UNIVERSITA'
DEGLI STUDI
DI BARI

Integrazione e Test di Sistemi Software

Homework 1 - Black box & White box

Student

Marco Porro 717061

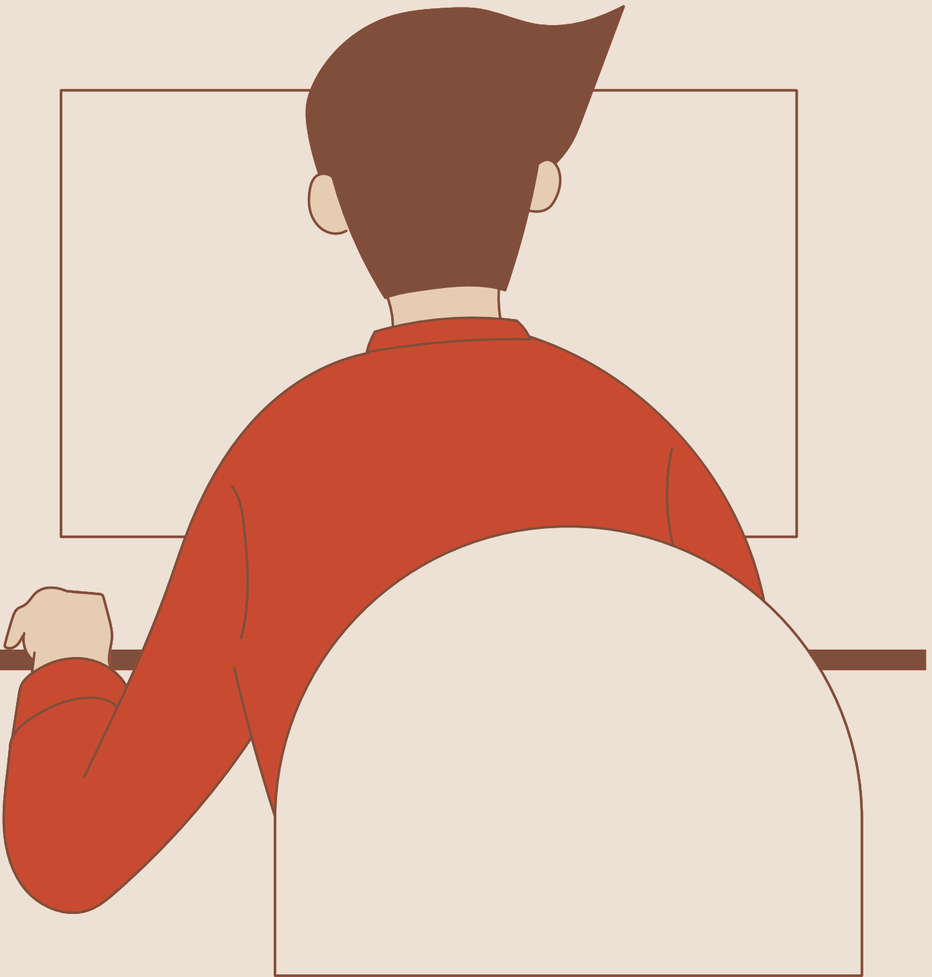
Student

Stefano Mansi 717448

Testing Workflow

I. Understanding the requirements

Analizziamo i requisiti del nostro programma andando ad identificare per ogni metodo**: obiettivo, input, output.**



Analisi del codice

Il codice è strutturato in due classi principali: **Main** e **OperazioniMath**. La prima gestisce l'interazione con l'utente attraverso l'input da console e le stampe dell'output, mentre la seconda contiene i metodi per la conversione di numeri e il calcolo delle soluzioni di un'equazione di secondo grado.

Metodi

- public String **convertiBase**(int numeroDecimale, int baseDestinazione);
- public double[] **calcolaSoluzioniEquazioneSecondoGrado**(double a, double b, double c)

1 **public String convertiBase(int numeroDecimale, int baseDestinazione)**

L'obiettivo di questo metodo è quello di convertire un numero decimale in un numero con base binaria, ottale o esadecimale.

Input

Il metodo riceve due parametri in input:

- **numeroDecimale**, il quale rappresenta il numero intero positivo decimale da convertire scelto dall'utente
- **baseDestinazione**, il quale rappresenta un numero intero che identifica la base in cui il numero sarà convertito.

