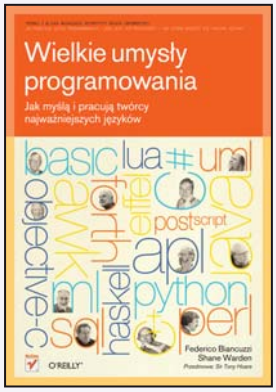
Wielkie umysły programowania. Jak myślą i pracują twórcy najważniejszych języków



Autorzy: Federico Biancuzzi, Shane Warden
Tłumaczenie: Radosław Meryk
ISBN: 978-83-246-2537-6
Tytuł oryginału: Masterminds of Programming: Conversations with the Creators of Major Programming Languages
Format: 168×237, stron: 584

Poznaj z bliska największe autorytety świata informatyki!

- Jak powstają języki programowania?
- Jaka jest ich przyszłość?
- Jak szybko nauczyć się takiego języka?

Droga od pomysłu do gotowej aplikacji jest długa i kręta. Najprawdopodobniej jednym z najdłuższych jej odcinków jest ten poświęcony na programowanie. Sztab ludzi, wiele języków programowania, technologii i narzędzi. Dzięki świetnej znajomości tych narzędzi powstają coraz nowsze, bardziej niezawodne aplikacje. Ale skąd biorą się języki programowania? Jak powstają i kto za tym stoi?

Na półce księgarni znajdziesz tysiące książek poświęconych językom programowania – i tylko tą jedną, która odpowiada na pytanie, co było na początku. Książka stanowi zbiór wywiadów z twórcami najbardziej znanych i najpopularniejszych języków. W trakcie pasjonującej lektury dowiesz się, co kierowało ludźmi, którzy postanowili stworzyć nowy język programowania, jakie mieli problemy, jak oceniają swoje dzieła z perspektywy czasu i jaką wróżą im przyszłość. Lektura tego tomu to niezwykła podróż przez historię informatyki w niesamowitym wydaniu.

W książce znajdziesz wywiady z autorami takich języków, jak:

- C++
- Python
- APL
- Forth
- BASIC
- AWK
- Lua
- Haskell
- ML
- SQL
- Java
- C#
- Perl

Inspirująca i pouczająca podróż przez historię informatyki!

SPIS TREŚCI

Słowo wstępne

Projektowanie języków programowania to wspaniały temat. Istnieje bardzo wielu programistów, którzy uważają, że potrafią zaprojektować lepszy język programowania od tego, którego aktualnie używają. Istnieje również bardzo wielu naukowców, którzy wierzą, że potrafią zaprojektować lepszy język programowania od jakiegokolwiek innego języka będącego w użyciu. Ich osądy są często uzasadnione, ale tylko kilka takich projektów opuściło dolną szufladę projektanta. O takich językach Czytelnik nie znajdzie informacji w tej książce.

Projektowanie języków programowania to poważny biznes. Niewielkie błędy w projekcie języka mogą być przyczyną dużych błędów w ostatecznym programie, a nawet niewielkie błędy w programach mogą mieć poważne konsekwencje i wiązać się z bardzo dużymi kosztami. Wrażliwe punkty powszechnie wykorzystywanego oprogramowania często umożliwiały ataki za pośrednictwem złośliwego oprogramowania. Czasami powodowało to w gospodarce światowej straty rzędu wielu miliardów dolarów. Bezpieczeństwo i zabezpieczenia języków programowania to motyw, który bardzo często pojawia się w niniejszej książce.

Projektowanie języków programowania to nieprzewidywalna przygoda. Języki projektowane pod kątem tworzenia uniwersalnych aplikacji, nawet jeśli są wspierane i sponsorowane przez potężne organizacje, czasami kończą na rynku niszowym.

Z drugiej strony języki projektowane z myślą o ograniczonych lub lokalnych zastosowaniach czasami zyskują liczną klientelę — także w środowiskach i aplikacjach, o których ich projektanci nigdy nawet nie marzyli. Niniejsza książka koncentruje się na językach tego drugiego typu.

Języki, które odniosły sukces, mają jedną istotną cechę: każdy z nich powstał w głowie jednego człowieka lub jest efektem pracy małej grupy podobnie myślących entuzjastów. Projektanci tych języków to mistrzowie programowania. Mają doświadczenie, wizję, energię, wytrwałość oraz prawdziwy geniusz. Cechy te pozwalają im przeprowadzić język od wstępnej implementacji, poprzez ewolucję wynikającą z doświadczeń, aż do standaryzacji będącej efektem wykorzystywania (standardy de facto) oraz przeprowadzanej oficjalnie (przez komitety standaryzacyjne).

W niniejszej książce Czytelnik spotka się z wieloma geniuszami. Z każdym z nich został przeprowadzony obszerny wywiad na temat historii języka oraz czynników, które wpłynęły na jego sukces. Wywiady te potwierdzają istotną rolę dobrych decyzji połączonych ze szczęściem. Publikacja słów wypowiadanych w wywiadzie pozwoli Czytelnikowi ocenić osobowość oraz motywacje projektantów. Jest to temat równie fascynujący jak sam projekt języka.

— Sir Tony Hoare

Sir Tony Hoare, zdobywca nagrody Turinga stowarzyszenia ACM oraz nagrody Kyoto Award, od 50 lat jest liderem badań nad algorytmami komputerowymi oraz językami programowania. W swoim pierwszym akademickim artykule, który został opublikowany w 1969 roku, opisał ideę dowodzenia poprawności programów i zasugerował, aby celem projektu języka programowania było ułatwienie pisania poprawnych programów. Jest zachwycony tym, że idea ta została stopniowo zaakceptowana przez projektantów języków programowania.

Przedmowa

Pisanie oprogramowania nie należy do łatwych zajęć. A już na pewno nie jest łatwe pisanie programów, które spełniają wymagania testów, czasu oraz różnych środowisk. W ciągu ostatnich pięciu dekad w branży inżynierii oprogramowania nie tylko czyniono wysiłki, aby pisanie oprogramowania stawało się coraz łatwiejsze, ale także projektowano języki pod tym kątem. Dlaczego jednak pisanie oprogramowania w ogóle jest trudne?

W większości książek i artykułów zajmujących się tym problemem mówi się o architekturze, wymaganiach oraz podobnych zagadnieniach związanych z *oprogramowaniem*. A co, jeśli cała trudność polega na *pisaniu*? Mówiąc inaczej, zastanówmy się, jakby wyglądał problem, gdyby programiści postrzegali swoją pracę bardziej w kategoriach komunikacji — *języka* — a mniej w kategoriach inżynierii.

Dzieci uczą się mówić przez pierwsze lata życia, natomiast czytania i pisania zaczynają się uczyć dopiero wtedy, gdy skończą pięć, sześć lat. Nie znam żadnego wielkiego autora, który nauczył się czytać i pisać jako dorosły. Czy znacie jakiegoś programistę, który nauczył się programowania w zaawansowanym wieku?

Jeśli dzieci znacznie łatwiej uczą się obcych języków niż dorośli, to co powiedzieć o uczeniu się programowania — czynności wymagającej poznawania nowego języka?

Wyobraźmy sobie, że uczymy się obcego języka i nie znamy nazwy jakiegoś przedmiotu. Potrafimy opisać go słowami, które znamy, z nadzieją, że nasz rozmówca zrozumie, co mamy na myśli. Czy to nie jest to samo, co robimy codziennie, pisząc programy? Opisujemy obiekt, który mamy na myśli, za pomocą języka programowania w nadziei, że nasz opis będzie wystarczająco czytelny dla kompilatora bądź interpretera. Jeśli coś nie zadziała, jeszcze raz przywołujemy obraz obiektu i próbujemy zrozumieć, co pominęliśmy lub co opisaliśmy niewystarczająco dokładnie.

Świadomy tych pytań postanowiłem przeprowadzić szereg badań na temat tego, po co tworzy się języki programowania, w jaki sposób technicznie się je opracowuje, jak się ich naucza, jak są przyswajane oraz jak rozwijają się z biegiem lat.

Shane i ja mieliśmy honor porozmawiania z 27 wielkimi projektantami języków. Osoby te przeprowadziły nas przez swoją pracę. Staraliśmy się zebrać ich wiedzę i doświadczenie, a następnie przedstawić je Czytelnikowi.

W książce *Masterminds of Programming* Czytelnik zapozna się z pewnymi schematami myślenia i czynnościami wymaganymi do stworzenia sprawnego języka programowania. Dowie się, co wpływa na to, że język staje się popularny, oraz zapozna ze sposobami rozwiązywania bieżących problemów, z jakimi spotykają się programiści używający tego języka. Jeśli zatem Czytelnik chce się dowiedzieć, w jaki sposób projektować języki programowania zdolne do osiągnięcia sukcesu, ta książka z pewnością będzie mu pomocna.

Osoby poszukujące inspiracji dotyczących oprogramowania i języków programowania będą potrzebowały kolorowego markera do zaznaczania interesujących fragmentów. Obiecuję, że na kartach niniejszej książki znajdą wiele informacji godnych zaznaczenia.

— Federico Biancuzzi

Organizacja materiału

Rozdziały w niniejszej książce uporządkowano w taki sposób, aby przekazywać Czytelnikom różne poglądy i prowokować. Polecamy smakować poszczególne wywiady i często do nich powracać.

Rozdział 1., „C++”, wywiad z Bjarne’em Stroustrupem.

Rozdział 2., „Python”, wywiad z Guidem van Rossumem.

Rozdział 3., „APL”, wywiad z Adinem D. Falkoffem.

Rozdział 4., „Forth”, wywiad z Charlesem H. Moore’em.

Rozdział 5., „BASIC”, wywiad z Thomasem E. Kurtzem.

Rozdział 6., „AWK”, wywiad z Alfredem Aho, Peterem Weinbergerem i Brianem Kernighanem.

Rozdział 7., „Lua”, wywiad z Luizem Henrique de Figueiredo i Robertem Ierusalimschym.

