TOWARZYSTWO NAUKOWE ORGANIZACJI I KIEROWNICTWA
Oddział w Szczecinie

PROGRAMOWANIE STRUKTURALNE

SZCZECIN 1980

TOWARZYSTWO NAUKOWE ORGANIZACJI I KIEROWNICTWA
Oddział w Szczecinie

PROGRAMOWANIE STRUKTURALNE

SZCZECIN 1980

Opracował:
ZDZISŁAW SZYJEWSKI
Recenzja:
WOJCIECH OLEJNICZAK
JANUSZ GWIAZDA
Przygotowanie do druku:
JANINA TARASZKIEWICZ

S.Z Graf. III. A 408/80 500 szt. x P-8/14

S p i s   t r e ś c i

Str

1. Oprogramowanie systemów informatycznych 5

1.1. Miejsce oprogramowania w projektowaniu systemów .. 7
1.2. Niezawodność oprogramowania 12
1.3. Stosowane techniki programowania 16
1.4. Rozwój metodyki programowania 20

2. Podejście strukturalne 23

2.1. Rozwój programowania strukturalnego 23
2.2. Charakterystyka programowania strukturalnego 29
2.3. Dlaczego bez GO TO ? 39
2.4. Sterowanie w programie 45

3. Programowanie modularnę 56

3.1. Określenie modułu 56
3.2. Logika programowania modularnego 59
3.3. Podział programu na moduły 68

4. Strukturogramy 74

4.1. Schemat blokowy programu 75
4.2. Techniki dokumentowania prac 79

5. Przykład zastosowania 93

5.1. Definicja zadania 93
5.2. Schemat przetwarzania 94

Str

5.3. Opis zbiorów 95

5.4. Schemat blokowy - strukturogram 96

L I T E R A T U R A 100

1. Oprogramowanie systemów informatycznych

Proces projektowania systemów informatycznych realizowany jest etapowo. Można wyróżnić następujące etapy prac:

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

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

- wypracowanie koncepcji systemu informacyjnego i jego automatyzacji,
- opracowanie charakterystyki technicznej, ekonomicznej i organizacyjnej.

Założenia systemu mogą być opracowane dwoma metodami:

- metodą diagnostyczną,
- metodą prognostyczną.

W praktyce żadna z tych metod nie jest stosowana w postaci czystej ale przeplatają się wzajemnie przy tworzeniu założeń systemu.

Projekt techniczny opracowywany jest na podstawie zatwierdzonych założeń systemu. Jego realizacja odbywa się w trzech fazach:

- weryfikacja założeń systemu,
- opracowanie i zatwierdzenie szczegółowych założeń projektu technicznego,
- opracowanie dokumentacji projektu technicznego.

Wynikiem tego etapu prac jest dokumentacja projektu technicznego, która stanowi podstawę do realizacji etapu oprogramowania. W dokumentacji projektu technicznego znajdują się projekty i opisy dokumentów wejściowych i wyjściowych systemu, stosowany system kodowania, opis algorytmów przetwarzania, podział na jednostki przetwarzania i programy, projekty i opisy zbiorów oraz wytyczne do programowania.

Oprogramowanie systemu realizuje się najczęściej w następujących fazach:

- opracowanie projektu oprogramowania[1/],
- opracowanie programów w formie źródłowej i binarnej oraz ich testowanie,
- opracowanie dokumentacji programów,
- sprawdzanie współdziałania programów.

Wynikiem tego etapu prac oprócz wytestowanych programów systemu winna być skompletowana z projektu technicznego i dokumentacji programów odpowiednia dokumentacja eksploatacyjna systemu.

Ostatni etap procesu projektowania to wdrożenie systemu, które realizowane jest w dwóch fazach:

1/ Opracowanie projektu oprogramowania jest fazą oprogramowania wprowadzoną wraz z zastosowaniem techniki programowania modularnego i polega najogólniej na wyodrębnieniu elementów wspólnych, na poziomie systemu, które mogłyby być jednorazowo programowane w postaci podprogramów.

- przygotowanie obiektu do wdrożenia systemu informatycznego,
- próbna eksploatacja systemu.

Etapowość prac projektowych graficznie prezentuje rysunek 1.

1.1. Miejsce oprogramowania w projektowaniu systemów

Programowanie jest to opracowywanie koncepcji, pisanie i sprawdzanie programu[1]. W procesie programowania można zatem wyróżnić czynności związane z:

- planowaniem programu,
- kodowaniem programu,
- testowaniem programu.

Do pierwszych czynności zaliczamy opracowywanie schematów sieci działań, do drugich - pisanie programu w wybranym języku programowania, do ostatnich - wykonanie przebiegów sprawdzających poprawne działanie programu, dokonywane bez pomocy komputera /tzw. suche przebiegi/ lub z jego pomocą na danych modelowych lub rzeczywistych.

Strategią opracowywania programów komputerowych będziemy nazywali całokształt czynności wykonywanych według określonych zasad i reguł. Ich stosowanie, dotrzymywanie i respektowanie w działalności programistycznej zminimalizuje czasochłonność i pracochłonność, a tym samym koszty prac programowych oraz zoptymalizuje jakość programu.

1/ PN-71/T-01016 Przetwarzanie danych i komputery. Podstawowe nazwy i określenia.

ETAPY PRZYGOTOWANIA TYPOWEGO SYSTEMU INFORMATYCZNEGO

ZADANIE PROJEKT.

ZAŁOŻENIA SYSTEMU

PROJEKT TECHNICZNY SYSTEMU

NADZÓR AUTORSKI PROJEKTANTÓW

1)

OPROGRAMOWANIE SYSTEMU

NADZÓR AUTORSKI PROGRAMISTÓW

- KOMPLETOWANIE DOKUMENTACJI EKSPLOATACYJNEJ

WDROŻENIE SYSTEMU

PRZYG. PILOTOWEGO OBIEKTU DO WDROŻENIA SYST.

PRÓBNA EKSPLOATACJA W PILOTOWYM OB.

- WERYFIKACJA DOKUMENTACJI SYSTEMU
- PRZEKAZANIE SYSTEMU DO UŻYTKOWEJ EKSPLOATACJI W PILOTOWYM OBIEKCIE

UŻYTKOWA EKSPLOATACJA SYSTEMU /W PILOTOWYM OBIEKCIE/

WDROŻENIE SYSTEMU W INNYCH OBIEKTACH
DOSKONALENIE SYSTEMU

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

Udział programistów w pracach nad projektem technicznym systemu, jeszcze przed rozpoczęciem prac programowych, dodatnio wpływa na jakość rozwiązań projektowych. Tego rodzaju współpraca programistów z projektantami ma jeszcze tę zaletę, iż programiści, a w szczególności kierownicy zespołów programujących niejako płynnie i szczegółowo wprowadzani są w temat projektu, co ma niebagatelne znaczenie w złożonych i trudnych systemach informatycznych.

Współuczestnictwo programistów w projektowaniu zbiorów, jednostek i przebiegów przetwarzania oraz w ustalaniu struktur programów korzystne jest również dla projektantów. Zawodowo pracujący programista zawsze lepiej zna możliwości komputera oraz dopuszczalne rozwiązania programowe dla określonej maszyny cyfrowej od projektanta czy też analityka systemów. Korzyść ze ścisłej współpracy będzie obopólna. Jeśli programiści nie biorą udziału w opracowaniu projektu technicznego powinni się zapoznać /przed przystąpieniem do prac programowych/ z dokumentacją techniczną, z założeniami programowymi i weryfikować je. Weryfikację przeprowadza w tym przypadku wiodący programista, przed rozpoczęciem właściwych prac programowych.

Jaka dokumentacja jest przedmiotem zainteresowania programistów w fazie oprogramowania systemu ?

Podstawą prac nad poszczególnymi programami są z a ł o ż e n i a p r o g r a m o w e . Ponadto muszą być dostarczone programistom dokumenty dotyczące struktur i opisu zbiorów danych, obowiązujących systemów kodowania informacji, systemów kontrolnych oraz występujących w systemie jednostek przetwarza-

nia danych. Inaczej mówiąc, programiści muszą być zapoznawani z tą częścią projektu technicznego, która definiuje prace realizowane przez komputer. Z chwilą otrzymania specyfikacji programista może rozpocząć pracę nad programem.

Specyfikacja programu powinna zawierać:

- ogólny opis funkcji danego programu oraz jego miejsce w systemie,
- charakterystykę zbiorów danych przetwarzanych przez program wraz z opisem rekordów występujących w zbiorach,
- opis procedur, jakie program winien realizować,
- opis wymaganych kontroli formalnych i logicznych,
- opis parametrów sterujących programem komputerowym,
- wskazówki dotyczące budowy programu w sensie jego wewnętrznej struktury, optymalnego sposobu rozwiązania procedur, budowy modułowej, segmentacji, strategii testowania programu, wyboru języka programowania oraz inne uwagi konieczne do jednoznacznego określenia zadań i funkcji programu.

W specyfikacji programu należy również wyszczególnić ograniczenia dotyczące obszaru pamięci zajętej przez program, konfiguracji komputera, przewidywanego czasu realizacji programu. Do specyfikacji może być ponadto dołączony ogólny schemat sieci działań, zwany również s c h e m a t e m   b l o k o w y m programu. Schemat ten należy traktować jako graficzne uzupełnienie opisu procedury programu.

Ponadto do specyfikacji dołącza się wzory wydawnictw komputerowych /tabulogramów/ oraz nośników danych, jeśli takie występ-

pują w programie.

Specyfikacja programu powinna być na tyle szczegółowa, by programista mógł przystąpić do pracy nad programem bez dodatkowych ustnych wyjaśnień uzupełniających. Pamiętać należy, iż właściwie opracowane i systematycznie aktualizowane specyfikacje programów stanowią podstawową dokumentację projektową systemu informatycznego. Tak jak nie można sobie wyobrazić wykonania określonego detalu maszyny bez uprzedniego sporządzenia rysunku technicznego, zawierającego wszystkie szczegóły i wymiary detalu, tak też przy opracowywaniu systemów informatycznych nie mogą mieć miejsca prace programowe niewłaściwie lub niedostatecznie udokumentowane. Niebezpieczeństwo tego rodzaju istnieje w szczególności w zespołach projektowo-programujących, gdzie te same osoby pełnią funkcje projektantów i programistów.

W zasadzie specyfikacje programów opracowywane są przez projektantów systemów informatycznych. W bardziej skomplikowanych przypadkach projektanci konsultują specyfikację ze starszymi programistami, w szczególności wtedy, gdy projektanci nie mają dostatecznego doświadczenia w zakresie oprogramowania komputerów, na których projekt ma być realizowany. Jeśli brak doświadczonych projektantów, specyfikacje programów opracowują samodzielnie starsi programiści. Spełniają oni w tym przypadku funkcje projektantów, gdyż - jak już wspomniano - specyfikacje programów wchodzą w skład dokumentacji projektu technicznego systemu informatycznego. Dokumenty te muszą przygotować z taką dokładnością, jak wymaga tego poprawnie opracowany projekt techniczny. Niedopuszczalne są uproszczenia wynikające z tego, że ta

sama osoba najpierw przygotowuje specyfikację, a później program.

Tak w zarysie można przedstawić tradycyjny sposób opracowania oprogramowania systemu informatycznego. Przedstawiony sposób programowania pozwala na budowę niezbyt skomplikowanych projektów. Przy wzroście złożoności problemów rozwiązywanych przez oprogramowanie, rosną rozmiary programów i komplikuje się proces programowania. Programy monolityczne są dzielone na mniejsze części oddzielnie programowane i łączone w całość, co upraszcza programowanie i pozwala na podział prac. Taka metoda pracy wymaga nowej organizacji całego procesu programowania. Nowa organizacja procesu programowania ma na celu wzrost niezawodności opracowywanego oprogramowania.

