

Idealy i historia

„Gdy ktoś powie,
że chce języka programowania,
w którym tylko wyrazi, co chce mieć zrobione,
daj mu lizaka.”

— Alan Perlis

W tym rozdziale krótko i wybiórczo przedstawimy historię języków programowania oraz opiszemy ideały, którym miały w zamierzeniu twórców służyć. Ideały i języki, które je wyrażają, są podstawą profesjonalizmu. Ponieważ w książce tej wykorzystujemy język C++, skoncentrujemy się właśnie na tym języku i językach, które miały na niego wpływ. Treść tego rozdziału ma stanowić tło i pozwolić Ci spojrzeć z dalszej perspektywy na opisywane w tej książce idee. Do opisu każdego języka dołączamy sylwetkę jego twórcy lub twórców, ponieważ żaden język programowania nie jest bytem abstrakcyjnym, tylko konkretnym rozwiązaniem zaprojektowanym przez ludzi, którzy chcieli rozwiązać napotkane przez siebie problemy.

22.1. Historia, ideały i profesjonalizm

22.1.1. Cele i filozofie języków programowania

22.1.2. Ideały programistyczne

22.1.3. Style i paradygmaty

22.2. Krótka historia języków programowania

22.2.1. Pierwsze języki

22.2.2. Korzenie nowoczesnych języków programowania

22.2.3. Rodzina Algol

22.2.4. Simula

22.2.5. C

22.2.6. C++

22.2.7. Dziś

22.2.8. Źródła informacji

22.1. Historia, ideały i profesjonalizm

Słynne powiedzenie Henry’ego Forda brzmi: „Historia to banialuki”. Jednak już od czasów starożytnych cytuje się słynne powiedzenie: „Kto nie zna historii, jest skazany na jej powtarzanie”. Trudność polega na tym, że nie wiadomo, co z historii zapamiętać, a o czym zapomnieć — „95% wszystkiego to banialuki” to kolejny cytat, który warto w tym momencie przytoczyć (choć wydaje nam się, że 95% to zbyt mała liczba w tym kontekście). Zgodnie z naszym pojmowaniem relacji między historią a teraźniejszością nie może być prawdziwego profesjonalizmu bez znajomości historii. Jeśli ktoś za słabo zna historię swojej dziedziny, jest naiwny, ponieważ w każdej dyscyplinie powstało mnóstwo pomysłów, które okazały się nietrafione. To, co jest w historii najbardziej wartościowe, to idee i ideały, które dowiodły swej wartości poprzez praktykę.

Z wielką przyjemnością napisalibyśmy o początkach najważniejszych idei w wielu językach programowania i wszelkiego rodzaju oprogramowaniu, jak systemy operacyjne, bazy danych, grafika, sieci, internet, skrypty itp., ale niestety będziesz musiał poszukać informacji na te tematy gdzieś indziej. Nam z ledwością wystarczy miejsca, aby pobeżnie przedstawić ideały i historię języków programowania.

Ostatecznym celem każdego projektu programistycznego zawsze jest powstanie przydatnego systemu. Często zapomina się o tym w ogniu dyskusji na temat technik i języków programowania. Nie zapominaj tego! Jeśli potrzebujesz przypomnienia, zajrzyj jeszcze raz do rozdziału 1.

22.1.1. Cele i filozofie języków programowania

Co to jest język programowania? Jakie funkcje ma on spełniać? Do popularnych odpowiedzi na pytanie, czym jest język programowania, należą:

- Narzędzie do wydawania maszynom instrukcji.
- Notacja do wyrażania algorytmów.
- Sposób komunikacji programistów.
- Narzędzie do przeprowadzania doświadczeń.
- Sposób na sterowanie urządzeniami zawierającymi komputer.
- Sposób na wyrażanie powiązań między koncepcjami.
- Sposób na opisywanie wysokopoziomowych projektów.

Nasza odpowiedź brzmi: „Wszystkie powyższe, a nawet więcej!”. Mamy tu (i w całym tym rozdziale) oczywiście na myśli języki programowania ogólnego przeznaczenia. Dodatkowo istnieją też języki specjalne i przeznaczone do zastosowań w określonych dziedzinach, które służą węższemu i zwykle lepiej określonym celom.

Jakie naszym zdaniem cechy powinien mieć język programowania:

- przenośność,
- bezpieczeństwo typów,
- precyzyjna definicja,
- wysoka wydajność,

- możliwość zwięzłego wyrażania myśli,
- wszystko, co ułatwia usuwanie błędów,
- wszystko, co ułatwia testowanie,
- dostęp do wszystkich zasobów komputera,
- niezależność od platformy,
- możliwość działania na wszystkich platformach (np. Linux, Windows, smartfony, układy wbudowane),
- stabilność przez wiele lat,
- szybkie poprawki w odpowiedzi na zmiany w obszarach zastosowań,
- łatwość do nauki,
- mały rozmiar,
- obsługa popularnych stylów programowania (np. programowanie obiektowe i ogólne),
- wszystko, co pomaga analizować programy,
- duża liczba narzędzi,
- wsparcie ze strony licznej społeczności,
- przyjazny dla początkujących (studentów, uczniów),
- pełne rozwiązania dla ekspertów (np. narzędzia do budowy infrastruktury),
- dostępność wielu narzędzi programistycznych,
- dostępność wielu komponentów programowych (np. bibliotek),
- wsparcie ze strony otwartej społeczności,
- obsługa przez najważniejsze platformy (Microsoft, IBM itp.).

Niestety nie da się spełnić tych wszystkich wymagań na raz. Jest to tym bardziej smutne, że każde z nich samo w sobie, patrząc obiektywnie, jest czymś wartościowym. Każde wprowadza coś wartościowego, a brak obsługi któregoś oznacza, że użytkownik takiego języka będzie musiał nałożyć pracy i poradzić sobie z większymi komplikacjami. Powód, dla którego nie da się spełnić wszystkich wymienionych wymagań, ma fundamentalny charakter — niektóre z nich wzajemnie się wykluczają. Nie można na przykład zapewnić 100% niezależności od platform i jednocześnie umożliwić dostępu do wszystkich zasobów systemu. Program, który korzysta z zasobów niedostępnych na wszystkich platformach, nie jest w pełni przenośny. Analogicznie każdy chce, aby język programowania (oraz związane z nim biblioteki i narzędzia) był mały i łatwy do nauki. Nie można spełnić tego życzenia, jeśli jednocześnie język ma w pełni obsługiwać wszystkie rodzaje systemów i nadawać się do użytku we wszystkich obszarach zastosowań.



W takich sytuacjach do gry wchodzi ideały. Na ich podstawie każdy projektant języka programowania, biblioteki i narzędzi programistycznych dokonuje wyborów i decyduje się na kompromisy. Tak, każdy, kto pisze program, jest projektantem, a więc musi podjąć różne decyzje projektowe.

22.1.2. Ideały programistyczne

Przedmowa książki *Język C++* zaczyna się od następującego zdania: „C++ jest językiem ogólnego przeznaczenia, zaprojektowanym z myślą o tym, by uprzyjemnić programowanie doświadczonym programistom”. Co takiego? Czy celem programowania nie jest dostarczanie produktów? Czy nie chodzi w nim o zapewnienie poprawności, wysokiej jakości i możliwości modyfikacji produktu w przyszłym czasie? Czy nie dotyczy ono czasu wprowadzania produktu na rynek i wspierania inżynierii oprogramowania? Oczywiście to też, ale nie można zapominać o programiście. Rozważmy inny przykład. Kiedyś Donald Knuth powiedział: „Największą zaletą Alto jest to, że nie działa szybciej w nocy”. Alto to nazwa komputera z centrum Xerox Palo Alto Research (PARC), który był jednym z pierwszych „komputerów osobistych” w przeciwieństwie do współdzielonych komputerów, które przez cały dzień były gromadnie oblegane.



Nasze narzędzia i techniki programistyczne zostały zaprojektowane w taki sposób, aby umożliwić programiście, człowiekowi, wydajniejszą pracę i osiąganie lepszych wyników. Nie zapominaj o tym. Jakże zatem wskazówki możemy dać programiście, który chce tworzyć jak najlepsze oprogramowanie przy jak najmniejszym nakładzie pracy? Nasze ideały wyraziliśmy na kartach wcześniejszych rozdziałów tej książki, a w tym rozdziale tylko je podsumujemy.



Powodem, dla którego staramy się pisać kod o dobrej strukturze, jest to, że dzięki niej można wprowadzać zmiany bez nadmiernego wysiłku. Im lepsza struktura, tym łatwiej wprowadzać zmiany, znajdować i poprawiać błędy, dodawać nowe funkcje, przenieść program na inną architekturę sprzętową, przyspieszyć działanie itd. To właśnie rozumiemy pod słowem „dobry”.

W pozostałej części tego podrozdziału:

- Powtórzymy, co chcemy osiągnąć, tzn. czego chcemy od naszego kodu.
- Zaprezentujemy dwa ogólne podejścia do rozwoju oprogramowania oraz udowodnimy, że ich kombinacja jest lepsza niż każde z osobna.
- Rozważymy najważniejsze aspekty wyrażanej w kodzie struktury programu:
 - bezpośrednie wyrażanie myśli,
 - poziom abstrakcji,
 - modułowość,
 - spójność i minimalizm.



Ideały są po to, aby z nich korzystać. Są to narzędzia wspomagające myślenie, a nie tylko frazesy, których głoszenie ma wprawić w zadowolenie kierowników i egzaminatorów. Programy mają w przybliżeniu odzwierciedlać ideały. Gdy utknijemy w pracy nad programem, zastanawiamy się przez chwilę, aby sprawdzić, czy ich źródłem nie jest odejście od któregoś z ideałów — to czasami pomaga. Oceniając program (najlepiej zanim zostanie wysłany do użytkowników), szukamy punktów niezgodnych z ideałami, które mogą w przyszłości powodować problemy. Stosuj ideały wszędzie, gdzie się da, ale pamiętaj, że uwarunkowania praktyczne (np. wydajność i prostota) i słabe strony języka programowania (żaden język nie jest perfekcyjny) często pozwolą Ci tylko na zbliżenie się do nich.

Ideały można wykorzystać jako wskazówki przy podejmowaniu technicznych decyzji. Nie można na przykład podjąć wszystkich decyzji dotyczących interfejsu biblioteki w oderwaniu od rzeczywistości (podrozdział 14.1). Gdyby tak się stało, wynik byłby opłakany. W zamian należy wrócić do swoich pierwszych zasad, zdecydować, co w danej bibliotece ma największe znaczenie, i dopiero wtedy opracować zwięzły zbiór interfejsów. Najlepiej, gdy opis zastosowanych zasad projektowych i kompromisowych decyzji znajdzie się w dokumentacji i komentarzach do kodu.

Zaczynając projekt, przejrzyj ideały i zobacz, jak odnoszą się do postawionego problemu i wczesnych pomysłów dotyczących rozwiązań. Może to być dobry sposób na zdobycie i dopracowanie pomysłów. W późniejszych fazach procesu rozwoju, gdy utkniesz, możesz spojrzeć na wszystko z perspektywy i zobaczyć, w którym miejscu kod najbardziej oddalił się od ideałów — tzn. gdzie istnieje największe ryzyko wystąpienia błędów i problemów projektowych. Jest to alternatywa dla najczęściej stosowanej techniki polegającej na wielokrotnym przeglądaniu kodu w tym samym miejscu i próbach rozwiązania problemów tymi samymi metodami. „Błędy są wszędzie, gdzie nie patrzysz — w przeciwnym wypadku już byś je znalazł”.

22.1.2.1. Czego chcemy

Najczęściej chcemy:

- *Poprawności* — mimo iż trudno może być zdefiniować, co ma się na myśli, mówiąc „poprawność”, zrobienie tego jest ważną częścią pracy. Często ktoś inny mówi nam, co znaczy „poprawność” w danym projekcie, ale wówczas trzeba te słowa zinterpretować na swój sposób.
- *Łatwość utrzymania* — każdy dobry program będzie poddawany w przyszłości modyfikacjom. Zostanie przeniesiony na nową platformę sprzętową i programową, doda mu się nowe funkcje oraz zostaną znalezione nowe błędy, które będzie trzeba poprawić. Ideału tego dotyczą poniższe podrozdziały poświęcone strukturze programu.
- *Wydajność* — wydajność („efektywność”) to pojęcie względne. Musi być odpowiednia dla danego rodzaju programu. Często słyszy się stwierdzenia, że aby napisać wydajny kod, należy programować na niskim poziomie, oraz że wszystko, co ma związek z dobrą, wysoko poziomą strukturą, powoduje straty wydajności. W rzeczywistości jest wręcz przeciwnie — zauważyliśmy, że akceptowalny poziom wydajności można osiągnąć poprzez trzymanie się ideałów i stosowanie zalecanych przez nas technik. Biblioteka STL jest przykładem kodu, który jest jednocześnie abstrakcyjny i bardzo wydajny. Słaba wydajność może być równie dobrze wynikiem obsesyjnego dbania o niskopoziomowe szczegóły, jak i ich lekceważenia.
- *Dostawa na czas* — dostarczenie idealnego programu o rok za późno zazwyczaj nie jest najlepszym pomysłem. Oczywiście ludzie oczekują rzeczy niemożliwych, ale musimy tworzyć dobrej jakości oprogramowanie w rozsądnym czasie. Istnieje mit, że projekt ukończony na czas jest tandetny. Wręcz przeciwnie. Z naszego doświadczenia wynika, że kładzenie nacisku na strukturę kodu (np. zarządzanie zasobami, stosowanie niezmienników oraz projekt interfejsu), projekt i przydatność do testowania oraz wykorzystanie odpowiednich bibliotek (często zaprojektowanych do konkretnego zastosowania lub obszaru zastosowań) są dobrymi sposobami na zmieszczenie się w wyznaczonym terminie.

