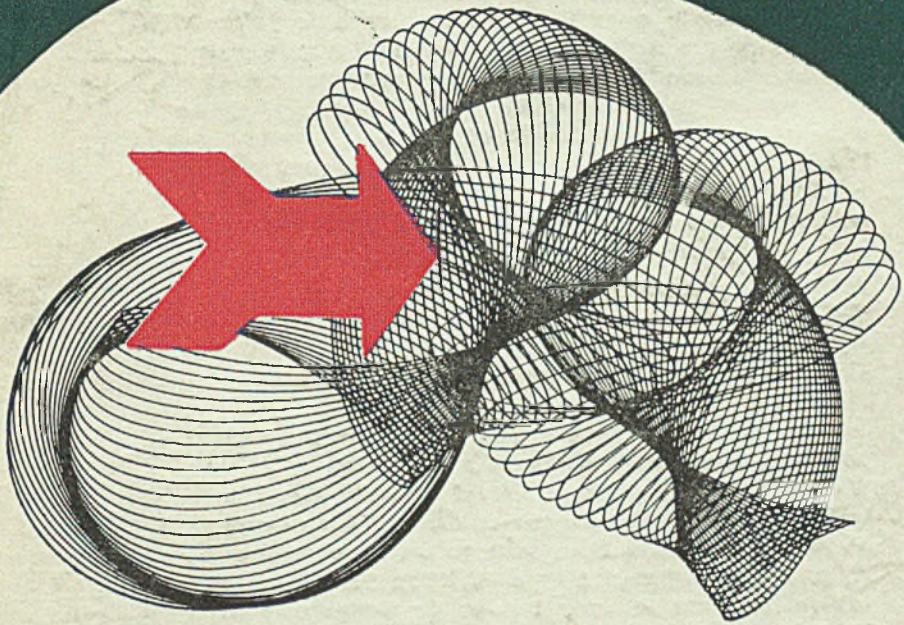
INFORMATYKA
W PRAKTYCE



Zdzisław Szyjewski

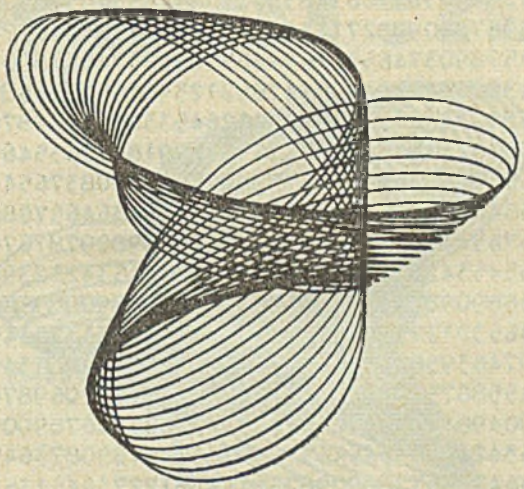
Programowanie strukturalne w przetwarzaniu danych

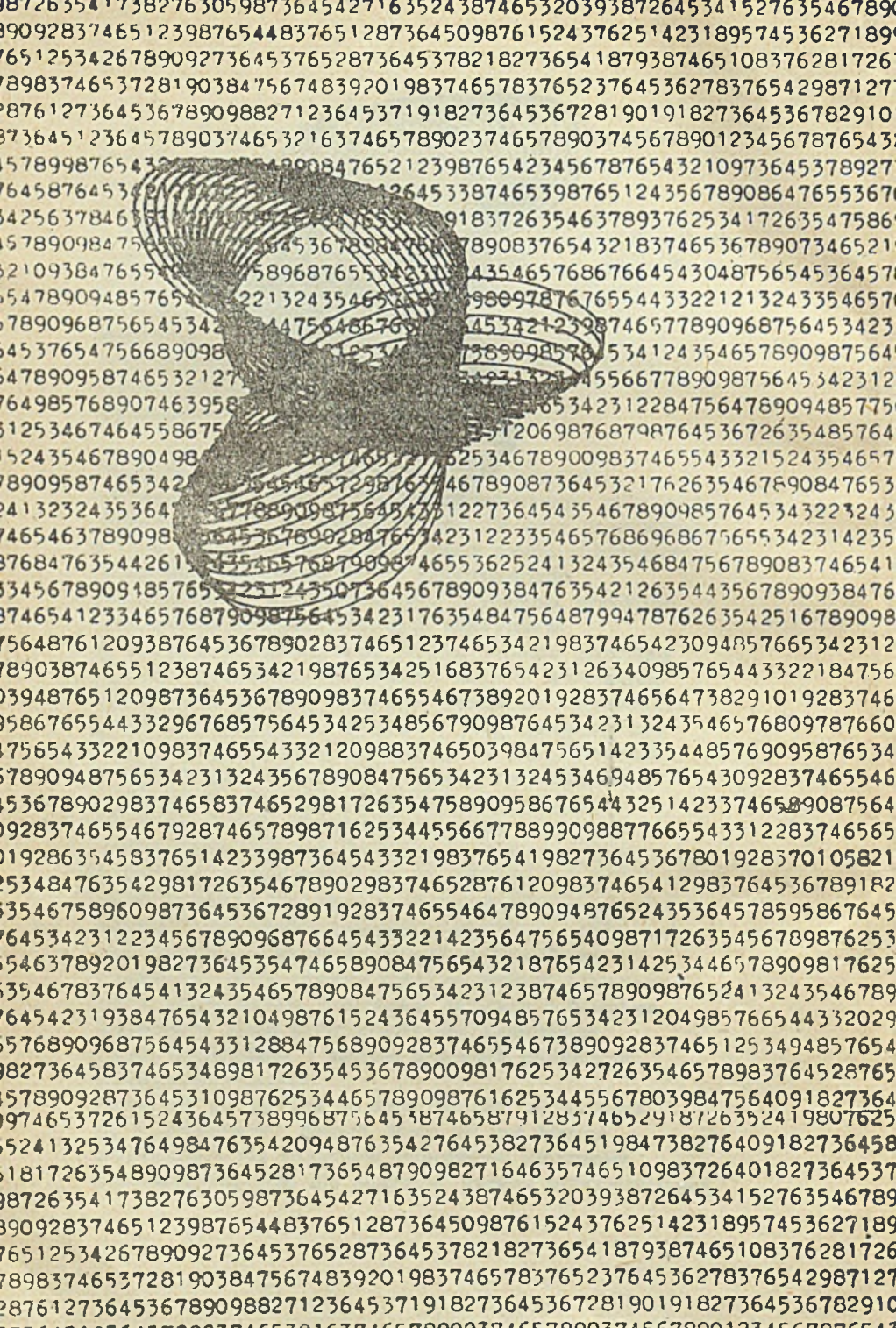
PWE

Seria wydawnicza „Informatyka w praktyce" zaspokaja potrzeby poznania literatury fachowej, poświęconej popularyzowaniu i wykorzystywaniu rozwiązań i zastosiwań informatyki, a zwłaszcza budowania i funkcjonowania systemów informatycznych w jednostkach gospodarczych oraz doskonalenia sposobów uzyskiwania informacji w zarządzaniu. Poszczególne pozycje serii są przeznaczone dla pracowników służby informacyjnej we wszystkich jednostkach gospodarczych, kadr kieriwniczych tych jednostek, studentów wyższych uczelni, uczestników kursów podyplomowych oraz osób interesujących się zagadnieniami i rozwojem informatyki.

*

Podejście strukturalne pozwala na rozwiązywanie złożonych problemów w sposób systematyczny, przejrzysty, czytelny i łatwy do modyfikowania. Te cchy, w połączeniu z dużą niezawodnością i sprawnością metod strukturalnych, są źródłem stosowania ich w programowaniu.
Programowanie strukturalne jest szczególnie przydatne jako metoda realizacji złożonych algorytmów.
W systemach przetwarzania danych ponadto algorytmy iperują na złożonych strukturach danych. Rozwiązanie problemu za pomocą komputera polega na opracowaniu poprawnie działającego programu, który powinien być również czytelnie napisany i łatwy do modyfikowania.
O poprawności rozwiązania i jakości opracowywanego prigramu decyduje faza projektowania programu.
Autor po podaniu charakterystyki programowania strukturalnego koncentruje się na metodach projektowania programów, wykorzystujących zasady podejścia strukturalnego i stosowanych w przetwarzaniu danych.
Metody projektowania programów oparte na strukturach przetwarzanych danych, są szczególnie przydatne w praktyce. Wiele miejsca poświęca również omówieniu metod projektowania programów Jacksona i Warniera.
Wyjaśnianie zasad postępowania jest ilustrowane bogato przykładami z zakresu typowych problemów przetwarzania danych.
Praca jest adresowana do informatyków zajmujących się przetwarzaniem danych. Może też być lekturą uzupełniającą dla studentów informatyki kierunków ekonomicznych oraz dla słuchaczy kursów informatycznych.





INFORMATYKA
W PRAKTYCE

Programowanie strukturalne w przetwarzaniu danych

Zdzisław Szyjewski

Państwowe Wydawnictwo Ekonomiczne
Warszawa 1984

S.03

Komitet Redakcyjny serii
INFORMATYKA W PRAKTYCE

JANUSZ GOŚCIŃSKI, TADEUSZ JAEGERMANN,
TADEUSZ PECHE (przewodniczący),
TADEUSZ WAŁCZAK, WŁADYSŁAW RADZIKOWSKI

Okładkę projektował
FRANCISZEK WINIARSKI

Redaktor
WANDA ROWICKA

S.13

007.52:681.327:519.256
681.3.06

POLITECHNIKA ŚLĄSKA
Biblioteka Główna
w Gliwicach

90272

© Copyright by
Państwowe Wydawnictwo Ekonomiczne
Warszawa 1984

ISBN 83-208-0373-X

A12.185

Spis treści

Od Autora 7

I. Elementy programowania komputerów 11

1. Miejsce programowania w projektowaniu systemów . . . 11
2. Istota procesu programowania 17
3. Metody i techniki programowania 26

II. Programowanie modularne 38

1. Podział programu monolitycznego 38
2. Podstawowe pojęcia programowania modularnego . . . 41
   a. Hierarchia modułów 42
   b. Pośrednictwo międzymodułowe 44
3. Opracowanie programu modularnego 44

III. Podejście strukturalne do programowania 50

1. Rozwój programowania strukturalnego 50
2. Charakterystyka programowania strukturalnego 54
3. Czytelność programu 64
4. Macierzowa metoda definiowania algorytmu dla tabulogramu 71

IV. Planowanie programu 79

1. Metody planowania programu 79
2. Metoda Jacksona 81
   a. Przykład wprowadzający 81
   b. Sekwencja 85
   c. Iteracja 88
   d. Selekcja 94
   e. Zasady postępowania 99
   f. Złożone przykłady stosowania metody 106
   g. Przykład obsługi końca zbioru 117
3. Metoda Warniera 126
   a. Założenia metody 126
   b. Struktura powtarzania 127
   c. Struktura wyboru 137

d. Struktury złożone 146

V. Kodowanie programu 156

1. Zasady kodowania strukturalnego 156
   a. Sposób wykorzystania języka programowania 159
   b. Mikroefektywność 160
   c. Komentarze w programie źródłowym 161
   d. Deklaracja danych 162
   e. Struktura programu 164
   f. Inne uwagi 166
2. Wspomaganie programowania strukturalnego przez języki programowania 167

VI. Dokumentowanie prac z wykorzystaniem podejścia strukturalnego 174

1. Schemat blokowy programu 174
2. Inne techniki dokumentowania prac 179

Literatura 189

Od Autora

Ostatnie lata charakteryzują się wyraźną poprawą w zakresie polskich wydawnictw książkowych z zakresu informatyki. Ukazały się liczne, ciekawe i wartościowe pozycje książkowe zarówno polskich autorów, jak i tłumaczenia zagranicznych wydawnictw. Na szczególną uwagę zasługują serie wydawnicze, takie jak „Biblioteka inżynierii oprogramowania" w Wydawnictwie Naukowo-Technicznym i „Informatyka w praktyce" w Państwowym Wydawnictwie Ekonomicznym.

Pierwsza seria koncentruje się na przedstawianiu trendów metodologicznych w programowaniu, druga natomiast — więcej uwagi poświęca zastosowaniom praktycznym informatyki. Programowanie strukturalne ma w metodologii programowania określoną, wysoką pozycję. Niniejsza praca jest próbą przedstawienia tej metody programowania dla potrzeb praktyki przetwarzania danych.

Wprowadzenie projektowania oprogramowania do metodyki projektowania systemów przetwarzania danych, jest ściśle związane ze stosowaniem ulepszonych metod programowania. W metodach tych szczególną uwagę koncentruje się na fazie planowania oprogramowania. Faza planowania programów jest decydująca dla przebiegu całego procesu programowania. Stąd też właśnie metody planowania programu są dominującą częścią tej książki.

Koncentrując się na metodach Jacksona i Warniera planowania programu należy podkreślić przede wszystkim walory praktyczne tych metod i dużą ich prostotę. Liczne przykłady wspierające omawiane problemy mają na celu ułatwienie Czytelnikowi zrozumienia prowadzonego wywodu, jak i dostarczenia typo-

wych wzorców, mogących mieć zastosowanie w praktyce programowania. W celu pełnego zrozumienia przedstawionych przykładów przydatna jest znajomość programowania w języku PL/1 lub języku COBOL, a brak umiejętności programowania w tych językach pozwala śledzić tok wywodu obu metod, bez ostatniej fazy, którą jest kodowanie.

Rozdział I jest wprowadzeniem w problematykę programowania systemów informatycznych. Po scharakteryzowaniu miejsca programowania oraz podstawowych funkcji etapu programowania systemów, omówiono zasady pisania programów. Szkicowe pokazanie procesu programowania pozwala na omówienie specyficznych cech niektórych metod programowania, których naturalnym rozwinięciem jest metoda programowania strukturalnego. Rozdział ma również na celu jednoznaczne określenie pojęć i terminów stosowanych w dalszych rozważaniach.

Wśród metod programowania szczególną rolę mają metody podziału programu na mniejsze jednostki programowe. Problematyce tej poświęcony jest II rozdział, w którym podane są podstawowe pojęcia i zasady programowania modularnego.

Rozdział III odpowiada na pytanie, na czym polega podejście strukturalne do programowania. Przedstawiając zasady strukturalnego programowania podkreślono cechy, jakie ma program tak tworzony. Przedstawione są rozwiązania wpływające na lepszą przejrzystość i czytelność źródłowego programu.

W następnych rozdziałach przedstawione są kolejne fazy tworzenia programu: planowanie, kodowanie i dokumentowanie. Szczególnie dużo miejsca poświęcono fazie planowania programu. W rozdziale IV omówiono metody Jacksona i Warniera planowania programu. Przedstawienie zasad postępowania w obu metodach wsparte jest licznymi przykładami.

Rozdział V zawiera wskazówki praktyczne dla strukturalnego kodowania programu. Oprócz ogólnych zaleceń przedstawione są niektóre typowe rozwiązania praktyczne. Ponadto w rozdziale tym zawarta jest próba porównania najbardziej popularnych języków programowania wyższego rzędu z punktu widzenia wspomagania metody programowania strukturalnego.

Ostatni rozdział poświęcono technikom dokumentowania prac programowych, które powstały dla opisu toku procesu progra-

mowania zgodnie z metodą programowania strukturalnego. Strukturalny sposób rozwiązywania problemu przedstawiono różnymi graficznymi metodami prezentacji, które najczęściej prowadzą do miniaturyzacji i standaryzacji na wzór rysunku technicznego.

Książka nie zawiera rozdziału poświęconego najbardziej pracochłonnemu etapowi prac programowych — testowaniu. Problematyka testowania jest zbyt ważna i dlatego skrótowe przedstawienie jej w tej książce mogłoby prowadzić do błędnych wniosków. Pomijając tę problematykę polecam jako lekturę uzupełniającą książkę z serii „Biblioteka inżynierii oprogramowania" napisaną przez Glenford J. Myers pt. *Projektowanie niezawodnego oprogramowania* (WNT, 1980 r.). Część trzecia tej książki poświęcona jest testowaniu oprogramowania i może być uzupełnieniem prowadzonego wywodu.

Szczecin, maj 1983 r.

I. Elementy programowania komputerów

1. Miejsce programowania w projektowaniu systemów

Trudno określić uniwersalny model procesu projektowania systemów informatycznych ze względu na jego różnorodność i skomplikowany charakter. W zależności od przyjętej strategii projektowania powstaje inny model teoretyczny. Istnieją jednak cechy ogólne, które występują w każdym modelu projektowania, a mianowicie:

— proces projektowania jest realizowany etapami,
— proces projektowania ma strukturę nieciągłą,
— w procesie projektowania występują cykle iteracyjne.

Skomplikowany charakter procesu projektowania i jego złożona struktura powodują, że jedynym praktycznie możliwym sposobem realizacji jest podział pracy na etapy. Charakterystyczne jest to, że każdy etap po zakończeniu, jest weryfikowany w różny sposób i w razie negatywnego wyniku weryfikacji, nie następuje przejście do kolejnego etapu, ale powrót do ponownego wykonania prac danego etapu. Taki sposób działania wyjaśnia istnienie cykli iteracyjnych, co jest cechą ogólną procesu projektowania. Etapowy sposób realizacji wiąże się ściśle z cechą, mówiącą o nieciągłości. Cały proces dzieli się na kolejne etapy, które wykonywane są sekwencyjnie jeden po drugim, z przeprowadzaniem weryfikacji każdego z nich. Prowadząc dalszy podział, można poszczególne etapy podzielić na fazy, a te z kolei na kroki. Podziału tego dokonuje się w zależności od stopnia złożoności problemu i konkretnych potrzeb wykonania. W ramach poszczególnych etapów, które podlegają całościowej we-

ryfikacji, fazy i kroki mogą być realizowane równolegle. Dominującą jest jednak technika sekwencyjnej realizacji.

Mimo różnorodnych podejść do procesu projektowania systemów informatycznych można wyodrębnić najczęściej występujące, następujące etapy prac:

— etap przedprojektowy, czyli rozpoznawania procesu,
— projektowanie właściwe, szczegółowe,
— etap kończący proces projektowania.

W odniesieniu do projektowania systemu informatycznego kolejne etapy prac bardziej szczegółowo można określić jako:

— opracowanie założeń systemu,
— opracowanie projektu technicznego systemu,
— oprogramowanie systemu,
— wdrożenie systemu.

Każdy z wymienionych etapów jest weryfikowany i dopiero po uzyskaniu pozytywnej opinii, rozpoczyna się dalsze prace nad systemem. Często poszczególne etapy są obiektem oddzielnych umów między przyszłym użytkownikiem systemu a zespołem wykonującym projekt. Zdarzają się wypadki przerwania prac po zapoznaniu się z wynikami początkowych etapów. Taka formuła współpracy użytkownika z zespołem projektującym oraz potrzeba weryfikacji etapu prac najczęściej w formie recenzji lub opinii, spowodowała potrzebę wykonywania dokumentacji zawierającej wyniki prac każdego z etapów. Stąd na projekt systemu informatycznego najczęściej składają się:

— koncepcja systemu,
— projekt techniczny,
— oprogramowanie systemu,
— zbiór instrukcji wdrożeniowych i eksploatacyjnych.

Założenia systemu zawierają ogólny model projektowanego systemu, czyli odpowiadają na pytanie co i jak należy zrobić. Założenia systemu najczęściej realizowane są w dwóch fazach:

— wypracowanie koncepcji systemu informacyjnego w obiekcie i jego automatyzacji,
— opracowanie charakterystyki technicznej, ekonomicznej i organizacyjnej obiektu w warunkach pracy systemu informatycznego.

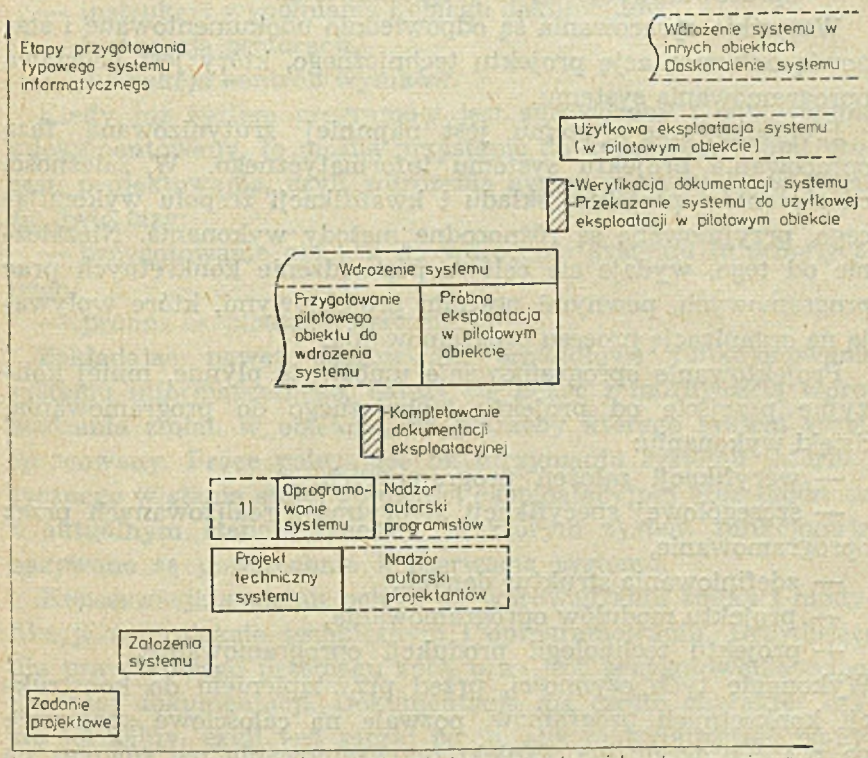
Weryfikacja tego etapu prac polega na akceptacji przez użyt-

kownika systemu rozwiązań zawartych w koncepcji. Zweryfikowane założenia systemu są obiektem dalszych szczegółowych prac nad projektem.

Projektowanie właściwe charakteryzuje się następującym tokiem prac projektowo-programowych:

— opracowanie projektu technicznego,
— projektowanie oprogramowania,
— opracowanie programów systemu,
— opracowanie projektu eksploatacyjnego (por. rys. 1).

Prace projektowo-programowe najczęściej są wykonywane przez wyspecjalizowane zespoły informatyków, które nie znając



Uwaga: Bloki przedstawionych etapów przygotowania typowego systemu informatycznego nie są ujęte w skali czasowej

1) Opracowanie projektu oprogramowania można już rozpocząć po zatwierdzeniu założeń systemu

Rys. 1. Kolejność realizacji etapów przygotowania typowego systemu informatycznego

i nie wnikając w strukturę obiektu, tworzą koncepcję zawartą w założeniach systemu. Taka organizacja prac wymaga wnikliwej weryfikacji pierwszego etapu.

Na opracowanie projektu technicznego systemu składają się przede wszystkim rutynowe prace informatyczne, mające na celu:

— opracowanie projektu dokumentów wejściowych,
— opracowanie projektu tabulogramów,
— opracowanie projektu zbiorów,
— opracowanie algorytmów,
— opracowanie systemu kodów,
— opracowanie założeń do programowania.

Wszystkie opracowania są odpowiednio udokumentowane i stanowią dokumentację projektu technicznego, który jest podstawą oprogramowania systemu.

Programowanie systemu jest najmniej zrutynizowaną fazą opracowania projektu systemu informatycznego. W zależności od złożoności projektu, składu i kwalifikacji zespołu wykonującego, przyjmowane są różnorodne metody wykonania. Niezależnie od tego, wydaje się celowe poprzedzenie konkretnych prac programowych, pewnymi pracami projektowymi, które wpływają na organizację procesu programowania.

Projektowanie oprogramowania umożliwia płynne, mniej kolizyjne przejście od projektu technicznego do programowania, dzięki wykonaniu:

— weryfikacji założeń programowych,
— szczegółowej specyfikacji algorytmów realizowanych przez oprogramowanie,
— zdefiniowania struktur danych,
— projektu modułów oprogramowania,
— projektu technologii produkcji oprogramowania.

Wykonanie tych czynności, przed przystąpieniem do konstrukcji konkretnych programów, pozwala na całościowe spojrzenie na prace programowe. Projekt oprogramowania ma również na celu efektywne wykorzystanie metod i dostępnych narzędzi programowych.

Opracowanie pojedynczych programów systemu jest wykonywane krokowo, gdzie kolejne kroki polegają na:

— opracowaniu schematu blokowego programu,
— napisaniu źródłowego programu, w kodzie wybranego języka programowania,
— realizacji przebiegów sprawdzających działanie programu,
— sprawdzeniu działania całego systemu programów,
— przygotowaniu opisów programów.

Opracowane, sprawdzone i odpowiednio udokumentowane programy mogą być eksploatowane.

Projekt eksploatacyjny jest zbiorem instrukcji umożliwiających prawidłowe funkcjonowanie systemu. Instrukcje te powstają w wyniku wcześniej wykonanych prac, lub są opracowane po zakończeniu programowania. Podstawowymi instrukcjami są:
— instrukcja wypełniania i obiegu dokumentów,
— instrukcja perforacji,
— instrukcja kontroli wyników.

Kiedy już system programów jest sprawdzony i odpowiednio udokumentowany, to można przystąpić do końcowego etapu procesu projektowania, czyli wdrożenia systemu, na który składają się dwie fazy:
— przygotowanie obiektu do wdrożenia systemu informatycznego,
— próbna eksploatacja systemu.

Zakładając nawet długotrwałe, prawidłowe funkcjonowanie systemu informatycznego, należy się liczyć z możliwością wprowadzania zmian w obiekcie, na potrzeby którego system został opracowany. Prace polegające na utrzymaniu systemu informatycznego w stanie stałej gotowości eksploatacyjnej oraz zgodności z aktualnym stanem obiektu, w którym system funkcjonuje, nazywane są powszechnie konserwacją systemu.

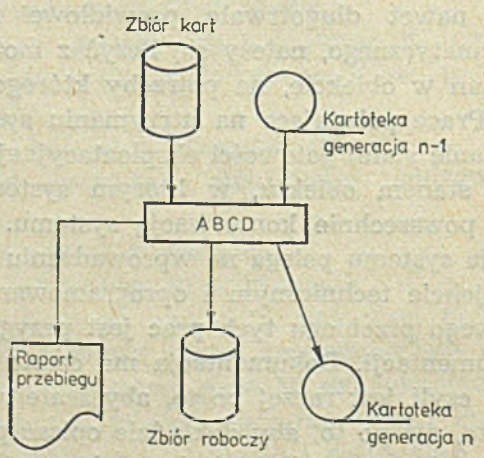
Konserwacja systemu polega na wprowadzaniu zmian i modyfikacji w projekcie technicznym i oprogramowaniu. Decydujące dla prawidłowego przebiegu tych prac jest przygotowanie odpowiedniej dokumentacji. Dokumentacja ma często charakter czysto formalny, czyli jest raczej po to, aby materializować pomysły projektanta, niż po to, aby dokładnie opisać przyjęte w projekcie rozwiązania w sposób czytelny i komunikatywny. Taka forma dokumentacji wynika z opracowywania jej obok głównych prac projektu, a nie na ich potrzeby.

Podstawową trudnością w prowadzeniu prac konserwacyjnych jest nieprzejrzystość oprogramowania, w połączeniu z mało czytelną i niekomunikatywną jego dokumentacją.

Przedstawiony w zarysie tok prac projektowych, wykonywany jest przy projektowaniu większości systemów informatycznych. O sprawności przebiegu prac nad projektem decyduje w dużej mierze podejście do programowania czyli metoda programowania. Ten etap prac będzie obiektem naszych dalszych szczegółowych rozważań, chociaż należy dodać, że etap programowania nie występuje samodzielnie i o jego strukturze i przebiegu decydują inne prace wykonywane nad projektem.

Na jakich założeniach opiera się programowanie? Programowanie opiera się na założeniach zawartych w schematach ideowych systemu, podsystemów, jednostek przetwarzania. Z tych schematów można wyodrębnić schemat ideowy dla konkretnego pojedynczego programu w systemie. Schemat taki (por. rys. 2) prezentuje podstawową funkcję programu oraz zbiory danych, które program ma przetwarzać.

Uzupełnieniem schematu jest opis szczegółowy funkcji realizowanych przez program. Funkcje realizowane przez program podawane są w formie algorytmów prezentowanych w różno-



Rys. 2. Przykładowy schemat ideowy programu

rodny sposób, od opisów słownych do wzorów matematycznych. Algorytmy zawierają opis działań na zbiorach danych, których struktura jest nieodłącznym elementem założeń do programowania.

Ponadto założenia do programowania uzupełniane są odpowiednimi, w zależności od potrzeb, wyciągami z projektu kodów oraz innymi informacjami niezbędnymi dla realizacji programu.

Tak skompletowany materiał jest podstawą prowadzenia prac programowych.

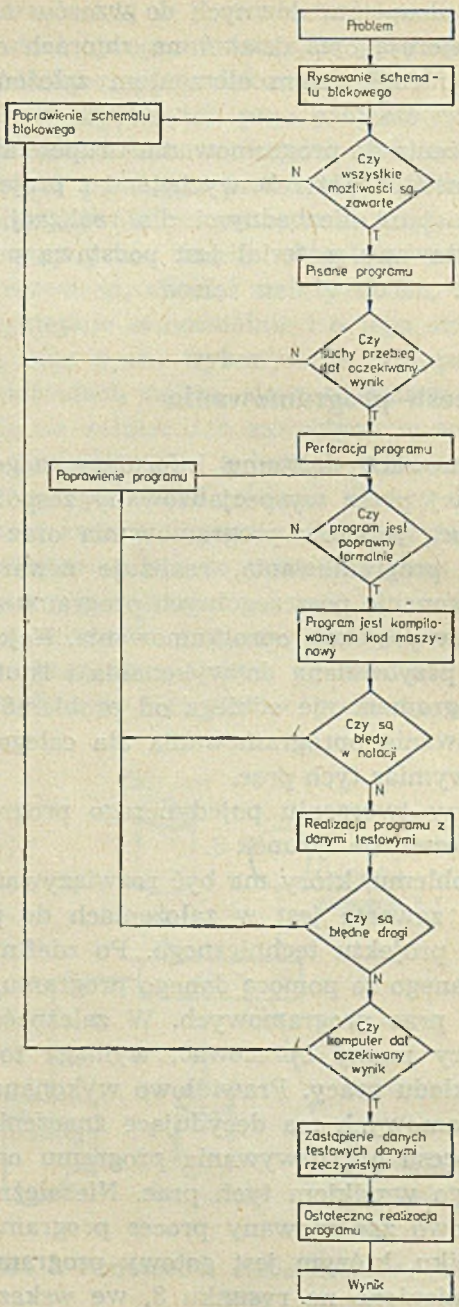
2. Istota procesu programowania

Proces programowania systemu informatycznego jest wykonywany najczęściej przez wyspecjalizowany zespół informatyków, który za pomocą języków programowania oraz innych dostępnych narzędzi programowania, realizuje zawarte w projekcie koncepcje. Wykonanie poszczególnych programów zależy od strategii przyjętej w projekcie oprogramowania. Najczęściej program jest jednostką przydzielaną do wykonania i istota prac nad pojedynczym programem nie odbiega od problemów powstających w toku opracowania oprogramowania dla całego projektu, chociaż inny jest wymiar tych prac.

Tok prac przy tworzeniu pojedynczego programu, w postaci sieci działań prezentuje rysunek 3.

Definicja problemu, który ma być rozwiązywany przez system informatyczny, zawarta jest w założeniach do programów, będących częścią projektu technicznego. Po zdefiniowaniu problemu rozwiązywanego za pomocą danego programu, przystępujemy do planowania prac programowych. W zależności od złożoności programu, który mamy opracować, wymaga to większego lub mniejszego nakładu pracy. Prawidłowo wykonana faza planowania prac programowych ma decydujące znaczenie dla płynności wykonania procesu opracowywania programu oraz jakości programu, będącego wynikiem tych prac. Niezależnie od tego, jak został szczegółowo zaplanowany proces programowania, w celu uzyskania wyniku, którym jest gotowy program, należy wykonać prace wymienione na rysunku 3, we wskazanej kolejności.

Biblioteka Główna



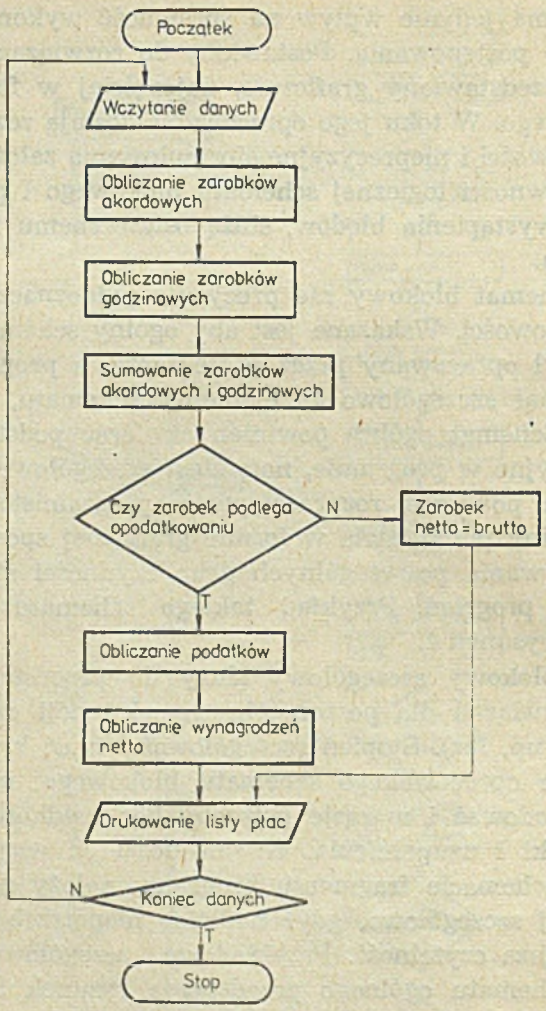
Rys. 3. Tok procesu programowania

Planowanie ma jedynie wpływ na sprawność wykonania kolejnych kroków postępowania. Postawiony do rozwiązania problem może być przedstawiony graficznie, najczęściej w formie schematu blokowego. W toku jego opracowania zostają rozstrzygnięte liczne wątpliwości i nieprecyzyjne sformułowania założeń. Sprawdzenie poprawności logicznej schematu blokowego i poprawienie go w razie wystąpienia błędów, służą ostatecznemu zdefiniowaniu problemu.

Pojęcie schemat blokowy nie precyzuje jednoznacznie stopnia jego szczegółowości. Wskazane jest aby ogólny schemat blokowy programu był opracowany przez projektanta, a programista jedynie rozwinął szczegółowo te elementy schematu, które tego wymagają. Schemat ogólny powinien zawierać podstawowe sytuacje decyzyjne w programie, natomiast szczegółowe algorytmy postępowania powinien rozpracowywać programista. Schemat blokowy ogólny przedstawia w formie graficznej sposób i kolejność wykonywania poszczególnych grup czynności składających się na dany program. Przykład takiego schematu blokowego przedstawia rysunek 4.

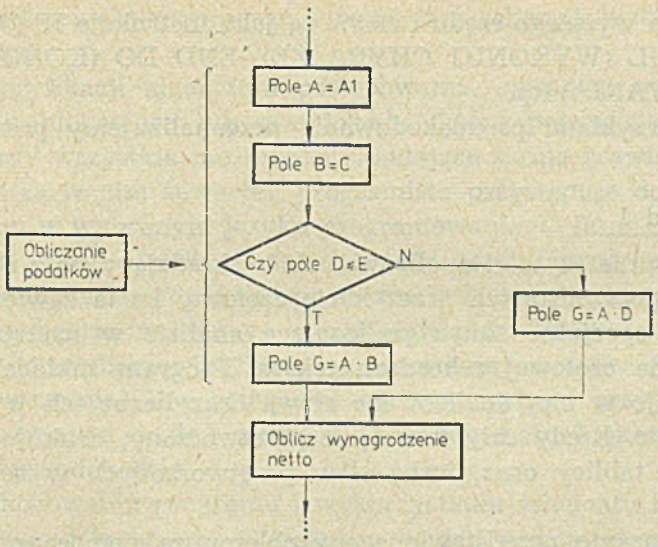
Schemat blokowy szczegółowy służy do przedstawienia dokładnych rozwiązań dla poszczególnych sekwencji działań (modułów, podgrup, faz). Stopień szczegółowości musi być taki, aby na podstawie opracowanego schematu blokowego można było bezbłędnie kodować i w razie potrzeby bez trudności wprowadzać poprawki i uzupełnienia. W zależności od wagi przedstawionego w schemacie fragmentu programu należy go przedstawiać bardziej szczegółowo, gdyż nadmiar niepotrzebnych szczegółów zmniejsza czytelność. Przykładowe, szczegółowe rozwinięcie części schematu ogólnego przedstawia rysunek 5.

Dobre projektowanie najczęściej przebiega z góry w dół (*top-down*). Wywodzi się to z potrzeby posiadania pełnego przeglądu projektowanego problemu. Ten sposób patrzenia przedstawia relacje między głównymi składnikami, ale nie uwzględnia szczegółów poszczególnych elementów. Różna szczegółowość schematów blokowych wspomaga ten sposób projektowania w metodologii tradycyjnej. Przy rozwiązywaniu złożonych i dużych problemów schemat blokowy jako nośnik specyfikacji i komunikacji procesu myślowego okazuje się zawodny ze względu na małą



Rys. 4. Schemat blokowy ogólny

przejrzystość i długość sieci działań. Poszukiwania nowych narzędzi specyfikacji i komunikowania w projektowaniu doprowadziły do wykształtowania się *pseudokodowania*. Głównym celem pseudokodowania jest umożliwienie indywidualnego sposobu myślenia w połączeniu z użyciem prostego języka w celu wyrażenia przyjętych rozwiązań.



Rys. 5. Fragment schematu blokowego szczegółowego

Pseudokodowanie skutecznie zastępuje schematy blokowe (szczegółowiej rozpatrzymy to w rozdz. IV). W tym miejscu dajemy jedynie krótką charakterystykę jego właściwości. Jest to sposób przedstawienia programu i jego poziomów. Główną zasadą prezentacji jest dyscyplina postępowania, znana każdemu projektującemu program, co dodatkowo wpływa na strukturę programu. Ponadto jest to narzędzie wspomagające odkrywanie logicznych segmentów programu. Wreszcie pseudokodowanie jest formą dokumentacji, która nie wymaga specjalnego nakładu pracy, ale powstaje w toku prowadzonych prac projektowych i jest prezentacją przebiegu procesu myślowego.

Pseudokodowanie nie jest projektowaniem w języku kompilacji. Pseudokodowanie odbywa się według kilku zasad syntaktycznych, umożliwiających pewną swobodę wyrażania myśli projektantowi w języku hipotetycznym, zamiast w języku programowania o szczegółowo rozbudowanej syntaktyce. Pseudokodowanie czyni projektowanie programu czytelniejszym i daje materiał do bardzo łatwej konwersji na kod wykonywalny maszynowo. Jednostki syntaktyczne wywodzą się z języków progra-

mowania wyższego rzędu i znane są jako instrukcje IF (JEŻELI), DOUNTIL (WYKONUJ CHYBA ŻE), END DO (KONIEC WYKONYWANIA) itp.

Dla przykładu pseudokodowania przeanalizujemy prosty problem.

Przykład 1

Program ma czytać i analizować zbiór składający się z rekordów tekstu, poprzedzonych przez kartę czołową; karta zawiera pojedyncze, specjalne słowa podlegające analizie w następujących po karcie czołowej rekordach tekstu. Program zakłada tablicę i zapisuje w niej analizowane słowa oraz liczbę ich wystąpień w tekście. Kiedy cały tekst jest przetworzony, drukuje się zawartość tablicy oraz liczbę słów przetworzonych w rekordach tekstu.

Rozwiązanie przedstawionego problemu za pomocą pseudokodu można przedstawić następująco:

```
początek programu
czytaj kartę czołową
czytaj pierwszy rekord tekstu
DOWHILE (W CZASIE KIEDY) występują rekordy tekstu
  DOWHILE (W CZASIE KIEDY) występują słowa w rekordzie
    pobierz następne słowo z tekstu
    przeszukaj tablicę słów dla pobranego słowa
    IF (JEŻELI) pobrane słowo zostało znalezione
      zwiększ licznik dla znalezionego słowa w tablicy
    ELSE (W PRZECIWNYM WYPADKU)
      wstaw pobrane słowo do tablicy
    ENDIF (KONIEC JEŻELI)
      zwiększ licznik słów przetworzonych
  ENDDO (KONIEC WYKONANIA) jak koniec rekordu
  czytaj następny rekord tekstu
ENDDO kiedy wszystkie rekordy tekstu zostały przeczytane
      drukuj tablicę i liczbę słów przetworzonych
      zakończenie programu.
```

Przekształcenie tego algorytmu postępowania, opisanego przy użyciu pseudokodu na język programowania wyższego rzędu nie powinno być większym problemem. Wynikiem procesu myślo-

wego nad projektem programu może być schemat blokowy lub pseudokod.

Po weryfikacji algorytmu postępowania, opisanego w postaci schematu blokowego lub pseudokodu (które polega na sprawdzeniu czy wszystkie możliwości składające się na rozwiązywany problem są w nim zawarte), programista przystępuje do pisania programu w wybranym języku programowania. O ile nie ma specjalnych zaleceń, jeśli chodzi o wybór języka programowania, to programista decyduje, który ze znanych mu języków programowania będzie najlepszy do zakodowania danego problemu. Decyzja o wyborze języka programowania najczęściej zapada jednak wcześniej i w zależności od tego wyboru przyjmowany jest odpowiedni stopień szczegółowości schematu blokowego lub opisu w pseudokodzie.

Po zakodowaniu programu i wykorzystaniu schematu blokowego lub pseudokodu oraz napisanego na ich podstawie programu w języku źródłowym, należy wykonać „suchy przebieg" programu. Suchy przebieg polega na wykonaniu przez programistę „na sucho", czyli bez użycia komputera, poszczególnych instrukcji programowych dla danych testowych. Dane winny być tak dobrane, aby nie wymagały wykonania zbyt skomplikowanych obliczeń i umożliwiały sprawdzenie węzłowych punktów programu. Realizacja programu na sucho ma na celu wykrycie błędów w przebiegu programu przez wykonanie kolejnych instrukcji programu (zgodnie z ich działaniem) dla konkretnych danych testowych. Ważne jest, aby wykonywanie programu odbywało się mechanicznie i konsekwentnie, instrukcja po instrukcji, bez dokonywania jakichkolwiek uproszczeń i skrótów. Dobrze jest, gdy suchy przebieg wykonuje programista, który nie jest autorem programu; sprzyja to wyeliminowaniu emocjonalnego podejścia do własnego dzieła i umożliwia pogłębioną kontrolę przyjętych rozwiązań.

Kolejnym krokiem postępowania jest perforacja programu [1]

---

[1] Przez pojęcie *perforacja programu* rozumie się przeniesienie programu na maszynowy nośnik informacji. Najczęściej jest to taśma papierowa lub karta perforowana. W związku z wchodzeniem do eksploatacji rejestratorów z taśmami magnetycznymi, *perforacja* może oznaczać również zapis na taśmie magnetycznej lub dowolny inny sposób tworzenia maszynowego nośnika informacji.

i sprawdzenie poprawności formalnej zgodnie z wymaganiami przyjętego języka programowania. Program źródłowy jest perforowany na takim nośniku danych, jaki przyjmuje kompilator danego języka programowania. Program na maszynowym nośniku danych poddawany jest procesowi kompilacji. W zależności od kompilatora, jaki został użyty, diagnostyka tego przebiegu maszynowego jest bogatsza lub zawiera tylko błędy w notacji. W razie wystąpienia błędów kompilacji należy poprawić postać źródłową programu. Poprawione instrukcje ponownie wyperforować i powtórzyć kompilację programu.

Proces ten jest kontynuowany tak długo, dopóki program nie wykazuje błędów w czasie kompilacji. Dopiero wówczas możemy przystąpić do zasadniczego sprawdzania poprawności funkcjonowania programu, czyli do testowania logiki działania programu.

Kolejne przebiegi programu z przygotowanymi danymi testowymi mają na celu sprawdzenie działania programu w różnych sytuacjach. Odpowiednie przygotowanie danych testowych winno mieć na celu sprawdzenie działania programu w warunkach późniejszej eksploatacji użytkowej.

Wykrycie błędnego działania programu na tym etapie prac, może spowodować:

— korektę programu źródłowego, np. błędnie użytej instrukcji,

— korektę w strukturze logicznej programu, czyli w schemacie blokowym.

Korekta programu źródłowego wymaga ponownego wykonania wszystkich czynności od perforacji poczynając, dla błędnej części programu. Wprowadzenie tego typu poprawki jest więc stosunkowo proste i może być szybko wykonane.

Błąd w logice programu, wykryty w czasie testowania, powoduje cofnięcie prac niemal do początku całego opisywanego procesu. Poprawienie takiego błędu wymaga dużo większego nakładu pracy, a ponadto kryje w sobie możliwość wprowadzenia nowych błędów, do już istniejącego programu. Po wprowadzeniu poprawki w logice programu praktycznie proces testowania logicznego należy w całości powtórzyć. Wykrycie oraz usunięcie błędów logicznych jest dużo trudniejsze i o wiele bardziej pracochłonne.

Jeśli rezultaty przebiegów testowych pokryją się z wynikami oczekiwanymi, to należy wykonać próbne przebiegi na danych rzeczywistych. Otrzymanie pozytywnych wyników oznacza, że program jest poprawny i może być przekazany do eksploatacji użytkowej.

Wynikiem rozwiązania problemu są:

— ostateczna wersja schematu blokowego programu,

— program źródłowy, czyli wylistowane instrukcje programu w wybranym języku programowania,

— program wynikowy, otrzymany w wyniku kompilacji programu źródłowego, zapisany w bibliotece programów systemu.

W toku wykonywania kolejnych kroków programowania tworzona jest bieżąco i poprawiana dokumentacja programu, która po przekazania go do eksploatacji jest ostatecznie kompletowana i stanowi materiał do użytkowania programu. Skompletowana dokumentacja jest podstawą wprowadzania zmian i modyfikacji w programie, w czasie jego eksploatacji użytkowej.

Na dokumentację programu najczęściej składają się:

— opis funkcji programu,

— schemat blokowy,

— wylistowany program źródłowy,

— instrukcja operowania,

— opis parametrów,

— opis danych wprowadzanych i wyników,

— lista komunikatów wprowadzanych i wyprowadzanych.

Problematyce dokumentowania programów więcej miejsca poświęcimy w rozdziale VI.

Istota procesu programowania całego projektu jest podobna do toku prac nad każdym pojedynczym programem. Po opracowaniu wszystkich programów składających się na cały system należy sprawdzić ich współdziałanie. Skompletowanie bibliotek programów dla systemu oraz odpowiedniej dokumentacji pozwala na przystąpienie do eksploatacji próbnej całego systemu.

W toku prac programowych można wyróżnić trzy następujące po sobie fazy:

— planowanie,

— kodowanie,

— testowanie.