1.2. Niezawodność oprogramowania

Jakość systemu informatycznego w dużej mierze zależy od sprawności i niezawodności oprogramowania systemu. Oprogramowanie systemu jest testowane na etapie tworzenia oprogramowania jak również na etapie próbnej eksploatacji. Każdy indywidualnie pisany program wchodzący w skład systemu informatycznego jest dokładnie testowany przed włączeniem do bibliotek systemu. Proces testowania jest powierzany autorowi programu a projektant systemu ogranicza się najczęściej do sprawdzenia działania programu na przygotowanych danych modelowych. W zależności od doboru danych testujących sprawdzenie programu jest bardziej lub mniej dokładne. Dopiero eksploatacja programu na danych rzeczy-

wistych pozwoli na jednoznaczne stwierdzenie o poprawności programu.

Problem testowania indywidualnie pisanego programu jest jednym z kluczowych elementów całego procesu tworzenia oprogramowania. Komputery posiadają dość ubogi aparat wspierający proces testowania programów i wiele zależy od zaangażowania i inwencji autora programu, który poprzez odpowiedni dobór danych testujących może sprawdzić różne warianty biegu programu lub ograniczyć się do sprawdzenia jedynie podstawowych dróg przebiegu programu. Testowanie programu winno być tak zorganizowane, aby w pierwszej kolejności sprawdzić bieg programu w przypadku realizacji podstawowej funkcji przy stosunkowo prostych danych testowych. W przypadku otrzymania pozytywnego wyniku takiego testowania należy w kolejnych przebiegach testowych sprawdzać kolejno inne drogi biegu programu poprzez odpowiedni dobór danych testujących. Dopiero po wytestowaniu i otrzymaniu pozytywnych wyników wszystkich możliwych dróg biegu programu można sprawdzać działanie programu przy wprowadzeniu bardzo różnorodnych danych testowych co pozwoli zaobserwować zachowanie się programu przy zmiennej realizacji różnych dróg biegu programu.

Doświadczenia praktyczne wskazują, że testowanie programu nigdy nie jest zbyt pełne, gdyż eksploatacja użytkowa programu prawie zawsze dostarcza takie dane, które powodują bieg programu drogą niezbyt dokładnie wytestowaną co zmusza do wprowadzania zmian w sytuacji przymusowej przy rygorach eksploatacji systemu. E. Dijkstra mówiąc o testowaniu programu stwierdził: "... testowanie programu może wykazać obecność pomyłki, ale nie może wyka-

zać nieobecności błędów". Nie prowadzi to do stwierdzenia, że testowanie jest zupełnie niepotrzebne, daje ono użyteczne informacje o zachowaniu się programu, należy jednak zwrócić uwagę także i na inne metody wykrywania błędów.

Proces testowania winien być - odpowiednio zorganizowany i przemyślany przez autora systemu. Najbardziej odpowiednią formą przygotowania się do testowania winno być opracowanie szczegółowego harmonogramu testowania, w którym będą uwzględnione wszystkie możliwe drogi biegu programu jak również ich możliwe kombinacje. Szczegółowe opracowanie takiego harmonogramu jak również jego precyzyjna realizacja pozwolą na pełne wytestowanie programu jak również na nie powtarzanie dróg biegu programu wcześniej wytestowanych. Do opracowanego harmonogramu testowania należy przygotować odpowiednio spreparowany zestaw danych, których ilość nie powinna być zbyt duża, aby nie wydłużać przebiegów testowych, ale dane winny być na tyle różnorodne aby osiągnąć cel danego kroku testowania.

Wprowadzanie poprawek do programu winno być kompleksowe po realizacji kilku kolejnych kroków testowania i zebraniu odpowiednio dużo doświadczenia odnośnie zachowania się programu. Każda wprowadzona poprawka do programu musi zostać sprawdzona poprzez ponowne wytestowanie na danych, które spowodowały nieprawidłowy bieg programu. Dopiero po otrzymaniu prawidłowych wyników testowania danego kroku możemy odnotować wytestowanie tego kroku. Po wytestowaniu wszystkich kroków zawartych w opracowanym harmonogramie możemy stwierdzić, że program jest wytestowany.

Czas wykorzystany na dokładne testowanie programu nie jest

czasem straconym nawet w przypadku zawsze pomyślnych wyników poszczególnych kroków ponieważ pozwala na sprawdzenie zachowania się programu przy różnorodnych danych źródłowych. Stworzenie dokumentacji procesu testowania, chociaż w postaci harmonogramu testowania, zwiększa niezawodność programu. W procesie eksploatacji użytkowej programu program dokładnie wytestowany nie powoduje niespodziewanych skutków, które mogą prowadzić do zachwiania terminarza eksploatacji. Sporządzona dokumentacja ułatwia również wprowadzanie zmian w eksploatowanym programie.

Proces opisany przy testowaniu pojedyńczego programu należy powtórzyć na poziomie oprogramowania całego systemu, gdy wszystkie programy wchodzące w skład oprogramowania są w wystarczającym stopniu wytestowane. Testowanie całego systemu winno posiadać swoją oddzielną dokumentację.

Niedokładne wytestowanie oprogramowania systemu lub pojedyńczych programów systemu powoduje przestoje w procesie eksploatacji użytkowej i potrzebę szybkiego, interwencyjnego wprowadzania poprawek, które z uwagi na wymogi czasowe nie są również dokładnie testowane i często powodują wprowadzenie dalszych błędów do oprogramowania.

Dokładne i kompleksowe testowanie oprogramowania decyduje o sprawności i niezawodności systemu informatycznego przekazanego do ciągłej eksploatacji.

Eksploatowany system informatyczny musi być konserwowany aby zapewnić jego sprawność działania. Na konserwację składają się zarówno poprawki wprowadzane do systemu w celu zapewnienia jego prawidłowego funkcjonowania a zachwianego w wyniku niedokładnego

testowania jak również wprowadzanie zmian zwiększających możliwości systemu. Odpowiednio przygotowana i ciągle uzupełniana dokumentacja oprogramowania systemu pozwala na elastyczne wprowadzanie zmian i poprawek do eksploatowanego systemu. Prowadzenie dokumentacji pozwala na poprawianie i uzupełnianie programów bez udziału autorów, którzy po pewnym czasie eksploatacji bądź są nieosiągalni albo nie pamiętają już szczegółów dawno pisanego programu.

Na niezawodność oprogramowania systemu informatycznego w dużej mierze składają się prawidłowe i dokładne testowanie oprogramowania systemu oraz odpowiednio przygotowana i prowadzona dokumentacja.

1.3. Stosowane techniki programowania

Biorąc pod uwagę dość niską sprawność oprogramowania indywidualnie tworzonego, częste awarie programów, problemy dokumentacyjne i związane z tym uzależnianie funkcjonowania systemu od nadzoru autorskiego twórców oprogramowania zaczęto odchodzić od indywidualnego pisania programów. Dużą popularność zyskały techniki tworzenia oprogramowania w oparciu o oprogramowanie standardowe komputerów.

Pisanie programów wielofunkcyjnych, sterowanych parametrami to pierwsze posunięcia na drodze odchodzenia od indywidualnego programowania. Program taki, jest dużo trudniejszy od napisania i dokładnego wytestowania, ale możliwość wielokrotnego wykorzystania w różnych systemach informatycznych pozwala na ponie-

sienie wysokich kosztów jego pisania i dokładnego testowania.

Szereg funkcji w systemie informatycznym jest identycznych niezależnie od dziedziny zastosowań jak np. zakładanie zbioru z dokumentów źródłowych, kontrola danych źródłowych, program wydruku zbiorów roboczych itp. Stworzenie jednego programu zakładania zbioru z dokumentów źródłowych pozwala na wmontowanie takiego programu do oprogramowania prawie każdego systemu informatycznego z odpowiednim zestawem parametrów uwzględniających specyfikę danego systemu.

Podobny kierunek działania podjęty został przez producentów komputerów, którzy wraz ze sprzętem gotowi są dostarczyć na żądanie użytkownika potrzebne oprogramowanie standardowe bardzo wysokiej jakości z pełną dokumentacją. Oprogramowanie to grupowane jest w grupy tematyczne i tworzą całe pakiety programów zabezpieczające oprogramowanie wielu dziedzin tematycznych zastosowań systemów informatycznych. Dostępne są pakiety programów zabezpieczające oprogramowanie systemu informatycznego gospodarki magazynowej, systemu zbytu czy oprogramowanie zabezpieczające rozwiązanie problemów transportowych, wykorzystania metod sieciowych czy wyszukiwania informacji.

Ten kierunek tworzenia oprogramowania systemów informatycznych pozwala na tworzenie oprogramowania w sposób bardzo szybki i tani. Ponadto oprogramowanie otrzymane charakteryzuje się wysoką sprawnością i niezawodnością oraz posiada pełną dokumentację. Te zalety oprogramowania pakietowego zapewniły dużą grupę zwolenników tworzenia oprogramowania systemu z oprogramowania dostarczanego przez producentów komputerów.

Uniwersalność oprogramowania pakietowego została jednak osiągnięta dzięki wprowadzeniu szeregu ograniczeń, które nie zawsze pozwalają na skorzystanie z posiadanych zalet. Powstał problem takiego projektowania systemów informatycznych aby pogodzić oprogramowania i wymogi narzucane przez przewidywane do zastosowania oprogramowanie a spełnienie wymagań przyszłego użytkownika systemu. Nie w każdym przypadku udało się pogodzić oba warunki co znacznie ograniczało zasięg zastosowalności oprogramowania pakietowego.

Rozwiązaniem kompromisowym, często stosowanym, było wmontowanie programów standardowych w te miejsca oprogramowania systemu, gdzie występowała taka możliwość. Pozwoliło to na częściowe obniżenie kosztów tworzenia oprogramowania oraz wprowadzało pewne rygory formalne dla reszty programów.

Inną techniką tworzenia oprogramowania jest stosowanie generatorów programów. Cechą charakterystyczną tej techniki pracy jest możliwość podania w pewien sformalizowany sposób, warunków jakie ma spełniać wygenerowany program w jednym z dostępnych języków programowania. Otrzymana postać źródłowa programu może być bez zmian włączona do oprogramowania systemu lub można włączać do tak otrzymanego programu specyficzne, dodatkowe procedury pisane oddzielnie.

Rozwinięciem techniki generatorów programów są tak zwane "Języki formularzowe", których typowym przedstawicielem jest język RPG /Reports Program Genereted/. Nieco inny kierunek reprezentują tablice decyzyjne. Technika tablic decyzyjnych zwraca na siebie szczególną uwagę z uwagi na koncentrację problemu na naj-

bardziej skomplikowanym elemencie każdego programu jakim są instrukcje warunkowe.

Prawidłowy ciąg instrukcji warunkowych decyduje o prawidłowym bądź błędnym biegu programu dla wprowadzonych danych. W przypadku stosowania techniki tablic decyzyjnych problem sprowadza się do:

- określania wszystkich możliwych warunków oraz
- sporządzenia wykazu działań jakie należy wykonać przy każdej kombinacji warunków.

Następnie elementy te umieszczone są w tablicy w sposób następujący:

wykaz warunków	zapis warunków
wykaz działań	zapis działań

W tablicy powyżej podwójnej linii poziomej zapisuje się warunki, a poniżej czynności /działania/. Po lewej stronie pionowej linii podwójnej występują wykazy a po prawej zapisy występowania lub niewystępowania warunku lub czynności. Przyjęto przez T oznaczać konieczność spełnienia warunku sformułowanego w pozycji wykazu, przez N określenie, że warunek nie może być spełniony, a - oznacza, że warunek jest nieistotny. Podobnie w polu działań zapis X wskazuje, że działanie z wykazu ma być wykonane przy spełnieniu warunków, a - oznacza potrzebę zignorowania danego działania.

