Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**ЗАСТОСУВАННЯ МУЛЬТИМЕТОДІВ У ПРИКЛАДНОМУ ПРОГРАМУВАННІ**»

Виконав: студент 3-го року навчання,

Освітньої програми «Інженерія програмного забезпечення», 121

Компанієць Олександр Олександрович

Керівник Бублик В. В

кандидат наук, доцент

Київ – 2024

Тема: **Застосування мультиметодів у прикладному програмуванні**

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	Листопад 2023 р.	
2.	Огляд літератури та джерел за темою роботи.	Листопад-грудень 2023 р.	
3.	Ознайомлення з мультиметодами в C++	Грудень-січень 2023 р.	
4.	Дослідницька робота	Січень-лютий 2024 р.	
5.	Написання роботи	Лютий-квітень 2024 р.	
6.	Створення слайдів для доповіді.	Квітень 2024 р	
7.	Надання роботи керівнику на перевірку	Квітень 2024 р.	

Студент _Компанієць Олександр Олександрович_____ Керівник Бублик Володимир Васильович____ “16” травня 2024_____

ЗМІСТ

ВСТУП

Застосування мультиметодів у прикладному програмуванні — це досить складна і актуальна тема в сучасному світі технологій, в якому ефективність програмних рішень граю ключову роль при розробці. До появи C++17 було розглянуто багато можливих рішень цієї проблеми, але вони мали свої критичні недоліки, які не були вирішені, наприклад як було зазначено у [1]. Застосування мультиметодів може значно покращити якість і швидкість програм, що вплине на різні сфери бізнесу, науки та інших областей.

Однак, на сьогоднішній день ця тема недостатньо вивчена та в її основі лежить складне теоретичне і практичне підґрунтя, варто зауважити, що використання мультиметодів не завжди доцільне в залежності від наданої задачі. Немає чітких критеріїв, в якому випадку їх краще застосувати. Саме через це виникає «тенденція» використовувати альтеративні, помилкові або неафективні підходи.

Метою даного дослідження полягає в з’ясуванні та втіленні найбільш ефективного варіанту реалізації мультиметодів у прикладному програмуванні. Мета зумовила наступні завдання:

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 або використати нововведення та розробити новий варіант.
5. Зробити висновки щодо можливостей застосування мультиметодів у прикладному програмуванні на основі отриманих даних та аналізу зробленої роботи.

Робота складається з трьох розділів.

Перший розділ присвячено теоретичним аспектам мультиметодів у прикладному програмуванні. Наведено огляд суті подвійної диспетчеризації та

одинарної диспетчеризації. Надано приклади реалізацій та їх аналіз найпростіших моделей для покращеного розуміння проблеми.

У другому розділі проведено практичний аналіз існуючих підходів реалізації мультиметодів, визначені плюси та недоліки кожного. Також наявна спроба оптимізувати відомі варіанти втілення.

Третій розділ присвячено використанню нововведень C++17 для реалізації мультиметодів та моделювання робочих прикладів. Виконано порівняльний аналіз.

Практичне значення проробленої роботи — це можливість використання розроблених варіантів реалізації і рекомендацій у програмному забезпеченні. Це сприятиме підвищенню якості майбутніх продуктів та буде корисним під час оптимізації процесів.

РОЗДІЛ 1: ТЕОРЕТИЧНІ АСПЕКТИ МУЛЬТИМЕТОДІВ У ПРИКЛАДНОМУ ПРОГРАМУВАННІ

1.1 Визначення мультиметодів

Основний тип «поліморфізму» який називається перевантаженням, дозволяє співіснування декількох функцій з однаковою назвою. Компілятор може розрізняти функції під час компіляції, якщо вони мають різні списки параметрів.

Статичні механізми диспетчеризації — це шаблонні функції. Вони забезпечують більш просунутий поліморфізм під час компіляції.

Виклики віртуальних функцій надають середовищу виконання, а не компілятору, право вибирати, яку фактичну реалізацію функції викликати. Реалізації функцій середовища виконання прив’язуються до імен за допомогою віртуальних функцій. Динамічний тип об'єкта, для якого ви здійснюєте віртуальний виклик, визначає, яка функція буде викликана**.**

Головна суть у тому, що мультиметодний механізм має перевести тип аргументів із статичного типу у відповідний динамічний перед тим, як викликати відповідне перевантаження. Це може бути дуже корисно, але неминуче веде до іншого питання про те, чи дійсно функції повинні пов’язані з класами в першу чергу.

Для кращого розуміння диспетчеризації пропонується розглянути одноразову диспетчеризацію, іншими словами single dispatch. Основна суть полягає у тому, що у нас клас та функція, яка залежить від самого типу об’єкта, тому в такій реалізації в C++ використовуються віртуальні функції. Вони в свою чергу зв’язують основний клас та його похідні класи і через це назва цих функцій поєднується з конкретною реалізацією під час виконання програми, який саме

варіант буде використано визначається через динамічний тип об’єкта, до якого вона застосовується

Розглянемо приклад:

```
#include <iostream>
using namespace std;

// Абстракція для представлення форми
class Shape {
public:
    virtual void collision() const = 0;
    virtual ~Shape() = default;
};

// Абстракція для представлення кола
class Circle : public Shape {
public:
    void collision() const override {
        cout << "Collision detected for a circle." << endl;
    }
    ~Circle() override = default;
};

// Абстракція для представлення прямокутника
class Rectangle : public Shape {
public:
    void collision() const override {
        cout << "Collision detected for a rectangle." << endl;
    }
    ~Rectangle() override = default;
};

int main() {
    unique_ptr<Shape> Testc(new Circle);
    unique_ptr<Shape> Testr(new Rectangle);

    Testc->collision();
    Testr->collision();
    return 0;
}
```

