Testing Estructural
Buenas Prácticas

Algoritmos y Estructuras de Datos I

Buenas prácticas

"Por buenas o mejores prácticas se entiende un conjunto coherente de acciones que han rendido buen o incluso excelente servicio en un determinado contexto y que se espera que, en contextos similares, rindan similares resultados."

- Con respecto al software... hay **mucha información** al respecto... pero hay que saber organizarla, depurarla, clasificarla, etc!
  - Buenas prácticas a nivel metodología, programación en determinado lenguaje, de diseño, de testing, de documentación, de experiencia de usuario, etc.
- Para no generalizar... en lugar de buenas prácticas, daremos **fuertes recomendaciones**

Utilizar nombres declarativos

```
1 int x = 0;
2 vector<double> y;
3 ...
4 for(int i=0;i<=4;i=i+1) {
5  x = x + y[i];
6 }
```

```
1 int totalAdeudado = 0;
2 vector<double> deudas;
3 ...
4 for(int i=0;i<=4;i=i+1) {
5  totalAdeudado = totalAdeudado + deudas[i];
6 }
```

Utilizar nombres declarativos

```
1
2 int x = aux1();
3 int y = aux2();
4 int z = x / y;
5 return z;
```

```
1
2 int cantidadPaisesLatinos = obtenerTotalPaisesLatinos();
3 int totalDePaises = obtenerTotalDePaises();
4 int promedio = cantidadPaisesLatinos / totalDePaises;
5 return promedio;
```

Encapsular, modularizar, abstraer

- Encapsular comportamiento dentro de funciones auxiliares permite:
  - Facilitar la reutilización
  - Mejorar la legibilidad del código

```
1 int x = aux1();
2 int y = aux2();
3 int z = (x + y)/2;
4 return z;
```

```
1 int col = notaColoquio();
2 int pra = notaPracticas();
3 int notaFinal = calcularPromedio(col, pra);
4 return notaFinal;
```

Indentación (sangrado)

```
1 int main () {
2 int i, j;
3 for (i = 0; i <= 10; i++){
4 for (j = 0; j <= 10; j++){
5 cout << i << " x " << j << " = " << i*j;
6 }
7 }
8 return 0;
9 }
```

```
1 int main () {
2   int i, j;
3   for (i = 0; i <= 10; i++){
4     for (j = 0; j <= 10; j++){
5       cout << i << " x " << j << " = " << i*j;
6     }
7   }
8   return 0;
9 }
```

Indentación (sangrado)

- Algunos lenguajes fuerzan la indentación (ejemplo: Python)
- Algunos IDEs ayudan a automatizar la indentación
  - CLion: Code → Reformat Code

Comentarios

- Además de escribir comandos, los lenguajes de programación permiten escribir **Comentarios**
- Estos comentarios permiten:
  - Describir el comportamiento esperado de una función.
  - Registrar decisiones importantes tomadas por el programador
- Tipos de comentarios en C++:
  - Comentario de línea: `// ...`
  - Comentario de bloque: `/* ... */`

Comentarios: Malos usos

- Ejemplo 1:

```
/**
  * Función que toma dos enteros y devuelve un float.
  */
float calculateModule(int x, int y) {...}
```

- Ejemplo 2:

```
while (x<y) { // itero hasta que x es mayor o igual que y
  ...
}
```

- Ejemplo 3:

```
cout << y; // Imprimo el valor de y
```

Comentarios: Buenos usos

- Ejemplo 1:

```
/**
  * Computa el módulo del vector de a partir de dos coordenadas.
  * Las coordenadas deben ser números no negativos.
  * Si la precondición no se cumple, retorna -1.
  */
float calculateModule(int x, int y) {...}
```

- Ejemplo 2:

```
// almacena los Ids de los clientes que compraron productos
vector<s> clientIds;
```

- Ejemplo 3:

```
// el llamado a f se hace siempre con y>0
x = f(y);
```

Variables: Inicialización

- **Siempre** inicializar las variables.
- Falta inicializar:

```
int i;
i = 10; // mal
```

- Bien inicializado:

```
int i = 10; // bien
```

Variables: Scope de declaración

- Usar el scope más *estrecho* posible
- Ejemplo: Definición fuera de scope más estrecho:

```
int main() { // scope 1
  int t = 0;
  while (...) { // scope 2
    while (...) { // scope 3
      t = ...
    }
  }
```

- Ejemplo: Definición en scope más estrecho:

```
int main() { // scope 1
  while (...) { // scope 2
    while (...) { // scope 3
      int t = ...
    }
  }
```