W toku wykonywania tych faz powstają elementy dokumentacji, która jest kompletowana po pomyślnym zakończeniu testowania.

Faza planowania programu ma na celu takie przygotowanie prac, aby dalsze czynności mogły być wykonane sprawnie, szybko, bezbłędnie. Planowanie prac programowych winno umożliwiać współbieżne prowadzenie dalszych faz procesu programowania, co można osiągnąć przez podział programu na autonomiczne moduły.

Kodowanie programu ma na celu zapisanie programu źródłowego w jednym z języków programowania.

Fazę testowania wykonuje się w dwóch krokach. Pierwszy polega na doprowadzeniu do poprawności formalnej i logicznej każdego z modułów składających się na program; drugi krok polega na testowaniu współdziałania modułów w ramach całego programu i systemu.

3. Metody i techniki programowania

Rozwój sprzętu komputerowego, mający na celu podniesienie jego szybkości, niezawodności, elastyczności konfiguracji i innych cech umożliwiających konstruowanie lepszych, bardziej niezawodnych systemów informatycznych, jest w ostatnim okresie widoczny. Za rozwojem sprzętu generalnie nie nadąża rozwój metod programowania. Doskonalenie języków programowania oraz organizacji prac programowych, to tylko niewielkie próby nadążania za szybkim rozwojem sprzętu. Programowanie metodą „prób i błędów" nie spełnia wymagań stawianych przed zespołami programującymi systemy informatyczne. Stąd liczne kierunki poszukiwań ulepszonych metod programowania.

W związku ze stałym wzrostem możliwości komputerów oraz rozbudowy zestawu urządzeń zewnętrznych dołączanych do pojedynczej jednostki centralnej, powstają nowe problemy technologiczne pracy z komputerem, a jednocześnie wpływa to na styl pracy projektanta i programisty. Przed rozbudowanymi instalacjami systemu cyfrowego stawia się wyższe wymagania.

A. Colin formułuje cztery warunki stosowane do każdego systemu cyfrowego:

1. Projektowanie, kodowanie i wykonywanie algorytmów musi być maksymalnie ułatwione. Z tego wynika:
   — zapotrzebowanie na odpowiednie języki dla wyrażenia algorytmów,
   — zapotrzebowanie na specjalne mechanizmy modyfikowania, testowania i potwierdzania poprawności tych algorytmów,
   — potrzeba uproszczenia pewnych wspólnych procedur, takich jak obliczanie funkcji elementarnych lub obsługa urządzeń zewnętrznych.
2. Cały system musi być niezawodny. Należy przez to rozumieć, że:
   — po zakończeniu prac nad uruchamianiem programu należy mieć zaufanie do otrzymywanych wyników,
   — obsługa programowa powinna być regularnie dostępna i nie powinna podlegać nieprzewidzianym opóźnieniom ani przerwom. Nie powinny zdarzać się „katastrofy", które mogłyby uszkodzić lub zniszczyć informację nie dającą się natychmiast odtworzyć.
3. Zasoby używane przez program powinny być kontrolowane. To może być znowu interpretowane na dwa sposoby:
   — nie można dopuścić, aby jakikolwiek program przekroczył granicę przydziału czasu, przydziału pamięci lub objętości wyników,
   — należy zapewnić, aby podział zasobów systemu między wszystkich jego użytkowników był zgodny z taktyką ustaloną przez kierownictwo ośrodka; w systemie, w którym popyt jest regulowany przez wnoszenie opłat, każdy użytkownik powinien być bieżąco informowany o zużytych zasobach, a tam gdzie popyt jest regulowany przez racjonowanie, nie powinno się dopuścić, aby jakikolwiek użytkownik przekroczył ustalony przydział.
4. System jako całość musi być „opłacalny" co oznacza, że wszystkie jego części muszą być jak najpełniej wykorzystane, a w im większym stopniu możliwe jest zapewnienie równoległego wykonywania operacji tym lepiej [2].

Z historycznego punktu widzenia do wymagań tych podchodzono różnie. W początkowym okresie organiczenia i możliwości

[2] Por. A. Colin, *Wstęp do systemów operacyjnych*. Warszawa 1976.

techniczne komputerów determinowały w znacznym stopniu urzeczywistnianie wymienionych wymagań. Wraz z doskonaleniem sprzętu problemem stało się zarządzanie pracą systemu.

W związku z tym nastąpił rozwój języków programowania i ich bibliotek. Pozostałe postulaty realizowało ciągle rozwijane oprogramowanie operacyjne komputerów.

Systemy operacyjne ze swymi szerokimi możliwościami, wraz z obszerną klasą języków programowania, stały się podstawą rozwoju metod i technik programowania. Języki programowania w różnym stopniu wspomagały powstające metody programowania dostępnym aparatem instrukcyjnym, jak również rozwój nowych metod stał się motorem powstania nowych języków programowania.

Ogólnie, języki programowania można podzielić na trzy klasy:
— wewnętrzne,
— symboliczne,
— wyższego rzędu.

Języki wewnętrzne w obecnie eksploatowanych komputerach nie są stosowane przy programowaniu systemów informatycznych; wykorzystywane są jedynie przez służby techniczne oraz dla programowania zupełnie specyficznych algorytmów. Do najbardziej popularnych w kraju języków symbolicznych należą język programowania PLAN dla maszyn serii ODRA i język programowania ASSEMBLER dla maszyn serii RIAD. Obydwa języki programowania oprócz bogatej listy rozkazów wspomagane są przez podprogramy wywoływane z wykorzystaniem makroinstrukcji danego języka. Tak rozbudowany język programowania staje się bardzo wygodnym instrumentem w ręku programisty. Instrument ten wymaga jednak dobrej i głębokiej znajomości logiki systemu cyfrowego, aby można było go efektywnie wykorzystywać w programowaniu. Umiejętności programowania w języku symbolicznym nie zdobywa się łatwo, stąd stosunkowo wąski krąg użytkowników wykorzystuje język symboliczny do pisania w nim całego oprogramowania systemu. Częściej przy użyciu języka symbolicznego programowane są wybrane algorytmy z całego oprogramowania.

Języki programowania PLAN i ASSEMBLER charakteryzują się tym, że jedna instrukcja programu źródłowego odpowiada

jednej instrukcji maszynowej w kodzie wewnętrznym. Taki sposób programowania współpracy z urządzeniami zewnętrznymi jest bardzo kłopotliwy i pracochłonny. Producent wychodząc naprzeciw potrzebom użytkowników wzbogaca język symboliczny o makroinstrukcje współpracujące z urządzeniami zewnętrznymi. Makroinstrukcje te, umieszczone w bibliotece dostarczanej wraz z komputerem, mogą być w prosty sposób wykorzystywane w programach użytkowych, pisanych w języku symbolicznym.

Duży stopień szczegółowości języków symbolicznych sprawia sporą trudność przy pisaniu programu, ale umożliwia programiście wykorzystanie wszystkich możliwości komputera. Niektóre, bardziej specjalistyczne właściwości listy rozkazów komputera są dostępne tylko i wyłącznie na poziomie programowania w języku symbolicznym.

Najbardziej popularnym językiem programowania wyższego rzędu dla maszyn serii ODRA jest język COBOL. Charakteryzuje się on prostą i przejrzystą konstrukcją programu, jak również dużymi możliwościami. Program napisany w języku COBOL składa się z czterech działów, które mają ściśle określoną funkcję i zawartość informacyjną. Działy dzielą się na sekcje, a te dalej są podzielone na paragrafy. Sformalizowany sposób zapisu programu źródłowego wymusza na programiście porządek i ułatwia analizowanie programu źródłowego.

COBOL charakteryzują nie tylko podstawowe jednostki syntaktyczne, ale i jednostki dodatkowe, uzupełniające. Te dodatkowe jednostki ułatwiają opis rozwiązywanych zagadnień. Czyni to COBOL szczególnie atrakcyjnym. Jedną z dodatkowych możliwości kompilatorów w języku COBOL jest możliwość wykorzystania biblioteki zdań źródłowych za pomocą operatora COPY. Biblioteka zdań źródłowych umożliwia jednorazowe opisanie części programu występującej wielokrotnie lub wykorzystywanej w kilku programach. Szczególnie dotyczy to pełnych opisów zbiorów w dziale danych. Wykorzystując operator COPY oraz fazę kopiowania kompilatorów w języku COBOL, w żądane miejsce opracowanego programu włączane są elementy biblioteki w postaci kodu źródłowego. Obsługa biblioteki zdań źródłowych i sposób korzystania z niej jest bardzo prosty i nie wymaga od programisty przy pisaniu programu żadnych specjal-

nych zabiegów. Dostępne implementacje kompilatorów w języku COBOL umożliwiają również stosowanie specjalnych technik programowania, jak tablice decyzyjne czy też programowanie systemów bazy danych.

Na maszynach serii RIAD język programowania COBOL jest dostępny w wersji tzw. „amerykańskiej" COBOL-ANS (*American National Standard*). Wersja „amerykańska" języka, oprócz znanych na maszynach ODRA właściwości, ma możliwość ułatwienia opisu sprawozdań.

Firma IBM proponuje jako podstawowy język programowania wyższego rzędu język PL/1. Język ten stworzony został z myślą o serii komputerów IBM/360 i oparty na doświadczeniach języków COBOL, ALGOL i FORTRAN. Połączenie cech tych trzech języków powoduje, że jest to język użyteczny zarówno do obliczeń ekonomicznych, jak i naukowo-technicznych. Inną zaletą tego języka jest jego przydatność dla programistów zarówno początkujących, jak i dysponujących bogatym doświadczeniem.

Struktura programu źródłowego pisanego w PL/1 jest zupełnie dowolna w przeciwieństwie do sztywnej struktury programu w języku COBOL. Budowa programu ma charakter blokowy (modułowy). Rozróżnia się dwa podstawowe bloki programowe: bloki typu BEGIN i typu PROCEDURE, które mogą być samodzielnie programowane. Istnieje możliwość lokalnej deklaracji danych co daje dużą swobodę operowania w ramach bloku. Dynamiczna gospodarka pamięcią zajmowaną przez program jest kolejnym udogodnieniem tego języka programowania.

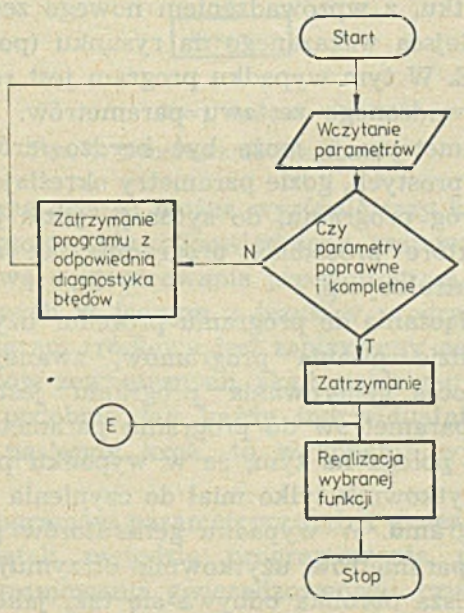
Język PL/1 jest ciągle doskonalony i rozbudowywany. Modułowa struktura języka pozwala na stosowanie wersji podstawowej wykonującej główne funkcje lub też pełnej, jaką umożliwia dostępny kompilator tego języka.

Do innych popularnych w kraju języków programowania wyższego rzędu należy zaliczyć ALGOL i FORTRAN, które ze względu na dostępny aparat instrukcji częściej wykorzystywane są do wykonania obliczeń inżynierskо-technicznych niż przy programowaniu systemów informatycznych.

Coraz szersze zastosowanie zdobywają specjalizowane języki programowania, jak języki symulacyjne, które wykorzystywane są przede wszystkim do wykonywania specyficznych algorytmów.

Z innych języków programowania wyższego rzędu warto wymienić język programowania BASIC, który jest szczególnie przydatny przy konwersacyjnym trybie współpracy z komputerem, oraz język PASCAL, uwzględniający najnowsze trendy w rozwoju metod programowania.

Języki programowania wraz z szerokimi możliwościami systemów operacyjnych są podstawą, na której rozwijały się różnorodne metody i techniki programowania. Naturalnym kierunkiem rozwoju metod programowania indywidualnego była parametryzacja programu. Założeniem parametryzacji było opracowanie takiego programu, który mógłby realizować kilka zbliżonych funkcji na podstawie danych wprowadzonych bezpośrednio przed użytkowym wykonaniem. Liczba elementów, które można modyfikować w tak przygotowanym programie decyduje o stopniu parametryzacji. Metoda sterowania ostateczną strukturą programu za pomocą parametrów, rozwijała się od prostych form przekazywania danych sterujących do bardziej złożonych, polegających na generowaniu kodu źródłowego programu.



Rys. 6. Struktura programu parametrycznego

Ogólną strukturę programu parametrycznego przedstawia rysunek 6. Program parametryczny oprócz wymaganej rozbudowy części zasadniczej, zawiera grupę instrukcji, która ma na celu wprowadzenie parametrów, analizę ich pod względem poprawności formalnej i kompletności niezbędnej dla realizacji.

Stosowane są dwa zasadnicze rozwiązania:

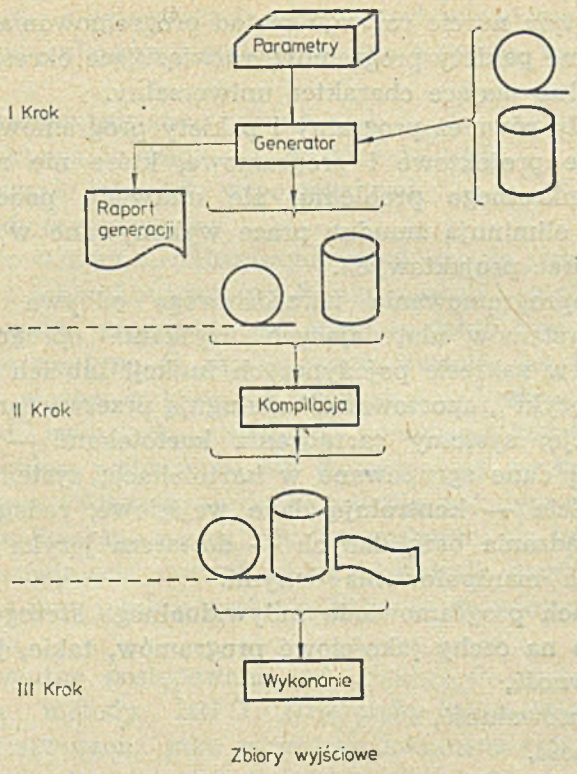
— program wprowadza cały komplet parametrów,

— parametry obowiązkowe są wprowadzane każdorazowo, a parametry opcjonalne mogą być pominięte, co spowoduje przyjęcie wartości domniemanych, określonych w toku pisania programu.

Każdorazowe wykrycie błędów formalnych w parametrach powoduje zatrzymanie programu z podaniem diagnostyki błędów. Po poprawieniu błędnych parametrów, należy ponownie uruchomić program. Gdy wszystkie parametry obowiązkowe zostaną dostarczone do programu, istnieje możliwość wyprowadzenia otrzymanej wersji programu do biblioteki prywatnej użytkownika. W kolejnych wykonaniach programu można go zrealizować od początku, z wprowadzeniem nowego zestawu parametrów lub od miejsca wskazanego na rysunku (por. rys. 6) jako punkt wejścia E. W tym wypadku program jest realizowany dla wcześniej wprowadzonego zestawu parametrów.

Stopień parametryzacji może być bardzo zróżnicowany, od sytuacji bardzo prostych, gdzie parametry określają wybór jednej z możliwych dróg programu, do sytuacji, gdzie parametrycznie określamy niektóre procedury programu, jak algorytm obliczania cyfry kontrolnej itp.

Możliwość dołączania do programu procedur użytkownika dała początek metodzie pisania programów, zwanej generowanie programów. Proces generowania programu jest podobny do wprowadzania parametrów do programu parametrycznego. Różnica zasadnicza polega na tym, że w wypadku programu parametrycznego użytkownik tylko miał do czynienia z postacią wykonywalną programu. W wypadku generatorów programów, po wprowadzeniu parametrów użytkownik otrzymuje postać źródłową, której dalsza obróbka odbywa się tak, jakby sam napisał wygenerowany program. Przebieg procesu generowania programu przedstawia rysunek 7.



Rys. 7. Generowanie programu

W procesie generowania można wyróżnić trzy kroki. Pierwszy polega na przygotowaniu odpowiedniego zestawu parametrów, które są podstawą wygenerowania przez program generator ciągu zdań źródłowych w jednym z języków programowania. Wygenerowany program źródłowy jest zapisywany na jednym z dostępnych nośników zewnętrznych, skąd poddawany jest procesowi kompilacji, podobnie jak każdy indywidualnie pisany program. Dopiero następny krok, to wykonanie wygenerowanego programu.

Tworzenie programów parametrycznych i generatorów programów dało początek metodzie programowania, polegającej na tworzeniu oprogramowania generalizowanego, czyli takiej grupy programów lub pojedynczych programów, które mogą być wielokrotnie wykorzystane przy rozwiązywaniu różnorodnych pro-

blemów. W tym nurcie rozwoju metod programowania powstały specjalizowane pakiety programów, rozwiązujące określoną klasę problemów lub mające charakter uniwersalny.

Powstawały również programy i pakiety programów wspomagające prace projektowe i programowe, które nie rozwiązują żadnego konkretnego problemu, ale ułatwiają podejmowanie decyzji lub eliminują żmudne prace wykonywane w tradycyjnym toku prac projektowych.

Rozwój oprogramowania narzędziowego odbywa się przez tworzenie systemów ułatwiających uzyskanie oprogramowania użytkowego w zakresie pojedynczych funkcji lub ich grup. Dla przykładu języki „raportowania" obsługują przetwarzanie, dostęp i komunikację; systemy zarządzania kartotekami — zakładają i aktualizują dane zgrupowane w kartotekach; systemy obsługi wejścia-wyjścia — kontrolują dane wejściowe, redagują dane; system zarządzania bazą danych — dostarcza języka opisu danych, języka manipulowania danymi.

W metodach programowania indywidualnego szczególną uwagę zwracano na cechy jakościowe programów, takie, jak:

— poprawność,
— przystosowalność,
— odporność,
— stabilność,
— optymalność.

Przemysł oprogramowania, a więc producenci sprzętu oraz wyspecjalizowani, niezależnie od nich działający, producenci oprogramowania, wypracowali liczne metody i sposoby programowania oraz specjalistyczne pomoce służące podnoszeniu jakości wymienionych cech. Zadaniem wypracowanych metod i narzędzi programowania jest zwiększenie efektywności procesu projektowania, co osiąga się, z jednej strony, przez podniesienie wydajności pracy programisty, z drugiej zaś — przez nadanie programowi większej przejrzystości wraz z dostarczeniem dokumentacji projektowej, która umożliwia łatwe i szybkie wniknięcie w rozwiązywany problem. Proponowane metody wpływają bezpośrednio na sprawność i wydajność procesu projektowania, natomiast pomoce mają na celu przede wszystkim ułatwienie i usprawnienie pracy projektowania.

Jak wynika z najnowszych prac, do technologii programowania wprowadzono ulepszone metody programowania, na które składają się:

— programowanie strukturalne,
— projektowanie programu „od góry do dołu",
— stosowanie biblioteki wspomagającej,
— metody organizacyjne.

Programowaniem strukturalnym zajmiemy się szczegółowo dalej, a teraz scharakteryzujemy krótko pozostałe metody.

Projektowanie programu „od góry do dołu" (*top-down*), charakteryzuje się tym, że zarówno opracowanie, jak i testowanie programu, można wykonywać równolegle. Postępując konsekwentnie od „ogółu" do „szczegółu" budowany jest projekt programu. Problem poddawany jest dekompozycji na kolejnych poziomach szczegółowości aż do określenia pojedynczych funkcji, które są realizowane na najniższym poziomie. Na kolejnych poziomach rozwinięcia rozważane są struktury danych, sytuacje decyzyjne i inne punkty węzłowe, które są decydujące dla całego rozwiązania.

Konsekwentne postępowanie od „ogółu" do „szczegółu" jest założeniem metody HIPO (*Hierarchy-Input-Process-Output*). Metoda ta stworzona jako narzędzie dokumentacyjne, wykorzystywana jest obecnie przy projektowaniu systemów, integrując proces projektowania i programowania systemu. HIPO dostarcza dwa rodzaje wykresów:

— wykres hierarchii, który ukazuje rozkład funkcji „od góry do dołu",
— wykres wejście-proces-wyjście, który dla każdej funkcji określa wejście, algorytm i wyjście; używana w metodzie technika graficzna umożliwia rozwijanie problemu „od góry do dołu" w zdyscyplinowany i przejrzysty sposób.

Biblioteka pomocnicza składa się z czterech elementów:

— biblioteki komputerowej dostępnej w procesie projektowania,
— biblioteki dokumentacji i opisów narzędzi programowych,
— procedur komputerowych,
— procedur postępowania.

Elementy te wspomagają proces projektowania systemu w ca-

łym cyklu realizacji. Wykorzystanie procedur postępowania sprawdzonych przy tworzeniu innych systemów umożliwia sprawne wykonanie procesu. Wykorzystanie procedur komputerowych, programów uniwersalnego zastosowania przez wmontowanie ich w projektowany system, eliminuje część prac programowych. Wykorzystanie metod i narzędzi programowych powstałych na potrzeby innych systemów jest możliwe dzięki odpowiednio prowadzonej bibliotece komputerowej oraz bibliotece dokumentacji. W ramach prac nad bieżąco wykonywanym projektem biblioteki te są rozszerzane o nowe elementy, które mogą być wykorzystywane w dalszych fazach.

Stosowanie biblioteki wspomagającej dostarcza zespołowi projektującemu:

— stale aktualnej reprezentacji wykonanych produktów,

— zdyscyplinowanego sposobu zarządzania całością prac wykonywanych przy realizacji projektu.

W dążeniu do zwiększenia wydajności procesu projektowania, oprócz stosowania programowania strukturalnego, projektowania „od góry do dołu" czy biblioteki wspomagającej, zwrócono także uwagę na metody natury organizacyjnej. Metody te mają na celu ścisłe rozdzielenie kompetencji członków zespołów. Szczególnie jest to widoczne w najbardziej popularnej metodzie proponowanej przez firmę IBM, a znanej pod nazwą „zespołu głównego programisty".

Rozwiązania organizacyjne tej metody, a zwłaszcza zróżnicowanie i rozłożenie odpowiedzialności, nastawione są w równej mierze na lepsze „dowodzenie" zespołem projektowym, jak i na wyegzekwowanie większej wydajności pracy poszczególnych jej członków. W zespole wyróżnia się trzy zasadnicze funkcje, głównego programisty, jego zastępcy oraz bibliotekarza zespołu.

Zadaniem głównego programisty, kierującego pracą całego zespołu, jest zaprojektowanie całości programu, podzielenie go na odpowiednie części z przydzieleniem prac poszczególnym członkom zespołu, a następnie koordynowanie i kontrolowanie takiego wykonania poszczególnych części, aby złożony z nich program spełniał założone wymagania.

Zastępca w równym stopniu znający wszystkie niuanse projektu współdecyduje w najważniejszych rozstrzygnięciach pro-

jektowych i jest w stanie w każdej chwili przejąć kierownictwo zespołu.

Bibliotekarz zespołu zajmuje się wszelkimi „papierkowymi" sprawami zespołu, takimi jak ewidencja postępu prac nad projektem, kompletowanie odpowiedniej dokumentacji, czuwanie nad biblioteką wspomagającą projekt.

Taka organizacja prac w połączeniu z ulepszonymi metodami programowania, umożliwia znaczne zwiększenie wydajności poszczególnych programistów, przyspieszenie prac nad projektem oraz podniesienie jakości produktu końcowego.

Z przedstawionych metod projektowania i programowania systemu widać wyraźnie, że problem odpowiedniej dekompozycji problemu na podproblemy jest myślą przewodnią większości z nich. Od sposobu wyodrębnienia elementów, ich rozmiaru oraz zachowania wzajemnych związków, w dużym stopniu zależy sprawność realizacji procesu projektowania, jak i jakość powstałego w wyniku końcowym projektu, będącego zbiorem niezależnie wykonywanych elementów.

II. Programowanie modularne

1. Podział programu monolitycznego

Rozwiązywanie skomplikowanych problemów powoduje, że programy mają coraz bardziej złożoną strukturę oraz zwiększa się liczba instrukcji źródłowych. Skonstruowanie programu monolitycznego rozwiązującego problem stało się coraz trudniejsze. Problemem stawało się nie tylko napisanie coraz większej liczby instrukcji programu, ale również umieszczenie całego programu w pamięci operacyjnej. Nie trudności techniczne, chociaż i one były dość istotne, ale dążenie do optymalnego wykorzystania pamięci operacyjnej skłaniało programistów do rezygnacji z monolitycznej postaci programu, w sytuacji gdy w danej chwili wykonywania programu tylko niewielka jego część jest aktywna.

Wydzielenie autonomicznych części składających się na cały program dało początek różnym metodom podziału programu. Najczęściej spotykaną metodą wydzielenia części autonomicznych w programie jest dość powszechnie przyjęta technika podprogramów, która może być stosowana z wykorzystaniem prawie wszystkich języków programowania. Technika ta jest wykorzystywana najczęściej w sytuacji, gdy niezbyt długa sekwencja instrukcji jest wielokrotnie powtarzana w programie i jednorazowe jej zakodowanie i wywoływanie oszczędza miejsce w pamięci operacyjnej oraz umożliwia bardziej przejrzystą konstrukcję programu. Liczne ograniczenia kompilatorów języków programowania nie pozwalają na powszechne stosowanie tej techniki dla podziału programu na większe bloki, które charakteryzowałyby się dużo większą autonomią niż podprogram.

Korzystając z niektórych języków programowania można prze-

prowadzić podział programu na stosunkowo duże bloki charakteryzujące się dużą autonomią. W języku programowania PLAN można np. przyjąć podział programu na segmenty oddzielnie kodowane i kompilowane. W języku COBOL natomiast istnieje podział działu procedur na sekcje, które mogą być wykonywane w wymaganej kolejności bez potrzeby przechowywania sekcji nieaktywnych w pamięci operacyjnej. W programie wyróżniona jest sekcja główna oraz pozostałe traktowane jako nakładki, które są przechowywane w pamięci zewnętrznej i wprowadzane czasowo do pamięci operacyjnej na jeden wspólny obszar tylko na czas wykonania. Sterowanie kolejnością wywoływania i wykonywania sekcji będących nakładkami, zakodowane jest w sekcji głównej. Technika nakładkowa umożliwia nie tylko podział programu źródłowego, ale również optymalizuje wykorzystanie obszaru pamięci operacyjnej wykorzystywanej przez program. Język programowania PL/1 daje jeszcze większe możliwości podziału programu na autonomicznie kodowane i wykonywane moduły programowe.

Podział programu monolitycznego na mniejsze elementy autonomiczne nie może dotyczyć wyłącznie części proceduralnej programu, ale wymaga odpowiedniej modyfikacji danych wykorzystywanych przez program. Wydzielenie autonomicznych bloków instrukcji jest czynnością stosunkowo prostą, ale wszystkie wyodrębnione bloki operują na wspólnych danych. W danych wykorzystywanych przez program można wyróżnić część danych wspólnych, wykorzystywanych przez różne części podzielonego programu oraz dane wykorzystywane w ramach tylko jednej części programu. Wyróżniamy więc dane *globalne* oraz dane *lokalne*. Dane globalne muszą być dostępne w każdej z części programu i są obszarami dla komunikowania się między wydzielonymi częściami programu. Dane lokalne są obszarami wykorzystywanymi tylko i wyłącznie przez jedną z części programu i nie muszą być dostępne w innych częściach.

Języki programowania umożliwiające podział programu, dostarczają instrumentów wyróżniających dane o znaczeniu lokalnym i globalnym. Wyjątkiem jest język programowania COBOL, gdzie dział danych jest jeden wspólny dla całego programu, niezależnie od podziału działu procedur.

Odejście od techniki monolitycznego programowania na rzecz podziału programu podniosło rangę fazy planowania programu. Określenie obszarów wspólnych, wydzielenie funkcji realizowanych przez autonomiczne części proceduralne, mają decydujący wpływ na dalszy tok prac programowych. Podział programu na autonomiczne części zmusza do zgłębienia rozwiązywanego problemu już w fazie planowania programu, gdyż dokonany podział musi szczegółowo wynikać ze struktury problemu. Wydzielenie autonomicznych części pozwala na oddzielne ich rozwiązywanie, co wydatnie wpływa na skrócenie czasu potrzebnego na opracowanie całego programu, gdyż rozwiązania poszczególnych części mogą być wykonywane współbieżnie przez różnych programistów. Rozbicie problemu na podproblemy wpływa na zmniejszenie stopnia trudności rozwiązania. Każda autonomicznie kodowana część może być samodzielnie kodowana, testowana i dokumentowana. Wytestowane części wymagają sprawdzenia ich współdziałania przed przekazaniem całego programu do eksploatacji.

Do sprawnego wykonania procesu dekompozycji jest proponowane systemowe spojrzenie na problem z góry w dół, realizowane w pięciu krokach postępowania, a mianowicie:

1) określenie struktury problemu w kilku (od 3 do 10) głównych procesach,

2) identyfikacja w strukturze problemu głównych strumieni wejściowych danych oraz głównych strumieni wyjściowych danych,

3) określenie punktu w strukturze problemu, gdzie zanika strumień wejściowy i punktu drugiego, gdzie po raz pierwszy pokazuje się strumień wyjściowy,

4) te dwa punkty dzielą problem na trzy sekcje: opiszmy te trzy sekcje i trzy realizowane przez nie funkcje, definiując moduł dla każdej z nich; są to źródłowe moduły do dalszych rozważań,

5) funkcje każdego z tych modułów określamy jako podproblem i powtarzamy postępowanie.

Proces ten należy kontynuować tak długo aż problem wejściowy do dekompozycji będzie odpowiednio mały.

Rozwój metod podziału programu dał początek metodzie programowania zwanej *programowaniem modularnym.*

2. Podstawowe pojęcia programowania modularnego

Przed przystąpieniem do rozwinięcia programu w strukturę modularną wymagane jest precyzyjne określenie efektu końcowego, a tym samym wymagana jest definicja modułów.

Podstawą programowania modularnego jest fakt, że problem rozwiązywany jest zbyt złożony, aby mógł być uchwycony w całości. Problem musi być podzielony na podproblemy, które mogą być dzielone coraz dalej, dopóki struktura problemu nie pozwala na dalszy podział lub zostanie osiągnięta wymagana wielkość modułu. Każdy z wydzielonych podproblemów staje się modułem programowym. Taki podział jest jedną z zasad programowania modularnego, przy założeniu tworzenia modułu dla każdego wydzielonego podproblemu.

Druga zasada polega na tym, że podział ten nie może być przypadkowy, ale powinien wynikać z rozważnej i szczegółowej analizy struktury logicznej problemu. Analiza struktury logicznej problemu, poprzedzająca prace nad programem, ma zasadnicze znaczenie dla stworzenia programu modułowego, w przeciwieństwie do programu monolitycznego lub podzielonego przypadkowo.

Fizyczna wielkość modułu nie jest stała, ale jeżeli wszystkie fazy programowania (planowanie, kodowanie i testowanie) mają być efektywnie kontrolowane, to moduły nie mogą być zbyt duże. Istotne jest również, aby podproblem, który ma być rozwiązany przez moduł mógł być ogarnięty przez programistę w całości, co również wymaga małych modułów. Górna granica wielkości modułu bywa określana w sposób różny. Wielkość modułu winna być taka, aby można go było zakodować i sprawdzić w ciągu jednego dnia [1]. Inni autorzy proponują, aby moduł był odpowiednikiem od 50 do 150 rozkazów [2].

Inną charakterystyczną cechą modułu programowego jest potrzeba niezależnego testowania, co powoduje, że moduł musi być

[1] Por. *Modular Programming techniques T.P. 5091*, ICL 1971.

[2] Por. J. Twichans, J. Heinemann, *Software Engineering—Sicherung und Kontroll der Software Qualität*, Data Exchange, III—IV, 1978; A. Boos, W. Trommer, *Modularprogramierung und rechnergestützte Programdokumentation in COBOL*, „Online Zfd" 1975, nr 10; R. Rackstein, H. Schmitz, *Strukturierte Programmierung und ihre graphische Interpretation*, „Online Zfd" 1975, nr 10.

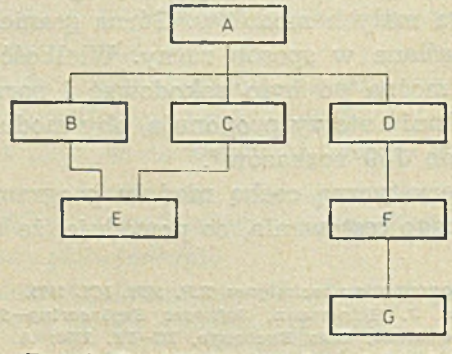
jednostką, która może być samodzielnie wykonana. Warunki takie w różnych językach programowania spełniają podprogramy czy procedury zewnętrzne. Samodzielne wykonanie modułu jest cechą potrzebną dla sprawdzenia poprawności jego działania, ale moduł występuje w większej całości, z którą musi mieć możliwość komunikowania się. Stąd wynika potrzeba wprowadzenia standardowego pośrednictwa (*interface*) międzymodułowego.

Zbierając wszystkie wymienione cechy można się pokusić o sformułowanie definicji modułu programowego.

Moduł jest elementem programu, który rozwiązuje jeden z podproblemów tego programu realizując jednoznaczny, w pełni zdefiniowany zestaw operacji, na określonych obszarach danych. Zakres podproblemu jest taki, że programista może łatwo uwzględnić całość wewnętrznego przetwarzania, wymaganego w tym samym czasie. Komunikacja modułu z otoczeniem odbywa się za pomocą standardowego pośrednictwa, które musi być kompatybilne z narzędziami programowymi wykorzystywanymi dla jego testowania.

a. Hierarchia modułów

Rozbicie problemu na podproblemy i wynikający stąd podział programu na moduły stwarza potrzebę przedstawienia wzajemnych relacji między modułami. Wykres hierarchii modułów jest metodą graficznej prezentacji podziału programu na moduły. Przykład wykresu hierarchii modułów przedstawia rysunek 8.



Rys. 8. Przykładowy wykres hierarchii modułów

Każda klatka wykresu reprezentuje moduł i literowe oznaczenie identyfikujące go. Linie łączące klatki przedstawiają komunikację między modułami. Linie wchodzące od góry do klatki, łączą go ze wszystkimi modułami, z których następuje wejście do modułu. Linie wychodzące z dołu klatki łączą moduł ze wszystkimi modułami, do których następuje wyjście z tego modułu. Na rysunku 8, moduł A przekazuje sterowanie do każdego z modułów B, C i D. Każdy moduł, z którego następuje wejście do danego modułu, jest określany jako moduł sterujący, a moduły, do których następuje wejście z danego modułu, są określane jako moduły podporządkowane. Na przedstawionym rysunku modułem sterującym dla modułu D jest moduł A, natomiast modułem podporządkowanym jest moduł F.

Na wykresie, moduł sterujący występuje fizycznie wyżej niż moduły, którymi steruje, co oznacza, że moduł sterujący wykorzystuje moduł podporządkowany w celu realizacji części swojej funkcji. Moduł podporządkowany rozwiązuje jeden z podproblemów, na jakie dzieli się problem rozwiązywany przez moduł sterujący. Poza tym miejsce na wykresie nie ma znaczenia, a w szczególności nie może dać żadnych wskazań co do kolejności wykonania poszczególnych modułów.

Wszystkie moduły, do których następuje bezpośrednie lub pośrednie wejście z danego modułu są częścią hierarchii tego modułu. W podanym na rysunku 8 przykładzie, hierarchią modułu F jest on sam, natomiast moduł E jest częścią hierarchii modułu B i C. Hierarchią modułu A jest cały program przedstawiony na rysunku.

Wykres hierarchii modułu umożliwia przejrzyste pokazanie powiązań między modułami, przedstawiając każdy moduł za pomocą odpowiedniego symbolu w powiązaniu z jego modułami sterującymi oraz jego modułami podporządkowanymi. Wykres nie zawiera żadnych informacji o decyzjach występujących w programie, ani o kolejności przetwarzania. Widać stąd, że w dokumentacji programu, wykres hierarchii modułu spełnia stosunkowo ograniczoną rolę, ale umożliwia przejrzystą prezentację podziału na moduły oraz może być użyty do wskazania przepływu informacji między modułami.

Główna funkcja wykresu hierarchii modułu polega na tym, że

służy on jako narzędzie w fazie projektowania programu. W fazie tej służy on jako środek podsumowania aktualnej struktury problemu, podczas wykonywania analizy logicznej oraz jako środek analizowania przepływu informacji, w którym to wypadku może wpłynąć na wybór struktury fizycznej.

Jako sposób komunikowania się między projektantem programu a programistą wykres hierarchii modułu jest związany przede wszystkim z organizacją programu jako całości, a nie logiką danego modułu. Celem wykresu jest pokazanie programiście, gdzie dany moduł jest umieszczony w ogólnej strukturze programu oraz przekazanie informacji o przepływie informacji między danym modułem a pozostałymi modułami.

Wykres hierarchii modułu spełnia ważną funkcję w udokumentowaniu programu, a szczególnie umożliwia szybkie i pełne rozpoznanie tych części programu, które mają być poprawione lub rozszerzone w toku konserwacji programu.

b. Pośrednictwo międzymodułowe

Moduł może komunikować się jedynie bezpośrednio ze swoim modułem sterującym oraz ze swoimi modułami podporządkowanymi. Na przykład moduł C (por. rys. 8) może komunikować się bezpośrednio jedynie z modułami A i E. Jeżeli moduł musi przekazać informacje do jakiegoś innego modułu, to komunikacja musi odbywać się przez ciągłą sekwencję powiązanych modułów, które łączą oba moduły. Z tego wynika, że komunikacja między modułami B i G, odbywa się przez moduł F.

Ogólnie biorąc, jeżeli dana informacja jest wymagana przez więcej niż jeden moduł, to musi być ona zdefiniowana i zamieszczona w module sterującym hierarchii, do której należą wszystkie moduły korzystające z tej informacji. Przekazywane pola danych winny mieć ustalone jednolite atrybuty, takie jak nazwa, długość, format.

3. Opracowanie programu modularnego

Projektowanie programu jest procesem analizowania rozwiązywanego problemu, a w szczególności zbiorów, które program ma

przetwarzać oraz decyzji, jakie należy podjąć w celu rozwiązania przedstawionego problemu. Struktura decyzji umożliwia rozbicie złożonego problemu na podproblemy. Na kolejnych poziomach podejmowania decyzji jest osiągane bardziej szczegółowe przedstawienie struktury programu, aż do uzyskania końcowego i dostatecznie szczegółowego rozwiązania całego problemu, które umożliwia szybkie i łatwe wykonanie kodowania.

Wcześniej zaproponowaliśmy jeden ze sposobów dekompozycji problemu na podproblemy. Dekompozycja ta nie może być jednak mechaniczna; winno to być działanie celowe, w wyniku którego moduły będą się charakteryzować prostotą i niezależnością. Niezależność modułu jest niezwykle istotną cechą. Z niezależności poszczególnych modułów wynikają z kolei następujące cechy:

— zredukowanie złożoności problemu, która jest skutkiem zależności poszczególnych modułów,

— usprawnienie procesu programowania, gdyż programiści mogą niezależnie i oddzielnie wykonywać poszczególne części programu; rozwiązanie problemu jest łatwiejsze, gdyż można go podzielić na niezależne części; rozwiązanie może być realizowane współbieżnie,

— poprawianie i modyfikowanie poszczególnych części może być realizowane przez różne osoby,

— organizacja prac nad problemem powinna być dokładna, gdyż potrzebny jest dodatkowy czas na kompletację modułów,

— czas potrzebny na zdefiniowanie problemu programiście jest minimalizowany, gdyż programista powinien zrozumieć tylko tę część programu, którą wykonuje,

— jakość programu będzie wynikiem wspólnego wysiłku; moduły mogą mieć charakter uniwersalny i mogą być biblioteką wspomagającą, wykorzystywaną przy rozwiązywaniu innych problemów.

Wysoką niezależność modułów można osiągnąć za pomocą dwóch metod postępowania:

1) maksymalizacji powiązań wewnątrzmodułowych,

2) minimalizacji powiązań zewnętrznych modułów.

Innymi słowami, chociaż program jako całość składa się z dużej liczby powiązań i relacji, to strategia podziału programu

na moduły ma na celu maksymalizację powiązań wewnątrz modułów oraz minimalizację połączeń międzymodułowych.

Niezależność modułu można określać za pomocą pewnych miar, takich jak moc modułu, połączenia międzymodułowe i inne miary [3].

Moc modułu jest miarą relacji wewnątrzmodułowych. Można wyróżnić pięć kategorii mocy, które wymieniamy w kolejności mocy wzrastającej:

1) nieistotna moc — moduł, w którym występują nieznaczące relacje między jego elementami,

2) logiczna moc — moduł, który w czasie każdego wywołania, spełnia funkcje zewnętrzne względem funkcji realizowanych przez moduł,

3) klasyczna moc — moduł, który spełnia dwie lub więcej funkcji; funkcje te są określone warunkami w procedurze głównej,

4) komunikacyjna moc — moduł, którego moc proceduralna elementów jest uzależniona od wykorzystywanych przez moduł danych,

5) funkcyjna moc — moduł spełnia jedną specyficzną funkcję.

Połączenie międzymodułowe jest natomiast miarą relacji między modułami. Oprócz mocy modułu jest to druga główna miara niezależności modułu. Można wyróżnić cztery kategorie połączeń międzymodułowych, które wymieniamy w porządku połączeń malejących (od najgorszych do najlepszych):

1) zadowalające połączenie — dwa moduły są zadowalająco połączone, jeżeli jeden odwołuje się bezpośrednio do drugiego,

2) zewnętrzne połączenie — grupa modułów jest zewnętrznie połączona, jeżeli odwołują się one do pewnych danych globalnych,

3) nadzorowane połączenie — dwa moduły są nadzorowanie połączone, jeżeli jeden jawnie kontroluje funkcje drugiego modułu,

4) połączenie danymi — dwa moduły są połączone danymi, jeżeli jeden moduł wywołuje drugi i wszystkie wejścia do niego

[3] Na temat miar niezależności modułu por. raport *IBM IPTO Support Group World Trade Systems Center*. Postbus 60 Zoetermeer, Netherlands, 079-253850; G. J. Myers, *Projektowanie niezawodnego oprogramowania*, Warszawa 1980.

i wyjścia z niego, dla modułu wywołanego są określane w postaci parametrów.

Kiedy moduły mają różne formy połączeń, to klasyfikuje się je według najgorszej formy połączenia. Dla przykładu, jeżeli dwa moduły są połączone zewnętrznie oraz są połączone danymi, to definiujemy je jako połączone zewnętrznie.

Inne miary niezależności

W celu określenia stopnia niezależności modułów wykorzystywane są również inne następujące miary.

1. Rozmiar modułu. Rozmiar modułu wpływa na niezależność modułu, przejrzystość programu i prostotę jego testowania. W zasadzie moduły winny zawierać od 10 do 100 zdań języka programowania wyższego rzędu, ale praktycznie wskazane jest określanie modułów, których rozmiar zawiera się między 40 a 60 zdań źródłowych.

2. Tablice informacji modułu. Dla wszystkich modułów należy opracować tablice informacji. W tablicy informacji można sekwencyjnie zapamiętać kilka różnych danych, natomiast między modułami przekazywany jest pojedynczy wskaźnik (lub adres), co zmniejsza wyraźnie liczbę kontaktów między modułami.

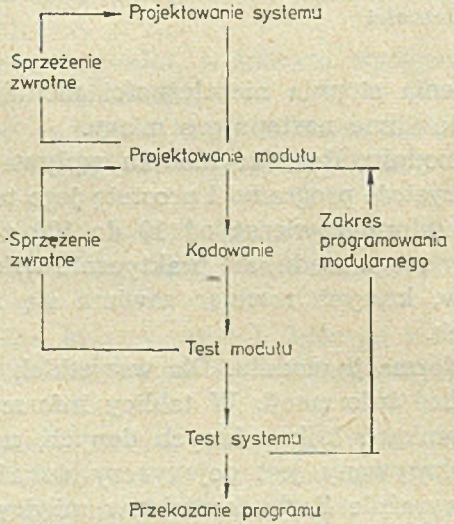
3. Struktury decyzyjne. Moduły i stany decyzyjne programu są ułożone w tablice zmiany stanu. Tablice te zawierają informacje o sytuacjach, gdy czynność wymagana przez program dla danej informacji jest zdeterminowana przez sekwencję poprzedniego wejścia (tzn. procedura jest uzależniona od wejścia).

4. Minimalizacja dostępu danych. W module winny być tylko te dane, które są niezbędne dla realizacji podstawowej funkcji modułu.

W wykonywanym procesie dekompozycji problemu należy wiele uwagi poświęcić niezależności modułów, ponieważ w dalszym procesie programowania modularnego każdy wydzielony moduł tworzy pewną całość i jest oddzielnie kodowany i testowany. Niezależność modułów pozwala na równoległe wykonywanie wielu prac.

Po zakończeniu procesu dekompozycji określona zostaje szczegółowo struktura logiczna programu. Wydzielone w jej toku pod-

programy nie muszą okazać się najlepszymi jednostkami do zakodowania jako moduły. Wymagana jest weryfikacja, której celem jest takie połączenie jednostek logicznych, aby tworzyły fizyczny moduł. W wyniku tych prac można wykreślić hierarchię modułów dla rozważanego problemu, co pozwala na rozpoczęcie prac nad poszczególnymi modułami.



Rys. 9. Tworzenie programu modularnego

Proces tworzenia programu modularnego przedstawia rysunek 9. Kodowanie wydzielonych modułów programowych może być wykonywane niezależnie, z zachowaniem warunków narzuconych przez pośrednictwo międzymodułowe. Oprócz możliwości współbieżnego kodowania modułów przez różnych programistów istnieje możliwość kodowania w różnych językach programowania, gdyż moduły łączone są w program w postaci wynikowej, która nie jest zależna od języka źródłowego programowania. Współbieżne kodowanie modułów umożliwia prawidłową organizację prac nad programem oraz skrócenie łącznego czasu wykonania programu.

Wielkość modułu umożliwia sprawne wykonanie kodowania oraz rozpoczęcie procesu testowania modułu, gdyż każdy moduł

powinien być jednostką programową, którą można wykonywać samodzielnie.

Testowanie modułu jest wykonywane podobnie, jak każdego innego programu. Ze względu na wielkość modułu oraz jego umiejscowienie w hierarchii i uwarunkowania pośrednictwa międzymodułowego przebieg testowania pojedynczego modułu nie powinien sprawiać zbyt wielkich trudności. Po wprowadzeniu niezbędnych poprawek i uzupełnień moduł w postaci gotowej do eksploatacji może być wmontowany w odpowiednie miejsce w hierarchii i testowany może być cały program, w skład którego wchodzi przygotowywany moduł. Testowanie programu ma na celu przede wszystkim sprawdzenie współdziałania modułów, czyli sprawdzana jest nie poprawność logiczna pojedynczych modułów, ale sposób ich konstrukcji w program.