Kombinacja pozycji wykazu i pozycji zapisu tworzy pewien warunek /powyżej podwójnej linii/ lub działanie /poniżej podwójnej linii/. Każda kolumna po prawej stronie pionowej linii podwójnej

jest zestawiana z kolumną po lewej stronie tej linii, w ten sposób powstaje reguła decyzyjna.

Technika tablic decyzyjnych pozwala nam na zapisanie niejako w sposób graficzny zdań typu "jeżeli ... to ...". Tablicowy sposób zapisu jest bardzo czytelny do analizowania i bardzo prosty w konstrukcji. Przygotowana tablica decyzyjna poddawana jest procesowi translacji przez odpowiednie programy standardowe i w wyniku przebiegu translacji otrzymujemy podprogram w jednym z dostępnych języków programowania.

Technika tablic decyzyjnych obok oczywistych zalet posiada jednak szereg niedoskonałości, do których należą w pierwszej kolejności duże rozmiary tablic przy opisie bardziej skomplikowanych problemów. Ponadto brak wygodnych translatorów tablic decyzyjnych co wyraźnie ogranicza możliwość stosowania tej techniki pracy.

Technika tablic decyzyjnych wymaga zupełnie innego podejścia do problemu, niż przy stosowaniu tradycyjnych technik programowania opartych na sieci działań. Inny sposób podejścia do rozwiązywanego problemu był niewątpliwym hamulcem rozwoju tej ciekawej i efektywnej techniki tworzenia programów.

1.4. Rozwój metodyki programowania

Mimo stosowania różnych technik tworzenia oprogramowania systemów informatycznych nie ma możliwości całkowitego wyeliminowania programowania indywidualnego. Doświadczenia praktyczne wskazują również na to, że indywidualnie pisane programy naj-

efektywniej realizują zadania stojące przed systemem informatycznym, pod warunkiem spełnienia szeregu wymogów niezawodnościowych poszczególnych programów jak również całego oprogramowania.

Obok prac zmierzających do wyeliminowania indywidualnego programowania prowadzone były prace nad doskonaleniem metodyki programowania. Prace te koncentrowały się głównie nad problemem osiągnięcia wysokiego współczynnika niezawodności tworzonego oprogramowania jak również podniesienia szybkości prac, co ściśle się łączy z kosztami opracowania oprogramowania oraz ułatwieniem modyfikowania i dokonywania zmian w programach.

Prace badawcze zmierzające do wypracowania doskonalszych metod tworzenia oprogramowania przebiegały w dwóch kierunkach. Jeden polegał na próbie ominięcia problemu przez zastosowanie wysokosprawnego oprogramowania pakietowego dostarczanego przez producenta komputera wrąz ze sprzętem a drugi kierunek koncentrował się na doskonaleniu metod programowania indywidualnego.

Stosowanie pakietów programów daje znaczne oszczędności na koszcie opracowania systemu. Dostępne pakiety są zwykle dobrze udokumentowane, a ich dostawcy są na ogół gotowi do dokonania odpowiednich zmian na zlecenie nabywcy. Jednym z hamulców szerokiego wykorzystania pakietów jest stosunkowo szczupła liczba dostępnych obecnie pakietów użytkowych.

W zależności od pakietu, oprogramowanie może być czasem wdrożone bez zmiany choćby jednego rozkazu, ale czasem może wymagać wprowadzenia dość istotnych i skomplikowanych zmian. Przykładem oprogramowania przyjmowanego bez zmian mogą być różnego

rodzaju generatory programów czy translatory tablic decyzyjnych. Natomiast pakiety użytkowe wymagają najczęściej zmian dla dostosowania ich do konkretnych potrzeb przetwarzania lub takiego projektowania systemu aby uwzględniać ograniczenia pakietu.

Rozwój metodyki programowania koncentrował się na takim doskonaleniu metod pisania programów aby produkt był niezawodny w działaniu, czas tworzenia był najkrótszy, koszt opracowania najniższy a program był przejrzysty, czytelny i łatwy do konserwacji. Kluczowym elementem stała się również odpowiednia, czytelna i pełna dokumentacja programu.

Rozbudowa zestawu sprzętowego komputera powodowała powstanie nowego typu problemów odpowiedniej organizacji i obsługi dużej ilości urządzeń. Rozwiązania inżynierskie pozwoliły na przejęcie niektórych funkcji kontrolno-sterujących bezpośrednio przez sprzęt. Rozwiązania techniczne pozwoliły również na częściowe zabezpieczenie zbiorów danych w systemach przed nieumyślnym zniszczeniem jak również przed niepowołanym dostępem do zapisanych danych.

Odrzielna grupa zagadnień to stała rozbudowa i doskonalenie funkcjonujących systemów operacyjnych jak również dostępnych języków programowania. Coraz większe możliwości języków wyższego rzędu powodują, że programowanie staje się mniej pracochłonne gdyż wspomaganie metod dostępu, sterowania i innych kluczowych funkcji jest coraz większe.

Programowanie indywidualne w dalszym ciągu stanowi istotny procent prowadzonych prac nad oprogramowaniem systemów informatycznych. Niedoskonałość dotychczas stosowanych metod programo-

wania pobudzała do prowadzenia stałych poszukiwań. Ostatnio niezwykle dużą popularność zdobyła metoda programowania strukturalnego, której pierwsze zastosowania wykazały nadspodziewanie dobre wyniki.

2. Podejście strukturalne

Koncepcja ogólna podejścia strukturalnego zrodziła się w firmie IBM i wywodzi się z struktury hierarchicznej nałozonej na proces projektowania i programowania systemów informatycznych. Podstawowym problemem do rozwiązania w procesie prac projektowych i programowych były sprawy przekazywania sterowania pomiędzy poszczególnymi obiektami systemu. Sterowanie wadliwie zorganizowane kryło 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 dała możliwość eliminacji błędów wynikających z nieczytelności i zagmatwania obrazu całości rozważanego problemu.

2.1. Rozwój programowania strukturalnego

Podstawy teoretyczne programowania strukturalnego dał Bohm i

Jacopini, którzy już w roku 1966 wyprowadzili matematyczny dowód, że wszystkie programy lub ich schematy blokowe dają się doprowadzić do ekwiwalentnej postaci, w której mogą być reprezentowane tylko przez trzy podstawowe przebiegi. Na rysunku 2 pokazane są te przebiegi graficznie zilustrowane w postaci znanej z notacji schematu blokowego.

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.

W językach programowania są to instrukcje różne od instrukcji skokowych czy innych dostępnych w danym języku instrukcji warunkowych, mogących wpływać na sekwencje wykonania rozkazów, składających się na program.

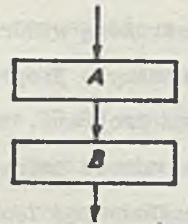
Struktura wyboru reprezentuje zwrotnicę w programie czyli instrukcję, która w zależności od stanu warunku badanego przez instrukcję powoduje wykonanie drogi A programu lub drogi B. Po realizacji wybranej drogi sterowania przekazywane jest do tego samego punktu programu, gdzie umieszczone są dalsze instrukcje programu.

W językach programowania są to typowe instrukcje warunkowe, które w językach wyżej zorganizowanych mają postać:

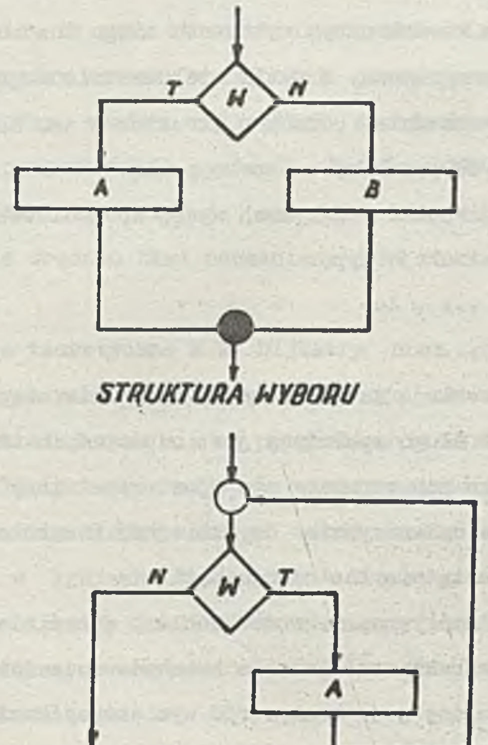
IF ... THEN ... , lub

IF ... THEN ... ELSE ... .

Przy pomocy instrukcji typu IF mamy możliwość wyboru jednej z dwóch dróg programu. Rozszerzeniem instrukcji IF jest instruk-



STRUKTURA SEKWENCYJNA



STRUKTURA WYBORU

STRUKTURA POWTARZANIA

RYS.2. PRZEBIEGI PODSTAWOWE WEDŁUG BÖHMA I JACOPINIEGO

cja CASE ... , która jest instrukcją wyboru wielokrotnego, czyli w zależności od stanu warunku mamy w programie nie dwie alternatywnie realizowane drogi biegu programu, ale kilka wariantów. Z tego widać, że wyrażenie warunkowe którego stan bada instrukcja, nie może mieć postaci boolowskiej lecz bardziej skomplikowaną.

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

Do organizacji pętli w językach wyżej zorganizowanych wykorzystywane są instrukcje typu:

DO ... WHILE ... , lub

DO ... UNTIL ... .

Instrukcje te pozwalają na wielokrotne wykonanie ciągu instrukcji tak długo jak długo spełniony jest odpowiednio określony warunek. W przypadku gdy warunek nie jest spełniony sterowanie jest przekazywane sekwencyjnie do kolejnej instrukcji programu występującej za ostatnią instrukcją pętli.

Zgodnie z matematycznym dowodem Bohma i Jacopiniego kombinacja tymi trzema strukturami pozwala na zbudowanie dowolnego programu. Zastanawiający jest brak wśród wymienionych struktur instrukcji skoku bezwarunkowego, instrukcji kluczowej dla tradycyjnego przekazywania sterowania w programie.

Na bazie tego fundamentalnego dowodu profesor E.W. Dijkstra w latach 1968 - 1972 stworzył podstawy teoretyczne programowania

strukturalnego. Znaczne poruszenie w świecie informatycznym wywołał jego list opublikowany w Communication of the ACM w roku 1968, pod tytułem "Rozkaz GO TO uznaje się za szkodliwy".

Dopóki sprawa programowania strukturalnego zdawała się mieć charakter teoretyczny, którego większość ludzi nie była w stanie w pełni zrozumieć, nic właściwie się nie działo. Dopiero zastosowania praktyczne i ich wyniki wzburzyły świat programistów.

Okazało się, że przy stosowaniu programowania strukturalnego osiąga się wyraźny wzrost wydajności pracy programistów i odpowiednio istotny spadek całkowitej częstotliwości występowania błędów w programowaniu systemów. Źródła publikowane podają, że średnio jeden błąd przypada na jeden rok pracy programisty, co oznacza, że średnio błąd popełniany jest raz na 10 000 wierszy programu.

Rozważania teoretyczne E.W. Dijkstry oraz pilotowane przez niego prace praktyczne pozwalają uważać go za ojca niektórych z koncepcji składających się na programowanie strukturalne.

W latach 70-tych obserwuje się wzrost zainteresowania techniką programowania strukturalnego wśród stale rosnącej liczby programistów w krajach zachodnich. Prace prowadzone zgodnie z zaleceniami Dijkstry jak i różnorodne modyfikacje pozwalają na dostarczenie dalszych argumentów za stosowaniem techniki programowania strukturalnego.

Nieco odmienne podejście mające cechy programowania strukturalnego opisał F.T. Baker w roku 1972 w IBM System Journal. Opisany przez niego przykład zastosowania programowania strukturalnego stanowi publikację, na którą powołują się różni autorzy pi-

szący o programowaniu strukturalnym i jego zaletach.

