DIETI UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

ELABORATO DI ESAME DI SOFTWARE TESTING

Tecniche di Testing

Giuseppe Ferrara

M63/001333

Sommario

INTRODUZIONE 3

CALENDARIO 3

ROMAN NUMERAL 4

INFLECTION 7

FTPFILE 8

FONTINFO 11

BYTEVECTOR 14

RANGE 21

RATIONAL NUMBER 30

Introduzione

In questo elaborato ho effettuato il testing di varie classi Java utilizzando diverse tecniche di testing per ottenere la più alta copertura possibile. Le tecniche che ho utilizzato sono diverse, spesso mi sono affidato alle classi di equivalenza per cercare di coprire tutte (o quasi) le casistiche differenti. In alcuni casi siccome i possibili input erano troppi ho utilizzato come oracolo un'altra classe.
Di seguito verranno illustrati i test più interessanti per ogni classe.

Calendario

La classe Calendario è una classe Java che ha 3 possibili valori in ingresso, un valore che rappresenta il giorno (intero), un valore che rappresenta il mese (stringa), un valore che rappresenta un anno (intero) e restituisce, se valido, il giorno della settimana corrispondente altrimenti restituisce la stringa Errore.
Per il testing di questa classe sono state applicate diverse strategie che illustrerò di seguito.
Questa classe ha il solo metodo calend, quindi tutte le casistiche puntano alla massima copertura possibile dello stesso metodo.
Di seguito la descrizione del caso di test in forma tabellare

ID	Precond	Input	Expected Output	Post Cond
1		d = 1; ms = gennaio; a = 2018;	Lunedi	
2		d = 0; ms = gennaio; a = 2018;	Errore	
3		d = 1; ms = Brumaio; a = 2018;	Errore	
4		d = 1; ms = gennaio; a = 1050;	Errore	

Questa tabella consente di generare i differenti Test Case da implementare in Junit.

```
@CsvSource({
        "1, gennaio, 2018, Lunedi",
        "0, gennaio, 2018, Errore",
        "1, Brumaio, 2018, Errore",
        "1, gennaio, 1050, Errore",
})
```

ID	Giorno	Mese	Anno	Ris. Atteso
1	1	gennaio	2018	Lunedi
2	0	gennaio	2018	Errore
3	1	Brumaio	2018	Errore
4	1	gennaio	1050	Errore

La prima strategia adottata è stata realizzata utilizzando la tecnica delle classi di equivalenza. Ho individuato le classi di equivalenza per i 3 input possibili, ognuno dei quali può essere valido o non valido. Per questo test è stata usata la tecnica di copertura BCC che implica un totale di 4 casi di test. Con questa tecnica di copertura non riusciamo a testare tutti i possibili valori dei mesi differenti e non riusciamo a testare casi particolari come il 29 febbraio o il 31 aprile.
La seguente tabella mostra l'esito dei test effettuati.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1		d = 1; ms = gennaio; a = 2018;	Lunedi	Lunedi		Success
2		d = 0; ms = febbraio; a = 2019;	Errore	Errore		Success
3		d = 4; ms = Brumaio; a = 2019;	Errore	Errore		Success
4		d = 5; ms = marzo; a = 1050;	Errore	Errore		Success

Utilizzando lo strumento ecfeed ho costruito la classe calendario dove ad ogni input gli ho assegnato un possibile valore valido e un possibile valore non valido.
In particolare, come valori possibili per il giorno ho scelto {0,1,28,29,30,31}, come valori possibili per il mese ho scelto {Febbraio, Giognof}, come valori possibili per l'anno ho scelto {1500,2015,2024}.
I valori scelti per il giorno rappresentano i valori limite, cioè quei valori che statisticamente è più probabile che possano portare problemi. Per l'anno ho scelto il mese di febbraio perché è quello che avendo un numero di giorni variabile e diverso da tutti gli altri è più difficile da gestire. Come valori per l'anno ho inserito un anno valido, uno non valido e uno bisestile.
Ho poi generato automaticamente 3 tipi di testcase, 2-wise, 3-wise e cartesian.
Essendo in questo caso 3 input differenti il numero dei test del caso 3-wise e del caso cartesian sono uguali.
Di seguito una descrizione tabellare di alcuni dei test

ID	Precond	Input	Expected Output	Post Cond
3		d = 30; ms = febbraio; a = 2024;	Errore	
7		d = 1; ms = febbraio; a = 2024;	Giovedi	
9		d = 31; ms = febbraio; a = 2024;	Errore	
13		d = 15; ms = febbraio; a = 2015;	Domenica	

Sono riuscito ad arrivare ad una copertura del 90%.
Come ultimo test ho generato, mediante uno script in python le date dal 1° gennaio 1581 al 31 marzo 2024. Per questo test ho utilizzato come oracolo una combinazione di classi, la classe **LocalDate** di java che ha lo scopo di verificare che la data inserita sia valida e la classe **Calendar** con la quale riesco a ricavare il giorno della settimana.
Ho utilizzato due classi perché calendar controlla se la data è valida ma è molto tollerante, inserendo 30 febbario di un anno non bisestile, ad esempio, la data viene convertita in 2 marzo automaticamente. Ho utilizzato quindi localdate per verificare la validità della data, successivamente creo un oggetto jCalendar di tipo Calendar e confronto la data ottenuta dalla classe Calendario.
Utilizzando queste tecniche di test sono riuscito ad arrivare ad una copertura della classe Calendario del 96%.

Roman Numeral

La classe Roman Numeral è una classe Java che ha un metodo che si occupa della conversione di un numero romano, rappresentato come stringa, al numero intero corrispondente.
Poiché testare questa classe con la tecnica delle classi di equivalenza non avrebbe dato buoni risultati in termini di copertura ho deciso di adottare un'altra strategia. Non avendo un oracolo automatico in questo caso e dovendolo fare ha mano ho limitato il numero di test case.

Di seguito una descrizione tabellare di alcuni dei casi di test.

ID	Precond	Input	Expected Output	Post Cond
1	RomanNumeral romanNumeral esiste	LXXI	71	
2	RomanNumeral romanNumeral esiste	DCCCXLIII	843	
3	RomanNumeral romanNumeral esiste	CCLVIII	258	
4	RomanNumeral romanNumeral esiste	XIV	14	
5	RomanNumeral romanNumeral esiste	AA	RuntimeException	
6	RomanNumeral romanNumeral esiste	S	RuntimeException	
7	RomanNumeral romanNumeral esiste	G	RuntimeException	

```java
@ParameterizedTest
  @CsvFileSource(resources = "/romani.csv", numLinesToSkip = 1)
  public void provaNumeroValidoDaFile(String romano, String expected) {
    if(expected.equals("INVALID")){
      assertThrows(RuntimeException.class, () -> romanNumeral.romanToInt("AI"),
        "Expected RuntimeException");
    } else {
      assertEquals(Integer.parseInt(expected), romanNumeral.romanToInt(romano));
    }
  }
```

Per testare questa classe ho generato casualmente 95 numeri romani compresi tra 1 e 1000 e manualmente ho scritto nel file csv il corrispondente valore in numero intero. Ho poi aggiunto circa 10 numeri romani errati per testare le varie casistiche di errore.
Mediante questi test ho raggiunto una copertura del 100% del codice.

Di seguito un report di alcuni dei casi di test descritti in precedenza

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	RomanNumeral romanNumeral esiste	LXXI	71	71		Success
2	RomanNumeral romanNumeral esiste	DCCCXLIII	843	843		Success
3	RomanNumeral romanNumeral esiste	CCLVIII	258	258		Success
4	RomanNumeral romanNumeral esiste	XIV	14	14		Success
5	RomanNumeral romanNumeral esiste	AA	RuntimeException	RuntimeException		Success
6	RomanNumeral romanNumeral esiste	S	RuntimeException	RuntimeException		Success
7	RomanNumeral romanNumeral esiste	G	RuntimeException	RuntimeException		Success

Questa strategia di testing riesce a raggiungere il massimo valore della copertura ma è poco efficiente, infatti sono necessari moltissimi test.

