



UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

INTEGRAZIONE E TEST DI SISTEMI

a.a 2023/2024

Gruppo: MAproject

Realizzato da:

Alessia Mansueto 738772

a.mansueto10@studenti.uniba.it

Corso: ITPS

Indice

HOMEWORK 1

Specification-based testing	3
1 – Comprendere i requisiti.....	3
2 – Esplora cosa fa il programma per vari input.....	4
3 – Esplorare input, output e identificare le partizioni	6
4 – Identificare i boundary case	8
5 – Definire i casi di test.....	8
6 – Automatizzare i casi di test.....	9
7 – Aumentare la suite di test con creatività ed esperienza	12
Structural testing	13
Code coverage.....	14

HOMEWORK 2

Property – based testing	16
1 - Esprimere le proprietà da testare	16
2 – Lasciare che sia il framework a scegliere i test	17
3 – Statistiche	20
4 – Code coverage.....	25

HOMEWORK 1

Specification-based testing

1 – Comprendere i requisiti

Il programma Java definisce una classe chiamata `ContoBancario` che rappresenta un conto bancario e fornisce diversi metodi per gestirlo. Questa classe deve gestire le operazioni bancarie di un conto. L'obiettivo principale è gestire le operazioni bancarie in modo sicuro e accurato, mantenendo la correttezza dei dati del conto.

Per il costruttore della classe `'ContoBancario'`

- Input:

- `'NumeroConto'` che è una stringa che rappresenta il numero di conto, deve essere composta da 9 cifre;

- `'saldo'` che è un `double` che rappresenta il saldo iniziale, deve essere un valore non negativo.

- Output: un nuovo oggetto `ContoBancario` inizializzato con il numero di conto e il saldo specificati

- Eccezione: se `NumeroConto` non è una stringa di 9 cifre e se `Saldo` è negativo.

Metodo `'deposito'`

- Input: `'importo'` un valore `double` che rappresenta l'importo da depositare sul conto bancario

- Output: il nuovo saldo (`double`) dopo aver aggiunto l'importo depositato al saldo corrente

Metodo `'prelievo'`

- Input: `'importo'` un valore `double` che rappresenta l'importo da prelevare dal conto bancario

- Output: il nuovo saldo (`double`) dopo aver sottratto l'importo prelevato dal saldo corrente

- Eccezione: se il saldo disponibile è insufficiente per coprire l'importo del prelievo.

Metodo `'trasferimento'`

- Input:

- `'destinazione'` un oggetto `ContoBancario` che rappresenta il conto di destinazione del trasferimento di denaro.

- `'importo'` un valore `double` che rappresenta l'importo da trasferire dal conto corrente al conto di destinazione

- Output: un valore boolean che indica se il trasferimento è avvenuto con successo (`true`) o meno (`false`). Se il saldo disponibile è insufficiente per coprire l'importo del trasferimento, il metodo restituisce `false`.

Metodo `'calcolaInteressi'`

- Input: `'tassoInteresse'` un valore `double` che rappresenta il tasso di interesse annuale espresso in percentuale.

- Output: l'importo degli interessi calcolati (`double`) basato sul saldo corrente e sul tasso di interesse specificato.

- Eccezione: se il `'tassoInteresse'` è negativo

Metodo `'cancellaInteressi'`

Questo metodo cancella tutti gli interessi calcolati per il conto bancario, svuotando la mappa `Interessi`

Metodo `'getSaldo'`

Indica il saldo attuale del conto bancario (`double`)

Metodo `'getInteressi'`

- Output: una mappa (Map <String, Double>) contenente gli interessi calcolati per il conto bancario. La chiave è il numero di conto e il valore è l'importo degli interessi.

La classe `ContoBancario` è progettata per gestire operazioni comuni di un conto bancario, come depositi, prelievi, trasferimenti di denaro e calcolo degli interessi. I metodi della classe assicurano la validità degli input attraverso controlli e lanciano eccezioni appropriate in caso di input non validi. Ogni metodo della classe ha chiari input e output, garantendo una manipolazione e gestione del conto bancario robusta ed efficiente.

2 – Esplora cosa fa il programma per vari input

Per comprendere meglio il funzionamento del programma `ContoBancario`, abbiamo eseguito una serie di test per verificare diverse situazioni di input e come il programma risponda ad esse. Abbiamo analizzato in particolare i seguenti aspetti:

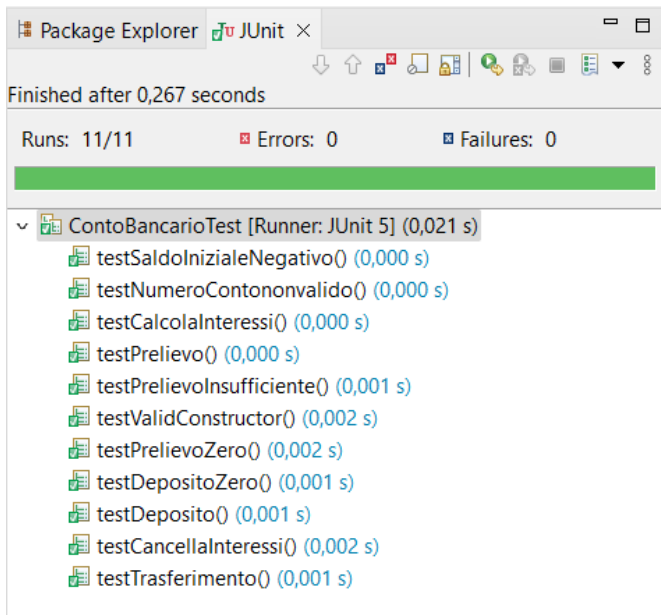
- *Costruttore del ContoBancario*: questo test verifica se il programma gestisce correttamente un numero di conto valido e un saldo iniziale positivo. Inoltre, abbiamo verificato se il programma solleva correttamente un'eccezione quando viene fornito un numero di conto non valido o un saldo iniziale negativo.
- *Deposito e prelievo*: questo test per verificare il deposito con importo positivo e con deposito pari a zero per vedere che non cambi il saldo con quest'ultimo. Per il prelievo per verificare il prelievo di un importo positivo, uno di un importo pari a zero (per non avere cambiamenti nel saldo) e inoltre, abbiamo verificato se il programma solleva un'eccezione quando viene tentato un prelievo di una somma superiore al saldo disponibile.
- *Trasferimento di denaro tra conti*: per verificare se il programma trasferisce correttamente denaro da un conto a un altro. Questo test è stato progettato per assicurarsi che il saldo dei conti coinvolti venga aggiornato correttamente dopo il trasferimento di una somma inferiore o uguale al saldo disponibile.
- *Calcolo degli interessi*: lo abbiamo testato per verificare se il programma calcola correttamente gli interessi in base al tasso di interesse fornito.
- *Cancellazione degli interessi*: lo abbiamo testato per confermare che il programma cancelli correttamente gli interessi calcolati, lasciando la mappa degli interessi vuota.

