


# UD3. Deseño e realización de Probas

DAM1-Contornos de Desenvolvemento 2024-25

<b>1. Introdución</b>	<b>2</b>
<b>2. Estratexia de probas do software</b>	<b>2</b>
2.1. Probas de unidade	2
2.2. Probas de integración	2
2.3. Probas do sistema	3
2.4. Proba de validación	4
<b>3. Técnicas de deseño de casos de proba</b>	<b>5</b>
3.1. Proba do camiño básico	5
3.1.1. Grafo de fluxo	6
3.1.2. Complexidade ciclomática	9
3.1.3. Camiños independentes	10
3.1.4. Obtención dos casos de proba	10
3.2. Particións ou clases de equivalencia	14
3.3. Análise de valores límite	18
<b>4. Documentación para as probas</b>	<b>20</b>
<b>5. Ferramentas de depuración</b>	<b>21</b>
5.1. Execución paso a paso	22
5.2. Puntos de ruptura (breakpoints)	22
5.3. Examinar variables	23
<b>6. Probas Unitarias</b>	<b>25</b>
<b>Anexos e Ligazóns</b>	<b>25</b>

-  [Diseño y realización de pruebas](#)

# 1. Introducción

As probas constitúen unha das tarefas do ciclo de vida do software e o seu obxectivo fundamental é a detección de erros que se cometeron involuntariamente durante o proceso de construción do software antes de que este se entregue ao cliente.

O obxectivo desta tarefa é a detección de defectos, non demostrar que o software non ten ningún. É imposible probar exhaustivamente o software, o que se pretende é realizar probas que non supoñan un esforzo excesivo per que teñan unha alta probabilidade de que detecten erros. O obxectivo último é detectar estes erros antes de entregar o software ao cliente.

## 2. Estratexia de probas do software

As probas sempre comezan polos compoñentes máis pequenos, é dicir, empézase probando pequenas porcións do software (métodos, clases, paquetes, etc.) e vaise incrementando de xeito progresivo o alcance da proba. Sóense levar a cabo catro tipos de probas secuenciais que veremos a continuación.

### 2.1. Probas de unidade

Comezase probando **módulos individuais ou clases**, dependendo do tipo de desenvolvemento realizado. No caso das clases, tamén haberá que realizar probas a nivel de método, isto é, para cada método non trivial haberá que comprobar se o seu comportamento é o adecuado.

Búscase eliminar erros na interface e na lóxica interna. Nesta actividade utilízanse **técnicas de caixa negra e caixa branca**.

Fanse probas sobre:

1. A **interfaz** do módulo, para asegurar que a información flúe adecuadamente.
2. As **estruturas de datos locais**, para asegurar que manteñen a integridade durante todos os pasos do proceso.
3. As **condicións límite**, para asegurar que funcionan correctamente nos límites establecidos durante o proceso.
4. Todos os **camiños independentes da estrutura de control**, co fin de asegurar que todas as sentenzas se executan polo menos unha vez.
5. Todos os **camiños de manexo de erros**.

Existen ferramentas para automatizar probas unitarias como son: [JUnit](#), CPPUnit, [PHPUnit](#), PyUnit, etc.

### 2.2. Probas de integración

- [Pruebas de Integración: qué son, tipos y ejemplos](#)

As probas de integración teñen como obxectivo verificar que os distintos módulos ou compoñentes dun programa funcionan correctamente cando se integran entre si para resolver as distintas funcionalidades do programa.

Consisten en combinar módulos ou compoñentes individuais previamente probados mediante probas unitarias e avaliar o seu funcionamento conxunto. Estas probas céntranse na comunicación e interacción entre os compoñentes, asegurándose de que non haxa problemas de transferencia de datos, sincronización ou compatibilidade entre eles

Existen distintos enfoques para levar a cabo as probas:

1. **Big bang ou non incremental:**
  - Todos os compoñentes intégranse simultaneamente e próbanse como unha única unidade.
  - Útil para sistemas pequenos, pero pode dificultar identificar a causa dos erros en sistemas complexos.
2. **Integración incremental:** Os módulos intégranse e próbanse en pequenos segmentos, neste caso os erros son máis fáciles de localizar.
  - **Ascendente (Bottom-up).** A construción e proba do programa empeza dende os módulos inferiores e máis sinxelos da estrutura do programa.
  - **Descendente (Top-Down).** A integración comeza no módulo principal movéndose cara abaixo pola xerarquía de control.
3. **Probas baseadas en hebra:** Estas probas están orientadas a sistemas que empregan múltiples hebras ou threads para executar tarefas concurrentes, como sistemas operativos, aplicacións en tempo real, etc.

## 2.3. Probas do sistema

Este tipo de probas consiste no proceso de proba dun sistema integrado de hardware e software para comprobar se cumpre cos requisitos especificados.

Existen distintos tipos de probas do sistema:

- **Proba de recuperación.** Fórzase o fallo do sistema e verifícase que a recuperación se leva a cabo axeitadamente.
- **Proba de seguridade.** Esta proba intenta verificar que o sistema está protexido contra accesos ilegais.
- **Proba de esforzo, resistencia ou stress.** Trata de enfrontar o sistema con situacións que demandan gran cantidade de recursos, por exemplo, deseñando casos de proba que requiran o máximo de memoria, incrementando a frecuencia de datos de entrada que dean problemas e nun sistema operativo virtual, etc.
- **Proba de rendemento:** Proba que o sistema cumpla certos requisitos de rendemento como, por exemplo, un tempo de resposta axeitado ante un evento.
- **Proba de despliegue:** Consiste en probar o funcionamento do sistema nas distintas plataformas nas que se debe utilizar. Por exemplo, unha aplicación web debe probarse en distintos navegadores e sistemas operativos, resolucións de pantalla, etc.