Opisana technika, określana mianem "zespołu głównego programisty /Chief, Programmer Team Management/, charakteryzuje podejście do projektowania i wdrażania systemów, która zastosowana została z powodzeniem w pracach wykonanych przez firmę IBM dla potrzeb New York Times'a. IBM opracował i wdrożył złożony system wyszukiwania informacji przy pomocy jedynie garstki programistów o dużych kwalifikacjach, i to w przeciągu dość krótkiego czasu. Co ważniejsze, twierdzi się, że opracowany system programów faktycznie nie zawierał błędów i działa zadowalająco od dnia wdrożenia. Podejście wówczas zastosowane łączy w sobie sprawne kierowanie i specjalne techniki programowania strukturalnego.

Duży wkład w prace nad powstaniem teorii programowania strukturalnego mają prace H. Millsa. Koncepcje Millsa kładą główny nacisk na jasność prezentacji i łatwość testowania programów.

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

Postać nieprzejrzysta

```
      .
      .
      .
   K = 0;
   L = 0;
   J = 1;
A: IF T/J/ = 999 THEN GO TO KON;
```

Postać przejrzysta

```
   .
   .
   .
K = 0;
L = 0;
J = 1;
DO WHILE /T /J/ ∠ 999/;
```

```
   IF T/J/ = 0 THEN GO TO B;        L = L + 1;
   L = L + 1;                       IF T/J/ = 0 THEN K = K + 1;
   J = J + 1;                       J = J + 1;
   GO TO A;                         END;
B: K = K + 1;                         .
   L = L + 1;                         .
   GO TO A;                           .
KON: .
     .
     .
```

Przy realizacji bardziej skomplikowanych zadań różnice mogą być jeszcze większe.

Publikacje zachodnie dotyczące sukcesów programowania strukturalnego, spowodowały zainteresowanie się tą koncepcją przez programistów krajów socjalistycznych. Po pierwszych pozytywnych eksperymentach zainteresowanie tą techniką stale wzrastało i w chwili obecnej, programie strukturalne jest najmodniejszą techniką programowania w Polsce.

Pozytywne wyniki stosowania tej techniki oraz brak konkurencyjnego rozwiązania zmierzającego do podniesienia wydajności pracy programisty i zwiększenia niezawodności programów pozwalają przypuszczać o dalszym rozwoju tej techniki programowania.

2.2. Charakterystyka programowania strukturalnego

Cechami charakterystycznymi programowania strukturalnego, zgodnie z koncepcjami reprezentowanymi przez Dijkstra, 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 podprogramie lub wpływać na nie,
5. Jedno wejście i jedno wyjście z modułu.

T. Baker w zmodyfikowanej koncepcji zwanej "zespołem głównego programisty", jako cechy charakterystyczne wyróżnia:

1. Budowanie systemu oprogramowania powinno przebiegać od góry do dołu /top - down design/,
2. Wdrażanie programów także powinno przebiegać od góry do dołu, zaś elementy zastępcze, 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. Ogólna piecza nad opracowywaniem programów powinna znajdować się w ręku bardzo doświadczonego i wysoko kwalifikowanego głównego programisty, na którego barki spadać też powinna sprawa łączenia z sobą poszczególnych modułów i testowania.

Te dwie koncepcje zawierają wszystkie najistotniejsze elementy składające się na technikę programowania strukturalnego. Obydwaj autorzy jako pierwszą, fundamentalną zasadę programowania strukturalnego wymieniają hierarchiczną strukturę programu konsekwentnie realizowaną z góry na dół.

Realizacja programu z góry do dołu powoduje najczęściej wydłużenie listy instrukcji programu. Z wydłużeniem się programów ściśle wiąże się zajętość pamięci przez program pisany z wykorzystaniem techniki programowania strukturalnego. Większa zajętość pamięci operacyjnej jest wadą omawianej techniki i jest często podnoszona jako kontrargument przez przeciwników programowania strukturalnego.

Nieoptymalna gospodarka pamięcią zajmowaną przez program jest powodowana chęcią osiągnięcia wyższej przejrzystości napisanego programu. Program optymalnie wykorzystujący pamięć posiada, z reguły tak skomplikowane sterowanie kolejnością wykonywania instrukcji i ich członów, że po pewnym czasie określenie drogi biegu programu dla określonej sytuacji jest nie do uchwycenia nie tylko przez innych programistów, ale również przez autora programu.
Wprowadzenie zmian czy modyfikacji wiąże się z dużym ryzykiem popsucia prawidłowego biegu programu w związku z brakiem przejrzystości. Biorąc pod uwagę nowe techniki gospodarowania pamięcią przez stronicowanie /pamięć wirtualna/ jak również stałe rozbudowywanie efektywnie dostępnej pamięci, problem optymalizacji zajętości pamięci przez programy nie jest decydujący przy pisaniu programów. Jednym z podstawowych problemów przy pisaniu

programów stał się problem elastyczności programu czyli możliwości wprowadzania zmian i modyfikacji jak również wysoka sprawność gotowego programu. Te cechy programu w dużym stopniu zabezpiecza przejrzystość programu.

Program napisany przejrzyście czyli pozwalający jednoznacznie i prosto określić drogę biegu daje się łatwo modyfikować czy też lokalizować i usuwać niesprawność. Przejrzystość programu można osiągnąć przez ograniczenie stosowania instrukcji skoku bezwarunkowego GO TO co postuluje Dijkstra. Sekwencyjna realizacja programu z góry na dół pozwala w sposób prosty śledzić bieg programu.

Technika pisania z góry na dół każdego programu jest więc fundamentalnym założeniem programowania strukturalnego. Pozostałe cechy wynikają z niej w sposób bezpośredni lub pośredni.

Problem jaki ma być rozwiązany przy pomocy programu jest rozbijany na mniejsze moduły funkcjonalne jak pokazuje to rysunek 3. W problemie A wydzielone zostały podproblemy B i C, które dalej dzielą się na kolejne podproblemy itd. Technika podziału na moduły zostanie omówiona w dalszej części opracowania.

Rysunek 3 przedstawia poszczególne moduły funkcjonalne w postaci prostokątów a linie łączące poszczególne prostokąty określają sposób wywoływania kolejnych modułów idąc z góry na dół gdyż tylko taki sposób poruszania się po strukturze jest przyjęty.

Reprezentacja na rysunku 3 zawiera jedynie informacje o układzie struktury modułów, a nie o przebiegu szczegółowego wykorzystania modułów. Oprócz tego w reprezentacji tej nie wszyst-

kie miejsca przecięcia interpretowane są graficznie za pomocą linii, a mianowicie wtedy, gdy pewien moduł wywoływany jest przez kilka miejsc innego modułu.

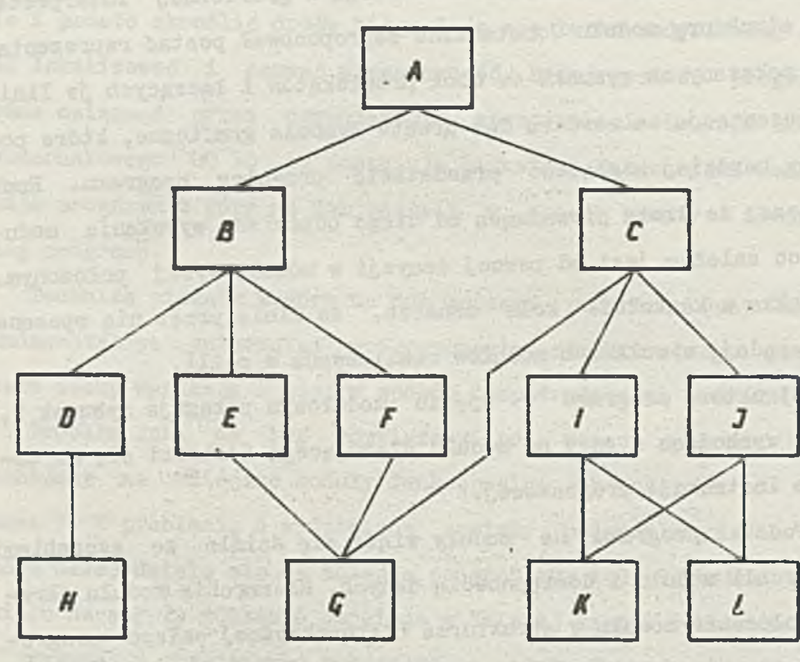
Aby zwiększyć zawartość informacji w graficznej interpretacji struktury modułu Constantine zaproponował postać reprezentacji pokazany na rysunku 4. Obok prostokątów i łączących je linii reprezentacja ta zawiera dwa proste symbole graficzne, które powinny bardziej poglądowo przedstawić przebieg programu. Romb oznacza, że linia prowadząca od niego odpowiada wywołaniu modułu, co zależne jest od pewnej decyzji w module wyżej położonym. Strzałka w kształcie koła oznacza, że linie przez nią opasane odpowiadają wywołaniom modułów realizowane w pętli.

Strukturę programu w ujęciu modułowym pokazuje rysunek 5, gdzie wychodząc z góry od modułu sterującego dochodzi się na dole do instrukcji programowej.

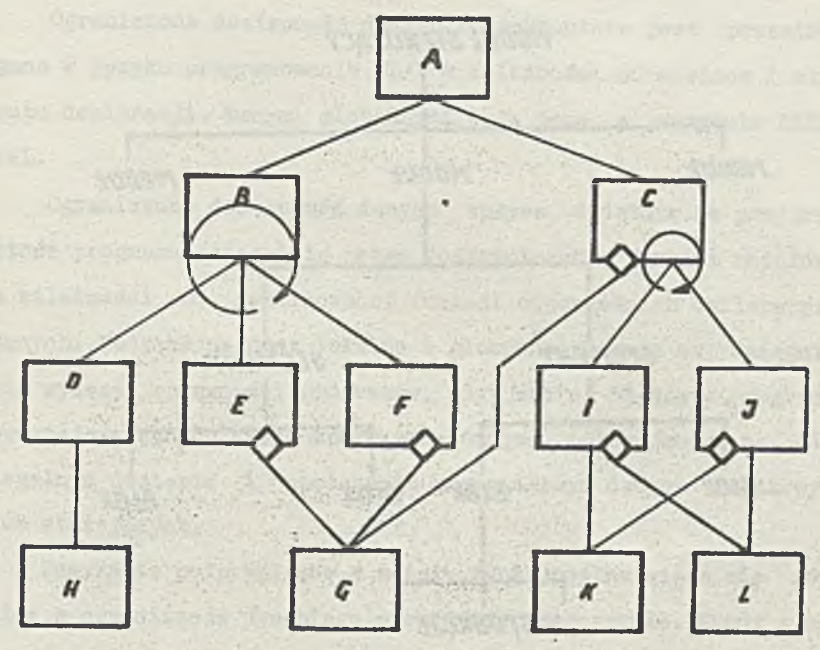
Podział programu na moduły wiąże się ściśle ze szczeblami hierarchii modułu i dostępnością danych. Hierarchię modułu określa położenie modułu w strukturze hierarchicznej całego programu. Każdy z modułów może operować na danych dostępnych w danym module oraz na danych z modułów podrzędnych z niższych szczebli hierarchii.

Dane w zależności od ich dostępności można podzielić na lokalne i globalne. Dane globalne dostępne są na każdym szczeblu hierarchii natomiast dane lokalne są dostępne jedynie w ramach jednego modułu oraz w modułach nadrzędnych dla niego.