To zmusza do zastanowienia się nad strukturą kodu:

- Jeśli w programie jest błąd (każdy duży program zawiera błędy), łatwiej go znaleźć, jeżeli program ten ma przejrzystą strukturę.
- Jeśli ktoś nowy musi zrozumieć lub zmodyfikować program, kod o dobrej strukturze jest nieporównywalnie łatwiej zrozumieć niż bałagan przeplatany niskopoziomowymi szczegółami.
- Jeśli wystąpią problemy z wydajnością programu, często łatwiej jest zoptymalizować kod wysokopoziomowy (taki, który stanowi dobre przybliżenie ideałów i ma dobrze zdefiniowaną strukturę) niż niskopoziomowy lub niechlujny. Dla początkujących wysokopoziomowy kod jest łatwiejszy do zrozumienia. Ponadto kod wysokiego poziomu jest często znacznie szybciej gotowy do optymalizowania i testowania niż niskopoziomowy.



Zwracamy szczególną uwagę na twierdzenie o zrozumiałości kodu. Wszystko, co pomaga zrozumieć program i rozumować na jego temat, jest dobre. Zasadniczo regularność jest lepsza od nieregularności — pod warunkiem że nie osiąga się jej poprzez nadmierne upraszczanie.

22.1.2.2. Ogólne podejścia

Są dwa podejścia do pisania poprawnego oprogramowania:

- *Od dołu* — składanie systemu ze składników, których poprawność jest udowodniona.
- *Od góry* — składanie systemu ze składników, które zakłada się, że mają błędy, i znajdowanie tych błędów.



Co ciekawe, najbardziej niezawodne systemy tworzy się, łącząc te dwa, zdawałoby się, przeciwstawne podejścia. Powód tego jest bardzo prosty — przy tworzeniu dużego systemu do realnego użytku nie da się zapewnić wymaganych poprawności, adaptowalności i utrzymywalności bez względu na zastosowane podejście:


- Nie da się zbudować wystarczającej liczby podstawowych komponentów i udowodnić ich poprawności, aby wyeliminować wszystkie możliwe źródła błędów.
- Nie da się całkowicie zrekompensować wad zawierających błędy podstawowych komponentów (bibliotek, podsystemów, hierarchii klas itp.), jeśli wciela się je do końcowej wersji systemu.

Jednak zbliżenie się do obu tych podejść na raz może dać lepszy wynik niż każde z nich z osobna — możemy tworzyć (albo pożyczać lub kupować) wystarczająco dobre komponenty, aby pozostające w nich problemy zrekompensować dobrą obsługą błędów i systematycznym przeprowadzaniem testów. Ponadto, jeśli będziemy budować lepsze komponenty, można będzie z nich skonstruować większą część systemu, co zredukuje zapotrzebowanie na niechlujny kod pisany „na szybkiego”.




Testowanie jest niezbędnym składnikiem procesu rozwoju oprogramowania. Opis tych technik znajduje się w rozdziale 26. Testowanie to systematyczne przeszukiwanie kodu w celu znalezienia błędów. Istnieje popularny slogan: „Testuj od wczesnych etapów pracy i rób to często”. Staramy się tak projektować programy, aby łatwo się je testowało i aby trudno było ukryć się błędem w kodzie.

22.1.2.3. Bezpośrednie wyrażanie myśli

Gdy pisze się kod wyrażający coś — bez względu na to, czy jest to kod wysoko czy niskopoziomowy — należy to wyrazić bezpośrednio, a nie opłótkami. Fundamentalny ideał dotyczący bezpośredniego reprezentowania myśli w postaci kodu występuje w kilku wersjach: 

- *Bezpośrednia reprezentacja myśli w postaci kodu.* Lepiej na przykład zaprezentować argument jako specyficzny typ (np. `Month` czy `Color`) niż bardziej ogólny (np. `int`).
- *Reprezentowanie niezależnych myśli w sposób niezależny.* Na przykład standardowa funkcja `sort()` może poza kilkoma wyjątkami posortować każdy standardowy kontener zawierający elementy dowolnego typu. Koncepcje sortowania, kryteriów sortowania, kontenera oraz typu elementów są wzajemnie niezależne. Gdybyśmy zbudowali „wektor obiektów alokowanych w pamięci wolnej, którego elementy muszą być typu będącego podklasą klasy `Object` oraz z funkcją składową `before()` przeznaczoną do używania przez `vector::sort()`”, otrzymalibyśmy znacznie mniej ogólną funkcję `sort()`, ponieważ przyjelibyśmy założenia na temat sposobu przechowywania, hierarchii klas, dostępnych funkcji składowych, uporządkowania itd.
- *Bezpośrednie reprezentowanie w kodzie związków między różnymi myślami.* Do najpowszechniejszych powiązań, które można bezpośrednio reprezentować w kodzie, należą dziedziczenie (np. obiekt klasy `Circle` jest rodzajem obiektu klasy `Shape`) i parametryzacja (np. `vector<T>` reprezentuje wspólną część wszystkich wektorów, niezależnie od typu elementów).
- *Nie wahaj się łączyć idei wyrażanych w postaci kodu — tylko gdy ma to sens.* Funkcja `sort()` na przykład pozwala na używanie rozmaitych typów elementów i kontenerów, pod warunkiem że obsługują operator `<` (jeśli nie, używa się funkcji `sort()` z dodatkowym argumentem określającym kryterium sortowania) oraz że kontenery obsługują iteratory dostępu wolnego.
- *Wyrażanie prostych myśli w prosty sposób.* Przestrzeganie opisanych wyżej zasad może doprowadzić do powstania zbyt ogólnego kodu. Może się na przykład okazać, że utworzona hierarchia klas ma bardziej skomplikowaną taksonomię (strukturę dziedziczenia), niż komukolwiek potrzebna, lub że każda (na pozór) prosta klasa ma siedem parametrów. Aby nie zmuszać wszystkich użytkowników do mierzenia się z każdym możliwym rodzajem komplikacji, staramy się tworzyć proste wersje, które pozwalają rozwiązać najczęściej spotykane lub najważniejsze problemy. Mamy na przykład funkcję `sort(b,e)`, która domyślnie do sortowania wykorzystuje operator `<` oraz funkcję `sort(b,e,op)` sortującą według kryterium przekazanego jako argument `op`. Moglibyśmy też utworzyć wersję `sort(c)` do sortowania standardowych kontenerów przy użyciu operatora `<` oraz `sort(c,op)` do sortowania standardowych kontenerów przy użyciu kryterium `op`.

22.1.2.4. Poziom abstrakcji

Preferujemy pracę na **najwyższym możliwym poziomie abstrakcji**. Innymi słowy dążymy do ideału, którym jest wyrażanie rozwiązań w tak ogólny sposób, jak to możliwe. 

Rozważmy na przykład, w jaki sposób można reprezentować pozycje w książce telefonicznej (do przechowywania np. w telefonie komórkowym). Można zaprezentować zbiór wartości (nazwa, wartość) w postaci wektora `vector< pair<string, Value_type> >`. Jeśli jednak dostęp do tego zbioru uzyskiwalibyśmy zawsze po nazwie, wyższym stopniem abstrakcji byłby kontener `map<string, Value_type>`, którego zastosowanie pozwoliłoby nam zaoszczędzić sobie pisanie (i debugowania) funkcji dostępowych. Z drugiej strony, `vector< pair<string, Value_type> >` reprezentuje wyższy poziom abstrakcji niż dwie tablice, `string[max]` i `Value_type[max]`, gdzie relacja między łańcuchem a jego wartością nie jest bezpośrednia. Najniższym poziomem abstrakcji może być coś takiego, jak `int` (liczba elementów) plus dwa `void*` (wskazujące jakąś formę reprezentacji znaną programiście, ale nie kompilatorowi). Wszystkie zaproponowane przez nas sugestie można uznać za zbyt niskopoziomowe, ponieważ skupiamy się na reprezentacji par wartości, a nie ich przeznaczeniu. Moglibyśmy zbliżyć się do aplikacji, definiując klasę, która bezpośrednio odzwierciedlałaby jej przeznaczenie. Można by było na przykład utworzyć klasę `Phonebook` z wygodnym interfejsem. Klasę tę można by było zaimplementować przy użyciu dowolnej z zasugerowanych wyżej technik.



Powodem, dla którego wolimy wyższy stopień abstrakcji (gdy mamy do dyspozycji odpowiedni mechanizm abstrakcyjny oraz jeśli używany przez nas język odpowiednio wydajnie ją obsługuje), jest to, że tego rodzaju sformułowania są bliższe naszemu sposobowi rozwiązywania problemów niż rozwiązania wyrażone na poziomie sprzętu komputerowego.

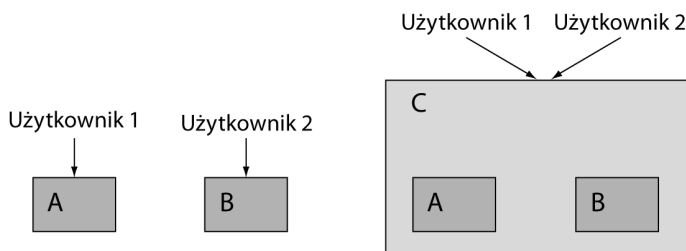
Najczęstszym powodem do zejścia na niski poziom abstrakcji jest chęć podniesienia „wydajności”. Należy to robić tylko wówczas, gdy jest to naprawdę potrzebne (punkt 25.2.2). Zastosowanie instrukcji niższego poziomu (bardziej prymitywnych) nie zawsze poprawia wydajność. Czasami wręcz uniemożliwia optymalizację. Jeśli na przykład użyta zostanie klasa `Phonebook`, możemy wybierać między różnymi implementacjami, np. `string[max]` plus `Value_type[max]` a `map<string, Value_type>`. W niektórych przypadkach lepiej sprawdzi się pierwsze rozwiązanie, a w innych drugie. Naturalnie wydajność nie byłaby znaczącym problemem w aplikacji do zarządzania tylko kontaktami jednej osoby. Kompromis ten staje się interesujący, dopiero gdy przyjdzie przechowywać i zarządzać milionami pozycji. Co ważniejsze, stosowanie niskopoziomowych narzędzi języka programowania zabiera programiście czas, którego później może zabraknąć do naniesienia poprawek (związanych z wydajnością lub innych).

22.1.2.5. Modułowość



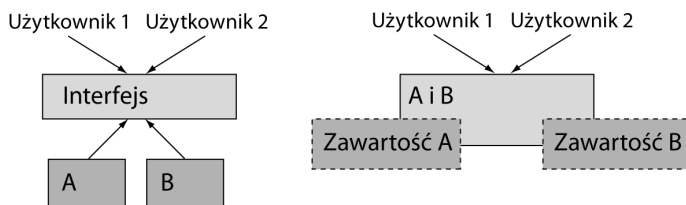
Modułowość jest ideałem. Powinniśmy budować systemy ze składników (funkcji, klas, hierarchii klas, bibliotek itp.), które można tworzyć, rozgryzać i testować osobno. Najlepiej, gdy składniki takie można projektować i implementować w taki sposób, aby nadawały się do użycia w więcej niż jednym programie (wielokrotne wykorzystanie). Technika **wielokrotnego wykorzystania** polega na tworzeniu systemów z gotowych przetestowanych składników, które były już używane gdzieś indziej. Poruszyliśmy ten temat przy opisie klas, hierarchii klas, projektowania interfejsów oraz programowania ogólnego. Większość tego, co piszemy na temat stylów programowania (punkt 22.1.3), ma związek z projektowaniem, implementowaniem i wykorzystaniem potencjalnie nadających się do wielokrotnego użytku składników. Należy zauważyć, że nie każdy składnik da się wykorzystać w więcej niż jednym programie. Niektóre przykłady kodu są zbyt wyspecjalizowane i nie da się ich łatwo poprawić, aby nadawały się do użytku gdzieś indziej.

Modułowość kodu powinna odzwierciedlać ważne logiczne rozróżnienia w aplikacji. Możliwości wielokrotnego wykorzystania kodu nie da się zwiększyć poprzez proste umieszczenie dwóch niezwiązanych ze sobą klas A i B w nadającym się do wielokrotnego użytku składniku C. Wprowadzenie C poprzez złączenie interfejsów A i B komplikuje kod:



Zarówno użytkownik 1, jak i użytkownik 2 używają C. Jeśli nie zajrzy się do wnętrza składnika C, można odnieść wrażenie, że użytkownicy ci odnoszą korzyści ze współdzielenia wspólnego komponentu. Można by było założyć (w tym przypadku błędnie), że do zalet współdzielenia (wielokrotnego wykorzystania) składników należy zaliczyć lepsze testowanie, mniej kodu, większą bazę użytkowników itp. Niestety poza odrobiną zbyt dużego uproszczenia nie jest to rzadkie zjawisko.

Co może pomóc? Może warto byłoby utworzyć wspólny interfejs do A i B:



Powyższe schematy mają za zadanie przywołać na myśl odpowiednio dziedziczenie i parametryzację. W obu przypadkach dostarczony interfejs musi być mniejszy niż proste połączenie interfejsów A i B, aby ćwiczenie było warte uwagi. Innymi słowy, A i B muszą łączyć fundamentalne cechy wspólne, z których użytkownicy mogliby skorzystać. Zauważ, że wróciliśmy do interfejsów (podrozdział 9.7 i punkt 25.4.2) i pośrednio do niezmienników (punkt 9.4.3).


22.1.2.6. Spójność i minimalizm

Spójność i minimalizm to główne ideały dotyczące wyrażania myśli. Można by zatem było je odrzucić, ponieważ dotyczą wyglądu. Niemniej jednak trudno jest elegancko zaprezentować niechlujny projekt, a zatem wymagania spójności i minimalizmu wykorzystać jako kryteria projektowe, które mogą mieć wpływ nawet na najdrobniejsze szczegóły programu:

- Nie dodawaj funkcji, jeśli masz wątpliwości co do jej użyteczności.
- Podobnym funkcjom nadawaj podobne interfejsy (i nazwy), ale tylko jeśli podobieństwo dotyczy fundamentalnych cech.
- Różnym funkcjom nadawaj różne nazwy (i twórz w miarę możliwości różne rodzaje interfejsów), ale tylko jeśli różnice dotyczą fundamentalnych cech.


