Heaven’s Light Is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Course Code: CSE 3205

Course Title: Software Engineering Sessional

Lab Project: Composite Design Pattern

Submitted By	**Submitted To**
Syed Shifat E Rahman	Farjana Parvin
Roll: **2003065**	Assistant Professor
Section: B, Session: 2020-2021	Department of CSE, RUET

Date: 03.06.2025

Table of Contents

Design Pattern: Composite

Design patterns are reusable solutions to common problems in software design. The Composite Pattern is part of the structural design pattern category and treats individual objects and compositions uniformly. Composite Design Pattern enables the composition of objects into tree structures to represent part-whole hierarchies. It allows clients to treat both individual objects and compositions of objects uniformly.

Design pattern

In software engineering, a design pattern is a general, repeatable solution to a commonly occurring problem in software design. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns make the code more flexible or efficient, reducing coupling among program components and memory overhead. A design pattern isn't a finished design that can be directly used in code.

There are 23 design patterns in three categories:

- Creational
- Structural
- Behavioral

Creational Patterns

Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code. There are 5 Creatonal design Patterns:

1. Factory method
2. Abstract factory
3. Builder
4. Prototype
5. Singleton

Structural Patterns

It explains how to assemble objects and classes into larger structures while keeping these structures flexible and efficient. 7 Structural Design Patterns:

1. Adapter
2. Bridge
3. **Composite**
4. Decorator
5. Facede
6. Flyweight
7. Proxy

Behavioral Patterns

It is concerned with algorithms and the assignment of responsibilities between objects. All the remaining patterns belong to Behavioral Pattern.

Composite Design Pattern

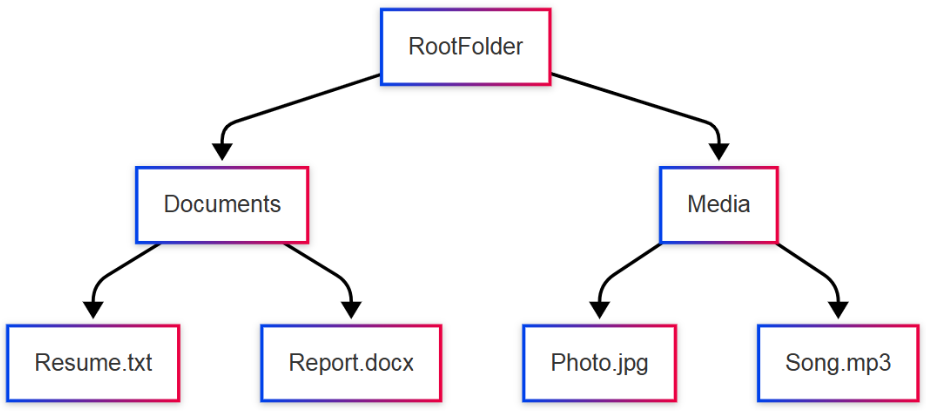
Intent

Composite Design Pattern structures objects into tree-like hierarchies to represent whole-part relationships. The Composite pattern enables clients to work with both single objects and groups of objects consistently.

Motivation

File management systems allow users to organize complex directory structures from simple files and folders. Users can group files into folders, which can then be grouped into higher-level directories, forming nested hierarchies. A straightforward implementation might define classes for basic file types, like Documents and Images, along with other classes that serve as containers, such as Folders to hold these files and other folders.

But treating files and folders differently in code can complicate the design, even though users interact with them similarly. The Composite pattern solves this by introducing a common abstract class, FileSystemEntity, that represents both files and folders. Primitive classes like TextFile or ImageFile implement basic operations but have no children. The Folder class, as a composite, can contain other FileSystemEntity objects and delegate operations like getSize() or remove() to its children. Since both files and folders share the same interface, the system can handle them uniformly, simplifying file management and supporting recursive folder structures.



Applicability

The Composite pattern is useful for building tree-like structures. It has two types of elements: simple ones (leaves) and complex ones (containers). Both types share the same interface. A container can hold leaves or other containers. This pattern helps when the client code should treat all elements the same way. Since all elements use the same interface, the client doesn't need to know if it's working with a single item or a group.

