Implementation of UML Diagram

Assigment 4. Evdokimova Daria, CSE-01

To improve quality of code for this task I use 3 suitable design patterns from lectures.

1. The Flyweight pattern

This pattern is designed to reduce memory usage when a program has many instances with shared attributes. The main principle involves creating a factory class that stores common templates for instances of the `File` class. If no such template is found when creating an instance, the factory will generate a new one. Thus, with a large number of objects, memory efficiency is achieved, as each instance will only store its unique fields (such as *name* and *id*) while shared fields (*owner*, *group*, *extension*, *read_only*) are managed by the Flyweight.

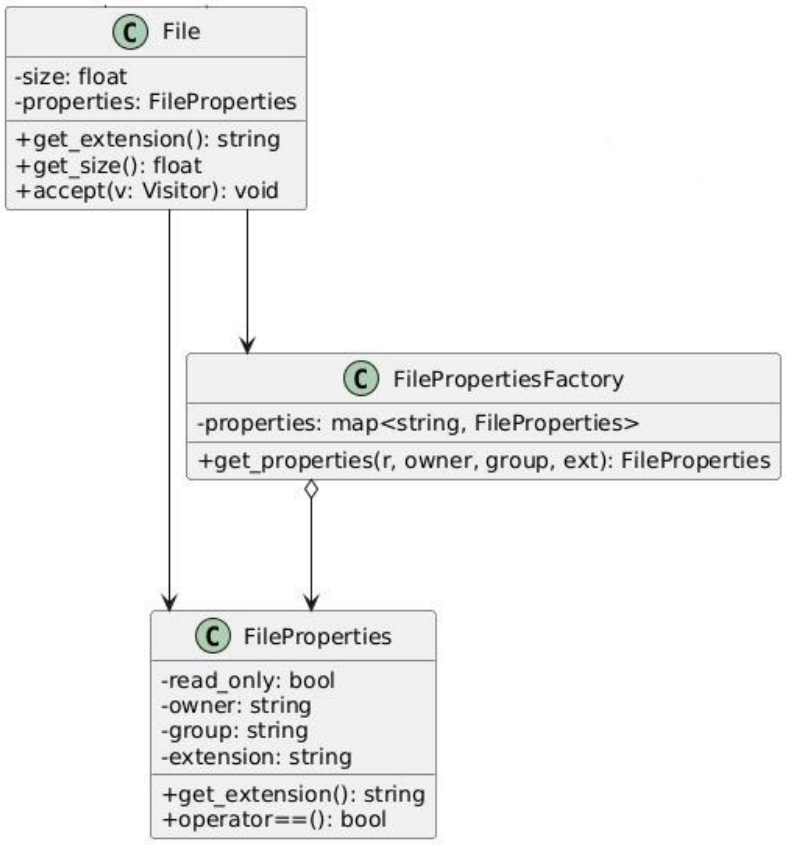
```
//Flyweight for storing sets of file attributes
class FileProperties {
    bool read_only;
    string owner;
    string group;
    string extension;

public:
    FileProperties(bool r, const string& o, const string& g, const string& ext) : read_only(r), owner(o), group(g), extension(ext) {}

    string get_extension() const { return extension; }

    //comparator for use in unordered_map
    bool operator==(const FileProperties& other) const {
        return read_only == other.read_only && extension == other.extension && owner == other.owner && group == other.group;
    }
};
```

```
//factory for managing and creating the above flyweights
class FilePropertiesFactory {
    static unordered_map<string, shared_ptr<FileProperties>> properties;
public:
    static shared_ptr<FileProperties> get_properties(bool r, const string& owner, const string& group, const string& extension) {

        //store by key equal to the combined attribute string
        string key = to_string( val: r) + owner + group + extension;
        //if such a set of attributes already exists, return it...
        if (properties.find( x: key) != properties.end()) {
            return properties[key];
        }
        //...if not, create and add a new one
        shared_ptr<FileProperties> new_format = make_shared<FileProperties>(r,  o: owner,  g: group,  ext: extension);
        properties[key] = new_format;
        return new_format;
    }
};
```

```cpp
//shared dictionary of all attribute sets
unordered_map<string, shared_ptr<FileProperties>> FilePropertiesFactory::properties;

//file class
class File : public Node {
    float size;
    //attributes stored in a flyweight
    shared_ptr<FileProperties> properties;
public:
    File(int id, bool r, const string &owner, const string &group, float size, const string &name, const string &extension) : Node(id, name), size(size) {
        properties = FilePropertiesFactory::get_properties(r, owner, group, extension);
    }
    //function to get extension, returns it from the flyweight
    string get_extension() {
        return properties->get_extension();
    }
```

