Software Quality and Testing Labs

@ParameterizedTest

```java
package delft;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.*;

class RomanNumeralTests {

    private final RomanNumeral numeral = new RomanNumeral();

    @ParameterizedTest
    @MethodSource("generator")
    void getNumber(String s, int expected) {
        assertEquals(expected, numeral.convert(s));
    }

    private static Stream<Arguments> generator() {
        return Stream.of(
            Arguments.of("", 0),
            Arguments.of("X",10),
            Arguments.of("XI",11),
            Arguments.of("IX",9));
    }
}
```

@BeforeEach

```java
import …

class BeforeEachNotation {

    private RomanNumeral roman;

    @BeforeEach
    void setup() {
        roman = new RomanNumeral();
    } // @Tests...
```

@CsvSource

```
    @ParameterizedTest(name = "small={0}, big={1}, total={2}, result={3}")
    @CsvSource({
      // The total is higher than the amount of small and big bars.
      "1,1,5,0", "1,1,6,1", "1,1,7,-1", "1,1,8,-1",
      // No need for small bars.
      "4,0,10,-1", "4,1,10,-1", "5,2,10,0", "5,3,10,0",
      // Need for big and small bars.
      "0,3,17,-1", "1,3,17,-1", "2,3,17,2", "3,3,17,2",
      "0,3,12,-1", "1,3,12,-1", "2,3,12,2", "3,3,12,2",
      // Only small bars.
      "4,2,3,3", "3,2,3,3", "2,2,3,-1", "1,2,3,-1"
    })
    void boundaries(int small, int big, int total, int expectedResult) {
        int result = new ChocolateBars().calculate(small, big, total);
        Assertions.assertEquals(expectedResult, result);
    }
```

Category Partition Method

1. Identify the parameters

2. Derive characteristics of each parameter. For example, an int year should be a positive integer number between 0 and infinite.

Some of these characteristics can be found directly in the specification of the program.

Others might not be found from specifications. For example, an input cannot be null if the method does not handle that well.

3. Add constraints in order to minimize the test suite. (equivalence portioning or impossible scenarios)

4. Generate combinations of the input values. (Like a cartesian product)

5. Exceptional cases can be just tested once and thus no need to combine them into the cartesian product (the constraints should be applied after the cartesian product).

```
/**
 * <p>Repeat a String {@code repeat} times to form a
 * new String.</p>
 *
 * @param str  the String to repeat, may be null
 * @param repeat  number of times to repeat str, negative treated as zero
 * @return a new String consisting of the original String repeated,
 *  {@code null} if null String input
 */
public String repeat(final String str, final int repeat) {
    // ...
}
```

What are the parameters of the method?

1. `String str` - the string that will be repeated
2. `int repeat` - the number of times to repeat the string

Derive the characteristics of each parameter.

1. Parameter `str`
   - `null` - output should be `null`
   - empty - output should be empty
   - non-empty - output should be the string repeated
2. Parameter `repeat`
   - negative - output should be empty
   - zero - output should be empty
   - positive - output should be the string repeated `repeat` times

null str null output precedes over repeat's negative and zero.

What are the constraints that you can use to minimize the test suite?

- `str` being `null`
- negative value for `repeat`
- empty string for `str` give exceptional behavior.

Combine the characteristics of the parameters to derive the test cases.

1. `str null`
   - `str`: `null`
   - `repeat`: 5
   - output: `null`
2. empty `str`
   - `str`: `""`
   - `repeat`: 4
   - output: `""`
3. `repeat` negative
   - `str`: `"hello"`
   - `repeat`: -2
   - output: `""`
4. `repeat` 0 with non-empty `str`
   - `str`: `"world"`
   - `repeat`: 0
   - output: `""`
5. positive `repeat` with non-empty `str`
   - `str`: `"ab"`
   - `repeat`: 3
   - output: `"abababab"`

Category Partition Method Unit Tests

```
package delft;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.Optional;
import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
class DelftStringUtilsTests {

    @ParameterizedTest
    @MethodSource("generator")
 void repeat(String testName, Optional<String> str, int repeat, Optional<String>
result) {

        DelftStringUtils util = new DelftStringUtils();
        assertEquals(result.isPresent() ? result.get() : null,
                util.repeat(str.isPresent() ? str.get() : null, repeat));
    }

    //source
    private static Stream<Arguments> generator() {
        return Stream.of(
Arguments.of("empty str", Optional.of(""), 4, Optional.of("")),
Arguments.of("nullstr", Optional.ofNullable(null), 5, Optional.ofNullable(null)),
Arguments.of("repeat negative", Optional.of("hello"), -2, Optional.of("")),
Arguments.of("0 non-empty str", Optional.of("world"), 0, Optional.of("")),
Arguments.of("+repeat non-empty str", Optional.of("ha"), 2, Optional.of("haha"))
        );
    }
}
```

Be careful with NullPointerException, therefore use Optional.ofNullable instead of Optional.of, do not forget package delft and the name of the classes.

Give a test suit for:

```
Given an integer `n`, return the string form of the number followed by `"!"`.
If the number is divisible by 3 use `"Fizz"` instead of the number,
and if the number is divisible by 5 use `"Buzz"` instead of the number,
and if the number is divisible by both 3 and 5, use `"FizzBuzz"`
```

$$n \to n! = 8,\ \ 3|n \to Fizz! = 6,\ \ 5|n \to Buzz! = 25,\ \ 3|n\ \wedge 5|\mathrm{n} \to \mathrm{FizzBuzz}! = 15$$

Give category/partitions for:

```
/**
* Puts the supplied value into the Map,
* mapped by the supplied key.
* If the key is already in the map, its
* value will be replaced by the new value.
* NOTE: Nulls are not accepted as keys;
*  a RuntimeException is thrown when key is null.
* @param key the key used to locate the value
* @param value the value to be stored in the HashMap
* @return the prior mapping of the key, or null if there was none.
  */
  public V put(K key, V value) {
  // implementation here
  }
```

1. Two parameters: key and value
2. The execution of the program does not depend on the value; it always inserts it into the map. We can define different characteristics of the key:
    - The key can already be present in the map or not.
    - The key can be null.
3. The requirements did not give a lot of parameters and/or characteristics, so we do not have to add constraints.
4. The combinations are each of the possibilities for the key with any value, as the programs execution does not depend on the value. We end up with three partitions:
    - New key
    - Existing key
    - null key

Which of the following statements is **true** about applying the category/partition method in the Java method below?

```
/**
* Puts the supplied value into the Map,
* mapped by the supplied key.
* If the key is already in the map, its
* value will be replaced by the new value.
*
* NOTE: Nulls are not accepted as keys;
*  a RuntimeException is thrown when key is null.
*
* @param key the key used to locate the value
* @param value the value to be stored in the HashMap
* @return the prior mapping of the key,
*  or null if there was none.
*/
public V put(K key, V value) {
   // implementation here
}
```

The number of tests generated by the category/partition method can grow quickly, as the chosen partitions for each category are later combined one-by-one. This is not a practical problem to the put() method because the number of categories and their partitions is small.

In an object-oriented language, besides using the method's input parameters to explore partitions, we should also consider the internal state of the object (i.e., the class's attributes), as it can also affect the behavior of the method.

The specification does not specify any details about the value input parameter, and thus, experience should be used to partition it, e.g., value being null and not null.

What actions could we take to reduce the number of combinations?

- **Pattern size**: empty, single character, many characters, longer than any line in the file.
- **Quoting**: pattern is quoted, pattern is not quoted, pattern is improperly quoted.

- **File name**: good file name, no file name with this name, omitted.
- **Occurrences in the file**: none, exactly one, more than one.
- **Occurrences in a single line, assuming line contains the pattern**: one, more than one.

---

1. We should treat file names 'no-filename with this name' and 'omitted' as exceptional, and thus, test them just once.
2. We should treat pattern size 'empty' as exceptional, and thus, test it just once.
3. We should constrain the options in the 'occurrences in a single line' category to happen only if 'occurrences in the file' are either exactly one or more than one. It does not make sense to have none occurrences in a file and one pattern in a line.
4. We should treat 'pattern is improperly quoted' as exceptional, and thus, test it just once.

What test cases should be created when taking both the partition of the input parameters and the internal state of the object into account?

```
/**
* Adds the specified element to this set if it
* is not already present.
* If this set already contains the element,
* the call leaves the set unchanged
* and returns false.
*
* If the specified element is NULL, the call leaves the
* set unchanged and returns false.
*
* If the set is full,
* the call leaves the set unchanged and return false.
* Use private method `isFull` to know whether the set is already full.
*
* @param e element to be added to this set
* @return true if this set did not already contain
*   the specified element
*/
public boolean add(E e) {
    // implementation here
}
```

Input parameter `e`:

- P1: Element not present in the set
- P2: Element already present in the set
- P3: NULL element.

Set is full? (`isFull` comes from the state of the class)

- isFull == true
- isFull == false

With the categories and partitions in hands, we can constrain the `isFull == true` and test it only once (i.e., without combining with other classes).

We then combine them all and end up with four tests:

- T1: `isFull` returns false, `e`: Element not present in the set
- T2: `isFull` returns false, `e`: Element present in the set
- T3: `isFull` returns false, `e`: Null element
- T4: `isFull` returns true, `e`: Element not present in the set

Boundary Testing

- On-point: the value we see in the condition (or parts of one (lazy)).
  - is not necessarily true (in-point)
- Off-point: The off-point is the value that is closest to the boundary and that flips the condition.
  - Inequalities have two off points.
- In-points: In-points are all the values that make the condition true.
- Out-points: Out-points are all the values that make the condition false.

