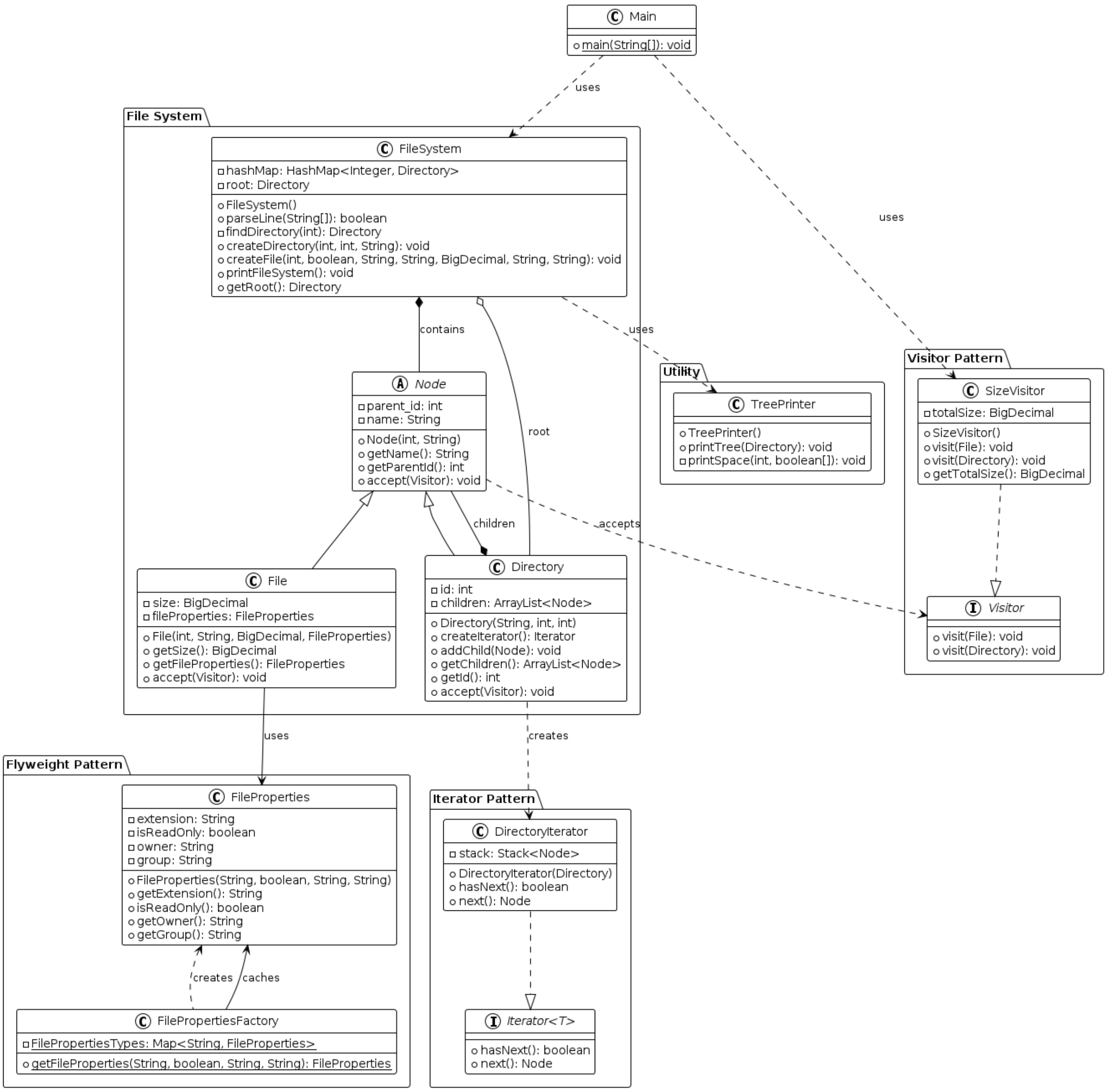
Architecture Description: Directory Walker

