Università di Catania

Uni ct MATEMATICA E INFORMATICA

Mutation & Regression Testing

Alessandro Midolo, PhD

Dipartimento di Matematica e Informatica

Università di Catania

alessandro.midolo@unict.it

https://www.dmi.unict.it/amidolo/

A.A. 2024/2025

Qualità di una test suite

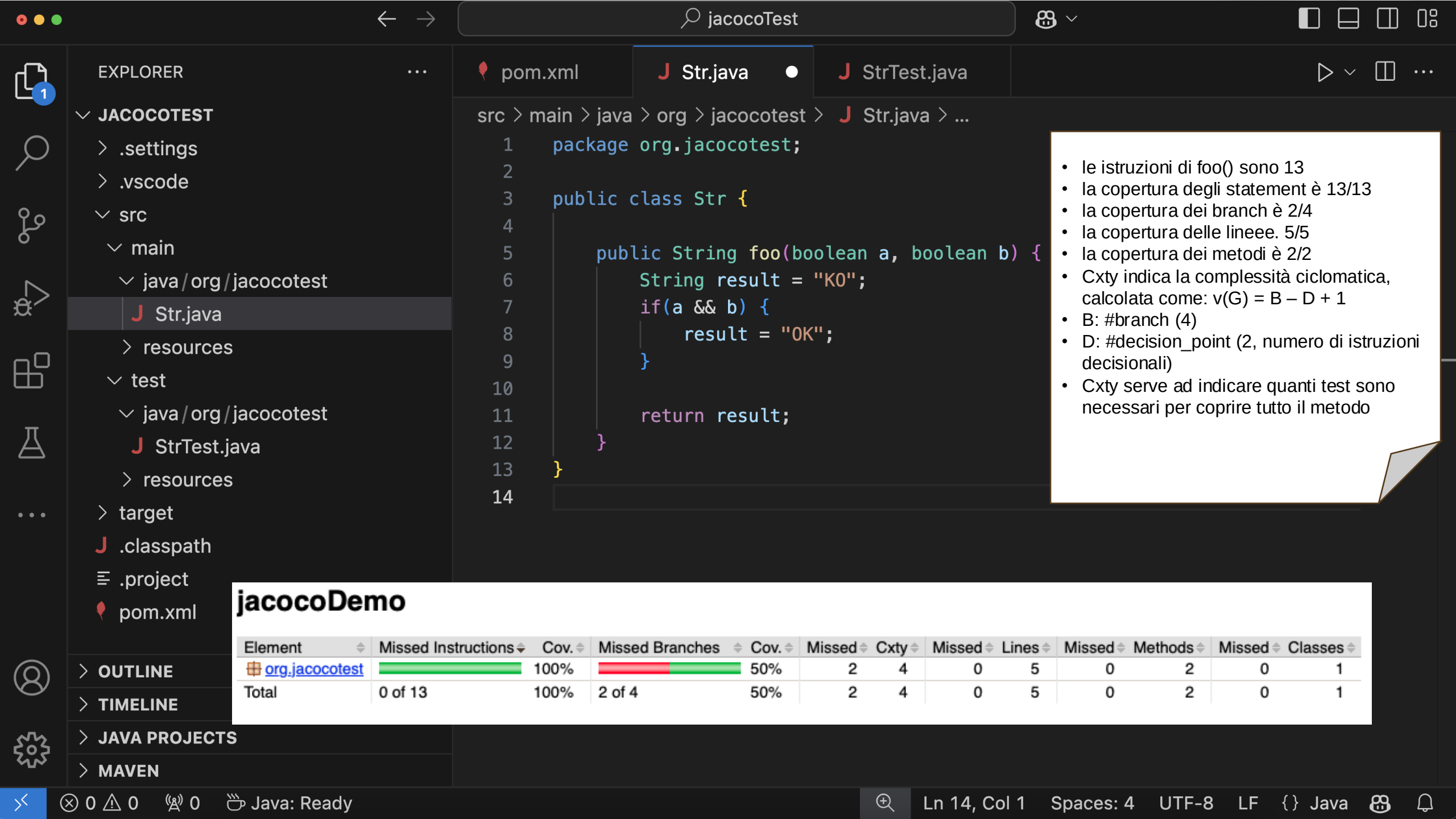
- L'obiettivo di una test suite è quello di esercitare il comportamento di un'applicazione, distinguendo un comportamento anomalo (almeno uno dei test fallisce - **fail**) dal comportamento atteso (tutti i test superati - **pass**)
- Quanti e quali test dovremmo aggiungere per avere una test suite efficace?
- In teoria, il miglior modo per misurare la **qualità** di una test suite sarebbe di valutarne la capacità di individuare **difetti reali (bug)** nell'applicazione
- "Il test di un programma può essere usato per mostrare la presenza di bug, ma mai per mostrare la loro assenza" – Dijkstra (1970)
- Una delle metriche maggiormente usate per misurare la qualità di una test suite è la **copertura** del codice durante l'esecuzione dei test
- Non è l'unica metrica di copertura, ma è certamente una delle più immediate ed usate, anche grazie alla presenza di numerosi tool integrati con gli IDE
  - Cobertura
  - JaCoCo
  - Codcov