Structure

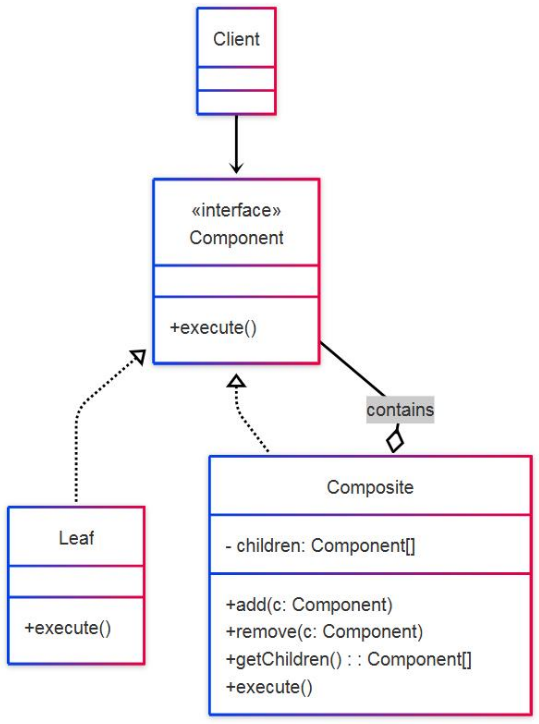


Figure 1: UML design for Composite Patterns

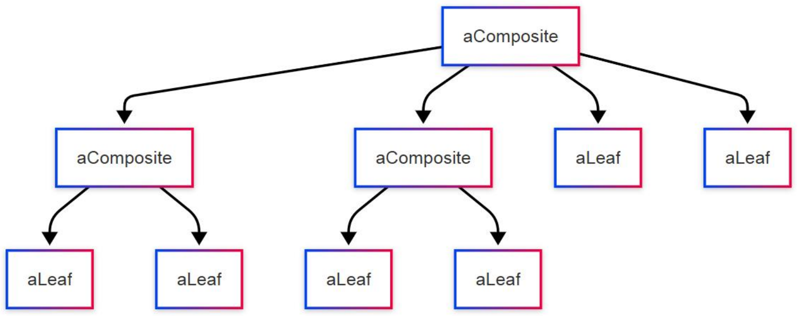


Figure 2: Object Structure

Participants

- **Component:** FileSystemItem
  - Declares the interface for all file system elements (both files and directories).
  - Can define default operations such as getName(), getSize(), or display().
  - May include methods to manage child items like add(), remove(), and getChildren(), even if not all subclasses use them.
  - May include a reference to a parent directory if navigating upward in the structure is needed.
- **Leaf:** File (TextFile, ImageFile, ExecutableFile)
  - Represents individual files (no children).
  - Implements behaviour like returning size, displaying file info, opening the file, etc.

    - Does not support child management methods.
- **Composite:** Directory or Folder
    - Represents directories that can contain other FileSystemItem objects (both files and directories).
    - Implements behavior specific to managing a collection of children (add/remove/get children).
    - Delegates operations (like calculating total size or displaying contents) to its children.
    - Implements the child management methods defined in FileSystemItem.
- Client: User or FileManagerApp
    - Uses the FileSystemItem interface to interact with all elements uniformly.
    - Can call methods like displayStructure() or getTotalSize() on any file or folder without worrying about their exact type.

