

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**  
**ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА**  
**ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИЙ ТЕХНОЛОГІЙ**  
**КАФЕДРА МАТЕМАТИЧНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ**

**Лабораторна робота №1**  
**з дисципліни: «Інженерія програмного забезпечення»**  
**за темою: «Патерн проектування Strategy»**

**Виконав:**  
студент 3 курсу  
Владислав Краковський

**Перевірив:**  
викладач  
Пенко В.Г.

## **Зміст**

1 ЗАВДАННЯ РОБОТИ.....	3
2 ХІД РОБОТИ.....	4
Висновок.....	11

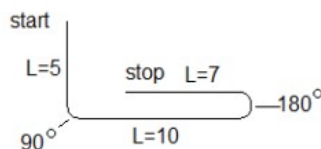
## 1 ЗАВДАННЯ РОБОТИ

**Мета роботи:** Ознайомитись з патерном проектування "Стратегія" (Strategy), навчитися реалізовувати його на мові програмування C++, зрозуміти переваги й недоліки використання даного патерну при проектуванні програмного забезпечення, а також навчитися застосовувати його для розв'язання типових задач.

### Умова роботи:

#### Приклад - Автомобільні гонки.

Траса складається з прямолінійних ділянок із заданою довжиною та поворотів ( $0 < \text{кут повороту} < 180^\circ$ ). Приклад траси:



Початкова швидкість - 0. Потрібно визначити час проходження всієї траси.

Щоб пройти трасу найшвидше, автомобіль повинен швидко проходити прямолінійні ділянки та повороти. Але це різні здібності, що рідко реалізуються в одному автомобілі. На даний момент існує 2 різновиди автомобілів:

- «відчайдушний» - на повороті втрачає швидкість за формулою  $v_{\text{нова}} = v_{\text{стара}} - 0.5 * \text{кут\_повороту}$  (у радіанах), прискорення на прямолінійних ділянках - 0.2 м/с.
- "обережний" - на повороті втрачає швидкість за формулою  $v_{\text{нова}} = v_{\text{стара}} - 0.3 * \text{кут\_повороту}$  (у радіанах), прискорення на прямолінійних ділянках - 0.1 м/с.

Програма повинна бути розроблена так, щоб не знадобилося великих зусиль для додавання нових типів автомобілів надалі.

Рисунок 1.1 - Умова

#### Варіанти самостійних завдань

1. Модифікувати попередню програму, щоб визначала час проходження траси автомобілем.

Рисунок 1.2 - Варіант задачі

## Система контролю версій Git та рефакторинг

### Варіанти самостійних завдань

У наведеному вище викладі кожного патерну наводяться структурний та реальний приклад. Реалізувати структурний приклад та зафіксувати його в системі контролю версій. Далі здійснити серію фіксацій, яка трансформує структурний приклад у реальний. Деякі з проміжних фіксацій повинні полягати у застосуванні того чи іншого прийому рефакторингу.

Рисунок 1.3 - Завдання з GitHub

## 2 ХІД РОБОТИ

Опис патерну Strategy:

**Патерн Strategy (Стратегія)** — це поведінковий патерн проєктування, який дозволяє визначати різні алгоритми (способи поведінки) та інкапсулювати їх у вигляді окремих класів, після чого робити ці алгоритми взаємозамінними в залежності від ситуації.

**Як реалізується патерн Strategy?**

1. Створюється загальний інтерфейс (абстрактний клас) для всіх алгоритмів (стратегій).
2. Для кожної конкретної стратегії створюється окремий клас, який реалізує цей інтерфейс.
3. Контекст (клас, який використовує стратегію) зберігає посилання на об'єкт-стратегію та делегує йому виконання алгоритму.
4. Використовувач (клієнт) може підміняти стратегію на будь-яку іншу реалізацію на льоту (під час виконання програми).

На C++ це виглядає як створення абстрактного базового класу і декількох класів-нащадків, що реалізують різні алгоритми, а також класу-контексту, який взаємодіє з цими стратегіями через інтерфейс.

## Для чого використовується патерн Strategy?

- Коли треба мати кілька різних варіантів алгоритмів для виконання певної задачі.
- Коли потрібно міняти поведінку об'єкта "на льоту" без зміни коду самого об'єкта.
- Для уникнення дублювання коду та виділення змінної поведінки у самостійні класи.