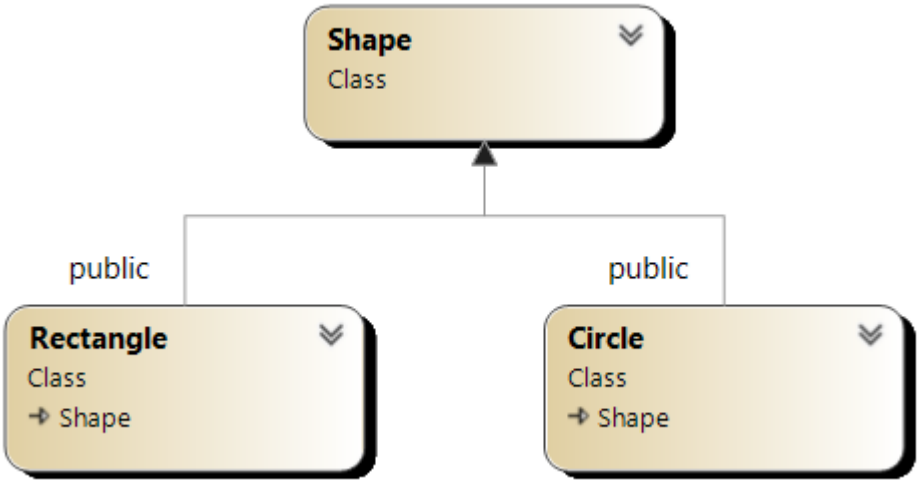

Рисунок 1

В цьому випадку Testc та Testr — це указники на Shape, але коли ми викликаємо метод зіткнення (collision), то ми отримуємо різний результат. Ми досягаємо цього через динамічний поліморфізм, а він у свою чергу вже реалізований за допомогою вище зазначених віртуальних функцій

1.2 Місце використання мультметодів

Розглянемо такий приклад: у вас є операція, яка використовує посилання або вказівники на базові класи для маніпулювання кількома поліморфними об’єктами. Ви хочете, щоб поведінка цієї операції змінювалася залежно від того, скільки з цих об’єктів є динамічними за своєю природою. Одним із поширених класів проблем, які найкраще вирішуються за допомогою багатьох підходів, є зіткнення з об’єктами. Наприклад, ви можете створити відеогру, у якій персонажі створюються з абстрактного класу під назвою Character. Залежно від того, які два типи стикаються воїн і монстр, воїн і скриня зі скарбами або воїн і магічний портал, ви хотіли б, щоб їхня зустріч реагувала по-різному.

1.3 Підхід «brute force»

Розглянемо найбільш прямолінійний підхід використання подвійної диспетчеризації. Для цього створимо метод, який буде приймати два параметри Shape. Для реалізації цього підходу ми можемо створити алгоритм перетину двох фігур однакових чи різних, але для створення мультиметода використаємо підхід перебору для визначення правильного алгоритму розрахунку перетину. Розглянемо варіант запропонований у [1]:

```
#include <iostream>
class Shape {
```

```
public:
    virtual ~Shape() {}
};

class Rectangle : public Shape {};
class Ellipse : public Shape {};
class Poly : public Shape {};

void Error(const std::string& message) {
    std::cerr << message << std::endl;
}

// Сигнатури функцій для різних алгоритмів перетину
void DoHatchArea1(Rectangle&, Rectangle&) {
    std::cout << "Intersecting Rectangle with Rectangle" << std::endl;
}

void DoHatchArea2(Rectangle&, Ellipse&) {
    std::cout << "Intersecting Rectangle with Ellipse" << std::endl;
}

void DoHatchArea3(Rectangle&, Poly&) {
    std::cout << "Intersecting Rectangle with Poly" << std::endl;
}

void DoHatchArea4(Ellipse&, Poly&) {
    std::cout << "Intersecting Ellipse with Poly" << std::endl;
}

void DoHatchArea5(Ellipse&, Ellipse&) {
    std::cout << "Intersecting Ellipse with Ellipse" << std::endl;
}

void DoHatchArea6(Poly&, Poly&) {
    std::cout << "Intersecting Poly with Poly" << std::endl;
}

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("Undefined Intersection");
        }
    }
    else if (Ellipse* p1 = dynamic_cast<Ellipse*>(&lhs)) {
        if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs)) {
            DoHatchArea2(*p2, *p1);
        }
        else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs)) {
            DoHatchArea5(*p1, *p2);
        }
        else if (Poly* p2 = dynamic_cast<Poly*>(&rhs)) {
            DoHatchArea4(*p1, *p2);
        }
        else {
            Error("Undefined Intersection");
        }
```

```
}
else if (Poly* p1 = dynamic_cast<Poly*>(&lhs)) {
    if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs)) {
        DoHatchArea3(*p2, *p1);
    }
    else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs)) {
        DoHatchArea4(*p2, *p1);
    }
    else if (Poly* p2 = dynamic_cast<Poly*>(&rhs)) {
        DoHatchArea6(*p1, *p2);
    }
    else {
        Error("Undefined Intersection");
    }
}
else {
    Error("Undefined Intersection");
}
   }
```