Rozdział 8., „Haskell”, wywiad z Simonem Peytonem Jonesem, Paulem Hudakiem, Philipem Wadlerem i Johnem Hughesem.

Rozdział 9., „ML”, wywiad z Robinem Milnerem.

Rozdział 10., „SQL”, wywiad z Donem Chamberlinem.

Rozdział 11., „Objective-C”, wywiad z Tomem Love’em i Bradem Coksem.

Rozdział 12., „Java”, wywiad z Jamesem Goslingiem.

Rozdział 13., „C#”, wywiad z Andersem Hejlsbergiem.

Rozdział 14., „UML”, wywiad z Ivarem Jacobsonem, Jamesem Rumbaughem i Gradym Boochem.

Rozdział 15., „Perl”, wywiad z Larrym Wallem.

Rozdział 16., „PostScript”, wywiad z Charlesem Geschkem i Johnem Warnockiem.

Rozdział 17., „Eiffel”, wywiad z Bertrandem Meyerem.

W dodatku „Współtwórcy” zamieszczono biografie wszystkich rozmówców.

Konwencje stosowane w książce

W niniejszej książce zastosowano następujące konwencje typograficzne:

Kursywa
: Oznacza nowe pojęcia, adresy URL, nazwy plików i narzędzi.

`Czcionka o stałej szerokości`
: Oznacza zawartość plików komputerowych oraz — ogólnie rzecz biorąc — wszystkiego, co można znaleźć w programach.

ROZDZIAŁ PIERWSZY

C++

C++ zajmuje interesujące miejsce wśród języków: jest zbudowany na bazie języka C, implementuje mechanizmy obiektowe pochodzące z języka Simula, jest ustandaryzowany przez ISO oraz zaprojektowany według dwóch idei: „nie płacisz za to, czego nie używasz" oraz „typy definiowane przez użytkownika powinny być obsługiwane na równi z wbudowanymi". Chociaż język ten został spopularyzowany w latach osiemdziesiątych i dziewięćdziesiątych jako narzędzie programowania obiektowego używane do tworzenia programów z graficznym interfejsem użytkownika (GUI), to jedną z jego największych zasług na polu rozwoju oprogramowania są techniki programowania generycznego udostępnione w bibliotece STL (ang. *Standard Template Library*). Próbowano zastąpić język C++ nowszymi językami, takimi jak Java czy C#, tymczasem do kolejnej wersji standardu C++ dodano nowe, długo oczekiwane własności. Twórcą języka C++ i jednym z jego gorących orędowników jest Bjarne Stroustrup.

Decyzje projektowe

Dlaczego zdecydował się pan rozszerzyć istniejący język, zamiast stworzyć nowy?

Bjarne Stroustrup: Kiedy zaczynałem — w 1979 roku — moim celem była pomoc programistom w budowaniu systemów. W dalszym ciągu przyświeca mi taki cel. Aby język programowania stanowił prawdziwą pomoc w rozwiązywaniu problemów, a nie był tylko akademickim ćwiczeniem, musi być kompletny w zakresie narzędzi do tworzenia aplikacji. A zatem potrzebny jest prosty język do rozwiązywania problemów. Problemy, które starałem się rozwiązać, dotyczyły projektowania systemów operacyjnych, obsługi sieci i symulacji. Ja i moi koledzy potrzebowaliśmy języka, za pomocą którego można by prezentować organizację programu, tak jak to można robić w języku Simula (potrzebne nam zatem były techniki obiektowe). Jednocześnie potrzebowaliśmy takiego języka, za pomocą którego można pisać wydajny kod niskopoziomowy — tak jak w języku C. W 1979 roku nie było języka, który posiadałby obie te cechy, a przynajmniej ja takiego nie znałem. Właściwie nie chciałem projektować nowego języka programowania. Chciałem tylko pomóc w rozwiązaniu kilku problemów.

Bazowanie na istniejącym języku programowania ma bowiem sens. Z języka bazowego bierzemy podstawową strukturę składni i semantyki, wykorzystujemy z niego użyteczne biblioteki i stajemy się częścią kultury. Gdybym nie wykorzystał języka C, oparłbym język C++ na jakimś innym języku. Dlaczego C? Miałem Dennisa Ritchiego, Briana Kernighana i innych wielkich twórców Uniksa w odległości piętra lub pokoju w Computer Science Research Center — Bell Labs, a zatem pytanie to może wydawać się bezzasadne. Ja jednak potraktowałem je zupełnie serio.

W szczególności system typów języka C był nieformalny i niezbyt dobrze przestrzegany (jak powiedział Dennis Ritchie: „C jest językiem o ścisłej typizacji (ang. *strongly typed*) i słabo kontrolowanych typach"). Cecha „słabo kontrolowane" mnie martwiła, zresztą do dziś sprawia ona programistom języka C++ wiele problemów. Język C nie był wtedy powszechnie używany, tak jak to się dzieje dziś. Oparcie języka C++ na języku C było wyrazem wiary w model przetwarzania będący podstawą języka C (cecha „ścisła typizacja") oraz wyrazem zaufania do moich kolegów. Wyboru dokonałem na podstawie wiedzy na temat większości języków wyższego poziomu wykorzystywanych w tamtych czasach do tworzenia systemów (znałem je zarówno jako użytkownik, jak i praktykujący programista). Warto pamiętać, że był to czas, w którym większość prac „blisko sprzętu" oraz wymagających dobrej wydajności było wykonywanych w asemblerze. Powstanie systemu Unix stanowiło poważny przełom pod wieloma względami — jednym z nich było użycie języka C nawet do najbardziej wymagających zadań programowania systemowego.

W związku z tym wybrałem wykorzystywany w języku C prosty model maszyny, który uznałem za bardziej wartościową cechę od lepszej kontroli typów występującej w innych językach. Tym, czego naprawdę potrzebowałem jako ramy (ang. *framework*)

do tworzenia programów, były klasy dostępne w Simuli. Dlatego przeniosłem je na model pamięci i przetwarzania właściwy językowi C. W efekcie powstało niezwykle ekspresywne i elastyczne narzędzie, które pod względem szybkości działania mogło rywalizować z asemblerem i nie wymagało stosowania rozbudowanego środowiska wykonawczego.

Dlaczego zdecydował się pan na wspieranie wielu paradygmatów?

Bjarne: Ponieważ kombinacja stylów programowania często prowadzi do stworzenia lepszego kodu, przy czym „lepszy" oznacza kod, który w bardziej bezpośredni sposób odzwierciedla projekt, działa szybciej, jest łatwiejszy w zarządzaniu itp. Dążąc do tworzenia lepszego kodu, ludzie albo próbują zdefiniować swój ulubiony styl programowania zawierający wszystkie użyteczne konstrukcje (na przykład „programowanie generyczne jest po prostu formą programowania obiektowego"), albo wyłączają pewne obszary aplikacji (na przykład zakładają, że „każdy ma maszynę z zegarem 1GHz i pamięcią 1 GB").

Język Java koncentruje się wyłącznie na programowaniu obiektowym. Czy to powoduje, że kod Javy jest w pewnych przypadkach — tam, gdzie w języku C++ można skorzystać z programowania generycznego — bardziej złożony?

Bjarne: No cóż, projektanci Javy — a może raczej osoby zajmujące się marketingiem — promowali programowanie obiektowe do granic absurdu. Kiedy Java pojawiła się po raz pierwszy, a jej twórcy podkreślali czystość i prostotę tego kodu, przewidziałem, że w przypadku sukcesu Java znacznie się rozrośnie i wzrośnie jej złożoność. Tak też się stało.

I tak wykorzystanie rzutowania do konwersji z typu `Object` podczas pobierania wartości z kontenera (na przykład `(Apple)c.get(i)`) jest absurdalną konsekwencją braku możliwości wyspecyfikowania typu, jaki powinny mieć obiekty w kontenerze. To sposób rozwlekły i nieefektywny. Teraz Java ma typy generyczne, zatem jest tylko nieco wolniejsza. Innymi przykładami zwiększonej złożoności języka (w celu pomocy programistom) są typy wyliczeniowe (enumeracje), odbicia oraz klasy wewnętrzne.

Faktem jest, że złożoność musi się gdzieś pojawić. Jeśli nie ma jej w definicji języka, to będzie w niezliczonych aplikacjach i bibliotekach. Na podobnej zasadzie obsesja, by w Javie każdy algorytm (operację) implementować w postaci klasy, prowadzi do absurdów — na przykład klas bez danych, które składają się w całości z funkcji statycznych. Istnieją powody, dla których w matematyce stosuje się zapisy `f(x)` i `f(x,y)` zamiast `x.f()`, `x.f(y)` czy też `(x,y).f()` — zapis `(x,y).f()` jest próbą wyrażenia idei „prawdziwie obiektowego sposobu" przedstawiania dwóch argumentów oraz uniknięcia asymetrii właściwej dla zapisu `x.f(y)`.

Dzięki połączeniu abstrakcji danych i technik programowania generycznego język C++ rozwiązuje wiele problemów dotyczących logiki i notacji wynikających z podejścia obiektowego. Klasycznym przykładem jest zapis `vector<T>`, w którym `T` może być

dowolnym typem możliwym do kopiowania — włącznie z typami wbudowanymi, wskaźnikami na hierarchie obiektowe oraz typami definiowanymi przez użytkowników — na przykład ciągami znaków i liczbami zespolonymi. Wszystko to osiągnięto bez dodatkowych kosztów w fazie działania, nakładania ograniczeń na rozmieszczenie danych w pamięci czy też stosowania specjalnych reguł dotyczących standardowych komponentów bibliotecznych. Innym przykładem, który nie pasuje do klasycznego modelu pojedynczej dyspozycji (ang. *single dispatch*) charakterystycznego dla programowania obiektowego, jest operacja wymagająca dostępu do dwóch klas, na przykład `operator*(Macierz,Wektor)`. Operacja ta naturalnie nie może być metodą żadnej z klas.

