Report for Assignment 3

Amanda Zakir, Dawa Arkhang, Edvin Livak, Jafar Mohammadi, Yusuf Can Ekin

February 20, 2026

Contents

List of Figures

List of Tables

1 Project

Name: Apache Commons Lang
URL: `https://github.com/apache/commons-lang`

Apache Commons Lang is a widely used Java library that provides utility functions for core Java classes such as `String`, `Object`, and date/time utilities. The project is mature, actively maintained, and contains more than 10,000 lines of Java code along with an automated test suite.

2 Onboarding experience

The project selected for this assignment is **Apache Commons Lang**. Before proceeding with the complexity analysis and coverage improvement, we conducted a preliminary verification to ensure the project met all assignment criteria.

2.1 Project Verification & Technical Setup

Upon forking and cloning the repository, we first verified the size and language requirements using `cloc` (Count Lines of Code). The project is predominantly written in Java and significantly exceeds the 10,000 LOC threshold required by the assignment.

```
% cloc . --include-lang=Java
    1233 text files.
    1214 unique files.
    1993 files ignored.

-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                           534          18032          73700         100795
-------------------------------------------------------------------------------
SUM:                           534          18032          73700         100795
-------------------------------------------------------------------------------
```

With over 100,000 lines of Java code, the project was deemed suitable in terms of size.

2.2 Build & Test Execution

The project uses **Apache Maven** as its build system. We executed the standard Maven lifecycle to verify that the project builds and tests run out of the box without requiring complex local configuration.

```
mvn clean verify
```

The build process automatically downloaded all necessary dependencies. The test suite, containing thousands of unit tests, executed successfully on the first attempt, confirming that the project is stable and has a mature testing infrastructure.

2.3 Coverage Tool Integration

We observed that the project already includes a `jacoco-maven-plugin` configuration in its `pom.xml`. This simplified the setup significantly, as we did not need to manually inject coverage agents. To generate the coverage reports for our analysis, we used the following command:

```
mvn jacoco:report
```

This generated HTML reports in `target/site/jacoco`, which served as our baseline for identifying functions with low branch coverage.

2.4 Function Selection Challenges

While the technical onboarding was straightforward, identifying suitable functions that satisfied all assignment criteria required a significant amount of analysis and iteration.
The assignment task specified finding functions with high Cyclomatic Complexity Number (CCN) that were also arguably "long" in terms of Lines of Code (LOC). Initially, we used the `lizard` tool to generate a list of candidates, sorting primarily by CCN and NLOC. We identified several large methods in utility classes like `DateUtils` and `StringUtils` that seemed like perfect candidates on paper.
However, a major challenge arose during the initial coverage analysis with JaCoCo. Since `Apache Commons Lang` is a mature and widely used library, we discovered that the vast majority of these "complex and long" functions already possessed extremely high branch coverage (often exceeding 95% or even 100%). For instance, several date parsing functions we initially selected left no room for meaningful coverage improvement, as every edge case was already tested.
Consequently, the group had to iterate on the selection process multiple times. We shifted our strategy from solely looking at the "largest" functions to finding a balance between algorithmic complexity and testability. We analyzed multiple modules to find functions that:

- Exhibited non-trivial control flow (high CCN),
- Were sufficiently long to require analysis (high NLOC),
- **Crucially**, had a branch coverage clearly below 100%, leaving room for the "Coverage Improvement" task.

3 Complexity

3.1 Results for five complex functions

Cyclomatic complexity was measured using `lizard`. We first identified functions of substantial size (based on NLOC and physical length), and from these selected five functions exhibiting relatively high cyclomatic complexity (CCN 15–25). Although the project contains functions with CCN above 50, those extreme outliers are very large utility methods. We chose moderately high-complexity functions to enable reliable manual counting, meaningful coverage improvement, and feasible refactoring analysis.

Function	CCN (Lizard)	NLOC	Manual A	Manual B
EqualsBuilder::reflectionAppend	16	47	16	16
DurationFormatUtils::lexx	23	85	23	23
StringUtils::getLevenshteinDistance	18	61	18	18
StringUtils::convertRemainingAccentCharacters	17	54	17	17
FastDatePrinter::appendFullDigits	15	52	15	15

Manual counting was performed independently by two group members. The results were then compared and reconciled. The only potential ambiguity concerns whether short-circuit boolean operators (`&&`, `||`) should be counted as separate decision points. We followed the same counting convention as Lizard to ensure consistency, which eliminated discrepancies between manual and tool-based measurements.

3.1.1 Manual Counting: DurationFormatUtils::lexx

The manual calculation for `lexx` followed the formula $M = \pi + 1$, where π represents the number of decision points. Following the group's convention to align with `lizard`, we excluded the `switch` header and the `default` case, while counting short-circuit operators.

- **Base Complexity:** +1 (Method entry point).
- **Iteration:** +1 for the `for` loop (line 701) that iterates through the format string.
- **Decision Points (If-statements):** +9 for the various guard clauses and state checks (lines 703, 711, 718, 724, 755, 762, 763, 773, 776).
- **Decision Points (Switch Branches):** +10 for the character-specific cases (lines 710–751) handling date tokens and literals.
- **Logical Operators:** +2 for the short-circuit `&&` operators (lines 703, 763) which create additional logical branches.

Summing these points ($1 + 10 + 2 + 9 = 22$) and adding the base value of 1, the final cyclomatic complexity is **23**.

3.1.2 Manual Counting: StringUtils::convertRemainingAccentCharacters

The manual count for `convertRemainingAccentCharacters` yields a cyclomatic complexity of **17**. This calculation follows the convention of counting the method entry, the loop, and the individual branches within the primary `switch` block.

- **Base Complexity:** +1 (Method entry).
- **Iteration:** +1 for the `for` loop (line 1378) used to traverse the character array.
- **Decision Points (Switch Cases):** +15 for the individual `case` labels (lines 1381–1434). Each case represents a distinct character replacement path.
- **Logical Operators:** +0 (Per the exclusion criteria for this specific measurement).

Summing these points ($1+15=16$) and adding the base value of 1, the final cyclomatic complexity is **17**.

3.1.3 Manual Counting: EqualsBuilder::reflectionAppend

The manual calculation for `reflectionAppend` follows the classical definition $M = \pi + 1$, where π denotes the number of decision (predicates) in the control flow. In alignment with the group convention and the behavior of `lizard`, short-circuit logical operators (`&&`, `||`) were counted as additional decision points, since they introduce separate control-flow edges due to short-circuit evaluation semantics.

- **Base Complexity:** +1 (Method entry point).
- **Decision Points (If-statements):** +8 for top-level and nested `if/else if` conditions controlling equality checks, type compatibility, and guard clauses.
- **Logical Operators:** +5 for short-circuit operators within compound predicates (e.g., null checks and bypass conditions). Each operator introduces an additional branch in the control flow graph.
- **Iteration:** +1 for the `while` loop traversing superclasses.
- **Exception Handling:** +1 for the `catch` block, representing an additional possible execution path.

Summing the decision points ($8+5+1+1=15$) and adding the base value of 1 yields a final cyclomatic complexity of **16**.

3.1.4 Manual Counting: FastDatePrinter::appendFullDigits

The manual count for `appendFullDigits` yields a cyclomatic complexity of **15**. This calculation follows the standard $V(G) = P + 1$ formula, where P represents the number of decision points.

- **Base Complexity: +1** (Method entry).
- **Decision Points (If-statements): +6**
  - **+1** for the top-level `if (value < 10000)` (Line 936).
  - **+3** for nested digit checks: `value < 1000`, `value < 100`, and `value < 10` (Lines 940, 942, 944).
  - **+2** for internal zero checks: `value >= 100` (Line 960) and `value >= 10` (Line 968).
- **Decision Points (Loops): +4**

  - **+1** for the `for` loop (Line 950).
  - **+3** for the `while` loops (Lines 984, 990, and 996).
- **Decision Points (Switch Cases): +4**
  - **+1** for `case 4`, **+1** for `case 3`, **+1** for `case 2`, and **+1** for `case 1` (starting at Line 954).
- **Logical Operators: +0** (No instances of or `||` are found in the function logic).

Summing the decision points $(6+4+4=14)$ and adding the base value of **1** for the method entry results in a final cyclomatic complexity of **15**. This total is consistent with the metrics reported by the Lizard tool and JaCoCo.

3.1.5 Manual Counting: StringUtils::getLevenshteinDistance

The manual count for `getLevenshteinDistance(s, t, threshold)` yields a cyclomatic complexity of **18**. This calculation follows the standard $V(G) = P + 1$ formula, where P represents the number of decision points. In alignment with the group convention, short-circuit logical operators (e.g., `||`) are counted as additional decision points since they introduce separate control-flow edges.

