UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

Specification-based testing

Azzurra Ragone

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 | Fax: +39.080.5442536
serlab.di.uniba.it

What is Specification-based Testing?

Specification-based testing
OR
Functional testing
OR
Black-box testing

Specification-based testing

Specification-based testing is guided by ***Requirements*** (*business rules* that the software implements and that we need to validate): you **derive tests from the requirements**

Functionality: what the software need to DO or NOT to DO

Requirements: ex. Agile user stories, UML use cases, plain text, etc.

Usually this is the first technique to go for in ST

SubstringsBetween example

Test method substrings- Between(), inspired by the Apache Commons Lang library (http://mng.bz/nYR5).

Method: `substringsBetween`

Searches a string for substrings delimited by a start and end tag, returning all matching substrings in an array.

SubstringsBetween example

```
public static String[] substringsBetween(final String str, final String open,
final String close)
```

INPUT:

str—The string containing the substrings. Null returns `null`; an empty string returns another empty string.

open—The string identifying the start of the substring. An empty string returns null.

close—The string identifying the end of the substring. An empty string returns null.

OUTPUT:

The program returns a string array of substrings, or `null` if there is no match.

SubstringsBetween example

Example:
str = “**a**x**ca**y**ca**x**c**”
open = “a”
close = “c”

What is the result?

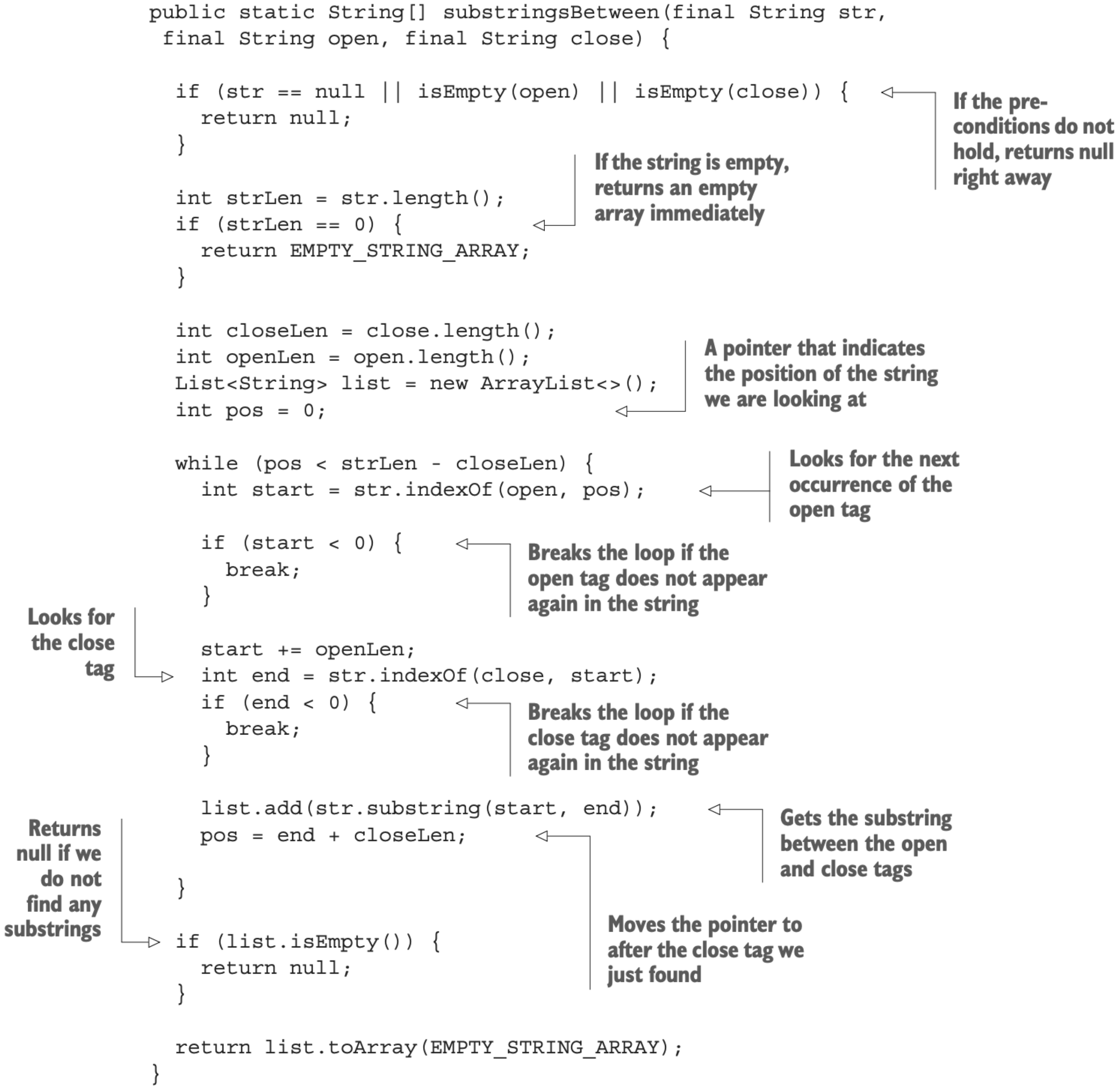
SubstringsBetween example

Example:
str = “**a**x**ca**y**ca**x**c**”
open = “a”
close = “c”

Output: array [“x”, “y”, “x”].

The “a<something>c” substring appears three times in the original string

Software Engineering Research LABoratory



```
public static String[] substringsBetween(final String str,
 final String open, final String close) {

  if (str == null || isEmpty(open) || isEmpty(close)) {
    return null;
  }

  int strLen = str.length();
  if (strLen == 0) {
    return EMPTY_STRING_ARRAY;
  }

  int closeLen = close.length();
  int openLen = open.length();
  List<String> list = new ArrayList<>();
  int pos = 0;

  while (pos < strLen - closeLen) {
    int start = str.indexOf(open, pos);

    if (start < 0) {
      break;
    }

    start += openLen;
    int end = str.indexOf(close, start);
    if (end < 0) {
      break;
    }

    list.add(str.substring(start, end));
    pos = end + closeLen;

  }

  if (list.isEmpty()) {
    return null;
  }

  return list.toArray(EMPTY_STRING_ARRAY);
}
```