## Типові приклади:

- Вибір способу сортування.
- Вибір алгоритму шифрування.
- Вибір способу оплати (банківська карта, PayPal, криптовалюта).

## Переваги патерну Strategy

- **Гнучкість** — легко додавати нові алгоритми без зміни коду клієнта та контексту.
- **Відокремлення алгоритму від контексту** — принцип єдиного обов'язку, код контексту не залежить від деталей реалізації алгоритму.
- **Можливість підміни алгоритму під час виконання програми.**

## Недоліки патерну Strategy

- **Ускладнення структури коду** — для кожної стратегії створюється окремий клас.
- **Клієнт має знати про різні стратегії** — і сам вибирати, яку саме застосовувати.
- **Можливий зріст кількості класів** при великій кількості алгоритмів.

Запишемо розв'язок поставленої задачі з використання патерну Strategy:

```
#include <iostream>
#include <cmath>
#include <memory>
#include <vector>
using namespace std;

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

// === Інтерфейси ===
class BrakeStrategy {
public:
    virtual double Brake(double velocity, double angle_deg) = 0;
    virtual ~BrakeStrategy() = default;
};

class DriveStrategy {
public:
    virtual pair<double, double> Accelerate(double velocity,
double length) = 0;
    virtual ~DriveStrategy() = default;
};

// === Стратегії гальмування ===
class AggressiveBrake : public BrakeStrategy {
public:
    double Brake(double velocity, double angle_deg) override {
        double angle_rad = angle_deg * M_PI / 180.0;
        return max(0.0, velocity - 0.5 * angle_rad);
    }
}
```

```
};
```

```
class CarefulBrake : public BrakeStrategy {
public:
    double Brake(double velocity, double angle_deg) override {
        double angle_rad = angle_deg * M_PI / 180.0;
        return max(0.0, velocity - 0.3 * angle_rad);
    }
};
```

```
// === Стратегії прискорення ===
```

```
class AggressiveDrive : public DriveStrategy {
public:
    pair<double, double> Accelerate(double velocity, double
length) override {
        double a = 0.2;
        double t = (-2 * velocity + sqrt(4 * velocity * velocity +
2 * a * length)) / (2 * a);
        double newVelocity = velocity + a * t;
        return { newVelocity, t };
    }
};
```

```
class CarefulDrive : public DriveStrategy {
public:
    pair<double, double> Accelerate(double velocity, double
length) override {
        double a = 0.1;
        double t = (-2 * velocity + sqrt(4 * velocity * velocity +
2 * a * length)) / (2 * a);
        double newVelocity = velocity + a * t;
        return { newVelocity, t };
    }
};
```

```
// === Клас автомобіля ===
class Car {
private:
    double velocity;
    double totalTime;
    unique_ptr<BrakeStrategy> brake;
    unique_ptr<DriveStrategy> drive;

public:
    Car(BrakeStrategy* b, DriveStrategy* d)
        : velocity(0.0), totalTime(0.0), brake(b), drive(d) {}

    double GetVelocity() const {
        return velocity;
    }

    double GetTotalTime() const {
        return totalTime;
    }

    void BrakeAtTurn(double angle_deg) {
        velocity = brake->Brake(velocity, angle_deg);
        totalTime += 1.0; // допущення: гальмування займає 1 сек
    }

    void AccelerateOnStraight(double length) {
        auto [newVel, timeSpent] = drive->Accelerate(velocity,
length);
        velocity = newVel;
        totalTime += timeSpent;
    }
}
```



```

void SetBrakeStrategy(BrakeStrategy* b) {
    brake.reset(b);
}

void SetDriveStrategy(DriveStrategy* d) {
    drive.reset(d);
}

};

// === Тестування ===
int main() {
    Car car1(new AggressiveBrake(), new CarefulDrive());
    Car car2(new CarefulBrake(), new AggressiveDrive());

    // Траса: пряма — поворот — пряма — поворот — пряма
    car1.AccelerateOnStraight(5);
    car1.BrakeAtTurn(90);
    car1.AccelerateOnStraight(10);
    car1.BrakeAtTurn(180);
    car1.AccelerateOnStraight(7);

    car2.AccelerateOnStraight(5);
    car2.BrakeAtTurn(90);
    car2.AccelerateOnStraight(10);
    car2.BrakeAtTurn(180);
    car2.AccelerateOnStraight(7);

    cout << "Car 1 final velocity: " << car1.GetVelocity() << "
m/s\n";
    cout << "Car 1 total time: " << car1.GetTotalTime() << " s\n\
n";

