Integrazione e Test di Sistemi Software

- ➔ In seguito allo svolgimento del primo home work in comune, la disgregazione del gruppo è avvenuta con comunicazione inviata via mail.
- ➔ L'approvazione dei codici riguarda l'Homework 1 e 2. La scelta è avvenuta considerando il numero dei componenti del gruppo.

BREAK TEAM

HOMEWORK 1: BINDING DINAMICO - VENDITA - VENDITA SCONTATA

Realizzato da: Porro Marco, Zappimpulso Antonella, Mansi Stefano

SINGLEBREAK

HOMEWORK 2: DiscountCalculator

HOMEWORK 3: UsernameValidation

Realizzato da: Zappimpulso Antonella

HOMEWORK 1

TESTING WORKFLOW

1. Comprendere i requisiti

Il nostro codice è composto da tre classi:

- Vendita: costituisce la vendita di un singolo elemento. Ignora tasse, sconti e qualsiasi elemento che potrebbe variare il prezzo.
  Il prezzo non può essere negativo e il nome non può essere una stringa vuota.
- VenditaScontata: è una classe che estende la classe Vendita, aggiungendo lo sconto che non può essere negativo.
- BindingDinamicoDemo: è la classe Main in cui vengono creati gli oggetti “Vendita” e “VenditaScontata”. L'obiettivo è quello di comprendere se il prodotto con prezzo scontato risulta conveniente rispetto al prodotto non scontato.

IN DETTAGLIO LA CLASSE BINDINGDINAMICODEMO

```
public static String methodPrezzoMinore (Vendita semplice,
VenditaScontata scontato){
  System.out.println(semplice.toString());
  System.out.println(scontato.toString());

  if (scontato.minoreDi(semplice))
     return "Il prodotto scontato costa meno.";
  else
     return"Il prodotto scontato non costa meno.";
}
```

- Dati i due oggetti di vendita: semplice e scontato, l’obiettivo di questo metodo è quello di comprendere se un prodotto scontato costa meno di un prodotto diverso non scontato.
- Il metodo riceve: “Vendita semplice” e “VenditaScontata scontato”
- Il metodo restituisce una stringa che indica il risultato dell’elaborazione.

```
public static String methodPrezzoUguale (Vendita
prezzoNormale, VenditaScontata prezzoSpeciale){
  System.out.println(prezzoNormale.toString());
  System.out.println(prezzoSpeciale.toString());

  if (prezzoSpeciale.uguaglianzaVendite(prezzoNormale))
     return "Il costo totale e' lo stesso.";
  else
     return "Il costo totale e' diverso.";
}
```

- Dati i due oggetti di vendita: semplice e scontato, l'obiettivo di questo metodo è quello di comprendere il costo totale di un prodotto è uguale o diverso.
- Il metodo riceve: "Vendita prezzoNormale" e "VenditaScontata prezzoSpeciale".
- Il metodo restituisce una stringa che indica il risultato dell'elaborazione.

IN DETTAGLIO LA CLASSE VENDITA:

```
public boolean setPrezzo(double nuovoPrezzo) {
     if (nuovoPrezzo >= 0) {
        prezzo = nuovoPrezzo;
        return true;
     } else {
        System.out.println("Errore: Prezzo negativo.");
        return false;
     }
}
```

- Questo metodo avvalora un oggetto di tipo vendita.
  Controlla se il valore è maggiore o uguale a zero, altrimenti indica la presenza di un valore errato.
- Il metodo riceve: "nuovoPrezzo"
- Il metodo restituisce un boolean: true se l'operazione è andata a buon fine, false in caso contrario.

```
public boolean setNome(String nuovoNome) {
  if(nuovoNome != null && nuovoNome != "") {
     nome = nuovoNome;
     return true;
  }
  else {
     System.out.println("Errore: nome errato.");
     return false;
  }
}
```

- Questo metodo avvalora un oggetto di tipo vendita.
  Controlla se il valore non è una stringa vuota o nulla, altrimenti indica la presenza di un valore errato.
- Il metodo riceve: "nuovoNome"
- Il metodo restituisce un boolean: true se l'operazione è andata a buon fine, false in caso contrario.

```
public boolean uguaglianzaVendite(Vendita altraVendita) {
  if(altraVendita == null)
     return false;
```

```
  else
     return (nome.equals(altraVendita.nome) &&
           totale() == altraVendita.totale());
}
```

.