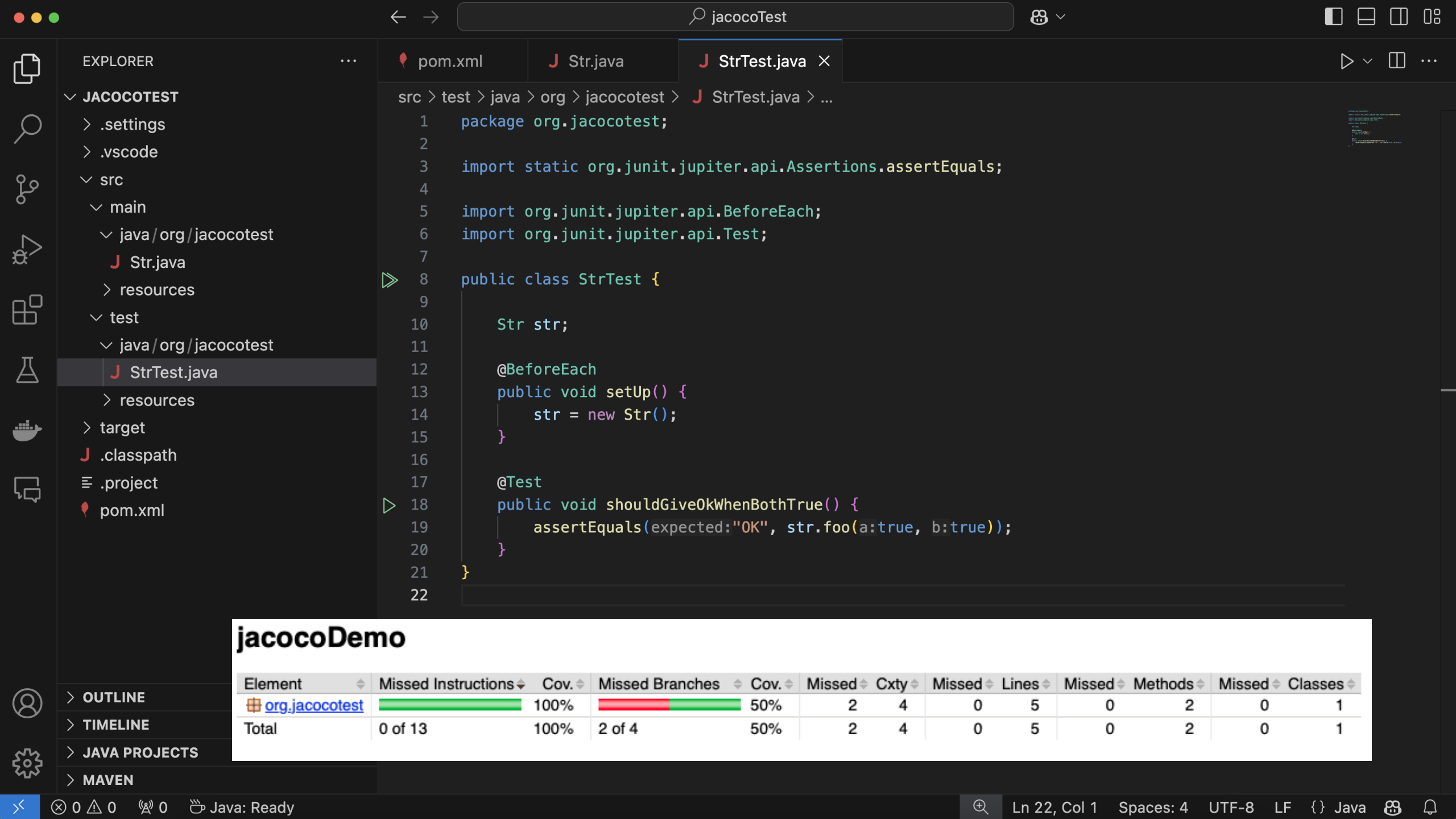
Code Coverage

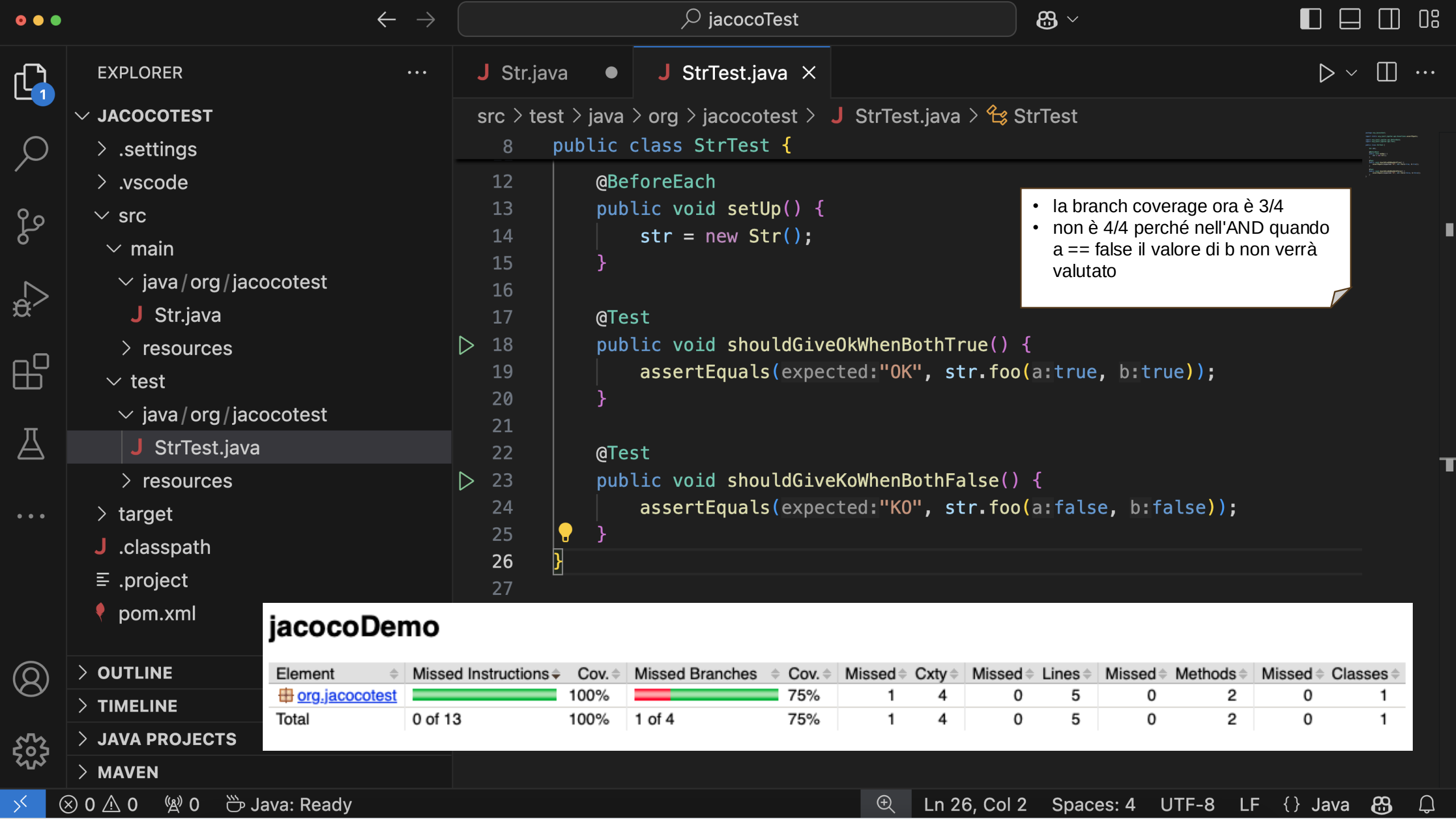
- **Class coverage**: percentuale di classi coperte
- **Method coverage**: percentuale di metodi coperti
- **Line coverage**: percentuale di linee di codice coperte
- **Statement coverage**: percentuale di istruzioni coperte
  - su una linea posso avere più di una istruzione
  - teoricamente più preciso, ma spesso usato come sinonimo di line coverage
- **Branch coverage**: percentuale di rami di esecuzione coperti
  - Es. if (a && b) ha 2 rami da coprire:
    - (a && b) è vera
    - (a && b) è falsa
- Condition coverage: percentuale di condizioni coperte
  - Es. if (a && b) ha 4 condizioni da coprire: (a) è vera, (a) è falsa, (b) è vera, (b) è falsa