Spójność nazw, interfejsów i stylu implementacji ułatwiają utrzymanie kodu. Gdy zasady te są zachowane, programista nie musi zapoznawać się z nowymi konwencjami dla każdej większej części systemu. Przykładem tego jest biblioteka STL (rozdziały 20. i 21. oraz punkty B.4 – B.6). Jeśli zachowanie spójności nie jest możliwe (np. z powodu korzystania z przestarzałego kodu lub kodu napisanego w innym języku), dobrym pomysłem może być dostarczenie interfejsu odpowiadającego reszcie programu. Można też pozwolić obcemu (dziwnemu, słabej jakości) kodowi zainfekować wszystkie te części własnego programu, które z niego korzystają.

Jednym ze sposobów na zachowanie minimalizmu i spójności jest skrupulatne (i spójne) dokumentowanie każdego interfejsu. Dzięki temu jest większa szansa, że wszelkie nieścisłości i duplikacje zostaną zauważone. Szczególnie korzystne może być dokumentowanie warunków wstępnych i końcowych oraz niezmienników, a także bardzo rozsądne zarządzanie zasobami i raportowanie błędów. Spójna obsługa błędów i dobre zarządzanie zasobami są niezbędne do zachowania prostoty (podrozdział 19.5).



Niektórzy programiści za swoją najważniejszą zasadę projektowania uważają tzw. KISS (ang. *Keep It Simple, Stupid* — „zachowaj prostotę, głupku”). Spotykaliśmy się nawet z twierdzeniami, że jest to jedyna słuszna zasada projektowania. My jednak wolimy mniej sugestywne sformułowania, jak np. „To co jest proste, niech pozostanie proste” i „Zachowaj prostotę: niech to będzie tak proste, jak to możliwe, ale ani trochę prostsze”. Drugi z tych cytatów należy do Alberta Einsteina. Jest w nim zawarte ostrzeżenie, że istnieje pewna granica upraszczania, której przekroczenie powoduje zniszczenie projektu. Oczywiście należy w tej sytuacji zadać pytanie: „Dla kogo ma być to proste i w porównaniu z czym?”.

22.1.3. Style i paradygmaty



Projektując i implementując program, należy dbać o spójność stylu. Język C++ umożliwia programowanie w czterech najważniejszych stylach, które można uznać za fundamentalne:

- programowanie proceduralne,
- abstrakcja danych,
- programowanie obiektowe,
- programowanie ogólne.

Nazywa się je, czasami nieco pompatycznie, paradygmatami programowania. Istnieje wiele więcej takich paradygmatów, jak programowanie logiczne (ang. *logic programming*), funkcyjne (ang. *functional programming*), programowanie regułowe (ang. *rule-based programming*), programowanie ograniczeniowe (ang. *constraints-based programming*) oraz programowanie aspektowe (ang. *aspect-based programming*). Jednak nie są one bezpośrednio obsługiwane przez język C++. Nie możemy napisać o „wszystkim” w książce dla początkujących, dlatego zdobycie wiedzy na temat paradygmatów i masę szczegółów, które musieliśmy pominąć, pozostawiamy czytelnikowi jako pracę na przyszłość. Na następnej stronie znajduje się krótki opis czterech obsługiwanych przez C++ paradygmatów:

- *Programowanie proceduralne* — polega na budowaniu programów z funkcji działających na argumentach. Do przykładów zastosowania tego paradygmatu należą m.in. biblioteki funkcji matematycznych, jak np. `sqrt()` i `cos()`. Obsługa tego stylu programowania w języku C++ wyraża się poprzez funkcje (rozdział 8.). Bardzo ważna jest tu możliwość

przekazywania funkcjom argumentów przez wartość, referencję i stałą referencję. Dane często są zorganizowane w struktury reprezentowane przez konstrukcję `struct`. Bezpośrednie techniki abstrakcji (np. prywatne dane składowe i funkcje składowe klas) nie są wykorzystywane. Należy zauważyć, że ten styl programowania — i funkcji — jest integralną częścią wszystkich pozostałych stylów.

- *Abstrakcja danych* — polega na utworzeniu zestawu typów odpowiednich dla danego obszaru zastosowań, a następnie napisaniu programu, który je wykorzystuje. Klasycznym przykładem są macierze (podrozdziały 24.3 – 24.6). Bezpośrednie ukrywanie danych (np. stosowanie prywatnych danych składowych klasy) jest często wykorzystywaną techniką. Do popularnych przykładów można zaliczyć standardowe typy `string` i `vector`, które pokazują silną zależność między abstrakcją danych a parametryzacją charakterystyczną dla programowania ogólnego. Nazywa się to abstrakcją, ponieważ dostęp do typu odbywa się poprzez jego interfejs, a nie jest uzyskiwany bezpośrednio do implementacji.
- *Programowanie obiektowe* — polega na zorganizowaniu typów w hierarchie, aby bezpośrednio w kodzie wyrazić występujące między nimi powiązania. Klasycznym przykładem jest hierarchia `Shape` z rozdziału 14. Ten styl jest bez wątpienia przydatny, gdy między typami występują rzeczywiste fundamentalne powiązania. Jest jednak silna tendencja do nadużywania tego paradygmatu, tzn. spotyka się hierarchie zbudowane z typów, których nie łączą żadne fundamentalne cechy. Gdy ktoś tworzy podklasy, należy spytać, dlaczego to robi. Co chce w ten sposób wyrazić? Co takie rozróżnienie bazowy-derywowany daje mi w danym przypadku?
- *Programowanie ogólne* — polega na „przerabianiu” konkretnych algorytmów w celu wzniesienia ich na wyższy poziom abstrakcji poprzez dodanie parametrów, które pozwalają na zmianę niektórych rzeczy bez wpływu na podstawową funkcję algorytmu. Prosty przykład takiej „przeróbki” jest funkcja `high()` opisana w rozdziale 20. Algorytmy `find()` i `sort()` z biblioteki STL to klasyczne algorytmy wyrażone w bardzo ogólnej formie przy użyciu programowania ogólnego. Zobacz rozdziały 20. i 21. oraz poniższy przykład.

Wszyscy razem! O stylach programowania (paradygmatach) często mówi się w taki sposób, jakby były niepowiązanymi wzajemnie alternatywnymi technikami — stosuje się albo programowanie ogólne, albo obiektowe. Aby rozwiązywać problemy w najlepszy możliwy sposób, należy jednak stosować mieszankę stylów. Piszac „najlepszy możliwy sposób”, mamy na myśli takie rozwiązanie, które jest łatwe do czytania, napisania i utrzymania oraz wystarczająco wydajne. Rozważmy przykład — początki klasycznego przykładu klasy `Shape` sięgają języka Simula (punkt 22.2.4). Często traktuje się go jako przykład programowania obiektowego. Pierwsze rozwiązanie mogłoby wyglądać następująco:


```
void draw_all(vector<Shape*>& v)
{
    for(int i = 0; i<v.size(); ++i) v[i]→draw();
}
```



Ten kod rzeczywiście wygląda na „obiektowy”. Przy jego pisaniu wykorzystano hierarchię klas i wywołanie funkcji wirtualnej `draw()`, które znajdzie odpowiednią jej wersję dla każdego konkretnego kształtu (`Shape`). Tzn. dla obiektu typu `Circle` zostanie wywołana funkcja `Circle::draw()`, a dla `Open_polyline` — `Open_polyline::draw()`. Jednak `vector<Shape*>` to konstrukcja z programowania ogólnego — wykorzystuje się w niej parametr (typ elementu), który zostaje konkretnie określony w czasie kompilacji. Możemy to podkreślić, stosując prosty algorytm z biblioteki standardowej do wyrażania iteracji przez elementy:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(), mem_fun(&Shape::draw));
}
```

Trzeci argument funkcji `for_each()` jest funkcją wywoływaną dla każdego elementu sekwencji określonej przez dwa pierwsze argumenty (dodatek B.5.1). Teraz to trzecie wywołanie funkcji należy traktować jako zwykłą funkcję (lub obiekt funkcyjny) wywoływaną za pomocą składni `f(x)`, a nie funkcję składową wywoływaną za pomocą składni `p->f()`. Użyliśmy więc funkcji `mem_fun()` z biblioteki standardowej (punkt B.6.2), aby powiedzieć, że tak naprawdę chcemy wywołać funkcję składową (funkcję wirtualną `Shape::draw()`). Chodzi nam o to, że funkcje `for_each()` i `mem_fun()`, będąc szablonami, nie są w rzeczywistości „bardzo obiektywne”. Wyrażnie należałoby je zaliczyć do programowania ogólnego. Co ciekawsze, funkcja `mem_fun()` jest niezależną funkcją (szablonową) zwracającą obiekt klasy. Innymi słowy, można ją z łatwością zaklasyfikować jako abstrakcję danych (brak dziedziczenia), a nawet przykład programowania proceduralnego (brak ukrywania danych). Można zatem powiedzieć, że w tym jednym wierszu kodu zostały zastosowane kluczowe aspekty wszystkich czterech podstawowych stylów programowania.



Ale po co mielibyśmy pisać drugą wersję przykładu „narysuj wszystkie figury (`Shape`)”? Zasadniczo robi to samo, co poprzednia wersja, a nawet wymaga użycia mniejszej liczby znaków! Można się sprzeczać, że wyrażenie pętli za pomocą funkcji `for_each()` jest „bardziej oczywiste i mniej podatne na błędy” niż pisanie pętli `for`, ale wielu osób argument ten nie przekonuje. Lepszym argumentem jest to, że „funkcja `for_each()` wyraża, co ma zostać zrobione (iteracja przez sekwencję), a nie jak należy to zrobić”. Dla wielu jednak najbardziej przekonującym argumentem jest przydatność — wskazuje drogę do uogólnienia (w najlepszej tradycji programowania ogólnego), które pozwala rozwiązać więcej problemów. Czemu figury są przechowywane w wektorze? Czemu nie w liście? Czemu nie jako ogólna sekwencja? Możemy więc napisać trzecią (jeszcze ogólniejszą) wersję:

```
template<class Iter> void draw_all(Iter b, Iter e)
{
    for_each(b, e, mem_fun(&Shape::draw));
}
```

To będzie działać ze wszystkimi rodzajami sekwencji i figur. Tę funkcję można nawet wywołać na rzecz elementów tablicy figur:

```
Point p{0,100};
Point p2{50,50};
Shape* a[] = { new Circle(p,50), new Triangle(p,p2,Point(25,25)) };
draw_all(a,a+2)
```


Ograniczając możliwość zastosowania tylko do kontenerów, możemy napisać prostszą wersję:

```
template<class Cont> void draw_all(Cont& c)
{
    for (auto& p : c) p->draw();
}
```

A w C++14 możemy nawet posłużyć się koncepcjami (punkt 19.3.3):

