Przedmowa

Scott Meyers

W roku 1991 ukazało się pierwsze wydanie mojej książki *Effective C++*. Nie było w niej prawie żadnych informacji o szablonach, ponieważ były one tak nowym elementem języka, że niewiele o nich wiedziałem. Nawet w wypadku niewielkiego kodu zawierającego szablony wysyłałem do znajomych listy elektroniczne z prośbą o sprawdzenie go, gdyż żaden z dostępnych mi kompilatorów nie obsługiwał szablonów.

W roku 1995 ukazała się kolejna moja książka *More Effective C++*[1]. I znów na temat szablonów nie było w niej prawie nic. Tym razem jednak przed opisaniem ich powstrzymał mnie nie brak wiedzy (brudnopis zawierał cały rozdział na ich temat) ani nie ograniczenia kompilatorów, lecz podejrzenie, że w podejściu społeczności C++ do tych zagadnień zajdą lada dzień tak gwałtowne zmiany, że wszystko, co na ten temat miałem do powiedzenia, stanie się oklepane, sztuczne lub wręcz błędne.

Istniały dwa powody, by tak sądzić. W styczniu 1995 roku w *C++ Report* ukazał się artykuł Johna Bartona i Lee Nackmana, w którym opisali oni, jak za pomocą szablonów zaimplementować przy zerowych kosztach wykonania bezpieczną typologicznie analizę rozmiarów macierzy. Sam spędziłem nad tym problemem trochę czasu i wiedziałem, że wielu szukało rozwiązania, ale nikomu się to nie udało. Rewolucyjne podjęcie Bartona i Nackmana uzmysłowiło mi, że zastosowania szablonów wykraczają daleko poza „pojemnik elementów typu T".

Oto jak Barton i Nackman implementują mnożenie dwóch mianowanych wielkości fizycznych:

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator* (Physical<m1, l1, t1> lhs,
                                         Physical<m2, l2, t2> rhs)
{
  return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

[1] Polskie wydanie: *Język C++ bardziej efektywny*, Warszawa, WNT 1998 r.

Nawet bez wyjaśnień zawartych w artykule łatwo spostrzec, że przedstawiony szablon funkcyjny ma sześć parametrów, lecz żaden z nich nie reprezentuje typu! Ten sposób wykorzystania szablonów był dla mnie takim olśnieniem, że zakręciło mi się w głowie.

Wkrótce zacząłem czytać o STL. Elegancka struktura biblioteki Alexandra Stepanova, w której pojemniki nie zawierają informacji o algorytmach, a algorytmy informacji o pojemnikach, w której iteratory zachowują się jak wskaźniki (ale mogą być obiektami), pojemniki i algorytmy akceptują zarówno wskaźnik do funkcji, jak i obiekt funkcyjny, a klienci biblioteki mogą ją rozszerzać bez konieczności korzystania z mechanizmu dziedziczenia lub przedefiniowywania funkcji wirtualnych sprawia, że znów poczułem, iż o szablonach nie wiem prawie nic.

W tej sytuacji w książce *More Effective C++* o szablonach nie napisałem niemal wcale. Czemu? Moja wiedza o nich była wciąż na poziomie „pojemnik elementów typu T", podczas gdy Barton, Nackman, Stepanov i inni jasno wykazywali, że to tylko wierzchołek góry lodowej.

W roku 1998 zacząłem za pomocą poczty elektronicznej korespondować z Andreiem Alexandrescu i niebawem zdałem sobie sprawę, że po raz kolejny muszę zrewidować swoje poglądy na temat szablonów. Barton, Nackman i Stepanov zadziwili mnie tym, co za pomocą szablonów da się zrobić, a Andrei tym, w jaki sposób.

Gdy mam kogoś wprowadzić w tematykę prac Andreia, posługuję się pewnym przykładem. To jedna z najprostszych rzeczy, do której popularyzacji Andrei się przyczynił. Chodzi mianowicie o szablon `CTAssert`, podobny w swojej funkcjonalności do makra `assert`, ale używany z warunkami, których wartość daje się wyliczyć podczas kompilacji. Oto on:

```
template<bool> struct CTAssert;
template<> struct CTAssert<true> {};
```

To wszystko. Zauważ, że szablon podstawowy `CTAssert` nigdzie nie jest zdefiniowany. Istnieje specjalizacja dla `true`, ale nie dla `false`. To, czego w tym przykładzie nie ma, jest równie ważne jak to, co jest. Zaczynamy więc patrzeć na szablony z zupełnie innej perspektywy, ponieważ znaczne fragmenty „kodu źródłowego" są z premedytacją opuszczone. To sposób myślenia bardzo odmienny od dotychczasowego. W tej książce zamiast `CTAssert` Andrei omawia o wiele bardziej wyrafinowany szablon `CompileTimerChecker`.