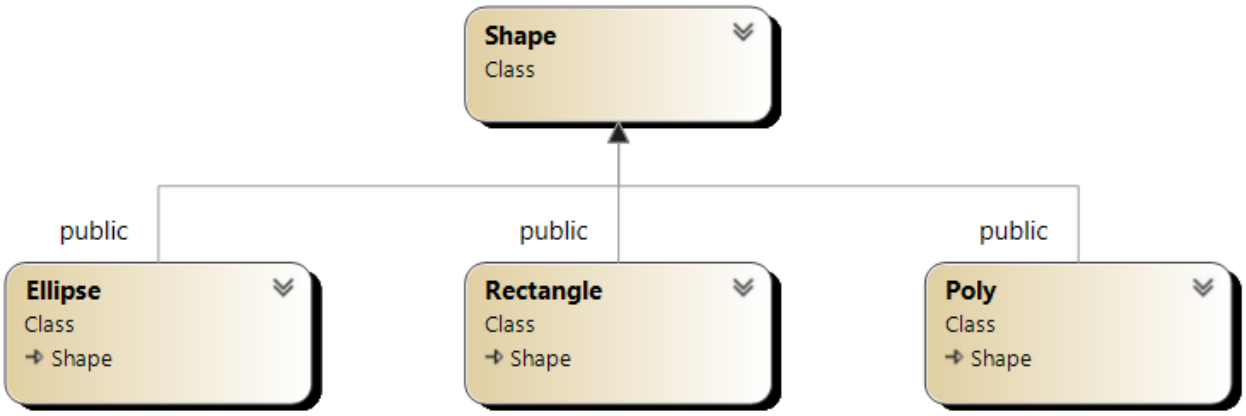

Рисунок 2

Як ми можемо побачити під час такої реалізації мультиметодів нам треба писати багато тексту, який повторюється, також треба зазначити, що проблемою є те, що таку велику кількість коду важко читати та додавати зміни до коду. Іншою важливою проблемою є те, що DoubleDispatch потрібно знати про існуючі класи. Можна додати ще те, що з плюсів такого підходу є швидкість для невеликої кількості існуючих класів, бо наша реалізація використовує пошук лінійної складності.

1.4 Динамічна диспетчеризація та мультиметоди

Типовим прикладом використання багатьох технік є операція перетину. Intersect() визначає чи перетинаються дві форми в ієрархії форм. Управління

кожною можливою композицією форм, включаючи ті, які користувачі бібліотеки надають пізніше, може бути дуже складним. Ще гірше, для використання найточнішого та ефективнішого алгоритму, програміст має мати додаткові знання про існуючі пари форм. Використовуючи синтаксис мультиметодів, з віртуальною індикацією диспетчеризації часу виконання [2].

Треба зазначити, що ціна виклику подвійної диспетчеризації для таких форм як прямокутник або лінія, може перевищувати саме використання алгоритму перетину [2].

Якщо прототип функції має два або більше параметри, які доповнені ключовим слово virtual, то це буде мультиметодною функцією. Реалізація мультиметодів має параметри зі static. Оголошення віртуальних функцій мають відбуватися перед визначеннями самих реалізацій мультиметодів [2].

Наведемо класичний приклад реалізації overlap(), яка посилається на базовий клас shape. Перевірка того чи перетинаються дві форми потрібно створювати інший код для кожної пари фігур[2].

```
struct shape            {...};
struct square: shape {...};
struct triangle: shape {...};
bool overlap (virtual shape& a, virtual shape& b);
bool overlap (static square& a, static triangle& b){...};
bool overlap (static triangle& a, static square& b) {...};
bool overlap (static shape& a, static square& b){...};
bool overlap (static square& a, static shape& b){...};[2]
```

Перекриття (virtual shape& a, virtual shape& b) замінюється функцією диспетчеризації з перекриттям прототипу (shape& a, shape& b). Всередині ця функція диспетчеризації використовує C++ RTTI для вибору однієї з доступних функцій overlap() на основі динамічних типів її параметрів. Якщо ж для якоїсь комбінації фігур не існує відповідної реалізації, то в такому випадку згенерована функція диспетчеризації такої функції призведе до винятку[2].

РОЗДІЛ 2: АНАЛІЗ ТА ОПТИМІЗАЦІЯ ІСНУЮЧИХ ПІДХОДІВ ДО МУЛЬТИМЕТОДІВ: ПРАКТИЧНЕ ВИВЧЕННЯ ПЛЮСІВ ТА МІНУСІВ РІЗНИХ РЕАЛІЗАЦІЙ

2.1 Реалізація мультиметодів тільки через віртуальні функції

Головний підхід такої реалізації закладається в тому, що використовується перенавантаження для кожного методу і воно також у самому класі буде позначатися віртуальним методом і після цього вже буде додано реалізацію кожного з цих методів. Головна мета такого підходу дозволяє уникнути в потребі створення класу виключення, якщо буде ситуація коли на вхід потрапить об’єкт, обробка якого не підтримується. Розглянемо приклад з GameObject:[3]

```
#include <iostream>

class SpaceShip;
class SpaceStation;
class Asteroid;

class GameObject {
public:
    virtual void collide(GameObject& otherObject) = 0;
    virtual void collide(SpaceShip& otherObject) = 0;
    virtual void collide(SpaceStation& otherObject) = 0;
    virtual void collide(Asteroid& otherObject) = 0;
};

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

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

```
    }
    void collide(SpaceStation& otherObject) override {
        std::cout << "SpaceStation collided with another SpaceStation" << std::endl;
    }
    void collide(Asteroid& otherObject) override {
        std::cout << "SpaceStation collided with Asteroid" << std::endl;
    }
};