```
void draw_all(Container& c)
{
    for (auto& p : c) p->draw();
}
```

Ten kod nadal jest wyraźnie obiektowy, ogólny i bardzo podobny do zwykłego kodu proceduralnego. W hierarchii klas i implementacji poszczególnych kontenerów bazuje na abstrakcji danych. Z powodu braku lepszego określenia programowanie przy użyciu odpowiedniej mieszanki stylów nazwano **programowaniem wieloparadygmatowym** (ang. *multi-paradigm programming*). Ja jednak doszedłem do wniosku, że należy to nazywać po prostu programowaniem: „paradygmaty” odzwierciedlają ograniczony obraz tego, jak można rozwiązać problemy, oraz słabości języków, w których wyrażamy nasze rozwiązania. Wróżę świetlaną przyszłość programowaniu przy użyciu ulepszonej techniki, lepszych języków programowania i narzędzi wspomagających.



22.2. Krótka historia języków programowania

Na samym początku programiści ręcznie wybijali dłutem zera i jedynki w kamieniu. Cóż, może nie do końca tak. Zaczniemy prawie od samego początku i krótko przedstawimy niektóre kamienie milowe w historii języków programowania ze szczególnym uwzględnieniem ich związku z językiem C++.

Istnieje masa języków programowania. Średnio w ciągu dekady powstaje około 2000 nowych języków i mniej więcej tyle samo „przestaje istnieć”. Tekst tego podrozdziału obejmuje prawie 60 lat i zawiera wzmianki tylko o 10 językach. Więcej informacji można znaleźć na stronie <http://research.ihost.com/hopl/HOPL.html>. Można na niej znaleźć artykuły trzech konferencji HOPL ACM SIGPLAN na temat historii języków programowania (ang. *History of Programming Languages* — historia języków programowania). Są to dokładnie przejrzane przez naukowców publikacje, a więc stanowią znacznie bardziej wiarygodne źródło informacji niż jakiekolwiek przeciętne internetowe źródło. Wszystkie opisywane przez nas tutaj języki zostały zaprezentowane na konferencji HOPL. Warto zauważyć, że jeśli w wyszukiwarce internetowej wpisze się tytuł znanej publikacji, istnieje duże prawdopodobieństwo jej znalezienia. Ponadto większość wymienionych tu naukowców ma własne strony internetowe, na których udostępniają dodatkowe materiały na temat swojej pracy.



Nasze prezentacje języków są z konieczności bardzo krótkie — każdy z wymienionych — i setki niewymienionych — zasługuje na poświęcenie mu całej książki. Starannie też wybraлиśmy informacje, które podajemy na temat każdego języka. Mamy nadzieję, że zachęci Cię to do dalszych poszukiwań informacji, a nie doprowadzi do konkluzji: „a więc to wszystko, co można powiedzieć o języku X”. Należy pamiętać, że każdy opisany tu język programowania stanowił bardzo duże osiągnięcie i wniósł poważny wkład do naszego świata. Nie ma szans, aby należycie opisać wszystkie te języki w tak krótkim tekście — ale nie wspomnieć o żadnym

z nich byłoby jeszcze gorsze. Chcielibyśmy przedstawić przykładowe fragmenty kodu w każdym z nich, ale niestety to nie jest miejsce na takie rzeczy (zobacz 5. i 6. zadanie pracy domowej).

Zbyt często dzieje się tak, że artefakty (np. języki programowania) przedstawia się po prostu jako to, czym są, lub wynik jakiegoś bliżej nieokreślonego procesu. Jest to niepoprawne interpretowanie historii. Język programowania to zazwyczaj — było tak zwłaszcza w początkowych latach, kiedy dyscyplina programowania dopiero się kształtowała — wypadkowa ideałów, pracy, osobistych preferencji i zewnętrznych ograniczeń jednej (lub częściej wielu) osoby. Dlatego szczególnie nacisk kładziemy na sylwetki najważniejszych osób zaangażowanych w powstawanie języków. IBM, Bell Labs, Cambridge University itp. nie projektują języków programowania. Robią to pracujące w tych instytucjach osoby, najczęściej ściśle współpracując z przyjaciółmi i kolegami z pracy.

Warto zwrócić uwagę na pewne zjawisko, które często wypacza nasze wyobrażenie o historycznych faktach. Zdjęcia słynnych naukowców i inżynierów są często robione w czasach, gdy są oni u szczytu sławy, należą do narodowych akademii, są członkami Towarzystwa Królewskiego, rycerzami świętego Jana, laureatami nagrody Turinga itd. — innymi słowy, gdy są o wiele lat starsi, niż gdy dokonywali swoich największych odkryć. Prawie wszyscy są lub należeli do najbardziej produktywnych postaci w swoim środowisku do późnych lat życia. Sięgając jednak do początków funkcji swojego ulubionego języka i technik programistycznych, spróbuj sobie jednak wyobrazić młodego mężczyznę (w nauce i inżynierii wciąż jest o wiele za mało kobiet), który zastanawia się, czy wystarczy mu pieniędzy, aby zaprosić dziewczynę do przyzwoitej restauracji, lub rodzica próbującego tak wykombinować, aby przedłożyć swoją najważniejszą publikację w takim czasie i miejscu, by móc połączyć wyjazd z rodzinnym wypoczynkiem. Szare brody, łysiejące głowy i zaniedbane ubrania pojawiają się wiele lat później.

22.2.1. Pierwsze języki



Gdy od roku 1949 zaczęły pojawiać się pierwsze „nowoczesne” elektroniczne komputery przechowujące programy, każdy z nich miał swój własny język. Między wyrażeniem algorytmu (np. obliczającego orbitę planety) a instrukcjami maszynowymi była zależność jeden do jednego. Oczywiście naukowiec (najczęstszymi użytkownikami byli naukowcy) miał notatki zawierające wzory matematyczne, ale program składał się z listy instrukcji maszynowych. Pierwsze prymitywne listy zawierały liczby dziesiętne lub ósemkowe — dokładnie odpowiadały swojej reprezentacji w pamięci komputera. Później pojawiły się asemblery i autokody, a więc opracowano języki, w których instrukcje maszynowe i maszyny (np. rejestry) miały symboliczne nazwy. Wówczas programista mógł napisać „LD R0 123”, aby załadować zawartość fragmentu pamięci o adresie 123 do rejestru 0. Jednak każda maszyna miała własny zestaw instrukcji i język.



Przedstawicielem projektantów języków programowania z tamtych czasów jest David Wheeler z University of Cambridge Computer Laboratory. Napisał on w 1949 roku pierwszy na świecie prawdziwy program, który działał na komputerze przechowującym programy (był to program tabliczki kwadratów, który opisaliśmy w punkcie 4.4.2.1). Jest jednym z około dziesięciu ludzi, którzy twierdzą, że napisali pierwszy kompilator (dla specyficznego dla konkretnej maszyny autokodu). Wynałazł wywołanie funkcji (tak, nawet coś tak na pozór prostego musiało zostać kiedyś wynalezione). W 1951 roku napisał błyskotliwy artykuł na temat projektowania bibliotek. Artykuł ten wyprzedzał czas swojego powstania o około 20 lat! Jest współautorem, obok Maurice'a Wilkesa (poszukaj informacji o nim) i D. J. Gilla, pierwszej na świecie książki o programowaniu. Jako pierwszy uzyskał stopień doktora nauk z dziedziny informatyki (na uniwersytecie Cambridge w 1951 roku). Dużo wniósł do prac nad sprzętem (architektury pamięci cache i wczesne sieci lokalne) i do algorytmiki (np. algorytm szyfrowania TEA [punkt 25.5.6] i transformata Burrowsa-Wheelera [algorytm kompresji używany w bzip2]). Tak się składa, że był on też prowadzącym pracy doktorskiej Bjarne Stroustrupa — informatyka to jeszcze młoda dyscyplina. Część swoich najważniejszych dokonań Wheeler osiągnął, będąc doktorantem. Później został profesorem na uniwersytecie Cambridge i członkiem Towarzystwa Królewskiego w Londynie.

Źródła

Burrows M., Wheeler David, „A Block Sorting Lossless Data Compression Algorithm”, Technical Report 124, Digital Equipment Corporation, 1994.

Campbell-Kelly Martin, „David John Wheeler”, Biographical Memoirs of Fellows of the Royal Society, 52, 2006 (biografia jego dokonań technicznych).

EDSAC: <http://en.wikipedia.org/wiki/EDSAC>.

Knuth Donald, *Sztuka programowania*, WNT; szukaj hasła „David Wheeler” w indeksie każdego tomu.

TEA: http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm.

Wheeler D. J., „The Use of Sub-routines in Programmes”, Proceedings of the 1952 ACM National Meeting (to jest ten artykuł na temat projektowania bibliotek z 1951 roku).

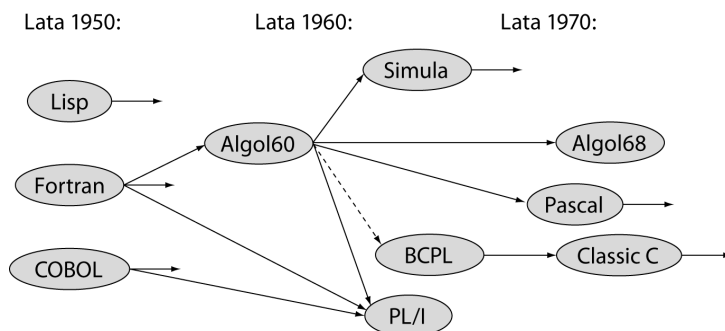
Wilkes M. V., Wheeler D., Gill D. J., *Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Press, 1951; wydanie drugie, 1957 (pierwsza książka o programowaniu).

Witryna Bzip2: www.bzip.org/.

Witryna Cambridge Ring: <http://koo.corpus.cam.ac.uk/projects/earlyatm/cr82>.

22.2.2. Korzenie nowoczesnych języków programowania

Poniżej znajduje się schemat przedstawiający wczesne języki programowania:



Wymienione na powyższym rysunku języki są ważne, ponieważ z jednej strony były (i niektóre nadal są) powszechnie używane, a z drugiej są przodkami innych ważnych języków programowania — często bezpośrednich spadkobierców występujących pod tą samą nazwą. W tym podrozdziale opiszemy trzy wczesne języki — Fortran, COBOL oraz Lisp. Większość nowoczesnych języków programowania wywodzi się właśnie od nich.

22.2.2.1. Fortran



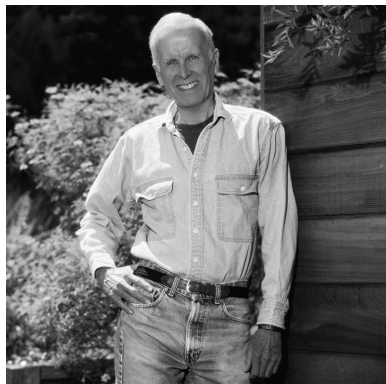
Prawdopodobnie najważniejszym wydarzeniem w historii rozwoju języków programowania było pojawienie się w 1956 roku języka Fortran. Jego nazwa jest akronimem utworzonym ze słów „Formula Translation”. Ideą, która przyświecała jego twórcom, było generowanie wydajnego kodu maszynowego z notacji zaprojektowanej dla ludzi, a nie maszyn. Jako model notacji tego języka posłużyło to, czym naukowcy i inżynierowie posługiwali się do rozwiązywania problemów przy użyciu matematyki, a nie instrukcje maszynowe udostępniane przez (wówczas będące nowością) komputery elektroniczne.

Z punktu widzenia naszych czasów Fortran można traktować jako pierwszą próbę bezpośredniego zaprezentowania dziedziny zastosowań w kodzie. Język ten pozwalał na wykonywanie obliczeń algebry liniowej w taki sam sposób, jak opisywano w książkach. Udostępniał tablice, pętle oraz standardowe funkcje matematyczne (przy użyciu standardowej notacji matematycznej, jak $x+y$ i $\sin(x)$). Utworzono standardową bibliotekę funkcji matematycznych, mechanizmy wejścia i wyjścia oraz była możliwość tworzenia przez użytkownika własnych funkcji i bibliotek.

Notacja języka Fortran była w dużym stopniu niezależna od maszyny, dzięki czemu napisany w nim kod często dało się wykorzystywać na różnych komputerach po wprowadzeniu niewielkich zmian. Stanowiło to **ogromny** skok do przodu. W ten sposób Fortran zyskał sobie miejsce w historii jako pierwszy wysokopoziomowy język programowania.

Uważano za rzecz zupełnie niezbędną, aby kod maszynowy generowany z kodu w języku Fortran był bliski optymalnemu pod względem wydajności. Wówczas maszyny miały wielkość pokoju, były strasznie drogie (przekraczały wielokrotnie roczną pensję zespołu dobrych programistów), według dzisiejszych standardów śmiesznie wolne (wykonywały około 100 000 instrukcji na sekundę) oraz miały absurdalnie mało pamięci (w granicach 8 KB). A jednak udało się pisać przydatne programy na tych maszynach i poprawianie notacji (prowadzące do lepszej wydajności programisty i przenośności kodu) nie mogło temu zaszkodzić.

Język Fortran odniósł ogromne sukcesy na polu, dla którego został utworzony, a więc obliczeń naukowych i inżynierskich i od chwili powstania ciągle ewoluuje. Najważniejsze jego wersje to II, IV, 77, 90, 95 oraz 03. Cały czas toczy się dyskusja, czy więcej osób używa języka Fortran77, czy Fortran90.