CORRECT

- Conformance
  - Test what happens when your **input** is not in conformance with what is expected. i.e. string instead of int, not an email, etc.
- Ordering
  - Test different input order (i.e. sometimes the method only worked for sorted arrays)
- Range
  - Test what happens when we provide inputs that are outside of the expected range. (i.e. negative numbers for age)
- Reference (for OOP methods)
  - What it references outside its scope
  - What external dependencies it has
  - Whether it depends on the object being in a certain state
  - Any other conditions that must exist
- Existence
  - Does the system behave correctly when something that is expected to exist, does not? i.e. null pointer errors
- Cardinality
  - Test loops in different situations, such as when it actually performs zero iterations, one iterations, or many.
- Time
  - What happens if the system receives inputs that are not ordered in regards to date and time?
  - Timing of successive events
  - Does the system handle timeouts well?
  - Does the system handle concurrency well? (multiple computations are happening at the same time.)
  - Time formats and time zones

Domain testing

(Specification based testing) + Category partition method + Boundary Testing

1. We read the requirement
2. We identify the input and output variables in play, together with their types, and their ranges.
3. We identify the dependencies (or independence) among input variables, and how input variables influence the output variable.
4. We perform equivalent class analysis (valid and invalid classes).
5. We explore the boundaries of these classes.
6. We think of a strategy to derive test cases, focusing on minimizing the costs while maximizing fault detection capability.
7. We generate a set of test cases that should be executed against the system under test.

```
/**
 * <p>Converts all the delimiter separated words in a String into camelCase,
 * that is each word is made up of a titlecase character and then a series of
 * lowercase characters.</p>
 *
 * <p>The delimiters represent a set of characters understood to separate
words.
 * The first non-delimiter character after a delimiter will be capitalized.
The first String
 * character may or may not be capitalized and it's determined by the user
input for capitalizeFirstLetter
 * variable.</p>
 *
 * <p>A <code>null</code> input String returns <code>null</code>.
 * Capitalization uses the Unicode title case, normally equivalent to
 * upper case and cannot perform locale-sensitive mappings.</p>
 *
 * @param str                   the String to be converted to camelCase, may
be null
 * @param capitalizeFirstLetter boolean that determines if the first
character of first word should be title case.
 * @param delimiters            set of characters to determine
capitalization, null and/or empty array means whitespace
 * @return camelCase of String, <code>null</code> if null String input
 */
public String toCamelCase(String str, final boolean capitalizeFirstLetter,
final char... delimiters) {
    // ...
}
```

The `final char... delimiters` allows you to pass any amount of delimiters by adding more `char` arguments.
For example `toCamelCase("hello:world", true, ':', ';')` would have two delimiters.
You can see this `delimiters` parameter as an array.

Variables

- `str` - the original string
- `capitalizeFirstLetter` - boolean
- `delimiters` - array of chars
- output - CamelCased string

Dependencies

There are no constraint dependencies among the input variables.

Equivalence and Boundary Analysis

- Variable `str`
  - null
  - empty
  - non-empty single word
  - non-empty multiple words
- Variable `capitalizeFirstLetter`
  - true
  - false
- Variable `delimiters`
  - `null`
  - no delimiter
  - single delimiter
  - multiple delimiters
- Variable `delimiters`, `str`:
  - delimiter exists in the string
  - delimiter does not exist in the string
  - string consists of only delimiters
- Boundaries:
  - No delimiter →single delimiter →• multiple delimiters
  - Single word →Multiple words

Strategy

- Single tests for null and empty.
- Combine `non-empty single word` with all partitions in `capitalize first letter`
- Combine `non-empty multiple words` with (`delimiters`, `str`). Choose either true or false for `capitalize first letter`; we will not combine with it as the number of tests would grow very large
- Note that we choose words that are not already in the correct format.

Test cases

- T1 null = `(null, true, '.') -> null`
- T2 empty = `("", true, '.') -> ""`
- T3 non-empty single word, capitalize first letter = `("aVOcado", true, '.') -> "Avocado"`

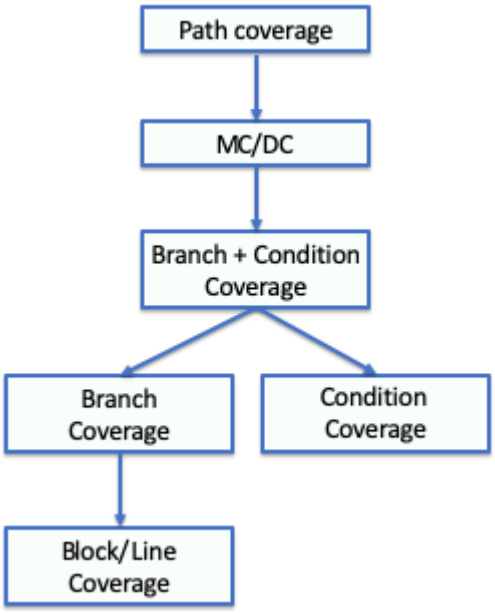
- T4 non-empty single word, not capitalize first letter = `("aVOcado", false, '.') -> "avocado"`
- T5 non-empty single word, capitalize first letter, no delimiter = `("aVOcado", true) -> "Avocado"`
- T6 non-empty single word, capitalize first letter, single existing delimiter = `("aVOcado", true, 'c') -> "AvoAdo"`
- T7 non-empty single word, capitalize first letter, single non-existing delimiter (skip as already tested with T4)
- T8 non-empty single word, capitalize first letter, multiple delimiters = `("aVOcado", true, 'c', 'd') -> "AvoAO"`
- T9 non-empty multiple words, capitalize first letter, no delimiter = `("aVOcado bAnana", true) -> "AvocadoBanana"`
- T10 non-empty multiple words, capitalize first letter, single existing delimiter = `("aVOcado-bAnana", true, '-') -> "AvocadoBanana"`
- T11 non-empty multiple words, capitalize first letter, single non-existing delimiter = `("aVOcado bAnana", true, 'x') -> "AvocadoBanana"`
- T12 non-empty multiple words, capitalize first letter, multiple delimiters, existing delimiter = `("aVOcado bAnana", true, ' ', 'n') -> "AvocadoBaAA"`
- T13 non-empty multiple words, capitalize first letter, multiple delimiters, non-existing delimiter = `("aVOcado bAnana", true, 'x', 'y') -> "AvocadoBanana"`
- T14 delimiters equal to `null` = `("apple", true, null) -> "Apple"`
- T15 only delimiters in the word = `("apple", true, 'a', 'p', 'l', 'e') -> "apple"`

Implementation

```
class Solution {

    private final DelftCaseUtilities delftCaseUtilities = new DelftCaseUtilities(
);

    @MethodSource("generator")
    @ParameterizedTest(name = "{0}")
    void domainTest(String name, String str, boolean firstLetter, char[] delimite
rs, String result) {
        assertThat(delftCaseUtilities.toCamelCase(str, firstLetter, delimiters)).
isEqualTo(result);
    }

    private static Stream<Arguments> generator() {
        return Stream.of(Arguments.of("null", null, true, null, null),
                Arguments.of("empty", "", true, new char[]{'.'}, ""),
                Arguments.of("non-
empty single word, capitalize first letter", "aVOcado", true, new char[]{'.'},
                        "Avocado"),
                Arguments.of("non-
empty single word, not capitalize first letter", "aVOcado", false, new char[]{'.'
},
                        "avocado"),
```

```
                Arguments.of("non-
empty single word, capitalize first letter, no delimiters", "aVOcado", true,
                        new char[]{}, "Avocado"),
                Arguments.of("non-
empty single word, capitalize first letter, single delimiter", "aVOcado", true,
                        new char[]{'.'}, "Avocado"),
                Arguments.of("non-
empty single word, capitalize first letter, multiple delimiters", "aVOcado", true
,
                        new char[]{'c', 'd'}, "AvoAO"),
                Arguments.of("non-
empty multiple words, capitalize first letter, no delimiters", "aVOcado bAnana",
true,
                        new char[]{}, "AvocadoBanana"),
                Arguments.of("non-
empty multiple words, capitalize first letter, single existing delimiter",
                        "aVOcado-bAnana", true, new char[]{'-
'}, "AvocadoBanana"),
                Arguments.of("non-
empty multiple words, capitalize first letter, single non-existing delimiter",
                        "aVOcado bAnana", true, new char[]{'x'}, "AvocadoBanana")
,
                Arguments.of("non-
empty multiple words, capitalize first letter, multiple existing delimiters",
                        "aVOcado bAnana", true, new char[]{' ', 'n'}, "AvocadoBaA
A"),
                Arguments.of("non-
empty multiple words, capitalize first letter, multiple non-existing delimiters",
                        "aVOcado bAnana", true, new char[]{'x', 'y'}, "AvocadoBan
ana"),
                Arguments.of("delimiters is null", "apple", true, null, "Apple"),
                Arguments.of("only delimiters in word", "apple", true, new char[]
{'a', 'p', 'l', 'e'}, "apple"));
    }
}
```

Structural Testing



- **Block coverage** = only difference from line coverage is that the minimal unit of coverage is the group of lines before reaching a true/false condition (so it does not depend on the developer verbose style)
- **Branch coverage** = possible route from the true/false condition is touched.
- **Condition coverage** = conditions might be composed of multiple sub-conditions, some of them with lazy operators. Condition coverage regards the extent of "touching" all conditions (so having them hit T and F, but not necessarily in all possible combinations)
- Path coverage = not only touching all conditions but also experiencing them in all the T/F combinations possible
  - MCDC = modified condition/decision coverage where only the sub-conditions that trigger a change in the decision are evaluated (for N conditions you need N+1 tests)

Given (A & B) | C which tests achieves 100% MC/DC?

Test case	A	B	C	(A & B) *or* C
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	T
8	F	F	F	F

{1,5} does not trigger change

Test case	A	B	C	(A & B) *or* C
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	T
8	F	F	F	F

{3,7} does not trigger change

Test case	A	B	C	(A & B) *or* C
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	T
8	F	F	F	F

{2,6} does trigger change

Test case	A	B	C	(A & B) *or* C
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	T
8	F	F	F	F

{4, 8} does not trigger change. We've exhausted all A cases ({2,6}). Do the same for B and C.