- **Base Complexity: +1** (Method entry).
- **Decision Points (If-statements): +13**
  - **+1** for input validation: `if (s == null || t == null)`.
  - **+1** for invalid threshold: `if (threshold < 0)`.
  - **+1** for empty `s`: `if (n == 0)`.
  - **+1** for nested threshold check under `n == 0`: `if (m <= threshold)`.
  - **+1** for empty `t`: `if (m == 0)`.
  - **+1** for nested threshold check under `m == 0`: `if (n <= threshold)`.
  - **+1** for length-difference cut-off: `if (Math.abs(n - m) > threshold)`.
  - **+1** for optional swap to reduce memory: `if (n > m)`.
  - **+1** for overflow guard inside the outer loop: `if (j > Integer.MAX_VALUE - threshold)`.
  - **+1** for stripe off-table early exit: `if (min > max)`.
  - **+1** for ignoring entry left of stripe: `if (min > 1)`.
  - **+1** for character match vs edit-cost update: `if (s.charAt(i - 1) == jOfT)`.
  - **+1** for final threshold acceptance: `if (p[n] <= threshold)`.
- **Decision Points (Loops): +3**
  - **+1** for the initialization loop: `for (int i = 0; i < boundary; i++)`.
  - **+1** for the outer loop over `t`: `for (int j = 1; j <= m; j++)`.
  - **+1** for the inner loop over the stripe: `for (int i = min; i <= max; i++)`.
- **Logical Operators: +1**
  - **+1** for the short-circuit operator `||` in `(s == null || t == null)`.

Summing the decision points $(13+3+1=17)$ and adding the base value of **1** for the method entry results in a final cyclomatic complexity of **18**. This total is consistent with the metrics reported by the Lizard tool.

3.2 Are the functions just complex, or also long?

The Lizard output reports both Cyclomatic Complexity Number (CCN) and Non-Comment Lines of Code (NLOC). This allows us to distinguish whether high complexity is primarily caused by size or by dense branching logic.

- **EqualsBuilder.reflectionAppend**: 47 NLOC, 61 physical lines, CCN 16.
- **DurationFormatUtils.lexx**: 85 NLOC, 90 physical lines, CCN 23.
- **StringUtils.getLevenshteinDistance**: 61 NLOC, 133 physical lines, CCN 18.
- **StringUtils.convertRemainingAccentCharacters**: 54 NLOC, 66 physical lines, CCN 17.
- **FastDatePrinter.appendFullDigits**: 52 NLOC, 68 physical lines, CCN 15.

Three structural patterns can be observed:

- **lexx** is relatively long and heavily branched due to parsing structured input.
- **getLevenshteinDistance** is moderately long but algorithmically dense due to nested loops.
- **reflectionAppend**, **convertRemainingAccentCharacters**, and **appendFullDigits** are shorter in terms of LOC but contain concentrated decision logic. Their complexity is caused by control-flow density rather than sheer size.

3.3 Purpose of the functions

The selected functions implement non-trivial and domain-specific logic:

- **EqualsBuilder.reflectionAppend**: Uses reflection to determine if two objects are equal by comparing their fields dynamically. The complexity arises from the need to handle security managers, traverse class hierarchies (checking superclasses), filter out transient fields, and manage a registry to prevent infinite recursion in objects with cyclic references.
- **DurationFormatUtils.lexx**: Tokenizes formatting patterns used by duration formatting utilities. It parses literal text and symbolic date/time tokens, requiring multiple conditional branches to distinguish pattern types.
- **FastDatePrinter.appendFullDigits**: Formats numeric date fields with zero-padding rules. The logic depends on field width and value ranges, resulting in several conditional branches.
- **StringUtils.convertRemainingAccentCharacters**: Replaces accented characters with ASCII equivalents. The branching logic reflects explicit case distinctions for different character groups.
- **StringUtils.getLevenshteinDistance**: Computes the edit distance between two strings using a dynamic programming algorithm. The branching structure arises from nested loops and conditional minimum computations.

The relatively high cyclomatic complexity in these functions is partly justified by the need to handle multiple edge cases and alternative execution paths.

3.4 Exceptions

In both manual counting and tool-based measurement, only explicit control-flow constructs were considered: `if`, `else-if`, `for`, `while`, `case`, `catch`, ternary operators, and short-circuit boolean operators (`&&`, `||`).
The selected functions do not contain explicit catch blocks (except for a defensive try-catch in `reflectionAppend`); therefore, exception handling does not significantly contribute to the reported cyclomatic complexity. Implicit runtime exceptions are not reflected in the CCN metric.

3.5 Documentation clarity

The API documentation generally describes the intended functionality of each method. However, it does not fully enumerate all possible execution paths induced by internal branching logic.
In particular, the recursive behavior and class hierarchy traversal in `reflectionAppend` and detailed token parsing behavior in `lexx` require careful inspection of the source code to fully understand all possible outcomes.
Thus, while the high-level purpose is clearly documented, the branching structure and its behavioral implications are not exhaustively described in the documentation.

4 Refactoring

4.1 Plan for refactoring complex code

4.1.1 EqualsBuilder::reflectionAppend

Diagnosis: The `reflectionAppend` method currently has a Cyclomatic Complexity Number (CCN) of 16. The method violates the Single Responsibility Principle by acting as a ”God Method” that combines three distinct and complex operations:

- **Class Resolution:** Determining the common superclass using nested `isInstance` checks.
- **Configuration Filtering:** Checking bypass lists and transient settings using complex boolean logic.
- **Recursive Hierarchy Traversal:** A `while` loop that walks up the inheritance tree.

Refactoring Plan: We plan to apply the ”Extract Method” technique to decompose these responsibilities into specialized helper methods. The main method will act only as a high-level orchestrator.

- **Step 1:** Extract the class resolution logic into `determineCommonClass(Object, Object)`.
- **Step 2:** Extract the bypass check logic into `isBypassRegistered(Class, Class)`.
- **Step 3:** Extract the hierarchy traversal loop into `appendFieldsInHierarchy(Object, Object, Class)`.

4.1.2 DurationFormatUtils::lexx

Diagnosis: the `lexx` method has a Cyclomatic Complexity of 23, which is relatively high for a single parsing function. The primary source of this complexity is not algorithmic necessity, but structural design. The method currently combines multiple responsibilities:

- Character iteration and parsing logic.
- State management (`inliteral, inOptional, buffer, previous`).
- Token emission and merging logic.
- Optional block validation and exception handling.
- Pattern-character mapping via a large `switch` statement.

The elevated complexity is mainly caused by:

- A long `switch` statement containing both structural control characters (`[, ] '`) and pattern tokens (`y, M, d, H, m, s ,S`).
- Nested conditional checks inside multiple `case` branches.
- Interleaving of scanning logic with toke creation logic.
- Mixing state transitions, literal collection and token emission inside the same loop.

The complexity is therefore primarily structural rather than essential. The method implements a small state machine (literal vs non-literal, optional vs non-optional), but this logic can be expressed more clearly with better separation of concerns.

While the current implementation is functionally correct, it is harder to read, maintain, and extend (e.g., handling escaped quotes in the TODO comment).

Refactoring plan: To reduce complexity while preserving behavior, we propose restructuring the method by making a clearer parsing pipeline using a small state object and a few focused helper methods.

Step 1: Introduce an Internal Parser State Object
Create a private static helper class LexxState to encapsulate the mutable state used while parsing:

- `inLiteral`
- `inOptional`
- `optionalIndex`
- `buffer`
- `previous`
- `list`

This removes multiple local variables from the method and centralizes parser state, improving clarity and reducing decision clutter in the main method.

Step 2: Extract literal-mode handling
Extract literal-mode consumption into the function `handleLiteralChar(LexxState st, char ch)`.
This helper checks whether the parser is inside a quoted literal and appends characters to the current literal buffer. It returns a boolean that lets the main loop skip further processing for that character. This reduces branching directly inside `lexx` and clarifies literal behavior.

Step 3: Extract non-literal parsing logic
Extract all processing that happens outside literal-mode into: `handleNonLiteralChar(LexxState st, char ch, int i)`.
This helper performs the branching-heavy logic:

- Opening/closing optional blocks ([and]) with validation.
- Toggling literal mode (’).
- Mapping token letters (`y, M, d, H, m, s, S`) to token values.
- Handling default literal characters. Emitting or merging tokens (including repeated token compression via `previous.increment()`).

This step moves the majority of the original `switch` + nested `if` logic out of `lexx`, making the main method significantly simpler.

Step 4: Extract final validation
Extract end-of-parse validation into: `validateLexxEnd(String format, LexxState st)`
This preserves the same error behavior as before while keeping `lexx` focused only on driving the parse loop.

Expected Outcome:
By distributing the parsing responsibilities across a small internal state object and three focused helper methods the cyclomatic complexity is expected to drop to 10 - 14. The method now primarily acts as a coordinator that iterates through the input string and delegates specific responsibilities (literal handling, non-literal handling, and validation) to dedicated helpers.

This separation of concerns makes the parsing logic easier to understand, maintain, and extend. The state machine behavior becomes more explicit, and complex branching logic is isolated into smaller units. Additionally, the refactored structure enables more targeted unit testing of individual parsing behaviors, improving long-term maintainability and testability.

4.1.3 StringUtils::getLevenshteinDistance(CharSequence, CharSequence, int)

Diagnosis: Using Lizard, `StringUtils::getLevenshteinDistance(CharSequence, CharSequence, int)` has a Cyclomatic Complexity Number (CCN) of **18**. This indicates the method contains many decision points and is harder to maintain and reason about than a typical utility method. The complexity mainly comes from the method combining multiple responsibilities, which violates the Single Responsibility Principle by acting as a "God Method":

- **Input validation and early exits:** Handling `null` inputs, negative thresholds, empty-string cases, and early `-1` returns (e.g., when the length difference exceeds the threshold).
- **Preprocessing / normalization:** Swapping the input sequences so the shorter string is used to reduce memory usage.
- **Core thresholded DP algorithm:** Implementing the banded dynamic programming computation with nested loops, stripe bound calculations, overflow/stripe checks, and the final "within threshold" decision.