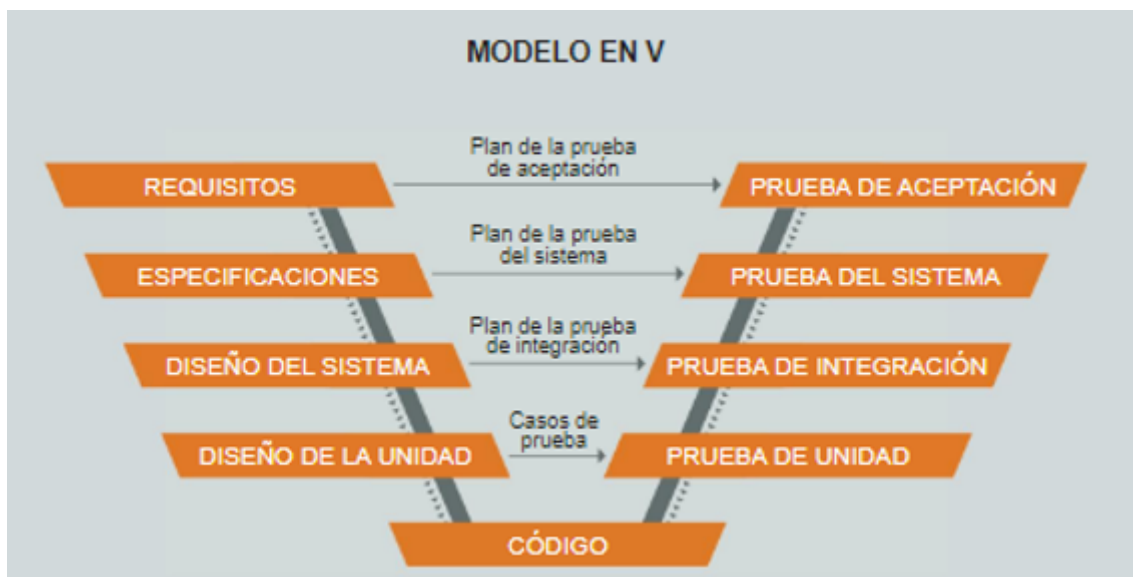
## 2.4. Proba de validación

O obxectivo desta proba, tamén chamada **proba de aceptación**, é determinar se o software é considerado válido por parte da persoa usuaria e se está preparado para a súa implantación.

Lévanse a cabo unha serie de probas de caixa negra que demostran a conformidade cos requisitos. As técnicas que se empregarán son:

- **Proba Alfa.** Lévanse a cabo polo cliente ou usuario no lugar de desenvolvemento. O cliente utiliza o software de forma natural baixo a observación do desarrollador, que irá rexistrando os erros e os problemas de uso.
- **Proba Beta.** Lévanse a cabo polos usuarios finais do software no seu lugar de traballo. O desarrollador non está presente. O usuario rexistra todos os problemas que atope, reais e/ou imaxinarios. Modifícanse e prepárase unha nova versión do produto.

Todas estas probas se amosan no seguinte Modelo en V, que é unha variante do modelo en cascada, onde se relaciona cada fase do ciclo de vida do software coa fase de probas asociada.



### 3. Técnicas de deseño de casos de proba

As técnicas de deseño de casos de proba son as diferentes formas en que se poden xerar casos de proba para probar o software. Dependendo do enfoque que se adopte, pódese aplicar un dos seguintes tipos de técnicas de deseño de casos de proba.

1. **Probas de caixa branca ou probas estruturais:** examinan os detalles de cada módulo, para o que se debe dispor do código fonte. A través de dito código, próbanse os diferentes camiños, os bucles, as variables, etc. Destas probas derivan casos de proba que:
  - Garanten que todas as rutas independentes dentro dun módulo se revisaron polo menos unha vez.
  - Revisan todas as decisións lóxicas nos seus lados verdadeiro e falso.
  - Executan todos os bucles nas súas fronteiras operativas.
  - Revisan estruturas de datos internas para garantir a súa validez.
2. **Probas de caixa negra ou probas funcionais:** considérase ao software como unha caixa negra que recibe unha serie de entradas e proporciona unha serie de saídas. O obxectivo destas probas é validar os requisitos funcionais. Estas probas lévanse a cabo sobre a interface do software, non fai falla coñecer a estrutura interna do programa nin o seu funcionamento. Preténdese obter casos de proba que demostren que as funcións do software son operativas, e dicir, que as saídas que devolve a aplicación son as esperadas en función das entradas que se proporcionan.

É habitual empregar os dous tipos de técnicas de deseño de casos de proba, pois se poden combinar e complementar perfectamente.

#### Probas de código

A proba de código consiste na execución do programa (ou parte del) co obxectivo de atopar erros. Pártese para a súa execución dun conxunto de entradas e unha serie de condicións de execución; obsérvanse e rexístranse os resultados e compáranse cos resultados esperados. Observarase se o comportamento do programa é o previsto ou non e por que.

#### 3.1. Proba do camiño básico

Esta **técnica de caixa branca**, proposta inicialmente por McCabe, trata de obter un valor de complexidade do módulo denominada **complexidade ciclomática**. Este valor permitirá deseñar un conxunto básico de camiños de execución e en base a ese conxunto crearanse os casos de proba que permitan recorrer todos eses camiños.

A proba se realiza en catro fases:

- Obtención do grafo de fluxo do programa.
- Cálculo da complexidade ciclomática.
- Obtención do conxunto básico de camiños independentes.

- Creación de casos de proba.

### 3.1.1. Grafo de fluxo

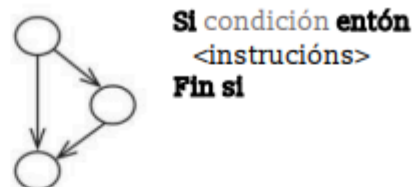
- [▶ Cómo crear el grafo de flujo de un programa](#)
- [▶ Grafo de flujo de un programa. Ejemplo complejo.](#)
- [▶ Complejidad ciclomática: cómo calcularla](#)
- [▶ Pruebas de cobertura. Ejemplo práctico](#)
- [▶ Prueba del camino básico. Pruebas de caja blanca](#)
- [▶ Tests Funcionales. Clases de equivalencia y valores límite.](#)
- [▶ MVC: Modelo Vista Controlador](#)

O grafo de fluxo mostra o recorrido do control lóxico do programa. Para representalo úsase a seguinte notación:

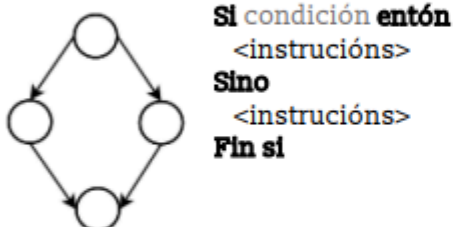
#### Secuencial



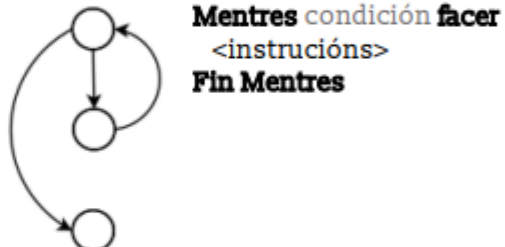
#### Condicional



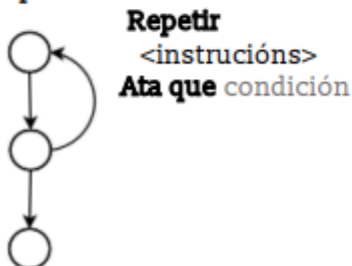
#### Condicional



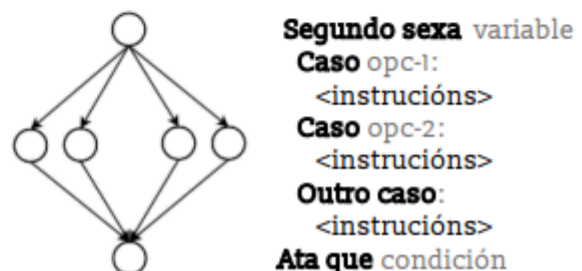
#### Facer Mentres



#### Repetir ata



#### Condicional múltiple



Para crear un diagrama de fluxo hai que ter en conta que cada instrución conforma un nodo, cada asignación de valores é un nodo e mesmo os parámetros dun método son un nodo.

