Assignment 2: Composite Pattern

Design Patterns Implementation

File System Explorer Application

Name: Colby Frison

Student ID: XXXXXXXXX

Course: CS 4213 - Software Design Patterns

Semester: Fall 2025

Assignment: DP Assignment 2 - Composite

Due Date: End of Week 12

January 20, 2026

Contents

1 Hierarchical System Context

1.1 Domain: File System Explorer

The domain selected for this assignment is a **File System Explorer**, which demonstrates hierarchical structures naturally. File systems are inherently hierarchical, with folders containing files and other folders, which can be nested to any depth.

1.2 Feature and Module Description

The file system explorer handles navigation and management of nested folder structures:

- **Multiple layers interact:** Each folder can contain individual files, subfolders, or a mix of both
- **Recursive operations:** The system needs to traverse through however many levels of nesting exist
- **Uniform interface:** Both files and folders respond to the same set of operations, even though they work differently internally

1.3 Uniform Operations Needed Across Levels

These operations work the same way whether called on a single file or an entire folder tree:

1. `display(indent)`: Shows the file or folder with indentation to indicate its level
2. `getSize()`: Returns size in bytes (a file's size, or the total for everything in a folder)
3. `search(name)`: Finds a file or folder by name
4. `getPath()`: Gets the full path from root to this item

1.4 Hierarchy Block Diagram

Figure 1 shows the hierarchical structure, illustrating how the root folder contains multiple levels of nested folders and files.

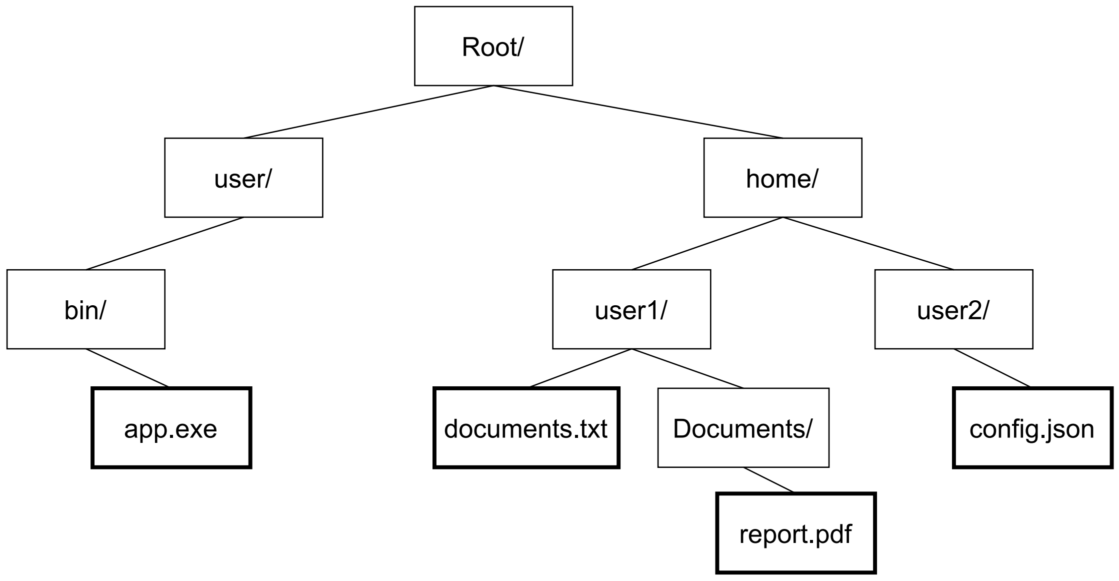

Figure 1: File System Hierarchical Block Diagram showing recursive directory and file relationships

The diagram shows:

- **Root Directory**: The top-level container
- **Intermediate Directories**: `user/`, `home/`, `bin/`, `user1/`, `user2/`, `Documents/`
- **Leaf Files**: `app.exe`, `documents.txt`, `config.json`, `report.pdf`

2 Design and Modeling

2.1 Applying the Composite Pattern

The design uses three roles from the Composite Pattern:

Component (FileSystemComponent): This is the abstract base that defines what both files and folders can do, including `display()`, `getSize()`, `search()`, and `getPath()`. It also includes folder-only operations like `add()` and `remove()`, which throw an error if called on a file.

Leaf (File): Represents individual files that don't contain anything else. Each file has a size and extension, and implements the operations in its own way.