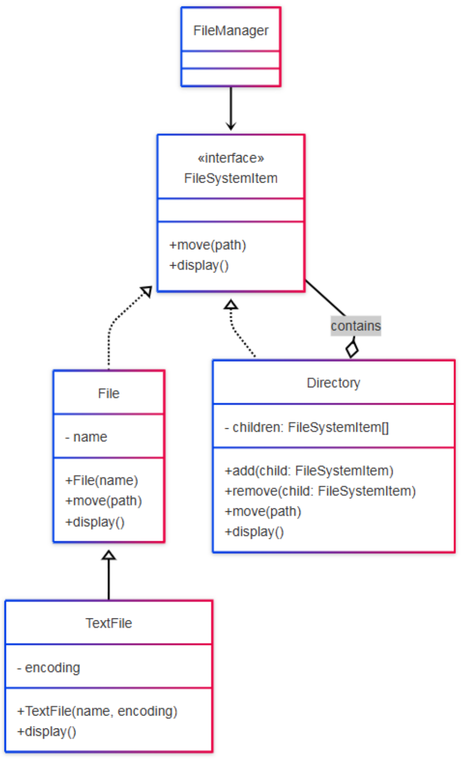
Collaborations

Clients interact with the objects in a composite structure through the interface defined by the Component class. When a Leaf receives a request, it processes it on its own. However, when a Composite receives the request, it typically passes the request along to its child components. In some cases, it may also perform extra operations either before or after delegating the request to its children.

Consequences

- Defines hierarchies of simple and composite objects that can be nested recursively.
- Simplifies client code by treating individual and grouped objects uniformly.
- Supports easy extension with new component types without altering existing code.
- Lacks strict control, making it harder to enforce specific child types in composites.

Pseudocode



Implementation

To implement the Composite pattern, one should first ensure that the application's core structure can be modelled as a tree. The model should be broken down into basic elements and containers. Containers must be capable of holding both individual elements and other containers. Next, a component interface should be defined, containing methods that are relevant to both simple and complex components. Leaf classes are then created to represent the simplest elements. There may be several types of leaf classes, depending on the application's needs. A container class is also implemented to represent complex elements. This class should include a field, typically an array or list, to store child components. This field must be of the component interface type to allow storage of both leaves and containers. When implementing the interface methods, the container should delegate most operations to its child components. Methods for adding and removing child elements should also be implemented in the container class. Optionally, these child management methods can be included in the component interface. Although this violates the Interface Segregation Principle since leaf classes will leave these methods unimplemented it allows clients to interact with all components uniformly when constructing the tree.

Advantages and Disadvantages

Advantages

- Uniform interface for simple and complex components.
- Easy to add new components.
- Supports recursive tree structures.
- Enhances code extensibility and flexibility.

Disadvantages

- May introduce design complexity.
- Difficult to restrict operations for specific components.
- Can lead to high memory usage.
- Reduced performance for deep or frequently updated trees.

Relations with Other Patterns

- The Builder pattern is useful for creating complex Composite trees by enabling recursive construction steps.
- Chain of Responsibility often complements Composite, allowing leaf nodes to pass requests up through their parent chain to the root.
- Iterators can traverse Composite trees, while the Visitor pattern executes operations across the entire tree.
- To save memory, shared leaf nodes in a Composite can be implemented as Flyweights.
- Though Composite and Decorator have similar recursive structures, Decorators wrap a single child to add behaviour, while Composites combine results from multiple children.
- These patterns can cooperate, with Decorators enhancing specific components inside a Composite tree.
- Designs relying on Composite and Decorator may benefit from the Prototype pattern to clone complex structures instead of rebuilding them.

Code Implementation: Composite Design Pattern

Code implemented here is the implementation of a File system.

The structure of the code

- **Component:** file_system_component
    - Declares the interface for all file system elements
    - Defines default operations such as getName(), getSize(), display() and composite functions add(), remove() and getChild().
- **Leaf:** file
    - Represents individual files.
    - Private properties: Name and Size.
    - Inherits getName(), getSize(), display().
    - Does not support child management methods.
- **Composite:** Directory
    - Represents directories that can contain other FileSystemItem objects both files and directories.
    - Private properties: Name, Children list.
    - Inherits: getName(), getSize(), display(), add(), remove(), getChild().
    - Extra child operation: getChildCount().
- Client: User or FileManagerApp
    - Uses the FileSystemItem interface to interact with all elements uniformly.

