Stosowanie wzorców projektowych w C++

Kod wielokrotnego wykorzystania w programowaniu zorientowanym obiektowo

—

Dmitri Nesteruk

Helion

Apress

Tytuł oryginału: Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design

Tłumaczenie: Krzysztof Bąbol, z wykorzystaniem fragmentów książki "Wzorce projektowe w .NET. Projektowanie zorientowane obiektowo z wykorzystaniem C# i F#" w przekładzie Radosława Meryka

ISBN: 978-83-283-7175-0

First published in English under the title Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design by Dmitri Nesteruk, edition: 1

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/stwzpr.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/stwzpr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Wprowadzenie

Temat wzorców projektowych brzmi sucho, akademicko nudnie i, szczerze mówiąc, jest implementowany na siłę w prawie każdym języku programowania, jaki można sobie wyobrazić. W tej grupie są także takie języki programowania, jak JavaScript, które nawet nie są prawidłowymi językami obiektowymi! Po co więc kolejna książka na ten temat?

Wydaje mi się, że głównym motywem, który skłonił mnie do napisania tej książki, jest to, że język C++ znowu wkroczył w okres świetności. Po długim okresie stagnacji ewoluuje, rozwija się i pomimo konieczności zachowania wstecznej kompatybilności z językiem C jest wiele zmian na lepsze, choć nie w takim tempie, jakiego wszyscy byśmy sobie życzyli. (Czekam m.in. na wprowadzenie modułów).

Przechodząc do wzorców projektowych — nie wolno nam zapominać, że *pierwsza* książka[1] na ten temat zawierała przykłady w językach C++ i Smalltalk. Od tamtego czasu w wielu przypadkach wzorce projektowe wkomponowano bezpośrednio w język programowania: np. w C# wbudowano wzorzec Obserwator ze wsparciem dla zdarzeń (i słowem kluczowym `event`). W języku C++ tak się *nie* stało, przynajmniej nie na poziomie składni, jednak wprowadzenie typów takich jak `std::function` w wielu sytuacjach uprościło pracę programistów.

Wzorce projektowe mogą również być podstawą ciekawego dochodzenia, jak można rozwiązać problem na wiele różnych sposobów, o różnym stopniu zaawansowania technicznego i z zastosowaniem różnego rodzaju kompromisów. Niektóre wzorce są bardziej lub mniej istotne oraz nieuniknione, podczas gdy inne omówiono przede wszystkim z powodu naukowej ciekawości (niemniej jednak omówiłem je, bo jestem zwolennikiem opisów kompletnych).

Czytelnicy powinni być świadomi, że kompleksowe rozwiązania niektórych problemów często prowadzą do nadinżynierii lub tworzenia struktur i mechanizmów, które są znacznie bardziej skomplikowane, niż to konieczne dla większości typowych scenariuszy. Chociaż nadinżynieria bywa zabawna (rozwiązujesz problem, a jednocześnie imponujesz współpracownikom), to często jej stosowanie nie jest możliwe ze względu na ograniczenia dotyczące czasu, kosztów i złożoności.

---

[1] Erich Gamma i in., *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, Helion, Gliwice 2010.

Preliminaria

Dla kogo jest ta książka?

Ta książka ma być współczesną aktualizacją do klasycznej książki „gangu czterech" (ang. *Gang of Four* — od czterech autorów) w odniesieniu do języka programowania C++. Bo w końcu ilu z Was pisze w języku Smalltalk? Podejrzewam, że niezbyt wielu.

Celem tej książki jest zbadanie, w jaki sposób możemy stosować nowoczesny C++ (najnowsze dostępne wersje języka C++) do implementacji klasycznych wzorców projektowych. Jednocześnie jest to również próba ukształtowania nowych wzorców i metod, które mogą być użyteczne dla programistów C++.

Wreszcie, w niektórych miejscach ta książka jest po prostu demonstracją technologii nowoczesnego języka C++. Prezentuje, w jaki sposób dzięki niektórym najnowszym cechom języka (na przykład koprocedurom, ang. *coroutines*) niektóre trudne problemy stały się o wiele łatwiejsze do rozwiązania.

O przykładach kodu

Wszystkie przykłady kodu zaprezentowane w tej książce nadają się do zastosowania w produkcji. Wprowadziłem jednak kilka uproszczeń w celu poprawy czytelności:

- Można zauważyć, że często używam słowa `struct` zamiast `class`, co pozwala uniknąć nadmiaru słów kluczowych `public`.
- Unikam przedrostka `std::`, który zaciemnia kod, zwłaszcza tam, gdzie jest on już zagęszczony. Jeśli pojawia się słowo `string`, można w ciemno założyć, że odnosi się ono do typu `std::string`.
- Nie dodaję wirtualnych destruktorów, chociaż w praktyce byłoby to zasadne.
- W nielicznych przypadkach tworzę i przekazuję parametry przez wartość, by nie mnożyć wystąpień `shared_ptr`/`make_shared`/itp. Inteligentne wskaźniki wprowadzają kolejny poziom złożoności, a ich integrację we wzorcach projektowych przedstawionych w tej książce pozostawiam jako zadanie dla Czytelników.
- Czasem opuszczam elementy zapewniające typowi pełną funkcjonalność (np. konstruktory przenoszące), bo zajmują one zbyt wiele miejsca.
- W wielu przypadkach pomijam modyfikator `const`, którego dodanie w normalnych okolicznościach byłoby zasadne. Poprawność stałych (ang. *const-correctness*) dość często powoduje rozbicie i podwojenie powierzchni interfejsu API, co niezbyt dobrze się sprawdza w formacie książkowym.

