

# Calidad, Pruebas y Mantenimiento del Software.

## Continuous Assessment Laboratory 1

Ana Cortés Cercadillo<sup>1</sup>, Carlos Javier Hellín Asensio<sup>2</sup>, and Daniel Ferreiro Rodríguez<sup>3</sup>

<sup>1</sup>a.cortesc@edu.uah.es, <sup>2</sup>carlos.hellin@edu.uah.es, <sup>3</sup>daniel.ferreiror@edu.uah.es

3 de Abril, 2022

## Contents

<b>1</b>	<b>Ejercicio 1</b>	<b>1</b>
1.1	Uso de Maven como gestión de proyectos . . . . .	1
1.2	Ejecución de los casos de prueba . . . . .	2
1.3	Medición de la cobertura . . . . .	4
1.4	Integración continua . . . . .	5
1.5	Replicación con Docker . . . . .	8
<b>2</b>	<b>Ejercicio 2</b>	<b>9</b>
<b>3</b>	<b>Trabajo común</b>	<b>10</b>
3.1	Medición de cobertura de pruebas . . . . .	12
3.1.1	Herramienta Expcov . . . . .	12
3.1.2	Ejecución de los casos de prueba . . . . .	14
3.2	Casos de prueba adicionales . . . . .	17
3.2.1	Defecto en el código . . . . .	22
3.3	Cálculo de la métrica de McCabe . . . . .	35

## 1 Ejercicio 1

Para este ejercicio se va a usar jMetal como programa Java de software libre disponible en GitHub. Se puede descargar desde su repositorio [7] en formato .zip o con el siguiente comando de git:

```
git clone https://github.com/jMetal/jMetal.git
```

### 1.1 Uso de Maven como gestión de proyectos

jMetal está dividido en subproyectos, de los cuales cada uno de ellos tiene un archivo pom.xml [4] como se puede ver a continuación.

```

ls jmetal-*
jmetal-algorithm:
pom.xml  src/

jmetal-core:
pom.xml  src/

jmetal-example:
pom.xml  src/

jmetal-experimental:
pom.xml  src/

jmetal-lab:
pom.xml  src/

jmetal-parallel:
pom.xml  src/

jmetal-problem:
pom.xml  src/

```

Por lo tanto utiliza Maven [1] como gestión de proyectos. Esto no solo permite facilitar la ejecución de los casos de prueba como se verá a continuación, sino que también compilar e instalar con un solo comando:

```
mvn install
```

También hay que tener en cuenta que, al seguir esta estructura de subproyectos, se va a tener, después de compilar, una carpeta para cada uno de los subproyectos con el resultado de la compilación. Esto mismo ocurrirá cuando se haga la ejecución de los tests.

## 1.2 Ejecución de los casos de prueba

Una vez descargado el proyecto de GitHub, se puede ver dónde están almacenados estos tests. Siguiendo la estructura anteriormente explicada, se puede comprobar que en cualquiera de esas carpetas de subproyectos, en la ruta *src/test/* se encuentran los ficheros .java con los tests. Por ejemplo, el fichero *ABYSSTest.java* almacenado en *jmetal-algorithm/src/test/java/org/uma/jmetal/algorithm/multiobjective/abyss/*, que importa JUnit para su uso en los tests:

```

1 package org.uma.jmetal.algorithm.multiobjective.abyss;
2
3 import static junit.framework.TestCase.assertFalse;
4 import static org.junit.Assert.assertEquals;
5 import static org.junit.Assert.assertTrue;
6
7 import java.util.ArrayList;
8 import java.util.List;
9 import org.junit.Before;

```

```

10 import org.junit.Test;
11 import org.springframework.test.util.ReflectionTestUtils;
12 import org.uma.jmetal.operator.crossover.impl.SBXCrossover;
13 import org.uma.jmetal.operator.localsearch.
    LocalSearchOperator;
14 import org.uma.jmetal.operator.localsearch.impl.
    BasicLocalSearch;
15 import org.uma.jmetal.operator.mutation.MutationOperator;
16 import org.uma.jmetal.operator.mutation.impl.
    PolynomialMutation;
17 import org.uma.jmetal.problem.doubleproblem.DoubleProblem;
18 import org.uma.jmetal.problem.doubleproblem.impl.
    AbstractDoubleProblem;
19 import org.uma.jmetal.solution.doublesolution.DoubleSolution
    ;
20 import org.uma.jmetal.util.archive.Archive;
21 import org.uma.jmetal.util.archive.impl.
    CrowdingDistanceArchive;
22 import org.uma.jmetal.util.comparator.DominanceComparator;
23 import org.uma.jmetal.util.pseudorandom.JMetalRandom;
24
25 /** Created by ajnebro on 11/6/15. */
26 public class ABYSSTest {
27     DoubleProblem problem;
28     LocalSearchOperator<DoubleSolution> localSearch;
29     MutationOperator<DoubleSolution> mutation;
30     Archive<DoubleSolution> archive;
31
32     ....

```

Ya visto dónde se localizan los tests, se procede a realizar su ejecución de los casos de prueba. Como se utiliza Maven para la gestión del proyecto, esto facilita que con un solo comando se ejecuten los tests:

```
mvn test
```

Este comando va a compilar cada uno de los subproyectos, creando la carpeta *target* en su interior donde se almacenará todos los resultados de ejecutar dicho comando. Los informes de los tests para saber si, por ejemplo, se han producido errores, se almacenan en la carpeta *surefire-reports* y los tests compilados en *.class* se encuentran en la carpeta *test-classes*.

También durante la ejecución de este comando, se va mostrando los resultados por pantalla que van dando los tests. Por ejemplo, para el proyecto *jmetal-algorithm* se muestra una de las posibles salidas:

```

-----
T E S T S
-----
Running org.uma.jmetal.algorithm.multiobjective.abyss.ABYSSTest
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.441 sec
Running org.uma.jmetal.algorithm.multiobjective.nsgaii.NSGAIIBuilderTest
2022-03-22 19:26:40.732 INFO: Loggers configured with null [org.uma.jmetal.util.
    JMetalLogger configureLoggers]

```

```

2022-03-22 19:26:40.765 INFO: Number of cores: 2 [org.uma.jmetal.util.evaluator.
    impl.MultiThreadedSolutionListEvaluator <init>]
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.431 sec
Running org.uma.jmetal.algorithm.multiobjective.pesa2.AdaptiveGridArchiveTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec
Running org.uma.jmetal.algorithm.multiobjective.pesa2.PESA2BuilderTest
2022-03-22 19:26:40.786 INFO: Number of cores: 2 [org.uma.jmetal.util.evaluator.
    impl.MultiThreadedSolutionListEvaluator <init>]
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 sec

```

Results :

Tests run: 26, Failures: 0, Errors: 0, Skipped: 0

Se puede ver cada uno de los tests que se van ejecutando, como ABYSSTest, NSGAIIBuilderTest, etc. Cada uno de ellos muestra el número de tests que se han ejecutado (*Tests run*), los que han fallado (*Failures*), los que han dado errores (*Errors*), los que se han saltado (*Skipped*) y, por último, el tiempo que ha tardado en segundos para ejecutar los tests (*Time elapsed*). Al final, se muestra en *Results* la suma total de estos valores de cada uno de los tests.

### 1.3 Medición de la cobertura

Como jMetal ya tiene integrada con Maven una herramienta para medir la cobertura, se va a aprovechar para usarla. Dicha herramienta es *jacoco* [5] y, con el uso de Maven, se puede ejecutar con el siguiente comando:

```
mvn test jacoco:report
```

Este comando creará la carpeta *site* dentro de *target* para cada uno de los subproyectos. En su interior se encuentran ficheros .html, por lo que si se abre el fichero *index.html* con un navegador web ya se acceden a los resultados de la cobertura en dicho subproyecto. Por ejemplo, para el caso de jmetal-algorithm se muestra a continuación una parte de estos resultados:

#### org.uma.jmetal:jmetal-algorithm

Element	Missed Instructions	Cov	Missed Branches	Cov	Missed	Cnty	Missed	Lines	Missed	Methods	Missed	Classes
org.uma.jmetal.algorithm.multiobjective.mopad	<div></div>	0 %	<div></div>	0 %	465	465	1 427	1 427	105	105	11	11
org.uma.jmetal.algorithm.multiobjective.cdq	<div></div>	0 %	<div></div>	0 %	264	264	637	637	79	79	7	7
org.uma.jmetal.algorithm.multiobjective.microfame.util	<div></div>	0 %	<div></div>	0 %	107	107	376	376	31	31	5	5
org.uma.jmetal.algorithm.multiobjective.smpso	<div></div>	0 %	<div></div>	0 %	162	162	416	416	103	103	5	5
org.uma.jmetal.algorithm.singleobjective.evolutionstrategy	<div></div>	0 %	<div></div>	0 %	117	117	293	293	63	63	6	6
org.uma.jmetal.algorithm.singleobjective.dmpso	<div></div>	0 %	<div></div>	0 %	116	116	332	332	72	72	3	3
org.uma.jmetal.algorithm.multiobjective.mombi.util	<div></div>	0 %	<div></div>	0 %	124	124	301	301	54	54	12	12
org.uma.jmetal.algorithm.singleobjective.particleswarmoptimization	<div></div>	0 %	<div></div>	0 %	93	93	233	233	42	42	2	2
org.uma.jmetal.algorithm.multiobjective.espea.util	<div></div>	0 %	<div></div>	0 %	109	109	268	268	41	41	8	8
org.uma.jmetal.algorithm.multiobjective.nsgaii.util	<div></div>	0 %	<div></div>	0 %	93	93	206	206	40	40	3	3
org.uma.jmetal.algorithm.singleobjective.evolutionstrategy.util	<div></div>	0 %	<div></div>	0 %	59	59	182	182	12	12	1	1
org.uma.jmetal.algorithm.multiobjective.nsgaii	<div></div>	17 %	<div></div>	13 %	81	98	227	275	58	72	6	9

Figura 1: Parte de la cobertura de jmetal-algorithm

Aquí se pueden identificar varias columnas:

- **Element:** el elemento que se analiza su cobertura (puede ser un paquete de Java, clase o método).
- **Missed Instructions:** instrucciones por las que los tests no han pasado.