Project file Structure:

Github link: https://github.com/shifat65/SE_Lab_project_composite.git

```
FileSystemComposite
├── include
│   ├── file_system_component.h   # Abstract base class
│   ├── file.h                    # Leaf class (File)
│   └── directory.h               # Composite class (Directory)
├── src
│   ├── file_system_component.cpp
│   ├── file.cpp
│   ├── directory.cpp
│   └── main.cpp                  # program
├── tests
│   └── test_file_system.cpp      # Unit tests
│   └── CMakeLists.txt
├── googletest                    # Google Test framework
├── CMakeLists.txt
└── README.md
```

Code:

include/file_system_component.h

```
#ifndef FILE_SYSTEM_COMPONENT_H
#define FILE_SYSTEM_COMPONENT_H

#include <string>
#include <vector>
#include <memory>

class FileSystemComponent {
public:
    virtual ~FileSystemComponent() = default;
    virtual std::string getName() const = 0;
    virtual size_t getSize() const = 0;
    virtual void display(int indent = 0) const = 0;

    // For composite operations
    virtual void add(std::shared_ptr<FileSystemComponent>
component);
    virtual void remove(std::shared_ptr<FileSystemComponent>
component);
    virtual std::shared_ptr<FileSystemComponent> getChild(int index)
const;
};

#endif
```

include/file.h

```
#ifndef FILE_H
#define FILE_H

#include "file_system_component.h"

class File : public FileSystemComponent {
public:
    File(const std::string& name, size_t size);
    std::string getName() const override;
    size_t getSize() const override;
    void display(int indent = 0) const override;

private:
    std::string name_;
    size_t size_;
};

#endif
```

include/directory.h

```
#ifndef DIRECTORY_H
#define DIRECTORY_H

#include "file_system_component.h"
#include <vector>
```

```
class Directory : public FileSystemComponent {
public:
    Directory(const std::string& name);
    std::string getName() const override;
    size_t getSize() const override;
    void display(int indent = 0) const override;

    // Composite operations
    void add(std::shared_ptr<FileSystemComponent> component)
override;
    void remove(std::shared_ptr<FileSystemComponent> component)
override;
    std::shared_ptr<FileSystemComponent> getChild(int index) const
override;
    size_t getChildCount() const;

private:
    std::string name_;
    std::vector<std::shared_ptr<FileSystemComponent>> children_;
};

#endif
```

src/file_system_component.cpp

```
#include "file_system_component.h"
#include "iostream"

void FileSystemComponent::add(std::shared_ptr<FileSystemComponent>
component) {
    throw std::runtime_error("Unsupported operation: add()");
}

void
FileSystemComponent::remove(std::shared_ptr<FileSystemComponent>
component) {
    throw std::runtime_error("Unsupported operation: remove()");
}

std::shared_ptr<FileSystemComponent>
FileSystemComponent::getChild(int index) const {
    throw std::runtime_error("Unsupported operation: getChild()");
}
```

src/file.cpp

```
#include "file.h"
#include <iostream>

File::File(const std::string& name, size_t size) : name_(name),
size_(size) {}

std::string File::getName() const { return name_; }
```

```
size_t File::getSize() const { return size_; }

void File::display(int indent) const {
    std::cout << std::string(indent, ' ') << "- " << name_
              << " (" << size_ << " bytes)" << std::endl;
}
```

src/directory.cpp

```
#include "directory.h"
#include "iostream"
#include <algorithm>

Directory::Directory(const std::string& name) : name_(name) {}

std::string Directory::getName() const { return name_; }

size_t Directory::getSize() const {
    size_t totalSize = 0;
    for (const auto& child : children_) {
        totalSize += child->getSize();
    }
    return totalSize;
}

void Directory::display(int indent) const {
    std::cout << std::string(indent, ' ') << "+ " << name_
              << " (dir, " << getSize() << " bytes)" << std::endl;
    for (const auto& child : children_) {
        child->display(indent + 2);
    }
}

void Directory::add(std::shared_ptr<FileSystemComponent> component)
{
    children_.push_back(component);
}

void Directory::remove(std::shared_ptr<FileSystemComponent>
component) {
    children_.erase(std::remove(children_.begin(), children_.end(),
component),
                  children_.end());
}

std::shared_ptr<FileSystemComponent> Directory::getChild(int index)
const {
    if (index < 0 || index >= children_.size()) {
        return nullptr;
    }
    return children_[index];
}

size_t Directory::getChildCount() const {
    return children_.size();
}
```

