УДК 004.432.2
DOI: 10.18523/2617-3808.2021.4.64-71

Бублик В. В.

ДО ПИТАННЯ СТВОРЕННЯ СТАТИЧНОГО ПАТЕРНУ ПРОЄКТУВАННЯ ДЛЯ ПОДВІЙНОЇ ДИСПЕТЧЕРИЗАЦІЇ МОДЕЛЬНИХ СИГНАТУР

У роботі досліджено можливість уникнення застосування поліморфізму при створенні мовою програмування C++ класів-моделей певної сигнатури, придатних для подвійної диспетчеризації функцій–членів цих класів стосовно варіантів їх інтерпретації. Із цією метою розглянуто створення невіртуальних ієрархій шляхом застосування методів узагальненого програмування із використанням так званої зворотної конкретизації шаблону. Це дало змогу запропонувати загальну схему патерну проєктування, що визначив архітектуру класів, придатну для статичної реалізації мультиметодів. Розглянуто приклад застосування патерну для спільної реалізації мультиметодів у різнорідних класах на прикладі комплексних чисел і рядків символів. Одержані в роботі результати знайшли застосування в курсах об'єктно-орієнтованого програмування на факультеті інформатики Києво-Могилянської академії.

Ключові слова: C++, об'єктно-орієнтоване програмування, патерн (зразок) проєктування, подвійна диспетчеризація (мультиметод), віртуальна функція, поліморфізм, узагальнене програмування.

Вступ

Під поліморфізмом зазвичай розуміють динамічне налаштування програми на тип даних об'єкта, з яким програма зустрінеться у процесі її виконання. Тобто за своїм характером поліморфізм є характеристикою чисто динамічною. Водночас у літературі і практиці застосування C++ час від часу вживають термін «статичний поліморфізм». На відміну від динамічного, статичний поліморфізм викликає багато різночитань аж до повного його заперечення. Найпростішим варіантом статичного поліморфізму є техніка довизначення (overloading) функцій. Звісно, за всієї привабливості довизначення як функцій–членів класу, так і статичних функцій–утиліт класів, називати його поліморфізмом, взагалі кажучи, сумнівно. Хоча, керуючись алгоритмом Герба Саттера і Андрея Александреску [12] (рекомендація 44), можна приймати виважені рішення стосовно доцільності вживання того чи того виду кожної конкретної функції. Однак, зважаючи на зауваження Скотта Мейєрса [7] щодо небезпечності функцій–членів класів, часто віддають перевагу утилітам, що може створювати додаткові проблеми у разі спроб застосування поліморфізму.

Узявши широко вживаний у літературі приклад двох зображень комплексних чисел в алгебричній і тригонометричній формах, позначивши відповідні класи через AComplex і TComplex, матимемо утиліти таких сигнатур – додавання для алгебричних комплексних чисел і множення для тригонометричних:

```
const AComplex operator+
(const AComplex&, const AComplex&);
const TComplex operator*
(const TComplex&, const TComplex&);
```

За наявності відповідних операторів зведення типів або конвертувальних конструкторів застосування утиліт дає змогу визначати адитивні операції лише в алгебричній, а мультиплікативні – у тригонометричній формі, користуючись, за необхідності, автоматичним зведенням типів. Важливу роль при цьому відіграє симетричність обох параметрів, що допускає зведення типів як за першим, так і за другим параметром.

Справді, за умови визначень

```
AComplex a(3.5, 2);
TComplex t1(1, pi), t2(1, pi/4);
```

вирази `a + t1; t1+a; t1+t2` обчислюються зведенням типів за допомогою однієї і тієї самої реалізації операції додавання + для класу `AComplex`.

Якщо ж перенести оператор + до членів класу `AComplex`

```
const AComplex AComplex::operator+
                (const AComplex&);
```

© *Бублик В. В.*, 2021

то, через несиметричність першого параметра з другим, зведення за першим параметром даватиме синтаксичну помилку, тож вирази `t1+a i t1+t2` стануть помилковими.