Output

Il metodo restituisce (in tipo **String**) il numero convertito in una nuova base binaria, ottale o esadecimale .

- Se la base non risulta essere 2 o 8 o 16 restituirà una Stringa di errore "**Base di destinazione non supportata"**.
- Se il numero risulta minore o uguale a zero restituirà una **stringa vuota**.

2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

L'obiettivo di questo metodo è quello di calcolare le soluzioni di un'equazione di secondo grado.

Input

Il metodo riceve come parametri in input:

- **a**,il quale rappresenta il coefficiente del termine di grado 2 (a!=0);
- **b**,il quale rappresenta il coefficiente del termine di grado 1;
- **c**,il quale rappresenta il termine noto.

Output

Il metodo restituisce :

- **due soluzioni reali** nel caso in cui il delta sia positivo;
- **un'unica soluzione reale** nel caso in cui il delta sia uguale a zero;
- un messaggio con scritto "**L'equazione non ha soluzioni reali"** nel caso in cui il delta sia minore di zero (due valori indefiniti);
- un valore **null** nel caso in cui a == 0, pertanto verrà mostrato un messaggio con scritto **"a uguale zero impossibile eseguire i calcoli"**.

II. Explore what the program does for various inputs

Partendo dai requisiti esaminiamo cosa il nostro programma fa per diversi input, in modo da avere un modello mentale chiaro su come il programma dovrebbe funzionare.

- Il metodo **convertiBase()** è progettato per convertire un numero intero positivo decimale in un numero con una base scelta dall'utente;
- Il metodo **calcolaSoluzioniEquazioneSecondoGrado()** è progettato per calcolare le soluzioni di un'equazione di secondo grado.

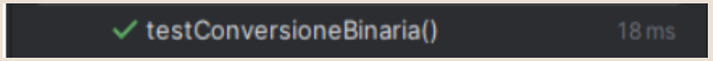
Per entrambi i metodi verifichiamone il comportamento con determinati tipi di input.

1 public String convertiBase(int numeroDecimale, int baseDestinazione)

testConversioneBinaria()

Verifica se dati un numero decimale(**14**) e una base di destinazione binaria(**2**) ci venga effettivamente restituito il numero binario desiderato(**1110**).

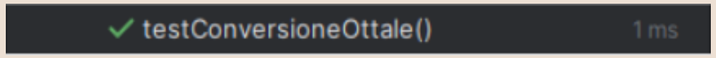
```java
@Test
void testConversioneBinaria() {
   assertEquals("1110", operazioniMath.convertiBase(14, 2));
}
```



testConversioneOttale()

Verifica se dati un numero decimale(**27**) e una base di destinazione ottale(**8**) ci venga effettivamente restituito il numero in base otto desiderato(**33**).

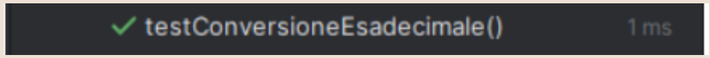
```java
@Test
void testConversioneOttale() {
   assertEquals("33", operazioniMath.convertiBase(27, 8));
}
```



testConversioneEsadecimale()

Verifica se dati un numero decimale(**255**) e una base di destinazione esadecimale(**16**) ci venga effettivamente restituito il numero in base sedici desiderato(**FF**).

```java
@Test
void testConversioneEsadecimale() {
   assertEquals("FF", operazioniMath.convertiBase(255, 16));
}
```



2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

testEquazioneDueSoluzioniReali()

Verifica se dati tre coefficienti(**1,-3,2**) distribuiti in modo che l'equazione abbia soluzioni reali ,il metodo restituisca effettivamente le soluzioni reali desiderate(**2.0,1.0**)

```
@Test
void testEquazioneDueSoluzioniReali() {
   assertArrayEquals(new double[]{2.0, 1.0},
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, -3, 2));
}
```

✓ testEquazioneDueSoluzioniReali() 1 ms

testEquazioneUnaSoluzioneReale()

