

# Relazione Test

## Desperate Colleagues 2.5 – GRUPPO 1

### **Data**

15/06/2023

### **Esame**

Integrazione e Test di Sistemi Software  
a.a. 2022/2023

REALIZZATO DA:

Detomaso Gabriele – 736743 – g.detomaso6@studenti.uniba.it

Pipoli Alessandra – 744391 – a.pipoli13@studenti.uniba.it

# Sommario

<b>Homework1 - Black Box Test</b> .....	4
assignMarkToStudent() .....	4
Comprensione dei requisiti .....	4
Identificazione delle partizioni .....	4
Identificazione dei boundary cases.....	4
Ideare casi di test .....	5
Test case template .....	5
Test su J-Unit.....	6
Modifiche al metodo post testing .....	8
addNewCourseAttender() .....	9
Comprensione dei requisiti .....	9
Identificazione delle partizioni .....	9
Identificazione dei boundary cases.....	9
Ideare casi di test .....	10
Test case template .....	10
Test su J-Unit.....	10
Modifiche al metodo post testing .....	12
getSpecifSubscription() .....	13
Comprensione dei requisiti .....	13
Identificazioni delle partizioni .....	14
Identificazione dei boundary cases.....	14
Ideare casi di test .....	14
Test case template .....	15
Test su J-Unit.....	15
<b>Homework2 – Task 1</b> .....	16
Report .....	16
Report dopo le eventuali modifiche .....	18
Conclusioni.....	19
<b>Homework2 – Task 2</b> .....	20

countMarksInInclusiveRange()	20
Comprensione dei requisiti	20
Ideare casi di test per il full code coverage	21
Test case template	21
Report	22
Test su J-Unit	22
Individuare le partizioni seguendo una metodologia basata sulle specifiche	23
Ideare casi di test utilizzando una metodologia basata sulle specifiche	24
<b>Homework 3</b>	25
getStudentsWithHigherMark()	25
Comprensione dei requisiti	25
Identificare le proprietà basate sui requisiti	26
Test case template	26
Property Based Test	26

# Homework1 - Black Box Test

## assignMarkToStudent()

```
public void assignMarkToStudent(int mark, String mat) {  
    CourseSubscription courseSubscription = getSpecificSubscription(mat);  
    courseSubscription.setMark(mark);  
}
```

### Comprensione dei requisiti

Il seguente metodo viene utilizzato per assegnare un voto ad uno specifico Studente identificato da una matricola univoca.

Lo studente deve essere già iscritto ad un corso.

Il metodo riceve due parametri in input:

- **mark**, di tipo Integer, rappresenta il voto assegnato ad uno studente.  
Range ipotizzato: da 18 a 30 e lode inclusi.
- **mat**, di tipo Stringa, rappresenta la matricola assegnata allo studente.  
Range ipotizzato: la lunghezza della matricola deve essere uguale a 6 e deve contenere solo caratteri numerici.

Il metodo non restituisce alcun valore (in quanto void).

### Identificazione delle partizioni

Parametro <b>mark</b>	Parametro <b>mat</b>
<ol style="list-style-type: none"><li>1. Interi positivi</li><li>2. Interi negativi</li><li>3. Valore 0</li></ol>	<ol style="list-style-type: none"><li>1. Stringa nulla</li><li>2. Stringa vuota</li><li>3. Stringa di lunghezza &gt; 0</li><li>4. Stringa alfanumerica</li><li>5. Stringa numerica</li></ol>

### Identificazione dei boundary cases

Parametro **mark**:

- **On point**: 18 e 31
- **Off point**: 17 e 32
- **In point**:  $18 \leq \text{mark} \leq 31$
- **Out point**:  $\text{mark} \leq 17$  OR  $\text{mark} \geq 32$

Parametro **mat**:

- **On point:** lunghezza = 6 AND numerica
- **Off point:** (lunghezza = 5 OR =7) OR alfanumerica
- **In point:** lunghezza = 6 AND numerica
- **Out point:** (lunghezza <= 5 OR >= 7) OR alfanumerica

### Ideare casi di test

**T1.** mark è un intero positivo valorizzato con 18

**T2.** mark è un intero positivo con valore minore di 18

**T3.** mark è un intero positivo valorizzato con 31

**T4.** mark è un intero positivo con valore maggiore di 31

**T5.** mark è un intero negativo

**T6.** mat è nulla

**T7.** mat ha valore vuoto

**T8.** La lunghezza di mat è minore di 6 e contiene solo caratteri numerici

**T9.** La lunghezza di mat è uguale a 6 e contiene caratteri alfanumerici

**T10.** La lunghezza di mat è maggiore di 6 e contiene solo caratteri numerici

**T11.** mat deve avere come valore la matricola di uno studente non iscritto ad alcun corso.

**T12.** mat deve avere come valore la matricola di uno studente non inserito nel sistema.

### Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

## Test su J-Unit

```
@BeforeAll
static void setup() {
    courseManager1 = new CourseManager( courseName: "Integrazione e test 1", LocalDate.now().plusDays( daysToAdd: 10));
    courseManager3 = new CourseManager( courseName: "Integrazione e test 3", LocalDate.now().minusDays( daysToSubtract: 1));
}

no usages  ▸ GabrieleDetomaso

@BeforeEach
void init() {
    // Aggiunta studenti per courseManager1
    try {
        courseManager1.addNewCourseAttender(s1, LocalDate.now());
        courseManager1.addNewCourseAttender(s2, LocalDate.now());
        courseManager1.addNewCourseAttender(s3, LocalDate.now());
    } catch (Exception e) {
        System.out.println("Eccezione lanciata 1");
    }
}

no usages  ▸ GabrieleDetomaso *

@AfterEach
void tearDown() {
    courseManager1.deleteCourseStudents();
    courseManager3.deleteCourseStudents();
}

no usages  ▸ GabrieleDetomaso

@AfterAll
static void clear() {
    courseManager1 = null;
    courseManager3 = null;
}
```

Il metodo **setup()** viene richiamato una sola volta prima dell'esecuzione di tutti i test e inizializza i vari corsi.

