Relazione Test

Desperate Colleagues
2.5 – GRUPPO 2

Data
15/06/2023

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

REALIZZATO DA:
Detomaso Giacomo – 735634 – g.detomaso7@studenti.uniba.it
Scorrano Roberto – 735877 – r.scorrano3@studenti.uniba.it

Sommario

Homework 1 - Black Box Test

getSubscriptionsByDate()

```java
public Set<CourseSubscription> getSubscriptionsByDate(LocalDate fromDate, LocalDate toDate, boolean inclusive) {
    Set<CourseSubscription> subsInRange = new LinkedHashSet<>();
    long fromDateLong = fromDate.getLong(ChronoField.EPOCH_DAY);
    long toDateLong = toDate.getLong(ChronoField.EPOCH_DAY);

    for (CourseSubscription courseSubscription : subscriptions) {
        LocalDate subDate = courseSubscription.getSubDate();
        long subDateLong = subDate.getLong(ChronoField.EPOCH_DAY);

        // True if the subDate belongs to the range (bounds excluded)
        boolean condition = subDateLong > fromDateLong && subDateLong < toDateLong;

        // True if inclusive param is true and the subDate is equals to
        // the lower or the upper bound of the interval
        boolean inclusiveCondition = inclusive && (subDateLong == fromDateLong || subDateLong == toDateLong);

        // If the inclusive parameter is set to true but the subDate is in the range (not equals
        // to one of the two bounds) inclusiveCondition will be evaluated as false,
        // but condition is evaluated to true. The behavior is identical to have inclusive set to false.
        if (condition || inclusiveCondition) {
            subsInRange.add(courseSubscription);
        }
    }

    return subsInRange;
}
```

Comprensione dei requisiti

Il seguente metodo restituisce un insieme di iscrizioni al corso, la cui data di iscrizione è compresa in un range di date. Di default i limiti (superiore e inferiore) del range sono esclusi dal risultato finale, a meno che la loro inclusione non sia esplicitamente richiesta.

Ad esempio, per un range di date del tipo (10-02-2023 a 20-02-2023), verrà restituito in output un set di iscrizioni con le cui date escluderanno i due limiti del range (ovvero 10 e 20 febbraio). Se specificato, invece, anche le iscrizioni avvenute in queste date saranno presenti.

Il metodo può ricevere tre parametri:

- fromDate, oggetto di tipo LocalDate che stabilisce il limite inferiore del range.
- toDate, oggetto di tipo LocalDate che stabilisce il limite superiore del range.
- inclusive, di tipo boolean che viene utilizzato per stabilire l'inclusione o no dei limiti superiore e inferiore del range.

Il metodo restituisce un Set nel quale sono presenti le iscrizioni al corso che ricadono nel range indicato.

Identificazione delle partizioni

Input individuali

fromDate	toDate	inclusive
1. Valore nullo 2. Date nel formato: *aaaa-mm-gg*	1. Valore nullo 2. Date nel formato: *aaaa-mm-gg*	1. True 2. False

Nei primi quattro casi di combinazione non è contato il campo inclusive in quanto il suo valore è irrilevante in quelle combinazioni.

Combinazioni di input

- fromDate = toDate con inclusive False
- fromDate = toDate con inclusive True
- fromDate > toDate

Classi di output

- Insieme vuoto
- Insieme con numero elementi >= 1
- Valore null

Identificazione dei boundary cases

I valori dei boundary cases variano a seconda del valore inclusive (true o false), quindi analizzeremo le variabili nelle loro possibili condizioni. Si ipotizza inoltre che le variabili vengano valutate insieme, nella medesima espressione logica:

Se inclusive è **false**:

- **On point:** fromDate e toDate [*rende la condizione falsa*]
- **Off point:** fromDate + 1 e toDate – 1 [*rende la condizione vera*]
- **In point:** tutte le date comprese fra fromDate + 1 e toDate - 1
- **Out point:** date minori di fromDate + 1 e maggiori di toDate - 1

Se inclusive è **true**:

- **On point:** fromDate e toDate [*rende la condizione vera*]
- **Off point:** fromDate - 1 e toDate + 1 [rende la condizione falsa]
- **In point:** tutte le date comprese fra fromDate e toDate inclusi
- **Out point:** date minori di fromDate e maggiori di toDate

Ideare casi di test

T1: fromDate valore nullo

T2: toDate valore nullo

T3: fromDate formato errato

T4: toDate formato errato

T5: iscrizione trovata in range valido con inclusive false

T6: iscrizione non trovata in range valido con inclusive false

T7: iscrizione trovata in range valido con inclusive true

T8: iscrizione non trovata in range valido con inclusive true

T9: fromDate maggiore di toDate

T10: fromDate uguale a toDate con inclusive false

T11: fromDate uguale a toDate con inclusive true

T12: singolo studente con iscrizione esattamente sul range inferiore con inclusive false

T13: singolo studente con iscrizione esattamente sul range superiore con inclusive false

T14: singolo studente con iscrizione esattamente sul range inferiore con inclusive true

T15: singolo studente con iscrizione esattamente sul range superiore con inclusive true

T16: nessuno studente iscritto al corso

T17: più di uno studente con iscrizione esattamente sul range superiore con inclusive true

T18: più di uno studente con iscrizione esattamente sul range inferiore con inclusive true

Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

Test su J-Unit

I metodi relativi al ciclo di vita di J-Unit usati per inizializzare gli oggetti di test, sono presenti al termine della relazione.

Test T1 e T2

I due test, simili, sono stati uniti eseguendo una assertAll, nel quale si controlla il comportamento del metodo con una data di input **null**.

```
Giacomo Detomaso
@Test // Uniti test: T1 e T2
@DisplayName("fromDate and toDate null")
void fromDateToDateNull() {
    Assertions.assertAll(
            () -> Assertions.assertThrows(Exception.class, () ->
                    courseManager1.getSubscriptionsByDate( fromDate: null, LocalDate.now(),  inclusive: false)),
            () ->  Assertions.assertThrows(Exception.class, () ->
                    courseManager1.getSubscriptionsByDate(LocalDate.now(),  toDate: null,  inclusive: false))
    );
}
```

Test T3 e T4

I due test, simili, sono stati uniti eseguendo una assertAll, nel quale si controlla il comportamento del metodo con una data di input **errata**.

```
@Test // Uniti test: T3 e T4
@DisplayName("Wrong date format")
void wrongDateFormat() {
    Assertions.assertAll(
            () -> Assertions.assertThrows(DateTimeException.class, () ->
                    courseManager1.getSubscriptionsByDate(LocalDate.parse( text: "04-08-2023"), LocalDate.now(),  inclusive: false)),

            () -> Assertions.assertThrows(DateTimeException.class, () ->
                    courseManager1.getSubscriptionsByDate(LocalDate.now(), LocalDate.parse( text: "04-08-2023"),  inclusive: false))
    );
}
```

Test T5 e T7

Si valuta il comportamento in un caso corretto del metodo, con il parametro booleano che cambia da true a false. Contestualmente anche la dimensione dell'output risulta diversa nei due casi. È stato sviluppato un test parametrico.

```
@ParameterizedTest // Uniti test: T5 e  T7
@DisplayName("Subscriptions found")
@ValueSource(booleans = {false, true})
void subscriptionsFound(boolean inclusive) throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-03");
    LocalDate toDate = LocalDate.parse( text: "2022-11-10");

    int equal; // parametro da usare nell'assert equals

    // In base al valore di inclusive cambia il numero di studenti resituiti
    if (inclusive) {
        equal = 2; // Se inclusive è vero vi sono
    } else {
        equal = 1;
    }

    Assertions.assertEquals(equal, courseManager1.getSubscriptionsByDate(fromDate, toDate, inclusive).size());
}
```