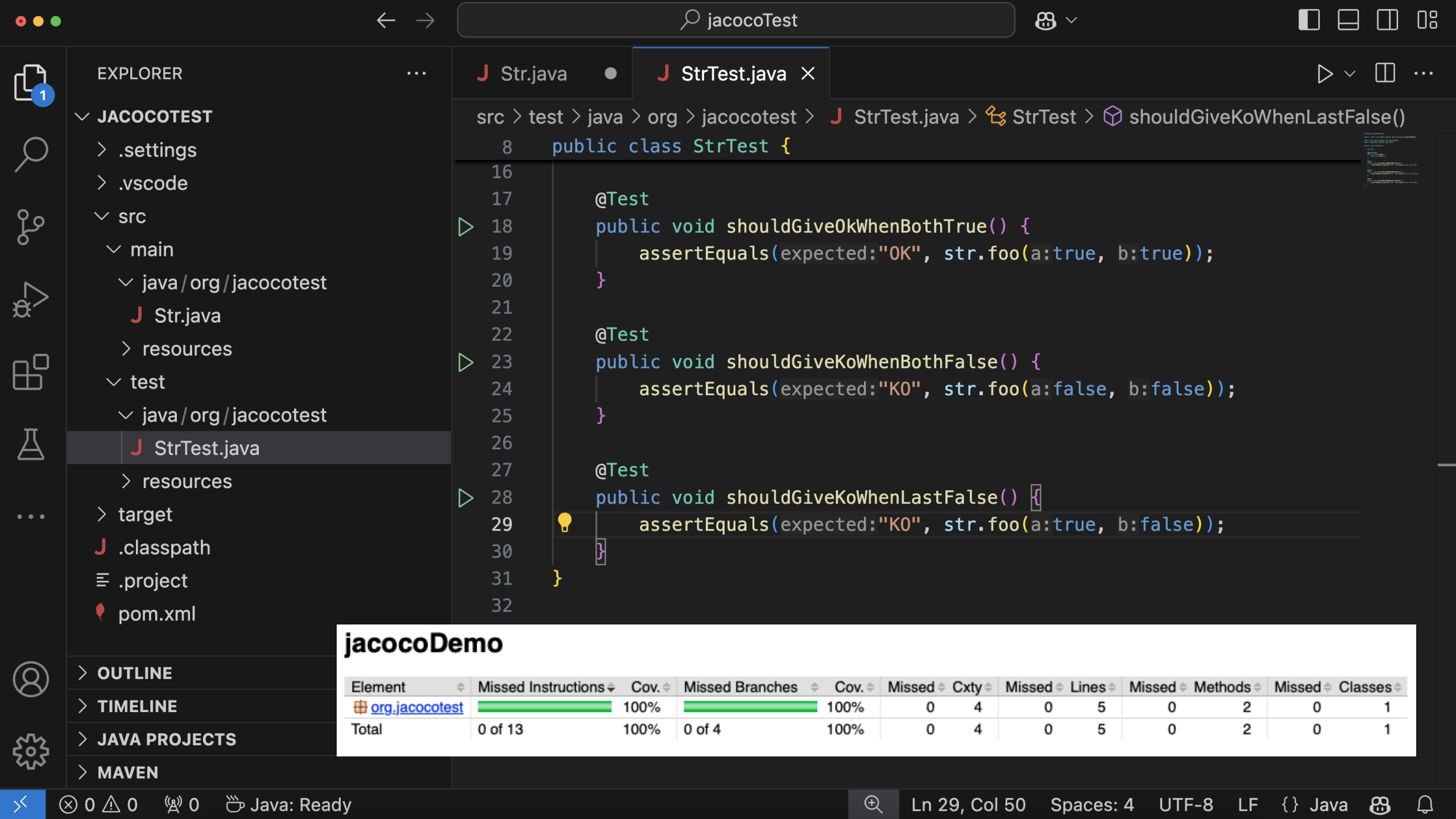
JaCoCo

- JaCoCo (**Java Code Coverage**) è una libreria open-source ampiamente utilizzata per misurare la copertura del codicenei progetti Java. Serve a verificare quanto del tuo codice è effettivamente testato tramite test unitari e di integrazione, generando report dettagliati che aiutano a individuare le parti non coperte e a migliorare la qualità dei test
- Possiamo eseguire il codice internamente da maven:
  - **mvn clean org.jacoco:jacoco-maven-plugin:0.8.8:prepare-agent test org.jacoco:jacoco-maven-plugin:0.8.8:report**


jacocoTest
EXPLORER
JACOCOTEST
.settings
.vscode
src
main
java/org/jacocotest
Str.java
resources
test
java/org/jacocotest
StrTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
pom.xml
Str.java
StrTest.java
src > main > java > org > jacocotest > Str.java > ...
1 package org.jacocotest;
2
3 public class Str {
4
5 public String foo(boolean a, boolean b) {
6 String result = "KO";
7 if(a && b) {
8 result = "OK";
9 }
10
11 return result;
12 }
13 }
14
• le istruzioni di foo() sono 13
• la copertura degli statement è 13/13
• la copertura dei branch è 2/4
• la copertura delle lineee. 5/5
• la copertura dei metodi è 2/2
• Cxty indica la complessità ciclomatica, calcolata come: v(G) = B – D + 1
• B: #branch (4)
• D: #decision_point (2, numero di istruzioni decisionali)
• Cxty serve ad indicare quanti test sono necessari per coprire tutto il metodo
jacocoDemo
Element
Missed Instructions
Cov.
Missed Branches
Cov.
Missed
Cxty
Missed
Lines
Missed
Methods
Missed
Classes
org.jacocotest
100%
50%
2
4
0
5
0
2
0
1
Total
0 of 13
100%
2 of 4
50%
2
4
0
5
0
2
0
1
0 0 0 Java: Ready
Ln 14, Col 1
Spaces: 4
UTF-8
LF
Java

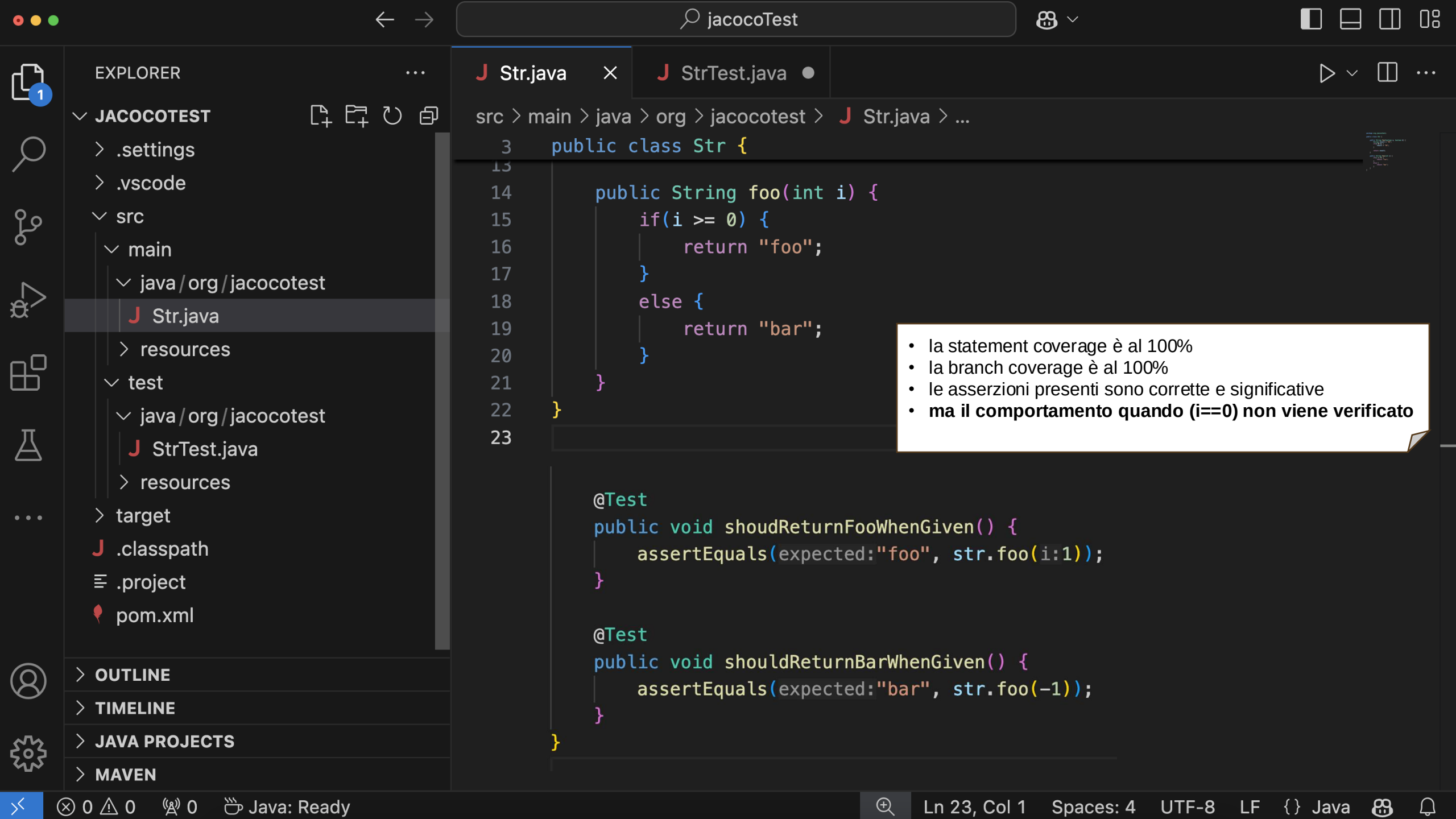

jacocoTest
EXPLORER
JACOCOTEST
.settings
.vscode
src
main
java/org/jacocotest
Str.java
resources
test
java/org/jacocotest
StrTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
pom.xml
Str.java
StrTest.java
src > test > java > org > jacocotest > StrTest.java > ...
1 package org.jacocotest;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.Test;
7
8 public class StrTest {
9
10 Str str;
11
12 @BeforeEach
13 public void setUp() {
14 str = new Str();
15 }
16
17 @Test
18 public void shouldGiveOkWhenBothTrue() {
19 assertEquals(expected:"OK", str.foo(a:true, b:true));
20 }
21 }
22
jacocoDemo
Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes
org.jacocotest | 100% | 50% | 2 | 4 | 0 | 5 | 0 | 2 | 0 | 1
Total | 0 of 13 | 100% | 2 of 4 | 50% | 2 | 4 | 0 | 5 | 0 | 2 | 0 | 1
Java: Ready
Ln 22, Col 1
Spaces: 4
UTF-8
LF
Java