Należy zwrócić uwagę, że w większości przykładów wykorzystałem najnowsze wersje C++ (C++11, 14, 17 i późniejsze) i, ogólnie rzecz biorąc, korzystam z najnowszych własności języka C++, które są dostępne dla programistów. Na przykład w książce tej nie ma zbyt wielu sygnatur funkcji z zapisem `-> decltype(...)` na końcu, bo w standardzie C++14 wprowadzono

automatyczną dedukcję typu zwracanego. Żaden z przykładów nie jest ukierunkowany na konkretny kompilator, ale jeśli w tym wybranym coś nie działa[2], trzeba szukać obejść.

Czasami odwołuję się do innych języków programowania, takich jak C# lub Kotlin. Od czasu do czasu warto dowiedzieć się, jak projektanci innych języków zaimplementowali konkretną cechę języka. C++ nierzadko zapożycza ogólnie dostępne pomysły z innych języków. Przykładem może być wprowadzenie słowa kluczowego `auto`. Taka inferencja typów w deklaracjach zmiennych i typów zwracanych występuje w wielu innych językach.

O narzędziach programistycznych

Przykłady kodu zawarte w tej książce pisałem z myślą o nowoczesnych kompilatorach C++, takich jak Clang, GCC czy MSVC. Poczyniłem ogólne założenie, że czytelnicy będą korzystać z ostatniej dostępnej wersji kompilatora, a co za tym idzie, z najnowszych i najlepszych cech języka. W razie stosowania starszych wersji kompilatorów należy zrezygnować z nowoczesnych konstrukcji językowych, ale nie zawsze jest to możliwe.

Zawarte w tej książce przykłady nie wymagają konkretnych narzędzi programistycznych, więc jeśli masz aktualny kompilator, będziesz mógł spokojnie je prześledzić: większość z nich to osobne pliki *.cpp*. Mimo wszystko chciałbym przy tej okazji przypomnieć, że wysokiej jakości narzędzia, takie jak CLion lub ReSharper C++, znacznie usprawniają pracę programisty. Po zainwestowaniu niewielkiej kwoty otrzymuje on mnóstwo dodatkowych funkcji, co bezpośrednio przekłada się na szybsze programowanie i lepszą jakość wytworzonego kodu.

Ważne koncepcje

Zanim zaczniemy, chciałbym pokrótce wspomnieć o kilku kluczowych koncepcjach ze świata C++, do których odwołuję się w tej książce.

Curiously Recurring Template Pattern (dosł. ciekawie rekurencyjny wzorzec szablonu)

Hej, to najwidoczniej wzorzec! Nie wiem, czy można go zakwalifikować jako oddzielny wzorzec *projektowy*, ale na pewno w świecie C++ jest swego rodzaju wzorcem. Pomysł jest w zasadzie prosty: jako argument szablonu do klasy bazowej przekazywana jest *sama* klasa dziedzicząca:

```
1   struct Foo : SomeBase<Foo>
2   {
3     ...
4   }
```

Możesz się zastanawiać, *po co* to w ogóle robić? No cóż, jednym z powodów jest możliwość dostępu do silnie typowanego wskaźnika `this` wewnątrz implementacji klasy bazowej.

[2] Firmo Intel, to o tobie!

Wyobraź sobie na przykład, że w każdej klasie dziedziczącej po `SomeBase` zaimplementowano potrzebną do iteracji parę funkcji `begin()/end()`. Czy da się w jakiś sposób iterować po obiekcie wewnątrz funkcji składowej klasy `SomeBase`? Intuicja podpowiada, że to niemożliwe, bo sama klasa `SomeBase` nie zapewnia interfejsu `begin()/end()`.Wykorzystanie wzorca CRTP pozwala jednak rzutować wskaźnik `this` do typu klasy pochodnej:

```
template <typename Derived>
struct SomeBase
{
  void foo()
  {
    for (auto& item : *static_cast<Derived*>(this))
    {
      ...
    }
  }
}
```

Konkretny przykład zastosowania tego podejścia znajduje się w rozdziale 9.

Dziedziczenie domieszek

W języku C++ klasę można zdefiniować tak, by dziedziczyła po własnym argumencie szablonu, np. tak:

```
template <typename T> struct Mixin : T
{
  ...
}
```

Takie podejście, nazywane **dziedziczeniem domieszek** (ang. *mixin inheritance*), pozwala na hierarchiczną kompozycję typów. Użycie zapisu `Foo<Bar<Baz>> x;` pozwala np. zadeklarować zmienną typu łączącego cechy wszystkich trzech klas bez konieczności rzeczywistego konstruowania całkiem nowego typu `FooBarBaz`.

Konkretny przykład zastosowania tego podejścia znajduje się w rozdziale 9.