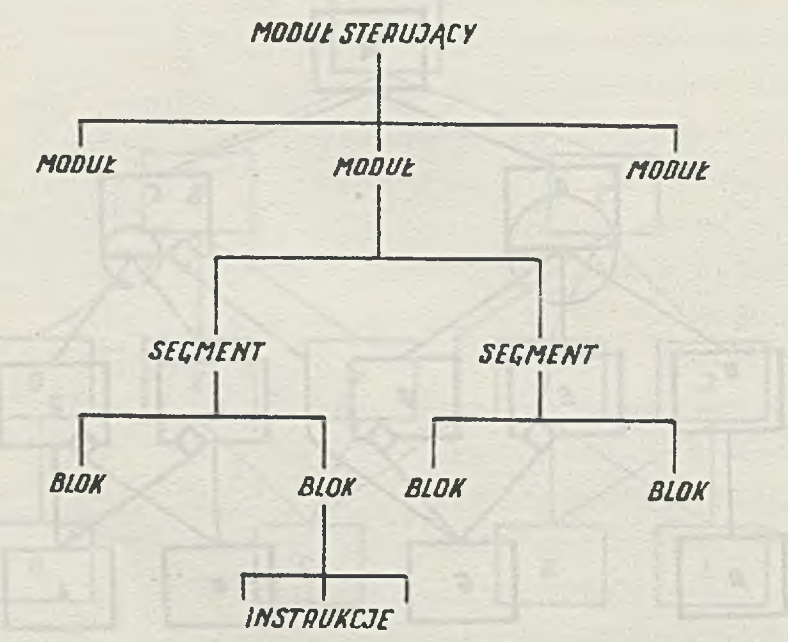
Rysunek 6 pokazuje struktury danych dostępnych dla trzech modułów P1, P2 i P3. Część danych zadeklarowanych w sposób sta-



RYS.3. STRUKTURA PODPROGRAMÓW BEZ INTERPUNKCJI PRZEBIEGU



RYS. 4. STRUKTURA PODPROGRAMÓW WRAZ Z INTERPUNKCJĄ PRZEBIEGU WEDŁUG CONSTANTINE'A



RYS.5. STRUKTURA PROGRAMU

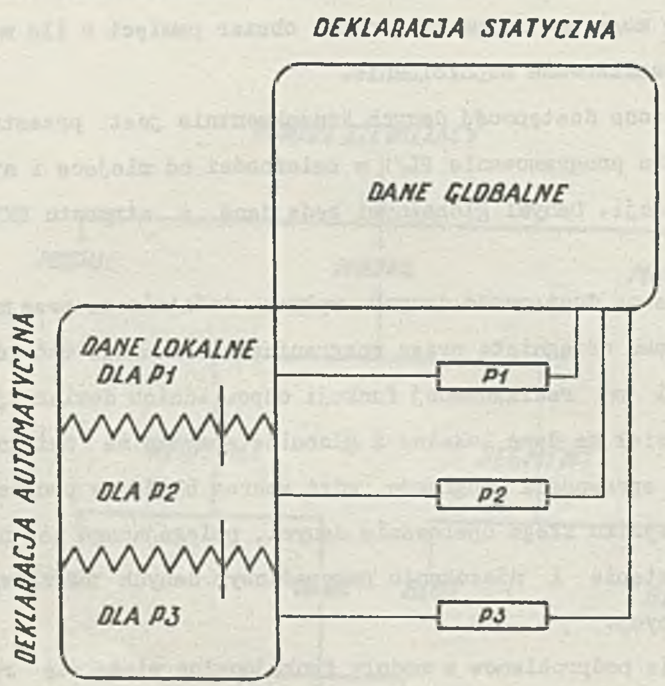
tyczny stanowi dane globalne dostępne dla wszystkich modułów. Dane wymagane jedynie lokalnie poprzez poszczególne moduły są deklarowane w sposób dynamiczny w momencie gdy występuje potrzeba przez co mogą one zajmować ten sam obszar pamięci o ile moduły nie są realizowane współbieżnie.

Ograniczona dostępność danych konsekwentnie jest przestrzegana w języku programowania PL/1 w zależności od miejsca i atrybutu deklaracji. Danymi globalnymi będą dane z atrybutu EXTERNAL.

Ograniczona dostępność danych wpływa dodatnio na przejrzystość programu osiągniętą przez rozgraniczenie funkcji modułów i w zależności od realizowanej funkcji odpowiednich deklaracjach danych. Podział na dane lokalne i globalne pozwala na osiągnięcie wyższej sprawności programów, gdyż szereg błędów w programie powstaje w wyniku złego operowania danymi, polegającego na nielegalnym dostępie i niszczeniu przypadkowym danych użytkowych lub sterujących.

Zamykanie podproblemów w moduły funkcjonalne wiąże się również z organizacją przebiegu sterowania w programie. Każdy z modułów winien mieć jedno wejście i jedno wyjście. Taka konstrukcja narzuca warunek zamknięcia zakresu instrukcji warunkowych do ram jednego modułu co oznacza, że decyzja napotkana w jednym module nie może wpływać na wykonanie innych modułów lub pokazywać sterowanie do innych modułów.

Generalnie sterowanie winno być skoncentrowane w jednej, głównej części programu a poszczególne moduły wywoływane z niej w zależności od potrzeb w kolejności góra, dół.



RYS.6. STRUKTURA DANYCH

Podział programu na moduły funkcjonalne pozwala na wykorzystanie symulacji przy testowaniu programu. Testowanie programu, dokładniej konstrukcji jego struktury, można rozpocząć w momencie określenia modułów z jakich program ma się składać i zbudowania struktury hierarchicznej programu. Poszczególne moduły mogą być symulowane w modelu i w miarę ich przekazywania zastępowane faktycznymi, oprogramowanymi i wytestowanymi odrębnie ciągami instrukcji. Takie wykorzystanie symulacji pozwala na bardzo wczesne rozpoczęcie testowania programu i co się z tym wiąże skrócenie czasu przekazania wykonanego, sprawnego całego programu.

Tak wykorzystana technika symulacji może być również wykorzystywana na etapie budowy całego oprogramowania systemu, gdzie obiektem symulowanym jest pojedyńczy program a obiektem badanym cały system, w szczególności jego oprogramowanie.

Do podanych cech ogólnych programowania strukturalnego T. Baker wprowadza ponadto odpowiednie wymogi organizacji prac programowych przez co jeszcze w większym stopniu podnosi efektywność programowania.

2.3. Dlaczego bez GO TO ?

Podział programu na moduły funkcjonalne nie stanowi jeszcze o stosowaniu metod programowania strukturalnego. Podstawowym problemem programowania strukturalnego jest przekazywanie sterowania. Wcześniej już zostało powiedziane, że moduł charakteryzuje się jednym wejściem i jednym wyjściem oraz, że w programie

należy konsekwentnie przestrzegać hierarchii góra - dół.

Rysunek 7 pokazuje jak z tych struktur podstawowych budować program poprzez wzajemne zanurzanie w sobie poszczególnych podstawowych struktur.

Struktura sekwencyjna jest prostym ciągiem instrukcji bezwarunkowych, które wykonywane są w kolejności występowania, bez przekazywania sterowania w inne miejsca programu.

Struktura wyboru może być zrealizowana przez instrukcje typu:

IF ..... THEN ...

IF ..... THEN ... ELSE ...

CASE ...

czyli instrukcje, które w zależności od stanu warunku realizują jeden z wybranych ciągów instrukcji bezwarunkowych.

Struktura powtarzania realizowana przez instrukcje:

DO ... WHILE ...

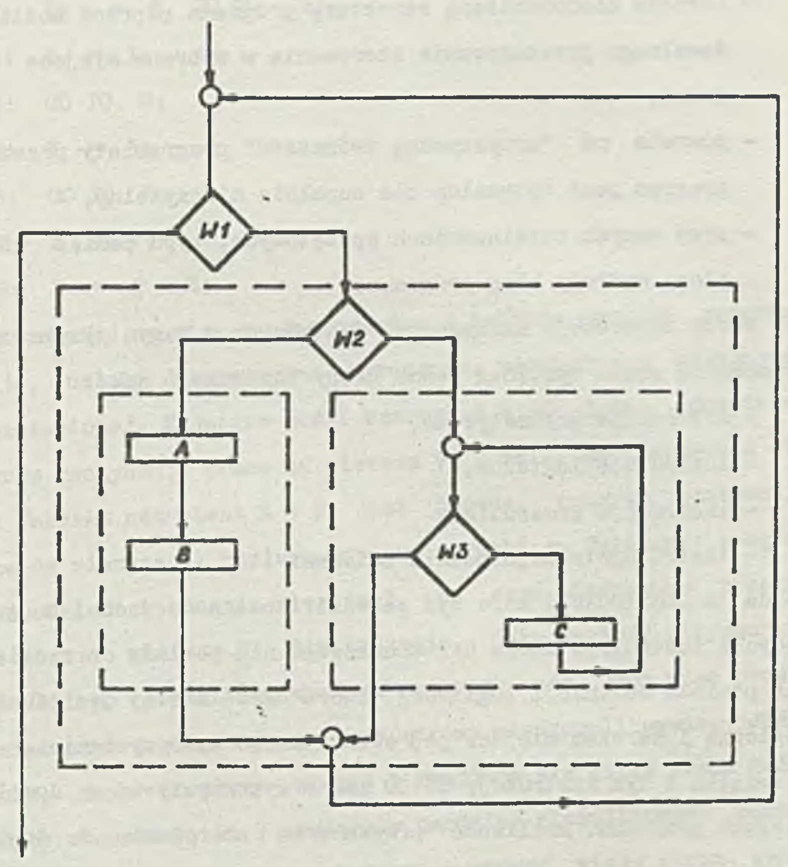
DO ... UNTIL ...

jest przykładem wielokrotnego powtarzania ciągu instrukcji bezwarunkowych czyli organizowania pętli programowych.

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

Pod adresem instrukcji GO TO zostało sformułowanych szereg zarzutów, z których podstawowe to:

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



RYS.7. PRZYKŁAD ZANURZANIA PRZEBIEGÓW PODSTAWOWYCH

- narusza hierarchiczną strukturę programu poprzez możliwość dowolnego przekazywania sterowania w wybrane miejsce programu,
- pozwala na "artystyczną twórczość" programisty przez co program jest optymalny ale 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 jak:
- 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 posiada przydziału. Nie posiada definicji logicznej, która określała by jej funkcję logiczną i zarazem miejsce jej ewentualnego wykorzystania. W związku z tym instrukcję GO TO można wykorzystywać w dowolnym miejscu programu. Możliwość przekazania sterowania w dowolne miejsce programu a ponadto możliwość modyfikowania tej instrukcji /np. przy pomocy instrukcji ALTER w COBOLU/ powoduje takie splątanie dróg sterowania w programie, że szczegółowa analiza realizacji biegu programu jest niezmiernie kłopotliwa.

