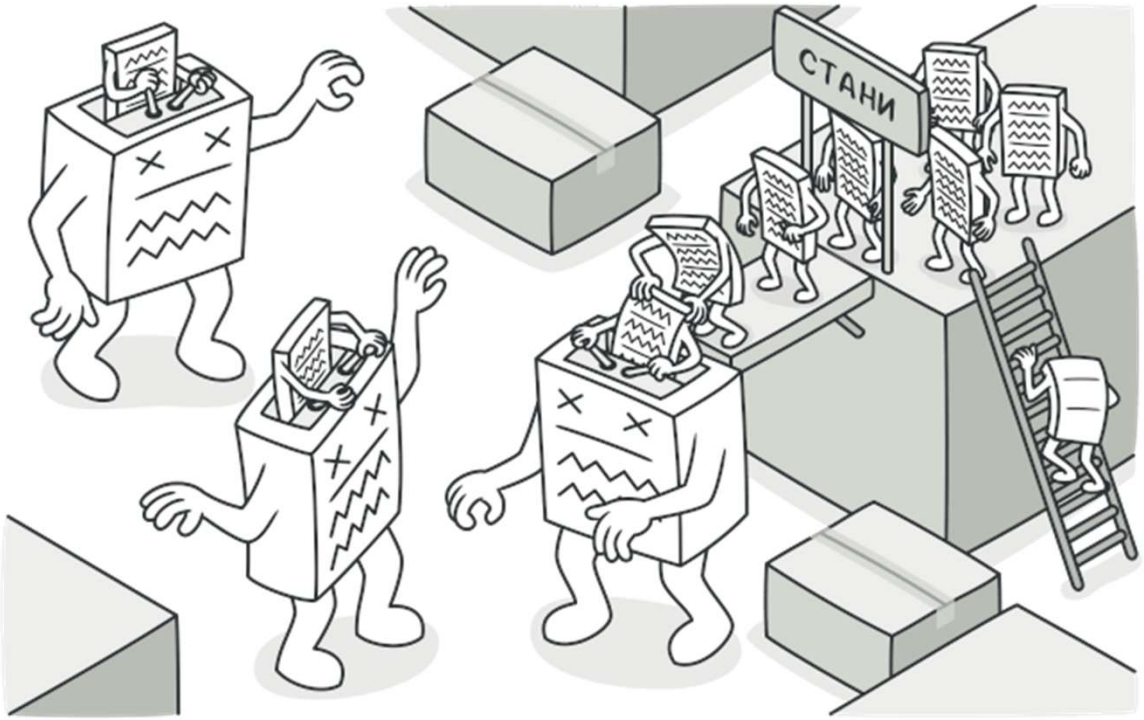
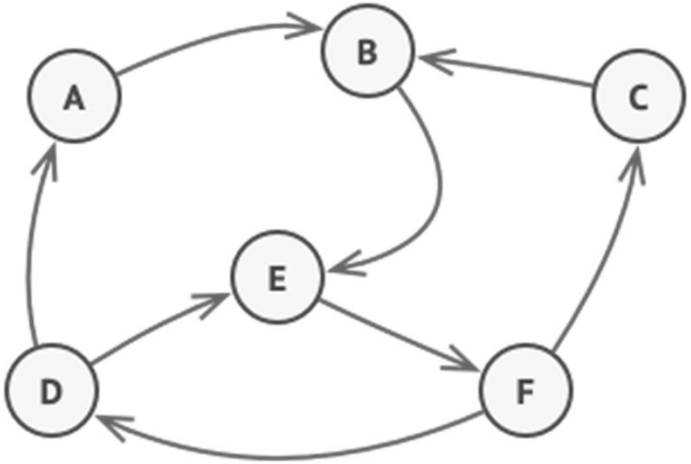
Стан