If the preconditions do not hold, returns null right away

If the string is empty, returns an empty array immediately

A pointer that indicates the position of the string we are looking at

Looks for the next occurrence of the open tag

Breaks the loop if the open tag does not appear again in the string

Looks for the close tag

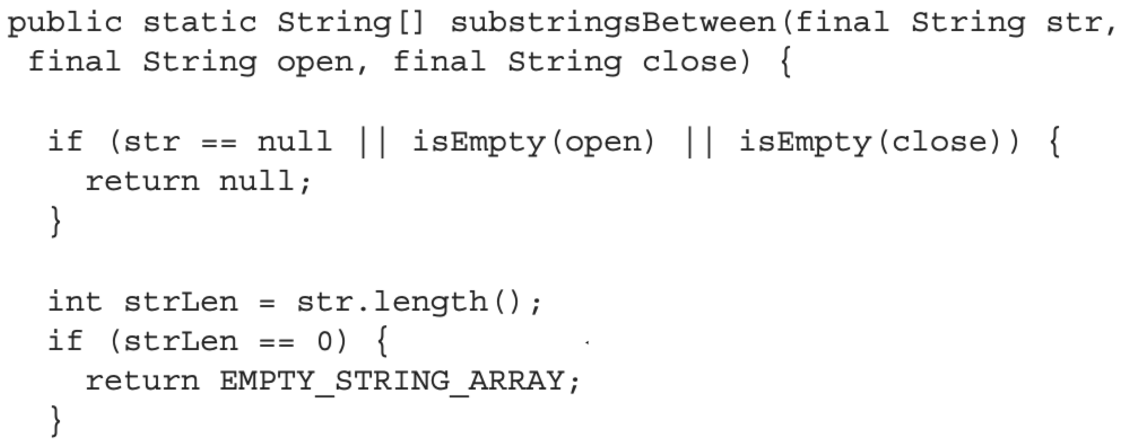
Breaks the loop if the close tag does not appear again in the string

Gets the substring between the open and close tags

Moves the pointer to after the close tag we just found

Returns null if we do not find any substrings

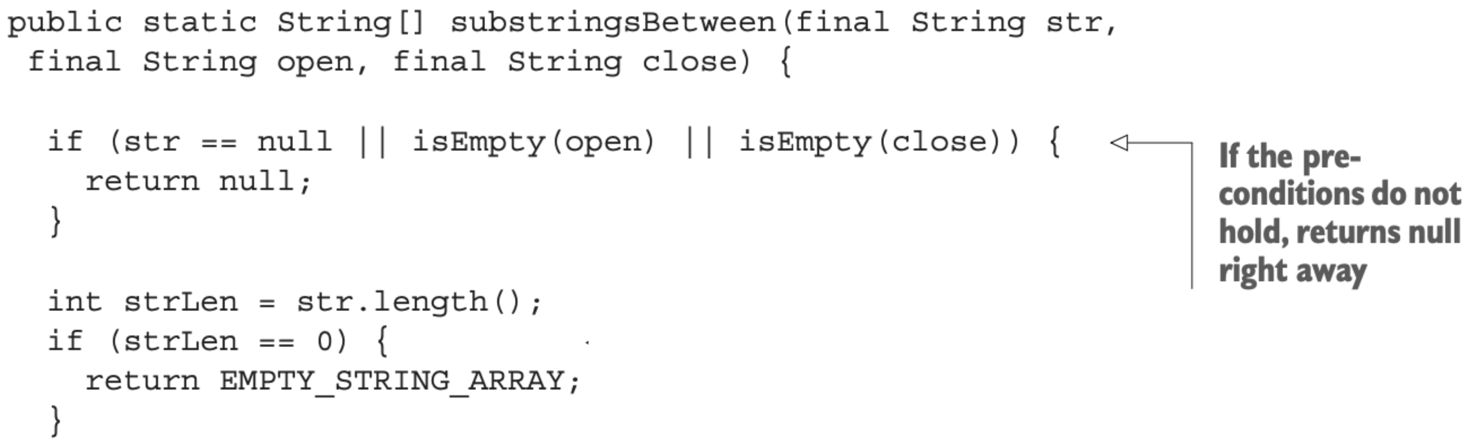
substringsBetween method



```
public static String[] substringsBetween(final String str,
 final String open, final String close) {

  if (str == null || isEmpty(open) || isEmpty(close)) {
    return null;
  }

  int strLen = str.length();
  if (strLen == 0) {
    return EMPTY_STRING_ARRAY;
  }
```

Software Engineering Research LABoratory

substringsBetween method

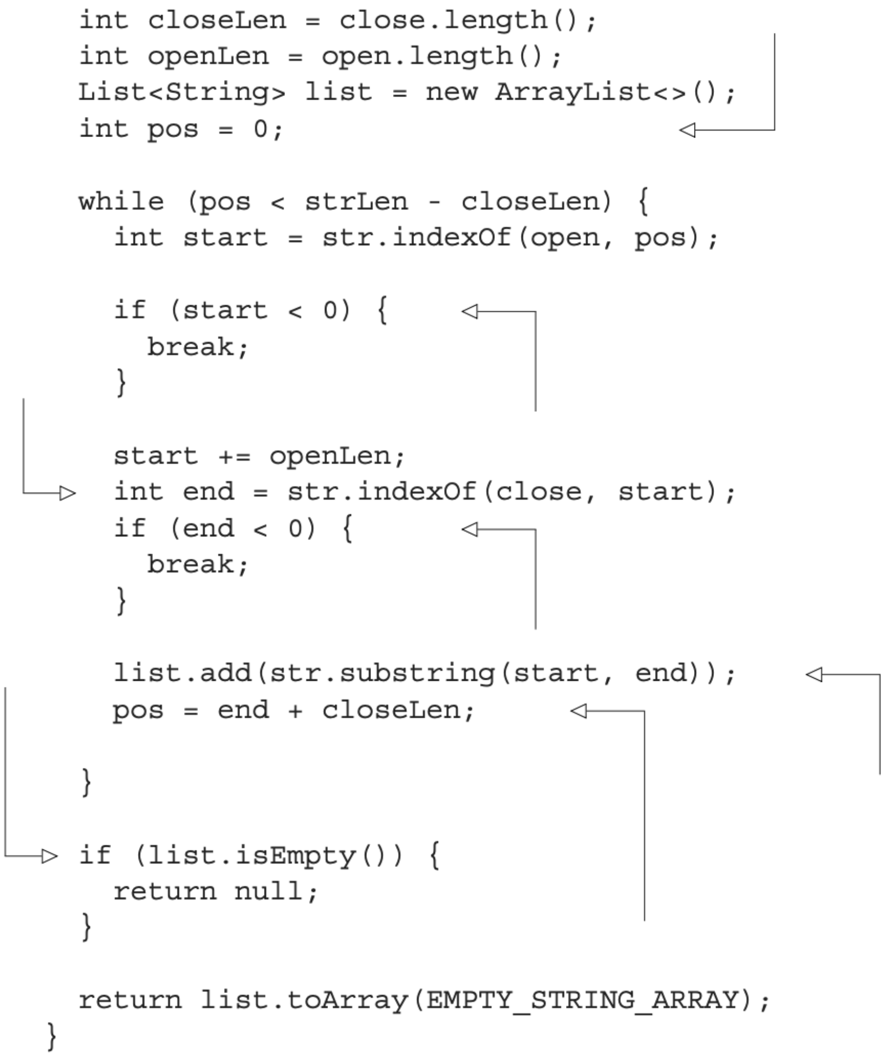


```
public static String[] substringsBetween(final String str,
 final String open, final String close) {

  if (str == null || isEmpty(open) || isEmpty(close)) {
    return null;
  }

  int strLen = str.length();
  if (strLen == 0) {
    return EMPTY_STRING_ARRAY;
  }
```