- Questo metodo controlla se due oggetti di tipo vendita sono uguali (si definiscono uguali se hanno lo stesso nome e stesso prezzo).
- Il metodo riceve: “altraVendita”
- Il metodo restituisce un boolean: true se i nomi e i totali delle vendite sono gli stessi, altrimenti restituisce false, restituisce false anche se l’oggetto è vuoto.

```
public boolean minoreDi(Vendita altraVendita) {
  if(altraVendita == null) {
     System.out.println("Errore: oggetto Vendita  null.");
     return false;
  }
  //else
  return (totale() < altraVendita.totale());
}
```

- Questo metodo controlla se il prezzo totale dell’oggetto è minore rispetto al prezzo totale del secondo oggetto.
- Il metodo riceve: “altraVendita”
- Il metodo restituisce un boolean: true se il totale dell'oggetto è minore del totale dell'oggetto altraVendita; altrimenti restituisce false, restituisce false anche se l’oggetto altraVendita è nullo.

```
public boolean equals(Object altroOggetto) {
  if(altroOggetto == null)
     return false;
  else if(!(altroOggetto instanceof Vendita))
     return false;
  else {
     Vendita altraVendita = (Vendita)altroOggetto;
     return (nome.equals(altraVendita.nome) &&
        (prezzo == altraVendita.prezzo));
  }
}
```

- Questo metodo controlla se due oggetti di tipo vendita sono uguali (si definiscono uguali se hanno lo stesso nome e stesso prezzo).
- Il metodo riceve: “altraOggetto”
- Il metodo restituisce un boolean: true se i nomi e i totali delle vendite sono gli stessi, altrimenti restituisce false, restituisce false anche se l’oggetto è vuoto e anche se l’oggetto non è un’ istanza di tipo Vendita.

IN DETTAGLIO LA CLASSE VENDITASCONTATA:

```
public double totale() {
```

```
  double frazione = sconto / 100;
  return (1 - frazione) * getPrezzo();
}
```

- Questo metodo calcola il prezzo totale del prodotto calcolando lo sconto e sottraendolo
- il metodo non riceve nulla
- il metodo restituisce un double: corrispondente al prezzo totale del prodotto

```
public boolean setSconto(double nuovoSconto) {
  if(nuovoSconto >= 0) {
     sconto = nuovoSconto;
     return true;
  }
  else {
     System.out.println("Errore: sconto negativo.");
     return false;
  }
}
```

- Questo metodo avvalora un oggetto di tipo vendita.
  Controlla se il valore è maggiore o uguale a zero, altrimenti indica la presenza di un valore errato.
- Il metodo riceve: “nuovoScontato”
- Il metodo restituisce un boolean: true se l’operazione è andata a buon fine, false in caso contrario.

```
public boolean equals(Object altroOggetto) {
  if (altroOggetto == null)
     return false;
  else if (!(altroOggetto instanceof VenditaScontata))
     return false;
  else {
     VenditaScontata altraVenditaScontata =
(VenditaScontata)altroOggetto;
     return (super.equals(altraVenditaScontata) && sconto ==
altraVenditaScontata.sconto);
  }
}
```

- Questo metodo controlla se due oggetti di tipo venditaScontata sono uguali (si definiscono uguali se hanno lo stesso nome, stesso prezzo e stesso sconto).
- Il metodo riceve: “altraOggetto”

- Il metodo restituisce un boolean: true se i nomi, i totale e gli sconti delle vendite sono gli stessi, altrimenti restituisce false, restituisce false anche se l'oggetto è vuoto e anche se l'oggetto non è un'istanza di tipo Vendita.

TEST

```
String methodPrezzoMinore (Vendita semplice, VenditaScontata scontato)
```

INPUT
parametro di sinistra:

1. Empty
2. Null
3. Avvalorato con prezzo minore rispetto al prodotto scontato
4. Avvalorato con prezzo maggiore rispetto al prodotto scontato
5. Avvalorato con prezzo uguale rispetto al prodotto scontato

parametro di destra:

1. Empty
2. Null
3. Avvalorato con prezzo minore rispetto al prodotto non scontato
4. Avvalorato con prezzo maggiore rispetto al prodotto non scontato
5. Avvalorato con prezzo uguale rispetto al prodotto non scontato

```
String methodPrezzoUguale (Vendita prezzoNormale, VenditaScontata prezzoSpeciale)
```

