The state pattern

START

Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

The state pattern

START

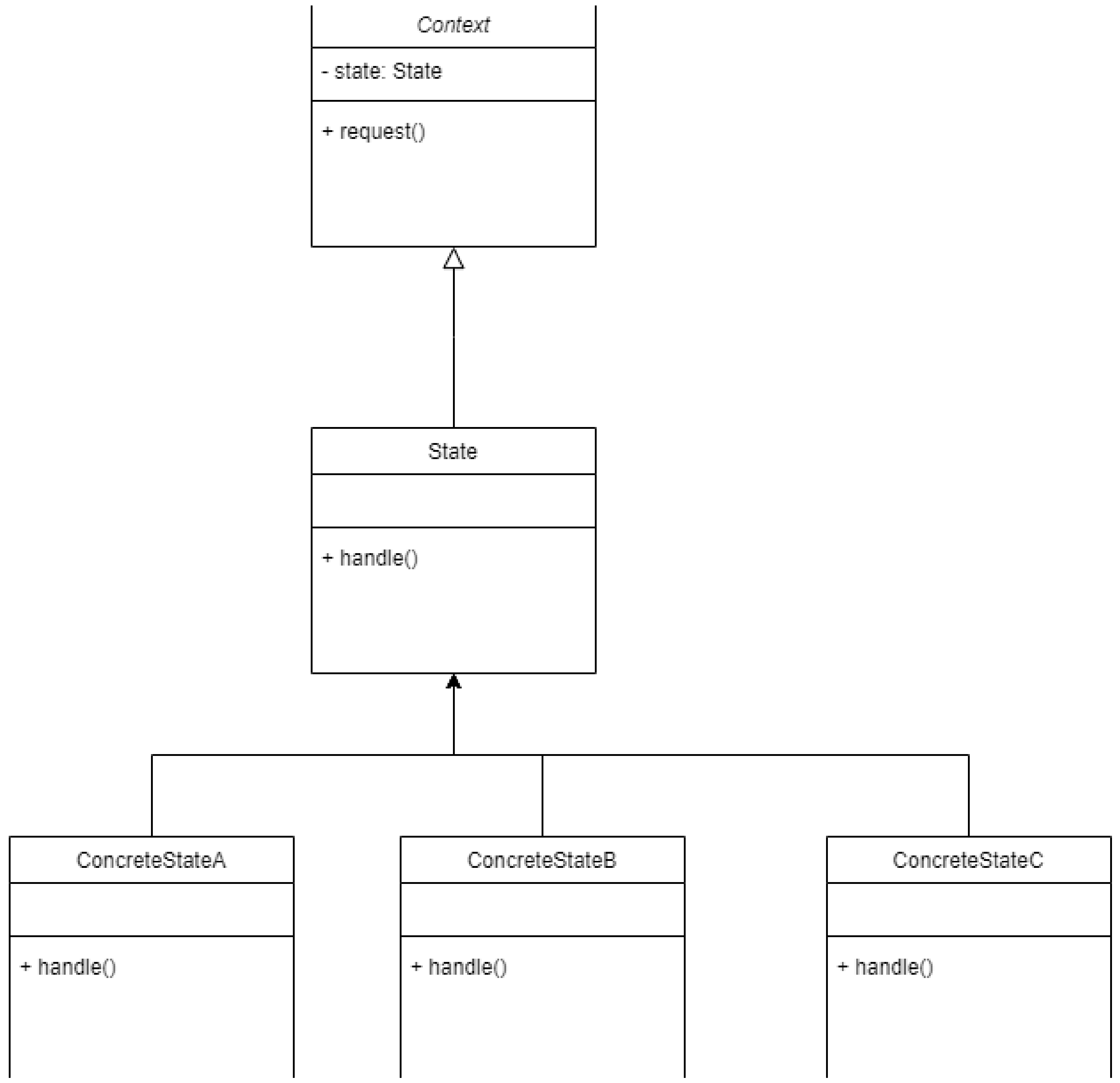
Example:
Smartphone buttons

Definition:

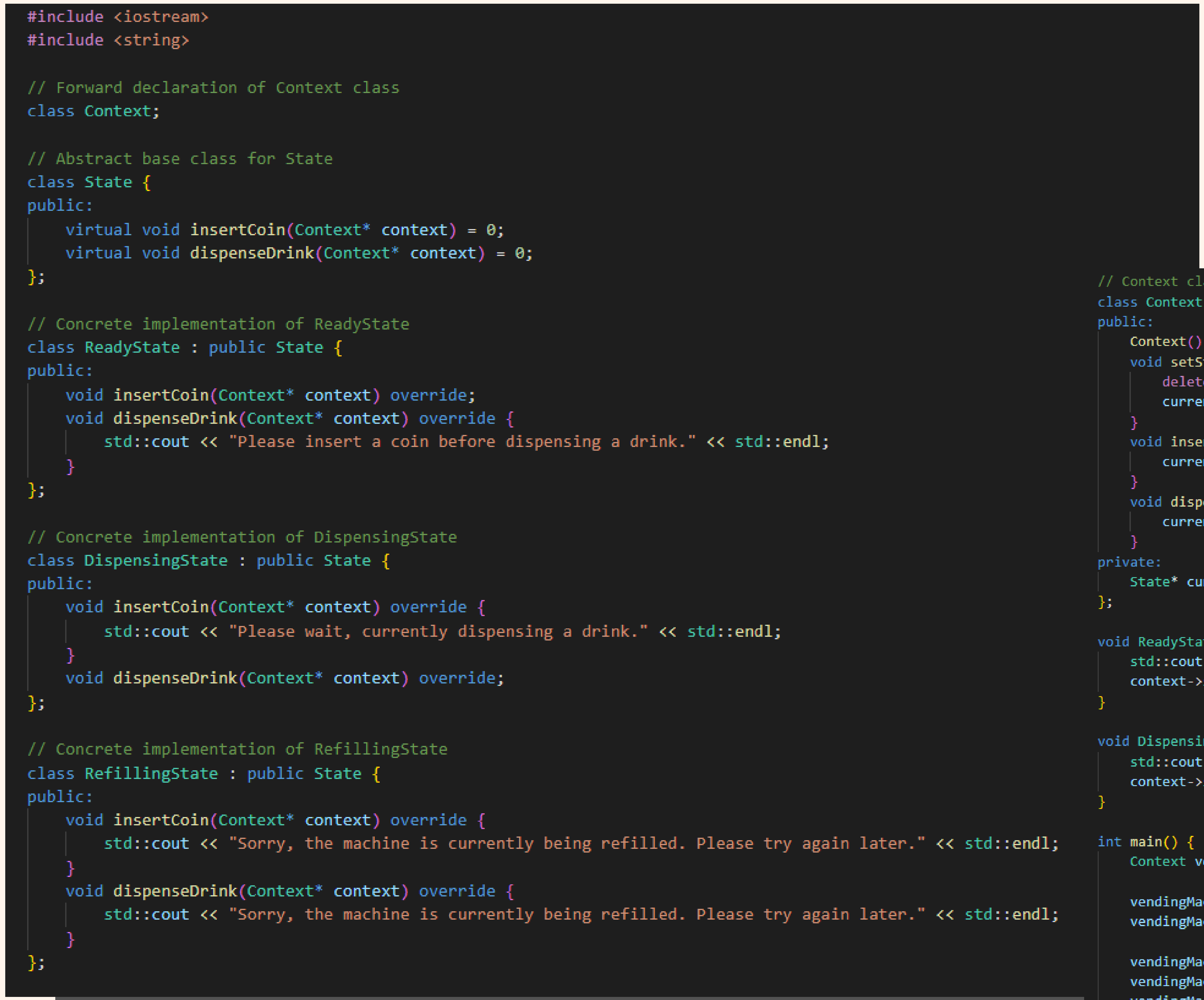
The State pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes. It is used when an object needs to behave differently based on its internal state, and the state transitions can be complex and involve multiple conditions.

.