Total cases that trigger change:

A: {**2**, **6**}

B: {**2**, **4**}

C: {*3*,**4**}, {*5*,**6**}, {7,8}

Minimal (A + B + C) = {2,3,4,6 } or {2,4,5,6}… *In the case where a letter doesn't have sets -> it is not possible to achieve MC/DC coverage and probably the expression can be simplified.*

- Always ignore the line of the method and lines that only contain "{" or "}" unless otherwise specified.
- basic condition coverage: only looking at the conditions themselves while ignoring the overall outcome of the decision block
- condition coverage = full condition coverage = full condition coverage + branch coverage
- A minimal test suite that achieves 100% branch coverage has the same number of test cases as a minimal test suite that achieves 100% full condition coverage. (full means whole)

What is the condition coverage T1 = 15 and T2 = 8 test cases give combined:

```
public String fizzString(int n) {
    if (n % 3 == 0 && n % 5 == 0)
         return "FizzBuzz!";
    if (n % 3 == 0)
         return "Fizz!";
    if (n % 5 == 0)
         return "Buzz!";
    return n + "!";
}
```

This yields 5/8 = 62.5%:

condition	T	F
n % 3 == 0 at line 2	✓	✓
n % 5 == 0 at line 2	✓	(skipped due to lazy operator)
n % 3 == 0 at line 4	(skipped from returns)	✓
n % 3 == 0 at line 6	(skipped from returns)	✓

Loop boundary adequacy criterion

- 100% path coverage of a loop is impossible (requires test all possible (infinite) iterations), line coverage, decision and condition coverage may be 100% covered by a single test. Adequacy:
  - Test the loop is run zero times.
  - Test the loop is run only once.
  - Test the loop is run multiple times.
  - Use additional specification based techniques to determine the exact number of times.

Line coverage Junit

```
package delft;
```

```
class DelftStringUtilities {

    private DelftStringUtilities() {
        // Override default constructor, to prevent it from getting considered in
 the
        // coverage report.
    }

    /**
     * A String for a space character.
     *
     * @since 3.2
     */
    private static final String SPACE = " ";

    /** The maximum size to which the padding constant(s) can expand. */
    private static final int PAD_LIMIT = 8192;

    /**
     * The empty String {@code ""}.
     *
     * @since 2.0
     */
    private static final String EMPTY = "";

    /**
     * Right pad a String with a specified String.
     *
     * <p>
     * The String is padded to the size of {@code size}.
     *
     * @param str
     *            the String to pad out, may be null
     * @param size
     *            the size to pad to
     * @param padStr
     *            the String to pad with, null or empty treated as single space
     * @return right padded String or original String if no padding is necessary,
     *         {@code null} if null String input
     */
    public static String rightPad(final String str, final int size, String padStr
) {
        if (str == null) {
            return null;
        }
```

```
        if (padStr == null || padStr.isEmpty()) {
            padStr = SPACE;
        }
        final int padLen = padStr.length();
        final int strLen = str.length();
        final int pads = size - strLen;
        if (pads <= 0) {
            // returns original String when possible
            return str;
        }
        if (padLen == 1 && pads <= PAD_LIMIT) {
            return rightPadChar(str, size, padStr.charAt(0));
        }
        if (pads == padLen) {
            return str.concat(padStr);
        } else if (pads < padLen) {
            return str.concat(padStr.substring(0, pads));
        } else {
            final char[] padding = new char[pads];
            final char[] padChars = padStr.toCharArray();
            for (int i = 0; i < pads; i++) {
                padding[i] = padChars[i % padLen];
            }
            return str.concat(new String(padding));
        }
    }

    /**
     * Right pad a String with a specified character.
     *
     * <p>
     * The String is padded to the size of {@code size}.
     *
     * @param str
     *            the String to pad out, may be null
     * @param size
     *            the size to pad to
     * @param padChar
     *            the character to pad with
     * @return right padded String or original String if no padding is necessary,
     *         {@code null} if null String input
     * @since 2.0
     */
    public static String rightPadChar(final String str, final int size, final cha
r padChar) {
```

```
        if (str == null) {
            return null;
        }
        final int pads = size - str.length();
        if (pads <= 0) {
            // returns original String when possible
            return str;
        }
        return str.concat(repeat(padChar, pads));
    }

    /**
     * Returns padding using the specified delimiter repeated to a given length.
     *
     * <p>
     * Note: this method does not support padding with
     * <a href="http://www.unicode.org/glossary/#supplementary_character">Unicode
     * Supplementary Characters</a> as they require a pair of {@code char}s to be
     * represented. If you are needing to support full I18N of your applications
     * consider using {@link #repeat(String, int)} instead.
     *
     * @param ch
     *            character to repeat
     * @param repeat
     *            number of times to repeat char, negative treated as zero
     * @return String with repeated character
     * @see #repeat(String, int)
     */
    public static String repeat(final char ch, final int repeat) {
        if (repeat <= 0) {
            return EMPTY;
        }
        final char[] buf = new char[repeat];
        for (int i = repeat - 1; i >= 0; i--) {
            buf[i] = ch;
        }
        return new String(buf);
    }
}
```

Solution

```
package delft;
```

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

import static org.assertj.core.api.Assertions.assertThat;

class DelftStringUtilitiesTest {

    @MethodSource("generator")
    @ParameterizedTest(name = "{0}")
    void lineTest(String name, String str, int size,String padStr, String result)
 {
        assertThat(DelftStringUtilities.rightPad(str,size,padStr)).isEqualTo(resu
lt);
    }

    private static Stream<Arguments> generator() {
        return Stream.of(Arguments.of("null str",null,4,"c",null),
                Arguments.of("not null str and null padstr","abc",5,null,"abc  ")
,
                Arguments.of("str size smaller than int size","abc",1,null,"abc")
,
                Arguments.of("pad len = 1 = remaning pads to fill","abc",4,".","a
bc."),
                Arguments.of("remaning pads to fill smaller than pad length","abc
",4,".+","abc."),
                Arguments.of("remaning pads to fill larger than pad length and pa
d length not 1","abc",6,".+","abc.+."),
                Arguments.of("remaning pads to fill = pad length and pad length n
ot 1","abc",6,".+!","abc.+!")
                );
    }

    @Test
    void nullStrPadChar(){
        assertThat(DelftStringUtilities.rightPadChar(null,420,'A')).isEqualTo(nul
l);
    }
    @Test
    void sizeSmallerThanStrLength(){
        assertThat(DelftStringUtilities.rightPadChar("abc",1,'A')).isEqualTo("abc
");
```

```
    }
    @Test
    void sizeBiggerThanLimit(){
        assertThat(DelftStringUtilities.rightPadChar("abc", 8200,'A')).isEqualTo(
"abcAAA … AAA"); //just copy paste the outcome from the failed test
    }
    @Test
    void negativeIntRepeat(){
        assertThat(DelftStringUtilities.repeat('A',-1)).isEqualTo("");
    }
}
```

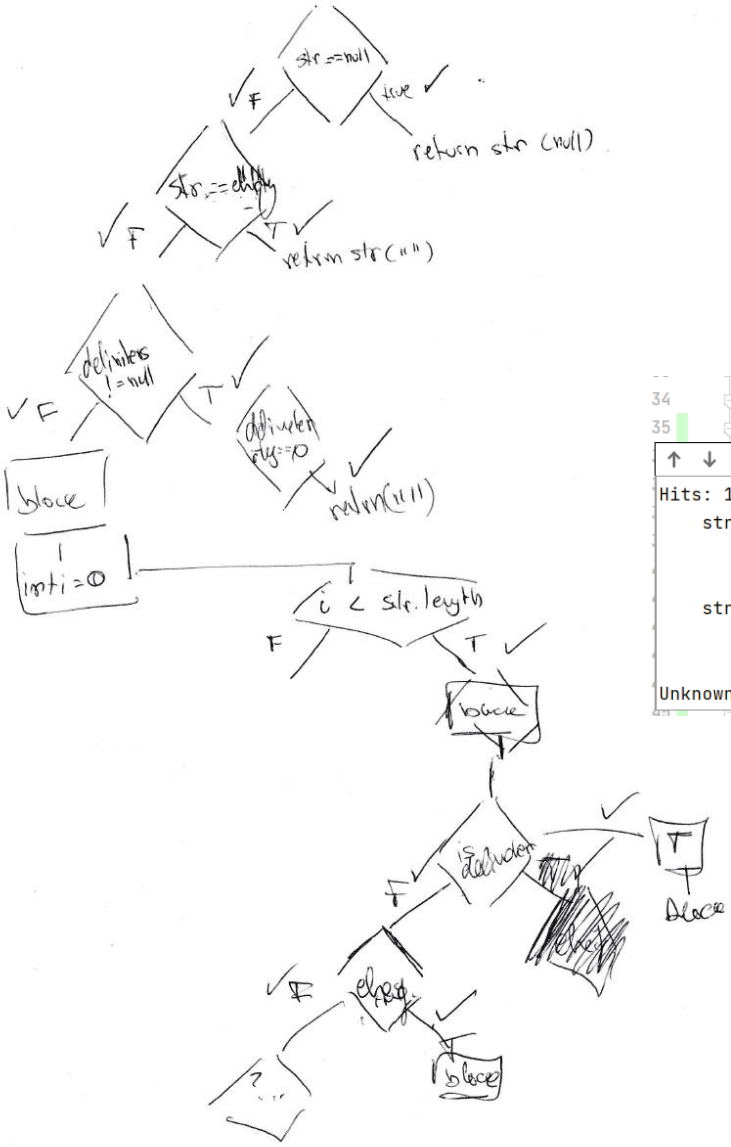
Condition coverage

```
package delft;

class DelftWordUtilities {

    private DelftWordUtilities() {
        // Override default constructor, to prevent it from getting considered in
 the
        // coverage report.
    }

    /**
     * The empty String {@code ""}.
     *
     * @since 2.0
     */
    private static final String EMPTY = "";

    /**
     * Extracts the initial characters from each word in the String.
     *
     * <p>
     * All first characters after the defined delimiters are returned as a new
     * string. Their case is not changed.
     *
     * <p>
     * If the delimiters array is null, then Whitespace is used. Whitespace is
     * defined by {@link Character#isWhitespace(char)}. A {@code null} input Stri
ng
     * returns {@code null}. An empty delimiter array returns an empty String.
     *
```

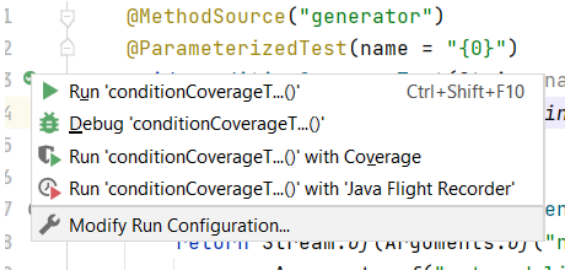
```
     * @param str
     *            the String to get initials from, may be null
     * @param delimiters
     *            set of characters to determine words, null means whitespace
     * @return String of initial characters, {@code null} if null String input
     * @see #initials(String)
     * @since 2.2
     */
    public static String initials(final String str, final char... delimiters) {
        if (str == null || str.isEmpty()) {
            return str;
        }
        if (delimiters != null && delimiters.length == 0) {
            return EMPTY;
        }
        final int strLen = str.length();
        final char[] buf = new char[strLen / 2 + 1];
        int count = 0;
        boolean lastWasGap = true;
        for (int i = 0; i < strLen; i++) {
            final char ch = str.charAt(i);
            if (isDelimiter(ch, delimiters)) {
                lastWasGap = true;
            } else if (lastWasGap) {
                buf[count++] = ch;
                lastWasGap = false;
            }
        }
        return new String(buf, 0, count);
    }