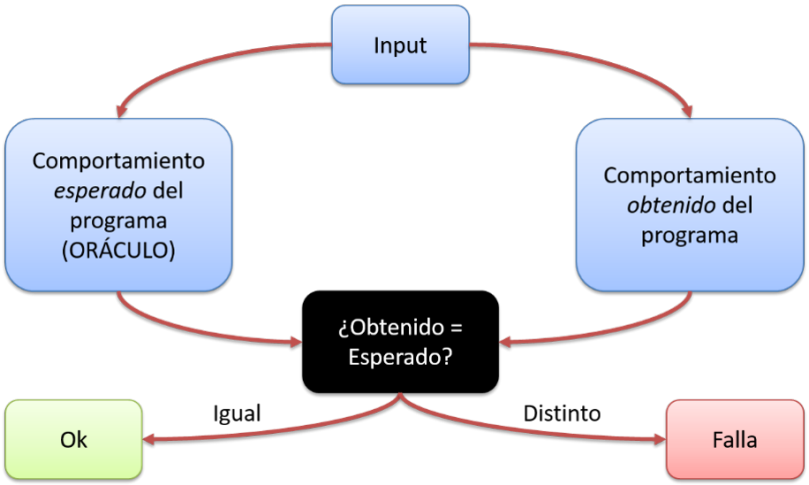
Recap: ¿Por qué escribir la especificación del problema?

- Nos ayuda a entender mejor el problema
- Nos ayuda a construir el programa
  - Derivación (Automática) de Programas
- Nos ayuda a prevenir errores en el programa
  - Testing
  - Verificación (Automática) de Programas

¿Qué es hacer testing?

- Es el proceso de ejecutar un producto para ...
  - Verificar que satisface los requerimientos (en nuestro caso, la especificación)
  - Identificar diferencias entre el comportamiento **real** y el comportamiento **esperado** (IEEE Standard for Software Test Documentation, 1983).
- Objetivo: encontrar defectos en el software.
- Representa entre el 30 % al 50 % del costo de un software confiable.

¿Cómo se hace testing?



Test Input, Test Case y Test Suite

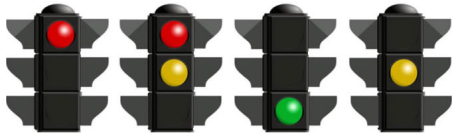
- **Programa bajo test**: Es el programa que queremos saber si funciona bien o no.
- **Test Input** (o dato de prueba): Es una asignación concreta de valores a los parámetros de entrada para ejecutar el programa bajo test.
- **Test Case:** Caso de Test (o caso de prueba). Es un programa que ejecuta el programa bajo test usando un dato de test, y chequea (automáticamente) si se cumple la condición de aceptación sobre la salida del programa bajo test.
- **Test Suite**: Es un conjunto de casos de Test (o de conjunto de casos de prueba).

Hagamos Testing

- ¿Cuál es el programa de test?
  - Es la implementación de una especificación.
- ¿Entre qué datos de prueba puedo elegir?
  - Aquellos que cumplen la **precondición** en la especificación
- ¿Qué condición de aceptación tengo que chequar?
  - La condición que me indica la **postcondición** en la especificación.
- ¿Qué pasa si el dato de prueba no satisface la precondición de la especificación?
  - Entonces no tenemos ninguna condición de aceptación

Ejemplo: Especificando un semáforo

- Sea la siguiente especificación de un semáforo:



- proc *avanzar*(inout v, a, r : Bool) {

  Pre {
  $esValido(v, a, r)$
  $\wedge v = v_0 \wedge r = r_0 \wedge a = a_0$
  }

  Post {
  $(esRojo(v_0, a_0, r_0) \rightarrow esRojoAmarillo(v, a, r))$
  $\wedge\ (esRojoAmarillo(v_0, a_0, r_0) \rightarrow esVerde(v, a, r)$
  $\wedge\ (esVerde(v_0, a_0, r_0) \rightarrow esAmarillo(v, a, r))$
  $\wedge\ (esAmarillo(v_0, a_0, r_0) \rightarrow esRojo(v, a, r))\}$
  }

Ejemplo: Semáforo

- Programa a testear:
  - `avanzar(bool &v, bool &a, bool &r)`
- Test Suite:
  - Test Case #1 (AvanzarRojo):
    - Entrada: ($v = false, a = false, r = true$)
    - Salida Esperada: $v = false, a = true, r = true$
  - Test Case #2 (AvanzarRojoYAmarillo):
    - Entrada: ($v = false, a = true, r = true$)
    - Salida Esperada: $v = true, a = false, r = false$
  - Test Case #3 (AvanzarVerde):
    - Entrada: ($v = true, a = false, r = false$)
    - Salida Esperada: $v = false, a = true, r = false$
  - Test Case #4 (AvanzarAmarillo):
    - Entrada: ($v = false, a = true, r = false$)
    - Salida Esperada: $v = false, a = false, r = true$

Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?

proc *valorAbsoluto*(in $n : \mathbb{Z}$, out $result : \mathbb{Z}$){
Pre {$True$}
Post {$result = \|n\|$}
}

- Probar valorAbsoluto con 0, chequear que result=0
- Probar valorAbsoluto con -1, chequear que result=1
- Probar valorAbsoluto con 1, chequear que result=1
- Probar valorAbsoluto con -2, chequear que result=2
- Probar valorAbsoluto con 2, chequear que result=2
- ...etc.
- ¿Cuántas entradas tengo que probar?