Dla przykładu przeanalizujemy wycinek programu napisanego przy użyciu PL/1, gdzie rozkaz GO TO jest używany 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;
```

Aby zrozumieć co się w tym prostym i krótkim wycinku programu dzieje, należy rozpatrywać wyrażenia warunkowe w kolejności przedstawionej. Najpierw jeśli warunek A = 10 jest prawdziwy program kontynuuje pracę od wiersza 7. Z kolei gdy warunek A =10 jest fałszem natomiast A = 5 jest prawdą, program kontynuuje pracę od wiersza 5. Dopiero gdy oba warunki są fałszem, program przechodzi do wykonania wiersza 3, po czym sterowanie przekazane jest do wiersza 8. Stwierdzenie, jaką wartość przyjmuje Y po przebiegu przez ten wycinek programu nie jest więc proste chociaż samo zagadnienie jest wyjątkowo nieskomplikowane. Jeśli jednak, nasz docelowy wiersz 8 znajduje się kilka stron dalej i gdyby wiersze 3, 5, 7 zawierały bardziej skomplikowane procedury obliczeniowe, trzeba byłoby wertować wiele stron, zanim zdołalibyśmy rozeznać, co program realizuje.

Ten sam wycinek programu można zapisać w postaci:

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

Postać ta nie wykorzystuje instrukcji GO TO, dzięki czemu program jest łatwiejszy do analizowania.

Programowanie strukturalne jako podstawową cechę programu stawia jego przejrzystość. Wykorzystanie instrukcji skoku bezwarunkowego jest zaprzeczeniem tego i w związku z tym jednym z haseł programowania strukturalnego jest programowanie bez GO TO.

Odpowiednie manipulowanie instrukcją GO TO pozwala na optymalizację programu pod względem zajętości pamięci i w zależności od inwencji programisty program może być pod tym względem dziełem artystycznym. Dzieło takie jest jednak zrozumiałe tylko przez autora i to przez pewien okres czasu od napisania.

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 tą samą funkcję napisane bez GO TO przez różnych programistów są do siebie bardzo podobne w przeciwieństwie do identycznie powstałych programów gdy można wykorzystywać skok bezwarunkowy GO TO.

W maszynach drugiej generacji program podczas realizacji w maszynie musiał w całości i przez cały czas biegu rezydować w pamięci operacyjnej. Z uwagi na taką organizację biegu programu optymalizacja zajętości pamięci, osiągana przez wykorzystanie GO TO, miała swoje uzasadnienie.

Już wprowadzenie techniki nakładania, czyli budowa programów nakładkowych, gdzie w pamięci rezyduje jedynie moduł główny i w zależności od potrzeb przywołuje on z pamięci zewnętrznej kolejne nakładki, które zajmują to samo miejsce w pamięci, wprowa-

dziło pewne ograniczenia na stosowanie instrukcji GO TO. Skoki bezwarunkowe winny się w zasadzie ograniczać do jednej nakładki gdyż każdy skok z nakładki do nakładki zmuszał do wykonania transmisji z pamięci zewnętrznej do pamięci operacyjnej co powodowało, że oszczędność na zajętości pamięci nie rekompensowała strat czasowych potrzebnych na manipulowanie nakładkami.

Problem stał się jeszcze bardziej widoczny przy zastosowaniu stronicowania pamięci operacyjnej. Podział programu na "strony" pamięci operacyjnej realizowany jest 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 realizowane jest odwołanie. Częste używanie instrukcji GO TO może spowodować, że czas efektywnego biegu programu będzie nikłym procentem przy całkowitym czasie realizacji.

Widać z tego, że nowe rozwiązania techniczne wymuszają ograniczenie stosowania instrukcji skoku bezwarunkowego przez co wychodzą naprzeciw postulatom programowania strukturalnego.

2.4. Sterowanie w programie

Zakaz używania instrukcji skoku bezwarunkowego GO TO nie rozwiązuje problemu o ile nie zostanie dostarczony programiście aparat umożliwiający realizację funkcji dotychczas realizowanych przy pomocy GO TO. Jeśli programista będzie dysponował wygodnym, łatwym w stosowaniu aparatem umożliwiającym realizację podstawowych cech programowania strukturalnego to bez żalu zrezygnuje

się tylko z GO TO, ale również innych rozwiązań sprzecznych z zasadami programowania strukturalnego.

Przyjmiemy jako podstawowe cechy programowania strukturalnego:

a/ Przebiegiem programu powinny sterować tylko trzy podstawowe struktury przebiegu:

- sekwencja,
- wybór,
- powtarzanie,

pokazane na rysunku 2.

b/ Program powinien składać się z kilku logicznie zamkniętych, hierarchicznie uporządkowanych bloków, mających zawsze po jednym wejściu i wyjściu jak na rysunku 5.

c/ Program winien mieć dane o strukturze dostosowanej do swojej struktury funkcjonalnej w taki sposób, żeby każda część programu rozporządzała tylko takimi danymi, jakie potrzebne jej są do spełnienia swojej funkcji /rys. 6/.

Biorąc pod uwagę te trzy własności język programowania powinien:

- wspomagać trzy struktury podstawowe przebiegu z pkt. a,
- umożliwiać dzielenie programu na małe jednostki kodowe z jednym wejściem i wyjściem, oraz
- umożliwiać stosowanie struktur danych z dostępem ogólnym i ograniczonym /dane GLOBAL i LOCAL/.

Z dwóch ostatnich wymagań wynikają dwa dalsze, a mianowicie:

- części programów muszą dawać się często sprzęgać i wstawiać jedna w drugą,
- możliwość udostępniania wspólnych danych jak i wymiany własnych danych.

Tablica z rysunku 8 pokazuje jak niektóre języki programowania wspomagają stosowanie struktur podstawowych. Wystąpienie znaku X oznacza występowanie danej instrukcji lub możliwości w danym języku programowania natomiast znak - oznacza brak tej możliwości. Gdy znak X jest ujęty w nawiasy oznacza to, że możliwość osiągnięcia zamierzonego skutku nie jest dostępne bezpośrednio a jedynie poprzez zastosowanie odpowiednich specjalnych rozwiązań programowych.

Z tabeli tej widać, że językiem najlepiej wspierającym programowanie strukturalne w realizacji cechy sterowania jest język PASCAL, który powstawał wówczas gdy koncepcja programowania strukturalnego była już lansowana. Z innych języków dość duże wspomaganie dla stosowania struktur podstawowych posiadają PL1, ALGOL i COBOL. Z języków wyższego rzędu FORTRAN jedynie w minimalnym stopniu wspomaga sterowanie przy użyciu struktur podstawowych podobnie jak język podstawowy Assembler.

Biorąc pod uwagę możliwość dzielenia programu na moduły, wspomaganie ze strony wymienionych języków programowania rozkłada się podobnie.

FORTRAN daje najmniejsze możliwości podziału. Program w FORTRANIE 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 tylko tym, że mogą one obliczać tylko jedną wartość. Struktura programu w FORTRANIE może przyjąć postać pokazaną na rysunku 9.

Podprogramy funkcji mają zasadniczo po jednym punkcie wej-

	FORTRAN	COBOL	PL/I	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)

( ) - OZNACZA WYSTĘPOWANIE W OGRANICZONEJ POSTACI

RYS. 8. PORÓWNANIE STEROWANIA W RÓŻNYCH JĘZYKACH PROGRAMOWANIA

ścia i wyjścia, ale podprogramy mogą mieć kilka wejść /ENTRY/ oraz po kilka wyjść /RETURN/. Programista chcący programować strukturalnie w FORTRANie musi zrezygnować z tych możliwości.

COBOL dopuszcza więcej możliwości podziału programu. I tak program można podzielić na sekcje, a te z kolei na paragrafy. Paragraf zawiera jedną lub kilka instrukcji, sekcja - jedną lub kilka paragrafów. Paragrafy można porównać z blokami w PL1, ale w przeciwieństwie do bloków paragrafy nie dają się fizycznie wbudować jeden w drugi. Można to osiągnąć poprzez zastosowanie instrukcji PERFORM jak pokazuje poniższy przykład:

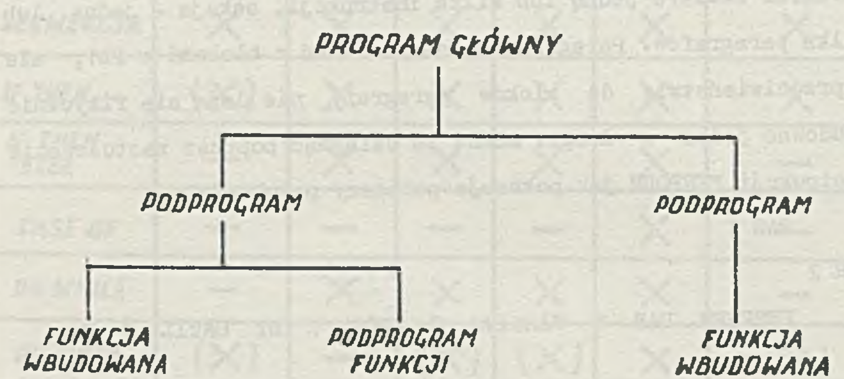
```
    PAR 1
PAR 2
    PERFORM PAR 1 VARYING J FROM 1 BY UNTIL J N
    ...
    PAR 3
```

Paragraf PAR 1 logicznie wbudowany w paragraf PAR 2 fizycznie znajduje się na początku lub końcu programu przez co nie przyczynia się do przejrzystości i jest szczególną wadą COBOLu.

Dalszą wadą COBOLu jest to, że nie posiada możliwości stosowania podprogramów funckji, a podprogramy zewnętrzne mogą mieć po kilka wejść i wyjść.

Niewątpliwą zaletą COBOLu jest natomiast możliwość stosowania instrukcji COPY przez co można wbudować w kod źródłowy program sekcje i paragrafy wszędzie tam, gdzie są one potrzebne do wykonania. Struktura programu w COBOLu może wyglądać tak, jak pokazano na rysunku 10.

PL 1 daje najwięcej możliwości podziału. PL 1 ma strukturę



RYS. 9. STRUKTURA PROGRAMU W JĘZYKU FORTRAN

RYS. 10. STRUKTURA PROGRAMU W JĘZYKU COBOL

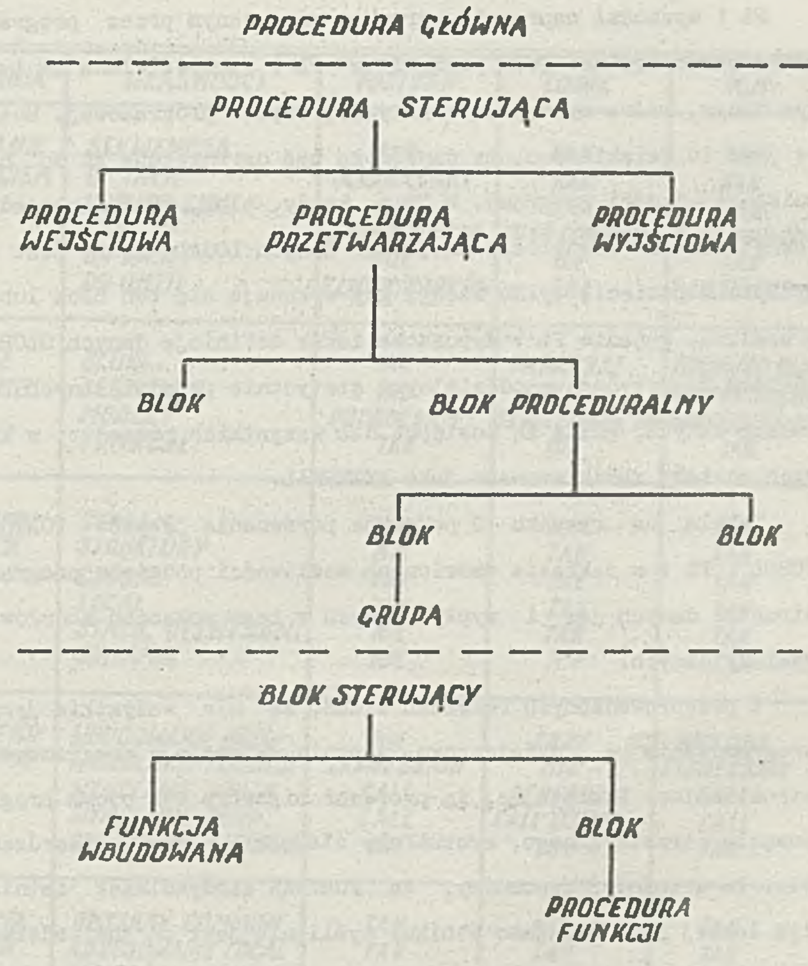
blokową, uznaje zarówno bloki proceduralne /PROC/ jak i bloki BEGIN czy DO, które dają się fizycznie wstawiać jeden w drugi. Procedury mogą być względem siebie zewnętrzne lub wewnętrzne, w PL 1 można stosować funkcje wbudowane i podprogramy funkcji.

Bloki BEGIN względnie DO mogą być wzajemnie zawarte w sobie przez co osiągamy fizyczne wstawianie bloków jednego w drugi. Strukturę programu w PL 1 pokazuje rysunek 11.

Jedynym odstępstwem od zasad programowania strukturalnego w PL 1 jest możliwość organizacji wielu wejść i wyjść do bloków.

Jeśli chodzi o struktury danych i ograniczenia dostępności danych do FORTRAN spełnia tylko pewną część wymogów programowania strukturalnego. Istnieje podział na dane GLOBAL i LOCAL. Dane nie określone jako COMMON, znane są tylko w programie głównym względnie podprogramie, w którym są zadeklarowane. Dane deklarowane jako COMMON są dostępne zarówno dla programu głównego jak i dla wszystkich podprogramów deklarujących je. Podczas przebiegu programu obecne są wszystkie pola wszystkich części programu. Wadą jest brak możliwości określenia hierarchicznych struktur danych.