If the pre-conditions do not hold, returns null right away

substringsBetween method

axcaycazc

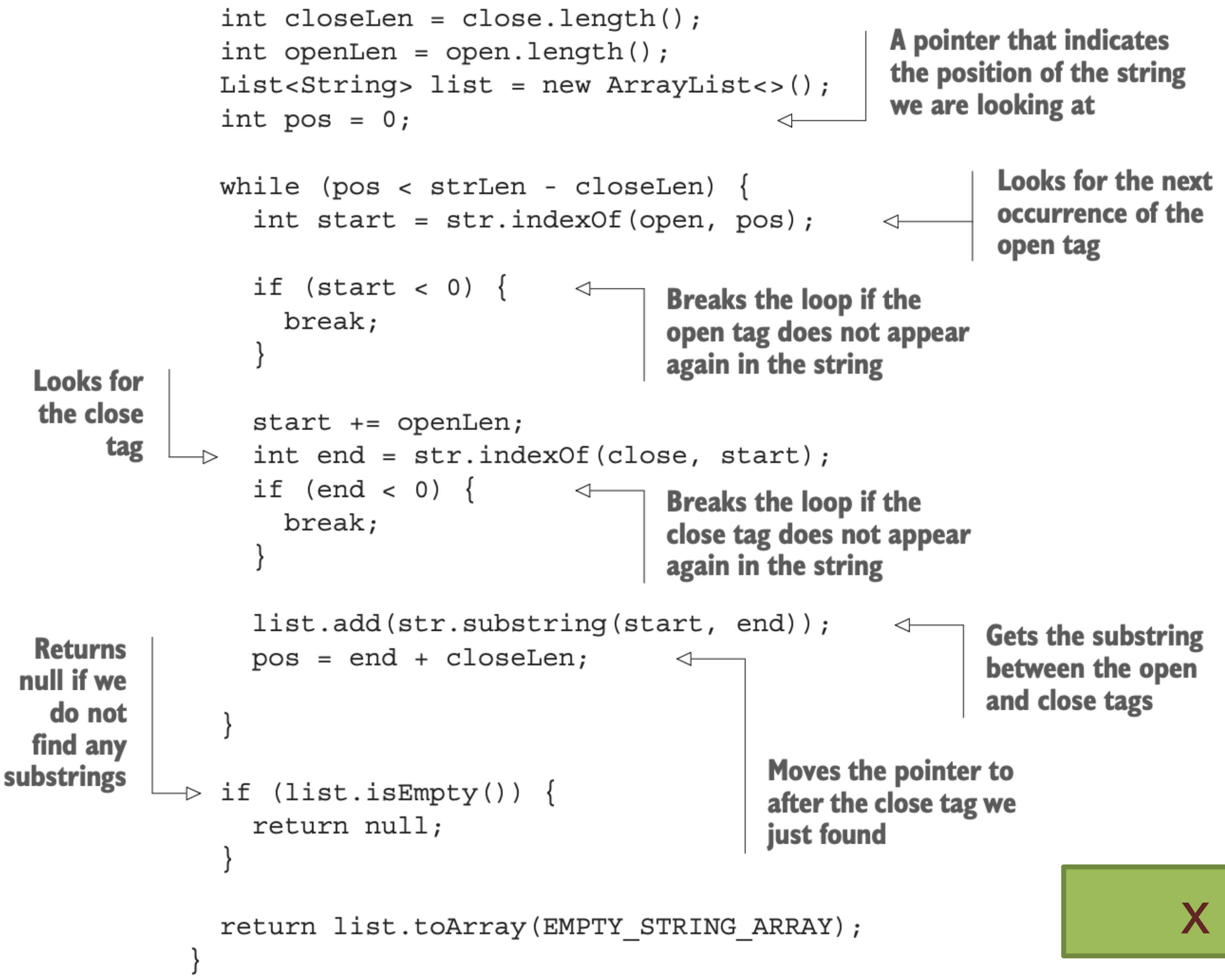


```
  int closeLen = close.length();
  int openLen = open.length();
  List<String> list = new ArrayList<>();
  int pos = 0;

  while (pos < strLen - closeLen) {
    int start = str.indexOf(open, pos);

    if (start < 0) {
      break;
    }

    start += openLen;
    int end = str.indexOf(close, start);
    if (end < 0) {
      break;
    }

    list.add(str.substring(start, end));
    pos = end + closeLen;

  }

  if (list.isEmpty()) {
    return null;
  }

  return list.toArray(EMPTY_STRING_ARRAY);
}
```

x , y, z

substringsBetween method

axcaycazc



```
  int closeLen = close.length();
  int openLen = open.length();
  List<String> list = new ArrayList<>();
  int pos = 0;

  while (pos < strLen - closeLen) {
    int start = str.indexOf(open, pos);

    if (start < 0) {
      break;
    }

    start += openLen;
    int end = str.indexOf(close, start);
    if (end < 0) {
      break;
    }

    list.add(str.substring(start, end));
    pos = end + closeLen;

  }

  if (list.isEmpty()) {
    return null;
  }

  return list.toArray(EMPTY_STRING_ARRAY);
}
```

A pointer that indicates the position of the string we are looking at

Looks for the next occurrence of the open tag

Breaks the loop if the open tag does not appear again in the string

Looks for the close tag

Breaks the loop if the close tag does not appear again in the string

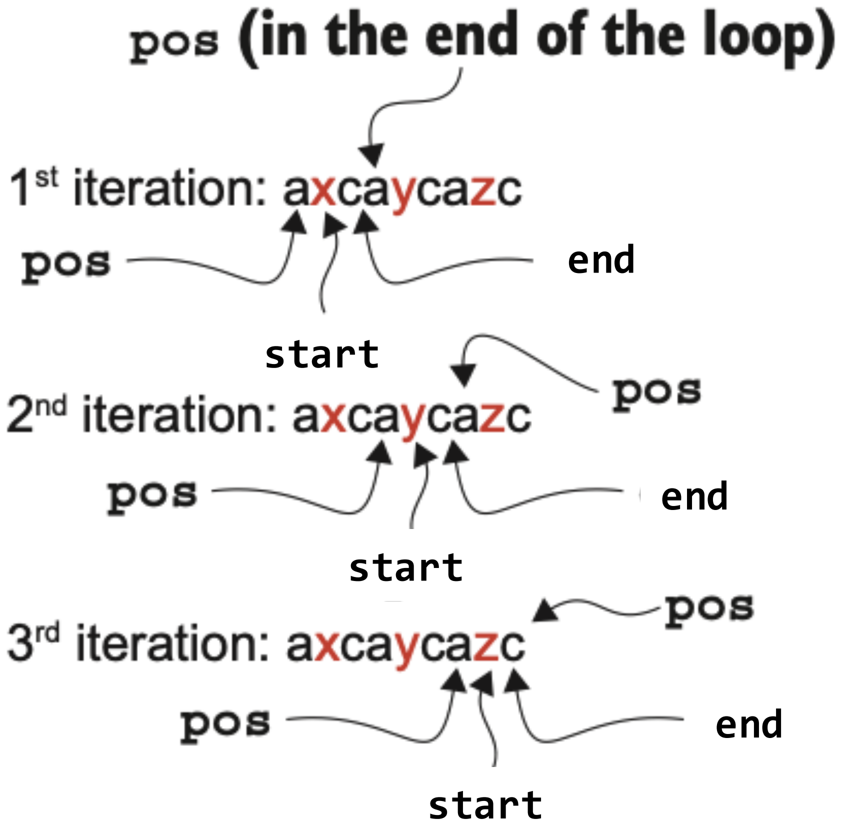
Gets the substring between the open and close tags