Provo ad effettuare il testing dello stesso metodo con una differente strategia, utilizzo le tabelle di decisione. Analizzando il codice della clase si nota che un numero romano può essere di una o più cifre appartenenti al seguente insieme {I,V,X,L,C,D,M}.
Nel caso in cui il numero è a più cifre la seconda viene sottratta (se si trova a sinistra) o aggiunta (se si trova a destra. Ho costruito quindi un nuovo scenario di test con i seguenti test case

ID	Precond	Input	Expected Output	Post Cond
1	RomanNumeral romanNumeral esiste	I	1	
2	RomanNumeral romanNumeral esiste	V	5	
3	RomanNumeral romanNumeral esiste	X	10	
4	RomanNumeral romanNumeral esiste	L	50	
5	RomanNumeral romanNumeral esiste	C	100	
6	RomanNumeral romanNumeral esiste	D	500	
7	RomanNumeral romanNumeral esiste	M	1000	
8	RomanNumeral romanNumeral esiste	IV	4	
9	RomanNumeral romanNumeral esiste	VI	6	
10	RomanNumeral romanNumeral esiste	P	RuntimeException	

Utilizzando questa strategia di testing ho ottenuto sempre una copertura del 100% ma con soli 10 casi di test. Questa strategia è quindi molto più efficiente della precedente.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	RomanNumeral romanNumeral esiste	I	1	1		Success
2	RomanNumeral romanNumeral esiste	V	5	5		Success
3	RomanNumeral romanNumeral esiste	X	10	10		Success
4	RomanNumeral romanNumeral esiste	L	50	50		Success
5	RomanNumeral romanNumeral esiste	C	100	100		Success
6	RomanNumeral romanNumeral esiste	D	500	500		Success
7	RomanNumeral romanNumeral esiste	M	1000	1000		Success
8	RomanNumeral romanNumeral esiste	IV	4	4		Success
9	RomanNumeral romanNumeral esiste	VI	6	6		Success
10	RomanNumeral romanNumeral esiste	P	RuntimeException	RuntimeException		Success

Inflection

La classe inflection è una classe che ha due metodi principali, pluralize e singularize. Entrambi prendono in ingresso una stringa contenente una parola e rispettivamente restituisce la versione plurale o singolare della parola in ingresso.
La classe trasforma la parola al singolare o al plurale utilizzando espressioni regolari statiche e parole irregolari o invarianti che sono presenti staticamente o possono essere aggiunte dinamicamente.
Per effettuare il testing di questa classe ho utilizzato la tecnica delle classi di equivalenza. In particolare, ho utilizzato le seguenti classi di equivalenza. Per il metodo pluralize ho individuato 3 classi di equivalenza che rappresentano una parola regolare, irregolare o invariante. Analogamente anche per il metodo singularize ho individuato le stesse classi di equivalenza.

ID	Descrizione	Valore
CE1	Singolare regolare	Dog
CE2	Singolare irregolare	Person
CE3	Singolare invariante	Equipment
CE4	Plurale regolare	Dogs
CE5	Plurale irregolare	People
CE6	Plurale invariane	equipment

A partire da queste classi di equivalenza ho generato i test case utilizzando il criterio di copertura ECC, utilizzando ogni classe una volta, sia per il metodo pluralize che per il singularize.

Test case pluralize

ID	Precond	Input	Expected Output	Post Cond
1		Dog	Dogs	
2		Person	People	
3		Equipment	Equipment	

Test case singularize

ID	Precond	Input	Expected Output	Post Cond
4		Dogs	Dog	
5		People	Person	
6		Equipment	Equipment	

```
@CsvFileSource(resources = "/pluralize.csv", numLinesToSkip = 1)
@ParameterizedTest
public void testPluralize(String word, String expected) {
    System.out.println("Testo pluralize con " + word + " e mi aspetto " + expected + " ed ottengo " + Inflection.pluralize(word));
    assertEquals(expected, Inflection.pluralize(word));
}



@CsvFileSource(resources = "/pluralize.csv", numLinesToSkip = 1)
@ParameterizedTest
public void testSingularize(String expected, String word) {
    System.out.println("Testo pluralize con " + word + " e mi aspetto " + expected + " ed ottengo " + Inflection.pluralize(word));
    assertEquals(expected, Inflection.singularize(word));
}
```

Per eseguire i test progettati ho scritto due funzioni differenti che invocano rispettivamente il metodo singularize e pluralize. I test realizzati sono test parametrici che ricevono in ingresso due stringhe che rispettivamente sono la parola da trasformare al plurale o al singolare e il valore atteso. Per semplicità come valori in ingresso ai test h utilizzato lo stesso file csv che viene richiamato dai due metodi di test con i parametri invertiti per testare sia il metodo pluralize che il metodo singularize, essendo quest'ultimo il duale.
In questo modo ho raggiunto il 96% di copertura del codice della classe.
Di seguito il report rispettivamente dei test eseguiti sul metodo pluralize e singularize.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1		Dog	Dogs	Dogs		Success
2		Person	People	People		Success
3		Equipment	Equipment	Equipment		Success

ID	Precond	Input	Expected Output	Output	Post Cond	Result
4		Dogs	Dog	Dog		Success
5		People	Person	Person		Success
6		Equipment	Equipment	Equipment		Success

FtpFile

La classe ftp file è una classe che rappresenta un file memorizzato su un ftp server. I primi 4 test sono test molto semplici che rispettivamente testano i costruttori e il metodo set e get della proprietà rawlisting. Questa proprietà è la rappresentazione in forma testuale di tutte le proprietà che ha un file, quindi il nome, la data e ora di modifica, la dimensione, il nome e il gruppo del proprietario e per tutti e 3 i livelli i privilegi.
Per testare il tipo di file sono stati realizzati due test parametrizzati, il primo che riceve in ingresso il tipo, crea un file, imposta il tipo e controlla che il tipo del file sia rimasto invariato. Il secondo invece imposta il tipo e per controllare non utilizza la get type ma utilizza le funzioni booleane che restituiscono true se il file è di un certo tipo.
Un test molto interessante è il test del metodo set permissions che setta i permessi di lettura, scrittura ed esecuzione a diversi livelli. I valori in ingresso sono 4:

- Access
- Permission
- Value
- Expected

Il primo indica il livello di accesso, varia da 0 a 2 e rappresenta l'utente il gruppo o chiunque. Il secondo che è anch'esso un intero che varia da 0 a 2 indica per il livello scelto precedentemente che tipologia di permesso si sta assegnando quindi se lettura, scrittura o esecuzione. Il terzo indica se il permesso è assegnato o meno. Expected indica se il permesso c'è o no. Per questo test ho realizzato 3 classi di equivalenza per i 3 input differenti. Queste classi possono assumere ciascuna un valore compreso tra {-1, 3}. Ho scelto questi valori perché rappresentano i possibili valori validi uniti a 2 valori al di fuori del limite.

Per realizzare i test case ho realizzato il seguente modello utilizzando lo strumento ctwedge.

```
1. Model Permissions
2.  Parameters:
3.    permission : [-1 .. 3]
4.    access: [-1 .. 3]
5.    value : Boolean
6.
```

Ho poi generato i test case automaticamente, siccome i test ottenuti non sarebbero stati tanti ho utilizzato il criterio 3-wise per ottenere la massima copertura possibile.

Di seguito riporto un estratto della tabella relativa ai test case realizzati.

ID	Precond	Input	Expected Output	Post Cond
1	FtpFile ftpFile esiste	permission = 0; access = 0; value = true;	true	permissions[0][0] = true
2	FtpFile ftpFile esiste	permission = 0; access = 0; value = false;	false	permissions[0][0] = false
3	FtpFile ftpFile esiste	permission = 1; access = 0; value = true;	true	permissions[1][0] = true
4	FtpFile ftpFile esiste	permission = 1; access = 0; value = false;	false	permissions[1][0] = false
5	FtpFile ftpFile esiste	permission = -1; access = -1; value = true;	error	
6	FtpFile ftpFile esiste	permission = 3; access = 3; value = false;	error	

```
@CsvFileSource(resources = "/test_permissions.csv", numLinesToSkip = 1)
@ParameterizedTest
public void testSetFTPFilePermissions(int access, int permission, boolean value, String expected) {
    FTPFile ftpFile = new FTPFile();
    if (expected.equals("true") || expected.equals("false")) {
        ftpFile.setPermission(access, permission, value);
        assertEquals(Boolean.parseBoolean(expected), ftpFile.hasPermission(access, permission));
    } else if (expected.equals("null")) {
        assertFalse(ftpFile.hasPermission(access, permission));
    } else
        assertThrows(ArrayIndexOutOfBoundsException.class, () -> ftpFile.setPermission(access, permission, value));
}
```

Di seguito un estratto del report dei test in forma tabellare.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	FtpFile ftpFile esiste	permission = 0; access = 0; value = true;	true	true	permissions[0][0] = true	Success
2	FtpFile ftpFile esiste	permission = 0; access = 0; value = false;	false	false	permissions[0][0] = false	Success