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

int main() {
    SpaceShip ship;
    SpaceStation station;
    Asteroid asteroid;

    std::cout << "Testing collisions between different GameObjects:" << std::endl;
    std::cout << "1. SpaceShip with SpaceStation:" << std::endl;
    ship.collide(station);

    std::cout << "2. SpaceShip with Asteroid:" << std::endl;
    ship.collide(asteroid);

    std::cout << "3. SpaceStation with SpaceShip:" << std::endl;
    station.collide(ship);

    std::cout << "4. SpaceStation with Asteroid:" << std::endl;
    station.collide(asteroid);

    std::cout << "5. Asteroid with SpaceShip:" << std::endl;
    asteroid.collide(ship);

    std::cout << "6. Asteroid with SpaceStation:" << std::endl;
    asteroid.collide(station);

    std::cout << "7. Asteroid with Asteroid:" << std::endl;
    asteroid.collide(asteroid);

    std::cout << "8. SpaceShip with SpaceShip:" << std::endl;
    ship.collide(ship);

    std::cout << "9. SpaceStation with SpaceStation:" << std::endl;
    station.collide(station);

    return 0;
      }
```

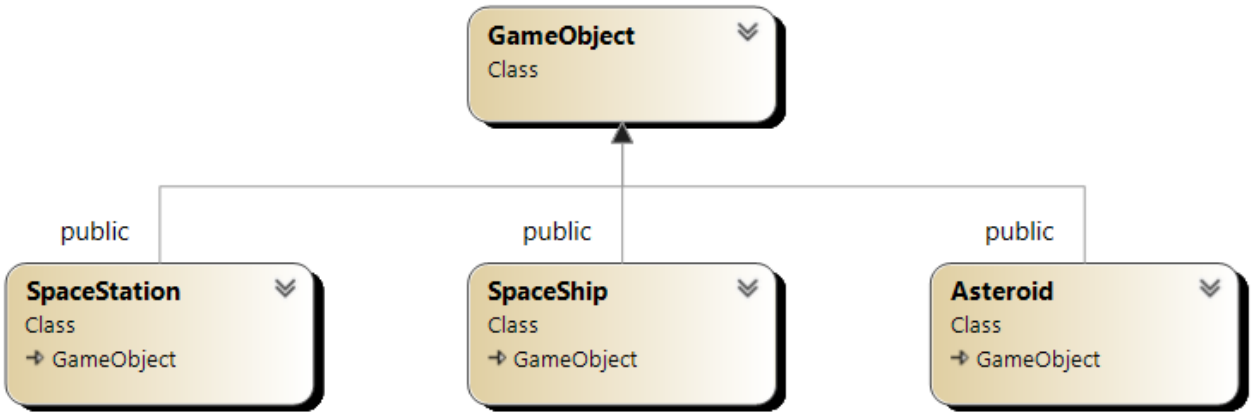

Рисунок 3

Основна ідея полягає у тому, щоб ми реалізували подвійну диспетчеризацію через дві одинарні диспетчеризації, тобто перший віртуальний виклик відповідає за визначення динамічного типу першого об’єкта, а другий робить це аналогічно для другого об’єкта. Через використання такого підходу не може виникнути такої ситуації, в якій сталося би непорозуміння — це і є суть використання тільки віртуальних функцій. Ця реалізація могла би бути ідеальною, якщо б не її фатальний недолік — це те, що кожен клас повинен знати про своїх родичів, тобто SpaceShip, SpaceStation, Asteroid знають про існування один одного. Ще одним мінусом можна виокремити те, що код потребує постійного оновлення, особливо після додавання нових класів, це може бути проблемою, якщо у вас немає доступу до вихідного коду. Можна також ще додати, що реалізація у такий спосіб порушує один з основних принципів SOLID, які є рекомендаціями щодо проектування гнучких і підтримуваних програмних систем, а саме принцип інверсії залежностей. Головна суть цього принципу полягає у тому, що модулі верхнього рівня не повинні бути пов’язаними з нижніми модулями, але у нашому прикладі GameObject потребує додаткову інформацію про його похідні класи: SpaceShip, SpaceStation, Asteroid.

2.2 Використання патерну програмування «Відвідувач»

Під час огляду декількох різних реалізацій мультиметодів можна виокремити те, що в кожному з них існують свої недоліки, тому ми спробуємо вирішити їх використавши патерн програмування «Відвідувач». Я обрав цей підхід, бо він дозволить нам частково обійти головну проблему при реалізації просто через віртуальні функції. Для того, щоб спробувати реалізувати мультиметоди використовуючи цей патерн програмування на знадобляться п’ять основних деталей у програмі: «Відвідувач», «Конкретний відвідувач», «Елемент» та «Конкретний елемент» [4].

«Відвідувач» відповідає за визначення операції «відвідати» для кожного «Конкретного елементу». Тут будуть визначені сигнатури, які допоможуть визначити підходящий варіант методу та отримати клас нашого елементу.

«Конкретний відвідувач» імплементує операції «відвідати», які були визначені «Відвідувачем»

«Елемент» визначає операцію «підтвердити», яка буде отримувати «Конкретного відвідувача» як аргумент

«Конкретний елемент» аналогічно імплементує операцію «підтвердити», яка вже свою чергу буде викликати реалізовану операцію «відвідати» [4].

Розглянемо приклад зіткнення однієї фігури з будь-якою іншою:

```
#include <iostream>

class Circle;
class Rectangle;

// Відвідуач
class CollisionVisitor {
public:
    virtual void visit(const Circle& circle) const = 0;
    virtual void visit(const Rectangle& rectangle) const = 0;
    virtual ~CollisionVisitor() = default;
};

// Елемент
class Shape {
public:
    virtual void accept(const CollisionVisitor& visitor) const = 0;
```

