The template method pattern

START

Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

The template method pattern

START

Example:
House construction

Definition:

The Template Method Pattern is a behavioral design pattern that defines the basic steps of an algorithm in a superclass, allowing its subclasses to provide specific implementations of some of those steps. In other words, it provides a skeleton of an algorithm, where some steps are left to be implemented by the subclasses..

Structure of Template Method Pattern

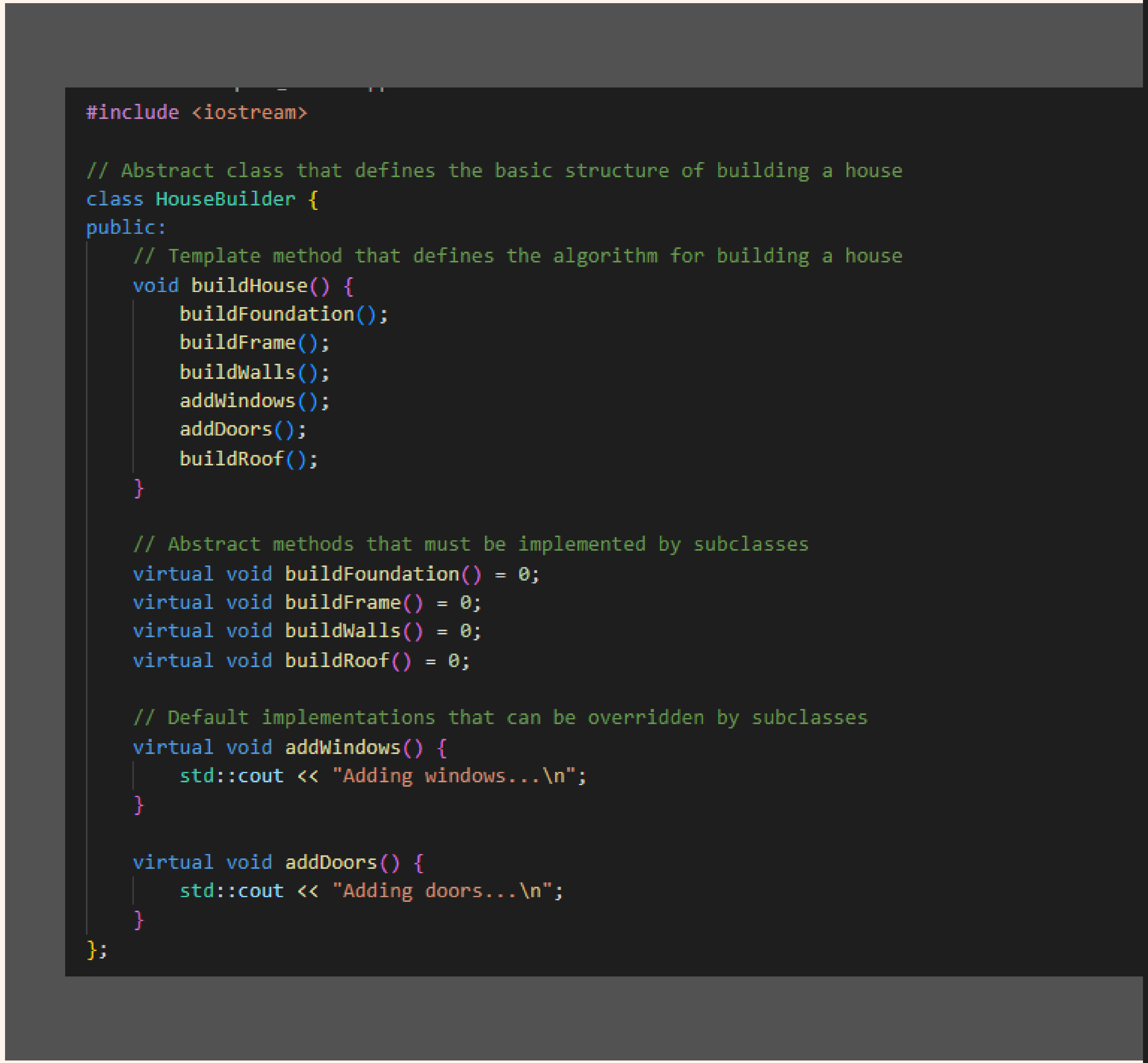
Abstract Class: Defines the basic structure of the algorithm or process, including abstract methods representing steps that can vary and concrete methods providing a default implementation for common steps.

Hook Method: A hook method is a method defined in the abstract class that provides a default implementation but can be overridden by the concrete classes if needed. This allows the concrete classes to modify the behavior of the algorithm or process at certain points.

Concrete Class: A concrete class that extends the abstract class and implements the abstract methods. Each concrete class can provide its own implementation for the abstract methods, allowing for variations in the algorithm or process.

Template Method: The template method is the main method in the abstract class that defines the overall structure of the algorithm or process by calling the abstract methods and hook methods in a specific order. This method cannot be overridden by the concrete classes.

