Object Oriented programming and software engineering – Lab 22

Adam Korytowski – 2025

1. Composite design pattern

What is the Composite Pattern?

The Composite pattern is a structural design pattern that allows you to treat individual objects and compositions of objects uniformly. It organizes objects into tree structures to represent part-whole hierarchies, so clients can interact with single objects and groups of objects through the same interface.

Why use the Composite Pattern?

- To simplify client code by allowing it to treat single objects and composites identically.
- To build complex tree-like structures where components can be simple leaves or containers (composites) of other components.
- To support recursive compositions, where composites can contain other composites.
- To enable flexible and extensible designs where you can add new components or composites without changing client code.

When to use the Composite Pattern?

- When you need to represent hierarchical part-whole relationships.
- When you want to manipulate groups and individual objects uniformly.
- When you need to add or remove components dynamically.
- When client code must be agnostic to whether it works with a single object or a composition.

Examples:

– Reasons to use Composite for Mage, Warrior, and Character classes:

1. You can control one character or a whole team the same way — just call attack() or move() without worrying if it's a single warrior or many characters.
2. You can easily build teams made of different characters or smaller teams, making it simple to organize and manage groups in your game.

– Code example – Vehicle class:

In the vehicle domain, you often deal with groups of vehicles that operate together as a unit (e.g., convoys, fleets, patrols). Using Composite allows you to:

- Model individual vehicles (Car, Truck) as well as groups of vehicles (e. g. Convoy) using the same Vehicle interface.
- Write client code that calls move() or other actions on a vehicle, without caring if it's a single car or a whole convoy.
- Easily compose complex vehicle structures (fleets of convoys, convoys of trucks and cars) recursively.
- Extend the system by adding new composite types (like e. g. EcoFleet) that add new behaviors without rewriting client code.

This leads to more maintainable, scalable, and flexible code, which is especially important in simulations, games, or logistics software where vehicles often act in groups.

```
class Vehicle
{
public:
    virtual void move() const = 0;
    virtual ~Vehicle() {}
};
```

```cpp
class Car : public Vehicle
{
private:
    std::string name;
public:
    Car(const std::string& name) : name(name) {}

    void move() const override
    {
        std::cout << "Car " << name << " drives on the road." << std::endl;
    }
};

class Truck : public Vehicle
{
private:
    std::string name;
public:
    Truck(const std::string& name) : name(name) {}

    void move() const override
    {
        std::cout << "Truck " << name << " hauls cargo." << std::endl;
    }
};
```

```cpp
class Convoy : public Vehicle
{
private:
    std::vector<Vehicle*> vehicles;
public:
    void add(Vehicle* vehicle)
    {
        vehicles.push_back(vehicle);
    }

    void move() const override
    {
        for (const auto& v : vehicles)
        {
            v->move();
        }
    }

    ~Convoy()
    {
        for (auto v : vehicles)
        {
            delete v;
        }
    }
};
```

Usage example:

```
Vehicle* car = new Car("Car");
Vehicle* truck = new Truck("Truck");

Convoy* convoy = new Convoy();
convoy->add(car);
convoy->add(truck);
convoy->move();
```

2. Tasks

Use your code from previous laboratories (having Base and Child classes)

- Refactor your class hierarchy to support composite objects. Create at least one class that acts as a container for other objects of the base class type. Demonstrate usage where composites contain both leaf and other composite objects (5 pts)
- Create a composite class that performs an action on a group of objects and accumulates a specific metric returned by each object (e.g., energy usage, score, risk level). The composite should output or process the total result after invoking the action on all its children. Design it so that it works regardless of the specific type of the leaf classes (e.g., vehicles, characters, sensors), as long as they share a common interface (3 pts)

Example:

```
double getCO2Emission() const override
{
    return 0.12; // e.g., 0.12 kg per move
}
```

```cpp
class EcoFleet : public Vehicle
{
private:
    std::vector<Vehicle*> vehicles;
public:
    void add(Vehicle* v)
    {
        vehicles.push_back(v);
    }

    void move() const override
    {
        std::cout << "EcoFleet dispatching vehicles:" << std::endl;
        double totalCO2 = 0.0;

        for (const auto& v : vehicles)
        {
            v->move();
            totalCO2 += v->getCO2Emission();
        }

        std::cout << "Total CO2 emissions: " << totalCO2 << " kg" << std::endl;
    }

    ~EcoFleet()
    {
        for (auto v : vehicles)
        {
            delete v;
        }
    }
};
```

Usage example:

```cpp
EcoFleet* ecoFleet = new EcoFleet();
ecoFleet->add(convoy);
ecoFleet->add(anotherCar);

ecoFleet->move();
```