Il metodo **init()** viene richiamato prima dell'esecuzione di ogni test e iscrive studenti nel 'courseManager1'.

Il metodo **tearDown()** viene richiamato dopo l'esecuzione di ogni test e disiscrive gli studenti da tutti i corsi.

Il metodo **clear()** viene richiamato una sola volta terminata l'esecuzione di tutti i test e assegna valore null a tutti i corsi.

I metodi descritti qui sopra fanno parte dei metodi del ciclo di esecuzione dei test su J-Unit quindi sono richiamati per tutti i metodi testati nell'Homework1 e nell'Homework2.

Per non ripetere la descrizione su ogni metodo testato sono stati riportati solo qui.

### Test T2, T4 e T5

I tre test sono stati uniti con un `assertThrows` perché simili, in quanto tutti danno voti non accettabili dal sistema. Di conseguenza devono essere lanciate delle eccezioni.

```
@ParameterizedTest //Uniti test: T2, T4 e T5
@MethodSource("marksProviderOfmarkWrong")
@DisplayName("Inserimento di vari voti non accettabili")
void markWrong(int mark) {

    Assertions.assertThrows(Exception.class, () ->
        courseManager1.assignMarkToStudent(mark, s1.getMat())
    );
}

private static Stream<Integer> marksProviderOfmarkWrong() { return Stream.of(...values:17, 32, -1); }
```

### Test T1 e T3

I due test sono stati uniti con un `assertDoesNotThrow` perché simili, in quanto entrambi danno voti accettabili. Di conseguenza ci si aspetta l'assegnazione del voto senza lancio di eccezioni.

```
@ParameterizedTest //Uniti test: T1 e T3
@MethodSource("marksProviderOfmarkRight")
@DisplayName("Inserimento di vari voti accettabili ad uno studente iscritto")
void markRight(int mark) {

    Assertions.assertDoesNotThrow(() ->
        courseManager1.assignMarkToStudent(mark, s1.getMat())
    );
}

private static Stream<Integer> marksProviderOfmarkRight() { return Stream.of(...values:18, 31); }
```

### Test T6, T7, T8, T9, T10, T11 e T12

I sette test sono stati uniti con un `assertThrows` perché simili, in quanto vengono passati al metodo chiamato dei valori per la matricola dello studente che sono in un formato errato o non sono accettabili dal sistema (esempio passaggio di una matricola di uno studente non iscritto al corso).

```

@ParameterizedTest //Uniti test: T6, T7, T8, T9, T10, T11 e T12
@NullAndEmptySource
@MethodSource("matsProvider")
@DisplayName("Assegnazione voto a studenti con matricole di formato sbagliato o non iscritti/esistenti")
void matWrong(String mat) {

    Assertions.assertThrows(Exception.class, () ->
        courseManager1.assignMarkToStudent(mark: 28, mat));
}

```

```

private static Stream<String> matsProvider() {
    return Stream.of(...values: "11111",
        "A2G27T",
        "1234567",
        s4.getMat(),
        "9999999"
    );
}

```

## Modifiche al metodo post testing

```

public void setMark(int mark) {
    if (mark >= 18 && mark <= 30) ●
        this.mark = mark;
    else
        throw new IllegalArgumentException("Mark value must be in range [18, 30]"); ●
}

```

```

public void setMark(int mark) {
    if (mark >= 18 && mark <= 31)
        this.mark = mark;
    else
        throw new IllegalArgumentException("Mark value must be in range [18, 31]");
}

```

Dopo il fallimento del test T3, causato dal rifiuto del voto 31, è stato incluso il suddetto valore modificando il range dei voti accettati. Questo range è specificato nel metodo setMark() richiamabile su un oggetto di tipo CourseSubstruction.



## addNewCourseAttender()

```
public boolean addNewCourseAttender(Student student, LocalDate subDate) throws NullStudentException {  
    if (subDate.isAfter(endSubDate))  
        throw new DateTimeException("The subscriptions are ended");  
  
    return subscriptions.add(new CourseSubscription(student, subDate));  
}
```

### Comprensione dei requisiti

Il seguente metodo permette di aggiungere un nuovo studente al corso.

Lo studente non deve già essere iscritto e la data di iscrizione deve precedere la data di scadenza delle iscrizioni.