Andrei zajął się opartymi na szablonach implementacjami popularnych idiomów C++ oraz wzorców projektowych, zwłaszcza tych pochodzących od Bandy Czterech[1]. Doprowadziło to do krótkiego spięcia z entuzjastami wzorców projektowych, którzy uważali, że wzorzec projektowy nie może być zapisany za pomocą kodu. Kiedy jednak się wyjaśniło, że rozwiązania Andreia polegają na automatycznym generowaniu *inplementacji* wzorców, a nie na kodowaniu wzorców jako takich, sprzeciw ustał, a ja z przyjemnością odnotowałem fakt, że dwa artykuły w *C++ Report*, dotyczące osiągnięć Andreia, powstały we współpracy z jednym z członków Bandy Czterech (Johnem Vlissidesem).

[1] „Banda Czterech" to określenie, które przylgnęło do Ericha Gammy, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa – autorów klasycznej książki o wzorcach projektowych pt. „Design Patterns: Elements of Reusable Object-Oriented Software" (polskie wydanie: „Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku"; Warszawa, WNT 2005).

Pracując nad szablonami, służącymi do generowania implementacji idiomów i wzorców, Andrei był zmuszony stanąć wobec różnych problemów projektowych, z którymi stykają się wszyscy programiści. Czy kod powinien być bezpieczny wątkowo? Czy dodatkowa pamięć powinna pochodzić ze stosu, sterty czy z puli statycznej? Czy inteligentne wskaźniki powinny wykonywać test na zero przed wyłuskaniem? Co powinno się stać przy wychodzeniu z programu, gdy destruktor jednego z singletonów próbuje użyć innego, już zniszczonego, singletonu? Celem Andreia było udostępnienie klientom jego biblioteki wszystkich możliwych wariantów projektowych bez wyróżniania któregokolwiek.

Rozwiązanie Andreia polega na zamknięciu tego typu decyzji projektowych wewnątrz tzw. *klas wytycznych* (ang. *policy classes*), umożliwieniu klientowi przekazywania ich do biblioteki jako parametrów szablonów, a także na zapewnieniu rozsądnych wytycznych domyślnych – tak, aby większość klientów nie musiała sobie nimi zawracać głowy. Rezultaty mogą być zaskakujące. Przedstawiony na przykład w tej książce szablon inteligentnego wskaźnika potrzebuje tylko czterech wytycznych, ale może wygenerować ponad 300 różnych typów tego wskaźnika, różniących się szczegółami dotyczącymi funkcjonalności. Programiści, którym wystarcza domyślne zachowanie inteligentnego wskaźnika, mogą zignorować wytyczne, podać tylko typ obiektu wskazywanego i praktycznie bez żadnego dodatkowego wysiłku czerpać korzyści z tej misternie zbudowanej klasy.

Podsumowując, w książce tej są przedstawione trzy tematy techniczne – każdy na swój sposób ważny. Po pierwsze, poznajemy zupełnie nowe spojrzenie na siłę i elastyczność szablonów w C++. (Jeżeli po przeczytaniu o listach typów nie opadnie Ci szczęka, to masz szczękościsk). Po drugie, poznajemy rozkład popularnych idiomów i wzorców projektowych na niezależne zagadnienia. Dla projektantów szablonów i implementatorów wzorców są to niezmiernie istotne informacje, ale darmo szukać ich w większości literatury z tej dziedziny. Po trzecie, kod źródłowy Loki (biblioteki szablonów opisanej w tej książce) jest dostępny za darmo, a zatem każdy może przyjrzeć się implementacjom odpowiadającym omawianym idiomom i wzorcom. Ten kod jest świetnym testem jakości obsługi szablonów przez kompilator, ale przede wszystkim jest nieoceniony jako punkt wyjścia przy projektowaniu własnych szablonów. Oczywiście, nic też nie stoi na przeszkodzie, by po prostu używać tego kodu choćby od zaraz. Wiem, że Andrei chciałby, by korzystano z owoców jego pracy.

Wiedza na temat szablonów języka C++ zmienia się obecnie równie szybko, jak w roku 1995, kiedy zdecydowałem się o nich nie pisać. Jeśli to tempo się utrzyma, być może *nigdy* mi się nie uda. Na szczęście są ludzie odważniejsi niż ja. Andrei jest pionierem. Z jego książki na pewno dużo się nauczycie. Tak jak ja.

Scott Meyers
Wrzesień 2000