Verifica se dati tre coefficienti(**1,-2,1**) distribuiti in modo che l'equazione ammetta una soluzione reale ,il metodo restituisca effettivamente il valore della soluzione reale desiderata(**1.0**)

```
@Test
void testEquazioneUnaSoluzioneReale() {
   assertArrayEquals(new double[]{1.0, 1.0},
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, -2, 1));
}
```

✓ testEquazioneUnaSoluzioneReale() 1 ms

testEquazioneNessunaSoluzioneReale()

Verifica se dati tre coefficienti(**2,1,2**), distribuiti in modo che l'equazione abbia il delta negativo , il metodo restituisce valori **NaN** (utilizzati per rappresentare risultati indefiniti o indeterminati in operazioni matematiche).

```
@Test
void testEquazioneNessunaSoluzioneReale() {
   assertArrayEquals(new double[]{Double.NaN, Double.NaN},
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(2, 1, 2));
}
```

✓ testEquazioneNessunaSoluzioneReale() 3 ms

III. Explore inputs, outputs and identify partitions

Identifichiamo **classi di input** che si comportano nello stesso modo per effettuare un singolo caso di test per classe.

Metodo convertiBase()

INPUT:

int numeroDecimale:
numeroDecimale >0;
numeroDecimale <=0;

int baseDestinazione:
baseDestinazione = 2;
baseDestinazione = 8;
baseDestinazione = 16;
baseDestinazione != 2 or !=8 or !=16

OUTPUT:

String risultato:
stringa numero convertito;
stringa vuota;
stringa di errore (Base di destinazione non supportata)

Metodo calcolaSoluzioniEquazioneSecondoGrado()

INPUT:

double a:
a < sogliamin || a > sogliamax;
a = 0;
sogliamin<a<+sogliamax

double b:
b < sogliamin || b > sogliamax;
sogliamin<b<sogliamax

double c:
c < sogliamin || c > sogliamax;
sogliamin<c<+sogliamax

OUTPUT:

double [] soluzioni:
soluzioni[0] != soluzioni[1];
soluzioni[0] == soluzioni[1];
soluzioni[0] e soluzioni[1] = Double.NaN;
null (quando a = 0);
AritmeticException

IV. Identify boundary cases (aka corner cases)

Identifichiamo i **boundary case** del nostro programma perché spesso è in prossimità dei limiti che si possono verificare errori o comportamenti inattesi.

Metodo convertiBase()

Basi ammissibili (**in point**) :

- baseDestinazione = 2;
- baseDestinazione = 8;
- baseDestinazione = 16;

Numeri decimali (**on point**) :

- numeroDecimale = 1;

Basi non ammissibili (**out Point**) :

- baseDestinazione !=2 e !=8 e !=16

Numeri decimali (**off point**) :

- numeroDecimale = 0;

Metodo calcolaSoluzioneEquazioneSecondoGrado()

coefficiente a ammissibile(**on point**):

- a=1 ;

coefficiente a non ammissibile(**off point**):

- a= 0;

V. Devise test cases

Ideiamo i casi di test andando a decidere quali partizioni dovremmo combinare con le altre e quali no.

Dopo aver compreso i requisiti, esplorato il comportamento del programma per diverse tipologie di input e identificato partizioni e casi limite, è cruciale sviluppare casi di test specifici che mettano alla prova ogni aspetto critico del nostro sistema.

La creazione di tali casi di test è fondamentale per garantire una copertura completa delle funzionalità e una rilevazione efficace di eventuali difetti.

1 public String convertiBase(int numeroDecimale, int baseDestinazione)

T1	numeroDecimale > 0 e baseDestinazione=2
T2	numeroDecimale > 0 e baseDestinazione=8
T3	numeroDecimale > 0 e baseDestinazione=16
T4	numeroDecimale = 0 e baseDestinazione =2 “boundary case off point”
T5	numeroDecimale = 0 e baseDestinazione =8 “boundary case off point”
T6	numeroDecimale = 0 e baseDestinazione =16 “boundary case off point”
T7	numeroDecimale > 0 e baseDestinazione sbagliata !=2 e !=8 e !=16
T8	numeroDecimale = 0 e baseDestinazione sbagliata !=2 e !=8 e !=16 “boundary case off point”
T9	numeroDecimale = 1 e baseDestinazione corretta “boundary case on point”

2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

T1	a, b e c definite in modo tale che il risultato sia composto da due soluzioni reali
T2	a, b e c definite in modo tale che il risultato sia una soluzione reale
T3	a, b e c definite in modo tale che non ci siano soluzioni reali
T4	a = 0, b ∈ R , c ∈ R in modo tale che restituisca null “boundary case off point”
T5	a = 1, b ∈ R , c ∈ R “ boundary case on point”

VI. Automate test cases

Ci concentriamo sull'automatizzazione dei casi di test precedentemente ideati.

