ЗАСТОСУВАННЯ МУЛЬТИМЕТОДІВ У ПРИКЛАДНОМУ ПРОГРАМУВАННІ

Робота виконана студентом 3 курсу
Компанійцем Олександром Олександровичем

МЕТА ДОСЛІДЖЕННЯ

1. Зробити критичний огляд відомих варіантів реалізації.

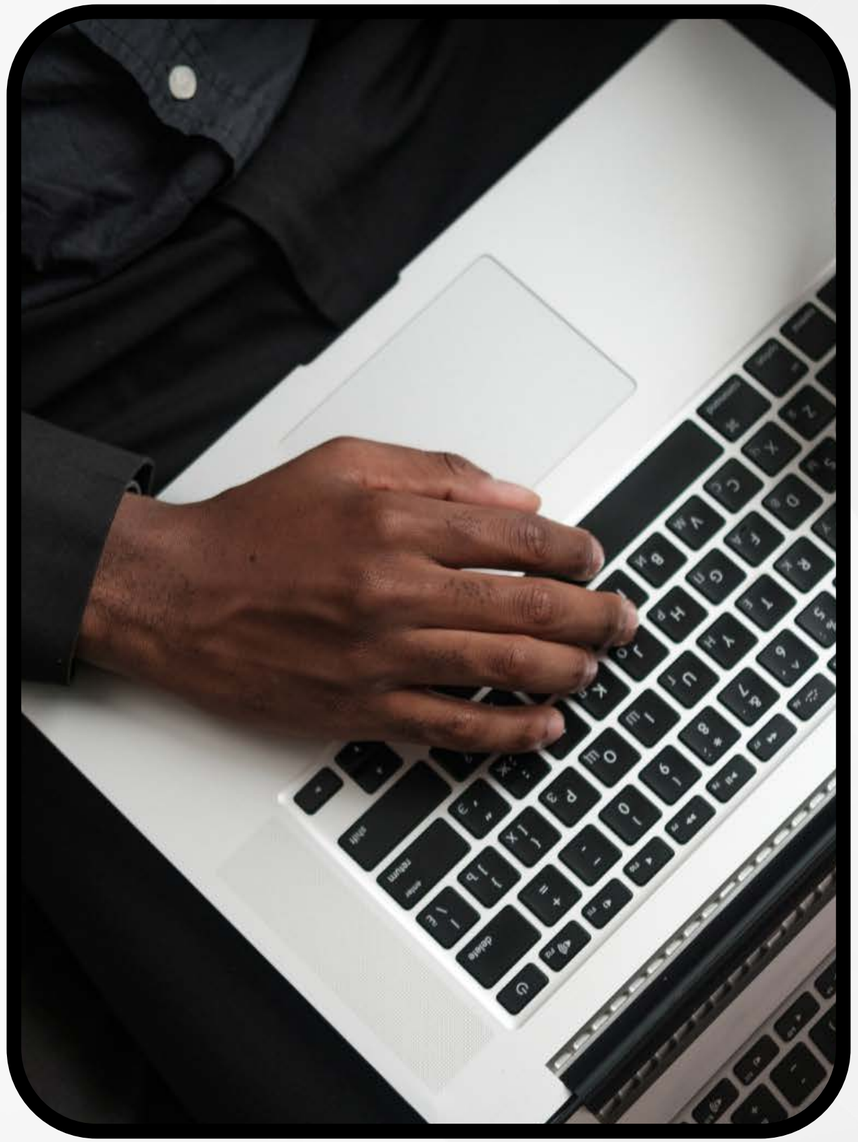
2. Виконати аналіз існуючих підходів для втілення мультиметодів.

3. Визначити особливості, плюси та недоліки підходів.

4. Оптимізувати існуючі, які використовувалися до появи C++17

5. Використати нововведення та розробити новий варіант.

6. Зробити висновки



BRUTE FORCE

Реалізація мультиметодів вимагає написання об'ємного та повторюваного коду.