COBOL pod względem struktur danych posiada dużo mniejsze możliwości. W COBOL-u 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 COBOLu nie ma również



RYS. 11. STRUKTURA PROGRAMU W JĘZYKU PL/1

dynamicznego przykładu pamięci.

PL 1 wychodzi naprzeciw warunkom narzucanym 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 ma obszar danych LOCAL, który jest fizycznie w pamięci, tylko wtedy, gdy wykonuje się ten blok lub ta procedura. 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.

Tabela na rysunku 12 pokazuje porównanie języków FORTRAN, COBOL i PL 1 w zakresie omówionych możliwości podziału programu, struktur danych jak i wypływających z tego połączeń modułów i wymiany danych.

Z przeprowadzonych rozważań widać, że nie wszystkie języki programowania w dostatecznym stopniu wspomagają programowanie strukturalne. Pamiętając, że profesor Dijkstra był ojcem 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 ...".

Języki później powstałe zawierają więcej elementów wspierających programowanie strukturalne.

WYMAGANIA	WŁASNOŚCI	FORTRAN	COBOL	PL/I
STEROWANIE PRZEBIEGIEM	SEKWENCJA	TAK	TAK	TAK
	IF THEN	OGRANICZONY	TAK	TAK
	IF THEN ELSE	NIE	TAK	TAK
	CASE OF	COMPUTER TO GO	GO TO DEPENDING	GO TO MARKENVAR
	DO WHILE	NIE	NIE	TAK
	DO UNTIL	Z LICZN. KROKÓW	TAK	Z LICZN.KROKÓW
PODZIAŁ PROGRAMU	BLOKI	NIE	PARAGRAF	BEGIN/DO BLOK/
	SEGMENTY	NIE	SEKCJA	PROCEDURA WEWN.
	MODUŁY	PODPROGRAM	ZEWNĘTRZNE UP	PROCEDURA ZEWN.
	FUNKCJE	TAK	NIE	TAK
STRUKTURY DANYCH	STREAM	TAK	NIE	TAK
	STRUKTURY	NIE	TAK	TAK
	GLOBAL	TAK	NIE	TAK
	LOCAL	TAK	TAK	TAK
	STATIC REENTRANT	NIE	TAK	TAK
	DYNAMIC	NIE	NIE	TAK
POŁĄCZENIE PROGRAMU	WBUDOWANIE KODU	NIE	COPY	INCLUDE
	WYWOŁANIE FUNKCJI	EXPRESSION	NIE	EXPRESSION
	WYWOŁANIE WEWN.	CALL	PERFORM	CALL
	WYWOŁANIE ZEWN.	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

RYS. 12. PORÓWNANIE JĘZYKÓW PROGRAMOWANIA

3. Programowanie modularne

3.1. Określenie modułu

Jedną z cech programowania strukturalnego jest podział programu na moduły funkcjonalne. Rozwój tej cechy doprowadził do wykształtowania się programowania modularnego.

Techniczną podstawę programowania strukturalnego stanowi technika podprogramów stosowana dość szeroko przy tradycyjnych metodach programowania. W miarę jak zwiększała się kompleksowość systemów przetwarzania danych, coraz większą uwagę zwracano na zasadę podprogramów - modułów, gdyż można od tej techniki pracy oczekiwać lepszej przejrzystości i elastyczności tworzonych programów.

Oparta na tej zasadzie technika programowania nosi nazwę programowania modularnego, a wydzielone bloki nazwano modułami.

Programowanie modularne przebiega w następujący sposób:

1. Podział problemu na kilka logicznych jednostek funkcyjnych.
2. Jednostki te dzieli się według pewnej hierarchii.
3. Przyporządkowanie modułów tym jednostkom.
4. Programowanie i testowanie modułów jako zamkniętych jednostek.
5. Integracja modułów w system programów w celu wykonania danych funkcji.

Taki sposób postępowania podlega wymogom i warunkom różnych zadań jak: planowanie, kierowanie, testowanie czy konserwacja

modułów.

Przeprowadzenie podziału problemu na funkcjonalne bloki wymaga dokładnego zrozumienia problematyki.
Ten zwiększony nakład pracy na planowanie umożliwia wczesne rozpoznanie i usunięcie większości logicznych luk. Podczas planowania i realizacji techniki modułowej kilkakrotnie nasuwa się okazja do sprawdzania.

Wyraźny podział funkcji na człony daje lepszą przejrzystość. Ma to szczególne znaczenie przy złożonych systemach i zmniejsza ryzyko błędnego projektowania.

Możliwość przydzielania pracy według stopnia trudności pozwala na uwzględnienie kwalifikacji programistów i równoczesne określenie odpowiedzialności każdego z nich, a rozważna standaryzacja modułów upraszcza kontrolę rezultatów programowania. Dysponowanie programistami staje się elastyczniejsze. Elastyczne dysponowanie pracownikami umożliwia elastyczne reagowanie na terminy. Częste duże zapotrzebowanie na siłę roboczą występuje tylko przez krótki okres czasu. Szybkie zmniejszenie szczytowego zapotrzebowania prowadzi do oszczędności kosztów, a kontynuacja jest zapewniona, ponieważ projekt może być dalej opracowywany przez stały personel. Do dalszej oszczędności kosztów może przyczynić się zlecenie kodowania na zewnątrz, a programiści opracowują opisy modułów.

Implementacja systemu programów wiąże się najczęściej z pewną reorganizacją systemu, dlatego też wskazane jest stopniowe przeprowadzanie jej. Programowanie modułowe jest dostatecznie elastyczne, aby sprostać temu wymaganiu.

Trzeba w tym jednakże widzieć pewne niebezpieczeństwo. Wyraźne ustalenie odpowiedzialności przy programowaniu modułowym programiści rozumieją przeważnie jako zbyt małą swobodę manipulowania, a zatem możliwości rozwoju.

Zalety testowania modułów wynikają z faktu, że manipuluje się tu małymi jednostkami, stanowiącymi dla siebie logiczną całość. W przeciwieństwie do konwencjonalnej metody testowania, w której drogą przebadania zadań stwierdzone zostaje czy program funkcjonuje, w technice modułowej uwaga zwrócona jest na poszczególne decyzje /IF/ w module - jako przedmiocie badania. Stwarza to możliwość systematycznego projektowania i przeprowadzania większości testów modułów. Zlokalizowanie jakiegoś błędu jest prostsze.

Przejrzysty podział modułu przyczynia się do łatwiejszego zrozumienia podzielonego na moduły systemu programów, krótszy jest więc czas potrzebny na zapoznanie się z problemem. Również dokumentacja ma układ modułowy. Schemat blokowy jest stosunkowo mały i łatwiej rozpoznać przebieg programu. Przy projektowaniu modułu, części podlegające konserwacji należy z jednej strony mocno odizolować od otoczenia, z drugiej zaś powinny one być tak pomyślane, aby dawały się dopasować. Programowanie modułowe jest alternatywą kilku technik programowania. Ma one wprawdzie wiele zalet, ale nie jest bezproblemowe. Dopiero właściwe posługiwanie się nim przynosi efekty. Programiści muszą zmienić sposób myślenia. Proces ten związany jest przeważnie ze szkoleniem i często potrzebne jest wpierw praktyczne doświadczenie. Podstawowe prace projektowe mają wprawdzie korzystny wynik, mogą

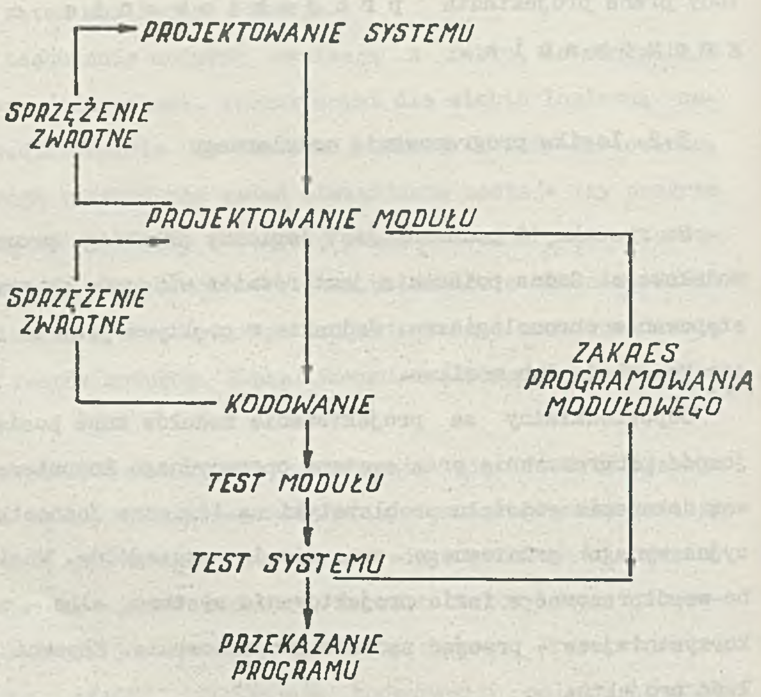
one jednakże prowadzić do nadmiernych kosztów i zakładają na ogół wyższe kwalifikacje programisty. Pomiędzy projektowanie systemu a kodowanie musi być włączona nowa dla konwencjonalnej metody praca projektanta p r o j e k t o w a n i e o p r o - g r a m o w a n i a .

3.2. Logika programowania modularnego

Na rysunku 13 pokazany jest logiczny przebieg programowania modułowego. Godne polecenia jest również widoczne na rysunku postępowanie chronologiczne. Jednakze w praktyce jest to nie zawsze konieczne lub możliwe.

Odpowiedzialny za projektowanie modułów musi posiadać znajomość programowania oraz systemu operacyjnego komputera, ponieważ dokonanie podziału problematyki na logiczne jednostki funkcyjne wymaga gruntownego zrozumienia szczegółów. Musi on albo współpracować w fazie projektowania systemu, albo - co jest korzystniejsze - przejąć zadania projektowania. Zapewni to ciągłość projektu.

Niektóre funkcje są standardowe prawie dla wszystkich systemów programów, np. identyfikacja kart wejściowych i zbiorów, manipulowania wejście-wyjściem, przygotowania list. Wiele błędnych manipulacji może być już tu wykrytych, jednakże niektóre warunki błędów zostaną rozpoznane dopiero przy opisywaniu modułu lub przy sporządzaniu schematu blokowego.
Właściwe przetwarzanie danych /np. aktualizacja/ różni się zazwyczaj bardzo w zależności od problemu, dlatego też trudno jest



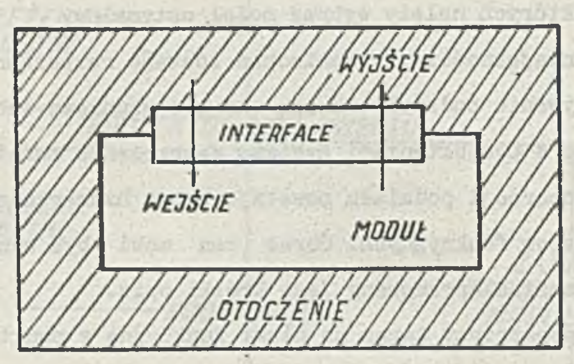
RYS.13. PROGRAMOWANIE MODUŁOWE

ustanowić generalne zasady podziału funkcji. Sposoby definiowania problemów są bardzo zróżnicowane i stąd ich hierarchiczne podziały na bloki funkcyjne będą miały odmienny wygląd. Problem musi być gruntownie przeanalizowany i w końcu ujęty znowu w pojedyńcze jednostki logiczne. Konieczna jest więc gruntowna znajomość problematyki. Dla tego samego problemu może powstać kilka modeli, z których należy wybrać model optymalny.

Większość niejasności i zaniedbań zostaje rozpoznanych dopiero przy dokonywaniu podziału. Muszą one być usunięte lub uzupełnione wspólnie z projektantami systemu /sprzężenie zwrotne/. Po zakończeniu operacji podziału powstaje obraz hierarchicznej struktury jednostek funkcyjnych. Obraz ten musi być udokumentowany oraz musi zostać opracowany jego krótki opis.

Jeśli popatrzymy teraz na bloki funkcyjne z punktu widzenia techniki programowania, to moduł przedstawia się jako zamknięta część programu, która spełnia określone funkcje logiczne.Najważniejszymi elementami podstawowymi modułu są informacje potrzebne do wykonania /wejścia do modułu/, przetwarzanie wejścia i generowanie wyników /wyjście z modułu/. Wymiana informacji ze środowiskiem może następować tylko za pomocą tzw. interfejsu /rys. 14/.

Ten interfejs jest najważniejszą częścią składową programowania. Należy nań zwracać szczególną uwagę. W tej fazie projektu muszą być ustalone i dokładnie udokumentowane atrybuty pól danych jak nazwa, długość i format. Uaktualnienie interfejsu, które będzie konieczne ze względu na zmiany otoczenia lub samego modułu, musi być przeprowadzane centralnie przez odpowiedzialną osobę "admi-



RYS.14. INTERFEJS MODUŁU

