪ É máis fácil observar os resultados das probas. 	<ul style="list-style-type: none"> ▪ É vantaxosa cando hai fallos nos niveis superiores do programa. ▪ Antes de incorporar E/S, é difícil representar casos e as entradas poden ser difíciles de crear. Tras incorporar funcións de E/S, é fácil a representación de casos. ▪ Antes de incorporar o último módulo, vese a estrutura previa do programa. ▪ Require ficticios subordinados. Os ficticios subordinados non son fáciles de crear. ▪ Difícil observar resultados. ▪ Induce a atrasar a terminación da proba dalgúns módulos.

A selección dunha estratexia de integración depende das características do software e, ás veces, da planificación do proxecto. Algunhas recomendacións poderían ser:

- Identificar e probar canto antes aqueles módulos considerados críticos ou problemáticos.
- En xeral, o mellor compromiso pode ser un enfoque combinado (ás veces denominado *sándwich*) que consiste en:
 - Usar a integración descendente para os niveis superiores da estrutura do programa.
 - Utilízase simultaneamente a ascendente para os niveis subordinados.
 - Continúase ata que ambas as aproximacións atópanse nalgún nivel intermedio.

 **Tarefa 3.3. Detallar etapas da integración incremental ascendente e da descendente para o seguinte grupo de módulos.**



Probas de sistema ou implantación

Son probas non funcionais realizadas por persoal técnico que permiten comprobar que o sistema completo, compre os requisitos funcionais e técnicos especificados e se relaciona correctamente con outros sistemas. Ademais realízanse outras probas como:

- **Probas de carga:** Este é o tipo máis sinxelo de probas de rendemento. Unha proba de carga realízase xeralmente para observar o comportamento dunha aplicación baixo unha cantidade de peticións esperada. Esta carga pode ser o número esperado de usuarios concorrentes utilizando a aplicación e que realizan un número específico de transaccións durante o tempo que dura a carga. Esta proba pode mostrar os tempos de resposta de todas as transaccións importantes da

aplicación. Se a base de datos, o servidor de aplicacións, etc. tamén se monitorizan, entón esta proba pode mostrar o colo de botella en la aplicación.

- **Probas de sobrecarga ou estrés:** Permiten avaliar o comportamento do sistema ao someter a unha situación límite como por exemplo demanda excesiva de peticións, utilización da máxima cantidade de memoria, traballar con pouca memoria, ter moitos usuarios realizando ao mesmo tempo a mesma operación, utilizar o máximo volume de datos.
- **Proba de estabilidade:** Esta proba normalmente faise para determinar se la aplicación pode aguantar una carga esperada continuada. Xeralmente esta proba realízase para determinar se hai algunha fuga de memoria na aplicación.
- **Proba de picos:** como o nome suxire, trata de observar o comportamento do sistema variando o número de usuarios, tanto cando baixan, como cando teñen cambios drásticos na súa carga. Esta proba recoméndase que sexa realizada cun software automatizado que permita realizar cambios no número de usuarios mentres que os administradores levan un rexistro dos valores a ser monitorizados.
- **Probas de compatibilidade:** Permiten probar o sistema en diferentes contornos, medios ou sistemas operativos e ver se hai fallos no aspecto ou no funcionamento.
- **Probas de seguridade e acceso a datos:** Permiten probar como responde o sistema ante ataques externos como por exemplo irrupcións non autorizadas ou camuflaxe de código malicioso nunha entrada de datos.
- **Probas de recuperación e tolerancia a fallos:** Permiten provocar anomalías externas como fallos eléctricos, de dispositivos, de software externo ou de comunicacións, e ver que o sistema se recupera sen perda de datos e sen fallos de integridade e o tempo que lle leva facelo.

Probas de validación ou aceptación

Son probas non funcionais realizadas por persoal non técnico e serven para comprobar se o produto final se axusta aos requisitos iniciais do software e está baseada nas accións visibles para o usuario e nas saídas do software que este pode recoñecer.

Poden existir proba alfa e/ou probas beta:

- As probas alfa asóciase normalmente ao software contratado. Realízaas o cliente nun contorno controlado baixo a supervisión da empresa que desenvolve o software. O desenvolvedor de software tomará nota dos posibles erros e problemas de uso.
- As probas beta asóciase normalmente ao software de interese xeral como por exemplo os paquetes ofimáticos pero tamén poden utilizarse para software contratado. Realízanas usuarios de confianza no seu entorno real de traballo sen control directo da empresa de software. O usuario informará dos resultados das probas. Ás veces, publícanse estas versións ao público en xeral, pero baixo esta etiqueta, para que se saiba que pode ter erros.

Probas de regresión

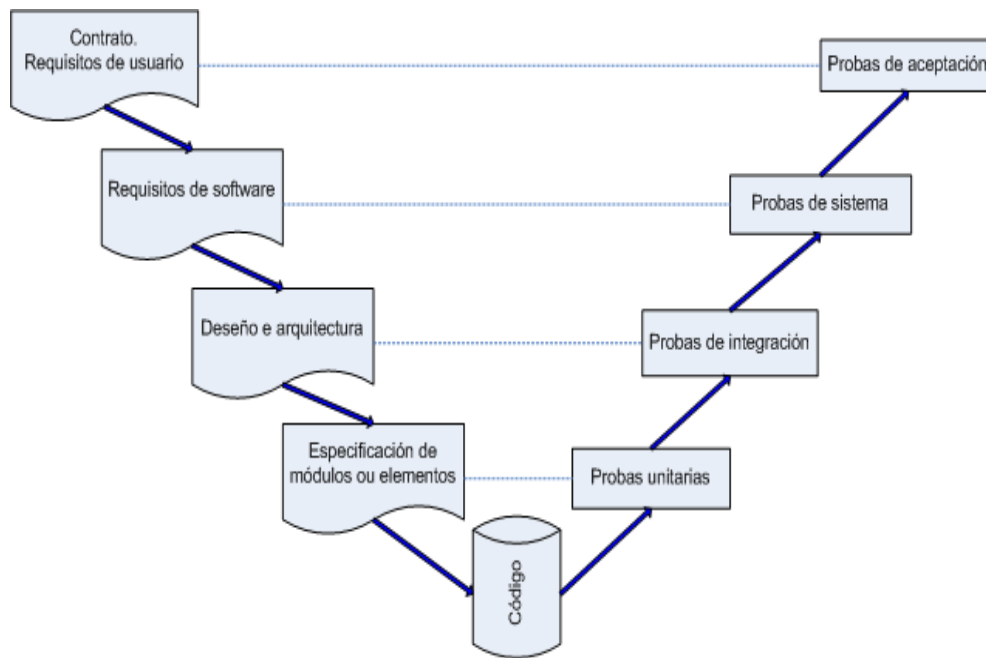
Son probas realizadas por persoal técnico para evitar efectos colaterais. Aplícanse cada vez que se fai un cambio no software para verificar que non aparecen comportamentos non desexados ou erros noutros módulos ou partes do software.

Un caso particular de cambio no software dáse cando se agrega un novo módulo nas probas de integración incremental. Pode ocorrer que un módulo que funcionaba ben deixe de facelo pola incorporación doutro. Neste caso as probas de regresión consistirán en volver a aplicar un

subconxunto significativo das probas realizadas aos módulos xa integrados.

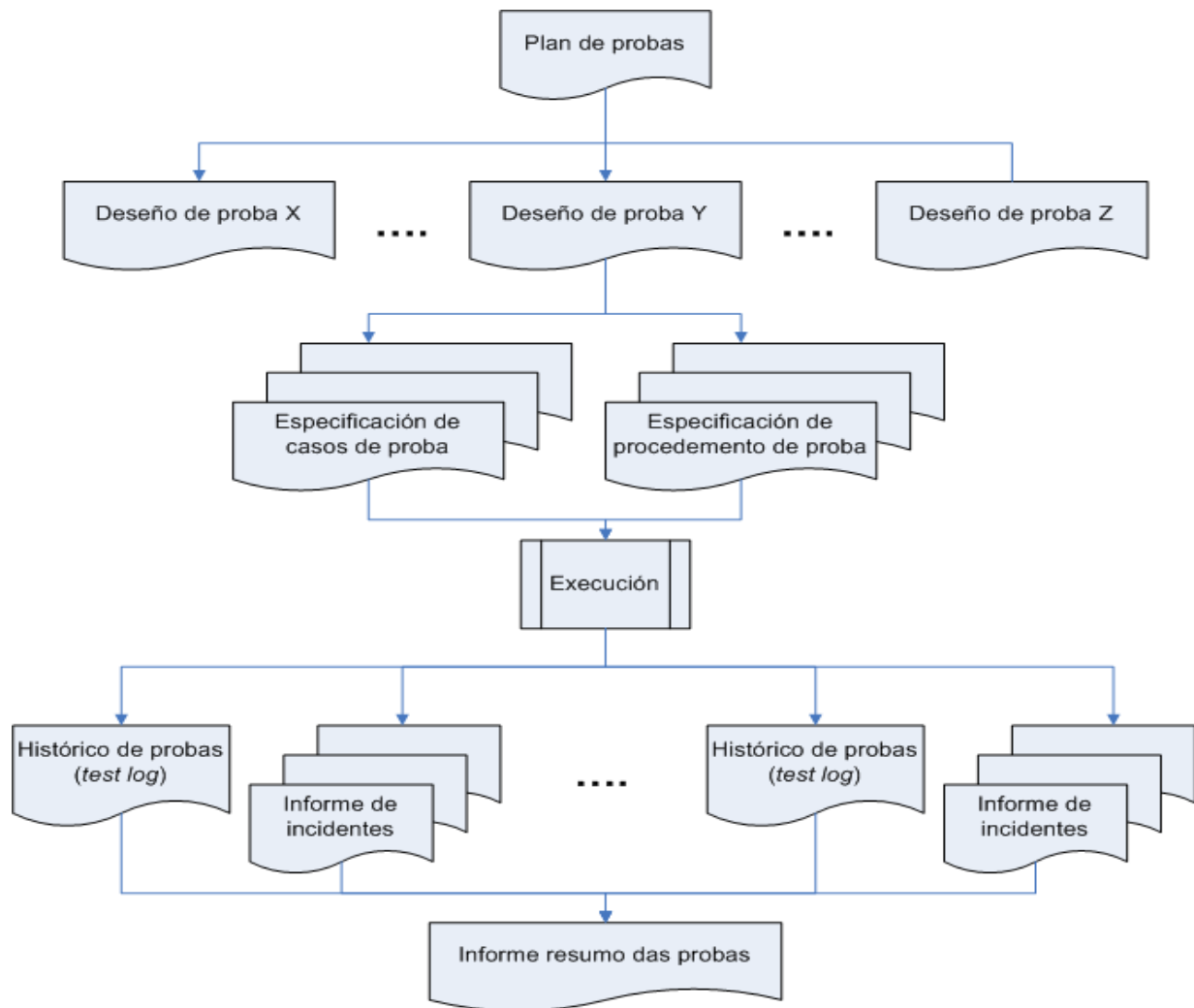
Estratexia de aplicación das probas no ciclo de vida clásico

A estratexia de aplicación e a planificación das probas pretende integrar o deseño dos casos de proba nunha serie de pasos ben coordinados, a través da creación de distintos niveis de proba, con diferentes obxectivos. En xeral a estratexia de probas adoita seguir durante o ciclo de vida do software as etapas que se mostran na seguinte imaxe.



1.4 Documentación do deseño das probas

A documentación das probas é necesaria para unha boa organización das mesmas, así como para asegurar a súa reutilización. Segundo o estándar IEEE 829, os documentos relacionados coas probas asóciase ás distintas fases das probas segundo se indica na gráfica seguinte.



O primeiro paso sitúase na planificación xeral do esforzo de proba (plan de probas) que inclúe o alcance do plan, os recursos a utilizar, a planificación das probas e as actividades a realizar. O segundo paso consiste no deseño das probas que xorde da ampliación e detalle do plan de probas e no que se especifican as características das diferentes probas. A partir deste deseño, especifícanse cada un dos casos de proba mencionados brevemente no deseño da proba e especifícase como proceder en detalle e desenvolver a execución dos devanditos casos (procedementos de proba). Tanto as especificacións de casos de proba como as especificacións dos procedementos deben ser os documentos básicos para a execución das probas. O último paso é o informe co resumo das probas no que se reflecten os resultados das actividades de proba e se achega unha avaliación do software baseada nos devanditos resultados.

A documentación da execución das probas é fundamental para a eficacia na detección e corrección de defectos, así como para deixar constancia dos resultados da execución das probas. Os documentos xerados para cada execución son:

- **Histórico de probas:** Documenta os feitos relevantes ocorridos durante a execución das probas.
- **Informe de incidentes:** Documenta cada incidente ocorrido na proba e que requira unha posterior investigación.

2. Depuración de código

2.1 Introducción

A operación de depuración serve para examinar o código da aplicación en tempo de execución e buscar solucións a erros detectados. Permite executar liñas de código ata un momento no que se pode examinar o estado para descubrir problemas.

Os exemplos de depuración desta actividade utilizarán un proxecto Java denominado *Estadísticos* cun paquete chamado *probas* que inclúe as clases *Estadísticos.java* e *Main.java*. A execución do proxecto permite teclear dous números enteiros positivos (o primeiro maior ou igual co segundo) e visualizar os estatísticos:

- Factorial de cada un dos números: produto de números enteiros dende 2 ata o número.
- Combinacións dos dous números: resultado de dividir o factorial do primeiro número polo produto do factorial do segundo número e o factorial da diferenza de ambos números.
- Variacións sen repetición dos dous números: resultado de multiplicar as combinacións polo factorial do segundo número.
- Variacións con repetición de dous números: resultado de elevar o primeiro número ao segundo.

O código de *Main.java* é:

```
package probas;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        System.out.print("\nCALCULOS ESTADÍSTICOS\n");
        Scanner teclado = new Scanner(System.in);
        boolean error;
        int m, n;
        do {
            try {
                error = false;
                System.out.print("Teclee m (>= 0): ");
                m = teclado.nextInt();
                System.out.print("Teclee n (>= 0 y <= m): ");
                n = teclado.nextInt();
                Estadísticos es = new Estadísticos(m, n);
                System.out.printf("Permutaciones(%d) = %f\n", n, es.factorial(n));
                System.out.printf("Permutaciones(%d) = %f\n", m, es.factorial(m));
                System.out.printf("Variaciones (%d,%d) = %f\n", m, n, es.variaciones());
                System.out.printf("Combinaciones(%d,%d) = %f\n", m, n, es.combinaciones());
                System.out.printf("Variaciones con repet. (%d,%d)= %f\n", m, n, es.variac_repet());
            } catch (Exception e) {
                System.out.println(e.getClass() + "->" + e.getMessage()); // Muestra el error
                error = true;
            }
        } while (error);
    }
}
```

O código de *Estadísticos.java* é:

```
package probas;

public class Estadísticos {

    private int m;
    private int n;
```



```

public Estadisticos(int m, int n) throws Exception {
    if (m < 0 || n < 0 || m < n) {
        throw new Exception("Error. " +
            "Los argumentos tienen que ser >=0 y el primero >= que el segundo");
    }
    this.n = n;
    this.m = m;
}
/* Cálculo Factorial o permutaciones de x */
public double factorial(int x) throws Exception {
    double resultado = 1;
    for (int i = 2; i <= x; i++) {
        resultado *= i;
    }
    return resultado;
}
/* Cálculo Combinaciones de m elementos tomados de n en n */
public double combinaciones() throws Exception {
    double combi = factorial(m)/(factorial(n)*factorial(m-n));
    return combi;
}
/* Cálculo Variaciones de m elementos tomados de n en n */
public double variaciones() throws Exception {
    double vari = combinaciones()*factorial(n);
    return vari;
}
/* Cálculo Variaciones con repetición de m elementos tomados de n en n */
public double variac_repet() throws Exception {
    double varirepe = Math.pow(m, n);
    return varirepe;
}
}

```

2.2 Sesión de depuración

Para iniciar unha sesión de depuración é indispensable que o arquivo ou proxecto a depurar teña método *main*, é dicir, ten que ser posible executalo.