Właściwości

Właściwość (ang. *property*) to nic innego jak (zazwyczaj prywatne) pole oraz połączenie gettera i settera. W standardowym C++ właściwość wygląda następująco:

```
class Person
{
  int age;
public:
  int get_age() const { return age; }
  void set_age(int value) { age = value; }
};
```

W wielu językach programowania (m.in. w C# i Kotlinie) zinternalizowano pojęcie właściwości, wprowadzając je bezpośrednio do języka. W języku C++ do tego nie doszło (i prawdopodobnie nigdy tak się nie stanie), ale większość kompilatorów (MSVC, Clang, Intel) pozwala na użycie niestandardowego specyfikatora deklaracji o nazwie `property`:

```
1   class Person
2   {
3     int age_;
4   public:
5     int get_age() const { return age_; }
6     void set_age(int value) { age_ = value; }
7     __declspec(property(get=get_age, put=set_age)) int age;
8   };
```

Korzysta się z tego następująco:

```
1   Person person;
2   p.age = 20; // wywołuje funkcję p.set_age(20)
```

Zasady projektowania SOLID

SOLID to akronim, który oznacza następujące zasady projektowania (i ich skróty):

- zasada pojedynczej odpowiedzialności (ang. *Single Responsibility Principle* — SRP),
- zasada otwarty-zamknięty (ang. *Open-Closed Principle* — OCP),
- zasada podstawiania Liskov (ang. *Liskov Substitution Principle* — LSP),
- zasada segregacji interfejsów (ang. *Interface Segregation Principle* — ISP),
- zasada odwracania zależności (ang. *Dependency Inversion Principle* — DIP).

Zasady te wprowadził Robert C. Martin na początku lat 2000. W gruncie rzeczy jest to wybór pięciu zasad spośród kilkudziesięciu, jakie Martin sformułował w swoich książkach i na swoim blogu. Wymienione pięć zasad zdominowało opis wzorców i ogólnie projektowania oprogramowania. Zanim zanurzymy się w zagadnienia dotyczące wzorców projektowych (wiem, że już nie możecie się tego doczekać), zaprezentujemy krótkie wprowadzenie na temat tego, na czym polegają zasady SOLID.

Zasada pojedynczej odpowiedzialności

Załóżmy, że zdecydowałeś się prowadzić dziennik swoich najbardziej intymnych myśli. Dziennik ma tytuł i kilka wpisów. Możesz zamodelować go w następujący sposób:

```
1   struct Journal
2   {
3     string title;
4     vector<string> entries;
5
6     explicit Journal(const string& title) : title{title} {}
7   };
```

Teraz możesz dodać funkcję dodawania wpisu do dziennika. Wpis będzie poprzedzony jego numerem porządkowym. To łatwe:

```
void Journal::add(const string& entry)
{
  static int count = 1;
  entries.push_back(boost::lexical_cast<string>(count++)
    + ": " + entry);
}
```

Z dziennika można teraz skorzystać w następujący sposób:

```
Journal j{"Drogi dzienniczku"};
j.add("Płakałem dzisiaj.");
j.add("Zjadłem robaka.");
```

Sensowne jest, aby ta funkcja była częścią klasy `Journal`, ponieważ dziennik powinien być odpowiedzialny za dodawanie do niego wpisów. Odpowiedzialnością dziennika jest przechowywanie wpisów, więc umieszczanie w klasie wszystkiego, co jest z tym związane, jest w porządku.

Załóżmy, że postanowiłeś utrwalić dziennik przez zapisanie go do pliku. W tym celu dodajesz do klasy `Journal` następujący kod:

```
void Journal::save(const string& filename)
{
  ofstream ofs(filename);
  for (auto& s : entries)
    ofs << s << endl;
}
```

To podejście jest problematyczne. Zadaniem dziennika jest *przechowywanie* wpisów, a nie zapisywanie ich na dysku. Jeśli dodasz funkcję utrwalania do klasy `Journal` i podobnych klas, każda zmiana w podejściu do utrwalania (powiedzmy, że zdecydujesz się zapisywać dziennik w chmurze zamiast na dysku) wymagałaby wielu drobnych zmian w każdej z klas, których dotyczy problem.

Chciałbym się tutaj zatrzymać i sformułować tezę: architektura, która prowadzi do wprowadzania wielu drobnych zmian w wielu klasach, niezależnie od tego, czy są one powiązane (tak, jak w hierarchii), czy nie, jest zwykle *cuchnącym kodem* (ang. *code smell*) — wskazuje, że coś jest nie tak. Wszystko zależy od sytuacji: jeśli zmieniasz nazwę symbolu, który jest używany w stu miejscach, to uważam, że ogólnie jest wszystko w porządku. ReSharper, CLion lub jakiekolwiek inne zintegrowane środowisko programistyczne (IDE), którego używasz, pozwoli Ci wykonać tę refaktoryzację i ją za Ciebie przeprowadzi. Kiedy jednak trzeba zupełnie przerobić interfejs, to taki proces może być bardzo bolesny!

Dlatego należy stwierdzić, że utrwalanie jest osobną kwestią, którą lepiej wyrazić w osobnej klasie, np. w następujący sposób:

```
struct PersistenceManager
{
  static void save(const Journal& j, const string& filename)
  {
    ofstream ofs(filename);
    for (auto& s : j.entries)
      ofs << s << endl;
```

```
8     }
9   };
```

Właśnie w taki sposób rozumiemy *pojedynczą odpowiedzialność*: każda klasa jest odpowiedzialna tylko za jedną rzecz, a zatem istnieje tylko jeden powód do jej modyfikacji. Klasa `Journal` będzie musiała się zmienić tylko wtedy, gdy będzie coś jeszcze do zrobienia w związku z przechowywaniem wpisów w pamięci. Na przykład możemy zdecydować, że każdy wpis będzie poprzedzony znacznikiem czasu. W takim przypadku trzeba będzie zmodyfikować funkcję `add()` tak, aby dokładnie to robiła. Z kolei aby zmienić mechanikę utrwalania, należy zmodyfikować klasę `PersistenceManager`.