Jedną z podstawowych różnic pomiędzy językami C++ i Javą jest sposób implementacji wskaźników. W pewnym sensie można powiedzieć, że język Java nie posiada prawdziwych wskaźników. Jakie różnice występują pomiędzy tymi dwoma podejściami?

Bjarne: Cóż, w języku Java oczywiście są wskaźniki. W rzeczywistości niemal wszystko w Javie to niejawne wskaźniki. Tyle że nazwano je *referencjami*. Niejawność wskaźników ma zarówno zalety, jak i wady. Podobnie istnieją zarówno zalety, jak i wady rzeczywistego występowania lokalnych obiektów (tak jak w C++).

Podejście zastosowane w C++, polegające na wspieraniu zmiennych lokalnych alokowanych na stosie oraz rzeczywistych zmiennych członkowskich dowolnego typu, gwarantuje wygodną jednolitą semantykę, kompaktowy układ i minimalne koszty dostępu oraz stanowi podstawę dla obsługi ogólnego zarządzania zasobami w języku C++. To jest bardzo ważne, a wszechobecne i niejawne stosowanie wskaźników w Javie (zwanych tam referencjami) zamyka drzwi do wszystkich tych mechanizmów.

Przeanalizujmy sprawy związane z rozmieszczeniem danych w pamięci. W języku C++ konstrukcja `vector<complex>(10)` jest reprezentowana jako uchwyt do tablicy 10 liczb zespolonych w wolnej pamięci. W sumie zajmuje ona 25 słów: 3 słowa na wektor plus 20 słów na liczby zespolone oraz dodatkowo 2 słowa na nagłówek tablicy w wolnej pamięci (stercie). Odpowiednik w Javie (dla zdefiniowanego przez użytkownika kontenera zawierającego obiekty typów niestandardowych) zająłby 56 słów: 1 na referencję do kontenera plus 3 na kontener plus 10 na referencje do obiektów plus 20 na obiekty plus 24 na przechowywane w wolnej pamięci nagłówki 12 niezależnie alokowanych obiektów. Oczywiście te liczby są przybliżone, ponieważ koszty obsługi wolnej pamięci (sterty) są zdefiniowane w implementacji obu języków. Konkluzja jest jednak oczywista: przez wszechobecne i niejawne stosowanie referencji w Javie uproszczono model programowania i implementację mechanizmu odśmiecania (ang. *garbage collector*), ale jednocześnie dramatycznie zwiększono koszty pamięciowe, podwyższono koszty dostępu do pamięci (z powodu większej liczby pośrednich operacji dostępu) oraz proporcjonalnie zwiększono koszty alokacji.

Tym, czego Java nie ma — i dobrze dla Javy — jest możliwość nieprawidłowego używania wskaźników w działaniach arytmetycznych na wskaźnikach. Jednak dobrze napisanego kodu w C++ ten problem i tak nie dotyka: zamiast wykonywać działania na wskaźnikach, programiści używają bardziej wysokopoziomowych abstrakcji, takich jak strumienie wejścia-wyjścia, kontenery, lub stosują zaawansowane algorytmy. W zasadzie wszystkie tablice — to samo dotyczy większości wskaźników — pozostają ukryte głęboko w implementacjach, czyli w miejscach, których większość programistów nie musi widzieć. Niestety, istnieje również wiele źle napisanego i niepotrzebnie niskopoziomowego kodu w języku C++.

Jest jednak istotne miejsce, w którym wskaźniki i działania na nich są wielką zaletą: bezpośrednie i wydajne prezentowanie struktur danych. Referencje Javy nie dają takich samych możliwości — na przykład w Javie nie można przedstawić operacji wymiany. Inny przykład to użycie wskaźników do niskopoziomowego bezpośredniego dostępu do pamięci (fizycznej). W każdym systemie trzeba to robić w jakimś języku i często tym językiem jest C++.

Z występowaniem wskaźników (i tablic w stylu języka C) wiąże się oczywiście ryzyko niewłaściwego wykorzystania: przepełnienia bufora, użycia wskaźników w odniesieniu do usuniętej pamięci, wystąpienia wskaźników niezainicjowanych itp. Jednak w dobrze napisanym kodzie C++ nie stanowi to wielkiego problemu. Problem ten po prostu nie występuje w przypadku wskaźników i tablic wykorzystywanych wewnątrz abstrakcji (na przykład `vector`, `string`, `map` itp.). Zarządzanie zasobami z wykorzystaniem zasięgów (ang. *scope*) załatwia większość potrzeb. Pozostałe problemy można rozwiązać dzięki zastosowaniu inteligentnych wskaźników i specjalizowanych uchwytów. Programistom z doświadczeniem głównie w języku C lub C++ starego stylu trudno będzie w to uwierzyć, ale zarządzanie zasobami bazujące na zasięgach to niezwykle mocne narzędzie. Zasięgi definiowane przez użytkownika z odpowiednimi operacjami pozwalają rozwiązywać klasyczne problemy za pomocą mniejszej ilości kodu w porównaniu ze starymi niebezpiecznymi sposobami. Oto na przykład najprostsza postać klasycznego problemu przepełnienia bufora stwarzającego zagrożenie dla bezpieczeństwa:

```
char buf[MAX_BUF];
gets(buf); // Oops!
```

Wystarczy skorzystać ze standardowej biblioteki *string*, a problem zniknie:

```
string s;
cin >> s; // odczyt znaków oddzielanych spacjami
```

Są to oczywiście trywialne przykłady, ale odpowiednie ciągi znaków i kontenery można zbudować w taki sposób, by spełniały właściwie wszystkie potrzeby. Dobrym miejscem do rozpoczęcia poszukiwań jest standardowa biblioteka.

Co pan rozumie pod pojęciami „semantyka wartości" oraz „ogólne zarządzanie zasobami"?

Bjarne: „Semantyka wartości" to termin powszechnie używany w odniesieniu do klas, w których obiekty mają właściwość dającą po skopiowaniu dwie niezależne kopie obiektów (z tą samą wartością).

Na przykład:

```
X x1 = a;
X x2 = x1; // Teraz x1 == x2
x1 = b;    // Zmienia się x1, ale nie x2
           // teraz x1! = x2 (pod warunkiem że X(a)! = X(b))
```

Taka cecha dotyczy oczywiście zwykłych typów numerycznych, jak liczby `int`, `double`, liczby zespolone oraz matematyczne abstrakcje, na przykład wektory. Jest to najbardziej użyteczna własność. C++ obsługuje ją dla typów wbudowanych oraz dla dowolnych typów definiowanych przez użytkownika, dla których chcemy ją mieć. Pod tym względem język C++ różni się od Javy — tu typy wbudowane, takie jak `char` i `int`, posiadają tę własność, ale typy definiowane przez użytkowników jej nie mają i nie mogą mieć. Tak jak w języku Simula, wszystkie typy definiowane przez użytkownika w Javie mają semantykę referencji. W C++ programista może zastosować dowolną spośród tych dwóch semantyk, zgodnie z wymaganiami typu. Język C# naśladuje język C++ (choć nie do końca) w obsłudze typów użytkownika z semantyką wartości.

Termin „ogólne zarządzanie zasobami" odnosi się do popularnej techniki posiadania zasobu przez obiekt (na przykład uchwytu do pliku lub blokady). Jeśli ten obiekt jest zmienną zdefiniowaną w pewnym zasięgu, to czas życia zmiennej wprowadza maksymalny limit czasu, przez jaki utrzymywany jest zasób. Zazwyczaj konstruktor alokuje zasób, a destruktor go zwalnia. Takie działanie, często określane terminem RAII (od ang. *Resource Acquisition Is Initialization* — dosł. zdobycie zasobu to inicjalizacja), doskonale się integruje z obsługą błędów z wykorzystaniem wyjątków. Oczywiście nie każdy zasób można obsłużyć w ten sposób, ale w wielu przypadkach jest to możliwe. Wtedy zarządzanie zasobami staje się niejawne i wydajne.

Wydaje się, że zasada „blisko sprzętu" była wiodącą regułą podczas projektowania języka C++. Czy można powiedzieć, że język C++ został zaprojektowany w układzie dół-góra, podczas gdy wiele języków programowania jest projektowanych w układzie góra-dół, w tym sensie, że dostarczają one abstrakcyjnie racjonalnych konstrukcji i zmuszają kompilator do tego, by dopasowywał te konstrukcje do dostępnego środowiska przetwarzania?

Bjarne: Uważam, że określenia góra-dół i dół-góra to złe sposoby charakteryzowania tych decyzji projektowych. W kontekście języka C++ i innych języków „blisko sprzętu" oznacza, że jest przyjęty model obliczeń danego komputera — sekwencje obiektów w pamięci oraz operacje definiowane na obiektach o stałym rozmiarze zamiast jakiejś

abstrakcji matematycznej. Tak jest zarówno w przypadku języka C++, jak i Javy, ale w odniesieniu do języków funkcyjnych jest inaczej. C++ różni się od Javy tym, że pod spodem jest maszyna fizyczna, a nie maszyna abstrakcyjna.

Prawdziwy problem polega na tym, jak przejść od ludzkiego sposobu postrzegania problemów i rozwiązań do ograniczonego świata maszyny. Można zignorować ludzkie rozterki i stworzyć kod maszynowy (lub gloryfikowany kod maszynowy w postaci złego kodu w języku C). Można też zignorować rozterki maszyny i uzyskać piękne abstrakcje, które pozwalają na wykonywanie dowolnych operacji olbrzymim kosztem i (lub) bez ograniczeń zdroworozsądkowych. Język C++ to próba uzyskania bezpośredniego dostępu do sprzętu (na przykład za pośrednictwem wskaźników i tablic) tam, gdzie jest taka potrzeba, z jednoczesnym dostarczeniem rozbudowanych mechanizmów abstrakcyjnych pozwalających na wyrażanie idei wysokopoziomowych (na przykład hierarchii klas i szablonów).

W związku z takimi założeniami podczas tworzenia języka C++ i jego bibliotek zwracano baczną uwagę na wydajność kodu wynikowego i zarządzanie pamięcią. Te aspekty przenikają zarówno podstawowe mechanizmy języka, jak i mechanizmy abstrakcyjne w sposób wyróżniający język C++ spośród innych języków.