A maiores, sempre hai un nodo inicio e un nodo fin.

Cada paso dunha instrución a outra é unha arista e aquelas instrucións iterativas e alternativas terán distintos camiños dependendo dos valores da condición.

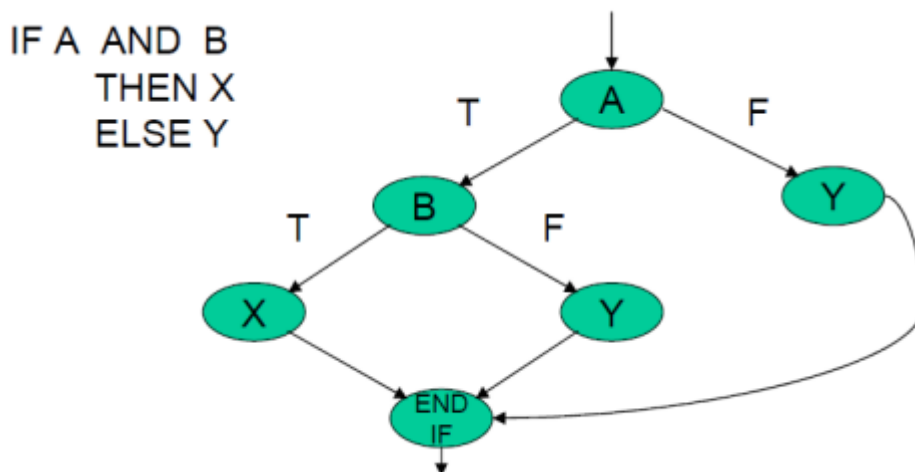
Se partimos dende un organigrama, numéranse cada un dos símbolos e os finais das estruturas de control. Cada nodo representa unha ou máis sentenzas procedimentais. Un nodo pódese corresponder cunha sentenza de símbolos do proceso e un rombo de decisión.

### Condicións

As condicións, xa sexa nunha instrución alternativa ou unha iterativa, conforman un nodo cunha arista por cada opción posible. En caso de que unha condición estea formada por varias operacións lóxicas unidas mediante AND ou OR, haberá que tratalas como condicións independentes aniñadas segundo o operador que as une.

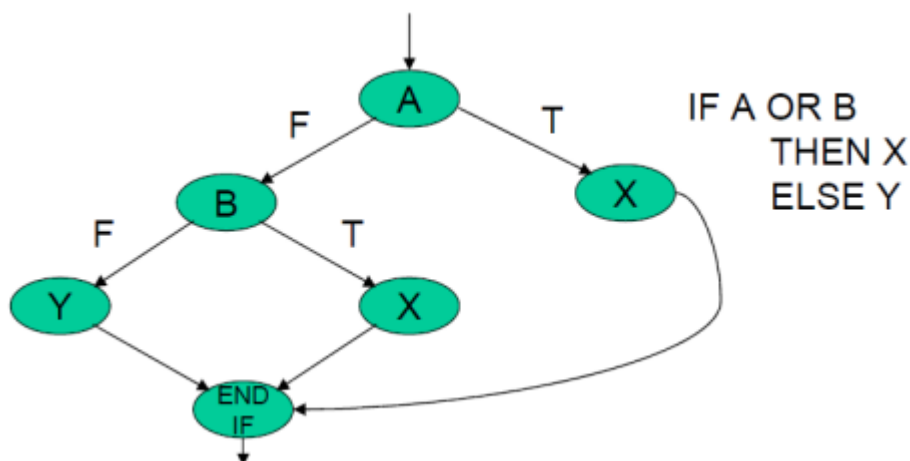
#### Condición AND

Nunha condición AND cada un dos operandos da condición é un nodo. As condicións valoranse en cascada cando dean *true*. Se algún de eles é *false*, saltarase ao caso contrario directamente. Só se todos son *true* entrará.



#### Condición OR

Nunha condición OR, cada un dos operandos tamén é un nodo. Valóranse en cascada cando algún sexa *false*. Se algún é *true*, entra no seu interior, se todos son *false*, saltará ao caso contrario.



### Instrucións alternativas

As instrucións alternativas terán dúas ramas se son un *if-else* ou múltiples se son un *switch*. Cos *switch*, crearase unha rama independente por cada caso posible tendo en conta que os que estean combinados tomaranse como un só e os que non teñan un *break* enlazaranse. En caso dos *if*, seguirá as mesmas regras que nas condicións comentadas anteriormente.

## **Bucles**

Nos bucles, o fluxo entra no corpo e volve atrás en función da condición do bucle.

### **While**

No *while*, a condición é un nodo e o interior outro que estarán conectados de forma cíclica do segundo ao primeiro e é o primeiro o que saltaría ao seguinte, tras o bucle, no caso de que a condición non se cumpra.

### **Do-while**

O bucle *do-while* é similar ao *while*, coa salvedade de que primeiro entra no corpo, e logo valora a condición.

### **For**

No bucle *for*, primeiro vai o nodo da asignación do valor á variable de control *i*, logo o nodo/s da condición, logo o/os nodo/s do corpo e por último o nodo do incremento do valor que volverá ao da condición. Se a condición non se cumpre, irá ó seguinte nodo tralo bucle.

## **Parámetros de entrada**

Os propios parámetros de entrada dun método forman un nodo conxunto, xa que a obtención destes é unha instrución.

## **Asignacións e declaracións**

As asignacións de valores conforman un nodo e agrúpanse se están de forma conxunta. As declaracións óbvianse ou se estipulan no mesmo nodo que a primeira asignación, formando parte do nodo.

## **Entradas e saídas**

As entradas e saídas conforman un nodo, podendo agruparse se son consecutivos.

## **Return**

Cada *return* é un nodo en si mesmo, independentemente de se devolve directamente un valor ou unha variable.