...

UML for pattern



I taste code with FlightWeight and without it and get these results:

with

```
Memory used: 4156 KB
1000
FILE 0 F root docker 827.34 gwaqj.cpp
```

...

```
├── gwaqj.cpp (827.34KB)
└── gwaqj.cpp (827.34KB)
Memory used: 4600 KB
```

4600 – 4156 = 444 KB

without

```
Memory used: 4112 KB
1000
FILE 0 F root docker 827.34 gwaqj.cpp
...
```

```
├── gwaqj.cpp (827.34KB)
├── gwaqj.cpp (827.34KB)
└── gwaqj.cpp (827.34KB)
Memory used: 4652 KB
```

4652 – 4112 = 540 KB

For testing, I used a file containing 1000 similar files, all depending on a single root. As a result, applying the Flyweight pattern reduced memory usage by approximately 17.8% for this test.

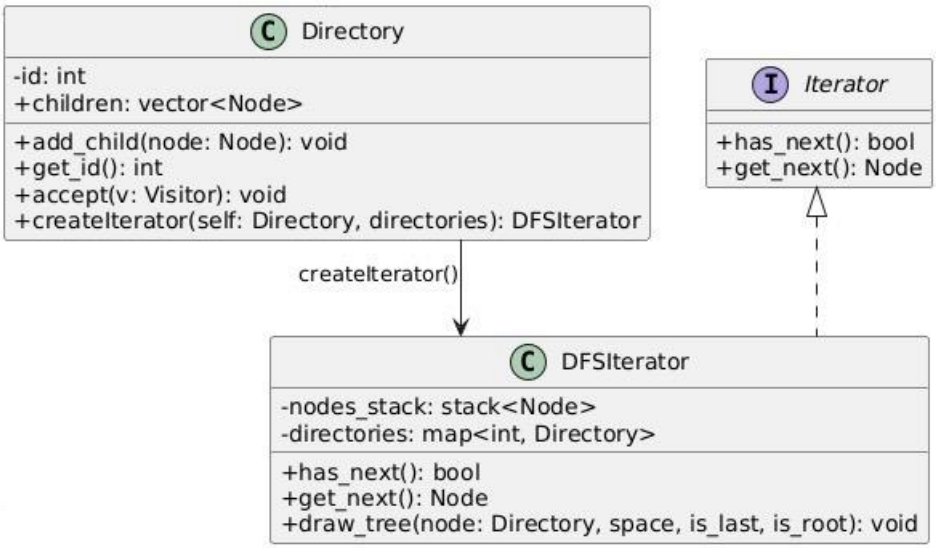
2. The Iterator pattern

Since the file structure is represented as a tree, we need a specialized way to traverse it. That's why the Iterator pattern is well-suited for this task. An *Iterator* interface is created with basic methods that are then implemented in the *DFSIterator* class. This way, using depth-first traversal, we can build the tree (e.g., in the *draw_tree()* method).

```
//forward declaration of DFSIterator for use in directory functions
class DFSIterator;

//directory class
class Directory : public Node {
    int id;
public:
    vector<shared_ptr<Node>> children;

    //two constructors since directories can be created with two or three variables (directly inherited from root)
    Directory(int id, const string& name) : Node( id: 0, name), id(id) {}
    Directory(int id, int parent_id, const string& name) : Node( id: parent_id, name), id(id) {}

    //forward declaration of function for creating an iterator
    unique_ptr <DFSIterator> createIterator(shared_ptr<Directory> self, const unordered_map<int, shared_ptr<Directory>>& directories);

    //getter for id
    int get_id() const { return id; }

    //function to add children to directory
    void add_child(shared_ptr<Node> node) { children.emplace_back(node); }

    //override accept function to use visitor
    void accept(Visitor& v) override {
        v.visit( &: *this);
        for (auto& child : shared_ptr<Node> &  : children)
            child->accept( &: v);
    }
};

//interface for creating DFSIterator
class Iterator {
public:
    virtual bool has_next() = 0;
    virtual shared_ptr<Node> get_next() = 0;
    virtual ~Iterator() = default;
};
```

```cpp
//class for working with the file tree (particularly for DFS output)
class DFSIterator : public Iterator {
    //stack to be used in DFS
    stack<shared_ptr<Node>> nodes_stack;
    //unordered_map to access all directories by id