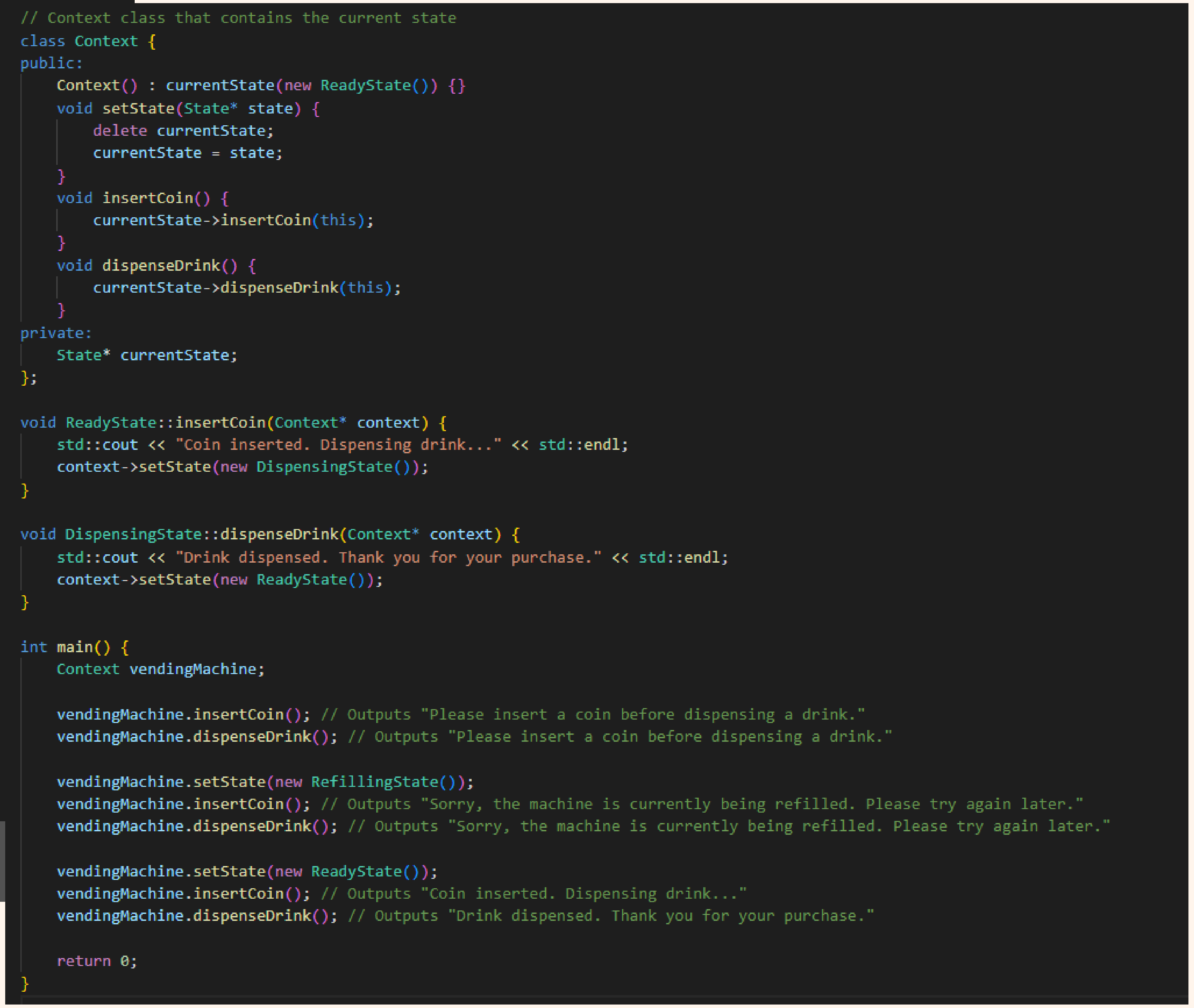
UML Diagram


Context
- state: State
+ request()
State
+ handle()
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()
ConcreteStateC
+ handle()

Implemetation



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

// Forward declaration of Context class
class Context;

// Abstract base class for State
class State {
public:
    virtual void insertCoin(Context* context) = 0;
    virtual void dispenseDrink(Context* context) = 0;
};

// Concrete implementation of ReadyState
class ReadyState : public State {
public:
    void insertCoin(Context* context) override;
    void dispenseDrink(Context* context) override {
        std::cout << "Please insert a coin before dispensing a drink." << std::endl;
    }
};

// Concrete implementation of DispensingState
class DispensingState : public State {
public:
    void insertCoin(Context* context) override {
        std::cout << "Please wait, currently dispensing a drink." << std::endl;
    }
    void dispenseDrink(Context* context) override;
};