Questo approccio di testing ci ha permesso di ottenere una comprensione completa del comportamento del programma `ContoBancario` e di confermare che funziona correttamente in una varietà di situazioni di input.

```

10 class ContoBancarioTest {
11     private ContoBancario conto;
12
13     @BeforeEach
14     void setUp() {
15         // Inizializza un nuovo conto bancario prima di ogni test
16         conto = new ContoBancario("123456789", 1000);
17     }
18
19     // Test per verificare il costruttore con un numero di conto valido e saldo positivo
20     @Test
21     void testValidConstructor() {
22         assertEquals("123456789", conto.NumeroConto);
23         assertEquals(1000, conto.getSaldo());
24     }
25
26     // Test per verificare il lancio di un'eccezione con un numero di conto non valido
27     @Test
28     void testNumeroContoNonValido() {
29         assertThrows(IllegalArgumentException.class, () -> new ContoBancario("12345", 1000));
30     }
31
32     // Test per verificare il lancio di un'eccezione con un saldo iniziale negativo
33     @Test
34     void testSaldoInizialeNegativo() {
35         assertThrows(IllegalArgumentException.class, () -> new ContoBancario("123456789", -1000));
36     }
37
38     // Test per verificare il deposito su un conto
39     @Test
40     void testDeposito() {
41         conto.deposito(500);
42         assertEquals(1500, conto.getSaldo());
43     }
44
45     // Test per verificare un deposito con importo zero (il saldo non dovrebbe cambiare)
46     @Test
47     void testDepositoZero() {
48         conto.deposito(0);
49         assertEquals(1000, conto.getSaldo());
50     }
51
52     // Test per verificare un prelievo su un conto con saldo sufficiente
53     @Test
54     void testPrelievo() {
55         conto.prelievo(200);
56         assertEquals(800, conto.getSaldo());
57     }
58
59     // Test per verificare un prelievo con importo zero (il saldo non dovrebbe cambiare)
60     @Test
61     void testPrelievoZero() {
62         conto.prelievo(0);
63         assertEquals(1000, conto.getSaldo());
64     }
65
66     // Test per verificare un prelievo con importo superiore al saldo (dovrebbe lanciare un'eccezione)
67     @Test
68     void testPrelievoInsufficiente() {
69         assertThrows(IllegalArgumentException.class, () -> conto.prelievo(1200));
70     }
71
72     // Test per verificare se il metodo trasferisce correttamente denaro da un conto a un altro
73     @Test
74     void testTrasferimento() {
75         ContoBancario destinazione = new ContoBancario("987654321", 500);
76         conto.trasferimento(destinazione, 200);
77         assertEquals(800, conto.getSaldo());
78         assertEquals(700, destinazione.getSaldo());
79     }
80
81     // Test per verificare se calcola correttamente gli interessi in base al tasso di interesse fornito
82     @Test
83     void testCalcolaInteressi() {
84         double tassoInteresse = 5.0; // 5%
85         double interessePrevisto = 50; // 5% di 1000
86         assertEquals(interessePrevisto, conto.calcolaInteressi(tassoInteresse));
87     }
88
89     // Test per verificare se cancella correttamente gli interessi calcolati
90     @Test
91     void testCancellaInteressi() {
92         conto.calcolaInteressi(5.0);
93         conto.cancellaInteressi();
94         assertTrue(conto.getInteressi().isEmpty());
95     }
96
97
98 }

```



3 – Esplorare input, output e identificare le partizioni

Classi di input (Individuali)

- NumeroConto
 - Null
 - Stringa vuota
 - Stringa di lunghezza 1
 - Stringa di lunghezza > 1
- Saldo
 - Null
 - Numero negativo
 - Numero zero
 - Numero positivo
- Importo
 - Null
 - Numero negativo
 - Numero zero
 - Numero positivo
- TassoInteresse
 - Numero negativo
 - Numero zero
 - Numero positivo
- Destinazione
 - Null
 - Oggetto ContoBancario valido

Combinazione di Input

- ContoBancario (NumeroConto, saldo)
 - NumeroConto null e saldo positivo
 - NumeroConto valido e saldo negativo

- NumeroConto valido e saldo positivo
- NumeroConto vuoto e saldo positivo
- NumeroConto valido e saldo zero
- Deposito (importo)
 - importo negativo
 - importo zero
 - importo positivo
- Prelievo (importo)
 - importo negativo
 - importo superiore al saldo
 - importo positivo e inferiore al saldo
 - importo zero
- Trasferimento (destinazione, importo)
 - destinazione null e importo positivo
 - destinazione valido e importo negativo
 - destinazione valido e importo superiore al saldo
 - destinazione valido e importo positivo
 - destinazione valido e importo zero
- CalcolaInteressi (tassoInteresse)
 - tassoInteresse negativo
 - tassoInteresse zero
 - tassoInteresse positivo

Classi di output (attesi)

- ContoBancario
 - Se NumeroConto è null o non valido, o se saldo è negativo, ci aspettiamo che venga lanciata un'eccezione 'IllegalArgumentException'
 - Se i valori di NumeroConto e saldo sono validi, ci aspettiamo che l'oggetto ContoBancario venga creato correttamente con i valori specificati
- Deposito
 - Se importo è negativo, ci aspettiamo che venga lanciata un'eccezione IllegalArgumentException.
 - Se importo è zero, ci aspettiamo che il saldo rimanga invariato.
 - Se importo è positivo, ci aspettiamo che il saldo venga incrementato di importo
- Prelievo
 - Se importo è negativo o superiore al saldo disponibile, ci aspettiamo che venga lanciata un'eccezione IllegalArgumentException.
 - Se importo è zero, ci aspettiamo che il saldo rimanga invariato.
 - Se importo è positivo e inferiore al saldo disponibile, ci aspettiamo che il saldo venga decrementato di importo
- Trasferimento
 - Se destinazione è null o se importo è negativo o superiore al saldo disponibile, ci aspettiamo che venga lanciata un'eccezione IllegalArgumentException.
 - Se importo è zero, ci aspettiamo che i saldi di entrambi i conti rimangano invariati.
 - Se importo è positivo e inferiore al saldo disponibile, ci aspettiamo che il saldo del conto di origine venga decrementato di importo e che il saldo del conto di destinazione venga incrementato di importo.

- `calcolaInteressi`
 - Se `tassoInteresse` è negativo, ci aspettiamo che venga lanciata un'eccezione `IllegalArgumentException`.
 - Se `tassoInteresse` è zero o positivo, ci aspettiamo che il saldo venga incrementato del valore calcolato come $\text{saldo} * \text{tassoInteresse} / 100$

4 – Identificare i boundary case

- *ContoBancario*
 - I casi on point sono: `NumeroConto` ha esattamente 9 cifre, e `saldo` è esattamente 0
 - I casi off point: `NumeroConto` con più di 9 cifre, `NumeroConto` con meno di 9 cifre, e `saldo` è negativo.
- *Deposito*
 - I casi on point sono: `Importo` è esattamente 0
 - I casi off point: `Importo` è negativo, `Importo` è un numero troppo grande
- *Prelievo*
 - I casi on point: `importo` è uguale al `saldo`
 - I casi off point: `importo` è maggiore del `saldo`, `importo` negativo
- *Trasferimento*
 - I casi on point: `importo` uguale al `saldo`
 - I casi off point: `importo` maggiore del `saldo`, `importo` negativo, `destinazione` null
- *CalcolaInteressi*
 - I casi on point: `tassoInteresse` è esattamente 0
 - I casi off point: `tassoInteresse` negativo, `tassoInteresse` è un numero troppo grande.

5 – Definire i casi di test

- *Casi eccezionali:*

- T1: `NumeroConto` è null.
- T2: `NumeroConto` è una stringa vuota.
- T3: `Saldo` è negativo.
- T4: `TassoInteresse` è negativo.
- T5: `Destinazione` è null.

- *Costruttore ContoBancario:*

- T6: `NumeroConto` è null e `saldo` positivo.
- T7: `NumeroConto` valido e `saldo` negativo.
- T8: `NumeroConto` valido e `saldo` positivo.

- *Deposito e Prelievo:*

- T9: `Importo` è negativo o zero (deposito).
- T10: `Importo` è positivo e inferiore o uguale al `saldo` (deposito).
- T11: `Importo` è superiore al `saldo` (prelievo).
- T12: `Importo` è negativo (prelievo).
- T13: `Importo` è positivo e inferiore o uguale al `saldo` (prelievo).

