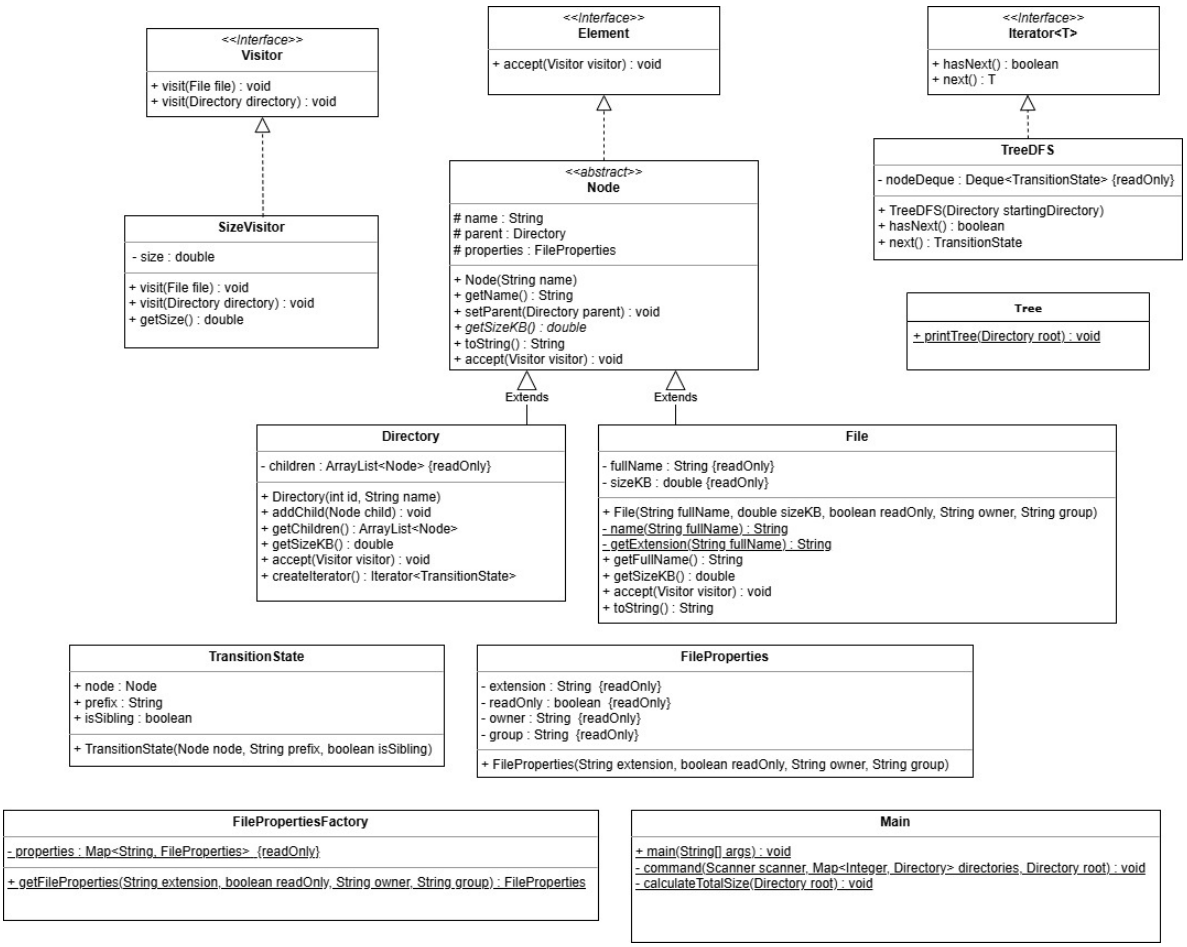

Figure 1: UML diagram of my code

In my project I used 3 patterns: 1 structural - Flyweight and 2 behavioral - Visitor, Iterator.

Structural pattern

1. In a file system simulation, numerous files may have identical attributes such as file extension, read-only status, owner, and group. Storing these attributes individually for each file would lead to redundancy and excessive memory use, particularly when dealing with thousands or millions of files. By applying the Flyweight pattern, these common attributes can be encapsulated within a single FileProperties object, which multiple File objects can reference.

1.1. The FileProperties class serves as the Flyweight in this pattern. It contains the intrinsic state — extension, readOnly, owner, and group — which represents the data that can be shared across multiple files.

```
/** Class representing file properties */
class FileProperties {  4 usages
    private final String extension;  1 usage
    private final boolean readOnly;  1 usage
    private final String owner;  1 usage
    private final String group;  1 usage

    /** Constructor for FileProperties ...*/
    public FileProperties(String extension, boolean readOnly, String owner, String group)
        this.extension = extension;
        this.readOnly = readOnly;
        this.owner = owner;
        this.group = group;
    }
}
```

1.2. The FilePropertiesFactory is responsible for creating and managing the cache of FileProperties

objects. It utilizes a HashMap to store existing instances and returns a shared one if the requested properties already exist.

```java
/** Factory class for creating and caching FileProperties instances */
class FilePropertiesFactory {  1 usage
    private static final Map<String, FileProperties> properties = new HashMap<>();  3 usages
    /** Gets a FileProperties instance, creating and caching it if necessary ...*/
    public static FileProperties getFileProperties(String extension, boolean readOnly, String owner, String group) {
        String key = extension + readOnly + owner + group;
        if (!properties.containsKey(key)) {
            properties.put(key, new FileProperties(extension, readOnly, owner, group));
        }
        return properties.get(key);
    }
}
```