Водночас суміщені з присвоєнням оператори не передбачають симетричності типів параметрів, а тому можуть залишитися функціями–членами відповідних класів

```
AComplex& AComplex::operator+=
         (const AComplex&);
AComplex& AComplex::operator*=
         (const AComplex&);
TComplex& TComplex::operator+=
         (const TComplex&);
TComplex& TComplex::operator*=
         (const TComplex&);
```

за обмеженіших можливостей автоматичного зведення типів. При цьому адитивні операції над числами у тригонометричній формі можуть і надалі застосовувати алгебричні реалізації за явного зведення типів. Оператор додавання, суміщений із присвоєнням, += для тригонометричного запису, наприклад, набуде вигляду:

```
TComplex& TComplex::operator+=
         (const TComplex& a)
{
    return *this=AComplex(*this) +=
          AComplex(a);
}
```

Те саме стосується мультиплікативних операторів в алгебричній формі.

Особливості створення динамічних мультиметодів

Можливість визначення ієрархій для типу комплексних чисел належить до класичних прикладів застосування поліморфізму. Ще у 1988 р. Бертран Маєр [5] сформулював як вправу задачу визначення абстрактного базового класу (інтерфейсу) `Complex` із чисто віртуальними операціями. Справді, якщо визначити інтерфейс класу комплексних чисел як сигнатуру операторів в такий спосіб

```
class Complex
{
public:
    virtual Complex& operator+=
            (const Complex&) = 0;
    virtual Complex& operator-=
            (const Complex&) = 0;
    virtual Complex& operator*=
            (const Complex&) = 0;
    virtual Complex& operator/=
            (const Complex&) = 0;
};
```

то спроба визначення інтерпретацій цієї сигнатури за допомогою ієрархії спадкувань, наприклад, таких

```
// Комплексні числа
// в алгебричній формі
class AComplex: public Complex
{
public:
    virtual Complex& operator+=
            (const Complex& z);
……….
};
// Комплексні числа
// в тригонометричній формі
class TComplex: public Complex
{
public:
    virtual Complex& operator+=
            (const Complex& z);
……….
};
```

виявиться невдалою.

У похідних класах, у наших термінах, `AComplex` і `TComplex`, віртуальні операції базового класу мають набути поліморфної реалізації. Однак залежність віртуальної функції від двох об'єктів – `this` і параметра операції – зумовлює застосування подвійної диспетчеризації, перетворюючи функцію–член класу на мультиметод. Проблему мультиметодів часто обговорювали в комп'ютерній літературі, наприклад у відомій книжці Андрея Александреску [2]. У ній мультиметодам присвячено цілий розділ із висновком про те, що немає вдалої концепції їх реалізації. Попри наявні пропозиції про попередній варіант доповнення C++ мультиметодами [10] і особисту участь Б'єрна Страструпа у дослідженні проблеми [11; 9], доповнення наступних стандартів C++ мультиметодами не передбачається.

Складність створення динамічної реалізації мультиметоду наочно демонструє простий приклад ієрархії комплексних чисел, побудованої за наведеною діаграмою. Для прикладу обмежимося одним оператором += (рис. 1).

Реалізація мультиметоду += передбачає доповнення кожного з класів ієрархії додатковими функціями. Обмежимося скороченим викладенням структури класів. У закритій частині базового класу розміщують дві додаткові функції-диспетчери, які доведеться реалізовувати в похідних класах. Диспетчери залежать від типів похідних класів, чим порушується один з основних п'яти принципів SOLID [4], а саме принцип інверсії залежностей: базові класи не повинні залежати

від похідних. Цей недолік мають також інші витонченіші способи реалізації мультиметодів.