Testowanie programu jako całości jest w pewnym sensie niezależne od testowania poszczególnych modułów wchodzących w jego skład, stąd możliwość równoległego testowania programu oraz poszczególnych modułów. Realizacja poszczególnych modułów będzie najczęściej różna w czasie z wielu przyczyn, stąd też testując program możemy w jego hierarchii umieszczać moduły już wytworzone lub symulować istnienie jeszcze nie wykonanych. Ze względu na sprawdzanie połączeń logicznych między modułami, bez wnikania w strukturę wewnętrzną każdego modułu, można je traktować jak swego rodzaju „czarne skrzynki". W praktyce symulowania modułów stosowane są dwa rozwiązania. Jedno polega na stosowaniu modułu ślepego[4], a drugie na budowie modułu zastępczego, który dostarcza do programu takie dane, jakie są przewidywane dla modułu rzeczywistego.

Wiele firm produkujących oprogramowanie oferuje pakiety programów przeznaczone do testowania modułów. Ośrodki obliczeniowe mają często własne pakiety standardowe testujące lub też pakiety testujące o pewnym podzbiorze możliwości, wykonane na potrzeby określonego systemu.

---

[4] Moduł ślepy to taki, który ma jedynie punkt wejścia i punkt wyjścia; nie dokonuje żadnego przetwarzania, a jedynie przekazuje sterowanie do modułu wywołującego z chwilą, gdy tylko zostanie sam wywołany.

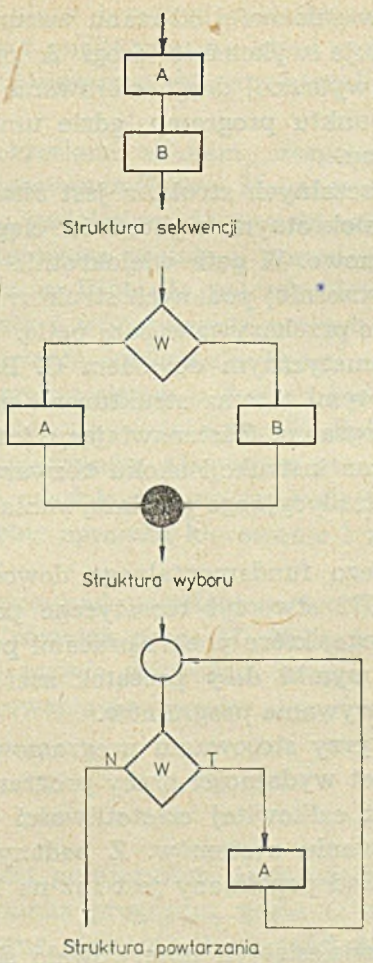
III. Podejście strukturalne do programowania

1. Rozwój programowania strukturalnego

Koncepcja ogólna podejścia strukturalnego zrodziła się w firmie IBM i wywodzi się ze struktury hierarchicznej, nałożonej na proces projektowania i programowania systemów informatycznych. Podstawowym problemem do rozwiązania w procesie prac projektowych i programowych są sprawy przekazywania sterowania między poszczególnymi obiektami systemu. Sterowanie wadliwie zorganizowane kryje w sobie potencjalne źródło powstawania błędów i niesprawności funkcjonujących systemów. W koncepcji podejścia strukturalnego przyjęto jako obowiązujące przekazywanie sterowania z góry na dół (*top-down*), co spowodowało wyraźną eliminację błędów wynikających z wadliwego przekazywania sterowania. Równocześnie hierarchiczna struktura z jednokierunkowym sposobem poruszania się, z góry na dół, zwiększyła przejrzystość całego problemu, przez co powstała możliwość eliminacji błędów wynikających z nieczytelności i zagmatwania obrazu całości rozważanego problemu.

Podstawy teoretyczne programowania strukturalnego dali C. Böhm i G. Jacopini, którzy w 1966 r. przeprowadzili matematyczny dowód, że wszystkie programy lub ich schematy blokowe dają się doprowadzić do ekwiwalentnej postaci, w której mogą być reprezentowane tylko przez trzy podstawowe struktury [1]. Na rysunku 10 pokazane są te struktury: sekwencyjna, wyboru, powtarzania zilustrowane w postaci znanej z notacji schematu blokowego.

[1] Por. C. Böhm, G. Jacopini, *Flow Diagrams, Turing Machiness and Languages with only Two Formulation Rules*, „Communication ACM" 1966, nr 5.



Rys. 10. Podstawowe struktury

Struktura sekwencyjna reprezentuje sekwencje instrukcji, które są wykonywane w kolejności występowania w programie i które nie powodują przekazywania sterowania w inne miejsce programu. Po wykonaniu instrukcji pierwszej wykonywana jest druga, po jej wykonaniu trzecia i tak dalej z góry programu w dół.

Struktura wyboru reprezentuje zwrotnicę w programie, czyli

instrukcję, która w zależności od stanu warunku badanego przez instrukcję, powoduje wykonanie drogi A programu lub drogi B. Po wykonaniu wybranej drogi, sterowanie przekazywane jest do tego samego punktu programu, gdzie umieszczone są dalsze instrukcje programu.

Trzecią z dopuszczalnych struktur jest *struktura powtarzania*, polegająca na wielokrotnym wykonaniu ciągu instrukcji, czyli tzw. pętle programowe. W pętli wielokrotnie mogą być wykonywane dowolne z wcześniej podanych struktur w ten sposób, żeby sterowanie nie było przekazywane poza pętlą.

Zgodnie z matematycznym dowodem C. Böhma i G. Jacopiniego operowanie tymi trzema strukturami umożliwia zbudowanie dowolnego programu. Zastanawiający jest brak wśród wymienionych struktur instrukcji skoku bezwarunkowego, instrukcji kluczowej dla tradycyjnego przekazywania sterowania w programie.

Na podstawie tego fundamentalnego dowodu E. W. Dijkstra w latach 1968—1972 stworzył teoretyczne podstawy programowania strukturalnego, które poparł pracami praktycznymi. Prace praktyczne i ich wyniki dały początek szerokiej popularyzacji tej metody opracowywania programów.

Okazało się, że przy stosowaniu programowania strukturalnego osiąga się wzrost wydajności pracy programistów i odpowiednio istotny spadek całkowitej częstotliwości występowania błędów w programowaniu systemów. Z badanych źródeł wynika, że średnio jeden błąd popełniany jest raz na 10 000 wierszy programu [2].

W latach siedemdziesiątych obserwowało się wzrost zainteresowania programowaniem strukturalnym wśród stale rosnącej liczby programistów w krajach zachodnich. Z prac prowadzonych zgodnie z zaleceniami E. W. Dijkstry, jak i różnorodnych modyfikacji dochodzą dalsze argumenty, przemawiające za stosowaniem tej metody programowania.

Nieco odmienne podejście, mające cechy programowania strukturalnego, opisał F. T. Baker w 1972 r. w IBM System Jour-

[2] Por. F. Miller, G. Lindamood, *Programming: Top-down Approach*, „Datamation" 1973, nr 12.

nal [3]. Opisany przez niego przykład zastosowania programowania strukturalnego jest publikacją, na którą powołuje się wielu autorów piszących o programowaniu strukturalnym oraz o jego zaletach.

Opisana metoda, określana mianem „zespołu głównego programisty" (*Chief Programmer Teams Management*), charakteryzuje się specyficznym podejściem do projektowania i wdrażania systemów, która zastosowana została z powodzeniem w pracach wykonanych przez firmę IBM dla potrzeb New York Times'a. Firma IBM opracowała i wdrożyła dla potrzeb tej gazety, złożony system wyszukiwania informacji. Prace wykonano za pomocą jedynie kilku programistów o dużych kwalifikacjach, i to w dość krótkim czasie. Co ważniejsze, F. Baker twierdzi, że opracowany system programów faktycznie nie zawierał błędów i działa zadowalająco od dnia wdrożenia. Podejście wówczas zastosowane łączy w sobie sprawne kierowanie i zasady pogramowania strukturalnego.

Duży wkład w prace nad powstaniem teorii programowania strukturalnego mają również prace H. Millsa [4]. W koncepcjach H. Millsa zwraca się przede wszystkim uwagę na jasność prezentacji i łatwość testowania programów.

Przykład 2

Jak istotną sprawą jest przejrzystość i jasność prezentacji może dowieść przykład wycinka programu, gdzie w tablicy T wyszukiwane są oraz zliczane elementy o wartości zero. Tablica jest zakończona elementem o wartości 999. Dodatkowo liczona jest liczba elementów tablicy. Jako język kodowania wybrano PL/1.

Przy wykonywaniu bardziej skomplikowanych zadań różnice mogą być jeszcze większe.

---

[3] Por. F. Balker, H. Mills, *Chief Programmer Teams*, „Datamation" 1973, nr 12.
[4] Por. H. Mills, *Mathematical Foundations for Structured Programming*, IBM 1971; H. Mills, *Top Down Programming in Large Systems. Debugging techniques in Large Systems*, Prentice 1971.

Postać nieprzejrzysta

```
                K = 0;
                L = 0;
                J = 1;
A: IF T(J)=999 THEN GO TO KON;
   IF T(J)=0 THEN GO TO B;
   L = L+1;
   J = J+1;
   GO TO A;
B: K = K+1;
   L = L+1;
   J = J+1;
   GO TO A;
   KON
```

Postać przejrzysta

```
K = 0;
L = 0;
J = 1;
DO WHILE (T(J) ¬=999);
L = L+1;
IF T(J)=0 THEN K=
=K+1;
J = J+1;
END;
```

Strukturalne podejście do prac programowych jest istotnym nowum w metodyce programowania. Pisanie programów przestało być „sztuką" dostępną dla stosunkowo wąskiej grupy programistów, a stało się dobrze zorganizowanym procesem, w wyniku którego powstaje produkt, który coraz częściej jako autora ma nie pojedynczego programistę, ale zespół ludzi, których wspólny wysiłek złożył się na ostateczny wynik. Zespołowy sposób opracowywania programu zwrócił uwagę na doskonalenie metod organizatorskich prac nad programem, które mają równie decydujący wpływ na jakość programu, jak umiejętności programowania poszczególnych członków zespołu. Rozwój programowania strukturalnego koncentruje się obecnie przede wszystkim na doskonaleniu organizacji prac programowych.

2. Charakterystyka programowania strukturalnego

Cechami charakterystycznymi programowania strukturalnego, prezentowanymi w pracach E. Dijkstry, są:

1) konsekwentna hierarchiczna struktura programu (*top-down*),

2) ograniczone sterowanie przebiegiem programu, bez stosowania instrukcji skoku bezwarunkowego GO TO,

3) ograniczona dostępność danych, podział programu na moduły funkcjonalne i operowanie danymi lokalnymi w ramach modułu i globalnymi na poziomie programu,

4) centralne sterowanie programem, polegające na tym, że decyzja napotkana w programie nie powinna bezpośrednio wywoływać pewnych operacji w drugim module lub wpływać na nie,

5) jedno wejście i jedno wyjście z modułu [5].

T. Baker i H. Mills w zmodyfikowanej koncepcji programowania strukturalnego, zwanej „zespołem głównego programisty" wyróżniają następujące cechy charakterystyczne:

1) budowanie systemu oprogramowania powinno przebiegać od góry do dołu (*top-down design*),

2) wdrażanie programów także powinno przebiegać od góry do dołu, a elementy zastępne, symulujące obecność modułów jeszcze będących w opracowaniu, należałoby wprowadzać możliwie jak najwcześniej,

3) pojedyncze moduły programów powinny być jak najkrótsze, możliwie nie dłuższe niż jedna strona wydruku z maszyny, dla ułatwienia dzielenia struktury logicznej na poszczególne człony dające się łatwo testować,

4) ogólną pieczę nad opracowywaniem programów powinien mieć bardzo doświadczony i wysoko kwalifikowany główny programista, który byłby również odpowiedzialny za sprawę łączenia z sobą poszczególnych modułów i testowanie [6].

Te dwie koncepcje zawierają wszystkie najbardziej istotne elementy składające się na metodę programowania strukturalnego. Wymienieni autorzy za pierwszą, podstawową zasadę programowania strukturalnego uważają hierarchiczną strukturę programu konsekwentnie realizowaną z góry na dół, za pomocą trzech struktur: sekwencji, wyboru i powtarzania.

Struktury te posiadają w językach programowania odpowiedni aparat instrukcji, umożliwiający proste ich stosowanie. Rozważmy przykłady wymienionych struktur.

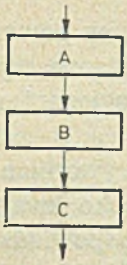
[5] Por. E. W. Dijkstra, op. cit.
[6] Por. F. Baker, H. Mills, op. cit.

Struktura sekwencji jest realizowana przez ciąg instrukcji bezwarunkowych w kolejności występowania bez przekazywania sterowania w inne miejsce programu.

```
A = 1;
POL = POL1—POL2;
CALL PROC2;
AL = SUBSTR (STR1, 1, 4).
```

Przedstawiony ciąg instrukcji języka PL/1 jest strukturą sekwencji i będzie wykonywany w kolejności występowania, instrukcja po instrukcji. Podobnie w innych językach programowania jest to ciąg instrukcji, których wykonanie nie jest uzależnione od spełnienia (lub niespełnienia) jakichkolwiek warunków, a następstwo wykonania jest wyznaczone porządkiem występowania instrukcji w tekście programu źródłowego.

W postaci graficznej, w notacji schematów blokowych, można to przedstawić jak na rysunku 11.



Rys. 11. Struktura sekwencji

Wybór

Wykonanie instrukcji sterujących uzależnione jest najczęściej od zdefiniowanych warunków. Dokonywany jest *wybór* jednego z alternatywnie opisanych ciągów operacji. Wybór odpowiedniego ciągu operacji uzależniony jest od badania czy warunek jest spełniony, czy nie spełniony. Warunek może przyjmować jedną z dwóch wartości logicznych: prawda lub fałsz. Inaczej mówiąc, warunek może być spełniony (prawda) lub nie spełniony (fałsz).

Opisany w programie warunek może być:

— warunkiem prostym, gdy operandy połączone są operatorem relacji (=, <, > itp.),

— warunkiem złożonym, gdy warunki proste połączymy operatorami logicznymi (OR, AND lub NOT),
— wyrażeniem warunkowym, gdy warunki złożone traktujemy jak warunki proste (gdy każdy z nich posiada określoną — prawda lub fałsz — wartość logiczną) i na nich wykonujemy operacje logiczne najczęściej z wykorzystaniem nawiasów.

W dalszych rozważaniach każdy element graficzny:

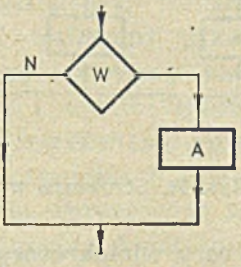
będzie oznaczał opisany w programie warunek, gdzie przyjęto następujące oznaczenia:

W — badanie, czy warunek jest spełniony,
T — tak (warunek przyjmuje wartość logiczną prawda),
N — nie (warunek przyjmuje wartość logiczną fałsz.)

Struktura wyboru może mieć postać:

— uproszczoną,
— pełną,
— wyboru wielokrotnego.

Postać uproszczoną przedstawia rysunek 12.



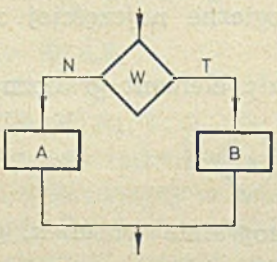
Rys. 12. Struktura IF THEN

Ta struktura wyboru może być opisana za pomocą instrukcji IF THEN (bez ELSE):

```
.
.
.
IF W THEN A;
.
.
.
```

lub kodując w części ELSE zdanie puste
IF W THEN A; ELSE;

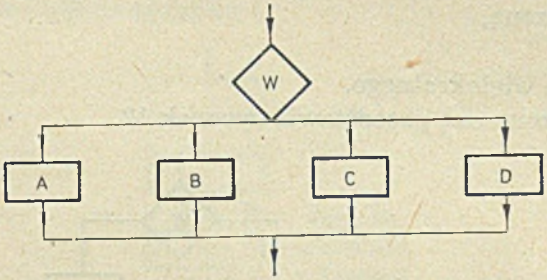
Postać pełną struktury wyboru można przedstawić za pomocą rysunku 13:



Rys. 13. Struktura IF THEN ELSE

oraz opisać w programie za pomocą instrukcji:
IF W THEN B; ELSE A;

Postać wyboru wielokrotnego przedstawia rysunek 14.



Rys. 14. Struktura wyboru wielokrotnego

Struktura wyboru wielokrotnego nie ma odpowiednich instrukcji we wszystkich językach programowania, ale można ją opisać przy odpowiedniej konstrukcji wyrażenia W. W języku programowania PASCAL wykonanie wyboru wielokrotnego odbywa się za pomocą instrukcji CASE.

Struktura wyboru jednego z dwóch lub wielu ciągów instrukcji charakteryzuje się wyborem odpowiedniego ciągu instrukcji w zależności od wartości logicznej, jaką przyjmuje warunek. Po wykonaniu wybranego ciągu instrukcji sterowanie jest przekazywane do następnej instrukcji zapisanej po instrukcji wyboru.

Miejsce przekazania sterowania jest wspólne dla wszystkich ujętych w strukturze wyboru sytuacji szczegółowych, co umożliwia spełnienie zasady przekazywania sterowania z góry w dół. Kierunek przebiegu sterowania jest konsekwentnie realizowany mimo, że część zakodowanych instrukcji w konkretnej sytuacji nie jest wykonana.

Powtarzanie

Struktura wyboru umożliwia kodowanie alternatywnie realizowanych ciągów operacji. Innym równie często występującym sposobem kodowania są cykle programowe. Cykle pozwalają na zapis czynności, które muszą być wykonane wielokrotnie. Iteracja jest sposobem postępowania wykorzystywanym nie tylko w obliczeniach naukowo-technicznych, ale również w przetwarzaniu danych. Struktura powtarzania umożliwia kodowanie takich procedur.

Struktura powtarzania może występować w postaci:

— DO UNTIL (WYKONUJ CHYBA ŻE)

— DO WHILE (WYKONUJ W CZASIE KIEDY)

Rozróżnienie tych dwóch postaci struktury powtarzania wywodzi się z logiki organizacji struktury. Warunek określający krotność powtarzania może być testowany przed wykonaniem powtarzanej części lub może być testowany na końcu powtarzanej części instrukcji. Miejsce i wynik testowania warunku mają wpływ na wykonanie lub niewykonanie powtarzanej części instrukcji. Przedstawiamy to na rysunku 15, na którym widać, że niezależnie od wyniku badania warunku, w wypadku struktury typu DOUNTIL część powtarzana A zostanie wykonana co najmniej jeden raz. Różnica w miejscu testowania warunku ma 3 ważne implikacje, które są zasadnicze przy konstrukcji struktury powtarzania.

— Warunek testuje pytanie czy jest to inne wykonanie części powtarzanej, czy też to samo. Testowanie warunku powinno następować przed każdym wykonaniem powtarzanej części, a szczególnie przed pierwszym.

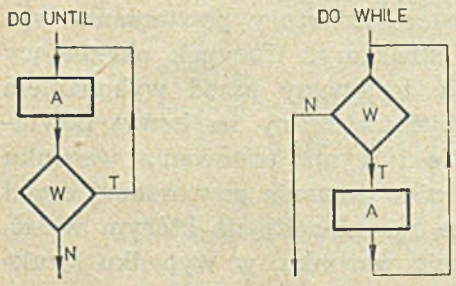
— Zerowe wykonanie części powtarzanej jest sytuacją prawidłową dla struktury; badanie warunku przed częścią powta-

rzaną pozwala na zerowe wykonanie czego nie zapewnia badanie warunku na końcu części powtarzanej.

— Struktura powtarzania zawsze składa się z maksymalnie możliwej liczby wykonań dla powtarzanej części. W strukturze typu UNTIL, spełnienie testowanego warunku (prawda) jest warunkiem koniecznym i wystarczającym dla zakończenia, natomiast spełnienie warunku (prawda) w strukturze typu WHILE jest wystarczające i konieczne dla każdego powtarzania.

Wybór odpowiedniej postaci struktury powtarzania jest bardzo istotny dla procesu obliczeniowego, szczególnie w sytuacji gdy pierwsze wykonanie części powtarzanej jest istotne dla całego procesu obliczeniowego. W większości wypadków wykorzystywana jest postać WHILE struktury powtarzania, jedynie język programowania FORTRAN wykorzystuje strukturę powtarzania typu UNTIL. Stosowanie struktury powtarzania w postaci UNTIL wymaga szczególnego przeanalizowania z powodu niekontrolowania warunku przy pierwszym wykonaniu części powtarzanej.

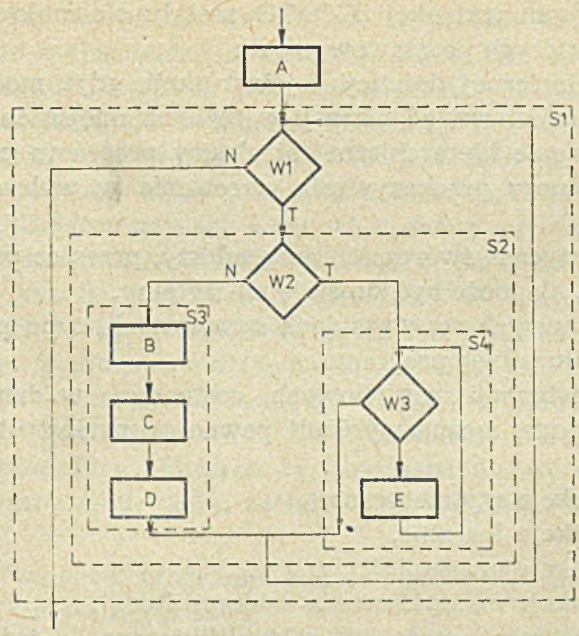
Obie struktury powtarzania można przedstawić w postaci schematu blokowego:



Rys. 15. Struktury powtarzania typu DOUNTIL I DOWHILE

Konsekwentne stosowanie wymienionych struktur umożliwia budowę hierarchicznej struktury programu z góry w dół. Struktury nie są stosowane w praktyce w postaci czystej, ale są w zależności od potrzeb odpowiednio zagnieżdżane. Przykład zagnieżdżania struktur podstawowych przedstawia rysunek 16.

Na rysunku przedstawiona jest struktura sekwencji składająca się z dwóch elementów: A i zagnieżdżonej w niej strukturze S1, która jest strukturą powtarzania. Częścią powtarzaną w strukturze S1 jest struktura wyboru S2. W strukturze wyboru



Rys. 16. Przykład zagnieżdżania struktur podstawowych

S2 zagnieżdżone są dwie struktury S3 i S4, gdzie S3 jest strukturą sekwencji składającą się z elementów B, C i D, natomiast struktura S4 jest strukturą powtarzania z częścią powtarzaną E.

Z przykładu tego wynika, jak konstruuje się program wykorzystując tylko podstawowe struktury sekwencji, wyboru i powtarzania. Metodą zagnieżdżania można budować dowolnie duże i skomplikowane programy nie burząc podstawowej zasady przekazywania sterowania z góry w dół. Każda z wymienionych struktur ma dokładnie jedno wejście do struktury i dokładnie jedno wyjście ze struktury. Poprzez zagnieżdżenie tych struktur w całym programie istnieje możliwość łatwego prześledzenia przebiegu sterowania.

Podstawowe struktury nie uwzględniają możliwości użycia instrukcji skoku bezwarunkowego GO TO. Rezygnacja z użycia instrukcji GO TO jest jednym z założeń programowania strukturalnego.

Pod adresem instrukcji GO TO zostały sformułowane liczne zarzuty, z których podstawowymi są:

— brak logicznej definicji tej instrukcji, gdyż może być stosowana w zależności od potrzeb w różnych miejscach programu,

— naruszenie hierarchicznej struktury programu przez możliwość dowolnego przekazywania sterowania w wybrane miejsce programu,

— „artystyczna twórczość" programisty, przez co program jest optymalny, ale może być zupełnie nieczytelny,

— przy nowych rozwiązaniach sprzętowych, typu pamięć wirtualna wydłuża bieg programu.

Wśród instrukcji maszynowych, dostępnych w danym języku programowania, można wyróżnić pewne grupy instrukcji, a mianowicie:

— instrukcje arytmetyczne,

— instrukcje logiczne,

— instrukcje przesunięć,

— instrukcje manipulowania zbiorami itd.

Każda z instrukcji może być zakwalifikowana do jednej z grup, jedynie instrukcja skoku bezwarunkowego nie daje się jednoznacznie przyporządkować do żadnej z grup. Instrukcję GO TO można wykorzystywać w dowolnym miejscu programu. Możliwość przekazania sterowania w dowolne miejsce programu, a ponadto możliwość modyfikowania tej instrukcji (np. za pomocą instrukcji ALTER w języku COBOL), powoduje takie splątanie dróg sterowania w programie, że szczegółowa analiza biegu programu jest niezwykle kłopotliwa.

Dla przykładu przeanalizujemy wycinek programu napisanego w języku programowania PL/1, gdzie instrukcja GO TO jest używana bardzo często:

```
1: IF A = 10 GO TO 7;
2: IF A = 5  GO TO 5;
3: X = 3;
4: GO TO 8;
5: X = 5;
6: GO TO 8;
7: X = 1;
8: Y = 2+X;
```

Rozpatrzymy wyrażenia warunkowe w przedstawionej kolejności, w celu wyjaśnienia, co dokonuje się w tym krótkim wycinku programu. Najpierw, jeśli warunek A = 10 jest prawdą, program przechodzi do wykonania instrukcji zapisanej w wierszu 7. Z kolei gdy warunek A = 10 jest fałszem, natomiast A = 5 jest prawdą, sterowanie przekazane jest do instrukcji z wiersza 5. Dopiero gdy oba warunki są fałszem, program przechodzi do wykonania instrukcji z wiersza 3, po czym sterowanie przekazane jest do instrukcji z wiersza 8. Stwierdzenie, jaką wartość przyjmie Y, po przebiegu przez ten wycinek programu, nie jest więc proste, chociaż samo zagadnienie jest nieskomplikowane. Jeśli jednak, nasz docelowy wiersz 8 znajduje się kilka stron dalej, i gdyby wiersze 3, 5 i 7 zawierały bardziej skomplikowane procedury obliczeniowe, to trzeba byłoby analizować wiele stron programu, zanim zdołalibyśmy stwierdzić co program wykonuje.

Ten sam wycinek programu można zapisać w równoważnej, przejrzystej i czytelnej postaci:

```
IF A = 10 THEN X = 1; ELSE
    IF A = 5 THEN X = 5; ELSE X = 3;
Y = 2+X.
```

W programowaniu strukturalnym za podstawową cechę programu uznaje się jego przejrzystość. Wykorzystanie instrukcji skoku bezwarunkowego jest zaprzeczeniem tego i w związku z tym jednym z haseł programowania strukturalnego jest programowanie bez GO TO.

Tendencje rozwoju oprogramowania wskazują na dążenie do elastyczności programów, do łatwości konserwacji programu, nie tylko przez autora, ale również przez innych programistów. Zaniechanie korzystania z rozkazu GO TO ogranicza „artystyczną twórczość" programistów. Programy realizujące tę samą funkcję napisane bez GO TO przez różnych programistów są do siebie bardzo podobne w przeciwieństwie do identycznie powstałych programów, gdy wykorzystywana jest instrukcja GO TO.

W maszynach drugiej generacji, program w maszynie był w całości podczas realizacji i przez cały czas znajdował się w pamięci operacyjnej. Ze względu na taką organizację biegu progra-

mu optymalizacja zajętości pamięci, osiągana przez wykorzystanie instrukcji GO TO, miała swoje uzasadnienie.

Użycie techniki nakładania, czyli budowa programów nakładkowych, gdzie w pamięci znajduje się jedynie moduł główny i w zależności od potrzeb przywołuje on z pamięci zewnętrznej kolejne nakładki, które zajmują to samo miejsce w pamięci, spowodowało pewne ograniczenia w stosowaniu instrukcji GO TO. Skoki bezwarunkowe winny się w zasadzie ograniczać do jednej nakładki, gdyż każdy skok z nakładki do nakładki zmusza do wykonania transmisji z pamięci zewnętrznej do pamięci operacyjnej, co powoduje że oszczędność na zajętości pamięci nie rekompensuje strat czasowych potrzebnych do manipulowania nakładkami.

Problem stał się jeszcze bardziej widoczny przy zastosowaniu stronicowania pamięci operacyjnej. Podział programu na „strony" pamięci operacyjnej jest realizowany poza programem, co znacznie utrudnia możliwość przewidywania tego podziału, a w związku z tym użycie instrukcji GO TO prawie zawsze będzie się łączyło z potrzebą sprowadzenia do pamięci strony, do której następuje odwołanie. Częste używanie instrukcji GO TO może spowodować, że czas efektywnego biegu programu będzie niewielkim procentem przy całkowitym czasie realizacji.

Nowe rozwiązania techniczne wymuszają ograniczenie stosowania instrukcji skoku bezwarunkowego przez co wychodzą naprzeciw postulatom programowania strukturalnego.

3. Czytelność programu

Programy wykorzystujące instrukcję skoku bezwarunkowego umożliwiają zastosowanie optymalnych rozwiązań zarówno z punktu widzenia projektowania programu, jak i jego eksploatacji. Najczęściej optymalność rozwiązań okupiona jest nieczytelnością i nikłą przejrzystością listu programu. Brak przejrzystości i czytelności oczywiście nie wpływa na walory eksploatacyjne programu tak długo, gdy program jest wykorzystywany w niezmienionej formie i funkcjonuje poprawnie. Problemy powstają, gdy trzeba program zmodyfikować lub poprawić. Prak-

tycznie tylko autor programu może zorientować się w zawiłościach następstwa rozkazów związanych z bezwarunkowym przekazywaniem sterowania w dowolne miejsca programu.

Konsekwentne stosowanie struktur podstawowych w konstrukcji programu podnosi wydatnie jego przejrzystość i czytelność, co jest nie bez znaczenia przy jego konserwowaniu i modyfikowaniu. Problematyka czytelności programu jest podnoszona do wysokiej rangi w metodzie programowania strukturalnego. Położenie nacisku na taki styl programowania, aby program był czytelny, wynika z faktu, że postać źródłowa programu jest bardziej przeznaczona do czytania przez ludzi niż przez komputer. Wydruk kodu źródłowego jest także dokumentem, do którego odwołujemy się w przyszłości, co każe przywiązywać dużą wagę do czytelności kodu.

Czytelność programu można osiągnąć nie tylko przez stosowanie struktur podstawowych, ale również przez odpowiednie zabiegi redakcyjne. Staranna i przejrzysta forma kodu źródłowego nie ma żadnego wpływu na warunki eksploatacyjne programu, a może wyraźnie zwiększyć czytelność programów. Stosowanie akapitów, odstępów, znaków przestankowych czy słów opcjonalnych instrukcji, dopuszczają w większości języki programowania wyższego rzędu.

Istotne znaczenie dla czytelności programu ma właściwy wybór nazw zmiennych wykorzystywanych w programie. Ten prosty i tani sposób zwiększenia czytelności kodu źródłowego nie zawsze jest wystarczająco akcentowany i stosowany przez programistów. Nazwy nadawane przez programistę nie powinny być przypadkowe, a ich struktura winna dostarczać czytelnikowi programu dodatkowych informacji.

Programista nadaje nazwy różnym elementom w programie i nazwy te mają najczęściej znaczenie lokalne w ramach pisanego programu. O ile nadawanie nazw etykietom, sekcjom, paragrafom może mieć charakter lokalny, to dla większej przejrzystości — dobrze byłoby — jeśli chodzi o cały system informatyczny — aby pole zawierające cenę asortymentu nie nazywało się w jednym programie A w drugim B, a jeszcze w innym CEN. Dążąc do zwiększenia czytelności programów w opracowywanym systemie informatycznym, dobrze jest zbudować za-

sady nazewnictwa na potrzeby całego systemu tak, aby uniknąć sytuacji opisanej. Oto jedno z możliwych rozwiązań tego problemu.

System informatyczny przetwarza dane, które są rozmieszczone w zbiorach. Dla każdego systemu można wydzielić skończoną kolekcję danych wykorzystywanych w systemie

$$D = \{d_1, d_2, ..., d_n\}, \tag{1}$$

gdzie d_i, są to kolejne dane występujące w różnych fazach realizacji systemu. Z technologicznego punktu widzenia ze zbioru D można wydzielić podzbiory:

$$\begin{aligned} DWE &= \{dwe_1, dwe_2, ..., dwe_k\} \\ DPR &= \{dpr_1, dpr_2, ..., dpr_l\} \\ DWY &= \{dwy_1, dwy_2, ..., dwy_m\} \end{aligned} \tag{2}$$

gdzie:
dwe_i — kolejne dane wejściowe,
dpr_i — kolejne dane stałe przechowywane w zbiorach systemu,
dwy_i — kolejne dane wyjściowe.

Pojedyncza dana d_k ze zbioru (1) występująca w systemie może w szczególności należeć równocześnie do wszystkich podzbiorów (2), ale musi należeć przynajmniej do jednego z nich. Oznacza to, że suma logiczna podzbiorów (2) daje zbiór (1).

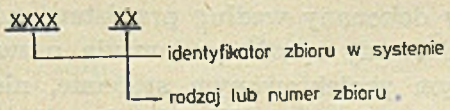
$$D = DWE \cup DPR \cup DWY.$$

Uświadomienie tych prostych i oczywistych zależności zachodzących między danymi, przetwarzanymi przez system, prowadzi do stwierdzenia, że każda dana ze zbioru (1) może mieć w całym systemie *jedną* nazwę i jeden opis. Dla przykładu dana zawierająca ilość „czegoś" może mieć w systemie nazwę ILOSC oraz jeden opis mówiący, że jest to przykładowo pole 5-znakowe. Jeśli rozważana dana występuje tylko w jednym z podzbiorów (2) sytuacja jest prosta, natomiast komplikuje się nieco, jeśli występuje w dwóch lub trzech podzbiorach. Nie wnikając w problematykę różnych postaci przechowywania i związanym z tym różnym opisem, zajmiemy się problemami nadawania nazw.

W celu osiągnięcia pełnej jednoznaczności nazw w systemie, proponuje się by w toku projektowania systemu wykonać następujące czynności.

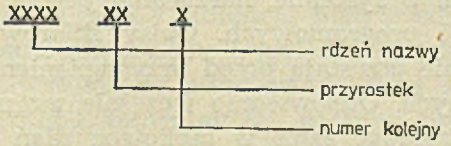
1. Opracować słownik nazw. Słownik ten zawiera skróty nazw wszystkich danych występujących w systemie. Nadana nazwa powinna w najkrótszy i unikalny sposób identyfikować dane systemu ze zbioru (1) [7].

2. Określić kod budowy nazwy zbiorów w sposób następujący:



Pierwszy człon kodu określa identyfikator zbioru, który umożliwia zidentyfikowanie zbioru oraz określenie jego przynależności do systemu. Człon drugi rozszerza identyfikator i określa rodzaj zbioru lub jego kolejny numer w systemie. Wykonanie tych dwóch czynności daje możliwość tworzenia unikalnych identyfikatorów danych.

3. Określić kod budowy nazw danych dla zbioru (1) w następujący sposób:



gdzie:

— rdzeń nazwy — skrót pobrany ze słownika,
— przyrostek — rodzaj lub numer zbioru, z którego pochodzi informacja,
— numer kolejny — numer kolejny dla takich samych danych lub typ rekordu, jeśli w rekordach różnych typów umieszczone są takie same dane.

Dołączenie numeru zbioru oraz typu rekordu (o ile ten występuje) do nazwy określonej w słowniku czyni taki identyfikator

[7] Nieco inny sposób skracania nazw, polegający na usuwaniu samogłosek, proponuje D. Van Tassel. Por. D. Van Tassel, *Praktyka programowania*, Warszawa 1978, s. 21.

unikalny w całym systemie oraz wiąże tak określoną daną z konkretnym zbiorem. Kontynuując przykładową ilość „czegoś" możemy mieć ILOSC-KAR, ILOSC-TM2, ILOSC-27 reprezentujące tę samą daną umieszczoną w różnych zbiorach systemu. W wypadku ilości „czegoś-innego" przechowywanego w tych samych zbiorach przyrostki będą te same, a zmianie ulegnie tylko rdzeń nazwy. Biorąc to pod uwagę możemy w dokumentacji projektowej oraz w programach źródłowych posługiwać się tak skonstruowaną nazwą bez obawy o jakąkolwiek niejednoznaczność.

Wybór nazw dokonany według przedstawionej metody wprowadza sformalizowany sposób nadawania nazw dla wszystkich danych w całym projektowanym systemie, niezależnie od podziału systemu na programy. Niezależnie od zwiększenia czytelności pojedynczych programów, istnieje możliwość konstruowania wspólnych modułów programowych, które zawierają definicję zbiorów systemu lub innych wspólnie wykorzystywanych kolekcji danych. Języki programowania wyższego rzędu dostarczają prostych narzędzi programowych włączenia tych modułów do kodu źródłowego programu (operator COPY dla języka COBOL, INCLUDE dla języka PL/1).

Tworząc bibliotekę zdań źródłowych, dla opisu zbiorów wykorzystywanych w systemie, narzucamy pewną organizację prac projektowych i programowych. Opisy zbiorów wykorzystywanych w systemie powstają przed przystąpieniem do prac czysto programistycznych. Odpowiednio wcześnie można więc dla każdego zbioru utworzyć zapis w bibliotece zdań źródłowych. Dla poszczególnych danych nadawane są unikalne nazwy, zgodnie z przedstawioną propozycją, oraz atrybuty zgodnie z zasadami przyjętego języka programowania. Zapisane w bibliotece zdań źródłowych opisy zbiorów mogą być wykorzystywane przez wszystkie programy, które przetwarzają dany zbiór.

W kolejnym przykładzie pokazujemy, jak wykorzystując bibliotekę zdań źródłowych w języku PL/1 włączono do programu opis zbioru TABINW. Operacja ta jest generowana przez wiersz 21 załączonego programu źródłowego.

W postaci źródłowej programu kodowanej przez programistę w miejscu opisu wykorzystywanego zbioru występuje wiersz %INCLUDE TABINW. W toku komputerowego przygotowania

```
COMPILE-TIME MACRO PROCESSOR
MACRO SOURCE LISTING
  1       OBSTAB: PROC OPTIONS(MAIN);     /* OBSLUGA TABLICY INWERSYJNEJ */     00000100
  2                                                                             00000200
  3        DCL                                                                  00000300
  4        1 TA(500)          /* TABLICA DLA JEDNOSTEK NOWO-WPROWADZANYCH */    00000400
  5        ,  2 JED CHAR(6)                                                     00000500
  6        ,  2 TER PIC '999'                                                   00000600
  7        ,  2 ODL PIC '999'                                                   00000700
  8        , WSK FIXED BINARY_INIT(0)  /* WSKAZNIK KONCA DANYCH */              00000800
  9        , 1 TABLR(500)     /* TABLICA ROBOCZA */                             00000900
 10        ,  2 JEDR CHAR(6)                                                    00001000
 11        ,  2 TERR PIC '999'                                                  00001100
 12        ,  2 ODLR PIC '999'                                                  00001200
 13        ;                                                                    00001300
 14        ON ENDFILE(SYSIN) WSK = 1;                                           00001400
 15        TA = 0; TABLR = 0;                                                   00001500
 16        DO WHILE(WSK = 0);  /* WCZYTANIE LISTY NOWO-WPROWADZANYCH */         00001600
 17          J = J + 1;                                                         00001700
 18          GET EDIT(JED(J),TER(J),ODL(J))(SKIP,A(6),X(1),A(3),X(1),A(3));     00001800
 19        END;                                                                 00001900
 20          J = J - 1;                                                         00001950
 21              CLOSE FILE(SYSIN);                                             00002000
 22        %INCLUDE TABINW;                                                     00002100
 23        PUT PAGE;                                                            00002120
 24        PUT EDIT('RAPORT PRZEBIEGU OBSLUGI TABLICY INWERSYJNEJ',             00002140
 25                 (44)'=','OBLICZENIA Z DNIA  (.DATE)                         00002160
 26                 (X(30),A,SKIP,X(30),A,SKIP,X(35),A,A);                     00002180
 27        PUT EDIT('WPROWADZONO NOWYCH JEDNOSTEK : ',J)(SKIP(2),A,F(5));       00002190
 28        CLOSE FILE(TABINW);                                                  00002195
 29        DO I = 2 TO J;          /* KONTROLA UPORZADKOWANIA */                00002200
 30          IF JED(I-1) < JED(I) THEN; ELSE DO;                                00002300
 31            PUT EDIT('NIEUPORZADKOWANY ZBIOR',JED(I-1),JED(I))               00002340
 32                    (SKIP,A,X(3),A,X(3) A);                                  00002360
 33                  DISPLAY ('NIEUPORZADKOWANY ZBIOR'); STOP;                  00002400
 34                                  END;                                       00002500
 35        END;                                                                 00002600
 36        L = 1;  I = 1;  K = 1;                                               00002700
 37      /* WSKAZNIKI I - DLA TABLICY K - DLA LISTY NOWYCH L - DLA TABL-WY      00002710
 38                J - ILOSC NOWO-WPROWADZANYCH */                             00002720
```

programu do pracy, w miejscu tego zdania źródłowego umieszczany jest odpowiedni zapis z biblioteki zdań źródłowych [8]. Z jednorazowego przygotowania opisu i umieszczenia go w bibliotece zdań źródłowych wynika, że poszczególne dane ze zbioru opatrzone są odpowiednim komentarzem, co przy stosowaniu opisu zbioru w każdym programie osobno jest rzadko wykorzystywane. W wylistowanym programie, występuje w początkowej części, wydruk dołączonych zdań, a w dalszej — kompletny już program, który jest poddawany procesowi kompilacji.

---

[8] Opisem włączanym w program źródłowy nie musi być tylko opis zbioru, może to być dowolny ciąg zdań źródłowych języka programowania. Mogą to być całe sekcje lub procedury programu. Po włączeniu są one traktowane tak samo, jak napisane jawnie w programie źródłowym. Na tej zasadzie tworzone są makroinstrukcje programisty, które są alternatywami dla podprogramów. Podprogram w odróżnieniu od makroinstrukcji nie jest włączany w odpowiednie miejsce programu, a występuje gdzieś indziej. W miejscu jego wykorzystania występuje tylko wywołująca go instrukcja (CALL), która powoduje czasowe przekazanie sterowania w inne miejsce programu. Wykorzystanie makroinstrukcji nie wymaga przekazania sterowania, gdyż jest to tekst włączony w wykonywaną listę instrukcji w toku komputerowego przygotowania programu.

```
INCDDDED TEXT FOLLOWS FROM DD.MEMBER =  SYSLIB  .TABINW

 85        DCL              /* ZBIOR ZAWIERAJACY TABLICE INWERSYJNA */          00000100
 86        TABINW FILE RECORD SEQUENTIAL UNBUFFERED                             00000200
 87      , 1 TABLINW        /* TABLICA INWERSYJNA */                             00000300
 88      ,  2 AKTROZ FIXED BINARY    /* AKTUALNY ROZMIAR TABLICY INWERSYJNEJ */ 00000400
 89      ,  2 TABL(1000)              /* TABLICA O MAKSYMALNYM WYMIARZE 1000 */ 00000500
 90      ,   3 JEDORGI CHAR(6)         /* KOD JEDNOSTKI ORGANIZACYJNEJ */       00000600
 91      ,   3 NRTERIN PIC '999'       /* KOD TERYTORIALNY */                   00000700
 92      ,   3 NRTAODI PIC '999'          /* NUMER W TABLICY ODLEGLOSCI */      00000800
 93      ;                                                                      00000900
 94        READ FILE(TABINW) INTO(TABLINW);                                     00001000
           OBSTAB: PROC OPTIONS(MAIN);     /* OBSLUGA TABLICY INWERSYJNEJ */    00000100


 MACRO DIAGNOSTIC MESSAGES



THE SUPPRESSION OF WARNING MESSAGES HAS BEEN REQUESTED.
           OBSTAB: PROC OPTIONS(MAIN);     /* OBSLUGA TABLICY INWERSYJNEJ */    00000100


                  SOURCE LISTING


STMT LEVEL NEST
  1                  OBSTAB: PROC OPTIONS(MAIN);     /* OBSLUGA TABLICY INWERSYJNEJ */
  2     1             DCL
                       1 TA(500)        /* TABLICA DLA JEDNOSTEK NOWO-WPROWADZANYCH */
                     ,  2 JED CHAR(6)
                     ,  2 TER PIC '999'
                     ,  2 ODL PIC '999'
                     , WSK FIXED BINARY INIT(0)  /* WSKAZNIK KONCA DANYCH */
                     , 1 TABLR(500)     /* TABLICA ROBOCZA */
                     ,  2 JEDR CHAR(6)
                     ,  2 TERR PIC '999'
                     ,  2 ODLR PIC '999'
                     ;
  3     1             ON ENDFILE(SYSIN) WSK = 1;
  5     1             TA = 0;   TABLR = 0;       J = 0;
  8     1             DO WHILE(WSK = 0);    /* WCZYTANIE LISTY NOWO-WPROWADZANYCH */
  9     1     1         J = J + 1;
 10     1     1         GET EDIT(JED(J),TER(J),ODL(J))(SKIP,A(6),X(1),A(3),X(1),A(3));
 11     1     1        END;
 12     1               J = J - 1;
 13     1                     CLOSE FILE(SYSIN);
 14     1             DCL              /* ZBIOR ZAWIERAJACY TABLICE INWERSYJNA */
                     TABINW FILE RECORD SEQUENTIAL UNBUFFERED
                     , 1 TABLINW        /* TABLICA INWERSYJNA */
                     ,  2 AKTROZ FIXED BINARY    /* AKTUALNY ROZMIAR TABLICY INWERSYJNEJ */
                     ,  2 TABL(1000)              /* TABLICA O MAKSYMALNYM WYMIARZE 1000 */
                     ,   3 JEDORGI CHAR(6)         /* KOD JEDNOSTKI ORGANIZACYJNEJ */
                     ,   3 NRTERIN PIC '999'       /* KOD TERYTORIALNY */
                     ,   3 NRTAODI PIC '999'          /* NUMER W TABLICY ODLEGLOSCI */
                     ;
 15     1             READ FILE(TABINW) INTO(TABLINW);
 16     1             PUT PAGE;
 17     1             PUT EDIT('RAPORT PRZEBIEGU OBSLUGI TABLICY INWERSYJNEJ',
                                (44)'=','OBLICZENIA Z DNIA    ',DATE)
                                (X(30),A,SKIP,X(30),A,SKIP,X(35),A,A);
 18     1             PUT EDIT('WPROWADZONO NOWYCH JEDNOSTEK : ',J)(SKIP(2),A,F(5));
 19     1              CLOSE FILE(TABINW);
 20     1             DO I = 2 TO J;          /* KONTROLA UPORZADKOWANIA */
 21     1     1         IF JED(I-1) < JED(I) THEN; ELSE DO;
 24     1     2             PUT EDIT('NIEUPORZADKOWANY ZBIOR',JED(I-1),JED(I))
                                     (SKIP,A,X(3),A,X(3),A);
 25     1     2                  DISPLAY ('NIEUPORZADKOWANY ZBIOR'); STOP;
 27     1     2                                  END;
 28     1     1        END;
 29     1             L = 1;  I = 1;  K = 1;
                     /* WSKAZNIKI I - DLA TABLICY K - DLA LISTY NOWYCH L - DLA TABL-WY
                                J - ILOSC NOWO-WPROWADZANYCH */
```

Stosowanie jednolitych i przejrzystych zasad nazewnictwa, odpowiednie wykorzystanie komentarzy, staranna redakcja mają decydujący wpływ na czytelność programu. Dbałość o czytelność i przejrzystość programu nie wiąże się ze wzrostem pracochłonności, a ma istotny wpływ na jego walory użytkowe, szczególnie w fazie konserwacji. Celem każdego programisty winno być opracowanie programu nie tylko sprawnego, ale również czytelnego.