- **Cov.:** el porcentaje de cobertura.
- **Missed Branches:** las ramas por la que los tests no han pasado.
- **Missed:** dependiendo de si la columna que está al lado es Ctxty, Lines, Methods o Classes se refiere a cada uno de ellos indicando el número por las que no han pasado los tests.
- **Cxty:** Complejidad ciclomática.
- **Lines:** Cantidad de líneas que tiene código.
- **Methods:** Cantidad de métodos que tiene el código.
- **Classes:** Cantidad de clases que tiene el código.

Se puede llegar a la conclusión con la Figura 1 que, mientras hay varios tests en el subproyecto jmetal-algorithm y no se han producido ningún error, a la hora de realizar el análisis de cobertura no se cubre lo suficiente, a lo mucho el paquete nsgaii con un 17% de cobertura en instrucciones. En los distintos subproyectos esto se mantiene, hay tests que pasan sin errores pero la cobertura es mínima.

Siguiendo con el uso de la herramienta *jacoco*, se puede obtener más información específica si se hace click al nombre de un paquete cualquiera, de la columna Element. Por ejemplo, en la Figura 2 se muestra la cobertura por clases y si se vuelve a hacer click, ahora se ve la cobertura por métodos como se puede observar en la Figura 3. Y ya por último, tenemos una visualización más clara dentro del propio código, como se puede ver en la Figura 4 dando más información por las ramas que no han pasado los tests con el uso de los rombos rojos. Todas estas nuevas tablas mantienen las mismas columnas explicadas anteriormente.

#### org.uma.jmetal.algorithm.multiobjective.moead

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MOEADD	<div><div></div></div>	0 %	<div><div></div></div>	0 %	260	260	756	756	28	28	1	1
MOEADSTM	<div><div></div></div>	0 %	<div><div></div></div>	0 %	55	55	187	187	15	15	1	1
AbstractMOEAD	<div><div></div></div>	0 %	<div><div></div></div>	0 %	38	38	143	143	9	9	1	1
MOEADIEpsilon	<div><div></div></div>	0 %	<div><div></div></div>	0 %	33	33	111	111	8	8	1	1
MOEADDRA	<div><div></div></div>	0 %	<div><div></div></div>	0 %	20	20	72	72	7	7	1	1
MOEADBuilder	<div><div></div></div>	0 %	<div><div></div></div>	0 %	30	30	62	62	24	24	1	1
ConstraintMOEAD	<div><div></div></div>	0 %	<div><div></div></div>	0 %	18	18	61	61	6	6	1	1
MOEAD	<div><div></div></div>	0 %	<div><div></div></div>	0 %	8	8	32	32	5	5	1	1
MOEADBuilder_Variant	<div><div></div></div>	0 %	<div><div></div></div>	n/a	1	1	1	1	1	1	1	1
AbstractMOEAD_FunctionType	<div><div></div></div>	0 %	<div><div></div></div>	n/a	1	1	1	1	1	1	1	1
AbstractMOEAD_NeighborType	<div><div></div></div>	0 %	<div><div></div></div>	n/a	1	1	1	1	1	1	1	1
Total	7.607 of 7.607	0 %	714 of 714	0 %	465	465	1.427	1.427	105	105	11	11

Figura 2: Cobertura de las clases en jmetal-algorithm

## 1.4 Integración continua

Como JMetal ya tiene su propio repositorio de GitHub, está ya habilitado desde ahí el uso de la integración continua permitiendo así compilar y ejecutar las pruebas de forma automática. Para ello, existe un fichero en *.github/workflows* llamado *integration-test.yml* cuyo contenido es el siguiente:

## MOEADD















Element	Missed Instructions	Cov	Missed Branches	Cov	Missed Ctxy	Missed Lines	Missed Methods
• <a href="#">nondominated_sorting_add(DoubleSolution)</a>		0 %		0 %	56	56	170
• <a href="#">updateArchive(DoubleSolution)</a>		0 %		0 %	35	35	112
• <a href="#">deleteCrowdRegion2(DoubleSolution_int)</a>		0 %		0 %	23	23	76
• <a href="#">matingSelection(int_int)</a>		0 %		0 %	20	20	58
• <a href="#">deleteCrowdRegion1(DoubleSolution_int)</a>		0 %		0 %	14	14	56
• <a href="#">nondominated_sorting_delete(DoubleSolution)</a>		0 %		0 %	23	23	50
• <a href="#">deleteRankOne(DoubleSolution_int)</a>		0 %		0 %	21	21	55

Figura 3: Cobertura de los métodos en jmetal-algorithm

```

497.     public int nondominated_sorting_add(S indiv) {
498.
499.         int flag = 0;
500.         int flag1, flag2, flag3;
501.
502.         // count the number of non-domination levels
503.         int num_ranks = 0;
504.         ArrayList<Integer> frontSize = new ArrayList<>();
505.         for (int i = 0; i < populationSize; i++) {
506.             int rankCount = countRankOnes(i);
507.             if (rankCount != 0) {
508.                 frontSize.add(rankCount);
509.                 num_ranks++;
510.             } else {
511.                 break;
512.             }
513.         }
514.     }

```

Figura 4: Cobertura del código en jmetal-algorithm

```

1  name: Integration Test
2
3  on:
4    push:
5      branches: [ master ]
6    pull_request:
7      branches: [ master ]
8
9  jobs:
10   build:
11
12     runs-on: ubuntu-latest
13
14     steps:
15     - uses: actions/checkout@v2
16     - name: Set up JDK 13
17       uses: actions/setup-java@v2
18       with:
19         java-version: '13'
20         distribution: 'adopt'
21         cache: maven
22     - name: Build with Maven
23       run: mvn -B integration-test --file pom.xml

```

Usando GitHub Actions [3], se establece con este fichero que cada vez que haya un push o pull request en la rama master (la parte de *on*), se realice la compilación y ejecución de las pruebas de integración desde una máquina virtual con Ubuntu instalado (la parte en *jobs*).

De esta forma, si se ven las Actions en el repositorio de GitHub de jMetal, se encuentran las ejecuciones de las pruebas, que hacen lo mismo como se ha explicado en la subsección 1.2, pero estas son automáticas y se realizan, como se ha explicado antes, cada vez que hay un push o pull request en la rama master, de forma que se hacen pruebas nada más subir nuevos cambios para así detectar fallos cuanto antes. Un ejemplo de ejecución automática de las pruebas sería en la siguiente Figura 5, donde se puede ver que ha pasado con éxito.

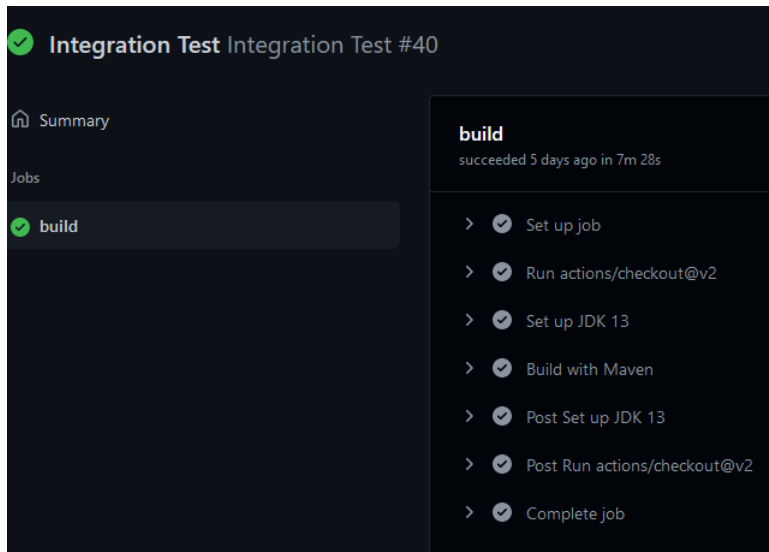


Figura 5: Ejecución automática de pruebas

## 1.5 Replicación con Docker

Una de las ventajas que ofrece la replicación con Docker es poder tener una imagen del entorno de pruebas. Cuando se tiene esta imagen creada, se puede lanzar varias instancias para realizar las pruebas o compartir la imagen con otros.

Para ello, se instala Docker [2] y se crea primero un fichero llamado *Dockerfile*, cuyo contenido es el siguiente:

```
FROM alpine

RUN apk add --no-cache git openjdk13 maven

RUN git clone https://github.com/jMetal/jMetal
WORKDIR /jMetal

CMD ["mvn", "test"]
```

Se utilice Alpine como distribución de Linux, ya que es bastante ligera. También se instala git, openjdk13 y maven. Lo justo y necesario para poder ejecutar los tests de jMetal. Después se realiza la clonación del repositorio de jMetal en GitHub y se ejecuta los tests con maven, como ya se explicó en anteriores subsecciones.