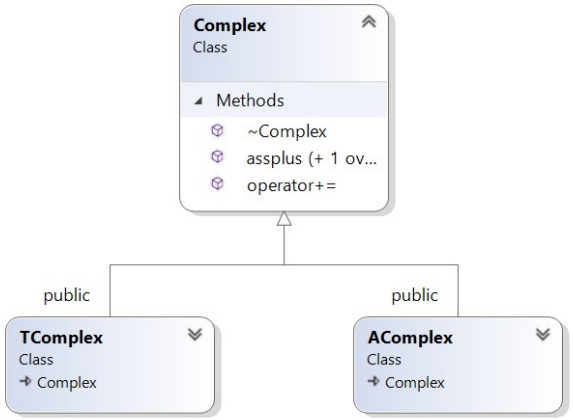

Рис. 1. Віртуальна ієрархія комплексних чисел

```
class Complex
{
public:
    virtual Complex& operator+=
            (const Complex&) = 0;
private:
    //Dispatchers
    virtual AComplex&
       assplus(AComplex&) const = 0;
    virtual TComplex&
       assplus(TComplex&) const = 0;
};
```

Похідні класи мають містити реалізації цих функцій-диспетчерів, одна з яких реалізується як проформа для забезпечення можливості лінкування.

```
class AComplex: public Complex
{
public:
    virtual Complex& operator+=
            (const Complex& z)
    {
          return z.assplus(*this);
    }
private:
    double _re, _im;
    virtual AComplex& assplus
            (AComplex& z) const
    {
          z._re += _re;
          z._im += _im;
          return z;
    }
// Pro forma realization for linker
    virtual TComplex& assplus
            (TComplex& z) const
    {
          TComplex* pc = nullptr;
          return *pc;
    }
};
```

Відповідно у тригонометричній формі матимемо:

```
class TComplex: public Complex
{
public:
    virtual TComplex& operator+=
             (const Complex& z)
    {
          return z.assplus(*this);
    }
private:
    TComplex& TComplex::assplus
              (TComplex& z) const
{
          TComplex u = AComplex(z);
          return u+=AComplex(*this);
}
// Pro forma realization for linker
    virtual AComplex& assplus
             (AComplex&) const
    {
          AComplex* p = nullptr;
          return *p;
    }
};
```

Серйозніші задачі потребуватимуть ще складніших побудов. Але кожного разу за потреби доповнення ієрархії новими похідними класами доведеться вносити зміни до самого базового класу.

Створення невіртуальної диспетчеризації

Водночас дослідження можливостей узагальненого програмування (templates) дають змогу переносити деякі динамічні проблеми на статичний рівень. Зокрема у [13] запропоновано варіант застосування поліморфізму без віртуальних функцій. Цей самий метод дає можливість визначити патерн проєктування (design pattern) мультиметодів для певного виду класів, які ми назвемо сигнатурними моделями. Мова йтиме про визначення патерну, що забезпечить застосування диспетчеризації для класів, які реалізуватимуть інтерфейс, визначений базовим абстрактним класом. Патерни проєктування (на думку автора, українською мовою їх правильніше було б називати зразками) набули значного поширення завдяки знаменитій книжці «Банда чотирьох» [3]. Зразок проєктування «Сигнатурний мультиметод» для прикладу за двома операціями `f` і `h` можна умовно зобразити діаграмою

(рис. 2). Важливо, що одна з операцій реалізує семантику значень, тоді як інша повертає результат відсилкою. У класі `Base`, параметризованому типовим параметром `Derived`, розташований атрибут `_derived` типу `Derived&`, який забезпечуватиме доступ до об'єктів похідних класів. Методи класу `Base` повертають результат типу `Derived` або навіть `Derived&`, а приймають параметр типу `const Base&`, за рахунок чого, власне, відбувається зведення подвійної диспетчеризації до одиночної. Клас `Product` (їх може бути скільки завгодно) визначається як похідний від зворотної конкретизації класу `Base` його похідним класом `Product` як значенням типового параметра.

Далі діаграма демонструє можливе застосування патерну. Статична функція `process()`, узагальнена типовим параметром `T`, приймає аргументи типу `Base<T>&`. Клієнт класу `Product` викликає функцію process() із параметрами власного типу, в нашому випадку `Product`. Спільний ефект спадкування і узагальнень забезпечує диспетчеризацію параметра `Base<Product>&` за типом `Product`.