3	FtpFile ftpFile esiste	permission = 1; access = 0; value = true;	true	true	permissions[1][0] = true	Success
4	FtpFile ftpFile esiste	permission = 1; access = 0; value = false;	false	false	permissions[1][0] = false	Success
5	FtpFile ftpFile esiste	permission = -1; access = -1; value = true;	error	true		Success
6	FtpFile ftpFile esiste	permission = 3; access = 3; value = false;	error	false		Success

L'ultimo test interessante è quello che testa il metodo **toFormattedString**.
Per la copertura di questo test ho utilizzato due classi di equivalenza, la prima che rappresenta un file con tutti i permessi possibili, la seconda che rappresenta un file che non è dotato di permessi.
Il metodo va a settare all'oggetto ftpFile tutti i parametri necessari e va poi a controllare dalla stringa generata che i permessi siano stati impostati correttamente.

```
@CsvSource({
    "false,?---------",
    "true,?rwxrwxrwx",
})
@ParameterizedTest
void testToFormattedString(Boolean permission, String expected) {
    ftpFile.setName("test.txt");
    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
            ftpFile.setPermission(i, j, permission);
        }
    }
    ftpFile.setHardLinkCount(1);
    ftpFile.setUser("user01");
    ftpFile.setGroup("group01");
    ftpFile.setSize(1024);
    String timestamp = "2021-06-01T10:23:00.000+02:00";
    ZonedDateTime zonedDateTime = ZonedDateTime.parse(timestamp);
    Calendar calendar = Calendar.getInstance();
    calendar.setTimeInMillis(zonedDateTime.toInstant().toEpochMilli());
    ftpFile.setTimestamp(calendar);
    String formattedString = ftpFile.toFormattedString();
    System.out.println(formattedString);
    assert(formattedString.contains(expected));

}
```

Siccome sia il nome del file, che l'utente o il gruppo o ancora la dimensione sono parametri che non hanno funzioni particolari, ma sono semplicemente stringhe, li ho lasciati costanti.

Con questi test la classe ftpFile è stata coperta al 91%.

FontInfo

La classe FontInfo è una classe che rappresenta le informazioni relative ad un font, queste sono famiglia, dimensione e informazioni sullo stile. Consente anche di creare un oggetto java.awt.Font e di essere inizializzato da quest'ultimo.
Il primo metodo rilevante è testConstructor2, che verifica il secondo costruttore, quello che riceve in ingresso un oggetto di tipo Font.
Questi test case sono stati realizzati sempre con la tecnica delle classi di equivalenza assumendo in alcuni casi anche più di un valore valido.
Nel test successivo siccome andiamo a testare il costruttore mediante la classe font assumiamo come precondizione di avere un oggetto della classe Font valido per poter testare la classe fontInfo. Nel caso in cui provassimo a creare un oggetto font non valido andrebbe sicuramente in errore il suo costruttore e comunque staremmo testando un'altra classe. Per questo motivo non avendo senso in questo caso testare i vari possibili valori non validi mi sono concentrato sui possibili valori validi e sull'unico valore non valido ammesso dalla classe FontInfo che è il valore null.
Di seguito la tabella di progettazione dei casi di test.

ID	Precond	Input	Expected Output	Output	Post Cond
1	Oggetto font valido	family = Arial; style = 1; size = 24;			fontInfo.family = "Arial"; fontInfo.size = 24; fontInfo.isItalic = true;
2	Oggetto font valido	family = Arial; style = 2; size = 24;			fontInfo.family = "Arial"; fontInfo.size = 24; fontInfo.isbold = true;
3	Oggetto font valido	family Arial; style = 1; size = 24;			fontInfo.family = "Arial"; fontInfo.size = 24; fontInfo.isItalic = false;
4	Oggetto font valido	family = Arial; style = 1; size = 12;			fontInfo.family = "Arial"; fontInfo.size = 12; fontInfo.isItalic = true;
5	Oggetto font valido	family = Times New Roman; style = 1; size = 24;			fontInfo.family = "Times New Roman"; fontInfo.size = 24; fontInfo.isItalic = true;
6	Oggetto font null	family = null; style = null; size = null;	Thrown Exception Illegal Argument	Thrown Exception Illegal Argument	

Per questo test ho utilizzato costruito 3 classi di equivalenza per i 3 possibili input. La famiglia del font può essere uno dei seguenti valori {Arial, Times New Roman, null}, lo stile può essere {1,2} e la dimensione {12,24}.
Per la realizzazione dei casi di test ho utilizzato la tecnica di copertura delle classi adiacenti.
L'ultimo caso di test in realtà utilizza il costruttore passando come argomento un oggetto null.

```
@CsvSource({
    "Arial, 1, 24",
    "Arial, 2, 24",
    "Arial, 1, 24",
    "Arial, 1, 12",
    "Times New Roman, 1, 24",
    "null, 2, 24",
})
```

Ho realizzato un test parametrizzato che riceve in ingresso 3 parametri, famiglia, stile e la dimensione.

```
@ParameterizedTest
public void testConstructor2(String family, int style, int size) {
    if(family.equals("null")) {
        assertThrows(IllegalArgumentException.class, () -> {
            FontInfo fontInfo = new FontInfo(null);
        });
        return;
    }
    Font font = new Font(family, style, size);
    FontInfo fontInfo = new FontInfo(font);
    assertTrue(fontInfo.getFamily().equals(family));
    assertTrue(fontInfo.getSize() == size);
    if(style == Font.ITALIC){
        assertTrue(fontInfo.isItalic());
    } else if(style == Font.PLAIN) {
        assertFalse(fontInfo.isItalic());
        assertFalse(font.isBold());
    } else {
        assertTrue(fontInfo.isBold());
    }
}
```

Nel test costruisco un oggetto font con i parametri in ingresso e da quello inizializzo l'oggetto fontInfo. Verifico poi uno alla volta che i valori in ingresso siano rimasti invariati nell'oggetto appena creato.

Di seguito la tabella che mostra il report dei test eseguiti

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	Oggetto font valido	family = Arial; style = 1; size = 24;			fontInfo.family = "Arial"; fontInfo.size = 24; fontInfo.isItalic = true;	Success
2	Oggetto font valido	family = Arial; style = 2;size = 24;			fontInfo.family = "Arial"; fontInfo.size = 24; fontInfo.isbold = true;	Success
3	Oggetto font valido	family = Arial; style = 1; size = 24;			fontInfo.family = "Arial"; fontInfo.size = 24; fontInfo.isItalic = false;	Success
4	Oggetto	family = Arial;			fontInfo.family = "Arial";	Success

	font valido	style = 1; size = 12;			fontInfo.size = 12; fontInfo.isItalic = true;	
5	Oggetto font valido	family = Times New Roman; style = 1; size = 24;			fontInfo.family = "Times New Roman"; fontInfo.size = 24; fontInfo.isItalic = true;	Success
6	Oggetto font null	family = null; style = 1; size = 24;	Thrown Exception Illegal Argument	Thrown Exception Illegal Argument		Success

La classe fontInfo può anche creare una stringa riepilogativa con le proprietà del font.
Per il test sono stati progettati i seguenti test case.

ID	Precond	Input	Expected Output	Post Cond
1	Oggetto font valido	family = new String("Arial"); style = new String("BOLD"); size = 24;	Arial, 24, bold	
2	Oggetto font valido	family = new String("Arial"); style = new String("ITALIC"); size = 24;	Arial, 24, italic	
3	Oggetto font valido	family = new String("Arial"); style = new String("PLAIN"); size = 24;	Arial, 24	
4	Oggetto font valido	family = new String("Arial"); style = new String("BOLD"); size = 12;	Arial, 12, bold	
5	Oggetto font valido	family = new String("Times New Roman"); style = new String("BOLD"); size = 24;	Times New Roman, 24, italic	

Anche questi sono stati progettati sulla base delle classi di equivalenza precedentemente descritte con la variante che style può assumere uno dei seguenti valori {BOLD, ITALIC, PLAIN}.
Ho poi costruito I test case con la tecnica delle classi di equivalenza adiacenti per poter trovare più facilmente la causa di un eventuale errore.

```
@CsvSource(value = {
    "Arial; 1; 24; Arial, 24, bold",
    "Arial; 2; 24; Arial, 24, italic",
    "Arial; 0; 24; Arial, 24",
    "Arial; 1; 12; Arial, 12, bold",
    "Times New Roman; 2; 24; Times New Roman, 24, italic",
}, delimiter = ';')
@ParameterizedTest
public void testToString(String family, int style, int size, String expected){
    Font font = new Font(family, style, size);
    FontInfo fontInfo = new FontInfo(font);
    assertEquals(expected, fontInfo.toString());
}
```

Questo test verifica la correttezza del metodo toString, il metodo che restituisce una stringa che indica le prorpietà dell'oggetto fontInfo.