4. Macierzowa metoda definiowania algorytmu dla tabulogramu

Podejście strukturalne do programowania musi być odpowiednio przygotowane we wcześniejszych fazach opracowania projektu. Konsekwentne przestrzeganie zasady hierarchicznej struktury programu, podział programu na moduły funkcjonalne, będą naturalnym sposobem postępowania dla przyjętych we wcześniejszych fazach opracowania projektu sposobów postępowania. Jak utworzenie biblioteki zdań źródłowych wymusza odpowiednią organizację prac programowych, tak też sposoby opracowania i opisu prac projektowych wpływają na proces programowania.

Sposoby prezentacji algorytmu obliczeniowego lub algorytmu postępowania zawsze nastręczały duże trudności. Metody zapisu zwykle okazywały się nieprecyzyjne i z reguły konieczny był dodatkowy komentarz lub objaśnienie autora algorytmu. Problem nabiera szczegółowej wagi przy opisie algorytmu obliczeniowego dla poszczególnych pól tabulogramu.

Wszystkie dane występujące w tabulogramie, to dane zawierające się w podzbiorze DWY z (2). Dane te można podzielić na:

— stałe,
— przenoszone,
— obliczane.

Dane stałe mają ściśle określoną wartość i najczęściej służą do identyfikacji lub czytelniejszej prezentacji wyników. Dane przenoszone natomiast muszą mieć swój odpowiednik w podzbiorach DPR lub DWE. Algorytm otrzymania takiej danej wyjściowej dwy_i jest stosunkowo prosty do przedstawienia, gdyż przy stosowaniu omówionych już zasad budowy nazw danych, zapis:

$$dwy_i = dpr_j$$

jest precyzyjny i jednoznacznie określa pochodzenie danej.

Dane obliczane równie jednoznacznie i precyzyjnie można określić za pomocą odpowiednio rozbudowanego zapisu matematycznego, w którym po lewej stronie znaku równości występuje dana wyjściowa, a po prawej matematyczny zapis sposobu jej otrzymania. Wykorzystanie w zapisie jednoznacznych nazw umożliwia dużą precyzję takiej prezentacji każdego algorytmu obliczeniowego.

Przykładowa dana ilość „czegoś" może być określona w sposób następujący:

$$\text{ILOS} - \text{TO3} = \left(\sum_{i=1}^{N} \text{ILOS} - \text{WE} \right) / N,$$

gdzie i przebiega przez wszystkie zapisy, w których dana KLU-WE ma wartość 15.

Uogólniając można powiedzieć, że każda dana wyjściowa da się przedstawić w postaci jednoznacznego wzoru, gdzie argumentami będą stałe wartości oraz jednoznaczne nazwy danych wejściowych lub przechowywanych w zbiorach systemu.

Szczególną klasą danych wyjściowych są tabulogramy, które najczęściej prezentują wyniki działania systemu. Duża ilość danych zawarta na kolejnych stronach tabulogramu nie sprzyja podaniu szczegółowego algorytmu obliczeniowego dla każdej z nich. Najczęściej programista, który ma oprogramować wydruk określonego tabulogramu, otrzymuje projekt szaty graficznej wydruku oraz pewne wyjaśnienia ogólne, jak dany tabulogram wyprowadzać. Dla tabulogramów typu *wykaz*[9], taka ilość informacji może być wystarczająca. Dla tabulogramów bardziej skomplikowanych, ze złożonymi procedurami obliczeniowymi, jest to materiał niewystarczający. Dołączany ewentualnie opis słowny jest również mało przydatny ze względu na swą długość, wynikającą z potrzeby opisu prawie każdego pola oraz małą precyzję zapisu. Precyzyjny i w miarę krótki zapis można osiągnąć dzięki zastosowaniu formuły matematycznej.

[9] Przez tabulogram typu *wykaz* rozumiemy tabulogram, w którym występuje niewielka różnorodność danych oraz są to najczęściej dane przenoszone lub stałe, a ewentualne dane obliczane posiadają bardzo proste algorytmy obliczeniowe.

Modelem tabulogramu może być macierz:

$$T = \begin{vmatrix} d_{11} & d_{12} & \dots & d_{1m} \\ d_{21} & d_{22} & \dots & d_{2m} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ d_{n1} & d_{n2} & \dots & d_{nm} \end{vmatrix}$$

której elementami d_{ij} są dane wyjściowe z podzbioru DWY, zawarte w tabulogramie, m jest ilością kolumn, w jakich dane są wyprowadzane na drukarkę, a n jest ilością wierszy tabulogramu. Wartość m, ze względu na szerokość papieru drukarki nie jest liczbą zbyt dużą. Każde wystąpienie kolumny jest jednak istotne, gdyż dotyczy najczęściej innej grupy danych wyjściowych. Wartość n, może przybierać bardzo różne wielkości, od kilku wierszy w wypadku tabulogramów małych objętościowo, do bardzo dużych liczb dla objętościowo dużych tabulogramów. W macierzy (3) trudno jest znaleźć kolumny równe, tzn. takie k i l aby:

$$d_{ik} = d_{il}, \qquad \text{dla } i = 1, 2, ..., n.$$

Równość dwóch kolumn macierzy oznaczałaby najczęściej wadliwy projekt tabulogramu, gdzie te same dane występują wielokrotnie. Wypadek równości kolumn macierzy nie zmienia proponowanego rozwiązania tylko wymaga rozważania takich kolumn tak, jak gdyby zawierały różne dane. W praktyce wypadki takie występują bardzo rzadko. Natomiast istnieje wiele takich j, że:

$$d_{jk} = d_{pk}, \qquad \text{dla określonego } p, \text{ gdzie } k = 1, 2, ..., m.$$

Oznacza to, że wiele wierszy tabulogramu zawiera te same dane wyjściowe [10]. Wiersze takie stanowią jedną klasę danych wyjściowych i w związku z tym można rozważać je wspólnie. W ma-

[10] Równość i nierówność danych rozumiana tutaj jest w sensie mnogościowym, tzn. dwa wiersze lub kolumny tabulogramu zawierające informacje o materiałach są równe, mimo że dotyczą różnych materiałów, jednolita jest natomiast ich struktura.

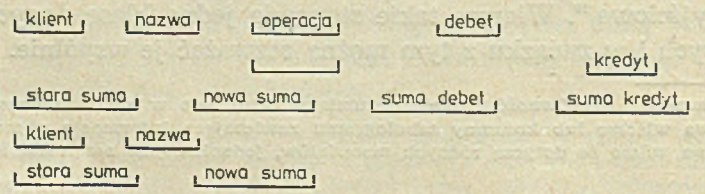
cierzy (3) można wyróżnić pewną, ściśle określoną liczbę klas wierszy. Każda klasa jest reprezentowana w tabulogramie przez dane wyjściowe o jednolitej strukturze. Dokonujemy kompresji macierzy T tak, że w macierzy (3) pozostawiamy jeden wiersz, który ma reprezentować całą klasę. Dzięki takiemu zabiegowi po dokonaniu kompresji macierzy (3), polegającej na zastąpieniu wierszy jednej klasy przez reprezentanta, otrzymujemy macierz T, gdzie liczba kolumn jest najczęściej równa m, a liczba wierszy jest ograniczona do n', przy czym zachodzi relacja: $n' \leqslant n$, gdzie n' jest liczbą wierszy w T', a n liczbą wierszy w T.

Macierz T' otrzymana w wyniku kompresji macierzy (3) opisuje jednoznacznie wszystkie dane zawarte w tabulogramie. Jak stwierdziliśmy, dane te pochodzą z podzbioru DWY, a te z kolei dają się przedstawić w postaci odpowiednich zapisów matematycznych, gdzie argumentami są dane z podzbiorów DWE i DPR.

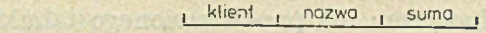
W ten sposób opisując każdy element macierzy T' mamy jednoznacznie, precyzyjnie i przejrzyście zdefiniowany algorytm otrzymania pola tabulogramu. Część elementów macierzy T będzie zawierała wartości stałe równe spacji, zeru lub jakiejś określonej wartości numerycznej lub alfanumerycznej. Elementy te również muszą być opisane dla otrzymania pełnego zapisu algorytmu. W wypadku danych przenoszonych będzie to zwykłe zrównanie dwóch nazw. W pozostałych wypadkach należy dążyć do wyrażenia poszczególnych elementów macierzy, będących odpowiednikiem pól tabulogramu, za pomocą wzorów matematycznych.

Przykład 3

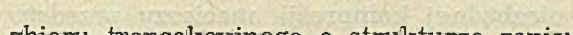
Dla lepszego zrozumienia przedstawionej metody rozważmy przykład, w którym zdefiniujemy algorytm otrzymania tabulogramu w postaci:



Tabulogram zawiera listę klientów banku z wyszczególnionymi operacjami za dany okres rozliczeniowy. Tabulogram powstaje z danych zbioru głównego o strukturze zapisu:



oraz ze zbioru transakcyjnego o strukturze zapisu:



gdzie kod przyjmuje wartości *D* (dla debet) lub *C* (dla kredyt).

Zasady tworzenia wierszy tabulogramu są następujące:

— wiersze typu 4 i 5 występują dla zapisów ze zbioru głównego, dla których nie ma zapisów transakcyjnych,

— wiersze typu 1, 2 i 3 występują tylko dla tych klientów, którzy wykonywali operacje w okresie rozliczeniowym, przy czym liczba wierszy typu 1 lub 2 dla danego klienta zależy od liczby zapisów w zbiorze transakcyjnym, natomiast wiersz typu 3 występuje jeden, dla każdego klienta, który miał operacje,

— przedstawiona struktura listy jest analogiczna dla wszystkich klientów zapamiętanych w zbiorze systemu.

Zgodnie z przyjętym w metodzie sposobem postępowania, pierwszym krokiem jest utworzenie odpowiedniego słownika nazw dla danych występujących w rozważanym problemie.

W rozważanym w przykładzie problemie pełną kolekcję danych oraz odpowiadający im słownik nazw stanowią:

KLIE — identyfikator klienta,
NAZW — nazwa klienta,
SUMA — stan rachunku,
SUMO — wartość operacji,
SUMS — stan rachunku stary (na początku okresu rozliczeniowego),
SUMN — stan rachunku nowy (na zakończenie okresu rozliczeniowego),
NUMO — numer operacji,
KODO — kod operacji.

Zbiór główny niech ma nazwę GLOW04, zbiór transakcyjny TRAN08, a tabulogram TABL45.

Zgodnie z określonymi wcześniej zasadami pole o nazwie KLIE08 oznacza indentyfikator klienta ze zbioru transakcyjnego.

Po wykonaniu tych czynności, mających na celu jednoznaczne określenie nazw dla wykorzystywanych danych możemy przystąpić do definiowania algorytmu dla przedstawionego tabulogramu. Po wykonaniu niezbędnej kompresji macierzy przedstawiającej pełny tabulogram, otrzymujemy macierz o pięciu kolumnach i pięciu wierszach, gdzie:

wiersz 1 — reprezentuje wszystkie pierwsze zapisy transakcyjne dla kolejnego klienta,

wiersz 2 — reprezentuje wszystkie dalsze zapisy transakcyjne dla danego klienta,

wiersz 3 — reprezentuje wszystkie wiersze sumujące dla transakcji danego klienta,

wiersz 4 — reprezentuje wszystkie wiersze dla rekordu zbioru głównego,

wiersz 5 — uzupełnienia wierszy 4.

Zapisując tabulogram jako macierz, mamy:

$$TABL45 = [a_{ik}], \quad \text{gdzie: } i = 1, 2, ..., 5, \quad k = 1, 2, ..., 5$$

Szczegółową definicję wartości pól tabulogramu można zapisać:

a_{11} = KLIE08

a_{12} = NAZW04

a_{13} = NUMO08

$$a_{14} = \begin{cases} \text{SUMO08} & \text{dla KODO08} = \text{D} \\ \text{spacje} & \text{dla KODO08} = \text{C} \end{cases}$$

$$a_{15} = \begin{cases} \text{spacje} & \text{dla KODO08} = \text{D} \\ \text{SUMO08} & \text{dla KODO08} = \text{C} \end{cases}$$

$a_{21} = a_{22}$ = spacje

a_{23} = NUMO08

a_{24} = jak a_{14}

a_{25} = jak a_{15}

a_{31} = SUMA04

$a_{32} = \text{SUMA04} + a_{34} - a_{35}$

a_{33} = spacje
$a_{34} = \Sigma(a_{14}+a_{24})$ dla tego samego KLIE08
$a_{35} = \Sigma(a_{15}+a_{25})$ dla tego samego KLIE08
a_{41} = KLIE04
a_{42} = NAZW04
$a_{43} = a_{44} = a_{45}$ = spacje
a_{51} = SUMA04

$$a_{52} = \begin{cases} \text{SUMA04} & \text{gdy brak } a_{32} \text{ dla danego KLIE04} \\ a_{32} & \text{gdy występuje } a_{32} \text{ dla danego KLIE04} \end{cases}$$

$a_{53} = a_{54} = a_{55}$ = spacje

Z przykładu tego widać wyraźnie sposób postępowania oraz wady i zalety tej metody. Przyjmując, że budowa słownika oraz zasady budowy nazw są opracowane jednorazowo na potrzeby całego systemu, to dla konkretnego tabulogramu problemem jest jedynie zbudowanie odpowiedniej macierzy oraz szczegółowe określenie algorytmów powstawania poszczególnych pól tabulogramu.

Wadą metody proponowanej jest wzrost pracochłonności, gdyż w wypadku bardziej skomplikowanych tabulogramów niż ten z omawianego przykładu, należy szczegółowo opracować algorytmy dla odpowiednio dużej macierzy. Ten wzrost pracochłonności rekompensowany jest dokładniejszym rozważaniem zawartości tabulogramu, nie tylko pod kątem przydatności występujących tam danych, ale również sposobem ich powstawania. Rozważania te zastępują opracowanie tzw. tablicy krzyżowej wejścia—wyjścia, proponowanej w ramach prac nad projektem technicznym przez niektóre metodyki projektowania systemów informatycznych.

Niekwestionowaną zaletą proponowanej metody jest jednoznaczność i kompletność definicji zadania programowego. Projekt graficzny tabulogramu wraz z takim opisem algorytmu jego powstawania jest materiałem do programowania bez potrzeby jakichkolwiek dodatkowych informacji. Opis algorytmów dla poszczególnych pól wskazuje dodatkowo na elementy wspólne oraz te, które wymagają szczególnego potraktowania ze względu na złożoność algorytmu. Rozbudowane algorytmy mogą być pro-

gramowane jako moduły programowe, co pozwala na podział prac programowych i ich współbieżną realizację.

Szczegółowo definiując algorytmy otrzymywania poszczególnych pól tabulogramu przygotowujemy proces programowania tego tabulogramu, czyli następuje przeniesienie części prac planowania programu na etap poprzedzający proces opracowania programu.

IV. Planowanie programu

1. Metody planowania programu

Strukturalne podejście do rozwiązania postawionego problemu zmienia szczegółowy tok prac projektowo-programowych. Podział programu na moduły funkcjonalne, hierarchiczne spojrzenie na cały projektowany system oraz inne wymagania podejścia strukturalnego powodują zmiany w podziale zadań na poszczególnych etapach procesu projektowego oraz wprowadzają nowe elementy w stosowanej metodyce postępowania [1].

Proces programowania omówiony w rozdziale I (pkt. 2) wraz ze szczegółowym tokiem postępowania przedstawionym na rysunku 3 musi zostać zmodyfikowany. Modyfikacja ta polega przede wszystkim na spojrzeniu na system programów jako jedną, spójną całość. Analizując cały system oprogramowania nie tylko inaczej organizujemy pracę nad pojedynczymi programami, wchodzącymi w jego skład, ale określamy elementy wspólne oraz szczegółową strategię wykonania całego systemu. Można powiedzieć, że *projektujemy oprogramowanie*, czyli na sam system oprogramowania należy spojrzeć podobnie jak na cały projektowany system.

Podejście strukturalne do procesu programowania powoduje przeniesienie akcentów z fazy pisania kodu źródłowego programu na fazę projektowania oprogramowania. Bardzo ważnym elementem tej fazy prac jest planowanie programu. Istotą prac planowania jest odpowiednie zorganizowanie procesu programowania tak, aby mógł być opracowany szybko, sprawnie i niezawod-

[1] Por. Z. Szyjewski, *Wpływ rozwoju oprogramowania na projektowanie systemów informacyjnych zarządzania*, Politechnika Szczecińska, Szczecin 1980.

nie. W toku prac planowania następuje szczegółowe poznanie struktury rozwiązywanego problemu, zapisanie go w odpowiednio sformalizowanej postaci, która umożliwia łatwą weryfikację zastosowanych rozwiązań oraz przygotowuje do kodowania.

Można zaobserwować kilka różnych sposobów podejścia do planowania programu.

E. Dijkstra proponuje czysto teoretyczne podejście do planowania polegające na określeniu niezależnych poziomów abstrakcji z rozdzieleniem urządzeń i innych zasobów komputera między wydzielone niezależne poziomy abstrakcji. Proponowana metoda planowania ma niewielkie walory praktyczne, ale dostarcza pełny opis formalny problemu.

L. Constantine i *G. Myers* proponują podobną metodę postępowania z tym, że określają kryteria, według których można dokonać podziału na poziomy abstrakcji. E. Dijkstra proponuje stosowanie metody zstępującej (*top-down*) przy dzieleniu programu na poziomy abstrakcji. L. Constantine i G. Myers zalecają rozszerzenie stosowania tej metody postępowania na wszystkie kroki planowania i kodowania programu. G. Myers proponuje metodę matematycznego dowodzenia poprawności programów przez formułowanie i dowodzenie twierdzeń o programach [2].

D. Parnas określa strukturę programu jako związki między modułami. Planowanie programu ma polegać na minimalizacji związków między modułami. Minimalizacja ta oznacza w praktyce dążenie do przekazywania między modułami jak najmniejszej ilości informacji. Pozwala to na dokonanie zmian w ramach modułu bez wpływu na działanie całego programu. Szczególnie istotnym problemem jest rozdzielenie danych między modułami. Jest zalecane, aby każdy moduł operował na własnych zbiorach danych.

M. Jackson i *J. Warnier* za podstawę planowania programu przyjmują struktury danych, na których program ma pracować. Na podstawie struktury danych określają strukturę logiczną programu. Szczegółowa specyfikacja danych służy do określenia podziału programu na moduły.

Metody Jacksona i Warniera planowania programu ze względu

[2] Por. G. J. Myers, op. cit., rozdz. 17 oraz S. Alagić, M. A. Arbib, *Projektowanie programów poprawnych i dobrze zbudowanych*, Warszawa 1982.

na duże walory praktyczne zdobyły popularność i zostaną szczegółowo omówione dalej. Inne metody planowania mają mniejsze walory praktyczne, szczególnie w przetwarzaniu danych. Spojrzenie na problem planowania programu przez pryzmat danych, na których program ma pracować jest podejściem bardzo praktycznym i stosunkowo łatwym w zastosowaniu.

2. Metoda Jacksona

a. Przykład wprowadzający

Mamy do rozwiązania problem i komputer jako narzędzie:

— problem jest akcją wykonywaną przez program,

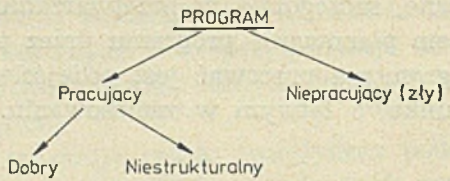
— komputer, to sprzęt i oprogramowanie pomocne przy wykonywaniu pewnego zbioru prymitywnych operacji — wykonanie instrukcji języka programowania.

Zadanie programisty polega na celowym wykorzystaniu tych prymitywnych operacji do rozwiązania problemu. Takie stanowisko prezentuje angielski programista z Cambridge, M. Jackson [3].

W praktyce najczęściej mamy do czynienia z problemami skomplikowanymi, które należy podzielić na mniejsze odcinki tak, aby można było łatwo zastosować zbiór instrukcji języka programowania. Niech program P rozwiązuje problem T. Dzielimy problem T na pewną liczbę podproblemów $T1$, $T2$, $T3$, które korespondują z odpowiednimi składnikami programu $P1$, $P2$, $P3$ tak, że $P1$ realizuje problem $T1$, $P2$ realizuje $T2$ itd. Wykonanie programu P polega na prawidłowym działaniu $P1$, $P2$ i $P3$ oraz ich wzajemnym współdziałaniu. Każdy ze składników programu jest samodzielnym programem i można go dalej dekomponować, jeżeli ułatwi nam to rozwiązanie problemu. Otrzymujemy w wyniku dalszej dekompozycji $P11$, $P12$, $P13$ itd. przy spełnieniu tych samych warunków, jak przy pierwotnym podziale P. Proces podziału kontynuujemy, poziom za poziomem, aż do składników, które są elementami niepodzielnymi i dającymi się prosto rozwiązać i zakodować w języku programowania.

[3] Por. M. Jackson, *Principles of Program Design*, New York 1975, s. 15.

Taką metodę tworzenia programów proponuje M. Jackson, który ponadto każdy program klasyfikuje jako:



Jako podstawowe kryterium podziału wyróżnia funkcjonowanie lub niefunkcjonowanie programu. Programy niepracujące są oczywiście złymi programami. Wśród programów pracujących, czyli wykonujących postawione przed nimi zadania, wyróżnia dwie grupy. Do jednej zalicza programy nie mające przejrzystej strukturalnej konstrukcji. Druga grupa programów pracujących to programy „dobre".

Program „dobry" to więcej niż program pracujący, to ponadto:
— łatwy do zrozumienia, czyli napisany przejrzyście,
— łatwy do utrzymania, konserwacji,
— łatwy do modyfikowania,
— łatwy do wprowadzania uzupełnień i poprawek.

M. Jackson proponuje metodę, która umożliwia pisanie „dobrych" programów. Podstawowym założeniem tej metody jest:

STRUKTURA PROGRAMU ⟺ STRUKTURA PROBLEMU

Zależność tę można określić jako wzajemny związek między strukturą problemu do rozwiązania a strukturą programu rozwiązującego ten problem. M. Jackson wychodzi z założenia, że stosowane powszechnie schematy blokowe, do graficznej prezentacji problemu, nie oddają adekwatnie jego istoty. W miejsce schematów blokowych proponuje:
— metodę z góry na dół (*top-down*),
— metodę kolejnych uściśleń (*stepwise refinement*),
— diagramy hierarchiczne.

M. Jackson, we wstępie do książki opisującej metodę projektowania programu, jako trzy podstawowe zasady programowania strukturalnego, wykorzystywanego w proponowanej metodzie, podaje:

1) problem należy podzielić na hierarchiczną strukturę składników, czemu ma towarzyszyć szczegółowa analiza programu dla

współzależnych struktur i składników; podziału dokonujemy aż do składników elementarnych, które nie są dalej podzielone,

2) na każdym poziomie dekompozycji należy dążyć do wykorzystania trzech podstawowych struktur: sekwencji, powtarzania i wyboru,

3) istrukcji GO TO należy unikać wszędzie tam, gdzie to jest możliwe [4].

Przykład 4

Dla zobrazowania tak rozumianego strukturalnego postępowania przy rozwiązywaniu problemu, rozważmy następujący prosty przykład [5]. Należy wygenerować i wydrukować tablicę o postaci:

```
1
2  4
3  6  9
4  8 12 16
.
.
.
10 20 30 40 ... 100
```

Przeanalizujemy kolejne kroki rozwiązania problemu, które polega na kolejnych uściśleniach. Postępowanie w każdym kolejnym kroku zapiszemy w postaci pseudokodu.

1. Krok definicji problemu na poziomie tablicy:

```
Start
   generowanie tablicy,
stop
```

2. Krok uściślenia definicji do poziomu wiersza tablicy:

```
A = 1
while  A ⩽ 10
do
   generowanie wiersza tablicy
   wydruk wiersza tablicy
   A = A+1
enddo
```

gdzie A — indeks wiersza tablicy.

[4] Por. tamże, s. 1.
[5] Tamże.

3. Krok uściślenia definicji do poziomu elementu wiersza:

```
zerowanie wektora wiersza:
S = 1
while  S ⩽ A
do
  M(S) = A * S
  S = S+1
enddo
```

gdzie S — indeks kolumny tablicy,
M(i) — wektor wiersza tablicy.

Dalsze uściślanie zdefiniowanego programu nie jest wymagane, gdyż osiągnęliśmy ostateczne jego rozwiązanie. Wstawiając kolejno rozwiązania szczegółowe do bardziej ogólnych, otrzymujemy ostateczne rozwiązanie strukturalne:

```
Start
  A = 1
  while  A ⩽ 10
do
  zerowania wektora wiersza
  S = 1
  while  S ⩽ A
do
  M(S) = A * S
  S = S+1
enddo
  wydruk wiersza tablicy
  A = A+1
enddo
stop
```

Zwróćmy uwagę że:

struktura programu jest uzależniona od struktury danych na których program operuje.

Sformułowana zasada stanowi podstawę metody Jacksona planowania programu.

W proponowanej metodzie dla graficznej prezentacji sposobu postępowania wykorzystuje się:

— diagram strukturalny,
— schemat logiczny,

dla opisu
- danych,
- programu.

Przeanalizujemy obecnie, jak struktury podstawowe sekwencji, wyboru i powtarzania są reprezentowane w metodzie dla opisu danych i programu.

b. Sekwencja

O strukturze sekwencji mówimy wtedy, gdy dwie lub więcej części są wykonywane jeden raz, w określonym porządku występowania.

Sekwencja jest naturalnym sposobem wykonywania instrukcji. Porządek wykonywania poszczególnych operacji jest wyznaczany przez zapis w programie. Realizacja programu polegająca na wykonywaniu instrukcji jedna po drugiej w kolejności występowania sprzyja poprawności programu. Dopiero skomplikowane przekazywanie sterowania, zmieniające ten naturalny porządek, powoduje najczęściej błędne funkcjonowanie programu. Podamy kilka przykładów sekwencji dla danych i procedur kodowanych w językach COBOL i PL/1.

```
P1. ADD X TO Y.
    CALL PODP.
    PERFORM PX.
```

Paragraf P1 jest sekwencją składającą się z 3 części: operacji dodawania, podprogramu PODP oraz paragrafu lub sekcji PX. Każdorazowe wykonanie paragrafu P_1 powoduje, że każda z 3 części jest raz wykonana w porządku występowania. Mimo, że w celu wykonania tych instrukcji sterowanie jest przekazywane w inne miejsce programu, to nie burzy to podstawowej zasady struktury sekwencji.

```
01 REKORD-1.
 02 POL1  PIC  X(5).
 02 POL2.
  03 POL21  PIC 999.
  03 POL22  PIC S9(5) COMP SYNC RIGHT.
 02 POL3 PIC X(30).
```

Grupa danych nazwana REKORD-1 jest sekwencją składającą się z 3 części: elementarnych POL1 i POL3 oraz grupy POL2.

Każde z opisanych pól występuje tylko jeden raz w porządku wyznaczonym kolejnością zapisu źródłowego.

```
DO;
  A = 1;
  B = C;
  CALL D;
END;
```

Grupa DO jest sekwencją składającą się z 3 części: podstawienia wartości za A i za B oraz wywołania podprogramu D.

```
DO;
  A = 1;
  B = C;
DO;
  D = E;
  CALL F;
END;
END;
```

Taka grupa DO jest również sekwencją składającą się z 3 części: podstawień do A i B oraz innej grupy DO.

```
ABC: PROC;
   PUT SKIP LIST (A, B, C);
   PUT SKIP DATA (I, J, K);
END ABC;
```

Procedura ABC jest również sekwencją składającą się z dwóch części operacji PUT LIST oraz PUT DATA.

Sekwencja danych w języku PL/1 jest analogiczna, jak struktury deklarowane w języku COBOL.

```
DCL
1 STRUKTURA
, 2 POLE1  CHAR(3)
, 2 POLE2
, 3 POLE21  FIXED BINARY (15)
, 3 POLE22  FIXED DECIMAL (7)
;
```

Z przedstawionych przykładów wynika, w jaki sposób można opisać struktury sekwencji za pomocą różnych jednostek syntaktycznych. Sposób zapisu ma pewne cechy ogólne, a zapis konkretny zależy od języka programowania. M. Jackson zaproponо-

wał notację o bardziej generalnych zasadach, która umożliwia zapisanie dowolnej sekwencji w postaci diagramu. Diagram, dla sekwencji A składającej się z B, C i D, przyjmuje postać:

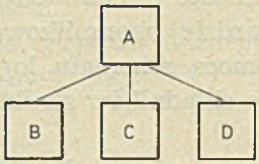
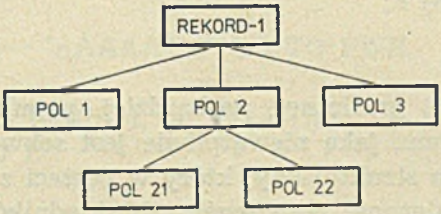


Diagram ten nie jest siecią działań. Linie służą do łączenia dwóch składników, określając jednocześnie ich zależność. W podanym przykładzie B jest częścią A, podobnie jak C i D.

Diagram może zawierać dowolną ilość poziomów w zależności od potrzeb. Zapis REKORD-1 kodowany wcześniej w języku COBOL można przedstawić w postaci następującego diagramu:



Diagramy strukturalne są wykorzystywane zarówno dla opisu programu, jak i struktur danych. Na podstawie diagramu strukturalnego można dla programu zbudować schemat logiczny, który w postaci bardziej zbliżonej do języka programowania przedstawia strukturę programu. M. Jackson proponuje wykorzystać dla tego celu pseudokodowanie. Schemat logiczny dla paragrafu P1 kodowanego w języku COBOL przedstawiamy w postaci:

```
P1  seq
        y := y+x;
        wykonaj PODP;
        wykonaj PX;
P1  end
```

Dla zapisu składników sekwencji w postaci schematu logicznego wykorzystywane są trzy formy. Dwie z nich są przedstawione w tym przykładzie. Pierwsza jest wykorzystywana do opisu elementarnych operacji i polega na stosowaniu zapisu w po-

staci instrukcji języka programowania (ADD X TO Y) lub w innej formie skrótowej (Y := Y+X). Druga służy do zapisu nieelementarnych składników, które należałoby opisać słownie „wykonaj następujące czynności o nazwach PODP i PX". Trzecia forma służy do opisu bardziej skomplikowanych składników, które są opisywane za pomocą schematu logicznego. Sytuacja taka występuje, gdy jeden ze składników struktury jest sam strukturą sekwencyjną.

Przedstawia to schemat logiczny dla drugiej grupy DO kodowanej w języku PL/1.

```
D1 seq
     A := 1;
     B := C;
     D2 seq
          D := E;
       wykonaj F;
     D2 end
D1 end
```

Dla prezentacji graficznej, najbardziej popularnej struktury danych i programu, jaką niewątpliwie jest sekwencja, wykorzystuje się diagram strukturalny, który w postaci zbliżonej do grafu prezentuje zależność poszczególnych składników. Schemat logiczny, wykorzystując pseudokodowanie, umożliwia bardzo czytelną prezentację struktury sekwencji, dając wygodny zapis do kodowania w dowolnym języku programowania.

c. Iteracja

O strukturze iteracji mówimy wówczas, gdy jedna część jest powtarzana zero jeden lub więcej niż jeden raz. Cykliczne powtarzanie pewnej partii instrukcji programowych jest znanym i powszechnie stosowanym sposobem konstruowania programu. Instrukcje programowe cyklicznie powtarzane tworzą tzw. pętlę programową. Ten sposób programowania umożliwia realizację takich samych algorytmów dla innego zestawu danych. Dane są dostarczane dla każdego cyklu oddzielnymi instrukcjami lub są zgrupowane w tablicy, która jest przykładem struktury powtarzania (iteracji) dla danych. Pojedyncze pole lub grupa pól jest

powtarzana wielokrotnie w tablicy. Każdy z powtarzanych elementów ma jednak takie same atrybuty. Kilka przykładów dla danych i procedur kodowanych w językach COBOL i PL/1.

```
PERFORM P1 7 TIMES.
```

Zdanie to jest iteracją, gdyż dla każdego jego wystąpienia, paragraf P1 jest powtarzany 7 razy.

```
01 TABLICA.
 02 TABL OCCURS 50.
  03 EL1 PIC X(7).
  03 EL2 PIC 9(4).
```

Grupa danych nazwana TABLICA jest iteracją: powtarzana jest 50 razy część nazwana TABL, dla każdego wystąpienia TABLICA.

Rozważmy sekcję, która przetwarza opisaną grupę danych.

```
WYD-TAB SECTION.
PW1.
     IF EL1 (I) = 'AAAAAAA' GO TO PWK.
PW2.
     DISPLAY ('TABL', I, '=', EL1 (I)).
     ADD 1 TO I.
PW3.
     GO TO PW1.
PWK.
     EXIT.
```

Sekcja WYD-TAB jest iteracją, gdzie częścią powtarzaną jest paragraf PW2. Paragraf ten może być wykonany zero lub wiele razy w zależności od wartości EL1 opisanej grupy danych. Każdorazowe wykonanie sekcji WYD-TABL spowoduje powtarzanie paragrafu PW2, dopóki element tablicy EL1 nie przyjmie wartości AAAAAAA. Opisany ciąg instrukcji można zakodować inaczej za pomocą instrukcji PERFORM.

```
PERFORM PW2 UNTIL EL1 (I) = 'AAAAAAA'.
```

Paragraf PW2 będzie wykonany zero razy, gdy EL1 (I) przyjmuje wartość AAAAAAA dla początkowej wartości I.

```
DO I = 1 TO 10;
    X(I) = X(I)+1;
    Y(I) = Y(I) * Y(I);
END;
```

Grupa DO jest iteracją, gdzie część powtarzana nie ma nazwy, a jest to sekwencja składająca się z dwóch instrukcji: nadania wartości X(I) oraz Y(I). Powtarzana część jest wykonana 10 razy dla każdego wystąpienia grupy DO.

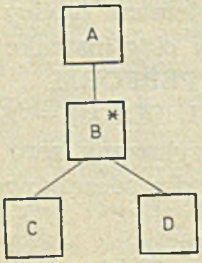
```
DCL
  1 A
, 2 B (10)
, 3 C CHAR (5)
, 3 D FIXED BINARY
;
```

Struktura A jest iteracją, gdzie częścią powtarzaną jest B, która jest powtarzana 10 razy dla każdego wystąpienia. Część powtarzana składa się z sekwencji dwóch elementów C i D.

Dla graficznej prezentacji iteracji M. Jackson proponuje diagram, w którym część powtarzana jest oznaczona gwiazdką.

Gwiazdka zapisana w prawym górnym rogu prostokąta zawierającego B oznacza wielokrotne wystąpienie B w ramach A. Iteracją jest A i wielokrotne wystąpienie B jest atrybutem A.

Dla opisanej w języku PL/1 struktury A mamy następujący diagram graficzny:



Schemat logiczny dla iteracji X, gdzie powtarzaną częścią jest Y, mamy:

X *iter*
wykonaj Y
X *end*

Oczywiście część powtarzana może być opisana w jednej z trzech wcześniej podanych formach zapisu dla sekwencji. Rozszerzając opis słowny dla iteracji, można wzbogacić go o specyfikację liczby powtórzeń dla powtarzanej części. Dla przykładu schemat logiczny dla iteracji zawartej w sekcji WYD-TAB można opisać:

```
WYD-TAB iter until EL1 (I) = 'AAAAAAA'
        PW2 seg
            drukowanie tablicy
            I := I+1;
        PW2 end
WYD-TAB end
```

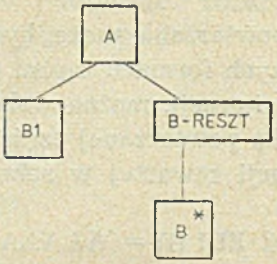
W przykładzie tym wyspecyfikowaliśmy warunek, dla którego iteracja zostanie zakończona. Należy zwrócić uwagę, że częścią powtarzaną jest struktura sekwencji. Warunek można wyspecyfikować w innej postaci:

```
WYD-TAB iter while EL1 (I) ≠ 'AAAAAAA'
        PW2 seg
            drukowanie tablicy
            I := I+1;
        PW2 end
WYD-TAB end
```

Przykłady różnego zapisu warunku przetwarzania dla iteracji, umożliwiają odróżnienie iteracji typu WHILE, od typu UNTIL, opisanych w rozdziale III (pkt 2). Iteracja typu WHILE jest dokładnie przedstawiona w rozbudowanej postaci opisu sekcji WYD-TAB. Z zapisu instrukcji wynika, że badanie warunku jest wykonywane przed każdą potencjalną realizacją paragrafu PW2.

Zgodnie z definicją iteracji, klasycznym typem iteracji jest iteracja typu WHILE. Użycie iteracji typu UNTIL winno być celowe i stosowane w specjalnych sytuacjach, gdy jest wymagane przynajmniej jedno wystąpienie powtarzanej części. W celu wyraźnego podkreślenia tego faktu, proponuje się pokazanie pierwszego wystąpienia na diagramie oddzielnie. Dla przykładu: gdy A składa się z pewnej liczby B takich, że pierwsze B musi

być przetworzone, to jest wskazane przedstawienie tego faktu wyraźnie na diagramie.



Takie przedstawienie iteracji A sugeruje specyficzne przetwarzanie pierwszego wystąpienia B. Otrzymujemy sekwencje: B1 i B-RESZT, gdzie B1 jest składnikiem elementarnym, natomiast B-RESZT jest częścią powtarzaną. Dodajmy, że każda iteracja może być rozwinięta w sekwencję przez rozbicie części powtarzanej na odpowiednio wiele pojedynczych składników.

Dla przykładu, tablicę można zadeklarować dwoma sposobami.

```
01 TABLICA.
   02 EL PIC 99 OCCURS 3.
```

lub

```
01 TABLICA.
   02 EL1 PIC 99.
   02 EL2 PIC 99.
   02 EL3 PIC 99.
```

Pierwsza deklaracja jest typową iteracją, gdzie część powtarzana EL jest powtarzana 3 razy. Druga deklaracja w wyniku daje to samo, z tym że elementy EL1, EL2, EL3 składają się na sekwencję trzech części.

Analogicznie instrukcje, w których argumentami są tablice opisane dwoma sposobami, stanowią iterację lub sekwencję.

```
PAR1.
    MOVE ZERO TO EL(I).
PAR2.
    PERFORM PAR1 VARYING I FROM 1 BY 1 UNTIL I≤3.
```

Tak zakodowany paragraf PAR2 jest iteracją, gdzie część powtarzana, paragraf PAR1 jest powtórzony 3 razy. Rozwiniemy go w sekwencję:

```
PAR1.
        MOVE ZERO TO EL1.
        MOVE ZERO TO EL2.
        MOVE ZERO TO EL3.
```

Tak zakodowany paragraf PAR1 jest sekwencją składającą się z trzech części. Wykonanie obydwu postaci jest równoważne, gdyż w obu wypadkach powoduje wyzerowanie elementów tablicy EL1.

Z przykładów tych wynika, w jaki sposób iteracje można rozwinąć w sekwencję. Działanie takie ma jednak pewne ograniczenia. Jedną z zalet iteracji jest krótki i zwięzły sposób kodowania części powtarzanych wielokrotnie. Rozwijając iteracje w sekwencję tracimy ten walor. Ekonomiczność stosowania iteracji wzrasta wraz ze wzrostem liczby powtórzeń. W rozważanym przykładzie, gdzie część powtarzana była trzykrotnie, ekonomiczność stosowania iteracji nie jest wyraźnie widoczna, ale w wypadku gdy część powtarzana byłaby przykładowo 100 razy, uwidacznia to się już wyraźnie.

Inną cechą charakterystyczną iteracji jest możliwość warunkowego określenia krotności powtórzeń, co jest niemożliwe przy stosowaniu rozwinięcia w sekwencję, gdyż wymagałoby to badania warunku po wykonaniu każdego składnika sekwencji.

Deklaracja tablicy:

```
DCL TABLICA (N) FIXED BINARY;
```

oraz zwiększanie każdego elementu o 1

```
DO I = 1 TO N;
    TABLICA (I) = TABLICA (I)+1;
END;
```

gdzie ilość elementów tablicy nie jest w czasie kodowania znana dokładnie, nie może być rozwinięta w sekwencję z powodu braku informacji szczegółowej o liczbie powtórzeń składających się na opisaną iterację.

W rozważanych przykładach jednym ze składników iteracji była struktura sekwencji. W zastosowaniach praktycznych iteracja może być traktowana jako składnik sekwencji. Budując programy, struktury te można w całości zagnieżdżać jedna w drugiej.

d. Selekcja

O strukturze selekcji mówimy wówczas, gdy występują dwie lub więcej części, z których jedna i tylko jedna wykonana jest raz dla każdego wystąpienia selekcji[6]. Wykonanie algorytmu realizowanego przez program bardzo często wymaga wyboru jednego z możliwych wariantów. Badanie stanu warunku lub wyrażenia warunkowego i w zależności od wyniku tego badania, wykonanie jednego wariantu lub innego jest sytuacją bardzo często spotykaną. Najczęściej zdarza się sytuacja, gdzie następuje realizacja jednego odgałęzienia programu przy spełnieniu badanego warunku, a drugiego — w przeciwnym wypadku. Wybrana część programu jest realizowana jednorazowo przy danym wykonaniu instrukcji, z równoczesnym pominięciem alternatywnej części programu. Podamy kilka przykładów kodowanych w językach COBOL i PL/1 dla opisu danych i kodowania procedur.

```
IF X = 0 MOVE A TO B.
         ELSE MOVE C TO D.
```

Zdanie IF jest selekcją, której częściami są dwa zdania MOVE. Każde wykonanie zdania IF polega na wykonaniu dokładnie raz i tylko jednego z dwóch zdań MOVE. Wybór zdania do wykonania uzależniony jest od wartości X; gdy X przyjmuje wartość 0, to wykonuje się pierwsze zdanie, gdy przyjmuje inną wartość — to drugie zdanie.

```
01 KARTA.
 02 KOD PIC X.
 02 KREDYT.
  03 NUMER-R PIC 9 (6).
  03 NUMER-K PIC 9 (7).
 03 WARTOŚĆ PIC 9 (6).
 03 NAZWA PIC X (22).
02 DEBET REDEFINES KREDYT.
 03 KLIENT PIC 9 (7).
```

[6] Wcześniej w rozdziale III pkt. 2.2 dla tego typu struktury używaliśmy określenia *struktura wyboru*, teraz dla jej określenia będziemy używać synonimu *selekcja*. Podyktowane to jest zgodnością z terminami przyjętymi przez M. Jacksona w schemacie logicznym (*select*). Nie jest to więc żadna nowa struktura, podobnie jak w rozdziale IV (pkt c.) dla struktury powtarzania używaliśmy synonimu *iteracja*.

```
03 WARTOŚĆ-D PIC 9 (8).
03 NAZWA-K PIC X (25).
02 TRANSFER REDEFINES DEBET.
03 KOD-TRAN PIC 99.
03 KLIENT1 PIC 9 (7).
03 KLIENT2 PIC 9 (7).
03 FILLER PIC X (24).
```

Trzy grupy pól o nazwach KREDYT, DEBET i TRANSFER są częściami selekcji. Grupa KARTA składa się z sekwencji KOD i wymienionych składników selekcji. Dla każdego wystąpienia danych KARTA będzie dokładnie jedno wystąpienie z KREDYT lub DEBET lub TRANSFER

```
IF WSK = 1 THEN PUT DATA (A);
           ELSE DO;
                  A = 0;
                  WSK = 1;
                END;
```

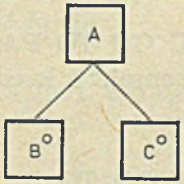
Tak zakodowana instrukcja IF jest selekcją, gdzie jedną częścią jest instrukcja PUT DATA, a drugą grupa DO, składająca się z dwóch zdań.

```
IF WSK = 0 THEN;
           ELSE CALL A;
```

Tak zapisana instrukcja IF jest również selekcją, gdzie jedną częścią jest zdanie puste, a drugą — zdanie wywołujące procedurę A.

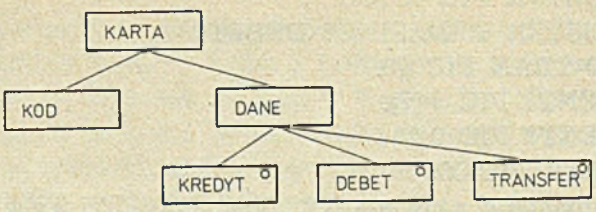
Selekcja, w której jedna z części jest zdaniem pustym, jest wypadkiem podobnym do sytuacji iteracji dla zera powtórzeń. Rozwiązanie takie stosuje się w wypadkach szczególnych.

Selekcję A, w skład której wchodzą części B i C, reprezentuje diagram:



Kółka w prostokątach B i C oznaczają, że A jest selekcją, a B i C wchodzącymi w jej skład czynnościami.