Test T6 e T8

Sviluppato un test parametrico nel quale è inserito un range in cui non sono presenti iscrizioni. Si valuta il risultato al variare del parametro inclusive.

```
@ParameterizedTest // Uniti test: T6 e  T8
@DisplayName("Subscriptions not found")
@ValueSource (booleans = {false, true})
void subscriptionsNotFound(boolean inclusive) throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-12-03");
    LocalDate toDate = LocalDate.parse( text: "2022-12-10");

    Assertions.assertEquals( expected: 0, courseManager1.getSubscriptionsByDate(fromDate, toDate, inclusive).size());
}
```

Test T9, T10 e T11

Questi tre test sono stati sviluppati singolarmente. Le operazioni effettuate rispettano quanto descritto nel template. I tre test effettuano operazioni di combinazione degli input.

```java
@Test // T9
@DisplayName("fromDate greater then toDate")
void fromDateGreaterThanToDate() {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-10");
    LocalDate toDate = LocalDate.parse( text: "2022-11-03");
    Assertions.assertThrows(RangeDateException.class, () -> courseManager1
            .getSubscriptionsByDate(fromDate, toDate,  inclusive: true));
}

 Giacomo Detomaso *
@Test // T10
@DisplayName("fromDate and toDate are the same with inclusive false")
void fromDateEqualsToDateNotInclusive() {
    LocalDate date = LocalDate.parse( text: "2022-11-10");
    Assertions.assertThrows(RangeDateException.class, () -> courseManager1
            .getSubscriptionsByDate(date, date,  inclusive: false));
}

 Giacomo Detomaso
@Test // T11
@DisplayName("fromDate toDate are the same with inclusive true")
void fromDateEqualsToDateInclusive() throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-10");
    LocalDate toDate = LocalDate.parse( text: "2022-11-10");

    Assertions.assertEquals( expected: 1, courseManager1.getSubscriptionsByDate(fromDate, toDate,  inclusive: true).size());
}
```

Test T16

Il test verifica il comportamento del metodo se nessuno studente è iscritto al corso.

```java
 Giacomo Detomaso
@Test // T16
@DisplayName("No students in course")
void noStudentsInCourse (){
    LocalDate fromDate = LocalDate.parse( text: "2022-11-01");
    LocalDate toDate = LocalDate.parse( text: "2022-11-10");

    courseManager1.deleteCourseStudents(); // azzera il corso

    Assertions.assertThrows(CourseEmptyException.class,
            () -> courseManager1.getSubscriptionsByDate(fromDate, toDate,  inclusive: true));
}
```

Test T12 e T14, T13 e T15

Verificano la presenza o meno, al variare di inclusive, di uno studente iscritto in una data corrispondente al limite inferiore (T12 T14) e superiore (T13 e T15).

```
Giacomo Detomaso
@ParameterizedTest // Uniti test: T12 e  T14
@DisplayName("Students on fromDate: inclusive true and false")
@ValueSource (booleans = {false, true})
void studentsOnFromDate(boolean inclusive) throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-04");
    LocalDate toDate = LocalDate.parse( text: "2022-11-05");

    Set<CourseSubscription> subs = courseManager1.getSubscriptionsByDate(fromDate, toDate, inclusive);
    Set<Student> students = new HashSet<>();

    // Recupera tutti gli studenti presenti nelle iscrizioni
    for (CourseSubscription sub: subs) {
        students.add(sub.getStudent());
    }

    // Controlla che lo studente s1 che risiede sul limite inferiore di questo range
    // se inclusive è true sia presente, altrimenti non presente
    if (inclusive) {
        Assertions.assertTrue(students.contains(s1));
    } else {
        Assertions.assertFalse(students.contains(s1));
    }
}
```

```
Giacomo Detomaso
@ParameterizedTest // Uniti test: T13 e  T15
@DisplayName("Students on toDate: inclusive true and false")
@ValueSource (booleans = {false, true})
void studentsOnToDate(boolean inclusive) throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-01");
    LocalDate toDate = LocalDate.parse( text: "2022-11-04");

    Set<CourseSubscription> subs = courseManager1.getSubscriptionsByDate(fromDate, toDate, inclusive);
    Set<Student> students = new HashSet<>();

    // Recupera tutti gli studenti presenti nelle iscrizioni
    for (CourseSubscription sub: subs) {
        students.add(sub.getStudent());
    }

    // Controlla che lo studente s1 che risiede sul limite inferiore di questo range
    // se inclusive è true sia presente, altrimenti non presente
    if (inclusive) {
        Assertions.assertTrue(students.contains(s1));
    } else {
        Assertions.assertFalse(students.contains(s1));
    }
}
```

Test T17 e T18

Test parametrici che verificano la presenza o meno di più studenti iscritto in una data corrispondente al limite inferiore (T18) e superiore(T17)

```java
@Test // T17
@DisplayName("Students on toDate inclusive multiple")
void studentsOnToDateInclusiveMultiple() throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-01");
    LocalDate toDate = LocalDate.parse( text: "2022-11-11");

    Set<CourseSubscription> subs = courseManager1.getSubscriptionsByDate(fromDate, toDate, inclusive: true);
    Set<Student> students = new HashSet<>();

    // Recupera tutti gli studenti presenti nelle iscrizioni
    for (CourseSubscription sub: subs) {
        students.add(sub.getStudent());
    }

    // Controlla che lo studente s3 ed s4 (che risiedono sul range superiore) siano present
    Assertions.assertAll(
            () -> Assertions.assertTrue(students.contains(s3)),
            () -> Assertions.assertTrue(students.contains(s4)),
            // Controllo comunque la dimensione dell'output (controllo ridondante -> controllato in test
            // precedenti. Lo si lascia per chiarezza e sicurezza). Il controllo è stato aggiunto perchè
            // nel range testato vi sono tutti e 4 gli studenti del corso
            () -> Assertions.assertEquals( expected: 4, students.size()));
}
```

```java
@Test // T18
@DisplayName("Students on fromDate inclusive multiple")
void studentsOnFromDateInclusiveMultiple() throws RangeDateException, CourseEmptyException {
    LocalDate fromDate = LocalDate.parse( text: "2022-11-11");
    LocalDate toDate = LocalDate.parse( text: "2022-11-20");

    Set<CourseSubscription> subs = courseManager1.getSubscriptionsByDate(fromDate, toDate, inclusive: true);
    Set<Student> students = new HashSet<>();

    // Recupera tutti gli studenti presenti nelle iscrizioni
    for (CourseSubscription sub: subs) {
        students.add(sub.getStudent());
    }

    // Controlla che lo studente s3 ed s4 (che risiedono sul range superiore) siano present
    Assertions.assertAll(
            () -> Assertions.assertTrue(students.contains(s3)),
            () -> Assertions.assertTrue(students.contains(s4)));
}
```

Modifiche al metodo post testing

```java
public Set<CourseSubscription> getSubscriptionsByDate(LocalDate fromDate, LocalDate toDate, boolean inclusive)
        throws RangeDateException, CourseEmptyException {
    Set<CourseSubscription> subsInRange = new LinkedHashSet<>();
    long fromDateLong = fromDate.getLong(ChronoField.EPOCH_DAY);
    long toDateLong = toDate.getLong(ChronoField.EPOCH_DAY); 1

    // Check the dates
    if (fromDate.isAfter(toDate)) { 2
        throw new RangeDateException("fromDate is greater then to date");
    }

    if (!inclusive && (fromDate.equals(toDate))) { 3
        throw new RangeDateException("fromDate is equal to toDate but inclusive is false. They cannot be equals");
    }

    // Check the course number
    if (subscriptions.isEmpty()) { 4
        throw new CourseEmptyException("The course is empty");
    }
```

