

Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 1

**Discipline**: TMPS

Elaborated: FAF-222, Ostafi Eugen

Checked: asist. univ.,

Chișinău 2024

**Project Title: Creational Design Patterns in C++**

**Objectives**

1. Study and understand Creational Design Patterns.
2. Choose a domain and define its main classes/models/entities.
3. Implement creational design patterns to handle object instantiation within a sample project.

**Theory**

Creational design patterns provide general solutions for optimizing object creation, controlling instantiation complexity, and improving object lifecycle management. This project will explore these patterns by implementing a real-world sample domain using C++.

The following creational design patterns will be implemented:

* **Singleton**
* **Builder**
* **Factory Method**

**Domain Selection**

The domain for this project is **Vehicle Manufacturing**. The project simulates creating vehicles of different types (e.g., Car, Truck, Motorcycle) with varying configurations.

**Classes and Entities**

* **Vehicle** (Base class for Car, Truck, Motorcycle)
* **VehicleFactory** (Factory for creating vehicles)
* **VehicleBuilder** (Builder for constructing complex vehicle objects)
* **VehiclePool** (Optional - managing reusable vehicle objects)

**Project Structure**

The project will be organized into modules/packages as follows:

* **client**: Contains main function and client-side logic for interacting with Vehicle objects.
* **domain**: Contains domain-specific classes (e.g., Vehicle, Car, Truck).
* **factory**: Contains factory classes that implement the Factory Method pattern.
* **builder**: Contains builder classes for constructing complex Vehicle objects.
* **models**: Contains classes for models of each specific vehicle type (subclasses of Vehicle).

**Design Patterns Implementation**

**1. Singleton Pattern**

* **Purpose**: Ensure a single instance of the VehicleManager class, which controls the creation of all vehicle objects.
* **Implementation**:
  + Define a VehicleManager class.
  + Use a private static instance variable and a static getInstance() method to manage instantiation.

class VehicleManager {

private:

static VehicleManager\* instance;

VehicleManager() {}

public:

static VehicleManager\* getInstance() {

if (instance == nullptr) {

instance = new VehicleManager();

}

return instance;

}

void manageVehicleCreation() {

}

};

**2. Builder Pattern**

* **Purpose**: Construct Vehicle objects with complex configurations (e.g., engine type, transmission, features).
* **Implementation**:
  + Define a VehicleBuilder interface with methods like setEngineType(), setTransmission().
  + Create concrete builders (e.g., CarBuilder, TruckBuilder) for specific vehicle types.
  + Use the VehicleBuilder to progressively build a complex Vehicle instance.

class VehicleBuilder {

protected:

    Vehicle\* vehicle;

public:

    virtual ~VehicleBuilder() { delete vehicle; }

    virtual VehicleBuilder& setEngineType(const std::string& engine) = 0;

    virtual VehicleBuilder& setTransmission(const std::string& transmission) = 0;

    virtual Vehicle\* build() = 0;

};

class CarBuilder : public VehicleBuilder {

public:

    CarBuilder() { vehicle = new Car(); }

    VehicleBuilder& setEngineType(const std::string& engine) override {

        vehicle->engineType = engine;  // Direct access

        return \*this;

    }

    VehicleBuilder& setTransmission(const std::string& transmission) override {

        vehicle->transmission = transmission;  // Direct access

        return \*this;

    }

    Vehicle\* build() override {

        return vehicle;

    }

};

**3. Factory Method Pattern**

* **Purpose**: Define an interface for creating an object but allow subclasses to alter the type of objects created.
* **Implementation**:
  + Define a VehicleFactory base class with a createVehicle() method.
  + Implement CarFactory, TruckFactory, and MotorcycleFactory that override createVehicle() to create specific vehicle types.

class VehicleFactory {

public:

Vehicle\* createVehicle(const std::string& type) {

if (type == "Car") return new Car();

if (type == "Truck") return new Truck();

if (type == "Motorcycle") return new Motorcycle();

return nullptr;

}

};

**Example Usage**

int main() {

    // Using Factory Method to create vehicles

    VehicleFactory factory;

    Vehicle\* car = factory.createVehicle("Car");

    Vehicle\* truck = factory.createVehicle("Truck");

    Vehicle\* motorcycle = factory.createVehicle("Motorcycle");

    // Display vehicle info

    car->displayInfo();

    truck->displayInfo();

    motorcycle->displayInfo();

    // Using Builder Pattern to create a customized car

    CarBuilder carBuilder;

    carBuilder.setEngineType("V8").setTransmission("Automatic");

    Vehicle\* customCar = carBuilder.build();

    customCar->displayInfo();

    // Clean up

    delete car;

    delete truck;

    delete motorcycle;

    delete customCar;

    return 0;

}