Probando (Testeando) programas

- Si los enteros se representan con 32 bits, necesitaríamos probar 2^{32} datos de test.
- Necesito escribir un test suite de 4,294,967,296 test cases.
- Incluso si lo escribo automáticamente, cada test tarda 1 milisegundo, necesitaríamos $1193,04$ horas (49 días) para ejecutar el test suite.
- Cuanto más complicada la entrada (ej: secuencias), más tiempo lleva hacer testing.
- La mayoría de las veces, el testing exhaustivo no es práctico.

Limitaciones del testing

- Al no ser exhaustivo, el testing NO puede probar (demostrar) que el software funciona correctamente.

"El testing puede demostrar la presencia de errores nunca su ausencia" (Dijkstra)

- Una de las mayores dificultades es encontrar un conjunto de tests adecuado:
  - **Suficientemente grande** para abarcar el dominio y maximizar la probabilidad de encontrar errores.
  - **Suficientemente pequeño** para poder ejecutar el proceso con cada elemento del conjunto y minimizar el costo del testing.

¿Con qué datos probar?

- **Intuición:** hay inputs que son "parecidos entre sí" (por el tratamiento que reciben)
- Entonces probar el programa con uno de estos inputs, ¿equivaldría a probarlo con cualquier otro de estos parecidos entre sí?
- Esto es la base de la mayor parte de las técnicas
- ¿Cómo definimos cuándo dos inputs son "parecidos"?
  - Si únicamente disponemos de la especificación, nos valemos de nuestra *experiencia*

Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?

proc $valorAbsoluto$(in $n : \mathbb{Z}$, out $result : \mathbb{Z}$){
  Pre $\{True\}$
  Post $\{result = \|n\|\}$
}

Ejemplo:

- Probar valorAbsoluto con 0, chequear que result=0
- Probar valorAbsoluto con un valor negativo x, chequear que result=-x
- Probar valorAbsoluto con un valor positivo x, chequear que result=x

Ejemplo: valorAbsoluto

- Programa a testear:
  - `int valorAbsoluto(int x)`
- Test Suite:
  - Test Case #1 (cero):
    - Entrada: $(x = 0)$
    - Salida Esperada: $result = 0$
  - Test Case #2 (positivos):
    - Entrada: $(x = 1)$
    - Salida Esperada: $result = 1$
  - Test Case #3 (negativos):
    - Entrada: $(x = -1)$
    - Salida Esperada: $result = 1$

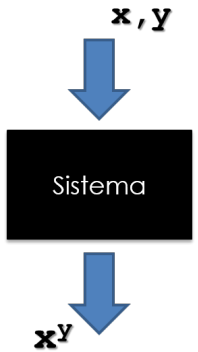
Retomando... ¿Qué casos de test elegir?

1. No hay un algoritmo que proponga casos tales que encuentren todos los errores en cualquier programa.
2. Ninguna técnica puede ser efectiva para detectar todos los errores en un programa arbitrario
3. En ese contexto, veremos dos tipos de criterios para seleccionar datos de test:
   - **Test de Caja Negra**: los casos de test se generan analizando la especificación sin considerar la implementación.
   - **Test de Caja Blanca**: los casos de test se generan analizando la implementación para determinar los casos de test.

Criterios de **caja negra** o funcionales

- Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

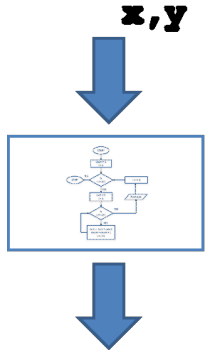
proc *fastexp*(in $x : \mathbb{Z}$, in $y : \mathbb{Z}$, out $result : \mathbb{Z}$){
  Pre $\{x \neq 0 \leftrightarrow y = 0\}$
  Post $\{result = x^y)\}$
}



Criterios de **caja blanca** o estructurales

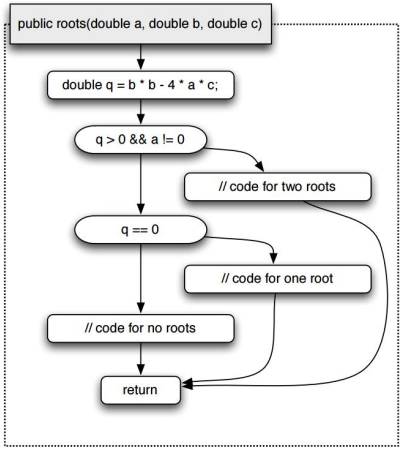
- Los datos de test se derivan a partir de la estructura interna del programa.

```
1  int fastexp(int x, int y) {
2    int z = 1;
3    while( y != 0 ) {
4      if(impar(y)) then {
5        z = z * x;
6        y = y - 1;
7      }
8      x = x * x;
9      y = y / 2;
10   }
11   return z;
12 }
```