1. Qualsiasi test eseguito su questo metodo falliva a seguito di eccezioni lanciate dal metodo stesso. Si è quindi analizzato il codice, e constato che vi era un errore sintattico. Al posto di richiamare il metodo getLong su toDate veniva richiamato il metodo get, che causava errore a runtime;
2. A seguito del fallimento del test **T9** è stato aggiunto un controllo per prevenire che fromDate sia maggiore di toDate, lanciando un'eccezione
3. A seguito del fallimento del test **T10** è stato aggiunto un controllo per prevenire che toDate sia uguale a fromDate con inclusive false, lanciando un'eccezione
4. A seguito del fallimento del test **T16** è stato aggiunto un controllo per prevenire che vengano effettuate ricerche se il corso non ha iscritti, lanciando un'eccezione

countMarksInInclusiveRange()

```java
public int countMarksInInclusiveRange(int from, int to) {
    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]");

    int count = 0;

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

    return count;
}
```

Comprensione dei requisiti

Il metodo in questione serve per individuare quanti studenti hanno preso un voto che ricade all'interno di un range, i cui limiti superiore e inferiore sono inclusi.

Il range di voti deve essere compreso tra 18 e 30 (inclusi).

I parametri di input sono:

- From: intero, indica il limite inferiore del range nel quale verranno contati i
- To: intero, indica il limite superiore del range nel quale verranno contati i voti

Il metodo restituisce in output il numero di voti che ricade nel range indicato in input.

Identificazione delle partizioni

Input individuali

from	to
- Interi positivi - Interi negativi - Valore nullo (0)	- Interi positivi - Interi negativi - Valore nullo (0)

Combinazione input

- from = to
- from > to

Output

- Valore nullo
- Valore >= 1
- Valore negativo (si ipotizza condizione di errore)

Identificazione dei boundary cases

Parametro **from**:

- **on point**: 18
- **off point**: 17
- **in point**: n >= 18
- **out point**: n < 18

Parametro **to**:

- **on point**: 30
- **off point**: 31
- **in point**: n <= 30
- **out point**: n > 30

Combinando i due parametri che si ipotizza vengono valutati in unica espressione:

- **on point**: 18 e 30 [rendono la condizione vera]
- **off point**: 17 e 31 [rendono la condizione falsa]
- **in point**: n >= 18 AND n <= 30
- **out point**: n < 18 AND n > 30

Ideare casi di test

T1: valore from corretto e to minore di 18

T2: valore from corretto e to maggiore di 30

T3: valore to corretto e from minore di 18

T4: valore to corretto e from maggiore di 30

T5: singolo studente che rientra nel range valido di valori

T6: nessuno studente che rientri nel range valido di valori

T7: voto dello studente esattamente uguale al valore di from (range valido)

T8: voto dello studente esattamente uguale al valore di to (range valido)

T9: from uguale a to con uno studente che abbia esattamente quel voto

T10: from maggiore di to

T11: nessuno studente iscritto al corso

T12: nessun voto assegnato agli studenti

Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

Test su J-Unit

Test T1, T2, T3 e T4

Test parametrico (i valori sono presi dal metodo **getInputMarkPairs** presente nella seconda immagine) in cui vengono testati i vari input errati di from e to.

```java
@ParameterizedTest //Uniti test: T1, T2, T3 e T4
@DisplayName("Wrong mark input")
@MethodSource("getInputMarkPairs")
void wrongMarkInput(int from, int to) {
    Assertions.assertThrows(IllegalArgumentException.class, () -> courseManager1.countMarksInInclusiveRange(from, to));
}
```

```java
public static Stream<Arguments> getInputMarkPairs() {
    return Stream.of(
            Arguments.of( ...arguments: 18, 17), // T1 di countMarksInInclusiveRange
            Arguments.of( ...arguments: 18, 31), // T2 di countMarksInInclusiveRange
            Arguments.of( ...arguments: 17, 18), // T3 di countMarksInInclusiveRange
            Arguments.of( ...arguments: 31, 17)// T4 di countMarksInInclusiveRange
    );
}
```

Test T5 e T6

Test nella quale viene applicato un range valido di valori from e to, inserendo prima tre studenti che rientrino nel range e poi nessuno studente che rientri.

```java
@Test // Uniti T5 e T6
@DisplayName("Students in range and not")
void studentsInRangeAndNot () {
    courseManager1.assignMarkToStudent( mark: 29,  mat: "111111");
    courseManager1.assignMarkToStudent( mark: 28,  mat: "111112");
    courseManager1.assignMarkToStudent( mark: 29,  mat: "111113");

    // Testiamo che nel primo caso il range contenga gli studenti, nel secondo caso no
    Assertions.assertAll(
            () -> Assertions.assertTrue( condition: courseManager1.countMarksInInclusiveRange(27, 30) > 0),
            () -> Assertions.assertEquals( expected: 0, courseManager1.countMarksInInclusiveRange(25, 27))
    );
}
```

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

Test nella quale vengono provate le seguenti condizioni:

- uno studente con voto esattamente su from (T7) e poi su to (T8)
- from uguale a to con uno studente che abbia esattamente quel voto (T9)
- from maggiore di to, viene testato il ritorno di un'eccezione per questi input errati (T10)
- nessuno studente iscritto al corso (T11)
- nessuno degli studenti iscritti al corso possiede voti (T12)

```
    @Test // T7
    @DisplayName("One student on from")
    void oneStudentOnFrom() throws CourseEmptyException {
        courseManager1.assignMarkToStudent( mark: 27,  mat: "111111");
        Assertions.assertEquals( expected: 1, courseManager1.countMarksInInclusiveRange(27, 30));
    }

    @Test // T8
    @DisplayName("One student on to")
    void oneStudentOnTo() throws CourseEmptyException {
        courseManager1.assignMarkToStudent( mark: 30,  mat: "111111");
        Assertions.assertEquals( expected: 1, courseManager1.countMarksInInclusiveRange(27, 30));
    }

    @Test // T9
    @DisplayName("From equal to")
    void fromEqualTo() throws CourseEmptyException {
        courseManager1.assignMarkToStudent( mark: 27,  mat: "111111");
        Assertions.assertTrue( condition: courseManager1.countMarksInInclusiveRange(27, 27) >= 1);
    }

    @Test // T10
    @DisplayName("From greater than to")
    void fromGreaterThanTo() {
        courseManager1.assignMarkToStudent( mark: 27,  mat: "111111");
        Assertions.assertThrows(IllegalArgumentException.class, () -> courseManager1.countMarksInInclusiveRange(27, 22));
    }

    @Test // T11
    @DisplayName("No student in course")
    void noStudentInCourse3() {
        courseManager1.deleteCourseStudents();
        Assertions.assertThrows(CourseEmptyException.class, () -> courseManager1.countMarksInInclusiveRange(19, 29));
    }

    @Test // T12
    @DisplayName("No marks in course")
    void noMarksInCourse() throws CourseEmptyException {
        Assertions.assertEquals( expected: 0, courseManager1.countMarksInInclusiveRange(20, 25));
    }
}
```

Modifiche al metodo post testing

```java
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) 1
        throw new IllegalArgumentException("from is greater than to");

    if (subscriptions.size() == 0) 2
        throw new CourseEmptyException();

    int count = 0;

    for (CourseSubscription courseSubscription : subscriptions) {
        if (courseSubscription.getMark() >= from && courseSubscription.getMark() <= to) { 3
            ++count;
        }
    }

    return count;
}
```

1. A seguito del fallimento del test **T10** è stato aggiunto un controllo per prevenire che from sia Maggiore di to,
2. A seguito del fallimento del test **T11** Aggiunto un controllo per prevenire che vengano effettuate operazioni se non vi sono studenti iscritti al Corso
3. A seguito del fallimento dei test **T7** e **T8** è stata aggiunta la condizione di inclusività, in quanto la specifica del metodo indica che nel risultato finale sono contati di default tutti gli studenti che hanno preso un voto che coincide con from e to

getStudentWithHigherMark()

```java
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 = orderedSet.first().getMark();

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

    return higherMarkStudents;
}
```

Comprensione dei requisiti

Il seguente metodo restituisce la matricola degli studenti con il voto più alto. Non possiede elementi in input. È possibile considerare come input "implicito" un insieme di studenti iscritti al corso, popolato con il metodo, analizzato dal **gruppo 1** *addNewCourseAttender*.