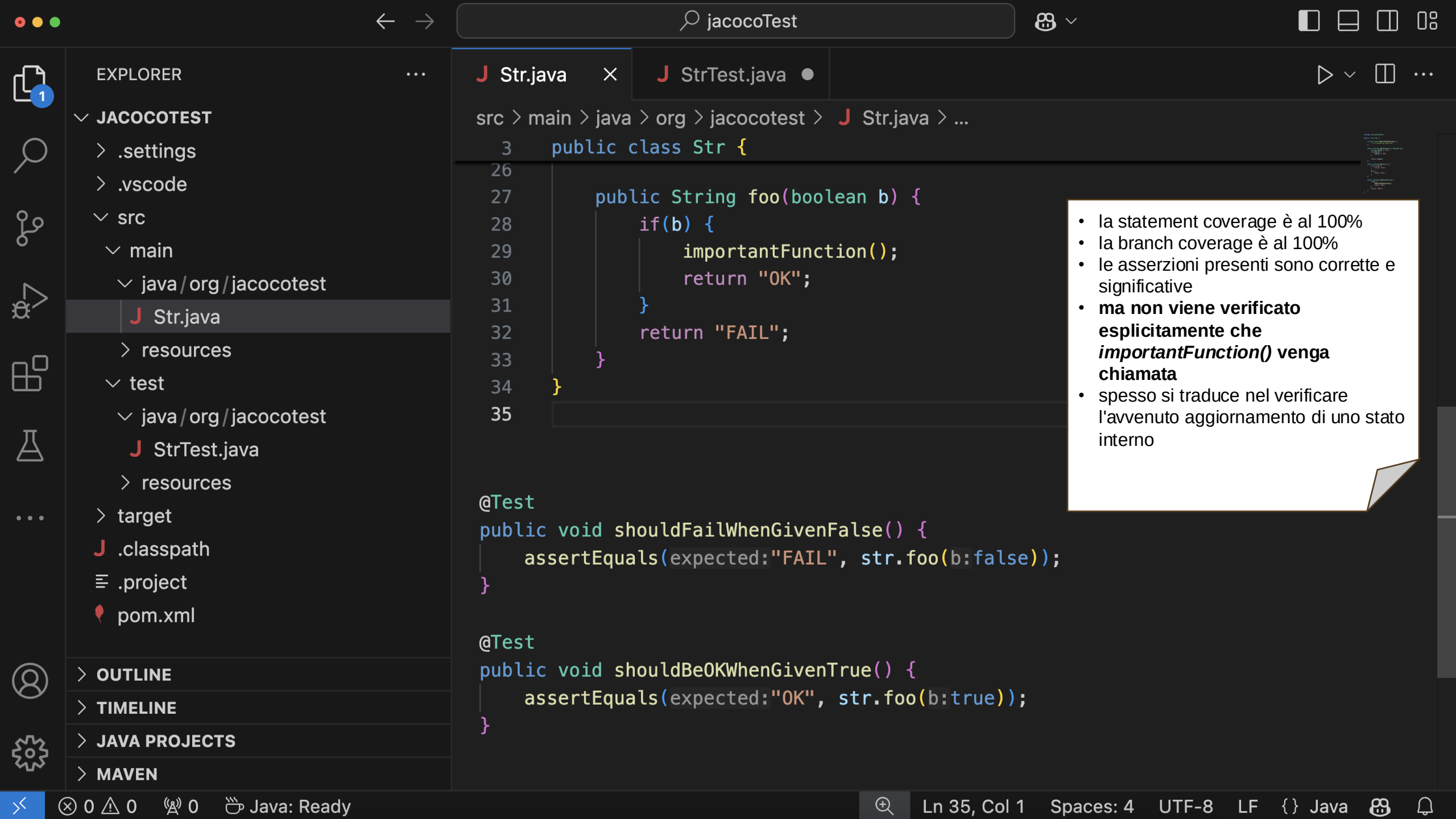

jacocoTest
EXPLORER
JACOCOTEST
.settings
.vscode
src
main
java / org / jacocotest
Str.java
resources
test
java / org / jacocotest
StrTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
Str.java
StrTest.java
src > test > java > org > jacocotest > StrTest.java > StrTest
8 public class StrTest {
12 @BeforeEach
13 public void setUp() {
14 str = new Str();
15 }
16
17 @Test
18 public void shouldGiveOkWhenBothTrue() {
19 assertEquals(expected:"OK", str.foo(a:true, b:true));
20 }
21
22 @Test
23 public void shouldGiveKoWhenBothFalse() {
24 assertEquals(expected:"KO", str.foo(a:false, b:false));
25 }
26 }
27
• la branch coverage ora è 3/4
• non è 4/4 perché nell'AND quando a == false il valore di b non verrà valutato
jacocoDemo
Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes
org.jacocotest | 100% | 75% | 1 | 4 | 0 | 5 | 0 | 2 | 0 | 1
Total | 0 of 13 | 100% | 1 of 4 | 75% | 1 | 4 | 0 | 5 | 0 | 2 | 0 | 1
0 0 0 Java: Ready
Ln 26, Col 2 Spaces: 4 UTF-8 LF Java


jacocoTest
EXPLORER
JACOCOTEST
.settings
.vscode
src
main
java/org/jacocotest
Str.java
resources
test
java/org/jacocotest
StrTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
Str.java
StrTest.java
src > test > java > org > jacocotest > StrTest.java > StrTest > shouldGiveKoWhenLastFalse()
8 public class StrTest {
16
17 @Test
18 public void shouldGiveOkWhenBothTrue() {
19 assertEquals(expected:"OK", str.foo(a:true, b:true));
20 }
21
22 @Test
23 public void shouldGiveKoWhenBothFalse() {
24 assertEquals(expected:"KO", str.foo(a:false, b:false));
25 }
26
27 @Test
28 public void shouldGiveKoWhenLastFalse() {
29 assertEquals(expected:"KO", str.foo(a:true, b:false));
30 }
31 }
32
jacocoDemo
Element
Missed Instructions
Cov.
Missed Branches
Cov.
Missed
Cxty
Missed
Lines
Missed
Methods
Missed
Classes
org.jacocotest
100%
100%
0
4
0
5
0
2
0
1
Total
0 of 13
100%
0 of 4
100%
0
4
0
5
0
2
0
1
0 0 0
Java: Ready
Ln 29, Col 50
Spaces: 4
UTF-8
LF
Java

Limiti della code coverage

- Avere una code coverage elevata è una condizione necessaria ma in certi casi può non essere sufficiente
- Il fatto che abbiamo ottenuto il 100% della copertura del codice non significa che abbiamo coperto tutti i possibili comportamenti dell'applicazione
- Cicli, ricorsione, ordine di esecuzione dei metodi, codice mancante…
- La copertura ci indica solo che abbiamo eseguito il codice durante i test, non che ne abbiamo testato il comportamento in maniera significativa
- Il caso limite è quello di test senza asserzioni
- Anche in presenza di asserzioni, queste devono essere scelte in modo tale che siano in grado di individuare la presenza di possibili comportamenti anomali
- Nella maggior parte dei casi, il problema non risiede nei test presenti, ma nei **test mancanti**: è necessario considerare dei test case aggiuntivi


jacocoTest
EXPLORER
JACOCOTEST
.settings
.vscode
src
main
java/org/jacocotest
Str.java
resources
test
java/org/jacocotest
StrTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
Str.java
StrTest.java
src > main > java > org > jacocotest > Str.java > ...
3 public class Str {
14 public String foo(int i) {
15 if(i >= 0) {
16 return "foo";
17 }
18 else {
19 return "bar";
20 }
21 }
22 }
23
@Test
public void shoudReturnFooWhenGiven() {
assertEquals(expected:"foo", str.foo(i:1));
}
@Test
public void shouldReturnBarWhenGiven() {
assertEquals(expected:"bar", str.foo(-1));
}
}
• la statement coverage è al 100%
• la branch coverage è al 100%
• le asserzioni presenti sono corrette e significative
• ma il comportamento quando (i==0) non viene verificato
0 0 0 Java: Ready
Ln 23, Col 1 Spaces: 4 UTF-8 LF Java


jacocoTest
EXPLORER
JACOCOTEST
.settings
.vscode
src
main
java/org/jacocotest
Str.java
resources
test
java/org/jacocotest
StrTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
Str.java
StrTest.java
src > main > java > org > jacocotest > Str.java > ...
3 public class Str {
26
27 public String foo(boolean b) {
28 if(b) {
29 importantFunction();
30 return "OK";
31 }
32 return "FAIL";
33 }
34 }
35
@Test
public void shouldFailWhenGivenFalse() {
assertEquals(expected:"FAIL", str.foo(b:false));
}
@Test
public void shouldBeOKWhenGivenTrue() {
assertEquals(expected:"OK", str.foo(b:true));
}
la statement coverage è al 100%
la branch coverage è al 100%
le asserzioni presenti sono corrette e significative
ma non viene verificato esplicitamente che importantFunction() venga chiamata
spesso si traduce nel verificare l'avvenuto aggiornamento di uno stato interno
0 0 0
Java: Ready
Ln 35, Col 1
Spaces: 4
UTF-8
LF
Java