Pierwsze definicje i pierwsza implementacja języka Fortran zostały opracowane w firmie IBM przez zespół prowadzony przez Johna Backusa: „Nie wiedzieliśmy, czego chcemy, i nie wiedzieliśmy, jak to zrobić. To jakoś tak nam wyszło”. Skąd miałby wiedzieć, skoro nikt wcześniej tego nie robił. Po drodze opracowali lub odkryli podstawową strukturę kompilatorów — analiza leksykalna, analiza składniowa, analiza semantyczna oraz optymalizacja. Fortran do dziś jest wiodącym narzędziem do optymalizowania obliczeń. Jedną z rzeczy, które pojawiły się (po pierwszej wersji Fortrana), była notacja do opisywania gramatyk: *Backus-Naur Form* (notacja Backusa-Naura, lub w skrócie BNF). Po raz pierwszy użyto jej w języku Algol60 (punkt 22.2.3.1) i stosuje się ją obecnie w większości nowych języków programowania. Jedną z wersji tej gramatyki wykorzystaliśmy w rozdziałach 6. i 7.

Wiele lat później John Backus zapoczątkował całkiem nową gałąź dyscypliny programowania (programowanie funkcyjne), w której stosowane jest matematyczne podejście do programowania w przeciwieństwie do maszynowego patrzenia na program odczytujący i zapisujący lokalizacje w pamięci. Należy zauważyć, że w czystej matematyce nie istnieją takie pojęcia, jak przypisanie czy akcja (ang. *action*). W zamian po prostu stwierdza się, że coś musi być prawdą na podstawie określonych warunków. Częściowo korzenie programowania funkcyjnego tkwią w języku Lisp (punkt 22.2.2.3), a niektóre jego zasady znalazły miejsce w bibliotece STL (rozdział 21.).



Źródła

Backus John, *Can Programming Be Liberated from the von Neumann Style?*, „Communications of the ACM”, 1977 (wykład Backusa w związku z otrzymaniem nagrody Turinga).

Backus John, *The History of FORTRAN I, II, and III*, „ACM SIGPLAN Notices”, 13, nr 8, „Special Issue: History of Programming Languages Conference”, 1978.

Hutton Graham, *Programming in Haskell*, Cambridge University Press, 2007.

ISO/IEC 1539, *Programming Languages — Fortran* (standard Fortran 95).

Paulson L. C., *ML for the Working Programmer*, Cambridge University Press, 1991.

22.2.2.2. COBOL

Język COBOL (ang. *Common Business-Oriented Language*) był (i w niektórych przypadkach nadal jest) tym dla programistów biznesowych, czym Fortran był (i w niektórych przypadkach nadal jest) dla programistów naukowych. Nacisk w nim położono na manipulowanie danymi:

- kopiowanie,
- przechowywanie i pobieranie (przechowywanie rekordów),
- drukowanie (raportów).

Wykonywanie obliczeń (często słusznie biorąc pod uwagę dziedzinę zastosowań języka COBOL) postrzegano jako coś drugorzędnego. Żywiono nadzieję, że język ten stanie się tak zbliżony do „biznesowej angielszczyzny”, że managerowie sami będą mogli pisać programy, co wyeliminowałoby potrzebę zatrudniania programistów. Nadzieje te były wielokrotnie wyrażane od wielu lat przez managerów chcących ciąć koszty ponoszone na programowanie. Nigdy nie udało się nawet zbliżyć do realizacji tego celu.



Pierwsza wersja języka COBOL została opracowana w latach 1959 – 1960 przez komisję (CODASYL) działającą z ramienia departamentu obrony USA i grupę największych producentów komputerów. Celem tych prac było zaspokojenie potrzeb związanych z przetwarzaniem związanym z biznesem. Projekt zbudowano bezpośrednio na bazie języka FLOW-MATIC wynalezionej przez Grace Hopper. Jednym z jej osiągnięć było wprowadzenie składni zbliżonej do języka angielskiego (w odróżnieniu od stosowanej w języku Fortran i dominującej także obecnie notacji matematycznej). Podobnie jak Fortran i inne odnoszące sukcesy języki, COBOL wielokrotnie ewoluował. Najważniejsze jego wersje to 60, 61, 65, 68, 70, 80, 90 oraz 04.

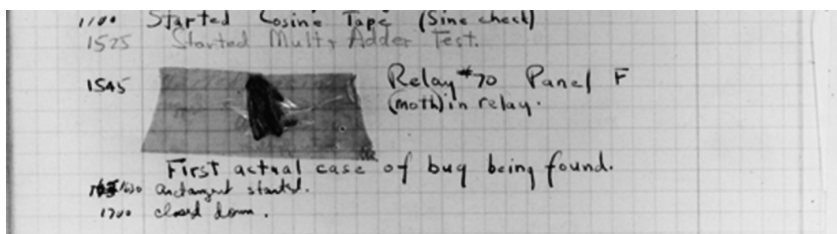
Grace Murray Hopper uzyskała doktorat z matematyki na uniwersytecie Yale. W czasie drugiej wojny światowej pracowała dla amerykańskiego wojska nad pierwszymi komputerami. Wróciła do wojska po kilku latach obecności w rodzącym się przemyśle komputerowym:



„Kontradmirał dr Grace Murray Hopper (U.S. Navy) była niezwykłą kobietą, która doskonale poradziła sobie z wyzwaniem, jakim było programowanie pierwszych komputerów. Będąc wiodącą postacią w dyscyplinie rozwoju oprogramowania, wniosła niemały wkład w przejście od prymitywnych technik programistycznych do zaawansowanych kompilatorów. Wyznawała zasadę, że to, iż „zawsze tak robiliśmy”, nie jest powodem, aby dalej tak robić”.

— Anita Borg, konferencja „Grace Hopper Celebration of Women in Computing”, 1994.

Mówi się, że to Grace Hopper jako pierwsza użyła słowa „Bug” na określenie błędu w programie komputerowym. Z pewnością należała do pierwszych użytkowników tego słowa w takim znaczeniu i udokumentowała to:



Jak widać, robak (ang. *bug*) rzeczywiście istniał i miał bezpośredni wpływ na sprzęt. Nowoczesne robale pojawiają się w oprogramowaniu i są mniej atrakcyjne wizualnie.

Źródła

Biografia G. M. Hopper: <http://tergestesoft.com/~eddysworld/hopper.htm>.

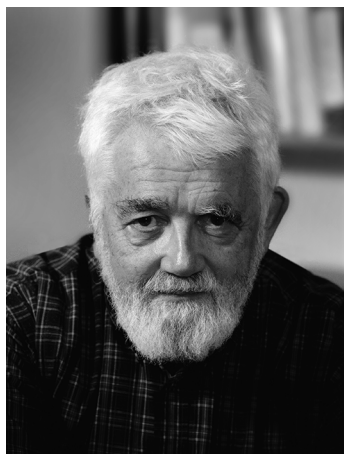
ISO/IEC 1989:2002, *Information Technology — Programming Languages — COBOL*.

Sammet Jean E., *The Early History of COBOL*, „ACM SIGPLAN Notices”, 13, nr 8, „Special Issue: History of Programming Languages Conference”, 1978.

22.2.2.3. Lisp

Język Lisp zaprojektował w 1958 roku pracujący w MIT John McCarthy. Miał to być język do przetwarzania list powiązanych i przetwarzania symbolicznego (stąd nazwa Lisp, która jest akronimem angielskich słów „LISt Processing” — przetwarzanie list). Pierwotnie Lisp był (i nadal często jest) językiem interpretowanym, w odróżnieniu od języków kompilowanych. Istnieją dziesiątki (najprawdopodobniej nawet setki) dialektów tego języka. W istocie mówi się nawet, że „Lisp jest domyślnie pluralistyczny”. Obecnie najpopularniejsze dialekty to Common Lisp i Scheme. Ta rodzina języków stanowi filar badań nad sztuczną inteligencją (choć dostarczane produkty często są napisane w językach C i C++). Jedną z głównych inspiracji wykorzystanych przez twórcę tego języka był matematyczny rachunek lambda.

Języki Fortran i COBOL zostały zaprojektowane do rozwiązywania realnych problemów w dziedzinach, do których były przeznaczone. Natomiast użytkownicy języka Lisp skupili się bardziej na samym programowaniu jako takim i elegancji programów. Działania te często kończyły się powodzeniem. Lisp stał się pierwszym językiem, którego definicję oddzielono od sprzętu, a składnię oparto na formie matematyki. Jeśli Lisp miał wyznaczoną jakąś dziedzinę zastosowań, to trudno ją precyzyjnie określić — sztuczna inteligencja czy przetwarzanie symboliczne nie znajdują tak dobrych odpowiedników w życiu codziennym, jak przetwarzanie biznesowe i programowanie dla celów naukowych. Echa języka Lisp (i ze skupionej wokół niego społeczności) można usłyszeć w wielu nowoczesnych językach programowania, zwłaszcza funkcyjnych.



John McCarthy uzyskał stopień licencjata w dziedzinie matematyki w California Institute of Technology, a doktorat z matematyki uzyskał na Uniwersytecie Princeton. Jak widać, wśród projektantów języków programowania można znaleźć wielu wybitnych matematyków.

Po niezapomnianym okresie pracy w MIT McCarthy przeniósł się w 1962 roku do Stanford, aby pomóc w budowie tamtejszego laboratorium sztucznej inteligencji. Powszechnie przypisuje mu się autorstwo terminu *artificial intelligence* (sztuczna inteligencja) — nazwy dyscypliny, w której wniósł duży wkład.

Źródła

Abelson Harold, Sussman Gerald J., *Structure and Interpretation of Computer Programs, Second Edition*, MIT Press, 1996.

ANSI INCITS 226-1994 (wcześniej ANSI X3.226:1994), *American National Standard for Programming Language — Common LISP*.

McCarthy John, *History of LISP*, „ACM SIGPLAN Notices”, 13, nr 8, „Special Issue: History of Programming Languages Conference”, 1978.

Steele Guy L. Jr., *Common Lisp: The Language*, Digital Press, 1990.

Steele Guy L. Jr., Gabriel Richard, *The Evolution of Lisp*, Proceedings of the ACM History of Programming Languages Conference (HOPL-2), „ACM SIGPLAN Notices”, 28, nr 3, 1993.

22.2.3. Rodzina Algol

Pod koniec lat 50. pojawiły się odczucia, że programowanie robi się zbyt skomplikowane, zbyt przypadkowe i za mało naukowe. Przeczutowano, że jest za dużo języków programowania oraz że istniejące już języki zostały skłcone ze zbyt małą dbałością o ogólność i bez zastosowania fundamentalnych zasad. Problem ten od tamtej pory był poruszany jeszcze wiele razy, chociaż pod skrzydłami IFIP (ang. *International Federation of Information Processing*) zaczęła działać grupa ludzi, którzy w ciągu kilku lat opracowali język, który zrewolucjonizował sposób myślenia o językach i ich definiowaniu. Większość nowoczesnych języków programowania (między innymi C++) dużo tym pracom zawdzięcza.

22.2.3.1. Algol60



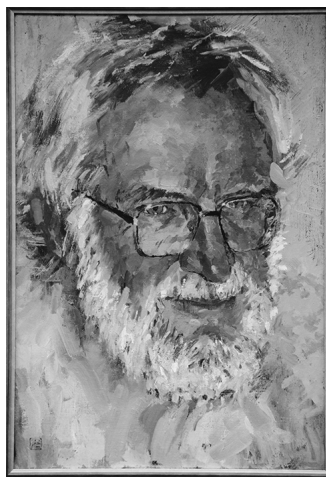
Algol (ang. *ALGOritmic Language*) jest wynikiem prac grupy IFIP 2.1. i stanowił przełom w nowoczesnych technikach projektowania języków programowania:

- zakres leksykalny,
- definicja języka przy użyciu gramatyki,
- wyraźne oddzielenie reguł syntaktycznych od semantycznych,
- wyraźne oddzielenie definicji języka od jego implementacji,
- systematyczne wykorzystywanie (statycznych, a więc czasu kompilacji) typów,
- bezpośrednia obsługa programowania strukturalnego.

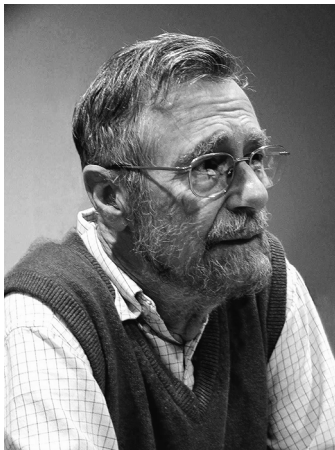
Samo pojęcie „języka do celów ogólnych” pojawiło się wraz z językiem ALGOL. Przed jego pojawieniem się języki programowania miały specjalistyczne zastosowania naukowe (np. Fortran), biznesowe (np. COBOL), były tworzone do przetwarzania list (np. Lisp), symulacji itd. Z tych wszystkich języków Algolowi najbliższe do Fortrana.

Niestety Algol60 nigdy nie zyskał popularności poza środowiskiem akademickim. Przez wiele osób z branży był uważany za „bardzo dziwaczny”, programiści używający języka Fortran uważali, że jest za wolny, użytkownicy języka COBOL zarzucali mu, iż „nie wspiera przetwarzania biznesowego”, dla programistów Lispa był „za mało elastyczny”, dla większości osób w branży informatycznej (włącznie z managerami, którzy decydowali o inwestowaniu w narzędzia) był „zbyt akademicki”, a dla wielu amerykańców „zbyt europejski”. Większość kierowanej pod jego adresem krytyki była słuszna. Na przykład w języku Algol60 nie było mechanizmów wejścia i wyjścia! Jednakże podobne krytyczne uwagi można było sformułować wobec każdego ówczesnego języka programowania — a Algol wyznaczył nowy standard dla wielu obszarów badań.

Jedyny problem z Algolem polegał na tym, że nikt nie wiedział, jak go zaimplementować. Rozwiązał go zespół programistów pracujący pod kierownictwem Petera Naura (redaktora raportu o Algolu60) i Edsgera Dijkstry:

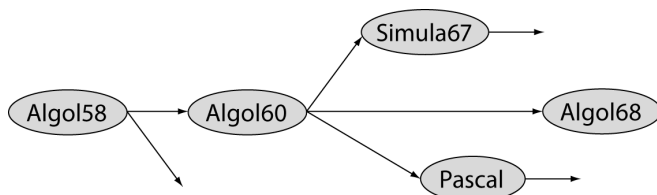