або ж State

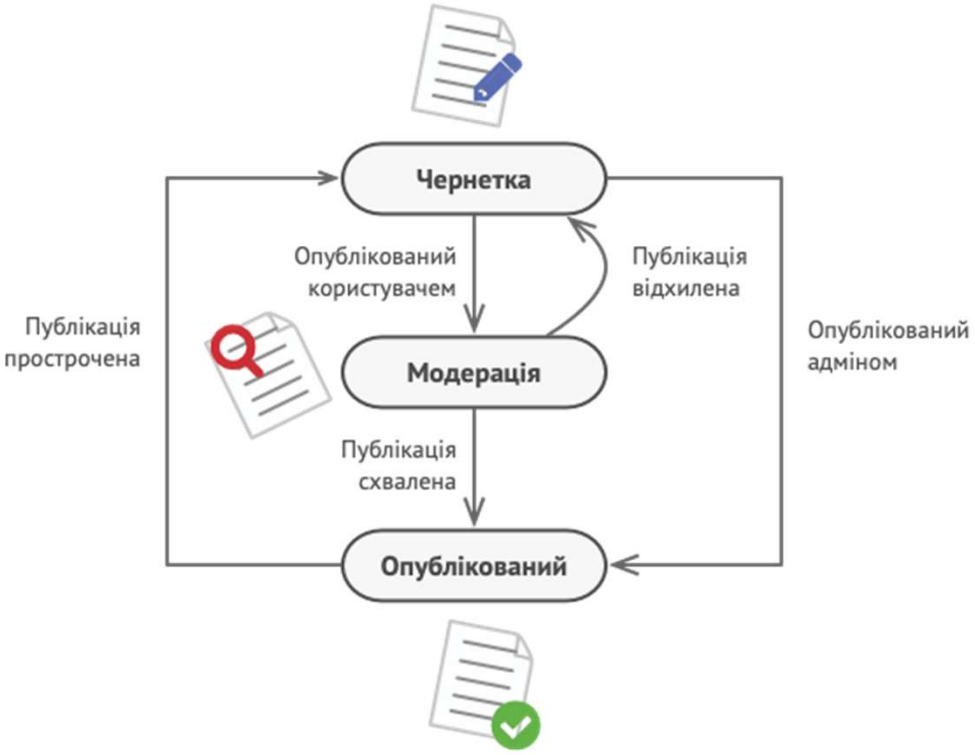


https://refactoring.guru/images/patterns/content/state/state-uk.png

Концепція



Скінченний автомат



Можливі стани документу та переходи між ними

Аналогія з реального світу

Ваш смартфон поводиться по-різному в залежності від поточного стану:

- Якщо телефон **розблоковано**, натискання кнопок телефону призведе до якихось дій.
- Якщо телефон **заблоковано**, натискання кнопок призведе до появи екрану розблокування.
- Якщо телефон **розряджено**, натискання кнопок призведе до появи екрану зарядки.

Чому не switch-case?

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

- Складно підтримувати
- Порушує принцип open/closed
- Не інкапсулює зв’язки між станами

Рішення

Контекст

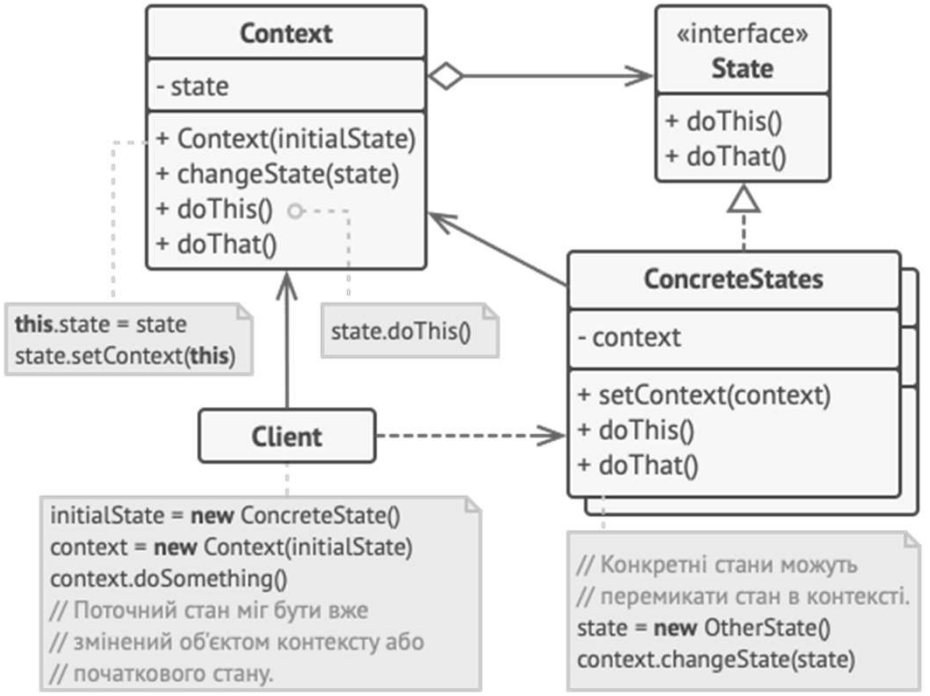
- Визначає інтерфейс, цікавий клієнтам
- Агрегує об’єкт стану, якому делегує увесь «мінливий» функціонал
- Має метод для зміни стану, який зазвичай викликатимуть об’єкти-стани