    const unordered_map<int, shared_ptr<Directory>>& directories;

public:
    DFSIterator(shared_ptr<Directory> root, const unordered_map<int, shared_ptr<Directory>>& directories) : directories(directories) {
        nodes_stack.push( x: root);
    }

    //function to check for next node
    bool has_next() override { return !nodes_stack.empty(); }

    //function to get next node and traverse tree with DFS
    shared_ptr<Node> get_next() override {
        //if no next element, exit...
        if (!has_next()) return nullptr;

        //...if exists, pop it
        auto current : shared_ptr<Node>  = nodes_stack.top();
        nodes_stack.pop();

        //if node is a directory, push all its children into the stack
        if (auto directory : shared_ptr<Directory>  = dynamic_pointer_cast<Directory>( r: current)) {
            for (auto x : reverse_iterator<...>  = directory->children.rbegin(); x != directory->children.rend(); ++x)
                nodes_stack.push( x: *x);
        }
        //return the current element
        return current;
    }
    //function to print the file tree
    void draw_tree(const shared_ptr<Directory>& node, const string& space = "", bool is_last = true, bool is_root = true) {

        //if the node is not root, draw edges depending on whether it's the last
        if (!is_root) {
            cout << space << (is_last ? "└── " : "├── ");
        }
        //print the node name
        cout << node->name << endl;

        //for all children of the node
        for (int i = 0; i < node->children.size(); ++i) {
            //get the child
            const auto& child : const shared_ptr<Node> &  = node->children[i];
            //check if it's the last child
            bool iss_last = (i == node->children.size() - 1);
            //depending on whether it's the last and whether it's the root, print either space or bar
            string new_space = space + (is_root ? "" : (is_last ? "    " : "│   "));

            //if the node is a directory...
            if (auto dir : shared_ptr<Directory>  = dynamic_pointer_cast<Directory>( r: child)) {
                //...recursively call draw_tree
                draw_tree( node: dir,  space: new_space,  is_last: iss_last,  is_root: false);
            }
                //if the node is a file...
            else if (auto file : shared_ptr<File>  = dynamic_pointer_cast<File>( r: child)) {
                //print information in the desired format including edge rendering
                cout << new_space << (iss_last ? "└── " : "├── ") << file->name << "." << file->get_extension() << " (" << file->get_size() << "KB)" << endl;
            }
        }
    }
};
```