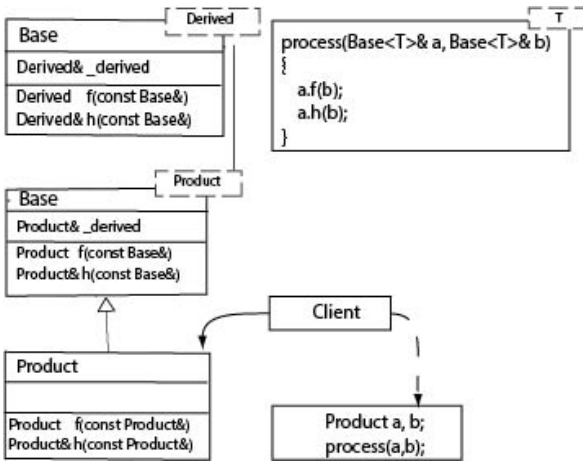

Рис. 2. Сигнатурний патерн подвійної диспетчеризації

Наведемо приклад застосування патерну.

```
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;
    }
    double mod() const
    {
          Derived c = *this;
          return c.mod();
    }
    Derived& operator+=(const Base& z)
    {
          Derived* c1 =
           static_cast<Derived*>(this);
          Derived c2 = z;
          *c1 += c2;
          return *c1;
    }
    Derived operator+(const Base& z)
    {
          Derived c1 = *this;
          Derived c2 = z;
          return c1+c2;
    }
};
```

Для дотримання принципу нетермінальної абстрактності Скотта Мейерса [6] (рекомендація 33) деструктор класу визначено як чисто віртуальний, що унеможливлює осібне використання його об'єктів без відповідних похідних класів.

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

Застосування невіртуальної диспетчеризації

Як приклад розглянемо знову комплексні числа, застосування яких також цікаве з академічної точки зору [5]. Візьмемо комплексні числа в алгебричній формі з операціями у вигляді членів класу. Для визначення класу застосуємо зворотну параметризацію шаблону ним самим.

```
class AComplex: public Base<AComplex>
{
private:
    double _re, _im;
public:
    AComplex(double re = 0, double im = 0)
:_re(re), _im(im), Base(*this) {}
    AComplex& operator=(const AComplex& a)
    {
          _re = a._re;
          _im = a._im;
    }
    double mod() const {
          return sqrt(_re*_re+_im*_im);
```

```
    }
    double arg() const {
          return asin(_im/mod());
    }
    AComplex& operator+=
              (const AComplex& z)
    {
          _re += z._re;
          _im += z._im;
          return *this;
    }
    const AComplex operator+
          (const AComplex& z) const
    {
          AComplex res = *this;
          res._re += z._re;
          res._im += z._im;
          return res;
    }
    double re() const
    {
          return _re;
    }
    double im() const
    {
          return _im;
    }
};
```

Розглянемо комплексні числа у тригонометричній формі. Виконання адитивних операцій відсилаємо до алгебричної форми. Це не обов'язкове рішення, вжите лише для того, щоб продемонструвати збереженість властивостей зведення типів, властиве звичайному визначенню у формі утиліт класу. Цікаво, що можливість зведення тепер поширюватиметься і на несиметричний стосовно типів параметрів варіант визначення операторів із застосуванням відсилок.

```
class TComplex : public Base<TComplex>
{
private:
    double _rho, _phi;
public:
    TComplex(double rho=0,double phi=0)
          :_rho(rho),_
phi(phi),Base(*this){}
    TComplex(const AComplex& a)
          :_rho(a.mod()), _phi(a.arg()),
           Base(*this) {}
    TComplex& operator=
           (const TComplex& t)
    {
          _rho = t._rho;
          _phi = t._phi;
          return *this;
    }
    operator AComplex() const
    {
          return AComplex
          (_rho * cos(_phi),
           _rho * sin(_phi));
    }
    double mod() const {
          return _rho;
    }
    TComplex& operator+=
              (const TComplex& z)
    {
          return (*this)=
         (AComplex(*this) +=
          AComplex(z));
    }
    const TComplex operator+
          (const TComplex& z) const
    {
          return AComplex (*this)+
                 AComplex(z);
    }
};
```