    /**
     * Is the character a delimiter.
     *
     * @param ch
     *            the character to check
     * @param delimiters
     *            the delimiters
     * @return true if it is a delimiter
     */
    private static boolean isDelimiter(final char ch, final char[] delimiters) {
        if (delimiters == null) {
            return Character.isWhitespace(ch);
        }
        for (final char delimiter : delimiters) {
```

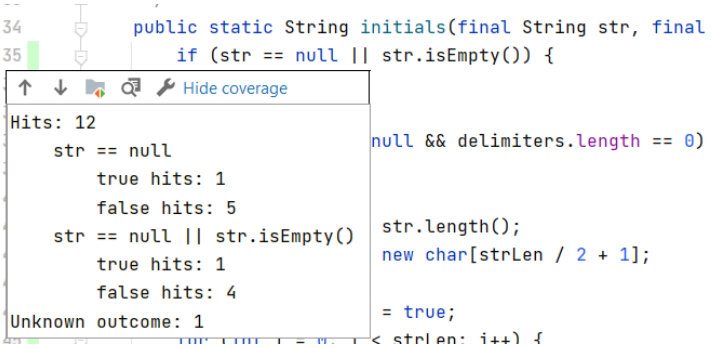
```
            if (ch == delimiter) {
                return true;
            }
        }
        return false;
    }
}
```

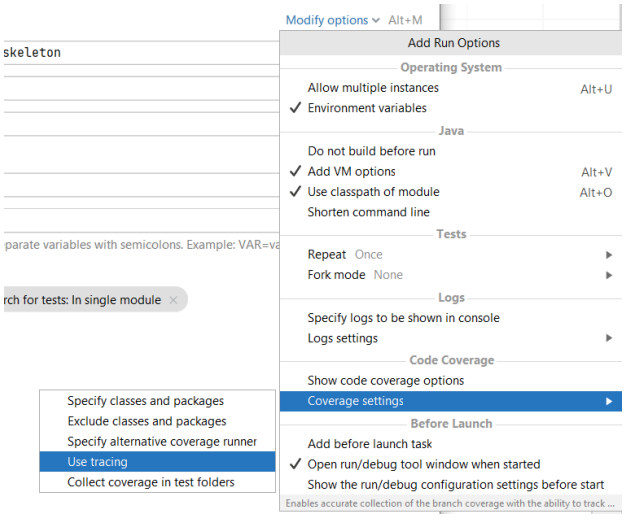
Because it's not basic but full condition coverage, split all the condition operators into separate conditions and make a CFG with True / False branches (T/F). Make test cases that hit all branches combined. (This CFG is not accurate but gets you close enough to then trial and error the last spec tests).



Furthermore, you can turn on branch by enabling tracing, or alternatively use JaCoCo runner.







Solution

```
package delft;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

import static org.assertj.core.api.Assertions.assertThat;

class DelftWordUtilitiesTest {


    @MethodSource("generator")
    @ParameterizedTest(name = "{0}")
    void lineTest(String name, String str, char[] delimiters, String result) {
        assertThat(DelftWordUtilities.initials(str, delimiters)).isEqualTo(result
);
    }

    private static Stream<Arguments> generator() {
        return Stream.of(Arguments.of("null str", null, new char[]{'c'}, null),
                Arguments.of("emtpy delimiter array", "sergio kirienko", new char
[0],""),
                Arguments.of("null delimiter array", "sergio kirienko", null,"sk"
),
                Arguments.of("emtpy str", "", null,""),
                Arguments.of("delimiter not found", "sergio kirienko",new char[]{
'c'},"s"),
                Arguments.of("only delimiter found", "ccc",new char[]{'c'},"")
        );
    }
}
```

Trick: Use .isNullOrEmpty() when you're not sure. Example:

```
package delft;

import java.util.ArrayList;
import java.util.List;

class DelftStringUtilities {
```

```
    private DelftStringUtilities() {
        // Override default constructor, to prevent it from getting considered in
 the
        // coverage report.
    }

    /**
     * Searches a String for substrings delimited by a start and end tag, returni
ng
     * all matching substrings in an array.
     *
     * <p>
     * A {@code null} input String returns {@code null}. A {@code null} open/clos
e
     * returns {@code
     * null} (no match). An empty ("") open/close returns {@code null} (no match)
.
     *
     * @param str
     *            the String containing the substrings, null returns null, empty
     *            returns empty
     * @param open
     *            the String identifying the start of the substring, empty return
s
     *            null
     * @param close
     *            the String identifying the end of the substring, empty returns
     *            null
     * @return a String Array of substrings, or {@code null} if no match
     * @since 2.3
     */
    public static String[] substringsBetween(final String str, final String open,
 final String close) {
        if (str == null || isEmpty(open) || isEmpty(close)) {
            return null;
        }
        final int strLen = str.length();
        if (strLen == 0) {
            return new String[0];
        }
        final int closeLen = close.length();
        final int openLen = open.length();
        final List<String> list = new ArrayList<>();
        int pos = 0;
        while (pos < strLen - closeLen) {
```

```
            int start = str.indexOf(open, pos);
            if (start < 0) {
                break;
            }
            start += openLen;
            final 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(new String[0]);
    }

    /**
     * Checks if a CharSequence is empty ("") or null. *
     *
     * <p>
     * NOTE: This method changed in Lang version 2.0. It no longer trims the
     * CharSequence. That functionality is available in isBlank().
     *
     * @param cs
     *            the CharSequence to check, may be null
     * @return {@code true} if the CharSequence is empty or null
     * @since 3.0 Changed signature from isEmpty(String) to isEmpty(CharSequence)
     */
    public static boolean isEmpty(final CharSequence cs) {
        return cs == null || cs.length() == 0;
    }
}
```

Solution

```
package delft;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;
```

```
import static org.assertj.core.api.Assertions.assertThat;


class DelftStringUtilitiesTest {

    @MethodSource("substringMethodGenerator")
    @ParameterizedTest(name = "{0}")
    void conditionCoverageTestSubstringMethod(String name, String str, String ope
n, String close, String result) {
        assertThat(DelftStringUtilities.substringsBetween(str, open, close)).isNu
ll();
    }

    private static Stream<Arguments> substringMethodGenerator() {
        return Stream.of(Arguments.of("null str", null, "open", "close", null),
                Arguments.of("null open", "str", null, "close", null),
                Arguments.of("null close", "str", "open", null, null),
                Arguments.of("not found open","sergio." , "(", ".", null),
                Arguments.of("not found close","(sergio" , "(", ".", null)
        );
    }

    @Test
    void conditionCoverageTestSubstringMethodNormal() {
        assertThat(DelftStringUtilities.substringsBetween("ser(gi)o" , "(", ")"))
.contains("gi");
    }

    @Test
    void emptyOpen() {
        assertThat(DelftStringUtilities.substringsBetween("ser(gi)o" , "", ")")).
isNull();
    }

    @Test
    void emptyClose() {
        assertThat(DelftStringUtilities.substringsBetween("ser(gi)o" , "(", "")).
isNull();
    }

    @Test
    void emmmpty() {
        assertThat(DelftStringUtilities.substringsBetween("" , "(", "a")).isNullO
rEmpty();
```

```
    }

    @Test
    void isEmpty() {
        CharSequence cs = new String();
        assertThat(DelftStringUtilities.isEmpty(cs)).isTrue();
    }

    @Test
    void isNotEmpty() {
        CharSequence cs = new String("Nah");
        assertThat(DelftStringUtilities.isEmpty(cs)).isFalse();
    }
}
```

Assertions

```
assert PRECONDITION;
//...
if (A) {
  // ...
  if (B) {
    // ...
    assert POSTCONDITION1;
    return ...;
  } else {
    // ...
    assert POSTCONDITION2;
    return ...;
  }
}
// ...
assert POSTCONDITION3;
return ...;
```

- Run java with -ea so that assertions are enabled, which in the case of not being true will throw an AssertionError. They are used to make debugging easier.

Invariant: What assertion(s), if any, can be turned into a class invariant?

```
public Square squareAt(int x, int y) {
  assert x >= 0;
  assert x < board.length;
  assert y >= 0;
  assert y < board[x].length;
  assert board != null;

  Square result = board[x][y];

  assert result != null;
```

```
  return result;
}
```

- Preconditions are those assertions that directly use the parameters of the method
- Postconditions are assertions that directly use the object that is going to be returned
- Invariants are assertions that use objects not specified in the method signature nor in the return statements (and that must remain true all the time (at least after method executions))
    - Static methods do not have invariants. Class invariants are related to the entire object, while static methods do not belong to any object (they are "stateless"), so the idea of (class) invariants does not apply to static methods.

Liskov Substitution Principle with Asserts

Interface (Parent) with Invariant I and {P} M {Q}

Implementation (Child) with I' and Q' stronger, P' weaker.

Example of good implementation: Mammal Parent, Child: Tiger. You can substitute land mammal with tiger, i.e. mammal.getFurColor(); meets assert animal.hasHair();

Example of violation: Mammal Parent, Child: Whale.

You cannot substitute a mammal with a whale (I know it's a mammal). Mammal.getFurColor() will not work with whale: whale's invariant is weaker (doesn't enforce fur).

Liskov Substitution Principle with JUnit Tests

```
package delft;

import java.util.HashMap;
import java.util.Map;

/** A square playing board. */
abstract class Board {

    /** The size of the board. */
    protected int size;

    /**
     * Creates a Board with a certain size.
     *
     * @param size
     *            the size of the board
     * @throws IllegalArgumentException
     *             if the size is negative
     */
    protected Board(int size) {
        if (size < 0) {
```

```
            throw new IllegalArgumentException("The size of the board cannot be n
egative.");
        }
        this.size = size;
    }