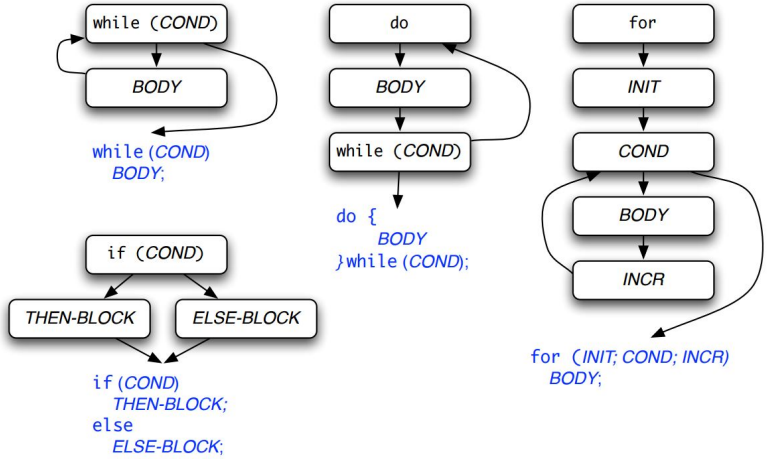
¿Qué pasa si y es potencia de 2?
¿Qué pasa si y = 2n -1?

Control-Flow Graph



- El control flow graph (CFG) de un programa es sólo una representación gráfica del programa.
- El CFG es independiente de las entradas (su definición es estática)
- Se usa (entre otras cosas) para definir criterios de adecuación para test suites.
- Cuanto más *partes* son ejercitadas (cubiertas), mayores las chances de un test de descubrir una falla
- *partes* pueden ser: nodos, arcos, caminos, decisiones...

Control Flow Patterns

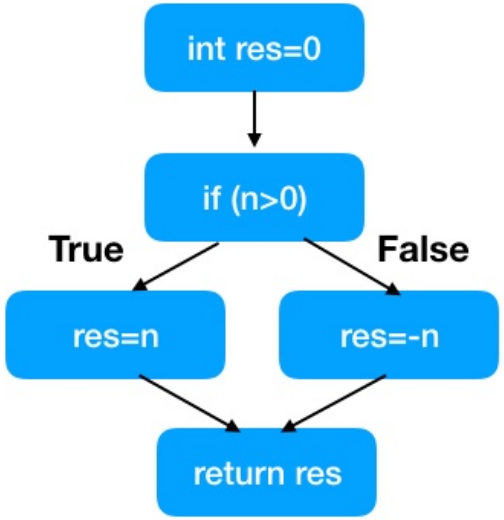


Ejemplo #1

proc *valorAbsoluto*(in $x : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre {*True*}
  Post {*result* = $\|x\|$}
}

```
1  int valorAbsoluto(int n) {
2    int res = 0;
3    if( n > 0 ) {
4      res = n;
5    } else {
6      res = −n;
7    }
8    return res;
9  }
```

CFG de valorAbsoluto

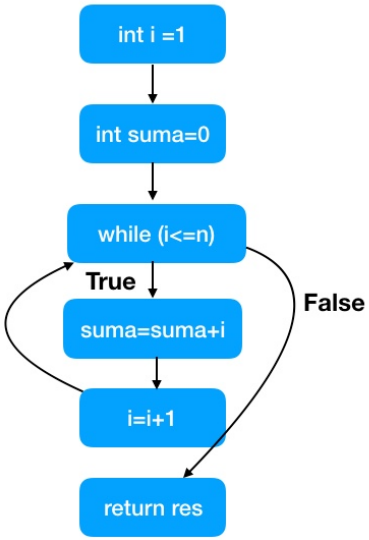


Ejemplo #2

proc *sumar*(in $n : \mathbb{Z}$, out *result* : $\mathbb{Z}$){
  Pre $\{n \geq 0\}$
  Post $\{result = \sum_{i=1}^{n} i\}$
}

```
1  int sumar(int n) {
2    int i = 1;
3    int suma = 0;
4
5    while( i <= n ) {
6      suma = suma + i;
7      i = i + 1;
8    }
9    return suma;
10 }
```

CFG de sumar

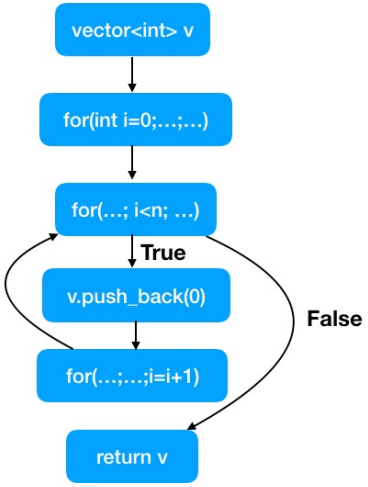


Ejemplo #3

proc *crearVectorN*(in $n : \mathbb{Z}$, out *result* : $seq\langle\mathbb{Z}\rangle$){
  Pre $\{n \geq 0\}$
  Post $\{|result| = n \wedge \#apariciones(result, 0) = n\}$
}

```
1  vector<int> crearVectorN(int n) {
2    vector<int> v;
3    for (int i=0; i<n;i=i+1) {
4      v.push_back(0);
5    }
6    return v;
7  }
```