Il parametro restituito in output è un set che contiene le matricole degli studenti che hanno avuto il voto più alto.

Identificazione delle partizioni

Input

Non essendoci input nel metodo, per identificare una partizione si considera l'input **implicito** citato precedentemente: un **insieme di studenti iscritto al corso "s"**.

Insieme "s"
• Valore null • Insieme vuoto • Insieme pieno

Output

- Valore null
- Insieme vuoto
- Insieme pieno

Identificazione dei boundary cases

Non disponendo di informazioni sugli input, si considera come boundary case la presenza o meno di studenti iscritti al corso. **Ipotizzando** che per ottenere un risultato in output sia necessario che ci sia **almeno** uno studente iscritto al corso:

- **on point:** 0 [rende la condizione falsa]
- **off point**: 1 [rende la condizione vera]
- **in point**: numero di studenti > 0
- **out point**: 0

Ideare casi di test

T1: singolo studente in corso con voto valido

T2: più studenti in corso con voto valido, ma solo uno studente possiede il voto più alto

T3: più studenti in corso con voto valido e più studenti hanno preso lo stesso voto più alto

T4: singolo studente senza voto registrato

T5: più studenti iscritti al corso senza voto registrato

Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

Test su J-Unit

Test T1

Verifica la presenza nell'output del metodo, dell'unico studente a cui è stato assegnato il voto, come studente con voto più alto.

```
@Test // T1
@DisplayName("One student with an assigned mark")
void oneStudentWithAssignedMark() throws CourseEmptyException {
    courseManager1.assignMarkToStudent( mark: 30,  mat: "111111");

    Set<Student> studentSet = courseManager1.getStudentsWithHigherMark();

    Assertions.assertAll(
            () -> Assertions.assertEquals( expected: 1, studentSet.size()),
            () -> Assertions.assertTrue(studentSet.contains(s1))
    );
}
```

Test T2

Verifica la presenza nell'output del singolo studente che ha preso il voto più alto, considerando però questa volta che a più studenti è stato assegnato un voto.

```
@Test // T2
@DisplayName("More students with an assigned mark, but with one highest one")
void moreStudentWithAssignedMarkWithOneHighestMark() throws CourseEmptyException {
    courseManager1.assignMarkToStudent( mark: 30,  mat: "111111");
    courseManager1.assignMarkToStudent( mark: 29,  mat: "111112");

    Set<Student> studentSet = courseManager1.getStudentsWithHigherMark();
    Assertions.assertAll(
            () -> Assertions.assertEquals( expected: 1, studentSet.size()),
            () -> Assertions.assertTrue(studentSet.contains(s1))
    );
}
```

Test T3

Verifica la presenza nell'output di tutti gli studenti che hanno preso il voto più alto, considerando però questa volta che a più studenti è stato assegnato un voto.

```
@Test // T3
@DisplayName("More students with an assigned mark, but with more highest ones")
void moreStudentWithAssignedMarkWithMoreHighestMarks() throws CourseEmptyException {
    courseManager1.assignMarkToStudent( mark: 30,  mat: "111111");
    courseManager1.assignMarkToStudent( mark: 29,  mat: "111112");
    courseManager1.assignMarkToStudent( mark: 30,  mat: "111113");

    Set<Student> studentSet = courseManager1.getStudentsWithHigherMark();
    Assertions.assertAll(
            () -> Assertions.assertEquals( expected: 2, studentSet.size()),
            () -> Assertions.assertTrue(studentSet.contains(s1)),
            () -> Assertions.assertTrue(studentSet.contains(s3))
    );
}
```

Test T4 e T5

Verifica che nei corsi a cui nessun voto è stato assegnato, la dimensione dell'insieme di output è zero.

```
@Test //T4 e T5
@DisplayName("No Students with marks")
void noStudentsWithMarks() {
    Set<Student> studentSet1 = courseManager1.getStudentsWithHigherMark();
    Set<Student> studentSet2 = courseManager2.getStudentsWithHigherMark();
    Assertions.assertAll(
            () -> Assertions.assertEquals( expected: 0, studentSet1.size()),
            () -> Assertions.assertEquals( expected: 0, studentSet2.size())
    );
}
```

Test T6

```
@Test //T6
@DisplayName("No students in course 2")
void noStudentInCourse2() {
    courseManager1.deleteCourseStudents();

    Assertions.assertEquals( expected: 0, courseManager1.getStudentsWithHigherMark().size());
}
```

Modifiche al metodo post testing

```java
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 1

    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());
            System.out.println(courseSubscription.getStudent().getMat());
        } else {
            break;
        }
    }

    return higherMarkStudents;
}
```

2

1. Aggiunto un controllo per ottenere un insieme vuoto a seguito della mancanza di studenti iscritti al corso
2. A seguito del fallimento dei test **T4** e **T5**, è stato aggiunto un controllo che permette di ottenere un insieme vuoto qualora nessuno studente abbia un voto assegnato

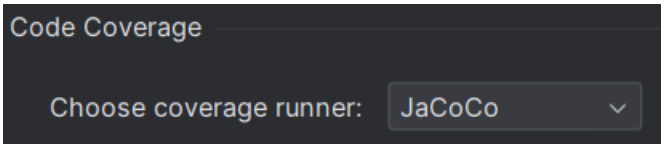
Homework 2 - Task 1

Analisi iniziale

Il primo step effettuato per eseguire lo **structural based testing**, dei metodi testati precedentemente con lo **specification based testing**, è stato quello di eseguire il tool di code coverage **Jacoco**. Di seguito, per ogni metodo viene riportata, l'analisi della code coverage, e gli eventuali test progettati per raggiungere il 100% di copertura.

