Testing

Table of contents

Outline . . . 1
Testing . . . 1
JUnit . . . 2
Black box: the bus example . . . 2
Summary: the black-box method . . . 2
Worked: coverage of `register` . . . 2
Exercise (HW): black box tests for `contains` . . . 3
Further material . . . 4

See csse2002[1] for course logistics.

Outline

1. Levels of Testing
2. Test Frameworks
3. Black Box Testing
4. White Box Testing
5. Test Driven Development (TDD)

Testing

See java-testing[2] — the four levels of testing (unit → integration → system → acceptance), regression testing, black-box vs white/glass-box testing (smoke tests, boundary tests, equivalence classes, code coverage), and TDD.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-testing.pdf

JUnit

See java-junit[3] — `@Test`, `@Before`/`@After`, `Assert` methods, and testing for expected exceptions.

Black box: the bus example

The black-box example is a bus. The *specification is all you get*: `getCurrentPassengers()` and `getAverageCount()` are given as Javadoc only, with their bodies elided (`{...}`). With a maximum capacity of 80, the equivalence classes and their boundary probes are:

Class	Values	Boundary tests
Valid – Empty	0	-1, 0, 1
Valid – Normal	1 – 79	Covered by above and below
Valid – Full capacity	80	79, 80, 81
Invalid	–	-1, 81

This gives the **minimal boundary set `-1, 0, 1, 79, 80, 81`**.

Summary: the black-box method

The black-box method in six points:

1. Black box testing: no knowledge of implementation, only specification.
2. JUnit is a test framework.
3. Tests must not violate preconditions.
4. Start with smoke tests.
5. Expand test suite with boundary tests.
6. Stop when confidence is high.

Worked: coverage of `register`

One short method, with the same question asked at each of the three levels of white-box coverage:

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

```java
public void register(int x) {
    if (x > 0) {
        positives += 1;
    }
    if (x % 2 == 0) {
        evens += 1;
    }
}
```

What input for `x` would give *statement* coverage? Any odd positive value, e.g. `x = 1`, gives only a partial run: `if (x > 0)`, `positives += 1` and the `if (x % 2 == 0)` test are all executed, but `evens += 1` is missed. **`x = 2`** alone executes every statement.

What input for `x` would give *branch* coverage? Branch coverage means each branch is tested for the `true` case and the `false` case. `x = 2` makes both conditions `true`, which is half of it. Adding `x = -1` makes both conditions `false` as well. So **`x = 2` and `x = -1`**: two tests.

What input for `x` would give *path* coverage? Path coverage means every path through the code is traversed. Drawn as a decision tree, the code has four `exit` leaves, so there are four paths, each with its own input:

Path	`x`
`x > 0 && x % 2 == 0`	`2`
`x > 0 && !(x % 2 == 0)`	`1`
`!(x > 0) && x % 2 == 0`	`-2`
`!(x > 0) && !(x % 2 == 0)`	`-1`

The tree, with the four inputs placed on the paths they drive, is under "White box technique: code coverage" in java-testing[4], along with the loop argument that follows it: a 100-iteration loop containing one `if` has 2^{100} paths, so exhaustive path coverage is abandoned in favour of the 0/1/2-iteration approximation.

The worked JUnit white-box example, `DiscountCalculator`, is in java-junit[5]: five reachable returns, five tests.

Exercise (HW): black box tests for `contains`

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

```java
/**
 * Determine if value is contained within the array.
 *
 * @requires array is sorted in ascending order.
 * @return true if value is in the array, false otherwise.
 */
boolean contains(int[] array, int value) {

}
```

Write **black box** tests for `contains` — from this specification only, without seeing the body.

Sketch of an answer

Worked the same way as the bus example:

- `@requires array is sorted in ascending order` is a **precondition**, so summary point 3 applies: an unsorted array is not a legal test case. Passing one proves nothing, whatever `contains` returns.
- **Equivalence classes** on the array: empty; one element; many elements. On the value: present in the array; absent.
- **Boundary probes** (from the collection and whole-number rows of the boundary table in java-testing[6]): empty array; single-element array with the value present and with it absent; value equal to the **first** element; value equal to the **last** element; value falling strictly between two neighbours; value below the minimum and above the maximum; `Integer.MIN_VALUE` / `Integer.MAX_VALUE`.
- `array == null` is a **reference-type** boundary, but the spec neither permits nor forbids it, so whether it is a legal test case depends on reading `@requires` as implying non-null.

Further material

> I hear and I forget. I read and I remember. I do and I understand.
>
> — Confucius (according to popular belief)

The CodeCamp playlist has supplementary material and the Git repo has the code samples. Muddy cards as usual.

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