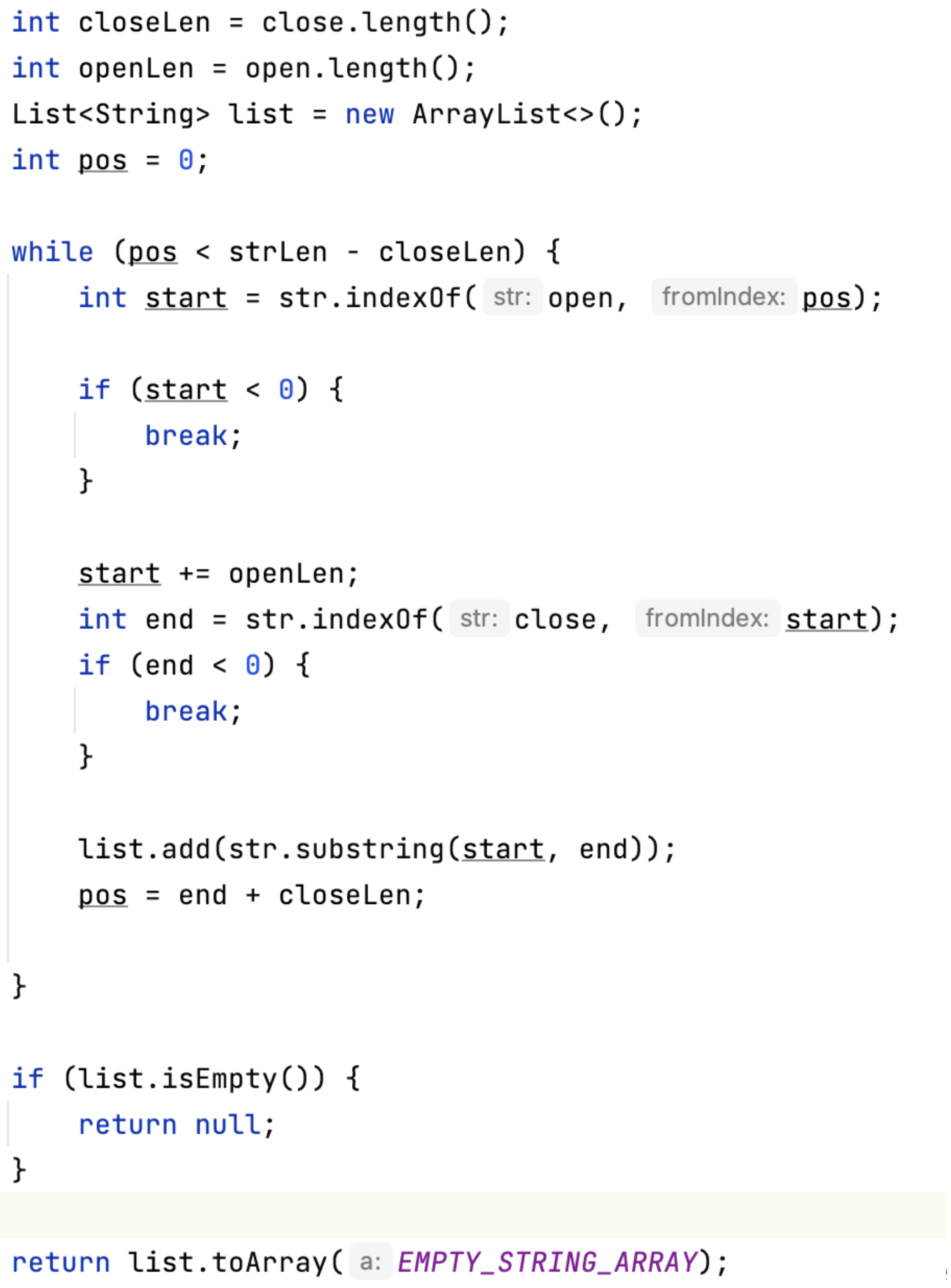
Moves the pointer to after the close tag we just found

Returns null if we do not find any substrings

x , y, z

substringsBetween method





```
int closeLen = close.length();
int openLen = open.length();
List<String> list = new ArrayList<>();
int pos = 0;

while (pos < strLen - closeLen) {
    int start = str.indexOf( str: open,  fromIndex: pos);

    if (start < 0) {
        break;
    }

    start += openLen;
    int end = str.indexOf( str: close,  fromIndex: start);
    if (end < 0) {
        break;
    }

    list.add(str.substring(start, end));
    pos = end + closeLen;

}

if (list.isEmpty()) {
    return null;
}


return list.toArray( a: EMPTY_STRING_ARRAY);
```

1st iteration:
pos=0, start=1, end=2

2nd iteration:
pos=3, start=4, end=5

3rd iteration=
pos=6, start=7, end=8

Testing workflow for Specification-based testing

1. Understanding the requirements (what the program must do, inputs, and outputs)
2. Explore what the program does for various inputs
3. Explore inputs, outputs and identify partitions
4. Identify boundary cases (aka corner cases)
5. Devise test cases (output Test Plan)
6. Automate test cases (output: Junit code)
7. Augment the test suite with creativity and experience

P.S. It is not possible to test all possible combinations of inputs (sometimes it is not even convenient or effective), exhaustive testing should be replaced by a pragmatic approach.

1) Understanding the requirements

Write down what the program **should do**, and how **inputs** are converted in expected **outputs**

1) Understanding the requirements

Write down what the program **should do**, and how **inputs** are converted in expected **outputs**

*1) The **goal** of this method is to collect all substrings in a string that are delimited by an open tag and a close tag (the user provides these)*

*2) The program receives three parameters as **input**:*

a) str, which represents the string from which the program will extract sub- strings