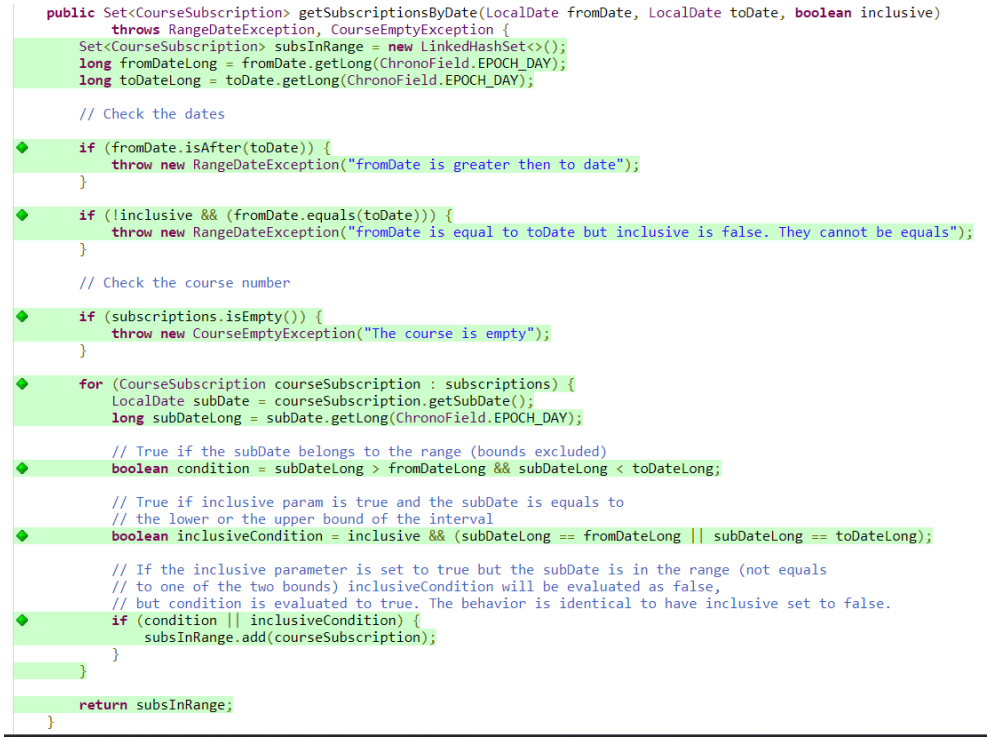
Il criterio di code coverage usato è quello di **branch coverage**. È sufficiente quindi esercitare i branch (true e false) per ciascun if presente, per raggiungere il massimo livello possibile di copertura per questo criterio.

Per i test case template, riferirsi al file Excel citato in precedenza per gli stessi metodi analizzati nell'homework 1.



getSubscriptionsByDate()

Come possibile notare dal seguente screen, per il metodo in questione i test black box effettuati sono risultati sufficienti per avere il 100% di code coverage.



```
public Set<CourseSubscription> getSubscriptionsByDate(LocalDate fromDate, LocalDate toDate, boolean inclusive)
        throws RangeDateException, CourseEmptyException {
    Set<CourseSubscription> subsInRange = new LinkedHashSet<>();
    long fromDateLong = fromDate.getLong(ChronoField.EPOCH_DAY);
    long toDateLong = toDate.getLong(ChronoField.EPOCH_DAY);

    // Check the dates

    if (fromDate.isAfter(toDate)) {
        throw new RangeDateException("fromDate is greater then to date");
    }

    if (!inclusive && (fromDate.equals(toDate))) {
        throw new RangeDateException("fromDate is equal to toDate but inclusive is false. They cannot be equals");
    }

    // Check the course number

    if (subscriptions.isEmpty()) {
        throw new CourseEmptyException("The course is empty");
    }

    for (CourseSubscription courseSubscription : subscriptions) {
        LocalDate subDate = courseSubscription.getSubDate();
        long subDateLong = subDate.getLong(ChronoField.EPOCH_DAY);

        // True if the subDate belongs to the range (bounds excluded)
        boolean condition = subDateLong > fromDateLong && subDateLong < toDateLong;

        // True if inclusive param is true and the subDate is equals to
        // the lower or the upper bound of the interval
        boolean inclusiveCondition = inclusive && (subDateLong == fromDateLong || subDateLong == toDateLong);

        // If the inclusive parameter is set to true but the subDate is in the range (not equals
        // to one of the two bounds) inclusiveCondition will be evaluated as false,
        // but condition is evaluated to true. The behavior is identical to have inclusive set to false.
        if (condition || inclusiveCondition) {
            subsInRange.add(courseSubscription);
        }
    }

    return subsInRange;
}
```

In questo specifico caso, visto l'alto numero di test progettati con lo specification based, si decide di fermarsi al criterio di branch coverage. La suite di test precedente non solo esercita efficacemente le condizioni in cui viene lanciata un'eccezione ma per quanto concerne la condizione più complessa, **la settima**, esercita numerose situazioni che vanno anche oltre la condizione di coverage imposta. Per l'espressione **condition**:

- Sono esercitate situazioni, al variare di inclusive, che prevedono che nessuno studente iscritto appartenga al range (test T6 e T8);
- Sono esercitate le situazioni, con inclusive false, che prevedono la presenza di studenti nel range specificato (test T5), o casi di boundary dove questi studenti sono iscritti in uno dei due limiti (test T12 e T13);

Tutte e quattro le condizioni per l'espressione condition sono esercitate dalla suite di test. Discorso analogo si può fare per **inclusiveCondition**:

- Testato il caso in cui inclusive è true e false (quasi tutta la suite di test lavora sulla variazione di questo parametro)

- Testati nei boundary test situazioni in cui studenti sono iscritti in date uguali ai due limiti e situazioni in cui sono iscritti in date facenti parte del range diverse dai due limiti, esplicitate nell'analisi del parametro condition.
    - Sono state testate nei test T17 e T18 anche situazioni in cui più di uno studente è iscritto, rispettivamente, in una data uguale al limite inferiore e superiore

Tutte le possibili condizion, precisamente quattro per la prima espressione e sei per la seconda, sono state esercitate, il che, ha permesso di ottenere il 100% di **c+b coverage**, che risulta essere un criterio sicuramente molto più rigoroso della sola branch coverage e sufficiente, secondo il team che ha progettato la suite di test, a stabilire l'efficacia della stessa.

countMarksInInclusiveRange()

Anche in questo caso, lo specification based testing è risultato efficace nel raggiungimento dell'obiettivo del 100% di code coverage.

```
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;
}
```

Similmente a quanto descritto per il precedente metodo, vista la suite di test progettata in fase di specification based testing, si è deciso di fermarsi al criterio di branch coverage.

Considerando l'espressione più complessa, presente all'interno del ciclo for, numerosi test prevedono l'esercitazione delle quattro condizioni presenti:

- Voto compreso tra from e to (test T5)
- Voto uguale a from e to (test T7 e test T8)

- Voti non appartenenti al range (test T6)

Anche per le prime due espressioni, i primi 4 casi di test progettati per questa suite permettono di esercitare le 8 condizioni totali.

Anche in questo caso, per queste espressioni complesse il criterio di c+b coverage è stato raggiunto al 100%.

getStudentWithHigherMark()

Come possibile notare in questo metodo i branch non coperti:

- Per l'if il caso in cui la condizione sia falsa, quindi che nessuno studente sia iscritto al corso
- Per il for il branch non coperto è quello che riguarda la non esecuzione del ciclo, condizione legata alla presenza o meno di studenti nel corso

```
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;
}
```

Per raggiungere il 100% di code coverage, quindi, basta un singolo test, riportato di seguito.

Ideare casi di test

T6: nessuno studente iscritto al corso

Test su J-Unit

Test T6

Non essendoci alcuno studente iscritto al corso, si verifica che l'output sia un insieme vuoto.

```
@Test //T6: sviluppato in seguito ai white box test
@DisplayName("No students in course 2")
void noStudentInCourse2() {
    courseManager1.deleteCourseStudents();
    Assertions.assertEquals( expected: 0, courseManager1.getStudentsWithHigherMark().size());
}
```

Di seguito è riportato lo screen del report di jacoco che mostra il 100% di code coverage per il metodo analizzato.

```
    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;
    }
```

Non essendo presenti espressioni complesse si è deciso di non effettuare ulteriori analisi, fermandosi al criterio di branch coverage del tool Jacoco.

Considerazioni finali

Avendo effettuato un processo di Black Box testing **sistematico ed efficace**, si può notare che in due metodi, sui tre totali, senza dovere effettuare test white box l'obiettivo di raggiungere il 100% di code coverage era già stato conseguito.