src/main.cpp

```
#include "iostream"
#include <memory>
#include "directory.h"
#include "file.h"

int main() {
    // Create files
    auto resume = std::make_shared<File>("resume.pdf", 2500);
    auto vacationPhoto = std::make_shared<File>("vacation.jpg",
4200);
    auto notes = std::make_shared<File>("notes.txt", 300);
    auto budget = std::make_shared<File>("budget.xlsx", 1800);

    // Create directories
    auto homeDir = std::make_shared<Directory>("Home");
    auto documentsDir = std::make_shared<Directory>("Documents");
    auto picturesDir = std::make_shared<Directory>("Pictures");
    auto projectsDir = std::make_shared<Directory>("Projects");

    // Build the file system structure
    homeDir->add(documentsDir);
    homeDir->add(picturesDir);
    homeDir->add(projectsDir);

    documentsDir->add(resume);
    documentsDir->add(notes);
    documentsDir->add(budget);

    picturesDir->add(vacationPhoto);

    // Display the file system
    std::cout << "File System Structure:" << std::endl;
    homeDir->display();

    // Show statistics
    std::cout << "\nStatistics:" << std::endl;
    std::cout << "Total size of Home: " << homeDir->getSize() << "
bytes" << std::endl;
    std::cout << "Documents folder size: " << documentsDir-
>getSize() << " bytes" << std::endl;
    std::cout << "Number of items in Documents: "
              << documentsDir->getChildCount() << std::endl;

    return 0;
}
```