Refactoring Plan: We plan to apply the *Extract Method* technique to split these responsibilities into small helper methods. The original method will become a high-level orchestrator that reads clearly and delegates complex logic to focused, testable helpers.

- **Step 1:** Extract validation and trivial-case handling into `validateAndHandleTrivialCases(CharSequence, CharSequence, int)`. This helper handles `null` checks, `threshold < 0`, `n == 0`, `m == 0`, and `abs(n-m) > threshold`.
- **Step 2:** Extract normalization / swap logic into `normalizeByLength(CharSequence, CharSequence)` to ensure the shorter input is used as the first sequence, keeping algorithm assumptions consistent and reducing memory use.
- **Step 3:** Extract stripe bound computation into `computeStripeBounds(int, int, int, int)`. This helper computes the `min/max` band indices (including overflow handling) and detects when the stripe runs off the table.

- **Step 4:** Extract the core DP computation into `computeLevenshteinWithThreshold(char[], char[], int)` to encapsulate the nested-loop update logic and return either the computed distance or `-1`.
- **Step 5:** Keep the public method as an orchestrator that (i) validates/handles trivial cases, (ii) normalizes inputs, and (iii) delegates to the computation helper. This reduces branching in the public method and localizes complexity inside small, testable components.

4.1.4 StringUtils::convertRemainingAccentCharacters

Diagnosis: The method `convertRemainingAccentCharacters` exhibits elevated Cyclomatic Complexity due to a large `switch` statement containing multiple explicit Unicode case branches. Each case represents a separate decision path, increasing the total complexity count.
The high complexity is not inherently necessary for functionality. Rather, it stems from the design choice of handling each special Unicode character through individual switch cases. This results in a method that:

- Combines iteration and transformation logic.
- Encodes character mappings as control flow rather than data.
- Becomes harder to extend and maintain as new mappings are added.

Although the implementation is functionally correct, the structure increases branching and reduces readability.

Refactoring Plan: To reduce complexity while preserving behavior, we propose restructuring the method to separate traversal logic from character mapping logic.

- **Step 1: Replace Switch Statement with Lookup Map** Introduce a static, immutable `Map<Character, Character>` containing all special Unicode replacements. This converts branching logic into data-driven mapping and removes multiple decision points.
- **Step 2: Extract Character Conversion Logic** Introduce a helper method such as `mapAccentCharacter(char)` that performs the lookup and returns either the mapped value or the original character.
- **Step 3: Keep Iteration Logic Focused** The main method should only iterate through the `StringBuilder` and delegate character transformation to the helper method. This enforces separation of concerns.

4.1.5 FastDatePrinter::appendFullDigits

Diagnosis:
The `appendFullDigits` method has a Cyclomatic Complexity of 15. The primary driver of this complexity is the "dual-path" architecture designed for performance optimization (LANG-1248). The method currently manages:

- **Optimized High-Performance Path:** A complex block (lines 936–977) using manual `switch` fall-throughs and nested `if` statements to format values $< 10,000$ without using arrays.

- **Standard Array Path:** A separate logic block (lines 982–998) that uses a temporary `work` array and multiple `while` loops to handle larger values.
- **Redundant Padding:** Zero-padding is implemented twice using different loop structures (`for` loop at line 950 and `while` loop at line 990), leading to code duplication and additional branches.

Refactoring Plan: To reduce complexity and improve maintainability, we propose applying the "Extract Method" technique to separate the optimization logic from the general logic.

- **Step 1: Extract Optimized Formatter:** Move the specialized small-value logic into a private method `appendOptimizedSmallDigits(Appendable, int, int)`. This encapsulates the switch-case complexity.
- **Step 2: Extract General Formatter:** Move the array-based logic for values $\geq 10,000$ into a method `appendStandardLargeDigits(Appendable, int, int)`.
- **Step 3: Consolidate Padding Logic:** Introduce a unified `applyZeroPadding(Appendable, int)` helper to replace the duplicated loops found at lines 950 and 990. This centralizes the padding requirement and removes redundant decision points.

Expected Outcome: By distributing these responsibilities, the Cyclomatic Complexity of the main `appendFullDigits` method will be reduced to 2 or 3, as it will function solely as a router. This separation of concerns makes the performance-optimized code easier to maintain and allows for targeted unit testing of individual formatting paths.

4.2 Estimated impact

4.2.1 EqualsBuilder::reflectionAppend

- **Current CCN:** 16
- **Expectation:** By extracting the three most complex blocks (conditional branching for class type, boolean operators for bypass checks, and the iteration loop), the complexity of the main method is expected to drop significantly.
- **Goal:** Since the identified blocks account for the majority of the decision points, we estimate the reduction will comfortably the required 35% threshold, resulting in a cleaner and more testable main method.

4.2.2 DurationFormatUtils::lexx:

- **Current CCN:** 23
- **Expectation:** The original method concentrated parsing iteration, state transitions, optional block validation, literal handling, and token emission into a single control-flow-heavy loop. By introducing an internal LexxState object and extracting literal handling, non-literal processing, and final validation into helper methods, the top-level lexx method becomes a simple orchestration loop with significantly fewer direct branches.
- **Goal:** Since the refactoring moves a majority of the `case` and `if` statements out of the `lexx` function, which is the complexity heavy part of the function, it is expected that the reduction should exceed the required 35% threshold.

4.2.3 StringUtils::getLevenshteinDistance:

- **Current CCN:** 18
- **Expectation:** The method has many early-return checks and nested decision points inside the stripe-based dynamic programming loop (min/max bounds, stripe-off-table guard, character match vs. edit cost, and array swapping). By applying *Extract Method* to separate (1) input validation and trivial cases, (2) swap/normalization logic, and (3) per-iteration stripe computation, the top-level method becomes a thin orchestrator with fewer direct branches.
- **Goal:** After refactoring, most decision points move into small helper methods, so the CCN of the main public method should drop by at least 35% (target ≤ 11). This improves readability, makes the algorithm easier to reason about, and enables more focused unit tests for each helper.

4.2.4 StringUtils::convertRemainingAccentCharacters:

This refactoring would:

- **Current CCN:** 17
- **Expectation:** After refactoring, it should significantly reduce Cyclomatic complexity by eliminating multiple switch branches, improve readability and maintainability as well as making it easier to map with new characters. it would also Improving testability by allowing isolated testing of the mappping function
- **Goal:** The high complexity of `convertRemainingAccentCharacters` is primarily structural rather than essential. By converting control-flow based mapping into data-driven mapping, this method can be simplified without altering its functional behavior.

4.2.5 FastDatePrinter::appendFullDigits:

- **Current CCN:** 15
- **Expectation:** The method currently handles two distinct execution paths: an optimized switch-case for values $< 10,000$ and a standard loop-based path for larger values. By applying *Extract Method* to separate (1) the switch-case optimization for small integers and (2) the iterative padding logic for values $\geq 10,000$, the complexity of the main method is significantly reduced.
- **Goal:** After refactoring, most decision points move into small helper methods, so the CCN of the main public method should drop by at least 35% (target $\leq 9,75$). This improves readability, makes the algorithm easier to reason about, and enables more focused unit tests for each helper.

5 Carried out refactoring (P+)

5.1 EqualsBuilder::reflectionAppend

The complex logic of `reflectionAppend` was extracted into three specialized helper methods. This addressed the violations of the Single Responsibility Principle and simplified the core logic.
Metrics:

Method	Original CCN	New CCN	Change
`reflectionAppend` (Main)	16	**10**	-6
`determineCommonClass` (Helper)	-	5	New
`isBypassRegistered` (Helper)	-	3	New
`appendFieldsInHierarchy` (Helper)	-	3	New

Analysis:

- **Reduction Rate:** 37.5%.
- **Pull Request:** `https://github.com/yusufcanekin/commons-lang/pull/15/changes`

5.2 DurationFormatUtils::lexx

The `lexx` function was split refactored with the help of helper functions that took on and concentrated the parsing-heavy logic of the original function.
Metrics:

Method	Original CCN	New CCN	Change
`lexx` (Main)	23	**3**	-20
`LexxState` (Helper class)	-	1	New
`handleLiteralChar` (Helper)	-	3	New
`handleNonLiteralChar` (Helper)	-	17	New
`validateLexxEnd` (Helper)	-	3	New

Analysis:

- **Reduction Rate:** 86.96%
- **Pull Request:** `https://github.com/yusufcanekin/commons-lang/pull/26/changes`
- **Important note:** The refactor does not reduce the inherent parsing complexity; it redistributes it. While the sum of CC across helpers is slightly higher, going from 23 to 27, the maximum complexity per method is reduced (23 to 17 for the most complex helper and 23 to 3 for the main entry). This improves maintainability because the most complex logic is isolated, and the main method becomes a clear orchestrator. Furthermore, since each new function will also have a base complexity of 1, each function will add more to the total complexity.

5.3 StringUtils::getLevenshteinDistance

The threshold-based Levenshtein implementation had become difficult to read and maintain due to many early-exit checks, swap/normalization logic, stripe bound calculations, and the dynamic-programming (DP) loop all living in one method. This resulted in a high Cyclomatic Complexity Number (CCN) and made it harder to reason about correctness.

