The composite pattern

START

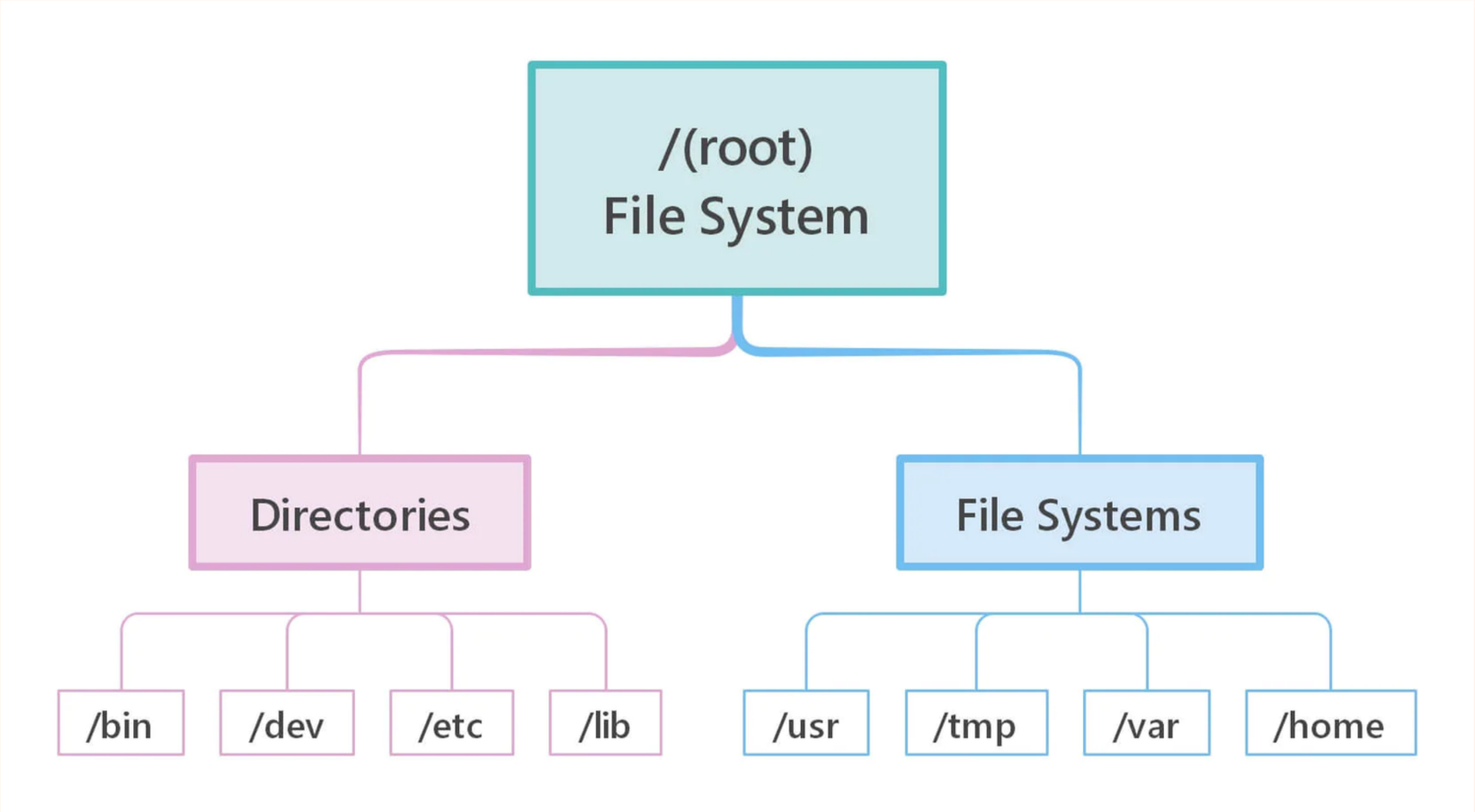
Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

The composite pattern

START

Example: File System



Definition:

The Composite Pattern is a design pattern used in object-oriented programming to represent part-whole hierarchies of objects. It allows you to treat individual objects and groups of objects in a uniform way.

Structure of Composite Pattern

Component: This is the base interface or abstract class that defines the operations that are common to both simple and complex objects in the composition. The component defines the behavior of the objects in the hierarchy and provides an interface for manipulating them.

Leaf: This represents the simplest or smallest objects in the composition. These objects do not have any child components. They implement the operations defined by the Component interface.

Structure of Composite Pattern

Composite: This is a complex object that has child components. The composite itself is a Component and contains a collection of child Components. The composite implements the operations defined by the Component interface, but may also provide additional operations to manipulate its child components.

Client: This is the application that uses the Component interface to manipulate the objects in the hierarchy. The client works with the components in the hierarchy through the Component interface, so it doesn't need to know whether it is working with a Leaf or a Composite.

Implemetation

```cpp
#include <iostream>
#include <list>
#include <string>

// Component interface
class Component {
public:
    virtual void add(Component* component) = 0;
    virtual void remove(Component* component) = 0;
    virtual void display() = 0;
};

// Leaf class
class File : public Component {
public:
    File(std::string name) : name(name) {}

    void add(Component* component) override {}
    void remove(Component* component) override {}
    void display() override {
        std::cout << "File: " << name << std::endl;
    }

private:
    std::string name;
};
```

```cpp
// Composite class
class Folder : public Component {
public:
    Folder(std::string name) : name(name) {}

    void add(Component* component) override {
        children.push_back(component);
    }
    void remove(Component* component) override {
        children.remove(component);
    }
    void display() override {
        std::cout << "Folder: " << name << std::endl;
        for (auto child : children) {
            child->display();
        }
    }

private:
    std::string name;
    std::list<Component*> children;
};

// Client code
int main() {
    // Create a file
    Component* file = new File("file.txt");

    // Create a folder and add the file to it
    Component* folder = new Folder("Folder 1");
    folder->add(file);

    // Create another folder and add the first folder to it
    Component* folder2 = new Folder("Folder 2");
    folder2->add(folder);

    // Display the contents of the second folder (which should show both the file and the first folder)
    folder2->display();

    // Clean up memory
    delete file;
    delete folder;
    delete folder2;

    return 0;
}
```

Avantages

1. Flexible hierachy
2. Uniform interface
3. Simplifies client code
4. Encapsulates complexity

Disavantages

1. Overhead
2. Limited functionatlity
3. Difficult to restrict component types
4. Difficult to maintain

THANK YOU!