```
        virtual ~Shape() = default;
    };

    // Конкретний елемент
    class Circle : public Shape {
    public:
        void accept(const CollisionVisitor& visitor) const override {
            visitor.visit(*this);
        }

        ~Circle() override = default;
    };

    // Конкретний елемент
    class Rectangle : public Shape {
    public:
        void accept(const CollisionVisitor& visitor) const override {
            visitor.visit(*this);
        }

        ~Rectangle() override = default;
    };

    // Конкретний відвідувач
    class ConsoleCollisionVisitor : public CollisionVisitor {
    public:
        void visit(const Circle& circle) const override {
            std::cout << "Collision between a circle and another shape." <<
std::endl;
        }

        void visit(const Rectangle& rectangle) const override {
            std::cout << "Collision between a rectangle and another shape." <<
std::endl;
        }
    };

    int main() {
        Circle circle;
        Rectangle rectangle;

        ConsoleCollisionVisitor collisionVisitor;

        circle.accept(collisionVisitor);
        rectangle.accept(collisionVisitor);

        return 0;
    }
```

При такій реалізації нас не виникає проблем і код чудово компілюється. Можна також додати, що такий підхід дає нам можливість без проблем додавати нові операції та розділяти зв'язані по контексту методи з незв'язаними. Тепер розглянемо спробу реалізації мультиметодів з використанням патерну «Відвідувач»:

```
    #include <iostream>

    class Circle;
    class Rectangle;
```

```cpp
// Абстрактний клас для відвідувача
class CollisionVisitor {
public:
    virtual void visit
                (const Circle& circle, const Rectangle& rectangle) const = 0;
    virtual void visit
                (const Rectangle& rectangle, const Circle& circle) const = 0;
    virtual ~CollisionVisitor() = default;
};

// Абстракція для представлення форми
class Shape {
public:
    virtual void accept
           (const CollisionVisitor& visitor, const Shape& other) const = 0;
    virtual ~Shape() = default;
};

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

    ~Circle() override = default;
};

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

    ~Rectangle() override = default;
};

// Реалізація відвідувача для виводу повідомлень
class ConsoleCollisionVisitor : public CollisionVisitor {
public:
    void visit(const Circle& circle, const Rectangle& rectangle) const override
{
        std::cout << "Collision between a circle and a rectangle." << std::endl;
    }

    void visit(const Rectangle& rectangle, const Circle& circle) const override
{
        std::cout << "Collision between a rectangle and a circle." << std::endl;
    }
};

int main() {
    Circle circle;
    Rectangle rectangle;

    ConsoleCollisionVisitor collisionVisitor;

    circle.accept(collisionVisitor, rectangle);
    rectangle.accept(collisionVisitor, circle);

    return 0;
```

```
}
```

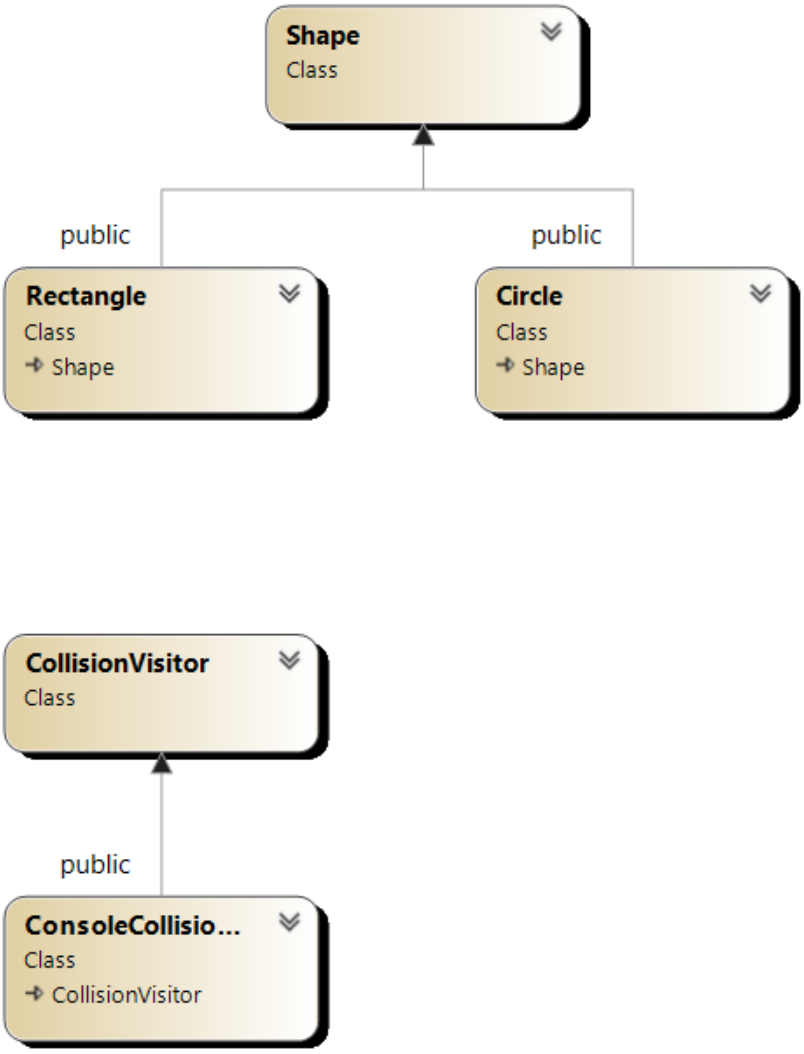

Рисунок 4

Така реалізація була б ідеальною для нас, але в конкретно цьому випадку у нас виникає проблема під час компіляції коду, бо відвідувач очікує уже визначений тип але ми просто передаємо Shape. Відштовхуючись від цієї реалізації можна спробувати оптимізувати її.

2.3 Спроба оптимізації патерну програмування «Відвідувач»

Для вирішення цієї проблеми розглянемо поєднання підходу Brute Force та підходу «Відвідувач», у результаті отримаємо:

```
#include <iostream>
#include <typeinfo>

class CollisionVisitor;


// Абстрактний клас для відвідувача
```

```
class CollisionVisitor {
public:
    virtual void visit
            (const Circle& circle, const Rectangle& rectangle) const = 0;
    virtual void visit(const Circle& circle, const Circle& circle2) const = 0;
    virtual void visit
           (const Rectangle& rectangle, const Circle& circle) const = 0;
    virtual void visit
           (const Rectangle& rectangle, const Rectangle& rectangle2) const = 0;
    virtual void visit(const Circle& circle) const = 0;
    virtual void visit(const Rectangle& rectangle) const = 0;
    virtual ~CollisionVisitor() = default;
};

// Абстракція для представлення форми
class Shape {
public:
    virtual void accept
           (const CollisionVisitor& visitor, const Shape& other) const = 0;
    virtual ~Shape() = default;
};

class Circle;
class Rectangle;

// Абстракція для представлення кола
class Circle : public Shape {
public:
    void accept
         (const CollisionVisitor& visitor, const Shape& other) const override;
    ~Circle() override = default;
};

// Абстракція для представлення прямокутника
class Rectangle : public Shape {
public:
    void accept(const CollisionVisitor& visitor, const Shape& other) const
override;
    ~Rectangle() override = default;
};

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

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

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

    if (circle) {
        visitor.visit(*this, *circle);
    }
```

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

        // Реалізація відвідувача для виводу повідомлень
        class ConsoleCollisionVisitor : public CollisionVisitor {
        public:
            void visit(const Circle& circle, const Rectangle& rectangle) const override
{
                std::cout << "Collision between a circle and a rectangle." << std::endl;
            }

            void visit(const Rectangle& rectangle, const Circle& circle) const override
{
                std::cout << "Collision between a rectangle and a circle." << std::endl;
            }
            void visit(const Circle& circle1, const Circle& circle2) const override {
                std::cout << "Collision between a circle and a circle." << std::endl;
            }
            void visit(const Rectangle& rectangle, const Rectangle& rectangle2) const
override {
                std::cout << "Collision between a rectangle and a rectangle." <<
std::endl;
            }
            void visit(const Rectangle& rectangle) const override {
                std::cout << "Collision between a rectangle and a shape." << std::endl;
            }
            void visit(const Circle& circle) const override {
                std::cout << "Collision between a circle and a shape." << std::endl;
            }
        };

        int main() {
            Circle circle1, circle2;
            Rectangle rectangle1, rectangle2;
            Shape& shape1 = rectangle1;
            Shape& shape2 = circle1;

            ConsoleCollisionVisitor collisionVisitor;

            circle1.accept(collisionVisitor, rectangle1);
            rectangle1.accept(collisionVisitor, circle1);
            circle1.accept(collisionVisitor, circle2);
            rectangle1.accept(collisionVisitor, rectangle2);
            rectangle1.accept(collisionVisitor, shape1);
            circle1.accept(collisionVisitor, shape2);
        }
```

При поєднанні цих двох підходів ми отримуємо працюючий результат з достатньо непоганою реалізацією, але через те, що ми використали дуже прямолінійний варіант «Brute Force» у нас виникає аналогічні проблеми такі як: велика кількість коду, кожному класу потрібно знати про вже інші існуючі класи, тобто Circle має знати про Rectangle, щоб взаємодіяти з ним. Одночасно з цим

можна додати, що при редакції коду та створенню нових класів спостерігаємо потребу в зміні усіх частин коду, що є великою незручністю.

Отже, хоч такий підхід і не порушує правил SOLID, але кількість мінусів перевищує можливості використання такого варіанту на практиці.

РОЗДІЛ 3: ЗАСТОСУВАННЯ НОВОВВЕДЕНЬ C++17 У МУЛЬТИМЕТОДАХ: ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА МОДЕЛЮВАННЯ РОБОЧИХ ПРИКЛАДІВ

3.1 Сучасний варіант використання патерну «Відвідувач»

У C++17 було додано юніон std::variant і функцію std::visit.

“std::variant” може зберігати у собі типи даних і під час виконання (рантайм) змінювати свій активний тип, саме це забезпечує цьому інструменту велику гнучкість. У свій час “std::visit” дозволяє виконати «відвідування» використовуючи відвідувача, при цьому він може бути як функцією так і лямбда виразом або функціональним об’єктом.