What we refactored: We applied *Extract Method* to separate the distinct responsibilities into small helper methods, while keeping the algorithm and behavior unchanged. The public method is now mainly an orchestrator:

- `validateLevenshteinArgs(...)` handles input validation.
- `levenshteinEarlyReturn(...)` handles empty-string and length-difference early exits.
- `ensureShorterInput(...)` centralizes the swap optimization (ensuring the shorter input is used for the DP row arrays).
- `initPreviousRow(...)` and `initCurrentRow(...)` isolate DP row initialization.
- `computeLevenshteinDistanceWithinThreshold(...)` contains the DP stripe computation.
- `stripeMinIndex(...)` and `stripeMaxIndex(...)` isolate stripe boundary calculations (including overflow-safe max logic).

Metrics:

Method	**Original CCN**	**New CCN**	**Change**
`getLevenshteinDistance(..., threshold)` (Main)	18	**4**	-14
`validateLevenshteinArgs` (Helper)	-	4	New
`levenshteinEarlyReturn` (Helper)	-	6	New
`ensureShorterInput` (Helper)	-	2	New
`initPreviousRow` (Helper)	-	2	New
`initCurrentRow` (Helper)	-	1	New
`computeLevenshteinDistanceWithinThreshold` (Helper)	-	9	New
`stripeMinIndex` (Helper)	-	1	New
`stripeMaxIndex` (Helper)	-	2	New

Analysis:

- **Reduction Rate:** $\frac{18-4}{18} \approx 77.8\%$.
- The main method now clearly communicates the algorithmic stages (validate $\rightarrow$ early-exit $\rightarrow$ normalize $\rightarrow$ init $\rightarrow$ compute).
- The complex stripe-based DP logic is isolated in a single helper, making the overall implementation easier to test and review.
- **Pull Request:** `https://github.com/yusufcanekin/commons-lang/pull/18/files`

5.4 StringUtils::convertRemainingAccentCharacters

The original `convertRemainingAccentCharacters` method used switch statement direcltly on each character to converted Unicode character to its ASCII character. While functional, the logic had a Cyclomatic Complexity (CC) of 17, due to many case branches which makes the code hard to read and maintain. The method was refactored to use a `HashMap` (`ACCENT_MAP`) that maps accented characters to their ASCII replacements. The method now iterates through the input StringBuilder and replaces each character if a corresponding entry exists in the map. Characters not present in the map are left unchanged.
Metrics:

Method	**Original CCN**	**New CCN**	**Change**
`convertRemainingAccentCharacters` (Main)	17	**3**	-14

- **Reduction Rate:** 82.4%.
- **Pull Request:** `https://github.com/yusufcanekin/commons-lang/pull/29`

5.5 FastDatePrinter::appendFullDigits

The complex logic of `appendFullDigits` was extracted into three specialized helper methods. This addressed the violations of the Single Responsibility Principle and simplified the core logic.
Metrics:

Method	**Original CCN**	**New CCN**	**Change**
`appendFullDigits` (Main)	15	**5**	-10
`appendOptimizedSmallDigits` (Helper)	-	7	New
`appendStandardLargeDigits` (Helper)	-	3	New
`applyZeroPadding` (Helper)	-	2	New

Analysis:

- **Reduction Rate:** 66.6%.
- **Pull Request:** `https://github.com/yusufcanekin/commons-lang/pull/23`

---

6 Coverage

6.1 Tools

JaCoCo was used to measure branch coverage. It was well documented and easy to integrate, as it was already configured in the project's Maven build file.
Coverage reports were generated automatically after running:

```
mvn clean test jacoco:report
```

6.2 Your own coverage tool

We implemented a custom branch coverage tool through manual source code instrumentation. The tool is centralized in a class named `BranchCoverage`, which manages branch registries and execution hits using thread-safe `ConcurrentHashMap` structures.

Patch and Instrumentation: The instrumented code and the DIY tool can be inspected in the following branch: `https://github.com/yusufcanekin/commons-lang/blob/diy-coverage/src/main/java/org/apache/commons/lang3/BranchCoverage.java`
Pull Request link below: `https://github.com/yusufcanekin/commons-lang/pull/20/changes`

Mechanism and Accuracy: The tool supports explicit branch outcomes by assigning unique IDs to each branch. It uses a `ShutdownHook` to automatically generate a report when the JVM terminates, writing the results to both the console and a file (`target/diy-coverage.txt`). The output is highly accurate for the instrumented lines, as it provides a binary [HIT] or [MISS] status for every registered branch ID.

However, Edvin had made a simpler branch coverage tool for his function before the BranchCoverage class had been created. Therefore his branch coverage was done with that tool instead. The implementation of this tool followed the instructions given in the task.pdf, where all branches in `lexx` were given an ID, at each branch outcome a flag was added. The flag data was kept using a simple data structure. With the help of the test file for the file `lexx` resided in, JUnit's @AfterAll was used to print all hits.
This code was also merged to the `diy-coverage` branch along with the other coverage tool.
Pull request: `https://github.com/yusufcanekin/commons-lang/pull/24/changes`

6.3 Evaluation

1. **Detail of Measurement:** Our tool provides granular branch-level coverage. By using the `branch(baseId, condition)` helper, we capture both the `true` and `false` outcomes of `if` statements separately, which aligns with the formal definition of branch coverage.

2. **Limitations:** The primary limitation is that the tool requires **manual instrumentation**. Unlike JaCoCo, it cannot automatically detect branches; every outcome must be manually tagged. It does not natively support ternary operators unless they are manually broken down into method calls, and it does not capture implicit exception branches (e.g., a `NullPointerException` skipping the rest of a block) unless specific `try-catch` blocks are instrumented.

3. **Consistency with Existing Tools:** The results are **consistent but not identical** to JaCoCo . While both tools identified the same high-level "weak spots" (e.g., the unreachable null check in `reflectionAppend`), our DIY tool reported higher local coverage percentages. This is because JaCoCo analyzes the entire class's bytecode, including generated methods and implicit branches we did not instrument . However, for the specific decision points we manually tagged, the [HIT]/[MISS] status mirrored JaCoCo's branch analysis, confirming the reliability of our manual logic.

7 Coverage improvement

Branch coverage was initially measured using JaCoCo. The following branch coverage values were observed before adding new test cases:

Function	CCN	Initial Coverage	Final Coverage
EqualsBuilder::reflectionAppend	16	85%	96%
DurationFormatUtils::lexx	23	97%	100%
FastDatePrinter::appendFullDigits	15	92%	96%
StringUtils::convertRemainingAccentCharacters	17	83%	100%
StringUtils::getLevenshteinDistance	18	94%	100%

Although none of the selected functions had 100% branch coverage, all exhibited uncovered execution paths, particularly in edge-case logic.
Four new test cases per function were added to improve coverage.

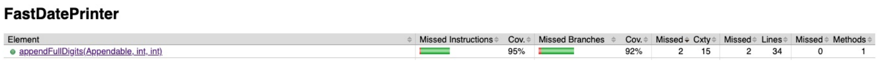
7.1 EqualsBuilder::reflectionAppend (Initial: 85%)

Coverage analysis revealed that **4 specific branches** were missed by the existing test suite.
New tests were added to target these specific scenarios:

- **Bypass Logic Combinations:** A test was added to cover complex boolean conditions (AND/OR logic) in the `bypassReflectionClasses` check. This covered scenarios where (GrandParent → Parent → Child) the list is null, empty, or contains specific class matches.
- **Hierarchy Traversal & Shadowing:** A test using a 3-level inheritance hierarchy (GrandParent → Parent → Child) with shadowed private fields was added. This forced the execution of the `while` loop that traverses superclasses, ensuring the reflection logic correctly accesses hidden fields in parent classes.
- **Transient Field Logic:** A test was added to explicitly toggle the `testTransients` flag, forcing the code to enter and skip the transient field filtering branch.
- **Loop Boundary & Path Coverage:** A test was introduced to verify the `reflectUpToClass` functionality. By explicitly setting a stop class, we validated the execution path where the hierarchy traversal terminates early, ensuring the loop condition `testClass != reflectUpToClass` is correctly evaluated.

Result: The new tests covered **3 out of the 4** missing branches and verified an additional execution path for loop termination. The remaining uncovered branch was identified as **Unreachable Code (Dead Code)** due to a logically impossible condition in the source code.