    /**
     * Returns the Unit at a position of the board. If no such a Unit has been se
t
     * before, it will return UNKNOWN.
     *
     * @param x
     *            x coordinate
     * @param y
     *            y coordinate
     * @return the unit at (x,y)
     */
    public abstract Unit getUnit(int x, int y);

    /**
     * Sets the unit of a certain position on the board.
     *
     * @param x
     *            x coordinate
     * @param y
     *            y coordinate
     * @param unit
     *            the new unit for the position (x,y)
     */
    public abstract void setUnit(int x, int y, Unit unit);

    /**
     * Checks whether the coordinates are in range and throws an exception if the
y
     * aren't.
     *
     * @param x
     *            x coordinate
     * @param y
     *            y coordinate
     * @throws IllegalArgumentException
     *             when the coordinates are out of the range of the board
     */
    protected void checkCoordinatesRange(int x, int y) {
        if (x < 0 || x >= size || y < 0 || y >= size) {
```

```
            throw new IllegalArgumentException(String.format("The position (%d, %
d) does not exist.", x, y));
        }
    }
}

class MapBoard extends Board {

    private final Map<Integer, Map<Integer, Unit>> board;

    public MapBoard(int size) {
        super(size);
        board = new HashMap<>();
    }

    @Override
    public Unit getUnit(int x, int y) {
        checkCoordinatesRange(x, y);
        if (board.containsKey(x) && board.get(x).containsKey(y)) {
            return board.get(x).get(y);
        }
        return Unit.UNKNOWN;
    }

    @Override
    public void setUnit(int x, int y, Unit unit) {
        checkCoordinatesRange(x, y);
        if (!board.containsKey(x)) {
            board.put(x, new HashMap<>());
        }
        board.get(x).put(y, unit);
    }
}

class ArrayBoard extends Board {

    private final Unit[][] board;

    public ArrayBoard(int size) {
        super(size);
        board = new Unit[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                board[i][j] = Unit.UNKNOWN;
            }
```

```
        }
    }

    @Override
    public Unit getUnit(int x, int y) {
        checkCoordinatesRange(x, y);
        return board[x][y];
    }

    @Override
    public void setUnit(int x, int y, Unit unit) {
        checkCoordinatesRange(x, y);
        board[x][y] = unit;
    }
}

enum Unit {
    FRIEND, ENEMY, UNKNOWN
}
```

Solution

```
package delft;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;
import static org.junit.jupiter.api.Assertions.*;

/**
 * Note that this test class is abstract! This means it wont be executed by itsel
f.
 * Only way to test the tests of this class is to create another non-
abstract class that extends it.
 */
abstract class BoardTest {

    /**
     * Initializes the abstract board with a special named constructor for abstra
ct classes (createClass)!
     */
    protected final Board board = createBoard(7);
    abstract Board createBoard(int size);
```

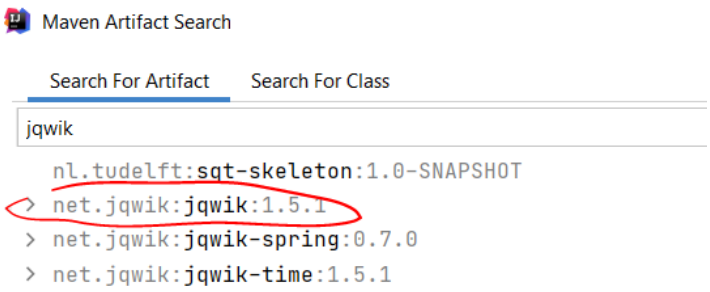
```
    /**
     * The method tested (createBoard(int size)) is equivalent to the Board const
ructor (Board(int size)).
     */
    @Test
    void illegalSizeTest(){
        assertThatThrownBy(() -> createBoard(-
3)).isInstanceOf(IllegalArgumentException.class);
    }

    /**
     * Another test for the abstract constructor to ensure 100% branch coverage.
     */
    @Test
    void createEmptyBoardTest() {
        assertThat(createBoard(0)).isNotNull();
    }

    //--
- Now we test the actual methods that appear in the abstract Board class with the
 default names
    @Test
    void getUnit() {
        assertThat(board.getUnit(0,0)).isEqualTo(Unit.UNKNOWN);
    }

    @Test
    void getUnitErrorXSmall() {
        assertThatThrownBy(() -> board.getUnit(-
1,0)).isInstanceOf(IllegalArgumentException.class);
    }

    @Test
    void getUnitErrorXBig() {
        assertThatThrownBy(() -
> board.getUnit(7,0)).isInstanceOf(IllegalArgumentException.class);
    }

    @Test
    void getUnitErrorY() {
        assertThatThrownBy(() -> board.getUnit(0,-
1)).isInstanceOf(IllegalArgumentException.class);
    }

    @Test
```

```
    void getUnitErrorYBig() {
        assertThatThrownBy(() -
> board.getUnit(0,7)).isInstanceOf(IllegalArgumentException.class);
    }

    @Test
    void setUnit() {
        board.setUnit(0,0,Unit.FRIEND);
        assertThat(board.getUnit(0,0)).isEqualTo(Unit.FRIEND);
    }

    @Test
    void getPreviouslySetTest() {
        board.setUnit(3, 2, Unit.ENEMY);
        assertThat(board.getUnit(3, 2)).isEqualTo(Unit.ENEMY);
    }

    @Test
    void getOtherElementOfPreviouslySetRowTest() {
        board.setUnit(3, 2, Unit.ENEMY);
        assertThat(board.getUnit(3, 4)).isEqualTo(Unit.UNKNOWN);
    }

    @Test
    void setPreviouslySetTest() {
        board.setUnit(0, 1, Unit.ENEMY);
        board.setUnit(0, 1, Unit.FRIEND);
        assertThat(board.getUnit(0, 1)).isEqualTo(Unit.FRIEND);
    }
}

class MapBoardTest extends BoardTest {

    /**
     * Important to override the abstract class constructor.
     * Doing this will execute all the tests of the abstract test class with the
MapBoard object.
     */
    @Override
    Board createBoard(int size) {
        return new MapBoard(size);
    }

    /**
```

```
     * We can test MapBoard specific tests too. It just happens that all of them
were direct implementations
     * of abstract methods, hence they could be written just in the abstract clas
s
     * so that it's also used by ArrayBoardTest.
     */
    @Test
    void example(){
        assertTrue(true);
    }
}

class ArrayBoardTest extends BoardTest {

    @Override
    Board createBoard(int size) {
        return new ArrayBoard(size);
    }
}
```

Property-Based Testing

- Automatically generated inputs for the tests
- Use jqwik engine by setting up the Maven dependency on IntelliJ:



Jqwik JUnit Example:

```
package delft;

class TaxIncome {

    public static final double CANNOT_CALC_TAX = -1;

    public double calculate(double income) {
        if (0 <= income && income < 22100) {
            return 0.15 * income;
        } else if (22100 <= income && income < 53500) {
            return 3315 + 0.28 * (income - 22100);
```

```
        } else if (53500 <= income && income < 115000) {
            return 12107 + 0.31 * (income - 53500);
        } else if (115000 <= income && income < 250000) {
            return 31172 + 0.36 * (income - 115000);
        } else if (250000 <= income) {
            return 79772 + 0.396 * (income - 250000);
        }
        return CANNOT_CALC_TAX;
    }
}
```

Solution

```
package delft;

import static org.junit.jupiter.api.Assertions.*;

import net.jqwik.api.*;
import net.jqwik.api.constraints.*;

class TaxIncomeTest {
    private final TaxIncome taxIncome = new TaxIncome();

    @Property
    void test0to22100(@ForAll @DoubleRange(min = 0, max = 22100, maxIncluded = fa
lse) double income) {
        assertEquals(taxIncome.calculate(income), 0.15 * income, Math.ulp(income)
);
    }

    @Property
    void test22100to53500(@ForAll @DoubleRange(min = 22100, max = 53500, maxInclu
ded = false) double income) {
        assertEquals(taxIncome.calculate(income), 3315 + 0.28 * (income - 22100),
 Math.ulp(income));
    }

    @Property
    void test53500to115000(@ForAll @DoubleRange(min = 53500, max = 115000, maxInc
luded = false) double income) {
        assertEquals(taxIncome.calculate(income), 12107 + 0.31 * (income - 53500)
, Math.ulp(income));
    }

    @Property
```

```
    void test115000to250000(@ForAll @DoubleRange(min = 115000, max = 250000, maxI
ncluded = false) double income) {
        assertEquals(taxIncome.calculate(income), 31172 + 0.36 * (income - 115000
), Math.ulp(income));
    }

    @Property
    void test250000(@ForAll @DoubleRange(min = 250000, minIncluded = false) doubl
e income) {
        assertEquals(taxIncome.calculate(income), 79772 + 0.396 * (income - 25000
0), Math.ulp(income));
    }

    @Property
    void invalid(@ForAll @Negative double income) {
        assertEquals(taxIncome.calculate(income), -1, Math.ulp(income));
    }
}
```

You can also add more than 1 parameter:

```
class TwoIntegersTest {
    private final TwoIntegers two = new TwoIntegers();

    @Property
    void normalTest(@ForAll @IntRange(min = 1, max = 99) int x,
                @ForAll @IntRange(min = 1, max = 99) int y) {
        assertEquals(two.sum(x,y), x+y);
    }
}
```

@Provide

You can also provide more specific parameters instead of using a range:

```
package delft;

class ChocolateBars {

    public static final int CANNOT_PACK_BAG = -1;

    public int calculate(int small, int big, int total) {
        int maxBigBoxes = total / 5;
        int bigBoxesWeCanUse = Math.min(maxBigBoxes, big);
        total -= (bigBoxesWeCanUse * 5);
```

```
        if (small < total)
            return CANNOT_PACK_BAG;
        return total;
    }
}
```

Solution

```
package delft;

import static org.assertj.core.api.Assertions.*;

import net.jqwik.api.*;
import net.jqwik.api.arbitraries.*;

class ChocolateBarsTest {

    private final ChocolateBars chocolateBars = new ChocolateBars();

    static class ChocolateBarsTestInput {

        int small;

        int big;

        int total;

        public ChocolateBarsTestInput(int small, int big, int total) {
            this.small = small;
            this.big = big;
            this.total = total;
        }
    }