- *Trasferimento:*

- T14: Destinazione è valida e importo è positivo e inferiore o uguale al saldo.
- T15: Destinazione è valida e importo è superiore al saldo.

- *CalcolaInteressi:*

- T16: TassoInteresse è negativo o zero.

- *CancellaInteressi:*

- T17: Verifica CancellaInteressi avvenuto

- **GetInteressi:**

- T18: Recupero interessi

6 – Automatizzare i casi di test

I primi sono i casi eccezionali, in ordine, ovvero:

T1: NumeroConto è null.

T2: NumeroConto è una stringa vuota.

T3: Saldo è negativo.

T4: TassoInteresse è negativo.

T5: Destinazione è null.

```
1 package com.example.FirstHomeworkTest;
2
3 import com.example.FirstHomework.*;
4
5 public class ContoBancarioTest {
6
7     // Casi di test eccezionali
8     @Test
9     public void testNumeroContoNull() {
10         Assertions.assertThrows(IllegalArgumentException.class, () -> {
11             new ContoBancario(null, 1000);
12         });
13     }
14
15     @Test
16     public void testNumeroContoVuoto() {
17         Assertions.assertThrows(IllegalArgumentException.class, () -> {
18             new ContoBancario("", 1000);
19         });
20     }
21
22     @Test
23     public void testSaldoNegativo() {
24         Assertions.assertThrows(IllegalArgumentException.class, () -> {
25             new ContoBancario("123456789", -1000);
26         });
27     }
28
29     @Test
30     public void testTassoInteresseNegativo() {
31         ContoBancario conto = new ContoBancario("123456789", 1000);
32         Assertions.assertThrows(IllegalArgumentException.class, () -> {
33             conto.calcolaInteressi(-1);
34         });
35     }
36
37     @Test
38     public void testDestinazioneNull() {
39         ContoBancario conto = new ContoBancario("123456789", 1000);
40         Assertions.assertThrows(IllegalArgumentException.class, () -> {
41             conto.trasferimento(null, 100);
42         });
43     }
44 }
```

I successivi sono i casi di test per il costruttore ContoBancario:

T6: NumeroConto è null e saldo positivo.

T7: NumeroConto valido e saldo negativo.

T8: NumeroConto valido e saldo positivo.

```
51 // Casi di test per il costruttore ContoBancario
52 @Test
53 public void testNumeroContoNullSaldoPositivo() {
54     Assertions.assertThrows(IllegalArgumentException.class, () -> {
55         new ContoBancario(null, 1000);
56     });
57 }
58
59 @Test
60 public void testNumeroContoValidoSaldoNegativo() {
61     Assertions.assertThrows(IllegalArgumentException.class, () -> {
62         new ContoBancario("123456789", -1000);
63     });
64 }
65
66 @Test
67 public void testNumeroContoValidoSaldoPositivo() {
68     ContoBancario conto = new ContoBancario("123456789", 1000);
69     Assertions.assertEquals(1000, conto.getSaldo());
70 }
```

Poi procediamo con i casi di test dei metodi Deposito e Prelievo:

T9: Importo è negativo o zero (deposito).

T10: Importo è positivo e inferiore o uguale al saldo (deposito).

T11: Importo è superiore al saldo (prelievo).

T12: Importo è negativo (prelievo).

T13: Importo è positivo e inferiore o uguale al saldo (prelievo).

```
74 // Casi di test Deposito e prelievo
75 @Test
76 public void testImportoNegativoOZeroDeposito() {
77     ContoBancario conto = new ContoBancario("123456789", 1000);
78     Assertions.assertThrows(IllegalArgumentException.class, () -> {
79         conto.deposito(-100);
80     });
81 }
82
83 @Test
84 public void testImportoPositivoInferioreOUgualeAlSaldoDeposito() {
85     ContoBancario conto = new ContoBancario("123456789", 1000);
86     conto.deposito(500);
87     Assertions.assertEquals(1500, conto.getSaldo());
88 }
89
90 @Test
91 public void testPrelievoConSaldoInsufficiente() {
92     ContoBancario conto = new ContoBancario("123456789", 1000); // Saldo iniziale di 1000
93     Assertions.assertThrows(IllegalArgumentException.class, () -> {
94         conto.prelievo(1500); // Tentativo di prelevare 1500, che è superiore al saldo disponibile
95     });
96 }
97
98 @Test
99 public void testPrelievoImportoNegativo() {
100     ContoBancario conto = new ContoBancario("123456789", 1000);
101     Assertions.assertThrows(IllegalArgumentException.class, () -> {
102         conto.prelievo(-100);
103     });
104 }
105
106 @Test
107 public void testPrelievoSuccesso() {
108     ContoBancario conto = new ContoBancario("123456789", 1000);
109     double nuovoSaldo = conto.prelievo(500);
110     assertEquals(500, nuovoSaldo);
111     assertEquals(500, conto.getSaldo()); // Verify that the balance is updated correctly
112 }
113
```

Per i casi di test del metodo Trasferimento abbiamo:

T14: Destinazione è valida e importo è positivo e inferiore o uguale al saldo.

T15: Destinazione è valida e importo è superiore al saldo.

```

115 // Casi di test Trasferimento
116 @Test
117 public void testDestinazioneValidaImportoPositivoInferioreOUgualeAlSaldoTrasferimento() {
118     ContoBancario conto = new ContoBancario("123456789", 1000);
119     ContoBancario destinazione = new ContoBancario("987654321", 0);
120     conto.trasferimento(destinazione, 500);
121     Assertions.assertEquals(500, conto.getSaldo());
122     Assertions.assertEquals(500, destinazione.getSaldo());
123 }
124
125 @Test
126 public void testDestinazioneValidaImportoSuperioreAlSaldoTrasferimento() {
127     ContoBancario conto = new ContoBancario("123456789", 1000);
128     ContoBancario destinazione = new ContoBancario("987654321", 0);
129     Assertions.assertFalse(conto.trasferimento(destinazione, 1500));
130 }

```

Per il metodo CalcolaInteressi, abbiamo:

T16: TassoInteresse è negativo o zero.

```

32 // Casi di Test CalcolaInteressi
33 @Test
34 public void testTassoInteresseNegativoOZeroCalcolaInteressi() {
35     ContoBancario conto = new ContoBancario("123456789", 1000);
36     Assertions.assertThrows(IllegalArgumentException.class, () -> {
37         conto.calcolaInteressi(-1);
38     });
39 }

```

Il metodo CancellaInteressi:

T17: Verifica CancellaInteressi avvenuto

```

43 //test cancellaInteressi
44 @Test
45 public void testCancellaInteressi() {
46     // Crea un nuovo conto bancario con un saldo iniziale
47     ContoBancario conto = new ContoBancario("123456789", 1000);
48
49     // Calcola gli interessi con un tasso del 5%
50     conto.calcolaInteressi(5);
51
52     // Verifica che la mappa degli interessi non sia vuota
53     assertFalse(conto.getInteressi().isEmpty());
54
55     // Cancella tutti gli interessi
56     conto.cancellaInteressi();
57
58     // Verifica che la mappa degli interessi sia vuota
59     assertTrue(conto.getInteressi().isEmpty());
60 }

```

Infine, abbiamo GetInteressi:

T18: Recupero interessi

```

163 //test get interessi
164 @Test
165 public void testGetInteressi() {
166     // Crea un nuovo conto bancario con un saldo iniziale
167     ContoBancario conto = new ContoBancario("123456789", 1000);
168
169     // Verifica che la mappa degli interessi sia inizialmente vuota
170     assertTrue(conto.getInteressi().isEmpty());
171
172     // Calcola gli interessi con un tasso del 5%
173     conto.calcolaInteressi(5);
174
175     // Verifica che la mappa degli interessi non sia vuota
176     assertFalse(conto.getInteressi().isEmpty());
177
178     // Verifica che gli interessi siano stati calcolati correttamente
179     assertEquals(50.0, conto.getInteressi().get("123456789"));
180 }

```

Dopo aver eseguito il JUnit test abbiamo ottenuto:

Finished after 0,324 seconds

Runs: 18/18	Errors: 0	Failures: 0
-------------	-----------	-------------

ContoBancarioTest [Runner: JUnit 5] (0,012 s)

- testDestinazioneNull() (0,000 s)
- testPrelievoSuccesso() (0,000 s)
- testNumeroContoValidoSaldoPositivo() (0,000 s)
- testImportoNegativoOZeroDeposito() (0,000 s)
- testNumeroContoVuoto() (0,000 s)
- testNumeroContoValidoSaldoNegativo() (0,001 s)
- testTassoInteresseNegativo() (0,000 s)
- testDestinazioneValidaImportoSuperioreAlSaldoTrasferimento() (0,000 s)
- testTassoInteresseNegativoOZeroCalcolaInteressi() (0,000 s)
- testDestinazioneValidaImportoPositivoInferioreOugualeAlSaldoTrasferimento() (0,000 s)
- testPrelievoImportoNegativo() (0,000 s)
- testNumeroContoNull() (0,000 s)
- testSaldoNegativo() (0,001 s)
- testImportoPositivoInferioreOugualeAlSaldoDeposito() (0,001 s)
- testNumeroContoNullSaldoPositivo() (0,002 s)
- testGetInteressi() (0,002 s)
- testCancellaInteressi() (0,001 s)
- testPrelievoConSaldoInsufficiente() (0,001 s)

7 – Aumentare la suite di test con creatività ed esperienza

T19: NumeroConto con spazi bianchi

```

151 // T19 Test del NumeroConto con spazi bianchi
152 @Test
153 public void testNumeroContoConSpaziBianchi() {
154     Assertions.assertThrows(IllegalArgumentException.class, () -> {
155         new ContoBancario(" 123456789", 1000);
156     });
157 }

```

T20: NumeroConto con caratteri speciali

```

159 // T20 Test NumeroConto con caratteri speciali
160 @Test
161 public void testNumeroContoConCaratteriSpeciali() {
162     Assertions.assertThrows(IllegalArgumentException.class, () -> {
163         new ContoBancario("123456789!", 1000);
164     });
165 }

```

T21: Test dei rollback delle trasazioni

Verifica che il saldo del conto rimanga invariato se il trasferimento fallisce

```

67 // T21 Test dei rollback delle transazioni
68 @Test
69 public void testRollbackTrasferimento() {
70     ContoBancario conto = new ContoBancario("123456789", 1000);
71     ContoBancario destinazione = new ContoBancario("987654321", 500);
72
73     Assertions.assertThrows(IllegalArgumentException.class, () -> {
74         conto.trasferimento(destinazione, -100); // Importo negativo, dovrebbe fallire
75     });
76
77     // Verifica che i saldi rimangano invariati
78     Assertions.assertEquals(1000, conto.getSaldo());
79     Assertions.assertEquals(500, destinazione.getSaldo());
80 }

```

Il risultato JUnit di questi test è:

Finished after 0,268 seconds

Runs: 21/21	Errors: 0	Failures: 0
-------------	-----------	-------------

<div> <div>ContoBancarioTest [Runner: JUnit 5] (0,021 s)</div> <div> <div>testDestinazioneNull() (0,000 s)</div> <div>testPrelievoSuccesso() (0,000 s)</div> <div>testNumeroContoValidoSaldoPositivo() (0,000 s)</div> <div>testNumeroContoConCaratteriSpeciali() (0,000 s)</div> <div>testImportoNegativoOZeroDeposito() (0,000 s)</div> <div>testNumeroContoVuoto() (0,000 s)</div> <div>testNumeroContoValidoSaldoNegativo() (0,000 s)</div> <div>testTassoInteresseNegativo() (0,000 s)</div> <div>testDestinazioneValidaImportoSuperioreAlSaldoTrasferimento() (0,000 s)</div> <div>testTassoInteresseNegativoOZeroCalcolaInteressi() (0,002 s)</div> <div>testDestinazioneValidaImportoPositivoInferioreOUgualeAlSaldoTrasferimento() (0,001 s)</div> <div>testPrelievoImportoNegativo() (0,001 s)</div> <div>testNumeroContoNull() (0,001 s)</div> <div>testNumeroContoConSpaziBianchi() (0,001 s)</div> <div>testSaldoNegativo() (0,001 s)</div> <div>testImportoPositivoInferioreOUgualeAlSaldoDeposito() (0,000 s)</div> <div>testNumeroContoNullSaldoPositivo() (0,001 s)</div> <div>testGetInteressi() (0,002 s)</div> <div>testCancellaInteressi() (0,001 s)</div> <div>testPrelievoConSaldoInsufficiente() (0,001 s)</div> <div>testRollbackTrasferimento() (0,001 s)</div> </div> </div>

Structural testing

Una volta completati gli specification-based testing, si passa agli structural testing, eseguendo la suite di test con uno strumento di code coverage per verificare la copertura raggiunta.

Code coverage

È stata effettuata una prima analisi di code coverage con JaCoCo sui test Black-box e il codice risulta coperto al 100%, quindi coprendo tutte le righe di codice.

```
9.
10. //Costruttore per la classe ContoBancario
11. public ContoBancario(String NumeroConto, double saldo) {
12.     // Controllo di validità per il numero di conto
13.     if (NumeroConto == null || !NumeroConto.matches("\\d{9}")) { // il numero di conto deve essere composto da 9 cifre
14.         throw new IllegalArgumentException("Il numero di conto non è valido. Deve essere composto da 9 cifre.");
15.     }
16.     this.NumeroConto = NumeroConto;
17.     // Controllo di validità per il saldo
18.     if (saldo < 0) {
19.         throw new IllegalArgumentException("Il saldo iniziale non può essere negativo.");
20.     }
21.     this.NumeroConto = NumeroConto;
22.     this.saldo = saldo;
23.     this.interessi = new HashMap<>();
24. }
25.
26. //Metodo per effettuare un deposito sul conto bancario
27. public double deposito(double importo){
28.     if (importo <= 0) {
29.         throw new IllegalArgumentException("Importo di deposito non può essere negativo.");
30.     }
31.     saldo += importo;
32.     return saldo;
33. }
34.
35. //Metodo per effettuare un prelievo dal conto bancario
36. public double prelievo(double importo) {
37.     if (importo <= 0) {
38.         throw new IllegalArgumentException("Importo di prelievo non può essere negativo o zero.");
39.     }
40.     if (saldo >= importo) {
41.         saldo -= importo;
42.         return saldo;
43.     } else {
44.         throw new IllegalArgumentException("Saldo insufficiente per effettuare il prelievo.");
45.     }
46. }
47.
48. //Metodo per trasferire denaro da questo conto a un altro conto bancario
49. public boolean trasferimento(ContoBancario destinazione, double importo) {
50.     if (destinazione == null) {
51.         throw new IllegalArgumentException("Il conto di destinazione non può essere null.");
52.     }
53.     if (importo <= 0) {
54.         throw new IllegalArgumentException("Importo di trasferimento deve essere positivo.");
55.     }
56.     if (saldo >= importo) {
57.         saldo -= importo;
58.         destinazione.deposito(importo);
59.         return true; // Restituisci true se il trasferimento è avvenuto con successo
60.     } else {
61.         return false; // Restituisci false se il trasferimento non è riuscito
62.     }
63. }
64.
65. //Metodo per calcolare gli interessi sul saldo del conto bancario
66. public double calcolaInteressi(double tassoInteresse) {
67.     if (tassoInteresse <= 0) {
68.         throw new IllegalArgumentException("Il tasso di interesse non può essere negativo.");
69.     }
70.     double interesse = saldo * tassoInteresse / 100;
71.     interessi.put(NumeroConto, interesse);
72.     return interesse;
73. }
74.
75. public void cancellaInteressi() {
76.     interessi.clear();
77. }
78.
79. //Metodo per ottenere il saldo attuale del conto bancario
80. public double getSaldo(){
81.     return saldo;
82. }
83.
84. //Metodo per ottenere gli interessi calcolati per il conto bancario
85. public Map<String, Double> getInteressi() {
86.     return interessi;
87. }
88.
```