// Concrete implementation of RefillingState
class RefillingState : public State {
public:
    void insertCoin(Context* context) override {
        std::cout << "Sorry, the machine is currently being refilled. Please try again later." << std::endl;
    }
    void dispenseDrink(Context* context) override {
        std::cout << "Sorry, the machine is currently being refilled. Please try again later." << std::endl;
    }
};
```



```cpp
// Context class that contains the current state
class Context {
public:
    Context() : currentState(new ReadyState()) {}
    void setState(State* state) {
        delete currentState;
        currentState = state;
    }
    void insertCoin() {
        currentState->insertCoin(this);
    }
    void dispenseDrink() {
        currentState->dispenseDrink(this);
    }
private:
    State* currentState;
};

void ReadyState::insertCoin(Context* context) {
    std::cout << "Coin inserted. Dispensing drink..." << std::endl;
    context->setState(new DispensingState());
}

void DispensingState::dispenseDrink(Context* context) {
    std::cout << "Drink dispensed. Thank you for your purchase." << std::endl;
    context->setState(new ReadyState());
}

int main() {
    Context vendingMachine;

    vendingMachine.insertCoin(); // Outputs "Please insert a coin before dispensing a drink."
    vendingMachine.dispenseDrink(); // Outputs "Please insert a coin before dispensing a drink."

    vendingMachine.setState(new RefillingState());
    vendingMachine.insertCoin(); // Outputs "Sorry, the machine is currently being refilled. Please try again later."
    vendingMachine.dispenseDrink(); // Outputs "Sorry, the machine is currently being refilled. Please try again later."

    vendingMachine.setState(new ReadyState());
    vendingMachine.insertCoin(); // Outputs "Coin inserted. Dispensing drink..."
    vendingMachine.dispenseDrink(); // Outputs "Drink dispensed. Thank you for your purchase."