Traduciamo i nostri casi di test in codice che può essere eseguito automaticamente, permettendoci di ripetere i test in modo rapido e affidabile.

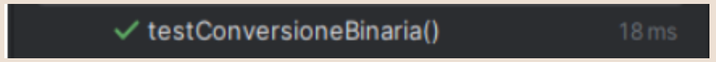
Durante questo passo, cerchiamo di garantire che essi coprano tutte le situazioni previste nei nostri casi di test.

1 public String convertiBase(int numeroDecimale, int baseDestinazione)

T1 testConversioneBinaria()

Verifichiamo il corretto comportamento del metodo con un numeroDecimale maggiore di zero (**14**) e come base di conversione quella binaria(**2**). Il risultato è stato conforme a quello atteso (**1110**).

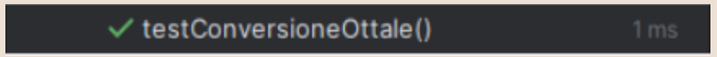
```
@Test
void testConversioneBinaria() {
   assertEquals("1110", operazioniMath.convertiBase(14, 2));
}
```



T2 testConversioneOttale()

Verifichiamo il corretto comportamento del metodo con un numeroDecimale maggiore di zero (**27**) e come base di conversione quella ottale(**8**).Il risultato è stato conforme a quello atteso (**33**).

```
@Test
void testConversioneOttale() {
   assertEquals("33", operazioniMath.convertiBase(27, 8));
}
```

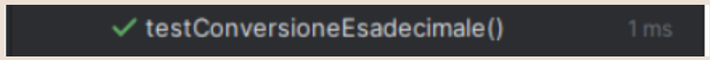


T3 testConversioneEsadecimale()

Verifichiamo il corretto comportamento del metodo con un numeroDecimale maggiore di zero (**255**) e come base di conversione quella esadecimale(**16**).

Il risultato è stato conforme a quello atteso (**FF**).

```
@Test
void testConversioneEsadecimale() {
   assertEquals("FF", operazioniMath.convertiBase(255, 16));
}
```



1 public String convertiBase(int numeroDecimale, int baseDestinazione)

T4 T5 T6 testNumeroDecimaleZeroBaseCorretta()

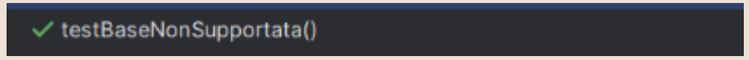
Verifichiamo come si comporta il metodo nel caso in cui gli venga passato in input un numeroDecimale uguale a zero e come base di conversione binaria, ottale ed esadecimale(**2,8,16**).
Abbiamo realizzato il test utilizzando dei test parametrici per semplificare l'esecuzione.
Il risultato è stato conforme a quello atteso (**una stringa vuota**).

```
@ParameterizedTest
@ValueSource (ints={2,8,16})
void testNumeroDecimaleZeroBaseCorretta(int base) {
   assertEquals("", operazioniMath.convertiBase(0, base));
}
```

T7 testBaseNonSupportata()

Verifichiamo come si comporta il metodo nel caso in cui gli venga passato in input un numeroDecimale maggiore di zero (**10**) e come base di conversione una errata (**10**). Il risultato è stato conforme a quello atteso (**la stringa di errore**).

```
@Test
public void testBaseNonSupportata() {
   assertEquals("Base di destinazione non supportata",
operazioniMath.convertiBase(10, 10));
}
```