    @Provide
    private Arbitrary<ChocolateBarsTestInput> onlySmall() {
        IntegerArbitrary small = Arbitraries.integers().greaterOrEqual(0);
        IntegerArbitrary big = Arbitraries.integers().greaterOrEqual(0);
        IntegerArbitrary total = Arbitraries.integers().greaterOrEqual(0);
        return Combinators.combine(small, big, total).as(ChocolateBarsTestInput::
new)
                .filter(k -> (k.total < 5 || k.big == 0) && k.small >= k.total);
    }

    @Provide
```

```
    private Arbitrary<ChocolateBarsTestInput> onlyBig() {
        IntegerArbitrary small = Arbitraries.integers().greaterOrEqual(0);
        IntegerArbitrary big = Arbitraries.integers().greaterOrEqual(0);
        IntegerArbitrary total = Arbitraries.integers().greaterOrEqual(0);
        return Combinators.combine(small, big, total).as(ChocolateBarsTestInput::
new)
                .filter(k -> (k.total <= 5 * k.big) && k.total % 5 == 0);
    }

    @Provide
    private Arbitrary<ChocolateBarsTestInput> both() {
        IntegerArbitrary small = Arbitraries.integers().greaterOrEqual(1);
        IntegerArbitrary big = Arbitraries.integers().greaterOrEqual(1);
        IntegerArbitrary total = Arbitraries.integers().greaterOrEqual(1);
        return Combinators.combine(small, big, total).as(ChocolateBarsTestInput::
new)
                .filter(k -
> (k.total - k.big * 5 > 0) && k.small >= k.total - k.big * 5);
    }

    @Provide
    private Arbitrary<ChocolateBarsTestInput> none() {
        IntegerArbitrary small = Arbitraries.integers().greaterOrEqual(0);
        IntegerArbitrary big = Arbitraries.integers().greaterOrEqual(0);
        IntegerArbitrary total = Arbitraries.integers().greaterOrEqual(1);
        return Combinators.combine(small, big, total).as(ChocolateBarsTestInput::
new)
                .filter(k -
> k.small < k.total - Math.min(k.total / 5, k.big) * 5);
    }

    @Property
    void onlySmallBars(@ForAll("onlySmall") ChocolateBarsTestInput input) {
        assertThat(chocolateBars.calculate(input.small, input.big, input.total)).
isEqualTo(input.total);
    }

    @Property
    void onlyBigBars(@ForAll("onlyBig") ChocolateBarsTestInput input) {
        assertThat(chocolateBars.calculate(input.small, input.big, input.total)).
isEqualTo(0);
    }

    @Property
    void bothBars(@ForAll("both") ChocolateBarsTestInput input) {
```

```
        assertThat(chocolateBars.calculate(input.small, input.big, input.total))
                .isEqualTo(input.total - Math.min(input.total / 5, input.big) * 5
);
    }

    @Property
    void noBars(@ForAll("none") ChocolateBarsTestInput input) {
        assertThat(chocolateBars.calculate(input.small, input.big, input.total)).
isEqualTo(-1);
    }

}
```

Test Doubles

- Dummy objects: object with dummy attributes
- Fake objects: poorman's version of a complex class (arraylist instead of a database)
- Stubs: 1-on-1 hard-coded answers for a fixed number of scenarios
- Spies: spies a dependency
- Mocks: you only mock the initialization of the object, but the behavior is determined by the rest of the implemented code (instead of by 1-on-1 hard-coded answers)
- Do not mock/stub the class under test
- You use mocks to control the external conditions and to observe the event being triggered.

Mockito

- Static class don't have invariants & static methods cannot be mocked by Mockito
    - (The invariant thing is and irrelevant fact regarding Mockito)
    - You can work around static methods by creating an abstraction on top of the dependencies that you do not own.

```
import static java.util.Arrays.asList;
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.when;

public class InvoiceFilterTest {
    private final IssuedInvoices issuedInvoices = Mockito.mock(IssuedInvoices.cla
ss);
…
 @Test
    void filterInvoices() {
    when(issuedInvoices.all()).thenReturn(asList(new Invoice("Mauricio", 20), new
 Invoice("Steve", 99)));
    assertThat(filter.lowestPrice()).isEqualTo(20);
 }
}
```

Reasons to mock:

- Slow dependency
- External dependency
- Hard to simulate cases

Reasons to not mock:

- Entities
- Native libraries

Trade-off:

- Reality and maintenance (changing code forces you to update mocks) vs convenience.

Example

```
package delft;

import java.util.Map;
import java.util.NoSuchElementException;
import java.util.Queue;

interface RequestService {

    /**
     * Returns a map with courses as keys and a list of requests as values. The
     * requests in each list are ordered chronologically in ascending order.
     *
     * @return the requests grouped by course and ordered chronologically
     */
    Map<String, Queue<String>> getRequestsByCourse();
}

class TheQueue {

    private final RequestService requestService;

    public TheQueue(RequestService requestService) {
        this.requestService = requestService;
    }

    /**
     * Gets the next request in the Queue for a specific course.
     *
```

```
     * @param course
     *            the course for which to get the next request
     * @return the next request
     */
    String getNext(String course) {
        Map<String, Queue<String>> requests = requestService.getRequestsByCourse(
);
        if (!requests.containsKey(course))
            throw new IllegalArgumentException("Course does not exist.");
        if (requests.get(course).isEmpty())
            throw new NoSuchElementException("There are no new requests.");
        return requests.get(course).poll();
    }
}
```

Solution

```
package delft;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.util.*;

import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatThrownBy;
import static org.mockito.Mockito.when;

class TheQueueTest {

    private RequestService requestService = Mockito.mock(RequestService.class);
    private TheQueue queue = new TheQueue(requestService);
    private Map<String,Queue<String>> requests = new HashMap<>();
    private Queue<String> spanishQueue = new LinkedList();

    @BeforeEach
    void setup(){
        when(requestService.getRequestsByCourse()).thenReturn(requests);
        requests.put("Spanish",spanishQueue);
        spanishQueue.addAll(List.of("Alex","Sergio"));
    }

    @Test
```

```
    void getNext() {
        assertThat(queue.getNext("Spanish")).isEqualTo("Alex");
    }

    @Test
    void getNextCourseNotFound() {
        assertThatThrownBy(()-
> queue.getNext("French")).isInstanceOf(IllegalArgumentException.class);
    }

    @Test
    void getNextEmptyCourse() {
        spanishQueue.clear();
        assertThatThrownBy(()-
> queue.getNext("Spanish")).isInstanceOf(NoSuchElementException.class);
    }
}
```

Verify

For void methods you can use verify to assert that they have been called. You don't use when(…).then…, once you pass the assert you can take the freedom to decide what happens next.

```
    @Test
    void alreadyEnqueued() {
        queue.enqueue("Sergio","Spanish");
        verify(requestService,times(0)).enqueue("Sergio","Spanish");
    }
```

Spy

You can spy on both, mocked classes and spied classes:

```
// this is a real list
List<String> list = new ArrayList<String>();

// this is a spy that will spy on the concrete list
// that is in the 'list' variable
List<String> spy = Mockito.spy(list);

// ...

spy.add(1); // this will call the concrete add() method

// ...

Mockito.verify(spy).add(1); // this will check whether add() was called
```

Design for testability

Controllability:

- Use dependency injection (put the objects in the method arguments signature as that will allow to mock the object much more easily)
    - If the objects that you want to inject are actually stable and used throughout the whole runtime process then it might be better to make them a private attribute and put the dependency injection in the class constructor.
- Static methods cant be mocked with Mockito, so they are bad for controllability too. Avoid them

Web testing

- Use a tool (driver and library) like Selenium WebDriver or Cypress to automatize clicks, searches, etc.
- Test for different browsers, screen sizes and mediums.
- Use snapshots to assert expected outcomes:

Snapshot testing is useful if you want detect unexpected changes to the component output (given the same set of inputs). For instance, when you do some refactoring, you might change the output of the component without realising it. The snapshot test then makes you aware of the change.

It would not be useful if you blindly update the snapshot every time there is a change, without inspecting the differences and deciding whether those are what you expected. If you would do so, the saved snapshot would not reflect the intended outcome, so the test would become incorrect.

- Web reaction times (loading a page) make tests flaky, although Cypress has retry-and-timeout logic built in

JavaScript Unit testing

- Keep js separate from html
- Be aware of dependency injection
- Be aware of how the DOM elements are retrieved (it's better to use id's as they are less likely to change than the rest of the HTML layout)

Without frame work example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <title>Date utils - Test</title>
</head>

<body>
    <p>View the console output for test results.</p>

    <script src="dateUtils.js"></script>
```

```
    <script>

        function assertEqual(expected, actual) {
            if (expected != actual) {
                console.log("Expected " + expected + " but was " + actual);
            }
        }

        /*
        Be aware that in JavaScript Date objects, months are zero-based,
        meaning that month 0 is January, month 1 is February, etc.
        So "new Date(2020, 1, 29)" represents February 29th, 2020.
        */

        // Test 1: incrementDate should add 1 day a given date object
        var date1 = new Date(2020, 1, 29);  // February 29th, 2020
        incrementDate(date1);
        // This succeeds:
        assertEqual(new Date(2020, 2, 1).getTime(), date1.getTime());

        // Test 2: dateToString should return the date in the form "yyyy-MM-
dd"
        var date2 = new Date(2020, 4, 1);   // May 1st, 2020
        // This fails because of time zone issues
        // (the actual value is "2020-04-30"):
        assertEqual("2020-05-01", dateToString(date2));
    </script>
</body>

</html>
```

JavaScript unit test with React:

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import DateIncrementer from './DateIncrementer';

test('renders initial date', () => {
  const { getByText } = render(<DateIncrementer initialDate={new Date(2020,
0, 1)} />);
  const dateElement = getByText("2020-01-01");
  expect(dateElement).toBeInTheDocument();
});

test('updates correctly when clicking the "+1" button', () => {
  const date = new Date(2020, 0, 1);
  const { getByText } = render(<DateIncrementer initialDate={date} />);
  const button = getByText("+1");

  fireEvent.click(button);

  const dateElement = getByText("2020-01-02");
  expect(dateElement).toBeInTheDocument();
});
```

Jest mocks