Peter Naur studiował (astronomię) na Uniwersytecie Kopenhaskim i pracował w kopenhaskiej Wyższej Szkole Technicznej (DTU) oraz dla duńskiego producenta komputerów Regnecentralen. Wcześniej (lata 1950 – 1951) nauczył się programować w laboratorium komputerowym w Cambridge w Anglii (w Danii nie było wówczas jeszcze komputerów) i później zrobił wybitną karierę na polu akademickim i przemysłowym. Jest jednym z wynalazców notacji Backusa-Naura, która służy do opisywania gramatyk, oraz jako pierwszy zaproponował formalne rozumowanie na temat programów (Bjarne Stroustrup po raz pierwszy — około roku 1971 — nauczył się stosowania niezmienników z jego artykułów). Naur zawsze z rozsądkiem patrzył na przetwarzanie komputerowe i nigdy nie zapominał o ludzkim aspekcie tego wszystkiego. Jego późniejsze prace można przyporządkować do dziedziny filozofii (choć sam Naur uważał, że konwencjonalna akademicka filozofia to stek bzdur). Był pierwszym profesorem w dziedzinie Datalogi na Uniwersytecie Kopenhaskim (duńskie słowo *datalogi* najlepiej przetłumaczyć jako **informatyka** — Naur nie znosił określenia *computer science*, ponieważ w centrum zainteresowania dyscypliny określanej po angielsku jako *computing* nie leżą komputery).



Kolejną wielką postacią w świecie komputerów był Edsger Dijkstra. Studiował fizykę w Lejdzie, ale swoje wczesne prace dotyczące komputerów wykonywał w Mathematisch Centrum w Amsterdamie. Później pracował w wielu różnych miejscach, między innymi na politechnice w Eindhoven, dla Burroughs Corporation oraz University of Texas (w Austin). Poza swoją nowatorską pracą nad językiem Algol, Dijkstra jako jeden z pierwszych naukowców był gorącym zwolennikiem stosowania w programowaniu logiki matematycznej, algorytmów oraz jednym z projektantów i implementatorów systemu operacyjnego THE — jednego z pierwszych, które obsługiwały współbieżność. Akronim THE pochodzi od słów Technische Hogeschool Eindhoven — to nazwa uczelni, dla której wówczas pracował Dijkstra. Jego najbardziej znana publikacja to najprawdopodobniej *Go-To Statement Considered Harmful*, w której przekonująco przedstawił problemy wywoływane przez brak struktury w przepływie sterowania.

Drzewo rodziny Algol jest imponujące:



Zwróć uwagę na języki Simula67 i Pascal. Są to przodkowie wielu (możliwe, że większości) nowoczesnych języków programowania.

Źródła

Dijkstra Edsger, *Algol 60 Translation: An Algol 60 Translator for the x1 and Making a Translator for Algol 60*, Report MR 35/61, Mathematisch Centrum (Amsterdam), 1961.

Dijkstra Edsger, *Go-To Statement Considered Harmful*, „Communications of the ACM”, 11, nr 3, 1968.

Lindsey C. H., *The History of Algol68*, Proceedings of the ACM History of Programming Languages Conference (HOPL-2), „ACM SIGPLAN Notices”, 28, nr 3, 1993.

Naur Peter (red.), *Revised Report on the Algorithmic Language Algol 60*, A/S Regnecentralen (Copenhagen), 1964.

Naur Peter, *Proof of Algorithms by General Snapshots*, „BIT”, 6, 1966, str. 310 – 316; prawdopodobnie pierwszy artykuł na temat dowodzenia poprawności programów.

Naur Peter, *The European Side of the Last Phase of the Development of ALGOL 60*, „ACM SIGPLAN Notices”, 13, nr 8, „Special Issue: History of Programming Languages Conference”, 1978.

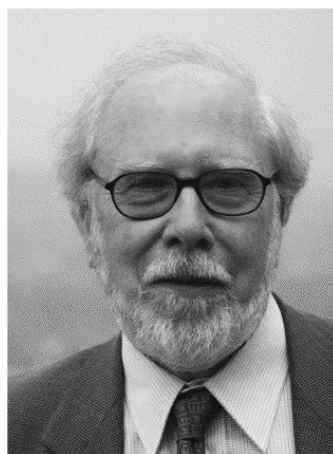
Perlis Alan J., *The American Side of the Development of Algol*, „ACM SIGPLAN Notices”, 13, nr 8, „Special Issue: History of Programming Languages Conference”, 1978.

van Wijngaarden A., Mailloux B. J., Peck J. E. L., Koster C. H. A., Sintzoff M., Lindsey C. H., Meertens L. G. L. T., Fisker R. G. (red.), *Revised Report on the Algorithmic Language Algol 68* (wrzesień 1973), Springer-Verlag, 1976.

22.2.3.2. Pascal

Należący do rodziny Algol język Algol68 był dużym i ambitnym projektem. Podobnie jak nad językiem Algol60, pracował nad nim zespół Algol (grupa robocza 2.1 IFIP), ale prace trwały bardzo długo i wiele niecierpliwych osób wątpiło, czy cokolwiek z tego wyjdzie. Jeden z członków zespołu, Niklaus Wirth, postanowił zaprojektować i zaimplementować własnego następcę dla języka Algol. W przeciwieństwie do języka Algol68, jego język, nazwany Pascalem, stanowił uproszczenie języka Algol60.

Prace nad Pascalem dobiegły końca w 1970 roku. Był to rzeczywiście prosty, choć przez to nieco mało elastyczny język. Często można było słyszeć opinie, że jest to język przeznaczony do nauki programowania, ale we wczesnych artykułach opisywano go jako alternatywę dla języka Fortran używanego w ówczesnych superkomputerach. Pascal był rzeczywiście łatwy do nauki i po ukazaniu się jego przenośnej implementacji stał się bardzo popularny wśród nauczycieli. Okazało się jednak też, że nie stanowił zagrożenia dla Fortrana.



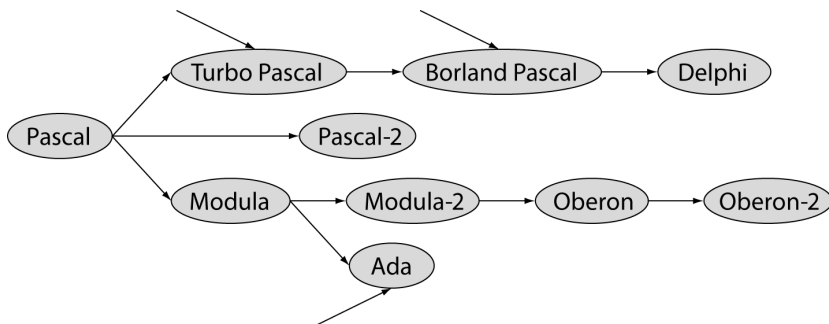
Pascal jest dziełem profesora Niklausa Wirtha (zdjęcia zrobiono w 1969 i 2004 roku) z uniwersytetu w Zurychu. Stopień doktora (w dziedzinie elektrotechniki i informatyki) uzyskał w University of California w Berkeley i pozostał przez całe życie związany z Kalifornią. Profesor Wirth był profesjonalistą od projektowania języków w każdym calu. Przez 25 lat zaprojektował i zaimplementował następujące języki:

- Algol W
- PL/360
- Euler
- Pascal
- Modula
- Modula-2
- Oberon
- Oberon-2
- Lola (język opisu sprzętu)

Wirth swoją pracę ocenia jako nigdy niekończącą się pogoń za prostotą. Jego prace miały bardzo duży wpływ na świat języków programowania. Badanie wszystkich jego języków jest niezwykle pasjonującym ćwiczeniem. Profesor Wirth jest jedyną osobą, która może pochwalić się prezentacją dwóch języków na konferencji HOPL.

Czysty Pascal okazał się w końcu zbyt prosty i sztywny, aby zdobyć popularność na dużą skalę. Przed zgubą uratował go w latach 1980 Anders Hejlsberg — jeden z założycieli firmy Borland. Najpierw zaprojektował i zaimplementował język Turbo Pascal (między innymi wprowadził do niego elastyczniejsze mechanizmy przekazywania argumentów), a później dodał model obiektowy podobny do języka C++ (ale bez wielodziedziczenia i z dobrą infrastrukturą modułową). Studiował na politechnice w Kopenhadze, gdzie od czasu do czasu wykladał Peter Naur — czasami nie można oprzeć się wrażeniu, że świat jest bardzo mały. Później Anders Hejlsberg zaprojektował Delphi dla firmy Borland i C# dla Microsoftu.

Drzewo rodzinne (z konieczności uproszczone) Pascala wygląda następująco:



Źródła

Borland i Turbo Pascal — http://en.wikipedia.org/wiki/Turbo_Pascal.

Hejlsberg Anders, Wiltamuth Scott, Golde Peter, *Język C#. Programowanie. Wydanie III*, Microsoft .NET Development Series, Helion, Gliwice 2009.

Wirth Niklaus, *The Programming Language Pascal*, „Acta Informatics”, Vol. 1 Fasc 1, 1971.

Wirth Niklaus, *Design and Implementation of Modula*, „Software—Practice and Experience”, 7, nr 1, 1977.

Wirth Niklaus, *Recollections about the Development of Pascal*, Proceedings of the ACM History of Programming Languages Conference (HOPL-2), „ACM SIGPLAN Notices”, 28, nr 3, 1993.

Wirth Niklaus, *Modula-2 and Oberon*, Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III), San Diego 2007; <http://portal.acm.org/toc.cfm?id=1238844>.

22.2.3.3. Ada

Język Ada został zaprojektowany w celu zaspokojenia potrzeb programistycznych departamentu obrony USA. Miał w szczególności służyć do tworzenia niezawodnego i łatwego w utrzymaniu oprogramowania systemów wbudowanych. Do jego najbardziej znanych poprzedników należą języki Pascal i Simula (punkty 22.2.3.2 i 22.2.4). Grupa opracowująca język Ada działała pod kierownictwem Jeana Ichbiaha — wcześniej był prezesem grupy użytkowników języka Simula. W projekcie języka Ada położono nacisk na:

- abstrakcję danych (ale do 1995 roku nie było dziedziczenia),
- solidne statyczne sprawdzanie typów,
- bezpośrednie wsparcie dla współbieżności ze strony języka.

Projekt języka Ada miał być ucieleśnieniem inżynierii oprogramowania w językach programowania. W wyniku tego departament obrony nie zaprojektował języka, tylko skomplikowany proces projektowania takiego języka. W proces projektowania zaangażowało się mnóstwo ludzi i organizacji. Prace polegały na organizowaniu szeregu konkursów, których celem było wyłonienie najlepszej specyfikacji, a następnie najlepszego języka odzwierciedlającego jej treść. Od 1980 roku zarządzanie tym olbrzymim przedsięwzięciem (trwającym w latach 1975 – 1998) przejął departament o nazwie AJPO (ang. *Ada Joint Program Office*).

W 1979 roku powstał język, który nazwano na cześć lady Augusty Ady Lovelace (córkę słynnego poety lorda Byrona). Lady Lovelace można nazwać pierwszym nowoczesnym programistą (przyjmując odpowiednią definicję nowoczesności), ponieważ pracowała z Charlesem Babbage (profesorem matematyki w katedrze Lucasa na uniwersytecie w Cambridge — to jest katedra, w której zasiadał kiedyś Newton!) w latach 40. XIX w. nad rewolucyjnym mechanicznym komputerem. Niestety jej urządzenie nie odniosło praktycznego sukcesu.





Dzięki przeprowadzeniu tak wyszukanego procesu projektowania język Ada uznawano za ostateczne rozwiązanie, jeśli chodzi o język zaprojektowany przez zespół. Lider zespołu projektanckiego, który wygrał konkurs, Jean Ichbiah z francuskiej formy Honeywell Bull, z poruszającą szczerością temu zaprzeczył. Podejrzewam nawet (na podstawie przeprowadzonych z nim rozmów), że mógłby zaprojektować lepszy język, gdyby nie był tak bardzo ograniczony przez ramy procesu.

Stosowanie języka Ada było przez wiele lat dozwolone w obszarach dotyczących wojska. Powstało nawet powiedzenie: „Ada to nie tylko dobry pomysł, to reguła!”. Początkowo stosowanie Ady było tylko „dozwolone”, ale gdy wiele projektów uzyskiwało zezwolenia na używanie innych języków (zazwyczaj C++), kongres USA wydał prawo zobowiązujące do stosowania Ady w większości wojskowych aplikacji. Później to prawo uchylono z komercyjnych i technicznych względów. Bjarne Stroustrup jest jednym z nielicznych, których prace zostały zakazane przez kongres USA.

Należy w tym miejscu zaznaczyć, że język Ada jest znacznie lepszy, niż wskazuje jego reputacja. Podejrzewamy, że gdyby departament obrony zarządzał nim nieco bardziej liberalnie (w kwestiach standardów dotyczących procesu rozwoju aplikacji, narzędzi programistycznych, dokumentacji itp.), język ten odniósłby znacznie większe sukcesy. Do dziś Ada zajmuje ważne miejsce w lotnictwie i podobnych do niego obszarach programowania systemów wbudowanych.

Język Ada został standardem wojskowym w 1980 roku, standardem ANSI w 1983 (pierwszą implementację wykonano w 1983 roku, a więc trzy lata po ukazaniu się pierwszego standardu!) oraz standardem ISO w 1987 roku. Standard ISO został gruntownie (ale oczywiście z zachowaniem zgodności z poprzednią wersją) zaktualizowany i zaklasyfikowany jako standard ISO z 1995 roku. Do najważniejszych usprawnień można zaliczyć zwiększenie elastyczności mechanizmów współbieżności oraz obsługę dziedziczenia.

Źródła

Barnes John, *Programming in Ada 2005*, Addison-Wesley, 2006.

Skonsolidowana instrukcja do języka Ada zawierająca międzynarodowy standard (ISO/IEC 8652:1995), *Information Technology — Programming Languages — Ada*, z poprawkami naniesionymi przez Technical Corrigendum 1 (ISO/IEC 8652:1995:TC1:2000).

Strona informacyjna na temat języka Ada: www.adaic.org/.

Whitaker William A., *ADA — The Project: The DoD High Order Language Working Group*, Proceedings of the ACM History of Programming Languages Conference (HOPL-2), „ACM SIGPLAN Notices”, 28, nr 3, 1993.

22.2.4. Simula