Mutation Testing

- Il mutation testing è una tecnica per **misurare l'efficacia di una test suite**
- Consiste nel generare in automatico delle versioni modificate dell'applicazione da testare (dette in gergo "**mutanti**") inserendo in ognuna un singolo difetto artificiale
- Un difetto consiste in una **piccola modifica** al codice sorgente, a cui dovrebbe corrispondere un comportamento anomalo individuabile tramite i test
- Se la test suite è in grado di rilevare il difetto (almeno uno dei test da **fail** come esito) il mutante è stato eliminato (**killed** in gergo)
- Se la test suite non è in grado di rilevare il difetto (tutti i test danno **pass** come esito) il mutante è **sopravvissuto**
- Se un mutante è sopravvissuto, uno sviluppatore/tester può ragionevolmente dedurre che i test non stiano controllando effettivamente tutti i possibili comportamenti dell'applicazione
- L'efficacia di una test suite è quindi **direttamente proporzionale** alla percentuale di **mutanti uccisi**
- Nota: il mutation testing viene eseguito solo su di una **green test suite**, ovvero una test suite in cui tutti i test vengono superati sulla versione originale del codice

Esempio di Mutazione

```java
public String foo(int i) {
    if(i >= 0) {
        return "foo";
    }
    else {
        return "bar";
    }
}
```

Originale

Conditionals_Boundary_Mutator

```java
public String foo(int i) {
    if(i > 0) {
        return "foo";
    }
    else {
        return "bar";
    }
}
```

Mutante

```java
@Test
public void shoudReturnFooWhenGiven() {
    assertEquals(expected:"foo", str.foo(i:1));
}

@Test
public void shouldReturnBarWhenGiven() {
    assertEquals(expected:"bar", str.foo(-1));
}
```

Test case eseguiti sul mutante

Esempio di Mutazione

- Nessuno dei due test fallirà, non riuscendo quindi ad uccidere il mutante rilevando il cambiamento nel comportamento causato dalla modifica inserita
- Questo suggerisce l'aggiunta del test case mancante

```
@Test
public void shouldReturnFooWhenGivenZero() {
    assertEquals(expected:"foo", str.foo(i:0));
}
```

Limite: Mutazioni Equivalenti

```java
public String foo() {
    int i = 2;
    if(i >= 1) {
        return "foo";
    }
    else {
        return "bar";
    }
}
```

Originale

Conditionals_Boundary_Mutator

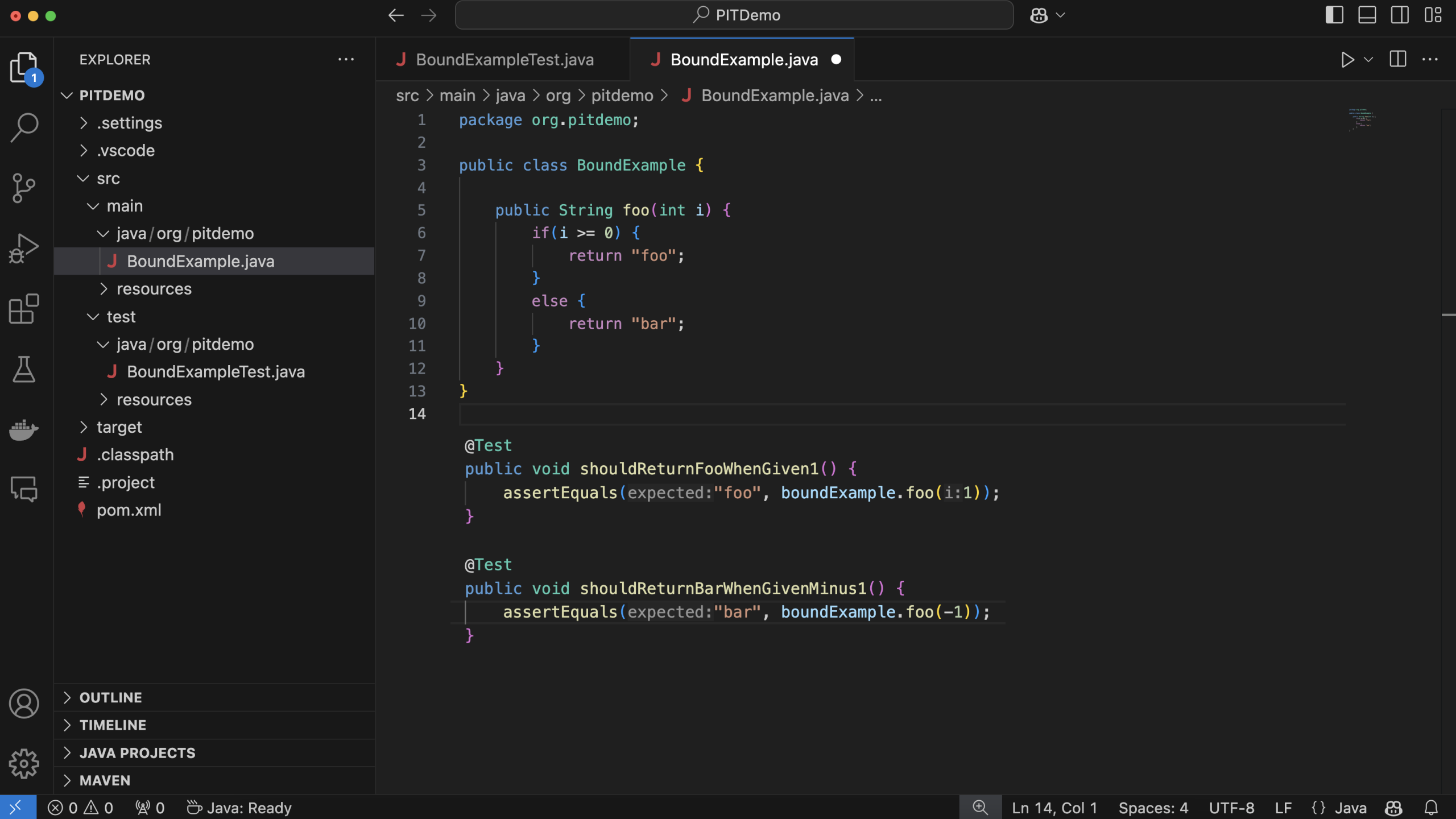
```java
public String foo() {
    int i = 2;
    if(i > 1) {
        return "foo";
    }
    else {
        return "bar";
    }
}
```

Mutante

- In questo caso la modifica ha creato un mutante con un comportamento equivalente a quello originale
- Il mutante non potrà quindi essere ucciso in alcun modo
- Un esempio comune è una mutazione applicata alle istruzioni di logging o di debug: in quel caso è il programmatore a non essere interessato al testing di questi comportamenti
- Dal report dell'analisi o dalla configurazione del tool di mutation testing è possibile in molti casi individuare o escludere queste casistiche

PITest

- Tool di mutation testing per Java e JVM
- Eseguito da riga di comando o come plugin per Maven, Gradle e Ant
- Dispone di diversi operatori di mutazione (mutatori)
- Le mutazioni vengono applicate a livello di bytecode
- Facile da configurare tramite Maven
- Report che integra copertura del codice e copertura dei mutanti
- Codice open source: https://github.com/hcoles/pitest
- Estendibile con nuovi operatori o nuovi engine
- Supporto all'extreme mutation testing
- Comando con maven: **mvn test-compile org.pitest:pitest-maven:mutationCoverage**