```

```

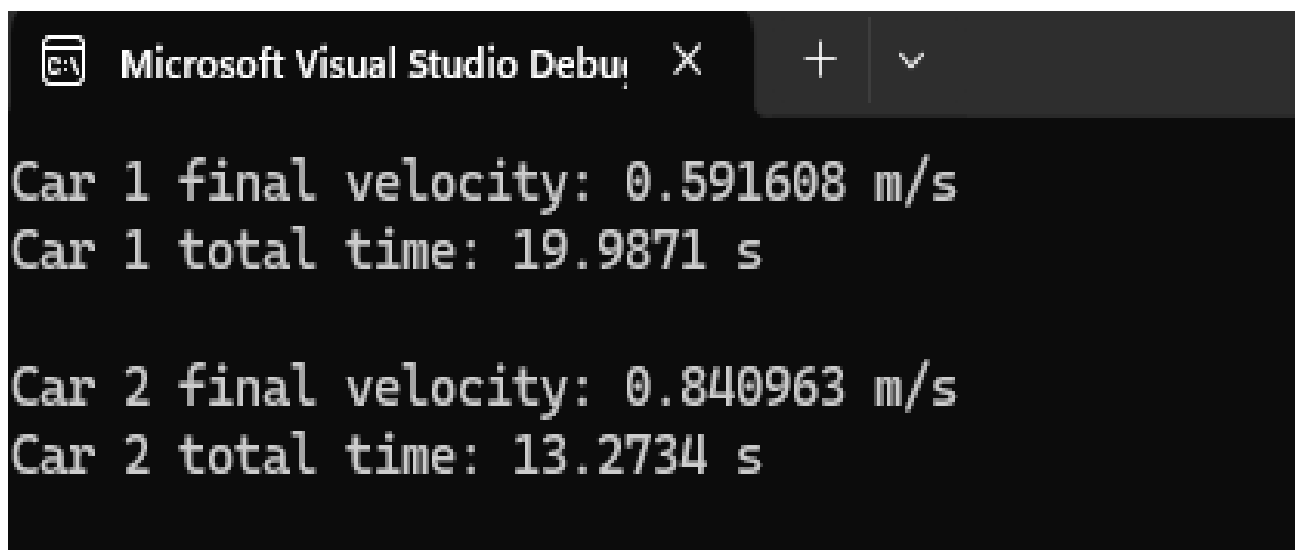
    cout << "Car 2 final velocity: " << car2.GetVelocity() << "
m/s\n";

    cout << "Car 2 total time: " << car2.GetTotalTime() << " s\n";

    return 0;
}

```

Подивимось скріншот з результатом на ілюстрації 2.1:



```

Microsoft Visual Studio Debug Console
+ v

Car 1 final velocity: 0.591608 m/s
Car 1 total time: 19.9871 s

Car 2 final velocity: 0.840963 m/s
Car 2 total time: 13.2734 s

```

Рисунок 2.1 — Результат роботи програми

Програма була завантажена на GitHub згідно з завданням, ілюстрація 2.2:

Add time calculation for car race track (task 1)	cf79b9d	<>
oouoz442 committed 2 weeks ago		
Add real-world Strategy example: car acceleration and braking	eebc62b	<>
oouoz442 committed 2 weeks ago		
Refactor Strategy pattern to real-world sorting strategies	f58aeb4	<>
oouoz442 committed 2 weeks ago		
Add structural implementation of Strategy pattern	03de262	<>
oouoz442 committed 2 weeks ago		

Рисунок 2.2 - GitHub

## **Висновок**

Патерн "Стратегія" — один із найважливіших та найуживаніших поведінкових патернів проєктування. Його застосування дозволяє підвищити гнучкість і розширюваність програмних систем за рахунок інкапсуляції змінних алгоритмів у окремі класи. Разом з тим, Strategy вимагає дисципліни в структурі проєкту і може збільшувати кількість класів, що потрібно враховувати при проєктуванні.