Така кількість коду ускладнює його читання та внесення змін, збільшуючи складність підтримки.

```
void DoubleDispatch(Shape& lhs, Shape& rhs) {
    if (Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs)) {
        if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs)) {
            DoHatchArea1(&:*p1, &:*p2);
        }
        else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs)) {
            DoHatchArea2(&:*p1, &:*p2);
        }
        else if (Poly* p2 = dynamic_cast<Poly*>(&rhs)) {
            DoHatchArea3(&:*p1, &:*p2);
        }
        else {
            Error(message:"Undefined Intersection");
        }
    }
```

ДИНАМІЧНА ДИСПЕТЧЕРИЗАІЯ

У більшості мов програмування динамічна диспетчеризація реалізується через механізм віртуальних функцій. Клас визначає віртуальну функцію, а похідні класи можуть перевизначити цю функцію, створюючи свої власні реалізації.

```
class GameObject {
public:
    virtual void collide(GameObject&
    virtual void collide(SpaceShip&
    virtual void collide(SpaceStatio
    virtual void collide(Asteroid& c
};
```

```
class SpaceShip : public GameObject {
public:
    void collide(GameObject& otherObject) override {
        otherObject.collide(& otherObject: *this);
    }
    void collide(SpaceShip& otherObject) override {
        std::cout << "SpaceShip collided with SpaceShip" <<
    }
    void collide(SpaceStation& otherObject) override {
        std::cout << "SpaceShip collided with SpaceStation"
    }
    void collide(Asteroid& otherObject) override {
        std::cout << "SpaceShip collided with Asteroid" <<
    }
};
```

ПАТЕРН “VISITOR”

Було розглянуто використання патерну "Visitor" та виявлено основну проблему: проблему компіляції через невідповідність типів, коли "Visitor" очікує конкретний тип, а отримує "Shape". Також зроблено спробу оптимізації методом brute force.

```
// Абстракція для представлення кола
class Circle : public Shape {
public:
    //void accept(const CollisionVisitor& visitor, const Shape
    //    visitor.visit(*this, other);
    //}

    ~Circle() override = default;
};
```

```
// Реалізація методів accept
void Circle::accept(const CollisionVisitor& visitor, const Shape& other
    const Circle* circle = dynamic_cast<const Circle*>(&other);
    const Rectangle* rectangle = dynamic_cast<const Rectangle*>(&other);

    if (circle) {
        visitor.visit(circle: *this, circle2: *circle);
    }
    else if (rectangle) {
        visitor.visit(circle: *this, *rectangle);
    }
    else {
        visitor.visit(circle: *this);
    }
```

НЕВІРТУАЛЬНА ДИСПЕТЧЕРИЗАЦІЯ

Була розглянута реалізація невіртуальної подвійної диспетчеризації, виявлено, що вона не порушує принципи SOLID і дозволяє додавати нові класи без змін у базовий клас.

```
private:
    Derived& _derived;
public:
    Base(Derived& derived) :_derived(derived)
    virtual ~Base() = 0;
    Base& operator=(const Base& c)
    {
        _derived = c._derived;
        return *this;
    }
    operator Derived& () const
```

```
Rectangle::Rectangle(double side)
    :_side(side),
    Base<Rectangle>(*this) {}
Rectangle::Rectangle(const Circle& t)
    :_side(t.radius()),
    Base<Rectangle>(*this) {}


ostream& operator<<(ostream& os, const Rectangle& z)
{
    return os << "RECTANGLE(" <<  z.side() << ')';
}
void Rectangle::intersect(const Rectangle& z) {
    std::cout << "Intersect:" << std::endl
        << *this << std::endl
        << z << std::endl;
}
```