Il metodo riceve due parametri in input:

- student, oggetto di tipo Studente, rappresenta lo studente che si vuole aggiungere al corso
- subDate, oggetto di tipo LocalDate, rappresenta la data dell'iscrizione

Il metodo restituisce un boolean con valore:

- Vero se l'inserimento va a buon fine
- Falso altrimenti

### Identificazione delle partizioni

Parametro <b>student</b>	Parametro <b>subDate</b>
1. Valore nullo	3. Valore nullo
2. Valorizzato	4. Valorizzato

Output <b>Boolean</b>
1. Valore true
2. Valore false

### Identificazione dei boundary cases

Parametro **subDate**:

- **On point**: subdate = alla data di chiusura iscrizione
- **Off point**: subDate = della data di chiusura iscrizione + un giorno
- **In point**: subDate <= della data di chiusura iscrizione
- **Out point**: subDate > della data di chiusura iscrizione

## Ideare casi di test

**T1.** student è nullo

**T2.** student è già stato iscritto al corso

**T3.** Student non è già iscritto al corso ma solo inserito nel sistema

**T4.** subDate è nulla

**T5.** subDate è valorizzata con la data antecedente, di un giorno, rispetto alla data di chiusura iscrizione

**T6.** subDate è valorizzata con la data uguale alla data di chiusura iscrizione

**T7.** subDate è valorizzata con la data maggiore, di un giorno, rispetto alla data chiusura iscrizione.

## Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

## Test su J-Unit

### Test T1

Questo test controlla che venga lanciata un'eccezione nel caso si tenti di passare il valore null al posto di un oggetto Student.

```
@Test // T1
@DisplayName("Studente nullo")
void studentNull() {
    Assertions.assertThrows(Exception.class, () ->
        courseManager1.addNewCourseAttender(student: null, LocalDate.now())
    );
}
```

### Test T2 e T3

I due test sono stati uniti con un assertEquals perché simili, in quanto entrambi cercano di iscrivere uno studente esistente ad un corso e restituiscono un valore

booleano. Questi valori booleani restituiti vengono confrontati con dei valori booleani attesi.

```
@ParameterizedTest // Uniti tet: T2 e T3
@MethodSource("studentsProviderOfRegistrationStudent")
@DisplayName("Studente già iscritto e studente non iscritto")
void registrationStudent(Boolean expectedResult, Student s) throws NullStudentException {

    Assertions.assertEquals(expectedResult, courseManager1.addNewCourseAttender(s, LocalDate.now()));
}

private static Stream<Arguments> studentsProviderOfRegistrationStudent() {

    return Stream.of(
        Arguments.of(...arguments: false, s1),
        Arguments.of(...arguments: true, s4)
    );
}
```

#### Test T4

Questo test verifica che venga lanciata un'eccezione nel caso si passi una data nulla in input al metodo chiamato.

```
@Test //T4
@DisplayName("Data nulla")
void dateNull() {

    Assertions.assertThrows(Exception.class, () -> courseManager1.addNewCourseAttender(s4, subDate: null));
}
```

#### Test T5 e T6

I due test sono stati uniti con un assertTrue perché simili, in quanto le date delle iscrizioni sono ritenute valide. Quindi si verifica che il metodo richiamato restituisca un valore true una volta eseguito.

```
@ParameterizedTest //Uniti test: T5 e T6
@MethodSource("datesProviderOfDateRight")
@DisplayName("Data antercedente e uguale alla data di chiusura iscrizione")
void dateRight(LocalDate testDate) throws NullStudentException {

    Assertions.assertTrue(courseManager1.addNewCourseAttender(new Student(name: "a", secondName: "a", mat: "111116"), testDate));
}

private static Stream<LocalDate> datesProviderOfDateRight() {
    return Stream.of(
        LocalDate.now().plusDays(daysToAdd:9),
        LocalDate.now().plusDays(daysToAdd:10)
    );
}
```

## Test T7

Questo test verifica, tramite un `assertThrows`, che non si possa aggiungere un'iscrizione una volta superata la data di chiusura iscrizioni. Il test ha successo se viene lanciata un'eccezione.

```
@Test //T7
@DisplayName("Data scaduta")
void dateExpired() {

    Assertions.assertThrows(Exception.class, () ->
        courseManager3.addNewCourseAttender(s4, LocalDate.now())
    );
}
```

## Modifiche al metodo post testing

```
public boolean addNewCourseAttender(Student student, LocalDate subDate) throws NullStudentException {
    if (subDate.isAfter(endSubDate))
        throw new DateTimeException("The subscriptions are ended");

    if (student == null)
        throw new NullStudentException("Lo studente è nullo");

    return subscriptions.add(new CourseSubscription(student, subDate));
}
```

Per ovviare al fallimento del test T1 si è aggiunto un controllo nel metodo, tale che venga lanciata un'eccezione specifica nel caso si tenti di richiamare il metodo passando in input il valore null al posto dello studente da iscrivere.

## getSpecifSubscription()

```
public CourseSubscription getSpecificSubscription(String mat) {
    TreeSet<CourseSubscription> orderedSetByMat = new TreeSet<>(subscriptions);
    List<CourseSubscription> allMat = new ArrayList<>(orderedSetByMat);

    // Perform a binary search
    int low = 0;
    int high = allMat.size() - 1;
    int middle = (high + low) / 2;

    while (low <= high) {
        String currentMat = allMat.get(middle).getStudent().getMat();
        if (currentMat.equals(mat)) {
            return allMat.get(middle);
        } else if (currentMat.compareTo(mat) < 0) {
            low = middle + 1;
        } else if (currentMat.compareTo(mat) > 0) {
            high = middle - 1;
        }

        middle = (low + high) / 2;
    }

    return null;
}
```

## Comprensione dei requisiti

Il seguente metodo permette di cercare tramite la matricola l'iscrizione di uno specifico studente al corso, nel caso lo studente è iscritto viene restituita l'iscrizione.