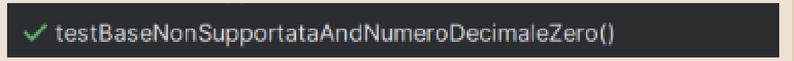


1 **public String convertiBase(int numeroDecimale, int baseDestinazione)**

T8 **testBaseNonSupportataAndNumeroDecimaleZero()**

Verifichiamo come si comporta il metodo nel caso in cui gli venga passato in input un numeroDecimale uguale a zero e come base di conversione una errata (**10**). Il risultato è stato conforme a quello atteso (**la stringa di errore**).

```
@Test
public void testBaseNonSupportataAndNumeroDecimaleZero() {
   assertEquals("Base di destinazione non supportata",
operazioniMath.convertiBase(0, 10));
}
```



T9 **testNumeroDecimaleUnoAndBaseCorretta()**

Verifichiamo il corretto comportamento del metodo con un numeroDecimale uguale a uno (**boundary case on point**) e come base di conversione binaria, ottale ed esadecimale(**2,8,16**). Abbiamo realizzato il Test utilizzando dei test parametrici per semplificare l'esecuzione. Il risultato è stato conforme a quello atteso (**1**).

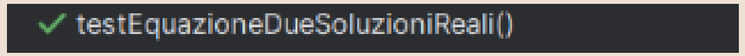
```
@ParameterizedTest
@ValueSource (ints={2,8,16})
void testNumeroDecimaleUnoAndBaseCorretta(int base) {
   assertEquals("1", operazioniMath.convertiBase(1, base));
}
```

2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

T1 testEquazioneDueSoluzioniReali()

Verifichiamo il corretto comportamento del metodo nel caso in cui gli venga passato come input coefficienti (**a=1; b = -3; c = 2**) definiti in modo tale da restituire due soluzioni reali. Il risultato è stato conforme a quello atteso (un **array** di **double** formato dalle due soluzioni **2.0** e **1.0**).

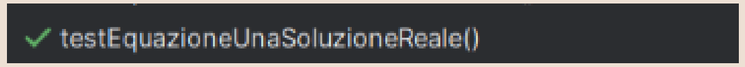
```
@Test
void testEquazioneDueSoluzioniReali() {
   assertArrayEquals(new double[]{2.0, 1.0},
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, -3, 2));
}
```



T2 testEquazioneUnaSoluzioneReale()

Verifichiamo il corretto comportamento del metodo nel caso in cui gli venga passato come input coefficienti (**a=1; b = -2; c = 1**) definiti in modo tale da restituire un'unica soluzione reale. Il risultato è stato conforme a quello atteso (un **array** di **double** formato dalle due soluzioni coincidenti **1.0** e **1.0**).

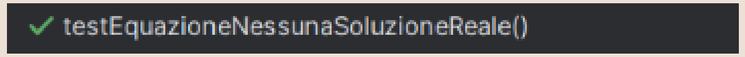
```
@Test
void testEquazioneUnaSoluzioneReale() {
   assertArrayEquals(new double[]{1.0, 1.0},
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, -2, 1));
}
```



T3 testEquazioneNessunaSoluzioneReale()

Verifichiamo il corretto comportamento del metodo nel caso in cui gli venga passato come input coefficienti (**a=2; b = 1; c = 2**) definiti in modo tale da avere il delta negativo e quindi nessuna soluzione reale. Il risultato è stato conforme a quello atteso (un **array** di **double** formato da due valori **Double.NaN**)..

```
@Test
void testEquazioneNessunaSoluzioneReale() {
   assertArrayEquals(new double[]{Double.NaN, Double.NaN},
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(2, 1, 2));
}
```



2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

T4 testCoefficienteAUgualeZero()

Verifichiamo il comportamento del metodo nel caso in cui gli venga passato come input il coefficiente di x alla seconda(a) uguale a zero mentre i coefficienti b e c appartengono a tutto l'insieme R (**a=0; b = 10; c = 20**).
Il risultato è stato conforme a quello atteso (valore **null**).

```
@Test
public void testCoefficienteAUgualeZero() {
  assertNull(operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(0, 10,
20));
}
```

✓ testCoefficienteAUgualeUno()

T5 testCoefficienteAUgualeUno()

Verifichiamo il corretto comportamento del metodo nel caso in cui gli venga passato come input il coefficiente di x alla seconda(a) uguale a uno (**boundary case on point**) mentre il coefficiente b e c appartenente a tutto l'insieme R (**a=1; b = -5; c = 6**).
Il risultato è stato conforme a quello atteso (una **array** di **double** formato dai due valori **3.0**, **2.0**).

```
@Test
public void testCoefficienteAUgualeUno() {
   assertArrayEquals(new double[]{3.0,
2.0},operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, -5, 6));
}
```

✓ testCoefficienteAUgualeZero()

VII. Augment the test suite with creativity and experience

L'obiettivo principale di questo step è elevare la suite di test garantendo che il nostro programma sia in grado di gestire situazioni complesse o impreviste.