Diagram dla struktury KARTA opisanej wcześniej ma postać:



Schemat logiczny dla selekcji A składającej się z B i C zapisujemy:

A *select*
wykonaj B
A *or*
wykonaj C
A *end*

W schemacie logicznym można dodatkowo zapisać warunek, który określa kryteria wyboru części do wykonania. Wracając do opisanego przykładu struktury KARTA i przyjmując, że pole KOD może przyjąć wartości K, D i T odpowiednio dla części KREDYT, DEBET i TRANSFER, to schemat logiczny można zapisać następująco:

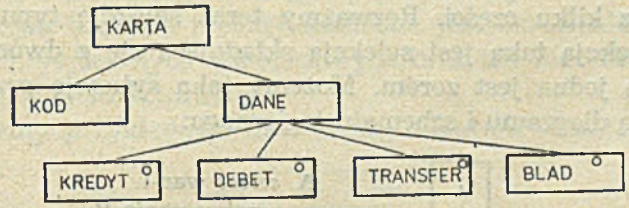
```
KARTA select KOD = 'K'
                    przetwarzanie KREDYT
KARTA or  KOD = 'D'
                    przetwarzanie DEBET
KARTA or  KOD = 'T'
                    przetwarzanie TRANSFER
KARTA end
```

Kodowanie w języku COBOL zapisanego schematu logicznego:

```
KARTA SECTION.
PAR1.
    IF KOD NOT = 'K' GO TO PAR2
      przetwarzanie transakcji kredyt
    GO TO PKON.
PAR2.
    IF KOD NOT = 'D' GO TO PAR3.
      przetwarzanie transakcji debet
    GO TO PKON.
```

```
PAR3.
        przetwarzanie transakcji transfer
PKON.
        EXIT.
```

Z szczegółowej analizy schematu logicznego oraz przedstawionego kodu w języku COBOL wynikają drobne rozbieżności. W schemacie logicznym dla każdej części selekcji jest zapisany w zdaniu warunek wyboru tej części. W programie zapisanym w języku COBOL kodowane są jedynie warunki dla dwóch części, dla pozostałych możliwości brak zapisu warunku. Rozwiązanie takie nie jest zalecane, gdyż może być źródłem poważnych błędów. Przyjmując, że wśród danych wprowadzanych nie wystąpią dane z kodem innym niż K, D i T, jest założeniem, które eliminuje możliwość wykrywania błędów i nie powinno być zalecane. Rozwiązanie zalecane przedstawia następujący diagram oraz wywodzący się z niego schemat logiczny.




```
KARTA select  KOD = 'K'
              przetwarzanie KREDYT
KARTA or      KOD = 'D'
              przetwarzanie DEBET
KARTA or      KOD = 'T'
              przetwarzanie TRANSFER
KARTA or      KOD ≠ 'K', ≠ 'D', ≠ 'T'
              przetwarzanie błędu
KARTA end
```

Kod źródłowy w języku COBOL przyjmuje w tym wypadku postać:

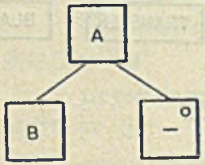
```
KARTA SECTION.
PAR1.
        IF KOD NOT = 'K' GO TO PAR2.
                przetwarzanie transakcji kredyt
```

```
        GO TO PKON
PAR2.
        IF KOD NOT = 'D' GO TO PAR3.
            przetwarzanie transakcji debet
        GO TO PKON.
PAR3.
        IF KOD NOT = 'T' GO TO PAR4.
         przetwarzanie transakcji transfer
        GO TO PKON.
PAR4.
        przetwarzanie błędu
PKON.
        EXIT.
```

Dla pełnej zgodności ze schematem logicznym należałoby w paragrafie PAR4 zakodować jako pierwsze zdanie:

IF KOD = 'K' OR KOD = 'D' OR KOD = 'T' GO TO PKON.

W rozważanym przykładzie zajmowaliśmy się selekcją składającą się z kilku części. Rozważmy teraz selekcję typu specjalnego. Selekcją taką jest selekcja składająca się z dwóch części, z których jedna jest zerem. Możemy taką sytuację przedstawić za pomocą diagramu i schematu logicznego:




```
A select war-1
    przetwarzanie B
A or war-2
    nic nie rób
A end
```

Sytuację specjalnego typu można przedstawić krócej w postaci:

```
A select war-1
    przetwarzanie B
A end
```

Kodowanie specjalnego typu selekcji polega na pominięciu kodowania części ELSE w instrukcji IF lub kodowania pomijanej części jako zdanie puste.

Przedstawione struktury sekwencji, iteracji i selekcji — o czym już mówiliśmy — wystarczają dla skonstruowania każdego pro-

gramu. Ich stosowanie polega na odpowiednim łączeniu wymienionych struktur w program. Łączenie to powinno polegać na zagnieżdżaniu jednej struktury w drugiej w całości jako odpowiedniej części struktury zewnętrznej. Przykładowo, selekcja może jako jedną część zawierać iterację, a jako drugą — sekwencję, przy czym te z kolei mogą dalej składać się ze składników, którymi mogą być wszystkie z trzech przedstawionych struktur.

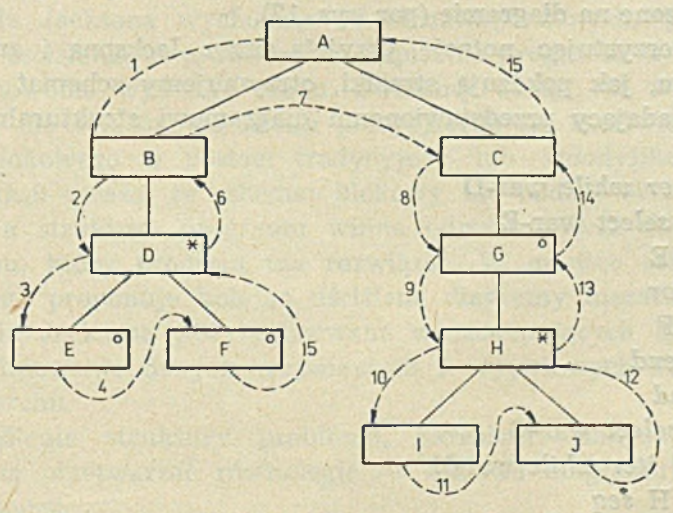
Przedstawione dla metody Jacksona diagramy strukturalne oraz schematy logiczne umożliwiają takie całościowe, hierarchiczne zagnieżdżenie struktur.

e. Zasady postępowania

Metoda Jacksona planowania programu ma na celu wygenerowanie strukturalnego kodu programu. W tym celu realizowane są kolejne kroki postępowania, w których wykorzystywane są tylko trzy podstawowe struktury. Operując diagramem strukturalnym oraz schematem logicznym osiągamy postawiony cel.

Pierwszym krokiem metody Jacksona jest określenie struktury danych wejściowych i wyjściowych oraz ich hierarchii w postaci diagramu strukturalnego.

Rozważmy przykładowo następujący diagram strukturalny.



Rys. 17. Analiza diagramu strukturalnego

W przedstawionym diagramie (por. rys. 17), zgodnie z przyjętą notacją Jacksona, prostokąty oznaczone w prawym górnym rogu znakiem gwiazdki, reprezentują strukturę powtarzania, a oznaczone znakiem kółka reprezentują strukturę wyboru. Na podstawie przedstawionego diagramu strukturalnego można zbudować odpowiadający mu schemat logiczny, stosując w odpowiedni sposób opis z lewej strony na prawo.

Zasada zamiany diagramu strukturalnego na schemat logiczny jest zaczerpnięta ze sposobów przechodzenia przez wierzchołki (węzły) struktur dendrytowych, wykorzystywanych przy translacji wyrażeń arytmetycznych zapisywanych w notacji polskiej. Diagram strukturalny traktujemy jak strukturę dendrytową, gdzie prostokąty spełniają rolę węzłów. Przy przetwarzaniu danych zapisanych w postaci struktury dendrytowej istotny jest porządek (kolejność) przechodzenia przez węzły takiej struktury. Kolejność przetwarzania węzłów jest dlatego sprawą istotną, ponieważ sama struktura jest nośnikiem informacji, oprócz informacji szczegółowej zapisanej w każdym z węzłów.

Schemat logiczny otrzymujemy z diagramu strukturalnego rozpoczynając analizę diagramu od wierzchołka i zapisując od strony lewej do prawej kolejno napotkane symbole. Graficznie sposób analizy przedstawiają ponumerowane kolejno przerywane linie zaznaczone na diagramie (por. rys. 17).

Wykorzystując notację przyjętą przez Jacksona i analizując diagram, jak pokazują strzałki, otrzymujemy schemat logiczny odpowiadający przedstawionemu diagramowi strukturalnemu:

```
A seq
  B iter while war-D
    D select war-E
        E
    D or
        F
    D end
  B end
    C select war-G
      G iter while war-H
        H seq
          I
```

J
H *end*
G *end*
C *end*
A *end*

Schemat logiczny jest programem napisanym w pseudokodzie z uwzględnieniem trzech podstawowych struktur. Zamiana tak zapisanego programu na wybrany język programowania wyższego rzędu jest już czynnością stosunkowo prostą.

Program w pseudokodzie, który powstaje w wyniku opisania diagramu strukturalnego, jest szkieletowym programem zawierającym wszystkie istotne operacje logiczne do wykonania programu. Tak powstały program wymaga jedynie uzupełnienia przez operacje, które należy wykonać w celu pełnego rozwiązania problemu, chociaż nie ulega zmianie struktura logiczna otrzymanego programu. Wstawiane sekwencje operacji mogą być dość liczne, ale niezależnie od ich rozmiaru nie mogą wpływać na przebieg sterowania w programie, które jest całkowicie określone w programie szkieletowym. Określenie operacji koniecznych do wykonania dla realizacji algorytmu programowego oraz ich alokacja na diagramie hierarchicznym, to kolejne kroki omawianej metody.

Metoda Jacksona wychodząc od struktury danych, poprzez kolejne uściślenia, doprowadza do napisania programu w języku programowania wyższego rzędu. Zwróćmy uwagę, że w metodzie tej nie wyróżniono kroku polegającego na budowie schematu blokowego w postaci tradycyjnej lub zmodyfikowanej. M. Jackson uważa, że schemat blokowy nie oddaje istoty problemu, a struktura programu winna odwzorowywać strukturę problemu, który program ma rozwiązać. W miejsce schematu blokowego proponuje kolejno uściślane diagramy hierarchiczne.

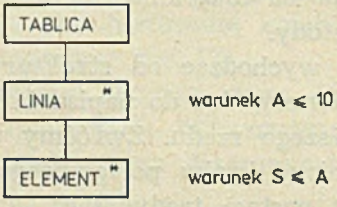
Metoda Jacksona jest realizowana w następujących krokach.

1. Zdefiniowanie danych wejściowych i wyjściowych oraz ich hierarchii.
2. Określenie struktury problemu, określenie danych, które można przetwarzać równolegle — budowa diagramu strukturalnego.
3. Specyfikacja operacji do wykonania.

4. Alokacja operacji do struktury programu.
5. Zamiana diagramu strukturalnego na program w pseudokodzie.
6. Kodowanie w wybranym języku programowania.

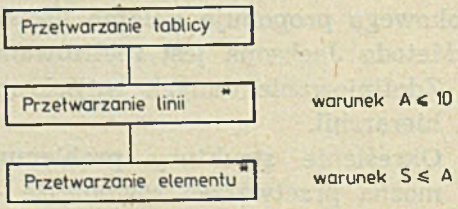
Przeanalizujemy kolejne kroki postępowania metody Jacksona przy rozwiązywaniu problemu zdefiniowanego i rozwiązanego wcześniej, a polegającego na generowaniu i wydrukowaniu tablicy z przykładu 4.

Pierwszy krok metody. Definiuje się dane, które program będzie przetwarzał oraz określa się ich hierarchię. Jest to przedstawione w postaci diagramu. Wyróżniono trzy poziomy danych. Pierwszy poziom najwyższy, to cała tablica, którą należy wygenerować i wydrukować. Poziomem niższym, który jest powtarzany w ramach całej tablicy, jest pojedynczy wiersz tablicy. Zgodnie z założeniem liczba wierszy nie przekracza 10. W ramach każdego wiersza można wyróżnić niższy poziom danych — element. Elementy będą się również powtarzały w ramach każdego wiersza, przy czym liczba ich jest nie większa niż numer wiersza. Używając gwiazdki na oznaczenie faktu powtarzania otrzymujemy diagram prezentujący strukturę danych.



Drugi krok metody. Struktura programu rozwiązującego postawiony problem jest analogiczna do struktury danych i można ją przedstawić w postaci następującego diagramu:

Struktura programu:



Przetwarzanie na poziomie wiersza jest powtarzane dopóty, dopóki liczba wierszy nie przekroczy 10. Natomiast przetwarza-

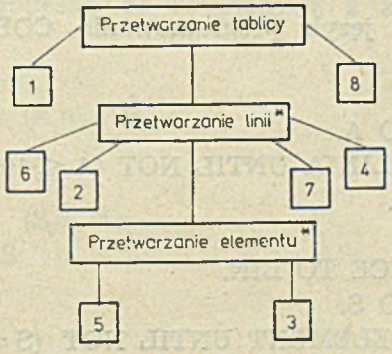
nie kolejnych elementów jest przetwarzane dopóty, dopóki liczba elementów nie jest większa od numeru wiersza. Struktura programu składa się z zagnieżdżonych w sobie struktur iteracji.

Trzeci krok metody. Specyfikujemy operacje, jakie należy wykonać w celu rozwiązania całego problemu. Wypisujemy je w dowolnej kolejności, numerując każdą operację.

Przyjmując: A jako indeks wiersza,
S jako indeks kolumny,
M(i) wektor wiersza tablicy
otrzymujemy następującą listę operacji:

1 A = 1
2 S = 1
3 S = S + 1
4 A = A + 1
5 M(S) = A * S
6 zerowanie wektora linii
7 drukowanie linii
8 stop

Czwarty krok metody. Wymienione operacje należy umieścić w odpowiednim miejscu diagramu przedstawiającego strukturę programu (por. s. 102).



Analizując problem, który ma być rozwiązany przez program, umieszczamy operacje z listy na kolejnych poziomach, wyznaczonych przy określaniu struktury programu. Poszczególne operacje mogą być wykorzystane jednorazowo, w ściśle określonym miejscu struktury programu, lub mogą być wykorzystane wielokrotnie na różnych poziomach struktury programu.

Piąty krok metody. Na podstawie diagramu określającego strukturę programu (z kroku drugiego metody) piszemy program w pseudokodzie, uwzględniając operacje z listy zgodnie z miejscem ich umieszczenia na diagramie. W toku pseudokodowania wykorzystujemy opisany wcześniej sposób analizy diagramu opisującego strukturę programu.

```
przetwarzanie — tablicy seq
                A = 1
              przetwarzanie — wiersza iter while A ⩽ 10
                zerowanie wektora wiersza
                S = 1
                przetwarzanie — elementu iter while S ⩽ A
                   M(S) = A * S
                   S = S+1
                przetwarzanie — elementu end
                drukowanie wiersza
                A = A+1
              przetwarzanie — wiersza end
              stop
przetwarzanie — tablicy end
```

Szósty krok metody. Kodowanie w wybranym języku programowania.

Wykorzystując język programowania COBOL otrzymujemy rozwiązanie:

```
TABLICA.
    MOVE 1 TO A.
    PERFORM LINIA UNTIL NOT (A ⩽ 10).
    STOP RUN.
LINIA.
    MOVE SPACE TO LIN.
    MOVE 1 TO S.
    PERFORM ELEMENT UNTIL NOT (S ⩽ A).
```

```
    DISPLAY (LIN).
    ADD 1 TO A.
ELEMENT.
    MULTIPLY A BY S GIVING M(S).
    ADD 1 TO S.
```

W przykładzie brakuje opisu danych, ale łatwo zauważyć, że elementy tablicy M wchodzą w skład drukowanej linii o nazwie programowej LIN.

Wykorzystując inną postać instrukcji PERFORM rozwiązanie to można przedstawić w postaci krótszej.

```
TABLICA.
        PERFORM LINIA VARYING A FROM 1 BY 1 UNTIL
                                        NOT (A ⩽ 10)
        STOP RUN.
LINIA.
        MOVE SPACE TO LIN.
        PERFORM ELEMENT VARYING S FROM 1 BY 1
                                UNTIL NOT (S ⩽ A).
        DISPLAY (LIN).
ELEMENT.
        MULTIPLY A BY S GIVING M(S).
```

Ze względu na dostępny aparat instrukcji języka COBOL, rozwiązanie to jest stosunkowo mało zbliżone do pseudokodu wytworzonego w piątym kroku metody. Ciąg logiczny programu odpowiada jednak dokładnie kolejnym uściśleniom rozwiązania, jakie proponuje autor metody. Wykorzystując w celu napisania programu język programowania PL/1 otrzymujemy kod źródłowy bardzo zbliżony do rozwiązania z kroku piątego.

```
TABLICA.
    A = 1;
    DO WHILE (A ⩽ 10);
      LIN = ' ';
      S = 1;
    DO WHILE (S ⩽ A);
    M(S) = A * S;
    S = S+1;
    END;
```

```
    PUT SKIP LIST (LIN);
    A = A+1;
    END;
  END TABLICA;
```

Analogicznie, jak w wypadku języka COBOL, wykorzystanie innej postaci instrukcji DO skraca rozwiązanie:

```
  TABLICA:
  DO A = 1 TO 10;
    LIN = ' ';
    DO S = 1 TO A;
        M(S) = A * S;
    END;
    PUT SKIP LIST (LIN);
  END;
END TABLICA;
```

Na podstawie przeprowadzonych rozważań, mających na celu rozwiązanie konkretnego chociaż bardzo prostego problemu, poznaliśmy podstawowe zasady postępowania przy korzystaniu z metody Jacksona. Wychodząc od struktury danych poprzez kolejne uściślanie, z wykorzystaniem jedynie trzech podstawowych struktur, dochodzimy do strukturalnego kodu źródłowego w wybranym języku programowania. Przy stosowaniu metody dla rozwiązania bardziej skomplikowanych problemów wystąpią nowe trudności, ale krokowy sposób postępowania przyczynia się do osiągnięcia dobrego strukturalnego rozwiązania.

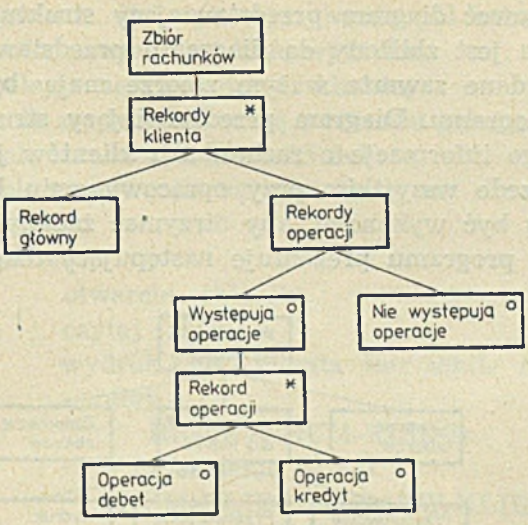
f. Złożone przykłady stosowania metody

W rozwiązywanym według metody Jacksona problemie wykorzystaliśmy tylko strukturę sekwencji i powtarzania. Rozważmy teraz bardziej skomplikowany problem, w celu rozwiązania którego potrzebne będą wszystkie dopuszczalne struktury.

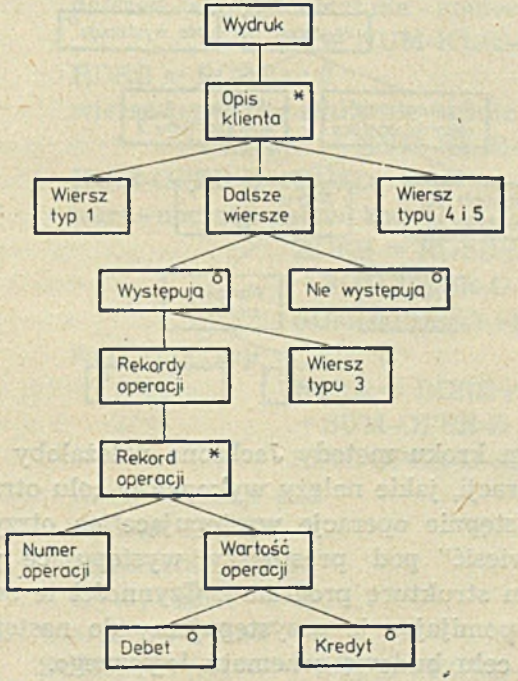
Rozwiążmy za pomocą metody Jacksona problem wydruku listy opisanej w przykładzie 3 (por. rozdz. III, pkt. 4).

Dane występujące w omawianym problemie zgrupowane są w dwóch zbiorach, których strukturę przedstawiają diagramy.

Struktura danych dla zbioru rachunków:

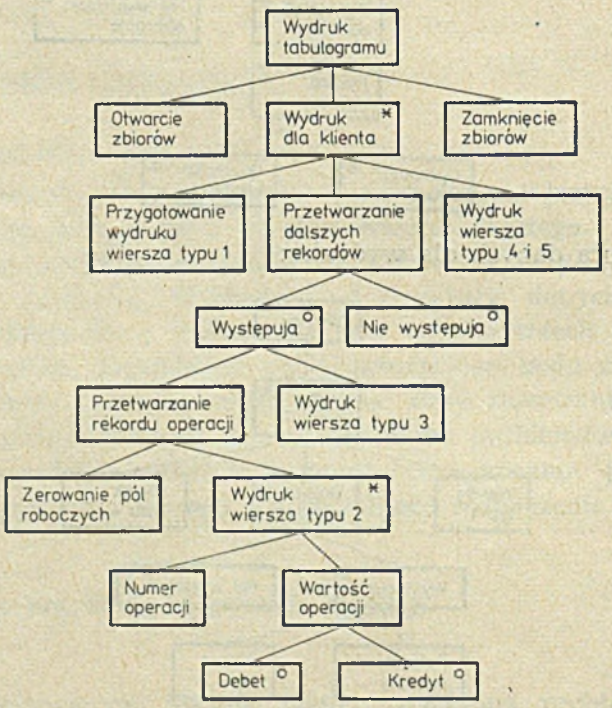


Struktura danych dla wydruku:



Na podstawie schematów przedstawiających strukturę danych można zbudować diagram przedstawiający strukturę programu. Diagram ten jest zbliżony do diagramu przedstawiającego wydruk, gdyż dane zawarte w tym zbiorze mają być wynikiem działania programu. Diagram przedstawiający strukturę zbioru zawierającego informacje o rachunkach klientów jest wykorzystywany przede wszystkim przy opracowywaniu listy operacji, które muszą być wykonane, aby otrzymać żądany wydruk.

Strukturę programu prezentuje następujący diagram:



W kolejnym kroku metody Jacksona należałoby wyspecyfikować listę operacji, jakie należy wykonać w celu otrzymania tego wydruku. Następnie operacje występujące na otrzymanej liście należy „podwiesić" pod prostokąty występujące w diagramie, prezentującym strukturę programu. Czynności te są stosunkowo proste, więc pomijając je przystępujemy do następnego kroku, mającego na celu budowę schematu logicznego.

W celu zwięzłego przedstawienia schematu przyjmujemy następujące oznaczenia dla wykorzystywania pól roboczych:

BUFD — bufor zawierający linię wydruku,
RNUM — pole zawierające numer klienta,
RSUM — pole zawierające sumę z rekordu głównego,
RDEB — pole sumowania wartości debetu,
RCRE — pole sumowania wartości kredytu.

Schemat logiczny:

```
Wydruk - tab seq
                otwarcie - zbiorów
                czytaj - zbiór
                wydruk - dla - klienta iter while nie - koniec -
                - zbioru
                        RNUM = NUM-KLIE-D
                        czyszczenie BUFD
                        NUM-KLIE-D i NAZW-KLIE-D→BUFD
                        RSUM = SUMA-OG-D
                        czytaj - zbiór
                dalsze - rekordy select nie - koniec - zbioru
                        i RNUM = NUM-KLIE-D
                RDEB = RCRE = 0
                wiersz-typ-2 iter while nie - koniec - zbioru
                        i RNUM = NUM-KLIE-D
                NUM-OPER-D→BUFD
                wart - operacji select kod D
                                RDEB = RDEB+
                                +SUM-OPER-D
                                SUM-OPER-D→BUFD
                wart - operacji or kod C
                                RCRE = RCRE+
                                +SUM-OPER-D
                                SUM-OPER-D→BUFD
                wart - operacji end
                wydruk BUFD
                czyszczenie BUFD
                czytaj - zbiór
        wiersz-typ-2 end
        SUMA-NOWA-W = RSUM+RCRE—RDEB
```

```
                RSUM, SUMA-NOWA-W, RCRE, RDEB→BUFD
            dalsze - rekordy or
                wydruk BUFD
                czyszczenie BUFD
                SUMA-NOWA-W = RSUM
            dalsze — rekordy end
            RSUM, SUMA-NOWA-W→BUFD
            wydruk BUFD
        wydruk - dla klienta end
        zamknij zbiory
        stop
    wydruk - tab end
```

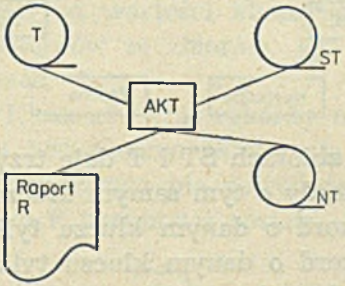
Zamiana przedstawionego schematu logicznego na kod źródłowy w języku programowania wyższego rzędu jest czynnością nie powodującą trudności. Jest to zarazem ostatni krok metody Jacksona.

Omówiony przykład przedstawia wykorzystanie wszystkich struktur programowania. Powtarzane są operacje wykonywane dla obsługi każdego kolejnego klienta aż do wyczerpania zbioru. Powtarzane są również operacje wykonywane dla wydruku wierszy typu 2, w ramach wykonywanych operacji dla jednego klienta. Dzięki takiemu zagnieżdżaniu struktur powtarzania, w programie wynikowym otrzymujemy pętlę zewnętrzną i zagnieżdżoną w niej pętlę wewnętrzną. W związku z odpowiednim ułożeniem diagramu, prezentującego strukturę danych, kodowanie zawierających się pętli jest proste i nie prowadzi do powstawania błędów logicznych w programie. Poprawność logiczna powstałego programu jest również zagwarantowana kompletnym rozważaniem sytuacji decyzyjnych. Oddzielnie rozważymy sytuację, w jakiej występują dalsze rekordy opisujące rachunek klienta (pień drzewa obrazującego strukturę danych) i oddzielnie sytuację, w której rekordy te nie występują. Podobnie inne czynności będą wykonywane w zależności od wartości operacji w występujących rekordach. Kompletne rozważenie obu możliwości w strukturze wyboru ustrzeże program wynikowy przed nieprzewidzianym działaniem w wypadku wystąpienia błędów w danych rzeczywistych. Inne operacje umieszczone są sekwencyjnie w miejscach ich występowania.

W przetwarzaniu danych wydruk tabulogramów jest problemem bardzo często występującym, stąd przedstawiony przykład może służyć jako wzorzec rozwiązywania dużej grupy programów powstających przy opracowywaniu systemów przetwarzania danych. Innym równie typowym problemem trudnym do rozwiązania przez konstruktora systemu przetwarzania danych jest aktualizacja zbioru podstawowego.

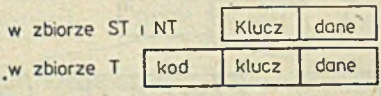
Przykład 5

Przeanalizujemy następujący typowy problem aktualizacji[1] przedstawiony w postaci schematu:



Należy wykonać program aktualizujący zbiór ST danymi zawartymi w zbiorze T. NOWA TASMA jest nowym zbiorem powstałym w wyniku aktualizacji, a raport R zawiera wydruk błędnych transakcji ze zbioru T.

Dla uproszczenia rozważań, struktury rekordów w zbiorach danych mają postać:

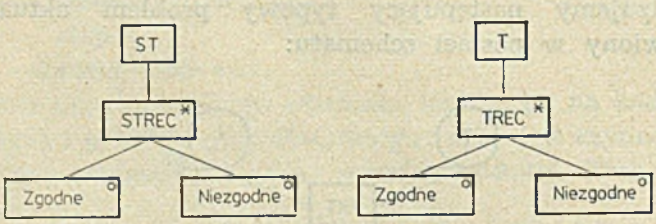


gdzie pole kod może przyjmować wartość:
M — dla modyfikacji rekordu w zbiorze podstawowym,
S — skreślenia (wymazania) rekordu ze zbioru,
W — wstawienie nowego rekordu.
Wszystkie zbiory są uporządkowane rosnąco według pola klucza.

[1] Tamże.

Analizę zbiorów wykorzystywanych przez program rozpoczniemy od zbiorów wejściowych. Zbiory ST i T składają się z pewnej liczby powtarzanych rekordów logicznych. Oba zbiory są uporządkowane jednakowo według pola klucza. Przetwarzając sekwencyjnie rekordy z obu zbiorów nie dla wszystkich wartości klucza ze zbioru ST muszą istnieć odpowiedniki w zbiorze T, co powoduje że występuje lub nie występuje zgodność kluczy obydwu zbiorów wejściowych.

Przeprowadzoną analizę zbiorów wejściowych można przedstawić za pomocą następujących diagramów hierarchicznych.



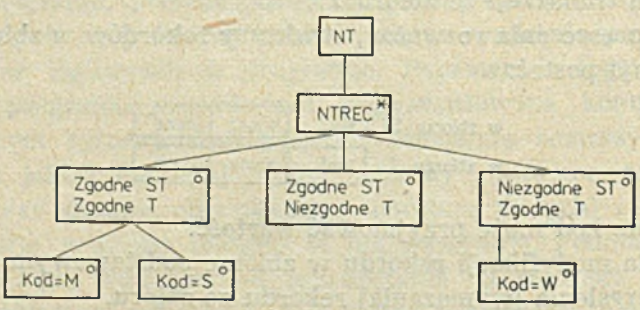
Układ danych w zbiorach ST i T daje trzy kombinacje:

— występują rekordy o tym samym kluczu w zbiorze ST i T,

— występuje rekord o danym kluczu tylko w zbiorze ST,

— występuje rekord o danym kluczu tylko w zbiorze T.

Te kombinacje danych wejściowych wyznaczają wszystkie możliwe sytuacje dla zbiorów wyjściowych NT i R, gdzie również występują iteracje rekordów. Diagram dla zbioru NT ma postać:



W diagramie uwzględniono trzy możliwe do wystąpienia sytuacje, zakładając rosnące uporządkowanie kluczy rekordów w przetwarzanych zbiorach.

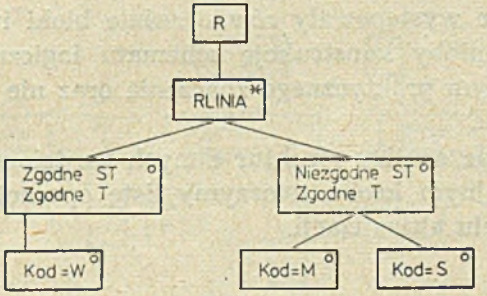
Pierwsza sytuacja ma miejsce wówczas, gdy zgodnie z wzrastającym ułożeniem kluczy rekordów, równocześnie występują rekordy z takimi kluczami w zbiorze ST i T. Na diagramie zostało to skrótowo oznaczone jako zgodne ST i zgodne T.

Druga sytuacja wystąpi wówczas, gdy zgodnie z wzrastającym uporządkowaniem wystąpił rekord w zbiorze ST, natomiast kolejny rekord w zbiorze T ma klucz o większej wartości; na diagramie zostało to oznaczone zgodne ST i niezgodne T.

Trzecia z możliwych sytuacji będzie wówczas, gdy zgodnie z uporządkowaniem wystąpił rekord w zbiorze T, natomiast kolejny rekord w zbiorze ST ma klucz o wartości większej; na diagramie zostało to oznaczone niezgodne ST i zgodne T.

Opisana zasada skrótowego opisu możliwych do wystąpienia sytuacji, w zależności od wartości kluczy pobieranych kolejno do przetwarzania rekordów ze zbiorów, jest stosowana również w dalszych diagramach.

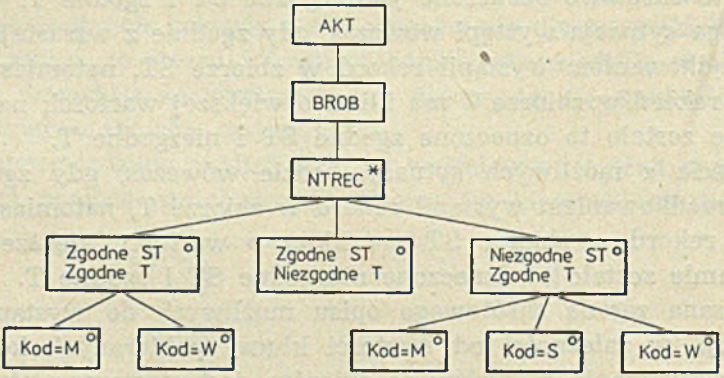
Natomiast raport R ma zawierać rekordy ze zbioru T, dla których kod transakcji jest błędny. Występują tylko dwie możliwości, gdyż kombinacja druga nie powoduje zapisu w raporcie R. Diagram dla zbioru R przyjmuje postać:



W wypadku zgodności kluczy rekordów zbiorów ST i T, kod transakcji W jest błędny, co powoduje wystąpienie rekordu w raporcie błędów. Wartości kodu M i S są natomiast niedopuszczalne, gdy występuje rekord ze zbioru T o kluczu niezgodnym z kluczem rekordu ze zbioru ST.

W procesie aktualizacji należy rozważyć przedstawione wcześniej trzy kombinacje rekordów w zbiorach ST i T oraz sytuację, w jakiej w zbiorze ST rekord nie występuje i nie występuje

również w zbiorze T. Sytuacja braku rekordów w zbiorach ST i T jako nieistotna może być pominięta. Strukturę programu aktualizacji można przedstawić w postaci diagramu.



Sytuacje, w których występuje zgodność kluczy rekordów ze zbiorów ST i T oraz rekord zbioru T ma kod transakcji S można pominąć, gdyż nie wymaga to żadnych działań, co powoduje wymazanie rekordu (rekord nie jest przepisywany do zbioru NT).

W diagramie tym został wprowadzony sztuczny blok BROB po to, aby nie występowały równocześnie bloki iteracji i selekcji, co utrudniłoby konstrukcję schematu logicznego. Blok ten nie ma poza tym praktycznego znaczenia oraz nie zmienia struktury problemu.

Na podstawie analizy struktur danych wejściowych i wyjściowych, w kolejnym kroku tworzymy listę operacji, jakie należy wykonać w celu aktualizacji.

1 otwarcie zbiorów

2 zamknięcie zbiorów

3 stop

4 czytanie zbioru ST

5 czytanie zbioru

6 N=min (klucz ST, klucz T) w celu porównania kluczy rekordów zbioru ST i T

7 pisz rekord do zbioru NT ze zbioru ST

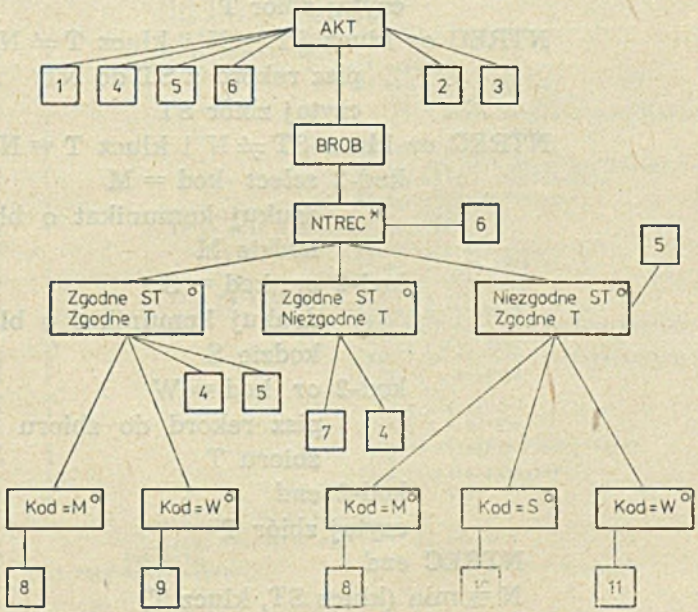
8 pisz rekord do zbioru NT ze zbioru T

9 drukuj komunikat o błędnym kodzie W

10 drukuj komunikat o błędnym kodzie M

11 drukuj komunikat o błędnym kodzie S

Poszczególne elementy z tej listy należy nanieść na diagram, przedstawiający strukturę programu zgodnie z potrzebami aktualizacji.



Na podstawie diagramu opisującego strukturę programu tworzymy schemat logiczny „przechodząc" po przedstawionym drzewie z lewej na prawo, uwzględniając podwieszone operacje.

Schemat logiczny:

```
AKT seg
      otwarcie zbiorów
      czytaj zbiór ST
      czytaj zbiór T
      N = min (klucz ST, klucz T)
BROB iter while N ⩽ max [8]
            NTREC select klucz ST = N i klucz T = N
                  kod-1 select kod = M
                        modyfikuj rekord z ST
                        pisz rekord do NT z ST
                  kod-1 or  kod = W
                        drukuj komunikat o błędnym
                        kodzie W
                  kod-1 end
                  czytaj zbiór ST
                  czytaj zbiór T
            NTREC or klucz ST = N i klucz T ≠ N
                     pisz rekord z ST do NT
                     czytaj zbiór ST
            NTREC or klucz ST ≠ N i klucz T = N
                  kod-2 select kod = M
                        drukuj komunikat o błędnym
                        kodzie M
                  kod-2 or  kod = S
                        drukuj komunikat o błędnym
                        kodzie S
                  kod-2 or  kod = W
                        pisz rekord do zbioru NT ze
                        zbioru T
                  kod-2 end
                  czytaj zbiór T
            NTREC end
            N = min (klucz ST, klucz T)
      BROB end
      zamknięcie zbiorów
      stop
AKT end
```

[8] W celu wykonania badania czy wystąpił koniec zbiorów, wprowadzamy maksymalną wartość klucza, która jest nadawana w momencie wystąpienia końca zbioru.

Na zakończenie rozwiązania pozostaje zamiana schematu logicznego na program w kodzie źródłowym. Wykorzystując język programowania PL/1 otrzymujemy następujący program:

```
STMT LEVEL NEST
 1                    AKT: PROC OPTIONS(MAIN);
 2                       DCL
                          ST FILE RECORD INPUT
                        , 1 STREC
                        ,  2 KLST CHAR(7)
                        ,  2 DANEST CHAR(70)
                        , NT FILE RECORD OUTPUT
                        , 1 NTREC
                        ,  2 KLDANE
                        ,    3 KLNT CHAR(7)
                        ,    3 DANENT CHAR(70)
                        , T FILE RECORD INPUT
                        , 1 TREC
                        ,  2 KOD CHAR(1)
                        ,  2 TREC
                        ,    3 KLT CHAR(7)
                        ,    3 DANET CHAR(70)
                        , N CHAR(7)
                        , MAX CHAR(7) INIT('9999999')
                        ;
 3     1                 ON ENDFILE(ST) KLST = MAX;
 5     1                 ON ENDFILE(T)  KLT = MAX;
 7     1                 OPEN FILE(ST), FILE(NT),FILE(T);
 8     1                 READ FILE(ST) INTO(STREC);
 9     1                 READ FILE(T) INTO(TREC);
10     1                 N = MIN(KLST,KLT);
11     1                 DO WHILE(N¬= MAX);
12     1     1            IF N = KLST & N = KLT THEN DO;
14     1     2                                IF KOD = 'M' THEN DO;
16     1     3                                         DANEST = DANET;
17     1     3                                         WRITE FILE(NT) FROM(STREC);
18     1     3                                                END;
19     1     2                                IF KOD ='W' THEN
20     1     2                                       PUT SKIP EDIT('W',TREC)(A,X(5),A);
21     1     2                                READ FILE(ST) INTO(STREC);
22     1     2                                READ FILE(T) INTO(TREC);
23     1     2                                         END;
24     1     1            IF N= KLST & N ¬= KLT THEN DO;
26     1     2                                WRITE FILE(NT) FROM(STREC);
27     1     2                                READ FILE(ST) INTO(STREC);
28     1     2                                         END;
29     1     1            IF N ¬= KLST & N = KLT THEN DO;
31     1     2                                IF KOD = 'M' THEN
32     1     2                                 PUT SKIP EDIT('M',TREC)(A,X(5),A);
33     1     2                                IF KOD = 'S' THEN
34     1     2                                 PUT SKIP EDIT('S',TREC)(A,X(5),A);
35     1     2                               IF KOD = ' W' THEN
36     1     2                                 DO;
37     1     3                                  KLNT = KLT;
38     1     3                                  DANENT = DANET;
39     1     3                                  WRITE FILE(NT) FROM(NTREC);
40     1     3                                 END;
41     1     2                                READ FILE(T) INTO(TREC);
42     1     2                                         END;
43     1     1              = MIN(KLST,KLT);
44     1     1            END;
45     1                 CLOSE FILE(ST), FILE(NT), FILE(T);
46     1              END AKT;
          AKT: PROC OPTIONS(MAIN);
```

g. Przykład obsługi końca zbioru

Przykład 6

Analizujemy wejściowy zbiór kart, zawierający karty czołowe oraz trzy typy kart T1, T2 i T3, które mają wartości 1, 2 i 3

odpowiednio dla typu w pierwszej kolumnie karty. Warunki analizy są następujące:

— zliczamy karty czołowe poprzedzające pierwszą kartę typu T1 (licznik LA),

— drukujemy pierwszą kartę typu T1, ale jej wystąpienie nie jest liczone,

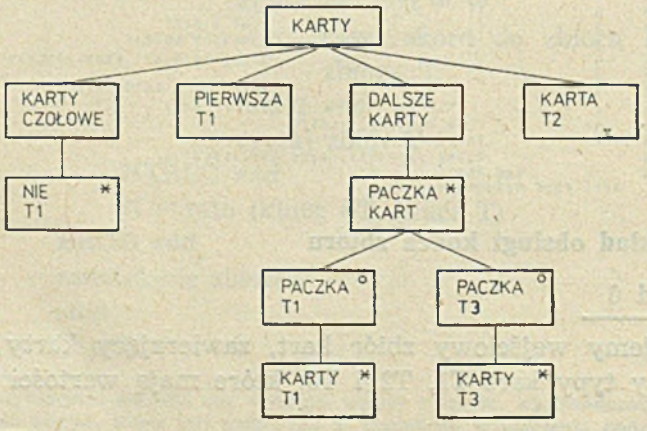
— drukujemy ostatnią kartę w zbiorze, którą jest karta typu T2; karta typu T2 ma być traktowana jako karta końca jedynie w razie wcześniejszego wystąpienia pierwszej karty typu T1; karty typu T2 występujące wcześniej należy zaliczyć do kart czołowych,

— zliczamy grupę kart występujących między pierwszą kartą typu T1 a kartą ostatnią; są to występujące pojedynczo lub w grupach karty typu T1 i T3 (licznik LB), gdyż karta typu T2 będzie potraktowana jako karta końca,

— zliczamy karty typu T1 występujące po pierwszej karcie T1 (licznik LC), nie licząc pierwszej,

— zliczamy paczki kart typu T3 występujące po pierwszej karcie typu T1 (licznik LD), w szczególności paczka może składać się z jednej karty.

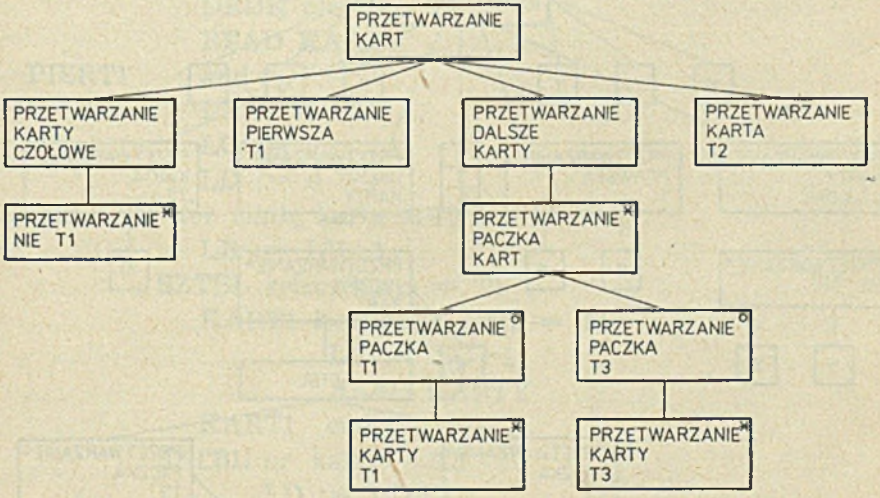
Po wydrukowaniu ostatniej karty, drukujemy wartości liczników. Zakładamy, że zbiór wejściowy kart istnieje i ma poprawny format, tzn. występuje co najmniej jedna karta typu T1; ostatnią kartą zbioru jest karta typu T2 i nie ma innych kart typu T2 między pierwszą kartą typu T1 a kartą ostatnią [1]. Powstaje diagram dla określenia struktury zbioru wejściowego.



[1] Por. M. Jackson, op. cit.

Diagram ten przedstawia zbiór wejściowy z punktu widzenia wymaganych analiz. Zostały w nim wyróżnione karty poprzedzające pierwszą kartę typu T1, które są iteracją kart typu różnego od T1. Wyróżniliśmy pierwszą kartę typu T1 oraz ostatnią kartę, którą jest karta typu 2. Między nimi występują karty typu T1 i T3 w postaci paczek kart. W ramach iteracji dla paczek kart, występuje struktura wyboru, w zależności od tego czy występująca paczka kart to karty typu T1 czy T3, ze względu na odmienny sposób ich zliczania.

Strukturę programu tworzymy w prosty sposób z diagramu przedstawiającego strukturę zbioru wejściowego, przez dodanie do każdego prostokąta w diagramie słowa *przetwarzanie*, gdyż zbiór wyjściowy nie ma specjalnej struktury. Zbiorem wyjściowym w rozwiązywanym przykładzie jest wydrukowanie informacji zgodnie z warunkami przetwarzania.

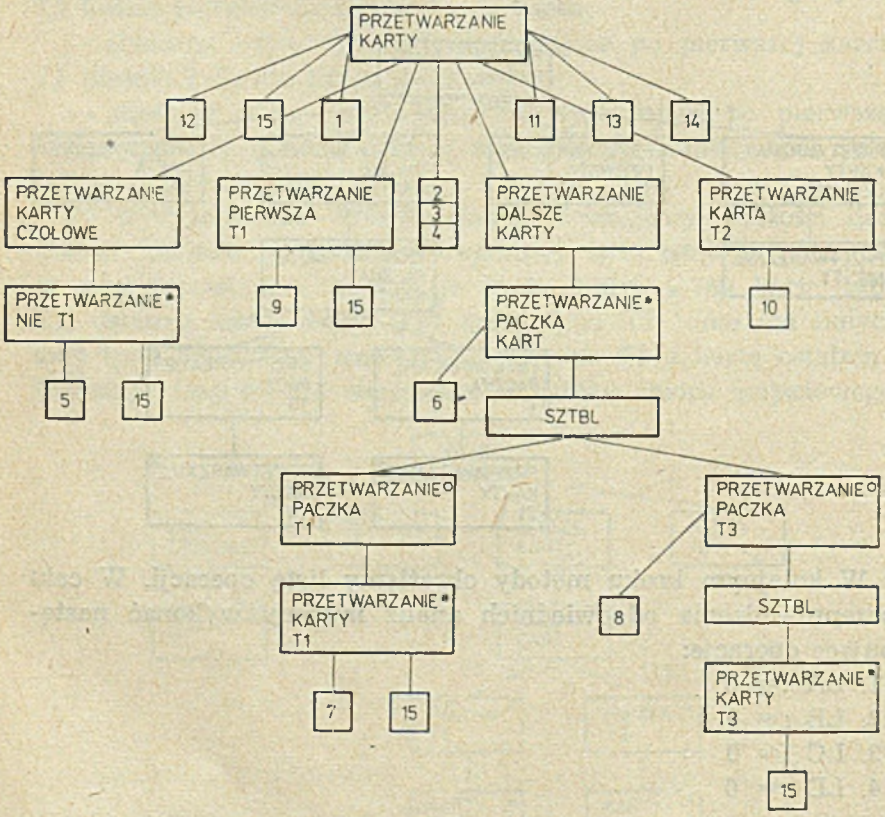


W kolejnym kroku metody określamy listę operacji. W celu przeprowadzenia odpowiednich analiz musimy wykonać następujące operacje:

1. LA := 0
2. LB := 0
3. LC := 0
4. LD := 0

5. LA := LA+1
6. LB := LB+1
7. LC := LC+1
8. LD := LD+1
9. DRUK pierwszej T1
10. DRUK karty T2
11. DRUK LA, LB, LC, LD
12. OPEN KARTY
13. CLOSE KARTY
14. STOP
15. READ KARTY

W kolejnym kroku metody wykonamy alokacje operacji z przedstawionej listy na diagramie przedstawiającym strukturę programu.