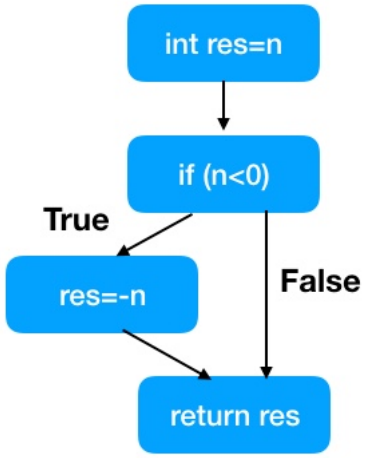
CFG de sumar



Ejemplo #4

```
1  int valorAbsoluto(int n) {
2    int res = n;
3    if( n < 0 ) {
4      res = −n;
5    }
6    return res;
7  }
```

CFG de valorAbsoluto



Criterios de Adecuación

- ¿Cómo sabemos que un *test suite* es *suficientemente bueno*?
- Un criterio de adecuación de test es un predicado que toma un valor de verdad para una tupla <*programa*, *test suite*>
- Usualmente expresado en forma de una regla del estilo:

  todas las sentencias deben ser ejecutadas

Cubrimiento de Sentencias

- Criterio de Adecuación: cada nodo (sentencia) en el CFG debe ser ejecutado al menos una vez por algún test case
- Idea: un defecto en un sentencia sólo puede ser revelado ejecutando el defecto
- Cobertura:

$$\frac{cantidad\ nodos\ ejercitados}{cantidad\ nodos}$$

Cubrimiento de Arcos

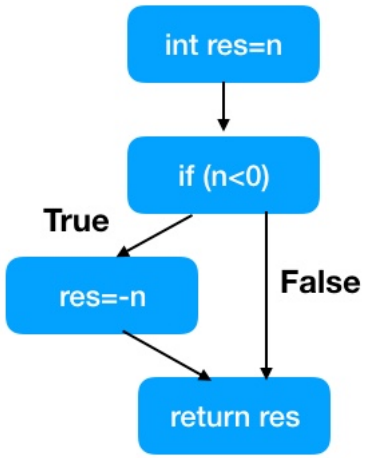
- Criterio de Adecuación: todo arco en el CFG debe ser ejecutado al menos una vez por algún test case
- Si recorremos todos los arcos, entonces recorremos todos los nodos. Por lo tanto, el cubrimiento de arcos incluye al cubrimiento de sentencias.
- Cobertura:

$$\frac{cantidad\ arcos\ ejercitados}{cantidad\ arcos}$$

- El cubrimiento de sentencias (nodos) no incluye al cubrimiento de arcos. ¿Por qué?

Cubrimiento de Nodos no incluye cubrimiento de Arcos

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los nodos pero que no cubra todos los arcos.

Cubrimiento de Decisiones (o Branches)

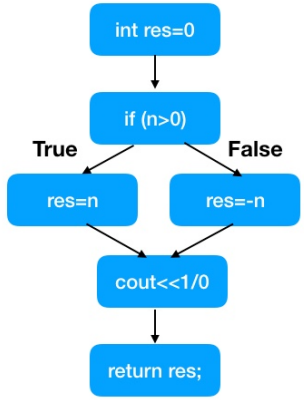
- Criterio de Adecuación: cada decisión (arco True o arco False) en el CFG debe ser ejecutado
- Por cada arco **True** o arco **False**, debe haber al menos un test case que lo ejercite.
- Cobertura:

$$\frac{cantidad\ decisiones\ ejercitadas}{cantidad\ decisiones}$$

- El cubrimiento de decisiones **no implica** el cubrimiento de los arcos del CFG. ¿Por qué?

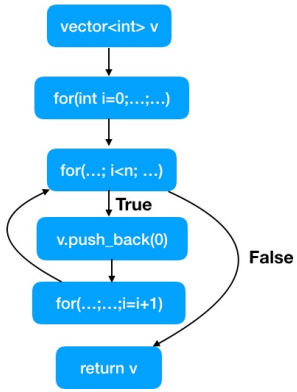
Cubrimiento de Branches no incluye cubrimiento de Arcos

Sea el siguiente CFG:



En este ejemplo, puedo construir un test suite que cubra todos los branches pero que no cubra todos los arcos.