Język Simula opracowali na początku lat 60. Kristen Nygaard i Ole-Johan Dahl w norweskim centrum komputerowym na uniwersytecie Oslo. Język Simula bez wątpienia należy do rodziny Algol. W istocie jest prawie w całości podzbiorem języka Algol60. Postanowiliśmy jednak opisać go osobno, ponieważ stanowi źródło fundamentalnych technik, które obecnie są znane pod nazwą „programowanie obiektowe”. Był to pierwszy język z dziedziczeniem i funkcjami wirtualnymi. Od niego pochodzą takie określenia jak **klasa** oznaczająca „typ zdefiniowany przez użytkownika” czy **wirtualny** w znaczeniu funkcji, którą można wywołać za pośrednictwem interfejsu klasy bazowej.

Wpływy języka Simula nie są ograniczone do narzędzi języka. Język ten był obiektowy, a jego projekt opierał się na pomysłach odtwarzania w kodzie zjawisk świata rzeczywistego:



- reprezentacja zjawisk jako klas i obiektów klas,
- reprezentacja relacji hierarchicznych jako hierarchii klas (dziedziczenie).

Zatem w tym języku program przestaje być monolitem i staje się zbiorem współpracujących obiektów.

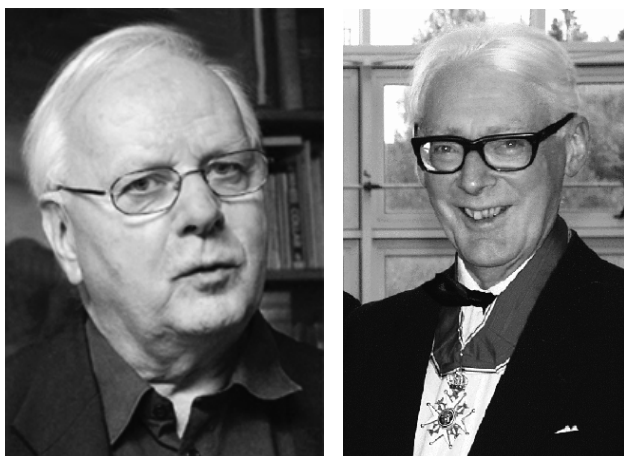


Kristen Nygaard — jeden z dwóch wynalazców (obok Ole-Johana Dahla — na zdjęciu po lewej w okularach) języka Simula 67 — był gigantem pod wieloma względami (także jeśli chodzi o wzrost), którego pasję i szczodrość można traktować jako wzór. Stworzył podwaliny programowania i projektowania obiektowego, zwłaszcza dziedziczenia, i przez dziesięciolecia śledził ich skutki. Nigdy nie wystarczały mu proste, krótkie i krótkowzroczne odpowiedzi. Przez wiele lat stale żywo uczestniczył w życiu społecznym. Można mu przypisać sporą zasługę w tym, że Norwegia nie przystąpiła do Unii Europejskiej. Uważał, że Unia może się

okazać scentralizowanym i biurokratycznym potworem nieczułym na potrzeby małego kraju usytuowanego na samym jej końcu. W połowie lat 70. Kristen Nygaard spędził sporo czasu na wydziale informatyki uniwersytetu w Aarhus w Danii (w tym czasie Bjarne Stroustrup odbywał tam swoje studia magisterskie).

Nygaard miał stopień magistra matematyki, który uzyskał na uniwersytecie Oslo. Zmarł w 2002 roku na miesiąc przed odbiorem (wraz ze swoim przyjacielem Ole-Johaniem Dahlem) przyznawanej przez ACM nagrody Turinga, która jest prawdopodobnie najznakomitszym uhonorowaniem dla informatyka.

Ole-Johan Dahl bardziej typowo działał w środowisku akademickim. Bardzo interesowały go zagadnienia związane ze specyfikacjami języków oraz metodami formalnymi. W 1968 roku został pierwszym profesorem zwyczajnym w dziedzinie informatyki na uniwersytecie w Oslo.



W sierpniu 2000 roku Dahl i Nygaard zostali odznaczeni przez króla Norwegii orderem św. Olafa. Nawet prawdziwi maniacy komputerowi mogą zyskać uznanie we własnym mieście!

Źródła

Birtwistle G., Dahl O.-J., Myhrhaug B., Nygaard K., *SIMULA Begin*, Studentlitteratur, Lund 1979.

Holmevik J. R., *Compiling SIMULA: A Historical Study of Technological Genesis*, „IEEE Annals of the History of Computing”, 16, nr 4, 1994, s. 25 – 37.

Krogdahl S., *The Birth of Simula*, Proceedings of the HiNC 1 Conference in Trondheim, czerwiec 2003 (IFIP WG 9.7, we współpracy z IFIP TC 3).

Nygaard K., Dahl O.-J., *The Development of the SIMULA Languages*, „ACM SIGPLAN Notices”, 13, nr 8, „Special Issue: History of Programming Languages Conference”, 1978.

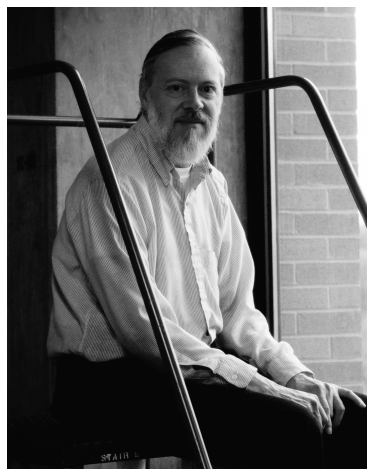
SIMULA Standard, *DATA processing — Programming languages — SIMULA*, standard szwedzki, Stockholm 1987.

22.2.5. C

W 1970 roku wiadomo było, że do programowania poważnych systemów — zwłaszcza implementacji systemów operacyjnych — trzeba używać kodu asemblera, a więc pisać nieprzenośny kod. Sytuację tę można porównać do programowania naukowego z czasów przed pojawieniem się języka Fortran. Do walki z tym problemem stanęło kilka osób i grup. W ostatecznym rozrachunku z wszystkich opracowanych rozwiązań zdecydowanie największy sukces odniósł język C (rozdział 27.).

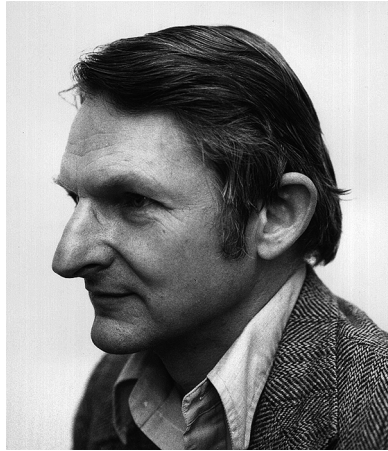


Język C został zaprojektowany i zaimplementowany przez Dennisa Ritchiego w centrum badań komputerowych Bell Telephone Laboratories w Murray Hill w stanie New Jersey. Piękno C polega na tym, że jest to prosty język programowania, który trzyma się bardzo blisko fundamentalnych aspektów sprzętu. Większość skomplikowanych własności (z których znaczna część znalazła się też dla zachowania zgodności między tymi językami w C++) została dodana później i w niektórych przypadkach wbrew woli Ritchiego. Sukces języka C należy częściowo przypisać jego szybkiemu udostępnieniu do użytku, ale jego prawdziwa siła tkwi w bezpośrednim związku między językiem a sprzętem (podrozdziały 25.4 i 25.5). Dennis Ritchie opisywał język C jako „język ze ścisłą kontrolą typów, ale słabym sprawdzaniem”. To znaczy, C ma statyczny (czasu kompilacji) system typów i programy, które używają obiektów inaczej, niż wynika z ich definicji, są niepoprawne. Problem w tym, że kompilator nie może tego sprawdzić. To miało sens, gdy kompilator C musiał działać w 48 kilobajtach pamięci. Szybko po wejściu C do użytku pojawił się program lint, który niezależnie od kompilatora sprawdzał zgodność programów z systemem typów.



Dennis Ritchie wraz z Kenem Thompsonem wynaleźli system Unix — najbardziej wpływowy system operacyjny wszech czasów. Język C był i jest związany z tym systemem, a za jego pośrednictwem z systemem Linux i ruchem otwartego oprogramowania.


Dennis Ritchie przez 40 lat pracował w centrum badań informatycznych Bell Laboratories. Studiował fizykę na uniwersytecie Harvarda, gdzie również obronił pracę doktorską z matematyki, jednak nie uzyskał tytułu doktora, ponieważ nie wniósł opłaty rejestracyjnej (60 dolarów).



W latach 1974 – 1979 na język C wpływ wywarło wielu ludzi pracujących w Bell Laboratories. Ulubionym przez wszystkich krytykiem, partnerem do dyskusji i człowiekiem dostarczającym pomysłów był Doug McIlroy. Wywarł on wpływ na języki C, C++, system Unix i wiele innych rzeczy.



Brian Kernighan jest wybitnym programistą i pisarzem. Zarówno jego kod, jak i teksty należy zaliczyć do wzorów klarowności. Styl tej książki częściowo jest wzorowany na jego arcydziele *Język ANSI C. Programowanie* (znanym jako K&R od nazwisk autorów Briana Kernighana i Dennisa Ritchie).

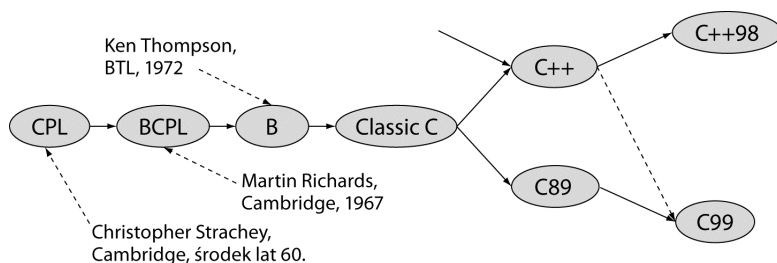
 Nie wystarczy mieć dobre pomysły. Aby idee stały się przydatne na dużą skalę, muszą zostać zredukowane do najprostszej formy i klarownie wyrażone w taki sposób, aby zrozumiało je jak najwięcej ludzi z grupy docelowej. Do największych wrogów prezentacji takich rzeczy należy rozwlekłość, podobnie jak niejasne wyrażanie i zbyt wysoki stopień abstrakcji. Puryści często kpią sobie z wyników takiej popularyzacji i wolą „oryginalne wyniki” przedstawione

w sposób rozumiały tylko dla ekspertów. My jesteśmy innego zdania — przekazanie nieprosty, ale wartościowych umiejętności nowicuszowi jest trudne, ma duże znaczenie dla powiększania się grona profesjonalistów oraz daje korzyści społeczeństwu.

Brian Kernighan uczestniczył w wielu znaczących projektach programistycznych i wydawniczych. Jako przykłady niech posłużą AWK (jeden z pierwszych języków skryptowych nazwany po inicjałach jego twórców — Aho, Weinberger i Kernighan) oraz AMPL — ang. *A Mathematical Programming Language* (matematyczny język programowania).

Aktualnie Kernighan jest profesorem pracującym na Princeton University. Jest oczywiście świetnym nauczycielem specjalizującym się w objaśnianiu skomplikowanych zagadnień. Ponad 30 lat przepracował w centrum badań komputerowych w Bell Laboratories. Z czasem Bell Laboratories przekształcono w AT&T Bell Labs, a później podzielono na AT&T Labs oraz Lucent Bell Labs. Studiował fizykę na University of Toronto, a tytuł doktora w dziedzinie elektrotechniki uzyskał w Princeton University.

Poniżej znajduje się drzewo rodzinne języka C:



Język C wywodzi się od nigdy nieukończonego, a rozpoczętego w Anglii projektu CPL, języka BCPL (ang. *Basic CPL*), który Martin Richards opracował podczas wizyty w MIT po odejściu z Cambridge University, oraz interpretowanego języka o nazwie B opracowanego przez Kena Thompsona.

Później opracowano standardy ANSI i ISO tego języka oraz pojawiło się w nim wiele wpływów języka C++ (np. sprawdzanie argumentów funkcji i `const`).

CPL był wspólnym przedsięwzięciem Cambridge University i Imperial College w Londynie. Początkowo prace były prowadzone w Cambridge, a więc oficjalnie C rozszyfrowywano jako skrót od nazwy tego miasta. Gdy do projektu przyłączył się Imperial College, C oficjalnie miało oznaczać „Combined” (połączony). W rzeczywistości (tak nam przynajmniej mówiono) litera C zawsze miała oznaczać Christopher od Christophera Stracheya — głównego projektanta CPL.

Źródła

Strony Briana Kernighana: <http://cm.bell-labs.com/cm/cs/who/bwk> i www.cs.princeton.edu/~bwk/.

Strona Dennisa Ritchiego: <http://cm.bell-labs.com/cm/cs/who/dmr>.

ISO/IEC 9899:1999, *Programming Languages — C* (standard języka C).

Kernighan Brian, Ritchie Dennis, *Język ANSI C. Programowanie*. Wydanie II, Helion, Gliwice 2010.

Lista członków centrum badań informatycznych Bell Labs: <http://cm.bell-labs.com/cm/cs/alumni.html>.

Ritchards Martin, *BCPL — The Language and Its Compiler*, Cambridge University Press, 1980.

Ritchie Dennis, *The Development of the C Programming Language*, Proceedings of the ACM History of Programming Languages Conference (HOPL-2), „ACM SIGPLAN Notices”, 28, nr 3, 1993.

Salus Peter, *A Quarter Century of UNIX*, Addison-Wesley, 1994.

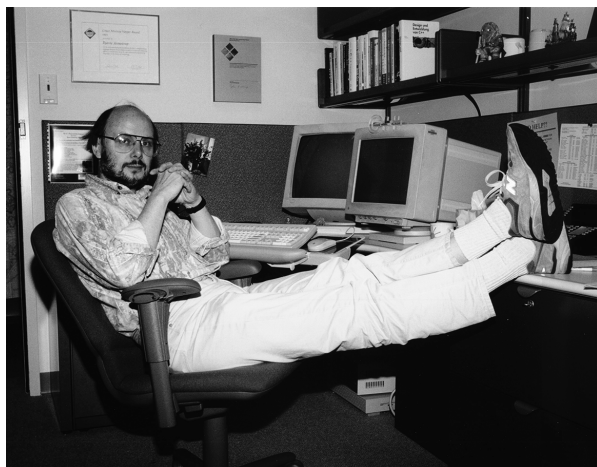
22.2.6. C++



C++ to język programowania ogólnego użytku z nakierowaniem na programowanie systemów, który:

- Jest lepszy niż C.
- Obsługuje abstrakcję danych.
- Obsługuje programowanie obiektowe.
- Obsługuje programowanie ogólne.

Został zaprojektowany i zaimplementowany przez Bjarne Stroustrupa w centrum badań informatycznych Bell Telephone Laboratories w Murray Hill w stanie New Jersey, a więc w bezpośrednim sąsiedztwie Dennisa Ritchiego, Briana Kernighana, Kena Thompsona, Douga McIlroya i innych wielkich ludzi ze świata Uniksa.



Bjarne Stroustrup uzyskał stopień magistra (w dziedzinie matematyki z informatyką) na uniwersytecie w swoim rodzinnym mieście Århus w Danii. Następnie przeniósł się do Cambridge, aby uzyskać stopień doktora informatyki, pracując dla Davida Wheelera. Główne cele C++ to:



- Umożliwić stosowanie technik abstrakcji w projektach głównego nurtu.
- Po raz pierwszy umożliwić stosowanie obiektowych i ogólnych technik programowania w dziedzinach, w których największe znaczenie ma wydajność.

Przed pojawieniem się języka C++ techniki te (często niedbale określane zbiorczo programowaniem obiektowym) były mało znane. Podobnie jak w przypadku programowania przed pojawieniem się języka Fortran i programowania systemów przed ukazaniem się C, było

„dobrze wiadomo”, że techniki te są zbyt kosztowne, aby je realnie stosować oraz zbyt skomplikowane dla „zwykłych programistów”.

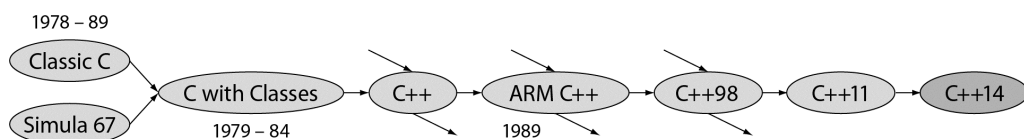