Una vez creado este fichero, se abre un terminal en la ubicación del Dockerfile y se ejecuta el siguiente comando para construir la imagen:

```
docker build -t jmetal .
```

Por último, se ejecuta la imagen creada con la siguiente instrucción y empezará a realizar los tests dentro del contenedor Docker.



```
docker run jmetal
```

Una vez finalizado, se podrá ver por pantalla que ha tenido éxito la compilación y los distintos tests.

```
INFO] Reactor Summary for org.uma.jmetal:jmetal 5.12-SNAPSHOT:
INFO]
INFO] org.uma.jmetal:jmetal ..... SUCCESS [ 4.357 s]
INFO] org.uma.jmetal:jmetal-core ..... SUCCESS [ 19.636 s]
INFO] org.uma.jmetal:jmetal-problem ..... SUCCESS [ 2.535 s]
INFO] org.uma.jmetal:jmetal-algorithm ..... SUCCESS [ 3.546 s]
INFO] org.uma.jmetal:jmetal-lab ..... SUCCESS [ 4.038 s]
INFO] org.uma.jmetal:jmetal-example ..... SUCCESS [ 0.637 s]
INFO] org.uma.jmetal:jmetal-experimental ..... SUCCESS [ 1.927 s]
INFO] org.uma.jmetal:jmetal-parallel ..... SUCCESS [ 11.781 s]
INFO] -----
INFO] BUILD SUCCESS
INFO] -----
INFO] Total time: 56.117 s
INFO] Finished at: 2022-04-02T21:24:10Z
INFO] -----
```

Figura 6: Salida final de ejecutar las pruebas en Docker

## 2 Ejercicio 2

Para este ejercicio se ha elegido la herramienta *EvoSuite* que genera automáticamente pruebas para proyectos hechos en Java. Junto a este documento, se incluye un vídeo tutorial donde se muestra su funcionamiento con una pequeña demo y un manual de uso de la herramienta.

### 3 Trabajo común

En esta parte de la práctica se toma un programa que determina, dadas las longitudes de sus 3 lados, el tipo de triángulo formado según sus lados, indicando en su caso si los datos dados realmente no conforman un triángulo. El código que se utilizará es este:

```
1  #include <math.h>
2  #include <stdio.h>
3
4  int checktriangle(int a, int b, int c)
5  {
6      int A, B, C;
7      if((c<a+b)&&(b<a+c)&&(a<c+b))
8      {
9          if(a==b && b==c)
10         {
11             printf("Equilateral Triangle \n");
12         }else if(a==b || b==c || a==c)
13         {
14             printf("Isosceles Triangle \n");
15         }else
16         {
17             printf("Scalene triangle \n");
18         }
19
20         int value = (a*a)+(b*b)-(c*c);
21
22         if(value==0)
23         {
24             printf("Right Triangle \n");
25         }else if(value>0)
26         {
27             printf("Acute Triangle \n");
28         }else
29         {
30             printf("Obtuse Triangle \n");
31         }
32     }
33     else
34     {
35         printf("Is not a Triangle \n");
36     }
37
38     return 0;
39 }
40
41 int main()
42 {
43     int a, b, c, i, n;
44     printf("Number of test cases: \n");
45     scanf("%d",&n); printf ("%d", n);
46 }
```

```

47     for (i = 1; i < n+1; i++)
48     {
49         printf("Set the values of the triangle sides (one per
           line): \n");
50         scanf("%d",&a); printf ("%d,", a);
51         scanf("%d",&b); printf ("%d,", b);
52         scanf("%d",&c); printf ("%d: ", c);
53         if(a>b && a>c)
54         {
55             int aux = c;
56             c=a;
57             a=aux;
58         }
59         else if(b>a && b>c)
60         {
61             int aux = c;
62             c=b;
63             b=aux;
64         }
65
66         checktriangle(a,b,c);
67     };
68
69     return 0;
70 }

```

Listing 1: Código

## 3.1 Medición de cobertura de pruebas

### 3.1.1 Herramienta Expcov

El código se prueba usando la herramienta web para medir la cobertura Expcov (<https://www.expcov.com/cgi-bin/expcov/demo.cgi>). De forma resumida, en la web encontramos cinco elementos importantes marcados en la Figura 7:

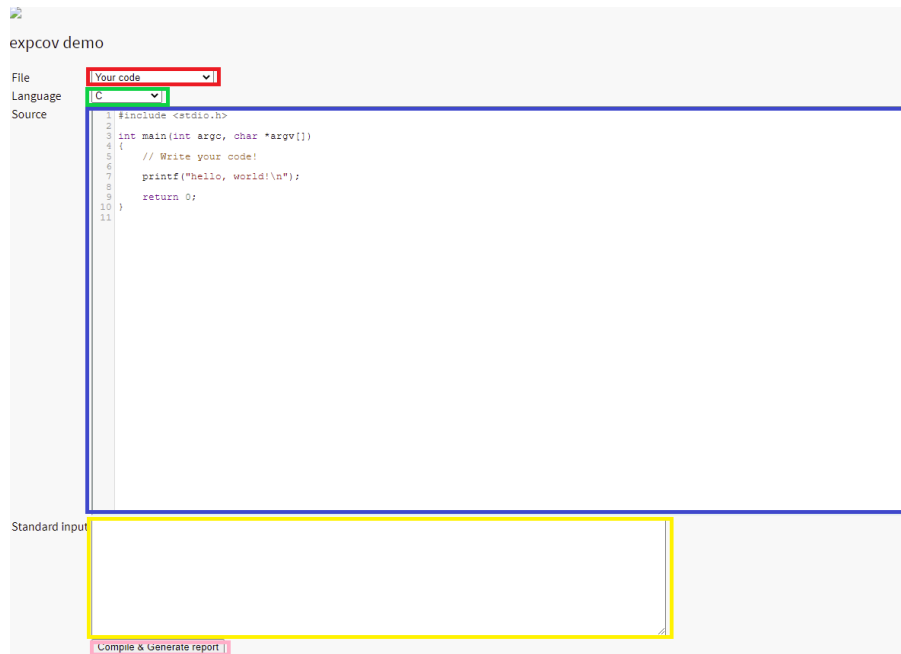


Figura 7: Interfaz de la herramienta.

1. **File**: tiene la opción de elegir un código de demostración de cualquiera de los tres lenguajes soportados o de introducir tu propio código.
2. **Lenguaje**: permite los lenguajes de programación C, C+ y Objective C.
3. **Source**: aquí se añade el código que se quiere probar.
4. **Standard input**: los casos de prueba con los que se quiere calcular la cobertura se añaden en esta zona.
5. **Compile & Generate report**: botón para generar el informe según los datos introducidos.

Esta herramienta crea un informe sobre la cobertura del código según los casos de prueba introducidos. Este informe detalla tres coberturas distintas: de funciones (Functions), de sentencias (Expressions) y de decisiones (Branches:if). Esta cobertura se muestra en forma de porcentaje.

Estos porcentajes salen de la siguiente división  $Cubierto/Total$ , que es lo que viene detallado después de mostrar el porcentaje en la Figura 8. En el caso de las funciones y de las expresiones se sigue esa nomenclatura, pero en el caso de las decisiones aparecen tres números después del porcentaje. Esto se debe

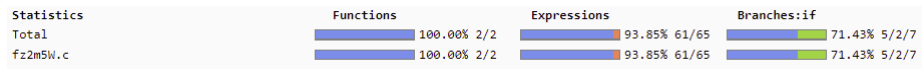


Figura 8: Cobertura de ejemplo obtenida.

a que se representan más datos, dependiendo de si se ha cubierto la opción de *True* y *False* o no se ha cubierto ni *True* ni *False*. El patrón que sigue es el siguiente *Cubierto/NoCubierto/Total*.

También se puede obtener un análisis más extenso como el de la Figura 9 pulsando en la zona de Statistics el nombre del fichero .c que en la Figura 8 aparece.

	Pass	Half	Fail	Excluded	Total
<b>Function</b> <input type="checkbox"/>	1		0	0	1
	100.00%		0.00%	0.00%	100%
<b>Expressions</b> <input type="checkbox"/>	2		0	0	2
	100.00%		0.00%	0.00%	100%
<b>Conditions</b> <input type="checkbox"/>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%
<b>MC/DC</b> <input type="checkbox"/>	0		0	0	0
	0.00%		0.00%	0.00%	100%
<b>Branches</b>					
if <input type="checkbox"/>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%
for <input type="checkbox"/>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%
while <input type="checkbox"/>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%
case <input type="checkbox"/>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%

Figura 9: Cobertura extendida de ejemplo.

La información que encontramos aquí son nuevamente las funciones (Functions) y expresiones (Expressions), las condiciones (Conditions), la cobertura de condición/decisión (MC/DC) y la cobertura de los diferentes tipos de ramas (Branches) que son if, for, while y case. De cada uno de ellos se muestra la cantidad y porcentaje pasados totalmente (Pass), pasados a la mitad (Half), no cubiertos (Fail), excluidos (Excluded) y totales (Total).