INPUT
parametro di sinistra:

1. Empty
2. Null
3. Avvalorato con prezzo minore rispetto al prodotto scontato
4. Avvalorato con prezzo maggiore rispetto al prodotto scontato
5. Avvalorato con prezzo uguale rispetto al prodotto scontato

parametro di destra:

1. Empty
2. Null
3. Avvalorato con prezzo minore rispetto al prodotto non scontato
4. Avvalorato con prezzo maggiore rispetto al prodotto non scontato
5. Avvalorato con prezzo uguale rispetto al prodotto non scontato

```
boolean setPrezzo(double nuovoPrezzo)
```

INPUT
parametro:

1. Empty
2. Null
3. Valore singolo
4. Valore multiplo

5. Valore negativo
6. Valore uguale a zero
7. Valore con la virgola
8. Massimo valore Double
9. MInimo valore Double

```
boolean setNome(String nuovoNome)
```

INPUT
parametro:

1. Empty
2. Null
3. String length 1
4. String length > 1

```
boolean uguaglianzaVendite(Vendita altraVendita)
```

INPUT
parametro:

1. Empty
2. Null
3. Avvalorato con nome e prezzo uguale
4. Avvalorato con nome uguale e prezzo diverso
5. Avvalorato con prezzo uguale e nome diverso
6. Avvalorato con nome e prezzo diverso

```
boolean minoreDi(Vendita altraVendita)
```

INPUT
parametro:

1. Empty
2. Null
3. Avvalorato con prezzo maggiore
4. Avvalorato con prezzo minore
5. Avvalorato con prezzo uguale

```
boolean equals(Object altroOggetto)
```

INPUT
parametro:

1. Empty
2. Null
3. Passando un oggetto diverso dal tipo Vendita
4. Avvalorato con nome e prezzo uguale
5. Avvalorato con nome uguale e prezzo diverso
6. Avvalorato con prezzo uguale e nome diverso
7. Avvalorato con nome e prezzo diverso

```
boolean setSconto(double nuovoSconto)
```

INPUT
parametro:

1. Empty
2. Null
3. Valore singolo
4. Valore multiplo
5. Valore negativo
6. Valore uguale a zero
7. Valore con la virgola
8. Massimo valore Double
9. Minimo Valore Double
10. Valore superiore a 100

```
public boolean equals(Object altroOggetto)
```

1. Empty
2. Null
3. Passando un oggetto diverso dal tipo VenditaScontata
4. Avvalorato con nome, prezzo e sconto uguale
5. Avvalorato con nome, sconto uguale e prezzo diverso
6. Avvalorato con prezzo, sconto uguale e nome diverso
7. Avvalorato con nome, prezzo uguale e sconto diverso
8. Avvalorato con nome, prezzo e sconto diverso