Używanie języka

W jaki sposób debuguje pan swój kod? Czy ma pan jakieś wskazówki dla programistów C++?

Bjarne: Przez wnikliwą analizę. Studiuję program i oglądam go mniej lub bardziej systematycznie tak długo, aż mam wystarczającą wiedzę do tego, by w inteligentny sposób odgadnąć miejsce wystąpienia błędu.

Testowanie to zupełnie inna sprawa. Podobnie jak odpowiedni projekt zmierzający do zminimalizowania liczby błędów. Bardzo nie lubię debugowania i robię wszystko, by go uniknąć. Jeśli jestem programistą fragmentu oprogramowania, buduję go na bazie interfejsów i niezmienników, dlatego trudno mi uzyskać kod z dużą liczbą błędów, który nie daje się skompilować i uruchomić. Następnie bardzo się staram, aby kod można było przetestować. Testowanie jest systematycznym poszukiwaniem błędów. Trudno testować w sposób systematyczny systemy, które mają nieprawidłową strukturę, dlatego jeszcze raz zalecam dbałość o czytelną strukturę kodu. Testowanie można zautomatyzować i wykonywać wielokrotnie, podczas gdy w przypadku debugowania nie da się tego uzyskać. Wykorzystanie stada gołębi, które losowo siadają na ekranie, po to, by zobaczyć, czy uda im się złamać aplikację okienkową, nie jest sposobem na zapewnienie wysokiej jakości systemów.

Moja rada? Trudno dawać uniwersalne rady, ponieważ najlepsze techniki często zależą od tego, co jest wykonalne w danym systemie oraz środowisku projektowym. Jeśli jednak mogę coś radzić: należy zidentyfikować kluczowe interfejsy, które można

systematycznie testować, a następnie napisać skrypty testowe, które to robią. Należy stosować automatyzację tam, gdzie to możliwe, i często uruchamiać takie automatyczne testy. Warto też często wykonywać testy regresji. Należy dążyć to tego, by każdy punkt wejścia do systemu i wyjścia z systemu był systematycznie przetestowany. System powinien być złożony z komponentów o wysokiej jakości: monolityczne programy są niepotrzebnie skomplikowane, przez co są trudne do zrozumienia i przetestowania.

Na jakim poziomie konieczna jest poprawa bezpieczeństwa oprogramowania?

Bjarne: Po pierwsze, bezpieczeństwo jest problemem systemowym. Żadne lokalne lub częściowe rozwiązanie nie zapewni sukcesu. Należy pamiętać, że nawet gdyby cały kod był perfekcyjny, to i tak udałoby się uzyskać dostęp do zapisanych sekretów komuś, kto ukradłby nasz komputer lub nośnik z kopią zapasową. Po drugie, bezpieczeństwo jest kompromisem pomiędzy kosztami a korzyściami: doskonałe bezpieczeństwo prawdopodobnie jest poza zasięgiem większości z nas. Można jednak zabezpieczyć system na tyle mocno, aby źli chłopcy woleli poświęcić swój czas na próby włamania do innego systemu. Sam dążę do tego, by nie przechowywać ważnych sekretów online, a problemy dotyczące poważnych zabezpieczeń pozostawiam ekspertom.

Co jednak z językami programowania i technikami programowania? Istnieje niebezpieczna tendencja do zakładania, że każda linijka kodu musi być bezpieczna (cokolwiek by to oznaczało). Niektórzy przyjmują nawet, że w ramach tego samego zespołu mogą działać osoby o złych intencjach, które próbują manipulować innymi częściami systemu. To bardzo niebezpieczne podejście, którego efektem jest zaśmiecanie kodu wieloma testami chroniącymi przed wyimaginowanymi zagrożeniami. W ten sposób kod staje się brzydki, wielki i powolny. Jeśli kod jest brzydki, to łatwo chowają się w nim błędy. Jeśli jest wielki, to z pewnością nie będzie dobrze przetestowany, a powolność zachęca do używania skrótów i złych praktyk. Te ostatnie należą do najczęstszych źródeł luk w zabezpieczeniach.

Uważam, że jedynym trwałym rozwiązaniem problemów zabezpieczeń jest prosty model bezpieczeństwa stosowany systematycznie przez wysokiej jakości sprzęt i/lub oprogramowanie w odniesieniu do wybranych interfejsów. Musi istnieć obszar za barierą, gdzie można pisać kod prosto, elegancko i wydajnie bez obaw, że losowe fragmenty kodu zniszczą losowe fragmenty innego kodu. Tylko w takim przypadku będziemy mogli skupić się na poprawności, jakości i prawdziwej wydajności. Zakładanie, że każdy może spreparować niezaufane wywołanie zwrotne, wtyczkę (ang. *plug-in*), przeciążoną metodę czy cokolwiek innego, jest po prostu głupie. Musimy odróżnić kod, który jest zabezpieczony przed oszustwami, od kodu, który zabezpiecza się przed sytuacjami nadzwyczajnymi.

Nie sądzę, aby można było stworzyć język programowania, który byłby całkowicie bezpieczny, a jednocześnie przydatny w praktycznych zastosowaniach. Oczywiście wszystko zależy od definicji słów „bezpieczeństwo” i „system”. Być może łatwiej

uzyskać bezpieczeństwo systemów w języku specyficznym dla konkretnej dziedziny. Moją dziedziną zainteresowania jest jednak programowanie systemowe (w bardzo szerokim znaczeniu tego terminu), włącznie z programowaniem systemów wbudowanych. Uważam, że w stosunku do tego, co oferuje C++, można by poprawić bezpieczeństwo typologiczne (ang. *type safety*) i z pewnością będzie ono poprawione, ale to tylko część problemu: bezpieczeństwo typologiczne nie jest tożsame z bezpieczeństwem w ogóle. Osoby programujące w C++, które używają wielu nieopakowanych tablic, operacji rzutowania oraz niestrukturalnych operacji `new` i `delete`, same proszą się o kłopoty. Osoby te pozostały na etapie stylu programowania z lat osiemdziesiątych. Aby dobrze korzystać z C++, trzeba stosować styl programowania, w którym jest jak najmniej naruszeń bezpieczeństwa typologicznego, a zasoby (łącznie z pamięcią) są zarządzane w prosty i systematyczny sposób.

Czy poleciłby pan język C++ do tworzenia niektórych systemów, na przykład oprogramowania systemowego i aplikacji wbudowanych, mimo że praktycy niechętnie wykorzystują go w tych zastosowaniach?

Bjarne: Oczywiście, że poleciłbym C++, a poza tym nie wszyscy są niechętni temu językowi. Właściwie nie zauważyłem zbytniej niechęci wykorzystywania C++ w tych obszarach, poza naturalną niechęcią do wypróbowywania czegoś nowego w instytucjach o ugruntowanej pozycji. Powiedziałbym nawet, że obserwuję stały i znaczący wzrost popularności języka C++. Dla przykładu — pomagałem pisać wskazówki kodowania dla kluczowego oprogramowania myśliwca Joint Strike Fighter (JSF) firmy Lockheed Martin. To samolot, którego oprogramowanie jest napisane w całości w C++. Czytelnicy najprawdopodobniej nie znają się na wojskowych samolotach, ale w sposobach używania języka C++ nie ma nic szczególnie militarnego. W niespełna rok z moich stron internetowych ściągnięto grubo ponad 100 000 kopii reguł kodowania JSF++. Z mojej wiedzy wynika, że informacje te interesowały głównie programistów niewojskowych systemów wbudowanych.

C++ jest używany do projektowania systemów wbudowanych od 1984 roku. W tym języku stworzono wiele przydatnych gadżetów, a liczba zastosowań języka dynamicznie wzrasta. Przykładami mogą być telefony komórkowe z systemami Symbian lub Motorola, urządzenia iPod oraz systemy GPS. Osobiście szczególnie podoba mi się zastosowanie C++ w lądownikach Mars Rovers, w których język C++ wykorzystano do analizy scen i autonomicznych podsystemów kontroli jazdy, większości naziemnych systemów komunikacyjnych oraz przetwarzania obrazów.

Osoby przekonane o tym, że język C musi być bardziej wydajny niż C++, odsyłam do mojego artykułu „Learning Standard C++ as a New Language”[1] [*C/C++ Users Journal*, maj 1999], w którym zamieściłem krótki opis filozofii projektu i zaprezentowałem wyniki prostych eksperymentów. Techniczny raport na temat

[1] *http://www.open-std.org/JTC1/sc22/wg21/docs/TR18015.pdf*.

wydajności opublikował także komitet standaryzacyjny ISO zajmujący się językiem C++. W raporcie podjęto kwestie wielu problemów i mitów związanych z takimi zastosowaniami C++, w których wydajność ma znaczenie. Autorzy artykułu zwrócili szczególną uwagę na problematykę systemów wbudowanych. (Tekst można znaleźć w internecie — wystarczy poszukać frazy *Technical Report on C++ Performance*).

Jądra systemów Linux lub BSD w dalszym ciągu są pisane w języku C. Dlaczego ich projektanci nie zdecydowali się przejść na C++? Czy istnieje jakiś problem z paradygmatem obiektowym?

Bjarne: W większości przypadków powodem jest konserwatyzm i siła inercji. Poza tym kompilator GCC wolno dojrzewał. Wydaje się, że niektóre osoby ze społeczności języka C wykazują niemal celową ignorancję popartą wieloletnim doświadczeniem. Od dziesięcioleci język C++ jest używany do tworzenia systemów operacyjnych, oprogramowania systemowego, a nawet twardych systemów czasu rzeczywistego oraz programów o kluczowym znaczeniu dla bezpieczeństwa. Oto kilka przykładów: Symbian, OS/400 i K42 firmy IBM, BeOS oraz częściowo Windows. Istnieje również wiele programów *open source* napisanych w C++ (na przykład KDE).