Стан

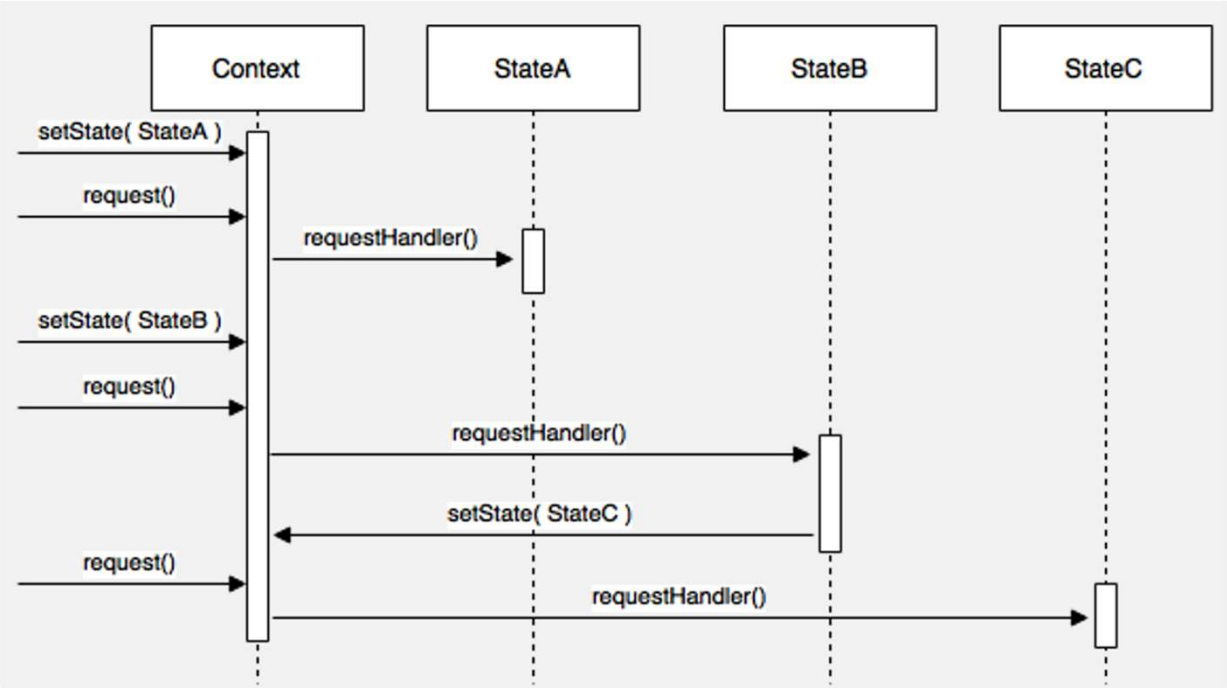
- Визначає інтерфейс, який інкапсулює поведінку, пов’язану з конкретним станом