### 3.1.2 Ejecución de los casos de prueba

Una vez vista y entendida la herramienta que se va a usar se procede a ejecutar los casos de prueba propuestos. Los valores de entrada con los siguientes:

- Caso 1: 6, 5, 10
- Caso 2: 3, 3, 3
- Caso 3: 4, 4, 5
- Caso 4: 7, 2, 3
- Caso 5: 4, 3, 0

Añadimos el código y las entradas en las zonas correspondientes. El formato de la entrada es el número de casos que se van a introducir y cada uno de los valores a continuación, en líneas distintas como se muestra en la Figura 10.

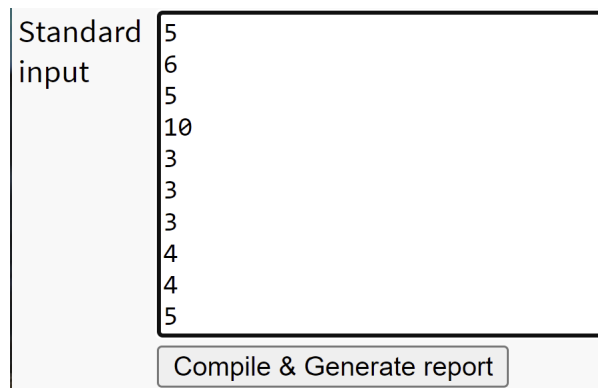


Figura 10: Casos de entrada propuestos.

Se ejecutan estos casos de prueba y, como se ha explicado anteriormente, se genera un informe con la cobertura del código conseguida. Esta cobertura es la de la Figura 11.

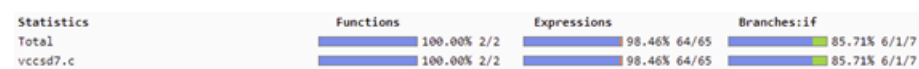


Figura 11: Cobertura de los casos propuestos.

En este caso obtenemos una cobertura en cuanto a funciones del 100% ya que de 2 funciones que hay, *checktriangle()* y *main()*, se ejecutan ambas. En cuanto a la cobertura de expresiones, se obtiene un 98,85%, es decir, de 65 expresiones se han cubierto 61. Finalmente, la cobertura de decisiones es del 71,43% con 5 decisiones cubiertas, 2 decisiones no cubiertas y 7 decisiones totales.

Para más detalle, se obtiene la información anterior pero con la cobertura extendida de la Figura 12.

En ella afirmamos la cobertura completa (100%) en cuanto a funciones como se ha dicho antes; 64 expresiones cubiertas (98,46%) pero 1 sin cubrir (1,54%) de las 65 totales; 16 condiciones pasadas totalmente (72,73%) y 6 cubiertas solo

	Pass	Half	Fail	Excluded	Total
Function	2 100.00%		0 0.00%	0 0.00%	2 100%
Expressions	64 98.46%		1 1.54%	0 0.00%	65 100%
Conditions	16 72.73%	6 27.27%	0 0.00%	0 0.00%	22 100%
MC/DC	7 58.33%		5 41.67%	0 0.00%	12 100%
Branches					
if	6 85.71%	1 14.29%	0 0.00%	0 0.00%	7 100%
for	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 12: Cobertura extendida de los casos propuestos.

a la mitad (27,27%) de las 22 totales; 7 MC/DC cubiertos (58,22%) y 5 sin cubrir (41,67%) de los 12 totales; 6 ramas if cubiertas totalmente (85,71%) y 1 cubierta a la mitad (14,29%) de las 7 totales; 1 rama for cubierta totalmente (100%) de 1 rama en total; en este código no aparecen ninguna rama while ni case.

La salida obtenida por la consola es la que se muestra en la Figura 13.

```

Number of test cases:
5Set the values of the triangle sides (one per line):
6,5,10: Scalene triangle
Obtuse Triangle
Set the values of the triangle sides (one per line):
3,3,3: Equilateral Triangle
Acute Triangle
Set the values of the triangle sides (one per line):
4,5,4: Isosceles Triangle
Acute Triangle
Set the values of the triangle sides (one per line):
7,2,3: Is not a Triangle
Set the values of the triangle sides (one per line):
4,3,0: Is not a Triangle

```

Figura 13: Salida obtenida de la ejecución.

Para observar mejor los casos que se proponen se crea la Tabla 1 en la que se valorará si las salidas esperadas y las verdaderas salidas coinciden y pasan la prueba o, por el contrario, no lo hacen y fallan la prueba.

No hay ningún error y todas las salidas son las previstas.

Casos de prueba	Valores de entrada	Salida esperada	Salida obtenida	Coinciden
Caso 1	(6, 5, 10)	Escaleno y Obtuso	Scalene triangle Obtuse Triangle	Sí
Caso 2	(3, 3, 3)	Equilátero y Agudo	Equilateral triangle Acute Triangle	Sí
Caso 3	(4, 5, 4)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 4	(7, 2, 3)	No triángulo	Is not a Triangle	Sí
Caso 5	(4, 3, 0)	No triángulo	Is not a Triangle	Sí

Tabla 1: Casos de prueba propuestos y sus salidas.



## 3.2 Casos de prueba adicionales

En esta parte se va a indicar exactamente qué opciones del código no se ejecutaron con las pruebas anteriores y, a continuación, se añadirán casos de entrada para tratar de conseguir el 100% de cobertura en decisiones y condiciones.

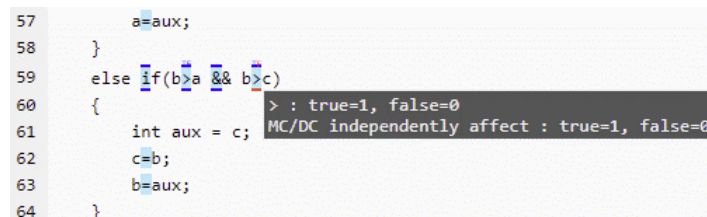
La herramienta utilizada añade el código junto con marcas de colores que indica por donde ha pasado la ejecución y por donde no. La marca azul significa que ha ejecutado esa orden sin problema y la marca roja quiere decir que no se ha ejecutado esa parte o que no se han probado todas las opciones que tiene de ejecutarse. Nos fijamos principalmente en las marcas rojas empezando con la función *main()* y luego la función *checktriangle()*.

En la función *main()* solo se encuentra una:

1. Figura 14: El motivo es que nunca se ha comprobado que la expresión  $b > c$  es falsa. Habrá que añadir un caso en el que el lado  $b$  no sea mayor que el lado  $c$ , más concretamente, que no se cumpla  $b > c$  pero se cumpla la primera condición  $b > a$ .

En la función *checktriangle()* se encuentran las siguientes marcas:

1. Figura 15: No se realiza una prueba en la que esa expresión sea falsa. Se añadirá un caso en el que el lado  $b$  sea mayor que la suma de  $a$  y  $c$  al mismo tiempo que se cumpla  $c < a + b$ .
2. Figura 16: De la misma forma ocurre con la siguiente expresión, nunca se comprueba que sea falsa. Se añadirá un caso en el que el lado  $a$  sea mayor que la suma de  $c$  y  $b$  cuando las expresiones booleanas anteriores del *if* sean verdaderas.
3. Figura 17: En este caso nunca se ha comprobado que la expresión  $b == c$  sea verdadera. Para que la ejecución llegue a esa expresión habrá que hacer  $a == b$  falso. Por lo tanto, la entrada debe tener el lado  $a$  distinto de  $b$  y el lado  $b$  igual al  $c$ .
4. Figura 18: El motivo es que nunca se ha comprobado que la expresión  $a == c$  es verdadera. Habrá que añadir un caso en el que el lado  $a$  sea igual que el lado  $c$  además de que  $b$  sea distinto de  $a$  y de  $c$ .
5. Figura 19: En este *if* no se llega a ejecutar en ningún momento la parte que hace verdadera a la expresión y, por lo tanto, el código de esta parte no se ha comprobado. Hay que añadir un caso para hacer que la variable *value* sea 0 en algún momento.



```
57     a=aux;
58 }
59 else if(b>a && b>c)
60 {
61     int aux = c;
62     c=b;
63     b=aux;
64 }
```

Figura 14: Decisión no cubierta función *main()* (1).

```

5 {
6     int A, B, C;
7     if((c<a+b)&&(b<a+c)&&(a<c+b))
8     {
9         if(a==b && b==c)
10    {

```

Figura 15: Decisión no cubierta función *checktriangle()* (1).

```

5 {
6     int A, B, C;
7     if((c<a+b)&&(b<a+c)&&(a<c+b))
8     {
9         if(a==b && b==c)
10    {

```

Figura 16: Decisión no cubierta función *checktriangle()* (2).

```

9     if(a==b && b==c)
10    {
11        printf("Equilateral Triangle \n");
12    }else if(a==b || b==c || a==c)
13    {
14        printf("Isosceles Triangle \n");
15    }else
16    {

```

Figura 17: Decisión no cubierta función *checktriangle()* (3).

```

9     if(a==b && b==c)
10    {
11        printf("Equilateral Triangle \n");
12    }else if(a==b || b==c || a==c)
13    {
14        printf("Isosceles Triangle \n");
15    }else
16    {

```

Figura 18: Decisión no cubierta función *checktriangle()* (4).

```

23     if(value==0)
24     {
25         printf("Right Triangle \n");
26     }else if(value>0)
27     {

```

Figura 19: Decisión no cubierta función *checktriangle()* (5).