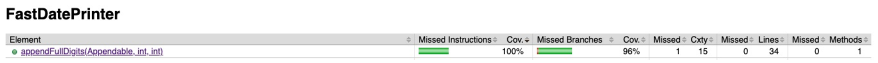
Pull Request: `https://github.com/yusufcanekin/commons-lang/pull/13/changes`



FastDatePrinter

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
appendFullDigits(Appendable, int, int)		95%		92%	2	15	2	34	0	1

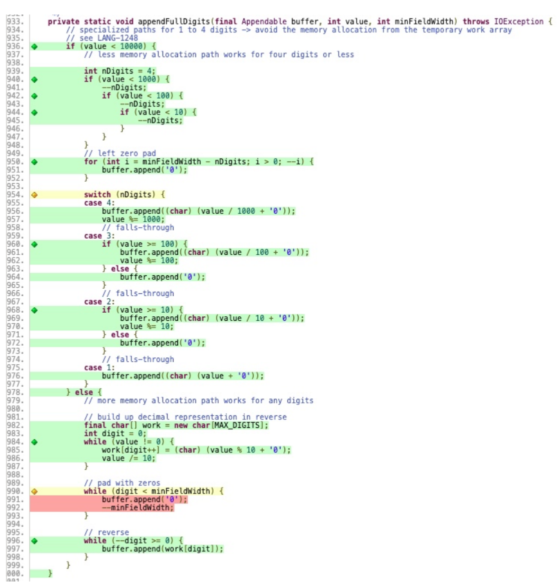
(a) Initial coverage (92%)



FastDatePrinter

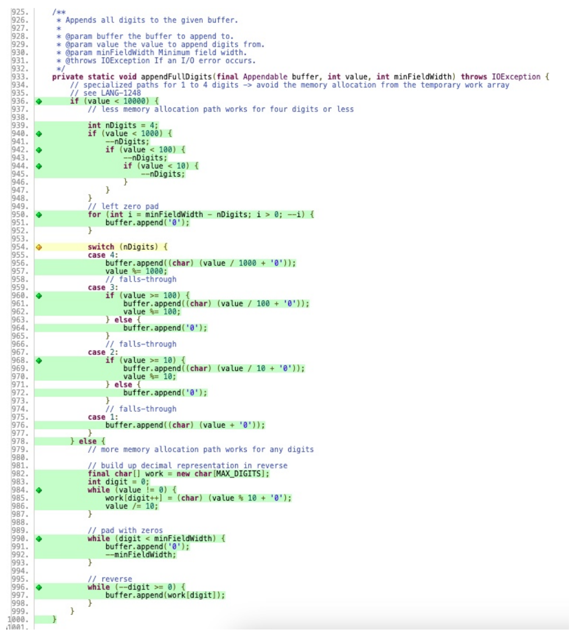
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
appendFullDigits(Appendable, int, int)		100%		96%	1	15	0	34	0	1

(b) Improved coverage (96%)



```java
    private static void appendFullDigits(final Appendable buffer, int value, int minFieldWidth) throws IOException {
        // specialized paths for 1 to 4 digits -> avoid the memory allocation from the temporary work array
        // see LANG-1248
        if (value < 10000) {
            // less memory allocation path works for four digits or less

            int nDigits = 4;
            if (value < 1000) {
                --nDigits;
                if (value < 100) {
                    --nDigits;
                    if (value < 10) {
                        --nDigits;
                    }
                }
            }
            // left zero pad
            for (int i = minFieldWidth - nDigits; i > 0; --i) {
                buffer.append('0');
            }

            switch (nDigits) {
            case 4:
                buffer.append((char) (value / 1000 + '0'));
                value %= 1000;
                // falls-through
            case 3:
                if (value >= 100) {
                    buffer.append((char) (value / 100 + '0'));
                    value %= 100;
                } else {
                    buffer.append('0');
                }
                // falls-through
            case 2:
                if (value >= 10) {
                    buffer.append((char) (value / 10 + '0'));
                    value %= 10;
                } else {
                    buffer.append('0');
                }
                // falls-through
            case 1:
                buffer.append((char) (value + '0'));
            }
        } else {
            // more memory allocation path works for any digits

            // build up decimal representation in reverse
            final char[] work = new char[MAX_DIGITS];
            int digit = 0;
            while (value != 0) {
                work[digit++] = (char) (value % 10 + '0');
                value /= 10;
            }

            // pad with zeros
            while (digit < minFieldWidth) {
                buffer.append('0');
                --minFieldWidth;
            }

            // reverse
            while (--digit >= 0) {
                buffer.append(work[digit]);
            }
        }
    }
```

(c) Initially missed branches



```java
    /**
     * Appends all digits to the given buffer.
     *
     * @param buffer the buffer to append to.
     * @param value the value to append digits from.
     * @param minFieldWidth Minimum field width.
     * @throws IOException If an I/O error occurs.
     */
    private static void appendFullDigits(final Appendable buffer, int value, int minFieldWidth) throws IOException {
        // specialized paths for 1 to 4 digits -> avoid the memory allocation from the temporary work array
        // see LANG-1248
        if (value < 10000) {
            // less memory allocation path works for four digits or less

            int nDigits = 4;
            if (value < 1000) {
                --nDigits;
                if (value < 100) {
                    --nDigits;
                    if (value < 10) {
                        --nDigits;
                    }
                }
            }
            // left zero pad
            for (int i = minFieldWidth - nDigits; i > 0; --i) {
                buffer.append('0');
            }

            switch (nDigits) {
            case 4:
                buffer.append((char) (value / 1000 + '0'));
                value %= 1000;
                // falls-through
            case 3:
                if (value >= 100) {
                    buffer.append((char) (value / 100 + '0'));
                    value %= 100;
                } else {
                    buffer.append('0');
                }
                // falls-through
            case 2:
                if (value >= 10) {
                    buffer.append((char) (value / 10 + '0'));
                    value %= 10;
                } else {
                    buffer.append('0');
                }
                // falls-through
            case 1:
                buffer.append((char) (value + '0'));
            }
        } else {
            // more memory allocation path works for any digits

            // build up decimal representation in reverse
            final char[] work = new char[MAX_DIGITS];
            int digit = 0;
            while (value != 0) {
                work[digit++] = (char) (value % 10 + '0');
                value /= 10;
            }

            // pad with zeros
            while (digit < minFieldWidth) {
                buffer.append('0');
                --minFieldWidth;
            }

            // reverse
            while (--digit >= 0) {
                buffer.append(work[digit]);
            }
        }
    }
```

(d) Final missed branches

Figure 1: Comparison of code coverage before and after adding new test cases for FastDatePrinter::appendFullDigits. Top: Overall percentage improvement. Bottom: Detailed branch analysis.

7.2 DurationFormatUtils::lexx (Initial: 97%)

Coverage analysis using the DIY tool showed that **2 specific branches** were missed by the existing test suite. The scenarios were covered with two new tests, one coverage test, and 2 additional path coverage tests were added to reinforce the testing:

- **Empty Format Handling:** A test (`testLexxEmptyFormatReturnsEmptyTokens`) was added to verify that an empty format string correctly skips the parsing loop and returns an empty token array. This ensures execution of the branch where the loop body is never entered.
- **Buffer Reuse for Consecutive Literals:** A test (`testLexxDefaultReusesExistingBuffer`) was added to confirm that consecutive non-pattern characters are appended to the same literal token rather than creating multiple tokens. This specifically triggers the branch where the existing buffer is reused.
- **Nested Optional Block Validation:** A dedicated test (`testLexxNestedOptionalThrows`) was added to explicitly trigger the exception path for nested optional blocks ([[). This verifies

correct validation logic and confirms that the exception branch is reached with the correct index.

- **Repeated Pattern Token Merging:** A test (`testLexxRepeatedPatternLettersMergeToSingleToken`) verifies that repeated pattern characters (e.g., "yyyy") are merged into a single token with an incremented count, covering the branch where previous.increment() is executed.
- **Quoted Literal Parsing:** A test (`testLexxLiteralInQuotesBecomesSingleLiteralToken`) ensures that quoted text is treated as a literal token and correctly handled in literal mode.

Result: The new tests covered the remaining two missing branches, while also adding path coverage to make the testing more robust.
Pull Request: `https://github.com/yusufcanekin/commons-lang/pull/25/changes`

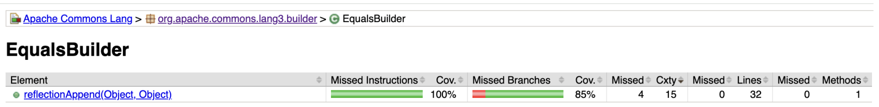
7.3 FastDatePrinter::appendFullDigits (Initial: 92%)

Coverage analysis revealed that **2 specific branches** were missed by the existing test suite. New tests were added to target these specific scenarios:

- **Large Value with Zero Padding:** The test `testLargeValueWithZeroPadding` targets the "True" branch of the padding loop (line 990) and the subsequent buffer append operations (lines 991-992). By setting a minimum field width of **10** for the value **10001**, the test forces the execution of the `while` loop to append five leading zeros, ensuring the requirement for padding large values is fully satisfied.
- **Large Value Path Coverage (No Padding):** The test `testLargeValueWithoutZeroPadding` was implemented to satisfy requirement **REQ-LARGE-VAL**, ensuring exact-width formatting for large values. By utilizing the value **10001** with a pattern width of **5**, the test forces the padding loop condition to be **false**. This successfully exercises the execution path where the loop is skipped, resolving the "Yellow Diamond" partial coverage at the loop entry point.
- **Internal Zero Preservation (Sparse Path):** The test `testPathSparseInternalZeros` validates the "Hole in the Middle" scenario using the value **1001**. This test forces the execution of the `else` branches (previously lines 963 and 971), ensuring the "Backup" logic correctly triggers to plug empty hundreds or tens places with '0' instead of skipping them in the output buffer.
- **Full Digit Calculation (Dense Path):** The test `testPathDenseNonZeroDigits` exercises the "Full House" scenario using the value **1234**. This test ensures that when every digit position is non-zero, the code correctly "opens the gates" by satisfying the `if` blocks (previously lines 960 and 968), verifying the standard arithmetic path for multi-digit formatting.