NetBeans permite iniciar sesión de depuración dun proxecto ou dun arquivo e elixir como empezar a depuración de diferentes maneiras. Por exemplo permite depurar:

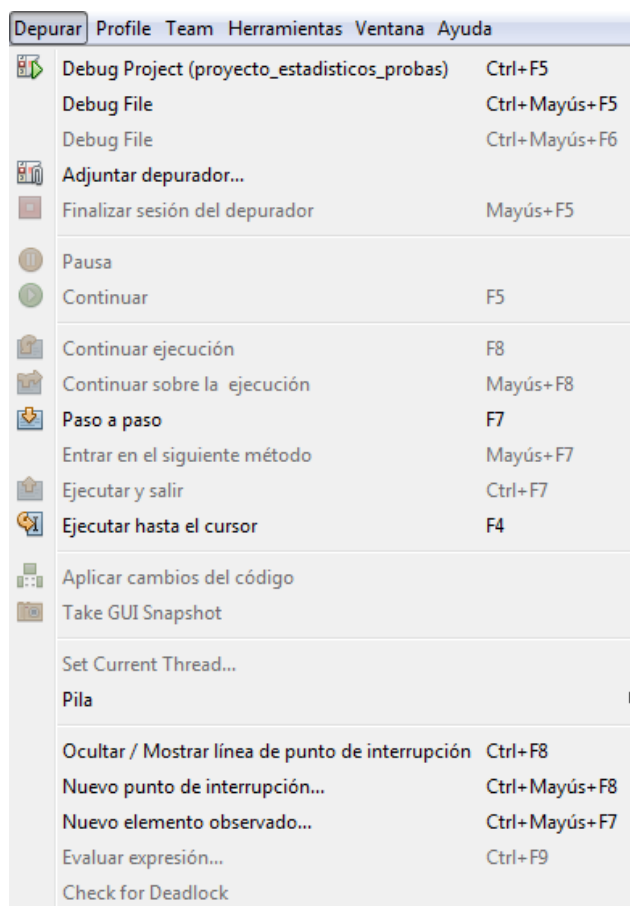
- Un proxecto establecido como proxecto principal, premendo Ctrl-F5 ou indo ao menú principal e elixindo *Debug -> Debug Project*. O proxecto principal fíxase con menú superior *Run > Set Main Project*.
- Un proxecto calquera dende a ventá *Proyectos*, seleccionando o proxecto, facendo clic dereito, e elixindo *Depurar*.
- Un arquivo fonte que se estea editando nese momento, indo ao menú principal e elixindo *Debug > Debug File*.
- Un arquivo calquera dende a ventá *Proyectos*, seleccionando o arquivo, facendo clic dereito, e elixindo *Debug File*.

En tódolos casos anteriores, a sesión de depuración empeza no arquivo seleccionado ou na clase principal do proxecto seleccionado e segue a execución ata que finalice normalmente o programa, encontre algún erro ou algún punto de interrupción.


Antes de comezar o debug, o mellor é crear un punto de interrupción antes das liñas conflitivas que queremos ver unha a unha. Para crear un punto de interrupción o máis sinxelo é hacer clic no número de liña, e marcará toda a liña con fondo vermello. Agora si podemos comezar a traza liña a liña.

Outra opción é comezar coa opción "Run to cursor"


No menú *Debug / Depurar*, pódese ver que a sesión de depuración tamén pode iniciarse para que a execución se faga paso a paso ou ata a liña na que estea o cursor.



Execución ata o cursor

A opción *Ejecutar hasta el cursor* (*Run to cursor*) permite executar o programa ata a localización do cursor no arquivo que se está editando e pausa o programa ata que se lle indica a seguinte operación a realizar na depuración. O arquivo editado debe ser chamado dende a clase principal do proxecto principal. Para realizar esta depuración débese situar o cursor no código fonte, premer **F4** ou seleccionar no menú principal *Debug>Ejecutar hasta el cursor* ou seleccionar  na barra de ferramentas da depuración.

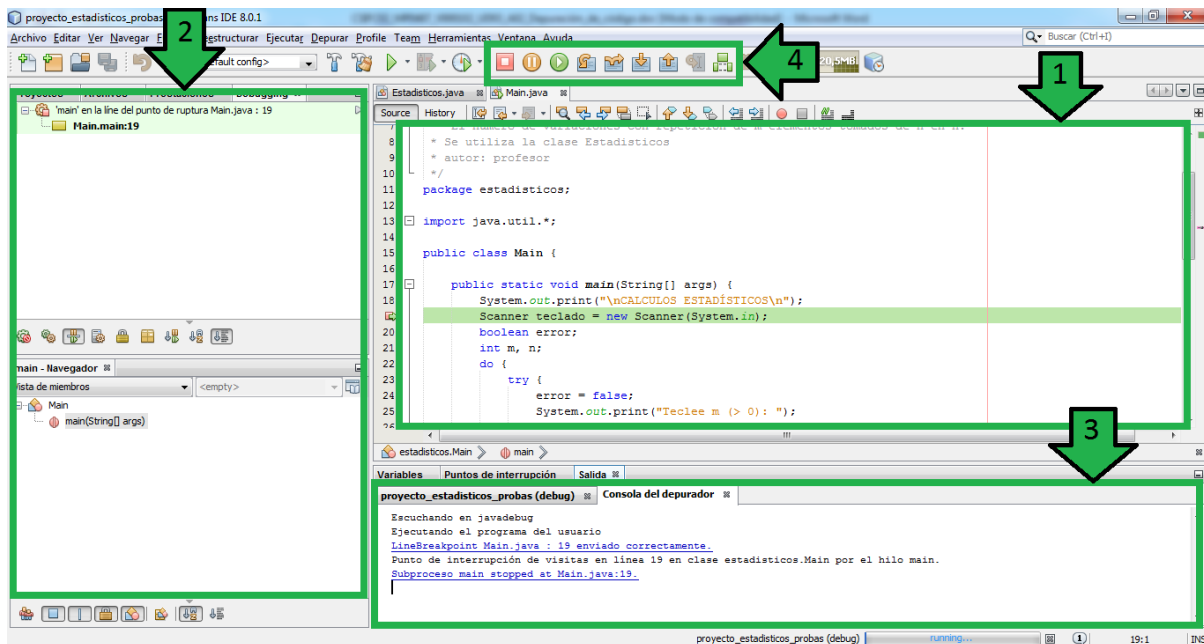
Execución paso a paso

A opción *Paso a paso* permite executar o programa liña a liña e pausa a execución ata que se lle indica a seguinte operación a realizar na depuración. Para iniciar a sesión de depuración paso a paso, elíxese *Depurar-> Paso a paso* no menú principal ou prémese **F7** ou  na barra de ferramentas da depuración e a execución deterase na liña coa primeira instrución executable. Cada vez que se queira executar paso a paso a seguinte liña, haberá que volver a pulsar F7. Se a liña de código está composta de varios métodos, aparecerá seleccionado cun borde negro o que se vai a executar paso a paso e poderase utilizar a tecla de tabulación para seleccionar un dos outros métodos. Pódese utilizar a tecla *[Enter]* ou *[F7]* para executar paso a paso o método seleccionado.

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

Pantalla de depuración

A pantalla de depuración aparece despois de iniciada a depuración e ten varias zonas. Por exemplo, despois de iniciada unha sesión de depuración paso a paso aparecen as zonas que se ven na imaxe seguinte.



Zona 1: Zona co código fonte en depuración. A seguinte liña a executar no proceso de depuración aparece marcada con cor de fondo verde e unha frecha verde na marxe esquerda.

Zona 2: Ventá *Debugging* con información sobre os procesos que se están executando. Na parte inferior desa ventá aparece un menú de iconas para poder cambiar a vista da información:

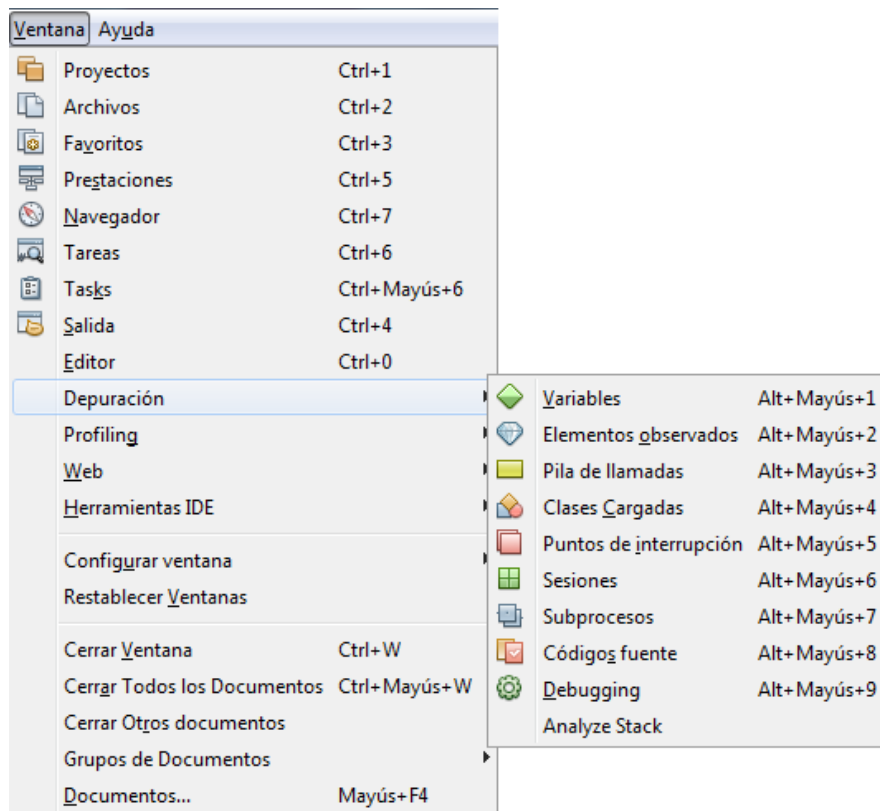


Zona 3: Zona de ventás:

- Ventás de saída ou *output* que se subdivide en:
 - *Consola del depurador* con información sobre o proceso de depuración.
 - *estadisticos(debug)*. Neste proxecto en concreto que ten entradas e saídas dende a consola realizaranse aquí as entradas de datos e verase a saída de resultados.
- Ventá *Variables* na que se pode ver e cambiar información sobre as variables locais e expresións.
- Ventá *Puntos de interrupción* ou *breakpoints* na que se pode ver e cambiar información sobre os puntos de interrupción.

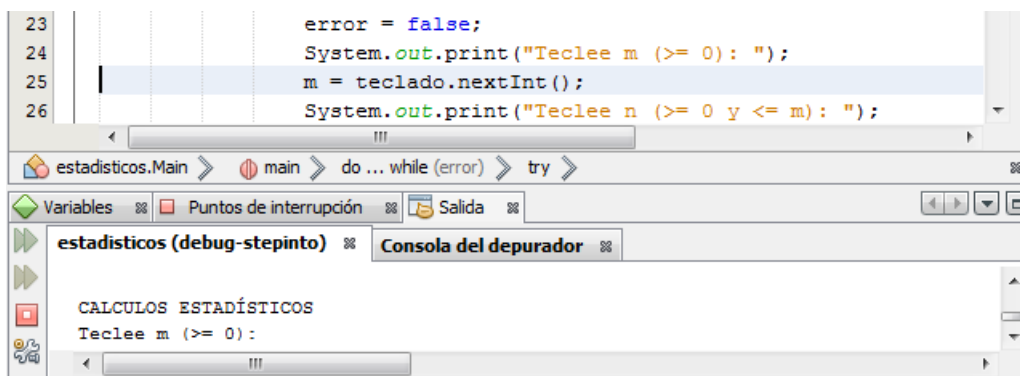
Zona 4: Barra de ferramentas de depuración para indicar o seguinte paso a realizar na depuración. Algunha das mesmas aparecen tamén activadas no menú principal *Depurar*. As barras de ferramentas que se ven poden decidilas desde o menú superior *View > Toolbars*.

Se non se ve algunha das ventás anteriores, pode accederse á opción *Ventana->Depuración* do menú principal e elixir a ventá que se desexa ver.













Durante o proceso de depuración pode ocorrer que a execución dunha liña de código precise dunha entrada por teclado e entón:

- A liña de código que ten a entrada pasa de ter fondo verde a ter fondo azul na ventá de edición.
- O proceso de depuración está detido ata que se teclee o dato na ventá de saída *estadísticos*.



Explicación rápida do significado de cada icona da barra de ferramentas de depuración (zona 4):



Icona	Descrición	Detalle
	Finalizar sesión do depurador (Maiúsculas+F5)	Finalizar instantaneamente a depuración.
	Pausa	Facer unha pausa no proceso de depuración.
	Continuar (F5)	Continuar a execución normal do programa despois de facer unha pausa.
	Continuar execución (F8)	Executa unha instrución. Se contén unha chamada a un método, este executase completo sen parar en cada unha das instrucións.
	Continuar sobre a execución (Maiúsculas+F8)	Continuar sobre a execución. É un refinamento de F8 para expresións con chamadas a métodos. Cada vez que nunha sesión de depuración paso a paso se utiliza este comando sobre unha expresión con chamada a métodos, párase a depuración antes de executar a chamada ao método actual podendo ver o historial dos valores de retorno dos métodos inmediatamente previos e o valor dos argumentos do método actual na ventá de Variables locais. Pode ser útil cando os valores de retorno dun método non se gardan nunha variable e por tanto non poden ser inspeccionados en tempo de depuración.
	Paso a paso (F7)	Executar Paso a paso. Permite entrar na execución paso a paso do método da liña actual. De haber máis dun método na liña, poderase: escoller o método que se vai depurar paso a paso utilizando as teclas de movemento do cursor ou a tecla tab e confirmando con F7, executar normalmente (F7) Nesta versión de NetBeans, por defecto, Step Into (F7) entra na execución paso a paso dos métodos API de Java. De querer que non se abran estes métodos e se executen, haberá que premer F8 en lugar de F7.
	Executar e saír (Ctrl+F7)	No caso de estar depurando dentro dun método, Ctrl+F7 finaliza a sesión de depuración do método e volve ao método que chamou ao actual. No caso de utilizarse no método principal, finaliza a sesión de depuración.
	Executar ata o cursor F4	Executar ata o cursor e espera instrucións para continuar coa depuración.
	Aplicar cambios do código	Aplicar cambios no código.
		Premer para activar a recollida de lixo.



Tarefa 3.4. Depurar de forma básica.

A tarefa consiste en realizar as seguintes depuracións sobre o proxecto Java *estadisticos*:

- Parte 1): Iniciar una depuración paso a paso do método *Main*, tecleando os valores $m=5$ e $n=2$. Executar paso a paso soamente o método *factorial(5)*. Finalizar a depuración e ver o resultado final.
- Parte 2): Colocar o cursor na liña 21 de *Main.java* e iniciar unha depuración ata o cursor, e teclear os valores $m=15$ e $n=3$. Finalizar esta depuración e ver o resultado final.

Recórdase que para abortar unha sesión de depuración antes de que acabe normalmente, hai que premer en  na barra de depuración e para terminar de executar de forma normal un método antes de que acabe a execución paso a paso, hai que premer en  na barra de depuración.

2.3 Inspección e modificación de variables, expresións e métodos

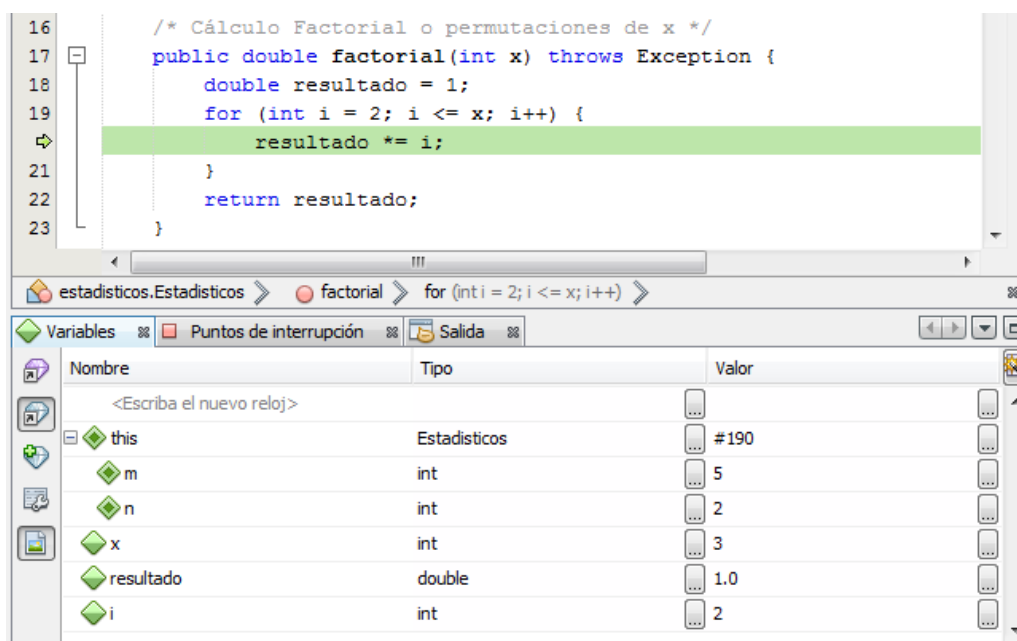
No código fonte

Durante unha sesión de depuración pódense ver o tipo e valor dunha variable situando o cursor sobre ela no código fonte:

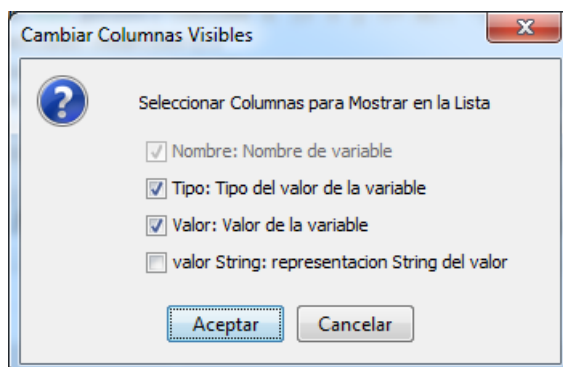
```
17 public double factorial(int x) throws Exception {
18     double res = (int) 2 = 1;
19     for (int i = 2; i <= x; i++) {
20         resultado *= i;
21     }
22     return resultado;
23 }
```

Na ventá Variables


Durante unha sesión de depuración pódese ver información sobre as variables locais na ventá *Variables* para variables locais e atributos de clase se existen.