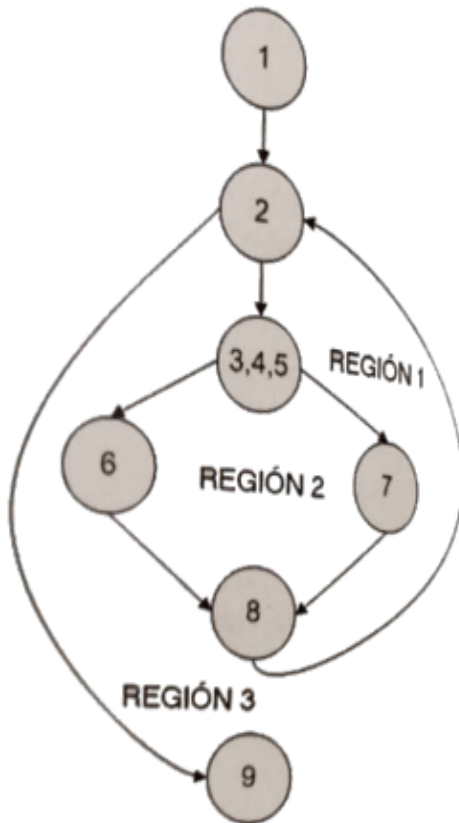


### 3.1.2. Complexidade ciclomática

A [complexidade ciclomática](#),  $V(G)$ , pódese calcular de 3 formas distintas:

1.  $V(G) = \text{Número de rexións do grafo}$
2.  $V(G) = \text{aristas} - \text{nodos} + 2$
3.  $V(G) = \text{Nodos predicado} + 1$

1. Unha vez debuxado o diagrama, contar o número de áreas creadas (zonas rodeadas de aristas) e sumarlle unha (exterior), deixando claro a irrelevancia dos elementos lineais.



Considérase unha **rexión** como unha zona do grafo pechada. Tamén se considera rexión a parte externa.

$$V(G) = \text{número de rexións} = 3$$

2. Outra forma de facer o cálculo, e contar o número de aristas, restarlle o número de nodos e sumarlle 2. Hai que ter en conta que, todos os elementos lineais, é dicir, aqueles que non crean ramas, achegan 1 nodo e 1 arista, polo que dá igual o número deles que haxa, xa que, segundo a fórmula da complexidade, a cada arista se lle resta un nodo. No exemplo hai 8 aristas e 7 nodos.

$$V(G) = (8 - 7) + 2 = 3$$

3. A complexidade pódese calcular contando os nodos predicado, aqueles dos que sae máis dunha arista, e sumarlle 1. Neste caso hai 2 nodos dos que saen máis de dúas aristas, nodo (2) e nodo (3,4,5).

$$V(G) = 2 + 1 = 3$$

O valor da complexidade ciclomática permite avaliar o risco do código analizado.

Complexidade	Avaliación do risco
1-10	Código ben estruturado. Capacidade de testeo alta. Custo e esforzo baixo
11-20	Código complexo. Capacidade de testeo media. Custo e esforzo medio
21-50	Código moi complexo. Capacidade de testeo baixa
>50	Non testeable. Custo e esforzo alto

Risco segundo a complexidade ciclomática

### 3.1.3. Camiños independentes

O valor de complexidade ciclomática  $V(G)$ , da o número de camiños independentes do conxunto básico dun programa. Un camiño independente é calquera camiño do programa que introduce, polo menos, un novo conxunto de sentenzas de proceso ou unha condición.

En termos do diagrama de fluxo, **un camiño independente está constituído polo menos por unha arista que non teña sido recorrida anteriormente** por outro camiño.

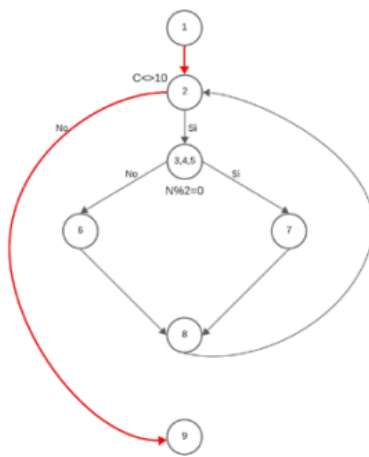
Neste caso, un grupo de camiños independentes sería:

Camiño 1: 1-2-9

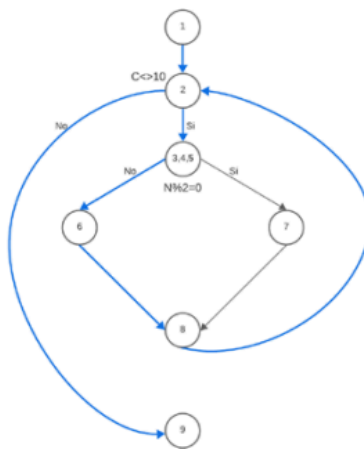
Camiño 2: 1-2-3,4,5-6-8-2-9

Camiño 3: 1-2-3,4,5-7-8-2-9

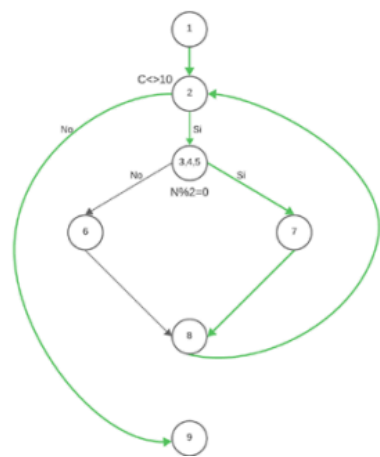
Camiño 1:



Camiño 2:



Camiño 3:



### 3.1.4. Obtención dos casos de proba

O último paso da proba do camiño básico é construír os casos de proba forzan a execución de cada camiño. Co fin de comprobar cada camiño, débense escoller os casos de proba de xeito que as condicións dos nodos predicado estean adecuadamente establecidas.

Seguindo co exemplo anterior:

Camiño	Caso de proba	Resultado esperado
1	Escoller algún valor de C tal que NON cumpra a condición $C <> 10$ . $C = 10$	Visualizar o número de impares e de pares.
2	Escoller algún valor de C que SI cumpra a condición $C <> 10$ . Escoller algún valor de N tal que NON cumpra a condición $N \% 2 = 0$ . $C = 1, N = 5$	Contar números impares

3	<p>Escoher algún valor de C que SI cumpra a condición <math>C \neq 10</math>. Escoher algún valor de N tal que SI cumpra a condición <math>N \% 2 = 0</math></p> <p style="text-align: center;"><math>C=1, N=4</math></p>	Contar números pares
---	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------

**Exemplo.** Dado o seguinte algoritmo:

```

int contar_letras(char cadena[10], char letra)
{
    int contador=0, n=0, lon;
    lon = strlen(cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra) n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}

```

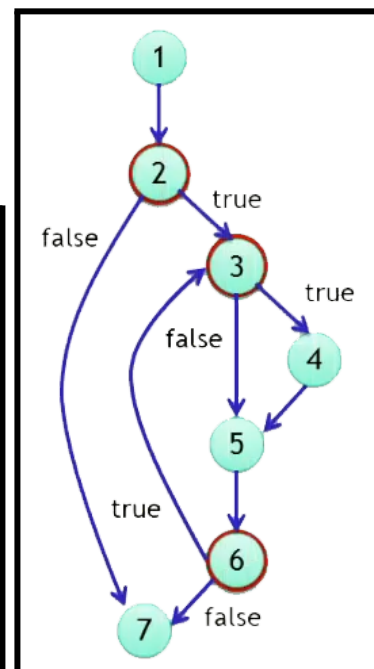
1. Realiza o grafo de fluxo
2. Calcula a complexidade ciclomática
3. Calcula os camiños independentes
4. Elabora os casos de proba