  Result: The new tests covered **1 out of the 2** missing branches and verified an additional execution path for loop termination. Path coverage analysis of the optimized switch-case block revealed a permanent "Yellow Diamond" on the `switch` statement itself. JaCoCo flags this as partially covered because it anticipates a default branch or a scenario where the switch value does not match any of the provided cases.

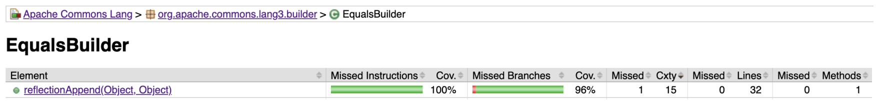
  Pull Request: `https://github.com/yusufcanekin/commons-lang/pull/22`


Apache Commons Lang > org.apache.commons.lang3.builder > EqualsBuilder

EqualsBuilder

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
reflectionAppend(Object, Object)	🟩	100%	🟥🟩	85%	4	15	0	32	0	1

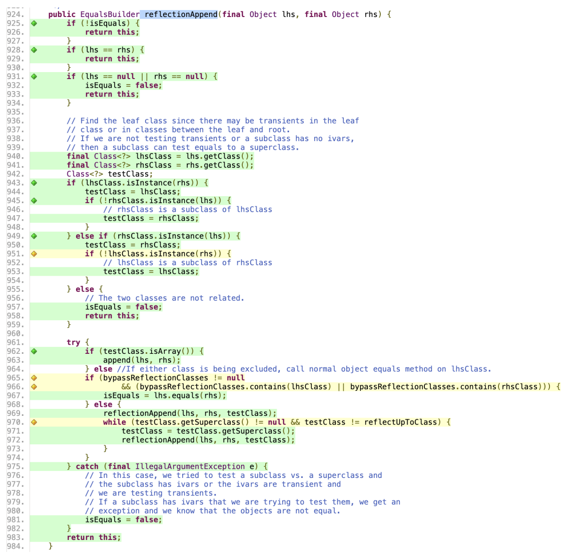
(a) Initial coverage (85%)


Apache Commons Lang > org.apache.commons.lang3.builder > EqualsBuilder

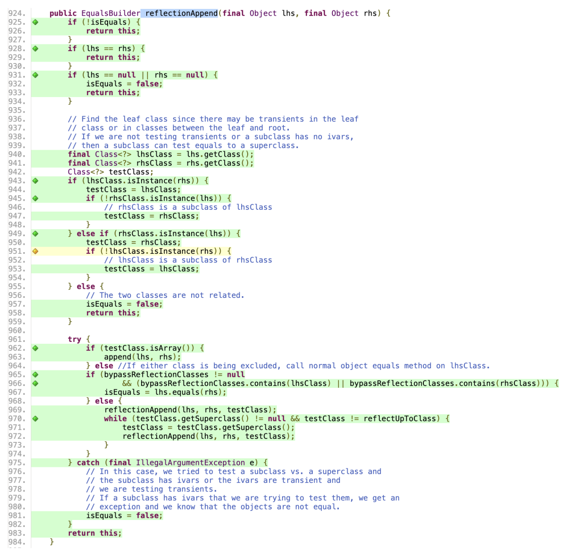
EqualsBuilder

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
reflectionAppend(Object, Object)	🟩	100%	🟩	96%	1	15	0	32	0	1

(b) Improved coverage (96%)


```java
    public EqualsBuilder reflectionAppend(final Object lhs, final Object rhs) {
        if (!isEquals) {
            return this;
        }
        if (lhs == rhs) {
            return this;
        }
        if (lhs == null || rhs == null) {
            isEquals = false;
            return this;
        }

        // Find the leaf class since there may be transients in the leaf
        // class or in classes between the leaf and root.
        // If we are not testing transients or a subclass has no ivars,
        // then a subclass can test equals to a superclass.
        final Class<?> lhsClass = lhs.getClass();
        final Class<?> rhsClass = rhs.getClass();
        Class<?> testClass;
        if (lhsClass.isInstance(rhs)) {
            testClass = lhsClass;
            if (!rhsClass.isInstance(lhs)) {
                // rhsClass is a subclass of lhsClass
                testClass = rhsClass;
            }
        } else if (rhsClass.isInstance(lhs)) {
            testClass = rhsClass;
            if (!lhsClass.isInstance(rhs)) {
                // lhsClass is a subclass of rhsClass
                testClass = lhsClass;
            }
        } else {
            // The two classes are not related.
            isEquals = false;
            return this;
        }

        try {
            if (testClass.isArray()) {
                append(lhs, rhs);
            } else //If either class is being excluded, call normal object equals method on lhsClass.
            if (bypassReflectionClasses != null
                    && (bypassReflectionClasses.contains(lhsClass) || bypassReflectionClasses.contains(rhsClass))) {
                isEquals = lhs.equals(rhs);
            } else {
                reflectionAppend(lhs, rhs, testClass);
                while (testClass.getSuperclass() != null && testClass != reflectUpToClass) {
                    testClass = testClass.getSuperclass();
                    reflectionAppend(lhs, rhs, testClass);
                }
            }
        } catch (final IllegalArgumentException e) {
            // In this case, we tried to test a subclass vs. a superclass and
            // the subclass has ivars or the ivars are transient and
            // we are testing transients.
            // If a subclass has ivars that we are trying to test them, we get an
            // exception and we know that the objects are not equal.
            isEquals = false;
        }
        return this;
    }
```

(c) Initially missed branches


```java
    public EqualsBuilder reflectionAppend(final Object lhs, final Object rhs) {
        if (!isEquals) {
            return this;
        }
        if (lhs == rhs) {
            return this;
        }
        if (lhs == null || rhs == null) {
            isEquals = false;
            return this;
        }

        // Find the leaf class since there may be transients in the leaf
        // class or in classes between the leaf and root.
        // If we are not testing transients or a subclass has no ivars,
        // then a subclass can test equals to a superclass.
        final Class<?> lhsClass = lhs.getClass();
        final Class<?> rhsClass = rhs.getClass();
        Class<?> testClass;
        if (lhsClass.isInstance(rhs)) {
            testClass = lhsClass;
            if (!rhsClass.isInstance(lhs)) {
                // rhsClass is a subclass of lhsClass
                testClass = rhsClass;
            }
        } else if (rhsClass.isInstance(lhs)) {
            testClass = rhsClass;
            if (!lhsClass.isInstance(rhs)) {
                // lhsClass is a subclass of rhsClass
                testClass = lhsClass;
            }
        } else {
            // The two classes are not related.
            isEquals = false;
            return this;
        }

        try {
            if (testClass.isArray()) {
                append(lhs, rhs);
            } else //If either class is being excluded, call normal object equals method on lhsClass.
            if (bypassReflectionClasses != null
                    && (bypassReflectionClasses.contains(lhsClass) || bypassReflectionClasses.contains(rhsClass))) {
                isEquals = lhs.equals(rhs);
            } else {
                reflectionAppend(lhs, rhs, testClass);
                while (testClass.getSuperclass() != null && testClass != reflectUpToClass) {
                    testClass = testClass.getSuperclass();
                    reflectionAppend(lhs, rhs, testClass);
                }
            }
        } catch (final IllegalArgumentException e) {
            // In this case, we tried to test a subclass vs. a superclass and
            // the subclass has ivars or the ivars are transient and
            // we are testing transients.
            // If a subclass has ivars that we are trying to test them, we get an
            // exception and we know that the objects are not equal.
            isEquals = false;
        }
        return this;
    }
```

(d) Final missed branches (only dead code)

Figure 2: Comparison of code coverage before and after adding new test cases for EqualsBuilder::reflectionAppend. Top: Overall percentage improvement. Bottom: Detailed branch analysis.

7.4 StringUtils::convertRemainingAccentCharacters (Initial: 83%)

Coverage analysis showed that 3 out of 16 branches inside `convertRemainingAccentCharacters` were not executed when testing only through the public method `stripAccents`.

To increase branch coverage, dedicated unit tests were added for each missing case. Each test directly invoked `convertRemainingAccentCharacters` with a `StringBuilder` containing the corresponding Unicode character and asserted that the correct ASCII replacement was performed.

This ensured execution of:

– Branch 8 (`\u1DA4` → 'i')

– Branch 9 (`\u1DA7` → 'I')

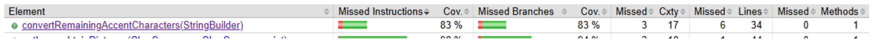
– Branch 13 (`\u1DB6` → 'u')

In addition to individual branch tests, a dedicated test case was constructed to achieve path coverage across these branches within a single execution. A `StringBuilder` containing the sequence `\u1DA4\u1DA7\u0166` was passed to the method.

Since the method iterates over each character and processes them independently through a `switch` statement, this input caused the execution path $8 \rightarrow 9 \rightarrow 13$ during one invocation of the method. Although the branches are not nested, this demonstrates coverage of a specific combination of branches in a single execution, thereby extending the analysis beyond pure branch coverage to path coverage.

