

# Praktyka programowania

*Referat z przedmiotu "Analiza, projektowanie i programowanie obiektowe"*

*Adam Sawicki, Grzegorz Stępnik*

*Informatyka, gr. II, sem. VI*

*26 marca 2006*

## 1. Styl Programowania

### **Programy mają być czytane przez ludzi**

W przyszłości możliwe, że będziemy mogli pisać programy w językach takich jak Angielski czy Polski. Jednak w tym momencie i zapewne jeszcze przez długi czas będziemy musieli zajmować się zasadami produkcji programów i pisanem ich w sposób zrozumiały przez człowieka. Styl programowania odnosi się do czytelności programów. Styl jest ważny, żeby człowiek mógł zrozumieć dany program, umieć go przeczytać. Gdy programiści przyzwyczajają się do określonego stylu, łatwiej jest im zrozumieć zarówno swój kod jak i inny.

Program jest przeznaczony do czytania bardziej przez ludzi niż przez komputer. Ludzie muszą przeczytać i zrozumieć kod, żeby poprawiać błędy, modyfikować lub konserwować program. Programy są także swojego rodzaju dokumentami, na których można powołać się w przyszłości. Można się z nich uczyć algorytmów. Wykorzystuje się je także pisząc inne programy. Autor programu zawsze powinien móc przeczytać swój program.

Gdy nie stosujemy się do określonego stylu, kod nie jest przejrzysty, wtedy gdy chcemy program zmodyfikować, kontynuować prościej jest zacząć od początku niż zrozumieć go by dalej pisać. Czytelny program stwarza wrażenie, że jego autor wiedział co robi. Sam program powinien mówić możliwie jak najwięcej o swojej kompozycji i budowie.

### **Standardy stylu**

Styl programowania jest sprawą prywatnych poglądów i gustów, nie powinien więc być ograniczony. Jeśli można coś zrobić na więcej niż jeden sposób, to należy wybrać jeden i się go trzymać. Wszystkie standardy to tylko zalecenia. Nie jest do końca prawdą, że mogą ograniczać przyszły rozwój i postęp. Dobry zestaw standardów może sprzyjać przyszłemu rozwojowi i postępowi dzięki kreowaniu uwagi programistów na nowe zagadnienia.

### **Komentarze**

Komentarze są bardzo ważną częścią programu. Bez nich trudno było by zrozumieć skomplikowane fragmenty kodu. Wielu początkujących programistów nie komentuje kodu, oszczędzając w ten sposób czas bądź usprawiedliwiając się tym, że "wstawi je później". Autor taki zdumiewająco szybko przekonuje się, że zapomniał wielu szczegółów w kodzie, i spędza więcej czasu na rozszyfrowanie o co mu chodziło. Pisząc komentarze zyskamy stracony na nie czas z nawiązką. Nieskomentowany program to chyba największy błąd jaki może popełnić programista.

Komentarze należy zatem wstawiać podczas pisania programu. Wtedy najlepiej zna się szczegóły. Komentarze wstawiane potem rzadko są zadowalające. Trudno też wówczas pamiętać co powinno się było skomentować.

Nie łatwo jest poczynić dobre komentarze. Mają one mówić jak zrozumieć program, powinny być więc przemyślane i zaprojektowane równie dobrze jak sam program.

Istnieją trzy typy komentarzy: komentarze wstępne, przewodniki i komentarze objaśniające.

## Komentarze wstępne

Dobrze jest gdy każdy program, procedura czy funkcja zaczynają się od pewnych instrukcji objaśniających.

Powinny się tu znaleźć przynajmniej:

- Opis działania programu
- Sposób użycia (jak wywołać program, jak go używać itp)
- Lista i opis ważniejszych zmiennych i tablic
- Pouczenie o wejściu/wyjściu
- Nazwy wszelkich metod specjalnych, które zostały użyte, wraz ze wskazaniem gdzie można znaleźć dalsze informacje
- Pewne informacje o czasie działania
- Ilość wymaganej pamięci operacyjnej
- Autor
- Data napisania

Najlepiej umieścić te informacje jako komentarz w samym programie.