Questo step rappresenta un momento cruciale per assicurare che il nostro programma non solo soddisfi gli standard di base, ma sia anche pronto a fronteggiare eventualità complesse con successo.

La creatività gioca un ruolo decisivo, consentendo di immaginare possibili scenari reali o di simulare condizioni estreme che potrebbero mettere alla prova la robustezza del sistema.

1 **public String convertiBase(int numeroDecimale, int baseDestinazione)**

testNumeroDecimaleNegativo()

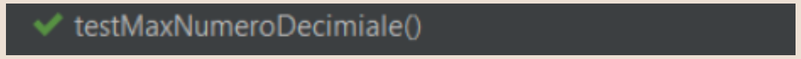
Verifica il comportamento della funzione quando viene passato un numero decimale negativo come input (**numeroDecimale=-1**). Abbiamo incluso questo caso per assicurarci che la funzione gestisca correttamente numeri negativi, e ci aspettiamo che restituisca una **stringa vuota** (**""**).

```
@Test
public void testNumeroDecimaleNegativo(){
   assertEquals("",operazioniMath.convertiBase(-1,2));
}
```

testMaxNumeroDecimale()

Verifica come la funzione si comporta quando le viene passato il valore massimo rappresentabile come numero decimale intero (**Integer.MAX_VALUE -> 2^31-1**). Ci aspettiamo che la funzione converte correttamente questo numero in binario, producendo la stringa rappresentante l'intero in base 2(**11111111111111111111111111111111**)

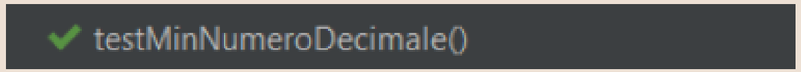
```
@Test
public void testMaxNumeroDecimiale(){

assertEquals("11111111111111111111111111111111",operazioniMath.convertiBas
e(Integer.MAX_VALUE,2));
}
```



testMinNumeroDecimale()

Verifica il comportamento della funzione quando viene passato il valore minimo rappresentabile come numero decimale intero (**Integer.MIN_VALUE-> -2^31-1**). Abbiamo incluso questo caso per assicurarci che la funzione gestisca correttamente il valore minimo, e ci aspettiamo che restituisca una **stringa vuota** (**""**).

```
@Test
public void testMinNumeroDecimale(){
assertEquals("",operazioniMath.convertiBase(Integer.MIN_VALUE,2));
}
```



2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

testSoluzioniZero()

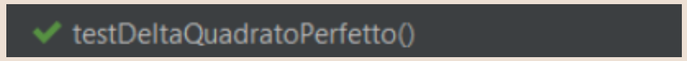
Abbiamo incluso questo test per verificare il comportamento della funzione quando il coefficiente quadratico dell'equazione di secondo grado è 1(**a=1**), mentre gli altri coefficienti sono zero(**b=0,c=0**). Ci aspettiamo che la funzione restituisca due soluzioni coincidenti, entrambe pari a zero. Il risultato atteso è un **array** contenente due zeri(**{0, 0}**). Inoltre, abbiamo utilizzato il parametro di **tolleranza 0.0001** per gestire eventuali errori di precisione nei calcoli.

```
@Test
public void testSoluzioniZero() {
   assertArrayEquals(new double[] { 0.0, 0.0 },
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, 0, 0),0.0001);
}
```

testDeltaQuadratoPerfetto()

Abbiamo deciso di includere questo test per verificare come la funzione gestisce un caso in cui il delta dell'equazione di secondo grado è un quadrato perfetto. Questo scenario si verifica quando il coefficiente lineare è zero(**b=0**) e il termine noto è il negativo(**c=-1**) del coefficiente quadratico(**a=1**). Il risultato atteso è un **array** contenente le due soluzioni, entrambe corrispondenti alla radice quadrata del coefficiente quadratico, con uno dei valori negativo(**{1.0, -1.0}**)

```
@Test
public void testDeltaQuadratoPerfetto() {
   assertArrayEquals(new double[] { 1.0, -1.0 },
operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(1, 0, -1));
}
```



