Black-box and Glass-box Testing

Table of contents

Applied class for 2026-04-16-solid-principles-ii[1] (Week 7). Applies the black-box/white-box testing concepts from java-testing[2] (Week 5) to a worked example. All of these tasks ask for *a* set of inputs achieving some coverage goal — many different answers are equally correct, so answers here are reveal-only worked examples rather than checkable exercises.

The `Bus` class

```
public class Bus {
    public Bus(int capacity) {...}
    public int getCurrent() {...}
    public int getAverageCount() {...}
    public void stop(int on, int off) {...}
}
```

[1] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[2] courses/csse2002/java-testing.pdf

Task 0 — write a smoke test

Working

A smoke test just exercises normal/expected operation:

```
Bus bus = new Bus(10);
bus.stop(5, 0);
bus.stop(3, 1);
assertEquals(7, bus.getCurrent());
assertEquals(6, bus.getAverageCount());
```

Task 1 — develop boundary scenarios (with justification)

Working

Scenario	Justification
Construct a bus with capacity 0	A bus that can have no passengers is an uncommon scenario we may expect bugs from.
Construct a bus with a negative capacity	Unclear how this should behave — worth clarifying and testing.
Ask for the average after no stops	Averages usually involve division by count; want to confirm no exception when count is 0.
Negative values for `on`/`off`	Unclear behaviour — does a negative `on` remove passengers?
More passengers get off than are currently on the bus	Results in undefined behaviour.
More passengers on the bus than capacity	Behaviour isn't specified, so worth observing.

Task 2 — how can we make `Bus`'s behaviour fully specified?

> **Working**
>
> Most of the boundary scenarios above come from unspecified/unclear behaviour. If we declare `getCurrent() <= capacity` an **invariant** of `Bus` (see java-specification[3]), we must protect it everywhere:
>
> 1. Constructor throws `IllegalArgumentException` if `capacity < 0`.
> 2. `stop` gets a precondition that both `on` and `off` are `>= 0`.
> 3. `stop` throws `IllegalStateException` if `getCurrent() + on - off` would be negative or exceed `capacity`.

Glass-box testing: code coverage levels

See java-testing[4] for the three levels (statement, branch, path). Consider:

```java
public static int countOccurrences(int[] array, int target) {
    int count = 0;
    for (int num : array) {
        if (num == target) {
            count++;
        }
    }
    return count;
}
```

A single array containing the target at least once (e.g. `[2, 3, 2]`, target 2) gives statement coverage (the `if` body runs); adding an array where the target never appears (e.g. `[3]`, target 2) gives branch coverage too.

Worked example: `ascending`

```java
/**
 * @require numbers != null
 * @ensure \result is true iff for all indices j such that
 *         0 <= j < numbers.length - 1, numbers[j] <= numbers[j + 1]
```

[3] courses/csse2002/java-specification.pdf

[4] courses/csse2002/java-testing.pdf

```
 */
public boolean ascending(int[] numbers) {
    boolean result = true;
    for (int i = 0; i < numbers.length - 1; i++) {
        if (numbers[i] > numbers[i + 1]) {
            result = false;
        }
    }
    return result;
}
```

Working

Statement coverage: `[2, 1]` (loop runs once, `if` body executes).
Branch coverage: `[2, 1, 2]`, or the pair `[2, 1]` and `[1, 2]` (need the `if` to be both true and false).
Path coverage (0, 1, and 2-iteration cases, crossed with the `if` outcome each time):

Input	Justification
`[]` or `[3]`	0 times through loop
`[3, 4]`	1 time through loop, `if` false
`[4, 3]`	1 time through loop, `if` true
`[3, 4, 5]`	2 times through loop, false, false
`[3, 4, 2]`	2 times through loop, false, true
`[4, 3, 5]`	2 times through loop, true, false
`[5, 4, 3]`	2 times through loop, true, true

Worked example: `maximum` (bonus)

```
/**
 * @require numbers != null
 * @ensure numbers.length > 0 ==> (
 *     (\forall int i; 0 <= i < numbers.length ==> \result >= numbers[i]) &&
 *     (\exists int i; 0 <= i < numbers.length && \result == numbers[i]))
 * @ensure numbers.length == 0 ==> \result = Integer.MIN_VALUE
 */
public int maximum(int[] numbers) {
    int currentMaximum = Integer.MIN_VALUE;
    for (int number : numbers) {
        if (number > currentMaximum) {
            currentMaximum = number;
```

```
        }
    }
    return currentMaximum;
}
```

> **Working**
>
> **Statement coverage**: `[1]`.
> **Branch coverage**: `[2, 1]` (first element sets a new max, second doesn't).
> **Path coverage**:

Input	Justification
`[]`	0 times through loop
`[Integer.MIN_VALUE]`	1 time through loop, `if` false
`[1]`	1 time through loop, `if` true
`[Integer.MIN_VALUE, Integer.MIN_VALUE]`	2 times through loop, false, false
`[Integer.MIN_VALUE, 1]`	2 times through loop, false, true
`[2, 1]`	2 times through loop, true, false
`[1, 2]`	2 times through loop, true, true

Worked example: `indexOf` (bonus)

```
/**
 * @require numbers != null
 * @ensure numbers[\result] = number ||
 *     (\result == -1 &&
 *      \forall int j; 0 <= j < numbers.length ==> numbers[j] != number)
 */
public int indexOf(int[] numbers, int number) {
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] == number) {
            return i;
        }
    }
    return -1;
}
```

Working

Statement coverage: `indexOf([], 1)` and `indexOf([1], 1)`.
Branch coverage: `indexOf([], 1)` and `indexOf([2, 1], 1)`.
Path coverage — note that with an early `return` inside the loop, "2 iterations, `if` true on the first" is unreachable (the method returns immediately), so two of the theoretically-possible paths for a 2-element input can't actually be tested:

Input	Justification
`indexOf([], 42)`	0 times through loop
`indexOf([1], 2)`	1 time through loop, `if` false
`indexOf([1], 1)`	1 time through loop, `if` true
`indexOf([1, 2], 3)`	2 times through loop, false, false
`indexOf([1, 2], 2)`	2 times through loop, false, true
Cannot be tested	2 times through loop, true, false
Cannot be tested	2 times through loop, true, true