Implemetation



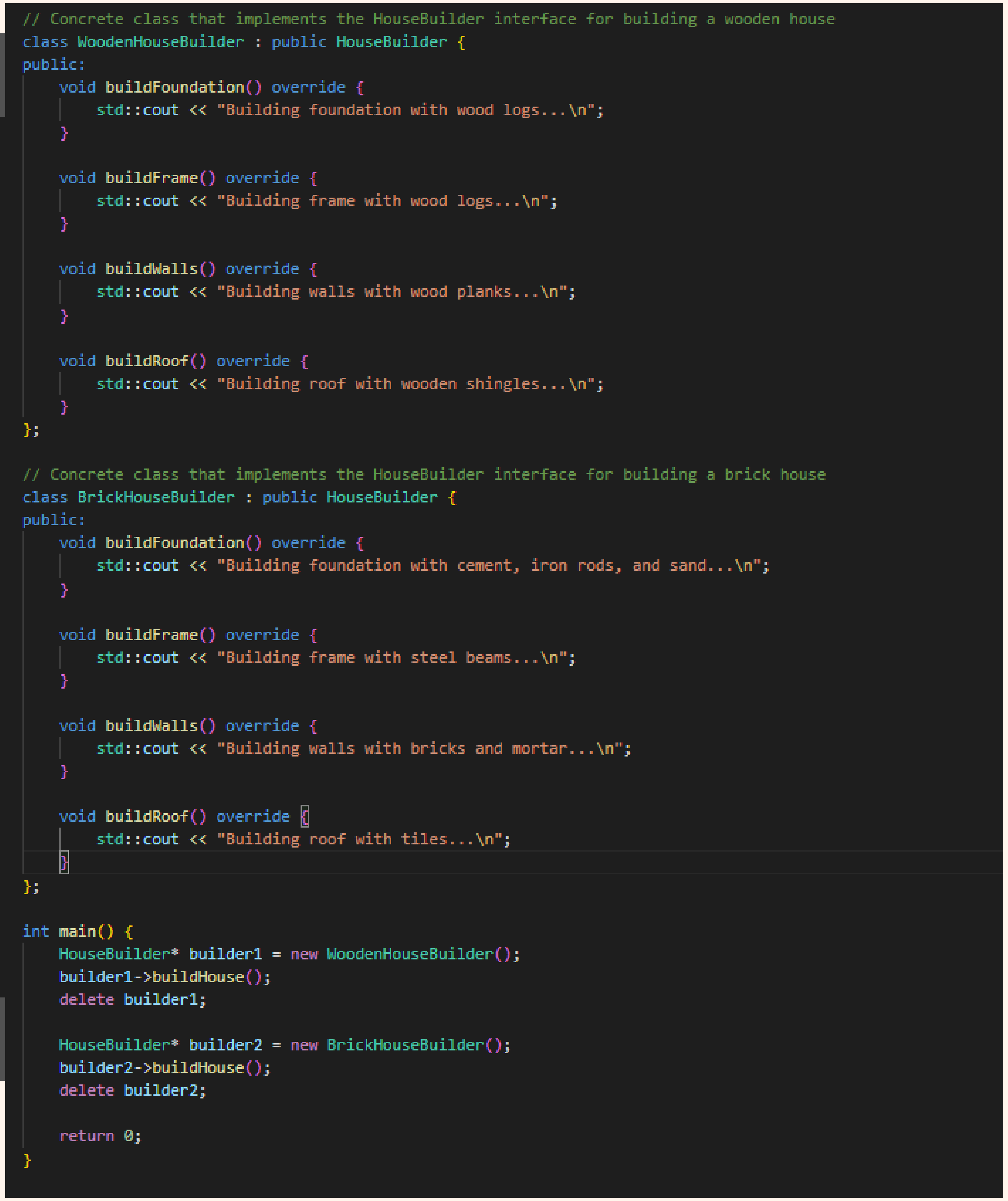
```cpp
#include <iostream>

// Abstract class that defines the basic structure of building a house
class HouseBuilder {
public:
    // Template method that defines the algorithm for building a house
    void buildHouse() {
        buildFoundation();
        buildFrame();
        buildWalls();
        addWindows();
        addDoors();
        buildRoof();
    }

    // Abstract methods that must be implemented by subclasses
    virtual void buildFoundation() = 0;
    virtual void buildFrame() = 0;
    virtual void buildWalls() = 0;
    virtual void buildRoof() = 0;

    // Default implementations that can be overridden by subclasses
    virtual void addWindows() {
        std::cout << "Adding windows...\n";
    }

    virtual void addDoors() {
        std::cout << "Adding doors...\n";
    }
};
```



```cpp
// Concrete class that implements the HouseBuilder interface for building a wooden house
class WoodenHouseBuilder : public HouseBuilder {
public:
    void buildFoundation() override {
        std::cout << "Building foundation with wood logs...\n";
    }

    void buildFrame() override {
        std::cout << "Building frame with wood logs...\n";
    }

    void buildWalls() override {
        std::cout << "Building walls with wood planks...\n";
    }

    void buildRoof() override {
        std::cout << "Building roof with wooden shingles...\n";
    }
};

// Concrete class that implements the HouseBuilder interface for building a brick house
class BrickHouseBuilder : public HouseBuilder {
public:
    void buildFoundation() override {
        std::cout << "Building foundation with cement, iron rods, and sand...\n";
    }

    void buildFrame() override {
        std::cout << "Building frame with steel beams...\n";
    }

    void buildWalls() override {
        std::cout << "Building walls with bricks and mortar...\n";
    }

    void buildRoof() override {
        std::cout << "Building roof with tiles...\n";
    }
};

int main() {
    HouseBuilder* builder1 = new WoodenHouseBuilder();
    builder1->buildHouse();
    delete builder1;

    HouseBuilder* builder2 = new BrickHouseBuilder();
    builder2->buildHouse();
    delete builder2;

    return 0;
}
```

Avantages

1 Clear separation of abstract algorithm and concrete implementations.

2 High degree of flexibility and extensibility.

3 Reduce code duplication & increase code reuse

Disadvantages

1. Tight coupling
2. Complexity
3. Difficult to understand

THANK YOU!