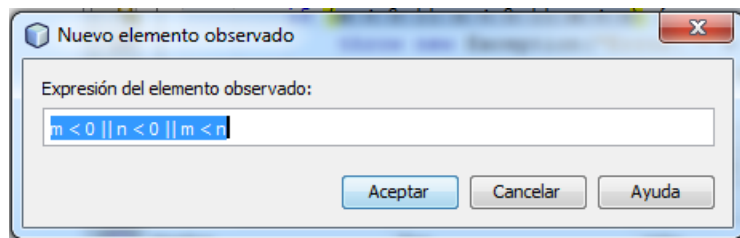
A icona situada á dereita  permite modificar a información que se ve.



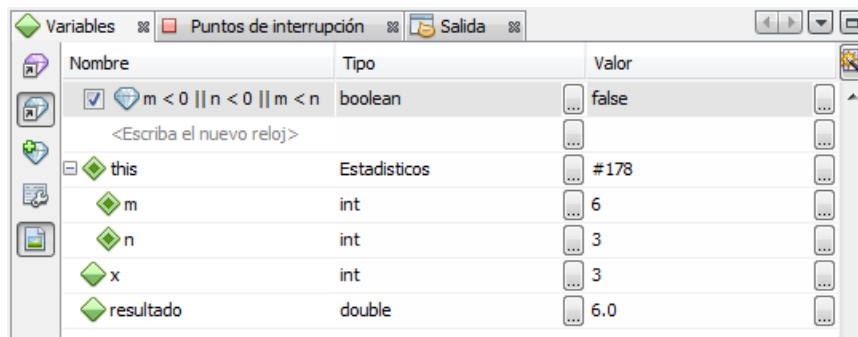
Ademais das variables locais pódense engadir expresións para observar (*watches*). Isto pódese facer de varias maneiras:


- Seleccionar a expresión no arquivo fonte editado, premer clic dereito e elixir *Nuevo elemento observado* ou premer Ctrl+Maiúsculas+F7.
- Seleccionar no menú principal *Depurar->Nuevo elemento observado*.
- Premer na icona  da ventá *Variables*.
- Teclear o novo elemento na liña da ventá *Variables* que pon *<Enter new watch>*.

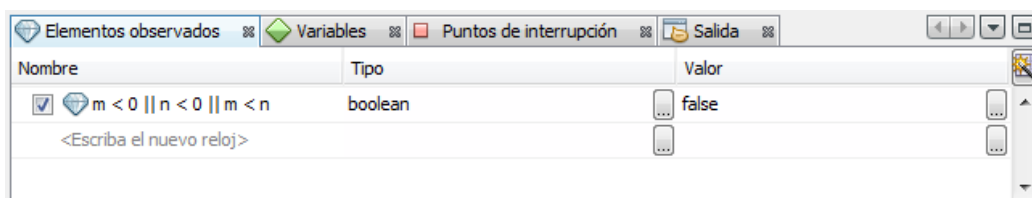
En calquera dos tres primeiros casos, aparece unha ventá na que ten que quedar definida a expresión que se quere observar.



A expresión aparece engadida na ventá *Variables* como un *watch*.




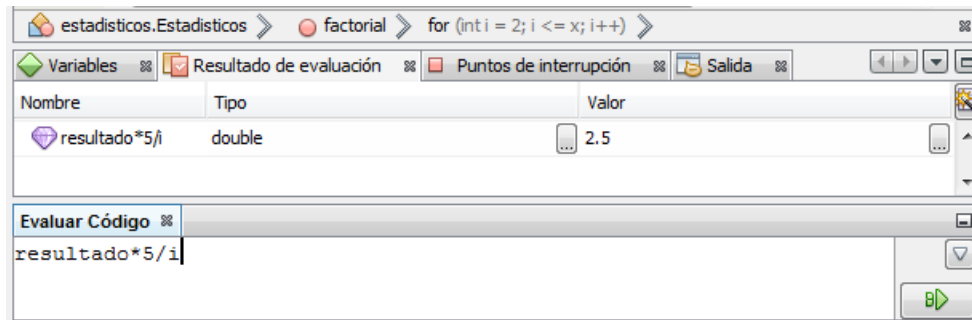
Os elementos observados poden moverse entre a ventá *Variables* e a ventá *Elementos observados* utilizando o interruptor .




Na ventá *Variables* tamén se pode facer clic dereito sobre o nome dunha variable ou expresión e facer cambios como por exemplo: eliminar desa ventá unha variable ou expresión, eliminar todas, ver o valor noutro formato ou editar.

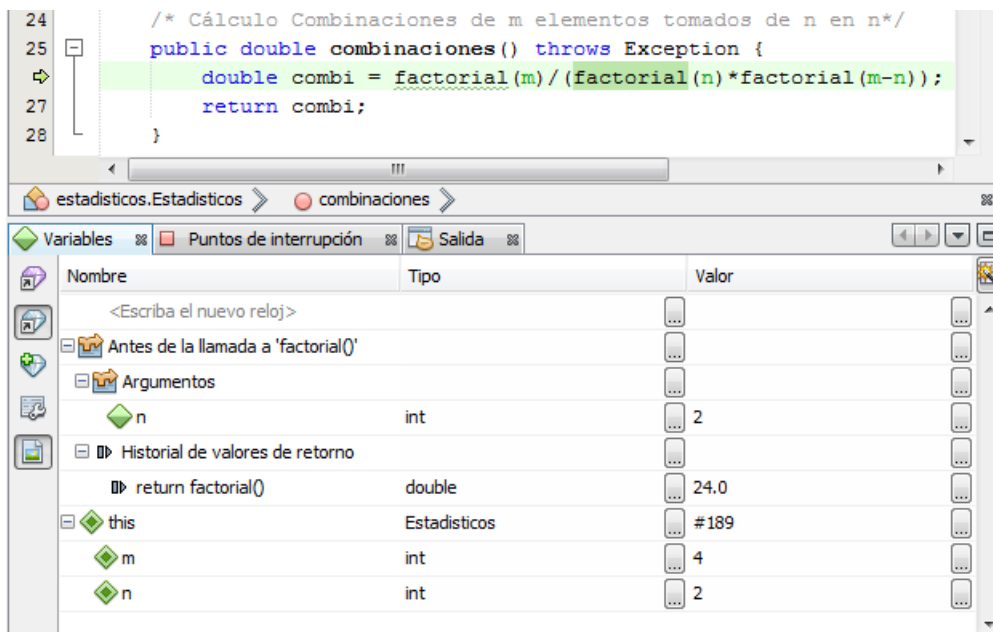
Co menú *Depuración->Evaluar expresión*

Durante unha sesión de depuración pódese ver o resultado dunha expresión dende o menú principal elixindo *Depurar->Evaluar expresión* ou premendo Ctrl+F9; ábrese a ventá *Evaluar Código* na que se pode teclear a expresión e avaliar o resultado nese momento premendo sobre o botón . Se a expresión non é posible no contexto actual non se podería ver o resultado. O resultado será similar ao seguinte:




Coa opción de depuración *Continuar sobre la ejecución*

Pódense ver os valores de retorno dos métodos previos (*Historial de valores de retorno*) e o valor dos parámetros do método seguinte (*Antes de la llamada a ...*), na ventá *Variables* cando nunha sesión de depuración se utiliza Maiúsculas+F8 (*Continuar sobre la ejecución*) ou se preme na icona  da barra de depuración sobre unha expresión con chamada a un método.



Modificación

Na ventá de avaliar expresións, na de variables e na de elementos observados pódese modificar o valor dunha variable e continuar o proceso de depuración con ese valor. Para iso faise clic ao carón do valor actual da variable, ou utilízase a icona  cando exista, para teclear o novo valor, prémese en *Aceptar* e continúaase coa depuración.



Tarefa 3.5. Depurar facendo inspección de variables, expresións e métodos, e modificando valores.

A tarefa consiste en realizar as seguintes depuracións sobre o proxecto Java *estadisticos*:

Parte 1) Depurar para poder ver o valor de retorno dos métodos factorial na liña 26 de Estadisticos.java.

```
double combi = factorial(m) / (factorial(n) * factorial(m-n));
```

Parte 2) Depurar para poder inspeccionar o valor da variable local *i* do método factorial(5) durante unha sesión de depuración.

Parte 3) Depurar para que en tempo de depuración se poida modificar o valor da variable local *i* do método factorial(5) e ver os cambios realizados nas variables locais.

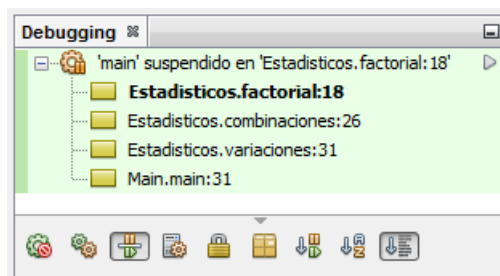
Parte 4) Depurar para que durante unha sesión de depuración se poida ver o valor da expresión *m-n*.

2.4 Pila (call stack)

A utilización da pila de chamadas é especialmente útil cando se utilizan varios fíos ou subprocesos durante a execución. Cando se inicia unha sesión de depuración, ábrese automaticamente a ventá *Debugging* con información sobre os subprocesos existentes e a pila de chamadas de cada un dos subprocesos suspendidos ou pausados; só unha desas chamadas é a chamada actual.

Ventá *Debugging*

A ventá *Debugging* ten o seguinte aspecto:



Esta imaxe indica unha sesión de depuración na que a execución de *main()* queda interrompida na liña 31 por unha chamada ao método *variaciones()*; a execución de *variaciones()* queda interrompida na liña 31 por unha chamada ao método *combinaciones()*; a execución de *combinaciones()* queda interrompida na liña 26 por unha chamada ao método *factorial()* e *factorial()* é o método actual pausado na liña 18.

A última chamada realizada é a chamada que se considera actual e indícase na pila en letra grosa. De consultar as variables locais veríanse as da chamada actual. Se os arquivos fontes están dispoñibles, pódese facer clic dereito na chamada e elixir *Ir a fonte* para ver o código fonte da chamada. A carón de cada proceso aparece unha icona con información sobre o proceso:

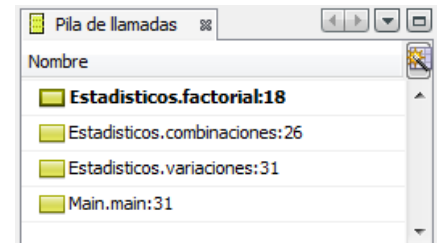
Icons	Description
	Indicates a thread that is running
	Indicates a thread suspended by hitting a breakpoint
	Indicates a suspended thread
	Indicates a thread group where all threads are running
	Indicates a thread group where all threads are suspended
	Indicates a thread group with running and suspended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

Pódese cambiar a vista desta información utilizando o menú de iconas da 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 a pila


A información sobre a pila de chamadas tamén se pode ver indo ó menú principal e elixindo *Ventana->Depuración->Pila de Chamadas* ou premer **Alt+Maiúsculas+F3** e abrírase a ventá *Pila de Chamadas* na zona de ventás. A información para cada chamada está marcada por unha icona e a descrición da chamada.



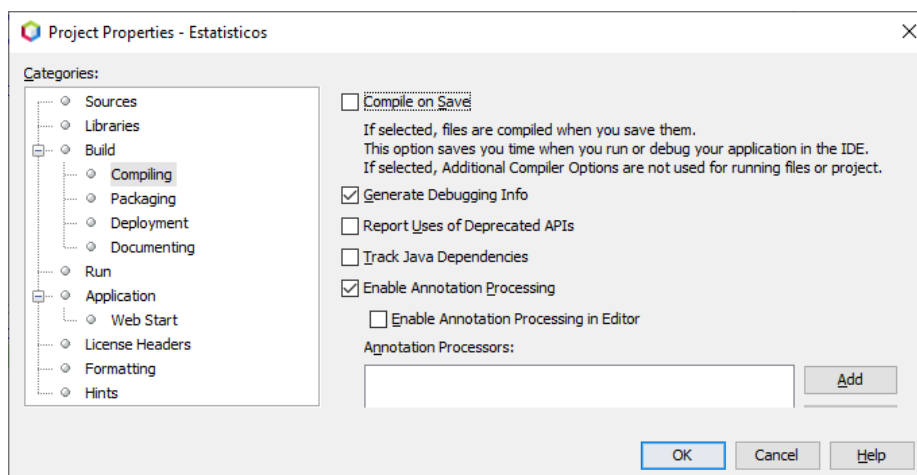
Tarefa 3.6. Depurar inspeccionando a pila.

A tarefa consiste en depurar o proxecto Java *estadísticos* ata ter ver na ventá da pila 4 chamadas.

2.5 Aplicar cambios no código

Pódense facer certas modificacións no código en tempo de depuración sen ter que reiniciar o programa. Para corrixir o código, débese corrixir na ventá de edición, ir ao menú principal e elixir *Depurar->Aplicar cambios del código* ou premer en  na barra de ferramentas de depuración, para recompilar e facer a reparación do código fonte.

(Olla: nas propiedades do proxecto hai que desactivar a opción "Compile on save", se non, ese botón aparece desactivado)



Consideracións xerais:

- Se hai erros durante a compilación, non se realizan os cambios e hai que arranxar os erros.

- Se non hai erros, o código obxecto resultante cambiarase polo que se estaba executando na depuración e:
 - se os cambios do código se fan dentro do método actual que se está a depurar, a pila de chamadas modifícase eliminando a chamada a ese método para permitir volver a chamar a ese módulo, pero
 - se os cambios se fan despois de haber chamado ao método, non se modificará a pila e para utilizar de novo o código modificado, debe eliminar as chamadas da pila que conteñan ese código.
- Están excluídas as seguintes modificacións:
 - Cambiar un modificador dun campo, un método ou unha clase.
 - Agregar ou quitar métodos ou campos.
 - Cambiar a xerarquía de clases.
 - Cambiar clases que non foron cargadas na máquina virtual.



Tarefa 3.7. Facer cambios no código mentres se depura.

A tarefa consiste en modificar do código do proxecto Java *Estadisticos* en tempo de depuración. A modificación a realizar pode ser para evitar que se calcule o factorial cando m ou n sexan maiores que 170 xa que se provocaría un desbordamento e retornaría o valor *Infinity*. Os pasos a seguir serían:

- Comezar a sesión de depuración, parando antes de que o usuario introduza os números
- Modificar o construtor de Estadisticos.java engadindo a condición `|| m > 170` ao if para que se produza a excepción, e tamén modificar o texto de aviso.
- Premer o botón “Aplicar cambios en el código”
- Seguir a execución introducindo para m un valor superior a 170.
- Comprobar que se efectuou a modificación no código.