Конкретні стани

- Реалізують методи інтерфейсу стану
- Визначають зв’язки між станами

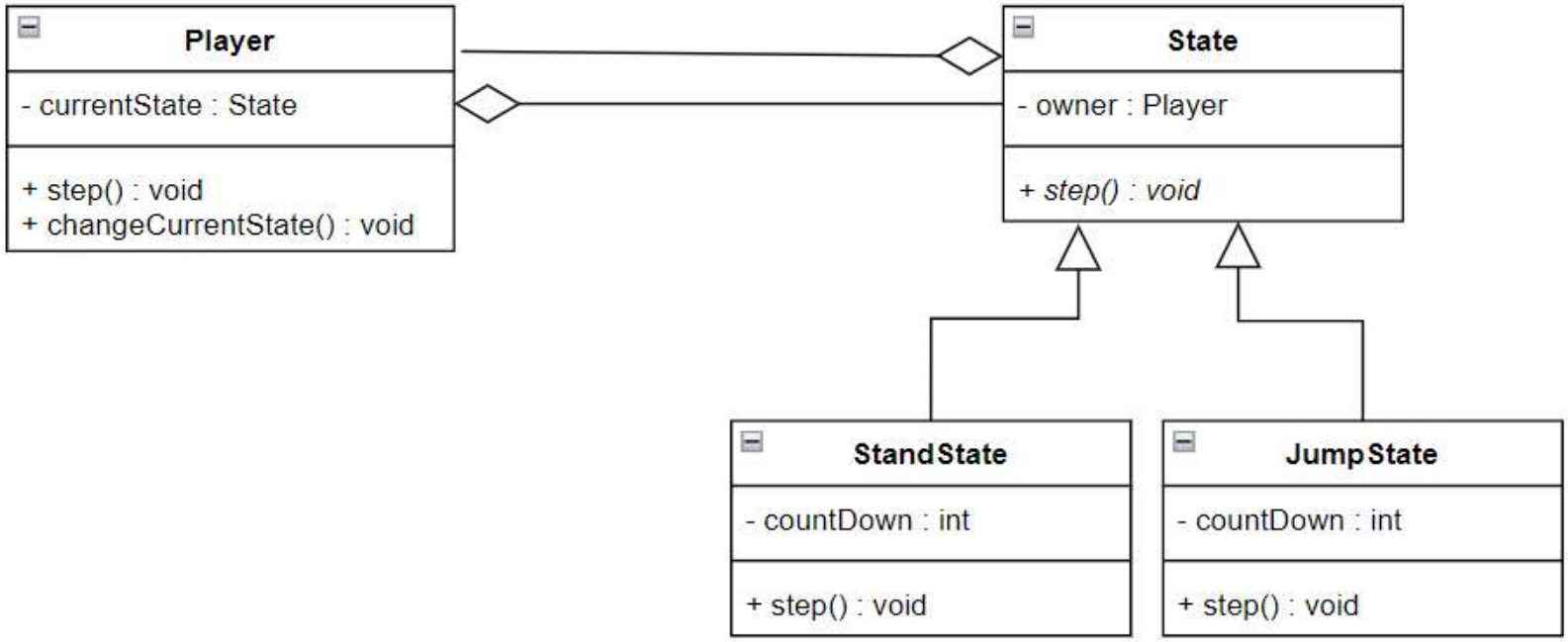


Діаграма послідовності

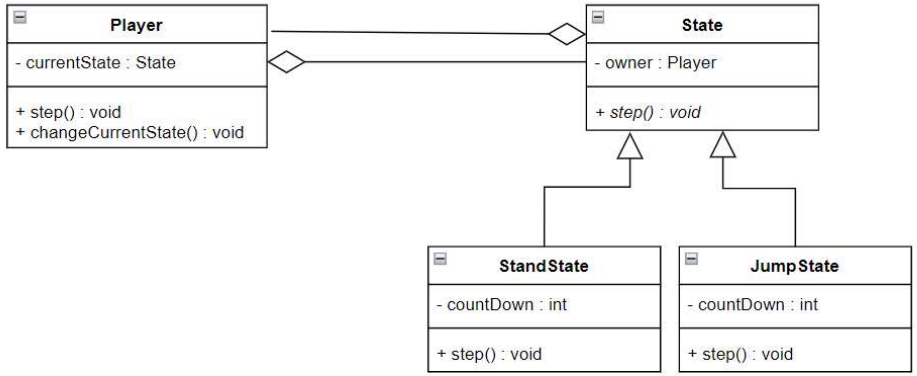


Приклад

Проект: state-player-example



Приклад




```cpp
#include <memory>

#include "State.h"

class Player {
public:
    Player();
    void step();

    template<typename T>
    void changeCurrentState();
private:
    std::unique_ptr<State> m_currentState;
};

template<typename T>
void Player::changeCurrentState() {
    m_currentState = std::make_unique<T>(*this);
}
```

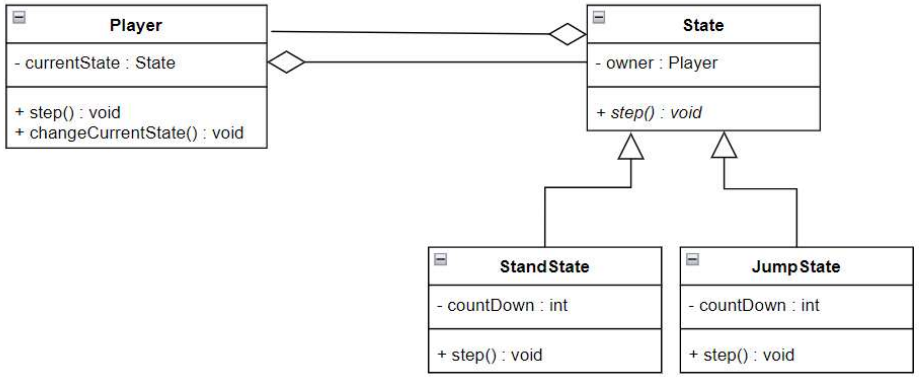
Зберігаємо поточний стан

```cpp
Player::Player() :
    m_currentState(std::make_unique<StandState>(& _Args: *this)) {

}

void Player::step() {
    m_currentState->step();
}
```