Селектори в класі AComplex необхідні для убезпечення операторів виведення.

```
ostream& operator<<(ostream& os,
         const AComplex& z)
{
    return os << '('<<
        z.re() << ',' << z.im()<<')';
}
ostream& operator<<(ostream& os,
         const TComplex& z)
{
    return os << AComplex(z);
}
```

Комплексними числами не обмежуються можливості запропонованого зразка. Доповнення патерну новими похідними класами не потребує жодних доповнень до базового класу, як це було у випадку динамічних мультиметодів. Як приклад, цікавий щодо прикладних застосувань, розглянемо рядки символів

```
class String : public Base<String>
{
private:
    int _len;
    char* _ch;

public:
    String(const char* ch)
           :_len(strlen(ch)),
            _ch(new char[_len+1]),
            Base(*this)
    {
          strcpy_s(_ch, _len+1, ch);
    }
    String(const String& s)
```

```
            :_len(s._len),
            _ch(new char[_len+1]),
            Base(*this)
    {
            strcpy_s(_ch, _len+1, s._ch);
    }
    ~String()
    {
            delete[]_ch;
            _len = 0;
            _ch = nullptr;
    }
    String& operator+=(const String& s)
    {
            char* newch =
             new char[_len + s._len + 1];
            char* tmp = newch;
            char* tmp1 = _ch;
            while (*tmp++ = *tmp1++);
            tmp1 = s._ch;
            --tmp;
            while (*tmp++ = *tmp1++);
            delete[]_ch;
            _ch = newch;
            _len += s._len;
            return *this;
    }
    const String operator+
           (const String& s) const
    {
            String res(*this);
            return res += s;
    }
};
```

Функція диспетчеризації залежить лише від базового класу, тоді як диспетчери віртуальної реалізації залежали від похідних класів.

```
template<class T>
void process(Base<T>& a, Base<T>& b)
{
    cout << "Process:" <<endl<< a
          << endl<< b << endl;
    cout << "Result +  :";
    cout << a + b << endl;
    cout << "Result += :";
    cout << (a += b) << endl;
}
```

Так виконується налаштування операторів + і += на тип переданих фактичних параметрів, наприклад алгебричних комплексних чисел:

```
AComplex a(1, 1);
AComplex b(1, 2);
process(a, b);
```

або на клас рядків символів

```
String a("Multimethod");
String b(" rules!");
process(a, b);
```

без явного втручання конкретизацій базового класу до клієнтського коду. При цьому уможливлюється приєднання різних варіантів визначення рядків символів, наприклад [8]. Важливим залишається лише дотримання прийнятої сигнатури.

Наведемо діаграму класів, створену студією MVS (рис. 3).

Діаграма на рис. 3 не повністю відтворює ситуацію. Наведено схему спадкування з шаблону, а насправді виникають три пари залежних конкретизацій узагальненого класу `Base`, по одній для кожного похідного класу.

Важливо, що створена реалізація виконує неявне зведення типів. Наприклад, якщо визначити

```
TComplex t1(1, 0), t2(2, pi/2);
AComplex a(1, 1);
```

то всі варіанти змішаних операцій, зокрема діагностовані як синтаксично некоректні за звичайних визначень, стають легітимними. Це стосу-