2.6 Punto de interrupción

Definición

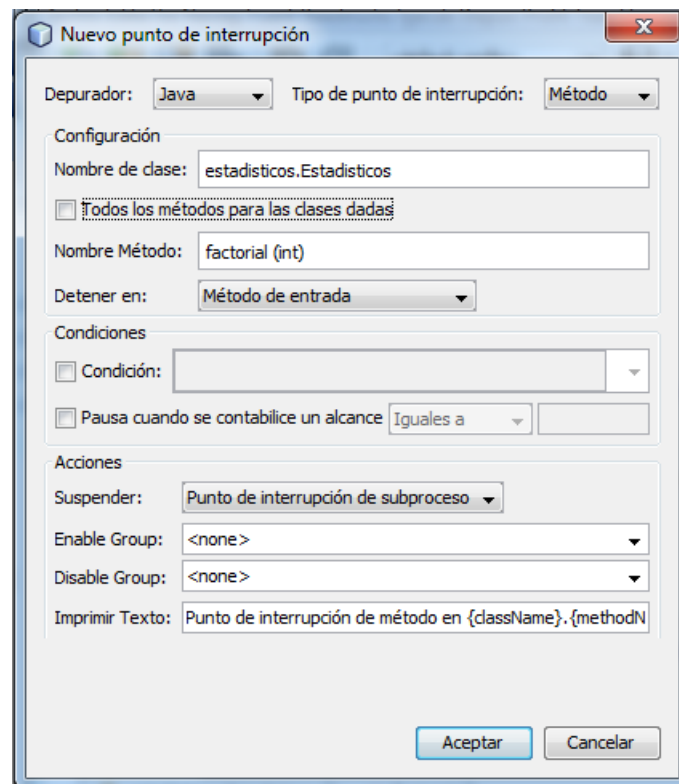
Un punto de interrupción ou ruptura ou breakpoint é unha marca no código fonte que indica ao depurador que se deteña nese punto e espere instrucións para continuar, podendo nese tempo facer operacións de inspección de variables, expresións ou código. Os puntos de interrupción de Java defínense a nivel global e afectan a tódolos proxectos que inclúan o código fonte que ten o punto de interrupción. Por exemplo, se Estadisticos.java tivera un punto de interrupción no método factorial(), cada vez que se depure un proxecto que inclúa esa clase, a sesión de depuración deteríase nese método. NetBeans permite varios tipos de puntos de interrupción:

- Liña: para que se pare ao chegar a esa liña.
- Clase: para que se pare no momento de cargar a clase.
- Excepción: para que se pare cando se detecte unha excepción independentemente de se o programa controla esa excepción.
- Campo: para que se pare cando se accede e/ou modifica o campo dunha clase.
- Método: para que se pare cando se entra e/ou se sae dun método.
- Subproceso: para que se pare cando se empeza e/ou finaliza un subproceso ou fío (*thread*).

Establecer un punto de interrupción

Para establecer un punto de interrupción, habrá que seleccionar o elemento do código no que se desexa establecer o punto de interrupción e elixir no menú *Depurar->Nuevo punto de interrupción* ou premer Ctrl+Maiúsculas+F8. Aparece a caixa de diálogo *Nuevo punto de interrupción* con información por defecto relacionada co elemento seleccionado e na que terán que facerse os axustes convenientes. O IDE indica o punto de interrupción establecido mediante unha icona na marxe esquerda do código fonte e os puntos de interrupción de liña indícaos ademais poñendo a liña con fondo roxo.

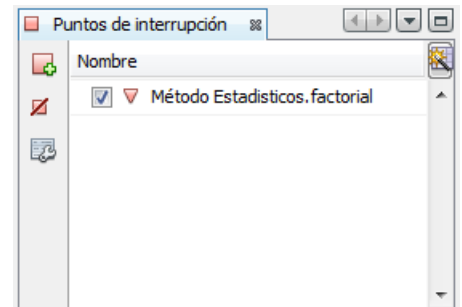
Na seguinte imaxe móstrase un exemplo de punto de interrupción de método para que a depuración se interrompa cando se entre no método factorial da clase *Estadísticos*:



No código fonte aparecerá o punto de interrupción de método marcado cunha icona como aparece na seguinte imaxe:

```
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      }
```


Durante a sesión de depuración, NetBeans comproba a validez dos puntos de interrupción e se o encontra non válido indícao mediante a icona "rota" no código fonte e mostra unha mensaxe de erro na consola do depurador. Ademais é posible ver a ventá *Puntos de interrupción* con información sobre os puntos de interrupción. Se a ventá non está aberta pódese abrir dende o menú principal *Ventana->Depuración->Puntos de interrupción* ou premendo Alt+Maiúsculas+F5.



As iconas posibles para marcar os puntos 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

Un punto de interrupción de liña é un os máis utilizados e por iso hai varias maneiras de establecelos:

- O máis sinxelo é facer clic sobre a marxe esquerda da ventá de edición á altura da liña na que se desexa colocar o punto de interrupción.
- Colocar o cursor sobre a liña na que se encontre a instrución onde queremos poñer dito punto, facer clic co botón dereito e seleccionar a opción *Ocultar/Mostrar liña de punto de interrupción* (Toggle Line Breakpoint) ou premer as teclas Ctrl + F8.
- Colocar o cursor sobre a liña na que se encontre a instrución onde queremos poñer dito punto, e no menú principal elixir *Depurar->Ocultar/Mostrar liña de punto de interrupción* ou premer as teclas Ctrl + F8.

No código fonte aparecerá o punto de interrupción marcado como se indica na seguinte imaxe.

```

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  }

```

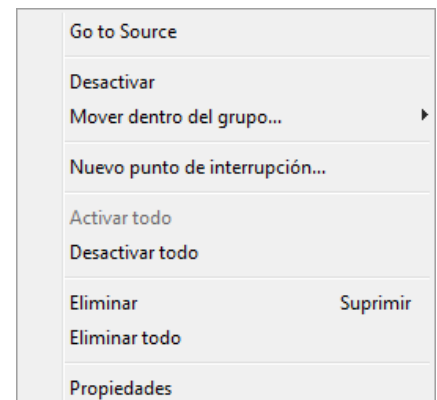
Executar ata o punto de interrupción

Para poder executar un proxecto principal ata un punto de interrupción, hai que ter definido o punto de interrupción e depurar o proxecto elixindo por exemplo no menú principal a opción *Depurar->Debug project(nome de proxecto)*, que executará o programa principal ata o primeiro punto de interrupción, ou excepción ou ata o final se non existen puntos de interrupción. A partir do punto de interrupción, pódese seguir depurando coas opcións xa vistas.

Modificar un punto de interrupción

Dende que se establece un punto de interrupción, pódese desactivar (queda rexistrado, pero non en uso) se é que está activado, activar se é que estaba desactivado, eliminar (desaparece), ou modificar. Todas estas operacións empézanse a realizar dende a ventá *Puntos de interrupción* e son:

- Unha forma rápida de activar ou desactivar un punto é marcar ou desmarcar o textbox correspondente na ventá anterior.
- Unha forma rápida de eliminar un punto é colocar o cursor sobre o nome do punto de interrupción na ventá anterior e premer Supr.
- Tódalas operacións de modificación dun punto de interrupción pódense facer colocando o rato sobre o nome do punto de interrupción na ventá anterior e facendo clic co botón dereito. Aparece unha lista de operacións posibles:
 - No caso de estar sobre un punto de interrupción activo, na ventá de opcións aparece dispoñible a opción *Desactivar*; se estivera desactivado, aparecería no seu lugar a opción *Activar*.
 - As opcións de desactivar todo, activar todo ou eliminar todo terán efecto sobre tódolos puntos de interrupción da ventá.
 - A opción *Propiedades* visualiza a ventá *Propiedades de punto de interrupción*, na que se pode axustar a configuración do punto de interrupción.
 - Tamén se pode crear un novo punto de interrupción.



Poñer condicións ao punto de interrupción

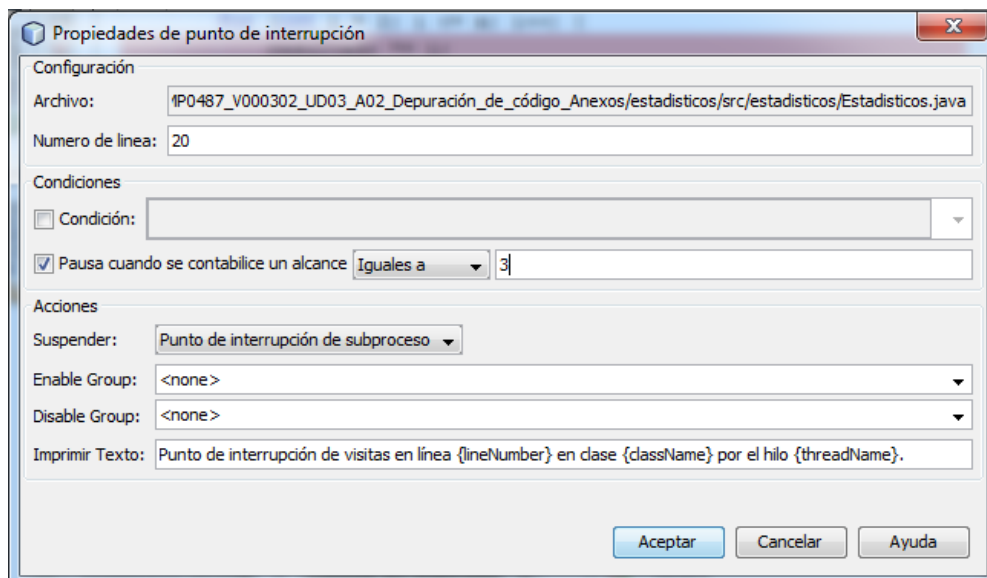
Pódense poñer condicións para que un punto de interrupción pause unha depuración; algunhas son comúns para tódolos puntos de interrupción e outras depende de se non son de fío, se son de clase ou se son de excepción.

Condições válidas para tódolos puntos de interrupción

Tódolos puntos de interrupción teñen a posibilidade de pausar unha depuración en función dunha frecuencia establecida, marcando o checkbox *Pausa cuando se contabilice un alcance*, seleccionando un

criterio da lista despregable (*Igual a, é maior que, é múltiplo de*) e establecendo un valor numérico para ese criterio na ventá de propiedades do punto.

A seguinte imaxe mostra a condición de que o punto de interrupción de liña situado na liña 20 de Estadisticos.java se active a terceira vez que se pase por ela durante unha sesión de depuración.

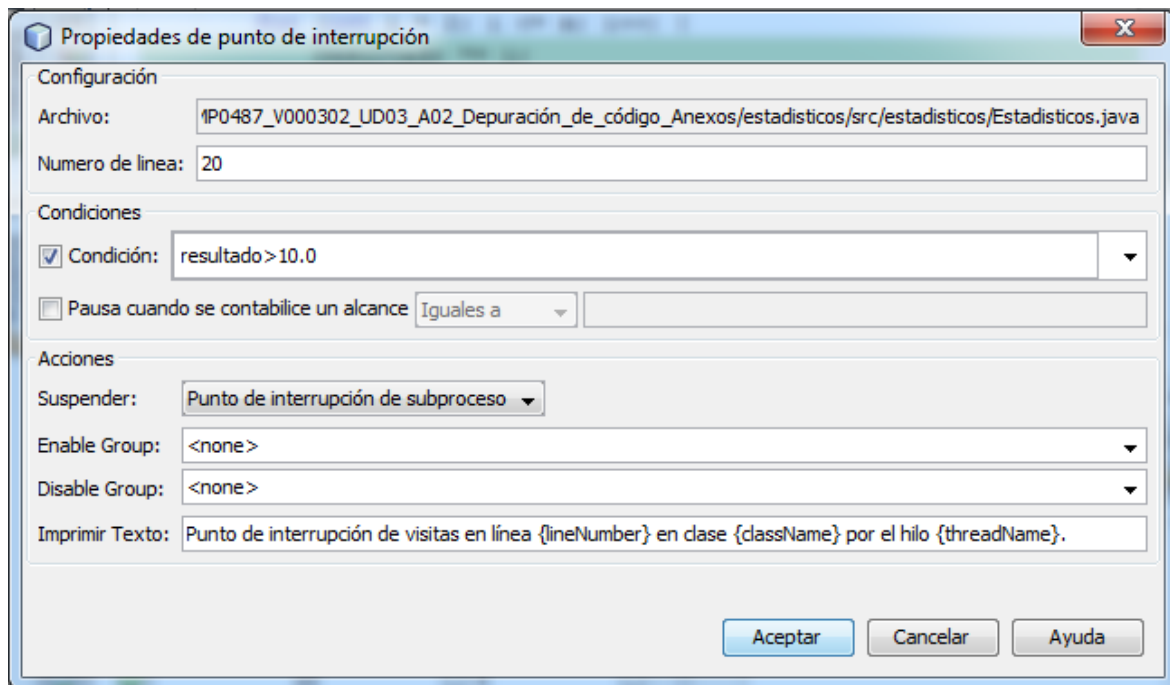


Condicions válidas para tódolos puntos agás os de tipos *thread*

Os puntos de interrupción que non son tipo fío, teñan a posibilidade de pausar unha depuración cando unha determinada condición é certa. Esta condición establécese na ventá de propiedades do punto de interrupción, seleccionando *Condición* e tecleando a condición. A condición debe seguir as normas de sintaxe de Java e pode incluír variables e métodos utilizados no contexto actual coas seguintes excepcións:

- As importacións son ignoradas. Débense usar nome completos como `obj instanceof java.lang.String`
- Non se pode acceder directamente a métodos e variables de clases externas. Débese utilizar `this.nome` ou `this.$1`

A seguinte imaxe mostra a condición de que o punto de interrupción de liña situado na liña 20 de Estadisticos.java se active cando a variable resultado teña un valor maior a 10 durante unha sesión de depuración.



Condicions específicas para os puntos de clase e excepción

Pódense dar as seguintes condicións específicas:

- Os puntos de interrupción de clases tamén permiten poñer como condición a exclusión dalgunha clase.
- Os puntos de interrupción de excepcións permiten poñer como condición un filtrado de clase a incluír ou excluír.



Tarefa 3.8. Depurar utilizando puntos de interrupción.

A tarefa consiste en facer as seguintes depuracións utilizando puntos de interrupción de liña, de método, de liña con condición e contador:

Parte 1) Definir un punto de interrupción na liña 21 de Main.java. Executar unha depuración para ver a saída de resultados ata ese momento. Finalizar a depuración. Desactivar ese 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. Executar a depuración e cada vez que se sae do método, ver como varía a pila na ventá da pila de chamadas, e ver os valores dos elementos *x* e *resultado* na ventá de elementos observados. Eliminar ese punto de interrupción.

Parte 3) Definir un punto de interrupción que pare a execución para os valores de *m*=4 e *n*=2, na liña 22 de Estadisticos.java.

Parte 4) Eliminar tódolos puntos de interrupción.



Tarefa 3.9. Depuración paso a paso

- 1) Crea un proxecto chamado DebugTelegrama cun paquete chamado probas e unha clase Telegrama.java co seguinte código:

```
package probas;
import java.util.Scanner;
```

```

/** Este programa calcula el costo de un telegrama dependiendo de su tipo y el
número de palabras.*/
public class Telegrama {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        String sTipoTelegrama;
        char tipoTelegrama;
        int numPalabras;
        double costo;

        // Lee el tipo de telegrama como una cadena
        System.out.println("Tipo de telegrama o/u? ");
        sTipoTelegrama = entrada.next();

        // Obten el primer caracter de la cadena
        tipoTelegrama = sTipoTelegrama.charAt(0);

        // Lee el número de palabras del telegrama
        System.out.println("Número de palabras? ");
        numPalabras = entrada.nextInt();

        // Si el tipo de telegrama es ordinario
        if(tipoTelegrama == 'O' || tipoTelegrama == 'o')

        // Si el número de palabras es menor o igual a 10
        if(numPalabras <= 10) costo = 25;

        // Si el número de palabras es mayor a 10
        else costo = 25 + 5 * (numPalabras - 10);

        // Si el tipo de telegrama es urgente
        else if(tipoTelegrama == 'U' || tipoTelegrama == 'u')

        // Si el número de palabras es menor o igual a 10
        if(numPalabras <= 10) costo = 40;


        // Si el número de palabras es mayor a 10
        else costo = 40 + 7.5 * (numPalabras - 10);
        else costo = 0;






        // Escribe el costo del telegrama
        System.out.println("Costo del telegrama: " + costo);
    }
}

```

- 2) Comeza o debug con Cntrl+F5.
- 3) Ver as saídas de consola na ventá "Output" e teclee "ordinario" na entrada solicitada.
- 4) Inspeccionar o contido da variable *sTipoTelegrama*
- 5) Executa a instrución que obtén que obtén o primeiro carácter da cadea e a almacena na variable *tipoTelegrama*. O seu valor é o esperado?
- 6) Sigue executando ata onde nos pide o número de palabra e teclee "8".
- 7) Desde a ventá de Inspección (new Watch), Inspecciona o valor da variable *numPalabras*. É o esperado? Cales son as teclas aceleradoras para *New Watch*?
- 8) Executa a seguinte instrución, o *if*, e verifica que o salto é o correcto. Canto vale a variable *costo*?
- 9) Cal é a seguinte instrución? Como remata o programa?
- 10) Repite todo o proceso con tipo de telegrama *ordinario* e 20 palabras.

Lembramos:

	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.
	Finish Debugger Session (Mayúsculas + F5). Termina la depuración del programa.