Lo studente deve essere stato già inserito nel sistema.

Il metodo riceve un parametro in input:

- mat, tipo String, rappresenta la matricola univoca di uno studente da ricercare Range ipotizzato: la lunghezza della matricola deve essere uguale a 6 e deve contenere solo caratteri numerici.

Il metodo restituisce:

- Un oggetto di tipo CourseSubscription nel caso lo studente sia già iscritto al corso
- Null altrimenti

## Identificazioni delle partizioni

Parametro <b>mat</b>
<ol style="list-style-type: none"><li>1. Stringa nulla</li><li>2. Stringa vuota</li><li>3. Stringa di lunghezza <math>&gt; 0</math></li><li>4. Stringa alfanumerica</li><li>5. Stringa numerica</li></ol>

Output <b>courseSubstruction</b>
<ol style="list-style-type: none"><li>1. Valore nullo</li><li>2. Valorizzato</li></ol>

## Identificazione dei boundary cases

Parametro **mat**:

- **On point:** lunghezza = 6 AND numerica
- **Off point:** (lunghezza = 5 OR =7) OR alfanumerica
- **In point:** lunghezza = 6 AND numerica
- **Out point:** (lunghezza  $\leq 5$  OR  $\geq 7$ ) OR alfanumerica

## Ideare casi di test

**T1.** mat è nulla

**T2.** mat ha valore vuoto

**T3.** La lunghezza di mat è minore di 6 e contiene solo caratteri numerici

**T4.** La lunghezza di mat è uguale a 6 e contiene caratteri alfanumerici

**T5.** La lunghezza di mat è uguale a 6, contiene solo caratteri numerici e appartiene ad uno studente iscritto al corso

**T6.** La lunghezza di mat è maggiore di 6 e contiene solo caratteri numerici

**T7.** mat deve avere come valore la matricola di uno studente inserito nel sistema ma non iscritto ad un corso

**T8.** mat deve avere come valore la matricola di uno studente non inserito nel sistema

## Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

## Test su J-Unit

### Test T1

Questo test verifica, tramite un `assertThrows`, che venga lanciata un'eccezione qualora si tenti di passare il valore `null` al posto della matricola dello studente del quale si vuole cercare l'iscrizione.

```
@Test // T1
@DisplayName("matricola nulla")
void matNull() {
    Assertions.assertThrows(Exception.class, () ->
        courseManager1.getSpecificSubscription(mat: null)
    );
}
```

### Test T2, T3, T4, T6, T7 e T8

Questi 6 test sono stati uniti con un `assertNull` perché simili, in quanto vengono passati al metodo chiamato dei valori per la matricola dello studente che sono in un formato errato o non sono accettabili dal sistema (esempio, passaggio di una matricola di uno studente non iscritto al corso). Ovviamente il valore atteso dall'esecuzione del metodo è `null`.

```
@ParameterizedTest //Uniti test: T2, T3, T4, T6, T7 e T8
@EmptySource
@MethodSource("matsProvider")
@DisplayName("Inserimento di matricole di formato sbagliato o non iscritti/esistenti")
void subscriberSearchWrong( String mat) {

    Assertions.assertNull( courseManager1.getSpecificSubscription(mat) );
}
```

```
private static Stream<String> matsProvider() {
    return Stream.of( ...values: "11111",
        "A2G27T",
        "1234567",
        s4.getMat(),
        "999999"
    );
}
```

## Test T5

Questo test permette di verificare, tramite un `assertInstanceOf`, se dando in input un parametro corretto viene effettivamente restituita l'iscrizione al corso dello studente cercato.

```
@Test // T5
@DisplayName("Inserimento matricola di uno studente iscritto")
void registerCorrect(){

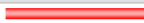
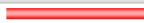













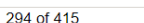

    Assertions.assertInstanceOf( CourseSubscription.class, courseManager1.getSpecificSubscription(s1.getMat()) );
}
```

## Homework2 – Task 1

### Report

Dopo aver testato i metodi con una metodologia black box, utilizzando il tool JaCoCo è stato generato il report di code coverage, il criterio utilizzato è: Condition + Branch Coverage.

### CourseManager

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getSubscriptionsByDate(LocalDate, LocalDate, boolean)		0%		0%	13	13	18	18	1	1
studentsAboveAverage(boolean)		0%		0%	8	8	16	16	1	1
countMarksInInclusiveRange(int, int)		0%		0%	10	10	14	14	1	1
getStudentsWithHigherMark()		0%		0%	5	5	16	16	1	1
getSpecificStudent(String)		0%	n/a	n/a	1	1	1	1	1	1
lambda\$getStudentsWithHigherMark\$(CourseSubscription)		0%	n/a	n/a	1	1	1	1	1	1
getSpecificSubscription(String)		100%		87%	1	5	0	16	0	1
addNewCourseAttender(Student, LocalDate)		100%		100%	0	3	0	5	0	1
CourseManager(String, LocalDate)		100%	n/a	n/a	0	1	0	5	0	1
assignMarkToStudent(int, String)		100%	n/a	n/a	0	1	0	3	0	1
deleteCourseStudents()		100%	n/a	n/a	0	1	0	2	0	1
Total	294 of 415	29%	65 of 76	14%	39	49	65	96	6	11