#### 1. Grafo de Fluxo

```

int contar_letras(char cadena[10], char letra)
{
    int contador=0, n=0, lon; } 1
    lon = strlen(cadena);
    if (lon > 0) { 2
        do { 3
            if (cadena[contador] == letra) n++; 4
            contador++; 5
            lon--;
        } while (lon > 0); 6
    }
    return n; 7
}

```



#### 2. Complexidade ciclomática = 4

3. Camiños: (cada novo camiño engade polo menos unha nova arista)

- 1) 1-2-7
- 2) 1-2-3-4-5-6-7
- 3) 1-2-3-5-6-7
- 4) 1-2-3-4-5-6-3-5-6-7

4. Casos de Proba

Número	Camiño	cadena	letra	n
1	1-2-7	" "	'a'	0
2	1-2-3-4-5-6-7	"a"	'a'	1
3	1-2-3-5-6-7	"b"	'a'	0
4	1-2-3-4-5-6-3-5-6-7	"ab"	'a'	1

**Exercicio.** Dado un algoritmo ou programa, por exemplo, o seguinte:

```
Leer NT;    // Nota_de_Teoría
Leer NP;    // Nota_de_Prácticas
Leer NTR;   // Nota_de_Trabajos
SI NP <= 4.5 ENTONCES
    NF = 4;
SINO
    SI NT <= 4.5 ENTONCES
        NF=NT
    SINO
        NF = NT + NTR;
    FINSI

    SI NF>10 ENTONCES
        NF = 10;
    FINSI
FINSI
Escribir "nota final " . NF
```

1. Realiza o grafo de fluxo
2. Calcula a complexidade ciclomática
3. Calcula os camiños independentes
4. Elabora os casos de proba

### Probos de bucles

Os bucles ou estruturas repetitivas constitúen unha estrutura de control fundamental en toda linguaxe de programación. As probas de bucle céntranse exclusivamente neste tipo de estruturas. Os casos de proba que se deben xerar para cada tipo de bucle son os seguintes:

**Bucles simples:** se o número máximo de iteracións a través do bucle é **n**, deberían probarse os seguintes casos:

1. Salta o bucle por completo.
2. Realiza unha iteración a través do bucle.
3. Realiza dúas iteracións a través do bucle.
4. Realiza **m** iteracións a través do bucle, con **m < n**.
5. Realiza **n-1**, **n** e **n+1** iteracións a través do bucle.

**Bucles anidados:** deben realizarse os seguintes pasos:

1. Comeza co bucle máis interno, establecendo valores mínimos para todos os demais bucles.
2. Realiza probas sinxelas de bucle para o bucle máis interno, mantendo os bucles exteriores nos seus valores mínimos.
3. ...

## 3.2. Particións ou clases de equivalencia

A técnica de **particionamento en clases de equivalencia** é un **método de proba de caixa negra** que permite ao tester seleccionar unha mostra representativa de entradas dentro de cada clase, en lugar de probar todas as posibles entradas. Isto mellora a eficiencia das probas sen comprometer a cobertura.

A idea principal detrás das clases de equivalencia é reducir o número de probas necesarias, identificando subconxuntos de datos que teñen unha probabilidade similar de producir resultados correctos ou incorrectos.

Unha **clase de equivalencia** é un conxunto de entradas ou condicións que se poden considerar como equivalentes porque se espera que se comporten de maneira similar durante as probas.

As clases de equivalencia divídense en xeral en dous tipos:

1. **Clases de equivalencia válidas:** Son os conxuntos de entradas que se espera que sexan procesados correctamente polo sistema. Por exemplo, se unha aplicación acepta números entre 1 e 100, os números entre 1 e 100 formarían unha clase de equivalencia válida.
2. **Clases de equivalencia inválidas:** Son as entradas que o sistema non debe aceptar, xa que son inadecuadas ou están fóra do rango permitido. Usando o exemplo anterior, os números menores que 1 ou maiores que 100 serían clases de equivalencia inválidas.

A partición equivalente é un **método de proba de caixa negra** que divide os valores dos campos de entrada dun programa en clases de equivalencia.

Unha **clase de equivalencia** é un conxunto de datos de entrada que comparten o mesmo comportamento ou esperan o mesmo resultado, que definen **estados válidos e non válidos** do sistema.

**Exemplo:** Una conta de aforros ten diferentes taxas de interese dependendo do **saldo** da conta. Para unha conta cun saldo entre 0€ e 100 € xera o 3%, se a conta está entre 100,01€ e 1000€ o interese é o 5% e un 7% se a conta está por riba dos 1000€.

Partición inválida	Válida (3%)	Válida (5%)	Válida (7%)
< 0	0.00 - 100.00	100.01 - 1,000.00	> 1,000.00

Obtéñense a partir das condicións de entrada descritas nas especificación.

- Clase válida: xera un valor esperado
- Clase non válida: xera un valor inesperado

As condicións de entrada veñen representadas por sentenzas na especificación.

Un valor específico	“..Introducir cinco valores..”
Un conxunto de valores relacionados	“..Palabras reservadas nunha linguaxe ..”
Un rango de valores	“..Valores entre 0 e 10..”
Unha condición lóxica	Condición “..debe ser..”

As clases de equivalencia defínense segundo unha serie de directrices:

- Se unha condición de entrada especifica **un rango**, defínese unha clase de equivalencia válida e dúas non válidas. Exemplo de rango: a nota debe ter un valor entre 1 e 10.
- Se unha condición require **un valor específico**, defínese unha clase equivalente válida e dúas non válidas. Exemplo de valor específico: o número de departamento pode ser branco ou ter dous díxitos.
- Se unha condición de entrada especifica **un membro dun conxunto**, defínese unha clase equivalente válida e unha non válida. Exemplo, o curso pode ter os seguintes valores: “DAM1”, “DAM2”, “SMR1” e “SMR2”.
- Se unha condición de entrada é **unha condición lóxica**, defínese unha clase de equivalencia válida e unha non válida. Exemplo de entrada lóxica: o salario debe ser maior que 0.
- Se se sabe que algúns elementos dunha clase non se tratan de igual forma que o resto dos seus elementos, débese dividir dita clase de equivalencia en varias clases menores.

A seguinte táboa resume o número de clases de equivalencia válidas e non válidas que hai que definir para cada tipo de condición de entrada.