Poiché è stata raggiunta la copertura del 100% con gli Specification-based testing, possiamo concludere che gli structural testing sono completi. È sicuro che ogni riga del codice della classe `ContoBancario` sia stata eseguita almeno una volta ed è verificato che il codice si comporti come previsto. La suite di test è completa e copre tutti gli aspetti dell'implementazione.

HOMework 2

Property – based testing

Per il secondo homework si è deciso di utilizzare lo stesso codice, in quanto si ritiene che sia possibile effettuare su questo anche i property – based testing

1 - Esprimere le proprietà da testare

- *Proprietà dell'input*

1. **Deposito** (deposito_nonNegativo):
 - 1.1 Importo Positivo: Verifica che l'importo del deposito sia positivo. Se non lo è, colleziona la statistica "Negative Deposit".
2. **Prelievo** (prelievo_nonNegativo):
 - 2.1 Importo Positivo: Verifica che l'importo del prelievo sia positivo. Se non lo è, colleziona la statistica "Negative Withdrawal".
 - 2.2 Saldo Sufficiente: Verifica che il saldo sia sufficiente per coprire il prelievo. Se non lo è, colleziona la statistica "Insufficient Balance".
3. **Trasferimento** (trasferimento_valido):
 - 3.1 Importo Positivo: Verifica che l'importo del trasferimento sia positivo. Se non lo è, colleziona la statistica "Negative Transfer".
 - 3.2 Saldo Sufficiente: Verifica che il saldo del conto sorgente sia sufficiente per coprire il trasferimento. Se non lo è, colleziona la statistica "Insufficient Balance Transfer".
4. **Calcolo degli Interessi** (calcolaInteressi_valido):
 - 4.1 Tasso Positivo: Verifica che il tasso di interesse sia positivo. Se non lo è, colleziona la statistica "Negative Interest Rate".
5. **Costruttore** (testCostruttore):
 - 5.1 Numero di Conto Valido: Verifica che il numero di conto sia una stringa di 9 cifre. Se non lo è, genera un'eccezione `IllegalArgumentException`.
 - 5.2 Saldo Non Negativo: Verifica che il saldo iniziale del conto sia non negativo. Se non lo è, genera un'eccezione `IllegalArgumentException`.
6. **Cancella Interessi** (testCancellaInteressi):
 - 6.1 Interessi Annullati: Verifica che dopo aver cancellato gli interessi, la mappa degli interessi sia vuota.
7. **Destinazione Conto Nullo** (testNullDestinationAccount):
 - 7.2 Eccezione per Conto Nullo: Verifica che venga generata un'eccezione `IllegalArgumentException` quando il conto destinazione è nullo durante il trasferimento.
8. **Numero di Conto Non Valido** (testInvalidAccountNumber):
 - 8.1 Eccezione per Numero di Conto Non Valido: Verifica che venga generata un'eccezione `IllegalArgumentException` quando il numero di conto non è valido (non è una stringa di 9 cifre).

- Proprietà dell'output

1. **Deposito** (deposito_nonNegativo):
 - Saldo Aggiornato: Verifica che il saldo del conto sia aggiornato correttamente dopo il deposito.
2. **Prelievo** (prelievo_nonNegativo):
 - Saldo Aggiornato: Verifica che il saldo del conto sia aggiornato correttamente dopo il prelievo.
3. **Trasferimento** (trasferimento_valido):
 - Saldo Aggiornato: Verifica che i saldi dei conti sorgente e destinazione siano aggiornati correttamente dopo il trasferimento.
4. **Calcolo degli Interessi** (calcolaInteressi_valido):
 - Interessi Calcolati: Verifica che gli interessi siano calcolati e registrati correttamente

2 – Lasciare che sia il framework a scegliere i test

1. Deposito

Questo test ha l'obiettivo di verificare il corretto funzionamento del metodo deposito su un conto bancario, garantendo che l'importo depositato sia sempre non negativo

```
14 @Property(tries = 100)
15     @Report(Reporting.GENERATED)
16     @StatisticsReport(format = Histogram.class)
17     void deposito_nonNegativo(
18         @ForAll("numeroConto") String numeroConto,
19         @ForAll @Positive double saldo,
20         @ForAll double importo) {
21
22         ContoBancario conto = new ContoBancario(numeroConto, saldo);
23         double saldoPrecedente = conto.getSaldo();
24
25         if (importo <= 0) {
26             assertThrows(IllegalArgumentException.class, () -> conto.deposito(importo));
27             Statistics.collect("Negative Deposit", importo);
28         } else {
29             double saldoDopoDeposito = conto.deposito(importo);
30             assertEquals(saldoPrecedente + importo, saldoDopoDeposito);
31             Statistics.collect("Importo di deposito", importo);
32         }
33     }
34 }
```

2. Prelievo

Questo test mira a verificare il corretto comportamento del metodo, gestendo l'operazione di denaro da un conto bancario. Se l'importo è negativo o zero, viene lanciata un'eccezione quando si tenta di effettuare un prelievo. Se l'importo è positivo e il saldo sufficiente viene eseguito il prelievo.

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void prelievo_nonNegativo(
    @ForAll("numeroConto") String numeroConto,
    @ForAll @Positive double saldo,
    @ForAll double importo) {

    ContoBancario conto = new ContoBancario(numeroConto, saldo);
    double saldoPrecedente = conto.getSaldo();