Di seguito la tabella riepilogativa dei test case descritti in precedenza.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	Oggetto font valido	family = Arial; style = 1; size = 24;	Arial, 24, bold	Arial, 24, bold		Success
2	Oggetto font valido	family = Arial; style = 2; size = 24;	Arial, 24, italic	Arial, 24, italic		Success
3	Oggetto font valido	family = Arial; style = 0; size = 24;	Arial, 24	Arial, 24		Success
4	Oggetto font valido	family = Arial; style = 1; size = 12;	Arial, 12, bold	Arial, 12, bold		Success
5	Oggetto font valido	family = Times New Roman; style = new 1 size = 24;	Times New Roman, 24, italic	Times New Roman, 24, italic		Success

```
@CsvSource( value = {
    "Arial; 1; 24; Arial; 2; 24; false",
    "Arial; 2; 24; Arial; 2; 24; true",
    "Arial; 0; 24; Times New Roman; 0; 24; false",
    "Arial; 1; 12; Arial; 1; 24; false",
    "Times New Roman; 2; 24; Times NEW Roman; 2; 24; true",
}, delimiter = ';')
```

Il test del metodo isEqual riceve in ingresso i parametri per costruire due font, ed un parametro che rappresenta il valore atteso dell'uguaglianza tra questi due.
Anche in questo caso per una maggiore facilità nell'individuare la causa di un eventuale fallimento ho costruito i test case utilizzando la tecnica delle classi adiacenti che prevede che i test case vengano costruiti variando solo un parametro alla volta in modo da trovare con certezza e più facilmente la causa del fallimento.
Con questi test ho raggiunto una copertura della classe maggiore dell'80%.

ByteVector

La classe bytevector è una classe che rappresenta un vettore di byte, è dotato di due attributi, uno dato dalla lunghezza e l'altro da un array di byte.
I primi due test verificano rispettivamente la correttezza del costruttore senza argomenti e del costruttore che riceve come argomento il valore intero size e che alloca un array di byte di quella dimensione.
Il metodo successivo della classe di test è il metodo testPutByte11 che testa l'omonima funzione.
Il metodo di test è un test parametrizzato che riceve in ingresso 3 valori, la dimensione iniziale dell'array data della classe, il primo byte e il secondo byte.
Per il testing di questo metodo ho realizzato dei test case con valori comuni di initial size multipli di 10 e due valori particolari, 0 e 1. Come valori da memorizzare e controllare che effettivamente

siano stati memorizzati ho scelto valori casuali positivi e negativi compresi tra -128 e 127. Questo limite è dovuto dal troncamento dell'intero a byte.

La tabella di progettazione dei casi di test è la seguente

ID	Precond	Input	Expected Output	Post Cond
1		initialSize = 10; b1 = -32; b2 = 48	b1 = -32; b2 = 48	bv.data.length >= initialSize
2		initialSize = 100; b1 = --17; b2 = 84	b1 = -17; b2 = 84	bv.data.length >= initialSize
3		initialSize = 1000; b1 = 19; b2 = 56	b1 = 19; b2 = 56	bv.data.length >= initialSize
4		initialSize = 10000; b1 = -92; b2 = 71	b1 = -92; b2 = 71	bv.data.length >= initialSize
5		initialSize = 100000; b1 = 5; b2 = -63	b1 = 5; b2 = -63	bv.data.length >= initialSize
6		initialSize = 0; b1 = 77; b2 = -19	b1 = 77; b2 = -19	bv.data.length >= initialSize
7		initialSize = 1; b1 = -94; b2 = 62	b1 = -94; b2 = 62	bv.data.length >= initialSize

```
@CsvSource({
        "10,-32,48",
        "100,-17,84",
        "1000,19,56",
        "10000,-92,71",
        "100000,5,-63",
        "0,77,-19",
        "1,-94,62"
})
@ParameterizedTest
public void testPutByte11(int initialSize, int b1, int b2) {
    ByteVector bv = new ByteVector(initialSize);
    bv.put11(b1, b2);
    assertEquals(2, bv.length);
    assertTrue(bv.data.length >= initialSize);
    assert (bv.data[0] == (byte) b1);
    assert (bv.data[1] == (byte) b2);
}
```

Di seguito un report dei test case effettuati

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1		initialSize = 10; b1 = -32; b2 = 48	b1 = -32; b2 = 48	b1 = -32; b2 = 48	bv.data.length >= initialSize	Success
2		initialSize = 100; b1 = --17; b2 = 84	b1 = --17; b2 = 84	b1 = -17; b2 = 84	bv.data.length >= initialSize	Success

3		initialSize = 1000; b1 = 19; b2 = 56	b1 = 19; b2 = 56	b1 = 19; b2 = 56	bv.data.length >= initialSize	Success
4		initialSize = 10000; b1 = -92; b2 = 71	b1 = -92; b2 = 71	b1 = -92; b2 = 71	bv.data.length >= initialSize	Success
5		initialSize = 100000; b1 = 5; b2 = -63	b1 = 5; b2 = -63	b1 = 5; b2 = -63	bv.data.length >= initialSize	Success
6		initialSize = 0; b1 = 77; b2 = -19	b1 = 77; b2 = -19	b1 = 77; b2 = -19	bv.data.length >= initialSize	Success
7		initialSize = 1; b1 = -94; b2 = 62	b1 = -94; b2 = 62	b1 = -94; b2 = 62	bv.data.length >= initialSize	Success

Il testPutShort è simile al precedente, riceve in ingresso la size iniziale e un numero intero. Ho scelto gli stessi valori di prima come initial size mentre come valore da memorizzare come short ho scelto dei valori casuali positivi aggiungendo però due casi limite, 125 e 126 che sono vicini all'ultimo numero rappresentabile con un byte senza andare in overflow.

Di seguito una tabella rappresentativa dei test case utilizzati per l'esecuzione del test.

ID	Precond	Input	Expected Output	Post Cond
1		initialSize = 10; s = 125	s = 125	bv.data.length >= initialSize
2		initialSize = 100; s = 126	s = 126	bv.data.length >= initialSize
3		initialSize = 1000; s = 19	s = 19	bv.data.length >= initialSize
4		initialSize = 10000; s = 71	s = 71	bv.data.length >= initialSize
5		initialSize = 100000; s = 0	s = 0	bv.data.length >= initialSize
6		initialSize = 0; s = 77	s = 77	bv.data.length >= initialSize
7		initialSize = 1; s = 94	s = 94	bv.data.length >= initialSize

```
@CsvSource({
        "10,125",
        "100,126",
        "1000,19",
        "10000,71",
        "100000,0",
        "0,77",
        "1,94"
})
@ParameterizedTest
public void testPuthShort(int initialSize, int s) {
    ByteVector bv = new ByteVector(initialSize);
    bv.putShort(s);
    assertEquals(2, bv.length);
    assertTrue(bv.data.length >= initialSize);
    assert((byte) (bv.data[0] >>> 8) == (byte) (s >>> 8));
    assert((byte) (bv.data[1]) == (byte) (s));
}
```

Il test verifica che il metodo putShort abbia memorizzato correttamente il numero intero come short, in particolare che sia stato convertito il numero intero in due byte e che i due byte in posizione 0, con lo shift di 8 bit, e in posizione 1 siano uguali con i byte memorizzati nelle rispettive posizioni dell'array data.

Di seguito una tabella riepilogativa dei test case eseguiti.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1		initialSize = 10; s = 125	s = 125	s = 125	bv.data.length >= initialSize	Success
2		initialSize = 100; s = 126	s = 126	s = 126	bv.data.length >= initialSize	Success
3		initialSize = 1000; s = 19	s = 19	s = 19	bv.data.length >= initialSize	Success
4		initialSize = 10000; s = 71	s = 71	s = 71	bv.data.length >= initialSize	Success
5		initialSize = 100000; s = 0	s = 0	s = 0	bv.data.length >= initialSize	Success
6		initialSize = 0; s = 77	s = 77	s = 77	bv.data.length >= initialSize	Success
7		initialSize = 1; s = 94	s = 94	s = 94	bv.data.length >= initialSize	Success

Il metodo **testPutInt** verifica la correttezza del metodo putInt della classe ByteVector.
In particolare, il test crea un oggetto di tipo ByteVector con una dimensione iniziale. Viene poi invocato il metodo putInt.