6. TEST CASES

IN DETTAGLIO VENDITATEST

```
public class VenditaTest {
   @ParameterizedTest
   @DisplayName("metodo che testa i valori ammissibili di
setPrezzo")
   @ValueSource ( doubles = { 9, 12, 14.5,0, Double.MAX_VALUE,
Double.MIN_VALUE} )
   void test1SetPrezzo(double n) {
       Vendita prodotto = new Vendita();
       Assertions.assertEquals(true, prodotto.setPrezzo(n));

   }
```

```
@Test
@DisplayName("metodo che testa i valori non ammissibili di setPrezzo
(-8, null, empty)")
void test2SetPrezzo() {
   Vendita prodotto = new Vendita();
   /*Double prezzonullo = null;*/
   /*Double prezzovuoto = new Double();*/
   Assertions.assertAll(() -> assertEquals(false,
prodotto.setPrezzo(-8))
                        /*() -> assertEquals(false,
list.get(1).setPrezzo(prezzonullo))*/
                        /*() -> assertEquals(false,
list.get(2).setPrezzo(prezzovuoto))*/);

}
```

```
@Test
@DisplayName ("metodo che testa i valori ammissibili e non
ammissibili (null, empty) di setNome")
void testSetNome(){
   Vendita prodotto = new Vendita();
   Assertions.assertAll(() -> assertEquals(true,
prodotto.setNome("a")),
                        () -> assertEquals(true,
prodotto.setNome("penna")),
                        () -> assertEquals(false,
prodotto.setNome(null)),
                        () -> assertEquals(false,
prodotto.setNome("")));
}
```

```
@Test
@DisplayName("test uguaglianza vendite")
void testUguaglianzaVendite() {
   Vendita prodotto = new Vendita("penna", 10);
   Assertions.assertAll(() -> assertEquals(false,
prodotto.uguaglianzaVendite(new Vendita())),
                        () -> assertEquals(false,
prodotto.uguaglianzaVendite(null)),
                        () -> assertEquals(true,
prodotto.uguaglianzaVendite(new Vendita("penna", 10))),
                        () -> assertEquals(false,
prodotto.uguaglianzaVendite(new Vendita("penna", 8.34))),
                        () -> assertEquals(false,
prodotto.uguaglianzaVendite(new Vendita("gomma", 10))),
```

```
                        () -> assertEquals(false,
prodotto.uguaglianzaVendite(new Vendita("matita", 12.5))));
}
```

```
@Test
@DisplayName("test minore di ")
void testMinoreDi() {
   Vendita prodotto = new Vendita("penna", 10);
   Assertions.assertAll(() -> assertEquals(false,
prodotto.minoreDi(new Vendita())),
           () -> assertEquals(false, prodotto.minoreDi(null)),
           () -> assertEquals(true, prodotto.minoreDi(new
Vendita("gomma", 13.67))),
           () -> assertEquals(false, prodotto.minoreDi(new
Vendita("penna", 8.34))),
           () -> assertEquals(false, prodotto.minoreDi(new
Vendita("gomma", 10))));

}
```

```
@Test
@DisplayName("test equals")
void testEquals() {
   Vendita prodotto = new Vendita("gomma", 12.50);
   Assertions.assertAll(() -> assertEquals(false,
prodotto.equals(new Vendita())),
           () -> assertEquals(false, prodotto.equals(null)),
           () -> assertEquals(false, prodotto.equals(new String())),
           () -> assertEquals(true, prodotto.equals(new
Vendita("gomma", 12.50))),
           () -> assertEquals(false, prodotto.equals(new
Vendita("gomma", 11.05))),
           () -> assertEquals(false, prodotto.equals(new
Vendita("matita", 12.50))),
           () -> assertEquals(false, prodotto.equals(new
Vendita("cancellino", 7.49))));

}
```

IN DETTAGLIO VENDITASCONTATATEST

```
@ParameterizedTest
@DisplayName("metodo che testa i valori ammissibili di setSconto")
@ValueSource( doubles = { 2, 11, 14.5,0,100, Double.MAX_VALUE,
Double.MIN_VALUE} )
void test1SetSconto(double n) {
```

```
    VenditaScontata prodotto = new VenditaScontata();
    Assertions.assertEquals(true, prodotto.setSconto(n));

}
```

```
@Test
@DisplayName("test equals")
void testEquals() {
    VenditaScontata prodotto = new VenditaScontata("penna", 21.50,
10);
    Assertions.assertAll(() -> assertEquals(false,
prodotto.equals(new VenditaScontata())),
            () -> assertEquals(false, prodotto.equals(null)),
            () -> assertEquals(false, prodotto.equals(new String())),
            () -> assertEquals(true, prodotto.equals(new
VenditaScontata("penna", 21.50, 10))),
            () -> assertEquals(false, prodotto.equals(new
VenditaScontata("penna", 11.05, 10))),
            () -> assertEquals(false, prodotto.equals(new
VenditaScontata("matita", 21.50, 10))),
            () -> assertEquals(false, prodotto.equals(new
VenditaScontata("cancellino", 7.49, 30))),
            () -> assertEquals(false, prodotto.equals(new
VenditaScontata("penna", 21.5, 17.40))));

}
```

```
@Test
@DisplayName("sconto negativo")
void testSetScontoNegativo() {
    VenditaScontata prodotto = new VenditaScontata();
    assertFalse(prodotto.setSconto(-10.0));
    assertEquals(0.0, prodotto.getSconto());
}
```

```
@Test
@DisplayName("caso stringa")
void testToStringSenzaSconto() {
    VenditaScontata prodotto = new VenditaScontata("Prodotto", 100.0, 0.0);
    String expected = "Componente = Prodotto, Prezzo = E100.0 Sconto =
0.0%\nCosto totale = E100.0";
    assertEquals(expected, prodotto.toString());
}
```