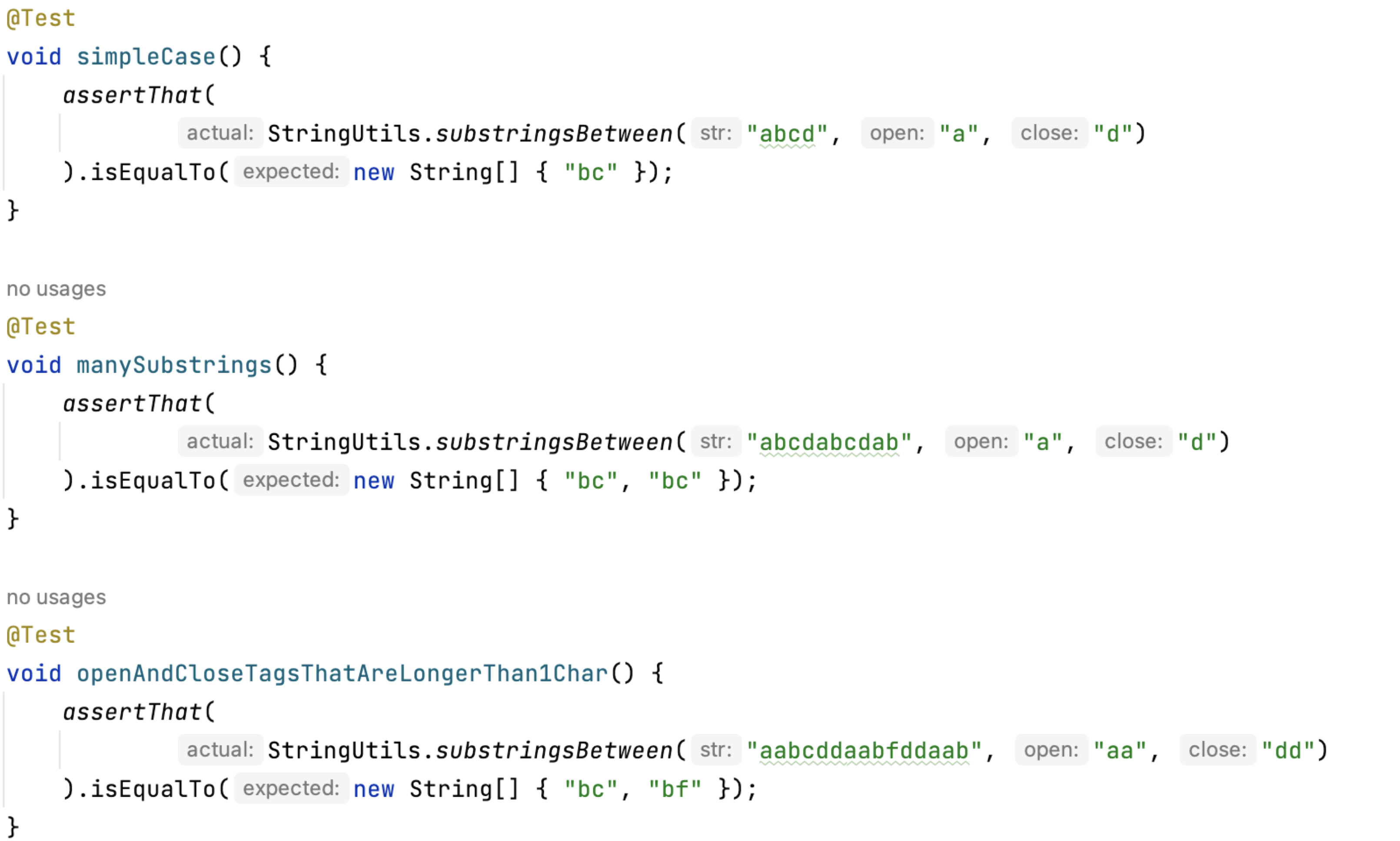
b) The open tag, which indicates the start of a substring

c) The close tag, which indicates the end of the substring

*3) The program returns an array composed of all the substrings found by the program (**output**).*

2) Explore what the program does for various inputs

This step is especially important if you did not write the code and you want to have a clear mental model of how the program should work



```
@Test
void simpleCase() {
    assertThat(
            actual: StringUtils.substringsBetween( str: "abcd",  open: "a",  close: "d")
    ).isEqualTo( expected: new String[] { "bc" });
}

no usages
@Test
void manySubstrings() {
    assertThat(
            actual: StringUtils.substringsBetween( str: "abcdabcdab",  open: "a",  close: "d")
    ).isEqualTo( expected: new String[] { "bc", "bc" });
}

no usages
@Test
void openAndCloseTagsThatAreLongerThan1Char() {
    assertThat(
            actual: StringUtils.substringsBetween( str: "aabcddaabfddaab",  open: "aa",  close: "dd")
    ).isEqualTo( expected: new String[] { "bc", "bf" });
}
```

3) Explore inputs, outputs and identify partitions

The number of possible inputs and outputs is nearly infinite

Some sets of inputs make the program behave the same way, regardless of the precise input value.

Test one single case that represents the entire class of inputs

Explore:

a - **Individual inputs** (classes of inputs)
b - **Combinations** of **inputs**
c - Classes of (expected) **outputs**

Individual Inputs

`str` parameter:

1 - Null string
2 - Empty string
3 - String of length 1
4 - String of length > 1
(any string)

`open` parameter:

1 - Null string
2 - Empty string
3 - String of length 1
4 - String of length > 1
(any string)

`close` parameter:

1 - Null string
2 - Empty string
3 - String of length 1
4 - String of length > 1
(any string)

Combinations of Inputs

`(str, open, close)` parameters:

1 - ***str*** contains neither the ***open*** nor the ***close*** tag.
2 - ***str*** contains the ***open*** tag but not the ***close*** tag.
3 - ***str*** contains the ***close*** tag but not the ***open*** tag.
4 - ***str*** contains both the ***open*** and ***close*** tags.
5 - ***str*** contains both the ***open*** and ***close*** tags multiple times.

Software Engineering Research LABoratory

Classes of (expected) outputs

`Array of strings` (output):

1 - Null array
2 - Empty array
3 - Single item
4 - Multiple items

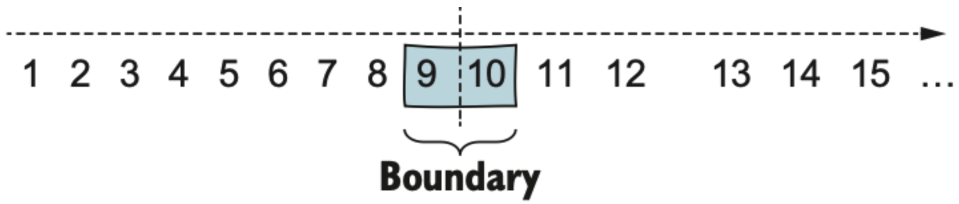
`Each individual string` (output):
1 - Empty
2 - Single character
3 - Multiple character

4) Identify boundary cases (aka corner cases)

Bugs love boundaries!

Boundary testing -> the program should work correctly when input are near the boundaries

Ex. If (a>=10)

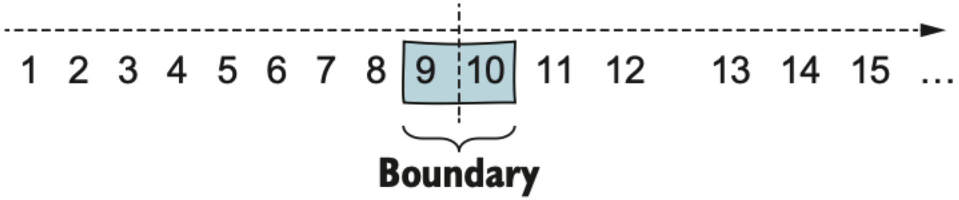


4) Identify boundary cases (aka corner cases)

ON/OFF POINTS

on point: the point that is on the boundary
off point: the point closest to the boundary that belongs to the other partition

Ex. If (a>=10)