```
jest.mock('./dateUtils');

import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import { dateToString, addOneDay } from './dateUtils';
import DateIncrementer from './DateIncrementer';

test('renders initial date', () => {
  dateToString.mockReturnValue("mockDateString");

  const date = new Date(2020, 0, 1);
  const { getByText } = render(<DateIncrementer initialDate={date} />);
  const dateElement = getByText("mockDateString");

  expect(dateToString).toHaveBeenCalledWith(date);
  expect(dateElement).toBeInTheDocument();
});

test('updates correctly when clicking the "+1" button', () => {
  const mockDate = new Date(2021, 6, 7);
  addOneDay.mockReturnValueOnce(mockDate);

  const date = new Date(2020, 0, 1);
  const { getByText } = render(<DateIncrementer initialDate={date} />);
  const button = getByText("+1");

  fireEvent.click(button);

  expect(addOneDay).toHaveBeenCalledWith(date);
  expect(dateToString).toHaveBeenCalledWith(mockDate);
});
```

Snapshot testing

In Jest, such a test can look like this:

```
test('renders correctly via snapshot', () => {
  const { container } = render(<DateIncrementer initialDate={new Date(2020,
0, 1)} />);
  expect(container).toMatchSnapshot();
});
```

End-to-end testing

- Test the flow of the application as a user would follow it while integrating all the used components
- Use Selenium Webdriver to also simulate the browser

Usability vs accessibility

Usability:

- User friendliness
- Easy to use

Accessibility:

- Accessible to people with disabilities
- WCAG = Web content accessibility guidelines
- AXE, screen reader, asking a disabled are ways to test for accessibility

Screenshot vs snapshot

- Snapshot (Jest) is a UI structure of a page, stored in xml-like file (.snap).
- Screenshot = visual regression, (BackstopJS) is just images
- Both testing techniques are the same in terms of comparing the current page state to the last one.

Given an arbitrary javascript library (yikes) see the following arguments in favor of "testable" and "not testable"

```
import { createLocal } from '../create/local';
import { cloneWithOffset } from '../units/offset';
import isFunction from '../utils/is-function';
import { hooks } from '../utils/hooks';
import { isMomentInput } from '../utils/is-moment-input';
import isCalendarSpec from '../utils/is-calendar-spec';

// ...

export function calendar(time, formats) {
    // Support for single parameter, formats only overload to the calendar
function
    if (arguments.length === 1) {
        if (isMomentInput(arguments[0])) {
            time = arguments[0];
            formats = undefined;
        } else if (isCalendarSpec(arguments[0])) {
            formats = arguments[0];
            time = undefined;
        }
    }
    // We want to compare the start of today, vs this.
    // Getting start-of-today depends on whether we're local/utc/offset or
not.
    var now = time || createLocal(),
        sod = cloneWithOffset(now, this).startOf('day'),
        format = hooks.calendarFormat(this, sod) || 'sameElse',
        output =
            formats &&
            (isFunction(formats[format])
                ? formats[format].call(this, now)
                : formats[format]);

    return this.format(
        output || this.localeData().calendar(format, this, createLocal(now))
    );
}
```

Yes, it is testable. The data used by the function are the input parameters (`time` and `formats`), some methods on `this`, and the imported functions/objects (`createLocal`, etc.).

- We can supply the parameters by just passing appropriate values to the function. We have to choose the first argument carefully, because depending on its structure, it has a different meaning (because of the fake 'overloading'), but it is possible.
- The properties on `this` (like `this.format()`) are a bit more tricky. You could do something like this:

```
var obj = { format: jest.fn(), localeData: jest.fn() };
var result = calendar.call(obj, 'nl');

// or:
// obj.calendar = calendar;
// var result = obj.calendar('nl');
```

-
- We can mock the imported functions/objects if necessary (or use them as is).
- We can 'spy' on the imported functions and see whether they have been called with the right parameters.

- No, it is not testable.

  - There is no documentation, so it is not clear what the function is supposed to do.
  - The function has so many dependencies (parameters and imports) that it becomes tedious to set up a test.
  - There are many functions that operate on `this`, and it is not clear what state `this` should be in for the functions to work.
  - The fake 'overloading' makes it harder to supply the right arguments.
  - Possible refactoring: We could create two functions with different names, so we avoid the 'overloading'.
  - Possible refactoring: We could make a choice between the OO style and the functional style. Right now, there seems to be a mix of both, making it unclear what happens and how it should be tested. In a purely functional implementation, testing the functions would become straightforward because we do not have to handle state any more. In a pure OO implementation, there would be a class with methods, and it would be clear how the methods interact.

Page vs state

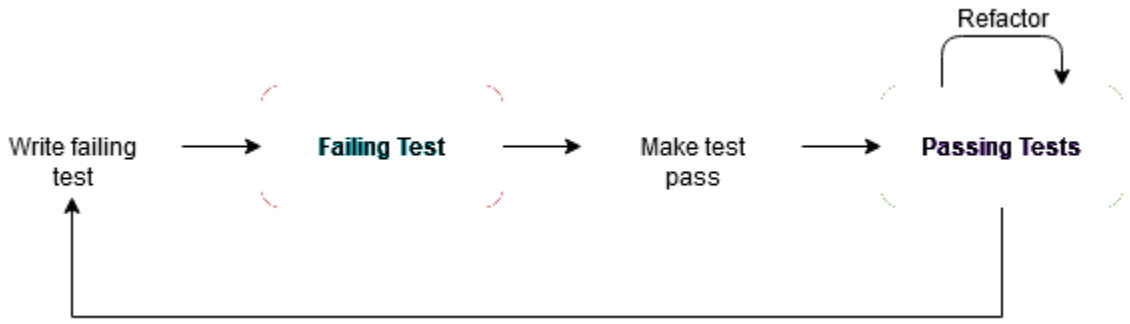
- Page objects are fully meant to represent/abstract some page of your web application. If your page has a form and a button, your page object would probably have methods such as `fillForm(...)` and `submit()`. The goal of a page object is to encapsulate the low-level details of how to handle that page, which at the end, simplifies and decouples your test code. There is no real constraint on how to model these Page Objects.
- State objects, on the other hand, try to represent the web app as a state machine. So, each state of the system becomes one state object. One state object may trigger an action that takes you to another state object.

Software regression

Not to be confused with visual regression, a software regression is a type of software bug where a feature that has worked before stops working.

- "guarding against UI regressions" means that you want to make sure that the output of your UI component does not change (or 'stop working correctly') when some unrelated code gets changed. That is exactly what snapshot tests are used for.
- A snapshot test just captures the actual output of the component; not whether that output matches the specifications.

TDD



- By creating the test first, we also look at the requirements first.
- We can control our pace of writing production code.
- Testable code from the beginning.
- Quick feedback on the code that we are writing.
- Baby steps
- You should use TDD when you do not know how to design and/or architect a part of the system (or dealing with a complex problem).
- You should **not** use TDD when you are familiar with the problem, or the design decisions are clear in your mind. If there is “nothing to be learned or explored”, TDD might not really afford any significant benefit.

TDD example. For this we use the Roman Numeral problem.

The Roman Numeral problem

It is our goal to implement a program that receives a string as a parameter containing a roman number and then converts it to an integer.

In roman numeral, letters represent values:

- I = 1
- V = 5

- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

Letters can be combined to form numbers.
For example we make 6 by using

$$5+1=6$$

and
have the roman number “VI”.
Example: 7 is “VII”, 11 is “XI” and 101 is “CI”.
Some numbers need to make use of a subtractive notation
to be represented.
For example we make 40 not by “XXXX”, but
instead we use

$$50-10=40$$

and have the roman number “XL”.
Other examples: 9 is “XI”, 40 is “XL”, 14 is “XIV”.

The letters should be ordered from the highest to the lowest value.
The values of each individual letter is added together.
Unless the subtractive notation is used in which a letter
with a lower value is placed in front of a letter with a higher value.

Combining both these principles we could give our
method “MDCCCXLII” and it should return 1842.

In your own IDE, write a testsuite and an implementation for the program specified above. Use the TDD cycle: write a test, adapt the implementation to make it pass, repeat.

Test suite in chronological order

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class TDDRomanNumeralTest {

    TDDRomanNumeral number;
    @BeforeEach
    void setup(){
        number = new TDDRomanNumeral();
    }
    @Test
    void one(){
```

```
    assertThat(number.convert("I")).isEqualTo(1);
}

@Test
void five(){
    assertThat(number.convert("V")).isEqualTo(5);
}

@Test
void ten(){
    assertThat(number.convert("X")).isEqualTo(10);
}

@Test
void fifty(){
    assertThat(number.convert("L")).isEqualTo(50);
}

@Test
void hundred(){
    assertThat(number.convert("C")).isEqualTo(100);
}

@Test
void fiveHundred(){
    assertThat(number.convert("D")).isEqualTo(500);
}

@Test
void thousand(){
    assertThat(number.convert("M")).isEqualTo(1000);
}

@Test
void two(){
    assertThat(number.sum("II")).isEqualTo(2);
}

@Test
void three(){
    assertThat(number.sum("III")).isEqualTo(3);
}

@Test
void four(){
    assertThat(number.calculate("IV")).isEqualTo(4);
}

@Test
void six(){
    assertThat(number.calculate("VI")).isEqualTo(6);
}

@Test
void MDCCCXLII(){
    assertThat(number.calculate("MDCCCXLII")).isEqualTo(1842);
}
```

```
}
```

Class Implementation

```
public class TDDRomanNumeral {
    public int convert(String str) {
        switch (str) {
            case "I":
                return 1;
            case "V":
                return 5;
            case "X":
                return 10;
            case "L":
                return 50;
            case "C":
                return 100;
            case "D":
                return 500;
            case "M":
                return 1000;
            default:
                return 0;
        }
    }

    public int sum(String str) {
        int i = 0;
        String[] chars = str.split("");
        for (String letter : chars) {
            i += convert(letter);
        }
        return i;
    }

    public int diff(String str) {
        return convert(str.substring(1)) - convert(str.substring(0, 1));
    }

    public int calculate(String str) {
        int n = 0;
        String[] chars = str.split("");
        for (int i = chars.length-1; i >= 0; i--){
            if (i >= 1 && convert(chars[i]) > convert(chars[i-1])){
                n += diff(chars[i-1] + chars[i]);
                i--;
            } else if (i >= 0) {
                n += convert(chars[i]);
            }
        }
        return n;
    }
}
```

Domain testing sample answer

Variables:

- a, explain
- b, explain

Dependencies:

I see some dependency between variables A and B, explain …

Equivalence partitioning and boundary analysis:

- variable A:
  - partition 1
  - partition 2
- variable B:
  - partition 3
  - partition 4
- (a, b)
  - partition 5
  - partition 6
- Boundaries
  - boundary 1: explanation
  - boundary 2: explanation

Structural testing

- Structural testing helped me in finding new test cases, A, B, and C…
- Domain tests and structural tests together achieve 100% branch+condition coverage.