    if (importo <= 0) {
        // Se l'importo è negativo o zero, lancia un'eccezione IllegalArgumentException
        assertThrows(IllegalArgumentException.class, () -> conto.prelievo(importo));
        Statistics.collect("Negative Withdrawal");
    } else if (saldo >= importo) {
        // Se l'importo è positivo e il saldo è sufficiente, effettua il prelievo
        double saldoDopoPrelievo = conto.prelievo(importo);
        assertEquals(saldoPrecedente - importo, saldoDopoPrelievo);
        Statistics.collect("Valid Withdrawal");
    } else {
        // Se l'importo è positivo ma il saldo è insufficiente, lancia un'eccezione IllegalArgumentException
        assertThrows(IllegalArgumentException.class, () -> conto.prelievo(importo));
        Statistics.collect("Insufficient Balance");
    }
}
```

3. Trasferimento

Questo test è stato progettato per verificare il comportamento del metodo, il quale gestisce il trasferimento di denaro da un conto sorgente ad un conto destinazione. Se l'importo è negativo o zero, o il conto destinazione è nullo, viene lanciata un'eccezione. Se l'importo è positivo e il conto destinazione è valido, viene eseguito il metodo tra il conto sorgente e quello di destinazione.

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void trasferimento_valido(
    @ForAll("numeroConto") String numeroContoSorgente,
    @ForAll("numeroConto") String numeroContoDestinazione,
    @ForAll @Positive double saldoSorgente,
    @ForAll @Positive double saldoDestinazione,
    @ForAll double importo) {

    ContoBancario contoSorgente = new ContoBancario(numeroContoSorgente, saldoSorgente);
    ContoBancario contoDestinazione = new ContoBancario(numeroContoDestinazione, saldoDestinazione);

    if (importo <= 0 || contoDestinazione == null) {
        assertThrows(IllegalArgumentException.class, () -> contoSorgente.trasferimento(contoDestinazione, importo));
        Statistics.collect("Trasferimento negativo", importo);
    } else {
        boolean trasferito = contoSorgente.trasferimento(contoDestinazione, importo);

        if (trasferito) {
            assertEquals(saldoSorgente - importo, contoSorgente.getSaldo(), 0.001);
            assertEquals(saldoDestinazione + importo, contoDestinazione.getSaldo(), 0.001);
            Statistics.collect("Trasferimento positivo", importo);
        } else {
            assertEquals(saldoSorgente, contoSorgente.getSaldo(), 0.001);
            assertEquals(saldoDestinazione, contoDestinazione.getSaldo(), 0.001);
            Statistics.collect("Trasferimento fallito", importo);
        }
    }
}
```

4. Calcolo interessi

Questo test aiuta a garantire che il metodo gestisca correttamente i tassi di interesse positivi e calcoli correttamente gli interessi, assicurandosi che il saldo e gli interessi siano aggiornati correttamente nel conto. Se il tasso di interesse è negativo o zero, viene lanciata un'eccezione. Se il tasso di interesse è positivo, viene eseguito il metodo

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void calcolaInteressi_valido(
    @ForAll("numeroConto") String numeroConto,
    @ForAll @Positive double saldo,
    @ForAll double tassoInteresse) {

    ContoBancario conto = new ContoBancario(numeroConto, saldo);

    if (tassoInteresse <= 0) {
        assertThrows(IllegalArgumentException.class, () -> conto.calcolaInteressi(tassoInteresse));
        Statistics.collect("Negative Interest Rate");
    } else {
        double interesseCalcolato = conto.calcolaInteressi(tassoInteresse);

        assertTrue(interesseCalcolato >= 0);
        assertTrue(conto.getInteressi().containsKey(numeroConto));
        assertEquals(interesseCalcolato, conto.getInteressi().get(numeroConto), 0.001);
        Statistics.collect("Valid Interest Calculation");
    }
}
```

5. Costruttore

Questo test assicura che il costruttore gestisca correttamente i casi in cui il numero di conto non è valido o il saldo iniziale è negativo, garantendo che il conto venga creato correttamente solo con parametri validi.

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void testCostruttore(
    @ForAll("numeroConto") String numeroConto,
    @ForAll double saldo) {

    if (numeroConto == null || !numeroConto.matches("\\d{9}")) {
        assertThrows(IllegalArgumentException.class, () -> new ContoBancario(numeroConto, saldo));
        Statistics.collect("Invalid Account Number");
    } else if (saldo < 0) {
        assertThrows(IllegalArgumentException.class, () -> new ContoBancario(numeroConto, saldo));
        Statistics.collect("Negative Initial Balance");
    } else {
        ContoBancario conto = new ContoBancario(numeroConto, saldo);
        assertEquals(numeroConto, conto.NumeroConto);
        assertEquals(saldo, conto.getSaldo());
        Statistics.collect("Valid Account Creation");
    }
}
```

6. CancellaInteressi

L'obiettivo di questo test è verificare il corretto funzionamento del metodo cancellaInteressi, nel rimuovere gli interessi accumulati su un conto bancario.

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void testCancellaInteressi(
    @ForAll("numeroConto") String numeroConto,
    @ForAll @Positive double saldo,
    @ForAll @Positive double tassoInteresse) {

    ContoBancario conto = new ContoBancario(numeroConto, saldo);
    conto.calcolaInteressi(tassoInteresse);
    assertFalse(conto.getInteressi().isEmpty());

    conto.cancellaInteressi();
    assertTrue(conto.getInteressi().isEmpty());
    Statistics.collect("Interest Cleared");
}
```

7. Destinazione conto Nullo

Questo verifica che il sistema reagisca in modo appropriato e prevedibile quando un conto di destinazione nullo viene passato come argomento al metodo di trasferimento.

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void testNullDestinationAccount(
    @ForAll("numeroConto") String numeroContoSorgente,
    @ForAll @Positive double saldoSorgente,
    @ForAll double importo) {

    ContoBancario contoSorgente = new ContoBancario(numeroContoSorgente, saldoSorgente);
    assertThrows(IllegalArgumentException.class, () -> contoSorgente.trasferimento(null, importo));
    Statistics.collect("Null Destination Account");
}
```

8. Numero di conto non valido

L'obiettivo di questo test è verificare che il costruttore gestisca correttamente il caso in cui un numero di conto non valido venga passato come argomento. Il test si aspetta che venga lanciata un'eccezione quando il numero di conto non valido viene utilizzato.

```
@Property(tries = 100)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void testInvalidAccountNumber(
    @ForAll("invalidAccountNumber") String numeroConto,
    @ForAll @Positive double saldo) {

    assertThrows(IllegalArgumentException.class, () -> new ContoBancario(numeroConto, saldo));
    Statistics.collect("Invalid Account Number");
}
```

3 – Statistiche

1. Deposito

#	label	count	
0	Importo di deposito 0.01	4	####
1	Importo di deposito 1.0	3	###
2	Importo di deposito 1.7976931348623157E308	2	##
3	Importo di deposito 0.12	2	##
4	Importo di deposito 727.19	1	#
5	Importo di deposito 6.93	1	#
6	Importo di deposito 21.49	1	#
7	Importo di deposito 1.1477992215999357E51	1	#
8	Importo di deposito 3.62	1	#
9	Importo di deposito 7.244526903839519E171	1	#
10	Importo di deposito 2.70938827204072576E17	1	#
11	Importo di deposito 5.227099033808372E41	1	#
12	Importo di deposito 43.97	1	#
13	Importo di deposito 3.766301854170145E271	1	#
14	Importo di deposito 2.74794926061751E142	1	#
15	Importo di deposito 3.197283121447172E156	1	#
16	Importo di deposito 1.4564127909975274E87	1	#
17	Importo di deposito 6.156063982327245E74	1	#
18	Importo di deposito 0.51	1	#
19	Importo di deposito 1.031509691267754E205	1	#
20	Importo di deposito 1.51	1	#
21	Importo di deposito 0.61	1	#
22	Importo di deposito 1.3851247999105193E272	1	#
23	Importo di deposito 0.31	1	#
24	Importo di deposito 135611.42	1	#
25	Importo di deposito 1.436921857809629E142	1	#
26	Importo di deposito 1363.5	1	#
27	Importo di deposito 1.46	1	#
28	Importo di deposito 4.89	1	#
29	Importo di deposito 55638.6	1	#
30	Importo di deposito 3.96597474346504E271	1	#
31	Importo di deposito 1.106500422071887E52	1	#
32	Importo di deposito 1.5261408310845206E100	1	#
33	Importo di deposito 3.3190011614470946E113	1	#
34	Importo di deposito 0.08	1	#
35	Importo di deposito 3.69	1	#
36	Importo di deposito 11.1	1	#
37	Negative Deposit 0.0	5	####
38	Negative Deposit -1.0	4	####
39	Negative Deposit -1.7976931348623157E308	4	####
40	Negative Deposit -0.01	3	###
41	Negative Deposit -0.77	2	##
42	Negative Deposit -4.391067478355759E62	1	#
43	Negative Deposit -1.26	1	#
44	Negative Deposit -195530.15	1	#
45	Negative Deposit -4.8289487858114635E272	1	#
46	Negative Deposit -4.058251830673754E220	1	#
47	Negative Deposit -5.177337768704022E289	1	#
48	Negative Deposit -2.4749863355519172E51	1	#
49	Negative Deposit -1.8039087163204778E255	1	#
50	Negative Deposit -1.3784850230066884E127	1	#

