Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	98%	40/41	71%	10/14	71%	10/14
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	98%	52/53	71%	22/31	73%	22/30
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	100%	27/27	88%	14/16	88%	14/16
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	100%	15/15	100%	7/7	100%	7/7
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	100%	12/12	100%	6/6	100%	6/6
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	100%	15/15	100%	6/6	100%	6/6
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	100%	21/21	75%	6/8	75%	6/8
NumberOfOperatorsCheck.java	0%	0/25	0%	0/8	0%	0/0

Name	Line Coverage		Mutation Coverage		Test Strength	
HalsteadLengthCheck.java	0%	0/41	0%	0/14	0%	0/0
HalsteadVolumeCheck.java	0%	0/53	0%	0/31	0%	0/0
NumberOfCommentLinesCheck.java	0%	0/27	0%	0/16	0%	0/0
NumberOfCommentsCheck.java	0%	0/15	0%	0/7	0%	0/0
NumberOfExpressionsCheck.java	0%	0/12	0%	0/6	0%	0/0
NumberOfLoopsCheck.java	0%	0/15	0%	0/6	0%	0/0
NumberOfOperandsCheck.java	0%	0/21	0%	0/8	0%	0/0
NumberOfOperatorsCheck.java	100%	25/25	75%	6/8	75%	6/8

Fault models factors:

1. HalsteadLengthCheck

Goal (informally): Count total number of operators + operands.

Likely faults:

1.1 **Paired symbols counted wrong**

- Counting `()`, `{}`, `[]` as *two* operators instead of one (or vice versa).
- E.g. `if (a > b) { ... }` – miscounting `(`, `)`, `{`, `}`.

1.2 **Method calls counted incorrectly**

- Counting method names as operands but forgetting to count the `()` or `.` as operators in `obj.method(a)`
- Or counting the same method identifier multiple times per call.

1.3 **Array access miscounted**

- `arr[i]` may treat `[]` as 0 operators, or count index `i` but not the `[ ]`.

1.5 **Literals/keywords not treated consistently**

---

2. `HalsteadVolumeCheck`

Goal (informally): Compute Halstead Volume = `N * log2(n)` (where N, n derived from operators/operands).

Likely faults:

2.1 **Using wrong formula**

- Using natural log or `log10` instead of `log2`.
- Using `n` instead of `log2(n)`.

2.2 **Using wrong N or n**

- Using only operators (or only operands) instead of operators+operands.
- Using distinct counts where total is needed, or vice versa.

2.3 **Integer vs floating-point errors**

- Doing integer division and truncating Volume.
- Rounding too aggressively (e.g., casting to `int` too early).

2.4 **Edge cases: n ≤ 1**

- `log2(1)` or `log2(0)` not handled → division by zero, NaN, or crash.
- Very small methods causing errors or negative values.

---

3. `NumberOfCommentLinesCheck`

Goal: Count the **number of lines that contain comments**.

Likely faults:

3.1 **Inline comments with code miscounted**

- Line like `int x = 0; // comment` may be counted as only code, not a comment line.
- Or counted twice.

3.2 **Multi-line block comments miscounted**

- `/* ... */` spanning several lines:
    - Counting only the first/last line, not the lines in between.
    - Counting blank lines inside the block as comment lines when they might not be desired (depending on your definition).

3.3 **Javadoc comments treated inconsistently**

- Ignoring `/** ... */` completely.
- Double counting Javadoc as both comment and something else.

3.4 **Comments inside strings mistakenly counted**

- `String s = "not // a comment";` – incorrectly treated as a comment line.

3.5 **Edge cases at beginning/end of file**

- Missing comments at top of file or at EOF when no newline is present.

---

4. `NumberOfCommentsCheck`

Goal: Count **individual comments**, not lines (e.g., one `//` vs one `/* */` block).

Likely faults:

4.1 **Counting comment lines instead of comment blocks**

- Treating a 5-line block comment as “5 comments” instead of 1.

4.2 **Merging separate comments on the same line**

- `int x = 0; // first // second` → should be 2 comments, but maybe counted as 1.

4.3 **Ignoring Javadoc**

- Not counting `/** ... */` as comments at all, or counting them separately/inconsistently.