En principio se observa que como máximo habrá que añadir 6 entradas más, 1 para la función *main()* y 5 para la función *checktriangle()*. Volvemos a la tabla de las entradas y salidas de la parte anterior para añadir nuevos casos que cubran los casos detallados anteriormente. Para ello se crea la Tabla 2. La salida obtenida por la consola es la que se muestra en la Figura 20.

```

Number of test cases:
8Set the values of the triangle sides (one per line):
6,5,10: Scalene triangle
Obtuse Triangle
Set the values of the triangle sides (one per line):
3,3,3: Equilateral Triangle
Acute Triangle
Set the values of the triangle sides (one per line):
4,5,4: Isosceles Triangle
Acute Triangle
Set the values of the triangle sides (one per line):
7,2,3: Is not a Triangle
Set the values of the triangle sides (one per line):
4,3,0: Is not a Triangle
Set the values of the triangle sides (one per line):
5,3,4: Scalene triangle
Right Triangle
Set the values of the triangle sides (one per line):
4,5,5: Isosceles Triangle
Acute Triangle
Set the values of the triangle sides (one per line):
5,4,5: Isosceles Triangle
Acute Triangle

```

Figura 20: Salida obtenida de la ejecución.

Añadiendo el Caso 6 se prueban los casos de las Figuras 14 y 19 detallados anteriormente. En la Figura 21 se muestra la cobertura después de añadir este caso de prueba a los anteriores y en la Figura 22 se extiende esa cobertura. Como se puede observar en la Figura 21, ya se ha alcanzado un 100% de cobertura tanto en funciones como en expresiones y en ramas if. En la cobertura extendida de la Figura 22 lo que cambió fue la de expresiones, llegando a 65 cubiertas completamente (100%) de 65 totales, la de condiciones, pasando a 18 cubiertas completamente (81,82%) y 4 cubiertas a la mitad (18,18%) de 22 totales, MC/DC, llegando a 8 cubiertas completamente (66,67%) y 4 no cubiertas (33,33%) de 12 totales, y la de ramas if, siendo ahora 7 cubiertas de 7 totales (100%).

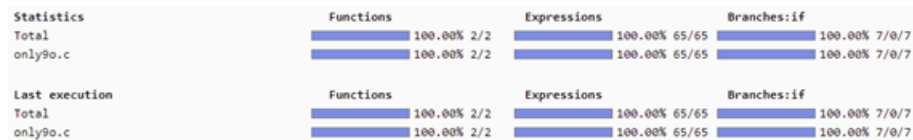


Figura 21: Cobertura después de añadir el Caso 6.

Casos de prueba	Valores de entrada	Salida esperada	Salida obtenida	Coinciden
Caso 1	(6, 5, 10)	Escaleno y Obtuso	Scalene triangle Obtuse Triangle	Sí
Caso 2	(3, 3, 3)	Equilátero y Agudo	Equilateral triangle Acute Triangle	Sí
Caso 3	(4, 5, 4)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 4	(7, 2, 3)	No triángulo	Is not a Triangle	Sí
Caso 5	(4, 3, 0)	No triángulo	Is not a Triangle	Sí
<b>Caso 6</b>	<b>(5, 3, 4)</b>	<b>Escaleno y Recto</b>	<b>Scalene triangle Right Triangle</b>	<b>Sí</b>
<b>Caso 7</b>	<b>(4, 5, 5)</b>	<b>Isósceles y Agudo</b>	<b>Isosceles triangle Acute Triangle</b>	<b>Sí</b>
<b>Caso 8</b>	<b>(5, 4, 5)</b>	<b>Isósceles y Agudo</b>	<b>Isosceles triangle Acute Triangle</b>	<b>Sí</b>

Tabla 2: Casos de prueba añadidos y sus salidas.

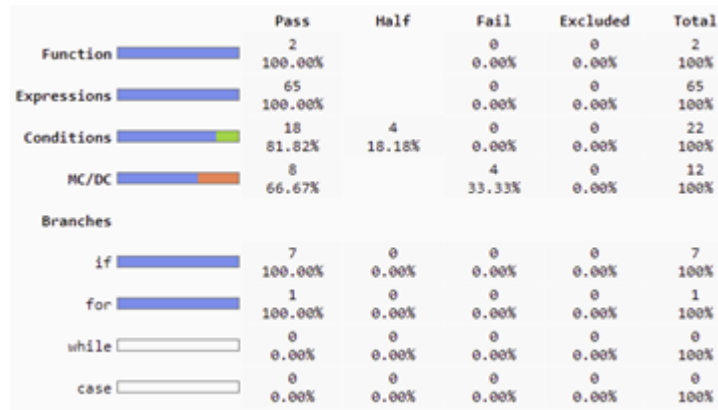


Figura 22: Cobertura extendida después de añadir el Caso 6.

Añadiendo el Caso 7 evitamos la marca roja de la Figura 17 convirtiéndola en azul. Los porcentajes de Conditions y MC/DC de la cobertura detallada también se actualizan como se muestra en la Figura 23. Las condiciones pasan a ser 19 cubiertas completamente (86,36%) y 3 cubiertas a la mitad (13,64%) de las 22 totales y las condiciones/decisiones avanzan hasta 9 cubiertas (75%) y 3 no cubiertas (25%) de 12 totales.

Añadiendo el Caso 8 ocurre lo mismo que con el caso anterior pero afectando a lo visto en la Figura 18. La cobertura actualizada es la de la Figura 24. Se vuelven a aumentar la cobertura de condiciones a 20 cubiertas completamente (90,91%) y 2 cubiertas a la mitad (9,09%) de las 22 totales y la MC/DC hasta 10 cubiertas (83,33%) y 2 no cubiertas (16,67%) de 12 totales.









	Pass	Half	Fail	Excluded	Total
Function 	2 100.00%		0 0.00%	0 0.00%	2 100%
Expressions 	65 100.00%		0 0.00%	0 0.00%	65 100%
Conditions 	19 86.36%	3 13.64%	0 0.00%	0 0.00%	22 100%
MC/DC 	9 75.00%		3 25.00%	0 0.00%	12 100%
Branches					
if 	7 100.00%	0 0.00%	0 0.00%	0 0.00%	7 100%
for 	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 23: Cobertura extendida después de añadir el Caso 7.



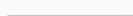





	Pass	Half	Fail	Excluded	Total
Function 	2 100.00%		0 0.00%	0 0.00%	2 100%
Expressions 	65 100.00%		0 0.00%	0 0.00%	65 100%
Conditions 	20 90.91%	2 9.09%	0 0.00%	0 0.00%	22 100%
MC/DC 	10 83.33%		2 16.67%	0 0.00%	12 100%
Branches					
if 	7 100.00%	0 0.00%	0 0.00%	0 0.00%	7 100%
for 	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 24: Cobertura extendida después de añadir el Caso 8.

### 3.2.1 Defecto en el código

En esta sección se va a señalar los defectos que se detectaron en el código con las pruebas, incluyendo explicación detallada sobre su detección, la naturaleza del defecto y su posible corrección.

Después de hacer un análisis del código, se llega a la conclusión de que no se pueden crear casos de prueba para completar los casos de las Figuras 15 y 16 no se llegan a ejecutar nunca debido a una zona de código que se ejecuta antes. La zona del conflicto que se localiza en la función *main()* es la siguiente:

```
1  if(a>b && a>c)
2  {
3      int aux = c;
4      c=a;
5      a=aux;
6  }
7  else if(b>a && b>c)
8  {
9      int aux = c;
10     c=b;
11     b=aux;
12 }
```

Listing 2: Código

Lo que se hace ahí es poner el lado de mayor tamaño en la variable *c*. Esto provoca que, después, al ejecutar la función *checktriangle()*, en la primera condición del *if* de la Figura 16,  $c < a + b$ , sí pueda darse la opción de que sea verdadera o falsa, pero el resto de condiciones, ya sea  $b < a + c$  o  $a < c + b$  no serán nunca falsas. Por ejemplo, en  $b < a + c$ , el lado *c* ya va a ser siempre mayor o igual que el lado *b*, por lo tanto, si se le suma otro número, tamaño del lado *a*, seguirá siendo mayor. Lo mismo ocurre con  $a < c + b$ .

Esta zona de código para lo que realmente es útil es para calcular la variable *value*, necesaria para obtener el mayor ángulo del triángulo, entonces se puede mover justo antes de este cálculo. El código con la modificación es el siguiente:

```
1  #include <math.h>
2  #include <stdio.h>
3
4  int checktriangle(int a, int b, int c)
5  {
6      int A, B, C;
7      if((c<a+b)&&(b<a+c)&&(a<c+b))
8      {
9          if(a==b && b==c)
10         {
11             printf("Equilateral Triangle \n");
12         }else if(a==b || b==c || a==c)
13         {
14             printf("Isosceles Triangle \n");
15         }else
16         {
17             }
```

```

18         printf("Scalene triangle \n");
19     }
20
21     if(a>b && a>c)
22     {
23         int aux = c;
24         c=a;
25         a=aux;
26     }
27     else if(b>a && b>c)
28     {
29         int aux = c;
30         c=b;
31         b=aux;
32     }
33
34     int value = (a*a)+(b*b)-(c*c);
35
36     if(value==0)
37     {
38         printf("Right Triangle \n");
39     }else if(value>0)
40     {
41         printf("Acute Triangle \n");
42     }else
43     {
44         printf("Obtuse Triangle \n");
45     }
46 }
47 else
48 {
49     printf("Is not a Triangle \n");
50 }
51
52 return 0;
53 }
54
55 int main()
56 {
57     int a, b, c, i, n;
58     printf("Number of test cases: \n");
59     scanf("%d",&n); printf ("%d", n);
60     for (i = 1; i < n+1; i++)
61     {
62         printf("Set the values of the triangle sides (one per
63             line): \n");
64         scanf("%d",&a); printf ("%d,", a);
65         scanf("%d",&b); printf ("%d,", b);
66         scanf("%d",&c); printf ("%d: ", c);
67
68         checktriangle(a,b,c);
69     };
70     return 0;

```

## Listing 3: Código

La cobertura extendida obtenida con esta nueva modificación del código y los casos de prueba de la Tabla 2 es la de la Figura 25.