Skrajnym przykładem antywzorca, który narusza zasadę SRP, jest wzorzec o nazwie *God Object* (dosł. obiekt-bóg). Obiekt-bóg to ogromna klasa, która stara się rozwiązać jak najwięcej problemów. Z tego powodu staje się monolitycznym potworem, z którym bardzo trudno się pracuje. W gruncie rzeczy każdy system o dowolnych rozmiarach można próbować zmieścić w pojedynczej klasie. Zazwyczaj jednak w efekcie powstaje niezrozumiały chaos. Na szczęście dla nas obiekty-bogowie są łatwe do rozpoznania — wizualnie lub automatycznie (wystarczy policzyć liczbę funkcji składowych), a dzięki systemom ciągłej integracji i kontroli wersji można szybko zidentyfikować odpowiedzialnego programistę i go odpowiednio ukarać.

Zasada otwarty-zamknięty

Załóżmy, że w bazie danych mamy zapisane dane o pewnych produktach. Każdy produkt ma kolor i rozmiar i jest zdefiniowany w następujący sposób:

```
1   enum class Color { Red, Green, Blue };
2   enum class Size { Small, Medium, Large };
3
4   struct Product
5   {
6     string name;
7     Color color;
8     Size size;
9   };
```

Teraz chcemy zapewnić pewne możliwości filtrowania dla wybranego zestawu produktów. Tworzymy więc następujący filtr:

```
1   struct ProductFilter
2   {
3     typedef vector<Product*> Items;
4   };
```

Aby zapewnić możliwość filtrowania produktów według koloru, definiujemy do tego funkcję składową:

```
1   ProductFilter::Items ProductFilter::by_color(Items items,  Color color)
2   {
3     Items result;
4     for (auto& i : items)
5       if (i->color == color)
6         result.push_back(i);
```

```
    return result;
  }
```

Nasze obecne podejście do filtrowania elementów według koloru jest dobre. Kod zostaje więc opublikowany, ale niestety, za jakiś czas szef prosi nas również o zaimplementowanie filtrowania według rozmiaru. Wracamy do pliku *ProductFilter.cpp*, dodajemy poniższy kod i ponownie kompilujemy:

```
ProductFilter::Items ProductFilter::by_size(Items items, Size size)
{
  Items result;
  for (auto& i : items)
    if (i->size == size)
      result.push_back(i);
  return result;
}
```

Czy nie wydaje Ci się, że ten kod jest prawie w całości zdublowany? Dlaczego nie napisać generycznej metody, która przyjmuje predykat (egzemplarz klasy szablonowej `function`)? No cóż, powodem może być to, że różne formy filtrowania mogą być wykonywane na różne sposoby: na przykład niektóre typy rekordów mogą być indeksowane i muszą być wyszukiwane w określony sposób; pewne typy danych można lepiej przeszukiwać na procesorach graficznych (GPU), podczas gdy inne nie.

OK, więc nasz kod wchodzi do produkcji, ale ponownie szef wraca i mówi nam, że teraz trzeba wprowadzić możliwość wyszukiwania według rozmiaru *i* koloru. Cóż więc innego zrobić, niż dodać kolejną funkcję?

```
ProductFilter::Items ProductFilter::by_color_and_size(Items
  items, Size size, Color color)
{
  Items result;
  for (auto& i : items)
    if (i->size == size && i->color == color)
      result.push_back(i);
  return result;
}
```

W tym scenariuszu chcemy wymusić *zasadę otwarty-zamknięty*, która stwierdza, że typ powinien być otwarty na rozszerzenia, ale zamknięty na modyfikacje. Innymi słowy, chcemy filtrowania, które można rozszerzać (być może za pomocą innego modułu) bez konieczności jego modyfikacji (i ponownej kompilacji czegoś, co już działa i być może zostało wysłane do klientów).

Jak można to osiągnąć? Po pierwsze rozdzielamy (zgodnie z zasadą SRP!) proces filtrowania na dwie części: filtrowanie (mechanizm, który pobiera wszystkie elementy i zwraca tylko niektóre) oraz specyfikację (predykat do zastosowania do elementu danych).

Możemy stworzyć bardzo prostą definicję interfejsu specyfikacji:

```
template <typename T> struct Specification
{
  virtual bool is_satisfied(T* item) = 0;
};
```

W tym kodzie `T` oznacza dowolny typ. Z pewnością może to być typ `Product`, ale może być także czymś innym. To sprawia, że ten kod nadaje się do wielokrotnego użycia.

Potrzebujemy teraz sposobu filtrowania na podstawie `Specification<T>`. W tym celu definiujemy, jak łatwo się domyślić, `Filter<T>`:

```
template <typename T> struct Filter
{
  virtual vector<T*> filter(
    vector<T*> items,
    Specification<T>& spec) = 0;
};
```

Teraz także nasze działania sprowadziły się do zdefiniowania sygnatury funkcji o nazwie `filter`, która pobiera wszystkie elementy i specyfikację i zwraca tylko te elementy, które są zgodne ze specyfikacją. Założyliśmy, że elementy będą przechowywane w kontenerze `vector<T*>`, ale w praktyce do funkcji `filter()` można przekazać parę iteratorów albo niestandardowy interfejs zaprojektowany specjalnie do przechodzenia po kolekcji. Niestety w języku C++ nie zdołano ustandaryzować pojęcia kolekcji, które istnieje w innych językach programowania (przykładem może być interfejs `IEnumerable` frameworka .NET).

Na podstawie powyższej definicji *implementacja* poprawionego filtra jest bardzo prosta:

```
struct BetterFilter : Filter<Product>
{
  vector<Product*> filter(
    vector<Product*> items,
    Specification<Product>& spec) override
  {
    vector<Product*> result;
    for (auto& p : items)
      if (spec.is_satisfied(p))
        result.push_back(p);
    return result;
  }
};
```

O przekazywanej specyfikacji `Specification<T>` możesz pomyśleć jako o silnie typowanym odpowiedniku `std::function`, który ma skończony zestaw konkretnych ustawień filtrów.