Il test verifica che il valore di length sia pari a 4 e che l'array data sia rimasto invariato o allargato dopo l'inserimento. Vengono poi controllati i byte presenti nelle 4 posizioni dell'array data con i bit a 8 a 8 dell'intero in ingresso al test.
Come valori del test sono stati scelti valori casuali uniti a valori limite, il massimo numero positivo rappresentabile con il tipo intero, il minimo numero negativo rappresentabile con il tipo intero e lo 0.

Ho realizzato questo test implementando i seguenti testcase

ID	Precond	Input	Expected Output	Post Cond
1		initialSize = 10; i = 125	s = 125	bv.data.length >= initialSize
2		initialSize = 100; i = 126	s = 126	bv.data.length >= initialSize
3		initialSize = 1000; i = 19	s = 19	bv.data.length >= initialSize
4		initialSize = 10000; i = 71	s = 71	bv.data.length >= initialSize
5		initialSize = 100000; i = 0	s = 0	bv.data.length >= initialSize
6		initialSize = 0; i = 77	s = 77	bv.data.length >= initialSize
7		initialSize = 1; i = 94	s = 94	bv.data.length >= initialSize
8		initialSize = 10; i = 2147483646	s = 2147483646	bv.data.length >= initialSize
9		initialSize = 4; i = -2147483647	s = -2147483647	bv.data.length >= initialSize

```java
@CsvSource({
        "10,125",
        "100,126",
        "1000,19",
        "10000,71",
        "100000,0",
        "0,77",
        "1,94",
        "10,2147483646",
        "4,-2147483647"
})
@ParameterizedTest
public void testPutInt(int initialSize, int i) {
    ByteVector bv = new ByteVector(initialSize);
    bv.putInt(i);
    assertEquals(4, bv.length);
    assertTrue(bv.data.length >= initialSize);
    assert((byte) (bv.data[0]) == (byte) (i >>> 24));
    assert((byte) (bv.data[1]) == (byte) (i >>> 16));
    assert((byte) (bv.data[2]) == (byte) (i >>> 8));
    assert((byte) (bv.data[3]) == (byte) (i));
}
```

Di seguito un riepilogo dei test effettuati

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1		initialSize = 10; i = 125	s = 125	s = 125	bv.data.length >= initialSize	Success
2		initialSize = 100; i = 126	s = 126	s = 126	bv.data.length >= initialSize	Success
3		initialSize = 1000; i = 19	s = 19	s = 19	bv.data.length >= initialSize	Success
4		initialSize = 10000; i = 71	s = 71	s = 71	bv.data.length >= initialSize	Success
5		initialSize = 100000; i = 0	s = 0	s = 0	bv.data.length >= initialSize	Success
6		initialSize = 0; i = 77	s = 77	s = 77	bv.data.length >= initialSize	Success
7		initialSize = 1; i = 94	s = 94	s = 94	bv.data.length >= initialSize	Success
8		initialSize = 10; i = 2147483646	s = 2147483646	s = 2147483646	bv.data.length >= initialSize	Success
9		initialSize = 4; i = -2147483647	s = -2147483647	s = -2147483647	bv.data.length >= initialSize	Success

Il metodo **putLong** è analogo, l'unica differenza è che confronta 8 byte invece di 4.

Il metodo **testPutUTF8** verifica la correttezza del metodo putUTF8. I caratteri utf non hanno dimensione fissa, possono essere memorizzati su uno o più byte. Il metodo, dopo aver inizializzato

un oggetto bytevector e chiamato la funzione putUtf8 crea un'array di bytes a partire dal 3 elemento dell'array bv.data. Bytevector memorizza nei primi 2 byte le informazioni sulla lunghezza dell'array. L'array appena creato viene convertito in stringa utilizzando la codifica utf 8, così come la stringa in ingresso al test e le due vengono confrontate.
Per questi test case ho generato casualmente delle stringhe utf di lunghezza variabile ed ho utilizzato come oracolo la classe String e la codifica UTF 8 nativa di Java.

ID	Precond	Input	Expected Output	Post Cond
1		initialSize = 10 s = "Hello"	String(bv.data[2...]) = "Hello"	
2		initialSize = 100 s = "Café"	String(bv.data[2...]) = "Café"	
3		initialSize = 1000 s = "こんにちは"	String(bv.data[2...]) = "こんにちは"	
4		initialSize = 10000 s = "Résumé"	String(bv.data[2...]) = "Résumé"	
5		initialSize = 100000 s = "Γειά σου"	String(bv.data[2...]) = "Γειά σου"	
6		initialSize = 0 s = "你好"	String(bv.data[2...]) = "你好"	
7		initialSize = 1 s = "Привет"	String(bv.data[2...]) = "Привет"	
8		initialSize = 10 s = "السلام عليكم"	String(bv.data[2...]) = "السلام عليكم"	
9		initialSize = 100 s = "प्रणाम"	String(bv.data[2...]) = "प्रणाम"	
10		initialSize = 1000 s = "1234567890"	String(bv.data[2...]) = "1234567890"	

In queste tabelle di descrizione dei casi di test realizzati ho imposto come risultato atteso che la stringa ottenuta dall'array di bytes dell'oggetto bv della classe ByteVector sia uguale alla stringa in ingresso però confrontando solo i valori dal 2 elemento dell'array in poi perché nei primi due elementi la classe salva informazioni aggiuntive e non direttamente i dati ottenuti dalla decodifica della stringa da utf a byte.

```
@CsvSource({
    "10,\"Hello\"",
    "100,\"Caf\u00E9\"",
    "1000,\"\u3053\u3093\u306B\u3061\u306F\"",
    "10000,\"R\u00E9sum\u00E9\"",
    "100000,\"\u0393\u03B5\u03B9\u03AC \u03C3\u03BF\u03C5\"",
    "0,\"\u4F60\u597D\"",
    "1,\"\u041F\u0440\u0438\u0432\u0435\u0442\"",
    "10,\"\u0627\u0644\u0633\u0644\u0627\u0645 \u0639\u0644\u064A\u0643\u0645\"",
    "100,\"\u092A\u094D\u0930\u0923\u093E\u092E\"",
    "1000,\"1234567890\""
})
```

```java
@ParameterizedTest
    public void testPutUTF8(int initialSize, String s) {
        ByteVector bv = new ByteVector(initialSize);
        bv.putUTF8(s);
        assertTrue(bv.data.length >= initialSize);
        int byteLength = ((bv.data[0] & 0xFF) << 8) | (bv.data[1] & 0xFF);
        byte[] bytes = new byte[byteLength];
        for (int i = 0; i < byteLength; i++) {
            bytes[i] = bv.data[i + 2];
        }
        String decoded = new String(bytes, StandardCharsets.UTF_8);
        String expected = new String(s.getBytes(StandardCharsets.UTF_8), StandardCharsets.UTF_8);
        assertEquals(expected, decoded);
    }
```

Di seguito il report ottenuto dall'esecuzione dei casi di test.