1.3. The File class contains the extrinsic state —data specific to each individual file, such as its name and size— and maintains a reference to a shared FileProperties object provided by the factory.

```java
/** Class representing a file in the file system */
class File extends Node {  4 usages
    private final double sizeKB;  3 usages
    private final String fullName;  2 usages

    /** Constructor for File ...*/
    public File(String fullName, double sizeKB, boolean readOnly, String owner, String group) {  1 usage
        super(name(fullName));
        this.fullName = fullName;
        this.sizeKB = sizeKB;
        String extension = getExtension(fullName);
        this.properties = FilePropertiesFactory.getFileProperties(extension, readOnly, owner, group);
    }
```

Behavioral patterns

2. The Visitor pattern introduces new operations to a class hierarchy (such as file system nodes: File and Directory) without altering their structure. To compute the total size of files in the file system tree, I traverse the structure and apply different logic depending on the node type.

2.1. The Visitor interface declares a visit method for each type of Element it can operate on.

```java
/** Visitor interface for visiting files and directories */
interface Visitor {  4 usages  1 implementation
    void visit(File file);  1 usage  1 implementation
    void visit(Directory directory);  1 usage  1 implementation
}
```

2.2. The Element interface defines an accept method that receives a Visitor as its parameter. This method is implemented by concrete element classes such as File and Directory, and its purpose is to delegate the call back to the appropriate visit method on the Visitor.

```java
/** Visitable interface for accepting visitors */
interface Element {  1 usage  3 implementations
    void accept(Visitor visitor);  2 usages  3 implementations
}

/** Abstract class representing a node in the file system */
abstract class Node implements Element {  14 usages  2 inheritors
    protected String name;
    protected Directory parent;  2 usages
    protected FileProperties properties;  1 usage

    @Override  2 usages  2 overrides
    public void accept(Visitor visitor) {}
```

2.3. The File and Directory classes implement the accept method.

```java
    @Override  2 usages
    public void accept(Visitor visitor) {
        visitor.visit( directory: this);
        for (Node child : children) {
            child.accept(visitor);
        }
    }
```

```java
    @Override  2 usages
    public void accept(Visitor visitor) {
        visitor.visit( file: this);
    }
```

2.4. The SizeVisitor class implements the Visitor interface. Its visit(File) method performs the calculation.

```java
/** Visitor implementation for calculating the total size of files */
class SizeVisitor implements Visitor {  2 usages
    private double size = 0;  2 usages

    @Override  1 usage
    public void visit(File file) {
        size += file.getSizeKB();
    }

    @Override  1 usage
    public void visit(Directory directory) {}
```

3. The Iterator pattern offers a standardized way to traverse the elements of an aggregate object (a tree or collection) sequentially, without revealing the internal structure of that object. This separation of traversal logic from the data structure enhances flexibility.

3.1. The interface Iterator (then implemented by treeDFS).

```java
Iterator interface for traversing nodes

Type parameters: <T>

interface Iterator<T> {  3 usages  1 imp
    boolean hasNext();  1 usage  1 imple
    T next();  1 usage  1 implementation
}
```

3.2. The Directory class is the aggregate object. It provides a method (createIterator) that returns an instance of the iterator.

```java
/** Class representing a directory in the file system */
class Directory extends Node {  19 usages
    private final ArrayList<Node> children = new ArrayList<>();

    /** @return iterator for traversing the directory tree */
    public Iterator<TransitionState> createIterator() {  1 usage
        return new TreeDFS( startingDirectory: this);
    }
```

3.3. The TreeDFS class implements Iterator$< TransitionState >$ to perform depth-first traversal of a tree structure. The class provides the standard hasNext() and next() methods defined by the Iterator interface, allowing clients to iterate over tree nodes without needing to understand the underlying traversal mechanics.

```java
/** Depth-First iterator for traversing the directory tree */
class TreeDFS implements Iterator<TransitionState> {  1 usage
    private final Deque<TransitionState> nodeDeque = new LinkedList<>();  4 usages

    /** Constructor for TreeDFS ...*/
    public TreeDFS(Directory startingDirectory) {  1 usage
        if (startingDirectory != null) {
            ArrayList<Node> initialChildren = startingDirectory.getChildren();
            for (int i = initialChildren.size() - 1; i >= 0; i--) {
                Node childNode = initialChildren.get(i);
                boolean isLastChild = (i == initialChildren.size() - 1);
                nodeDeque.push(new TransitionState(childNode,  prefix: "", isLastChild));
            }
        }
    }

    @Override  1 usage
    public boolean hasNext() { return !nodeDeque.isEmpty(); }

    @Override  1 usage
    public TransitionState next() {...}
}
```

In my project I used pattern Flyweight for saving memory. For testing I used my own test with 1.000.000 lines. I estimated the memory usage by built-in runtime methods. (You can see it here: link).

The memory usage before implementing Flyweight pattern.

```
--- Memory state ---
Total memory:   1452 MB (1522532352 bytes)
Free memory:        550 MB (577326712 bytes)
Used memory:    901 MB (945205640 bytes)
------------------------
```

The memory usage after implementing Flyweight pattern.

```
--- Memory state ---
Total memory:   1272 MB (1333788672 bytes)
Free memory:        750 MB (786924240 bytes)
Used memory:    521 MB (546864432 bytes)
------------------------
```