Pozostało to, co łatwe. Aby utworzyć filtr według koloru, implementujemy specyfikację koloru:

```
struct ColorSpecification : Specification<Product>
{
  Color color;

  explicit ColorSpecification(const Color color) :   color{color} {}

  bool is_satisfied(Product* item) override {
    return item->color == color;
  }
};
```

Mając do dyspozycji tę specyfikację oraz listę produktów, możemy teraz je filtrować w następujący sposób:

```
Product apple{ "Jabłko", Color::Green, Size::Small };
Product tree{ "Drzewo", Color::Green, Size::Large };
Product house{ "Dom", Color::Blue, Size::Large };

vector<Product*> all{ &apple, &tree, &house };

BetterFilter bf;
ColorSpecification green(Color::Green);

auto green_things = bf.filter(all, green);
for (auto& x : green_things)
  cout << x->name << " jest zielone" << endl;
```

Powyższy kod zwraca produkty „Jabłko” i „Drzewo”, ponieważ oba mają kolor zielony. Teraz jedyną rzeczą, której do tej pory nie zaimplementowaliśmy, jest wyszukiwanie według rozmiaru *i* koloru (a właściwie wyjaśnienie, w jaki sposób szukać według rozmiaru *lub* koloru bądź łączyć ze sobą różne kryteria). Odpowiedź brzmi następująco: należy stworzyć specyfikację *złożoną* (czyli *kombinator*). Na przykład dla logicznego `AND` można to zrobić w następujący sposób:

```
template <typename T> struct AndSpecification : Specification<T>
{
  Specification<T>& first;
  Specification<T>& second;

  AndSpecification(Specification<T>& first,
  Specification<T>& second)
    : first{first}, second{second} {}

  bool is_satisfied(T* item) override
  {
    return first.is_satisfied(item) && second.is_satisfied(item);
  }
};
```

Teraz możesz tworzyć złożone warunki na podstawie prostszych specyfikacji `Specification`. Ponowne skorzystanie ze stworzonej wcześniej specyfikacji `green`, żeby znaleźć coś zielonego i dużego, sprowadza się teraz do następującego kodu:

```
SizeSpecification large(Size::Large);
ColorSpecification green(Color::Green);
AndSpecification<Product> green_and_large{ large, green };

auto big_green_things = bf.filter(all, green_and_big);
for (auto& x : big_green_things)
  cout << x->name << " jest duże i zielone" << endl;

// Drzewo jest duże i zielone
```

Napisaliśmy dużo kodu! Należy jednak pamiętać, że dzięki możliwościom języka C++ można wprowadzić operator && dla dwóch obiektów `Specification <T>`, dzięki czemu proces filtrowania według dwóch (lub większej liczby!) kryteriów jest wyjątkowo prosty:

```
template <typename T> struct Specification
{
  virtual bool is_satisfied(T* item) = 0;

  AndSpecification<T> operator &&(Specification&& other)
  {
    return AndSpecification<T>(*this, other);
  }
};
```

Jeśli teraz zrezygnujemy z tworzenia osobnych zmiennych dla specyfikacji rozmiaru i koloru, to złożoną specyfikację będzie można zapisać w jednym wierszu:

```
auto green_and_big =
  ColorSpecification(Color::Green)
  && SizeSpecification(Size::Large);
```

Podsumujmy więc, czym jest zasada OCP i jak jest egzekwowana w prezentowanym przykładzie. Ogólnie rzecz biorąc, zasada OCP mówi, że nie powinieneś mieć potrzeby wracania do kodu, który już został napisany i przetestowany, po to, żeby go zmienić. Właśnie tak się dzieje w naszym przykładzie! Zdefiniowaliśmy interfejsy `Specification<T>` i `Filter<T>` i odtąd wszystko, co musimy robić, aby zaimplementować nową mechanikę filtrowania, to implementacja jednego z tych interfejsów (bez modyfikacji samych interfejsów). To właśnie oznacza „otwarty na rozszerzenia, zamknięty na modyfikację".

Zasada podstawiania Liskov

Zasada podstawiania Liskov, nazwana na cześć Barbary Liskov, stwierdza, że jeśli interfejs akceptuje obiekt typu `Rodzic`, to powinien on również, bez żadnej szkody dla jego działania, przyjmować obiekt typu `Dziecko`. Rzućmy okiem na sytuację, w której zasada LSP jest złamana.

Oto klasa reprezentująca prostokąt; ma właściwości reprezentujące szerokość i wysokość oraz kilka getterów i setterów do obliczania powierzchni:

```
class Rectangle
{
protected:
  int width, height;
public:
  Rectangle(const int width, const int height)
    : width{width}, height{height} { }

  int get_width() const { return width; }
  virtual void set_width(const int width) { this->width = width; }
  int get_height() const { return height; }
  virtual void set_height(const int height) { this->height = height; }

  int area() const { return width * height; }
};
```

Załóżmy, że stworzyliśmy specjalny rodzaj prostokąta, zwany kwadratem, reprezentowany przez klasę `Square`. Ten obiekt przesłania settery, które ustawiają zarówno szerokość, jak i wysokość:

```
class Square : public Rectangle
{
public:
  Square(int size): Rectangle(size,size) {}
  void set_width(const int width) override {
    this->width = height = width;
  }
  void set_height(const int height) override {
    this->height = width = height;
  }
};
```