Widzę, że utożsamia pan język C++ z programowaniem obiektowym. C++ nie jest i nigdy nie miał być językiem, w którym stosuje się wyłącznie techniki programowania obiektowego. W 1995 roku napisałem artykuł „Why C++ is not just an Object-Oriented Programming Language". Jest on dostępny w internecie[2]. Ideą języka C++ było — i jest nią nadal — wspieranie wielu stylów programowania (paradygmatów, jeśli woli pan używać trudnych słów) oraz ich kombinacji. W kontekście wysokiej wydajności i bliskości sprzętu oprócz programowania obiektowego najważniejszym spośród innych paradygmatów jest używanie technik programowania generycznego (na jego określenie czasami używa się skrótu GP — od ang. *Generic Programming*). Typowa biblioteka C++ standardu ISO — STL — zawierająca framework dla algorytmów i kontenerów to w większym stopniu techniki GP niż OO (od ang. *Object Oriented*). Programowanie generyczne, w typowym dla języka C++ stylu opartym na intensywnym wykorzystywaniu szablonów, stosuje się powszechnie wszędzie tam, gdzie jest potrzebna zarówno abstrakcja, jak i wydajność.

Nigdy nie spotkałem się z programem, który byłby lepiej napisany w C niż w C++. Nie sądzę, aby taki program w ogóle mógł istnieć. W najgorszym razie można pisać kod C++ w stylu zbliżonym do języka C. Nic nie zmusza programistów do przesadnego wykorzystywania wyjątków, hierarchii klas czy szablonów. Dobry programista wykorzystuje zaawansowane własności tam, gdzie pozwalają one na bardziej bezpośrednie wyrażanie idei i nie zmuszają do ponoszenia zbędnych kosztów.

[2] *http://www.research.att.com/~bs/oopsla.pdf*.

Dlaczego programiści mieliby przenosić kod z języka C do C++? Jakie korzyści mogą wyniknąć z zastosowania C++ jako języka programowania generycznego?

Bjarne: Wydaje mi się, że zakłada pan, jakoby kod najpierw był pisany w języku C, a programista zaczynał pisanie wyłącznie w języku C. W wielu przypadkach programów C++ i programistów C++ (myślę, że już od długiego czasu dotyczy to większości sytuacji) tak nie jest. Niestety podejście polegające na wychodzeniu od języka C zdarza się w wielu kręgach, ale na szczęście nie jest już czymś, co przyjmuje się za pewnik.

Ktoś może przejść z języka C na C++ dlatego, że obsługa stylu programowania, którą realizował wcześniej w języku C, jest lepsza w C++ niż w C. Kontrola typów w języku C++ jest bardziej ścisła (nie można zapomnieć zadeklarować funkcji lub typów jej argumentów), istnieje także bezpieczne typologicznie wsparcie notacji wielu popularnych operacji, takich jak tworzenie obiektów (włącznie z inicjalizacją) czy stałych. Znam programistów, którzy właśnie z tych powodów przeszli na język C++ i są bardzo zadowoleni, że udało im się pozbyć wielu problemów. Zwykle takiemu przejściu towarzyszy adopcja niektórych bibliotek języka C++, czasami obiektowych — na przykład standardowej biblioteki obsługi wektorów, biblioteki obsługi interfejsu GUI oraz niektórych bibliotek specyficznych dla aplikacji.

Użycie niestandardowego typu, na przykład `vector`, `string` czy `complex`, nie wymaga zmiany paradygmatu. Można po prostu, jeśli się tego chce, korzystać z nich tak jak z typów wbudowanych. Czy ktoś, kto stosuje konstrukcję `std::vector`, używa technik OO? Powiedziałbym, że nie. Czy ktoś, kto używa mechanizmów obsługi interfejsu GUI języka C++, ale nie dodaje nowych funkcji, używa technik OO? Skłaniam się do odpowiedzi twierdzącej, ponieważ używanie ich zwykle wymaga od użytkowników zrozumienia i posługiwania się mechanizmami dziedziczenia.

Wykorzystanie C++ jako języka programowania generycznego daje programiście dostęp do gotowych standardowych kontenerów i algorytmów (w ramach standardowej biblioteki). Jest to wielkie udogodnienie w przypadku wielu zastosowań i wejście na wyższy poziom abstrakcji w porównaniu z językiem C. Poza tym programiści mogą korzystać z bibliotek, na przykład *Boost*, oraz stosować niektóre techniki programowania funkcyjnego właściwe programowaniu generycznemu.

Myślę jednak, że pytanie jest trochę mylące. Nie chcę przedstawiać języka C++ jako języka OO lub języka GP. Chciałbym raczej pokazać, że jest to język dający dostęp do własności takich jak:

- programowanie w stylu języka C,
- abstrakcja danych,
- programowanie obiektowe,
- programowanie generyczne.

Co najważniejsze, język ten obsługuje wiele stylów programowania (programowanie wieloparadygmatowe — jeśli ktoś woli) i jest ukierunkowany na programowanie systemowe.

Programowanie obiektowe i współbieżność

Przeciętna złożoność i rozmiary (licząc w wierszach kodu) oprogramowania wydają się rozrastać z roku na rok. Czy techniki programowania obiektowego dobrze komponują się w tej rzeczywistości, czy też tylko bardziej wszystko komplikują? Mam wrażenie, że dążenie do tworzenia obiektów wielokrotnego użytku komplikuje wiele rzeczy, a poza tym podwaja pracochłonność. Po pierwsze, trzeba zaprojektować narzędzie dające się wielokrotnie wykorzystać. Później, kiedy zajdzie konieczność wprowadzenia zmian, trzeba będzie napisać coś, co dokładnie wypełni lukę pozostawioną przez poprzednią wersję, a to oznacza ograniczenia dla rozwiązania.

Bjarne: To dobry opis poważnego problemu. OO to zbiór technik gwarantujących duże możliwości. Mogą one być pomocne, ale żeby były pomocne, muszą być dobrze wykorzystywane i stosowane w odniesieniu do takich problemów, dla których techniki te mają coś do zaoferowania. Zaprojektowanie dobrej klasy bazowej (interfejsu dla wielu nieznanych wcześniej klas) wymaga umiejętności przewidywania i doświadczenia. Jest to poważny problem podczas tworzenia kodu w całości bazującego na dziedziczeniu z wykorzystaniem interfejsów kontrolowanych statycznie. Skąd projektant klasy bazowej (klasy abstrakcyjnej, interfejsu czy jak to nazwiemy) ma wiedzieć, że uwzględnił wszystko, co jest potrzebne dla wszystkich klas, które w przyszłości będą dziedziczyły z klasy bazowej? Skąd ma wiedzieć, że to, co zaprojektuje, będzie można sensownie zaimplementować we wszystkich klasach, które w przyszłości będą dziedziczyły z jego klasy bazowej? Jak ma się dowiedzieć, że zaprojektowana klasa bazowa nie będzie poważnie kolidowała z czymś, co jest wymagane przez pewne klasy, które w przyszłości będą dziedziczyły z jego klasy bazowej?

Ogólnie rzecz biorąc, nie ma możliwości, by się tego dowiedzieć. W środowisku, w którym możemy wymusić nasz projekt, programiści dostosują się — często poprzez pisanie mało eleganckich obejść. Jeśli nie ma organu koordynującego, powstaje wiele niekompatybilnych interfejsów dla tego samego zestawu funkcji.

Nie można rozwiązać tych problemów w sposób ogólny, ale programowanie generyczne wydaje się być dobrym rozwiązaniem w wielu ważnych obszarach, w których podejście obiektowe się nie sprawdza. Przykładem godnym przytoczenia są proste kontenery: za pomocą hierarchii dziedziczenia nie można dobrze wyrazić pojęcia bycia elementem. Nie można również dobrze wyrazić pojęcia bycia kontenerem. Relacje te mogą jednak dostarczyć skutecznych rozwiązań za pomocą programowania generycznego. Przykładem może być biblioteka ***STL*** (wchodząca w skład standardowej biblioteki C++).

Czy to jest problem specyficzny dla języka C++, czy też w równym stopniu dotyczy on innych języków programowania?

Bjarne: Problem jest wspólny dla wszystkich języków, które bazują na statycznie kontrolowanych interfejsach hierarchii klas. Przykładem są języki C++, Java i C#. Problem ten nie dotyczy języków z dynamiczną kontrolą typów, takich jak Smalltalk czy Python. W języku C++ tę kwestię można rozwiązać za pomocą programowania generycznego. Dobrym przykładem są kontenery C++ i algorytmy należące do standardowej biblioteki. Kluczową cechą języka są w tym przypadku szablony dostarczające model późnej kontroli typów. Jest to odpowiednik mechanizmu dynamicznej kontroli typów, tyle że w fazie kompilacji, a nie wykonywania programu. Wprowadzone ostatnio do języków Java i C# typy generyczne są próbą pójścia w kierunku wyznaczonym przez C++. Według mojej opinii często niesłusznie mówi się o nich jako o usprawnieniu szablonów.

Szczególnie popularną techniką jest refaktoryzacja, która polega na próbie rozwiązania problemu techniką siłową poprzez przeorganizowanie kodu, w przypadku gdy początkowy projekt interfejsu był nieprawidłowy.

Jeśli jest to ogólny problem programowania obiektowego, to jaką mamy pewność, że zalety programowania OO są większe niż wady wynikające ze stosowania tej techniki? Być może problem z trudnością uzyskania dobrego projektu obiektowego jest źródłem wszystkich innych problemów.

Bjarne: Istnienie problemu w niektórych lub nawet w wielu przypadkach nie zmienia faktu, że wiele doskonałych, wydajnych i łatwych do zarządzania systemów napisano w językach obiektowych. Techniki obiektowe należą do podstawowych sposobów projektowania systemów, a interfejsy kontrolowane statycznie oprócz wad mają również zalety.