Durante l'esecuzione del test sono stati ottenuti i seguenti risultati:

- Scenari di deposito con importi specifici: il test ha prodotto 4 casi di depositi con l'importo di 0.01, 3 casi di depositi con l'importo 1.0 e diversi altri importi di deposito sono stati testati, ciascuno con un caso singolo, evidenziando la diversità degli scenari esaminati.
- Scenari di tentativi di deposito negativi: test ha rilevato 4 casi di tentativi di deposito con un importo negativo di -1.0, diversi altri importi negativi sono stati testati, ciascuno con un caso singolo, per verificare la gestione corretta di tentativi di deposito non validi.

Questo test è stato eseguito con successo su 100 tentativi, confermando la correttezza del metodo 'deposito' nel gestire sia i depositi validi che quelli non validi. I casi di Negative deposit evidenziano la capacità del sistema di gestire correttamente i tentativi di deposito con importi negativi, rifiutando tali operazioni e sollevando l'eccezione appropriata.

2. Prelievo

#	label	count	
0	Insufficient Balance	20	#####
1	Negative Withdrawal	58	#####
2	Valid Withdrawal	22	#####

Durante il test si sono ottenuti i seguenti risultati:

- Insufficient Balance: sono stati registrati 20 casi in cui il metodo prelievo ha lanciato correttamente un'eccezione `IllegalArgumentException` a causa di un saldo insufficiente per effettuare il prelievo. Questo scenario è cruciale per garantire che il sistema gestisca correttamente i casi in cui l'utente tenta di prelevare più fondi di quelli disponibili sul conto.
- Negative Withdrawal: Sono stati registrati 58 casi in cui il metodo prelievo ha lanciato un'eccezione `IllegalArgumentException` a causa di un importo negativo
- Valid Withdrawal: Sono stati registrati 22 casi in cui il prelievo è stato eseguito correttamente con un saldo sufficiente nel conto e un importo valido. Questa statistica verifica che il sistema esegua correttamente i prelievi quando le condizioni sono adeguate, contribuendo a garantire che la funzionalità principale del metodo di prelievo sia conforme alle aspettative.

L'analisi delle statistiche indica una distribuzione significativa tra i vari tipi di prelievi eseguiti durante il test.

3. Trasferimento

#	label	count
0	Trasferimento fallito 1.0	3
1	Trasferimento fallito 9.627052519677505E51	1
2	Trasferimento fallito 3.902498861580381E254	1
3	Trasferimento fallito 2.0808865483865833E255	1
4	Trasferimento fallito 1.7976931348623157E308	1
5	Trasferimento fallito 3.421575255131527E41	1
6	Trasferimento fallito 3.3493708079373584E157	1
7	Trasferimento fallito 484.09	1
8	Trasferimento fallito 8.7947089695289E11	1
9	Trasferimento fallito 10.78	1
10	Trasferimento fallito 8.493140527905314E23	1
11	Trasferimento fallito 0.15	1
12	Trasferimento fallito 8.799793155646251E307	1
13	Trasferimento fallito 9.48093978171365E127	1
14	Trasferimento fallito 8.12513303969261E51	1
15	Trasferimento fallito 1.2890148155252554E63	1
16	Trasferimento fallito 37561.14	1
17	Trasferimento fallito 1.0108299439503234E289	1
18	Trasferimento fallito 4.759152426489949E51	1
19	Trasferimento fallito 1.5712706342264274E308	1
20	Trasferimento fallito 1.36617104312486672E17	1
21	Trasferimento fallito 3.9439404853824685E51	1
22	Trasferimento fallito 4.849367261247072E254	1
23	Trasferimento fallito 2.7160780633018946E307	1
24	Trasferimento fallito 2.890421991469814E32	1
25	Trasferimento fallito 2.3279182302269355E23	1
26	Trasferimento negativo 0.0	7
27	Trasferimento negativo -1.0	6
28	Trasferimento negativo -1.7976931348623157E308	2
29	Trasferimento negativo -0.01	2
30	Trasferimento negativo -1.3030029613353917E87	1
31	Trasferimento negativo -1.0036977226925216E113	1
32	Trasferimento negativo -4.628948711014953E41	1
33	Trasferimento negativo -6303060.26	1
34	Trasferimento negativo -1.3666991056574274E32	1
60	Trasferimento positivo 1.0	3
61	Trasferimento positivo 5.313242757783957E254	1
62	Trasferimento positivo 1.1440018668032602E128	1
63	Trasferimento positivo 1827.79	1
64	Trasferimento positivo 2.453324509899928E32	1
65	Trasferimento positivo 191.19	1
66	Trasferimento positivo 0.03	1
67	Trasferimento positivo 5.2299631798403E11	1
68	Trasferimento positivo 6.9523571338655E11	1
69	Trasferimento positivo 2.4065711318324715E32	1
70	Trasferimento positivo 8.736628934808721E62	1
71	Trasferimento positivo 0.5	1
72	Trasferimento positivo 2.383241795130732E127	1
73	Trasferimento positivo 5.5254563193586E11	1
74	Trasferimento positivo 1.3363343427156221E290	1
75	Trasferimento positivo 9.50177994749851E99	1
76	Trasferimento positivo 6.548058488501725E172	1
77	Trasferimento positivo 204219.73	1
78	Trasferimento positivo 1.88	1
79	Trasferimento positivo 69872.95	1
80	Trasferimento positivo 489.19	1

Durante l'esecuzione del test sono stati raccolti i seguenti risultati statistici:

- Scenari di trasferimento falliti: il test ha evidenziato diversi casi di trasferimento fallito con importo specifico, in cui il trasferimento di fondi non è stato completato correttamente. Questi casi hanno sollevato eccezioni o gestito il fallimento del trasferimento.
- Scenari di trasferimento negativo: il test ha rilevato diversi casi in cui è stato tentato un trasferimento con importo negativo. Questi casi hanno sollevato eccezioni appropriatamente per prevenire operazioni non valide.
- Scenari di trasferimento positivo: il test ha confermato diversi casi in cui il trasferimento di fondi è stato eseguito con successo. Questi casi hanno mostrato che il metodo trasferimento è in grado di gestire correttamente transazioni valide tra conti bancari.

Questo test ha prodotto risultati diversificati, ma complessivamente positivi.

I casi di Trasferimento Fallito dimostrano la capacità del sistema di gestire eccezioni e di interrompere il trasferimento quando necessario. I numerosi casi di Trasferimento Negativo confermano la robustezza del sistema nel rifiutare trasferimenti con importi negativi.

D'altra parte, i casi di Trasferimento Positivo evidenziano che il metodo trasferimento funziona correttamente nel trasferire fondi tra conti bancari quando gli importi sono validi e le condizioni lo consentono.