CFG de sumar



- ¿Cuántos nodos (sentencias) hay? 6
- ¿Cuántos arcos (flechas) hay? 6
- ¿Cuántas decisiones (arcos True y arcos False) hay? 2

Cubrimiento de Condiciones Básicas

- Criterio de Adecuación: cada condición básica de cada decisión en el CFG debe ser evaluada a verdadero y a falso al menos una vez
- Decisiones no implica Condiciones Básicas
  - **if(a && b)**
  - Decisiones: $a \wedge b, \neg(a \wedge b)$; pueden faltar $\neg a$ y $\neg b$
- Cobertura:

$$\frac{cantidad\ condiciones\ ejecutadas}{cantidad\ condiciones}$$

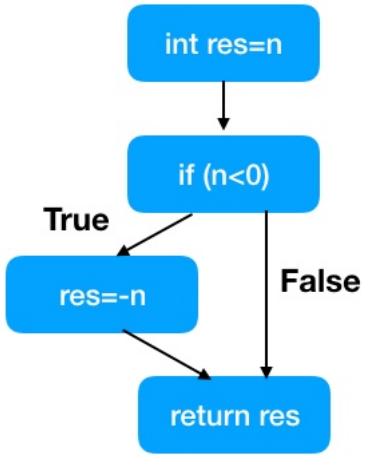
Cubrimiento de Caminos

- Criterio de Adecuación: cada camino en el CFG debe ser transitado por al menos un test case
- Cobertura:

$$\frac{cantidad\ caminos\ transitados}{cantidad\ total\ de\ caminos}$$

Caminos para el CFG de valorAbsoluto

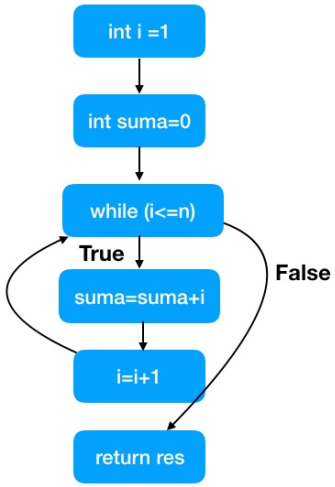
Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? 2

Caminos para el CFG de sumar

Sea el siguiente CFG:



¿Cuántos caminos hay en este CFG? La cantidad de caminos no está acotada (∞)

Recap: Criterios de Adecuación Estructurales

- En todos estos criterios se usa el CFG para obtener una métrica del test suite
- **Sentencias**: cubrir todas los nodos del CFG
- **Arcos**: cubrir todos los arcos del CFG
- **Decisiones (Branches)**: Por cada `if`, `while`, `for`, etc., la guarda fue evaluada a verdadero y a falso.
- **Condiciones Básicas**: Por cada componente básico de una guarda, este fue evaluado a verdadero y a falso.
- **Caminos**: cubrir todos los caminos del CFG. Como no está acotado o es muy grande, se usa muy poco en la práctica.

Intervalo

Break!

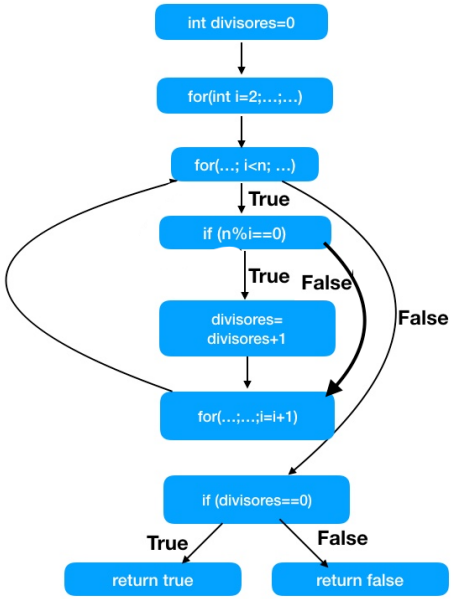
esPrimo()

Sea la siguiente implementación que decide si un número $n > 1$ es primo:

```
1  bool esPrimo(int n) {
2      int divisores = 0;
3      for(int i=2; i<n; i=i+1) {
4          if( n  % i == 0 ) {
5              divisores = divisores + 1;
6          } else {
7              // no hacer nada
8          }
9      }
10     if (divisores == 0) {
11         return true;
12     } else {
13         return false;
14     }
15 }
```

Graficar el CFG de la función `esPrimo()`.

CFG esPrimo()



Cubrimientos

Sea el siguiente test suite para esPrimo():

- Test Case #1: valorPrimo
  - Entrada: $n = 3$
  - Salida esperada: $result = true$
- Test Case #2: valorNoPrimo
  - Entrada: $n = 4$
  - Salida esperada: $result = false$
- ¿Cuál es el cubrimiento de sentencias (nodos) del test suite?

$$Cov_{sentencias} = \frac{9}{9} = 100\,\%$$

- ¿Cuál es el cubrimiento de decisiones (brances) del test suite?

$$Cov_{branches} = \frac{6}{6} = 100\,\%$$

Cubrimientos

Sea el siguiente test suite para esPrimo():

- Test Case #1: valorPar
  - Entrada: $n = 2$
  - Salida esperada: $result = true$
- Test Case #2: valorImpar
  - Entrada: $n = 3$
  - Salida esperada: $result = true$
- ¿Cuál es el cubrimiento de sentencias (nodos) del test suite?

$$Cov_{sentencias} = \frac{7}{9} \sim 77\,\%$$

- ¿Cuál es el cubrimiento de decisiones (brances) del test suite?

$$Cov_{branches} = \frac{4}{6} \sim 66\,\%$$

Discusión

- ¿Puede haber partes (nodos, arcos, branches) del programa que no sean alcanzables con **ninguna** entrada válida (i.e. que cumplan la precondición)?
- ¿Qué pasa en esos casos con las métricas de cubrimiento?
- Existen esos casos (por ejemplo: código defensivo o código que sólo se activa ante la presencia de un estado inválido)
- El 100 % de cubrimiento suele ser no factible, por eso es una medida para analizar con cuidad y estimar en función al proyecto (ejemplo: 70 %, 80 %, etc.)