Tarefa 3.10. Execución ata unha instrución dada

Sobre a clase da tarefa anterior:

- 1) Coloca o cursor na liña co *if* e executa “Run to Cursor” (F4).
- 2) Coloca o cursor nunha liña do final e repite a operación.
- 3) Prema F5, *Continue*. Que ocorre?
- 4) Finalizamos a sesión de depuración mediante o menú *Debug > Finish* (May +F5).



Tarefa 3.11. Punto de ruptura

Substitúe o código anterior polo seguinte:

```
package pruebas;
import java.util.Arrays;
public class pruebaDebug {
    public static void main(String[] args) {
        int [] arr = new int [] {1,2,3,4,5};
        for (int i=0;i<arr.length;i++)
            arr[i]++;
        System.out.println (Arrays.toString(arr));
        for (int i=0;i<arr.length;i++)
            arr[i]*=2;
        System.out.println (Arrays.toString(arr));
    }
}
```

- 1) Coloca o cursor na liña co *println*, entre os dous “for” e selecciona a opción *Toggle Breakpoint* (Ctrl + F8).
- 2) Mostrar na ventá de Watch o contido de todas as posicións do Array.
- 3) Selecciona *Debug / Debug Project* (Cntr+ F5)
- 4) Mostrar na ventá de Watch o contido de todas as posicións do Array.

3. Probas unitarias

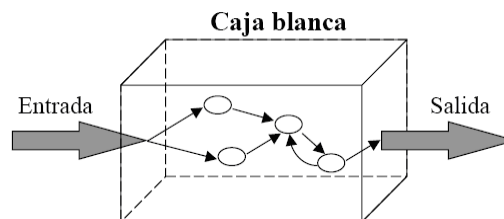
3.1 Técnicas de deseño de casos de proba

O deseño de casos de proba está limitado pola imposibilidade de probar exhaustivamente o software. Por exemplo, de querer probar tódolos valores que se poden sumar nun programa que suma dous números enteiros de dúas cifras (do 0 ao 99), deberíamos probar 10000 combinacións distintas (variacións con repetición de 100 elementos tomados de 2 en 2 = 100 elevado a 2) e aínda teríamos que probar todas as posibilidades de erro ao introducir datos (como teclear unha letra no canto dun número).

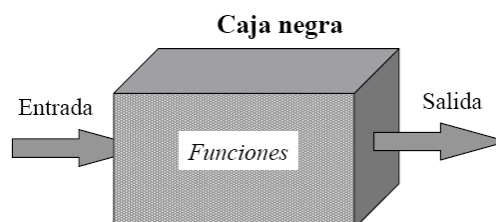
As técnicas de deseño de casos de proba teñen como obxectivo conseguir unha confianza aceptable en que se detectarán os defectos existentes, xa que a seguridade total só pode obterse da proba exhaustiva, que non é practicable sen consumir unha cantidade excesiva de recursos. Toda a disciplina de probas debe moverse nun equilibrio entre a dispoñibilidade de recursos e a confianza que achegan os casos para descubrir os defectos existentes.

Xa que non se poden facer probas exhaustivas, a idea fundamental para o deseño de casos de proba consiste en elixir algúns deles que, polas súas características, considéranse representativos do resto. A dificultade desta idea é saber elixir os casos que se deben executar xa que unha elección puramente aleatoria non proporciona demasiada confianza en detectar os erros presentes. Existen tres enfoques principais para o deseño de casos non excluíntes entre si e que se poden combinar para conseguir unha detección de defectos máis eficaz:

- Enfoque estrutural ou de caixa branca tamén chamado enfoque de caixa de cristal: Fíxase na implementación do programa para elixir os casos de proba².



- Enfoque funcional ou de caixa negra: Consiste en estudar a especificación das funcións, as súas entradas e saídas³.



- Enfoque aleatorio: Utiliza modelos, moitas veces estatísticos, que representen as posibles entradas ao programa para crear a partir delas os casos de proba.

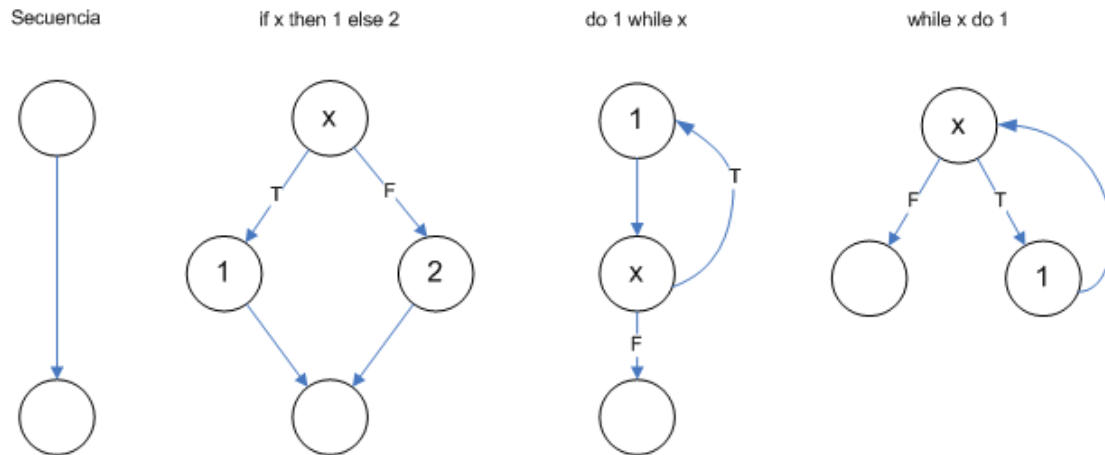
² Imaxe extraída de http://osl2.uca.es/wikihaskell/index.php/Pruebas_Unitarias_para_Haskell

³ Imaxe extraída de http://osl2.uca.es/wikihaskell/index.php/Pruebas_Unitarias_para_Haskell

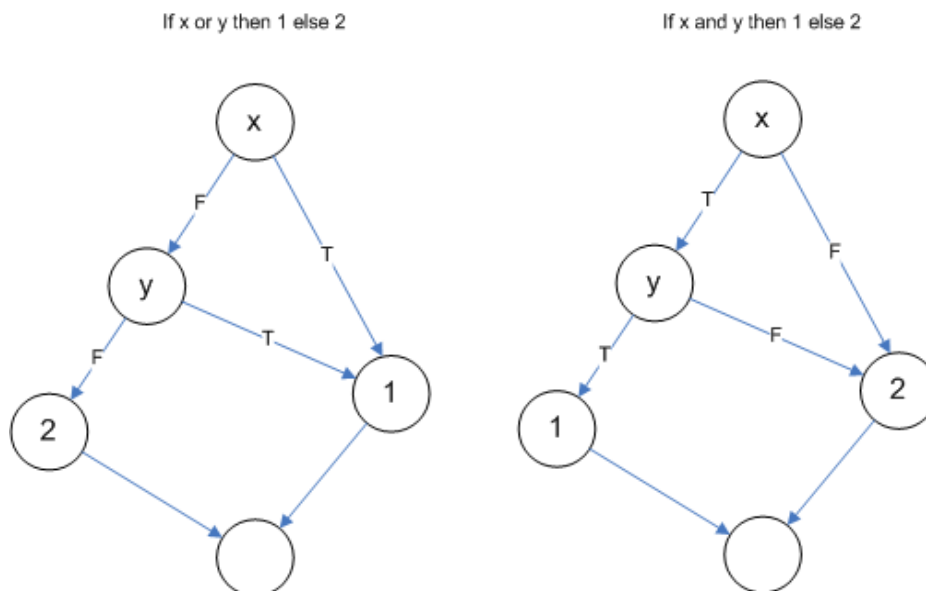
3.2 Probas unitarias estruturais

Para traballar coas técnicas estruturais imos realizar grafos de fluxo dos programas ou funcións a probar. Isto non é estritamente necesario pero debuxalos axuda a comprender o funcionamento das técnicas de proba de caixa branca. Os grafos que se usarán serán grafos fortemente conexos, é dicir, sempre existe un camiño entre calquera par de nodos que se elixan e para subsanar que o nodo primeiro e o último estean directamente conectados, engadirase un arco ficticio que os una.

Grafos básicos de fluxo:



Grafos para condicións múltiples:



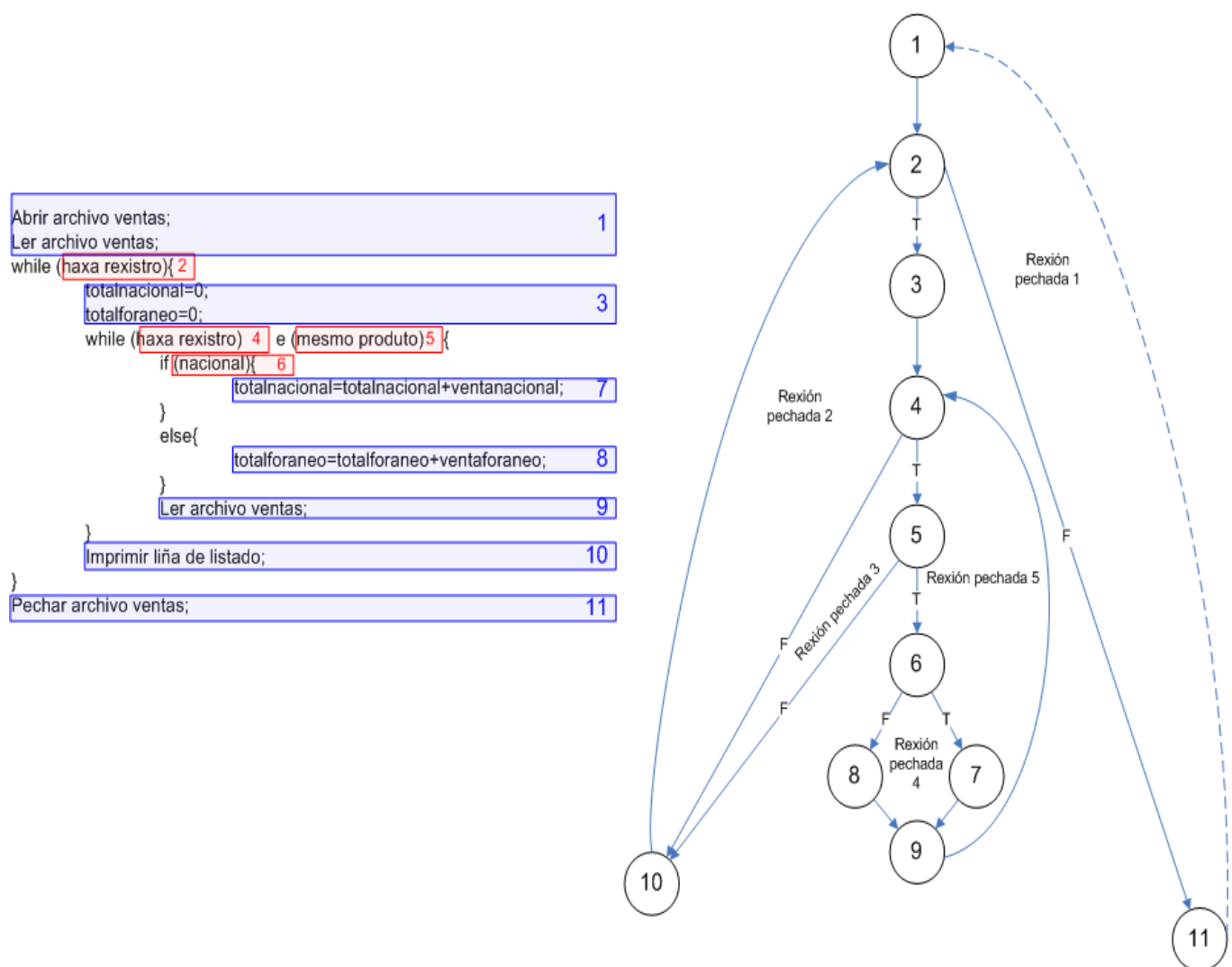
Criterios de cobertura lóxica de Myers

O deseño de casos ten que basearse na elección de camiños importantes que ofrezan unha seguridade aceptable en descubrir defectos. Utilízanse os chamados criterios de cobertura lóxica. Os criterios de cobertura lóxica de Myers móstranse ordenados de menor a maior esixencia, e por tanto, custo económico:

- Cobertura de sentenzas: xerar os casos de proba necesarios para que cada sentenza se execute, polo menos, unha vez.

- Cobertura de decisións: xerar casos para que cada decisión teña, polo menos unha vez, un resultado verdadeiro e, polo menos unha vez, un falso.
- Cobertura de condicións: xerar os casos de proba necesarios para que cada condición de cada decisión adopte o valor verdadeiro polo menos unha vez e o falso polo menos unha vez. Por exemplo, a decisión: `if ((a==3) || (b==2))` ten dúas condicións: `(a==3)` e `(b==2)`.
- Criterio de decisión/condición: consiste en esixir o criterio de cobertura de condicións obrigando a que se cumpra tamén o criterio de decisións.
- Criterio de condición múltiple: No caso de que se considere que a avaliación das condicións de cada decisión non se realiza de forma simultánea, pódese considerar que cada decisión multicondicional descomponse en varias decisións unicondicionales.
- Criterio de cobertura de camiños: Débese executar cada un dos posibles camiños do programa polo menos unha vez. Defínese camiño como unha secuencia de sentenzas encadeadas desde a sentenza inicial do programa até a sentenza final. O número de camiños, mesmo nun programa pequeno, pode ser impracticable para as probas. Para reducir o número de camiños a probar, pode utilizarse a complexidade ciclomática de McCabe.

Exemplo de grafo de fluxo fortemente conexo dun anaco de pseudocódigo:



Complexidade ciclomática de McCabe

A complexidade ciclomática é unha métrica que nos indica o número de camiños independentes que ten un grafo de fluxo. McCabe definiu como un bo criterio de proba o probar un número de camiños independentes igual ao da métrica. Un camiño independente é calquera camiño que introduce, polo menos, un novo conxunto de sentenzas de proceso ou unha condición, respecto dos camiños existentes. En termos do diagrama de fluxo, un camiño independente está constituído polo menos por unha aresta que non fose percorrida nos camiños xa definidos anteriormente. Na identificación dos distintos camiños débese ter en conta que cada novo camiño debe ter o mínimo número de sentenzas novas ou condicións novas respecto dos que xa existen. Desta maneira téntase que o proceso de depuración sexa máis sinxelo.

A complexidade de McCabe $V(G)$ pódese calcular das seguintes tres maneiras, a partir dun grafo de fluxo G fortemente conexo:

- $V(G) = a - n + 2$
 - a : número de arcos ou arestas sen contar o que une o primeiro nodo có último
 - n : número de nodos
- $V(G) = r$
 - r : número de rexións pechadas do grafo contando a que forma o arco que une o primeiro nodo có último
- $V(G) = c + 1$
 - c : número de nodos de condición

Por exemplo, a complexidade ciclomática de McCabe para o grafo de fluxo do anaco de pseudocódigo anterior é 5, é dicir, é o valor obtido por calquera das tres fórmulas anteriores:

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

Unha vez calculada a complexidade ciclomática debemos elixir tantos camiños independentes como o valor da complexidade. Para elixir estes camiños, comezamos definindo un camiño inicial e a continuación imos creando novos camiños variando o mínimo posible o camiño inicial. Para executar un camiño pode ser necesaria a súa concatenación con algún outro.

Para o exemplo anterior teríamos 5 camiños que se representan mediante o número dos nodos do grafo que recorren e os bucles mediante puntos suspensivos despois do nodo de control do bucle:

- 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-...



Tarefa 3.12. Representar grafos de fluxo, calcular a complexidade ciclomática de McCabe e obter os camiños

Podes empregar a ferramenta online draw.io.

A tarefa consiste en representar o grafo de fluxo, calcular a complexidade de McCabe e detallar os camiños para os métodos seguintes:

Parte 1) Método *calcularDivision* do proxecto *proyecto_division*.

Parte 2) Método *factorial* do proxecto *proyecto_factorial*.

Parte 3) Método *busca* do proxecto *proyecto_arrays*.

Parte 4) Método *obtenerAcronimo* do proxecto *proyecto_acronimos*.