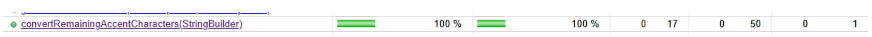
Result: The newly added tests successfully executed the previously uncovered branches, increasing overall structural branch coverage to 100%.

Pull Request: `https://github.com/yusufcanekin/commons-lang/pull/28`



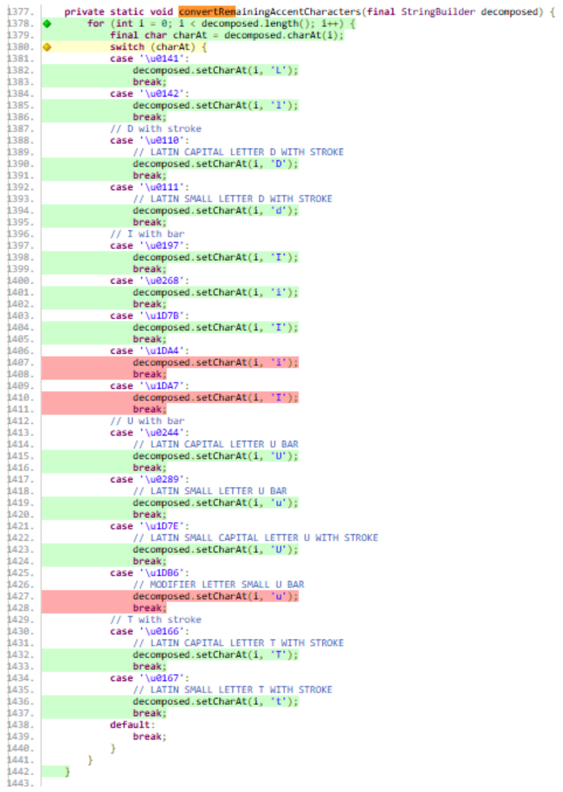
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
convertRemainingAccentCharacters(StringBuilder)	🟥🟩	83 %	🟥🟩	83 %	3	17	6	34	0	1

(a) Initial coverage (83%)



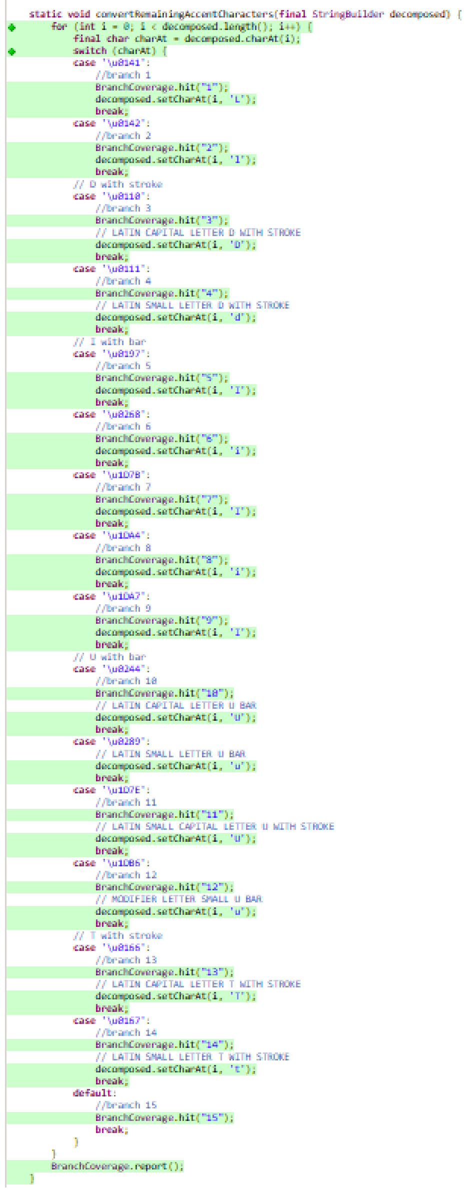
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
convertRemainingAccentCharacters(StringBuilder)	🟩	100 %	🟩	100 %	0	17	0	50	0	1

(b) Improved coverage (100%)



```java
    private static void convertRemainingAccentCharacters(final StringBuilder decomposed) {
        for (int i = 0; i < decomposed.length(); i++) {
            final char charAt = decomposed.charAt(i);
            switch (charAt) {
            case '\u0141':
                decomposed.setCharAt(i, 'L');
                break;
            case '\u0142':
                decomposed.setCharAt(i, 'l');
                break;
            // D with stroke
            case '\u0110':
                // LATIN CAPITAL LETTER D WITH STROKE
                decomposed.setCharAt(i, 'D');
                break;
            case '\u0111':
                // LATIN SMALL LETTER D WITH STROKE
                decomposed.setCharAt(i, 'd');
                break;
            // I with bar
            case '\u0197':
                decomposed.setCharAt(i, 'I');
                break;
            case '\u0268':
                decomposed.setCharAt(i, 'i');
                break;
            case '\u1D7B':
                decomposed.setCharAt(i, 'I');
                break;
            case '\u1DA4':
                decomposed.setCharAt(i, 'i');
                break;
            case '\u1DA7':
                decomposed.setCharAt(i, 'I');
                break;
            // U with bar
            case '\u0244':
                // LATIN CAPITAL LETTER U BAR
                decomposed.setCharAt(i, 'U');
                break;
            case '\u0289':
                // LATIN SMALL LETTER U BAR
                decomposed.setCharAt(i, 'u');
                break;
            case '\u1D7E':
                // LATIN SMALL CAPITAL LETTER U WITH STROKE
                decomposed.setCharAt(i, 'U');
                break;
            case '\u1DB6':
                // MODIFIER LETTER SMALL U BAR
                decomposed.setCharAt(i, 'u');
                break;
            // T with stroke
            case '\u0166':
                // LATIN CAPITAL LETTER T WITH STROKE
                decomposed.setCharAt(i, 'T');
                break;
            case '\u0167':
                // LATIN SMALL LETTER T WITH STROKE
                decomposed.setCharAt(i, 't');
                break;
            default:
                break;
            }
        }
    }
```

(c) Initially missed branches



```java
    static void convertRemainingAccentCharacters(final StringBuilder decomposed) {
        for (int i = 0; i < decomposed.length(); i++) {
            final char charAt = decomposed.charAt(i);
            switch (charAt) {
            case '\u0141':
                //branch 1
                BranchCoverage.hit("1");
                decomposed.setCharAt(i, 'L');
                break;
            case '\u0142':
                //branch 2
                BranchCoverage.hit("2");
                decomposed.setCharAt(i, 'l');
                break;
            // D with stroke
            case '\u0110':
                //branch 3
                BranchCoverage.hit("3");
                // LATIN CAPITAL LETTER D WITH STROKE
                decomposed.setCharAt(i, 'D');
                break;
            case '\u0111':
                //branch 4
                BranchCoverage.hit("4");
                // LATIN SMALL LETTER D WITH STROKE
                decomposed.setCharAt(i, 'd');
                break;
            // I with bar
            case '\u0197':
                //branch 5
                BranchCoverage.hit("5");
                decomposed.setCharAt(i, 'I');
                break;
            case '\u0268':
                //branch 6
                BranchCoverage.hit("6");
                decomposed.setCharAt(i, 'i');
                break;
            case '\u1D7B':
                //branch 7
                BranchCoverage.hit("7");
                decomposed.setCharAt(i, 'I');
                break;
            case '\u1DA4':
                //branch 8
                BranchCoverage.hit("8");
                decomposed.setCharAt(i, 'i');
                break;
            case '\u1DA7':
                //branch 9
                BranchCoverage.hit("9");
                decomposed.setCharAt(i, 'I');
                break;
            // U with bar
            case '\u0244':
                //branch 10
                BranchCoverage.hit("10");
                // LATIN CAPITAL LETTER U BAR
                decomposed.setCharAt(i, 'U');
                break;
            case '\u0289':
                // LATIN SMALL LETTER U BAR
                decomposed.setCharAt(i, 'u');
                break;
            case '\u1D7E':
                //branch 11
                BranchCoverage.hit("11");
                // LATIN SMALL CAPITAL LETTER U WITH STROKE
                decomposed.setCharAt(i, 'U');
                break;
            case '\u1DB6':
                //branch 12
                BranchCoverage.hit("12");
                // MODIFIER LETTER SMALL U BAR
                decomposed.setCharAt(i, 'u');
                break;
            // T with stroke
            case '\u0166':
                //branch 13
                BranchCoverage.hit("13");
                // LATIN CAPITAL LETTER T WITH STROKE
                decomposed.setCharAt(i, 'T');
                break;
            case '\u0167':
                //branch 14
                BranchCoverage.hit("14");
                // LATIN SMALL LETTER T WITH STROKE
                decomposed.setCharAt(i, 't');
                break;
            default:
                //branch 15
                BranchCoverage.hit("15");
                break;
            }
        }
        BranchCoverage.report();
    }
```

(d) Final missed branches

Figure 3: Comparison of code coverage before and after adding new test cases for convertRemainingAccentCharacters. Top: Overall percentage improvement. Bottom: Detailed branch analysis.

7.5 StringUtils::getLevenshteinDistance (Initial: 94%)

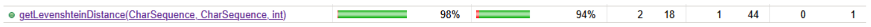
Coverage analysis of the threshold-based Levenshtein implementation revealed that **2 specific branches** were not exercised by the existing test suite. These missed paths were related to (1) the early-exit behavior when one input is empty and exceeds the threshold, and (2) the defensive condition where the diagonal stripe window runs off the DP table (`min > max`).