Per il terzo metodo invece, è bastato un solo test; quest'ultimo riguardava un caso già trattato negli altri metodi e che in fase di Black Box testing non è stato considerato, **erroneamente**, per questo specifico metodo.

Ciò dimostra l'importanza dei White Box test, il cui corretto utilizzo ha permesso di individuare un caso di test non considerato in ambito di Black Box testing.

Homework 2 - Task 2

studentsAboveAverage()

Il seguente metodo è usato per conteggiare tutti quegli studenti i cui voti sono superiori alla media di voti del corso. Il parametro di input booleano inclusive, determina se conteggiare o meno quegli studenti il cui voto è esattamente uguale alla media.

```java
public int studentsAboveAverage(boolean inclusive) {
    double avg = 0;
    double sum = 0;
    int hasMark = 0;
    int nStudent = 0;

    for (CourseSubscription courseSubscription : subscriptions) {
        if (courseSubscription.getMark() != -1) {
            hasMark++;
            sum += courseSubscription.getMark();
        }
    }

    if (hasMark != 0)
        avg = sum / hasMark;

    for (CourseSubscription courseSubscription : subscriptions) {
        int mark = courseSubscription.getMark();

        if ((inclusive && mark >= avg) || (!inclusive && mark > avg)) {
            nStudent++;
        }

    }

    return nStudent;
}
```

Ideare casi di test minimi per 100% di code coverage

T1: nessuno studente iscritto al corso

T2: almeno una iscrizione al corso con voto valido, con inclusive false

T3: almeno una iscrizione al corso con voto valido, con inclusive true

Test case template

Vedasi il file Excel consegnato nella cartella della relazione.

Test su J-Unit

Test T1

Non avendo alcun studente iscritto al corso, si verifica che l'output sia uguale a 0.

```
new *
@Test
@DisplayName("Course is empty")
void courseEmpty() {
    courseManager1.deleteCourseStudents();
    Assertions.assertEquals( expected: 0, courseManager1.studentsAboveAverage( inclusive: true));
}
```

Test T2 e T3

Test nella quale viene verificato che l'output sia corretto, avendo studenti con voto valido iscritti al corso

```
@Test
@DisplayName("Course is full with mark assigned")
void courseFullWithMarks() {
    courseManager1.assignMarkToStudent( mark: 26,  mat: "111112");
    courseManager1.assignMarkToStudent( mark: 27,  mat: "111113");
    courseManager1.assignMarkToStudent( mark: 28,  mat: "111114");

    Assertions.assertAll(
            () -> Assertions.assertEquals( expected: 2, courseManager1.studentsAboveAverage( inclusive: true)),
            () -> Assertions.assertEquals( expected: 1, courseManager1.studentsAboveAverage( inclusive: false))
    );
}
```

Casi di test non coperti

Analisi input e output

Considerando l'input booleano inclusive:

- Assume valore true
- Assume valore false

Non sono presenti combinazioni di input, essendo esso solo uno.

Per quanto concerne l'output le classi di output esso può assumere:

- Valore 0: quando nessuno studente è presente nel corso o con nessun voto assegnato

- Valore > 0: quando almeno uno studente ha voto superiore alla media generale
- Non può assumere valori negativi

Casi di test

T4: un solo studente iscritto senza voto assegnato

T5: più studenti iscritti, tutti senza voto assegnato

T6: più studenti iscritti, tutti con voto uguale e inclusive false

T7: più studenti iscritti, tutti con voto uguale e inclusive true

Ulteriore analisi MC/DC

Di seguito si riporta un'ulteriore analisi effettuata con la tecnica di code coverage MC/DC, al fine di individuare ulteriori test mancanti.

L'espressione analizzata è la seguente:

- (inclusive && mark >= avg) || (!inclusive && mark > avg)

Risulta però impossibile allo stato attuale raggiungere il 100% di questo criterio: inclusive e !inclusive non possono assumere lo stesso valore.

L'if è però scomponibile in due **if indipendenti** contenenti le condizioni tra parentesi. Se è vero uno, allora l'altro non lo sarà e viceversa. Questa scomposizione permette di analizzare l'MC/DC per ognuna delle espressioni estratte.

Si considera l'espressione: (inclusive && mark >= avg):

Test case	**inclusive**	**mark >= avg**	Decision
K1	T	T	T
K2	T	F	F
K3	F	T	F
K4	F	F	F

inclusive	{K1,K3}	già coperto
mark >= avg	{K1,K2}	già coperto

In questo caso le coppie indipendenti di test trovate sono state già esercitate:

- Il caso K1 corrisponde al test T3 o T7;
- Il caso K3 corrisponde al test T2 o T6;
- Il caso K2 corrisponde ai casi di test in cui non vi è nessuno studente in corso o con voto non assegnato;

Si considera ora l'espressione (!inclusive && mark > avg):

Test case	!inclusive	mark > avg	Decision
J1	T -> F	T	F
J2	T -> F	F	F
J3	F -> T	T	T
J4	F -> T	F	F

!inclusive:	{J1,J3}	già coperto
mark > avg	{J3,J4}	già coperto

- Il caso J1 corrisponde al test T2 o T6;
- Il caso J3 corrisponde al test T3 o T7;
- Il caso J4 corrisponde ai casi di test in cui non vi è nessuno studente in corso o con voto non assegnato;

Considerazioni finali

Com'è possibile notare, sono bastati due test per avere il 100% di code coverage, ciò dimostra che, come da attesa, la metrica di code coverage da sola non è sufficiente per coprire un vasto numero di casi di test. I casi di test non coperti ci dimostrano che effettuare solo analisi di code coverage senza considerare le operazioni effettuate nella fase di Specification Based testing come ad esempio analisi delle partizioni, identificazione dei boundary cases, porta all'ottenimento di una suite di test incompleta anche se coperta al 100%.

Homework 3 – Property Based Testing

getSubscriptionsByDate()

L'analisi di questo metodo, per specification based test e structural based test è stata effettuata nei precedenti homework.

Proprietà del metodo

- **fail**: date in input corrette e output vuoto. Il valore di inclusive è indifferente;
- **invalid**: date in input non corrette (fromDate > toDate o fromDate = toDate con inclusive true). Il valore inclusive deve essere sempre settato a false, in quanto se sono presenti date uguali esse sono considerate non corrette solamente in questo caso;
- **success**: input corretti e output pieno. Il valore del parametro inclusive genera alcune differenze.

Gli screen per gli output delle statistiche sono inseriti in un paragrafo separato dalla descrizione dei singoli metodi di PBT. Nonostante ciò, una descrizione delle eventuali operazioni eseguite per l'ottenimento di una determinata statistica, è riportata in quella del metodo che la implementa. Tale suddivisione è resa necessaria per una maggiore chiarezza grafica.

Per i test case template, riferirsi al file Excel citato in precedenza per gli stessi metodi analizzati nell'homework 1.

Definizione PBT per fail property (T1)

La proprietà fail prevede di ricevere in input due date corrette. Siccome nessuno studente è iscritto al corso nel range formato dalle due date fornite, l'output del metodo risulta essere un insieme vuoto.