Prace nad językiem C++ zaczęły się w 1979 roku, aby w 1985 roku pojawiła się pierwsza komercyjna wersja. Później Bjarne Stroustrup rozwijał go, korzystając z pomocy przyjaciół z Bell Labs i innych, aż w 1990 roku rozpoczął się proces standaryzacji. Od tamtej pory definicję C++ rozwija ANSI (amerykańska organizacja standaryzacyjna) i od 1991 roku ISO (międzynarodowa organizacja standaryzacyjna). Bjarne Stroustrup czynnie w tych pracach uczestniczy jako przewodniczący kluczowej podgrupy zajmującej się nowymi własnościami języka. Pierwszy międzynarodowy standard (C++98) został zatwierdzony w 1998 roku, a drugi — w 2011 (C++11). Następny standard ISO będzie nosił nazwę C++14, a po nim może pojawić się C++17, niekiedy nazywany C++1y.

Do najważniejszych etapów w rozwoju języka C++, które nastąpiły po początkowej fazie, było opracowanie biblioteki STL — algorytmów i kontenerów biblioteki standardowej. Był to wynik dziesiątek lat pracy — głównie Aleksa Stepanova — której celem było utworzenie najogólniejszego i najwydajniejszego oprogramowania. W pracach tych inspirowano się pięknem matematyki.



Alex Stepanov jest autorem biblioteki STL i pionierem programowania ogólnego. Studiował na uniwersytecie w Moskwie i pracował w dziedzinach robotyki, algorytmów i wielu innych, gdzie korzystał z różnych języków programowania (m.in. Ada, Scheme i C++). Od 1979 roku pracuje w amerykańskim środowisku akademickim i gospodarce — laboratoriach GE, AT&T Bell Labs, Hewlett-Packard, Silicon Graphics oraz Adobe.

Drzewo rodzinne języka C++ wygląda następująco:



Język „C with Classes” stanowił połączenie języków C i Simula. Umarł natychmiast po pojawieniu się implementacji swojego następcy — języka C++.

W istocie większość danych statystycznych, które znajdujemy w sieci (i w innych źródłach), nie jest lepsza od zwykłych plotek, ponieważ słabo korelują z rzeczywistym użytkowaniem języków. Są to np. liczba wystąpień nazwy danego języka na stronach internetowych, zamówienia kompilatorów, artykuły naukowe, sprzedaż książek itp. Wszystkie te miary faworyzują nowości wobec tego, co jest utrwalone. Przy okazji, kto to jest programista? Ktoś, kto codziennie używa jakiegoś języka programowania? Jak w takim razie zaklasyfikować studenta piszącego małe programy w ramach ćwiczeń albo profesora, który tylko mówi o programowaniu? Czy programistą jest fizyk, który prawie co roku pisze jakiś program? Czy profesjonalnym programistą jest ktoś, kto w każdym tygodniu używa języka programowania wiele razy, czy tylko raz? Na każde z tych pytań można znaleźć różne odpowiedzi w zależności od zastosowanej metody statystycznej.



Czujemy się jednak zobowiązani do przedstawienia swojej opinii. W roku 2014 na świecie żyło około 10 milionów zawodowych programistów. Informację tę uzyskaliśmy z danych IDC (firmy zajmującej się gromadzeniem danych), dzięki rozmowom z wydawcami i dostawcami kompilatorów oraz z różnych źródeł internetowych. Możesz spierać się o szczegóły, ale mamy pewność, że liczba ta jest większa niż 1 milion i mniejsza niż 100 milionów bez względu na to, którą definicję programisty się przyjmie. Jakich języków ci ludzie używają? Prawdopodobnie około 90% wszystkich programów zostało napisane w językach Ada, C, C++, C#, COBOL, Fortran, Java, PERL, PHP, Python i Visual Basic.

Moglibyśmy wymienić jeszcze dziesiątki, a nawet setki innych języków programowania. Nie widzimy jednak sensu, aby to robić, poza chęcią oddania należnego szacunku tym językom, które są interesujące lub ważne. W razie potrzeby samodzielnie poszukaj informacji. Profesjonalista zna kilka języków programowania i w razie potrzeby uczy się nowych. Nie ma „jedyne go najlepszego języka” dla wszystkich ludzi i dziedzin. W istocie wszystkie znane nam większe systemy zostały zbudowane przy użyciu więcej niż jednego języka programowania.

22.2.8. Źródła informacji

Do opisu każdego języka programowania powyżej dołączona została lista tekstów źródłowych. Dotyczą one kilku języków:

Więcej odnośników do stron projektantów języków i ich zdjęć

www.angelfire.com/tx4/cus/people/

Kilka przykładów języków

<http://dmoz.org/Computers/Programming/Languages/>

Podręczniki

Scott Michael L., *Programming Language Pragmatics*, Morgan Kaufmann, 2000.

Sebesta Robert W., *Concepts of Programming Languages*, Addison-Wesley, 2003.

Historia

Bergin T. J., Gibson R. G. (red.), *History of Programming Languages*, t. II, Addison-Wesley, 1996.

Hailpern Brent, Ryder Barbara G. (red.), *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*, San Diego 2007; <http://portal.acm.org/toc/cfm?id=1238844>.

Lohr Steve, *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts—The Programmers Who Created the Software Revolution*, Basic Books, 2002.

Sammet Jean, *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969.

Wexelblat Richard L. (red.), *History of Programming Languages*, Academic Press, 1981.

Ćwiczenia

1. Czemu służy historia?
2. Do czego można użyć języka programowania? Podaj kilka przykładów.
3. Wymień kilka podstawowych własności języków programowania, które są bezwzględnie dobre.
4. Co mamy na myśli, pisząc „abstrakcja” i „wyższy poziom abstrakcji”?
5. Jakie są nasze cztery wysokopoziomowe ideały kodu?
6. Wymień kilka potencjalnych zalet programowania wysokopoziomowego.
7. Co to jest wielokrotne użycie kodu i co dobrego z tego wynika?
8. Na czym polega programowanie proceduralne? Podaj konkretny przykład.
9. Co to jest abstrakcja danych? Podaj konkretny przykład.
10. Na czym polega programowanie obiektowe? Podaj konkretny przykład.
11. Na czym polega programowanie ogólne? Podaj konkretny przykład.
12. Na czym polega programowanie wieloparadygmatowe? Podaj konkretny przykład.
13. Jaki był pierwszy program uruchomiony na komputerze przechowującym program w pamięci?
14. Czym wsławił się David Wheeler?
15. Czym głównie zasłużył się pierwszy język Johna Backusa?
16. Jaki pierwszy język zaprojektowała Grace Murray Hopper?
17. Na jakim polu w informatyce najaktywniej udzielał się John McCarthy?
18. Jaki wkład w język Algol60 wniósł Peter Naur?
19. Czym wsławił się Edsger Dijkstra?
20. Jakie języki zaprojektował i zaimplementował Niklaus Wirth?
21. Jakie języki zaprojektował Anders Hejlsberg?
22. Jaką rolę odegrał w projekcie języka Ada Jean Ichbiah?
23. Jaki rodzaj programowania zapoczątkował język Simula?
24. Gdzie (poza Oslo) uczył Kristen Nygaard?
25. Czym wsławił się Ole-Johan Dahl?
26. Głównym projektantem którego systemu operacyjnego był Ken Thompson?
27. Czym wsławił się Doug McIlroy?
28. Jaki tytuł nosi najbardziej znana książka Briana Kernighana?
29. Gdzie pracował Dennis Ritchie?
30. Czym wsławił się Bjarne Stroustrup?
31. Jakich języków programowania próbował używać Alex Stepanov do zaprojektowania biblioteki STL?

32. Wymień dziesięć języków programowania, których nie opisano w podrozdziale 22.2.
33. Dialektem którego języka jest Scheme?
34. Wymień dwóch najbardziej znanych przodków języka C++.
35. Co oznacza litera C w nazwie C++?
36. Czy nazwa Fortran jest akronimem? Jeśli tak, to podaj jego rozwinięcie.
37. Czy nazwa COBOL jest akronimem? Jeśli tak, to podaj jego rozwinięcie.
38. Czy nazwa Lisp jest akronimem? Jeśli tak, to podaj jego rozwinięcie.
39. Czy nazwa Pascal jest akronimem? Jeśli tak, to podaj jego rozwinięcie.
40. Czy nazwa Ada jest akronimem? Jeśli tak, to podaj jego rozwinięcie.
41. Który język programowania jest najlepszy?

Terminologia

W tym rozdziale terminy stanowią nazwy języków programowania i organizacji oraz nazwiska ludzi:

- | | |
|-----------------------|--|
| • Języki: | • Organizacje: |
| • Ada | • Bell Laboratories |
| • Algol | • Borland |
| • BCPL | • Cambridge University |
| • C | • ETH (szwajcarski federalny obszar politechnik i instytutów technicznych) |
| • C++ | • IBM |
| • COBOL | • MIT |
| • Fortran | • Norwegian Computer Center |
| • Lisp | • Princeton University |
| • Pascal | • Stanford University |
| • Scheme | • U.S. Department of Defense |
| • Simula | • U.S. Navy |
| | • Wyższa Szkoła Techniczna w Kopenhadze |
| • Ludzie: | |
| • Charles Babbage | • Doug McIlroy |
| • John Backus | • Peter Naur |
| • Ole-Johan Dahl | • Kristen Nygaard |
| • Edsger Dijkstra | • Dennis Ritchie |
| • Anders Hejlsberg | • Alex Stepanov |
| • Grace Murray Hopper | • Bjarne Stroustrup |
| • Jean Ichbiah | • Ken Thompson |
| • Brian Kernighan | • David Wheeler |
| • John McCarthy | • Niklaus Wirth |

Praca domowa

1. Zdefiniuj **programowanie**.
2. Zdefiniuj **język programowania**.
3. Przeglądnij winiety rozdziałów w tej książce. Które są autorstwa informatyków? Opisz krótko osiągnięcia każdego z nich.
4. Przeglądnij winiety rozdziałów w tej książce. Które nie są autorstwa informatyków? Znajdź kraj ich pochodzenia oraz dziedzinę, którą się zajmowali.
5. Napisz program typu „Witaj, świecie” w każdym z opisanych w tym rozdziale języków.
6. Poszukaj w popularnych podręcznikach na temat każdego z opisanych języków coś, co się w nich uznaje za pierwszy pełny program. Napisz ten program we wszystkich pozostałych językach. Uwaga: może się okazać, że będzie trzeba napisać bardzo dużo programów.
7. Pominęliśmy wiele ważnych języków. W szczególności byliśmy zmuszeni pominąć wszystkie języki po C++. Utwórz listę pięciu nowoczesnych języków programowania, które Twoim zdaniem powinny zostać opisane i na temat każdego z nich napisz półtorastronicowy tekst.
8. Do czego i dlaczego używa się języka C++? Napisz na ten temat raport długości od 10 do 20 stron.
9. Do czego i dlaczego używa się języka C? Napisz na ten temat raport długości od 10 do 20 stron.
10. Wybierz dowolny język programowania (nie C ani C++) i napisz na jego temat około 10 – 20 stronicowy tekst, w którym opiszysz jego pochodzenie, przeznaczenie oraz możliwości. Podaj dużo konkretnych przykładów. Kto z niego korzysta i w jakim celu?
11. Kto aktualnie jest profesorem w katedrze Lucasa w Cambridge?
12. Kto z wymienionych w tym rozdziale projektantów języków programowania jest z wykształcenia matematykiem, a kto nie?
13. Kto z wymienionych w tym rozdziale projektantów języków programowania ma doktorat (i w jakiej dziedzinie), a kto nie?
14. Kto z wymienionych w tym rozdziale projektantów języków programowania jest laureatem nagrody Turinga? Co to za nagroda? Znajdź notki o laureatach wymienionych w tym rozdziale.
15. Napisz program, który wczytuje plik zawierający pary (nazwa,rok), np. (Algol,1960) i (C,1974), i umieszcza je na osi czasu.
16. Zmodyfikuj powyższy program, aby wczytywał plik zawierający krotki (nazwa,rok,(przodkowie)), np. (Fortran,1956,()), (Algol,1960,(Fortran)) czy (C++,1985,(C,Simula)) i umieszczał je na osi czasu oraz rysował strzałki biegnące od przodków do potomków. Użyj tego programu do narysowania ulepszonych wersji schematów z punktów 22.2.2 i 22.2.7.

Podsumowanie

Oczywiście tylko liźnęliśmy historii języków programowania i ideałów, które są motorem dążenia do coraz lepszego oprogramowania. Historia i te ideały są dla nas tak ważne, że jest nam przykro, iż musimy te tematy potraktować tak powierzchownie. Mamy nadzieję, że udało się nam przekazać część naszej fascynacji oraz w jakimś stopniu uwidocznić, jak wygląda dążenie do powstawania lepszego oprogramowania i lepszego programowania, co manifestuje się poprzez projektowanie i implementację języków programowania. Pamiętaj zatem, że programowanie — tworzenie dobrej jakości oprogramowania — jest zagadnieniem o fundamentalnym znaczeniu. Język programowania jest tylko narzędziem.