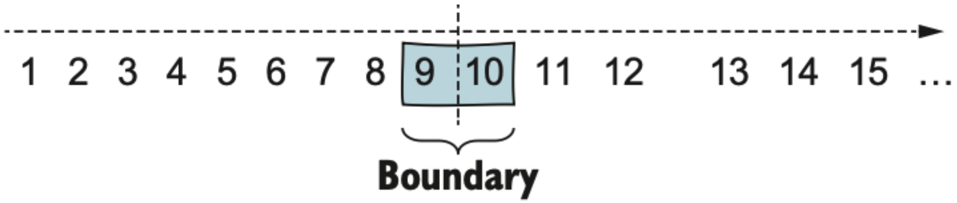
4) Identify boundary cases (aka corner cases)

ON/OFF POINTS

on point: the point that is on the boundary
off point: the point closest to the boundary that belongs to the other partition

Ex. If (a>=10)

10 is the on point
9 is the off point



Software Engineering Research LABoratory

4) Identify boundary cases (aka corner cases)

IN/OUT POINTS

in point: the points that make the condition true
out point: the points that make the condition false

Ex. If (a=10)

4) Identify boundary cases (aka corner cases)

IN/OUT POINTS

Ex. If (a=10)

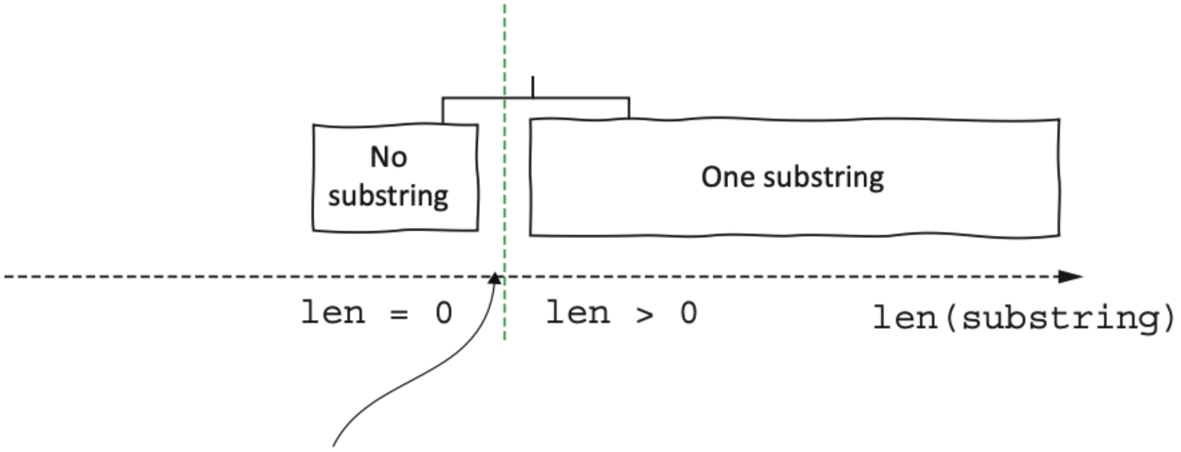
10 is the in point
6, 8, 12, 52 are the out points

4) Identify boundary cases (aka corner cases)

In our example we have two tests, one for each side of the boundary:

1 `str` contains both `open` and `close` tags, with *no* characters between them
2 `str` contains both `open` and `close` tags, with characters between them

(We discard test num 2, as other tests already cover this situation)



5) Devise test cases

`str` parameter:

1 - Null string
2 - Empty string
3 - String of length 1
4 - String of length > 1 (any string)

`open` parameter:

1 - Null string
2 - Empty string
3 - String of length 1
4 - String of length > 1 (any string)

`close` parameter:

1 - Null string
2 - Empty string
3 - String of length 1
4 - String of length > 1 (any string)

`(str, open, close)` parameters:

1 - ***`str`*** contains neither the ***`open`*** nor the ***`close`*** tag.
2 - ***`str`*** contains the ***`open`*** tag but not the ***`close`*** tag.
3 - ***`str`*** contains the ***`close`*** tag but not the ***`open`*** tag.
4 - ***`str`*** contains both the ***`open`*** and ***`close`*** tags.
5 - ***`str`*** contains both the ***`open`*** and ***`close`*** tags multiple times.

If you combine all possible inputs: 4 × 4 × 4 × 5 = 320 tests.
Is it useful to have all these tests?

5) Devise test cases

Pragmatically decide which partitions should be combined with others and which should not

a) Test exceptional cases only once and do not combine them (e.g. null, empty):

T1: `str` is null.
T2: `str` is empty.

T3: `open` is null.
T4: `open` is empty.

T5: `close` is null.
T6: `close` is empty.

5) Devise test cases

b) For *string of length 1* we may test just these four cases:

T7: The single character in `str` matches the `open` tag.

T8: The single character in `str` matches the `close` tag.

T9: The single character in `str` does not match either the `open` or the `close` tag.

T10: The single character in `str` matches both the `open` and `close` tags.

5) Devise test cases

c) A first combination of inputs:

`str` length > 1
`open` length = 1
`close` length = 1

T11: `str` does not contain either the `open` or the `close` tag.

T12: `str` contains the `open` tag but does not contain the `close` tag.

T13: `str` contains the `close` tag but does not contain the `open` tag.

T14: `str` contains both the `open` and `close` tags.

T15: `str` contains both the `open` and `close` tags multiple times.

5) Devise test cases

d) A second combination of inputs:

`str` length > 1
`open` length > 1
`close` length > 1

T16: str does not contain either the open or the close tag.

T17: str contains the open tag but does not contain the close tag.

T18: str contains the close tag but does not contain the open tag.

T19: str contains both the open and close tags.

T20: str contains both the open and close tags multiple times.

5) Devise test cases

e) Boundary test

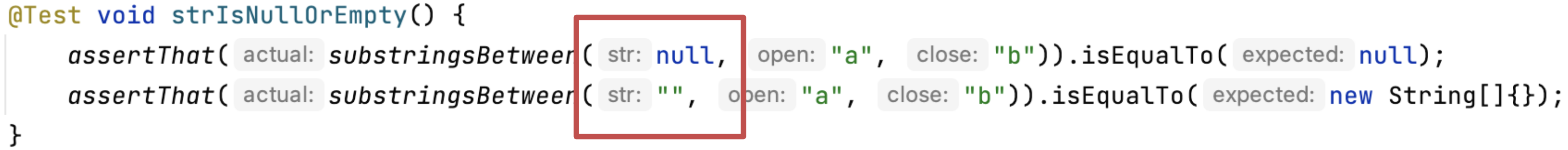
T21: str contains both the open and close tags with no characters between them.

Final note:
we end up with ***21*** *tests rather than* ***320***!

6) Automate test cases

T1: `str` is null.

T2: `str` is empty.