PITDemo
EXPLORER
PITDEMO
.settings
.vscode
src
main
java/org/pitdemo
BoundExample.java
resources
test
java/org/pitdemo
BoundExampleTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
BoundExampleTest.java
BoundExample.java
src > main > java > org > pitdemo > BoundExample.java > ...
1 package org.pitdemo;
2
3 public class BoundExample {
4
5 public String foo(int i) {
6 if(i >= 0) {
7 return "foo";
8 }
9 else {
10 return "bar";
11 }
12 }
13 }
14
@Test
public void shouldReturnFooWhenGiven1() {
assertEquals(expected:"foo", boundExample.foo(i:1));
}
@Test
public void shouldReturnBarWhenGivenMinus1() {
assertEquals(expected:"bar", boundExample.foo(-1));
}
0 0 0
Java: Ready
Ln 14, Col 1
Spaces: 4
UTF-8
LF
Java

BoundExample.java

```
1   package org.pitdemo;
2
3   public class BoundExample {
4
5       public String foo(int i) {
6   2       if(i >= 0) {
7   1           return "foo";
8           }
9           else {
10  1           return "bar";
11          }
12      }
13  }
```

Mutations

```
6   1. changed conditional boundary → SURVIVED Covering tests
    2. negated conditional → KILLED
7   1. replaced return value with "" for org/pitdemo/BoundExample::foo → KILLED
10  1. replaced return value with "" for org/pitdemo/BoundExample::foo → KILLED
```

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- org.pitdemo.BoundExampleTest.[engine:junit-jupiter]/[class:org.pitdemo.BoundExampleTest]/[method:shouldReturnFooWhenGiven1()] (4 ms)
- org.pitdemo.BoundExampleTest.[engine:junit-jupiter]/[class:org.pitdemo.BoundExampleTest]/[method:shouldReturnBarWhenGivenMinus1()] (50 ms)

Report generated by PIT 1.17.1

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	4/4	75%	3/4	75%	3/4

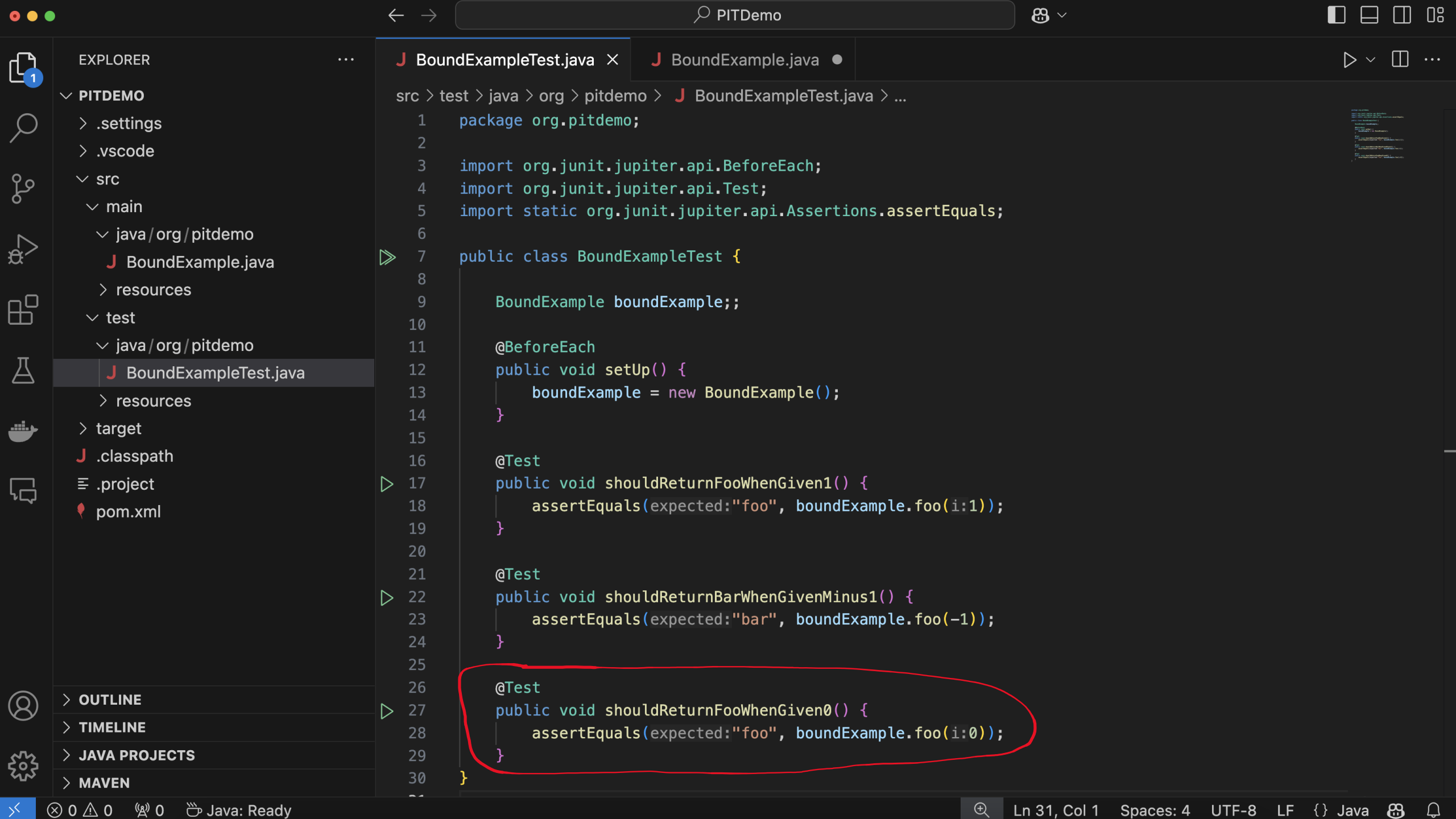
Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.pitdemo	1	100%	4/4	75%	3/4	75%	3/4

Report generated by PIT 1.17.1

Enhanced functionality available at arcmutate.com

- 100% di code (line) coverage ma 75% di mutation coverage
- Il report mostra che 1 dei 2 mutanti in riga 6 è sopravvissuto
- Mutatore: **CONDITIONALS_BOUNDARY_MUTATOR**
- Il mutatore converte
  - >= in > e viceversa
  - <= in < e viceversa
- Questo suggerisce l'aggiunta di un nuovo test case per controllare il comportamento ai bordi (i == 0)


PITDemo
EXPLORER
PITDEMO
.settings
.vscode
src
main
java/org/pitdemo
BoundExample.java
resources
test
java/org/pitdemo
BoundExampleTest.java
resources
target
.classpath
.project
pom.xml
OUTLINE
TIMELINE
JAVA PROJECTS
MAVEN
BoundExampleTest.java
BoundExample.java
src > test > java > org > pitdemo > BoundExampleTest.java > ...
1 package org.pitdemo;
2
3 import org.junit.jupiter.api.BeforeEach;
4 import org.junit.jupiter.api.Test;
5 import static org.junit.jupiter.api.Assertions.assertEquals;
6
7 public class BoundExampleTest {
8
9 BoundExample boundExample;;
10
11 @BeforeEach
12 public void setUp() {
13 boundExample = new BoundExample();
14 }
15
16 @Test
17 public void shouldReturnFooWhenGiven1() {
18 assertEquals(expected:"foo", boundExample.foo(i:1));
19 }
20
21 @Test
22 public void shouldReturnBarWhenGivenMinus1() {
23 assertEquals(expected:"bar", boundExample.foo(-1));
24 }
25
26 @Test
27 public void shouldReturnFooWhenGiven0() {
28 assertEquals(expected:"foo", boundExample.foo(i:0));
29 }
30 }
0 0 0 Java: Ready
Ln 31, Col 1 Spaces: 4 UTF-8 LF Java

BoundExample.java

```
1   package org.pitdemo;
2
3   public class BoundExample {
4
5       public String foo(int i) {
6   2       if(i >= 0) {
7   1           return "foo";
8           }
9           else {
10  1           return "bar";
11          }
12      }
13  }
```

Mutations

```
6   1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
7   1. replaced return value with "" for org/pitdemo/BoundExample::foo → KILLED
10  1. replaced return value with "" for org/pitdemo/BoundExample::foo → KILLED
```

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- org.pitdemo.BoundExampleTest.[engine:junit-jupiter]/[class:org.pitdemo.BoundExampleTest]/[method:shouldReturnFooWhenGiven0()] (2 ms)
- org.pitdemo.BoundExampleTest.[engine:junit-jupiter]/[class:org.pitdemo.BoundExampleTest]/[method:shouldReturnFooWhenGiven1()] (3 ms)
- org.pitdemo.BoundExampleTest.[engine:junit-jupiter]/[class:org.pitdemo.BoundExampleTest]/[method:shouldReturnBarWhenGivenMinus1()] (49 ms)