- O método *calcularDivision* recibe un dividendo e un divisor de tipo float e devolve o resultado da división tamén de tipo float sempre que o divisor non sexa 0, en cuxo caso xera unha excepción.

```
package proyecto_division;
public class Division {
    public float calcularDivision(float dividendo, float divisor) throws Exception {
        if (divisor == 0) {
            throw (new Exception("Error. El divisor no puede ser 0."));
        }
        float resultado = dividendo / divisor;
        return resultado;
    }
}
```

- O método *factorial* recibe un número n de tipo byte e devolve o seu factorial de tipo float agás no caso de que sexa negativo, en cuxo caso xera unha excepción. O factorial dun número n é o produto de tódolos números menores que el ata o número 2. Casos especiais do factorial son factorial(0)=1 e factorial(1)=1.

```
package proyecto_factorial;
public class Factorial {
    public float factorial(byte n) throws Exception {
        if (n < 0) {
            throw new Exception("Error. El número tiene que ser >=0");
        }
        float resultado = 1;
        for (int i = 2; i <= n; i++) {
            resultado *= i;
        }
        return resultado;
    }
}
```

- O método *busca* recibe un carácter c e un array de caracteres v de 10 elementos como máximo ordenados de forma ascendente. Devolve o valor booleano *true* ou *false* segundo encontre o carácter no array ou non. A busca é dicotómica, é dicir, a primeira busca faise tendo en conta todo o array pero nas seguintes só se ten en conta un segmento del obtido mediante o cálculo do índice metade do segmento e a comparación de c có elemento almacenado nese índice; se coincide, finaliza a busca e encontrouse o carácter no array; se c é menor, o seguinte segmento será a primeira metade do actual; se c é maior, o seguinte segmento será a segunda metade do actual. Se este proceso finaliza cun segmento nulo e non se encontrou o carácter é que non existe.

```
package proyecto_arrays;
public class OperacionesArrays {
    public boolean busca(char c, char[] v) {
        int a, z, m;
        a = 0;
        z = v.length - 1;
        boolean resultado=false;
        while (a <= z && resultado==false) {
            m = (a + z) / 2;
            if (v[m] == c) {
```

```

        resultado=true;
    }
    else
    {
        if (v[m] < c) {
            a = m + 1;
        }
        else{
            z = m - 1;
        }
    }
}
return resultado;
}}

```

- O método *obtenerAcronimo* recibe unha cadea de caracteres e retorna unha cadea co acrónimo correspondente. O acrónimo está formado polo primeiro carácter de cada palabra seguidos dun punto cando o carácter é diferente de espazo en branco.

```

package proyecto_acronimos;
public class Acronimos {
    public String obtenerAcronimo(String cadena){
        String resultado="";
        char character;
        int n=cadena.length();
        for(int i=0;i<n;i++){
            character=cadena.charAt(i);
            if(character!=' '){
                if (i==0){
                    resultado=resultado+character+'.';
                }
                else{
                    if(cadena.charAt(i-1)==' '){
                        resultado=resultado+character+'.';
                    }
                }
            }
        }
        return resultado;
    }
}

```

3.3 Probas unitarias funcionais

Chegaría cunha proba de caixa branca para considerar probado un anaco de código? A resposta é non, e demóstrase co seguinte código:

```

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

```

Hai dous camiños posibles que se recorren cos valores $x=5$, $y=5$, $z=5$ e $x=4$, $y=3$, e $z=5$, que confirman a validez do código, pero cos valores $x=2$, $y=5$, e $z=3$ fallaría o código. Do que se deduce, que se necesitan outro tipo de probas como as probas funcionais para complementar as probas estruturais.

As probas funcionais ou de caixa negra céntranse no estudo da especificación do software, da análise das funcións que debe realizar, das entradas e das saídas. As probas funcionais exhaustivas tamén adoitan ser impracticables polo que existen distintas técnicas de deseño de casos de caixa negra.

Clases de equivalencia

Cada caso de equivalencia debe cubrir o máximo número de entradas. Debe tratarse o dominio de valores de entrada dividido nun número finito de clases de equivalencia que cumpran que a proba dun

valor representativo dunha clase permite supor “razoablemente” que o resultado obtido (se existen defectos ou non) será o mesmo que o obtido probando calquera outro valor da clase. O método para deseñar os casos consiste en identificar as clases de equivalencia e crear os casos de proba correspondentes.

Imos ver algunhas regras que nos axudan a identificar as clases de equivalencia tendo en conta as restricións dos datos que poden entrar ao programa:

- De especificar un rango de valores para os datos de entrada, como por exemplo, "o número estará comprendido entre 1 e 49", crearase unha clase válida: $1 \leq \text{número} \leq 49$ e dúas clases non válidas: $\text{número} < 1$ e $\text{número} > 49$.
- De especificar un número de valores para os datos de entrada, como por exemplo, "código de 2 a 4 caracteres", crearase unha clase válida: $2 \leq \text{número de caracteres do código} \leq 4$, e dúas clases non válidas: menos de 2 caracteres e máis de 4 caracteres.
- Nunha situación do tipo "debe ser" ou booleana como por exemplo, "o primeiro carácter debe ser unha letra", identificarase unha clase válida: é unha letra e outra non válida: non é unha letra.
- De especificar un conxunto de valores admitidos que o programa trata de forma diferente, crearase unha clase para cada valor válido e outra non válida. Por exemplo, se temos tres tipos de inmobles: pisos, chalés e locais comerciais, faremos unha clase de equivalencia por cada valor e unha non válida que representa calquera outro caso como por exemplo praza de garaxe.
- En calquera caso, de sospeitar que certos elementos dunha clase non se tratan igual que o resto da mesma, deben dividirse en clases de equivalencia menores.

Para crear os casos de proba séguense os pasos seguintes:

- Asignar un valor único a cada clase de equivalencia.
- Escribir casos de proba que cubran todas as clases de equivalencia válidas non incorporadas nos anteriores casos de proba.
- Escribir un caso de proba para cada clase non válida ata que estean cubertas todas as clases non válidas. Isto faise así xa que se introducimos varias clases de equivalencia non válidas xuntas, poida que a detección dun dos erros, faga que xa non se comprobe o resto.

Exemplo: Unha aplicación bancaria na que o operador proporciona un código de área (número de 3 díxitos que non empeza nin por 0 nin por 1), un nome para identificar a operación (6 caracteres) e unha orde que disparará unha serie de funcións bancarias ("cheque", "depósito", "pago factura" ou "retirada de fondos"). Todas as clases numeradas son:

Entrada	Clases válidas	Clases inválidas
Código área	(1) $200 \leq \text{código} \leq 999$	(2) $\text{código} < 200$ (3) $\text{código} > 999$
Nome para identificar operación	(4) 6 caracteres	(5) menos de 6 caracteres (6) máis de 6 caracteres
Orde	(7) "cheque" (8) "depósito" (9) "pago factura" (10) "retirada fondos"	(11) "divisas"

Os casos de proba, supoñendo que a orde de entrada dos datos é: código-nome-orde son os seguintes:

- Casos válidos:

Código	Nome	orde	Clases
200	Nómina	cheque	(1) (4) (7)
200	Nómina	depósito	(1) (4) (8)
200	Nómina	pago factura	(1) (4) (9)
200	Nómina	retirada fondos	(1) (4) (10)

- Casos no válidos:

Código	Nome	orde	Clases
180	Nómina	cheque	(2)
1032	Nómina	cheque	(3)
200	Nómin	cheque	(5)
200	Nóminas	cheque	(6)
200	Nómina	divisas	(11)

Análise de valores límite (AVL)

A experiencia constata que os casos de proba que exploran as condicións límite dun programa producen un mellor resultado para a detección de defectos. Podemos definir as condicións límite para as entradas, como as situacións que se encontran directamente arriba, abaixo e nas marxes das clases de equivalencia e dentro do rango de valores permitidos para o tipo desas entradas. Podemos definir as condicións límite para as saídas, como as situacións que provocan valores límite nas posibles saídas. É recomendable utilizar o enxeño para considerar todos os aspectos e matices, ás veces sutís, para a aplicación do AVL. Algunhas regras para xerar os casos de proba:

- Se para unha entrada especificase un rango de valores, débense xerar casos válidos para os extremos do rango e casos non válidos para situacións xusto máis aló dos extremos. Por exemplo a clase de equivalencia: " $-1.0 \leq \text{valor} \leq 1.0$ ", casos válidos: -1.0 e 1.0, casos non válidos: -1.01 e 1.01, no caso no que se admitan 2 decimais.
- Se para unha entrada especificase un número de valores, hai que escribir casos para os números máximo, mínimo, un máis do máximo e un menos do mínimo. Por exemplo: "o ficheiro de entrada terá de 1 a 250 rexistros", casos válidos: 1 e 250 rexistros, casos non válidos: 251 e 0 rexistros.
- De especificar rango de valores para a saída, tentarán escribirse casos para tratar os límites na saída. Por exemplo: "o programa pode mostrar de 1 a 4 listaxes", casos válidos: 1 e 4 listaxes, casos non válidos: 0 e 5 listaxes.
- De especificar número de valores para a saída, hai que tentar escribir casos para tratar os límites na saída. Por exemplo: "desconto máximo será o 50%, o mínimo será o 6%", casos válidos: descontos do 50% e o 6%, casos non válidos: descontos do 5.99%, e 50.01% se o desconto é un número real.
- Se a entrada ou saída é un conxunto ordenado (por exemplo, unha táboa, un arquivo secuencial, ...), os casos deben concentrarse no primeiro e no último elemento.

Conxectura de erros:

A idea básica desta técnica consiste en enumerar unha lista de erros posibles que poden cometer os programadores ou de situacións propensas a certos erros e xerar casos de proba en base a dita lista. Esta técnica tamén se denominou xeración de casos (ou valores) especiais, xa que non se obteñen en

base a outros métodos senón mediante a intuición ou a experiencia. Algúns valores a ter en conta para os casos especiais poderían ser os seguintes:

- O valor 0 é propenso a xerar erros tanto na saída como na entrada.
- En situacións nas que se introduce un número variable de valores, como por exemplo unha lista, convén centrarse no caso de non introducir ningún valor e un só valor. Tamén pode ser interesante que todos os valores sexan iguais.
- É recomendable supor que o programador puidese interpretar mal algo nas especificacións.
- Tamén interesa imaxinar as accións que o usuario realiza ao introducir unha entrada, mesmo coma se quixese sabotar o programa. Poderíase comprobar como se comporta o programa se os valores de entrada están fóra dos rangos de valores límites permitidos para ese tipo de variable. Por exemplo, se unha variable de entrada é de tipo *int*, deberíase comprobar o que ocorre se o valor de entrada está fóra do rango de valores permitido para un *int* ou mesmo se ten decimais ou é unha letra. Poderíase comprobar que na entrada non vai camuflado código perigoso. Por exemplo, comprobar a posible inxección de código nunha entrada a unha base de datos.
- Completar as probas de caixa branca e de caixa negra para o caso de bucles. Procurar que un bucle se execute 0 veces, 1 vez ou máis veces. De coñecer o número máximo de iteracións do bucle (*n*), habería que executar o bucle 0, 1, *n*-1 e *n* veces. Se hai bucles anidados, os casos de proba aumentarían de forma exponencial, polo que se recomenda comezar probando o bucle máis interior mantendo os exteriores coas iteracións mínimas e ir creando casos de proba cara o exterior do anidamento.



Tarefa 3.13. Definir clases de equivalencia, realizar análise dos valores límite e conxectura de erros. A tarefa consiste en definir as clases de equivalencia, realizar a análise dos valores límite e realizar conxectura de erros para:

Parte 1) Método *calcularDivision* do proxecto *proyecto_division*.

Parte 2) Método *factorial* do proxecto *proyecto_factorial*.

Parte 3) Método *busca* do proxecto *proyecto_arrays*.

Parte 4) Método *obtenerAcronimo* do proxecto *proyecto_acronimos*.

3.4 Probas unitarias aleatorias

Nas probas aleatorias simúlase a entrada habitual do programa creando datos para introducir nel que sigan a secuencia e frecuencia coas que poderían aparecer na práctica diaria, de maneira repetitiva (próbanse moitas combinacións). Para iso utilízanse habitualmente ferramentas denominadas xeradores automáticos de casos de proba.

3.5 Enfoque recomendado para o deseño de casos

As distintas técnicas vistas para elaborar casos de proba representan aproximacións diferentes. O enfoque recomendado consiste na utilización conxunta de ditas técnicas para lograr un nivel de probas “razoable”. Por exemplo:

- Elaborar casos de proba de caixa negra para as entradas e saídas utilizando clases de equivalencia, completar co análise do valor límite e coa conxectura de erros para engadir novos casos non contemplados nas técnicas anteriores.

- Elaborar casos de proba de caixa branca baseándose nos camiños do código para completar os casos de proba de caixa negra.



Tarefa 3.14 Elaborar casos de proba. A tarefa consiste en elaborar casos de proba para:

Parte 1) Método *calcularDivision* do proxecto *proyecto_division*.

Parte 2) Método *factorial* do proxecto *proyecto_factorial*.

Parte 3) Método *busca* do proxecto *proyecto_arrays*.

Parte 4) Método *obtenerAcronimo* do proxecto *proyecto_acronimos*.

3.6 Junit

JUnit é un framework ou conxunto de clases Java que permiten ao programador construír e executar automaticamente casos de proba para métodos dunha clase Java. Os casos de proba quedan reflectidos en programas Java que quedan arquivados e poden volver a executarse tantas veces como sexa necesario. Estes casos de proba permiten avaliar se a clase se comporta como se esperaba, é dicir, a partir duns valores de entrada, avalíase se o resultado obtido é o esperado. JUnit foi escrito por Erich Gamma e Kent Beck e é un produto de código aberto distribuído baixo unha licenza Common Public License - v 1.0. A páxina oficial de JUnit é : <https://junit.org/junit5/>

Os IDE como NetBeans e Eclipse contan con complementos para utilizar JUnit, permitindo que o programador se centre na proba e no resultado esperado e deixe ao IDE a creación das clases que permiten a proba.

JUnit5 ten una estrutura distinta a das súas versións anteriores. Agora xa non é unha soa biblioteca, se non que é un conxunto de tres subproxectos: JUnit Platform, JUnit Jupiter y JUnit Vintage.

- JUnit Platform é a base que nos permite o lanzamento dos frameworks de proba na JVM e, entre outras cousas, tamén é o encargado de proporcionarnos a posibilidade de lanzar a plataforma desde liña de comandos e dos plugins para Gradle e Maven.
- JUnit Jupiter é o que máis empregaremos á hora de programar. Permítenos empregar o novo modelo de programación para a escritura dos novos test de JUnit 5.
- JUnit Vintage é o encargado dos test de JUnit3 e 4, por compatibilidade.

Neste documento utilizarase JUnit5 para o IDE NetBeans 11 e faranse as probas deste apartado sobre o sinxelo proxecto *proyecto_division* composto das clases *Division.java* e *Main.java* (ver Anexo).

Xerar probas en JUnit

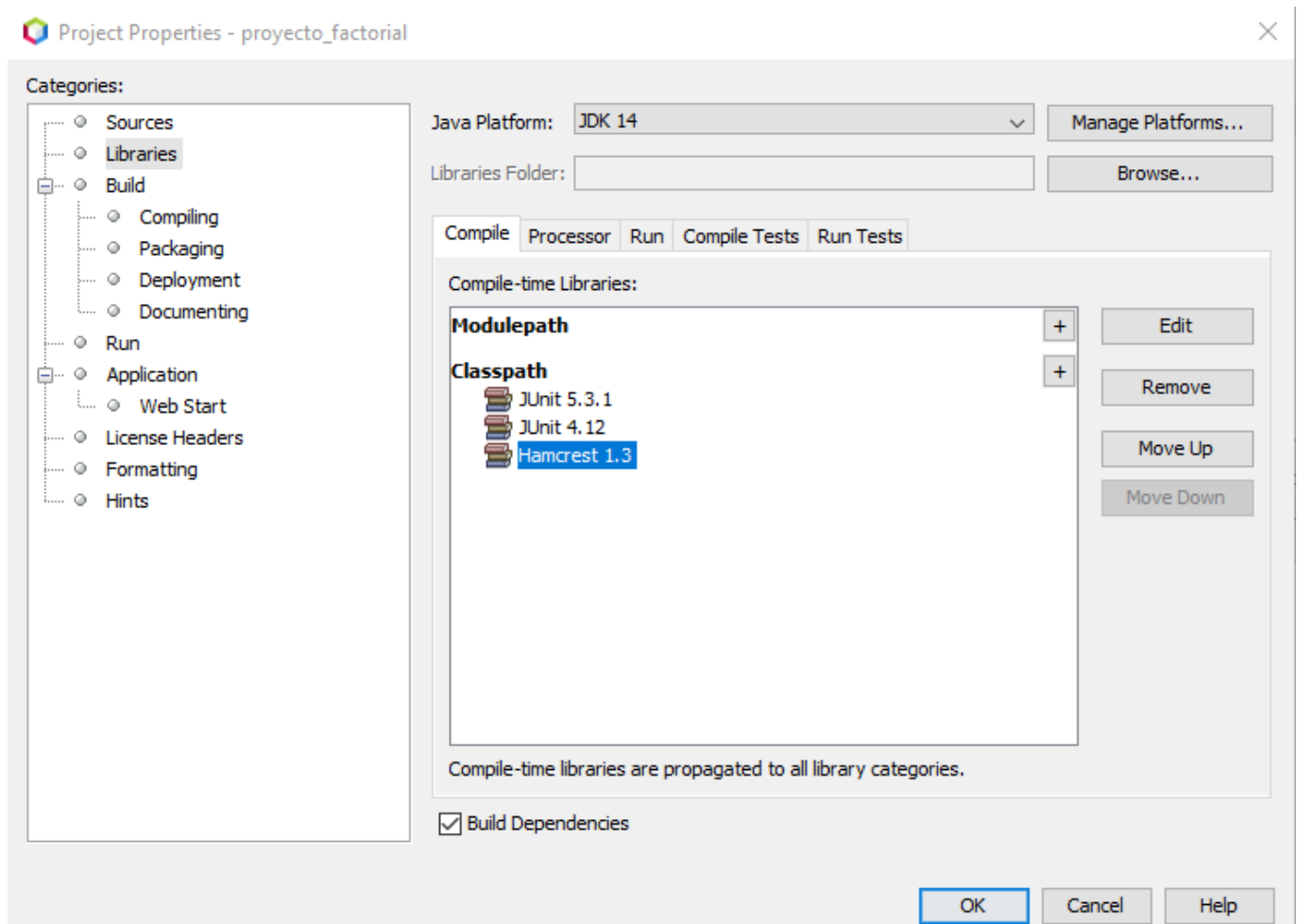
En NetBeans pódense xerar tres tipos de probas: para probar os métodos dunha clase, un conxunto de probas para probar un conxunto de clases dun paquete ou un caso de probas JUnit baleiro.