Da questo emerge che il metodo `getSpecificSubscription()` risulta non totalmente coperto.



```

108.     public CourseSubscription getSpecificSubscription(String mat) {
109.         TreeSet<CourseSubscription> orderedSetByMat = new TreeSet<>(subscriptions);
110.         List<CourseSubscription> allMat = new ArrayList<>(orderedSetByMat);
111.
112.         // Perform a binary search
113.         int low = 0;
114.         int high = allMat.size() - 1;
115.         int middle = (high + low) / 2;
116.
117.         while (low <= high) {
118.             String currentMat = allMat.get(middle).getStudent().getMat();
119.             if (currentMat.equals(mat)) {
120.                 return allMat.get(middle);
121.             } else if (currentMat.compareTo(mat) < 0) {
122.                 low = middle + 1;
123.             } else if (currentMat.compareTo(mat) > 0) {
124.                 high = middle - 1;
125.             }
126.
127.             middle = (low + high) / 2;
128.         }
129.
130.         return null;
131.     }

```

Precisamente la riga 123 è evidenziata in giallo, questo equivale a dire che viene valutato solo uno dei 2 rami dell'else if. Effettivamente eseguendo il codice con il debug si è confermato che durante l'esecuzione del metodo, in caso di condizione (dell'else if a riga 123) vera l'esecuzione passa dal *ramo* vero (viene eseguita la riga 124), nel caso di condizione falsa l'esecuzione si ferma prima (nello specifico al primo if o al secondo else if).

Due motivazioni:

1. **Mutua esclusione dell'if – else if.** Il motivo per cui si usa la struttura if – else if è proprio l'ottimizzazione delle espressioni logiche da calcolare. Appena una condizione risulta vera le successive vengono ignorate.
2. **Presenza del return.** Fa' sì che, durante l'esecuzione, una volta entrati nell'if di riga 120 il metodo termini restituendo il risultato, saltando così l'esecuzione delle righe successive.

Per coprire totalmente il metodo si potrebbero modificare le else if in semplici if, in modo da garantire la valutazione di tutte le condizioni al momento di esecuzione del metodo. Inoltre, si dovrebbero creare due nuove variabili, una conterrà il risultato da restituire mentre l'altra è un Boolean che funge da flag di segnalazione.

L'introduzione di queste due nuove variabili serve a posticipare il return dopo l'esecuzione di tutte le condizioni; infatti, solo quando viene trovato l'elemento ricercato si setta il flag a 'true' così che possa essere eseguito il break a riga 133 che permette di terminare il ciclo e ritornare il valore.

```

106.     public CourseSubscription getSpecificSubscription(String mat) {
107.         TreeSet<CourseSubscription> orderedSetByMat = new TreeSet<>(subscriptions);
108.         List<CourseSubscription> allMat = new ArrayList<>(orderedSetByMat);
109.
110.         // Perform a binary search
111.         int low = 0;
112.         int high = allMat.size() - 1;
113.         int middle = (high + low) / 2;
114.
115.         boolean flag = false;
116.
117.         CourseSubscription courseSubscription = null;
118.
119.         while (low <= high) {
120.             String currentMat = allMat.get(middle).getStudent().getMat();
121.             if (currentMat.equals(mat)) {
122.                 courseSubscription = allMat.get(middle);
123.                 flag = true;
124.             }
125.             if (currentMat.compareTo(mat) < 0) {
126.                 low = middle + 1;
127.             }
128.             if (currentMat.compareTo(mat) > 0) {
129.                 high = middle - 1;
130.             }
131.
132.             if (flag)
133.                 break;
134.
135.             middle = (low + high) / 2;
136.         }
137.
138.         return courseSubscription;
139.     }

```

## Report dopo le eventuali modifiche

### CourseManager

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getSubscriptionsByDate(LocalDate, LocalDate, boolean)	<div><div></div></div>	0%	<div><div></div></div>	0%	13	13	18	18	1	1
studentsAboveAverage(boolean)	<div><div></div></div>	0%	<div><div></div></div>	0%	8	8	16	16	1	1
countMarksInInclusiveRange(int, int)	<div><div></div></div>	0%	<div><div></div></div>	0%	10	10	14	14	1	1
getStudentsWithHigherMark()	<div><div></div></div>	0%	<div><div></div></div>	0%	5	5	16	16	1	1
getSpecificStudent(String)	<div><div></div></div>	0%	<div><div></div></div>	n/a	1	1	1	1	1	1
lambda\$getStudentsWithHigherMark\$0(CourseSubscription)	<div><div></div></div>	0%	<div><div></div></div>	n/a	1	1	1	1	1	1
getSpecificSubscription(String)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	6	0	21	0	1
addNewCourseAttender(Student, LocalDate)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	3	0	5	0	1
CourseManager(String, LocalDate)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	5	0	1
assignMarkToStudent(int, String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	3	0	1
deleteCourseStudents()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
Total	294 of 423	30%	64 of 78	17%	38	50	65	101	6	11

## Conclusioni

Questa ovviamente è una soluzione da adottare per raggiungere il 100% di code coverage, ma va a discapito dell'ottimizzazione del codice. Infatti, se la condizione del primo if risulta vera le altre due condizioni presenti nei successivi else if non vengono calcolate, condizioni che con questa soluzione verrebbero calcolate a prescindere dal risultato della prima, appesantendo così il carico di lavoro del metodo e aggiungendo di fatto calcoli inutili.

Per tali motivi si è deciso di fare roll back delle modifiche, così da preservare l'ottimizzazione del metodo, in quanto il codice parzialmente coperto è comunque testato nello specification-based test fatto precedentemente e non sono stati individuati faults.