Pit Test Coverage Report

Package Summary

org.pitdemo

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	4/4	100%	4/4	100%	4/4

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
BoundExample.java	100%	4/4	100%	4/4	100%	4/4

Report generated by PIT 1.17.1

Operatori di mutazione di default

Mutatori	Descrizione
Conditionals Boundary	<= ←→ < ; >= ←→ >
Increments	i++ ←→ i--
Invert Negatives	-x ←→ x
Math	a+b ←→ a-b; a*b ←→ a/b
Negate Conditionals	== ←→ != ; >= ←→ < ; <= ←→ >
Void Method Calls	elimina una chiamata ad un metodo void
Empty Returns	restituisce un valore "vuoto" in base al tipo di ritorno
False Returns	come il precedente, ma con False
True Returns	come il precedente, ma con True
Null returns	restituisce null

Funzionamento di PIT

- Per ogni classe dell'applicazione, PIT visita ogni suo metodo cercando istruzioni a cui applicare i mutatori selezionati (visitor pattern)
- Il **numero di mutanti** generati dipende quindi sia dal numero di **mutatori** che dalla **dimensione** dell'applicazione (numero di istruzioni)
- Per **ciascuno dei mutanti** generati verrà effettuata **l'esecuzione dei test**, in modo da verificare se il mutante viene ucciso o meno
- Questo comporta l'esecuzione di un gran numero di test: il **mutation testing è molto oneroso** rispetto alla misura di copertura del codice
- Per **limitare il numero di test** da eseguire per ciascun mutante, prima di generare i mutanti PIT esegue **un'analisi della copertura** del codice
- Per ogni test viene tenuta traccia delle istruzioni coperte
- Preso quindi un mutante, verranno **selezionati solo quei in grado di coprire le istruzioni modificate** dall'operatore di mutazione

Regression Testing

- Attività di testing che viene eseguita per accertarsi che l'introduzione di nuove modifiche non abbia **compromesso** la correttezza delle **funzionalità** esistenti
- Questo permette di individuare tempestivamente eventuali difetti inavvertitamente introdotti nel codice anche in parti non direttamente oggetto delle nuove modifiche
  - Es. l'aggiornamento del formato di output di una funzionalità di libreria ha causato un errore di parsing in un componente che la richiama
- Essendo un'operazione frequente, tipicamente ci si avvale di **strumenti automatici** per l'esecuzione dei test di regressione (JUnit, CI pipelines…)
- Il modo più semplice consiste nel rieseguire tutti i test (**retest all**), ma al crescere dell'applicazione il numero di test presenti aumenta (es. il numero di unit test è proporzionale al numero di metodi)
- Per rendere il regression testing **cost-effective**, vari approcci sono stati proposti:
  - **test suite minimisation**: identificare ed ignorare i test case ridondanti e obsoleti
  - **test suite reduction**: identificare ed eliminare i test case ridondanti e obsoleti
  - **test case selection**: selezionare i test che esercitano il codice influenzato dalle modifiche introdotte (sia direttamente che indirettamente)
  - **test case prioritisation**: ordinare i test secondo un criterio scelto (es. copertura)

Test Suite Minimisation & Reduction

- Entrambi puntano ad individuare test case ridondanti o obsoleti con la differenza che:
  - per **minimisation** questi test case vengono solo **ignorati** (es. per avere una test suite più efficiente da eseguire spesso, posticipando un retest all a dopo le 18:00)
  - per **reduction** questi test case vengono **eliminati**, riducendo permanentemente la dimensione della test suite
- Tipicamente viene risolto come un **problema di ottimizzazione**:
  - scelto un criterio di copertura (ed. del codice o dei mutanti)
  - viene individuato un **sottoinsieme di test case** in grado di mantenere la **stessa copertura**
  - con un minor numero di test o un minor tempo di esecuzione
- Gli approcci si distinguono inoltre in:
  - **adeguati**: non vi è perdita di copertura nella test suite risultante
  - **non adeguati**: accettata una percentuale di perdita di copertua (rilassamento)

Test Suite Selection

- L'obiettivo è selezionare solo quei test relativi a parti di codice **direttamente** o **indirettamente coinvolte** dalle ultime modifiche introdotte nel codice
- Questo per individuare rapidamente difetti introdotti con le ultime modifiche, correggendoli in maniera tempestiva
- È immediato individuare i test che chiamano direttamente il codice modificato (es. il metodo o la classe in cui sono state inserite le modifiche)
- Meno immediato se l'interazione è indiretta (il test chiama un metodo che chiama un metodo…)
- La soluzione naturale è:
  - dopo aver apportato delle modifiche eseguire tutti i test una volta sola, per poi rieseguire solo i test falliti, finché questi non vengono risolti
  - ma il problema è proprio evitare di rieseguire tutti i test, anche se una sola volta
- La soluzione intuitiva è usare le informazioni sulla copertura e rieseguire solo i test che **coprono il codice modificato**:
  - Male per ottenere le informazioni di **copertura aggiornate** dovrei rieseguire tutti i test sulla nuova versione del codice
  - Si può limitare usando **l'analisi statica delle dipendenze** e **l'esecuzione simbolica** per determinare quali test potrebbero "potenzialmente" coprire il codice modificato

Test Suite Prioritisation

- Mira ad individuare **l'ordinamento** ideale dei test case, in modo da ottenere il massimo dei benefici anche se l'esecuzione dei test viene interrotta prematuramente
- L'obiettivo è quello di massimizzare la probabilità di individuare dei difetti eseguendo solo una (prima) parte dei test a disposizione (**maximize early fault detection**)
- Alcuni approcci usano un ordinamento basato su un **criterio di copertura** (es. eseguo prima i test in grado di uccidere più mutanti)
- Altri approcci usano informazioni **sulle statistiche dei difetti rilevati** da ciascun test
- Altri danno priorità ai **test falliti di recente**, perché hanno una maggior probabilità di fallire ancora

Generazione Automatica di Test

- La creazione di test da parte dello sviluppatore è un'operazione necessaria ma faticosa e in alcuni casi ripetitiva
- Raggiungere un buon livello di copertura può richiede la creazione di un numero considerevole di test case
- Spesso, certe parti del test sono ripetute (boilerplate) e trovare il giusto modo per riusare test (o porzioni) precedentemente scritti può aiutare la creazione di nuove test
- In letteratura diversi approcci e tool sono stati proposti per la generazione automatica di test case

EvoSuite

- Genera automaticamente test JUnit completi di asserzioni (https://github.com/EvoSuite/evosuite)
- Disponibile sia come jar stand alone, che come plugin per Eclipse o per Jenkins
- Evosuite genere test per ciascuna classe: ognuna sarà di volta in volta la Class Under Test (CUT)
- I test vengono generati usando un approccio **evolutivo tramite algoritmo genetico**
- Data una o più classi da testare:
  - usa un approccio randomico per generare una **popolazione di test suite** (nota, non test case, intere test suite!)
  - usando un **criterio di copertura** come **fitness function** (es. branch coverage raggiunta da ciascuna test suite) vengono selezionati i migliori individui per procreare la prossima generazione
  - come operatore di **crossing** tra due individui (test suite) viene usato lo **scambio casuale di test case**
  - come operatore di **mutazione** di un individuo, vengono **aggiunti**, **eliminati** o **modificati** i test case presenti nella test suite (modifica di un test case: aggiunta, eliminazione o modifica di istruzioni/parametri)
- Il processo continua in maniera iterativa finché non si arriva a convergenza o si raggiunge un limite massimo di iterazioni/tempo, selezionando la test suite che massimizza la copertura
- La test suite selezionata è oggetto di ulteriori passi di test suite reduction per eliminare i test case e le istruzioni che non contribuiscono al miglioramento della copertura

Automatic Generation of Accurate Test Templates based on JUnit Asserts

Alessandro Midolo*
Dipartimento di Matematica e Informatica, University of Catania, Italy

Emiliano Tramontana
Dipartimento di Matematica e Informatica, University of Catania, Italy

ABSTRACT

Software testing is a much needed activity to ensure quality of software systems, and a large and effective test suite has the potential to identify defects. However, substantial effort is required when implementing test cases, due to the knowledge that developers have to acquire on the structure and behaviour of the system under test, and for the time needed to write many test cases to cover most of the execution paths. This paper proposes an automatic approach to generate templates of test cases. A generated test template consists of a JUnit assert, a method call and some default parameters, and can be customised by the developer, since generated code is easy to read. In the proposed approach, some selected industrial software repositories have been analysed to determine the most frequent assert used according to the return type of a method call. Then, such a frequency is used to guide the generation of new test cases for other software systems. The approach has been assessed on real-world Java projects, proving that generated test cases increase application code coverage considerably, while exhibiting readability and effectiveness.