2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

testCoefficientiMaxMin()

Abbiamo ideato questo caso di test perché vorremmo verificare come si comporta il programma nel caso in cui i valori dei coefficienti siano estremamente piccoli o estremamente grandi. Per testare questo caso specifico abbiamo stabilito una soglia di valori , identificando come valore estremamente grande **1000000000** mentre come valore estremamente piccolo **-1000000000** anziché fare affidamento su valori come MAX_VALUE o MIN_VALUE.Utilizzando soglie personalizzate, abbiamo avuto maggiore flessibilità nel determinare range specifici che fossero rilevanti per il nostro caso d'uso.Inoltre, evitando di utilizzare i valori massimi o minimi predefiniti,abbiamo evitato la rappresentazione di risultati infiniti che avrebbero reso difficile interpretare o confrontare i risultati ottenuti durante i test(**{0.6180339887498949,-1.618033988749895}**).

```
@Test
public void testCoefficientiMaxMin(){
   assertArrayEquals(new
double[]{0.6180339887498949,-1.618033988749895},operazioniMath.calcolaSol
uzioniEquazioneSecondoGrado(1000000000,1000000000,-1000000000));
}
```

testCoefficientiMaxMin()

N.B.

Durante lo svolgimento dello step 7 è emersa la necessità di definire una sogliaMax e sogliaMin anziché fare affidamento su valori come MAX_VALUE o MIN_VALUE. Questa consapevolezza è giunta soltanto attraverso la scrittura del codice, evidenziando la limitazione intrinseca delle rappresentazioni numeriche predefinite. Questo aggiornamento è stato fondamentale per garantire una rappresentazione precisa dei limiti superiori e inferiori, contribuendo così a una maggiore precisione e affidabilità del programma nel gestire dati numerici.

Con l'utilizzo di **Double.MAX_VALUE** e **Double.MIN_VALUE** il valore attuale restituito dal metodo è **Infinity** -->

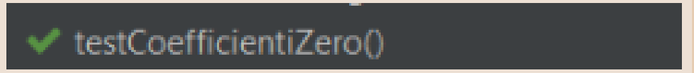
```
org.opentest4j.AssertionFailedError: array contents differ at index [0],
Expected :0.0
Actual   :Infinity
```

2 public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c)

testCoefficientiZero()

Abbiamo incluso questo test per verificare il comportamento della funzione quando tutti i coefficienti dell'equazione di secondo grado sono zero(**a=0,b=0,c=0**). Questo caso rappresenta una situazione particolare in cui l'equazione diventa essenzialmente una costante zero. Il risultato atteso è la mancanza di soluzioni reali, e pertanto ci aspettiamo che la funzione restituisca un **valore null**. Questo test è stato incluso per assicurarci che la funzione gestisca correttamente situazioni in cui l'equazione non ha soluzioni reali.

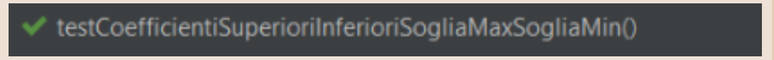
```
@Test
public void testCoefficientiZero(){
assertNull(operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(0,0,0));
}
```



testCoefficientiSuperioriInferioriSogliaMaxSogliaMin()

Abbiamo ideato questo caso di test per verificare se il metodo gestisce correttamente le situazioni in cui i coefficienti dell'equazione di secondo grado superano i valori massimi o minimi consentiti (**1000000000** e **-1000000000**). Si prevede che il metodo sollevi un'**eccezione** di tipo **ArithmeticException** in quanto almeno uno dei coefficienti è al di fuori dei limiti definiti.

```
@Test
public void testCoefficientiSuperioriInferioriSogliaMaxSogliaMin()
{
   assertThrows(ArithmeticException.class,() ->{

operazioniMath.calcolaSoluzioniEquazioneSecondoGrado(-1000000001,10000000
01,1000000001);
   });
}
```



Code Coverage e White-Box testing

Attraverso il **Code Coverage** abbiamo misurato quanto il nostro codice venisse esplorato e valutato durante i test.