To podejście jest *złe*. Jeszcze tego nie widać, ponieważ wygląda naprawdę bardzo niewinnie: settery po prostu ustawiają oba wymiary (tak aby kwadrat zawsze był kwadratem). Co może pójść nie tak? No cóż, jeśli przyjmiemy te założenia, możemy niechcący stworzyć funkcję wykorzystującą obiekt klasy `Rectangle`, która zawiedzie po otrzymaniu kwadratu:

```
void process(Rectangle& r)
{
  int w = r.get_width();
  r.set_height(10);

  cout << "Oczekiwane pole powierzchni = " << (w * 10)
    << ", uzyskano " << r.area() << endl;
}
```

Powyższa funkcja przyjmuje formułę `Area = Width × Height` jako niezmiennik. Pobiera szerokość, ustawia wysokość na 10 i słusznie oczekuje, że iloczyn będzie równy obliczonemu polu powierzchni. Wywołanie tej funkcji z obiektem `Square` zwraca jednak nieprawidłową wartość:

```
Square s{5};
process(s); // Oczekiwane pole powierzchni 50, uzyskano 25
```

Z przykładu tego (muszę przyznać, nieco sztucznego) płynie wniosek, że funkcja `process()` narusza zasadę LSP przez to, że jest całkowicie niezdolna przyjąć typ pochodny `Square` zamiast typu bazowego `Rectangle`. Jeśli poda się jej obiekt klasy `Rectangle`, wszystko jest w porządku, więc może minąć trochę czasu, zanim problem wyjdzie na jaw podczas testów (albo po wdrożeniu produkcyjnym — oby nie!).

Jak można to rozwiązać? No cóż, na wiele sposobów. Osobiście obstawałbym przy tym, że typ `Square` nie powinien w ogóle istnieć: zamiast tego można utworzyć fabrykę (patrz rozdział 3.) do tworzenia zarówno prostokątów, jak i kwadratów:

```
struct RectangleFactory
{
  static Rectangle create_rectangle(int w, int h);
  static Rectangle create_square(int size);
};
```

Można też próbować wykryć, czy prostokąt jest w istocie kwadratem:

```
1   bool Rectangle::is_square() const
2   {
3     return width == height;
4   }
```

Wariantem ekstremalnym w tym przypadku jest zgłaszanie wyjątku w funkcjach `set_width()` i `set_height()` klasy `Square`, oznajamiającego, że operacje te nie są obsługiwane i zamiast tego należy korzystać z funkcji `set_size()`. To podejście narusza jednak zasadę „najmniejszego zaskoczenia" (ang. *least surprise*), bo można się spodziewać, że po wywołaniu `set_width()` nastąpi pewna zmiana… nieprawdaż?

Zasada segregacji interfejsów

Oto kolejny wymyślony przykład, który jednak wystarczy do zilustrowania problemu. Załóżmy, że postanowiłeś stworzyć drukarkę wielofunkcyjną — urządzenie, które potrafi drukować, skanować, a także faksować dokumenty. Można je zdefiniować w następujący sposób:

```
1   struct MyFavouritePrinter /* : IMachine */
2   {
3     void print(vector<Document*> docs) override;
4     void fax(vector<Document*> docs) override;
5     void scan(vector<Document*> docs) override;
6   };
```

Definicja wygląda dobrze. Załóżmy teraz, że zdecydowaliśmy się zdefiniować interfejs, który musi zostać zaimplementowany przez każdego, kto chce stworzyć wielofunkcyjną drukarkę. W tym celu możesz użyć funkcji *Extract Interface* w swoim ulubionym środowisku IDE. W efekcie otrzymasz następujący interfejs:

```
1   struct IMachine
2   {
3     virtual void print(vector<Document*> docs) = 0;
4     virtual void fax(vector<Document*> docs) = 0;
5     virtual void scan(vector<Document*> docs) = 0;
6   };
```

Taki interfejs nie jest zbyt dobry. Chodzi o to, że niektórzy implementatorzy tego interfejsu mogliby zrezygnować ze skanowania lub faksowania i zaimplementować tylko drukowanie. Jednak ze względu na taką definicję interfejsu zmuszasz ich do zaimplementowania tych dodatkowych funkcji. Oczywiście można by pozostawić niepotrzebne funkcje puste, ale po co się tym przejmować?

Zasada ISP mówi, że zamiast tego należy rozdzielić interfejsy tak, aby implementatorzy mogli wybrać je w zależności od potrzeb. Ponieważ drukowanie i skanowanie to różne operacje (tzn. skaner nie może drukować), definiujemy dla nich osobne interfejsy:

```
1   struct IPrinter
2   {
3     virtual void print(vector<Document*> docs) = 0;
4   };
5
```

```
struct IScanner
{
  virtual void scan(vector<Document*> docs) = 0;
};
```

W tym przypadku drukarka lub skaner mogą zaimplementować *tylko* wymaganą funkcjonalność:

```
struct Printer : IPrinter
{
  void print(vector<Document*> docs) override;
};

struct Scanner : IScanner
{
  void scan(vector<Document*> docs) override;
};
```

Teraz, jeśli naprawdę chcemy interfejsu dla urządzenia wielofunkcyjnego, definiujemy go jako kombinację wyżej wymienionych interfejsów:

```
struct IMachine: IPrinter, IScanner /* IFax itd. */
{
};
```