## Homework2 – Task 2

### countMarksInInclusiveRange()

```
public int countMarksInInclusiveRange(int from, int to) throws CourseEmptyException {
    if (from < 18 || from > 30)
        throw new IllegalArgumentException("From parameter should be in range [18, 30]");

    if (to < 18 || to > 30)
        throw new IllegalArgumentException("To parameter should be in range [18, 30]");

    if (from > to)
        throw new IllegalArgumentException("from is greater than to");

    if (subscriptions.size() == 0)
        throw new CourseEmptyException();

    int count = 0;

    for (CourseSubscription courseSubscription : subscriptions) {
        if (courseSubscription.getMark() >= from && courseSubscription.getMark() <= to) {
            ++count;
        }
    }

    return count;
}
```

### Comprensione dei requisiti

Il seguente metodo permette di conteggiare il numero di studenti, iscritti in un determinato corso, con voto incluso in un range passato in input.

Il metodo riceve due parametri in input:

- **from**, tipo int, rappresenta il limite inferiore del range dei voti da prendere in considerazione per il conteggio degli studenti.
- **to**, tipo int, rappresenta il limite superiore del range dei voti da prendere in considerazione per il conteggio degli studenti.

Sia il parametro from, che il parametro to, devono avere come valore un numero che vada da 18 a 30 inclusi. Inoltre, from deve essere minore di to. Ovviamente è richiesto che ci debba essere almeno uno studente iscritto al corso.

La mancanza di una di queste precondizioni comporterà il lancio di una eccezione da parte del metodo durante la sua esecuzione.

Il metodo restituisce:

- Un valore int che rappresenta il numero di studenti iscritti al corso con un voto presente nel **range** di voti accettabili -> [18, 30].

### Ideare casi di test per il full code coverage

Il criterio utilizzato è: Condition + Branch Coverage.

- T1.** from è un intero positivo minore di 18
- T2.** from è un intero positivo maggiore di 30
- T3.** to è un intero positivo minore di 18
- T4.** to è un intero positivo maggiore di 30
- T5.** from e to sono valori contenuti nel range e from è maggiore di to
- T6.** from e to sono valori contenuti nel range e non ci sono iscritti nel corso
- T7.** from e to sono valori contenuti nel range e ci deve essere un solo studente iscritto al corso che abbia il voto compreso nel range
- T8.** from e to sono valori contenuti nel range e ci devono essere solo studenti che abbiano il voto minore al limite inferiore del range
- T9.** from e to sono valori accettabili nel range e ci devono essere solo studenti che abbiano il voto maggiore al limite superiore del range

### Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

## Report

Una volta implementati i test per raggiungere il 100% di code coverage è stato generato il report:

```
183.     public int countMarksInInclusiveRange(int from, int to) throws CourseEmptyException {
184.         ◆ if (from < 18 || from > 30)
185.             throw new IllegalArgumentException("From parameter should be in range [18, 30]");
186.
187.         ◆ if (to < 18 || to > 30)
188.             throw new IllegalArgumentException("To parameter should be in range [18, 30]");
189.
190.         ◆ if (from > to)
191.             throw new IllegalArgumentException("from is greater than to");
192.
193.         ◆ if (subscriptions.size() == 0)
194.             throw new CourseEmptyException();
195.
196.         int count = 0;
197.
198.         ◆ for (CourseSubscription courseSubscription : subscriptions) {
199.             ◆ if (courseSubscription.getMark() >= from && courseSubscription.getMark() <= to) {
200.                 ++count;
201.             }
202.         }
203.
204.         return count;
205.     }
```

## Test su J-Unit

Di seguito i 2 test parametrici scritti:

Il test **homeworkTask2\_1** verifica i casi in cui il metodo richiamato lancia un'eccezione.

```
@ParameterizedTest
@MethodSource("intsProviderOfHomework2Task2_1")
@DisplayName("Eccezioni del metodo countMarksInInclusiveRange")
void homework2Task2_1(int from, int to) {
    Assertions.assertThrows(Exception.class, ()
        -> courseManager3.countMarksInInclusiveRange(from, to));
}

1 usage new *
public static Stream<Arguments> intsProviderOfHomework2Task2_1() {
    return Stream.of(
        Arguments.of( ...arguments: 17, 18),
        Arguments.of( ...arguments: 32, 18),
        Arguments.of( ...arguments: 18, 17),
        Arguments.of( ...arguments: 18, 32),
        Arguments.of( ...arguments: 25, 18),
        Arguments.of( ...arguments: 18, 25)
    );
}
```

Il test **homeworkTask2\_2** verifica i casi in cui il metodo termina l'esecuzione restituendo un valore.

```
@ParameterizedTest
@MethodSource("intsProviderOfHomework2Task2_2")
@DisplayName("Confronto valore ritornato dal metodo countMarksInInclusiveRange")
void homework2Task2_2(int from, int to, int countExpeted){
    try {
        courseManager3.addNewCourseAttender(s1, LocalDate.now().minusDays( daysToSubtract: 2));
    } catch (Exception e) {
        e.getMessage();
    }
    courseManager3.assignMarkToStudent( mark: 20, s1.getMat());

    try{
        Assertions.assertEquals(countExpeted, courseManager3.countMarksInInclusiveRange(from, to));
    }catch (CourseEmptyException cee){
        cee.getMessage();
    }
}

1 usage new *
public static Stream<Arguments> intsProviderOfHomework2Task2_2() {
    return Stream.of(
        Arguments.of( ...arguments: 18, 25, 1),
        Arguments.of( ...arguments: 18, 19, 0),
        Arguments.of( ...arguments: 21, 25, 0)
    );
}
```

Nonostante la copertura totale del codice del metodo, risulta evidente la presenza di un bug.