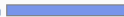



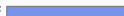
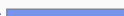


	Pass	Half	Fail	Excluded	Total
<b>Function</b> 	2 100.00%		0 0.00%	0 0.00%	2 100%
<b>Expressions</b> 	65 100.00%		0 0.00%	0 0.00%	65 100%
<b>Conditions</b> 	17 77.27%	5 22.73%	0 0.00%	0 0.00%	22 100%
<b>MC/DC</b> 	8 66.67%		4 33.33%	0 0.00%	12 100%
<b>Branches</b>					
if 	7 100.00%	0 0.00%	0 0.00%	0 0.00%	7 100%
for 	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 25: Cobertura extendida de la modificación.

Ahora se comprueba que partes del código no están cubiertas para añadir los nuevos casos de prueba necesarios. En la función *checktriangle()* encontramos las siguientes decisiones sin cubrir:

1. Figura 26: Hay que añadir un caso en el que la suma de los 2 primeros lados ( $a$  y  $b$ ) sea menor o igual que el lado  $c$ , es decir, que la condición  $c < a + b$  sea falsa.
2. Figura 27: En este caso se necesita que la operación and ( $\&\&$ ) entre las condiciones  $c < a + b$  y  $b < a + c$  sea falsa. Para ello se necesita que una de las dos condiciones sea verdadera y la otra se dé como falsa.
3. Figura 28: Para satisfacer este caso hay que conseguir que la suma de los lados  $a$  y  $c$  sea menor o igual que el lado  $b$ , es decir, que la condición  $b < a + c$  sea falsa.
4. Figura 29: En este caso se necesita que la condición  $b == c$  sea falsa en alguna ocasión y, para ello, la condición  $a == b$  tiene que darse como verdadera, por lo tanto, los lados  $a$  y  $b$  tienen que ser iguales pero el  $c$  distinto.
5. Figura 30: Hay que añadir un caso en el que los 2 primeros lados ( $a$  y  $b$ ) sean iguales entre sí, es decir, que la condición  $a == b$  sea verdadera.

Volvemos a la Tabla 2 de las entradas y salidas de la parte anterior para añadir nuevos casos que cubran los casos detallados anteriormente. Para ello se crea la Tabla 3. La salida obtenida por la consola es la que se muestra en la Figura 31.

Añadiendo el Caso 9 se soluciona la cobertura de las decisiones de las Figuras 26 y 27. Se actualiza la cobertura a la de la Figura 32.



```

6   int A, B, C;
7   if((c<a+b)&&(b<a+c)&&(a<c+b))
8   {
9       < : true=8, false=0
10      MC/DC independently affect : true=6, false=0
11  }

```

Figura 26: Decisión no cubierta función *checktriangle()* (1).

```

6   int A, B, C;
7   if((c<a+b)&&(b<a+c)&&(a<c+b))
8   {
9       && : true=8, false=0
10      if(a==b && b==c)
11      {

```

Figura 27: Decisión no cubierta función *checktriangle()* (2).

```

6   int A, B, C;
7   if((c<a+b)&&(b<a+c)&&(a<c+b))
8   {
9       < : true=8, false=0
10      if(a==b && MC/DC independently affect : true=6, false=0
11      {

```

Figura 28: Decisión no cubierta función *checktriangle()* (3).

```

8   {
9       if(a==b && b==c)
10      {
11          == : true=1, false=0
12          printf("E MC/DC independently affect : true=1, false=0
13      }else if(a==b || b==c || a==c)

```

Figura 29: Decisión no cubierta función *checktriangle()* (4).

```

11      printf("Equilateral Triangle \n");
12  }else if(a==b || b==c || a==c)
13  {
14      == : true=0, false=5
15      MC/DC independently affect : true=0, false=2
16      printf("Isosceles Triangle \n");

```

Figura 30: Decisión no cubierta función *checktriangle()* (5).

Number of test cases:  
11Set the values of the triangle sides (one per line):  
6,5,10: Scalene triangle  
Obtuse Triangle  
Set the values of the triangle sides (one per line):  
3,3,3: Equilateral Triangle  
Acute Triangle  
Set the values of the triangle sides (one per line):  
4,5,4: Isosceles Triangle  
Acute Triangle  
Set the values of the triangle sides (one per line):  
7,2,3: Is not a Triangle  
Set the values of the triangle sides (one per line):  
4,3,0: Is not a Triangle  
Set the values of the triangle sides (one per line):  
5,3,4: Scalene triangle  
Right Triangle  
Set the values of the triangle sides (one per line):  
4,5,5: Isosceles Triangle  
Acute Triangle  
Set the values of the triangle sides (one per line):  
5,4,5: Isosceles Triangle  
Acute Triangle  
Set the values of the triangle sides (one per line):  
1,1,4: Is not a Triangle  
Set the values of the triangle sides (one per line):  
1,4,1: Is not a Triangle  
Set the values of the triangle sides (one per line):  
4,4,5: Isosceles Triangle  
Acute Triangle

Figura 31: Salida obtenida de la ejecución.

	Pass	Half	Fail	Excluded	Total
<b>Function</b>	2		0	0	2
	100.00%		0.00%	0.00%	100%
<b>Expressions</b>	65		0	0	65
	100.00%		0.00%	0.00%	100%
<b>Conditions</b>	19	3	0	0	22
	86.36%	13.64%	0.00%	0.00%	100%
<b>MC/DC</b>	9		3	0	12
	75.00%		25.00%	0.00%	100%
<b>Branches</b>					
<b>if</b>	7	0	0	0	7
	100.00%	0.00%	0.00%	0.00%	100%
<b>for</b>	1	0	0	0	1
	100.00%	0.00%	0.00%	0.00%	100%
<b>while</b>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%
<b>case</b>	0	0	0	0	0
	0.00%	0.00%	0.00%	0.00%	100%

Figura 32: Cobertura extendida después de añadir el Caso 9.

Casos de prueba	Valores de entrada	Salida esperada	Salida obtenida	Coinciden
Caso 1	(6, 5, 10)	Escaleno y Obtuso	Scalene triangle Obtuse Triangle	Sí
Caso 2	(3, 3, 3)	Equilátero y Agudo	Equilateral triangle Acute Triangle	Sí
Caso 3	(4, 5, 4)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 4	(7, 2, 3)	No triángulo	Is not a Triangle	Sí
Caso 5	(4, 3, 0)	No triángulo	Is not a Triangle	Sí
Caso 6	(5, 3, 4)	Escaleno y Recto	Scalene triangle Right Triangle	Sí
Caso 7	(4, 5, 5)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 8	(5, 4, 5)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
<b>Caso 9</b>	<b>(1, 1, 4)</b>	<b>No triángulo</b>	<b>Is not a Triangle</b>	<b>Sí</b>
<b>Caso 10</b>	<b>(1, 4, 1)</b>	<b>No triángulo</b>	<b>Is not a Triangle</b>	<b>Sí</b>
<b>Caso 11</b>	<b>(4, 4, 5)</b>	<b>Isósceles y Agudo</b>	<b>Isosceles Triangle Acute Triangle</b>	<b>Sí</b>

Tabla 3: Casos de prueba añadidos y sus salidas.

Añadiendo el Caso 10 se soluciona la cobertura de la decisión de la Figura 28. Se actualiza la cobertura a la de la Figura 33.

Añadiendo el Caso 11 se soluciona la cobertura de las decisiones de las Figuras 29 y 30. Se actualiza la cobertura a la de la Figura 34.

Finalmente, conseguimos un 100% de cobertura en todos los aspectos realizando esta modificación en el código. Para terminar, hay un caso de los propuestos inicialmente que se pueden quitar de la tabla de pruebas y se seguiría obteniendo una cobertura total. Este caso es el Caso 4. Con esto, la tabla de casos de prueba final sería la Tabla 4 con 10 casos totales.









	Pass	Half	Fail	Excluded	Total
<b>Function</b> 	2 100.00%		0 0.00%	0 0.00%	2 100%
<b>Expressions</b> 	65 100.00%		0 0.00%	0 0.00%	65 100%
<b>Conditions</b> 	20 90.91%	2 9.09%	0 0.00%	0 0.00%	22 100%
<b>MC/DC</b> 	10 83.33%		2 16.67%	0 0.00%	12 100%
<b>Branches</b>					
if 	7 100.00%	0 0.00%	0 0.00%	0 0.00%	7 100%
for 	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 33: Cobertura extendida después de añadir el Caso 10.