Przykład komentarza wstępnego:

```
//  
//  LOAD_FROM_FILE UNIT  
//  
//  Autor: Grzegorz Stepnik  
//  
//  Opis:  
//  Modul zawiera funkcje:  
//  
//  |-----|  
//  | void GReadFile(char* name, map<string, string> &hash); |  
//  |-----|  
//  name - nazwa pliku z danymi  
//  hash - kontener typu std::map<string, string> do przechowywania danych z  
//  pliku  
//  
//  Funkcja zwraca przez referencje mape hash z danymi z pliku.  
//  Dane z pliku maja nastepujacy format:  
//  
//  "nazwa klucza" "wartosc"  
//  
//  Zasady:  
//  1. nazwa klucza oraz wartosc  musza byc ujetne w " " i moga miec spacje. Znak  
//  zastrzezony w nazwie klucza i wartosci to to: "  
//  2. klucz oraz wartosc moga miec kilka linijek  
//  3. poza znakami " " moze byc dowolny tekst z wyjatkiem '"' np. (  
//  ==="klucz"---"wartosc"===)  
//  4. #komentarz# - komentarz ograniczony znakami # nie jest uwzgledniany przy  
//  czytaniu z pliku  
//  (moze byc nawet wewnatrz tresci klucza lub wartosci, moze takze byc  
//  wielolinijkowy! :)  
//  
//  Przyklady:  
//  
//  "pozycja x" = "100"  
//  "pozycja y""20"  
//  "kolor"-  
//  -"czarny"  
//  
//  Funkcja GLoadFile znajduje sie w przestrzeni GLoad
```

## Przewodniki

Jeśli program jest bardzo długi warto sporządzić tzw. przewodnik czy spis treści w formie komentarza na początku programu. Spis treści powinien podawać nazwę, miejsce i funkcję każdego modułu. Należy liczyć, że każdy moduł jest już skomentowany.

## Komentarze objaśniające

Komentarze objaśniające umieszcza się w programie w celu objaśnienia każdego nie jasnego fragmentu kodu. Wstawiać należy komentarze zawsze tam gdzie coś może być nie całkiem oczywiste dla kogoś innego. Ważny jest rodzaj komentarza. Komentarz w stylu:

```
//sprawdzenie czy większe lub równe
```

nie jest dobrym komentarzem. Programista znający język wie, że sprawdzany jest taki warunek. Nie wie jednak poco i to należy właśnie opisać w komentarzu.

```
//wykonanie transakcji jeśli zapłata większa lub równa od ceny.
```

Komentarz powinien od razu pokazywać cel, to co się dzieje w strukturze programu.

Program jest dobrze skomentowany, jeśli programista może przeczytać same komentarze i zrozumieć co robi program bez odwołania się do innej dokumentacji. Często komentarze są niewystarczające, bo pisząc program przeceniamy własne umiejętności. Wierzymy, że bez trudu zapamiętamy pewne części programu.

Błędne komentarze to gorzej niż zupełny ich brak. Komentarze muszą być poprawne. Wraz ze zmianą programu zmiany muszą również ulec komentarze.

## Puste linie

Stosowanie odstępów to metoda na przejrzysty kod. Pisząc referat czy wypracowanie w tekście używa się pustych linii oddzielających akapity. Podobnie można użyć pustych linii do oddzielania składowych programu. Jedna pusta linia może służyć do oddzielenia podobnych instrukcji. Można zastosować kilka pustych linii do oddzielenia dużych fragmentów programu.

## Stosuj odstępy dla poprawienia czytelności

Odstępy najczęściej są opcjonalne. Pomijanie ich jednak nie jest dobrą cechą. Pisząc zdanie bez odstępów między wyrazami jesteśmy w stanie je odczytać, jednak zajęło by nam to o wiele więcej czasu. Odstępy w kodzie powinny być stosowane wszędzie tam, gdzie poprawiły by one czytelność.

Możliwe jest napisanie instrukcji następująco:

```
zmienna=11/zmienna2+2;
```

Dużo jednak łatwiej odczytać to:

```
zmienna = 11 / zmienna2 + 2;
```

Należy dawać odstępy między operatorami (+, -, \*, /), zmiennymi itd.

Czasem jednak można użyć odstępów do pokazania kolejności wykonywania działań:

```
a*b + c
```

## Wybór nazw zmiennych

Nazwy zmiennych należy wybierać tak, aby najlepiej identyfikowały reprezentowane przez siebie wielkości. Można stosować nazwy odpowiednio długie jednak nie za długie.

Przykład:

Lepiej jest użyć takich nazw:

```
cena = koszt + zysk;
```

niż takich:

```
c = k + z;
```

Właściwy dobór nazw zmiennych to jedna z najważniejszych zasad dotyczących czytelności programu. Dobierając nazwy zmiennych spróbuj stwierdzić, co ma zmienna oznaczać w potocznych tekstach, po czym wybierz najważniejsze słowo.