РЕАЛІЗАЦІЯ ПАТЕРНУ “VISITOR” ЗА СТАНДАРТАМИ C++17

Основною інновацією є інтеграція сучасних функцій мови C++17, зокрема std::variant і std::visit, які дозволяють більш гнучко і безпечно реалізувати мультиметоди. Приклад застосування методу до розв'язку модельної задачі був продемонстрований на розгляді зіткнень фігур

```cpp
struct Shape {
    virtual ~Shape() = default;
};

struct Circle : Shape {
    double x, y, radius;

    Circle(double x, double y, double radius) : x(x), y(y), radius(radius) {}
};

struct Rectangle : Shape {
    double x, y, width, height;

    Rectangle(double x, double y, double width, double height)
        : x(x), y(y), width(width), height(height) {}
};
```

```cpp
struct IntersectVisitor {
    bool operator()(const Circle& circle, const Rectangle& rect) const {

        double deltaX = circle.x - std::max(rect.x, std::min(circle.x, rect.x +
        double deltaY = circle.y - std::max(rect.y, std::min(circle.y, rect.y +
        return (deltaX * deltaX + deltaY * deltaY) < (circle.radius * circle.radi
    }

    bool operator()(const Rectangle& rect, const Circle& circle) const {

        return (*this)(circle, rect);
    }

    bool operator()(const Circle& c1, const Circle& c2) const {

        double dx = c1.x - c2.x;
        double dy = c1.y - c2.y;
        double distance = sqrt(dx * dx + dy * dy);
        return distance < (c1.radius + c2.radius);
    }

    bool operator()(const Rectangle& r1, const Rectangle& r2) const {

        return !(r1.x + r1.width < r2.x || r1.x > r2.x + r2.width ||
            r1.y + r1.height < r2.y || r1.y > r2.y + r2.height);
    }
};
```

НЕДОЛІКИ

Така реалізація має свою незручність під час реального використання. Ми використовуємо std::variant тільки на етапі тестування та ми маємо знати про всі класи, щоб повністю реалізувати нашу систему. Також це обмежує динаміність нашої системи

```
int main() {
    Circle c1(0, 0, 5);
    Rectangle r1(4, -3, 10, 6);

    std::variant<Circle, Rectangle> shape1 = c1;
    std::variant<Circle, Rectangle> shape2 = r1;

    bool intersects = std::visit(IntersectVisitor{}, shape1, shape2);
    std::cout << "Intersection: " << (intersects ? "Yes" : "No") << std::endl;

    return 0;
}
```

ОПТИМІЗАЦІЯ

Була надана оптимізація варінту наданого раніше. Було використано патерн до визначеннь застосований до виразів lambda, це надало можливість створити Shape разом з std::visit Це також дозволяє нам створити код, який не буде залежати від ієрархій. Створення такого “контейнера” Shape для збереження надає нам гнучкість структури.

```
struct Circle {
    float x, y, radius;
};

struct Rectangle {
    float x, y, width, height;
};

using Shape = std::variant<Circle, Rectangle>;

template<class... Ts> struct overloaded : Ts... { usin
template<class... Ts> overloaded(Ts...)->overloaded<Ts
```

```
bool checkIntersection(const Shape& s1, const Shape& s2) {
    return std::visit(overloaded{
        [](const Circle& c1, const Circle& c2) { return intersec
        [](const Rectangle& r1, const Rectangle& r2) { return in
        [](const Circle& c, const Rectangle& r) { return interse
        [](const Rectangle& r, const Circle& c) { return interse
        [](const auto&, const auto&) { return false; }
        }, s1, s2);
}
```