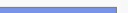


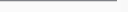




	Pass	Half	Fail	Excluded	Total
<b>Function</b> 	2 100.00%		0 0.00%	0 0.00%	2 100%
<b>Expressions</b> 	65 100.00%		0 0.00%	0 0.00%	65 100%
<b>Conditions</b> 	22 100.00%	0 0.00%	0 0.00%	0 0.00%	22 100%
<b>MC/DC</b> 	12 100.00%		0 0.00%	0 0.00%	12 100%
<b>Branches</b>					
if 	7 100.00%	0 0.00%	0 0.00%	0 0.00%	7 100%
for 	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 34: Cobertura extendida después de añadir el Caso 11.

Casos de prueba	Valores de entrada	Salida esperada	Salida obtenida	Coinciden
Caso 1	(6, 5, 10)	Escaleno y Obtuso	Scalene triangle Obtuse Triangle	Sí
Caso 2	(3, 3, 3)	Equilátero y Agudo	Equilateral triangle Acute Triangle	Sí
Caso 3	(4, 5, 4)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 4	(4, 3, 0)	No triángulo	Is not a Triangle	Sí
Caso 5	(5, 3, 4)	Escaleno y Recto	Scalene triangle Right Triangle	Sí
Caso 6	(4, 5, 5)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 7	(5, 4, 5)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 8	(1, 1, 4)	No triángulo	Is not a Triangle	Sí
Caso 9	(1, 4, 1)	No triángulo	Is not a Triangle	Sí
Caso 10	(4, 4, 5)	Isósceles y Agudo	Isosceles Triangle Acute Triangle	Sí

Tabla 4: Casos de prueba final y sus salidas.

Otra opción es no mover el lugar de la zona conflictiva y quitar las condiciones que siempre dan verdadero, ya que no interfieren para nada en el resultado booleano que evalúa el *if*. Con esta opción el código quedaría así:

```
1  #include <math.h>
2  #include <stdio.h>
3
4  int checktriangle(int a, int b, int c)
5  {
6      int A, B, C;
7      if(c<a+b)
8      {
9          if(a==b && b==c)
10         {
11             printf("Equilateral Triangle \n");
12         }else if(a==b || b==c || a==c)
13         {
14             printf("Isosceles Triangle \n");
15         }else
16         {
17             printf("Scalene triangle \n");
18         }
19
20         int value = (a*a)+(b*b)-(c*c);
21
22         if(value==0)
23         {
24             printf("Right Triangle \n");
25         }else if(value>0)
26         {
27             printf("Acute Triangle \n");
28         }else
29         {
30             printf("Obtuse Triangle \n");
31         }
32     }
33     else
34     {
35         printf("Is not a Triangle \n");
36     }
37
38     return 0;
39 }
40
41
42 int main()
43 {
44     int a, b, c, i, n;
45     printf("Number of test cases: \n");
46     scanf("%d",&n); printf ("%d", n);
47     for (i = 1; i < n+1; i++)
48     {
49         printf("Set the values of the triangle sides (one per
```

```

        line): \n");
50     scanf("%d",&a); printf ("%d,", a);
51     scanf("%d",&b); printf ("%d,", b);
52     scanf("%d",&c); printf ("%d: ", c);
53     if(a>b && a>c)
54     {
55         int aux = c;
56         c=a;
57         a=aux;
58     }
59     else if(b>a && b>c)
60     {
61         int aux = c;
62         c=b;
63         b=aux;
64     }
65
66     checktriangle(a,b,c);
67 };
68
69     return 0;
70 }

```

Listing 4: Código

Con los casos de prueba de la Tabla 2, sin añadir ninguno más, ya se obtiene el porcentaje de cobertura al 100% en todos los aspectos, como se puede ver en esta nueva actualización de la cobertura extendida en la Figura 35.







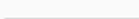

	Pass	Half	Fail	Excluded	Total
<b>Function</b> 	2 100.00%		0 0.00%	0 0.00%	2 100%
<b>Expressions</b> 	59 100.00%		0 0.00%	0 0.00%	59 100%
<b>Conditions</b> 	18 100.00%	0 0.00%	0 0.00%	0 0.00%	18 100%
<b>MC/DC</b> 	9 100.00%		0 0.00%	0 0.00%	9 100%
<b>Branches</b>					
if 	7 100.00%	0 0.00%	0 0.00%	0 0.00%	7 100%
for 	1 100.00%	0 0.00%	0 0.00%	0 0.00%	1 100%
while 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%
case 	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 100%

Figura 35: Cobertura extendida de la segunda modificación.

Para terminar, hay un caso de los propuestos inicialmente que se pueden quitar de la Tabla 2 y se seguiría obteniendo una cobertura total. Este caso es el Caso 4. Con esto, la tabla de casos de prueba final sería la Tabla 5 con 7 casos totales.

Casos de prueba	Valores de entrada	Salida esperada	Salida obtenida	Coinciden
Caso 1	(6, 5, 10)	Escaleno y Obtuso	Scalene triangle Obtuse Triangle	Sí
Caso 2	(3, 3, 3)	Equilátero y Agudo	Equilateral triangle Acute Triangle	Sí
Caso 3	(4, 5, 4)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 4	(4, 3, 0)	No triángulo	Is not a Triangle	Sí
Caso 5	(5, 3, 4)	Escaleno y Recto	Scalene triangle Right Triangle	Sí
Caso 6	(4, 5, 5)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí
Caso 7	(5, 4, 5)	Isósceles y Agudo	Isosceles triangle Acute Triangle	Sí

Tabla 5: Casos de prueba final y sus salidas.



Hasta ahora se ha podido reducir los casos de prueba hasta 7, pero existe una mejora que queremos mencionar [6]. Está mejora elimina la zona de conflicto que se detectó en la subsección 3.2.1, y se reemplaza con un cálculo de cuadrados usando float para conocer el ángulo de los triángulos. Esto hace que los casos de prueba sean menos, incluso algunos de los casos que se dan en el enunciado se podrían quitar.

```

1  #include <math.h>
2  #include <stdio.h>
3
4  int checktriangle(float a, float b, float c)
5  {
6      float A, B, C;
7
8
9      if((c<a+b)&&(b<a+c)&&(a<c+b))
10     {
11         if(a==b && b==c){
12             printf("Equilateral Triangle \n");
13         }else if(a==b || b==c || a==c){
14             printf("Isosceles Triangle \n");
15         }else{
16             printf("Scalene triangle \n");
17         }
18
19         A = (pow(b,2) + pow(c,2) - pow(a,2)) / (2*b*c);
20         B = (pow(a,2) + pow(c,2) - pow(b,2)) / (2*a*c);
21         C = (pow(a,2) + pow(b,2) - pow(c,2)) / (2*a*b);
22
23         if (A < 0 || B < 0 || C < 0){
24             printf("Obtuse Triangle \n");
25         } else if (A == 0 || B == 0 || C == 0){
26             printf("Right Triangle \n");
27         } else {
28             printf("Acute Triangle \n");
29         }
30     }
31     else{
32         printf("Is not a Triangle \n");
33     }
34
35     return 0;
36 }
37
38 int main()
39 {
40     int i, n;
41     float a, b, c;
42     printf("Number of test cases: \n");
43     scanf("%d",&n); printf ("%d", n);
44     for (i = 1; i < n+1; i++)
45     {
46         printf("Set the values of the triangle sides (one per

```

```

        line): \n");
47     scanf("%f",&a); printf ("%f,", a);
48     scanf("%f",&b); printf ("%f,", b);
49     scanf("%f",&c); printf ("%f: ", c);
50
51     checktriangle(a,b,c);
52 };
53
54     return 0;
55 }

```

Listing 5: Código

### 3.3 Cálculo de la métrica de McCabe

En esta sección se va a obtener las métricas, incluida la complejidad ciclomática y la comprobación manual de su valor del código proporcionado. Se comparará el diagrama creado manualmente con el generado por la herramienta Code2flow (<https://code2flow.com/app>) realizando un análisis en detalle de ambos.

Esta herramienta se encarga de crear el diagrama de flujo a partir del código introducido. En la Figura 36 se observa que a la izquierda se encuentra la zona donde se introduce el código deseado y a la derecha se muestra el diagrama de flujo creado.

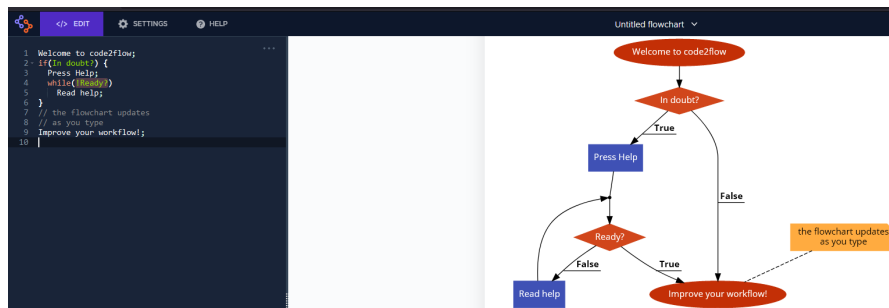


Figura 36: Interfaz de la herramienta.

El primer paso es introducir el código que vamos a usar en su lugar correspondiente. Al instante se observa el diagrama de flujo creado como se muestra en la Figura 37.