W diagramie tym wprowadzono dwa sztuczne bloki (SZTBL), w celu lepszego przedstawienia miejsca zwiększania liczników LB i LD, z tym że nie mają one wpływu na strukturę programu.

Na podstawie otrzymanego diagramu budujemy schemat logiczny.

```
KARTY seq
        OPEN KARTY
        READ KARTY
        LA := 0
CZOLKAR iter while karta ≠ T1
            LA := LA+1
            READ KARTY
CZOLKAR end
PIERT1  seq
            DRUK pierwsza T1
            READ KARTY
PIERT1  end
            LB := 0
            LC := 0
            LD := 0
DALKAR iter while karta ≠ T2
            LB := LB+1
          SZTBL select karta = T1
            KART1 iter while karta = T1
                    LC := LC+1
                    READ KARTY
            KART1 end
          SZTBL or karta = T3
                LD := LD+1
            KART3 iter while karta = T3
            READ KARTY
            KART3 end
          SZTBL end
DALKAR end
          KART2 seq
                DRUK karty T2
          KART2 end
```

```
                DRUK LA, LB, LC, LD
                CLOSE KARTY
                STOP
        KARTY end
```

M. Jackson jako jedno z możliwych rozwiązań dla omawianego przykładu proponuje następujące kodowanie w języku COBOL.

```
    ...
    77 EOF PIC X VALUE SPACE.
    ...
    PROCEDURE DIVISION.
    KASTSEK.
        OPEN INPUT KARTY.
        READ KARTY AT END MOVE E TO EOF.
        MOVE O TO LA.
    CZOLKAR.
        IF T1 GO TO CZOLEND.
        ADD 1 TO LA.
        READ KARTY AT END MOVE "E" TO EOF.
        GO TO CZOLKAR.
  CZOLEND.
        DISPLAY REKKART.
        READ KARTY AT END MOVE "E" TO EOF.
        MOVE O TO  LB, LC, LD.
  DALKAR.
     IF (NOT T1) AND (NOT T3) GO TO DALKAREND.
     ADD 1 TO LB.
  SZTBL.
    IF NOT T1 GO TO SZTBLOR.
  KART1.
     IF NOT T1 GO TO KART1END.
     ADD 1 TO LC.
     READ KARTY AT END MOVE "E" TO EOF.
     GO TO KART1.
  KART1END.
     GO TO SZTBLEND.
  SZTBLOR.
     ADD 1 TO LD.
```

```
KART3.
    IF NOT T3 GO TO KART3END.
    READ KARTY AT END MOVE "E" TO EOF.
    GO TO KART3.
KART3END.
SZTBLEND.
    GO TO KART1.
DALKAREND.
    DISPLAY REKKART.
    DISPLAY LA, LB, LC, LD.
    CLOSE KARTY.
    STOP RUN.
```

W przedstawionym kodzie program jest mniej przejrzysty niż w pseudokodzie ze względu na liczne instrukcje skoku bezwarunkowego GO TO. Struktura logiczna programu zawarta w diagramie określającym strukturę programu została zachowana, ale większą przejrzystość programu można osiągnąć przez użycie instrukcji PERFORM zamiast bardzo często przekazywanego sterowania za pomocą GO TO.

O wiele wygodniej kodować przedstawiony schemat logiczny, gdy wykorzystujemy język PL/1. Przedstawiamy inne kodowanie schematu logicznego w tym języku programowania (rozwiązanie autora — p. str. 124).

Struktura programu w PL/1 jest przejrzysta i czytelna. W programie kodowanym w PL/1 wykorzystano strumieniowe instrukcje wprowadzania i wyprowadzania, operujące na systemowych urządzeniach wejścia i wyjścia, co umożliwiło opuszczenie szczegółowej deklaracji zbiorów wraz z podawaniem ich nazw.

Wejściowy zbiór karty jest zbiorem seryjnym, a więc i sposób jego obsługi jest typowy dla tego rodzaju zbiorów. W omawianym przykładzie jedynie sposób zakończenia przetwarzania jest nietypowy. Zgodnie z założeniami przykładu, przetwarzanie jest zakończone z chwilą napotkania karty typu T2, która jest ostatnią kartą zbioru. W praktyce programowania najczęściej wykorzystuje się standardowy rekord końca zbioru. W zależności od nośnika danych przetwarzanego zbioru i systemu oprogramowania są to rekordy o różnej wartości.

```
STMT LEVEL NEST
  1                    KARTY: PROC OPTIONS(MAIN);
  2     1                     DCL
                              REKIN CHAR(80)
                             , 1 REK DEF REKIN
                             ,   2 TK PIC '9'
                             ,   2 RESZ CHAR(79)
                             , (LA,LB,LC,LD) FIXED BINARY(15)
                             ;
  3     1                     GET EDIT(REKIN)(A(80));
  4     1                     LA = 0;
  5     1                     DO WHILE(TK ^= 1);
  6     1     1                LA = LA + 1;
  7     1     1                GET EDIT(REKIN)(A(80));
  8     1     1               END;
  9     1                     PUT SKIP EDIT(REKIN)(A);
 10     1                     GET EDIT(REKIN)(A(80));
 11     1                     LB = 0;
 12     1                     LC = 0;
 13     1                     LD = 0;
 14     1                     DO WHILE(TK ^= 2);
 15     1     1                LB = LB + 1;
 16     1     1                IF TK = 1 THEN DO;
 18     1     2                                 DO WHILE(TK = 1);
 19     1     3                                  LC = LC + 1;
 20     1     3                                  GET EDIT(REKIN)(A(80));
 21     1     3                                 END;
 22     1     2                                END;
 23     1     1                           ELSE DO;
 24     1     2                                 LD = LD + 1;
 25     1     2                                 DO WHILE(TK = 3);
 26     1     3                                  GET EDIT(REKIN)(A(80));
 27     1     3                                 END;
 28     1     2                                END;
 29     1     1               END;
 30     1                     PUT SKIP EDIT(REKIN)(A);
 31     1                     PUT DATA(LA,LB,LC,LD);
 32     1                END KARTY;
          KARTY: PROC OPTIONS(MAIN);
```

W zbiorach przetwarzanych sekwencyjnie koniec zbioru jest sygnalizowany przez wystąpienie standardowego rekordu końca zbioru. Rekord taki jest odczytywany tak samo, jak inne rekordy zbioru. Języki programowania w aparacie instrukcji mają mechanizmy wykrywania rekordu końca zbioru i podejmowania odpowiedniej akcji.

W języku COBOL, instrukcja wprowadzania danych

READ ZB-WE AT END ZDANIE.

umożliwia zakodowanie zdania bezwarunkowego (najczęściej jest to instrukcja GO TO lub PERFORM), które wywołuje podjęcie akcji odpowiedniej dla wystąpienia końca zbioru. Opcja AT END musi występować w każdej instrukcji wprowadzania, a jest rea-

lizowana jedynie wówczas, gdy przetwarzanym rekordem jest standardowy rekord końca zbioru.

W języku PL/1 koniec zbioru jest obsługiwany przez specjalnie programowany warunek ON.

```
ON ENDFILE (ZB-WE) BEGIN,
   ...
   END;
```

w zakodowanym bloku BEGIN kodujemy wymagane dla końca zbioru działanie programu. Blok ten będzie wykonany wówczas, gdy jedna z kolejnych instrukcji wprowadzania rekordów ze zbioru ZB-WE odczyta standardowy rekord końca zbioru.

W rozważanym przykładzie, ani w programie kodowanym w języku COBOL (chociaż kodowana była opcja AT END), ani w kodzie PL/1, nie obsługiwaliśmy standardowego rekordu końca zbioru, co wynikało z założenia, że ostatnim rekordem w zbiorze jest karta T2. Chcąc uniknąć błędnego działania programu, gdy brak karty typu T2 na końcu zbioru kartowego, należałoby zmodyfikować oba programy tak, aby badane było wystąpienie standardowego rekordu końca zbioru.

W programie kodowanym w języku COBOL należy wprowadzić uzupełnienia.

```
...
77 EOF PIC X VALUE SPACE.
88 WAR-EOF VALUE "E".
...
KART1.
  IF WAR-EOF GO TO DALKAREND.
...
KART3.
  IF WAR-EOF GO TO DALKAREND.
...
```

W programie kodowanym w języku PL/1 należy dołożyć warunek ON ENDFILE dla zbioru SYSIN:

```
WSK PIC '9' INIT (0),
ON ENDFILE (SYSIN) WSK = 1;
```

oraz zmodyfikować instrukcją DO, przez dołożenie warunku badania wartości wskaźnika WSK.

```
DO WHILE (TK = 2 OR WSK = 1);
```

Po wprowadzeniu tych modyfikacji program będzie funkcjonował prawidłowo, niezależnie od tego czy karta typu T2 wystąpi jako ostatnia w zbiorze, czy też koniec zbioru zostanie wykryty przez standardową kartę końca zbioru.

3. Metoda Warniera

a. Założenia metody

Francuski inżynier J. D. Warnier pracując w firmie Honeywell Bull w 1974 r. opracował, a w dwa lata później zmodyfikował, metodę, która miała na celu:

— skrócenie czasu opracowania programu o 1/5 lub 1/6 normalnie wymaganego,

— skrócenie w sposób znaczący rozmiaru pamięci zajmowanej przez program,

— skrócenie czasu zajętości jednostki centralnej wykorzystywanej przez program,

— wykonanie dokumentacji, która umożliwia prawidłową konserwację wykonanego programu.

Autor tej metody opartej na zasadach algebry Boole'a i teorii zbiorów opracował szczegółową listę zdań, które mogą być łatwo zakodowane przy użyciu dowolnego języka programowania. Metoda ta podobnie jak metoda Jacksona oparta jest na strukturach danych, które są przetwarzane przez program oraz uwzględnia podstawowe zasady programowania strukturalnego, ze szczególnym wykorzystaniem struktur sekwencji, powtarzania i wyboru.

Metoda postępowania proponowana przez Warniera jest wykonywana krokami, a za podstawę rozważań przyjmuje strukturę danych wykorzystywanych przez program. W odróżnieniu od metody Jacksona bardziej szczegółowo są rozważane operacje wykonywane w programie. Szczególną uwagę zwraca się na sytuacje decyzyjne oraz związane z nimi warunki.

M. Jackson w swojej metodzie nie wykorzystuje schematu blokowego dla graficznego przedstawienia struktury programu, proponując w to miejsce diagram strukturalny. Warnier natomiast wykorzystuje graficzną metodę prezentacji struktury pro-

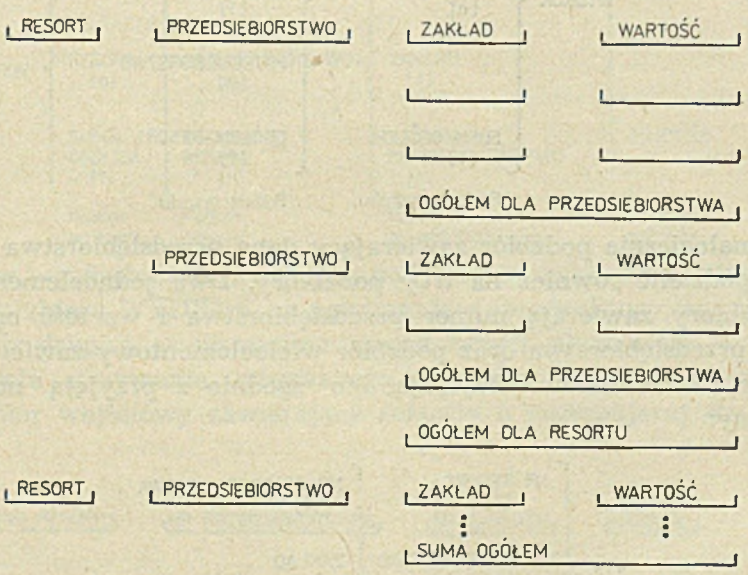
gramu za pomocą organigramu, którego podstawowe założenia nie odbiegają od techniki schematu blokowego; organigram powstaje nieco inaczej, a poszczególne bloki są numerowane w odpowiedni sposób.

Struktura sekwencji ze względu na swą prostotę nie wymaga specjalnego omówienia, natomiast pozostałe struktury podstawowe przedstawimy za pośrednictwem przykładów.

b. Struktura powtarzania

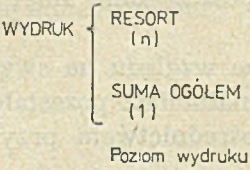
Przykład 7

Dla przedstawienia sposobu wykonania struktury powtarzania przez Warniera przeanalizujemy problem polegający na opracowaniu wydruku, który zawiera wykaz wartości produkcji z podziałem na resorty, przedsiębiorstwa i zakłady. Wydruk ma postać:



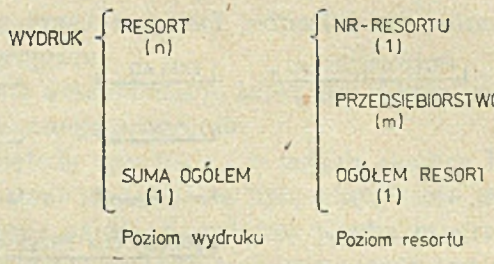
Traktując te dane jako zbiór w sensie matematycznym, można ten zbiór podzielić na podzbiory. W zbiorze danych wydruku można wyróżnić podzbiór wieloelementowy, dla każdego resortu oraz podzbiór jednoelementowy— zawierający sumę ogółem —

dla wszystkich resortów. Można to zapisać, zgodnie z notacją proponowaną przez Warniera:

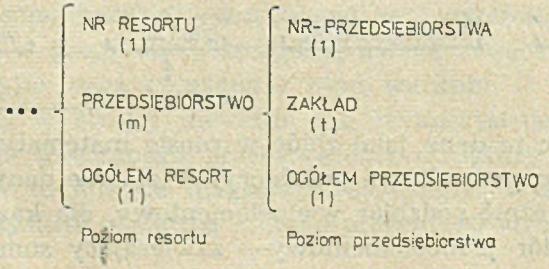


Oznaczenie w nawiasie określa liczność zbioru.

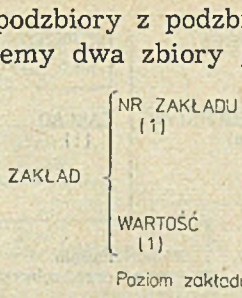
Podzbiór wieloelementowy, w omawianym przykładzie zawierający dane resortu, można dalej podzielić, wyróżniając trzy kolejne podzbiory. Podzbiór jednoelementowy zawierający numer resortu i podzbiór jednoelementowy zawierający wartość ogółem dla resortów oraz podzbiór wieloelementowy, zawierający dane dla przedsiębiorstwa. Zapisujemy to w sposób zgodny z proponowaną notacją, wyróżniając kolejny poziom:



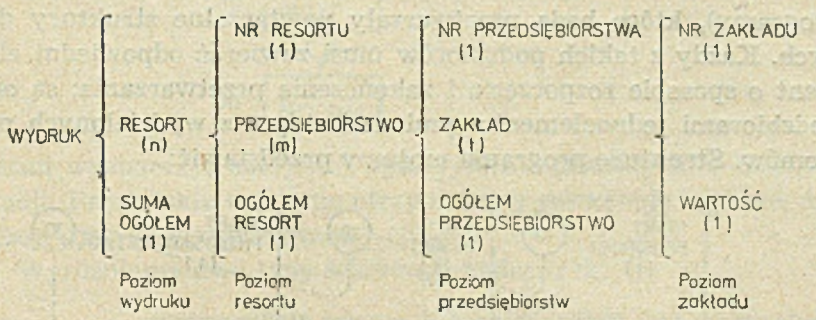
Analogicznie podzbiór zawierający dane przedsiębiorstwa można podzielić również na trzy podzbiory. Dwa jednoelementowe podzbiory zawierają numer przedsiębiorstwa i wartość ogółem dla przedsiębiorstwa oraz podzbiór wieloelementowy zawierający dane dla zakładów. Zapisując to zgodnie z przyjętą notacją mamy:



Wydzielając dalsze podzbiory z podzbioru zawierającego dane dla zakładu, otrzymujemy dwa zbiory jednoelementowe:



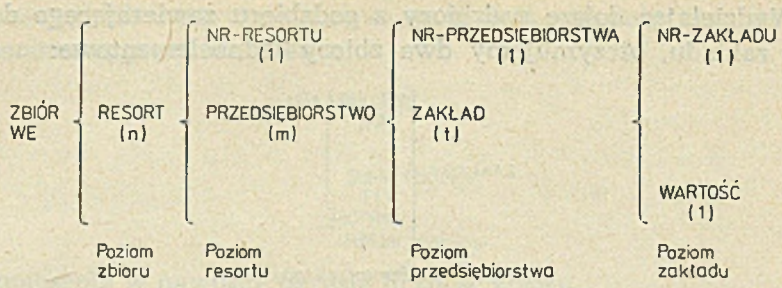
W wyniku kolejnych podziałów dochodzimy do najniższego poziomu, na którym otrzymamy podzbiory jednoelementowe. Na poziomie zakładu otrzymamy dwa podzbiory jednoelementowe, co uniemożliwia dalszy proces wydzielania podzbiorów. Oznacza to koniec procesu podziału zbioru wydruku na podzbiory. Składając wszystkie wydzielone poziomy, otrzymujemy następującą strukturę danych:



Przez wyodrębnienie kolejnych podzbiorów wykonamy dekompozycję zbioru WYDRUK na powtarzające się struktury, zawarte jedna w drugiej, aż do poziomu pojedynczych elementów.

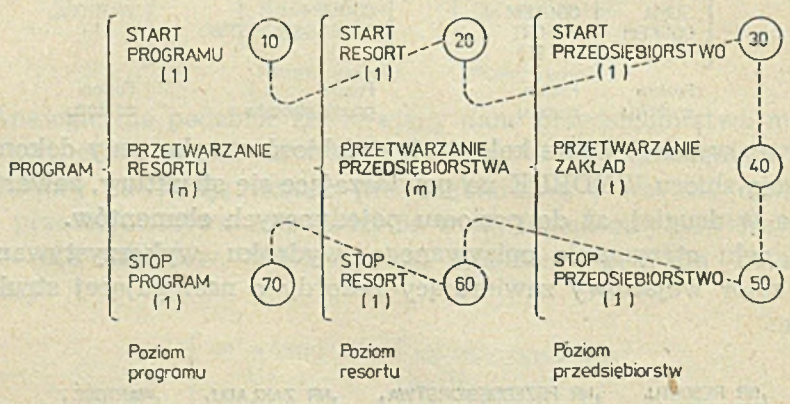
W celu otrzymania opisywanego wydruku wykorzystywany jest zbiór wejściowy zawierający rekordy o następującej strukturze:

Zbiór ten jest uporządkowany rosnąco według wzrastających wartości trzech pierwszych pól. Postępując analogicznie, jak w wypadku zbioru WYDRUK, można przez kolejne rozwinięcia poziomów przedstawić strukturę zbioru wejściowego:



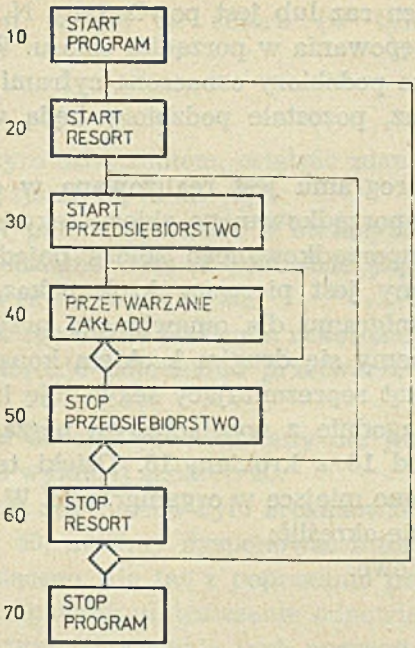
Zbiór wejściowy składa się ze struktur powtarzalnych dla resortu, przedsiębiorstwa i zakładu.

Hierarchia programu rozwiązującego opisany problem wywodzi się z hierarchii zbioru wejściowego. Powtarzalne struktury danych powodują wystąpienie struktury powtarzania w programie. W strukturze programu można wyróżnić podzbiory (podprogramy), które będą przetwarzały powtarzalne struktury danych. Każdy z takich podzbiorów musi zawierać odpowiedni element o sposobie rozpoczęcia i zakończenia przetwarzania; są one podzbiorami jednoelementowymi na każdym z wydzielonych poziomów. Strukturę programu możemy przedstawić:



Wychodząc z przedstawionej struktury programu można zbudować organigram, który reprezentuje rozwiązanie problemu w formie graficznej.

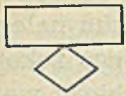
W przedstawionym organigramie każdy podzbiór z programu jest reprezentowany przez prostokąt. Prostokąty połączone strzał-



kami wyznaczają kolejność wykonywania zawartych w nich operacji. Przez takie działanie otrzymujemy sekwencję logiczną operacji ujętych w organigram.

Wyróżniamy dwa typy sekwencji logicznych:

reprezentuje zdanie lub zbiór zdań; jest to struktura sekwencyjna,



zbiór zdań zawierający decyzję, która jest podejmowana w ostatnim zdaniu sekwencji; w zależności od stanu warunku powtarzany jest ciąg wcześniejszych operacji lub wykonywana jest kolejna w porządku operacja. Jest to znana już struktura powtarzania typu DO UNTIL.

Wyrażając to w nomenklaturze zbiorów możemy stwierdzić, że *program jest uporządkowanym zbiorem sekwencji logicznych*, co można dodatkowo zapisać:

$$P = \{L\},$$

gdzie L jest sekwencją logiczną.

Każdy podzbiór składający się na program jest wykonywany co najmniej jeden raz lub jest powtarzany N razy w zależności od miejsca występowania w porządku zbioru. W przedstawionym przykładzie tylko podzbiory oznaczone cyframi 10 i 70 będą wykonane jeden raz, pozostałe podzbiory będą wykonywane wielokrotnie.

Konstrukcja programu jest realizowana w dwóch krokach:
— konstrukcja uporządkowanego zbioru sekwencji logicznych,
— konstrukcja uporządkowanego zbioru pojedynczych zdań.

Jak realizowany jest pierwszy krok pokazaliśmy przy konstruowaniu organigramu dla omawianego przykładu.

Obecnie zajmiemy się drugim krokiem konstrukcji programu.

Każdy prostokąt reprezentujący sekwencję logiczną otrzymuje kolejny numer zgodnie z wyznaczonym porządkiem. Przyjmuje się numerację od 10 z krokiem 10. Dzięki temu każde zdanie ma ściśle określone miejsce w organigramie. W każdym wypadku musimy dokładnie określić:

1. Dane wejściowe.
2. Decyzje.
3. Obliczenia.
4. Dane wyjściowe.
5. Wywoływane podprogramy.

Zajmiemy się tym na omówionym wcześniej przykładzie kolejno dla wymienionych operacji.

Dane wejściowe

Analizując dane wejściowe należy wziąć pod uwagę liczbę rekordów oraz sposób ich przetwarzania. Przede wszystkim należy zwrócić uwagę na przetwarzanie pierwszego i ostatniego rekordu oraz sposób przetwarzania pozostałych rekordów zbioru wejściowego. Umieszczając instrukcje operujące na danych wejściowych w odpowiednich miejscach organigramu, należy wiedzieć, które sekwencje są wykonywane tylko raz, a które wielokrotnie i w zależności od tego, umieścić odpowiednie instrukcje. (W omawianym przykładzie są to sekwencje o numerach 40 i 10). W wyniku przeprowadzonych rozważań można dla danych wejściowych wyspecyfikować następujące operacje:

10 czytaj pierwszy rekord ze zbioru,
40 czytaj jeden rekord ze zbioru (w tym rekord końca zbioru).

Decyzje

Zgodnie z przyjętym oznaczeniem, ostatnie zdania w sekwencjach 40, 50 i 60 są decyzjami. Są to decyzje:

40 jeżeli numer przedsiębiorstwa w bieżącym rekordzie = numerowi przedsiębiorstwa w rekordzie poprzednio przetwarzanym, to wykonaj skok do 40,
50 jeżeli numer resortu w bieżącym rekordzie = numerowi resortu w rekordzie poprzednio przetwarzanym, to wykonaj skok do 30,
60 jeżeli rekord obecnie przetwarzany nie jest rekordem końca zbioru, to wykonaj skok do 20.

Należy dodać, że aby można było zrealizować decyzje zawarte w sekwencji 40 i 50, musimy dysponować informacjami nie tylko z rekordu bieżącego, ale też z poprzednio przetwarzanego. W tym celu należy zapewnić umieszczenie odpowiednich informacji w obszarze roboczym. Wykonanie tych operacji oraz miejsce ich występowania musi być dokładnie określone, stąd operacje:

20 przenieść numer resortu do obszaru roboczego,
30 przenieść numer przedsiębiorstwa do obszaru roboczego.

Obliczenia

W rozważanym przykładzie wykonujemy sumowanie na trzech poziomach. Obliczamy sumy ogólne, na poziomie resortu oraz na poziomie przedsiębiorstwa. Na początku obliczeń pola zawierające odpowiednie sumy należy wyzerować, a następnie w kolejno powtarzanych strukturach zwiększać o wielkości zapisane w bieżąco powtarzanych rekordach; wynikają stąd następujące operacje:

10 zeruj licznik dla sumy ogółem,
20 zeruj licznik dla sumy resortu,
30 zeruj licznik dla sumy przedsiębiorstwa,
40 zwiększ licznik dla sumy przedsiębiorstwa o wartości,
50 zwiększ licznik dla sumy resortu o sumę przedsiębiorstwa,
60 zwiększ licznik dla sumy ogółem o sumę resortu.

Operacje wykonywane na danych wyjściowych to drukowanie odpowiednich informacji dla każdego wiersza oraz spacjowanie obszaru buforowego po wydrukowaniu każdego wiersza. Oznaczając bufor wyjściowy przez WY, mamy następujące operacje:

20 numer resortu → WY,
30 numer przedsiębiorstwa → WY,
40 numer zakładu i wartość → WY,
40 drukuj WY, spacjowanie WY,
50 suma przedsiębiorstwa → WY,
50 drukuj WY, spacjowanie WY,
60 suma dla resortu → WY,
60 drukuj WY, spacjowanie WY,
70 suma ogółem → WY,
70 drukuj WY, spacjowanie WY.

Przez wykonanie kolejnych czynności otrzymaliśmy, w ramach każdej oddzielnie ponumerowanej sekwencji logicznej, pewną grupę operacji, które należy dodatkowo uporządkować. Porządkowanie operacji wewnątrz każdej sekwencji wykonujemy według następujących zasad:

1) czynności przygotowawcze do decyzji,
2) czynności przygotowawcze do obliczeń,
3) obliczenia,
4) czynności przygotowawcze dotyczące danych wyjściowych,
5) dane wyjściowe,
6) dane wejściowe,
7) decyzje.

Jeżeli pewna grupa operacji występuje w takim samym porządku w więcej niż jednej sekwencji, to można zapisać ją w postaci podprogramu i w miejscu jej występowania wpisać instrukcję wywołującą ten podprogram.

Grupując wszystkie wymienione w poprzednich krokach operacje zgodnie z podanymi zasadami porządkowania, otrzymujemy następujący uporządkowany zbiór zdań składających się na program:

10 zeruj licznik dla sumy ogółem,
czytaj pierwszy rekord w zbiorze,
20 przenieś numer resortu do obszaru roboczego,
zeruj licznik dla sumy resortu,
numer resortu → WY,
30 przenieś numer przedsiębiorstwa do obszaru roboczego,
zeruj licznik dla sumy przedsiębiorstwa,
numer przedsiębiorstwa → WY,
40 zwiększ licznik dla sumy przedsiębiorstwa o wartość,
numer zakładu i wartość → WY,
drukuj WY, spacjowanie WY,
czytaj jeden rekord ze zbioru,
jeżeli numer przedsiębiorstwa w bieżącym rekordzie = numerowi przedsiębiorstwa w rekordzie poprzednio przetwarzanym, wykonaj skok do 40,
50 zwiększ licznik dla sumy resortu o sumę dla przedsiębiorstwa,
suma przedsiębiorstwa → WY,
drukuj WY, spacjowanie WY,
jeżeli numer resortu w bieżącym rekordzie = numerowi resortu w rekordzie poprzednio przetwarzanym, wykonaj skok do 30,
60 zwiększ licznik dla sumy ogółem o sumę resortu,
suma dla resortu → WY,
drukuj WY, spacjowanie WY,
jeżeli rekord obecnie przetwarzany nie jest rekordem końca, wykonaj skok do 20,
70 suma ogółem → WY,
drukuj WY, spacjowanie WY.

Pozostaje kodowanie tak opisanego programu w jednym z języków programowania. W wypadku języka programowania wyższego rzędu, poszczególne zapisy odpowiadają instrukcjom języka, co umożliwia proste i szybkie otrzymanie kodu nadającego się do maszynowego przetwarzania.

Rozwiązując konkretnie zdefiniowany problem prześledziliśmy kolejne kroki postępowania w metodzie Warniera. Reasumując omówione szczegółowo postępowanie, można wyróżnić kolejne kroki tej metody:

1) określenie hierarchii zbiorów wyjściowych,
2) określenie hierarchii zbiorów wejściowych,
3) określenie hierarchii programu,
4) budowa organigramu,
5) konstruowanie uporządkowanego zbioru zdań,
6) kodowanie programu.

Przy określaniu hierarchii zbiorów i programu są stosowane następujące główne zasady:

a) każdy zbiór danych jest zbiorem w sensie matematycznym,
b) przy określaniu struktury zbiorów zakładamy, że:
   — zbiór składa się z podzbiorów, które można uporządkować w hierarchii wzajemnego zawierania się w sensie teorii zbiorów,
   — na najniższym poziomie, gdzie zbiór nie ulega podziałowi, muszą być nadane nazwy wszystkim elementom,
   — pewien poziom dekompozycji zbioru, który nie wystąpił w zasadach przetwarzania nie może wystąpić w hierarchii innego zbioru; inaczej, liczba poziomów dekompozycji zbiorów musi być taka sama dla wszystkich zbiorów,
c) każdy program jest zbiorem uporządkowanych zdań,
d) przetwarzane struktury danych implikują programowe struktury powtarzania,
e) podzbiór programowy zawiera jeden podzbiór rozpoczynający proces i jeden podzbiór kończący proces, które są wykonywane tylko raz w ramach danego podzbioru programowego.

Przy konstruowaniu uporządkowanego zbioru zdań składającego się na program należy w podanej kolejności rozważyć:

a) dane wejściowe,
b) decyzje,
c) czynności przygotowawcze do decyzji,
d) czynności przygotowawcze do obliczeń,
e) obliczenia,
f) czynności przygotowawcze w odniesieniu do danych wejściowych,
g) dane wyjściowe,
h) podprogramy wywoływane.

Konstruując szczegółową listę zdań w ramach pojedynczych sekwencji logicznych należy przestrzegać uporządkowania według następującej kolejności:

1) przygotowanie do decyzji,
2) przygotowanie do obliczeń,
3) obliczenia,
4) przygotowanie w odniesieniu do danych wyjściowych,
5) dane wyjściowe,
6) dane wejściowe,
7) decyzje.

c. Struktura wyboru

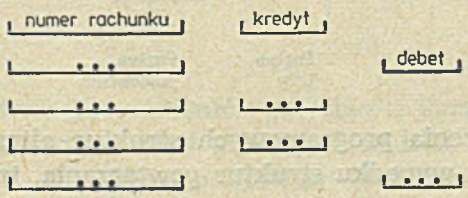
Poznaliśmy ogólne zasady metody Warniera na przykładzie, dla rozwiązania którego wystarczyły struktury sekwencyjna i powtarzania. Omówimy obecnie strukturę wyboru stosowaną również w tej metodzie.

Struktura wyboru występuje wówczas, gdy w rozwiązywanym problemie występuje przynajmniej jeden element alternatywny. W odniesieniu do zbioru danych oznacza to: zbiór danych zawiera przynajmniej jeden podzbiór, który może wystąpić lub nie w strukturze danych. Dla takiego wypadku przyjmujemy notację (0,1). W odniesieniu do zbioru programu oznacza to: podzbiór programu zawiera oprócz „start programu" i „stop programu", podzbiory spośród których przynajmniej dwa są wykonywane alternatywnie (przy czym w szczególności jeden z podzbiorów może być pusty analogicznie do struktury typu IF THEN).

Rozważmy te sytuacje na przykładach.

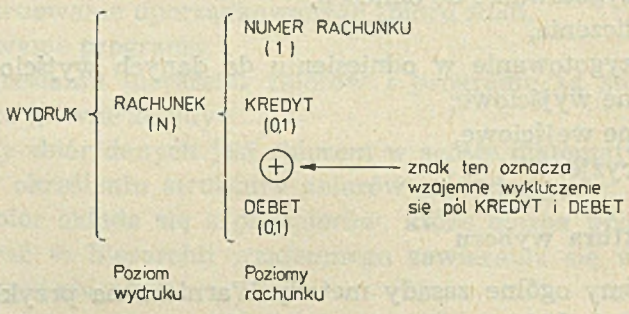
Przykład 8

Rozważmy wydruk bankowy w postaci:



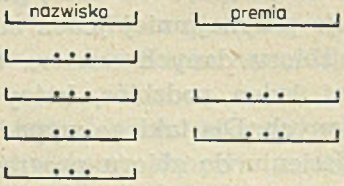
W tym przykładzie podzbiory „kredyt" i „debet" są podzbiorami alternatywnymi. Jeśli występuje podzbiór „kredyt" nie występuje podzbiór „debet" i na odwrót.

Struktura zbioru WYDRUK ma postać:



Przykład 9

Rozważmy listę pracowników w postaci:



Jeśli przy nazwisku nie występuje zapis, oznacza to że dany pracownik nie otrzymuje premii. W tym wypadku podzbiór premia w strukturze programu powoduje występowanie „pustego procesu", tzn. brak akcji. Struktura zbioru ma postać:

Zasady tworzenia programowych struktur alternatywnych są podobne, jak w wypadku struktur powtarzania, tzn. alternatyw-

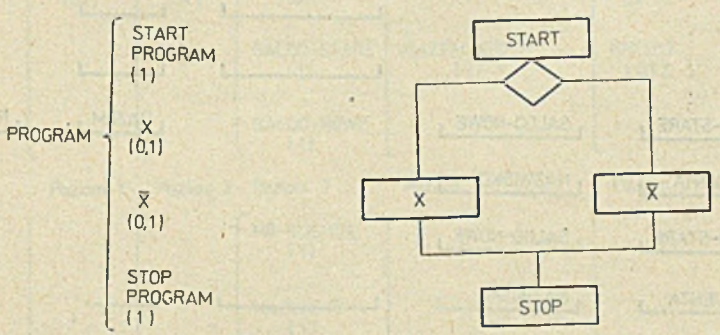
na struktura danych daje w rezultacie alternatywną strukturę programową. Zgodnie z wcześniejszym stwierdzeniem oznacza to, że podzbiór programowy zawiera:

— podzbiór „start program", który jest wykonywany tylko raz,

— dwa lub więcej podzbiorów, które są wykonywane alternatywnie,

— podzbiór „stop programu", który jest wykonywany tylko raz.

Struktura programu i odpowiadający mu organigram w wypadku alternatywnie wykonywanych procesów X i $\bar{X}$ przyjmują formy:



W szczególnym wypadku jeden z procesów może być procesem pustym.

W metodzie Warniera oprócz struktury sekwencyjnej wykorzystywane są struktury powtarzania, gdy podzbiór jest wykonywany N razy, oraz struktura wyboru, gdy podzbiór ma być wykonany jeden raz lub ma nie być podejmowane żadne działanie.

Znając wszystkie możliwe do wykorzystania struktury zastosujemy je w przykładzie. Rozwiążemy za pomocą metody Warniera następujące zadanie.

Przykład 10

Wykonać wydruk listy klientów ze zbioru kart, który zawiera jako pierwszą kartę, rekord główny, a za nim karty transakcji danego klienta. Dla jednego rekordu głównego może wystąpić

od zera do N rekordów transakcji. Rekordy zawierają następujące informacje:

Rekord główny: | nr-klienta | nazwisko | saldo |

Rekord transakcyjny: | nr-klienta | nr-transakcji | wartość | kod |

Pole kod może przyjmować wartość D dla debet, a C dla kredyt. Zbiór kart jest uporządkowany według wzrastających numerów klienta. Żądany wydruk listy klientów ma postać:

NR-KLIENTA	NAZWISKO	NR-TRANSAKCJI	DEBET	KREDYT
SALDO-STARE	SALDO-NOWE		RAZEM	RAZEM

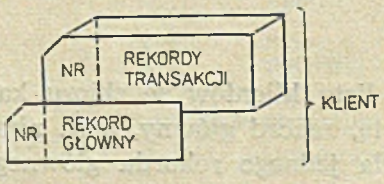
NR-KLIENTA	NAZWISKO
SALDO-STARE	SALDO-NOWE
NR-KLIENTA	NAZWISKO

Jeśli podzbiór transakcji dla jednego klienta nie jest zbiorem pustym, to wartość nowego salda obliczamy według wzoru:

$$\text{NOWE SALDO} = \text{STARE SALDO} + \text{DEBET} - \text{KREDYT}$$

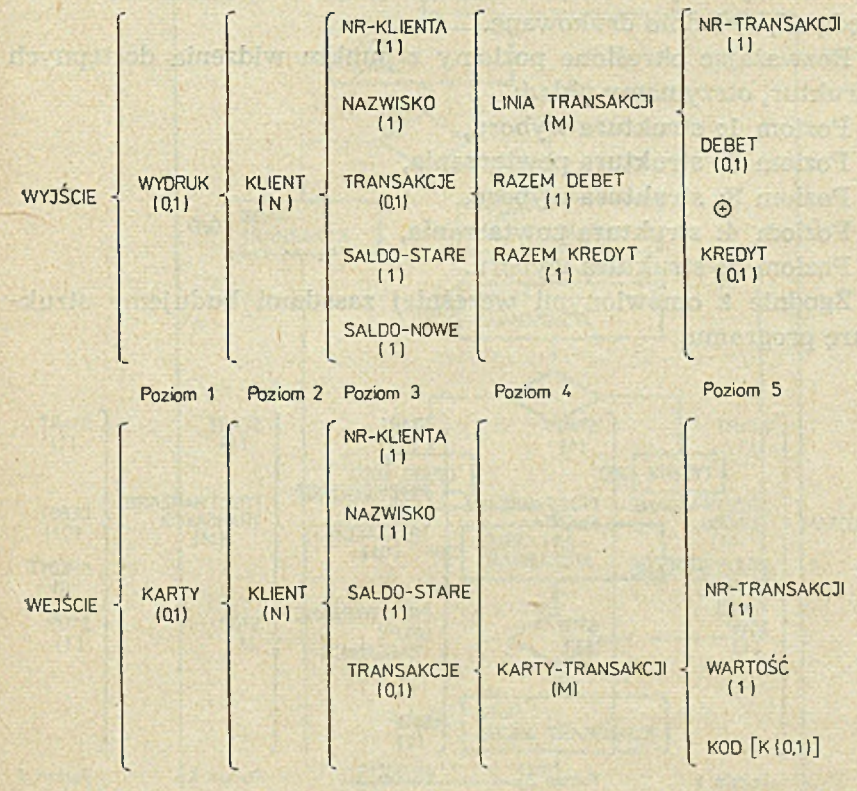
W razie, gdy dla danego numeru klienta brak rekordów transakcyjnych wartości starego i nowego, salda są sobie równe. W takim wypadku wydruk na liście ogranicza się do wydruku dwóch linii zawierających numer klienta i jego nazwisko oraz wartości sald.

Zbiór wejściowy składa się z uporządkowanego zbioru kart:



Wartość, numer klienta występują na wszystkich kartach, co oznacza, że występuje wielokrotnie w zbiorze, ale jeden raz w każdym rekordzie (na jednej karcie).

Przedstawmy strukturę zbioru wyjściowego i wejściowego zgodnie z notacją w metodzie Warniera.



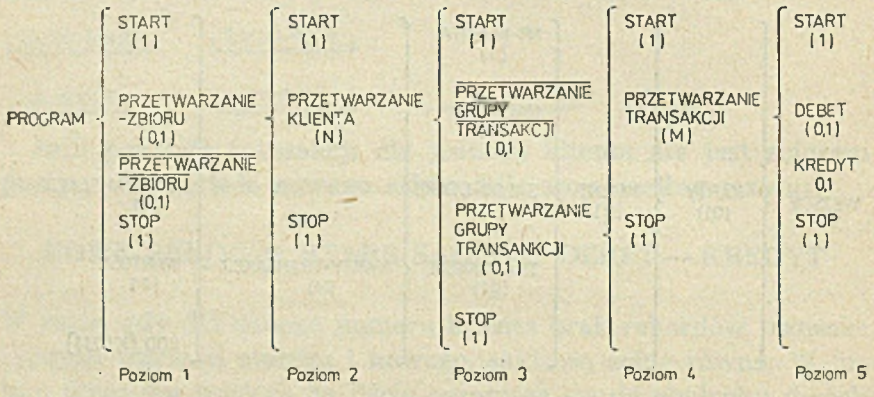
Jak to wynika z przedstawionego schematu w zbiorze wyjściowym i wejściowym wyróżniona została taka sama liczba poziomów. Poziom pierwszy dotyczy zbiorów i w obu wypadkach rozważana jest sytuacja czy zbiór może wystąpić, czy nie. Poziom drugi jest poziomem pojedynczego klienta, który może występować N razy. Poziom trzeci — to informacje o kliencie, gdzie w zbiorze wyjściowym mogą występować lub nie wartości transakcji, jak i w zbiorze wejściowym mogą występować lub nie karty transakcji. Poziom czwarty dotyczy jednej linii transakcji na

wydruku oraz jednej karty w zbiorze kart transakcyjnych. Poziom najniższy piąty odnosi się do informacji elementarnych, przy czym w zbiorze wyjściowym alternatywnie występują pola DEBET i KREDYT (co zostało oznaczone znakiem $\oplus$), natomiast w zbiorze wejściowym pole KOD może przyjąć dwie wartości, co oznaczyliśmy K (0,1). W zależności od wartości kodu wiersze będą odpowiednio drukowane.

Rozważając określone poziomy z punktu widzenia dostępnych struktur, otrzymamy układ:

Poziom 1: struktura wyboru,
Poziom 2: struktura powtarzania,
Poziom 3: struktura wyboru,
Poziom 4: struktura powtarzania,
Poziom 5: struktura wyboru.

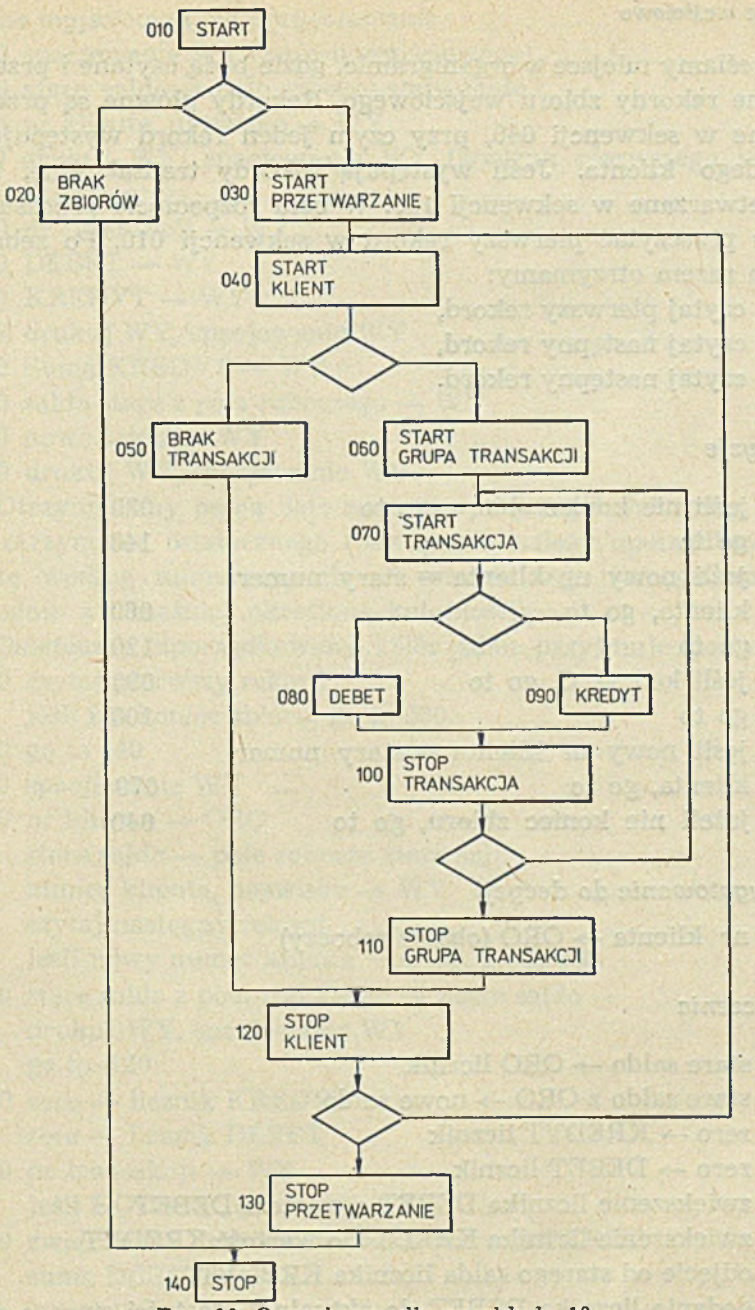
Zgodnie z omówionymi wcześniej zasadami budujemy strukturę programu:



Poszczególne poziomy zostały rozpisane zgodnie z wcześniej określonymi strukturami, przy czym każdy z nich został uzupełniony o podzbiór START i STOP.

W kolejnym kroku metody Warniera na podstawie przedstawionej struktury programu budujemy organigram (por. rys. 18).

Kolejny krok metody polega na konstruowaniu uporządkowanego zbioru zdań składających się na program. Rozważmy postępowanie zgodnie z wcześniej określonym porządkiem.