HOMEWORK 2-PARTE 1
SINGLEBREAK

Nella prima parte dell'HomeWork 2 bisogna testare la classe VenditaScontata.
Questa classe contiene i metodi in cui viene determinato lo sconto da applicare al prezzo.
Subito dopo aver concluso la prima parte dell'homework, ossia dopo la creazione dei test di tipo black-box, è stato usato il tool JaCoCo per controllare il code-coverage della nostra suite di test. Grazie a questo tool ho notato che il risultato era del 100% e di conseguenza non è stato necessario usare degli structural test per aumentare la nostra suite di test.

```
        public VenditaScontata(String ilNome, double ilPrezzo, double loSconto) {
                super(ilNome, ilPrezzo);
                setSconto(loSconto);
        }


        public double totale() {
                double frazione = sconto / 100;
                return (1 - frazione) * getPrezzo();
        }

        public double getSconto() {
                return sconto;
        }

        /**
         Precondizione: nuovoSconto ? non negativo.
        */
        public boolean setSconto(double nuovoSconto) {
                if(nuovoSconto >= 0) {
                        sconto = nuovoSconto;
                        return true;
                }
                else {
                        System.out.println("Errore: sconto negativo.");
                        return false;
                }
        }


        public String toString() {
                return ("Componente = " + getNome() +
                                ", Prezzo = E" + getPrezzo() +
                                " Sconto = " + sconto + "%\n" +
                                "Costo totale = E" + totale());
        }

        public boolean equals(Object altroOggetto) {
                if (altroOggetto == null)
                        return false;
                else if (!(altroOggetto instanceof VenditaScontata))
                        return false;
                else {
                        VenditaScontata altraVenditaScontata = (VenditaScontata)altroOggetto;
                        return (super.equals(altraVenditaScontata) && sconto == altraVenditaScontata.sconto);
                }
        }
} }
```

HOMEWORK 2-PARTE 2

Il codice scelto per l'Home Work 2, parte 2 è una classe contenente due metodi per calcolare gli sconti applicabili su un prodotto.

```java
    1 usage
    public static boolean isDiscountApplicable(double price, int quantity) {
        if (price <= 0 || quantity <= 0) {
            return false;
        }
        return true;
    }
    3 usages
    public static double calculateProductDiscount(double price, int quantity) {
        if (!isDiscountApplicable(price, quantity)) {
            return 0.0;
        }

        double discount = 0.0;

        if (quantity >= 10 && quantity < 20) {
            discount = 0.1;
        }else {
            discount = 0.15;
        }

        return price * quantity * (1 - discount);
    }

}
```

ANALISI DEL PROBLEMA

L'obiettivo è quello di raggiungere il code coverage al 100% usando il minore numero possibile di test. Per poter eseguire questo compito ho eseguito dei test procedendo per step.

ANALISI

1. Comprendere i requisiti (cosa deve fare il programma, gli input e output)

La classe considerata, costituita da due metodi. Analizzando i singoli metodi presi singolarmente, possiamo notare che:

```java
public static boolean isDiscountApplicable(double price, int quantity) {
```

Prende in input il prezzo del prodotto e la quantità.
È necessario che sia il prezzo che la quantità siano positivi, affinché il metodo restituisca true, in caso contrario restituirà false.
Presi questi input, il metodo quindi determina se lo sconto è applicabile o no.
Invece, il metodo:

```java
public static double calculateProductDiscount(double price, int quantity) {
```

Va a determinare il valore dello sconto e lo applica a un prodotto in base al prezzo e alla quantità.
Anche in questo caso gli input sono il prezzo e la quantità del prodotto. Il valore restituito sarà un double: lo sconto calcolato.

2. Esplorare cosa fa il programma per i vari input

In relazione ai vari input, consideriamo i due metodi singolarmente per studiarne il comportamento.
isDsicountApplicable:

- Se il prezzo o la quantità è minore o uguale a zero, allora restituisce false perché il prezzo non può essere negativo e di conseguenza lo sconto non può essere applicato a un valore negativo.

- Se sia il prezzo che la quantità sono positivi, allora restituisce true perché entrambe le condizioni sono soddisfatte.

CalculateProductDiscount:

- Se il metodo di cui sopra restituisce false, allora questo metodo restituirà 0.0
- Se invece la quantità è maggiore o uguale a 10 e minore di 20, allora lo sconto da applicare sarà uguale a 0.1 e l'output sarà proprio l'importo calcolato
- In tutti gli altri casi lo sconto da applicare è di 0.15.