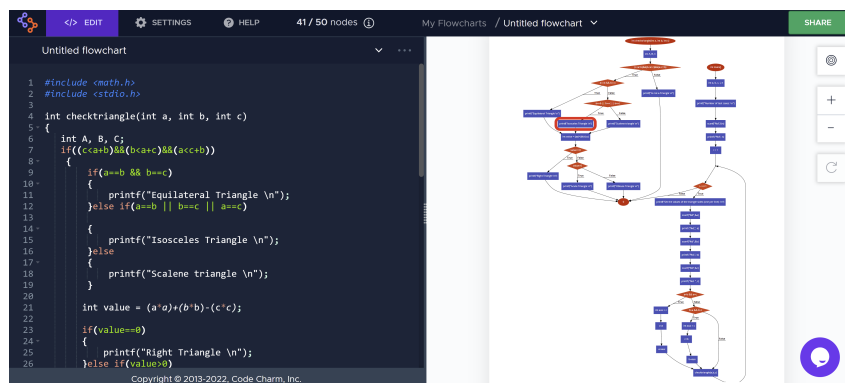


Figura 37: Herramienta con el código introducido.

La Figura 38 es el diagrama completo obtenido por esta herramienta.



Para comprobar el diagrama manualmente, dividimos el código en partes como se muestra a continuación:

```
1  #include <math.h>
2  #include <stdio.h>
3
4  // Nodo 13
5  int checktriangle(int a, int b, int c)
6  {
7      // Nodo 14
8      int A, B, C;
9      // Nodo 15, 16 y 17
10     if((c<a+b)&&(b<a+c)&&(a<c+b))
11     {
12         // Nodo 19 y 20
13         if(a==b && b==c)
14         {
15             // Nodo 21
16             printf("Equilateral Triangle \n");
17             // Nodo 22, 23 y 24
18             }else if(a==b || b==c || a==c)
19             {
20                 // Nodo 25
21                 printf("Isosceles Triangle \n");
22             }else
23             {
24                 // Nodo 26
25                 printf("Scalene triangle \n");
26             }
27         // Nodo 27
28         int value = (a*a)+(b*b)-(c*c);
29         // Nodo 28
30         if(value==0)
31         {
32             // Nodo 29
33             printf("Right Triangle \n");
34             // Nodo 30
35             }else if(value>0)
36             {
37                 // Nodo 31
38                 printf("Acute Triangle \n");
39             }else
40             {
41                 // Nodo 32
42                 printf("Obtuse Triangle \n");
43             }
44         }
45     }
46     else
47     {
48         // Nodo 18
49         printf("Is not a Triangle \n");
50     }
51     // Nodo 33
```

```

52     return 0;
53 }
54 // Nodo 1
55 int main()
56 {
57     // Nodo 2
58     int a, b, c, i, n;
59     printf("Number of test cases: \n");
60     scanf("%d",&n); printf ("%d", n);
61     // Nodo 3
62     for (i = 1; i < n+1; i++)
63     {
64         // Nodo 4
65         printf("Set the values of the triangle sides (one per
           line): \n");
66         scanf("%d",&a); printf ("%d,", a);
67         scanf("%d",&b); printf ("%d,", b);
68         scanf("%d",&c); printf ("%d: ", c);
69         // Nodo 5 y 6
70         if(a>b && a>c)
71         {
72             // Nodo 7
73             int aux = c;
74             c=a;
75             a=aux;
76         }
77         // Nodo 8 y 9
78         else if(b>a && b>c)
79         {
80             // Nodo 10
81             int aux = c;
82             c=b;
83             b=aux;
84         }
85         // Nodo 11
86         checktriangle(a,b,c);
87     };
88     // Nodo 12
89     return 0;
90 }

```

Listing 6: Código

Cada nodo marcado en el código se corresponde a un nodo dibujado manualmente en la Figura 39.

Cabe destacar una diferencia notable entre el diagrama de la herramienta y el hecho a mano. La herramienta considera como un nodo único las decisiones que implican varias condiciones y en el hecho a mano se separan para calcular más fácilmente la complejidad ciclomática.

La complejidad ciclomática se puede calcular de las siguientes maneras:

$$V(G) = \text{NumeroRegionesCerradas} \quad (1)$$

$$V(G) = \text{Aristas} - \text{Nodos} + 2 \quad (2)$$

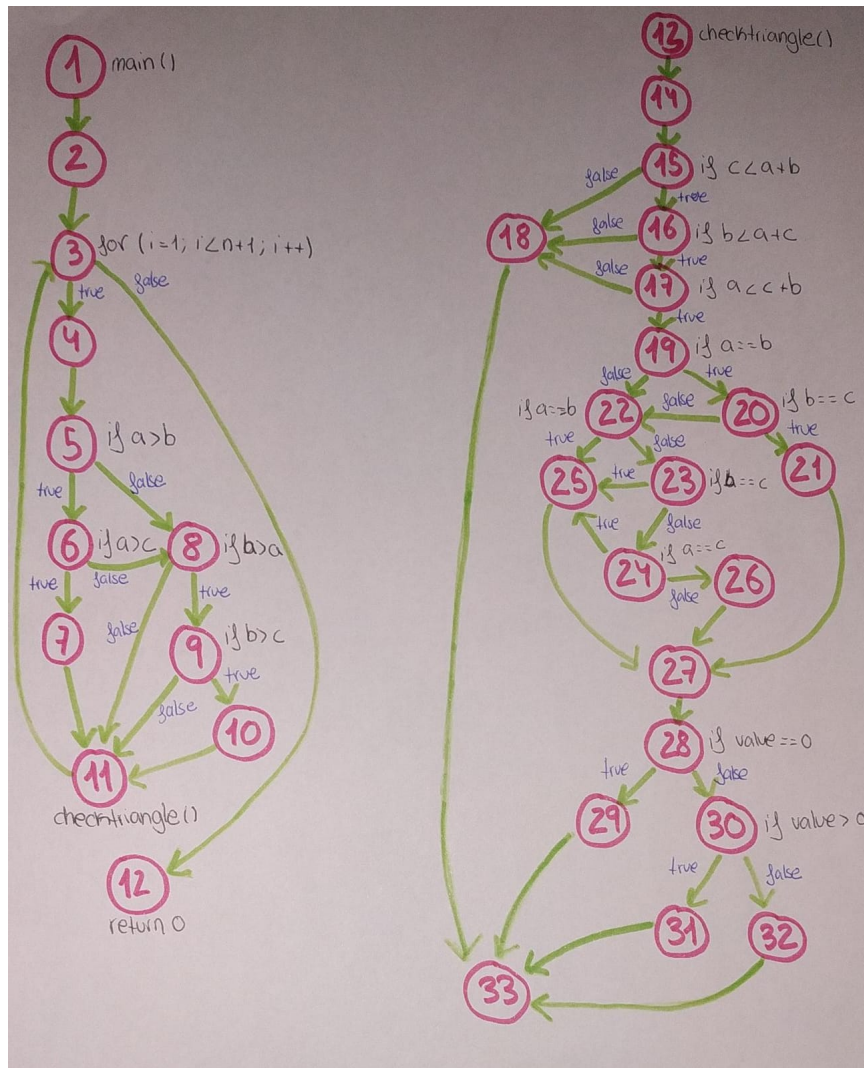


Figura 39: Diagrama de flujo hecho manualmente.

$$V(G) = \text{NodosPredicado} + 1 \quad (3)$$

En el diagrama obtenido por la herramienta los resultados a las anteriores ecuaciones son:

$$V(G) = \text{NumeroRegionesCerradas} = 9 \quad (4)$$

$$V(G) = \text{Aristas} - \text{Nodos} + 2 = 47 - 40 + 2 = 9 \quad (5)$$

$$V(G) = \text{NodosPredicado} + 1 = 8 + 1 = 9 \quad (6)$$

La complejidad ciclomática del diagrama obtenido por la herramienta es 9.

En el diagrama hecho a mano los resultados a las anteriores ecuaciones son:

$$V(G) = \text{NumeroRegionesCerradas} = 16 \quad (7)$$

$$V(G) = \text{Aristas} - \text{Nodos} + 2 = 46 - 32 + 2 = 16 \quad (8)$$

$$V(G) = \text{NodosPredicado} + 1 = 15 + 1 = 16 \quad (9)$$

Se ha contado el nodo 12 y el 33 (return 0;) como uno solo, como se hace en el diagrama de la herramienta. La complejidad ciclomática del diagrama obtenido por la herramienta es 16.

Las complejidades no son iguales debido a la diferencia mencionada anteriormente. En el diagrama de la herramienta no se han dividido las decisiones en sus condiciones y en el hecho manualmente sí, por ello, es más correcto este último.



## References

- [1] *Apache Maven Project*. <https://maven.apache.org/> [Último acceso 14/marzo/2022].
- [2] *Docker Desktop overview*. <https://docs.docker.com/desktop/> [Último acceso 26/marzo/2022].
- [3] *GitHub - About continuous integration*. <https://docs.github.com/es/actions/automating-builds-and-tests/about-continuous-integration> [Último acceso 16/marzo/2022].
- [4] *Introduction to the POM*. <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html> [Último acceso 14/marzo/2022].
- [5] *JaCoCo Java Code Coverage Library*. <https://www.jacoco.org/jacoco/> [Último acceso 15/marzo/2022].
- [6] Inés López Baldominos. “Aplicación de herramientas y análisis de datos para el aseguramiento de calidad del software”. In: (2020).
- [7] *Página del framework JMetal*. <https://github.com/jMetal/jMetal> [Último acceso 14/marzo/2022].