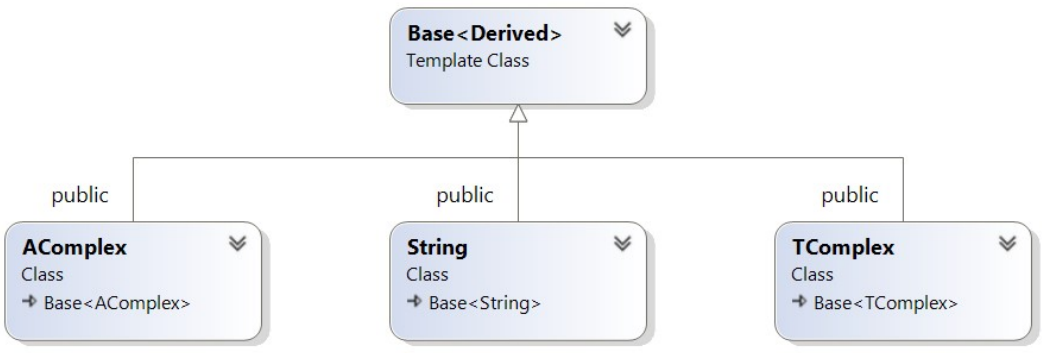

Рис. 3. Діаграма невіртуальної ієрархії

ється не лише виразів, що обчислюються за семантикою значень, як-от: `a+t1; t1+a; t1+t2`, а також суміщених присвоєнь виду `a+=t1; t1+=a; t1+=t2`.

Але за межами можливостей сигнатурних моделей залишилася диспетчеризація змішаних операцій, наприклад за сигнатурою

```
template<class T, class U>
void process(Base<T>& a, Base<U>& b),
```

що потребуватиме подальшого дослідження.

Висновки

У роботі було розглянуто питання подвійної диспетчеризації функцій мовою програмування C++, в якій передбачено лише одинарну диспетчеризацію. Було зауважено, що віртуальні методи моделювання подвійної диспетчеризації мають недолік – порушення одного з принципів SOLID, що негативно впливає на якість створюваного програмного забезпечення.

Запропоновано варіант невіртуальної подвійної диспетчеризації, узагальнений у вигляді створеного патерну проєктування «Сигнатурний мультиметод». Патерн придатний для застосування до різновидів інтерпретацій визначеної у ньому сигнатури. Розглянуто загальну схему застосування новоствореного патерну, проілюстровано на прикладі реалізації класів комплексних чисел і рядків символів. Показано, що немає порушень принципів SOLID, і продемонстровано можливість доповнення ієрархії новими класами реалізації сигнатури без необхідності втручання до структури базового класу.

Повну версію наведених у роботі програмних кодів можна знайти на сайті дистанційного навчання Києво-Могилянської академії [1]. Автор дякує Данилу Фітелю за обговорення і цінні рекомендації.

Список літератури

1. Бублик В. В. До питання електронного навчання програмуванню / В. В. Бублик // Наукові записки НаУКМА. – 2013. – Т. 151. Комп'ютерні науки. – С. 112–115.
2. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied / A. Alexandrescu. – Addison-Wesley, 2001.
3. Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlisside. – Addison-Wesley, 1994.
4. Martin R. The Dependency Inversion Principle [Electronic resource] / R. Martin // C++ Report, 1996. – 2021. – Mode of access: https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf.
5. Meyer B. Object-Oriented Software Construction / B. Meyer. – Prentice Hall, 1988.
6. Meyers S. More Effective C++ 35 New Ways to Improve Your Programs and Designs / S. Meyers. – Addison-Wesley, 2008.
7. Meyers S. How Non-Member Functions Improve Encapsulation [Electronic resource] / S. Meyers // Dr. Dobb's, 2000. – Mode of access: https://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197.
8. Ormrod N. The strange details of std::string at Facebook [Electronic resource] / N. Ormrod. – Mode of access: https://www.youtube.com/watch?v=kPR8h4-qZdk.
9. 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. – Pp. 123–134.
10. Smith J. Draft proposal for adding Multimethods to C++ [Electronic resource] / J. Smith. – Mode of access: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html.
11. Stroustrup B. The Design and Evolution of C++ / B. Stroustrup. – Addison-Wesley, 1994.
12. Sutter H. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, / H. Sutter, A. Alexandrescu. Addison-Wesley, 2005.
13. Tauber R. C++ Runtime Polymorphism without Virtual Functions [Electronic resource] / R. Tauber. – Mode of access: https://www.codeproject.com/Articles/603818/Cplusplus-Runtime-Polymorphism-without-Virtual-Fun.