Per la logica con cui è stata progettata la classe il range di voti accettabili dovrebbe essere [18, 31] essendo possibile assegnare 31 come voto ad uno studente iscritto. Se si inserisce 31 come valore di from o to il metodo lancia un'eccezione.

Individuare le partizioni seguendo una metodologia basata sulle specifiche

Parametro <b>from</b>	Parametro <b>to</b>
1. Valore negativo 2. Valore positivo	3. Valore negativo 4. Valore positivo

Output <b>count</b>
1. Valore 0 2. Valore > 0

Ideare casi di test utilizzando una metodologia basata sulle specifiche

Per il ragionamento fatto in precedenza è importante includere altri test che potrebbero individuare possibili comportamenti logicamente sbagliati.

**T1** From ha valore 31

**T2** To ha valore 31

**T3** From e To devono avere uno stesso valore e devono esserci almeno uno studente iscritto con lo stesso voto



## Homework 3

### getStudentsWithHigherMark()

```
public Set<Student> getStudentsWithHigherMark() {
    LinkedHashSet<Student> higherMarkStudents = new LinkedHashSet<>();

    TreeSet<CourseSubscription> orderedSet = new TreeSet<>(Comparator
        .comparingInt(CourseSubscription::getMark)
        .thenComparing(o -> o.getStudent().getMat()));
    orderedSet.addAll(subscriptions);

    orderedSet = (TreeSet<CourseSubscription>) orderedSet.descendingSet();

    int higherMark = -1; // inserimento di un valore a caso < 0

    if (orderedSet.size() > 0) {
        higherMark = orderedSet.first().getMark();

        if (higherMark == -1) {
            return higherMarkStudents;
        }
    }

    for (CourseSubscription courseSubscription : orderedSet) {
        if (courseSubscription.getMark() == higherMark) {
            higherMarkStudents.add(courseSubscription.getStudent());
        } else {
            break;
        }
    }

    return higherMarkStudents;
}
```

### Comprensione dei requisiti

Il seguente metodo permette di conteggiare il numero di studenti con il voto più alto, iscritti in un determinato corso.

Il metodo non riceve nulla in input, ma utilizza l'insieme delle iscrizioni contenute nello specifico corso per ricercare gli studenti con voto più alto.

Il metodo restituisce:

- Un Set non vuoto di studenti, se nel corso è presente almeno un'iscrizione e almeno uno studente ha un voto assegnato.
- Un Set vuoto di studenti, se nel corso non ci sono iscrizioni o se nessuno studente iscritto ha un voto assegnato.

## Identificare le proprietà basate sui requisiti

**Restituzione insieme pieno** di studenti con voto maggiore - Il corso ha almeno uno studente iscritto e almeno uno studente ha il voto assegnato.

**Restituzione insieme** di studenti **vuoto** - Il corso non ha studenti iscritti o tutti gli iscritti non hanno un voto assegnato.

## Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

## Property Based Test

### Restituzione insieme pieno

#### METODO PROVIDE

```
@Provide
public ListArbitrary<Student> studentsProvider (){
    Arbitrary <String> names = Arbitraries.strings().alpha().ofLength(10);
    Arbitrary <String> surnames = Arbitraries.strings().alpha().ofLength(15);
    Arbitrary <String> mats = Arbitraries.strings().numeric().ofLength(6);

    Arbitrary<Student> students = Combinators.combine(names, surnames, mats).as(Student:: new);

    return students.list().uniqueElements(Student::getMat);
}
```

Il metodo 'studentsProvider' restituisce al metodo property chiamante un oggetto di tipo ListArbitrary (contenente oggetti Arbitrary) che servirà per generare la lista di studenti casuali.

Sono stati inserite varie costrizioni come la lunghezza del nome, cognome e matricola dello studente, i vari valori con cui generare le stringhe e il vincolo che la matricola sia diversa per ogni studente.

## METODO PROPERTY

```
@Property(tries = 50)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Label("Restituzione insieme pieno")
void studentsWithMarkAndWithout(@ForAll("studentsProvider") @Size(min = 20, max = 35) List<Student> students,
                                @ForAll @Size(value = 35) List<@DateRange(min = "2023-05-01", max = "2023-05-31") LocalDate> localDates,
                                @ForAll @Size(value = 35) List<@IntRange(min = 18, max = 31) Integer> marks,
                                @ForAll @Size(value = 5) @UniqueElements List<@IntRange(max = 34) Integer> pos
                                ) throws NullStudentException{

    courseManager4 = new CourseManager( courseName: "Corso1", LocalDate.parse( text: "2023-10-31"));
    Student s = null;
    int mark = 0;
    int higherMark = 0;
    Set<Student> highMarkStudents = new LinkedHashSet<>(); // Insieme che conterrà gli studenti con voto più alto

    // calcolo del voto più alto generato
    for (int i = 0; i < students.size(); i++)
        if (!pos.contains(i))
            if (marks.get(i) > higherMark)
                higherMark = marks.get(i);
}
```

```
// Iscrizione degli studenti nel corso
for (int i = 0; i < students.size(); i++)
{
    s = students.get(i);
    mark = marks.get(i);

    courseManager4.addNewCourseAttender(s, localDates.get(i)); //Iscrizione studente

    //Assegnazione del voto se l'indice non è contenuto nell'insieme delle posizioni da lasciare senza voto
    if (!pos.contains(i))
    {
        if (mark == higherMark)
            highMarkStudents.add(s); // Studente con voto più alto inserito nell'insieme

        courseManager4.assignMarkToStudent(marks.get(i), s.getMat()); // voto assegnato
        //Statistics.label("Mark range assigned").collect(mark);
    }
}

Statistics.label("Higher mark range assigned").collect(mark); // Statistica del voto più alto assegnato ad ogni prova

// statistica del range di voti generato
for (int m: marks)
    Statistics.label("Mark range generated").collect(m);

Assertions.assertEquals(highMarkStudents, courseManager4.getStudentsWithHigherMark());
}
```