Composite (Directory): Represents folders that can hold files and other folders. It keeps a list of children and handles operations recursively, such as `getSize()` adding up everything inside, or `search()` looking through all children until it finds a match.

2.2 Adding a Creation Pattern: Factory Method

The `FileSystemFactory` provides methods for creating files and folders:

- `createFile(name, size, extension)`: Creates a new file
- `createDirectory(name)`: Creates a new folder
- `createFromPath(path)`: Scans an actual file system path and builds the whole structure automatically

This keeps clients from having to know about the concrete File and Directory classes directly.

2.3 UML Class Diagram

Figure 2 shows how all the classes relate to each other, including how File and Directory both inherit from FileSystemComponent, and how the Factory creates them.

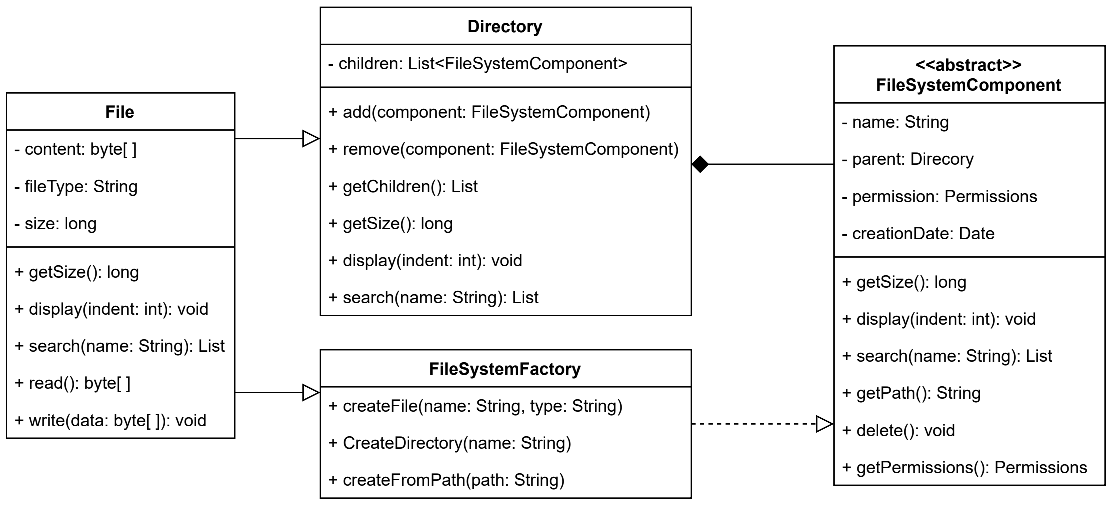
File
- content: byte[ ]
- fileType: String
- size: long
+ getSize(): long
+ display(indent: int): void
+ search(name: String): List
+ read(): byte[ ]
+ write(data: byte[ ]): void
Directory
- children: List<FileSystemComponent>
+ add(component: FileSystemComponent)
+ remove(component: FileSystemComponent)
+ getChildren(): List
+ getSize(): long
+ display(indent: int): void
+ search(name: String): List
FileSystemFactory
+ createFile(name: String, type: String)
+ CreateDirectory(name: String)
+ createFromPath(path: String)
<<abstract>>
FileSystemComponent
- name: String
- parent: Direcory
- permission: Permissions
- creationDate: Date
+ getSize(): long
+ display(indent: int): void
+ search(name: String): List
+ getPath(): String
+ delete(): void
+ getPermissions(): Permissions

Figure 2: UML Class Diagram of File System using Composite and Factory Patterns

2.4 Activity Diagram: File Search Workflow

Figure 3 shows how the search operation works: it checks the current item's name, and if it is a folder, recursively searches through all the children until it finds a match.

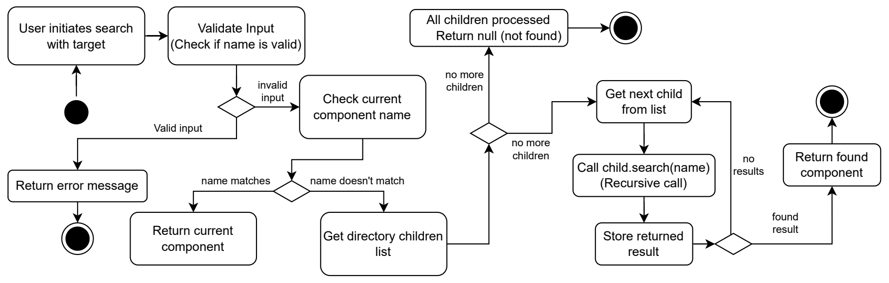

Figure 3: Activity Diagram: File Search Operation showing recursive traversal

3 Pattern Specification and Interpreter

3.1 Decision Table for Permission Checking

The decision table shows the rules for determining whether someone can access a file or folder based on their ownership and permissions.