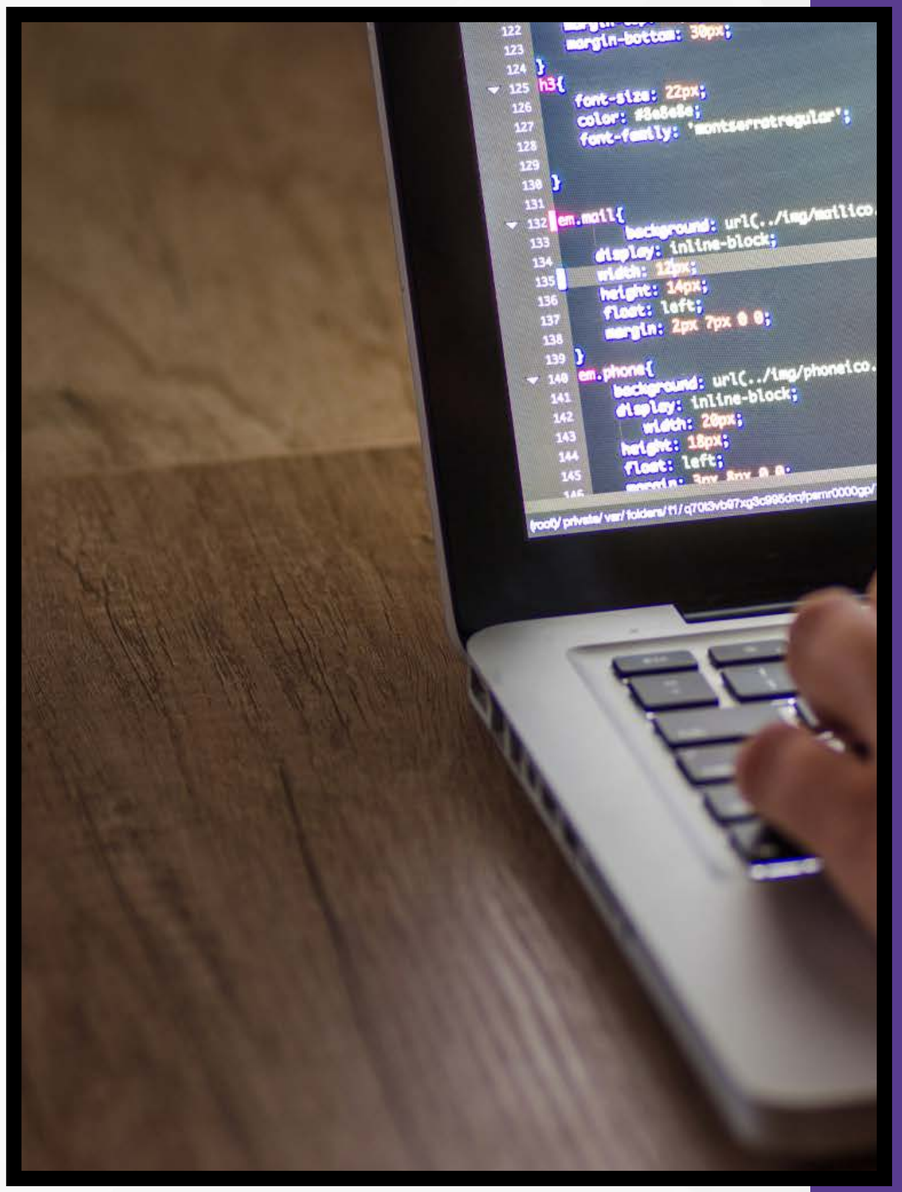
```
int main() {
    Circle c1{ 0, 0, 5 };
    Rectangle r1{ 0, 0, 10, 10 };
    Shape shape1 = c1;
    Shape shape2 = r1;

    bool result = checkIntersection(shape1, shape2);
    std::cout << "Intersection result: " << result << std::e

    return 0;
}
```

НОВИЗНА ТА ПЕРСПЕКТИВИ

Наукова новизна результатів роботи полягає в адаптації і оптимізації використання мультиметодів у контексті сучасних версій мови програмування C++.

Практична значимість результатів роботи проявляється у підвищенні ефективності програмного забезпечення.

Перспективи та рекомендації для подальших досліджень включають розширення експериментальної бази, розробку плагінів та інструментів для автоматизації впровадження мультиметодів.



ДЯКУЮ ЗА УВАГУ