test/test_file_system.cpp

```
#include <gtest/gtest.h>
#include <memory>
#include "directory.h"
#include "file.h"

class FileSystemTest : public ::testing::Test {
protected:
    void SetUp() override {
        file1 = std::make_shared<File>("report.doc", 1200);
        file2 = std::make_shared<File>("presentation.ppt", 3500);
        rootDir = std::make_shared<Directory>("Root");
    }

    std::shared_ptr<File> file1;
    std::shared_ptr<File> file2;
    std::shared_ptr<Directory> rootDir;
};

// Component tests
TEST_F(FileSystemTest, FileCreation) {
    EXPECT_EQ(file1->getName(), "report.doc");
    EXPECT_EQ(file1->getSize(), 1200);

    testing::internal::CaptureStdout();
    file1->display();
    std::string output = testing::internal::GetCapturedStdout();
    EXPECT_EQ(output, "- report.doc (1200 bytes)\n");
}

TEST_F(FileSystemTest, DirectoryCreation) {
    EXPECT_EQ(rootDir->getName(), "Root");
    EXPECT_EQ(rootDir->getSize(), 0);
    EXPECT_EQ(rootDir->getChildCount(), 0);
}

// Directory Operations Tests
TEST_F(FileSystemTest, AddSingleFile) {
    rootDir->add(file1);
    EXPECT_EQ(rootDir->getChildCount(), 1);
    EXPECT_EQ(rootDir->getSize(), 1200);
    EXPECT_EQ(rootDir->getChild(0), file1);
}

TEST_F(FileSystemTest, AddMultipleFiles) {
    rootDir->add(file1);
    rootDir->add(file2);
    EXPECT_EQ(rootDir->getChildCount(), 2);
    EXPECT_EQ(rootDir->getSize(), 4700);
}

TEST_F(FileSystemTest, RemoveFile) {
    rootDir->add(file1);
    rootDir->remove(file1);
    EXPECT_EQ(rootDir->getChildCount(), 0);
    EXPECT_EQ(rootDir->getSize(), 0);
}

TEST_F(FileSystemTest, RemoveNonexistentFile) {
    rootDir->add(file1);
    rootDir->remove(file2); // Not added
    EXPECT_EQ(rootDir->getChildCount(), 1); // Should remain
unchanged
}

// Edge Case Tests
TEST_F(FileSystemTest, GetInvalidChildIndex) {
    rootDir->add(file1);
    EXPECT_EQ(rootDir->getChild(-1), nullptr); // Negative index
    EXPECT_EQ(rootDir->getChild(1), nullptr); // Out of bounds
}

TEST_F(FileSystemTest, EmptyDirectoryOperations) {
    EXPECT_EQ(rootDir->getChild(0), nullptr);
    testing::internal::CaptureStdout();
    rootDir->display();
    std::string output = testing::internal::GetCapturedStdout();
    EXPECT_EQ(output, "+ Root (dir, 0 bytes)\n");
}

//Nested Structure Tests

TEST_F(FileSystemTest, NestedDirectories) {
    auto subDir = std::make_shared<Directory>("Subfolder");
    subDir->add(file1);
    rootDir->add(subDir);

    EXPECT_EQ(rootDir->getChildCount(), 1);
    EXPECT_EQ(rootDir->getSize(), 1200);
    EXPECT_EQ(subDir->getSize(), 1200);
}

TEST_F(FileSystemTest, DeeplyNestedStructure) {
    auto dir1 = std::make_shared<Directory>("Level1");
    auto dir2 = std::make_shared<Directory>("Level2");
    auto dir3 = std::make_shared<Directory>("Level3");

    dir3->add(file1);
    dir2->add(dir3);
    dir1->add(dir2);
    rootDir->add(dir1);

    EXPECT_EQ(rootDir->getSize(), 1200);
    EXPECT_EQ(dir1->getSize(), 1200);
    EXPECT_EQ(dir2->getSize(), 1200);
    EXPECT_EQ(dir3->getSize(), 1200);
}
// Invalud Operation Tests
TEST_F(FileSystemTest, InvalidOperationsOnFiles) {
    EXPECT_THROW(file1->add(file2), std::runtime_error);
    EXPECT_THROW(file1->remove(file2), std::runtime_error);
    EXPECT_THROW(file1->getChild(0), std::runtime_error);
}
// Display formatting tests
TEST_F(FileSystemTest, DisplayFormatting) {
    auto subDir = std::make_shared<Directory>("Sub");
    subDir->add(file1);
    rootDir->add(subDir);
    rootDir->add(file2);

    testing::internal::CaptureStdout();
    rootDir->display();
    std::string output = testing::internal::GetCapturedStdout();

    std::string expected =
        "+ Root (dir, 4700 bytes)\n"
        "  + Sub (dir, 1200 bytes)\n"
        "    - report.doc (1200 bytes)\n"
        "  - presentation.ppt (3500 bytes)\n";

    EXPECT_EQ(output, expected);
}
//Stress Tests
TEST_F(FileSystemTest, LargeNumberOfFiles) {
    const int NUM_FILES = 1000;
    for (int i = 0; i < NUM_FILES; i++) {
        auto file = std::make_shared<File>("file" +
std::to_string(i), i);
        rootDir->add(file);
    }
    EXPECT_EQ(rootDir->getChildCount(), NUM_FILES);
    EXPECT_EQ(rootDir->getSize(), NUM_FILES * (NUM_FILES - 1) / 2);
// Sum of 0..999
}

TEST_F(FileSystemTest, DeepDirectoryTree) {
    std::shared_ptr<Directory> current = rootDir;
    const int DEPTH = 50;

    for (int i = 0; i < DEPTH; i++) {
        auto newDir = std::make_shared<Directory>("Level" +
std::to_string(i));
        current->add(newDir);
        current = newDir;
    }

    current->add(file1);
    EXPECT_EQ(rootDir->getSize(), 1200);
    EXPECT_EQ(rootDir->getChildCount(), 1);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

test/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

# Set include directories for tests
include_directories(
    ${CMAKE_SOURCE_DIR}/include
    ${CMAKE_SOURCE_DIR}/googletest/googletest/include
)

# Create test executable
add_executable(file_system_tests
    test_file_system.cpp
    ../src/file_system_component.cpp
    ../src/file.cpp
    ../src/directory.cpp
)

# Link with Google Test libraries
target_link_libraries(file_system_tests
    PRIVATE
    gtest
    gtest_main
)

# Add test
add_test(NAME file_system_tests COMMAND file_system_tests)
```