Strategy:

- Combine everything from domain and structural testing…
- Total number of tests = 3

Test cases:

- T1 = …
- T2 = …
- T3 = …

Test code quality and test smells

FIRST properties

- Fast: (tests should be fast…)
- Isolated: should work without dependencies
- Repeatable: should work for unlimited iterations
- Self-validating: should assert whatever it pretends to test
- Timely: should be written at the same rate as the production code

Test Desiderata

- Isolated
- Composable
- Fast
- Inspiring
- Writable
- Readable
    - Understandable
    - Arrange, Act, Assert
- Behavioral
- Structure-insensitive
- Automated
- Specific
- Deterministic

Testcode smells

- Code duplication
- Assertion roulette: avoid multiple assertions in a test and make the fewer assertions very specific to the unit test
- Resource optimism: assuming (an external) resource is always available
- Test run war: when developers running the same test suite at the same time creates problems
- General fixture: don't have a shared set of inputs but create inputs specific to each test
- Indirect tests: tests that test irrelevant classes (a test should be focused on 1 class)
- Eager tests: these exercise more than 1 method or behavior
- Sensitive equality: test should have same outcomes after refactoring and changing external stuff
- Inappropriate assertions: i.e. use assertEquals(a,b) instead of assertTrue(a == b)
- Mystery guest: i.e. non obvious external object (such as a database)

Test Data Builder

```
public class InvoiceBuilder {

    private String country = "NL";
    private CustomerType customerType = CustomerType.PERSON;
    private BigDecimal value = new BigDecimal("500.0");

    public InvoiceBuilder withCountry(String country) {
        this.country = country;
        return this;
    }
```

```
    public InvoiceBuilder asCompany() {
        this.customerType = CustomerType.COMPANY;
        return this;
    }

    public InvoiceBuilder withAValueOf(String value) {
        this.value = new BigDecimal(value);
        return this;
    }

    public Invoice build() {
        return new Invoice(value, country, customerType);
    }
}
```

With the much clearer test:

```
@Test
void taxesForCompanies() {
    var invoice = new InvoiceBuilder()
            .asCompany()
            .withCountry("NL")
            .withAValueOf("2500")
            .build();

    var calculatedValue = invoice.calculate();

    assertThat(calculatedValue).isCloseTo(new BigDecimal("250"), within(new
BigDecimal("0.001")));
}
```

Flaky tests

- May depended on External/shared resources
- May have improper time-outs
- May have hidden interactions between different test methods
- If it gets worse overtime:
    - Probably resource leakage
    - XOR non-deterministic test
- Lonely test: only works when executed alone
- Interacting test: only works after another test sets up the desired state

Security testing

- Anything that can potentially corrupt the memory can influence where the instruction pointer points to. Such as:
    - Buffer overflows
    - Type confusion
    - Deserialising bugs
- Example of a software bug that also qualifies as a security vulnerability.:

- An infinite loop can be triggered by attackers to do a Denial of Service attack.

Update attack happens when reflection is used to update the logic of codebase. The malicious logic is updated at run-time. Prior to that, the code seems benign. Because static analysis checks code at compile time, the malicious logic isn't added yet. Though some testing tools may flag reflection code, the update attack itself cannot be detected until run-time.

Objectives and techniques

1. Testing like an attacker -> Penetration testing
2. Branch reachability analysis -> Symbolic execution
3. Detect pre-defined patterns in code -> Regular expressions
4. Generate complex test cases -> Fuzzing

How can you use Tainting to detect spyware?

Spyware usually accesses information that benign software does not and sends information to parties that benign software does not.
Tracking which data the program under test (spyware) accesses, and where that data flows is one way to detect spyware. For example, we know that `/etc/passwd` is sensitive, so we taint it. If a program requests it, we can follow the taint and see if it is being sent over the network.

Basically you CTRL+F over the network for data exactly equal to sensitive data that you think is of an attacker's interest. So that data is "tainted", if there is a match, we can look up which program has sent it over the network, that's our infected software.

A packet-sniffer is a type of software that can intercept and read all network traffic that is transmitted from a device over a network.

How can taint analysis be used to detect packet sniffers?

Sample data to be transmitted over the network is tainted and tracked. If a packet sniffer is present on the system, it will try to access this data (because that's what sniffers do) because of which taint will be spread to the sniffer app, and can be detected.

- The most likely injection vulnerability to be detected by static analysis is Format String injection.

Perform Reaching Definitions Analysis on the following piece of code.
Which values of the variables does the analysis produce?

```
int x = 0;
int y = 1;
while (y < 5) {
  y++;
```

```
  if (y > 7)
    x = 12;
  else
    x = 21;
}
return x;
```

```
x = {0,12,21}
y = {1,2,3,...}
```

Since reaching definitions analysis produces all the possible values a variable might take, resulting in over-generalization and false positives.

Intelligent testing

- REGULAR EXPRESSIONS CANNOT DETECT SEMANTICS
- Static analysis produces over-generalized results with some false positives, hence Sound but Incomplete.

Static testing

- Pattern matching: RegEx to find bugs
- Syntax analysis: Parse Tree (or Abstract Syntax Tree, which ignores semicolons etc) uses code syntax patterns to find bugs
- Static analysis: Static analysis produces over-generalized results with some false positives. Therefore it is Sound but Incomplete.

Mutation

- Hypotheses suggest that the mutant size should be small:
    - Competent Programmer: only tiny syntax mistakes are made (i.e. operators)
    - Coupling effect: complex faults are made of simple faults, therefore catching most small faults also means catching most complex faults.
- Operators:
    - Arithmetic Operator Replacement (+, −, *, /, %)
    - Relational Operator Replacement (<, >, <=, >=, !=, ==)
    - Conditional Operator Replacement (&, |, &&, ||, !, ^)
    - Assignment Operator Replacement (=, +=, −=, /=)
    - Scalar variable replacement: a variable of a given type for other one declared of the same type
- Language specific operators:
    - Java
        - Implements
        - Expands
        - Public
        - Static
        - Private
    - C
        - Access modifier change

  - Hiding variable deletion
  - Hiding variable insertion
  - Overriding method deletion
  - Parent constructor deletion
  - Declaration type change
- Mutation analysis: assessing the quality of the test suite by computing the mutation score
- Mutation testing: improving the quality of the test suite by using mutants

Example: Provide an upper-bound estimate for the number of possible mutants of the method above

```
/**
 * <p>Gets the minimum of three {@code int} values.</p>
 *
 * @param a  value 1
 * @param b  value 2
 * @param c  value 3
 * @return  the smallest of the values
 */
public static int min(int a, final int b, final int c) {
    if (b < a) {
        a = b;
    }
    if (c < a) {
        a = c;
    }
    return a;
}
```

Which of the following mutation operators can be applied to the method in order to obtain a *mutant*? A) single-order mutation B) higher-order mutation

Single order mutation:

We would have 2*(6-1) for Relational Operator Replacement
We would have 2*(4-1) for Assignment Operator Replacement
We would have 9*(3-1) Scalar Variable Replacments

Total: 10 + 6 + 18 = 34 mutants

Higher-order mutation:

`6^2 * 4^2 * 3^9 - 1 = 11337407` mutants.

- We calculate all the possible variants and then subtract `1` for the original method so that we are left with only the mutants.
- As you can see, the number of all possible mutants is quite significant (and too expensive to be used in practice).

Mutation score

- the process of determining the mutation score cannot be fully automated as an accurate score depends on weeding out equivalent mutations, which is an undecidable/NP-hard problem.
- Programs do not know the implicit behavior of a program, they only adapt themselves to the existing assertions. Therefore they can generate equivalent mutants which need to be reviewed by the developer.

$$mutation\ score = \frac{killed\ mutants}{nonequivalent\ mutants}$$

Similar Mutants

- If similar mutants are not discarded, the runtime of mutation testing would be very long, greatly reducing usability. Not discarding similar mutants also skews the mutation score.

Fuzz testing

- Bombard the system with random inputs to find unexpected bugs (from crash, memory, exceptions, etc).
    - Symbolic execution limits the search-space covered by a fuzzer by specifying bounds on variable values that cover a desired path

Search-based software testing

- Using genetic algorithms to generate automatic test cases
- Randoop can be used to find bugs in your program and to create regression tests
- Oracle problem
    - Random based tools to generate cases do not know the implicit results that the method should give, they only adapt themselves to match the assertions.
    - Not a problem: use randomization to find crashes and exceptions
    - Randoop checks for the following contracts:
        - Contracts over Object.equals():
            - Reflexivity: o.equals(o) == true
            - Symmetry: o1.equals(o2) == o2.equals(o1)
            - Transitivity: o1.equals(o2) && o2.equals(o3) ⇒ o1.equals(o3)
            - Equals to null: o.equals(null) == false
            - it does not throw an exception
        - Contracts over Object.hashCode():
            - Equals and hashcode are consistent: If o1.equals(o2)==true, then o1.hashCode() == o2.hashCode()
            - it does not throw an exception
        - Contracts over Object.clone():
            - it does not throw an exception, including CloneNotSupportedException
        - Contracts over Object.toString():
            - it does not throw an exception
            - it does not return null
        - Contracts over Comparable.compareTo() and Comparator.compare():
            - Reflexivity: o.compareTo(o) == 0 (implied by anti-symmetry)
            - Anti-symmetry: sgn(o1.compareTo(o2)) == -sgn(o2.compareTo(o1))

        - Transitivity: o1.compareTo(o2)>0 && o2.compareTo(o3)>0 ⇒ o1.compareTo(o3)>0
        - Substitutability of equals: x.compareTo(y)==0 ⇒ sgn(x.compareTo(z)) == sgn(y.compareTo(z))
        - Consistency with equals(): (x.compareTo(y)==0) == x.equals(y) (this contract can be disabled)
        - it does not throw an exception
    - Contracts over checkRep() (that is, any nullary method annotated with @CheckRep):
        - it does not throw an exception
        - if its return type is boolean, it returns true
- Violation of any of these contracts is highly likely to indicate an error.