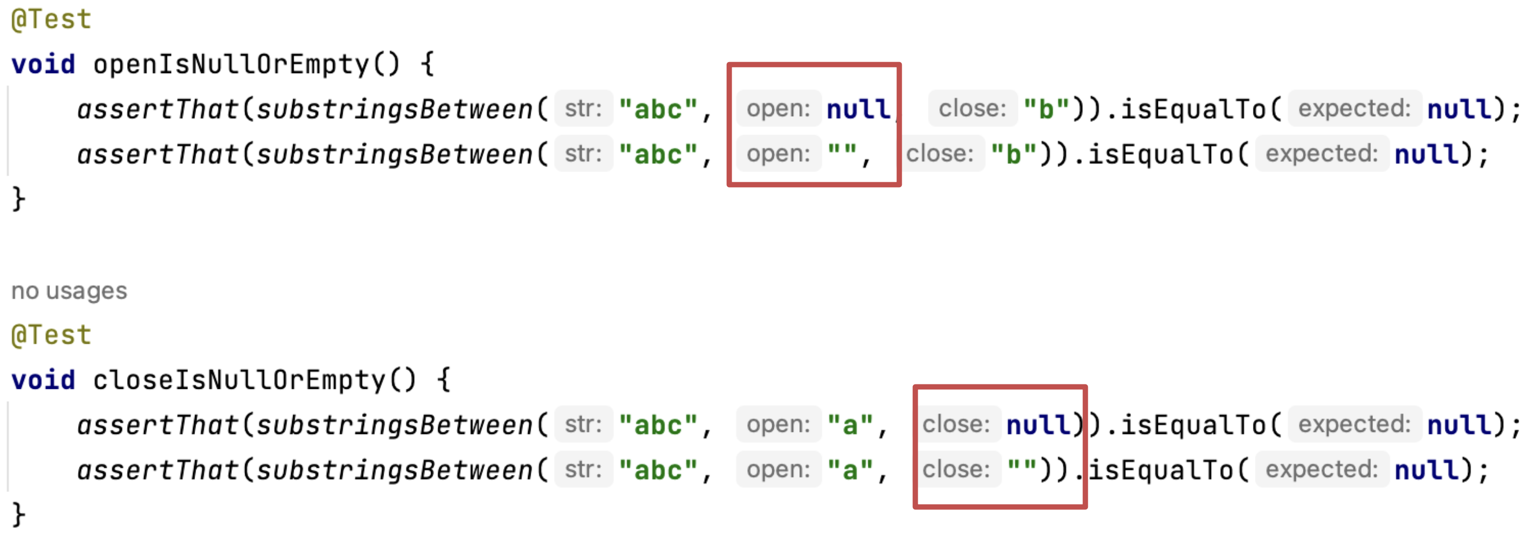


```
@Test void strIsNullOrEmpty() {
    assertThat( actual: substringsBetween( str: null, open: "a", close: "b")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "", open: "a", close: "b")).isEqualTo( expected: new String[]{});
}
```

6) Automate test cases

T3: `open` is null.
T4: `open` is empty.

T5: `close` is null.
T6: `close` is empty.



```
@Test
void openIsNullOrEmpty() {
    assertThat(substringsBetween( str: "abc", open: null, close: "b")).isEqualTo( expected: null);
    assertThat(substringsBetween( str: "abc", open: "", close: "b")).isEqualTo( expected: null);
}

no usages
@Test
void closeIsNullOrEmpty() {
    assertThat(substringsBetween( str: "abc", open: "a", close: null)).isEqualTo( expected: null);
    assertThat(substringsBetween( str: "abc", open: "a", close: "")).isEqualTo( expected: null);
}
```

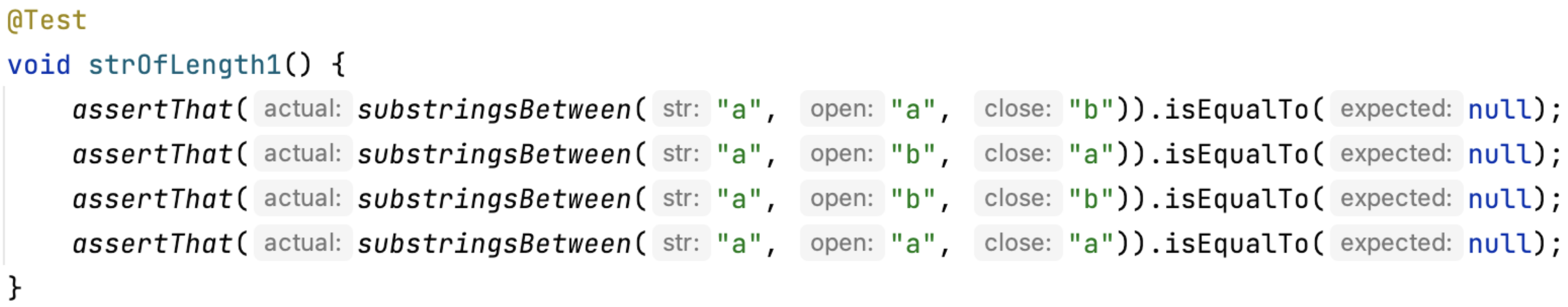
6) Automate test cases

T7: The single character in `str` matches the `open` tag.

T8: The single character in `str` matches the `close` tag.

T9: The single character in `str` does not match either the `open` or the `close` tag.

T10: The single character in `str` matches both the `open` and `close` tags.



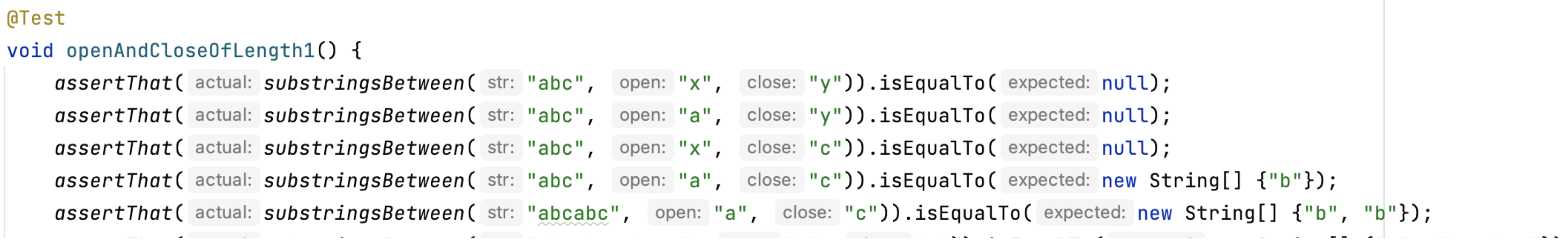
```
@Test
void strOfLength1() {
    assertThat( actual: substringsBetween( str: "a",  open: "a",  close: "b")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "a",  open: "b",  close: "a")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "a",  open: "b",  close: "b")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "a",  open: "a",  close: "a")).isEqualTo( expected: null);
}
```

Software Engineering Research LABoratory

6) Automate test cases

`str` length > 1, `open` length = 1, `close` length = 1

T11: `str` does not contain either the `open` or the `close` tag.
T12: `str` contains the `open` tag but does not contain the `close` tag.
T13: `str` contains the `close` tag but does not contain the `open` tag.
T14: `str` contains both the `open` and `close` tags.
T15: `str` contains both the `open` and `close` tags multiple times.