Kiedy trzeba zaimplementować interfejs w konkretnym urządzeniu wielofunkcyjnym, to należy użyć właśnie tego interfejsu. Na przykład moglibyśmy użyć prostej delegacji, aby upewnić się, że obiekt `Machine` korzysta z funkcjonalności dostarczanej przez określoną implementację `IPrinter` i `IScanner` (jest to dobra ilustracja wzorca Dekorator):

```
struct Machine : IMachine
{
  IPrinter& printer;
  IScanner& scanner;

  Machine(IPrinter& printer, IScanner& scanner)
    : printer{printer},
      scanner{scanner}
  {
  }

  void print(vector<Document*> docs) override {
    printer.print(docs);
  }

  void scan(vector<Document*> docs) override
  {
    scanner.scan(docs);
  }
};
```

Podsumowując, pomysł polega na rozdzieleniu części skomplikowanego interfejsu na osobne interfejsy, aby uniknąć zmuszania klientów do implementowania funkcjonalności, których w gruncie rzeczy nie potrzebują. Ilekroć piszesz wtyczkę do jakiejś skomplikowanej aplikacji i dostajesz do zaimplementowania interfejs z 20 mylącymi funkcjami z różnymi brakami implementacji i zwracającymi wartości `nullptr`, istnieje duże prawdopodobieństwo, że autorzy API naruszyli zasadę ISP.

Zasada odwracania zależności

Oryginalna definicja zasady odwracania (nazywanej także inwersją) zależności określa, co następuje[3]:

A. *Wysokopoziomowe moduły nie powinny zależeć od modułów niskiego poziomu. I jedne, i drugie powinny zależeć od abstrakcji.*

   To zdanie, ogólnie rzecz biorąc, oznacza, że jeśli jesteś zainteresowany logowaniem, to komponenty logujące nie powinny zależeć od konkretnej implementacji `ConsoleLogger`, ale mogą zależeć od interfejsu `ILogger`. W tym przypadku uważamy, że komponent logujący jest wysokiego poziomu (bliżej dziedziny biznesowej), podczas gdy logowanie, jako zasadnicze działanie (rodzaj plikowego systemu wejścia-wyjścia lub mechanizmu obsługi wątków), jest uważane za moduł niskopoziomowy.

B. *Abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji.*

   To jest ponowne stwierdzenie, że zależności od interfejsów lub klas bazowych są lepsze niż zależności od typów konkretnych. Mamy nadzieję, że to zdanie jest oczywiste, ponieważ takie podejście wspiera lepsze możliwości konfiguracji i testowania, pod warunkiem że korzystasz z dobrego frameworka do obsługi tych zależności.

Zasadnicze pytanie jest teraz takie: jak to wszystko w praktyce zaimplementować? Z całą pewnością będzie to wymagać znacznie większego nakładu pracy, bo trzeba jawnie określać, że np. klasa `Reporting` jest zależna od interfejsu `ILogger`. Można to wyrazić np. w taki sposób:

```
1   class Reporting
2   {
3     ILogger& logger;
4   public:
5     Reporting(const ILogger& logger) : logger{logger} {}
6     void prepare_report()
7     {
8       logger.log_info("Przygotowywanie raportu");
9       ...
10    }
11  };
12  }
```

Problem polega na tym, że aby zainicjować powyższą klasę, należy dokonać jawnego wywołania w postaci `Reporting{ConsoleLogger{}}` lub czegoś podobnego. A co będzie, jeśli klasa `Reporting` jest zależna od pięciu różnych inferfejsów? Albo jeśli `ConsoleLogger` ma swoje własne zależności? *Można* sobie z tym poradzić i napisać wiele kodu, ale istnieje lepsza metoda.

Nowoczesnym, stylowym i modnym sposobem rozwiązania tego problemu jest **wstrzykiwanie zależności** (ang. *Dependency Injection*): zasadniczo wymaga ono skorzystania

[3] Robert C. Martin, *Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki*, Helion, Gliwice 2015, s. 141 – 148.

z biblioteki, takiej jak Boost.DI[4], która *automatycznie* zaspokaja wymagania danego komponentu wobec zależności.

Rozważmy przykład samochodu, który ma silnik, ale musi też logować pewne dane. Możemy w zasadzie stwierdzić, że auto *jest uzależnione* od obu tych rzeczy. Na początek zdefiniujemy silnik:

```
1   struct Engine
2   {
3     float volume = 5;
4     int horse_power = 400;
5
6     friend ostream& operator<< (ostream& os, const Engine& obj)
7     {
8       return os
9         << "pojemność: " << obj.volume
10        << "moc w KM: " << obj.horse_power;
11    } // dzięki, ReSharper!
12  };
```

Teraz od nas zależy, czy chcemy, czy nie chcemy wyodrębniać interfejs `IEngine` i przekazywać go do samochodu. Możemy, ale nie musimy tego robić; zwykle jest to decyzja projektowa. Jeśli planujesz utworzenie hierarchii silników albo widzisz potrzebę posiadania klasy `NullEngine` (patrz rozdział 19.) do celów testowych, wtedy rzeczywiście potrzebujesz wyodrębnić interfejs.