ID	Pre	Input	Expected Output	Output	PC	Result
1		initialSize = 10 s = "Hello"	String(bv.data[2...]) = "Hello"	String(bv.data[2...]) = "Hello"		Success
2		initialSize = 100 s = "Café"	String(bv.data[2...]) = "Café"	String(bv.data[2...]) = "Café"		Success
3		initialSize = 1000 s = "こんにちは"	String(bv.data[2...]) = "こんにちは"	String(bv.data[2...]) = "こんにちは"		Success
4		initialSize = 10000 s = "Résumé"	String(bv.data[2...]) = "Résumé"	String(bv.data[2...]) = "Résumé"		Success
5		initialSize = 100000 s = "Γειά σου"	String(bv.data[2...]) = "Γειά σου"	String(bv.data[2...]) = "Γειά σου"		Success
6		initialSize = 0 s = "你好"	String(bv.data[2...]) = "你好"	String(bv.data[2...]) = "你好"		Success
7		initialSize = 1 s = "Привет"	String(bv.data[2...]) = "Привет"	String(bv.data[2...]) = "Привет"		Success
8		initialSize = 10 s = "السلام عليكم"	String(bv.data[2...]) = "السلام عليكم"	String(bv.data[2...]) = "السلام عليكم"		Success
9		initialSize = 100 s = "प्रणाम"	String(bv.data[2...]) = "प्रणाम"	String(bv.data[2...]) = "प्रणाम"		Success
10		initialSize = 1000 s = "1234567890"	String(bv.data[2...]) = "1234567890"	String(bv.data[2...]) = "1234567890"		Success

I due metodi testPutByte e testPutByteArray sono simili a quest'ultimo. In particolare, il secondo riceve in ingresso la dimensione iniziale del bytevector, una stringa che sarà l'array di byte, un offset e una lunghezza. I valori di offset e lunghezza servono ad indicare da che elemento copiare e quanti elementi copiare a partire da quest'ultimo.

```
@CsvSource({
        "10,\"Hello\",3,3",
        "100,\"Caf\u00E9\",0,5",
        "1000,\"\u3053\u3093\u306B\u3061\u306F\",2,4",
        "10000,\"R\u00E9sum\u00E9\",3,1",
        "100000,\"\u0393\u03B5\u03B9\u03AC \u03C3\u03BF\u03C5\",3,7",
        "0,\"\u4F60\u597D\",2,3",
        "1,\"\u041F\u0440\u0438\u0432\u0435\u0442\",2,7",
        "10,\"\u0627\u0644\u0633\u0644\u0627\u0645 \u0639\u0644\u064A\u0643\u0645\",1,4",
        "100,\"\u092A\u094D\u0930\u0923\u093E\u092E\",1,4",
        "1000,\"1234567890\",1,2"
})
@ParameterizedTest
public void testPutByteArray(int initialSize, String s, int offset, int length) {
    byte[] b = s.getBytes(StandardCharsets.UTF_8);
    byte[] expected = new byte[initialSize > length ? initialSize : length];
    ByteVector bv = new ByteVector(initialSize);
    bv.putByteArray(b, offset, length);
    assertTrue(bv.data.length >= initialSize);
    System.arraycopy(b, offset, expected, 0, length);
    assertArrayEquals(expected, bv.data);
}
```

Per questo test come valori in ingresso ho usato le stesse stringhe del test precedente come array di byte, mentre come valori di offset ho utilizzato valori compresi tra 0 e 3 e per la lunghezza ho utilizzato valori casuali compresi tra 2 e 7.

Range

La classe range è una classe che implementa un range offrendo dei metodi per verificare che un oggetto sia presente o meno in un range, che sia più grande o più piccolo ed altri.
Per testare questa classe, invece di generare dei test case utilizzando le classi di equivalenza ho preferito utilizzare una tecnica esplorativa e puntare alla copertura dei valori validi possibili. Ho generato dei test case in modo da coprire i diversi casi possibili per ogni metodo cercando di coprire tutto il codice con un minor numero possibile di test.
Essendo questa classe parametrica e quindi utilizzabile con qualunque tipo scalare od oggetto che sia dotato di un comparatore ho pensato che sarebbe stato inutile costruire le classi di equivalenza considerando input validi o no perché sarebbe ricaduto nel test dell'oggetto specifico e non dell'oggetto range. Ho deciso quindi di costruire i casi di test ad hoc in modo da rappresentare tutte le casistiche possibili metodo per metodo basandomi sul range e sull'elemento in ingresso.

Il primo metodo di cui si vuole verificare la correttezza è il metodo **testInRange**.

In questo test sono stati usati 8 casi di test che rappresentano diversi scenari possibili. Questi sono in ordine:
1) Il valore si trova nel range.

2) Il valore è maggiore del massimo del range.
3) Il valore è minore del minimo del range con minimo e massimo invertiti.
4) Il valore è uguale al minimo e al massimo del range.
5) Il valore è minore del minimo del range.
6) Il valore è maggiore del massimo del range con minimo e massimo invertiti.
7) Il valore è uguale al minimo del range.
8) Il valore è uguale al massimo del range.

Di seguito la rappresentazione dei vari test case in formato tabellare

ID	Precond	Input	Expected Output	Post Cond
1		min = -90; max = 563; value = 200;	true	
2		min = -100; max = 100; value = 150;	false	
3		min = 200; max = 100; value = -150;	false	
4		min = 0; max = 0; value = 0;	true	
5		min = -50; max = -60; value = -70;	false	
6		min = 200; max = 100; value = 300;	false	
7		min = -100; max = 100; value = -100;	true	
8		min = -100; max = 100; value = 100;	true	

```
@CsvSource({
        "-90, 563, 200, true",
        "-100, 100, 150, false",
        "200, -100, -150, false",
        "0, 0, 0, true",
        "-50, -60, -70, false",
        "200, 100, 300, false",
        "-100, 100, -100, true",
        "-100, 100, 100, true",
})

@ParameterizedTest
public void testIntRange(int min, int max, int value, boolean expected) {
    Range<Integer> range = Range.between(min, max);
    assertEquals(expected, range.contains(value));
}
```

Di seguito rappresento in tabella i vari test case

ID	min	max	value	expected
1	-90	563	200	true
2	-100	100	150	false
3	200	-100	-150	false
4	0	0	0	true

5	-50	-60	-70	false
6	200	100	300	false
7	-100	100	-100	true
8	-100	100	100	true

Il report dei test case effettuati, in forma tabellare, è il seguente.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1		min = -90; max = 563; value = 200;	true	true		Success
2		min = -100; max = 100; value = 150;	false	false		Success
3		min = 200; max = 100; value = -150;	false	false		Success
4		min = 0; max = 0; value = 0;	true	true		Success
5		min = -50; max = -60; value = -70;	false	false		Success
6		min = 200; max = 100; value = 300;	false	false		Success
7		min = -100; max = 100; value = -100;	true	true		Success
8		min = -100; max = 100; value = 100;	true	true		Success

Nei test successivi utilizzerò come valori in ingresso delle stringhe e l'ordine sarà l'ordine alfabetico.

Il seguente test verifica la correttezza del metodo **between** che si occupa della creazione di un range.
Il metodo between testato di seguito è quello con un comparatore diverso dal comparatore di default.
In questo caso ho costruito dei range di stringa ed ho utilizzato il comparatore che confronta le stringhe in ordine alfabetico.
Ho realizzato un test parametrico che prende in input i valori di minimo e massimo per la creazione di un range.
Il parametro 'value' è il valore che si vuole verificare se è contenuto nel range.
Il parametro 'expected' è il valore atteso che deve essere uguale al risultato del test.
Per verificare se un elemento si trova nel range viene utilizzato poi il metodo compare.

Per questo test sono stati scritti 5 test case che rappresentano diversi scenari possibili. Questi sono in ordine:
1) Il valore si trova nel range.
2) Il valore si trova prima del range.
3) Il valore è uguale al minimo del range.
4) Il valore è uguale al massimo del range.
5) Il valore si trova dopo il range.

Di seguito le classi di equivalenza scritte in forma tabellare

ID	min	max	value	expected
1	apple	zimbawe	banana	true
2	orange	zimbawe	banana	false
3	apple	orange	apple	true
4	orange	zimbawe	zimbawe	true
5	Apple	banana	orange	false

I test case realizzati, con il report realizzato dopo l'esecuzione sono

ID	Precond	Input	Expected Output	Output	Post Cond	Result

1		min = new String("apple");
		max = new String("zimbawe");
		value = new String("banana");
			true	true		Success

2		min = new String("orange");
		max = new String("zimbawe");
		value = new String("banana");
			false	false		Success

3		min = new String("apple");
		max = new String("orange");
		value = new String("apple");
			true	true		Success

4		min = new String("orange");
		max = new String("zimbawe");
		value = new String("zimbawe");
			true	true		Success

5		min = new String("Apple");
		max = new String("banana");
		value = new String("orange");
			false	false		Success

```
@CsvSource({
        "apple,zimbawe,banana, true",
        "orange,zimbawe,banana, false",
        "apple,orange,apple, true",
        "orange,zimbawe,zimbawe, true",
        "Apple,banana,orange, false",
})
@ParameterizedTest
public void testStringRange(String min, String max, String value, boolean expected) {
    Comparator<String> comparator = Comparator.comparing(String::toString, Comparator.naturalOrder());
    Range<String> range = Range.between(min, max, comparator);
    assertEquals(expected, range.contains(value));
}
```

Il prossimo test, simile al precedente, è il test che verifica la correttezza del metodo **isAfter**.
Viene creato un range di stringhe con comparatore il comparatore naturale (ordine alfabetico).
Il parametro 'value' è il valore che si vuole verificare se è contenuto nel range.
Il parametro 'expected' è il valore atteso che deve essere uguale al risultato del test.
Il metodo isAfter verifica se il range è dopo il valore passato come parametro.