Para crear unha proba JUnit, pódese seleccionar *Archivo Nuevo* na opción *Archivo* do menú principal, seleccionar a categoría *Unit Tests* e o tipo de arquivo de entre os tres posibles:

- *JUnit Test* que crea un caso de proba baleiro
- *Test for Existing Class* que crea un caso de proba para os métodos dunha clase
- *Test Suite* que crea probas para un paquete Java seleccionado (esta opción xa non está soportada en JUnit5 polo que hai que facelo co modulo *junit5.vintage* e incorporar a dependencia).

Dependendo do tipo de proba seleccionada, haberá que completar de forma diferente as seguintes ventás que aparecen.

Importante Proxectos “Ant”: O primeiro paso en caso de proxectos ‘Ant’ sería ir Propiedades do Proxecto (premendo no botón dereito do rato sobre o nome do proxecto na ventá de proxectos), e na opción lateral “Libraries” engadir as 3 que se ven na seguinte imaxe, **pero nós faremos proxectos Maven.**

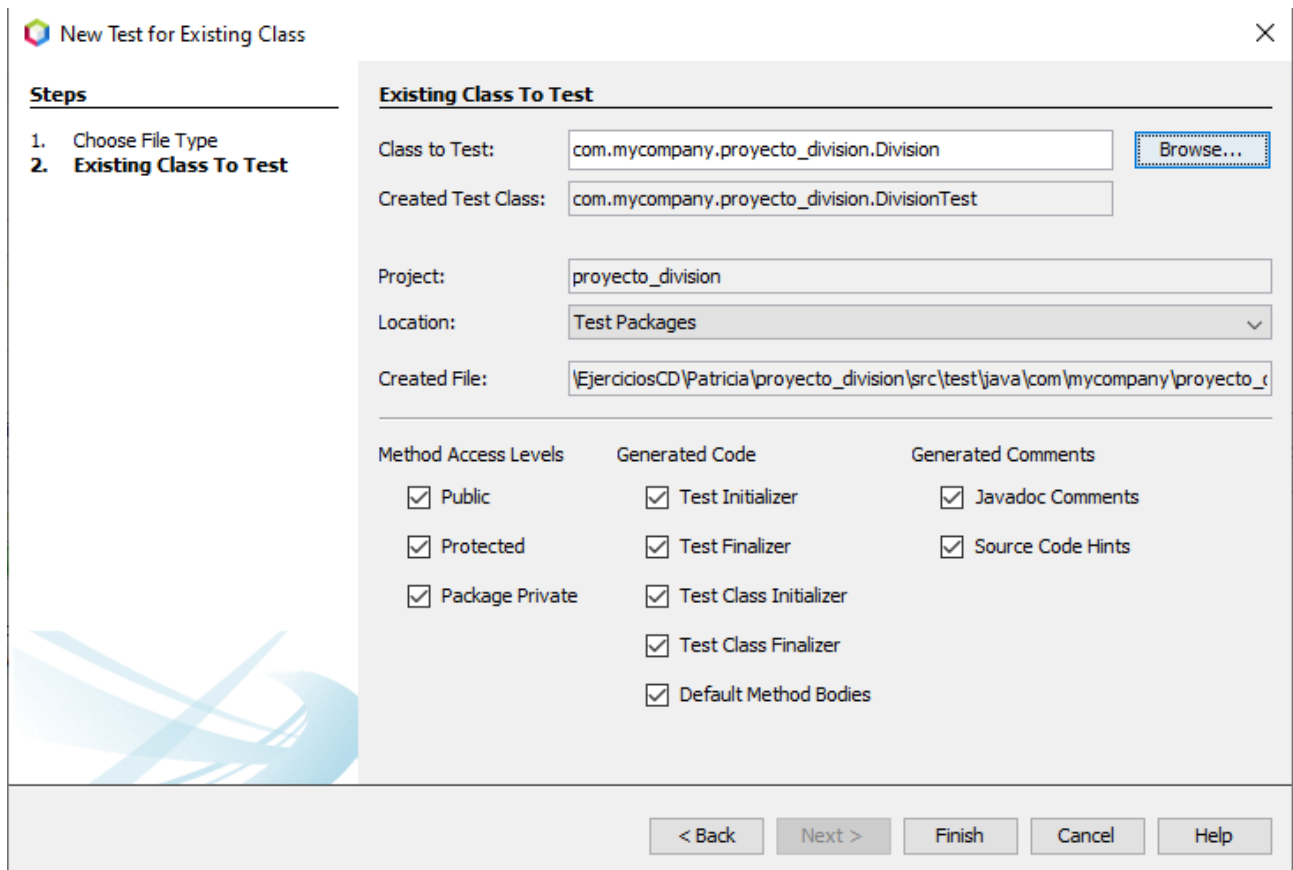


Para unha clase

De elixir no menú principal *File->New File-> Unit Tests->Test for Existing Class*, aparecerá a ventá *Existing Class To Test* na que hai que indicar: o nome da clase de proba (recoméndase deixar o nome por defecto: nome da clase a probar seguido de *Test*), a localización onde se gardará a proba (recoméndase deixar o valor por defecto) e as características da xeración de código dividida en tres apartados:

- Apartado *Method Access Levels* para indicar o nivel de acceso aos métodos de proba.
- Apartado *Generated Code* para indicar se a proba terá métodos para executarse antes de iniciar a proba (*Test Initializer*), despois de finalizar a proba (*Test Finalizer*) ou código exemplo para unha proba (*Default Method Bodies*).
- Apartado *Generated Comments* para indicar que as probas leven comentarios Javadoc e comentarios para suxerir como implementar os métodos de proba (*Source Code Hints*).

Déixase todo seleccionado e prémese en *Finish*.



O código que xera NetBeans por defecto para a proba do método *calcularDivision()* contempla un só caso de proba (0/0=0). Pódese modificar este método e engadir novos casos de proba ou engadir outros métodos de proba. NetBeans suxire borrar as dúas últimas liñas de código da anotación *Test*. Pódense borrar ou comentar as liñas das anotacións agás a anotación *Test* e as liñas *import* correspondentes ás anotacións borradas, se é que non van a ser utilizadas.

```
package proyecto_division;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class DivisionTest {

    public DivisionTest () {

    }

    @BeforeAll
    public static void setUpClass () {

    }

    @AfterAll
    public static void tearDownClass () {

    }

    @BeforeEach
    public void setUp () {

    }
}
```

```

@AfterEach
public void tearDown() {

}

@Test
public void testCalcularDivision() throws Exception {
    System.out.println("calcularDivision");
    float dividendo = 0.0F;
    float divisor = 0.0F;
    Division instance = new Division();
    float expectedResult = 0.0F;
    float result = instance.calcularDivision(dividendo, divisor);
    assertEquals(expResult, result, 0.0);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
}

```

Estrutura da proba

As anotacións e métodos que aparecen por defecto son:

- A anotación *@BeforeAll* marca ao método *setUpClass()* para ser executado antes de empezar a proba de clase, é dicir, execútase unha soa vez e antes de empezar a execución dos métodos de proba. Pódese utilizar por exemplo para crear unha conexión cunha base de datos.
- A anotación *@AfterAll* marca ao método *tearDownClass()* para ser executado ao finalizar a proba de clase. Pódese utilizar por exemplo para pechar a conexión coa base de datos realizada antes.
- A anotación *@BeforeEach* marca ao método *setUp()* para ser executado antes da execución de cada un dos métodos de proba, é dicir, execútase tantas veces como métodos de proba existan. Utilízase para inicializar recursos, variables de clase ou atributos que sexan iguais para tódalas probas.
- A anotación *@AfterEach* marca ao método *tearDown()* para executarse xusto despois da execución de cada un dos métodos de proba.
- A anotación *@Test* marca cada método de proba.
- O método *assertEquals*. Este método afirma que o primeiro argumento (resultado esperado) é igual ao segundo (resultado obtido). Se os dous argumentos son reais, pode ter un terceiro argumento chamado valor delta, que é un número real igual á máxima diferenza en valor absoluto entre o valor esperado e o actual para que a afirmación sexa un éxito.

```
Math.abs(esperado-obtido)<delta
```

Na proba para o método *calcularDivision()* utilizada de exemplo, pódese modificar o test para poder comprobar que a división entre 1 e 3 dá como resultado 0.33 cun valor delta de 1E-2. Se consideramos que o método *calcularDivision(1,3)* devolve 0.333, os valores esperados 0.34 e 0.33 serían equiparables ao valor real e o valor 0.32 non, xa que $\text{abs}(0.333-0.33) < 0.01$, $\text{abs}(0.333-0.34) < 0.01$ e $\text{abs}(0.333-0.32) > 0.01$. O código do test podería ser:

```

@Test
public void testCalcularDivision() throws Exception {

```

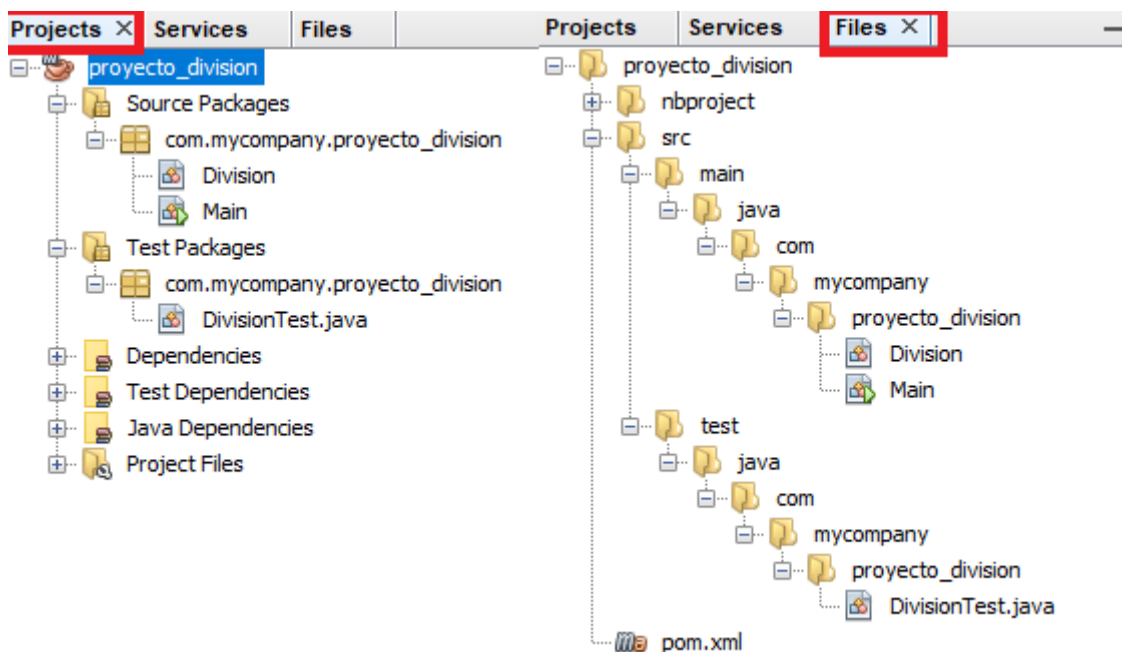
```

        System.out.println("Caso: 1/3=0.33 con valor delta 1E-2");
        Division instance = new Division();
        float resultado = instance.calcularDivision(1F, 3F);
        assertEquals(0.33, resultado, 1E-2);
    }
}

```

Carpeta para as probas

Nas ventás de arquivos e de proxectos poden verse a estrutura lóxica e física das carpetas nas que se gardan as probas JUnit. Pódense agregar outras carpetas de proba no cadro de diálogo de propiedades do proxecto pero tendo en conta que os arquivos de proba e os fontes non poden estar na mesma carpeta.



Executar a proba

Pasos para executar a proba dun proxecto inteiro:

- Selecciónase calquera nodo ou arquivo do proxecto na ventá de proxectos ou na de arquivos e elíxese no menú principal *Ejecutar->Probar Project(nome_do_proxecto)* ou prémese Alt-F6.
- O IDE executa tódolos métodos de proba do proxecto. De querer executar un subconxunto das probas do proxecto ou executar as probas nunha orde específica, debe crearse unha *Test Suite* que especifique as probas a executar.

Para executar a proba dunha clase, pódese elixir unha das dúas posibilidades seguintes:

- Selecciónase a clase na ventá de proxectos ou na de arquivos, clic dereito e elíxese *Probar archivo* ou prémese Alt-F6.
- Selecciónase a proba da clase e elíxese no menú principal *Ejecutar -> Probar archivo* ou prémese Alt-F6.

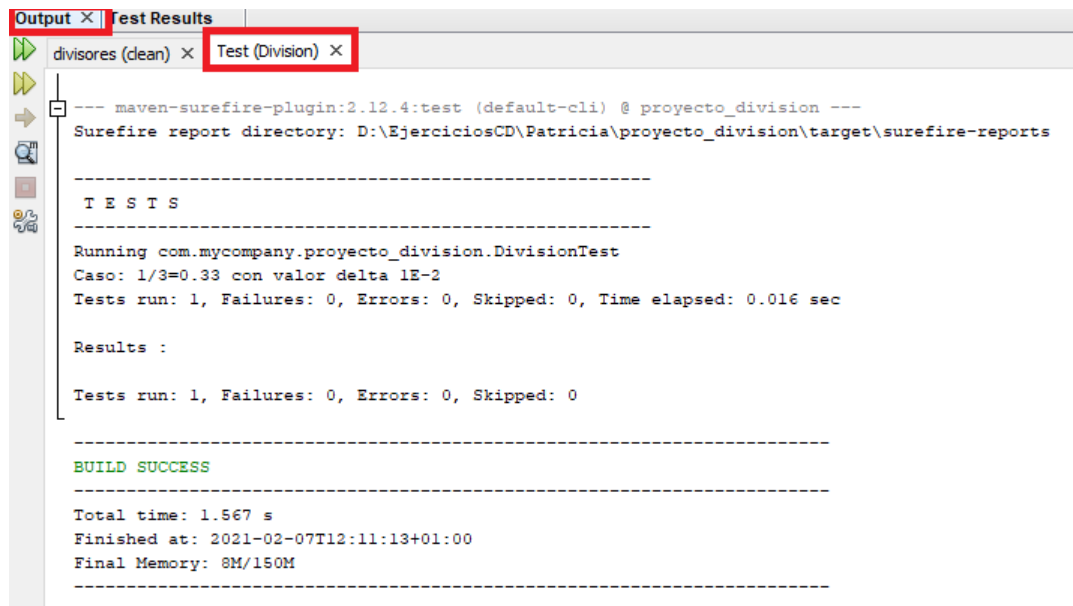
Pasos para executar un caso de proba:

- Execútase a proba que contén ese caso de proba.