Condições de entrada	Nº de clases de equivalencia válidas	Nº de clases de equivalencia non válidas
<b>Rango</b>	1 clase que contemple los valores del rango (1)	2 clases non válidas 1 valor por encima do rango (2) 1 valor por debaixo do rango (3)
<b>Valor específico</b>	1 clase que contemple dicho valor (4)	2 clases non válidas 1 valor por encima (5) 1 valor por debaixo (6)
<b>Membro de un conxunto</b>	1 clase de equivalencia por cada elemento (7)	1 clase non válida 1 elemento fora do conxunto (8)
<b>Lóxica</b>	1 clase que cumpra a condición (9)	1 clase non válida 1 valor que non cumpra a condición (10)

Para a creación dos casos de proba lévanse a cabo os seguintes tres pasos:

1. Asignar un número a cada clase de equivalencia (na táboa anterior entre parénteses).

2. Crear o número mínimo de casos de prueba que inclúan todas as clases válidas.
3. Crear un caso de prueba por cada clase no válida. O caso de prueba debe incluír unha clase non válida e todas as demais clases debe ser válidas.

#### Identificación y etiquetado de las clase de equivalencia

Parámetro de entrada	Regla heurística a aplicar	Clases válidas	Clases inválidas
Código de área	Condición booleana (¿es un número?) + rango valores ([200..999])	1. $200 \leq \text{código} \leq 999$	2. código < 200 3. código > 999 4. no es número

### Actividad resuelta 3.3

#### Prueba de particiones o clases de equivalencia

En una aplicación, la persona usuaria debe introducir su edad, que debe ser un número entre 18 y 65; su NIF, que debe constar de 8 números y una letra; y, además, debe indicar si tiene o no nacionalidad española, de forma que tener nacionalidad española es un requisito.



**Tabla 3.1.** Condiciones de entrada para el supuesto, junto con las clases de equivalencia válidas y no válidas de acuerdo con las normas especificadas

Condición de entrada	Clases válidas	Clases no válidas
Edad	$18 \leq \text{edad} \leq 65$ (1)	edad < 18 (2) edad > 65 (3) No es un número (4)
NIF	Una cadena de 9 caracteres compuesta por 8 números y una letra (5)	< 9 caracteres (6) > 9 caracteres (7) Alguno de los 8 primeros caracteres no es un número (8) El último carácter no es una letra (9)
Nacionalidad	Española (10)	No española (11)

En la Tabla 3.2, se muestran los casos de prueba válidos y, en la Tabla 3.3, los no válidos. Por cada caso de prueba, se indica, en la última columna, las clases de equivalencia incluidas. Recuerda que, para el caso de las clases válidas, se debe crear el número mínimo de casos de prueba que incluyan todas las clases válidas; en este caso, se requiere un único caso de prueba.

**Tabla 3.2.** Caso de prueba con clases de equivalencia válidas

Edad	NIF	Nacionalidad	Clases incluidas
35	32323267G	Española	(1), (5), (10)

Para las clases no válidas, hay que crear un caso de prueba por cada una de ellas. Como se puede observar en la Tabla 3.1, al haber 8 clases de equivalencia no válidas, se deben crear 8 casos de prueba (Tabla 3.3).

**Tabla 3.3.** Casos de prueba con clases de equivalencia no válidas

Edad	NIF	Nacionalidad	Clases incluidas
16	78787654Z	Española	(2), (5), (10)
73	88788888U	Española	(3), (5), (10)
AB	56837483Y	Española	(4), (5), (10)
45	879847F	Española	(1), (6), (10)
23	6767676762 <sup>a</sup>	Española	(1), (7), (10)
64	TT789009R	Española	(1), (8), (10)
19	569832349	Española	(1), (9), (10)
23	98828282C	No española	(1), (5), (11)

**Ejercicio.** Dada a seguinte especificación:

Un programa toma como entrada un ficheiro co formato de rexistro seguinte:

Numero-alumno Nome-alumno Nota-obtida

onde:

- Numero-alumno é un campo de números enteiros positivos de 3 díxitos (excluído o 000).

- Nome-alumno é un campo alfanumérico de 10 caracteres.
- Nota-obtida é un enteiro positivo entre 0 e 10.

O programa asigna un texto (que se imprime nun listado) a cada alumno segundo a nota obtida:

- “Moi deficiente” a o alumnado cunha nota inferior a 3
- “Insuficiente” o alumnado cunha nota maior o igual a 3 e menor que 5
- “Suficiente” se o alumno ten unha nota igual a 5 e menor que 7
- “Notable” se o alumno ten unha nota igual a 7 e menor que 9
- “Sobresaínte” se o alumno ten unha nota maior ou igual a 9

Desenvolve a estratexia de Clases de Equivalencia:

1. **Crea unha táboa de Clases de Equivalencia** (as clases deberán ser numeradas) na que se indiquen as seguintes columnas en cada fila:
  - Condición de entrada que se analiza
  - Clases válidas
  - Clases non válidas que se xeran para a condición
2. **Xera os casos de proba** (especificando a entrada en todos os casos e a saída esperada só nos casos válidos) para as clases creadas usando a técnica de particións de equivalencia, indicando en cada caso as clases que cubre.

### 3.3. Análise de valores límite

Esta técnica baséase en que os erros teñen unha maior probabilidade de producirse nos valores límite ou nos extremos dos valores de entrada. **Complementa a técnica das particións equivalentes** pero ten algunhas diferencias:

1. En lugar de seleccionar calquera elemento dentro dunha clase de equivalencia, selecciónanse aquel ou aqueles que están xusto nos límites da clase.
2. En lugar de centrarse unicamente nas **condicións de entrada**, esta técnica tamén xera casos de proba tendo en conta as **condicións de saída**.

As regras para xerar casos de proba son as seguintes:

1. Se unha **condición de entrada** especifica un **intervalo** de valores, por exemplo, entre -5 e 10, débense xerar casos válidos para os extremos do intervalo (-5 e 10) e casos non válidos xusto por riba e por baixo dos extremos (-5,01 e 10,01, no caso de que se admitan dous decimais).  
Outro exemplo, se unha entrada require un rango de valores enteiros comprendidos entre 1 e 10, hai que escribir casos de proba para os valores 1, 10, 0 e 11.
2. Se unha **condición de entrada** especifica un **número de valores** (entre 1 e 100), débense xerar casos de proba para os valores mínimos e máximos (1 e 100), un menos que o mínimo (0) e un máis que o máximo (101).  
Outro exemplo, se o programa require de dous a dez datos de entrada, hai que escribir casos de proba válidos para 2 e 10 datos de entrada, e casos non válidos para 1 e 11 datos de entrada.
3. Aplicar a regra 1 para cada **condición de saída** que especifica un **intervalo** de valores.

Por exemplo, se o salario mínimo é de 1.200,00 € e o máximo é de 3.600,00 €, deberán xerarse casos de proba que resulten en salarios de 1.200,00 €, 3.600,00 €, 1.199,99 € e 3.600,01 €.

Outro exemplo, se se debe aplicar sobre un campo de saída un desconto de entre un 10% mínimo e un 50% máximo (dependendo do tipo de cliente); haberá que deseñar casos de proba para o 9.99%, 10%, 50% e 50.01%.