W nazwach zmiennych można stosować znak podkreślenia. Jeśli zmienna składa się z kilku członów znak ten poprawia czytelność zmiennej.

Przykład:

```
lista_wartosci  
koszt_dod
```

## **Standardowe skróty**

Dobrym pomysłem jest ustalenie sobie bądź w firmie listy standardowych skrótów. Ułatwia to czytanie programu programistom nie będącym autorami programu. Np:

```
li_jednostek  
li_budynkow
```

```
l_surowcow  
l_jednostek
```

W tym przypadku "li" znaczyło by słowo lista, a "l" słowo liczba.

## **Przenoszenie słów**

Gdy wiersz przekracza 80 znaków warto przenieść resztę do nowej linii. Jeśli słowo nie mieści się w linii należy je w całości przenieść do nowej. Przenosząc instrukcję powinno się na końcu zostawiać operator. Wiadomo wtedy, że w kolejnej linii jest kontynuacja. Np:

```
a = b + c -  
(d + 2);
```

jest lepsze niż:

```
a = b + c  
- (d + 2);
```

## **Rozmieszczenie instrukcji**

Większość języków w tym C++ dopuszcza wiele instrukcji w jednej linii. np:

```
x = a * 2;  if (x > 10) { b = x }
```

Grupowanie wielu instrukcji w jednym wierszu zazwyczaj jest niekorzystne. Program staje się trudno czytelny. Poza tym uniemożliwia to zastosowanie np wcięć w kodzie poprawiających czytelność.

Lepszym rozwiązaniem jest umieszczanie tylko jednej instrukcji w wierszu:

```
x = a * 2;  
if (x > 10)  
{  
    b = x  
}
```

## **Nawiasy kosztują mniej niż błędy**

Prawidłowo użyte nawiasy znacznie poprawiają czytelność programów. Wykonywanie operacji arytmetycznych i logicznych zależy od priorytetów operacji. Programista może przestać na niewielkiej ilości nawiasów i obliczenia wykonają się poprawnie. Jednak często warto mimo poprawności wprowadzić dodatkowy nawias. Bardzo ułatwia to czytelność kodu. Można też pomylić się nie zauważając jakiegoś priorytetu i dzięki nawiasom można uniknąć takich błędów.

Przykładowe zastosowanie nawiasów dodatkowo dla czytelności:

```
a*b*c/(d*e*f)    //czytelnie
(a*b*c)/(d*e*f)  //bardziej czytelnie
```

## **Stosuj wcięcia dla uwidocznienia struktury programu**

Stosowanie wcięć w kodzie jest bardzo ważne. Nie mają one wpływu na kompozycję programu ale znacznie poprawiają jego czytelność. Przykład wcięć w kodzie:

kod bez wcięć:

```
if(data[i] == '#')
{
  if(stan != 4)
  {
    save_stan = stan;
    stan = 4;
  }
  else
  stan = save_stan;
}
```

oraz kod z wcięciami:

```
if(data[i] == '#')
{
    if(stan != 4)
    {
        save_stan = stan;
        stan = 4;
    }
    else
        stan = save_stan;
}
```

Wcinanie ma przede wszystkim związek z grupami instrukcji wyodrębniania przez przykładowe pary:

w pascalu:

```
begin ... end;
```

w C++:

```
{ ... }
```

Miejsce, w którym często można stosować wcięcia są pętle. Aby uzmysłowić czytelnikowi które 2 ciągi stanowią parę należy zapisać wcięciami na tym samym poziomie. Pod bloki takie jak wewnątrz pętli czy ciąg instrukcji w wyrażeniu warunkowym “if” wcinamy o poziom dalej, żeby uwidocznić, które fragmenty kodu należą do którego wyrażenia. Skomplikowane instrukcje “if” zawierające złożone zagnieżdżone wyrażenia (np kilka instrukcji “if” w “if”) są o wiele czytelniejsze z odpowiednimi wcięciami.