4.4 **Nested / sequential comments miscounted**

- `/* first */ /* second */` on one or multiple lines may be miscounted as 1 or 3 instead of 2.

4.5 **Comment markers inside strings**

- `String s = "/* */";` counted as a comment by mistake.

---

5. `NumberOfExpressionsCheck`

Goal: Count the number of **expressions** in the code (depending on your definition: expression statements, subexpressions, etc.).

Likely faults:

5.1 **Counting only expression *statements***

- Ignoring expressions that are part of larger constructs:
    - Conditions in `if`, `while`, `for`
    - Arguments in method calls
    - Return expressions.

5.2 **Under-counting complex expressions**

- `a = b + c * d;` counted as 1 expression where the intended metric wants multiple (e.g., subexpressions like `b + c * d`, `c * d`).

5.3 **Over-counting operators as separate expressions**

- Each operand or operator treated as its own expression instead of one tree.

---

6. `NumberOfLoopsCheck` (typoed as `NuerOfLoopsCheck`)

Goal: Count loop constructs such as `for`, `while`, `do-while`, and possibly enhanced `for`.

Likely faults:

6.1 **Ignoring enhanced for-loops**

- `for (Item i : items)` not counted as a loop.

6.2 **Miscounting nested loops**

- Only counting the outer loop, ignoring inner nested loops.
- Or double-counting the same loop if visiting AST nodes multiple times.

6.3 **Missing do-while loops**

- Handling `while` and `for` but forgetting `do { ... } while (...)`.

6.4 **Loops in anonymous classes or lambdas ignored**

- A `for` or `while` inside an anonymous inner class body or lambda expression is not counted.

6.5 **False positives from comments or strings**

- `// while (true)` or `"for (int i=0; i<n; i++)"` being counted as real loops.

6.6 **Foreach over streams suspiciously counted**

- Calling `list.forEach(...)` is a library call, not a language loop, but might be incorrectly counted as a “loop”.

---

7. `NumberOfOperandsCheck`

Goal: Count **operands** (variables, literals, constants, field names, etc.).

Likely faults:

7.1 **Not counting literals as operands**

- `42`, `"text"`, `true`, `false`, `null` might be skipped.

7.2 **Static field / qualified names miscounted**

- `ClassName.CONST` → counting only `CONST` or only `ClassName`, or counting both incorrectly.

7.3 **Array and field access**

- `arr[i].field` – miscounting `arr`, `i`, `field` (too many or too few).

7.4 **Parameters vs local variables**

- Parameters not counted at all, or double-counted when used.

7.5 **Method names counted as operands**

- Treating `foo()` as operand `foo` instead of operator for Halstead purposes.

7.6 **Operands inside expressions skipped**

- Operands in conditions, return statements, and lambda bodies not consistently included.

---

8. `NumberOfOperatorsCheck`

Goal: Count **operators** (`+`, `-`, `*`, `/`, `=`, `==`, `&&`, `||`, `? :`, `++`, etc.).

Likely faults:

8.1 **Missing certain operators entirely**

- `&&`, `||`, `?:`, `instanceof`, `new`, `->` (lambda arrow) not recognized.

8.2 **Unary vs binary again**

- `++i`, `i++`, `-a` ignored or miscounted compared to `a + b`.

8.3 **Assignment vs comparison confusion**

- `=` and `==` treated identically or one of them not counted.

8.4 **Separators counted as operators (or not)**

- Commas, semicolons, dots (`.`) sometimes incorrectly included or excluded depending on your definition.

8.5 **Combined/multi-character operators**

- `+=`, `-=`, `>>`, `>>>`, `>>=`, `>>>` miscounted as one or multiple operators incorrectly.

8.6 **Operators in expressions not visited**

- Operators in lambda expressions, generics (`<T>`), annotations, or `for` headers not considered

Observations: Class testing examines class as a whole unit when testing rather than testing the individual methods of the class. It discusses interactions of the states and fields of the class so testing would revolve around this factor. Because we are testing the class as a whole and we do not need to test each individual function, we have an overall reduced number of tests and broader coverage with fewer tests, ensuring that each test covers more than a single method test. Unit tests cannot easily reveal mutations or errors in tests with hidden interactions or dependencies such as one method relying on another method's outputs as its parameters.