Rys. 18. Organigram dla przykładu 10

Dane wejściowe

Określamy miejsce w organigramie, gdzie będą czytane i przetwarzane rekordy zbioru wejściowego. Rekordy główne są przetwarzane w sekwencji 040, przy czym jeden rekord występuje dla każdego klienta. Jeśli występują rekordy transakcyjne, to są przetwarzane w sekwencji 100. W celu rozpoczęcia procesu, należy przeczytać pierwszy rekord w sekwencji 010. Po zebraniu tego razem otrzymamy:

010 czytaj pierwszy rekord,
040 czytaj następny rekord,
100 czytaj następny rekord.

Decyzje

010 jeśli nie koniec zbioru, go to 030
020 go to 140
040 jeśli nowy nr klienta = stary numer klienta, go to 060
050 go to 120
070 jeśli kod = C, go to 090
080 go to 100
100 jeśli nowy nr klienta = stary numer klienta, go to 070
120 jeżeli nie koniec zbioru, go to 040

Przygotowanie do decyzji

040 nr klienta → ORO (obszar roboczy)

Obliczenia

040 stare saldo → ORO licznik
050 stare saldo z ORO → nowe saldo
060 zero → KREDYT licznik
060 zero → DEBET licznik
080 zwiększenie licznika DEBET o wartość DEBET
090 zwiększenie licznika KREDYT o wartość KREDYT
110 odjęcie od starego salda licznika KREDYT
110 dodanie licznika DEBET do aktualnej wartości starego salda

Dane wyjściowe i ich przygotowanie
030 spacjowanie WY (buforu wyjściowego)
040 stare saldo → pole robocze stare saldo
040 nr klienta, nazwisko → WY
050 drukuj WY, spacjowanie WY (wydruk pierwszego wiersza dla klienta)
070 nr transakcji → WY
080 DEBET → WY
090 KREDYT → WY
100 drukuj WY, spacjowanie WY
110 Suma KREDYT → WY
120 saldo stare z pola roboczego → WY
120 nowe saldo → WY
120 drukuj WY, spacjowanie WY

Otrzymaliśmy pełną listę zdań dla poszczególnych grup; w celu otrzymania ostatecznego rozwiązania należy uporządkować tę listę według numeracji, ale w ramach tego samego numeru, zgodnie z wcześniej określoną kolejnością.

Ostateczny uporządkowany zbiór zdań przyjmuje postać:

```
010 czytaj pierwszy rekord
    jeśli nie koniec zbioru, go to 030
020 go to 140
030 spacjowanie WY
040 nr klienta → ORO
    stare saldo → pole robocze stare saldo
    numer klienta, nazwisko → WY
    czytaj następny rekord
    jeśli nowy numer klienta = ORO, go to 060
050 stare saldo z pola roboczego → nowe saldo
    drukuj WY, spacjowanie WY
    go to 120
060 zero → licznik KREDYT
    zero → licznik DEBET
070 nr transakcji → WY
    jeśli KOD = C, go to 090
080 zwiększenie licznika DEBET o wartość DEBET
    suma DEBET → WY
    go to 100
```

090 zwiększenie licznika KREDYT o wartość KREDYT
suma KREDYT → WY
100 drukuj WY, spacjowanie WY
czytaj następny rekord
jeśli nowy numer klienta = ORO, go to 070
110 odjęcie od starego salda licznika KREDYT
dodanie licznika DEBET do aktualnej wartości
starego salda
suma DEBET → WY
suma KREDYT → WY
120 stare saldo z pola roboczego →WY
nowe saldo → WY
drukuj WY, spacjowanie WY
jeśli nie koniec zbioru, go to 040
130 koniec przetwarzania
140 koniec programu.

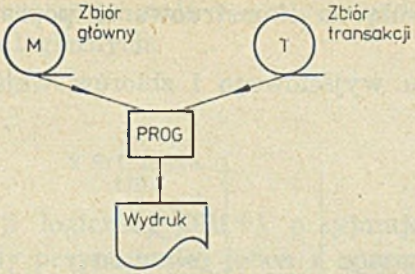
Po otrzymaniu tak uporządkowanego zbioru zdań, praktycznie mamy ostateczne rozwiązanie, gdyż zamiana każdego ze zdań na instrukcje programowe jest bardzo łatwa przez użycie języka programowania wyższego rzędu.

d. Struktury złożone

W dotychczas rozważanych przykładach mieliśmy do czynienia z prostymi strukturami wyboru i powtarzania, gdyż w obu wypadkach na wejściu i na wyjściu mieliśmy po jednym zbiorze, w których można było wyróżnić jednakową liczbę zbliżonych poziomów. W praktyce dość często występują bardziej złożone problemy, które powstają już w momencie wystąpienia dwóch zbiorów na wejściu. Przy rozwiązywaniu takich problemów należy korzystać ze złożonych struktur wyboru.

Przykład 11

Rozważmy przykład programu o przedstawionym schemacie na s. 147.



Zbiór główny może zawierać lub nie zawiera (0,1) rekord identyfikowany przez pole INDEKS o strukturze ogólnej:

| INDEKS | DANE |

Podobną strukturę mają rekordy w zbiorze transakcji, które również mogą, ale nie muszą, wystąpić.

W tabulogramie mogą występować 3 typy wierszy:

— wiersz typu X zawiera dane z rekordu-transakcji, dotyczącej INDEKSU, dla którego nie występuje rekord w zbiorze głównym ($M\bar{T}$),

— wiersz typu Y zawiera dane z rekordu, dla którego występuje rekord w zbiorze głównym, a nie występuje odpowiadający mu rekord w zbiorze transakcji ($\bar{M}T$),

— wiersz typu Z zawiera dane z rekordów, dla których zarówno w zbiorze głównym, jak i w zbiorze transakcji występują rekordy o takim samym polu INDEKS (MT).

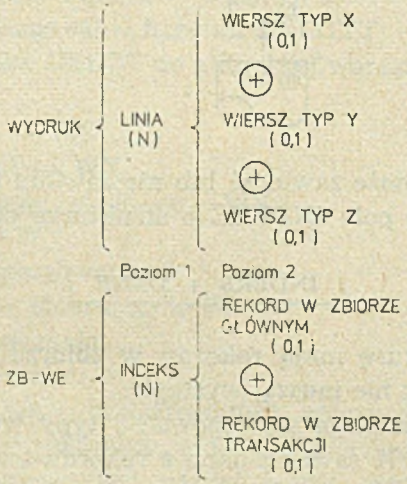
Rekordy w obu zbiorach są uporządkowane według pola INDEKS w porządku rosnącym.

Przedstawimy teraz schemat strukturalny dla zbiorów wejściowych i wyjściowych.

W wypadku zbiorów wyjściowych i wejściowych można znaleźć proste przejście z wejścia na wyjście na poziomie 1. Na poziomie 2 przejście takie nie jest proste, gdyż w zbiorze wyjściowym występuje jeden z wzajemnie się wykluczających zapisów; natomiast w zbiorze wejściowym występują lub nie występują rekordy dla danego indeksu w zbiorze głównym i zbiorze transakcyjnym lub tylko w jednym z nich. Rekordy te nie wykluczają się wzajemnie. Okazuje się, że znane dotychczas metody postę-

powania nie pozwolą na skonstruowanie poprawnej hierarchii programu.

Struktura zbioru wyjściowego i zbiorów wejściowych:



Do znanych już zasad:

— powtarzalne struktury danych dają w rezultacie programową strukturę powtarzania,

— alternatywne struktury danych dają w rezultacie programową strukturę wyboru,

dochodzi nowa zasada:

— dla konstrukcji programu ze złożonymi strukturami danych należy generować tablice decyzji.

Mówiąc inaczej struktura programu wywodzi się bezpośrednio ze struktury zbiorów wejściowych i wyjściowych z wyjątkiem wypadku występowania złożonych struktur danych.

Metoda Warniera oparta jest na matematycznych podstawach teorii zbiorów [10]. W metodzie wykorzystuje się ponadto algebrę Boole'a ze zmiennymi 0 (fałsz) i 1 (prawda) oraz zasady programowania strukturalnego.

Mają zastosowanie operacje logiczne AND, OR i NOT. Wynik operacji logicznej AND (.) przyjmuje wartość prawda (1) tylko

[10] Por. K. Kuratowski, *Wstęp do teorii mnogości i topolgii*, Warszawa 1977, rozdz. I i II.

wtedy, gdy oba operandy mają wartość prawda (1). Przedstawia to tablica działań logicznych:

$0 . 0 = 0$
$0 . 1 = 0$
$1 . 0 = 0$
$1 . 1 = 1$

Wynik operacji logicznej OR(+) przyjmuje wartość prawda (1) wtedy, gdy przynajmniej jeden z operandów ma wartość prawda (1).

$0+0 = 0$
$0+1 = 1$
$1+0 = 1$
$1+1 = 1$

Wynik operacji logicznej NOT (¯) polega na negacji prawdy (1), co daje fałsz (0) i na odwrót. Oznacza to:

$\bar{0} = 1$
$\bar{1} = 0$

Przy konstrukcji wyrażeń logicznych przestrzegane są zasady kolejności działań zgodnie z hierarchią:

— grupowanie /( )/: nawiasy,
— negacja (–): operacja NOT,
— iloczyn logiczny (.): operacja AND,
— suma logiczna (+): operacja OR.

Ponadto w metodzie Warniera wprowadzone są terminy:
MAKS: sekwencja zmiennych boolowskich powiązanych operatorem
OR, (np. $A+\bar{B}+\bar{C}+D$)
MIN: sekwencja zmiennych boolowskich powiązanych operatorem AND (np. $\bar{A} . B . C$)
Każde wyrażenie logiczne może być opisane jako iloczyn MAKS lub suma MIN.

Dla przykładu:
Jeżeli przyjmiemy oznaczenia:
A — rekord główny,
B — rekord transakcyjny,
C — rekord wynikowy,
to w sytuacji gdy: występuje rekord główny i nie występuje re-

kord transakcyjny, a należy wprowadzić rekord wyjściowy, można zapisać jako:

$MIN = A \cdot \bar{B} \cdot C$

Dla lepszego zrozumienia powiązań teorii zbiorów z algebrą Boole'a rozważmy następujący przykład:

Przykład 12

Niech X będzie zbiorem składającym się z wykładowców i studentów Politechniki

$X = \{$wykładowcy i studenci Politechniki$\}$

Niech S i W będą dwoma podzbiorami X, takimi że:

$S = \{$studenci$\}$,

$K = \{$kobiety$\}$.

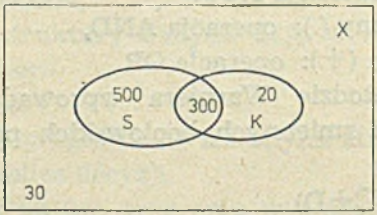
Jak łatwo zauważyć, zachodzą relacje zawierania się:

$S \subset X$ i $K \subset X$

Niech ponadto skład zbioru X będzie następujący:

— 30 wykładowców (mężczyzn)
— 20 wykładowców (kobiet)
— 500 studentów (mężczyzn)
— 300 studentów (kobiet)

Można to przedstawić w postaci diagramu Venna:



Następnie diagram Venna można przekształcić w odpowiadający mu diagram:

	$\bar{S}$	S
K	20	300
$\bar{K}$	30	500

Można to również zapisać jako MIN:

$\bar{K} \cdot \bar{S}$ — mężczyźni wykładowcy
$\bar{K} \cdot S$ — mężczyźni studenci

$K \cdot \bar{S}$ — kobiety wykładowcy
$K \cdot S$ — kobiety studenci

Jak to widać każdy MIN koresponduje z jednym kwadratem diagramu.

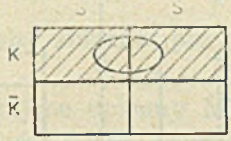
Można stwierdzić, że dla dwóch podzbiorów będziemy mieli 2^2, czyli 4 kwadraty w diagramie lub ogólnie dla n podzbiorów 2^n kwadratów w odpowiednim diagramie.

Wyrażenia boolowskie opisujące konkretną sytuację można zredukować wykorzystując prawa algebry zbiorów.

Dla przykładu, wyrażenie:

$$\bar{S} \cdot K + S \cdot K = K,$$

można zinterpretować graficznie:



Dla przedstawionego przykładu, z faktu związków i przecięć podzbiorów korespondujących wzajemnie oraz zasad algebry Boole'a wynika następujący wzór:

$$X = \{\bar{K} \cap \bar{S}\} \cup \{\bar{K} \cap S\} \cup \{K \cap \bar{S}\} \cup \{K \cap S\}.$$

W zależności od liczby występujących zbiorów otrzymujemy bardziej skomplikowane diagramy, które reprezentują tablicę decyzji, opisującą wszystkie możliwe stany zbiorów.

Diagram dla 1 zbioru ma postać:

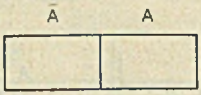


Diagram dla 2 zbiorów przedstawiliśmy w przykładzie, dla 3 zbiorów ma postać:

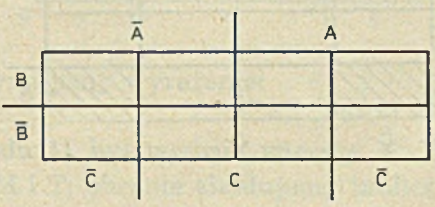


Diagram dla 4 zbiorów składa się już z 16 kombinacji:

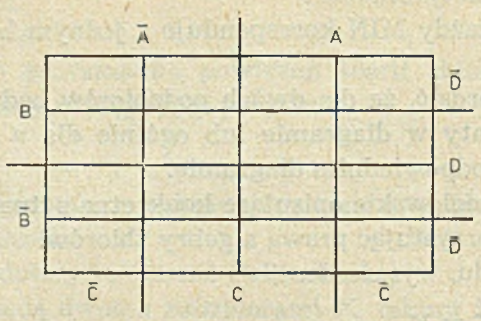
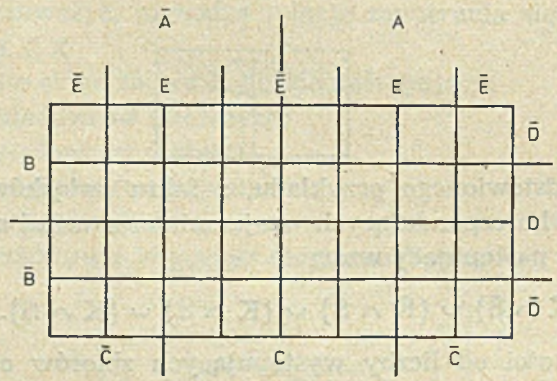
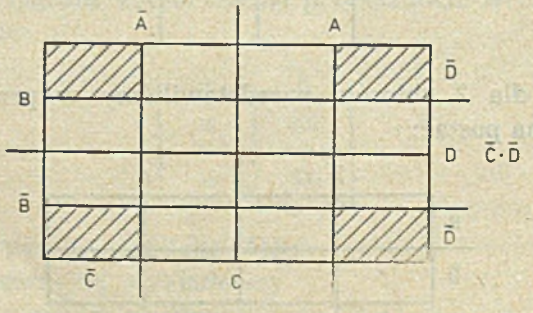
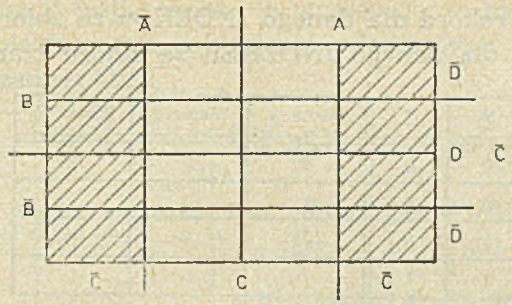


Diagram dla 5 zbiorów ma postać:



W celu wyjaśnienia sposobu wykorzystania diagramów oznaczymy dla przykładu kilka wyrażeń boolowskich na diagramie dla 4 zbiorów:





Wykorzystując diagramy dla opisu stanów zbiorów należy stosować następujące zasady:

— wyrażenia boolowskie mogą być przekształcane w sumy MIN przez użycie prawa negacji (znanych pod nazwą praw de Morgana),

— w zależności od liczby zbiorów przygotowujemy odpowiedni diagram,

— kwadraty reprezentujące opisany MIN należy zaciemnić,

— zaciemnione kwadraty należy w miarę możliwości grupować,

— grupy opisane w postaci MIN należy łączyć za pomocą operatora +

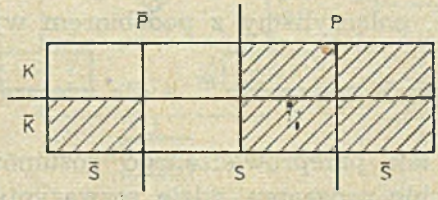
Dla przykładu rozważmy, jak można zredukować następujące wyrażenie:

$$P \cdot K + \overline{K+S} + P \cdot \bar{K} \cdot S$$

Przedstawimy to wyraźnie w formie sumy MIN:

$$P \cdot K + \bar{K} \cdot \bar{S} + P \cdot \bar{K} \cdot S$$

i zobrazujemy je na diagramie:



Po redukcji otrzymujemy wyrażenie:

$$P + \bar{P} \cdot \bar{S} \cdot \bar{K}$$

Celem przykładu 11 był wydruk wierszy X, Y i Z dla rekordów ze zbiorów M i T; obecnie zbudujemy tablicę decyzji, w któ-

rej M oznacza rekord dla danego INDEKSu ze zbioru głównego, a T — rekord dla danego INDEKSu ze zbioru transakcyjnego.

M	T	Wiersz X	Wiersz Y	Wiersz Z
0	0	0	0	0
0	1	1		
1	0		1	
1	1			1

Zbudujemy teraz diagram dla rekordów z dwóch zbiorów M i T:

	M	$\bar{M}$
$\bar{T}$	Y	—
T	Z	X

W diagramie przedstawione są 4 różne sytuacje dla rekordów ze zbiorów M i T. Rekordy z obu wymienionych zbiorów mogą występować na wejściu lub nie. Dla danego INDEKSu, w zależności od wartości logicznych przyporządkowanych parze odczytanych rekordów drukujemy wiersz X, Y lub Z. Jeżeli dla danego INDEKSu nie występuje ani rekord ze zbioru M, ani rekord ze zbioru T, to nie podejmujemy żadnej akcji, czyli jest to podzbiór pusty. Wykorzystując definicję zadania oraz wprowadzone wcześniej pojęcia, możemy zdefiniować sytuacje dla drukowania odpowiednich wierszy na podstawie tablicy decyzji, za pomocą odpowiedniej sumy MIN.

Dla wiersza typu X mamy:

$\bar{M} \cdot T + \bar{M} \cdot T = \bar{M}$

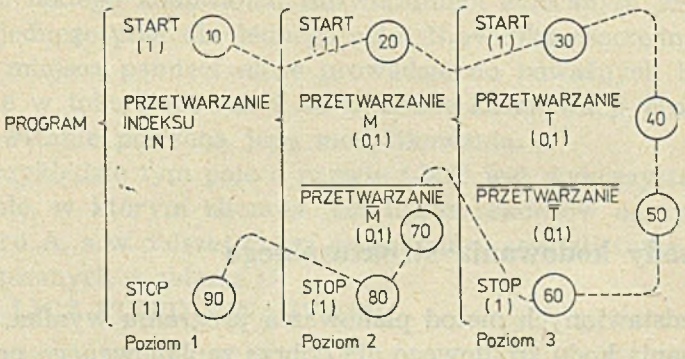
Jak to wynika z zapisu, podzbiór pusty, nie mający wpływu na przetwarzanie, połączyliśmy z podzbiorem wydruku wiersza typu X.

Dla wiersza typu Y mamy $M \cdot \bar{T}$, a dla wiersza typu Z — odpowiednio $M \cdot T$.

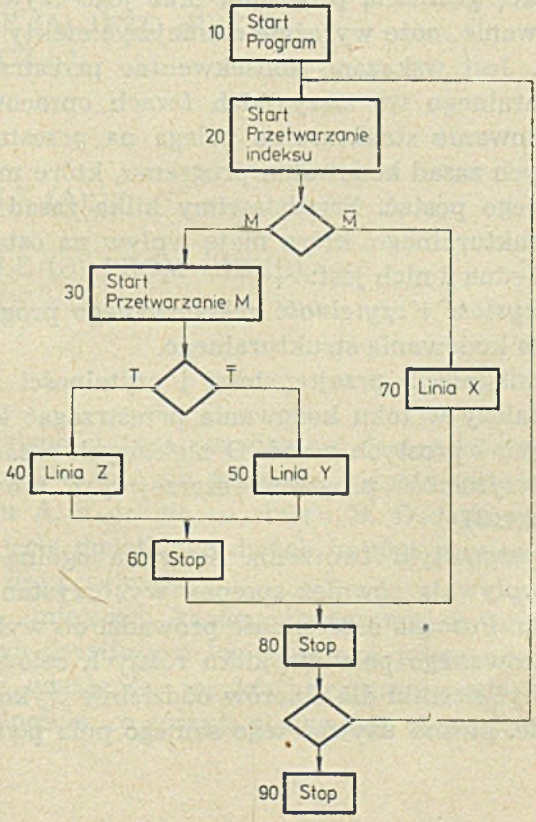
Na podstawie tak przeprowadzonego rozumowania możemy zbudować hierarchię programu, gdzie rozważymy wszystkie sytuacje wynikające z tablicy decyzji (p. s. 155).

Na poziomie 2 rozważamy sytuację występowania lub nie rekordu w zbiorze M. Jeśli rekord występuje, to taka sytuacja wymaga dalszego rozważenia i stąd wywodzi się poziom 3; nato-

miast w sytuacji jeśli rekord nie występuje, to nie wymaga dalszego pogłębiania, gdyż drukujemy wiersz typu X lub nie podejmujemy żadnej akcji.



Wykorzystując naniesioną na hierarchię programu numerację łatwo można zbudować odpowiedni organigram i wykonać kolejne kroki metody Warniera aż do otrzymania listy zdań.



V. Kodowanie programu

1. Zasady kodowania strukturalnego

Z przedstawionych metod planowania programu wynika, że proces pisania kodu źródłowego dla dobrze zaplanowanego programu jest czynnością prostą i ma niewielki wpływ na jego efektywność użytkową. Sposób kodowania ma jednak istotne znaczenie, jeśli chodzi o szatę graficzną programu oraz jego czytelność. Niestaranne kodowanie może wyraźnie zmniejszyć efekty prawidłowego planowania. Jest wskazane konsekwentne przestrzeganie podejścia strukturalnego we wszystkich fazach opracowywania programu. Kodowanie strukturalne polega na przestrzeganiu pewnych ogólnych zasad kodowania programu, które mają wpływ na ostateczną jego postać. Przedstawimy kilka zasad ogólnych kodowania strukturalnego, które mają wpływ na ostateczną postać programu. Jedną z nich jest:

— *przejrzystość i czytelność* zakodowanego programu — jako istotna cecha kodowania strukturalnego.

W celu osiągnięcia przejrzystości i czytelności zakodowanego programu należy w toku kodowania przestrzegać konsekwentnie kilku znanych i prostych zasad. O niektórych zasadach wpływających na czytelność programu szerzej już mówiliśmy (por. rozdz. III, pkt. 3).

Oprócz systematyki tworzenia nazw na ogólną przejrzystość programu wpływają również sposoby wykorzystania deklarowanych danych. Pozorna oszczędność prowadzi do wykorzystywania raz zadeklarowanego pola dla kilku różnych celów. Przykładem tego mogą być liczniki dla zbiorów oddzielnie wykorzystywanych w programie. Można używać tego samego pola podczas przetwa-

rzania jednego zbioru, a po zakończeniu przetwarzania ponownie je wykorzystać do przetwarzania innego zbioru. Rozwiązanie takie choć poprawne, nie jest zalecane ze względu na małą przejrzystość takiego kodowania. Rozwiązaniem zalecanym jest używanie jednego pola dla jednego celu. Niewielka oszczędność zajętości miejsca pamięci może prowadzić do poważnych błędów, jeśli nie w toku kodowania, to w czasie konserwacji programu, gdy powstanie potrzeba jego modyfikowania.

W przykładzie tym pole o nazwie LICZ jest wykorzystane raz jako pole, w którym zliczana jest liczba rekordów odczytanych ze zbioru A, a w dalszej części programu w celu zliczania rekordów zapisanych w zbiorze B.

```
DCL LICZ FIXED BINARY (15);
   .
   .
   .
LICZ = 0;
READ FILE (A) INTO (REKA);
LICZ = LICZ+1;
   .
   .
   .
CLOSE FILE (A);
LICZ = 0;
WRITE FILE (B) FROM (REKB);
LICZ = LICZ+1;
   .
   .
   .
```

Mimo poprawności tego rozwiązania jest zalecane zadeklarowanie dwóch pól o różnych nazwach, jedno w celu zliczenia rekordów zbioru A, a drugie — zbioru B. Dodatkowa deklaracja spowoduje zajęcie dodatkowo dwóch bajtów pamięci, ale uczyni program bardziej przejrzystym.

Kluczowym miejscem każdego programu są instrukcje warunkowe, które decydują o logice całego programu. Kodowanie warunków, w większości języków programowania, charakteryzuje się dowolnością, co sprzyja stosowaniu złożonych warunków.

Program kodowany z wykorzystaniem warunków złożonych jest mało przejrzysty i trudny do analizowania. W strukturalnych zasadach kodowania uwzględnia się różne sposoby kodowania warunków, ale jest wskazane unikanie stosowania skomplikowanej składni warunków. Natomiast w razie ich użycia zaleca się stosowanie nawiasów, które czynią zapisy bardziej przejrzystymi i pozwalają na uniknięcie wieloznaczności.

Zaleca się niestosowanie etykiet programowych. Sekwencja instrukcji realizujących podstawowe struktury powinna być wykonywana konsekwentnie z góry w dół i tylko podprogramy muszą być w odpowiedni sposób nazwane, aby można je było wywołać. Brak etykiet nie utrudnia kodowania, a skutecznie eliminuje instrukcję skoku bezwarunkowego, która zakłóca sekwencję góra—dół i jest potencjalnym źródłem trudnych do wykrycia błędów. Etykietowanie może mieć zastosowanie przy odpowiedniej redakcji kodowanego programu. Zastosowane nazwy dla odpowiednich partii kodu mogą zastępować komentarz lub można je wykorzystywać do opisu programu.

W celu zwiększenia przejrzystości programu jest wskazane również kodowanie jednego zdania w jednej linii. Nie tylko podnosi to przejrzystość programu, ale jest wygodne przy poprawianiu i modyfikowaniu programu.

Opisując zbiór kartowy w języku COBOL można to uczynić na dwa równoważne sposoby:

```
FD KARTA RECORDING MODE IS F LABEL RECORDS
    ARE OMITTED
    RECORD CONTAINS 80 CHARACTERS DATA RECORD
    IS KARTA-1.
```

lub

```
FD KARTA
    RECORDING MODE IS F
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS KARTA-1.
```

Oba opisy są poprawne i równoważne, ale drugi sposób zapisu jest bardziej czytelny i łatwiejszy od modyfikowania, gdyż każdy atrybut zbioru opisany jest w oddzielnym wierszu.

a. Sposób wykorzystania języka programowania

Języki programowania w różny sposób wspomagają stosowanie zasad programowania strukturalnego, o czym dokładniej będziemy mówić dalej (por. pkt. 2). Niezależnie od podstawowego repertuaru instrukcji dostępnych w konkretnym języku programowania, istnieją liczne możliwości specjalne. Kodowanie programu w konkretnym języku programowania pozwala oczywiście na pełne wykorzystanie jego możliwości. Najczęściej jednak tylko podstawowy podzbiór języka jest powszechnie znany i wystarcza do kodowania większości programów. Wykorzystanie specjalnych możliwości języka wyższego rzędu tam, gdzie można posłużyć się podzbiorem podstawowym, jest często stosowane bez uzasadnienia. Rozwiązania takie mogą się okazać bardzo kłopotliwe wówczas, gdy zajdzie potrzeba zmodyfikowania programu po pewnym czasie, gdyż nie będzie wiadomo czy rozwiązanie takie jest niezbędne właśnie w tym konkretnym wypadku. Możliwości specjalne powinny być wykorzystywane tylko wtedy, gdy podzbiór podstawowy języka nie wystarcza do zakodowania konkretnej sekwencji instrukcji. We wszystkich pozostałych wypadkach jest zalecane wykorzystywanie podstawowego podzbioru języka.

Przykładem „udziwniania" programu może być wykonanie mnożenia i dzielenia pól przez potęgi dwójki za pomocą instrukcji arytmetycznego przesuwania. W celu pomnożenia zawartości rejestru 5 przez 16 przy wykorzystaniu języka programowania ASSEMBLER można napisać:

```
SLA 5, 4
```

chociaż bardziej wskazane, ze względu na czytelność programu, jest zapisanie tego za pomocą instrukcji:

```
M 5, ALA
ALA DC F '16'
```

W toku opracowywania oprogramowania systemu liczne funkcje cząstkowe są identyczne lub podobne. W takim wypadku jest wskazane kodowanie ich w postaci podprogramów lub funkcji wbudowanych. Utworzenie biblioteki systemu zawierającej podprogramy i funkcje wbudowane wielokrotnego użycia nie tylko podnosi sprawność programowania, ale również zmusza do sto-

sowania podobnego stylu programowania przez wszystkich programistów.

Najprostszym przykładem podprogramu, który może być wykorzystywany przez kilka programów projektowego systemu jest podprogram drukowania nagłówka tabulogramu. Opracowanie takiego podprogramu zwalnia innych programistów od powtarzania tej samej pracy oraz umożliwia przyjęcie jednolitej formy i standardu dla wszystkich tabulogramów systemu.

Kompilatory większości języków programowania obok diagnostyki błędów formalnych wyprowadzają również informacje ostrzegawcze o możliwości powstania błędów. Ostrzeżenia kompilatora nie powodują wstrzymania dalszej obróbki programu, co często skłania programistów do ignorowania informacji ostrzegawczych. Komunikaty ostrzegawcze powinny być dokładnie analizowane, co ustrzeże programistę przed niespodziankami na etapie testowania.

b. Mikroefektywność

Tak samo postawiony problem można rozwiązać na wiele różnych poprawnych sposobów. Nowe rozwiązania często nasuwają się w toku procesu programowania. Usprawnianie niezakończonego programu najczęściej doprowadza do wydłużenia czasu testowania, spowodowanego chęcią zastąpienia jednego rozwiązania innym, bardziej optymalnym. Optymalizowanie rozwiązań przed ich pełnym sprawdzeniem nie jest zalecane. Natomiast jest zalecane korzystanie z optymalizacji programu wykonywanej przez kompilator. Optymalizacja programu w toku jego kodowania powinna być wykonywana tylko wtedy, gdy jest to bezwzględnie konieczne. Z praktyki wynika, że najprostsze algorytmy są najszybsze.

Podnoszenie sprawności programu, jeśli jest stosowane, powinno być oparte na konkretnych obliczeniach, a nie na przypuszczeniach, które często są mylne. Podnoszenie sprawności programu nie może być realizowane kosztem czytelności. Najlepiej ilustruje to znany z literatury przykład inicjalizacji diagonalnej macierzy jednostkowej kodowany w języku FORTRAN.

```
      DO 10 I = 1,N
      DO 10 J = 1,N
10    X (I,J) = (I/J) * (J/I)
```

Podnoszenie sprawności rozwiązań nie powinno być wykonywane w toku tworzenia kodu programu, ale dopiero po osiągnięciu poprawności działania.

c. Komentarze w programie źródłowym

Języki programowania pozwalają na komentowanie poszczególnych instrukcji, zmiennych, czy też części programu. Możliwość ta powinna być stosowana z umiarem. Zarówno brak komentarzy nie podnosi czytelności programu, jak i ich nadmiar. Stosując komentarze należy pamiętać, że komentarz powinien być odpowiedzią dla czytającego program. Nie powinien omawiać instrukcji komentowanej, ale uzasadniać przyczyny jej użycia lub spodziewane skutki. Komentarze należy wstawiać zawsze tam, gdzie jest niejasne, co robi się w danym miejscu programu.

Szczególnie przydatne jest komentowanie zmiennych używanych w programie. Stosowanie komentarzy w połączeniu z omówionymi już zasadami tworzenia nazw oraz wykorzystania pól eliminują liczne błędy logiczne, wynikające z nieprawidłowego wykorzystania deklarowanych danych.

Zasady umieszczenia komentarza w ciągu zdań źródłowych programu umożliwiają wstawienie go w to miejsce, gdzie jest najlepiej czytelny. Komentarz nie powinien jednak zlewać się z kodem źródłowym. Fizyczne rozdzielenie komentarza od kodu programu jest elementem podnoszącym przejrzystość programu.

W tym fragmencie przykładu programu kodowego w języku PL/1 prawidłowo skomentowane zostały dane oraz instrukcja organizująca pętlę:

```
   DCL
     WSKA FIXED BINARY (15)    /* WSKAZNIK KONCA
                                  ZBIORU A */
   , LICZ1 FIXED BINARY (15)   /* LICZNIK REKORDOW
   , 1 REKA                      ZBIORU A */
                               /*STRUKTURA REKORDU
                                  ZBIORU A */
```

```
    , 2  POL1 PIC '99'
    , 2  POL2 CHAR (78)
.
.
.
DO  WHILE (WSKA = 0);          /* POWTARZANIE OBLICZEN
                                  DO WYCZERPANIA
                                  REKORDOW ZBIORU A */
```

d. Deklaracja danych

Niektóre problemy związane z wykorzystaniem danych już omówiliśmy wcześniej; istotną jednak sprawą jest sama deklaracja danych wykorzystywanych w programie. Języki programowania umożliwiają niedeklarowanie niektórych zmiennych wykorzystywanych w programie. Z udogodnienia tego nie należy korzystać zbyt często; tak samo przyjmowanie niektórych atrybutów deklarowanych zmiennych przez domniemanie może prowadzić do trudnych do wykrycia błędów logicznych. Tylko deklaracja wszystkich wykorzystywanych danych z kompletnym opisem atrybutów umożliwia prawidłowe wykorzystanie danych w programie.

Najczęściej w programie są deklarowane zmienne, natomiast wartości stałe wykorzystywane są w postaci literali. Rozwiązanie takie nie jest zalecane z wielu powodów. Jest wskazane deklarowanie wartości stałych i wykorzystywanie ich przez nadaną nazwę w instrukcjach programu.

Innym dość często stosowanym sposobem jest nadawanie wielu nazw dla tego samego obszaru pamięci. Języki programowania dostarczają odpowiednich instrukcji, które umożliwiają wielokrotne nazywanie tego samego obszaru. Rozwiązanie to jest dość efektywne, ale nie wpływa na poprawność logiczną programu, jak i jego czytelność. Również w sytuacji odwrotnej, polegającej na użyciu jednej nazwy dla wielu obszarów, nie jest zalecane z powodu braku jednoznaczności interpretacji.

W programach wykorzystujących procedury wewnętrzne jest wskazane, aby każda procedura miała własne, lokalne deklaracje danych, a łączność z innymi procedurami była utrzymywana przez wyraźnie deklarowane obszary wspólne. Rozdzielanie de-

klaracji między moduły składające się na program, jest jedną z głównych zasad kodowania strukturalnego. Stosowanie danych lokalnych wydatnie podnosi czytelność zakodowanego programu.

Częstym źródłem powstawania błędów w programie są konwersje danych. Przy kodowaniu należy tak wykorzystywać deklarowane dane, aby unikać zbędnych konwersji, nie mieszać typów operacji i zawsze pamiętać o atrybutach danych, wykorzystywanych w instrukcjach, w połączeniu z zasadami stosowanej arytmetyki. Eliminowanie zbędnej konwersji danych dzięki odpowiedniemu doborowi atrybutów oraz instrukcji nie tylko podnosi czytelność programu, ale również wydatnie wpływa na szybkość realizacji programu.

W tym przykładzie pole o nazwie SUMPOL2 powinno być zadeklarowane analogicznie jak pole POL2, gdyż jest ono wykorzystywane do otrzymania sumy pól POL2 i deklaracja z atrybutem PICTURE nie jest niczym uzasadniona.

```
   DCL
     1 STR
,      2 POL1 CHAR (7)
,      2 POL2 FIXED BINARY (15)
,      2 POL3 CHAR (15)
,   SUMPOL2 PIC '99' INIT (0)
,   A FILE RECORD
;

   OPEN FILE (A) INPUT;
     .
     .
     .
   READ FILE (A) INTO (STR);
   SUMPOL2 = SUMPOL2 + POL2;
     .
     .
     .
   IF SUMPOL2 < 100 THEN...
     .
     .
     .
```

e. Struktura programu

W programowaniu strukturalnym szczególnego znaczenia nabiera struktura programu, która jest tworzona na etapie planowania programu. W toku kodowania są podejmowane decyzje na niższym poziomie szczegółowości, ale ich znaczenie jest równie ważne dla kształtu całości programu.

Na etapie planowania programu definiowane są moduły, z których program będzie zbudowany. Najczęściej jest to najmniejsza jednostka analizowana na tym etapie prac. Struktura modułu jest natomiast określana w toku jego kodowania. W trakcie kodowania modułu nie należy organizować więcej niż jedno wejście i jedno wyjście. Wykonanie odpowiednich deklaracji w kodzie źródłowym modułu może zmuszać do wejścia do modułu i wyjścia z niego w ściśle określonych miejscach. Deklaracje dodatkowych punktów wejścia, jak również organizowanie różnych punktów wyjścia z modułu, mogą skomplikować strukturę dobrze zaplanowanego programu.

Efektywnym chwytem programowym jest kodowanie programu, który sam się modyfikuje w czasie pracy. Mimo efektywności takiego rozwiązania, jak i dostępnych instrukcji (np. instrukcja ALTER dostępna w języku COBOL), chwyt ten wprowadza do kodowanego programu potencjalne źródło błędów trudnych do wykrycia. Należy raczej rezygnować z tego typu możliwości.

Innym źródłem trudnych do znalezienia błędów są zagnieżdżone zdania IF. Za pomocą jednego zdania typu IF można przez odpowiednią składnię badanego warunku wykonać szereg skomplikowanych badań. Bardziej efektywne okazuje się korzystanie z prostych warunków i ich kolejne logiczne kodowanie. W wypadku kodowania warunków złożonych, bezwzględnie jest zalecane wykorzystanie nawiasów, które nie tylko podnoszą przejrzystość, ale umożliwiają głębsze przeanalizowanie. Istotną sprawą jest wykonanie kompletnych badań, tzn. analizowanie wszystkich przewidywanych do wystąpienia sytuacji. Kodowanie instrukcji wyboru powoduje wykonanie jednej z dwóch dróg programu. Częstym jednak zjawiskiem jest pełne kodowanie tylko w sytuacji, gdy warunek jest prawdziwy, bez umieszczania specjalnej informacji o podjętym działaniu, gdy warunek jest

fałszywy [1]. Kodowanie ELSE dla każdego THEN, nawet gdy jest to zdanie puste, podniesie czytelność programu, jak i zmusi do pełnego analizowania kodowanej sytuacji decyzyjnej.

Instrukcja wyboru kodowana w języku PL/1:

```
IF (A = 0) & (B = 5) THEN CALL OBLICZ;
                     ELSE CALL BLAD;
```

w sposób bardzo czytelny i przejrzysty obrazuje nie tylko jaki warunek jest badany, ale również jasno precyzuje akcje, która będzie podjęta w zależności od wartości logicznej, jaką przyjmie wyrażenie warunkowe. Jeśli procedury o nazwach OBLICZ I BLAD nie są wykorzystywane w innym miejscu programu oraz nie są zbyt długie, to można je opisać bezpośrednio w instrukcji IF jako grupy DO. Jeśli jednak umieszczenie pełnego opisu w instrukcji wyboru miałoby ujemnie wpływać na czytelność tego fragmentu programu, to wskazane jest zakodowanie oddzielnej procedury i wywołanie jej za pomocą instrukcji CALL.

Równie często, jak instrukcje wyboru, wykorzystywane są instrukcje powtarzania. Szczególną uwagę przy ich kodowaniu należy zwracać na wybór instrukcji organizujących pętlę typu DO UNTIL lub typu DO WHILE. Istotne różnice w ich wykonaniu, polegające na miejscu badania warunku, mogą stać się źródłem poważnych błędów [2]. Innym zaleceniem, do tej grupy instrukcji, jest unikanie zamykania wielu zagnieżdżonych pętli za pomocą jednej instrukcji końca pętli.

W przykładzie kodowanym w PL/1 użycie pętli typu DO WHILE spowoduje, że przetwarzanie rekordu wykona się co najmniej jeden raz, co w wypadku zbioru pustego jest poważnym błędem. Błędem jest również przetwarzanie rekordu końca zbioru, które przy takim kodowaniu również będzie realizowane.

```
.
.
.
WSKA FIXED BINARY (15) INIT (0)
.
.
.
```

[1] Język programowania FORTRAN nie daje takich możliwości.

[2] Problem ten omówiony był dokładniej w rozdziale III, pkt. 2.

```
ON ENDFILE (A)  WSKA = 1;
   .
   .
   .
DO  WHILE (WSKA = 0);
READ FILE (A) INTO (REKA);
      .
      .
      .    /* przetwarzanie rekordu */
END;
```

Uwzględniając sposób realizacji pętli typu DO WHILE należy oddzielnie rozwiązać problem pierwszego rekordu zbioru. Przedstawiona sytuacja prawidłowo powinna być zakodowana w sposób następujący:

```
      .
      .
      .
   READ FILE (A) INTO (REKA);
   DO  WHILE  (WSKA = 0);
      .
      .    /* przetwarzanie rekordu */
      .
   READ FILE (A) INTO (REKA);
   END;
```

Wyniesienie pierwszej instrukcji czytania przed pętlę oraz cykliczne czytanie rekordów, po zakończeniu przetwarzania ostatnio przeczytanego, pozwoli na wyeliminowanie wymienionych błędów. Dla zbioru pustego oraz po przeczytaniu rekordu końca zbioru, nie będzie zrealizowane przetwarzanie rekordu.

f. Inne uwagi

Nawet konsekwentne przestrzeganie wymienionych zasad kodowania nie przyniesie spodziewanych efektów, jeśli nie będziemy dbali o szatę graficzną całego programu. Postać źródłowa programu powinna być czytelna również ze względu na stosowaną standaryzowaną zewnętrzną formę kodu. Odpowiednie rozmiesz-

czenie instrukcji na arkuszu programowym, umiejętne stosowanie akapitów, wyraźnie wpłyną na czytelność programu. W językach programowania wyższego rzędu jest dopuszczalne stosowanie skrótów dla słów kluczowych oraz pomijanie słów opcjonalnych. Korzystanie z tych możliwości jest tylko pozorną oszczędnością. Program, w którym są kodowane słowa kluczowe w pełnym brzmieniu wraz z występującymi w kodzie słowami opcjonalnymi, jest dla każdego programisty czytelny i przejrzysty, niezależnie od tego, ile czasu minęło od jego napisania.

Stosowanie przedstawionych zaleceń dla fazy kodowania nie ma tak istotnej roli jak poprawne planowanie programu, ale niestosowanie się do nich może zniweczyć wysiłek planowania poprawnej struktury programu. W przedstawionych zasadach kodowania kładzie się główny nacisk na prostotę, kompletność oraz dbałość o czytelność kodu programu. Cechy te nie wpływają na wykonanie programu, ale są niezwykle istotne w fazie konserwacji, kiedy po pewnym czasie zachodzi potrzeba szybkiego rozpoznania struktury programu w celu dokonania zmiany lub uzupełnienia.

2. Wspomaganie programowania strukturalnego przez języki programowania

Biorąc pod uwagę główne cechy programowania strukturalnego właściwości języka programowania powinny:

— wspomagać trzy podstawowe struktury przebiegu,

— umożliwiać dzielenie programu na małe jednostki kodowe z jednym wejściem i wyjściem,

— dopuszczać stosowanie struktur danych z dostępem ogólnym i ograniczonym (dane typu GLOBAL i LOCAL).

Ponadto wydzielone jednostki programowe powinny dawać się łatwo łączyć i wstawiać jedna w drugą oraz powinny istnieć możliwości udostępniania wspólnych danych oraz mechanizmów wymiany danych.

Przeanalizujemy podstawowe języki programowania w zakresie wspomagania wymienionych cech. Jako podstawowe w rozważaniach porównywane będą najbardziej popularne języki programowania wyższego rzędu COBOL, FORTRAN i PL/1. Nie-

które cechy zostaną porównane również z innymi językami programowania.

Wspomaganie podstawowych struktur przebiegu przedstawiamy w tablicy 1.

TABLICA 1

Wspomaganie struktur podstawowych przez języki programowania

	FORTRAN	COBOL	PL/1	ALGOL	PASCAL	ASSEMBLER
Sekwencja	X	X	X	X	X	X
IF THEN ...	(X)	X	X	X	X	(X)
IF THEN ... ELSE	—	X	X	X	X	—
CASE OF	—	—	—	—	X	—
DO WHILE	—	X	X	X	X	—
DO UNTIL	(X)	—	(X)	(X)	X	(X)

X — możliwość stosowania danej struktury w języku programowania
(X) — występowanie w ograniczonej postaci

Z danych tablicy 1 wynika, że jedynie język programowania PASCAL ma możliwość stosowania wszystkich struktur w sposób bezpośredni, natomiast języki PL/1 i ALGOL nie mają możliwości stosowania wyboru wielokrotnego. Najmniej wspomagają stosowanie struktur podstawowych języki programowania FORTRAN i ASSEMBLER.

W języku FORTRAN struktura wyboru może być stosowana tylko w postaci IF THEN..., gdzie można napisać tylko jedną instrukcję. Wykonanie grupy instrukcji jako wyniku warunku można osiągnąć dzięki odpowiedniemu zastosowaniu instrukcji skokowych. Również struktura powtarzania jest słabo wspomagana, gdyż jedyny dopuszczalny sposób organizacji pętli, to pętle sterowane liczbowo, co pozwala na przerwanie powtarzania jedynie przez licznik.

Organizacja struktur powtarzania w języku COBOL polega na wykorzystaniu instrukcji PERFORM. Jedna z postaci tej instrukcji, dla przerwania powtarzania wykorzystuje warunek typu UNTIL, ale mimo to miejsce badania warunku (przed wykonaniem części powtarzanej) i programowa realizacja pozwalają na konstrukcję struktury typu WHILE.

Język programowania PL/1 nie daje możliwości stosowania wprost struktury typu DO UNTIL, ale można ją łatwo osiągnąć przez użycie pętli DO WHILE z odpowiednio sformułowanym warunkiem:

```
LOG = '1' B;
DO WHILE (LOG);
      .
      .
      .
   IF WAR THEN LOG = '1' B; ELSE LOG = '0' B;
END;
```

Biorąc pod uwagę możliwość dzielenia programu na moduły, wspomaganie ze strony wymienionych języków programowania rozkłada się podobnie.