Podczas deklaracji zmiennych warto typ zmiennej i nazwę zmiennej pisać na tym samym poziomie: np:

```
int            zmienna1;
string         zmienna2;
std::vector<int> dluga_zmienna;
```

## Podsumowanie

Początkujący programiści uważają, że piszą programy dla maszyny. Doświadczeni programiści wiedzą, że piszą je dla ludzi. Znaczna część programów będzie czytana przez wielu ludzi. Najpierw autor wielokrotnie przeczyta swój program, pisząc go i konserwując. Czytając program uczymy się pisać w danym języku. Istnieją programy których nikt nie rozumie. Dzieje się tak gdy autor programu przestał się nim zajmować, a inni zmieniali go, wcale go nie rozumiejąc.

Dodatkowy wysiłek, zmierzający do zapewnienia czytelności programu, jest niewielki w porównaniu z kosztami sprawdzania, znajdowania błędów, czy ponownego pisania.

Są dwa główne powody pisania niechlujnych programów:

1. Ma to być program napisany “byle jak, byle szybciej” nie mający ogólnego zastosowania
2. Program jest pisany w pośpiechu, bo upływa termin jego wykonania (tzw. deadline)

Najlepiej pisać program od początku dobrze.

## Projektowanie programu

*Zawsze łatwiej jest zmienić kod, którego jeszcze nie napisałem.*

Projektowanie jest bardzo istotne. Każdy poważniejszy program trzeba zaprojektować przed rozpoczęciem jego implementacji, bo przystąpić do niej wiedząc już dokładnie, jak program ma działać i jak ma być zbudowany. Jest to konieczne, ponieważ dużych programów nie jest w stanie ogarnąć swoim umysłem jedna osoba i informacje o takich programach (a nie tylko same programy) powinny zostać usystematyzowane i spisane, by służyły wielu ludziom.

Pojęcie projektowania (rozumiane ogólnie, jako jego planowanie przed przystąpieniem do implementacji) współcześnie dzieli się na kilka faz, spośród których można wyróżnić co najmniej trzy główne:

1. **Określenie wymagań**, polegające na opisanu w sposób bardziej lub mniej usystematyzowany wymogów, jakie ma użytkownik co do tworzonego programu. Opis wymagań powinien obejmować zewnętrzne zachowanie systemu – powinien mówić “co” program ma robić, a nie “jak” ma to robić.
2. **Analiza**, polegająca na stworzeniu modelu systemu. Model opisuje, jak program będzie realizował swoje funkcje i jak będzie zbudowany (z jakich części będzie się składał), ale tylko od strony logicznej – bez wdawania się w szczegóły implementacyjne zależne od środowiska, w którym będzie tworzony.
3. **Projektowanie**, polegające na zaplanowaniu szczegółów fizycznej implementacji programu w konkretnym środowisku, z uwzględnieniem jego charakterystyki, możliwości i ograniczeń.

## Dążenie do prostoty

*Wszystko powinno być tak proste, jak to tylko możliwe, ale nie prostsze.*  
(Albert Einstein)

Tworząc programy należy dążyć do jak największej prostoty. Ta prawda znana jest jako reguła KISS - *Keep It Simple, Stupid*. Swój kod powinno się zawsze pisać bez udziwnień, w sposób prosty i przejrzysty - zarówno na wysokim poziomie (struktura programu powinna być logiczna i czytelna), jak i na niskim (instrukcje programu powinny być proste i zrozumiałe).

Używanie zawiłych konstrukcji i wymyślnych sztuczek może się niektórym wydawać dobre, choćby dlatego, że programiści piszący w ten sposób stają się w firmie niezastąpieni. Z punktu widzenia dyrekcji jednak nie jest to korzystne. Poza tym programiści sami po jakimś czasie będą

mięli problemy z modyfikowaniem własnego kodu. Dlatego warto dążyć do prostoty i konsekwentnie trzymać się w swoim kodzie pewnych konwencji, które zapewnią mu czytelność.

## **Czytanie programów**

*Napisać kod czytelny dla komputera jest łatwo.*

*Sztuką jest napisać kod czytelny dla człowieka.*

Pisanie kodu tym różni się od pisania utworów literackich, że kod po napisaniu nie tylko może być czytany, ale i wykonuje się – działa. Prawdopodobnie właśnie to czyni programowanie tak fascynującym. [2] Jednakże w praktyce zawodowego programisty podobno częściej czyta się kod, niż się go pisze. Dlatego kod powinien być nie tylko poprawny i dający się uruchomić na komputerze, ale i czytelny dla programisty.

Czytanie kodu to także dobry sposób na znajdowanie błędów. Są to tzw. testy statyczne. Kod warto również czytać już na etapie powstawania, gdyż świadomość tego faktu mobilizuje programistów do pisania kodu lepszego, bardziej zrozumiałego i lepiej zorganizowanego, niż gdyby pisali go sami.