Descrizione implementazione

```
@Property // T1
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void fail(
        @ForAll @DateRange(min = FROM_MIN, max = FROM_MAX) LocalDate from,
        @ForAll @DateRange(min = TO_MIN, max = TO_MAX) LocalDate to) throws NullStudentException,
        RangeDateException, CourseEmptyException {
    CourseManager courseManagerPBT = generateFixedCourse();
    Assertions.assertEquals( expected: 0, courseManagerPBT
            .getSubscriptionsByDate(from, to, inclusive: false).size());

    // Dimostra che ogni possibile combinazione delle date fornite dal range di input è considerata
    Statistics.collect(from, to);
}
```

Il caso di test definito prevede la definizione di due input:

- Il limite inferiore del metodo testato, generato da un **DateRange** che va da un valore di data minimo a un massimo (inclusi entrambi)
- Il limite superiore del metodo testato, generato da un **DateRange** che va da un valore di data minimo a un massimo (inclusi entrambi)

Dopo aver definito il corso, popolato da valori fissi, che soddisfano la proprietà testata, è eseguita un **assertEquals** che controlla che nessuno studente rientri nel range di date considerate. Il metodo di test è eseguito per ogni possibile combinazione degli input generati.

La statistica calcolata è usata per dimostrare che ogni possibile combinazione di date from e to viene effettuata.

Definizione PBT per invalid property (T2)

La proprietà invalid prevede il lancio di un'eccezione nel caso in cui i dati forniti in input non siano corretti. Essi sono i seguenti:

- fromDate > toDate
- fromDate = toDate con inclusive false

Ne consegue che per testare questa proprietà, se nel primo caso il valore del parametro inclusive sia indifferente, nel secondo invece è essenziale che sia false per soddisfare la proprietà.

Descrizione implementazione

```
@Property // T2
@Report(Reporting.GENERATED)
void invalid(
        @ForAll @DateRange(min = TO_MIN, max = TO_MAX) LocalDate from,
        @ForAll @DateRange(min = FROM_MAX, max = TO_MIN) LocalDate to) throws NullStudentException {
    CourseManager courseManagerPBT = generateFixedCourse();
    Assertions.assertThrows(
            RangeDateException.class, () ->
                    courseManagerPBT.getSubscriptionsByDate(from, to, inclusive: false)
    );

    // Dimostra la presenza di date uguali o non uguali
    Statistics.collect(from.equals(to) ? "Uguali" : "from > to");
}
```

A livello operativo il metodo di test, dopo aver generato i range di date in maniera analoga al metodo precedente si accerta del lancio dell'eccezione **RangeDateException** per ogni combinazione di date.

La statistica definita indica il numero di date uguali rispetto a quelle in cui semplicemente from è > to.

Definizione PBT per success property (T3)

La proprietà success prevede che l'output sia un insieme pieno composto da tutti gli studenti iscritti in date appartenenti al range fornito in input. Si distinguono due casistiche:

- Se inclusive è settato a true la proprietà success garantisce che l'output contenga tutti gli studenti iscritti al corso, *nel range specificato*, compresi anche gli studenti la cui data di iscrizione coincide con il limite superiore o inferiore;
- Se inclusive è settato a false la proprietà success garantisce che l'output contenga tutti gli studenti iscritti al corso, *nel range specificato*, senza gli studenti la cui data di iscrizione coincide con il limite superiore o inferiore.

Descrizione implementazione

```
@Property(tries = 40)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Label("PBT - SUCCESS")
void success(@ForAll("generateRandomSubscriptions")List<Tuple2<Student, LocalDate>> subscriptions) {
    LocalDate from = LocalDate.parse(FROM_MIN);
    LocalDate to = LocalDate.parse(TO_MAX);

    // Conta quante date sono state generate in range escludendo il limite
    // superiore ed inferiore
    int notInclusiveSubsCounter = 0;

    String rangeGlobal = "";

    // Questo ciclo è usato per collezionare la statistica e per
    // controllare quante date di iscrizione sono state generate nel range
    // non inclusivo dei due limit inferiore e superiore
    for (Tuple2<Student, LocalDate> sub : subscriptions) {
        LocalDate subDate = sub.get2();

        // Colleziona valori per il range usato nella statistica
        if (subDate.equals(from) || subDate.equals(to)) {
            rangeGlobal = "Inclusive sub";
        } else {
            notInclusiveSubsCounter++;
            rangeGlobal = "Not inclusive sub";
        }

        // Genera due statistiche. La prima individua in che percentuali vengono generate iscrizioni
        // in range inclusive (quindi iscrizioni che corrispondono ai due limiti del range) e
        // quante vole sono generate iscrizioni in date appartenenti al range ma diverse dai due limiti.
        Statistics.label("Global range").collect(rangeGlobal);
    }
```

Si considera il range di date ***FROM_MIN – TO_MAX***. L'input del metodo è composto da una lista, **subscriptions,** di studenti iscritti nelle date generate dal metodo specificato dall'annotazione **@ForAll("generateRandomSubscriptions")** il cui metodo **@Provide** verrà discusso in seguito. La lista è composta da un numero di tuple pari alla costante **LIMIT_GENERATIONS** pari a mille, contenti in ordine:

1. **Student**: lo studente generato arbitrariamente
2. **LocalDate**: la data generata arbitrariamente, compresa nel range

Nella prima parte, il metodo, definisce il range nel quale sono iscritti gli studenti generati, memorizzandolo nella variabili from e to nelle variabili from e to. Il ciclo for ha un duplice scopo: collezionare la statistica e contare il numero di date di iscrizione generate appartenenti al range ma diverse dai due limiti. Il risultato è memorizzato in una variabile che verrà utilizzata nella seconda parte del metodo di test, di seguito riportato.

```
    // Le lambda expression ammettono valori final come parametri
    final int finalNotInclusiveSubsCounter = notInclusiveSubsCounter;
    CourseManager courseManager = fillCourseManager(subscriptions);

    // Testa i casi in cui inclusive sia false e true
    Assertions.assertAll(
            // Con inclusive true, ogni studente iscritto nel range specificato
            // deve essere presente nell'output.
            () -> Assertions
                    .assertEquals(LIMIT_GENERATIONS,
                            courseManager.getSubscriptionsByDate(from, to, inclusive: true).size()),

            // Con inclusive false l'output è composto da tutti gli studenti, tranne quelli
            // iscritti in una data che coincide con uno dei due limiti.
            () -> Assertions
                    .assertEquals(finalNotInclusiveSubsCounter,
                            courseManager.getSubscriptionsByDate(from, to, inclusive: false).size())
    );

}
```

Nella seconda parte del metodo, dopo aver riempito il CourseManager viene controllato tramite **assertAll** che:

- Il numero di studenti di getSubscriptionsByDate **corrisponda al numero di iscrizioni generate**, essendo esse definite nel range indicato.
- Il numero di studenti corrisponda a quello di un contatore globale, che all'interno del metodo annotato dal @ForAll, conta tutte le date generate diverse dai due limiti ma all'interno nel range

L’uso dell’assertAll è dovuto alla necessità di dover testare la situazione in cui il parametro inclusive assume valori false e true. È già stato ampiamente discusso e dimostrato, nel primo homework, infatti, che il comportamento del metodo testato varia in funzione del valore del parametro booleano **inclusive**.