Utilizzando il tool interno all'IDE IntelliJ abbiamo eseguito il code coverage con il seguente risultato:

Coverage: OperazioniMathTest

Element	Class, %	Method, %	Line, %
all	100% (2/2)	100% (26/26)	100% (65/65)
OperazioniMath	100% (1/1)	100% (5/5)	100% (44/44)
OperazioniMathTest	100% (1/1)	100% (21/21)	100% (21/21)

Infine abbiamo esportato il **report** del Code Coverage che è presente nella cartella “codeCoverage” del nostro progetto.

```java
public class OperazioniMath {

    // Metodo per la conversione della base (basi ammissibili 2, 8, 16; altrimenti restituisce una stringa di errore)
    public String convertiBase(int numeroDecimale, int baseDestinazione) {
        switch (baseDestinazione) {
            case 2:
                return convertiBinario(numeroDecimale);
            case 8:
                return convertiOttale(numeroDecimale);
            case 16:
                return convertiEsadecimale(numeroDecimale);
            default:
                return "Base di destinazione non supportata";
        }
    }

    // Metodo per la conversione in binario (se il numero è minore di zero restituiremo una stringa vuota)
    private String convertiBinario(int numeroDecimale) {
        StringBuilder risultato = new StringBuilder();
        while (numeroDecimale > 0) {
            int resto = numeroDecimale % 2;
            risultato.insert(0, resto);
            numeroDecimale /= 2;
        }
        return risultato.toString();
    }

    // Metodo per la conversione in ottale (se il numero è minore di zero restituiremo una stringa vuota)
    private String convertiOttale(int numeroDecimale) {
        StringBuilder risultato = new StringBuilder();
        while (numeroDecimale > 0) {
            int resto = numeroDecimale % 8;
            risultato.insert(0, resto);
            numeroDecimale /= 8;
        }
        return risultato.toString();
    }

    // Metodo per la conversione in esadecimale (se il numero è minore di zero restituiremo una stringa vuota)
    private String convertiEsadecimale(int numeroDecimale) {
        StringBuilder risultato = new StringBuilder();
        while (numeroDecimale > 0) {
            int resto = numeroDecimale % 16;
            risultato.insert(0, Integer.toHexString(resto).toUpperCase());
            numeroDecimale /= 16;
        }
        return risultato.toString();
    }

    public double[] calcolaSoluzioniEquazioneSecondoGrado(double a, double b, double c) {
        double[] soluzioni = new double[2];
        double sogliaMax = 1000000000;
        double sogliaMin = -1000000000;
        if (a==0){
            return null;
        }
        // Se uno tra i tre coefficienti dovesse risultare maggiore di sogliaMax o minore di sogliaMin allora lancia l'eccezione
        if (a>sogliaMax||b>sogliaMax||c>sogliaMax||a<sogliaMin||b<sogliaMin||c<sogliaMin){
            throw new ArithmeticException();
        }
        double delta = b * b - 4 * a * c;


        if (delta > 0) {
            // Due soluzioni reali
            double radiceDelta = Math.sqrt(delta);
            soluzioni[0] = (-b + radiceDelta) / (2 * a);
            soluzioni[1] = (-b - radiceDelta) / (2 * a);
        } else if (delta == 0) {
            // Una soluzione reale (delta uguale a zero)
            soluzioni[0] = -b / (2 * a);
            soluzioni[1] = soluzioni[0]; // La stessa soluzione in entrambi i casi
        } else {
            // Nessuna soluzione reale (delta negativo)
            soluzioni[0] = Double.NaN;
            soluzioni[1] = Double.NaN;
        }
```

Considerazioni Finali

Grazie al raggiungimento del **100% di code coverage** ed a un'ulteriore **analisi del codice**, abbiamo verificato che tutte le condizioni all'interno dei cicli e delle strutture di selezione sono state coperte in modo completo.

I test effettuati in precedenza hanno esaminato **tutte le possibili combinazioni** di input e hanno attraversato con successo tutti i rami condizionali del codice. Pertanto la copertura ottenuta è stata sufficiente per garantire un'adeguata esplorazione della logica interna del programma.

Non è stato necessario eseguire ulteriori test di white box.