4. Usar a regra 2 para cada **condición de saída** que especifica un **número de valores**.

Por exemplo, se o programa especifica que se poden utilizar de un a tres descontos, haberá que xerar casos de proba para conseguir cero, un, tres e catro descontos.

Outro exemplo, se a saída dun programa é unha táboa de temperaturas de 1 a 10 elementos, débense deseñar casos de proba para que a saída do programa produza 0, 1, 10 e 11 elementos. Tanto nesta regra como na anterior, hai que ter en conta que non sempre se poderán xerar resultados fora do rango de saída.

5. Se a entrada ou saída dun programa é un conxunto ordenado ou unha **estrutura de datos internas** (por exemplo, unha táboa ou un ficheiro secuencial) que teñen límites preestablecidos (por exemplo, un array de 100 elementos), hai que deseñar casos de proba que proben a estrutura de datos nos seus límites, é dicir, no primeiro e último elemento.

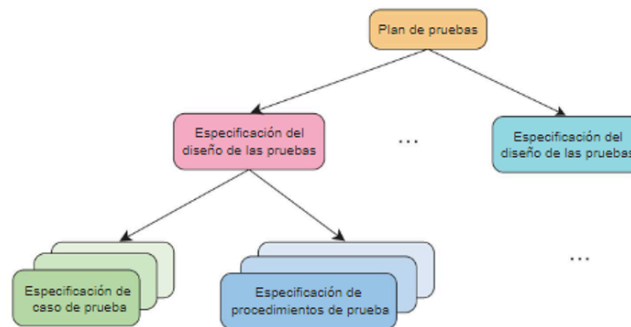
Exemplos:

	Condições de Entrada e saída	Casos de proba
<b>Código</b>	Enteiro de 1 a 100	Valores: 0, 1, 100 e 101
<b>Posto</b>	Alfanumérico de ata 4 caracteres	Lonxitude de caracteres: 0, 1, 4, 5
<b>Antigüidade</b>	De 0 a 25 anos (Real)	Valores: 0, 25, -0.1 , 25.01
<b>Horas semanais</b>	De 0 a 60	Valores: 0, 60, -1, 61
<b>Arquivo de entrada</b>	Ten de 1 a 100 rexistros	Para ler 0, 1, 100, 101 rexistros
<b>Arquivo de saída</b>	Poderá ter de 0 a 10 rexistros	Para xerar 0, 10 e 11 rexistros, (non se pode xerar -1 rexistro)
<b>Array interno</b>	De 20 cadeas de caracteres	Para o primeiro e o último elemento

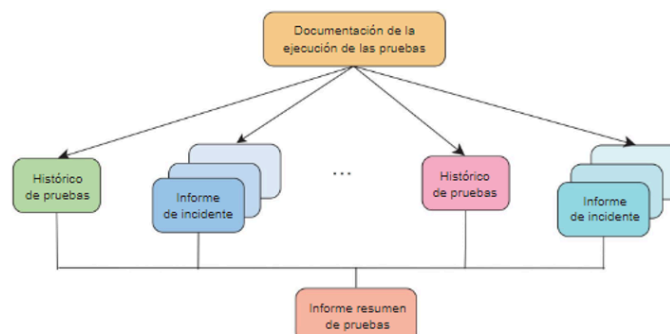
## 4. Documentación para as probas

A documentación das probas é conveniente para unha boa organización e para asegurar a súa reutilización. Hai que documentar tanto o deseño das probas como o resultado da súa execución. O estándar IEEE 829-1998 describe o conxunto de documentos que poden producirse durante o proceso de proba. Son os seguintes:

- **Plan de probas.** Recolle a planificación xeral das probas da aplicación que se está a crear. Describe o alcance, o enfoque, os recursos e o calendario das actividades de proba. Identifica os elementos a probar, as características que se van probar, as tarefas que se van realizar, o persoal responsable de cada tarefa e os riscos asociados ao plan.
- **Especificacións de proba.** Detalla o plan de probas
  - **Especificación dos casos de proba**, documenta os valores reais utilizados para a entrada, xunto cos resultados esperados e as posibles dependencias entre casos de proba.
  - **Especificación dos procedementos de proba**, indica os pasos para executar os casos de proba especificados, incluíndo o contorno necesario, o persoal responsable.



- **Informes de probas.**
  - **Histórico de probas**, que rexistra os feitos relevantes durante a execución da proba: data e hora, contorno, elementos probados, resultados e referencia ao informe de incidentes se é o caso.
  - **Informe de incidentes**, que describe calquera evento que se produce durante a execución da proba que require maior investigación.
  - **Informe resumen** das actividades de proba, que inclúe resumo da avaliación dos elementos probados, variacións do software como resultado das probas, valoración da cobertura, etc.

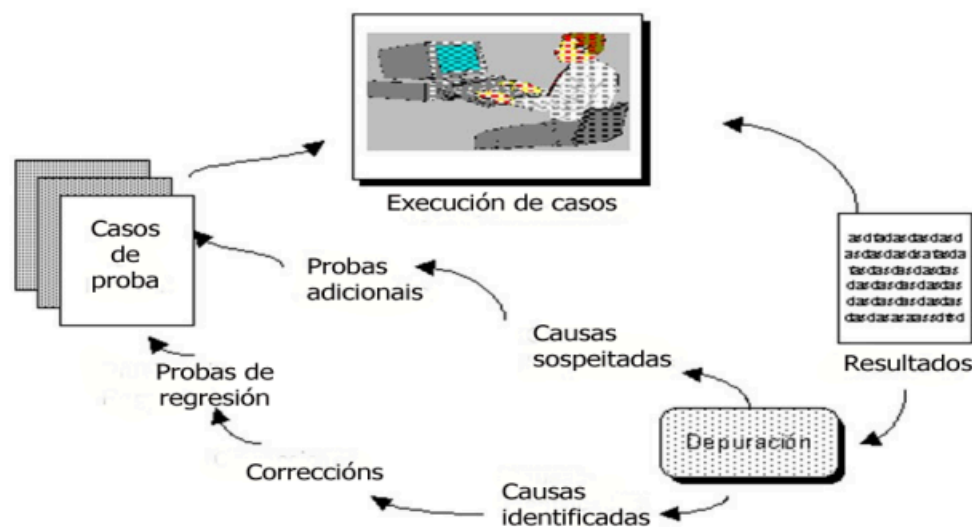


## 5. Ferramentas de depuración

O proceso de depuración comeza coa execución dun caso de proba. Avalíanse os resultados da execución e froito desa avaliación compróbase que hai unha falta de correspondencia entre os resultados esperados e os obtidos realmente.

O proceso de depuración sempre ten un dos resultados seguintes:

1. Atópase a causa do erro, corríxese e elimínase
2. Non se atopa a causa do erro. Neste caso, a persoa encargada da depuración debe sospeitar a causa, deseñar os casos de proba que axuden a confirmar as sospeitas e volver a repetir as probas para identificar e corrixir os erros (probas de regresión, repetición selectiva de probas para detectar fallos introducidos durante a modificación).