Technika ta znajduje zastosowanie m.in. w tzw. programowaniu ekstremalnym (XP – *Extreme Programming*), którego jednym z założeń jest programowanie parami. Zawsze jeden programista pisze kod, a drugi się temu przygląda, czyta, analizuje i sprawdza oraz zadaje pytania wyjaśniające.

## **Określenie problemu**

*Każdy klient wie dokładnie, czego NIE chce.*

Bardzo ważne jest określenie wymagań wobec projektowanego programu. Bez tego nie sposób napisać dobrego programu. Specyfikacja wymagań powinna być sporządzona w sposób pisemny, kompletna, spójna i niesprzeczna. Klient powinien dokładnie wiedzieć, czego oczekuje (mieć wizję zamawianego systemu), a programiści (bądź analitycy albo inni pracownicy firmy tworzącej oprogramowanie) powinni dobrze poznać i zrozumieć potrzeby klienta. Tylko wówczas powstaje porozumienie, które pozwala napisać taki program, jakiego klient oczekuje.

Problem z poprawnym określeniem wymagań leży po obydwu stronach. Klient nie zawsze jest w stanie myśleć w sposób systematyczny i jasno precyzować swoje myśli, a przedstawiciel firmy nie musi być obeznany z dziedziną problemu, którego ma dotyczyć program. Tymczasem błędy popełnione na etapie określania wymagań są bardzo kosztowne, gdyż prowadzą do całkowicie błędnego realizowania etapów dalszych.

Sposobów na lepsze określenie wymagań klienta jest wiele. Należy zawsze dążyć do jak najczęstszych kontaktów z klientem i dawać mu do weryfikacji efekty pracy, by na bieżąco zgłaszał swoje uwagi i sugestie. Ponadto pomocne może się okazać szybkie zbudowanie prototypu, który umożliwi klientowi ujście szkieletu działającego systemu, a tym samym lepsze zrozumienie i sprecyzowanie swoich oczekiwań.

## **Wybór algorytmu**

Ważne jest też dobranie jak najlepszego algorytmu, odpowiedniego do rozwiązywanego zadania. Przed rozpoczęciem kodowania należy rozważyć kilka możliwych algorytmów i wybrać ten najlepszy. Liczy się jego asymptotyczna złożoność, ale także praktyczna przydatność w danym miejscu, wymagana zajętość pamięci i czas wykonywania oraz inne parametry.

Źródłem algorytmów mogą być klasyczne książki z algorytmiki (jak [3], [4]) i inne poświęcone temu tematowi, a także literatura dotycząca danej dziedziny związanej z tworzoną programem.

## **Opis danych**

Równie istotny (albo jeszcze istotniejszy) jak wybór algorytmu jest odpowiednie dobranie struktur danych, czyli sposób zapisu i reprezentowania danych w pamięci. Determinuje on sposób, w jakim program będzie miał dostęp do tych danych i będzie mógł na nich operować, a tym samym decyzja ta musi być w jakiś sposób związana z wyborem algorytmu. Możliwe do zastosowania struktury danych oraz ich wydajność i łatwość implementacji w dużej mierze zależy także od używanego języka programowania.

Informacje o strukturach danych można znaleźć wraz z algorytmami na nich operującymi w książkach poświęconych algorytmice. Istnieją też reprezentacje danych charakterystyczne dla różnych dziedzin np. metody zapisywania macierzy w metodach numerycznych czy techniki podziału przestrzeni w programowaniu gier.

## **Wybór języka programowania**

Język programowania trzeba dobrać odpowiednio do rozwiązywanego problemu. Oczywiście istnieją też inne ograniczenia, jak język stosowany w danej firmie, umiejętności i znajomość języków swego i innych członków zespołu czy opłaty licencyjne na kompilatory i inne narzędzia.

Istnieją języki uniwersalne (jak C++) oraz specjalizowane, przeznaczone albo po prostu lepiej nadające się do konkretnych zastosowań. Do wyboru są języki operujące na różnym poziomie – od bardzo niskopoziomowych (jak assembler), poprzez C i C++, narzędzia typu RAD aż po języki deklaratywne, np. SQL.