Rushan Shafeev

May 2025

1 UML Diagram



2 Architecture Description

The Directory Walker is a Java-based file system simulator designed to model a hierarchical file structure, process input commands, compute total file sizes, and render a Linux-style tree visualization. It processes input lines (e.g., `DIR id parent_id name` or `FILE id readOnly owner group size name.extension`) to build a tree of directories and files. The system leverages three design patterns—Flyweight, Iterator, and Visitor—to ensure memory efficiency, modular traversal, and extensible operations. The core class hierarchy includes:

- `Node`: An abstract base class defining common attributes (`parent_id`, `name`) and an `accept(Visitor)` method for the Visitor pattern.
- `Directory`: Extends `Node`, managing an `ArrayList<Node>` for children, a unique `id`, and an iterator for traversal.
- `File`: Extends `Node`, storing a `BigDecimal` size and a `FileProperties` object for metadata.
- `FileSystem`: Orchestrates the hierarchy, using a `HashMap<Integer, Directory>` for O(1) directory lookups by ID, parsing input, and coordinating size calculations and tree printing.

Auxiliary classes include `FileProperties` for metadata, `FilePropertiesFactory` for Flyweight caching, `DirectoryIterator` for tree traversal, `SizeVisitor` for size aggregation, and `TreePrinter` for ASCII tree rendering. The architecture prioritizes scalability, handling large file systems (e.g., millions of nodes) while maintaining low memory overhead through shared metadata and efficient data structures.

2.1 Flyweight Pattern

The Flyweight pattern optimizes memory by sharing immutable `FileProperties` objects, which encapsulate file metadata: `extension` (e.g., `txt`), `isReadOnly` (boolean), `owner` (e.g., `user`), and `group` (e.g., `group`). The `FilePropertiesFactory` maintains a `HashMap<String, FileProperties>` to cache instances, using a composite key (`extension + "-" + isReadOnly + "-" + owner + "-" + group`) to identify unique combinations. When creating a file, `FileSystem` calls `FilePropertiesFactory.getFileProperties`, which returns an existing instance or creates a new one if none exists. This ensures that files with identical metadata, such as multiple `txt` files owned by `user:group`, share a single `FileProperties` object, significantly reducing memory redundancy. For example, in a system with 1,000,000 files sharing the same metadata, only one `FileProperties` instance is stored, compared to 1,000,000 instances without Flyweight. The pattern is seamlessly integrated into `FileSystem.createFile`, enhancing scalability for large-scale file systems and minimizing heap usage, as demonstrated in the memory comparison experiment.

2.2 Iterator Pattern

The Iterator pattern facilitates depth-first traversal of the file system hierarchy, decoupling traversal logic from the `Directory` structure. The `Iterator` interface defines two methods: `hasNext()` to check for remaining nodes and `next()` to retrieve the next node.

The DirectoryIterator class implements this interface, using a Stack<Node> to perform a depth-first search. It initializes with a Directory and pushes its children onto the stack in reverse order to maintain correct traversal order. The Directory class provides a createIterator() method, enabling clients like TreePrinter to traverse the tree without accessing the internal ArrayList<Node> directly. TreePrinter uses the iterator to generate a Linux-style tree with ASCII characters, tracking node depths and sibling counts via additional stacks. This design enhances modularity, allowing alternative traversal strategies (e.g., breadth-first) or new operations (e.g., searching) to be implemented without modifying Directory. The iterator's stack-based approach ensures efficient memory usage, even for deep or wide file systems.

2.3 Visitor Pattern

The Visitor pattern separates operations like size calculation from the file system's node hierarchy, promoting extensibility and maintainability. The Visitor interface declares visit(Directory) and visit(File) methods, allowing different behaviors for each node type. The SizeVisitor class implements this interface, accumulating file sizes (stored as BigDecimal for precision) in a totalSize field while ignoring directories, which contribute no direct size. Each Node subclass implements accept(Visitor), with Directory recursively forwarding the visitor to its children via its ArrayList<Node>. This enables SizeVisitor to traverse the entire tree, summing file sizes without modifying node classes. The FileSystem class uses SizeVisitor to compute the total size, which is formatted using BigDecimal.stripTrailingZeros() for clean output (e.g., 100KB instead of 100.000KB). The Visitor pattern allows new operations, such as counting files or collecting extensions, to be added by creating new visitor classes, without altering File or Directory. This decoupling enhances the system's flexibility for future enhancements, such as reporting or analysis tasks.