Anche per questo test ho realizzato delle classi di equivalenza che coprissero tutte le casistiche possibili.

1) Range ordinato a destra dell'elemento
2) Range ordinato con elemento all'interno
3) Range ordinato con elemento uguale al minore
4) Range al contrario a sinistra dell'elemento
5) Range al contrario a destra dell'elemento
6) Range ordinato con elemento uguale all'estremo maggiore
7) Range ordinato con elemento nullo

Di seguito una rappresentazione in tabella dei test case.

ID	min	max	value	expected
1	banana	zimbawe	apple	true
2	apple	banana	zimbawe	false
3	apple	orange	apple	false
4	orange	apple	zimbawe	false
5	orange	banana	apple	true
6	apple	orange	orange	false
7	orange	zimbawe	null	false

Per rappresentare i test ho realizzato una tabella che scompone gli input del metodo di test nelle stringhe che rappresentano gli estremi del range e della stringa per la quale si vuole verificare la posizione rispetto al range.

```
@CsvSource({
        "banana,zimbawe,apple, true",
        "apple,banana,zimbawe, false",
        "apple,orange,apple, false",
        "orange,apple,zimbawe, false",
        "orange,banana,apple,true",
        "apple,orange,orange,false",
        "orange,zimbawe,null, false",
  })

@ParameterizedTest
  public void testIsAfter(String min, String max, String value, boolean expected) {
    Comparator<String> comparator = Comparator.comparing(String::toString, Comparator.naturalOrder());
    Range<String> range = Range.between(min, max, comparator);
    if (value.equalsIgnoreCase("null")) {
      value = null;
    }
    assert(range.isAfter(value) == expected);
  }
```

Di seguito il report dei test case rappresentato in forma tabellare.

ID	Precond	Input	Expected Output	Output	Post Cond	Result

1		min = new String("banana");
		max = new String("zimbawe");
		value = new String("apple");
			true	true		Success

2		min = new String("apple");
		max = new String("banana");
		value = new String("zimbawe");
			false	false		Success

3		min = new String("apple");
		max = new String("orange");
		value = new String("apple");
			false	false		Success

4		min = new String("orange");
		max = new String("apple");
		value = new String("zimbawe");
			false	false		Success

5		min = new String("orange");
		max = new String("banana");
		value = new String("apple");
			true	true		Success

6		min = new String("apple");
		max = new String("orange");
		value = new String("orange");
			false	false		Success

7		min = new String("orange");
		max = new String("zimbawe");
		value = null;
			false	false		Success

I metodi **testStartedBy** e **testIsEndedBy** sono molto simili, verificano la correttezza dei rispettivi metodi della classe Range che controllano se l'elemento in ingresso è uguale all'estremo minore o maggiore.
Per testare i metodi ho scelto come valori in ingresso sia dei range ordinati con valori da cercare sia validi che non validi, ma anche range indicati al contrario che sono stati convertiti correttamente; infatti, dei due elementi che passiamo la classe Range determina automaticamente il maggiore il minore.

```
@CsvSource({
        "banana,zimbawe,zimbawe,0",
        "apple,banana,zimbawe,1",
        "apple,orange,banana,0",
        "orange,banana,zimbawe,1",
        "orange,zimbawe,apple,-1",
        "orange,zimbawe,null,-2",
})
```

Il test indicato di seguito si occupa di verificare il metodo compare to che verifica se un elemento è maggiore del massimo, minore del minimo elemento di un range o se è contenuto e prende in ingresso gli estremi di un range e l'elemento da confrontare ed il valore atteso. Il valore atteso può essere -1, 0 oppure 1.

ID	min	max	value	expected
1	banana	zimbawe	zimbawe	0
2	apple	banana	zimbawe	1
3	apple	orange	banana	0
4	orange	banana	zimbawe	1
5	orange	zimbawe	apple	-1
6	orange	zimbawe	null	-2

Di seguito viene indicata la tabella relativa ai test eseguiti.

ID	Precond	Input	Expected Output	Output	Post Cond	Result

1		min = new String("banana");
		max = new String("zimbawe");
		value = new String("zimbawe");
			0	0		Success

2		min = new String("apple");
		max = new String("banana");
		value = new String("zimbawe");
			1	1		Success

3		min = new String("apple");
		max = new String("orange");
		value = new String("banana");
			0	0		Success

4		min = new String("orange");
		max = new String("banana");
		value = new String("zimbawe");
			1	1		Success

5		min = new String("orange");
		max = new String("zimbawe");
		value = new String("apple");
			-1	-1		Success

6		min = new String("apple");
		max = new String("orange");
		value = null;
			Errore	Errore	Exception Thrown	Success

```java
@ParameterizedTest
  public void testCompareTo(String min, String max, String value, int expected) {
    Comparator<String> comparator = Comparator.comparing(String::toString, Comparator.naturalOrder());
    Range<String> range = Range.between(min, max, comparator);
    if (value.equalsIgnoreCase("null")) {
      final String nullValue = null;
      assertThrows(NullPointerException.class, () -> range.elementCompareTo(nullValue));
    } else {
      assert(range.elementCompareTo(value) == expected);
    }
  }
```

Il metodo **containsRange**, **isBeforeRange** e **isAfterRange** sono equivalenti come concetto ai metodi analoghi descritti in precedenza, con la differenza però che in ingresso ricevono dei range invece che degli elementi. Il test viene quindi effettuato in maniera analoga solo con dei range,

questi sono stati scelti in modo da rispecchiare la situazione vista nel test dei metodi precedentemente descritti.

Il metodo **testIsOverLappedBy** è un metodo che prende in ingresso due range ed un valore atteso. Verifica se tra il primo e il secondo range in ingresso ci sono intersezioni. Di seguito viene mostrata la tabella che riepiloga i test case eseguiti.

Per questo test ho realizzato dei test case che potessero realizzare le seguenti situazioni

1) Intersezione parziale con il minimo del secondo range minore del minimo del primo
2) Intersezione completa
3) Range identici ma minimo e massimo inseriti al contrario
4) Intersezione parziale con un estremo in comune
5) Nessuna intersezione con primo range a destra rispetto al secondo
6) Nessuna intersezione con primo range a sinistra rispetto al secondo
7) Intersezione parziale con il massimo del primo range minore del massimo del secondo
8) Secondo range nullo

ID	Precond	Input	Expected Output	Output	Post Cond	Result

1		min = new String("banana");
		max = new String("zimbawe");
		min2 = new String("apple");
		max2 = new String("orange");
			true	true		Success

2		min = new String("apple");
		max = new String("zimbawe");
		min2 = new String("banana");
		max2 = new String("orange");
			true	true		Success

3		min = new String("apple");
		max = new String("orange");
		min2 = new String("orange");
		max2 = new String("apple");
			true	true		Success

4		min = new String("orange");
		max = new String("banana");
		min2 = new String("banana");
		max2 = new String("apple");
			true	true		Success

5		min = new String("cucumber");
		max = new String("orange");
		min2 = new String("apple");
		max2 = new String("banana");
			false	false		Success

6		min = new String("broccoli");
		max = new String("cucumber");
		min2 = new String("apple");
		max2 = new String("banana");
			false	false		Success

7		min = new String("apple");
		max = new String("broccoli");
		min2 = new String("banana");
		max2 = new String("cucumber");
			true	true		Success

8		min = new String("apple");
		max = new String("cucumber");
		min2 = null;
		max2 = null;
			false	false	Exception Thrown	Success

```
@CsvSource({
        "banana,zimbawe,apple,orange,true",
        "apple,zimbawe,banana,orange,true",
        "apple,orange,orange,apple,true",
        "orange,banana,banana,apple,true",
        "cucumber,orange,apple,banana,false",
        "broccoli,cucumber,apple,banana,false",
        "apple,broccoli,banana,cucumber,true",
        "apple,cucumber,null,null,false"
  })
```