W dziedzinie wytwarzania oprogramowania nie istnieje jedna recepta na wszystkie problemy. Projektowanie jest trudne pod wieloma względami. Programiści często nie doceniają intelektualnych i praktycznych trudności związanych z tworzeniem rozbudowanych systemów zawierających elementy oprogramowania. Tworzenie takich systemów nie sprowadza się i nigdy nie będzie się sprowadzać do mechanicznego procesu produkcyjnego. Do stworzenia stosunkowo dużego systemu niezbędne są kreatywność, stosowanie zasad inżynierskich oraz ewolucyjne zmiany.

Czy istnieją powiązania pomiędzy paradygmatem programowania obiektowego a współbieżnością? Czy coraz większe potrzeby poprawy współbieżności doprowadzą do zmiany implementacji, czy natury projektów obiektowych?

Bjarne: Od bardzo długiego czasu istnieje powiązanie pomiędzy programowaniem obiektowym a współbieżnością. W Simuli 67, języku programowania, w którym po raz pierwszy były bezpośrednio dostępne techniki programowania obiektowego, istniały również mechanizmy do przedstawiania operacji współbieżnych.

Pierwszą biblioteką C++ była biblioteka obsługująca mechanizm, który dziś nazwalibyśmy wątkami. W Bell Labs już w 1988 roku wykorzystywaliśmy język C++ na komputerze sześcioprocesorowym i nie byliśmy jedynymi, którzy używali go w takich zastosowaniach. W latach dziewięćdziesiątych istniało co najmniej kilkadziesiąt eksperymentalnych dialektów języka C++ oraz bibliotek zajmujących się problemami programowania rozproszonego i równoległego. Współczesne zachłyśnięcie się systemami wielordzeniowymi nie jest moim pierwszym kontaktem ze współbieżnością. Przetwarzanie rozproszone było tematem mojej pracy doktorskiej i od tamtych czasów śledzę tę dziedzinę.

Ludzie, którzy po raz pierwszy stykają się ze współbieżnością, wieloma rdzeniami itp., często robią błąd, nie doceniając kosztów uruchamiania operacji na innym procesorze. Koszt uruchomienia operacji na innym procesorze (rdzeniu) oraz dostępu tej operacji do danych w pamięci procesora wywołującego (kopiowanie bądź też zdalny dostęp) może być tysiąc (lub więcej) razy większy niż koszt zwykłego wywołania funkcji. Poza tym ryzyko popełnienia błędów okazuje się dużo wyższe, jeśli zastosuje się współbieżność. Aby skutecznie wykorzystać udostępniane przez sprzęt możliwości przetwarzania równoległego, trzeba przemyśleć organizację oprogramowania.

Na szczęście możemy wykorzystać wyniki badań prowadzonych przez dziesięciolecia (które jednak mogą nas również wprowadzić w błąd). Ogólnie rzecz biorąc, jest tak wiele badań, że niemal niemożliwe okazuje się stwierdzenie, które są wartościowe, a tym bardziej — które są najlepsze. Dobrym punktem wyjścia może być artykuł na temat języka Emerald, wygłoszony na konferencji HOPL-III. Język ten był pierwszym, który zajmował się interakcją pomiędzy problemami języka a problemami systemowymi z uwzględnieniem kosztów. Ważną rzeczą jest również rozróżnienie pomiędzy równoległym przetwarzaniem danych, stosowanym od dziesięcioleci do wykonywania obliczeń naukowych (głównie w języku FORTRAN), a uruchamianiem komunikujących się ze sobą jednostek zwykłego sekwencyjnego kodu (na przykład procesów lub wątków) na wielu procesorach. Uważam, że aby język programowania mógł zdobyć powszechną akceptację we współczesnym świecie wielu rdzeni i klastrów, powinien obsługiwać oba rodzaje współbieżności, a najlepiej jeśli obsługiwałby po kilka odmian każdego z nich. Nie jest to łatwe, a problemy wykraczają poza tradycyjne sprawy dotyczące języków programowania — trzeba łącznie uwzględnić problemy związane z językiem, systemami i aplikacjami.

Czy język C++ jest przygotowany do obsługi współbieżności? Oczywiście można stworzyć biblioteki, które będą obsługiwały wszystko, ale czy język i standardowa biblioteka wymagają poważnych zmian pod kątem obsługi współbieżności?

Bjarne: Jest prawie przygotowany. Wersja C++0x będzie przygotowana. Aby język mógł obsługiwać współbieżność, musi przede wszystkim mieć dokładnie zdefiniowany model pamięci. Dzięki temu autorzy kompilatora mogą skorzystać z nowoczesnego sprzętu (wyposażonego w potoki, duże pamięci podręczne, bufory przewidywania

rozgałęzień, mechanizmy statycznego i dynamicznego przestawiania instrukcji itp.). Wtedy wystarczą niewielkie rozszerzenia języka: lokalna pamięć masowa z obsługą wątków oraz atomowe typy danych. Następnie można dodać obsługę współbieżności w postaci bibliotek. Naturalnie pierwszą nową biblioteką standardową będzie biblioteka obsługi wątków, umożliwiająca przenośne programowanie pomiędzy systemami, na przykład Linux i Windows. Oczywiście takie biblioteki istnieją od wielu lat, ale nie są to biblioteki standardowe.

Wątki wraz z pewną formą blokowania mającą na celu uniknięcie wyścigów to jeden z najgorszych sposobów bezpośredniej obsługi współbieżności. Jednak w języku C++ taki mechanizm jest potrzebny, aby można było obsłużyć istniejące aplikacje oraz aby język mógł spełnić swoją rolę — rolę systemowego języka programowania w tradycyjnych systemach operacyjnych. Prototypy takiej biblioteki istnieją — powstały na podstawie wielu lat aktywnego użytkowania języka.

Jednym z kluczowych problemów współbieżności jest sposób opakowania zadania, by mogło ono być wykonane współbieżnie z innymi zadaniami. Przewiduję, że w języku C++ rozwiązaniem tego problemu będzie obiekt funkcyjny. Obiekt może zawierać potrzebne dane i przekazywać je według potrzeb. Standard C++98 dobrze obsługuje ten mechanizm dla nazwanych operacji (nazwanych klas, z których egzemplifikujemy obiekty funkcyjne). Technika ta jest też powszechnie stosowana do parametryzacji w bibliotekach generycznych (na przykład STL). W standardzie C++0x ułatwiono pisanie prostych jednorazowych obiektów funkcyjnych poprzez wprowadzenie funkcji lambda, które mogą być pisane w kontekstach wyrażeń (na przykład jako argumenty funkcji) i odpowiednio generują obiekty funkcji (domknięcia).

Następne kroki są bardziej interesujące. Natychmiast po opublikowaniu standardu C++0x komisja planuje wydanie technicznego raportu na temat bibliotek. Niemal na pewno biblioteki będą obsługiwać pule wątków oraz jakąś formę „wykradania" pracy. Mam tu na myśli to, że będzie istniał standardowy mechanizm pozwalający użytkownikowi na współbieżne wykonywanie stosunkowo niewielkich jednostek pracy (zadań). Dzięki korzystaniu z tego mechanizmu użytkownik nie będzie się musiał martwić tworzeniem wątków, ich niszczeniem, blokadami itp. Mechanizmy te będą prawdopodobnie wbudowane w obiekty funkcyjne jako zadania. Poza tym `użytkownik` będzie mógł korzystać z mechanizmów komunikacji między geograficznie zdalnymi procesami za pomocą gniazd, strumieni wejścia-wyjścia itp., podobnych do tych oferowanych w bibliotece `boost::networking`.

W mojej opinii większość interesujących elementów współbieżności pojawi się w postaci wielu bibliotek obsługujących logicznie rozłączne modele współbieżności.

Wiele współczesnych systemów jest podzielonych na komponenty i rozproszonych w sieci. Era aplikacji webowych i aplikacji mashup może podkreślić ten trend. Czy w języku powinny się znaleźć mechanizmy obsługujące te aspekty pracy sieciowej?

Bjarne: Istnieje wiele form współbieżności. Celem niektórych z nich jest poprawa przepustowości lub czasu odpowiedzi programu na pojedynczym komputerze bądź klastrze, niektóre mają na celu obsługę geograficznej dystrybucji, a jeszcze inne znajdują się poniżej poziomu, jakim zwykle zajmują się programiści (potoki, pamięć podręczna itp.).

Standard C++0*x* dostarczy zbioru mechanizmów i gwarancji zabezpieczających programistów przed niskopoziomowymi szczegółami. Wprowadza on model maszyny, który będzie mógł pełnić rolę kontraktu pomiędzy architektami komputera a autorami kompilatorów. Dostarczy również bibliotekę obsługi wątków, w której znajdzie się proste odwzorowanie kodu na procesory. Skorzystanie z tych podstaw pozwala dostarczyć inne modele za pośrednictwem bibliotek. Chciałbym, aby w bibliotece standardu C++0*x* znalazły się prostsze w użytkowaniu, wyżejpoziomowe modele współbieżności, ale teraz wydaje się to mało prawdopodobne. Później — mam nadzieję, że wkrótce po opublikowaniu standardu C++0*x* — pojawi się więcej bibliotek wyspecyfikowanych w raporcie technicznym: pule wątków i obiekty `future`, a także biblioteka obsługi strumieni wejścia-wyjścia w sieci rozległej (na przykład TCP/IP). Takie biblioteki istnieją, ale nie wszyscy uważają, że są one na tyle dobrze wyspecyfikowane, aby mogły stać się standardowe.

Wiele lat temu miałem nadzieję, że standard C++0*x* zajmie się starymi dla języka C++ problemami dystrybucji poprzez wyspecyfikowanie standardowej formy serializacji, ale tak się nie stało. A zatem społeczność programistów języka C++ będzie zmuszona rozwiązywać bardziej wysokopoziomowe problemy przetwarzania rozproszonego i aplikacji rozproszonych za pomocą niestandardowych bibliotek i/lub platform framework (na przykład CORBA lub .NET).