Niezależnie od tego potrzebujemy również logować dane, a ponieważ można to robić na wiele sposobów (w konsoli, poprzez wiadomości e-mail, SMS, pocztą gołębią…), z pewnością będziemy potrzebować interfejsu `ILogger`:

```
1   struct ILogger
2   {
3     virtual ~ILogger() {}
4     virtual void Log(const string& s) = 0;
5   };
```

a także jakiejś jego implementacji:

```
1   struct ConsoleLogger : ILogger
2   {
3     ConsoleLogger() {}
4
5     void Log(const string& s) override
6     {
7       cout << "DZIENNIK: " << s.c_str() << endl;
8     }
9   };
```

Teraz definiowany przez nas samochód jest uzależniony zarówno od silnika, *jak i* od komponentu logowania. Potrzebujemy ich obu, ale od nas zależy, jak je będziemy przechowywać: możemy użyć zwykłego wskaźnika, referencji, wskaźnika `unique_ptr` lub `shared_ptr` albo czegoś jeszcze innego. Zdefiniujemy oba zależne komponenty jako parametry konstruktora:

---

[4] W tej chwili biblioteka Boost.DI nie jest jeszcze częścią właściwej kolekcji Boost, ale wchodzi w skład repozytorium `boost-ext` w serwisie GitHub.

```
1   struct Car
2   {
3     unique_ptr<Engine> engine;
4     shared_ptr<ILogger> logger;
5
6     Car(unique_ptr<Engine> engine,
7         const shared_ptr<ILogger>& logger)
8       : engine{move(engine)},
9         logger{logger}
10    {
11      logger->Log("produkcja samochodu");
12    }
13
14    friend ostream& operator<<(ostream& os, const Car& obj)
15    {
16      return os << "samochód z silnikiem: " << *obj.engine;
17    }
18  };
```

Zapewne spodziewasz się teraz, że podczas inicjacji obiektu klasy `Car` zobaczysz wiele wywołań `make_unique/make_shared`. Nie będziemy jednak robić nic takiego. Zamiast tego użyjemy biblioteki Boost.DI. Najpierw zdefiniujemy powiązanie interfejsu `ILogger` z klasą `ConsoleLogger`; mówi ono po prostu: „za każdym razem, gdy ktoś poprosi o interfejs `ILogger`, daj mu klasę `ConsoleLogger`”:

```
1   auto injector = di::make_injector(
2     di::bind<ILogger>().to<ConsoleLogger>()
3   );
```

Skoro skonfigurowaliśmy iniektor, możemy go wykorzystać do utworzenia samochodu:

```
1   auto car = injector.create<shared_ptr<Car>>();
```

W pokazanym powyżej wierszu tworzony jest inteligentny wskaźnik `shared_ptr<Car>` wskazujący *całkowicie zainicjowany* obiekt klasy `Car`, czyli dokładnie to, czego chcieliśmy. Wspaniałe w tym podejściu jest to, że w celu zmiany typu używanego loggera wystarczy dokonać zmiany w jednym miejscu (wywołaniu iniektora `bind`), a wtedy w każdym miejscu, w którym pojawi się interfejs `ILogger`, będzie używany inny dostarczony przez nas komponent logujący. Podejście to pomaga też w testach jednostkowych i pozwala stosować zaślepki (ang. *stubs*) (albo wzorzec Pusty obiekt) zamiast atrap (ang. *mocks*).

Czas na wzorce!

Dzięki zrozumieniu zasad projektowania SOLID jesteśmy gotowi przyjrzeć się samym wzorcom projektowym. Przygotujcie się; to będzie długa (miejmy jednak nadzieję, że nie nudna) przejażdżka!

Wzorce projektowe w C++: gwarancja najlepszej architektury!

Wzorce projektowe powinny się znaleźć w przyborniku każdego profesjonalnego programisty. Ich zaletą jest nie tylko łatwość tworzenia kodu wielokrotnego użytku, ale także możliwość szybkiego rozwiązywania złożonych zagadnień. Osoby, które chcą rozwijać swoje umiejętności programistyczne, mogą również wykorzystać wzorce projektowe do inspirującego dochodzenia, jak można rozwiązać konkretny problem na wiele sposobów — o zróżnicowanym stopniu zaawansowania technicznego i z zastosowaniem różnego rodzaju kompromisów. Taka zabawa jest bardzo zajmująca i pouczająca.

To książka przeznaczona dla programistów C++, którzy chcą poszerzyć swoją wiedzę na temat wzorców projektowych przy użyciu standardu C++17. Opisano tu zarówno klasyczne, jak i całkiem nowoczesne wzorce projektowe ułatwiające rozwiązywanie konkretnych problemów programistycznych w optymalny sposób. Przedstawiono też znaczenie niektórych najnowszych cech języka C++ dla implementacji wzorców. Treść została zilustrowana szeregiem przykładów i scenariuszy pokazujących wykorzystanie wzorców, ich alternatywy i wzajemne relacje. Co więcej, przykłady kodu — przy zachowaniu czytelności — prezentują sobą wysoką jakość oprogramowania produkcyjnego.

W tej książce między innymi:

- zasady korzystania z wzorców projektowych w nowoczesnym C++
- wzorce kreacyjne: Budowniczy, Fabryka, Prototyp i Singleton
- wzorce strukturalne: Adapter, Most, Dekorator, Fasada
- wzorce zachowań: Łańcuch Odpowiedzialności, Polecenie, Iterator, Mediator
- funkcyjne wzorce projektowe, takie jak Monada

Dmitri Nesteruk jest analitykiem giełdowym i programistą. Występuje na konferencjach, tworzy kursy i pisze książki techniczne. Zawodowo interesuje się integracją rozwiązań w dziedzinie obliczeń, finansów i algorytmicznego handlu. Z upodobaniem programuje w C# i C++ i implementuje wysokowydajne przetwarzanie danych za pomocą takich technologii jak CUDA oraz FPGA. W 2009 roku za osiągnięcia w dziedzinie C# otrzymał tytuł MVP.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-7175-0
9 788328 371750

Cena: 57,00 zł
Apress