Na podejmowaną decyzję o wyborze języka programowania powinny mieć wpływ rozważania dotyczące kryteriów takich jak wydajność kodu tworzonego w danym języku, łatwość i wygoda programowania oraz prawdopodobieństwo popełnienia błędów (na które ma wpływ choćby ścisła kontrola typów), dostępne narzędzia i wyposażenie biblioteki standardowej danego języka oraz wiele innych cech.

## **Biblioteki**

*Nie warto wyrażać otwartych drzwi.*

Warto korzystać z gotowych bibliotek. Do wykonania jakiegoś zadania trudno byłoby napisać od podstaw wszystkie czynności elementarne dla niego potrzebne. Trzeba zawsze operować na pewnym poziomie abstrakcji żeby być w stanie ukończyć program, dlatego bardziej podstawowe procedury, o ile to możliwe, powinny być zapewnione przez zewnętrzne biblioteki.

Staje się to tym istotniejsze, im większe i bardziej złożone są programy. Modularna budowa jest wtedy koniecznością, a montaż z gotowych komponentów jedną z ważnych technik programowania. Jakkolwiek nie sposób złożyć dobrego i wydajnego programu z samych tylko elementów zewnętrznych, to korzystanie z bibliotek jest ważne i potrzebne – czy to będzie biblioteka standardowa danego języka programowania, biblioteki dostępne za darmo wraz z kodem źródłowym czy komercyjne, rozbudowane moduły.

## **Postać wejścia-wyjścia**

Programy należy pisać tak, żeby postać danych wejściowych i wyjściowych była jak najkorzystniejsza. Jest to istotne nie tylko w sensie tekstowych wydruków pełnych liczb, ale także dziś, w dobie nowoczesnych programów współpracujących ze sobą w graficznym, okienkowym systemie.

Warto by program obsługiwał jak największą liczbę różnych formatów plików, w tym również otwarte standardy, których specyfikacja jest dostępna i które nie są objęte żadnymi obostrzeniami prawnymi, licencjami czy patentami.

Wyboru własnego formatu reprezentacji danych zachowywanych na dysku czy przesyłanych przez sieć należy dokonywać biorąc pod uwagę kryteria takie jak oszczędność miejsca i szybkość przetwarzania. Może to być np. postać binarna, tekstowa czy XML.

## **Wygoda operatora**

Wygoda obsługi programu dla użytkownika końcowego to coś, o czym często zapomina się na etapie projektowania i potem implementacji, a co jest tak naprawdę najistotniejsze. Dlatego należy zawsze postawić się w roli odbiorcy programu i spojrzeć na aplikację z jego perspektywy – bądź umożliwić użytkownikom jego przetestowanie i ocenę wygody obsługi.

Jest to zagadnienie istotne także dzisiaj, kiedy nie ma potrzeby zatrudnienia operatora zakładającego taśmy czy karty perforowane do komputera. W czasach kiedy komputery i inne urządzenia sterowane oprogramowaniem (jak choćby telefony komórkowe) są powszechnie dostępne a od użytkownika nie można oczekiwać żadnego specjalnego wykształcenia, obsługa programów musi być tym bardziej prosta, wygodna i intuicyjna.

## **Cele**

Warto jasno określić cele, jakie przyświecają twórcom programu. Jest wiele pozytywnych cech i do wszystkich nich warto dążyć – np. wysoka niezawodność, ukończenie na czas, łatwość konserwacji i wprowadzania zmian, wydajność, uniwersalność, łatwość użycia. Jednakże w konkretnej sytuacji jedne spośród nich mają większe znaczenie, a inne mniejsze. Dlatego należałoby ustalić, co jest priorytetem w przypadku danego programu. Precyzyjne określenie celów pozwoli uniknąć sytuacji, w której każdy ze współtwórców programu patrzy na niego z innego punktu widzenia.

Ponadto warto zachować skromność celów, by nie dążyć do uczynienia programu nazbyt rozbudowanym, uniwersalnym i wszechstronnym. Takie podejście nie jest dobre, ponieważ łatwo można przekroczyć założone terminy i budżet, napisać program bardzo obszerny i powolny, pełen nieużytecznych i niezwiązanych ze swoim przeznaczeniem funkcji czy nawet w ogóle go nie ukończyć.