    return 0;
}
```

Avantages

1 Flexibility

2 Separation of Concerns

3 Encapsulation

4 Code Reusability

Disavantages

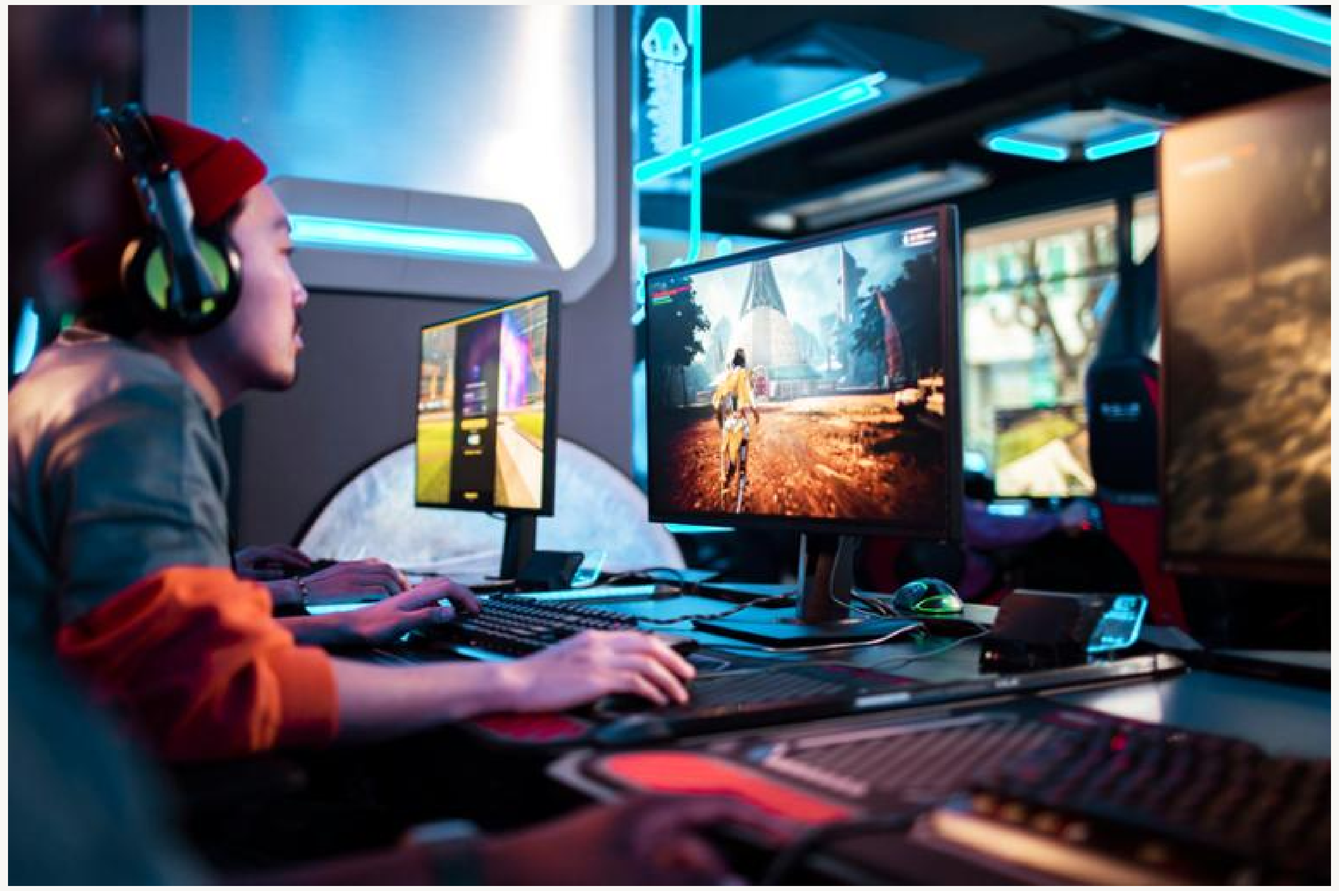
1 Overhead

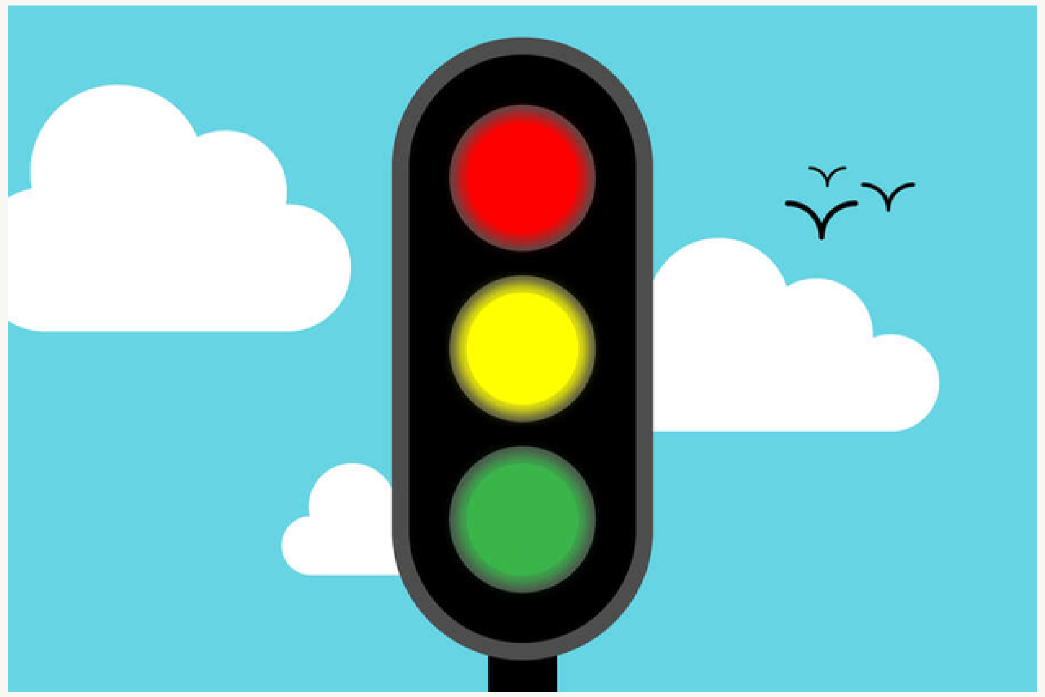
3 Tight Coupling

2 Complexity

4 Size of Codebase

REAL-LIFE EXAMPLE





THANK YOU!