Metodi providers per i PBT

Arbitrary generateRandomSubscriptions()

```
no usages  Giacomo Detomaso +1
@Provide
Arbitrary<List<Tuple2<Student, LocalDate>>> generateRandomSubscriptions() {
    Arbitrary<String> name = Arbitraries.strings().ofLength(4);
    Arbitrary<String> secondName = Arbitraries.strings().ofLength(5);
    Arbitrary<String> mat = Arbitraries.strings().numeric().ofLength(6);

    // Combinando gli input viene generato lo studente arbitrario
    Arbitrary<Student> studentArbitrary =
            Combinators
            .combine(name, secondName, mat)
            .as(Student::new);

    LocalDate from = LocalDate.parse(FROM_MIN);
    LocalDate to = LocalDate.parse(TO_MAX);

    LocalDateArbitrary localDateArbitrary = new DefaultLocalDateArbitrary();
    localDateArbitrary = localDateArbitrary.between(from, to);

    // Restituisce, dopo la combinazione, una lista di Tuple2.
    return Combinators
            .combine(studentArbitrary, localDateArbitrary)
            .as(Tuple::of)
            .list()
            // La prima entry di ogni tupla nella lista (lo Studente) deve essere univoca.
            // Lo studente possiede un metodo equals che stabilisce che due
            // studenti uguali possiedono la stessa matricola. Pertanto, nell'arbitrary in output,
            // nessuno studente possiederà la stessa matricola.
            .uniqueElements(Tuple.Tuple1::get1)
            .ofSize(LIMIT_GENERATIONS);
}
```

Il metodo **generateRandomSubscriptios** usato come provider per gli input del PBT success ha il compito di restituire una lista di tuple (la cui struttura è già stata descritta) con la caratteristica di contenere studenti NON duplicati.

Secondo la definizione del metodo equals della classe Student infatti due studenti sono duplicati se possiedono la stessa matricola.

Dopo aver generato lo studente arbitrario usando il Combinator sui parametri name, secondName e mat, è stata generata anche la data arbitraria nel range usato dal test. Il metodo **between** genera date in un range inclusivo.

Nella parte conclusiva del metodo, tramite un **chaining** sono state:

- Combinate date e studente arbitrario all'interno di una tupla;
- A partire dalla tupla arbitraria, si genera una lista di tuple anch'essa arbitraria;
- La condizione con cui è generata la lista è espressa dal metodo **uniqueElements**, dove con la lambda passata in input si specifica che il parametro della tupla non duplicato deve essere quello dello studente.

CourseManager fillCourseManager()

```
/**
 * Il metodo riempie e restituisce un corso sulla base delle iscrizioni
 * generate arbitrariamente.
 *
 * @param subscriptions le iscrizioni generate
 * @return il corso riempito con le iscrizione generate
 * */
private CourseManager fillCourseManager(List<Tuple2<Student, LocalDate>> subscriptions) {
    CourseManager courseManager = new CourseManager(
            courseName: "Integrazione e test Gruppo 2 - getSubscriptionByDate",
            LocalDate.now());


    for (Tuple2<Student, LocalDate> k : subscriptions) {
        try {
            courseManager.addNewCourseAttender(k.get1(), k.get2());
        } catch (Exception e) {
            System.out.println("Eccezione lanciata: " + e.getMessage());
        }
    }

    return courseManager;

}
```

Il metodo è usato, a partire dalla lista di tuple generate dal metodo generateRandomSubscriptions(), per riempire con tali valori un corso e restituirlo.

CourseManager generateFixedCourse()

```
@Provide
CourseManager generateFixedCourse() throws NullStudentException {
    CourseManager courseManagerPBT = new CourseManager(
            courseName: "Integrazione e test Gruppo 2 - PBT",
            LocalDate.now());

    courseManagerPBT.addNewCourseAttender(s1, LocalDate.parse(text: "2023-01-01"));
    courseManagerPBT.addNewCourseAttender(s2, LocalDate.parse(text: "2023-01-02"));
    courseManagerPBT.addNewCourseAttender(s3, LocalDate.parse(text: "2023-01-03"));
    courseManagerPBT.addNewCourseAttender(s4, LocalDate.parse(text: "2023-01-04"));

    return courseManagerPBT;
}
```

Il metodo genera un corso con un numero prefissato di elementi.

Statistiche

T1

La statistica dimostra che ogni combinazione di date viene esercitata.

#	label	count	
0	2023-02-01 2023-02-15	1	#
1	2023-02-01 2023-02-16	1	#
2	2023-02-01 2023-02-17	1	#
3	2023-02-01 2023-02-18	1	#
4	2023-02-01 2023-02-19	1	#
5	2023-02-01 2023-02-20	1	#
6	2023-02-01 2023-02-21	1	#
7	2023-02-01 2023-02-22	1	#
8	2023-02-01 2023-02-23	1	#
9	2023-02-01 2023-02-24	1	#
10	2023-02-01 2023-02-25	1	#
11	2023-02-02 2023-02-15	1	#
12	2023-02-02 2023-02-16	1	#
13	2023-02-02 2023-02-17	1	#
14	2023-02-02 2023-02-18	1	#
15	2023-02-02 2023-02-19	1	#
16	2023-02-02 2023-02-20	1	#
17	2023-02-02 2023-02-21	1	#
18	2023-02-02 2023-02-22	1	#
19	2023-02-02 2023-02-23	1	#
20	2023-02-02 2023-02-24	1	#
21	2023-02-02 2023-02-25	1	#
22	2023-02-03 2023-02-15	1	#
23	2023-02-03 2023-02-16	1	#
24	2023-02-03 2023-02-17	1	#
25	2023-02-03 2023-02-18	1	#

T2

Solo in un caso accade che from sia uguale a to, come ci si aspettava, visto che le combinazioni di input vengono esercitate una sola volta.

```
from > to (65) : 98 %
Uguali    ( 1) :  2 %
```

T3

Il numero di iscrizioni la cui data risiede nel range ma è diversa dei limiti è di molto superiore. Di seguito i valori in percentuale:

- Inclusive: circa 8%
- Not inclusive sub: circa il 92%

Il tutto su 1000 studenti generati, per ogni lista con 40 tentativi a disposizione. Il totale è quindi di 40000 generazioni arbitrarie.

```
    # |            label | count |
-----|------------------|-------|-----------------------------------------------------------------------------------
    0 |    Inclusive sub |  3266 | #######
    1 | Not inclusive sub | 36734 | ################################################################################
```

Appendice

Junit life cycle

BeforeAll

```
@BeforeAll
static void setup() {
    courseManager1 = new CourseManager(
            "Integrazione e test Gruppo 2 - getSubscriptionByDate",
            LocalDate.now());

    courseManager2 = new CourseManager(
            "Integrazione e test Gruppo 2 - getStudentWithHigherMark",
            LocalDate.now());
}
```

Nel metodo BeforeAll effettuiamo l'istanziazione dei due oggetti courseManager che vengono utilizzati in tutti i casi di test.

BeforeEach

```
@BeforeEach
void init()  {
    // Aggiunta studenti per courseManager1
    try {
        courseManager1.addNewCourseAttender(s1, LocalDate.parse("2022-11-04"));
        courseManager1.addNewCourseAttender(s2, LocalDate.parse("2022-11-10"));
        courseManager1.addNewCourseAttender(s3, LocalDate.parse("2022-11-11"));
        courseManager1.addNewCourseAttender(s4, LocalDate.parse("2022-11-11"));
    }
    catch(Exception e) {
        System.out.println("Eccezione lanciata 1: " + e.getMessage());
    }

    try {
        courseManager2.addNewCourseAttender(s1, LocalDate.parse("2022-11-04"));
    }
    catch(Exception e) {
        System.out.println("Eccezione lanciata 2: " + e.getMessage());
    }
}
```

Nel metodo BeforeEach andiamo a valorizzare i due oggetti creati in precedenza, effettuando l'atto di iscrizione degli studenti definiti al corso.

Tale operazione, come descritto nel template, è dovuta a questioni di ottimizzazione. Per tutti i casi di test si è infatti lavorato con lo stesso insieme di studenti.

AfterEach

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

Nel metodo AfterEach eliminiamo tutti gli studenti da courseManager1.

AfterAll

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

Nel metodo AfterAll viene imposto a null courseManager1 al termine dell'esecuzione della suite di test.