COMP2006 Assignment report

Mohammadsam Hassanpour

Student ID: 22321819

03/09/2025

Overview

In this assignment I implemented a java-based program to search and analyze files in a directory tree. This app recursively loads a directory (“.” default input, or given through command line arguments), then allows users to set search criteria (text or regex, include/exclude), choose output formats (count or show), and generate reports on the terminal.

Design patterns

Composite pattern

The composite pattern is used to represent the directory tree struct, which allows for uniform treatment of files and directories during the report generation. The FsNode interface defines the common operations such as getName(), isDirectory(), getContents(), getChildren(), addchild(). DirectoryNode acts as the composite and holds a list of child FsNode objects (files or subdir) and implements a recursive operation like addChild and getChildren. FileNode is the leaf and stores file contents as a list of strings. Default methods in FsNode (eg empty getChildren for files, throwing addChildren for files) ensures safe polymorphism. This system allows for recursive traversal in report creation (CountOutput.printCount and ShowOutput.printShow) which makes hierarchical outputs with indentation, like directory totals followed by child details.

Strategy pattern

The strategy pattern lets you handle different report output formats (count and show); it allows for flexible switching between the two without modifying core logic. The OutputStart interface defines a single method, generate (FsNode, CriteriaSet), implemented by CountOutput and ShowOutput. CountOutput counts matching lines per file, and piles totals for directories, printing hierarchically. ShowOutput displays matching lines with 1-based line numbers (eg 1 public class). The App class dynamically sets the strategy via user input in setOutputFormat, though count is the default format.

Design choices

1) Directory loading (DirectLoader.java)

- Recursively loads all files and directories using java.nio.file. Texts files are read (FileNode.getContents) while directories store children (DirectoryNode.getChildren).

- Hidden files and hidden directories (starting with ".") are skipped to avoid reading system files and ignored files. This is done by returning NULL, logged as INFO to exclude such files. This reduces unnecessary analysis and helps to keep a clean report.

2) Criteria handling (CriteriaSet, Criterion, LineMatcher.java)

- Criteria are parsed from user input (set by the user (eg "+t hello") into CriteriaSet containing Criterion objects, each combining an include/exclude flag with a LineMatcher (polychromic via PlainTextMatch for substring matching or RegexMatcher for regex with Pattern.find()). A default criterion of "+ r.*" (include all lines via regex) is set in the CriteriaSet construct and as a fallback in setCriteria if no valid criteria is provided by the user.

- Exclusive criteria are checked first. Overriding inclusions. Matching is case sensitive for both text and regex, ensuring consistency (hello != Hello).

- Invalid inputs (bad regex, wrong type) are logged and skipped, with a fallback to the default criterion.

3) Error handling and logging

- I used java.util.logging.Logger for tracing this system (INFO), warnings (eg unreadable file and invalid criteria) and severe errors such as invalid dir. This helps with debugging and monitoring.

- I/O errors such as file read failures result in empty contents or skipped nodes, ensuring the app continues running. Invalid criteria inputs are logged and the user is notified on the terminal.

4) output formatting

- Outputs are indented hierarchically with directories sorted alphabetically via Comparator in DirectoryNode.getChildren or ShowOutput.

- CountOutput precomputes the total number of directories to print before children, this ensures correct hierarchy (though it traverses twice, functional but less efficient)

- ShowOutput shows line numbers and content for matching lines, which preserves the file structure.

Use of container

I used java.util.ArrayList as the primary container for dynamic collection. It ensures flexibility and performance within the system.

- DirectoryNode: Stores child FsNode objects (files or subdir) in an ArrayList<FsNode>. This allows for dynamic addition of children during recursive loading (DirectLoader.load) and sorting for alphabetical output.

- FileNode: Holds file contents as an ArrayList<String> allowing variable sized line storage and efficient iteration during matching.

- CriteriaSet: Maintains a list of Criterion objects in an ArrayList<Criterion>, allowing for dynamic additions and removal of user defined criteria (setCriteria).

- App: Uses ArrayList<String> to collect user input lines for criteria (setCriteria), once again allowing for variable input sizes.

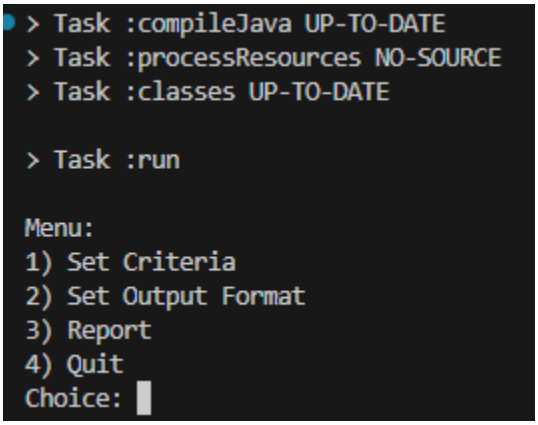
ArrayList was chosen for its o(1) append performance, dynamic resizing and support for iteration which suits the applications needs such ass adding children, reading file lines and parsing criteria.

Testing and example output

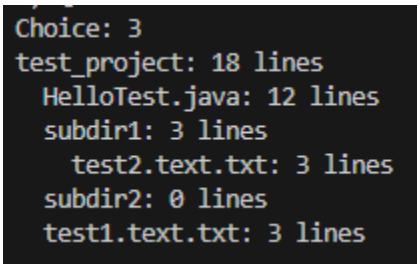
To run the app, open the terminal and set working directory to the root directory (eg C:\Users\username\Desktop\base_java_project> ...). Then type in the following:

./gradlew run --args="target\dir" and hit enter.

Example output:



Selecting 3) Report will output a report with the default criteria, with the default reporting format which is counting.



Here is a showcase of + t Hello:

```
test_project: 5 lines
  HelloTest.java: 3 lines
  subdir1: 1 lines
    test2.text.txt: 1 lines
  subdir2: 0 lines
  test1.text.txt: 1 lines
```

Here are the same criteria but with ShowOutput:

```
Choice: 3
test_project:
  HelloTest.java:
    5 public class HelloTest
    8     void testHello()
    10          assertEquals("Hello world!", Hello.getHello());
  subdir1:
    test2.text.txt:
      2 Hello again
  subdir2:
  test1.text.txt:
    1 Hello world
```

Here is a show case of + r [a-z]+ AND – r [0-9]+

```
app: 3 lines
  file2.txt: 1 lines
  file3.txt: 1 lines
  subdir: 1 lines
    file1.txt: 1 lines
```

Same criteria, this time with ShowOutput:

```
app:
  file2.txt:
    2 hello
  file3.txt:
    3 pattern
  subdir:
    file1.txt:
      3 xyz
```

Note: Output format and criteria need to be reset/changed manually after each use. This could be a weak point of the app, which I can improve on in the future. All the test files are in the base_java_project. Text focused files are in the root directory, in the test_project folder while regex focused files are in base_java_project\src\test\java\edu\curtin\app.

Conclusion

All in all, this system addresses the core requirements of recursive directory traversal, flexible criteria matching and reporting (either count or show). The composite pattern ensures seamless tree handling, while the strategy pattern supports extensible output formats. I think the choices I have made such as case sensitive matching and ignoring hidden or ignored files align with practical software project analysis. I made sure that this system follows modularity principles by making each class with purpose and they all have one focused purpose.