To improve coverage, dedicated unit tests were added to target these scenarios:

- **Empty-string boundary beyond threshold:** A test was added where the left input is empty (`n=0`), the right input length is greater than the threshold, and the expected return is `-1`. This explicitly covers the branch where `m > threshold` for the `n == 0` case.
- **Stripe window runs off the DP table (defensive branch):** A test was introduced to trigger the internal condition `min > max`, forcing an immediate `-1` return. This validates the defensive path that prevents invalid DP stripe processing.

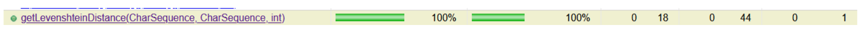
Result: The added tests executed all previously missed branches, improving the structural branch coverage from **94%** to **100%** in the JaCoCo report.

Pull Request: `https://github.com/yusufcanekin/commons-lang/pull/17/files`



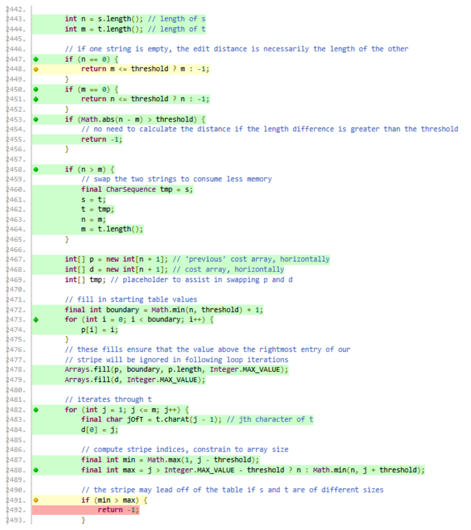
```
getLevenshteinDistance(CharSequence, CharSequence, int)    98%    94%    2    18    1    44    0    1
```

(a) Initial coverage (94%)



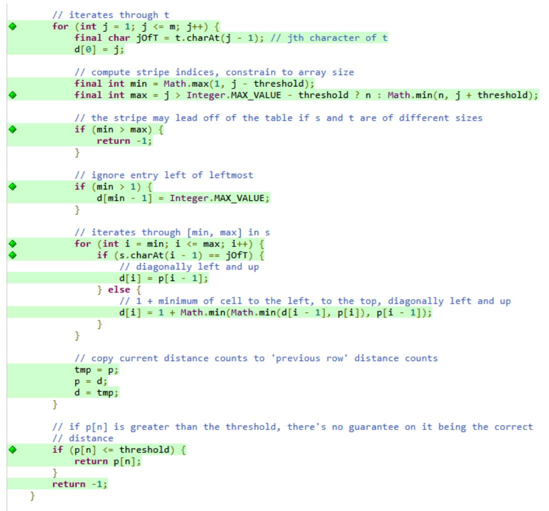
```
getLevenshteinDistance(CharSequence, CharSequence, int)    100%    100%    0    18    0    44    0    1
```

(b) Improved coverage (100%)



```java

        int n = s.length(); // length of s
        int m = t.length(); // length of t

        // if one string is empty, the edit distance is necessarily the length of the other
        if (n == 0) {
            return m <= threshold ? m : -1;
        }
        if (m == 0) {
            return n <= threshold ? n : -1;
        }
        if (Math.abs(n - m) > threshold) {
            // no need to calculate the distance if the length difference is greater than the threshold
            return -1;
        }

        if (n > m) {
            // swap the two strings to consume less memory
            final CharSequence tmp = s;
            s = t;
            t = tmp;
            n = m;
            m = t.length();
        }

        int[] p = new int[n + 1]; // 'previous' cost array, horizontally
        int[] d = new int[n + 1]; // cost array, horizontally
        int[] tmp; // placeholder to assist in swapping p and d

        // fill in starting table values
        final int boundary = Math.min(n, threshold) + 1;
        for (int i = 0; i < boundary; i++) {
            p[i] = i;
        }
        // these fills ensure that the value above the rightmost entry of our
        // stripe will be ignored in following loop iterations
        Arrays.fill(p, boundary, p.length, Integer.MAX_VALUE);
        Arrays.fill(d, Integer.MAX_VALUE);

        // iterates through t
        for (int j = 1; j <= m; j++) {
            final char jOfT = t.charAt(j - 1); // jth character of t
            d[0] = j;

            // compute stripe indices, constrain to array size
            final int min = Math.max(1, j - threshold);
            final int max = j > Integer.MAX_VALUE - threshold ? n : Math.min(n, j + threshold);

            // the stripe may lead off of the table if s and t are of different sizes
            if (min > max) {
                return -1;
            }
```

(c) Initially missed branches



```java
        // iterates through t
        for (int j = 1; j <= m; j++) {
            final char jOfT = t.charAt(j - 1); // jth character of t
            d[0] = j;

            // compute stripe indices, constrain to array size
            final int min = Math.max(1, j - threshold);
            final int max = j > Integer.MAX_VALUE - threshold ? n : Math.min(n, j + threshold);

            // the stripe may lead off of the table if s and t are of different sizes
            if (min > max) {
                return -1;
            }

            // ignore entry left of leftmost
            if (min > 1) {
                d[min - 1] = Integer.MAX_VALUE;
            }

            // iterates through [min, max] in s
            for (int i = min; i <= max; i++) {
                if (s.charAt(i - 1) == jOfT) {
                    // diagonally left and up
                    d[i] = p[i - 1];
                } else {
                    // 1 + minimum of cell to the left, to the top, diagonally left and up
                    d[i] = 1 + Math.min(Math.min(d[i - 1], p[i]), p[i - 1]);
                }
            }

            // copy current distance counts to 'previous row' distance counts
            tmp = p;
            p = d;
            d = tmp;
        }

        // if p[n] is greater than the threshold, there's no guarantee on it being the correct
        // distance
        if (p[n] <= threshold) {
            return p[n];
        }
        return -1;
    }
```

(d) Final missed branches

Figure 4: Comparison of branch coverage before and after adding new test cases for StringUtils::getLevenshteinDistance. Top: Overall percentage improvement. Bottom: Detailed branch analysis.

8 Self-assessment: Way of working

Contribution

The workload for this assignment was distributed through modular ownership, ensuring that each member took full responsibility for a high-complexity function while also participating in a cross-verification process to guarantee metric accuracy.

- **Yusuf:** Completed complexity reduction and branch coverage improvements for `reflectionAppend`. Performed manual CCN verification for Amanda's function.
- **Amanda:** Conducted complexity and coverage enhancements for `convertRemainingAccentCharacters` Performed manual CCN verification for Edvin's function.
- **Edvin:** Executed complexity refactoring and coverage improvements for `lexx`. Performed manual CCN verification for Jafar's function.
- **Jafar:** Responsible for complexity reduction and coverage optimization of `getLevenshteinDistance`. Performed manual CCN verification for Dawa's function.
- **Dawa:** Carried out complexity and coverage improvements for `appendFullDigits`. Performed manual CCN verification for Yusuf's function.

Essence Self-Evaluation

We evaluate our current way of working as being in the **In Use** state. Building on the solid foundation from our previous projects, we effectively adapted our established "Working Well" practices to meet the specific technical demands of legacy code refactoring and coverage improvement. Our focus shifted toward ensuring that project-specific constraints (such as the 35% CCN reduction target and the handling of "dead code" in a mature library) were met with high precision. Our collaboration remained fluid through the continued use of PR-based reviews and maven tests, which served as our primary safety net to prevent regressions in the library's behavior while validating our new tests. While we encountered some initial friction during the manual counting of deeply nested switch statements, we resolved these through open discussion and by standardizing our counting rules to align strictly with the `lizard` tool. This level of synchronization and mutual trust allowed us to work on completing the assignment mission without the need for excessive process overhead.

9 Overall experience

This assignment bridged the gap between theoretical software metrics and the reality of maintaining a mature open-source project. While we initially expected that "High Complexity" would directly correlate with "Low Coverage," our experience with *Apache Commons Lang* proved otherwise.

Key takeaways from our work include:

- **The "Mature Project" Paradox:** We discovered that in widely-used libraries, highly complex functions are often the *most* heavily tested parts of the system. Finding functions with missing coverage was significantly harder than expected. This taught us that complexity metrics alone are insufficient to identify weak spots in legacy codebases; they must be cross-referenced with coverage data.

  – **Defensive Coding vs. Coverage:** We learned that ”100% Coverage” is not always a practical or mathematically possible goal. As observed in our analysis, mature code often contains defensive checks (failsafes) that are logically impossible to trigger given the strict antecedent validations. Distinguishing between ”missing tests” and ”unreachable dead code” required deep manual inspection, which automated tools could not provide.
  – **Testability as a Refactoring Driver:** Attempting to write tests for complex, monolithic methods revealed significant friction. We realized that if a method requires excessive setup (e.g., mocking deep hierarchies or complex state) to test a single branch, it likely violates the Single Responsibility Principle. This practical struggle demonstrated *why* refactoring is essential for long-term maintainability, even more than the metrics themselves suggest.

Ultimately, this project demonstrated that software metrics are valuable starting points for analysis, but they must always be interpreted through the lens of architectural understanding and manual code review.