References

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley.

Bublyk, V. V. (2013). Do pytannia elektronnoho navchannia prohramuvanniu. *Naukovi zapysky NaUKMA, 151. Kompiuterni nauky*, 112–115 [in Ukrainian].

Gamma, E., Helm, R., Johnson, R., & Vlisside, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Martin, R. (1996). *The Dependency Inversion Principle*. Retrieved from https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.

Meyers, S. (2008). *More Effective C++ 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley.

Meyers, S. (2000). *How Non-Member Functions Improve Encapsulation.* Retrieved from https://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197.

Ormrod, N. (2020). *The strange details of std::string at Facebook*. Retrieved from https://www.youtube.com/watch?v=kPR8h4-qZdk.

Pirkelbauer, P., Solodkyy, Yu., & Stroustrup, B. (2007). Open Multi-Methods for C++. In *Proceedings of the 6th international conference on Generative programming and component engineering* (pp. 123–134). https://doi.org/10.1016/j.scico.2009.06.002

Smith, J. (2003). *Draft proposal for adding Multimethods to C++*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Sutter, H., & Alexandrescu, A. (2005). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley.

Tauber, R. (2014). *C++ Runtime Polymorphism without Virtual Functions*. Retrieved from https://www.codeproject.com/Articles/603818/Cplusplus-Runtime-Polymorphism-without-Virtual-Fun.

V. Boublik

TOWARDS CREATING A STATIC DESIGN PATTERN FOR DOUBLE DISPATCHING MODEL SIGNATURES

The paper investigates a possibility of developing a non-virtual hierarchy for a special case of class signature, which may possess different interpretations. The approach is similar to double dispatching in the C ++ programming language. As an alternative to polymorphism, a non-polymorphic hierarchy has been suggested based on generic programming templates. This hierarchy is based on inverse parametrization for templates enabling constructing a general scheme for the design pattern. The pattern defined a class architecture suitable for static implementation of double dispatched multimethod for a special case of signature-defined interfaces.

In fact, any abstract base class (interface) with purely virtual operations must acquire a polymorphic implementation. Besides, the polymorphism itself, the dependence of a virtual function on two objects – "this" and another parameter – requires the use of double dispatch, turning a class member function into a double dispatched multimethod.

A preliminary consideration deals with issues of double dispatching in the C++ programming language. Inheritance with polymorphic class member functions is used. This requires special efforts of adding to both bases and derived classes a couple of virtual functions to support dispatching. In any case, this approach, besides using virtual functions, has a disadvantage of violating one of the SOLID principles, namely the principle of dependency inversion: base classes should not depend on derivatives, which negatively affects the quality of the software.

Polymorphism is usually understood as the dynamic tuning of a program to the data type of the object that the program will encounter during its execution. That is, by its nature, polymorphism is a purely dynamic characteristic. However, in C++ literature and in practice, you can come across the term "static polymorphism".

At the same time, research of possibilities of generalized programming (templates) allows transferring some dynamic problems to the static level. In particular, a variant of static polymorphism application without virtual functions can be considered.

A variant of non-virtual double scheduling has been proposed, generalized in the form of a created design pattern "Signature multimethod". The use of the newly created pattern is illustrated with an example of implementing classes of complex numbers. The absence of violations of SOLID principles is shown, and the possibility of supplementing the hierarchy with new derived classes without the need to interfere with the structure of the base class is demonstrated.

The approach suggested in this work has been used in courses in object-oriented programming at the Faculty of Informatics of Kyiv-Mohyla Academy.

Keywords: C++, object-oriented programming, design pattern, double dispatching (multimethod), virtual function, polymorphism, generalized programming.

Матеріал надійшов 02.06.2021

Creative Commons Attribution 4.0 International License (CC BY 4.0)