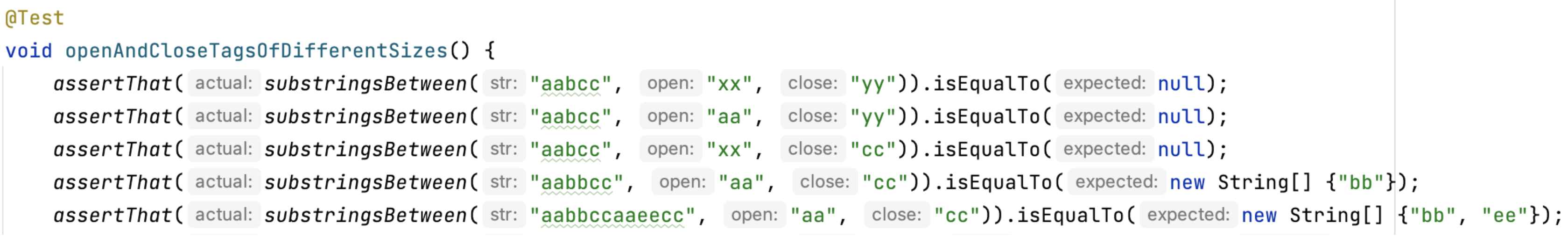


```
@Test
void openAndCloseOfLength1() {
    assertThat( actual: substringsBetween( str: "abc", open: "x", close: "y")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "abc", open: "a", close: "y")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "abc", open: "x", close: "c")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "abc", open: "a", close: "c")).isEqualTo( expected: new String[] {"b"});
    assertThat( actual: substringsBetween( str: "abcabc", open: "a", close: "c")).isEqualTo( expected: new String[] {"b", "b"});
```

6) Automate test cases

`str` length > 1, `open` length > 1, `close` length > 1

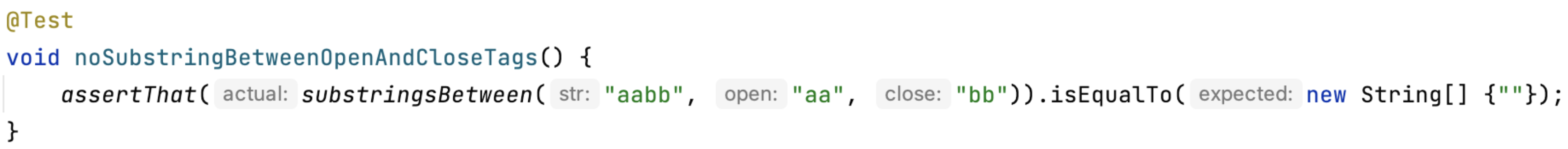
T16: `str` does not contain either the `open` or the `close` tag.
T17: `str` contains the `open` tag but does not contain the `close` tag.
T18: `str` contains the `close` tag but does not contain the `open` tag.
T19: `str` contains both the `open` and `close` tags.
T20: `str` contains both the `open` and `close` tags multiple times.



```
@Test
void openAndCloseTagsOfDifferentSizes() {
    assertThat( actual: substringsBetween( str: "aabcc",  open: "xx",  close: "yy")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "aabcc",  open: "aa",  close: "yy")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "aabcc",  open: "xx",  close: "cc")).isEqualTo( expected: null);
    assertThat( actual: substringsBetween( str: "aabbcc",  open: "aa",  close: "cc")).isEqualTo( expected: new String[] {"bb"});
    assertThat( actual: substringsBetween( str: "aabbccaaeecc",  open: "aa",  close: "cc")).isEqualTo( expected: new String[] {"bb", "ee"});
```

6) Automate test cases

T21: str contains both the open and close tags with no characters between them.



```
@Test
void noSubstringBetweenOpenAndCloseTags() {
    assertThat( actual: substringsBetween( str: "aabb",  open: "aa",  close: "bb")).isEqualTo( expected: new String[] {""});
}
```

<u>Final note</u>: decide how to group test in test method (refer to partitions)

7) Augment the test suite with creativity and experience

In test you should always consider Variations: testing special characters is always a good idea. What about white space?

T22: "abcabyt byrc " -> Space in the string, open=a, close=c

T23: "a abb ddc ca abbcc" -> Space in open & close tags open=‘a a’ , close= ‘c c’

```
T22: assertThat(substringsBetween("abcabyt byrc", "a", "c"))
                              .isEqualTo(new String[] {"b", "byt byr"});
```

```
T23: assertThat(substringsBetween("a abb ddc ca abbcc", "a a", "c c")).
                              isEqualTo(new String[] {"bb dd"});
```

Specification-based testing: recap

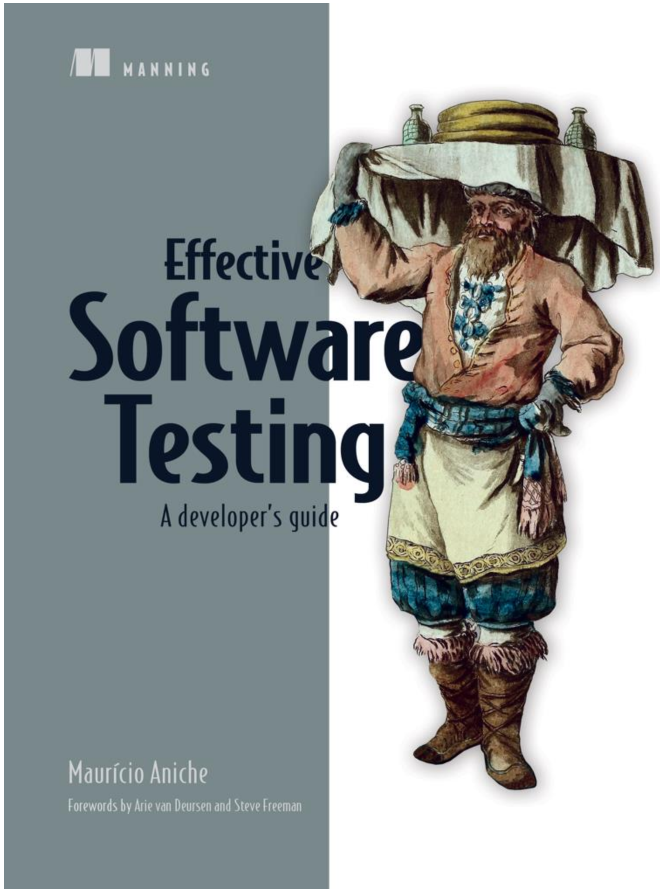
7 steps to create the test suite:

1. Understanding the requirements (what the program must do, inputs, and outputs)
2. Explore what the program does for various inputs
3. Explore inputs, outputs and identify partitions
4. Identify boundary cases (aka corner cases)
5. Devise test cases
6. Automate test cases
7. Augment the test suite with creativity and experience

Reference book:

Effective Software Testing. A developer's guide. Mauricio Aniche. Ed. Manning. (**Chapter 2**)

Use the "au35ani" discount code for a 35% off the price.



References:

- AssertJ - fluent assertions java library: https://assertj.github.io/doc/
- Assertj core javadoc: https://www.javadoc.io/doc/org.assertj/assertj-core/latest/index.html
- Assertj core javadoc: Assertions: https://www.javadoc.io/doc/org.assertj/assertj-core/latest/org/assertj/core/api/Assertions.html
- Introduction to AssertJ: https://www.baeldung.com/introduction-to-assertj

SERLAB

Software Engineering Research

Azzurra Ragone

Department of Computer Science- Floor VI – Room 616
Email: azzurra.ragone@uniba.it