CCS CONCEPTS

• **Software and its engineering**; • **Object oriented development.**;

KEYWORDS

testing, code generation, code quality

ACM Reference Format:
Alessandro Midolo and Emiliano Tramontana. 2023. Automatic Generation of Accurate Test Templates based on JUnit Asserts. In *the 7th International Conference on Algorithms, Computing and Systems (ICACS 2023), October 19–21, 2023, Larissa, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3631908.3631926

1 INTRODUCTION

An effective test suite can validate the correctness of a software system and improve its maintainability, reusability and robustness over time [1, 13, 25]. Moreover, a test suite can help developers check that enhancements and refactoring activities do not introduce defects [3, 21]. However, manually implementing a test suite could be challenging and time-consuming, since the developer should produce at least a test case executing each method and feature in different scenarios. Conversely, poor test coverage decreases the effectiveness of a test suite, leading to the misidentification of defects [16]. The state of the art provides several approaches to assist developers during software testing. The selection of proper input values can be a critical task to fulfil when implementing test cases, indeed a broad and accurate spectrum of values can improve the quality of the test suite. E.g., a combinatorial approach for the selection of input values can be very effective [4, 6]; moreover, a solution that automates the selection of expected output values to check the desired behaviour could make use of a proper model that represents the desired system [7].

For generating the code of a test suite, many tools have been proposed. Some tools focus on the random generation of a sequence of calls and values that constitute a test case [5, 9, 18, 24]. Generally, random testing tools generate many test cases, and consequently their execution time increases significantly, then developers need some criteria and tools to select among these a smaller set of test cases [2, 14]. Moreover, generated test cases can often carry several test smells, lowering the effectiveness and quality of the test suite [20]. Test cases that have been generated by a random approach can be improved by removing duplicate and redundant tests while keeping the same code coverage [15, 17, 22]. One of the most effective test generation tools is Evosuite [11, 27], which is search-based and uses evolutionary search to automatically generate test suites aiming at maximising code coverage [19]. Although Evosuite has greater coverage and accuracy than other generation tools, test generation takes considerable time, i.e. on two time budgets of 30 and 120 seconds, tests achieve 65.7% and 77.1% of mean line coverage respectively [27]. Moreover, a large amount of computational resources and time are needed to properly run it, otherwise several *OutOfMemoryException* crashes could occur, or the generated test suite exhibits low coverage [27]. The need to increase usability and readability of generated tests has been pointed out, as generated tests are less readable than manually implemented ones [12, 23].

This paper aims at automatically generating a set of test case templates for methods of an application that have not been previously tested. The template contains the method to be called, a JUnit assert, and as many input parameters as possible. Our proposed approach uses statistics on the most used JUnit asserts for each return type of a method, which have been gained by means of a static analysis on software repositories including test suites. The statistics on return types of methods have been collected for most of the standard Java library methods. Then, for an application that needs to have (more) test cases, our approach (and corresponding tool) generates a customised test case template for each method, selecting the assert statistically more relevant for the return type of the method to be called.

*Alessandro Midolo is the corresponding author.

ICACS 2023, October 19–21, 2023, Larissa, Greece

ACM ISBN 979-8-4007-0909-8/23/10.
https://doi.org/10.1145/3631908.3631926

2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE) | 978-1-7281-6975-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/WETICE49692.2020.00049

Automatic Generation of Effective Unit Tests based on Code Behaviour

Andrea Fornaia, Alessandro Midolo, Giuseppe Pappalardo, Emiliano Tramontana
Dipartimento di Matematica e Informatica, University of Catania, Italy
Email: surname@dmi.unict.it

***Abstract*—A large amount of test cases is very useful to check the correctness of a software system while it is developed. Often a considerable time is dedicated by human programmers to designing effective test cases. This paper proposes an approach for automatically generating test cases tailored to the characteristics of the code under test. For this, the classes of a software system to be tested are characterised by a static code analysis aiming at summarising and representing their behaviour. As test cases check the behaviour of code, classes that exhibit a close behaviour may be checked using similar test cases. Therefore, in the approach proposed, for classes having a comparable behaviour, test cases are generated by taking as a template the test cases available for one of the classes among the similar ones. The approach has been assessed on a few open source projects and has proved to be viable for generating applicable and effective test cases for the classes.**

***Index Terms*—test case generation, static code analysis, test templating, verification**

I. Introduction

Producing test cases is an effective way to check the correctness of a software system, and to check that evolutionary changes aiming at improving functionalities are not introducing defects to previously correct code [11], [26]. However, developing tests is a time consuming activity. When designing tests, a developer has to take into account the behaviour of the component under test to determine the set of inputs and expected output, which are essential to implement a test case. Moreover, during implementation some boilerplate code needs to be added to give the needed context to the test case.

The existing literature on tests suggests several approaches for assisting the work of developers. Since one of the tasks of the developers is the selection of input values to be given to a method, combinatorial approaches for input values can be very effective and many tools have been implemented to find values among given validity ranges, as well as outside validity ranges [3], [25]. Another task developers have to perform is implementing methods calls that check the behaviour of a class, assistance for it has been given by tools that e.g. randomly generate a sequence of calls [7], [17], [23]. Moreover, for the task of finding expected output values to check against the resulting execution behaviour, often the solution is building a model of the system [4]. Other approaches for producing tests and make them robust include the generation of code variations to make sure that tests can find the erroneous behaviour [12]. Moreover, some approaches have been proposed to automatically generate code that passes all tests [10].

Most of the approaches aim at having and executing as many test cases as possible, which is worthy for checking a large amount of execution scenarios. However, given the large number of possible input data and execution paths inside the software system under test, execution could take an amount of time larger than the time frame available to have a timely feedback. This is mainly relevant for agile practices, which prescribe both as much tests as possible, as well as developing components, integrating and testing the overall system several times a day, to ensure minimal design and correct execution [1], [8]. For this, during development, test execution time is curbed, and test cases to be executed have to be selected among available ones [13], [15], [21].

This paper aims at automatically generating test cases tailored to the behaviour of the class under test. Our approach provides a class with a test by using static analysis to determine its behaviour, then such a behaviour is checked against the behaviour gathered for other classes, finally tests are generated starting from previously known tests for classes having a similar behaviour. Since generated tests are tailored to the code to be tested, they are effective for code coverage and for finding bugs. We have used our approach to generate tests for several software systems, whose source code is available. According to our experiments, the tests we have generated manage to extend the amount of code coverage significantly, both by executing new paths within classes that had some test cases, and by generating tests for classes that had not been previously tested.

The rest of the paper is organised as follows. Section II describes how we perform the analysis of classes to gather their behaviour. Section III reports our approach for generating tests according to the knowledge on the behaviour of classes and other tests used as samples. Section IV shows the results of the analysis of several software systems. Section V compares our approach with relevant related works. Finally, conclusions are expressed in Section VI.

II. Analysis of Software Systems

Generally, the developer creates a test case once he has gathered some knowledge on the expected behaviour of the class to be tested. Then, for each method of the class, he determines a set of inputs, among valid or invalid ranges for the needed parameters, and an expected output to be compared with the output provided by the method execution.

2641-8169/20/$31.00 ©2020 IEEE
DOI 10.1109/WETICE49692.2020.00049

Università di Catania

Uni ct MATEMATICA E INFORMATICA

The End 

Alessandro Midolo, PhD

Dipartimento di Matematica e Informatica

Università di Catania

alessandro.midolo@unict.it

https://www.dmi.unict.it/amidolo/

A.A. 2024/2025