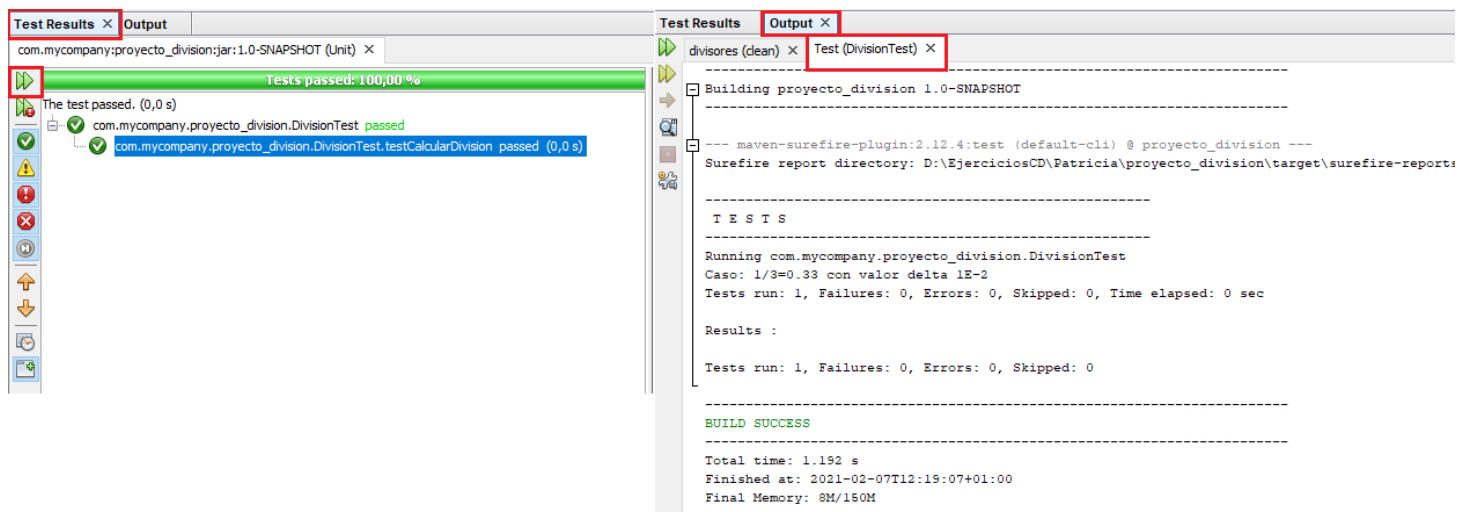
- Na ventá de resultados faise clic dereito sobre o método e elíxese *Run Again*.

Ventás de saída e resultados

A ventá de saída reflicte o detalle do proceso de execución das probas en formato texto.



A ventá de resultados ten dúas zonas. A zona da esquerda contén un resumo dos casos de proba superados e non superados e unha descrición deles en formato gráfico. Ao pasar o rato por riba da descrición dun método de proba, aparece nun recadro a saída correspondente a ese método. A zona da dereita contén a saída textual.



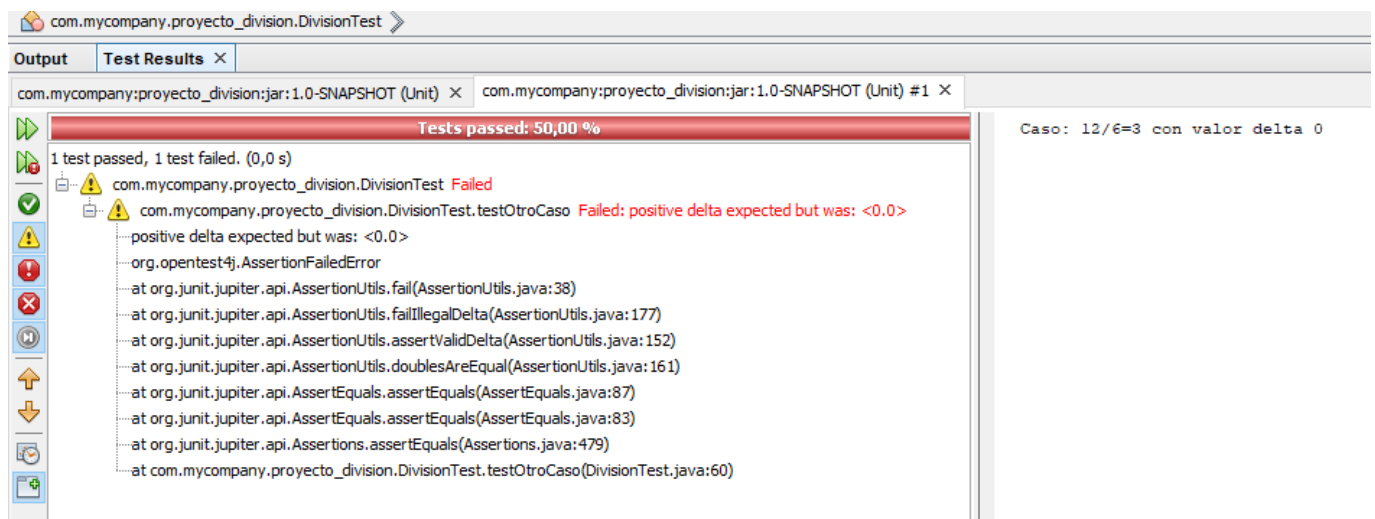
Algunhas das iconas que se poden utilizar na ventá de resultados son:

	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 ventás cambian se non se supera algunha proba. Por exemplo, ao executar un método de proba que espera un resultado 3 para o caso de proba 12/6 como o seguinte:

```
@Test
public void testOtroCaso() throws Exception {
    System.out.println("Caso: 12/6=3 con valor delta 0");
    float dividendo = 12.0F;
    float divisor = 6.0F;
    Division instance = new Division();
    float expResult = 3F;
    float result = instance.calcularDivision(dividendo, divisor);
    assertEquals(expResult, result, 0.00);
}
```

Aparecerá na ventá de resultados o detalle sobre o caso de proba non superado.



Asertos

A clase *Assert* permite comprobar se a saída do método que se está probando concorda cos valores esperados. Pódese ver a sintaxe completa dos métodos asertos (afirmacións) que se poden usar para as probas en:

<https://junit.org/junit5/docs/5.3.0/api/org/junit/jupiter/api/Assertions.html>

Unha breve descrición deles é:

- *void assertEquals(valor esperado, valor actual)* é un método con sobrecarga para cada tipo en java e que permite comprobar se a chamada a un método devolve un valor esperado. No caso de valores reais ten un terceiro argumento para indicar o valor delta ou número real igual á máxima diferenza en valor absoluto entre o valor esperado e o actual para que a afirmación sexa un éxito.
- *void fail()* para cando se espera que o programa falle. Utilízase cando a proba indica que

hai un erro ou cando se espera que o método que se está probando chame a unha excepción.

- `void assertTrue(boolean)` a proba é un éxito se a expresión booleana é certa.
- `void assertFalse(boolean)` a proba é un éxito se a expresión booleana é falsa.
- `void assertNull(Object)` a proba é un éxito se o obxecto é nulo.
- `void assertNotNull(Object)` a proba é un éxito se o obxecto non é nulo.
- `void assertEquals(Object, Object)` a proba é un éxito se os dous obxectos son o mesmo.
- `void assertNotEquals(Object, Object)` a proba é un éxito se os dous obxectos non son mesmo.

Anotacións

As anotacións aportan información sobre un programa e poden ser usadas polo compilador (por exemplo: `@Override` para informarlle que se está sobrescribindo un método, `@Deprecated` para indicarlle que está en desuso, `@SuppressWarnings` para indicarlle que non avise de *warnings* ou advertencias), por ferramentas de software que as poden procesar (por exemplo para xerar código ou arquivos xml) ou para ser procesadas en tempo de execución. As anotacións poden aplicarse a declaracións de clases, campos, métodos e outros elementos dun programa.

As anotacións máis importantes nunha proba son `@Ignore` que serve para desactivar un test e colócase xusto antes de `@Test`, e `@Test`.

A anotación `@Test` serve para anotar unha proba. Pode ter dous parámetros opcionais *expected* e *timeout*. O primeiro define a clase de excepción que se espera que lance a proba para que sexa superada e o segundo define os milisegundos que como máximo debe durar a execución para que a proba sexa superada.

Exemplo con *timeout*: o seguinte test non se superará xa que a execución da proba supera os 100 milisegundos:

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

Se queremos comprobar que se produce unha excepción, o `assertEquals` non serve. Temos que usar outra estrutura: `assertThrows`. (ollo: proxecto Maven, con Ant produce erro de compilación)

```
@Test
public void testDivException() {
    Division instance = new Division();
    assertThrows(Exception.class, new Executable() {
        @Override
        public void execute() throws Throwable {
            instance.calcularDivision(5,0);
        }
    });
}
```

Olo: para que funcione a sentencia anterior temos que facer o seguinte import:

```
import org.junit.jupiter.api.function.Executable;
```

A recomendación oficial é empregar funcións Lambda pero nós aínda non as coñecemos. A sintaxe sería así:

```
@Test
public void testDivException() {
    Division instance = new Division();
    assertThrows(Exception.class, () -> instance.calcularDivision(5,0));
}
```

}



Tarefa 3.15. Xerar e executar probas en JUnit e documentar incidencias

A tarefa consiste en xerar probas en JUnit, executalas e documentar incidencias, para os métodos:

Parte 1) O método *calcularDivision* do proxecto *proyecto_division*.

Parte 2) O método *factorial* do proxecto *proyecto_factorial*.

Parte 3) O método *busca* do proxecto *proyecto_arrays*.

Parte 4) O método *obtenerAcronimo* do proxecto *proyecto_acronimo*.

Recoméndase utilizar un método de proba para cada caso de proba nas tres primeiras tarefas xa que é máis fácil ver o resultado de cada un deles. Na última tarefa deberanse de agrupar varios casos de proba nun mesmo método de proba.

3.7 Proba do camiño básico

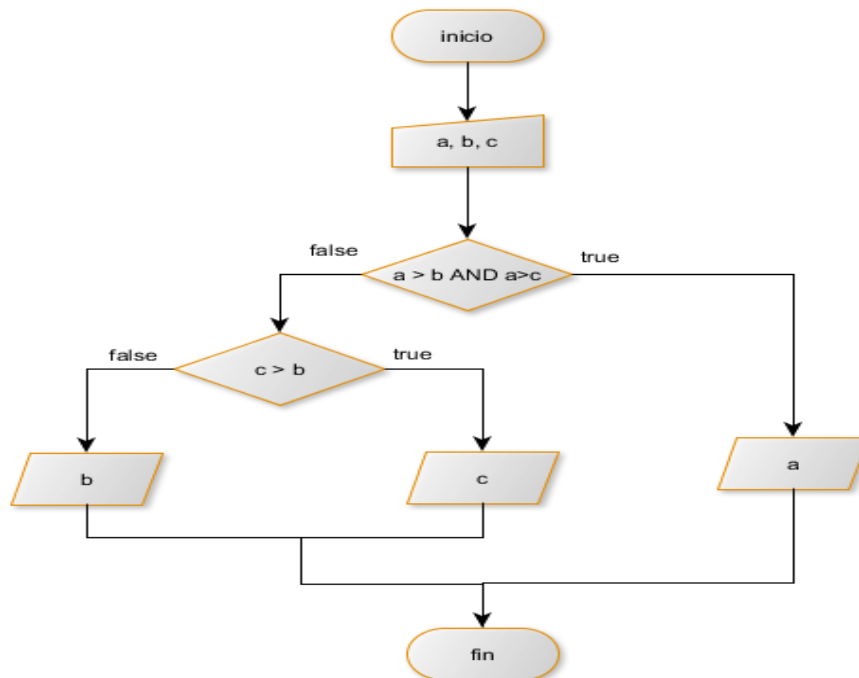
A proba do camiño básico é unha proba de «caixa branca» que consiste en verificar o código de nosos sistemas de xeito que comprobemos que todo funciona correctamente, é dicir, débese verificar que todas as instrucións do programa se executan polo menos unha vez.

Os pasos para desenvolver a proba do camiño básico son:

- 1.- Debuxar o grafo de fluxo
- 2.- Calcular a complexidade ciclomática
- 3.- Determinar o conxunto básico de camiños independentes
- 4.- Implementar as probas en JUnit

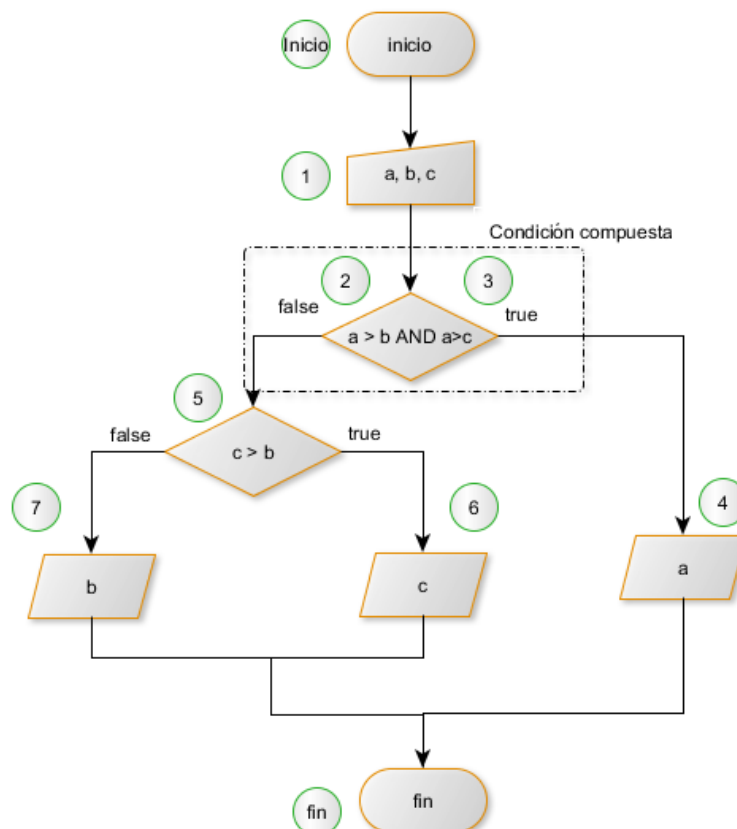
Vexamos un exemplo:

O seguinte diagrama de fluxo corresponde ao algoritmo para determinar o número maior de 3 valores dados.

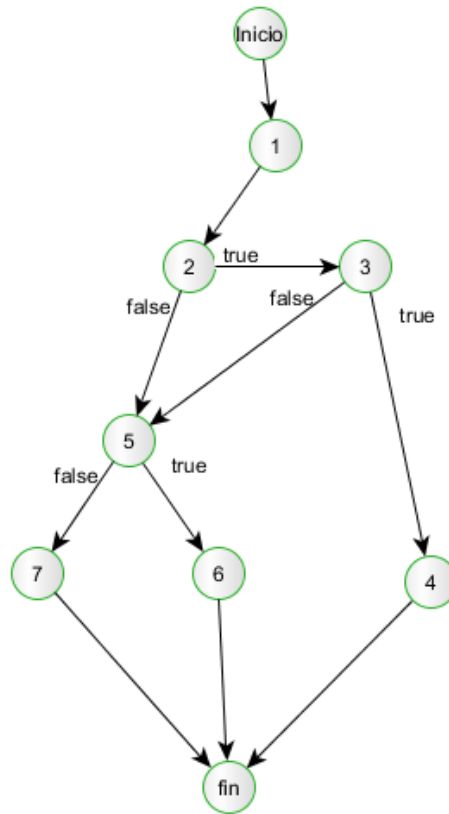


Paso 1: Debuxar o grafo de fluxo

Detectamos os nodos que conformaran o grafo de fluxo así como os camiños que se poden percorrer durante a execución do programa. Se temos unha condición composta, como é o noso caso ($a > b$ AND $a > c$), debemos descompoñela creando un nodo para cada unha das condicións.



A continuación, debuxamos o grafo de fluxo:



Paso 2: Complexidade Ciclomática

A complexidade ciclomática mide o número de camiños independentes:

$V(G) = \text{arestas} - \text{nodos} + 2$, no noso caso: $11 - 9 + 2 = 4$

Debemos entón facer 4 probas para asegurarnos de que cada instrución se executa a lo menos unha vez.

Paso 3: Camiños independentes

Imos formando os camiños independentes desde o máis longo ao máis curto, vento o noso grafo de fluxo.

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

Paso 4: Probas en JUnit

Implementamos eses casos de proba nunha clase JUnit en NetBeans.



Tarefa 3.16. Realiza o proceso completo das probas do camiño básico incluíndo as

probas unitarias con JUnit para o seguinte código:

```
static int  contarLetras (String cadena, char letra) {  
    int contPos=0, conVeces= 0, longCadena = 0;  
    longCadena = cadena.length();  
    if (longCadena > 0) {  
        do {  
            if (cadena.charAt(contPos)== letra) conVeces++;  
            contPos++;  
        } while (contPos < longCadena);  
    }  
    return conVeces;  
}
```