Ao desenvolver programas cométese dous tipos xerais de erros: erros de compilación e erros lóxicos.

- Os **erros de compilación** son doados de corrixir xa que normalmente se emprega un IDE para codificar e escribir as sentenzas. O contorno proporciona información da localización do erro e de como solucionalo, especialmente nas linguaxes compiladas, nas que o programa non pode compilarse ata que se corrixan todos os erros.
- Os **erros lóxicos**, comunmente chamados *bugs*, estes non impiden que o programa compile con éxito pero poden provocar que o programa devolva resultados incorrectos, que non sexan os esperados, ou poden provocar que o programa remate abruptamente antes de tempo ou que non remate nunca.

Para solucionar erros lóxicos, as contornas de desenvolvemento incorporan unha ferramenta coñecida como depurador ou *debugger*. O depurador permite analizar todo o programa, mentres este execútase. Permite suspender a execución dun programa, examinar e establecer os valores das variables, comprobar os valores devoltos por un determinado método, o resultado dunha comparación lóxica ou relacional etc.

## 5.1. Execución paso a paso

### Debugger

- [Depuración paso a paso de entornos de desarrollo como Netbeans](#)
- [Run and Debug Java in Visual Studio Code](#)

Para depurar un programa podemos utilizar o depurador para executar o programa de diferentes formas:

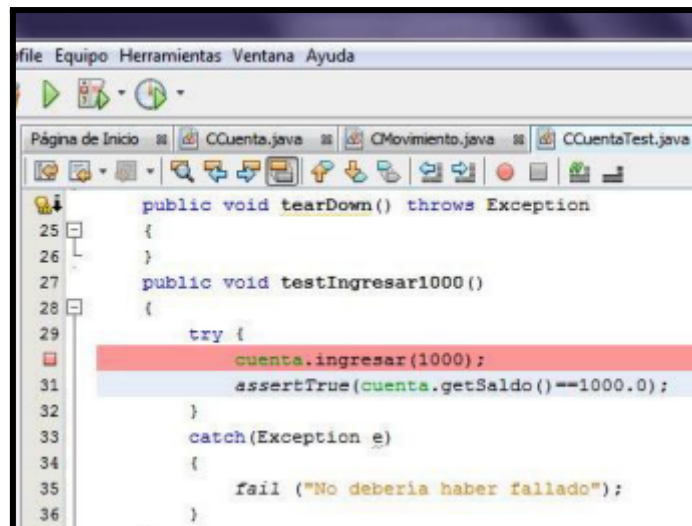
- A execución **paso a paso** permite executar un programa liña por liña para buscar e corrixir erros lóxicos.
- A execución **paso a paso por procedementos** permítenos executar un método ou función completamente. É útil, cando xa comprobamos que un procedemento funciona correctamente e só nos interesa o valor que devolve.
- Na **execución ata unha instrución**, o depurador executa o programa, e detense na instrución onde se atopa o cursor, a partir dese punto, podemos facer unha depuración paso a paso ou por procedementos.
- Na **execución dun programa ata o final**, executamos o resto do programa completamente sen deternos nas instrucións intermedias.

Nos distintos IDE's, no menú de depuración, pódense seleccionar os modos de execución especificados, e algúns máis. O obxectivo é poder examinar todas as partes que se consideren necesarias, de maneira rápida, sinxela e o máis clara posible.

## 5.2. Puntos de ruptura (*breakpoints*)

Dentro das opcións de depuración contamos coa posibilidade de inserir puntos de ruptura (*breakpoint*). Selecciónase a liña de código onde queremos que o programa se pare para a partir dela, inspeccionar variables, ou realizar unha execución paso a paso, para verificar a corrección do código.

Durante a proba dun programa, pode ser interesante a verificación de determinadas partes do código. Non nos interesa probar todo o programa, xa que delimitamos o punto concreto onde inspeccionar. Para iso, utilizamos os puntos de ruptura.



Punto de ruptura en NetBeans

Os puntos de ruptura son marcadores que **poden establecerse en calquera liña de código executable** (non sería válido nun comentario, ou nunha liña en branco). Unha vez inserido o punto de ruptura, e iniciada a depuración, o programa a avaliar executaríase ata a liña marcada co punto de ruptura. Nese momento, pódense realizar diferentes labores, por unha banda, pódense examinar as variables, e comprobar que os valores que teñen asignados son correctos, ou pódese iniciar unha depuración paso a paso, e ir comprobando o camiño que toma o programa a partir do punto de ruptura. Unha vez realizada a comprobación, podemos abortar o programa, ou continuar a execución normal do mesmo.

Dentro dunha aplicación, pódense inserir varios puntos de ruptura, e pódense eliminar coa mesma facilidade coa que se inseriron.

### 5.3. Examinar variables

Mentres o código está sendo analizado polo depurador, podemos ver os tipos e contidos das variables a medida que avanza o programa. Tamén é posible cambiar o valor das variables ou avaliar expresións.

Os **examinadores de variables**, forman un dos elementos máis importantes do proceso de depuración dun programa. Iniciado o proceso de depuración, normalmente coa execución paso a paso, o programa avanza instrución por instrución. Ao mesmo tempo, as distintas variables, van tomando diferentes valores. Cos examinadores de variables, podemos comprobar os distintos valores que adquieren as variables, así como o seu tipo. Esta ferramenta é de gran utilidade para a detección de erros.



The screenshot displays the NetBeans IDE interface. The top pane shows a Java code editor with the following code:

```
21 public double potencia (double base, double exponente)
22 {
23     int i;
24     double result=0;
25     try
26     {
27         for (i=0;i<exponente;i++)
28         {
29             result=result*base;
30         }
31     }
32     catch (Exception ex){
33         System.out.println("Se ha producido un error");
34     }
35 }
```

The line `double result=0;` is highlighted in green. Below the code editor, the 'Variables' window is open, showing the current state of variables during a debug session. The window has tabs for 'Variables', 'Puntos de Interru...', 'Salida', and 'Tareas'. The 'Variables' tab is active, displaying a table with columns 'Nombre', 'Tipo', and 'Valor'.

Nombre	Tipo	Valor
<Escriba el nuevo reloj>		
this	CFunciones	#46
base	double	2.0
exponente	double	3.0

At the bottom of the IDE, the status bar indicates 'Pruebas\_de\_Software (debug)' and 'running...'.

Examinando variables en NetBeans



## 6. Probas Unitarias

- [Probas Unitarias con JUNIT](#)

1. Creación dunha clase de proba
2. Preparación e execución das probas
3. Anotacións
4. Probas parametrizadas
5. Suite de probas
6. Cobertura

## Anexos e Ligazóns

- <https://www.codigocero.com/As-vantaxes-da-metodoloxia-TDD-centran-un-novo-curso-de-Vitae-entre-marzo-e>