Можемо задати початковий стан

Делегуємо виконання

Приклад




```cpp
class State {
public:
    State(Player& owner);
    virtual ~State() {}
    virtual void step() = 0;
protected:
    Player& m_owner;
};
```

Змінюємо стан контексту в залежності від нашої логіки

```cpp
void JumpState::step() {
    m_countDown--;
    std::cout << "JUMP State... countdown is " << m_countDown << "\n";
    if (m_countDown <= 0) {
        m_owner.changeCurrentState<StandState>();
        return;
    }

    // Be careful! We are deleted here
}
```

```cpp
void StandState::step() {
    m_countDown--;
    std::cout << "STAND State... countdown is " << m_countDown << "\n";
    if (m_countDown <= 0) {
        m_owner.changeCurrentState<JumpState>();
        return;
    }

    // Be careful! We are deleted here
}
```

Проблеми (нюанси) реалізації

- Хто визначає переходи між станами?
- Табличний підхід
- Час життя станів

Хто визначає переходи між станами?

- Якщо критерії для зміни стану фіксовані, це можна робити у контексті
- Варіант, коли стани самі визначають перехід до наступного, є більш гнучким, але додає залежності між станами
- У такому разі контекст має надавати інтерфейс для зміни стану

Таблиця переходів

Проект: state-table-based

```
map<State, vector<pair<Trigger, State>>>    transition_table;

transition_table[State::OffHook] = {
    { Trigger::CallDialed, State::Connecting },
    { Trigger::StopUsingPhone, State::OnHook }
};

transition_table[State::Connecting] = {
    { Trigger::HungUp, State::OffHook },
    { Trigger::CallConnected, State::Connected }
};

transition_table[State::Connected] = {
    { Trigger::LeftMessage, State::OffHook },
    { Trigger::HungUp, State::OffHook },
    { Trigger::PlacedOnHold, State::OnHold }
};

transition_table[State::OnHold] = {
    { Trigger::TakenOffHold, State::Connected },
    { Trigger::HungUp, State::OffHook }
};
```

Плюси

- Гнучко: можна змінювати дані замість коду

Мінуси

- Знаходження ключа у таблиці зазвичай дорожче, ніж виклик віртуальної функції
- Зберігання у такому однорідному форматі роблять переходи менш явними і тому складнішими для розуміння
- Основна різниця: взірець State **моделює поведінку, притаманну конкретним станам**, табличний підхід концентрується на **визначенні самих переходів між станами**

Час життя станів

А. Створюємо об’єкти коли потрібно і видаляємо після використання

- Стани невідомі заздалегідь
- Нечасті зміни станів
- Не створюємо непотрібних станів, особливо коли вони тримають багато інформації

Б. Створюємо їх завчасно і на весь час роботи програми

- Стани часто видаляються
- Може бути незручно, оскільки треба вести облік усіх станів
- Використовуємо **Flyweight** для реалізації, якщо стани інкапсулюють поведінку без даних

Застосування Стану

Використовуйте цей взірець, коли:

Поведінка об'єкта залежить від його стану

- Можна виділити інтерфейс, який залежить від стану, в окремий
- Можемо змінювати свою поведінку під час виконання залежно від цього стану

Операції мають великі складні умовні оператори, які залежать від стану об’єкта.

- Стан зазвичай представлено однією або декількома пронумерованими константами (enum).
- Часто декілька операцій міститимуть ту саму умовну структуру
- Шаблон State поміщає кожну гілку умовного виразу в окремий клас
- Це дає змогу розглядати стан контексту як окремий об’єкт, який може мати свій незалежний стан

Переваги/Недоліки Стану

Позбавляє від безлічі великих умовних операторів скінченного автомату

Концентрує в одному місці код, пов’язаний з певним станом.

Спрощує код контексту.

Може невиправдано ускладнити код, якщо станів мало, і вони рідко змінюються

Зв’язок з іншими взірцями

Strategy

- Обидва патерни спираються на композицію
- Різні імплементації стратегій не знають одна про одну
- Стани можуть знати один про одного, і можуть змінювати стан контексту
- Стратегії не можуть вплинути на стратегію виконання контексту
- Як реалізувати стратегію замість стану: проект state-graphical-editor

Flyweight

- Стани не мають стану ☺
- Дуже часті переходи між станами, можемо кешувати ці об’єкти

Singleton

- Стани можуть стати Singleton, якщо ми можемо перевикористати ці об’єкти

Дякую за увагу!