## **Złożoność**

Wszyscy znamy dwa wymiary programu: pamięć i czas. Jednak istnieje jeszcze trzeci wymiar, nad którym trzeba sprawować kontrolę – złożoność. Programy komputerowe należą do najbardziej złożonych struktur, jakie kiedykolwiek budował człowiek. Stają się one przy tym coraz bardziej skomplikowane, a wzrost ten następuje w bardzo szybkim tempie – dużo szybszym, niż rozwój technik tworzenia oprogramowania. Zjawisko to nazywamy kryzysem oprogramowania i właśnie jego zażegnaniu służy inżynieria oprogramowania. [5]

Metod zapanowania nad złożonością jest kilka. Najważniejsza z nich to zdecydowanie podział problemu na mniejsze, możliwie niezależne części i rozpatrywanie ich osobno. Każda z tych części musi mieć przy tym określony zakres odpowiedzialności, funkcje które ma realizować oraz mieć jasno i dobrze zdefiniowany interfejs, czyli wejście-wyjście, przez które komunikuje się z resztą systemu. Sposobem na zapanowanie nad złożonością danego modułu jest technika programowania zstępującego.

## **Programowanie strukturalne**

*Nic w programowaniu nie jest trudne, wszystko sprowadza się do odpowiedniego wywołania*

*odpowiednich funkcji z odpowiednimi parametrami w odpowiedniej kolejności ;)*  
(Dexio)

Dużym krokiem w rozwoju metod programowania było przejście z programowania liniowego (polegającego na pisaniu ciągu instrukcji przeplatanych instrukcjami skoku) do programowania strukturalnego (polegającego na wpisywaniu instrukcji warunkowych, pętli i wywołań funkcji). Podstawowe zasady programowania strukturalnego to:

1. Projektowanie zstępujące
2. Programowanie modularne
3. Kodowanie strukturalne

Projektowanie zstępujące polega na sporządzanie projektu od ogółu do szczegółu. Zaczyna się od najbardziej ogólnego opisu, by potem przejść do szczegółów każdego wypisanego kroku, szczegółów tych szczegółów itd. tworząc swoistą hierarchię. Taka organizacja procesu projektowania umożliwia powstanie dobrego projektu obejmującego podział programu na części oraz ich dokładne określenie. Ten sposób myślenia dokładnie odpowiada bowiem organizacji strukturalnego kodu, w którym funkcja główna wywołuje pewne funkcje, te wywołują kolejne funkcje itd.

Programowanie modularne zakłada podział programu na moduły i ich niezależne pisanie, każdego osobno. Moduły powinny stanowić logiczną jednostkę programu, powinny być możliwie jak najmniejsze i jak najbardziej niezależne od innych, a z resztą systemu komunikować się za pomocą jasno zdefiniowanego i prostego interfejsu (zbioru funkcji).

Mitem jest, że całkowite zakazanie stosowania instrukcji skoku oraz narzucenie ścisłego przestrzegania zasad programowania strukturalnego wraz z całą "ideologią", jaka się za nim kryje zagwarantuje wysoką jakość projektów. Ani wcale się tak nie stanie automatycznie samo z siebie, ani stosowanie instrukcji skoku nie jest zawsze złe. Wszystko zależy od programisty.

Obecnie programy są bardziej złożone o całe rzędy wielkości, a dominującą techniką ich tworzenia jest programowanie obiektowe. To kolejny poziom w rozwoju metod programowania i inżynierii oprogramowania (metodyka obiektowa nie dotyczy bowiem wyłącznie kodowania, ale także analizy i projektowania), jednak fundamentalne zasady nadal pozostają aktualne. Ścisłe trzymanie się zasad obiektowości samo nie zagwarantuje wysokiej jakości projektów. Dlatego większość współczesnych języków programowania pozwala też na stosowanie konstrukcji znanych z programowania strukturalnego, jak funkcje i zmienne globalne.

## **Praca w zespole**

Tworzenie dużych programów wymaga współpracy wielu osób. Dlatego istotna jest organizacja pracy. Model "zespołu głównego programisty" zakłada, że zespół powinien się składać z programisty głównego, programisty pomocniczego oraz bibliotekarza. Programista główny musi być osobą o dużej wiedzy i zdolnościach organizacyjnych, aby kierować całym przedsięwzięciem. Jego obowiązkiem jest całościowe zarządzanie, zarówno z punktu widzenia tworzonego programu, jak i personalnego. To on decyduje o kształcie ogólnego projektu i o podziale zadań. Programista pomocniczy pomaga programiście głównemu, a w razie potrzeby musi być w stanie go zastąpić. Bibliotekarz ma nieco inny zakres obowiązków, niż dwaj pierwsi. Wykonuje czynności pomocnicze, jak utrzymywanie biblioteki funkcji.