3 Flyweight Memory Usage

This report evaluates the memory efficiency of the Flyweight design pattern in a file system application implemented in the Directory_Walker package. The Flyweight pattern is used to share FileProperties instances, which store file metadata (extension, read-only status, owner, and group). The experiment compares memory usage when creating 1,000,000 files with and without the Flyweight pattern.

3.1 Methodology

The experiment used a Java application to simulate a file system with 1,000,000 files, each with identical properties (extension="txt", isReadOnly=true, owner="user", group="group"). Memory usage was measured using the Runtime class, calculating the difference in used memory (totalMemory - freeMemory) before and after creating the files, with garbage collection enforced via Runtime.getRuntime().gc().

Two implementations were tested by modifying the createFile method in the FileSystem class:

- **With Flyweight**: Used the FilePropertiesFactory to cache and reuse FileProperties instances.

- **Without Flyweight**: Created a new `FileProperties` instance for each file.

The toggled code in `FileSystem.createFile` is shown below:

```
// With Flyweight
FileProperties fileProperties = FilePropertiesFactory.
    getFileProperties(extension, isReadOnly, owner, group);

// Without Flyweight
FileProperties fileProperties = new FileProperties(extension,
    isReadOnly, owner, group);
```

The test was executed using the following `Main` class, which simulates the creation of 1,000,000 files and measures memory usage:

```
package Directory_Walker;

public class Main {
    public static void main(String[] args) {
        long memoryFlyweight = measureMemoryWithFlyweight();
        System.out.println(memoryFlyweight + " MB");
    }

    private static long measureMemoryFlyweight() {
        Runtime runtime = Runtime.getRuntime();
        // Force garbage collection
        runtime.gc();
        long memoryBefore = runtime.totalMemory() - runtime.
            freeMemory();

        // Initialize file system
        FileSystem fileSystem = new FileSystem();
        // Simulate input: create 1,000,000 files with similar
            properties
        simulateInput(fileSystem);

        // Calculate total size
        SizeVisitor sizeVisitor = new SizeVisitor();
        fileSystem.getRoot().accept(sizeVisitor);

        // Force garbage collection again
        runtime.gc();
        long memoryAfter = runtime.totalMemory() - runtime.
            freeMemory();

        return (memoryAfter - memoryBefore) / 1024 / 1024;
    }

    private static void simulateInput(FileSystem fileSystem) {
        fileSystem.createDirectory(1, 0, "testDir");

        for (int i = 0; i < 1000000; i++) {
            String[] line = {
```

```
                        "FILE", String.valueOf(1), "T", "user", "group", "
                            100", "file" + i + ".txt"
                    };
                    fileSystem.parseLine(line);
                }
            }
}
```

3.2 Results

Table 1 presents the memory usage for both implementations.

Table 1: Memory Usage Comparison for 1,000,000 Files

Implementation	Memory Usage (MB)
With Flyweight	134
Without Flyweight	210

3.3 Analysis

The Flyweight pattern reduced memory usage by approximately 36.2 %, from 210 MB to 134 MB. This reduction is due to reusing a single `FileProperties` instance for all 1,000,000 files in the Flyweight implementation, compared to creating 1,000,000 separate instances in the non-Flyweight case. Each `FileProperties` instance includes three `String` fields and a `boolean`, and the object overhead (approximately 16–24 bytes per instance on a 64-bit JVM) accumulates significantly without Flyweight. The `HashMap` used by `FilePropertiesFactory` added negligible overhead, as only one unique `FileProperties` instance was cached.

3.4 Conclusion

The Flyweight pattern significantly reduces memory usage in applications with many objects sharing identical metadata. In this file system application, it achieved a 36.2 % memory reduction for 1,000,000 files, confirming its effectiveness for optimizing memory in large-scale systems.