ID	min	max	min2	max2	expected
1	banana	zimbawe	apple	orange	true
2	apple	zimbawe	banana	orange	true
3	apple	orange	orange	apple	true
4	orange	banana	banana	apple	true
5	cucumber	orange	apple	banana	false
6	broccoli	cucumber	apple	banana	false
7	apple	broccoli	banana	cucumber	true
8	apple	cucumber	null	null	false

```
@ParameterizedTest
  public void testIsOverLappedBy(String minRange1, String maxRange1, String minRange2, String maxRange2,
        boolean expected) {
    Comparator<String> comparator = Comparator.comparing(String::toString, Comparator.naturalOrder());
    Range<String> range1 = Range.between(minRange1, maxRange1, comparator);
    Range<String> range2 = Range.between(minRange2, maxRange2, comparator);
    if (minRange2.equalsIgnoreCase("null") && maxRange2.equalsIgnoreCase("null")) {
      range2 = null;
    }

    assertEquals(expected, range1.isOverlappedBy(range2));
  }
```

Questo test verifica la correttezza del metodo **intersectionWith** della classe Range.
Il metodo è simile al precedente ma invece di restituire un valore booleano restituisce gli estremi del range dato dall'intersezione.

```
@CsvSource({
        "apple,orange,banana,broccoli,banana,broccoli",
        "apple,orange,banana,zimbawe,banana,orange",
        "banana,orange,apple,broccoli,banana,broccoli",
        "apple,banana,broccoli,orange,null,null",
        "apple,broccoli,apple,broccoli,apple,broccoli",
  })
  @ParameterizedTest
  public void testIntersectionWith(String minRange1, String maxRange1, String minRange2, String maxRange2,
        String minExpRange, String maxExpRange) {
    Comparator<String> comparator = Comparator.comparing(String::toString, Comparator.naturalOrder());
    Range<String> range1 = Range.between(minRange1, maxRange1, comparator);
    Range<String> range2 = Range.between(minRange2, maxRange2, comparator);
    if (minExpRange.equalsIgnoreCase("null") && maxExpRange.equalsIgnoreCase("null")) {
      assertThrows(IllegalArgumentException.class, () -> range1.intersectionWith(range2));
      return;
    }
    Range<String> expRange = Range.between(minExpRange, maxExpRange, comparator);
    assertEquals(expRange, range1.intersectionWith(range2));
  }
```

Il test **testHashCode** è un test che verifica la validità del metodo hashCode della classe Range. L'hashcode è calcolato in base gli attributi dell'oggetto, se due oggetti sono uguali hanno hashcode uguali, altrimenti l'hashcode sarà diverso.
Per verificare la correttezza del metodo quindo ho scritto il test seguente

```
  @CsvSource({
    "10,100, true",
    "10,10, false"
  })
  @ParameterizedTest
  public void testHashCode(int min, int max, boolean expected) {
    Range<Integer> range = Range.between(min, max);
    Range<Integer> range2 = Range.between(10, 100);
    assertEquals(expected, range.hashCode() == range2.hashCode());
  }
```

Che riceve in ingresso i parametri per creare un range e confronta l'hash con un range fisso che fa da riferimento.
I parametri in ingresso consentono di creare un range uguale al riferimento e uno diverso per coprire entrambi i possibili casi.

Con questi test ho coperto il 90% del codice della classe Range.

Rational Number

La classe Rational Numner è una classe che realizza l'astrazione di un numero razionale. Prende in ingresso due numeri interi, dividendo e divisore e restituisce il numero decimale corrispondente. Effettua anche l'operazione inversa andando a calcolare la frazione più vicina ad un numero razionale utilizzando la tecnica delle approssimazioni successive.
Per la realizzazione dei test di questa classe in quasi tutti i test ho utilizzato la tecnica delle classi di equivalenza. Come classi di equivalenza ho utilizzato le stesse sia per il numeratore che per il denominatore, queste sono

CE1	CE2	CE3	CE4	CE5
-1	0	1	-500	989

Ho poi costruito il modello su ctwedge

```
1. Model RationalNumber
2.  Parameters:
3.    num : {-1,0,1,500,-989}
4.    den : {-1,0,1,500,-989}
5.
```

E ho generato i test case con la tecnica di copertura 1-wise.
Il test **testFactoryMethod** verifica la correttezza del metodo **factoryMethod** e confronta il risultato utilizzando il valore doubleValue della classe RationalNumber.
Questo metodo inizializza un numero razionale mediante i valori di numeratore e denominatore.
Per verificare la correttezza utilizzo il metodo doubleValue della classe rational number e lo confronto con il valore atteso.

Di seguito una tabella riepilogativa dei test eseguiti.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	Rational number esiste	num = -1; den = 0;	Infinity	Infinity	No Exception Thown	Success
2	Rational number esiste	num = 0; den = 1;	0	0	No Exception Thown	Success
3	Rational number esiste	num = 1; den = 500;	-0.002	-0.002	No Exception Thown	Success
4	Rational number esiste	num = 500; den = 989;	0.5055611729019212	0.5055611729019212	No Exception Thown	Success
5	Rational number esiste	num = -989; den = 1;	989.0	989.0	No Exception Thown	Success

```
    @CsvSource({
        "-1,0,Infinity",
        "0,1,0.0",
        "1,-500,-0.002",
        "500,989,0.5055611729019212",
        "-989,-1,989.0",
    })
    @ParameterizedTest
    public void testFactoryMethod(long numerator, long divisor, double expected) {
        RationalNumber rational = RationalNumber.factoryMethod(numerator, divisor);
        double actual = rational.doubleValue();
        assertEquals(expected, actual, 1E-8);

    }
```

Altri test molto simili, che si differenziano solo per il tipo del valore atteso, sono testNegate, testDoubleValue e testFloatValue.

Il metodo testIntValue invece è simile ma il valore atteso viene calcolato mediante il troncamento del numero razionale ad intero. Per questo test ho utilizzato gli stessi test case realizzati in precedenza ma ho eliminato il primo perché il tipo int in Java non supporta, a differenza del double, la rappresentazione Infinity.

```
@CsvSource({
        "0,1,0",
        "1,-500,0",
        "500,989,0",
        "-989,-1,989",
})
@ParameterizedTest
public void testIntValue(long numerator, long divisor, int expected) {
    RationalNumber rational = RationalNumber.factoryMethod(numerator, divisor);
    int actual = rational.intValue();
    assertEquals(expected, actual);

}
```

Il metodo testValueOf invece verifica la correttezza del metodo valueOf che riceve in ingresso una stringa e calcola numeratore e denominatore utilizzando la tecnica delle approssimazioni successive.
Questo test è differente dai precedenti e non ho potuto utilizzare le stesse classi di equivalenza.
Il metodo riceve in ingresso un double e restituisce il numeratore e denominatore ottenuto utilizzando la tecnica delle approssimazioni successive.
Per testare questo metodo ho utilizzato le seguenti classi di equivalenza

CE	Valore	Descrizione
CE1	7	Intero positivo
CE2	1.89	Numero decimale maggiore di 1
CE3	0.333333333333333	Numero decimale periodico
CE4	-0.5242	Numero decimale compreso tra -1 e 0
CE5	0	0
CE6	-1.89	Numero decimale minore di -1
CE7	0.5242	Numero decimale compreso tra 0 e 1

```
@CsvSource(value = {
        "7;7;1",
        "1.89;189;100",
        "0.333333333333333;1;3",
        "-0.5242;-2621;5000",
        "-1.89;-189;100",
        "0.5242;2621;5000",
        "0;0;1",
}, delimiter = ';')
```

```java
@ParameterizedTest
public void testValueOf(double n, double expectedNumerator, double expectedDivisor) {
    RationalNumber rational = RationalNumber.valueOf(n);
    assertEquals(expectedNumerator, rational.numerator);
    assertEquals(expectedDivisor, rational.divisor);
}
```

Riporto l'esito dei test case eseguiti.

ID	Precond	Input	Expected Output	Output	Post Cond	Result
1	Rational number esiste	numString = 7	num = 7; den = 1;	num = 7; den = 1;	No Exception Thown	Success
2	Rational number esiste	numString = 1.89	num = 189; den = 100;	num = 189; den = 100;	No Exception Thown	Success
3	Rational number esiste	numString = 0.333333333333333	num = 1; den = 3;	num = 1; den = 3;	No Exception Thown	Success
4	Rational number esiste	numString = -0.5242	num = -2621; den = 5000;	num = -2621; den = 5000;	No Exception Thown	Success
5	Rational number esiste	numString = 0	num = 0; den = 1;	num = 0; den = 1;	No Exception Thown	Success
6	Rational number esiste	numString = -1.89	num = -189; den = 100;	num = -189; den = 100;	No Exception Thown	Success
7	Rational number esiste	numString = 0.5242	num = 2621; den = 5000;	num = 2621; den = 5000;	No Exception Thown	Success

Per questa classe ho ottenuto una copertura dell'86%.