CMakeLists.txt (root)

```
cmake_minimum_required(VERSION 3.10)
project(CompositePattern)

set(CMAKE_CXX_STANDARD 17)

# Set include directories
include_directories(
    ${CMAKE_SOURCE_DIR}/include
)

# Add Google Test as a subdirectory (if using local copy)
add_subdirectory(googletest)

# Main executable
add_executable(file_system_demo
    src/main.cpp
    src/file_system_component.cpp
    src/file.cpp
    src/directory.cpp
)

# Testing
enable_testing()
add_subdirectory(tests)
```

Sample Output:

```
File System Structure:
        + Home (dir, 8800 bytes)
        + Documents (dir, 4600 bytes)
                - resume.pdf (2500 bytes)
                - notes.txt (300 bytes)
                - budget.xlsx (1800 bytes)
        + Pictures (dir, 4200 bytes)
                - vacation.jpg (4200 bytes)
        + Projects (dir, 0 bytes)

Statistics:
Total size of Home: 8800 bytes
Documents folder size: 4600 bytes
Number of items in Documents: 3
```

Testing approach

The test approach followed here is a top-down approach. Development starts with the main module, using stubs for lower-level modules. Each stub is gradually replaced with the actual module, followed by testing. After each integration, regression testing can be done to check for new issues.

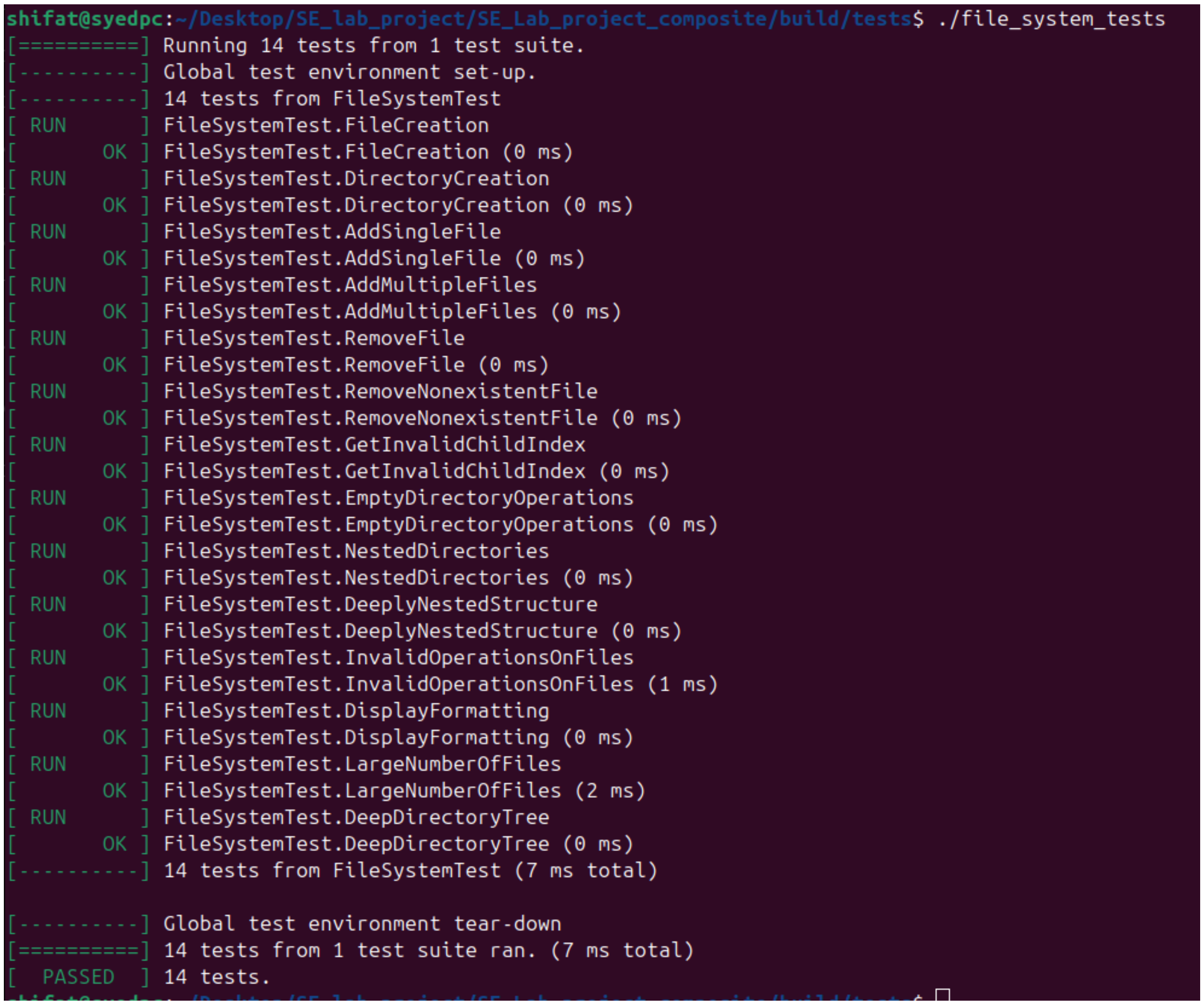
Here, the upper-level module, like file_system_components, was tested first. Then the directory system is tested. And then lower-level files are tested.