4. Calcolo degli interessi

#	label	count	
0	Negative Interest Rate	47	#####
1	Valid Interest Calculation	53	#####

Durante l'esecuzione del test, sono stati raccolti i seguenti risultati:

- Scenario con tasso di interesse negativo: il test ha rilevato 47 casi in cui è stato fornito un tasso di interesse negativo. In tutti questi casi, il sistema ha gestito correttamente l'eccezione, rifiutando il calcolo degli interessi non validi.
- Scenario con calcolo valido degli interessi: il test ha generato 53 casi di calcolo corretto degli interessi. In questi casi, il metodo `calcolaInteressi` ha restituito un valore non negativo e ha aggiornato correttamente la mappa degli interessi associata al conto bancario.

È stato eseguito con successo su 100 tentativi, durante i quali sono stati esplorati scenari critici come il trattamento di tassi di interesse negativi. La presenza di 47 casi di tassi di interesse negativi evidenzia la capacità del sistema di gestire correttamente input non validi, i 53 casi di calcolo valido degli interessi confermano che il metodo è in grado di eseguire il calcolo corretto degli interessi quando il tasso fornito è valido, contribuendo alla corretta gestione e aggiornamento degli interessi accumulati sul conto bancario.

5. Costruttore

#	label	count	
0	Negative Initial Balance	45	#####
1	Valid Account Creation	55	#####

Durante l'esecuzione del test sono stati ottenuti i seguenti risultati statistici:

- Negative Initial Balance: Il test ha evidenziato 45 casi di tentativi di creazione di conti con saldo iniziale negativo. Questo risultato è significativo poiché conferma che il sistema impedisce la creazione di conti con saldo iniziale non valido, contribuendo alla sicurezza e alla coerenza dei dati finanziari.
- Valid Account Creation: Sono stati registrati 55 casi di creazione di conti bancari validi. Questa statistica è cruciale poiché verifica che il sistema permetta la creazione di conti con saldo iniziale valido, assicurando che i nuovi conti vengano istanziati correttamente e siano utilizzabili nel sistema. I risultati raccolti durante il test indicano che il costruttore della classe `ContoBancario` funziona come previsto nel gestire sia i casi di saldo iniziale negativo che quelli validi. La presenza di 14 edge cases testati e tutti riusciti conferma che il sistema è stato esaminato anche in situazioni limite, assicurando una maggiore affidabilità e robustezza.

6. Cancella Interessi

#	label	count	
0	Interest Cleared	100	#####

Dall'esecuzione del test abbiamo ottenuto i seguenti risultati:

Interest Cleared: Sono stati registrati 100 casi in cui il metodo `cancellaInteressi` è stato eseguito con successo, ovvero gli interessi accumulati sono stati rimossi correttamente dal conto bancario. I risultati ottenuti durante il test indicano che il metodo `cancellaInteressi` ha funzionato correttamente in tutti i 100 casi testati. Questo dimostra che il sistema è in grado di gestire efficacemente la rimozione degli interessi accumulati su un conto bancario.

7. Destinazione conto nullo

#	label	count
0	Null Destination Account	100

Durante l'esecuzione del test con 100 tentativi, sono stati ottenuti i seguenti risultati:
- il test ha prodotto 100 casi di tentativi falliti dovuti a un account di destinazione nullo. Questo risultato evidenzia che il sistema ha gestito correttamente tutti i tentativi in cui è stato fornito un account destinazione nullo, rifiutando il trasferimento e sollevando l'eccezione.

8. Numero di conto non valido

#	label	count
0	Invalid Account Number	100

Durante l'esecuzione del test sono stati ottenuti i seguenti risultati statistici:
- Invalid Account Number: sono stati registrati 100 casi di tentativi di creazione di conti bancari con numeri di conto non validi. Questa statistica è significativa in quanto conferma che il sistema riconosce e gestisce correttamente i casi in cui il numero di conto fornito non rispetta i requisiti stabiliti, evitando la creazione di conti non utilizzabili o potenzialmente problematici. I risultati raccolti durante il test indicano che il costruttore della classe ContoBancario è stato efficace nel rilevare e trattare correttamente i numeri di conto non validi. Tutti i 6 edge cases testati hanno avuto esito positivo, dimostrando che il sistema è stato sottoposto anche a situazioni limite per garantire la sua robustezza e affidabilità.

Finished after 1,824 seconds

Runs: 8/8 Errors: 0 Failures: 0


- ContoBancarioTest [Runner: JUnit 5] (1,525 s)
 - trasferimento valido (0,650 s)
 - prelievo nonNegativo (0,181 s)
 - testNullDestinationAccount (0,159 s)
 - calcolaInteressi valido (0,132 s)
 - deposito nonNegativo (0,107 s)
 - testCostruttore (0,097 s)
 - testInvalidAccountNumber (0,100 s)
 - testCancellaInteressi (0,092 s)

4 – Code coverage

```

15. public ContoBancario(String NumeroConto, double saldo) {
16.     // Controllo di validità per il numero di conto
17.     if (NumeroConto == null || !NumeroConto.matches("\\d{9}")) { // il numero di conto deve essere composto da 9 cifre
18.         throw new IllegalArgumentException("Il numero di conto non è valido. Deve essere composto da 9 cifre.");
19.     }
20.     this.NumeroConto = NumeroConto;
21.     // Controllo di validità per il saldo
22.     if (saldo < 0) {
23.         throw new IllegalArgumentException("Il saldo iniziale non può essere negativo.");
24.     }
25.     this.NumeroConto = NumeroConto;
26.     this.saldo = saldo;
27.     this.interessi = new HashMap<>();
28. }
29.
30. //Metodo per effettuare un deposito sul conto bancario
31. public double deposito(double importo){
32.     if (importo <= 0) {
33.         throw new IllegalArgumentException("Importo di deposito non può essere negativo.");
34.     }
35.     saldo += importo;
36.     return saldo;
37. }
38.
39. //Metodo per effettuare un prelievo dal conto bancario
40. public double prelievo(double importo) {
41.     if (importo <= 0) {
42.         throw new IllegalArgumentException("Importo di prelievo non può essere negativo o zero.");
43.     }
44.     if (saldo >= importo) {
45.         saldo -= importo;
46.         return saldo;
47.     } else {
48.         throw new IllegalArgumentException("Saldo insufficiente per effettuare il prelievo.");
49.     }
50. }
51.
52. //Metodo per trasferire denaro da questo conto a un altro conto bancario
53. public boolean trasferimento(ContoBancario destinazione, double importo) {
54.     if (destinazione == null) {
55.         throw new IllegalArgumentException("Il conto di destinazione non può essere null.");
56.     }
57.     if (importo <= 0) {
58.         throw new IllegalArgumentException("Importo di trasferimento deve essere positivo.");
59.     }
60.     if (saldo >= importo) {
61.         saldo -= importo;
62.         destinazione.deposito(importo);
63.         return true; // Restituisci true se il trasferimento è avvenuto con successo
64.     } else {
65.         return false; // Restituisci false se il trasferimento non è riuscito
66.     }
67. }
68.
69. //Metodo per calcolare gli interessi sul saldo del conto bancario
70. public double calcolaInteressi(double tassoInteresse) {
71.     if (tassoInteresse <= 0) {
72.         throw new IllegalArgumentException("Il tasso di interesse non può essere negativo.");
73.     }
74.     double interesse = saldo * tassoInteresse / 100;
75.     interessi.put(NumeroConto, interesse);
76.     return interesse;
77. }
78.
79. public void cancellaInteressi() {
80.     interessi.clear();
81. }
82.
83. //Metodo per ottenere il saldo attuale del conto bancario
84. public double getSaldo(){
85.     return saldo;
86. }
87.
88. //Metodo per ottenere gli interessi calcolati per il conto bancario
89. public Map<String, Double> getInteressi() {
90.     return interessi;
91. }

```

Element	Missed Instructions	Cov.
 ContoBancario	<div style="width: 100%;"></div>	100%
Total	0 of 154	100%