Niveles de Test

- Test de Sistema
  - Comprende todo el sistema. Por lo general constituye el test de aceptación.

- Test de Integración
  - Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto
  - Testeamos la interacción, la comunicación entre partes

- Test de Unidad
  - Se realiza sobre una unidad de código pequeña, claramente definida.
    - ¿Qué es una unidad? Depende...

Niveles de Test → en el contexto del TP

- Test de Sistema
  - *Pruebas antes de entregar / pruebas de los docentes*

- Test de Integración
  - *Pruebas despues de unir la parte que hizo cada miembro del grupo*

- Test de Unidad
  - *Pruebas de la partecita que hizo cada miembro del grupo*

Google C++ Testing Framework

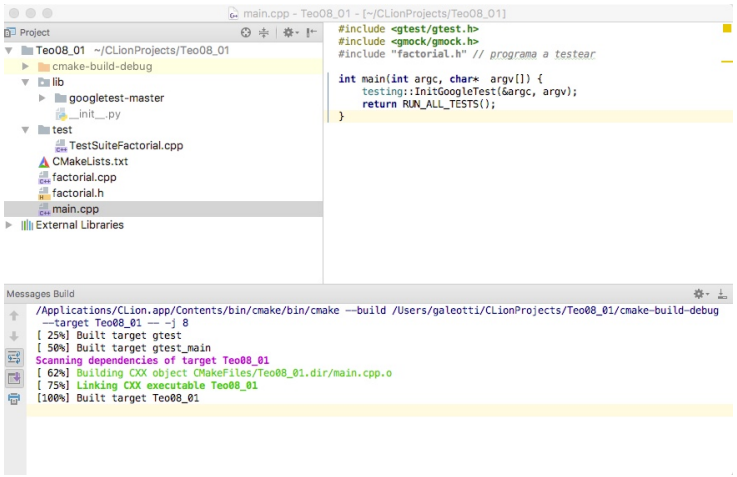
- **¿Por qué un framework de testing?**
- Nos olvidamos del *scaffolding*
- Ayuda a crear test independientes y repetibles
- Permite organizar mejor la información de debug
- Brinda herramientas de soporte

http://code.google.com/p/googletest

Google C++ Testing Framework

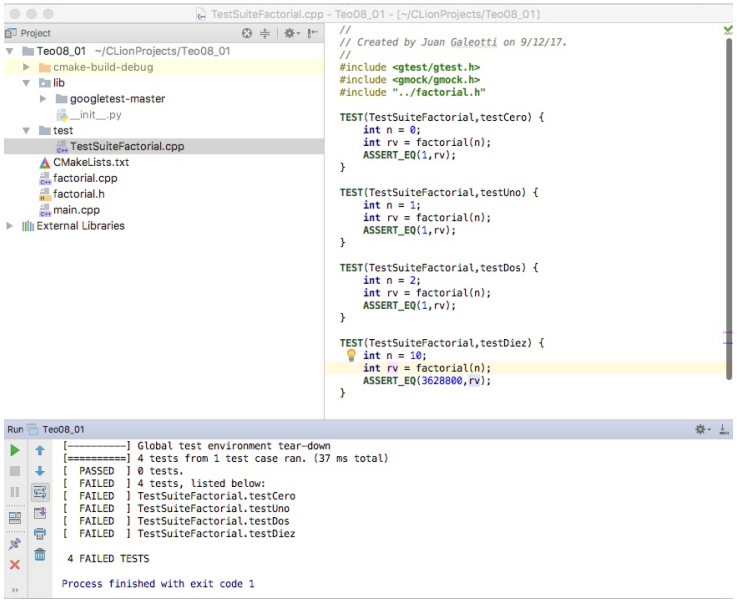
- **Estructura**
  - Cada test por separado, con la macro TEST
  - El test tiene un nombre y un grupo
- **Expectations / Assertions**
  - Assert() : ASSERT_* y EXPECT_*
  - * puede ser: TRUE, FALSE, EQ, LT, STREQ, etc
  - EXPECT_* : NO FATAL
  - ASSERT_* : FATAL (Interrumpe el test)

CLion+GTest (Usar el ejecutable/main)