Pierwsza biblioteka języka C++ (w rzeczywistości pierwszy kod w języku C z obsługą klas) dostarczała lekkiej obsługi współbieżności. Przez lata w C++ powstały setki bibliotek i frameworków obsługujących przetwarzanie współbieżne, równoległe i rozproszone, ale społeczność nie zdołała się porozumieć co do standardów. Przypuszczam, że częściowo problem ten wynika z tego, że zrobienie czegoś poważnego w tej dziedzinie wymaga znacznych nakładów finansowych, a wielcy gracze wolą wydawać pieniądze na własne zastrzeżone biblioteki, frameworki i języki. Nie jest to dobre dla społeczności języka C++ jako całości.

Przyszłość

Czy kiedykolwiek powstanie standard C++ 2.0?

Bjarne: To zależy, co pan rozumie przez „C++ 2.0". Jeśli oczekuje pan nowego języka, stworzonego prawie od podstaw, zawierającego wszystko, co najlepsze w C++, i pozbawionego wszystkiego tego, co złe (dla określonych definicji tego, co dobre, i tego, co złe), to moja odpowiedź brzmi: „Nie wiem". Chciałbym, aby powstał nowy język wywodzący się z tradycji C++, ale nie widzę takiego na horyzoncie, zatem pozwoli pan, że skoncentruję się na następnym standardzie ISO języka C++, znanym pod nazwą C++0*x*.

Dla wielu osób będzie to C++ 2.0, ponieważ dostarczy nowych własności języka i nowych standardowych bibliotek, ale jednocześnie będzie niemal w 100% zgodny z C++98. Nazwaliśmy go C++0*x* z nadzieją, że nazwa ta przekształci się w C++09. Jeśli się spóźnimy — w związku z czym *x* będzie musiał przyjąć wartość cyfry szesnastkowej — to zarówno ja, jak i inni członkowie zespołu będziemy smutni i zakłopotani.

Standard C++0*x* będzie niemal w 100% zgodny z C++98. Naszym celem nie jest doprowadzenie do tego, by istniejący kod przestał działać. Najbardziej znaczące niezgodności wynikają z użycia kilku nowych słów kluczowych, takich jak `static_assert`, `constexpr` i `concept`. Staraliśmy się zminimalizować ich oddziaływanie na kod, wybraliśmy więc nowe słowa kluczowe, które nie są często wykorzystywane. Najważniejsze usprawnienia to:

- Obsługa nowoczesnych architektur komputerów i współbieżności: model maszyny, biblioteka wątków, lokalna pamięć masowa wątków i operacje atomowe oraz mechanizmy asynchronicznego zwracania wartości (obiekty `future`).
- Lepsza obsługa programowania generycznego: typ `concept` (system typologiczny dla typów, kombinacji typów oraz kombinacji typów i liczb `integer`) umożliwiający lepszą kontrolę definicji i zastosowań szablonów oraz lepsze przeciążanie szablonów. Dedukcja typów bazująca na inicjalizatorach (`auto`), uogólnione listy inicjalizacyjne, uogólnione wyrażenia stałe (`constexpr`), wyrażenia lambda i wiele innych.
- Wiele drobnych rozszerzeń języka, takich jak statyczne asercje, semantyka przenoszenia (ang. *move semantics*), poprawione enumeracje, nazwa pustego wskaźnika (*nullptr*) itp.
- Nowe biblioteki standardowe obsługi dopasowywania wyrażeń regularnych, tablic skrótów (na przykład `unordered_map`), inteligentnych wskaźników itp.

Szczegółowe informacje można znaleźć w witrynie internetowej komitetu standaryzacyjnego C++[3]. Sumaryczne zestawienie zamieściłem na prowadzonej przeze mnie stronie internetowej C++0*x* FAQ[4].

[3] *http://www.open-std.org/jtc1/sc22/wg21/.*

[4] *http://www.research.att.com/~bs/C++0xFAQ.html.*

Warto zwrócić uwagę, że kiedy mówię o zachowaniu działania kodu, odnoszę się do rdzenia języka oraz biblioteki standardowej. Stary kod przestanie oczywiście działać, jeśli wykorzystuje niestandardowe rozszerzenia od jakiegoś dostawcy kompilatora lub antyczne niestandardowe biblioteki. Z mojego doświadczenia wynika, że jeśli ktoś skarży się na to, że kod przestał działać lub że jest niestabilny, zazwyczaj przyczyną są zastrzeżone własności i biblioteki. Jeśli na przykład zmienimy system operacyjny, a w programie nie skorzystaliśmy z jednej z przenośnych bibliotek GUI, prawdopodobnie będziemy zmuszeni do wykonania pewnych operacji z kodem interfejsu użytkownika.

Co pana powstrzymuje przed stworzeniem całkowicie nowego języka?

Bjarne: Natychmiast pojawia się kilka kluczowych pytań:

- Jakie problemy miałby rozwiązywać nowy język?
- Czyje problemy rozwiązywałby ten język?
- Jakie nowości można by wprowadzić (w porównaniu z każdym z istniejących języków)?
- Czy nowy język może być skutecznie wdrożony (w świecie, w którym istnieje wiele języków o mocnej pozycji)?
- Czy praca nad nowym językiem ma być tylko przyjemnym oderwaniem się od ciężkiej pracy polegającej na wspomaganiu tworzenia lepszych narzędzi i systemów?

Dotychczas nie udało mi się w zadowalający mnie sposób odpowiedzieć na te pytania.

Nie oznacza to jednak, że uważam C++ za perfekcyjny w swojej klasie. Nie jest on perfekcyjny. Jestem przekonany, że można by zaprojektować język, który miałby rozmiary nieprzekraczające jednej dziesiątej rozmiaru C++ (niezależnie od sposobu mierzenia rozmiaru) i który w mniejszym lub w większym stopniu dawałby te same możliwości co C++. Tymczasem tworzenie nowego języka to coś więcej niż tylko powielenie operacji wykonywanych w istniejącym języku, tyle że nieco lepiej i nieco bardziej elegancko.

Jakie wnioski z lekcji na temat powstania, dalszego rozwoju i przystosowywania się pańskiego języka do warunków współczesnych mogą wyciągnąć programiści, którzy tworzą systemy komputerowe dziś oraz będą je tworzyć w najbliższej przyszłości?

Bjarne: To doskonałe pytanie — czy potrafimy uczyć się z historii? Jeśli tak, to jak i czego możemy się nauczyć? W początkowej fazie tworzenia języka C++ sformułowałem zbiór reguł oczywistych. Można je znaleźć w książce *The Design and Evolution of C++* [Addison-Wesley]. Omówiłem je także w dwóch referatach

wygłoszonych na konferencji HOPL. Oczywiste jest, że każdy poważny projekt języka programowania wymaga zbioru zasad, które powinny być sformułowane jak najwcześniej. To są właśnie wnioski z doświadczeń tworzenia języka C++. Nie sformułowałem zasad projektowych języka C++ dostatecznie wcześnie i nie doprowadziłem do tego, aby zasady te zostały dostatecznie rozpowszechnione. W efekcie wiele osób opracowało własne reguły projektowania w C++. Niektóre z nich były dość zabawne i wprowadziły sporo zamieszania. Do dziś niektórzy postrzegają C++ jako raczej nieudaną próbę zaprojektowania języka przypominającego Smalltalk (nie, język C++ nie miał przypominać Smalltalka, C++ naśladuje model programowania obiektowego z języka Simula) lub jako tylko próbę rozwiązania niektórych wad pisania w języku C (nie, język C++ nie miał być tylko poprawką języka C).

Celem języka programowania (jeśli nie jest to język eksperymentalny) jest pomoc w budowaniu dobrych systemów. Pojęcia projektowania systemów i projektowania języka są ze sobą blisko związane.

Moja definicja słowa „dobry" w tym kontekście brzmi: „poprawny, łatwy w pielęgnacji i zużywający zasoby na akceptowalnym poziomie". Oczywistym brakującym komponentem jest „łatwy do pisania", ale dla tego rodzaju systemów, o których przede wszystkim myślę, ma to drugorzędne znaczenie. Ideologia szybkiego wytwarzania aplikacji (ang. *RAD development*) nie jest moim ideałem. Czasami stwierdzenie tego, co nie jest podstawowym celem, okazuje się równie ważne jak to, co nim jest. Na przykład nie mam nic przeciwko szybkiemu wytwarzaniu oprogramowania — nikt o zdrowych zmysłach nie chce poświęcać projektowi więcej czasu, niż to konieczne — ale za ważniejszy od szybkiego wytwarzania uważam brak ograniczeń w pewnych obszarach aplikacji oraz brak ograniczeń wydajności. Moim celem podczas tworzenia języka C++ było i jest bezpośrednie wyrażenie idei, czego wynikiem jest kod wydajny zarówno pod względem czasu działania, jak i zajmowanego miejsca.

Języki C i C++ gwarantują stabilność na dziesięciolecia. Ma to niezwykłe znaczenie dla strategicznych użytkowników języka. Znam wiele małych programów, które nie były zmieniane od początku lat osiemdziesiątych. Taka stabilność ma swoją wartość, ale języki, które jej nie zapewniają, po prostu nie nadają się do stosowania w dużych, długo realizowanych projektach. Języki korporacyjne oraz języki, które próbują gonić trendy, zupełnie się tu nie sprawdzają, a próby ich stosowania powodują wiele szkód.

Prowadzi to do myślenia o właściwym sposobie zarządzania ewolucją. Ile można zmieniać? Jaka powinna być szczegółowość zmian? Zmienianie języka co rok, w takim tempie, w jakim powstają nowe wydania produktów, jest zbyt gwałtowne i prowadzi do stosowania wielu rozwiązań częściowych, niedokończonych bibliotek i konstrukcji języka i/lub konieczności aktualizacji na masową skalę. Poza tym rok to po prostu zbyt krótki okres inkubacji dla ważnych własności. Takie podejście prowadzi zatem do powstawania rozwiązań połowicznych oraz martwych punktów. Z drugiej strony

dziesięcioletni cykl ISO dla języków standaryzowanych, takich jak C lub C++, jest zbyt długi i powoduje zastój części społeczności języka (a także części komitetu standaryzacyjnego).