FORTRAN daje najmniejsze możliwości podziału. Program w języku FORTRAN można podzielić tylko na program główny i podprogramy, ewentualnie podprogramy funkcji, przy czym podprogramy można dzielić głębiej na kolejne podprogramy. Podprogramy funkcji różnią się od innych tym, że mogą one obliczać tylko jedną wartość.

Podprogramy funkcji mają zasadniczo po jednym punkcie wejścia i wyjścia, ale podprogramy mogą mieć kilka wejść (ENTRY) oraz po kilka wyjść (RETURN). Programista chcąc programować strukturalnie w języku FORTRAN musi zrezygnować z tych możliwości.

COBOL dopuszcza więcej możliwości podziału programu. Program można podzielić na sekcje, a te z kolei na paragrafy. Paragraf zawiera jedną lub kilka instrukcji, sekcja — jeden lub kilka paragrafów. Paragrafy można porównać z blokami w PL/1, ale w przeciwieństwie do bloków, paragrafy nie dają się fizycznie wbudować jeden w drugim. Można to osiągnąć przez zastosowanie instrukcji PERFORM:

```
PAR1. ...
PAR2.
PERFORM PAR1 VARYING J FROM 1 BY 1 UNTIL J<N.
...
PAR3
```

Paragraf PAR1 logicznie wbudowany w paragraf PAR2 fizycznie znajduje się na początku programu. Taki sposób zapisu nie przyczynia się do przejrzystości i jest szczególną wadą języka COBOL. Inną wadą tego języka jest to, że nie ma w nim możliwości stosowania podprogramów funkcji, a podprogramy zewnętrzne mogą mieć po kilka wejść i wyjść, podobnie jak w języku FORTRAN.

Niewątpliwą natomiast zaletą języka COBOL jest możliwość stosowania instrukcji COPY, dzięki której można wbudować w kod źródłowy programu sekcje i paragrafy wszędzie tam, gdzie są one potrzebne do wykonania. Stosowanie operatora COPY nie jest klasycznym przykładem fizycznego wbudowania bloku instrukcji, ale w połączeniu z biblioteką zdań źródłowych daje duże możliwości fizycznego rozszerzenia kodu źródłowego o paragraf czy sekcje, pobrane z biblioteki przygotowanej wcześniej. Stosowanie operatora COPY w dziale danych doskonale wspomaga proponowaną wcześniej metodę standardowego nazewnictwa dla zbiorów, rekordów i pól w nich występujących.

PL/1 charakteryzują największe możliwości podziału. Program pisany w PL/1 ma strukturę blokową; mogą występować zarówno bloki proceduralne (PROC), jak i bloki BEGIN czy DO, które dają się fizycznie wstawiać jeden w drugi. Bloki proceduralne mogą być względem siebie zewnętrzne lub wewnętrzne. Można stosować również funkcje wbudowane i podprogramy funkcji. Bloki BEGIN lub DO mogą być wzajemnie zawarte w sobie, przez co osiągamy fizyczne wstawianie bloków jednego w drugi. Przy czym bloki BEGIN nie są wykonywane sekwencyjnie, a muszą być uaktywnione za pomocą odpowiedniej instrukcji, natomiast bloki typu DO są wykonywane sekwencyjnie i łączenie grupy instrukcji w blok ma na celu jedynie zabieg umożliwiający umieszczenie grupy instrukcji tam, gdzie kompilator dopuszcza wystąpienie tylko jednej instrukcji. W celu osiągnięcia takiego efektu w języku FORTRAN, trzeba stosować instrukcje skoku bezwarunkowego, np. dla realizacji instrukcji IF, gdy wymagane jest wykonanie więcej niż jednej instrukcji dla spełnionego warunku.

Funkcję analogiczną do operatora COPY z języka COBOL, dla języka PL/1 spełnia operator INCLUDE, którego zastosowanie

użytkowe jest niezwykle efektywne w połączeniu z rozbudowaną strukturą blokową języka.

Jedynym odstępstwem od zasad programowania strukturalnego w PL/1 jest możliwość organizacji wielu wejść i wyjść do bloków.

Z punktu widzenia struktury danych i ograniczenia dostępności danych, to język FORTRAN spełnia tylko pewną część wymagań programowania strukturalnego. Istnieje podział na dane GLOBAL i LOCAL.

Dane nie określone jako COMMON, znane są tylko w programie głównym lub podprogramie, w którym są zadeklarowane. Dane deklarowane jako COMMON są dostępne zarówno dla programu głównego, jak i dla wszystkich podprogramów je deklarujących. Podczas przebiegu programu obecne są w pamięci operacyjnej pola wszystkich części programu. Wadą języka jest brak możliwości określenia hierarchicznych struktur danych.

COBOL pod względem struktur danych charakteryzują dużo mniejsze możliwości. W języku tym nie są znane dane typu GLOBAL. Dział DATA DIVISION zawsze ma w programie znaczenie lokalne. Jeśli programista chce przekazać dane z jednego programu do drugiego, to musi je deklarować w LINKAGE SECTION i w wywołaniu CALL wyspecyfikować jako parametry. Na poziomie sekcji lub paragrafu nie jest możliwe jakiekolwiek ograniczenie rozporządzania danymi. W obrębie programu wszystkie dane są LOCAL. W języku COBOL nie ma również dynamicznego przydziału pamięci.

PL/1 wychodzi naprzeciw wymaganiom narzuconym przez programowanie strukturalne. Pola pamięci w PL/1 przydzielane są tylko tym danym, które występują w aktywnej części programowej. Możliwe jest to dzięki temu, że dane mogą być ograniczone aż do najmniejszej części programu. W PL/1 każdy blok, każda procedura wewnętrzna i zewnętrzna mają obszar danych LOCAL, który jest fizycznie w pamięci tylko wtedy, gdy wykonuje się ten blok lub tę procedurę. Ponadto PL/1 dopuszcza także definicję danych GLOBAL. Obszary te rezydują w oddzielnych statycznie przydzielanych modułach danych, gdzie są dostępne dla wszystkich procedur, w których zostały zadeklarowane jako EXTERNAL.

W tablicy 2 porównujemy języki: FORTRAN, COBOL i PL/1 w zakresie omówionych możliwości podziału programu, wypływających z tego połączeń modułów i wymiany danych.

TABLICA 2

Porównanie języków programowania

Wymagania	Własności	FORTRAN	COBOL	PL/1
Podział programu	bloki	nie	paragraf	BEGIN (do blok)
	segmenty	nie	sekcja	procedura wewnętrzna
	moduły	podprogram	zewnętrzne	procedura zewnętrzna
	funkcje	tak	nie	tak
Struktury danych	STREAM	tak	nie	tak
	struktury	nie	tak	tak
	GLOBAL	tak	nie	tak
	LOCAL	tak	tak	tak
	STATIC	nie	tak	tak
	DYNAMIC	nie	nie	tak
Połączenie programu	wbudowanie kodu	nie	COPY	INCLUDE
	wywołanie funkcji	EXPRESSION	nie	EXPRESSION
	wywołanie wewnętrzne	CALL	PERFORM	CALL
	wywołanie zewnętrzne	CALL	CALL USING	CALL
	struktura blokowa	nie	nie	tak
Wymiana danych	obszary COMMON	tak	nie	tak
	adresowanie LOCAL	tak	tak	tak
	wartości LOCAL	tak	nie	tak
	operatory LOCAL	nie	nie	tak
	wskaźniki LOCAL	nie	nie	tak

Z przeprowadzonych rozważań wynika, że nie wszystkie języki programowania w dostatecznym stopniu wspomagają programowanie strukturalne. Pamiętając, że E. W. Dijkstra był prekursorem programowania strukturalnego, zrozumiałe staje się jego stwierdzenie „[...] Im wcześniej zapomnimy, że FORTRAN kiedykolwiek istniał tym lepiej, ponieważ jako wehikuł myśli

nie jest on już adekwatny; marnuje nasz potencjał umysłowy i jest za ryzykowny, a tym samym za drogi, aby go używać [...]" [3].

Języki później powstałe zawierają więcej elementów wspierających programowanie strukturalne, czego przykładem może być język programowania PL/1 czy PASCAL, o którym mówiliśmy niewiele.

[3] E. W. Dijkstra, *The Humble Programmer*, „Communication ACM" 1972, nr 10.

VI. Dokumentowanie prac z wykorzystaniem podejścia strukturalnego

1. Schemat blokowy programu

Każdy produkt programowy oprócz poprawnego i niezawodnego funkcjonowania musi być przed przekazaniem do eksploatacji użytkowej odpowiednio udokumentowany. Dokumentacja jest opracowywana w celu prostego i czytelnego przedstawienia zastosowanych rozwiązań tak, aby można było bez większych trudności wgłębić się w przyjęte rozwiązania w celu wprowadzenia uzupełnień, zmian lub usunięcia usterek nie wykrytych na etapie testowania.

Właściwie stosowana inżynieria oprogramowania to nie tylko strukturalne programowanie i planowanie, ale także odpowiednia dokumentacja programów, umożliwiająca innym łatwe i szybkie, a więc efektywne rozpoznanie programu. Postacie reprezentacji i sposoby opisywania, z których korzysta się przy opracowywaniu oprogramowania, są nieporównywalne z metodami, stosowanymi w innych dziedzinach techniki, ponieważ nie dają możliwości dokumentowania rezultatów pracy w graficznej, poglądowej, a zarazem zwartej postaci. Schemat przepływu danych i schemat blokowy nie mają takiego znaczenia, jakie mają rysunki techniczne w budowie maszyn czy układ połączeń w elektrotechnice. W praktyce schemat blokowy często opracowuje się dopiero po zakończeniu pracy nad programem i uważa się go nie za środek pomocniczy, ale za uciążliwy wymóg dokumentacyjny. Przenosząc taki sposób rozumowania na budowę maszyn oznaczałoby to, że najpierw buduje się maszynę, a dopiero po tym wykonuje się rysunki dokumentacyjne.

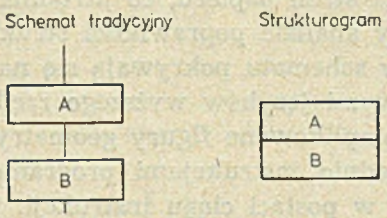
Geneza takiego podejścia do schematów blokowych programów wywodzi się z trudności w operowaniu powszechnie używanymi symbolami graficznymi. Symbole są zbyt duże i mało komunikatywne, przez co schemat blokowy nie jest przejrzysty i zwykle zajmuje kilka stron papieru, co utrudnia praktyczne wykorzystanie go przy analizie poprawności struktury logicznej programu. Elementy schematu pokrywają się najczęściej z instrukcjami programowymi języków wyższego rzędu. Programista zamiast kreślić skomplikowane figury geometryczne, woli posługiwać się bezpośrednio instrukcjami programowymi. Jeżeli jest gotowy program w postaci ciągu instrukcji, to nie ma potrzeby kreślenia adekwatnego schematu blokowego, który należałoby aktualizować wraz z programem w miarę postępów w testowaniu programu. Dublowanie prac, na etapie testowania, nie przynosi spodziewanych efektów, gdyż materiałem do analizy jest zawsze wydruk programu, a nie schemat blokowy. Najczęściej schemat blokowy powstaje dopiero po pełnym wytestowaniu programu na podstawie wylistowanych instrukcji programu, co staje się dokładnym odwróceniem problemu.

Wraz z wprowadzeniem nowych metod programowania zaczęto nadawać schematowi blokowemu programu lub jego odpowiednikowi duże znaczenie, nie tylko jako materiałowi dokumentującemu wykonaną pracę, ale jako narzędziu — wykorzystywanemu w procesie pisania i testowania programów. Przez nowe podejście do konstrukcji schematów blokowych programu eliminuje się niedoskonałości dotychczas stosowanej techniki, jak również są uwzględniane zmiany w technologii powstawania programu, czyli programowania z góry w dół i podziału programu na moduły.

Jak już mówiliśmy, każdy program można zbudować, korzystając jedynie z trzech podstawowych struktur: sekwencji, wyboru i powtarzania. Przez odpowiednie zanurzanie i łączenie, z tych trzech struktur można skonstruować dowolny program. Wykorzystując tę zasadę przyjęto dla każdego z przebiegów podstawowych jeden odpowiadający mu znak graficzny. Z tak określonych znaków graficznych można budować schemat blokowy programu w sposób czytelny i dobrze oddający logikę programu. W zależności od przyjętej szczegółowości schematu blokowego można osiągnąć ogólne schematy blokowe lub wnikając w coraz

głębsze poziomy zanurzania, wybrane elementy schematu opisywać w sposób coraz to bardziej szczegółowy.

Dla struktury sekwencyjnej jako znak graficzny przyjęto prostokąt podzielony poziomo na kolejne „piętra" (por. rys. 19).

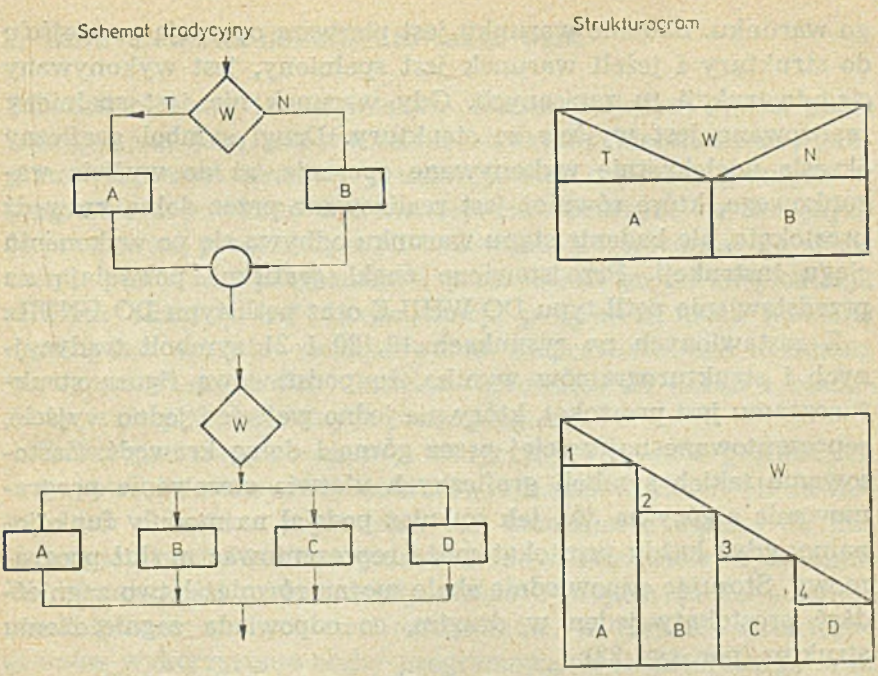


Rys. 19. Symbole graficzne dla struktury sekwencji

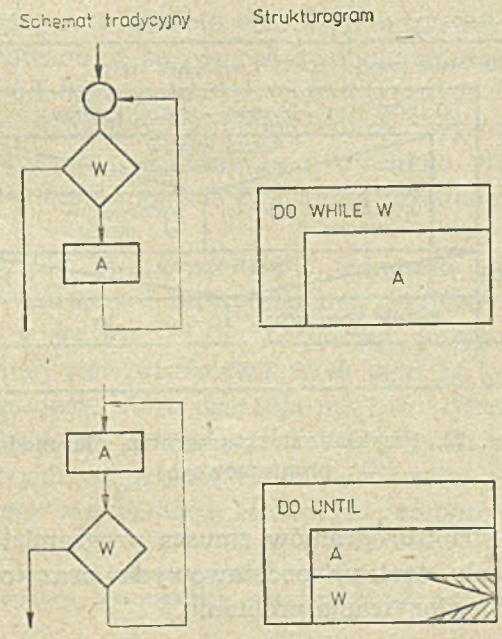
W zależności od tego, ile instrukcji (operacji) wykonywanych jest sekwencyjnie, tyle „pięter" występuje w prostokącie. Realizacja instrukcji zawartych w takim prostokącie odbywa się sekwencyjnie z góry w dół, czyli od najwyższego „piętra" prostokąta do najniższego przez wszystkie pośrednie. Tak określona struktura ma jedno wejście, którym jest górna krawędź prostokąta i jedno wyjście — jakim jest dolna krawędź prostokąta, czyli może być modułem indywidualnie programowanym, o ile nie ma innych przeciwwskazań.

Dla struktury wyboru określono dwa symbole (por. rys. 20). Jeden reprezentuje wybór dwuwartościowy, a drugi wielowartościowy. W górnej części prostokąta (wejście do prostokąta wykonuje się również przez górną krawędź) opisany jest warunek, od którego spełnienia (tak — T) lub niespełnienia (nie — N) zależy dokonanie wyboru dwuwartościowego lub wielowartościowego. W dalszej części prostokąta opisana jest akcja programu realizowana w konkretnej sytuacji. Wejście realizowane jest przez dolną krawędź prostokąta po wykonaniu ciągu instrukcji wybranych po zbadaniu warunku. Mimo opisu dwu różnych wariantów operacji dla konkretnej sytuacji, realizowany jest tylko jeden z nich i wyjście z niego jest końcem struktury.

Struktura powtarzania (por. rys. 21) jest również reprezentowana przez dwa znaki graficzne. Górny znak przedstawia wielokrotne wykonanie ciągu instrukcji aż do spełnienia określone-



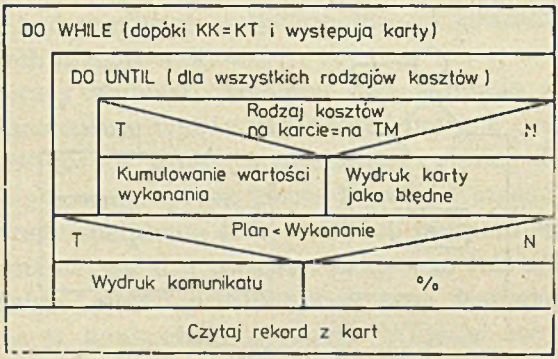
Rys. 20. Symbole graficzne dla struktury wyboru



Rys. 21. Symbole graficzne dla struktury powtarzania

go warunku. Badanie warunku jest pierwszą operacją po wejściu do struktury i jeżeli warunek jest spełniony, jest wykonywany ciąg instrukcji tu zapisanych. Gdy warunek nie jest spełniony realizowane jest wyjście ze struktury. Drugi symbol graficzny określa wielokrotnie wykonywane operacje, aż do wyjścia warunkowego, które również jest realizowane przez dolną krawędź prostokąta, ale badanie stanu warunku odbywa się po wykonaniu ciągu instrukcji. Przedstawione znaki graficzne pozwalają na przedstawienie pętli typu DO WHILE oraz pętli typu DO UNTIL.

Z zestawionych na rysunkach 19, 20 i 21 symboli tradycyjnych i strukturogramów wynika, że podstawową figurą strukturogramu jest prostokąt, który ma jedno wejście i jedno wyjście, reprezentowane najczęściej przez górną i dolną krawędź. Zastosowanie takich symboli graficznych ułatwia stosowanie programowania z góry na dół, jak również podział na moduły funkcjonalne, gdyż każdy prostokąt może reprezentować moduł programowy. Stosując odpowiednią skalę można również łatwo zagnieżdżać prostokąty jeden w drugim, co odpowiada zagnieżdżaniu struktur (por. rys. 22).



Rys. 22. Przykład strukturogramu dla modułu programowego

Stosowanie strukturogramów zmusza programistę do stosowania tylko trzech struktur podstawowych oraz odpowiedniego modułowego rozwiązywania problemu.

2. Inne techniki dokumentowania prac

Schemat blokowy jest elementem dokumentującym pewną pracę koncepcyjną, jaką jest tworzenie struktury hierarchicznej programu. Schemat blokowy prezentuje sieć działań logicznych w programie i powinien być materiałem roboczym do kolejnych faz tworzenia programu, czyli kodowania i testowania. Po skończeniu pracy nad programem i przekazaniu go do eksploatacji, schemat blokowy jest dokumentacją programu, która powinna umożliwiać szybkie zlokalizowanie nieprawidłowego działania w wypadku awarii lub ułatwiać wprowadzenie zmian i rozbudowy funkcji realizowanych przez program.

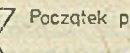
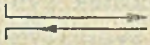
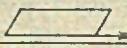
Biorąc pod uwagę rozmiary tradycyjnych znaków graficznych oraz potrzebę stosowania dodatkowych opisów słownych zawierających niezbędne objaśnienia lub komentarze dążono do miniaturyzacji schematów i jednoznaczności stosowanej symboliki. W dalszej części przedstawimy niektóre propozycje dokumentowania prac za pomocą strukturogramów. W technice strukturogramów wykorzystano cechy programowania strukturalnego oraz wprowadzono nowe „skondensowane" symbole graficzne.

W strukturalnym schemacie przebiegu wszystkie rozkazy programu zintegrowane są za pomocą symbolu graficznego, którego rodzaj zależy od typu jednostek syntaktycznych. Rozróżnia się osiem typów jednostek syntaktycznych, występujących we wszystkich językach programowania wyższego rzędu i używanych niemal wyłącznie do konstruowania programów w tych językach.

Na rysunku 23[1] podane są typy jednostek syntaktycznych wraz z odpowiadającymi im symbolami graficznymi i przykładem rozkazu w języku PL/1. Jednostka przyporządkowująca wartość zmiennej jest przedstawiona za pomocą koła. Jednostka decyzyjna ustalająca, która operacja ma być wykonana jako kolejna, symbolizowana jest tak, jak w dotychczas znanych schematach blokowych, za pomocą rombu. Jednostkę pętli, nie oznaczoną dotychczas w schemacie blokowym żadnym symbolem, re-

[1] Rysunek opracowano na podstawie pracy: A. Schmitza, *Das strukturierte Ablaufdiagram zur graphischen Interpretation von Programablaufen*, „Online Zfd" 1975, nr 10.

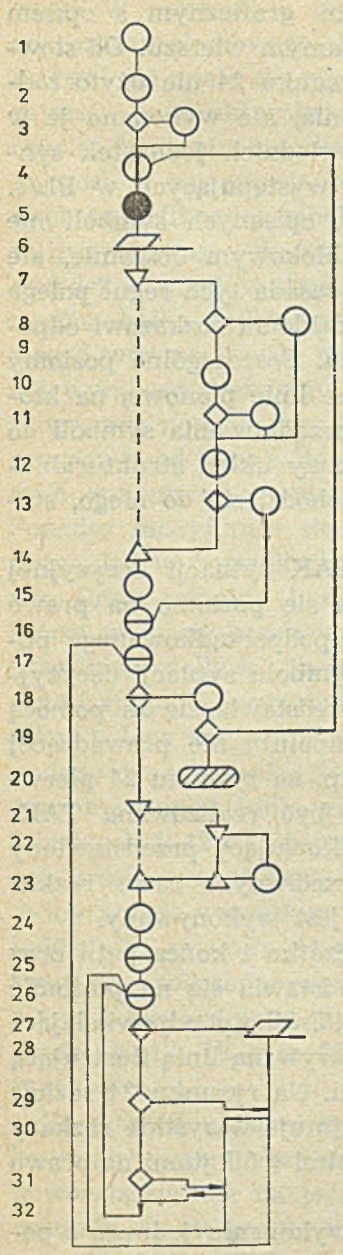
prezentują dwa trójkąty znajdujące się jeden nad drugim; górny trójkąt oznacza początek pętli, zaś dolny — jej koniec. Jednostkę wykonania podprogramu, nazywaną wywołaniem podprogramu, przedstawiają dwie strzałki symbolizujące wejście do podprogramu i wyjście z niego. Wykonanie tej jednostki polega na przenoszeniu sterowania z programu głównego do wywołanego podprogramu, a po jego zakończeniu z powrotem do programu głównego. Jednostkę skokową symbolizuje strzałka i linia ciągła prowadząca do określonego miejsca w programie. Zakreskowany owal oznacza jednostkę zakończenia programu lub podprogramu. Jednostki we-wy przedstawia się podobnie jak w schemacie blokowym za pomocą równoległoboku. Jeśli dodatkowo wprowadzona strzałka wskazuje na równoległobok — oznacza to operację czytania, jeśli zaś jest skierowaną na zewnątrz — operację pisania.

Typ rozkazu	Symbol graficzny	Przykład rozkazu w PL/1
Rozkaz przyporządkowujący		A=B+C-3
Rozkaz decyzyjny		IF, A=B THEN...ELSE...
Rozkaz pętli	 Początek pętli Koniec pętli	DO, I=1 BY 1 TO 18 END
Wywołanie podprogramu		CALL...
Rozkaz skokowy		GO TO...
Rozkaz zakończenia podprogramu		END, RETURN
Rozkaz czytania		READ, GET...
Rozkaz pisania		WRITE, PUT...

Rys. 23. Typy rozkazów wraz z odpowiadającymi im symbolami graficznymi

Z rysunku 24 [2] wynika, że strukturalny schemat przebiegu jest czysto graficzną prezentacją, podawaną równocześnie z formułowaniem rozkazów. W odróżnieniu od schematu blokowego, tekstu rozkazów nie wpisuje się tu w symbole graficzne, lecz obok nich.

[2] Rysunek opracowano na podstawie pracy A. Schmitza. Por. A. Schmitz, op. cit.



1. SZA = 1
   SMENDE = SPAR (RSE) - SPAR (RSA)
2. Jeśli MM = 1 to SMENDE = SMENDE + 1
3. ZDS = 1
   BAHN = BAHN + 1
   Napisz REKORD 1 (BAHN, kropki)
   Utwórz pętlę: SM = SMA do SMENDE
   3.1. Jeśli MA(1, SM) = 1 to HILF = KB
   i wykonaj 3.4
   3.2 HILF = SEN
   3.3 Jeśli MA (1, SM) = 0, to HILF = 0
   3.4 ZDS = ZDS + HILF
   3.5 Jeśli ZDS > 132, to SME = SM - I
4. SME = SMENDE
5. ZM = 0
6. ZM = ZM + 2
7. Jeśli ZM > AZMU, SMA = SME + I
   7.1 Jeśli, SME = SMENDE, wykonaj 3.
   7.2 Zakończ podprogram
8. Utwórz pętlę: DZ = 1 do KH
   8.1 Utwórz pętlę: DS = 1 do 132
   8.1.1 DRUCK (DZ, DS) = 0
9. CDS = 1
   A SM = SMA - 6
10. SM = SM + 1
11. Jeśli SM > SME, napisz REKORD 2 (DRUCK/DZ, DS)
    i wykonaj 6
12. Jeśli MA/1, SM/ = 1, wykonaj podprogram KNOTEN
    wykonaj 10.
13. Jeśli MA(1, SM/ = 0 wykonaj podprogram STELM
14. Wykonaj 10

Rys. 24. Strukturalny schemat przebiegu z sekwencją rozkazów

Bezpośredni związek między symbolem graficznym a opisem słownym powstaje przez zapis w tym samym wierszu. Do słownego opisu rozkazów pokazanych na rysunku 24 nie użyto żadnego ze znanych języków programowania, ale wykonano je w języku formalnym, korzystając z odpowiednich jednostek syntaktycznych, zbliżonych do jednostek występujących w PL/1. W strukturalnym schemacie przebiegu, opisanych symboli nie rozmieszcza się tak jak w schemacie blokowym dowolnie, ale według określonych reguł. Podstawowa zasada tych reguł polega na tym, że w graficznej reprezentacji każdemu rozkazowi odpowiada pewien określony poziom rozkazu. Poszczególne poziomy rozkazów reprezentowane są przez różne linie pionowe, na których znajdują się te symbole. Z przyporządkowania symboli do poszczególnych poziomów wynika właściwy układ strukturalnego schematu przebiegu i w zasadzie dochodzi się do niego, stosując trzy reguły postępowania.

Pierwsza reguła mówi, że wyjście TAK sytuacji decyzyjnej zawsze reprezentuje linia odgałęziająca się poziomo na prawo od symbolu graficznego i prowadzi do podporządkowanego poziomu rozkazu (leżącego na prawo od symbolu sytuacji decyzyjnej). Natomiast wyjście NIE zawsze przedstawia się za pomocą linii odgałęziającej się pionowo od symbolu i nie prowadzącej do poziomu podporządkowanego. Jeśli np. na rysunku 24 pierwsza sytuacja decyzyjna (MM = 1) ma być realizowana TAK, to zostaje wykonany rozkaz przyporządkowujący przedstawiony na poziomie podporządkowanym. W przeciwnym razie rozkaz ten z poziomu podporządkowanego nie jest wykonywany.

Druga reguła określa, że symbole początku i końca pętli oraz wszystkie rozkazy w obrębie pętli przedstawia się na poziomie podporządkowanym (a więc na prawo). Trójkąty odpowiadające początkowi i końcowi pętli łączy się przerywaną linią ilustrującą powrót przy wykonywaniu rozkazu pętli. Na rysunku 24 rozkaz pętli po pierwszym rozkazie pisania obejmuje wszystkie rozkazy przedstawione między dwoma odpowiednimi trójkątami na prawo od przerywanej linii łączącej te trójkąty.

Trzecia reguła polega na tym, że po wykonaniu jednego z pozostałych rozkazów nie może nastąpić żadne przejście na podporządkowany poziom rozkazu — z wyjątkiem rozkazu skoko-

wego. W pokazanym przykładzie reguła ta odnosi się również do rozkazu skokowego, jednakże przy ograniczonym stosowaniu na ogół nie udaje się jej przestrzegać. Jak widać rozkaz skokowy zajmuje w strukturalnym schemacie przebiegu szczególne znaczenie, gdyż nie ma dla niego żadnych ograniczeń ze względu na poziom następnego rozkazu. Na rysunku 24 dokonano pewnych modyfikacji symboli podanych na rysunku 23. Modyfikacje te przyczyniają się do zwiększenia przejrzystości informacji przedstawionych w strukturalnym schemacie przebiegu. Tak więc zamiast kółka podstawowego symbolu rozkazu przyporządkowującego wprowadza się również duży punkt wtedy, kiedy zmiennej wynikowej (zmienna zawierająca wynik z wykonanego podprogramu) nadana zostaje pewna wartość. Dzięki temu w podprogramie od razu widoczne są tylko te nieliczne miejsca, w których zmiennym wynikowym nadana została pewna wartość. Ponadto modyfikuje się rozkaz przyporządkowujący i sytuacje decyzyjne w wypadkach, gdy powołują się one na wspólny licznik. Przez wspólny licznik rozumie się tu zmienną, za pomocą której w czasie przebiegu podprogramu obliczona zostaje pewna określona wielkość. Ta modyfikacja rozkazu przyporządkowującego i sytuacji decyzyjnej mająca na celu użycie wspólnego licznika, polega na wprowadzeniu pionowej kreski. W przykładzie podanym na rysunku 24 zmienna ZDS służy jako licznik i dlatego też odpowiednio zmodyfikowane są symbole rozkazów w wierszach 4, 12 i 13 powołujące się na ZDS. Za pomocą poprzecznej kreski modyfikuje się rozkaz przydzielenia i rozkaz decyzyjny przede wszystkim wtedy, gdy powołują się one na wspólny indeks. Na rysunku 24 zmienne ZM i SM są wskaźnikami użytymi do oznaczenia wiersza i kolumny macierzy. W związku z tym odpowiednio modyfikowane są symbole rozkazów w wierszach 16, 17, 18 oraz 25, 26, 27, powołujące się na te zmienne indeksowe.

Za pomocą ukośnej kreski modyfikuje się rozkaz przyporządkowujący oraz rozkaz decyzyjny, przede wszystkim wtedy, gdy powołują się one na jedną cechę (zmienna Boole'a). Przez cechę rozumie się tu pewną zmienną, która może przyjąć tylko dwie wartości i służy jedynie do zbadania czy jakiś określony symbol przedstawia wartości prawda czy fałsz. Tę ukośną kreskę pochyla

się w lewo lub w prawo w zależności od tego, która z tych dwóch wartości zostaje przyporządkowana lub badana. Poza tymi zmodyfikowanymi symbolami określone zmienne przebiegu programu można wykazywać za pomocą różnych barw, co ułatwia jeszcze bardziej orientację w strukturalnym schemacie przebiegu.

Zmiany symboli przeprowadzone zgodnie z tymi wyjaśnieniami nie są głównym elementem strukturalnego schematu przebiegu, ale sprawdziły się w praktyce i okazały się bardzo wygodne przy tworzeniu strukturogramów. Podane uwagi mają jedynie podkreślić, że pomysłowa modyfikacja symboli w strukturalnym schemacie przebiegu może zwiększyć przejrzystość graficznej prezentacji. Czysto graficzna interpretacja programów, jaką jest strukturalny schemat przebiegu, może być czytana niezależnie od podanego obok niej tekstu rozkazów. W połączeniu z tym tekstem umożliwia jednak bardzo zwartą prezentację przebiegu programu oraz znacznie szybszą orientację w nim, niż w wypadku stosowania zwykłego schematu blokowego, w którym informacje graficzne są pomieszane z informacjami słownymi. Równoległa prezentacja strukturalnego schematu przebiegu i słowne sformułowanie rozkazów zapewniają bezpośrednie przejście z liniowej (jednowymiarowej) prezentacji przebiegu programu do dwuwymiarowej prezentacji, co osiąga się przez wylistowanie rozkazów; jest ona lepiej dostosowana do możliwości spostrzegania ludzkiego oka.

Często ze względu na rozmiar problemu i potrzebną powierzchnię, schemat blokowy dzieli się na schematy cząstkowe i za pomocą łączników tworzy się powiązania między nimi. Graficzna interpretacja przebiegu programu redukuje częściowo potrzebę stosowania łączników. Dalsza istotna zaleta strukturalnego schematu przebiegu, w porównaniu ze schematem blokowym polega na możliwości graficznej prezentacji operacji pętli. Główna zaleta strukturalnego schematu przebiegu polega na tym, że w graficznie poglądowej postaci można oddać nadrzędność lub podrzędność rozkazów, a więc ich wzajemną zależność. W tradycyjnym schemacie blokowym nie jest to możliwe ze względu na brak odpowiedniego symbolu graficznego. W schemacie blokowym zarówno nadrzędność jak i podrzędność rozka-

zów widoczne są tylko wtedy, gdy nie wprowadzono rozkazów skokowych i przewidziano tylko jedno zakończenie programu.

Poza wymienionymi zaletami strukturalny schemat przebiegu okazał się korzystny w praktycznym stosowaniu, ponieważ:

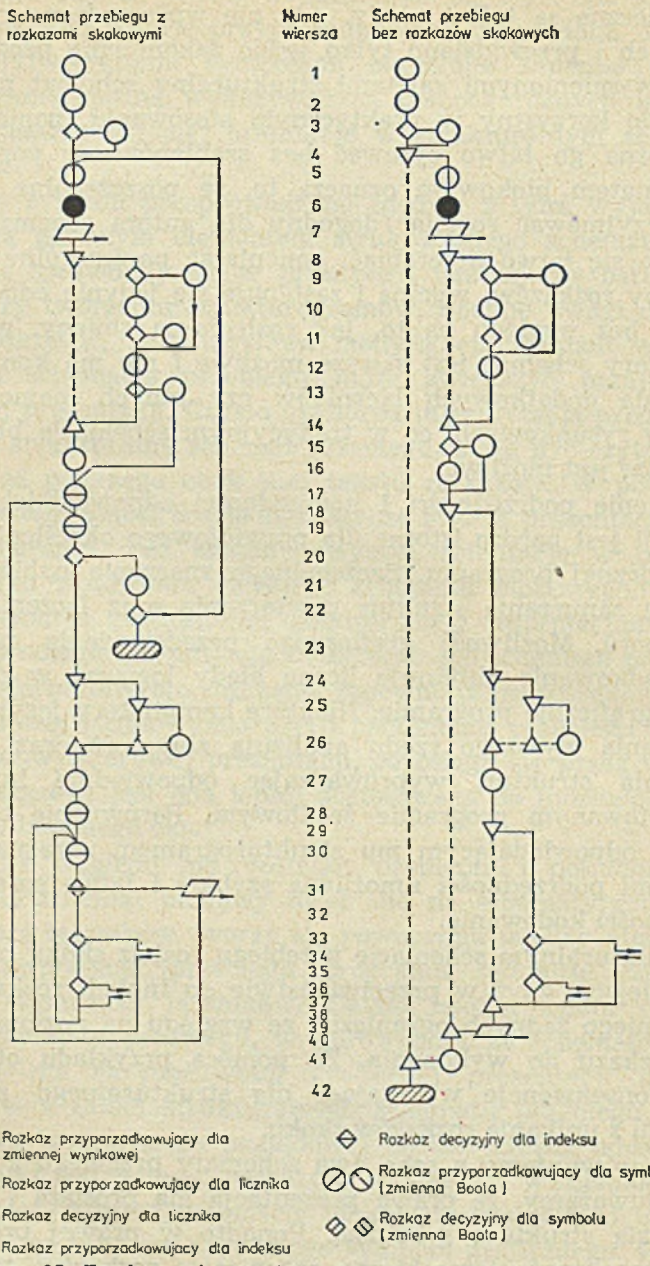
1) można go łatwo rysować bez szablonów; w porównaniu ze schematem blokowym oznacza to, że poszczególne symbole mogą przyjmować rozmiar dogodny dla autora schematu,

2) daje się łatwo przerabiać; zmieniając poszczególne rozkazy czy grupy rozkazów, wycina i zastępuje się jedynie odpowiednie miejsca, bez względu na to, jak rozległe są zmiany, włączenie ich w stary schemat jest zawsze możliwe i nie ma konieczności stosowania dodatkowych łączników czy innych pomocniczych chwytów rysunkowych, co w tradycyjnym schemacie blokowym nie zawsze jest możliwe.

Określenie podrzędności i nadrzędności odpowiednich ciągów instrukcji jest bardzo istotne dla prawidłowego określenia struktury logicznej programu. Szczególnego znaczenia nabiera to w wypadku zanurzania struktur powtarzania oraz łączenia struktur wyboru. Możliwość graficznego przedstawienia poziomów podporządkowania eliminuje liczne błędy logiczne w prezentowanym graficznie programie. Niektóre kompilatory języków programowania wyższego rzędu analizują złożoność oraz poziomy zanurzania struktur, wyprowadzając odpowiednie informacje w wylistowanym programie źródłowym. Porównanie tego wydruku z odpowiadającym mu strukturogramem z zaznaczonymi poziomami podrzędności umożliwia szybkie i łatwe sprawdzenie poprawności kodowania.

W strukturalnym schemacie przebiegu rozkaz skoku ma szczególne miejsce, gdyż w przeciwieństwie do innych rozkazów nie ma dla niego żadnych ograniczeń ze względu na poziom następnego rozkazu do wykonania. Za pomocą przykładu objaśnimy bliżej konsekwencje wynikające dla strukturalnego schematu przebiegu z unikania rozkazów skoku.

Gdy porównujemy z sobą dwa schematy przebiegu z rysunku 25 to zauważamy, że prawa prezentacja ma prostszą i bardziej przejrzystą strukturę graficzną. Ponadto w prawej prezentacji można rozpoznać hierarchiczną organizację rozkazów, tak typową dla programowania strukturalnego. Jeśli zrezygnuje się



Rys. 25. Porównanie strukturalnych schematów przebiegu dwóch ekwiwalentnych sekwencji rozkazów

z części ELSE rozkazów decyzyjnych, to rozmaite poziomy rozkazów w graficznej prezentacji przedstawiają hierarchię rozkazów w jednoznaczny sposób. Rozkazy w części ELSE sytuacji decyzyjnej są pod tym względem pewnym wyjątkiem, ponieważ mogą się one znajdować zarówno w wierszach 10 i 11 na tym samym poziomie, co rozkaz w wierszu 9, ale zgodnie z organizacją hierarchiczną są one podporządkowane temu rozkazowi.

Hierarchiczna organizacja rozkazów w prawej prezentacji ułatwia wyobrażenie przebiegu programu jako procesu dynamicznego, ponieważ już na pierwszy rzut oka są widoczne punkty odniesienia, brane pod uwagę przy określaniu dowolnego stanu w przebiegu programu. Takimi punktami odniesienia są pętle i decyzje, od których zależy wykonanie rozkazu. Rozkaz skoku narusza tę prostą i jasną hierarchiczną organizację rozkazów — jak to można zobaczyć w lewej prezentacji. Za pomocą tej graficznej postaci interpretacji udało się to, czego nie można było osiągnąć za pomocą tradycyjnego schematu blokowego — oddaje ona hierarchiczną organizację w poglądowej postaci tak, jak tego wymaga strukturalne programowanie.

Podejście strukturalne do rozwiązywanych problemów znalazło zastosowanie nie tylko w programowaniu, ale również w projektowaniu systemów informatycznych. Techniki dokumentowania projektowanego systemu oparte są na hierarchicznej strukturze systemu, a na dokumentację składa się schemat struktury z wyraźnym określeniem poziomów hierarchii oraz zestaw diagramów przepływu informacji. Szczególnie dużą popularność zdobyła dokumentacja HIPO (*Hierarchy + Input, Processing, Output*).

Literatura

Alagić S., Arbib M. A., *Projektowanie programów poprawnych i dobrze zbudowanych*, WNT, Warszawa 1982.

Baker F., Mills H., *Chief Programmer Teams*, „Datamation" 1973, nr 12.

Böhm C., Jacopini G., *Flow Diagrams, Turing Machines and Languages with only Two Formulation Rules*, „Communication ACM" 1966, nr 9.

Boos H., Trommer W., *Modularprogrammierung und rechnergestützte Programmdokumentation in COBOL*, „Online Zfd" 1975, nr 10.

Colin A., *Wstęp do systemów operacyjnych*, PWN, Warszawa 1976.

Dahl O. J., Dijkstra E. W., Hoare C. A. R., *Struktured Programming*, Academic Press, London 1972.

Dijkstra E. W., *The Humble Programmer*, „Communication ACM" 1972, nr 10.

Hackstein R., Schmitz A., *Strukturierte Programmierung und ihre graphische Interpretation*, „Online Zfd" 1975, nr 10.

Jackson M., *Principles of Program Design*, Academic Press, New York 1975.

Kuratowski K., *Wstęp do teorii mnogości i topologii*, PWN, Warszawa 1977.

Miller F., Lindamood G., *Programming: Top — Down Approach*, „Datamation" 1973, nr 12.

Mills H., *Mathematical Foundations for Structured Programming*, IMB 1971.

Mills H., *Top Down Programming in Large Systems. Debugging Techniques in Large Systems*, Rustin R., Prentice 1971.

Myers G. J., *Projektowanie niezawodnego oprogramowania*, WNT, Warszawa 1980.

Schmitz A., *Das strukturierte Ablaufdiagramm zur graphischen Interpretation von Programmablaufen*, „Online Zfd" 1975, nr 10.

Sueed H., *Strukturierte Programmierung mit FORTRAN, COBOL i PL/1*, część 1, „Online Zfd" 1975, nr 10, część 2, „Online Zfd" 1975, nr 11.

Szyjewski Z., *Wpływ rozwoju oprogramowania na projektowanie syste-*

mów informatycznych zarządzania (praca doktorska), Politechnika Szczecińska, Szczecin 1980.
Van Tassel D., *Praktyka programowania*, WNT, Warszawa 1978.
Turski W., *Metodologia programowania*, WNT, Warszawa 1978.
Twichans J., Heinemann J., *Software Engineering-Sicherung und Kontroll der Software Qualitat*, Data Exchange, III—IV 1978.
Warnier J., *Logical Construction of Programs*, Leiden: H. E. Stemfort Kroese B.V. 1974.

Redaktor techniczny
DANUTA SŁOWIŃSKA

Korektor

MAGDALENA LĄKOCY

Printed in Poland
Państwowe Wydawnictwo Ekonomiczne, Warszawa 1984

Zlec. 104/80. Wydanie I. Nakład 1850+150 egz.
Ark. wyd. 9,3. Ark. druk. 12,0
Papier druk. sat. kl. IV, 70 g. Format 61X86/16
Oddano do składania 14 IX 1983. Podpisano do druku
i druk ukończono w sierpniu 1984. Cena zł 80,—

Łódzka Drukarnia Dziełowa, Zakład nr 2, Pabianice, ul. Piotra Skargi 40
Zam. 1108. T-81

Nakładem
PAŃSTWOWEGO WYDAWNICTWA
EKONOMICZNEGO
ukazały się:

...SCHOCKE
...zne *przetwarzanie danych*
...odarce materialnej
...rszawa 1975, s. 118, cena zł 18,—

AGATA ROJEK-GROSZEWSKA
...RZEJ ZALEWSKI
Gromadzenie danych do
elektronicznego przetwarzania
Warszawa 1976, s. 350, cena zł 40,—

...IT GRINDLEY, JOHN HUMBLE
Skuteczność wykorzystania komputera
...szawa 1976, s. 245, cena zł 45,—

STANISŁAW ZADROŻNY
Organizacja zbiorów w małej
informatyce
Warszawa 1977, s. 159, cena zł 25,—

EDWARD KOLBUSZ, EDWARD KRAM
Wdrażanie systemów informatycznych
w przedsiębiorstwach przemysłowych
Warszawa 1977, s. 268, cena zł 36,—

ANDRZEJ JORDAN
Organizacja zbiorów w pamięciach
dyskowych
Warszawa 1977, s. 129, cena zł 19,—

PRACA ZBIOROWA
Przechowywanie danych
(tłum. w jęz. niem.)
Warszawa 1977, s. 154, cena zł 23,—

ZYGMUNT RYZNAR
Bank danych w przedsiębiorstwach
przemysłowych
Warszawa 1978, s. 127, cena zł 26,—

ANTONI NOWAKOWSKI
WOJCIECH OLEJNICZAK
Minikomputery biurowe
Warszawa 1978, s. 168, cena zł 25,—

BRONISŁAW OBIREK
Przygotowanie przedsiębiorstwa do
zastosowania informatyki
w zarządzaniu
Warszawa 1978, s. 151, cena zł 23,—

IGNACY DZIEDZICZAK
Model księgowości informatycznej
w przedsiębiorstwie
Warszawa 1979, s. 211, cena zł 38,—

JANUSZ ILCZUK, MARIA JERCZYŃSKA
Efektywność systemów
informatycznych zarządzania
Warszawa 1979, s. 208, cena zł 38,—

ANDRZEJ Z. IDŹKIEWICZ
Ochrona informacji w procesie
przetwarzania
Warszawa 1979, s. 147, cena zł 26,—

ROMUALD JAGIELSKI
Komputery Jednolitego Systemu
Warszawa 1980, s. 362, cena zł 53,—

ZOFIA MENET
Mała informatyka. Środki
i zastosowanie
Warszawa 1980, s. 170, cena zł 30,—

JERZY KISIELNICKI
Ekonomiczne problemy
zautomatyzowanych systemów
zarządzania
Warszawa 1981, s. 288, cena zł 50,—

W. N. LIEBIEDEW, A. P. SOKOŁOW
System operacyjny OS JS Podstawy
użytkowania
Warszawa 1982, s. 226, cena zł 60,—

IGNACY DZIEDZICZAK
Organizacja bazy danych księgowych
Warszawa 1982, s. 220, cena zł 60,—

STANISŁAW KOWALSKI
Klasyfikacja i kody w automatyzacji
przetwarzania danych
Warszawa 1983, s. , cena zł

Politechniki Śląskiej
BG Politechniki Śląskiej
nr inw.: 11 - 15264
Mg 90272

ISBN 83-208-0373-x