3. Esplorare gli ingressi, le uscite e identificare le partizioni

INPUT:

1. PRICE
   - > 0
   - = 0
   - < 0
2. QUANTITY
   - > 0
   - = 0
   - < 0

PARTIZIONI

- prezzo positivo, quantità positiva
- Prezzo positivo, quantità nulla
- Prezzo positivo, quantità negativa
- Prezzo nullo, quantità positiva
- Prezzo negativo, quantità positiva
- Prezzo nullo, quantità nulla
- Prezzo nullo, quantità negativa
- Prezzo negativo, quantità nulla
- Prezzo negativo, quantità negativa

4. Identificare i casi limite o d'angolo

1. Prezzo positivo e quantità nulla o negativa:
   - onPoint: price > 0, quantity > 0
   - offPoint: isDiscountApplicable → false
     calculateProductDiscount→ 0.0
2. Prezzo nullo o negativo, quantità positiva
   - onPoint: price = 0 or price < 0, quantity > 0
   - offPoint: isDiscountApplicable → false
     calculateProductDiscount→ 0.0
3. Prezzo nullo o negativo, quantità nulla o negativa
   - onPoint: price = 0 or price < 0, quantity < 0 or quantity = 0
   - offPoint: isDiscountApplicable → false
     calculateProductDiscount→ 0.0

5. Creare casi di test

Osservando riga per riga il codice, è stato compreso che i valori da testare per poter raggiungere l'obiettivo del code coverage al 100% sono:

- Prezzo e quantità negativi

```
no usages
@Test
public void testNoDiscountApplicable() {
    double price = -10.0;
    int quantity = -4;
    double expectedDiscountedPrice = 0.0; // Lo sconto non è applicabile, quindi il prezzo sconta

    double actualDiscountedPrice = DiscountCalculator.calculateProductDiscount(price, quantity);

    assertEquals(expectedDiscountedPrice, actualDiscountedPrice, delta: 0.001);
}
```

- Prezzo e quantità positivi:

```
@Test
public void testDiscountIsApplicable() {
    double price = 10.0;
    int quantity = 5;
    double expectedDiscountedPrice = 42.5; // price * quantity * (1 - 0.15) = 10 * 5 * 0.95 = 47

    double actualDiscountedPrice = DiscountCalculator.calculateProductDiscount(price, quantity);

    assertEquals(expectedDiscountedPrice, actualDiscountedPrice, delta: 0.001);
}
```

- Prezzo e quantità positivi:

```
@Test
public void testDiscountIsApplicablecase2() {
    double price = 10.0;
    int quantity = 12;
    double expectedDiscountedPrice = 108.0; // price * quantity * (1 - 0.05) = 1

    double actualDiscountedPrice = DiscountCalculator.calculateProductDiscount(p

    assertEquals(expectedDiscountedPrice, actualDiscountedPrice, delta: 0.001);
}
```

Andando a eseguire i codici con l'analisi del code coverage è stato ottenuto il risultato del 100% come previsto.

```
4      public static boolean isDiscountApplicable(double price, int quantity) {
5          if (price <= 0 || quantity <= 0) {
6              return false;
7          }
8          return true;
9      }
10
11     public static double calculateProductDiscount(double price, int quantity) {
12         if (!isDiscountApplicable(price, quantity)) {
13             return 0.0;
14         }
15
16         double discount = 0.0;
17
18         if (quantity >= 10 && quantity < 20) {
19             discount = 0.1;
20         }else {
21             discount = 0.15;
22         }
23
24         return price * quantity * (1 - discount);
25     }
26
27 }
```

HOMEWORK 3

Questo Homework richiede di eseguire i property test.
Il codice scelto è composto dalla classe “UsernameValidation” contenente il metodo “validateUsername”.
Così come indica il nome stesso, questo metodo controlla che un nome utente sia composto da caratteri, in particolare:

- I primi tre devono essere caratteri alfanumerici
- Il quarto carattere deve essere un “.”
- Le 4 cifre finali devono essere delle cifre numeriche.

Questo metodo restituisce un valore booleano:

- True se l’username è valido
- False se l’username non rispetta le caratteristiche indicate precedentemente.