Wokół języka, który odnosi sukces, tworzy się społeczność, która współdzieli techniki, narzędzia i biblioteki. Języki korporacyjne mają tu wielką przewagę: korporacje mogą kupić udział w rynku za pomocą technik marketingowych, konferencji oraz darmowych bibliotek. Taka inwestycja może doprowadzić do rozwoju społeczności — staje się ona liczniejsza i bardziej aktywna. Wysiłki firmy Sun dotyczące języka Java pokazały, jak amatorskie i niedostatecznie finansowane były wcześniejsze próby stworzenia języka ogólnego przeznaczenia. Ostrym kontrastem dla działań firmy Sun mogą być wysiłki Departamentu Obrony USA do nadania dominującej roli językowi Ada oraz niedostatecznie dofinansowane próby nadania takiej roli językowi C++ (przeze mnie i moich kolegów).

Nie mogę powiedzieć, że popieram wszystkie posunięcia firmy Sun związane z językiem Java — na przykład kwestionuję sprzedaż typu góra-dół (ang. *selling top-down*) firmom nieprogramistycznym — choć na tej podstawie łatwo zobaczyć, co można zrobić. Spośród społeczności języków niekorporacyjnych sukces odniosły te, które są skupione wokół Pythona i Perla. Sukcesów społeczności użytkowników języka C++ było zbyt mało i były one zbyt ograniczone, jeśli wziąć pod uwagę rozmiar społeczności. Konferencje ACCU są doskonałe. Dlaczego jednak mniej więcej od 1986 roku nie przeprowadza się dorocznych międzynarodowych konferencji na temat języka C++? Biblioteki *Boost* są świetne, dlaczego jednak nie stworzono centralnego repozytorium bibliotek C++ (co również można było zrobić około 1986 roku)? W użytku są tysiące bibliotek typu open source napisanych w C++. Komercyjnych bibliotek jest chyba jeszcze więcej. Nie będę teraz odpowiadał na te pytania. Chcę jedynie wskazać, że każdy nowy język musi jakoś zarządzać swoimi zasobami w dużej społeczności. Jeśli tego nie zrobi, poniesie poważne konsekwencje.

Język ogólnego przeznaczenia potrzebuje informacji i aprobaty wielu społeczności — na przykład programistów branżowych, wykładowców, akademickich pracowników naukowych, osób zajmujących się badaniami naukowymi w ośrodkach przemysłowych oraz społeczności open source. Społeczności te nie są rozłączne. Często jednak pojedyncze podspołeczności postrzegają siebie jako grupę samowystarczalną, która posiada wiedzę na temat tego, co jest prawidłowe, oraz pozostaje w konflikcie z innymi społecznościami, które z jakiegoś powodu tego nie rozumieją. Może stąd wynikać znaczący problem praktyczny. Na przykład część społeczności open source sprzeciwia się używaniu języka C++, ponieważ „to język firmy Microsoft" (a to nieprawda) lub „jest własnością AT&T" (to także nieprawda), natomiast niektórzy kluczowi gracze w branży uważają, że problemem języka C++ jest to, że nie są jego właścicielami.

Zasadniczym problemem jest to, że wiele podspołeczności propaguje ograniczony i zaściankowy pogląd na to, czym naprawdę jest programowanie oraz co jest naprawdę potrzebne. Gdyby wszyscy robili wszystko tak, jak należy, problemu by nie było. Trzeba

dążyć do zrównoważenia różnych potrzeb w celu stworzenia większej i bardziej zróżnicowanej społeczności. Dla bardziej doświadczonych programistów uniwersalność i elastyczność języka są ważniejsze od dostarczania optymalnych rozwiązań ograniczonego zakresu problemów.

Wróćmy do spraw technicznych. W dalszym ciągu uważam, że elastyczny i uniwersalny system bazujący na typach statycznych jest świetny. Moje doświadczenie w języku C++ jeszcze wzmacnia ten pogląd. Jestem również gorącym zwolennikiem prawdziwych zmiennych lokalnych typów definiowanych przez użytkowników: stosowane w języku C++ techniki obsługi ogólnych zasobów na podstawie zmiennych definiowanych w zasięgach nie miały sobie równych, jeśli chodzi o wydajność. Konstruktory i destruktory, często używane razem z technikami RAII (od ang. *Resource Acquisition Is Initialization*), pozwalają na uzyskanie bardzo eleganckiego i wydajnego kodu.

Edukacja

Opuścił pan branżę, by zostać pracownikiem akademickim. Dlaczego?

Bjarne: Właściwie do końca nie opuściłem branży, ponieważ utrzymuję kontakty z AT&T Labs oraz z ludźmi z AT&T oraz co roku spędzam sporo czasu z ludźmi z branży. Moje związki z branżą uważam za kluczowe, ponieważ to one pozwalają umiejscowić moją pracę w realiach.

Pięć lat temu zacząłem pracować jako wykładowca na Uniwersytecie Texas A&M (po prawie 25 latach pracy dla AT&T Labs), ponieważ odczuwałem potrzebę zmiany i uważałem, że w edukacji mam coś do zaoferowania. Poważnie rozważałem także dość idealistyczne pomysły, by zacząć bardziej gruntowne badania po wielu latach prowadzenia praktycznych badań i projektów.

Większość badań w informatyce albo jest zbyt odległych od codziennych problemów (nawet od hipotetycznych problemów przyszłości), albo są one tak zagłębione w codzienne problemy, że stają się czymś, co niewiele odbiega od transferu technologii. Oczywiście nie mam nic przeciwko transferowi technologii (bardzo go potrzebujemy), ale powinno istnieć silne sprzężenie zwrotne pomiędzy praktyką branżową a zaawansowanymi badaniami. Krótkowzroczność planowania wielu osób w branży oraz wymagania powstawania publikacji akademickich prowadzą do odwrócenia uwagi od kluczowych problemów.

Czego dowiedział się pan podczas swojej pracy akademickiej o nauczaniu programowania początkujących?

Bjarne: Najbardziej konkretnym rezultatem mojej pracy na uniwersytecie (oprócz obowiązkowych artykułów) jest nowy podręcznik nauczania programowania skierowany do osób, które nigdy wcześniej nie programowały: *Programming: Principles and Practice Using C++* [Addison-Wesley].

To moja pierwsza książka dla początkujących. Zanim stałem się wykładowcą akademickim, po prostu nie znałem wystarczająco dużo niedoświadczonych osób, abym mógł napisać taką książkę. Uważałem jednak, że zbyt wielu programistów było źle przygotowanych do wykonywania zadań w branży i w innych obszarach. Teraz, kiedy mam za sobą doświadczenie w nauczaniu ponad 1200 początkujących programistów, zyskałem nieco większą pewność, że moje idee w tym obszarze będą skalowalne.

Książka dla początkujących musi spełniać kilka celów. Przede wszystkim powinna zapewniać dobrą podstawę do dalszej nauki (jeśli podręcznik odniesie sukces, jego lektura będzie stanowiła punkt wyjścia do długofalowego procesu), a także uczyć pewnych praktycznych umiejętności. Trzeba też pamiętać, że programowanie, a ściślej — wytwarzanie oprogramowania, nie jest wyłącznie umiejętnością teoretyczną. Nie jest też czymś, co można robić dobrze bez przyswojenia pewnych podstawowych pojęć. Niestety, nazbyt często w procesie nauczania zapomina się o zachowaniu równowagi pomiędzy teorią (zasadami) a praktyką (technikami). W konsekwencji spotykamy ludzi, którzy właściwie gardzą programowaniem (uznają wyłącznie kodowanie) i uważają, że oprogramowanie można tworzyć po zapoznaniu się z pierwszymi zasadami, bez żadnych praktycznych umiejętności. Z drugiej strony są ludzie, którzy uważają, że każdy kod jest dobry i wszystko można osiągnąć poprzez szybki wgląd w podręcznik online oraz trochę wycinania i wklejania. Spotkałem programistów, którzy implementację K&R języka C uważali za zbyt skomplikowaną i teoretyczną. W mojej opinii oba podejścia są zbyt ekstremalne i prowadzą do powstawania kodu o złej strukturze, niewydajnego i trudnego do pielęgnacji, mimo że czasami udaje się stworzyć działające fragmenty aplikacji.

Jaka jest pana opinia o przykładach kodu zamieszczanych w podręcznikach? Czy powinny uwzględniać one kontrolę błędów (obsługę wyjątków)? Czy powinny to być kompletne programy, które można skompilować i uruchomić?

Bjarne: Jestem zwolennikiem przykładów, które za pomocą jak najmniejszej liczby wierszy pozwalają zilustrować ideę. Takie fragmenty programów często są niekompletne, chociaż sądzę, że moje skompilują się i uruchomią, jeśli osadzę je na odpowiedniej ramie. Ogólnie mój styl prezentacji kodu wywodzi się z implementacji K&R. W mojej nowej książce wszystkie przykłady kodu będą dostępne w postaci możliwej do skompilowania. Niewielkie fragmenty kodu umieszczone w tekście opisującym ich działanie przeplatam z dłuższymi, bardziej kompletnymi fragmentami kodu. W kluczowych miejscach wykorzystuję obie techniki dla pojedynczego przykładu, tak aby czytelnik mógł dwukrotnie przyjrzeć się kluczowym instrukcjom.

Niektóre przykłady powinny zawierać mechanizmy kontroli błędów, a wszystkie powinny odzwierciedlać projekt, który da się sprawdzić. Oprócz opisów błędów i ich obsługi, zamieszczonych w różnych miejscach książki, znalazły się w niej również

osobne rozdziały poświęcone obsłudze błędów i testowaniu. Preferuję przykłady pochodzące z realnie działających programów. Nie znoszę sztucznych przykładów w rodzaju drzew dziedziczenia zwierząt oraz głupich łamigłówek matematycznych. Być może powinienem opatrzyć moją książkę etykietą: „Przykłady w tej książce nie zawierają aktów maltretowania zwierząt".