Po tak wielu latach rozwoju branży informatycznej nadal nie ma jasności co do skuteczności takiej a nie innej organizacji pracy w zespole podczas pisania programów. Także dziś proponuje się różne modele, jak choćby programowanie ekstremalne. Jasne jest, że do zrealizowania dwukrotnie większego programu nie wystarczy dwa razy więcej ludzi i dwa razy więcej czasu. Statystycznie rzecz biorąc wydajność programistów może wynosić zaledwie kilka instrukcji dziennie, gdyż reszta czasu poświęcana jest na projektowanie, a także testowanie, uruchamianie i komunikację między członkami zespołu. Trudno uwzględnić wszystkie te czynniki.

## **Biblioteka wspomagająca**

Van Tassel wspomina o bibliotece wspomagającej konstrukcję jako zbiorze wszystkich funkcji, który powinien być pod opieką specjalnie w tym celu zaangażowanej osoby – bibliotekarza. Dziś ten problem nadal pozostaje aktualny, choć innego słownictwa używa się na jego określanie.

Współczesne narzędzia wspomagające całosciowy proces tworzenia oprogramowania – tzw. narzędzia CASE – posiadają bazę danych zwaną repozytorium, w której składuje się informacje o wszystkich elementach systemu i ich wzajemnych powiązaniach, jak komponenty, klasy czy pojęcia dotyczące modelu zebrane w słownik. Kolejne wersje kodu są natomiast w miarę postępu prac zbierane i zachowywane w wielodostępnej bazie z pomocą programów zwanych systemami kontroli wersji, jak CVS, SVN, Visual SourceSafe czy TeamSystem. To znacznie usprawnia wspólną pracę nad kodem programu.

## **Konserwacja**

*Nowa wersja programu to program, w którym poprawiono stare błędy, a zrobiono nowe.*

Pisząc program należy z góry myśleć o tym, aby był on łatwy w konserwacji. Na ukończeniu programu jego cykl życia się bowiem nie kończy i zawsze zachodzi potrzeba jego poprawiania, ulepszania czy innego modyfikowania. Dlatego lepiej myśleć o tym od początku i już na etapie projektowania uczynić program dostosowanym do przyszłych modyfikacji, aby te były łatwe i w ogóle możliwe. Dobry program powinien być:

- przenośny – dający się łatwo dostosować do działania w innym środowisku
- adaptowalny – dający się łatwo dostosować do zmieniających się wymagań użytkownika
- łatwy w znajdowaniu i usuwaniu błędów
- łatwy w rozbudowie o nowe możliwości
- łatwy we wprowadzaniu zmian i poprawek

## **Dokumentacja**

Zawsze trzeba sporządzać dokumentację programu, aby był on dobrze opisany. Cokolwiek nie zostanie zapisane, popadnie w niepamięć. Informacje na temat programu służą nie tylko innym programistom, ale i samemu autorowi danego fragmentu kodu, gdyż za jakiś czas zapomni szczegółów z nim związanych. Dokumentację trzeba zacząć sporządzać już od początku tworzenia programu, a nie dopiero po jego ukończeniu. W sytuacji braku dobrej dokumentacji korzystne może się bowiem okazać w przyszłości napisanie programu od nowa, niż analizowanie jego kodu.

## **Pisanie od nowa**

Jeśli jednak nie ma już innego wyjścia, nie należy się bać pisania od nowa. Programowanie to praca twórcza i jak każda taka, często wymaga odrzucenia pewnych efektów pracy. Jeśli kod jest niezrozumiały, nieudokumentowany, pełen błędów albo nieprzemyślany i źle zorganizowany, lepiej i szybciej jest napisać dany fragment od początku, niż przerabiać istniejący. Nigdy nie należy na siłę wprowadzać modyfikacji do takiego kodu, by nie brnąć w coraz większe kłopoty.

## **Bibliografia**

1. Dennie van Tassel, Praktyka programowania. Wyd. WNT.
2. Brooks Frederick P., Jr., Mityczny osobomiesiąc. Eseje o inżynierii oprogramowania. Wyd. WNT.
3. Cormen, Leiserson, Rivest. Wprowadzenie do algorytmów. Wyd., WNT.
4. Knuth Donald E. Sztuka programowania. Wyd. WNT.
5. Jaskiewicz Andrzej, Inżynieria oprogramowania. Wyd. Helion.