Використаємо наданий нам новий функціонал у C++17 і розглянемо приклад реалізації патерну:

```
 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) {}
};


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

    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);
    }
};

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;
      }
```

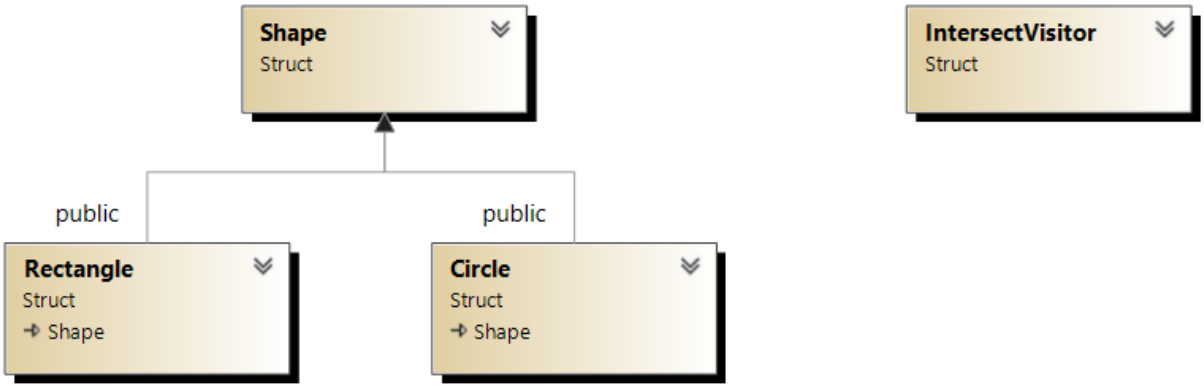

Рисунок 5

Після аналізу коду можна зазначити, що така реалізація не порушує основних правил SOLID, являється гнучким, так як додавання нових функціональних методів або типів не потребує додаткових модифікацій, зміни відбуваються через додавання нових методів обробки до відвідувача. Використання “std::variant” в цьому прикладі обмежує наші типи, так як вони мають бути відомі заздалегідь а також він використовується тільки на етапі тестування, це обмежує динамічність нашої системи. Спробуємо оптимізувати цей підхід:

```
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... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...)->overloaded<Ts...>;
```

Цю частину варто детальніше розглянути. Це оголошення створює новий тип Shape, який грає роль «перехідника» або контейнера для збереження Rectangle та Circle. Воно надає нам можливість легше маніпулювати цими формами. Тут було використано «overloaded lambda», саме вона дозволяє нам використати Shape разом з std::visit, дозволяючи нам створювати функціональні об’єкти, які потім будуть використані як відвідувачі.

```
bool intersects(const Circle& c1, const Circle& c2) {
    float dx = c1.x - c2.x;
    float dy = c1.y - c2.y;
    float distance = sqrt(dx * dx + dy * dy);
    return distance < (c1.radius + c2.radius);
}

bool intersects(const Rectangle& r1, const Rectangle& r2) {
    return !(r1.x + r1.width < r2.x || r2.x + r2.width < r1.x ||
        r1.y + r1.height < r2.y || r2.y + r2.height < r1.y);
}

bool intersects(const Circle& c, const Rectangle& r) {
    float closestX = std::clamp(c.x, r.x, r.x + r.width);
    float closestY = std::clamp(c.y, r.y, r.y + r.height);
    float dx = c.x - closestX;
    float dy = c.y - closestY;
    return (dx * dx + dy * dy) < (c.radius * c.radius);
}

bool intersects(const Rectangle& r, const Circle& c) {
    return intersects(c, r);
}

bool checkIntersection(const Shape& s1, const Shape& s2) {
    return std::visit(overloaded{
        [](const Circle& c1, const Circle& c2) { return intersects(c1, c2); },
        [](const Rectangle& r1, const Rectangle& r2) { return intersects(r1, r2); },
        [](const Circle& c, const Rectangle& r) { return intersects(c, r); },
        [](const Rectangle& r, const Circle& c) { return intersects(r, c); },
        [](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::endl;

    return 0;
}
```

Рисунок 6

Детально розглянувши цей приклад можемо сказати, що такий підхід аналогічно не порушує правил SOLID, гарно масштабується, так як додавання нових типів легко виконується розширенням частин Shape з std::variant, а створення нових взаємодій відбувається додаванням відповідних методів у відвідувач. Також можна зауважити, що код не залежить від ієрархій спадкування, що спрощує архітектуру системи, а також використання лямбда-функцій робить код чистішим і зрозумілішим.

Отже, такий підхід чудово вирішує основні проблеми мультиметодів та ілюструє, як гнучко можна обробляти взаємодії в універсальному контексті. Але варто зауважити, що автоматичний вибір функцій відвідувачем може викликати додаткові витрати під час виконання.

3.2 Застосування невіртуальної диспетчеризації

Дослідження можливостей узагальненого програмування дають змогу переносити деякі динамічні проблеми на статичний рівень. Цей підхід був запропонований [5], як варіант застосування поліморфізму без використання віртуальних функцій. Розглянемо цей шаблон, який відомий як Curiously Recurring Template Pattern, CRTP. Суть цього шаблону полягає спадкуванні на основі класів-шаблонів, клас “Derived” успадковує від базового класу “Base”, але і він інстанційований класом “Derived”. Саме це дає нам можливість

використовувати статичний поліморфізм, а не динамічний. Розберемо приклад (оснований на [6]) класу “Base”

```
template <class Derived>
class Base
{
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
	{
		return _derived;
	}

	void intersect(const Base& z)
	{
		Derived* c1 = static_cast<Derived*>(this);
		Derived c2 = z;
		cout << "Intersect:" << endl << c1 << endl << c2 << endl;

	}

};
template <class Derived>
Base<Derived>::~Base() {}
```

В цьому прикладі було наведено варіант методу «intersect», який може бути застосовний у реальній програмі, варто зауважити його особливість. Цей метод використовує static_cast<Derived*>(this) для того, щоб конвертувати this з Base до Derived, ми можемо зробити це, так як Base завжди успадковується від Derived.