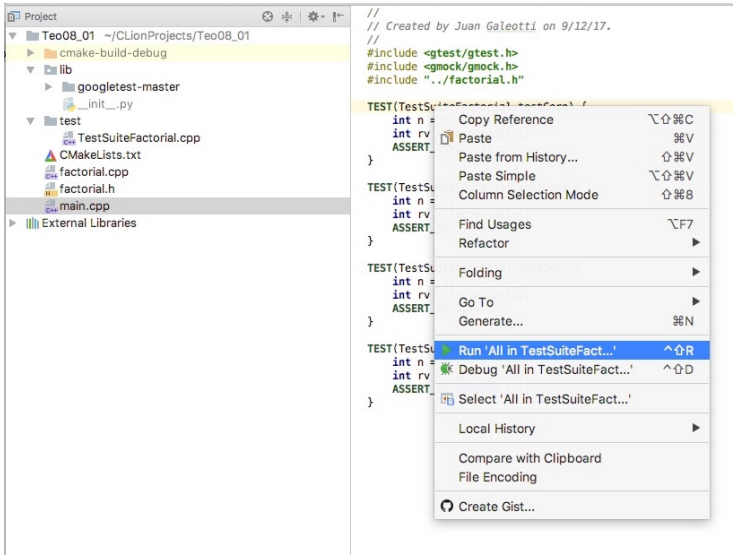


https://www.jetbrains.com/help/clion/google-test-support.html

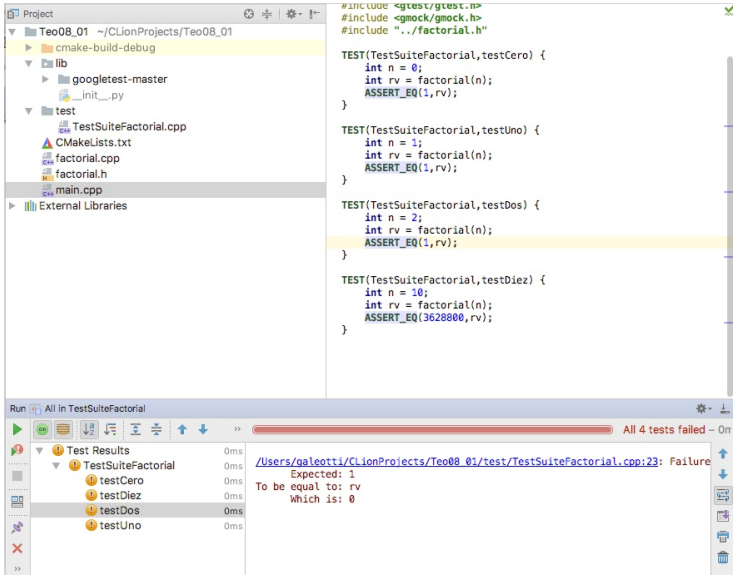
CLion+GTest (Usar el ejecutable/main)



CLion+GTest (Usar el Test Runner)



CLion+GTest (Usar el Test Runner)



¿Cómo organizar un Unit Test?

La ejecución de un test de unidad puede pensarse como tres fases secuenciales

- **Setup**: responsable de la inicialización de las variables (datos de test).
- **Exercise**: es la ejecución propiamente dicha de los casos de test.
- **Check**: es la constatación de los resultados del test

Existe una cuarta fase, denominada **teardown**, en la cual se deshace todo aquello que el test haya modificado (para respetar el principio de que cada test de unidad debe ser independiente).

¿Cómo organizar un Unit Test?

Ventajas de la organización

- No es una buena práctica (se confunden las etapas)

```
1 TEST(FactorialTest, negative){
2         EXPECT_EQ(1, factorial(−5));
3 }
```

- Es una buena práctica (se explicitan las etapas)

```
1 TEST(FactorialTest, negative){
2         //Setup
3         int a = −5;
4         //Exercise
5         int retVal = factorial(a);
6         //Check
7         EXPECT_EQ(1, retVal);
8 }
```

Cubrimientos

- Test Case #1: valorPrimo
  - Entrada: $n = 3$
  - Salida esperada: *result* = *true*

```
1 TEST(TestSuiteEsPrimo, valorPrimo) {
2   // setup
3   int n =3;
4   // exercise
5   bool result = esPrimo(n);
6   // check
7   EXPECT_TRUE(result);
8 }
```

Cubrimientos

- Test Case #2: valorNoPrimo
  - Entrada: $n = 4$
  - Salida esperada: *result* = *false*

```
1 TEST(TestSuiteEsPrimo, valorNoPrimo) {
2   // Setup
3   int n =4;
4   // Exercise
5   bool result = esPrimo(n);
6   // Check
7   EXPECT_FALSE(result);
8 }
```

Bibliografía

- David Gries - The Science of Programming
  - Chapter 22 - Notes on Documentation
    - Chapter 22.1 - Indentation
    - Chapter 22.2 - Definitions and Declarations of Variables
- Pezze, Young - Software Testing and Analysis
  - Chapter 1 - Software Test and Analysis in a Nutshell
  - Chapter 12 - Structural Testing
- Google Inc. - Google Test Primer documentation
  - https://github.com/google/googletest/blob/master/googletest/docs/Primer.md