Tests

The project includes 14 unit tests covering:

- File Operations Path
  - File creation and properties
  - File display formatting
  - Invalid operations on files
- Directory Operations Path
  - Empty directory operations
  - Single file operations
  - Multiple file operations
  - File removal cases
- Nested Structure Path
  - Shallow nesting
  - Deep nesting
  - Mixed files and directories
- Error Handling Path
  - Invalid indices
  - Operations on wrong types
  - Nonexistent removal
- Display Formatting Path
  - Empty display
  - Simple display
  - Complex hierarchical display
- Stress Test Path
  - Large number of files
  - Deep directory structures
  - Memory management

Google test result:



```
shifat@syedpc:~/Desktop/SE_lab_project/SE_Lab_project_composite/build/tests$ ./file_system_tests
[==========] Running 14 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 14 tests from FileSystemTest
[ RUN      ] FileSystemTest.FileCreation
[       OK ] FileSystemTest.FileCreation (0 ms)
[ RUN      ] FileSystemTest.DirectoryCreation
[       OK ] FileSystemTest.DirectoryCreation (0 ms)
[ RUN      ] FileSystemTest.AddSingleFile
[       OK ] FileSystemTest.AddSingleFile (0 ms)
[ RUN      ] FileSystemTest.AddMultipleFiles
[       OK ] FileSystemTest.AddMultipleFiles (0 ms)
[ RUN      ] FileSystemTest.RemoveFile
[       OK ] FileSystemTest.RemoveFile (0 ms)
[ RUN      ] FileSystemTest.RemoveNonexistentFile
[       OK ] FileSystemTest.RemoveNonexistentFile (0 ms)
[ RUN      ] FileSystemTest.GetInvalidChildIndex
[       OK ] FileSystemTest.GetInvalidChildIndex (0 ms)
[ RUN      ] FileSystemTest.EmptyDirectoryOperations
[       OK ] FileSystemTest.EmptyDirectoryOperations (0 ms)
[ RUN      ] FileSystemTest.NestedDirectories
[       OK ] FileSystemTest.NestedDirectories (0 ms)
[ RUN      ] FileSystemTest.DeeplyNestedStructure
[       OK ] FileSystemTest.DeeplyNestedStructure (0 ms)
[ RUN      ] FileSystemTest.InvalidOperationsOnFiles
[       OK ] FileSystemTest.InvalidOperationsOnFiles (1 ms)
[ RUN      ] FileSystemTest.DisplayFormatting
[       OK ] FileSystemTest.DisplayFormatting (0 ms)
[ RUN      ] FileSystemTest.LargeNumberOfFiles
[       OK ] FileSystemTest.LargeNumberOfFiles (2 ms)
[ RUN      ] FileSystemTest.DeepDirectoryTree
[       OK ] FileSystemTest.DeepDirectoryTree (0 ms)
[----------] 14 tests from FileSystemTest (7 ms total)

[----------] Global test environment tear-down
[==========] 14 tests from 1 test suite ran. (7 ms total)
[  PASSED  ] 14 tests.
```

=0=