Як приклад застосування цього підходу розглянемо реалізацію перетину фігур.

```
class Circle : public Base<Circle>
{
private:
	double _radius;
public:
	Circle(double radius = 0);
	Circle(const Rectangle& a);
	Circle& operator=(const Circle& t)
	{
		_radius = t.radius();
		return *this;
	}

	void intersect(const Circle& z);
	const double& radius() const
	{
		return _radius;
	}
```

```
};

ostream& operator<<(ostream& os, const Circle& z);

class Rectangle : public Base<Rectangle>
{
private:
	double _side;
public:
	Rectangle(double side = 0);
	Rectangle(const Circle& t);

	const Rectangle& operator=(const Rectangle& a)
	{
		_side = a.side();
		return *this;
	}

	void intersect(const Rectangle& z);


	const double& side() const
	{
		return _side;
	}

};
ostream& operator<<(ostream& os, const Rectangle& z);
```

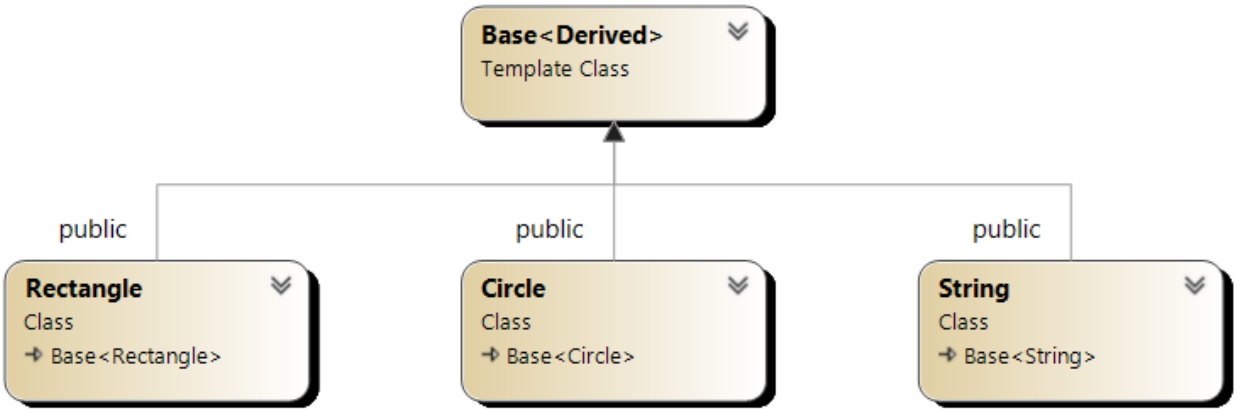

Рисунок 7

Важливо зауважити що ці класи мають конструктори, які використовуються для перетворення Rectangle в Circle і навпаки. Такий зразок не являється обмеженням по темі або сфері використання. Також доповнення цього патерну новими класами не потребує додаткових доповнень до класу Base, а також такий підхід не порушує правила SOLID, що є важливим плюсом.

Варто зазначити, що невіртуальна диспетчеризація має свої недоліки такі як: обмежена гнучкість, складність структур, збільшення архітектури веде до збільшення вразливості до помилок.

Висновки по роботі та рекомендації для подальших досліджень

Проведене дослідження та розробка сучасного варіанту застосування мультиметодів у прикладному програмуванні є значущим внеском у розвиток цієї сфери. У роботі представлені ефективні підходи реалізації подвійної диспетчеризації. Також був проведений глибокий теоретичний аналіз і практичне дослідження існуючих варіантів втілення мультиметодів, зокрема через використання патерну «Відвідувач», динамічної, та невіртуальної(статичної) диспетчеризації. Були з’ясовані найбільш ефективні та вдалі варіанти реалізації, а саме сучасний підхід через патерн «Відвідувач» та невіртуальна диспетчеризація. Результати цього дослідження можуть бути застосовані для покращення швидкості та якості програмних продуктів, що є важливим під час розробки комерційного програмного забезпечення.

Як рекомендації для майбутніх досліджень можна зазначити такі можливі варіанти: розширення експериментальної бази, дослідження case-study з реальними проектами, розробка плагінів і інструментів. Майбутні дослідження мають включати у себе застосування мультиметодів у більш широкому спектрі мов, програм та проектів, що дозволить детальніше розглянути подвійну диспетчеризацію краще зрозуміти обмеження та можливості мультиметодів. Розробка плагінів та інструментів для використання мультиметодів може значно популяризувати цей підхід через автоматизацію процесів інтеграції.

СПИСОК ЛІТЕРАТУРИ

[1] Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied 2001

[2] Cmm (C++ with MultiMethods) user documentation [Electronic resource] Mode access: http://www.op59.net/cmm/cmm-0.28/users.html

[3] Meyers S. More Effective C++ 35 New Ways to Improve Your Programs and Designs / S. Meyers. – Addison-Wesley, 2008

[4] Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlisside. – Addison-Wesley, 1994.

[5] Tauber R. C++ Runtime Polymorphism without Virtual Functions / R. Tauber

[6] До питання створення статичного патерну проєктування для подвійної диспетчеризації модельних сигнатур Бублик В. В. 2021

[7] Pirkelbauer P. Open Multi-Methods for C++ / P. Pirkelbauer, Yu. Solodkyy, B. Stroustrup // Proceedings of the 6th international conference on Generative programming and component engineering. – Salzburg, Austria, 2007

[8] . Smith J. Draft proposal for adding Multimethods to C++ [Electronic resource] / J. Smith. Mode access:

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html

[9] Multi Methods [Electronic resource] Mode access: https://wiki.c2.com/?MultiMethods

[10] Design and evaluation of C++ open multi-methods☆ Peter Pirkelbauer ∗ , Yuriy Solodkyy∗ , Bjarne Stroustrup 2009