CODICE:

```
public class UsernameValidation {

    3 usages
    public static boolean validateUsername(String username) {
        if (username == null || username.isEmpty()) {
            throw new IllegalArgumentException("Nessun valore inserito");
        }

        boolean hasAlphanumericPrefix = true;
        boolean hasDot = false;
        boolean hasNumericSuffix = true;

        for (int i = 0; i < username.length(); i++) {
            char c = username.charAt(i);

            if (i < 3) {
                if (!Character.isLetterOrDigit(c)) {
                    hasAlphanumericPrefix = false;
                    break;
                }
            } else if (i == 3) {
                if (c != '.') {
                    return false;
                }
                hasDot = true;
            } else {
                if (!Character.isDigit(c)) {
                    hasNumericSuffix = false;
                    break;
                }
            }
        }

        return hasAlphanumericPrefix && hasDot && hasNumericSuffix;
    }
}
```

Questo codice mi permette di prevedere tutte e tre le partizioni, quindi passed, invalid e failed.

Il caso di invalid si verifica quando al metodo si passa una stringa vuota.
Mentre il caso passed e failed si verificano rispettivamente se l'input passato al codice è valido e quindi tutti i controlli effettuati hanno esito positivo, oppure al contrario almeno uno non ha esito positivo.

TEST SI USERNAMEVALIDATION

TEST PASSED:

```java
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void passTestUsername(@ForAll("validUsernames") String username) {
    boolean isValid = UsernameValidation.validateUsername(username);
    assertTrue(isValid);
    String prefix = username.substring(0, 3);

    boolean startsWithLetter = Character.isLetter(prefix.charAt(0));
    boolean startsWithNumber = Character.isDigit(prefix.charAt(0));
    boolean hasDotAtFourthChar = username.charAt(3) == '.';
    boolean onlyAlphanumeric = prefix.matches( regex: "[A-Za-z0-9]+");

    if (startsWithLetter) {
        Statistics.collect( ...values: "StartsWithLetter");
    }
    if (startsWithNumber) {
        Statistics.collect( ...values: "StartsWithNumber");
    }
    if (hasDotAtFourthChar) {
        Statistics.collect( ...values: "HasDotAtFourthChar");
    }
    if (onlyAlphanumeric) {
        Statistics.collect( ...values: "OnlyAlphanumeric");
    }
}
```

```java
no usages
@Provide
Arbitrary<String> validUsernames() {
    return Arbitraries.strings().alpha().numeric().ofLength(3)
            .flatMap(prefix -> {
                char dot = '.';
                String suffix = generateNumericSuffix();
                return Arbitraries.just( value: prefix + dot + suffix);
            });
}
```

Statistiche per Passed:

```
estamp = 2023-08-08T18:56:49.607439400, [UsernameValidationTest:passTestUsername] (3000) statistics =
    # |              label | count |
 -----|--------------------|-------|----------------------------------------------------------------------------
    0 | HasDotAtFourthChar |  1000 | ##############################################################################
    1 |   OnlyAlphanumeric |  1000 | ##############################################################################
    2 |   StartsWithLetter |   830 | ###################################################################
    3 |   StartsWithNumber |   170 | ##############
```

TEST FAILED:

```
no usages
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void failedTestUsername(@ForAll("invalidUsernames") String username) {
    boolean isValid = UsernameValidation.validateUsername(username);
    assertFalse(isValid);
}

no usages
@Provide
Arbitrary<String> invalidUsernames() {
    return Arbitraries.strings().withCharRange(' ', 'Z').ofLength(8)
            .filter(username -> username.charAt(3) != '.');
}
```

Per la parte Failed non ho previsto la generazione di statistiche poiché per username non validi e quindi per fare fallire il metodo, la stringa non deve contenere “.” Alla quarta posizione.

TEST INVALID

```
no usages
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void invalidFormatTestUsername(@ForAll("invalidFormatUsernames") String username) {
    assertThrows(IllegalArgumentException.class, () -> UsernameValidation.validateUsername(username));
}

no usages
@Provide
Arbitrary<String> invalidFormatUsernames() {
    return Arbitraries.strings().ofMaxLength(0);
}
```

Per la partizione invalid non ho previsto la generazione di statistiche come nel caso di passed perché non lo ritengo opportuno. Inoltre, un username per essere invalido non deve contenere il punto in quarta posizione oppure è una stringa vuota.