Table 1: File Access Permission Decision Table

Rule	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**
Conditions						
Is Owner?	Y	Y	Y	N	N	N
Has Group Access?	-	-	-	Y	Y	N
Is Read Permission?	Y	Y	N	Y	N	-
Is Write Permission?	Y	N	-	N	Y	-
Is Directory?	-	-	Y	-	Y	-
Actions						
Grant Full Access	X	-	-	-	-	-
Grant Read Only	-	X	-	X	-	-
Grant Write (Dir Only)	-	-	X	-	X	-
Deny Access	-	-	-	-	-	X
Log Access Attempt	X	X	X	X	X	X

Rule Descriptions:

1. **R1**: Owner with read and write permissions gets full access
2. **R2**: Owner with only read permission gets read-only access
3. **R3**: Owner can create/delete in owned directories
4. **R4**: Group member with read permission gets read-only access
5. **R5**: Group member with write permission can modify directories
6. **R6**: Non-owner/non-group user is denied access

3.2 Encoding Logic with the Interpreter Pattern

These decision table rules can be encoded using the Interpreter Pattern. The approach involves building complex permission checks from simple, composable pieces:

- **Context**: Holds the user's permission information (whether the user is the owner, has read access, etc.)
- **Expression Interface**: Each expression can evaluate itself given the context
- **Basic Checks**: Simple expressions like IsOwner, HasReadPerm, HasWritePerm
- **Combinations**: Logical operators (AND, OR, NOT) that combine other expressions

Example rule expressions:

```
Rule1 = (IS_OWNER AND HAS_READ_PERM AND HAS_WRITE_PERM)
Rule2 = (IS_OWNER AND HAS_READ_PERM)
GrantAccess = Rule1 OR Rule2
```

This approach enables building new rules by combining existing pieces, and allows each piece to be tested independently.

4 Reflection

Why Composite Fits This Design

File systems are naturally hierarchical, because folders contain files and other folders, and they can be nested to any depth. The Composite Pattern fits this structure very well, since it enables treating both files and folders in the same way through a shared interface. When calling `getSize()` on a folder, it can walk through the tree and add up all the files inside without requiring special case code for recursion. The same `display()` or `search()` call works whether dealing with a single file or an entire folder tree. This uniform treatment means developers do not need to constantly write if statements that ask "is this a file or a folder?" in different parts of the system. The pattern also scales naturally, because adding more nesting levels or introducing new component types such as shortcuts does not require rewriting existing client code.

The Advantage of Factory Method

Adding the Factory Method gives the design a cleaner and more controlled way to create files and folders. Instead of client code directly calling `new File()` or `new Directory()`, it asks the factory for the components it needs. This is especially helpful for a method such as `createFromPath()`, which can scan the actual file system and automatically build the whole hierarchy, deciding whether each path is a file or a folder and creating the appropriate type. The factory also centralizes any initialization logic, such as setting default permissions or metadata, eliminating the need to duplicate the same setup code across multiple places. For testing, a factory can be swapped in that produces mock components instead of real ones, which makes it easier to test behavior without touching the real file system.

How Pattern-Based Logic Models Improve Maintainability

Decision tables make permission rules easy to see and easy to discuss. Instead of hunting through nested if statements scattered across different classes, all the rules are located in one place where both developers and non-technical stakeholders can read and review them. The Interpreter Pattern builds on this idea by breaking complex rules into small, reusable pieces, such as basic checks like `IsOwner` or `HasReadPermission`, which can then be combined using logical operators like AND, OR, and NOT. When permission requirements change, typically only one of these smaller pieces needs to be adjusted, rather than searching for every site where permissions are checked. Each expression can be tested on its own, and new rules can be assembled from existing expressions without modifying the old ones. Keeping the rule definitions separate from the rest of the business logic helps keep the system understandable and makes it easier to evolve over time.

References

1. Kung, David C. (2013). *Software Engineering: An Object-Oriented Perspective* (2nd ed.). Course textbook chapters 3.4, 3.5, 10.1–10.6.