UML for pattern



3. The Visitor Pattern

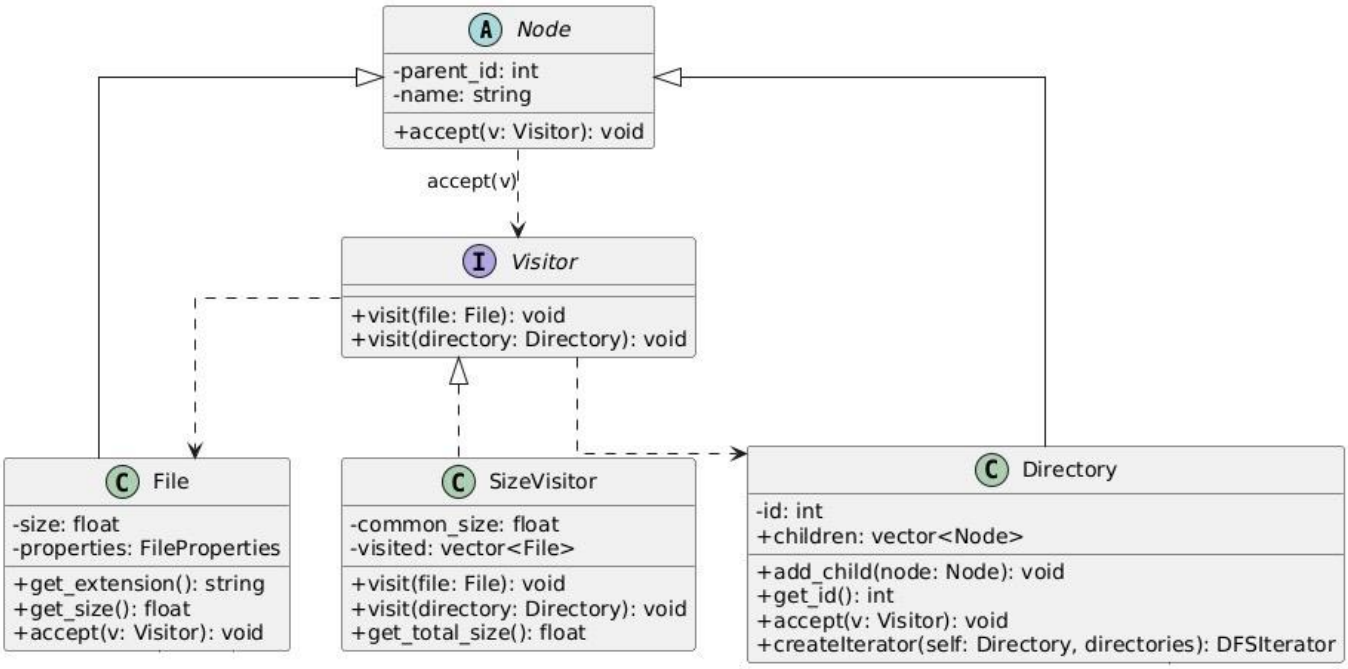
To calculate the total size of a tree (the combined size of all files in the tree), the *Visitor* pattern is used. The main idea is that the *File* and Directory classes have different structures, so the *visit()* method is implemented separately for each class accordingly. An interface is created with *visit()* methods for each class. Then, in the *SizeVisitor* class, these methods are implemented for *File* and *Directory*. These classes contain an *accept()* method, which invokes the corresponding *visit()* method.

Thus, calculating the total size of the tree does not modify the structure of the *File* and *Directory* classes. Instead, they delegate the task to the *SizeVisitor* class, which already contains the logic for data aggregation and stores the *common_size*.

```
//interface for future creation of SizeVisitor class
class Visitor {
public:
    //separate visit methods for file and directory classes
    virtual void visit(File &file) = 0;
    virtual void visit(Directory &directory) = 0;

    virtual ~Visitor() = default;
};
```

```cpp
//class for calculating total size of all files
class SizeVisitor : public Visitor {
    float common_size = 0;
    vector<File> visited;
public:
    SizeVisitor() = default;
    //override method for files...
    void visit(File &file) override {
        if (find( first: visited.begin(),  last: visited.end(),  val: file) == visited.end()) {
            //just add its size to the total
            common_size += file.get_size();
            visited.emplace_back(file);
        }
    }

    //...and separately for directories
    void visit(Directory &directory) override {
        for (auto& child : shared_ptr<Node> &  : directory.children)
            //call the method for all its children
            child->accept( &: *this);
    }
    //function to get final total size
    float get_total_size() const {
        return common_size;
    }
};
```

UML for pattern



UML for whole program