il metodo property 'studentWithMarkAndWithout' genera 4 parametri:

- Una lista di studenti di dimensione variabile [20, 35]
- Una lista di 35 date con intervallo di date generabili [2023-05-01, 2023-05-31]
- Una lista di 35 voti con intervallo di voti generabili [18, 31]
- Una lista di 5 interi con intervallo di interi generabili [0, 35], segnala 5 posizioni in cui lo studente non avrà voto.

Una volta istanziato il nuovo corso chiamato 'courseManager4' si procede a salvare in 'higherMark' il voto più alto generato facendo attenzione ad ignorare i voti generati per le posizioni degli studenti senza voto.

Successivamente viene riempito il corso con gli studenti (e il relativo voto) generati, tenendo sempre conto di lasciare senza voto gli studenti con posizione contenuta nella lista 'pos'. Contemporaneamente gli studenti con voto più alto vengono inseriti anche in un Set utile poi per confrontare il risultato restituito dal metodo testato.

Il test ha successo se il Set di studenti restituito è uguale al Set 'higherMarkStudents'.

## STATISTICHE

### Higher mark range assigned

```
timestamp = 2023-05-29T21:08:21.918348100, [CourseManagerGroup1Test:studentsWithMarkAndWithout] (50) Higher mark range assigned =
# | label | count |
----|-----|-----|-----
0 | 18 | 1 | #
1 | 19 | 5 | #####
2 | 20 | 3 | ###
3 | 21 | 4 | ####
4 | 22 | 5 | #####
5 | 23 | 4 | ####
6 | 24 | 6 | #####
7 | 25 | 4 | ####
8 | 26 | 2 | ##
9 | 27 | 4 | ####
10 | 28 | 4 | ####
11 | 29 | 1 | #
12 | 30 | 1 | #
13 | 31 | 6 | #####
```

Questa statistica riporta quante volte ogni voto è stato il più alto del corso su 50 prove.

### Mark range generated

```
timestamp = 2023-05-29T18:10:45.117271300, [CourseManagerGroup1Test:studentsWithMarkAndWithout] (1750) Mark range generated =
# | label | count |
----|-----|-----|-----
0 | 18 | 161 | #####
1 | 19 | 150 | #####
2 | 20 | 101 | #####
3 | 21 | 111 | #####
4 | 22 | 110 | #####
5 | 23 | 118 | #####
6 | 24 | 121 | #####
7 | 25 | 122 | #####
8 | 26 | 111 | #####
9 | 27 | 120 | #####
10 | 28 | 110 | #####
11 | 29 | 105 | #####
12 | 30 | 143 | #####
13 | 31 | 167 | #####
```

Questa seconda statistica riporta tutti i voti generati in 50 prove, indipendentemente se siano stati assegnati o meno. Come si può notare i valori più generati sono gli estremi dell'intervallo, dove è più probabile che ci siano dei bug.

Questa osservazione non è inaspettata in quanto il framework Jqwik si concentra maggiormente sul generare i valori per testare i boundary case.

## Restituzione insieme vuoto

### METODO PROVIDE

Il provide 'studentsProvider' per generare la lista degli studenti è lo stesso descritto in precedenza.

### METODO PROPERTY

```
@Property (tries = 50)
@Report(Reporting.GENERATED)
@Label("Restituzione insieme vuoto")
void allStudentsWithoutMark (@ForAll("studentsProvider") @Size(min = 0, max = 35) List<Student> students,
                             @ForAll @Size(value = 35) List<@DateRange(min = "2023-05-01", max = "2023-05-31") LocalDate> localDates
                             ) throws NullStudentException{

    courseManager4 = new CourseManager( courseName: "Corso1", LocalDate.parse( text: "2023-10-31"));

    //Riempimento del corso con i valori da generare
    for (int i = 0; i < students.size(); i++){
        courseManager4.addNewCourseAttender(students.get(i), localDates.get(i));
    }

    Assertions.assertTrue(courseManager4.getStudentsWithHigherMark().isEmpty());
}
```

Il metodo property 'allStudentsWithoutMark' genera 2 parametri:

- Una lista di studenti di dimensione variabile [0, 35]
- Una lista di 35 date con intervallo di date generabili [2023-05-01, 2023-05-31]

Il corso 'courseManager4' una volta istanziato viene riempito con gli studenti presenti nella lista 'students', ma a nessuno viene assegnato il voto. Nel caso la lista abbia 0 elementi il corso rimane vuoto.

Il test ha successo se il metodo restituisce un Set vuoto di studenti.