nistratora danych", aby uniknąć zamieszania powodowanego przez nieupoważnione lub niezamierzone wtargnięcia w definicję interfejsu. Sporządzona w ten sposób lista pól danych wraz z kompletnymi objaśnieniami jest częścią składową dokumentacji przy przekazywaniu programu i ułatwia konserwację. W definicjach pól danych określone jest jakie pola są w dyspozycji jako wejścia informacji i co one zawierają wchodząc do modułu. Ponadto powiedziane jest jakie powinny być generowane wyniki, gdzie mają być one zapamiętane, co zostanie zrobione w przypadku błędu i jak powinny być ustalone warunki, które wpływają na dalszy przebieg programu.

Jak duży powinien być moduł ? Nie ma w tym zakresie żadnych norm. Liczba instrukcji powinna zostać ograniczona do kilkuset, aby zagwarantować sens i cel programowania modułowego. Jak dalece skomplikowany może być moduł ? Przez pytanie to rozumie się w pierwszym rzędzie liczbę logicznych decyzji. W miarę jak pozwala na to logika problematyki liczba ta powinna być mała ze względu na późniejsze testowanie modułu.
Jak często używany jest moduł ? Ten atrybut modułu ma znaczenie dla struktur OVERLAY. Częściej używane modele powinny dłużej pozostawać w pamięci. To częste używanie należy mieć szczególnie na uwadze podczas przeprowadzania konserwacji. Czy moduł użytkowany jest przez kilka innych modułów ? Podział modułu można przeprowadzić w ten sposób, żeby powstało możliwie dużo wspólnych modułów. Nie jest to jednak godne polecenia. Pierwszeństwo powinny mieć logika problematyki i prostota ukształtowania modułów. Jeśli jakiś moduł ma być użytkowany przez wiele innych mo-

dułów, to musi on być dokładnie dopasowany do specjalnych życzeń. Taki moduł może wkrótce stracić swoje zalety, a zmiany otoczenia połączone są z wielkimi trudnościami. Moduły muszą więc być możliwie wyraźnie zdefiniowane.

Jak już wspomniano, zaletą techniki modułowej jest łatwość przeprowadzania konserwacji. Zaletę tę można jeszcze rozwinąć, jeśli zwróci się na to uwagę w fazie projektowania modułu, kładąc na przykład mocny nacisk na wyodrębnienie modułu. Nie powinno się dopuszczać przelatania się modułów.

Wejście i wyjście: Moduł powinien mieć tylko jedno wejście i jedno wyjście, które muszą być zgodne z konwencją powiązań.

Używa się kilku określeń dla modułu sterującego: root module, manline, main lub root segment. W skrajnym przypadku moduł sterujący zawiera tylko instrukcje aktywizacji modułów. W dużych systemach może on mieć złożoną postać /w szczególności z powodu operacji WE/WY/ i powinien go napisać i wytestować doświadczony programista /najlepiej projektant modułów/.

Głównym zadaniem modułu sterującego jest sterowanie scentralizowanymi wejściami i wyjściami. Należą tu np.: OPEN, CLOSE i manipulowanie błędami WE/WY. Moduł sterujący musi dostarczać informacji innym modułom.

Kolejnym ważnym zadaniem jest logiczne sterowanie modułem przetwarzania, tj. przebiegiem przetwarzania. Najprotszym sposobem realizacji sterowania jest ustawienie ciągu instrukcji CALL, ale sterowanie zależy przeważnie od kilku kryteriów.

Decyzja o ewentualnym nienormalnym zakończeniu wykonania programu powinna następować w module sterowania i powinny być tu wpro-

wadzone czynności wymagane dla manipulowania błędami.

Moduły obsługi błędów są konieczne we wszystkich systemach programów. Typowymi częściami składowymi są wyprowadzanie komunikatu alarmującego o błędach i zakończenie przetwarzania. Prawie zawsze trudno jest przeprowadzić korektę umożliwiającą dalsze przetwarzanie.

Przeważnie dąży się do tego by warunkami błędów zarządzać przy pomocy jednego modułu. Nie zawsze daje się to zrealizować. Drugą skrajnością byłoby manipulowanie warunkami błędów decentralnie - w miejscu ich powstawania. Zaprzecza to jednakże technice modułowej. Ponieważ z jednej strony moduły te /przede wszystkim te, które znajdują się na najniższym poziomie/ muszą tworzyć zamknięte jednostki, tj. nie mogą one decydować o tym czy program powinien być przerwany, musi to być zdecydowane na wyższym poziomie, aby zabezpieczony był przegląd warunków błędów. Z drugiej strony moduły te muszą być uwolnione od operacji WE/WY.Problemy występują również wtedy, gdy manipulowanie błędami zostaje scentralizowane w jednym module.

Godne polecenia jest zawsze centralne zarządzanie wejściem/wyjściem, przy czym jest ono nadzorowane tylko przez moduł sterujący. Wszystkie inne moduły są całkowicie wolne od operacji WE/WY. Centralizacja uwarunkowana jest nie tylko technicznie przez program ale jest konieczna również ze względów organizacyjnych. Zdecentralizowane operacje WE/WY wymagałyby możliwości dokładnego śledzenia przez jakie moduły,rekordy zostały przeczytane lub zapisane. Gdyby to w ogóle było możliwe, zależność zwiększyłaby się do tego stopnia, że zaprzeczony zostałby sens i

cel techniki modułowej. Ścisłe przestrzeganie centralizacji może jednakże nadmiernie skomplikować moduł sterujący. W module tym powinien być również zrealizowany dostęp do jakiegoś zbioru, dotyczący tylko jednego modułu.

Odpowiedzialny programista musi przed przystąpieniem do kodowania zapoznać się z opisami interfejsów, konwencjami powiązań i zadaniami danego modułu. Mając dobrą dokumentację projektu modułu może on pracować samodzielnie. W przypadku złożonych modułów potrzebna mu będzie pomoc. Rozpoczyna on normalnie od schematu blokowego. Zostają znowu rozpoznane niejasności i luki. Od tej fazy aż do zakończenia tekstu modułu zapotrzebowanie siły roboczej osiąga maksimum.

Przy konwencjonalnym testowaniu prawidłowość systemu programów można zbadać dopiero za pomocą tabulogramów wyjściowych. Zostają przy tym dobrze wytestowane przewidziane sytuacje normalne, ale sytuacje wyjątkowe i wiele kombinacji czynników mających wpływ na wynik nie jest sprawdzane. Dlatego też częściej występują błędy i system programów wymaga częstej konserwacji. Konsekwentny test modułu może ten problem w znacznej mierze rozwiązać. Przedmiotem badania podczas przeprowadzania testu modułu są logiczne rozgałęzienia programu. Moduł jest bez zarzutu, gdy całkowicie wystestowane są wszystkie jego rozgałęzienia. Moduł nie jest zależny od zewnętrznych operacji WE/WY, a zatem nie ma żadnego wyjścia w postaci tabulogramu, który mówiłby coś o sposobie jego pracy. Konieczny jest więc inny sposób postępowania. Proponuje się tu badanie rozgałęzień modułu. Zastosowanie tej metody testowania umożliwia usystematyzowanie testu programu oraz

dokładne wytestowanie programu.

Testując system należy zbadać czy gładko przebiega współdziałanie modułów. Problemy które można tu napotkać to:

- braki w projektowaniu, w szczególności niedostateczna koordynacja komunikacji modułów;
- brak pełnego zrozumienia przez programistę opisu modułów,
- niepełne testy modułów.

Niebezpieczeństwo testów systemu może polegać na tym, że w czasie ich przeprowadzania nie dysponuje się już częścią programistów, którzy brali udział w programowaniu modułów, kiedy było szczytowe zapotrzebowanie na siłę roboczą. Utrudnia to bardzo wyszukiwanie błędów.
Aby uniknąć takiej sytuacji i zmniejszyć ryzyko związane z testem systemu, trzeba prowadzić test krok po kroku. Musi on być tak zorganizowany, żeby można było montować cały system moduł po module. Ten sposób postępowania okazał się w praktyce dobry. Stosunkowo szybko można zlokalizować błędy, gdyż występują one z największym prawdopodobieństwem w najnowszym module. Można je więc w porę rozpoznać i usunąć.
Moduły WE/WY należy zakończyć w pierwszej kolejności. Jako kolejny opracowuje się moduł sterujący, wszystkie inne moduły mogą być opracowywane równolegle. Przy krokowym przeprowadzaniu testu systemu brakujące moduły muszą być albo pominięte albo symulowane.

3.3. Podział programu na moduły

Programiści i kierownicy działów programowania uznają generalnie, że modularyzacja jest praktycznym narzędziem programowania. Korzyści wynikające z rozbicia obszernego programu na kilka mniejszych stanowiących dla siebie całość /modułów/ są tak oczywiste, że nie wymagają żadnych dodatkowych uzasadnień. Jak dotąd niewielu tylko ośrodkom udaje się pomyślnie programować przy pomocy tej techniki, a powstające programy modułowe nie są często lepsze od programów monolitycznych, które zastępują. W rzeczywistości programy modułowe często nie są niczym więcej jak tylko programami nomolitycznymi rozbitymi na arbitralne moduły, z których każdy wykazuje charakterystykę programu monolitycznego jaki zastępuje. Dlaczego tak się dzieje ? Dlaczego modularyzacja jest tak trudna ? Jak rozbić program na moduły, aby uzyskać korzyści, których wszyscy w skrytości ducha oczekujemy ? Przyjęto, że modularyzacja jest prawidłowym podejściem, a jedynym kłopotem jest nieumiejętność przeprowadzania jej. W pewnym sensie słuszne musi być rozbicie dużego programu na mniejsze jednostki, ale jak to zrobić ? Aby odpowiedzieć na to pytanie zreasumujmy wpierw konwencjonalne podejście do modularyzacji, a potem popatrzmy na ten sam problem z odmiennego punktu widzenia. Okazuje się to bardziej obiecującym podejściem i takim, które może mieć większy związek z codziennymi problemami napotykanymi przez programistę. Podany przykład ilustruje to podejście. Program modułowy jest to program o strukturze pokazanej na rys. 15 A, B, C, D i E są to moduły. Ich współzależność nazywamy

hierarchią. A, B, C, D i E tworzą razem hierarchię modułową lub program.

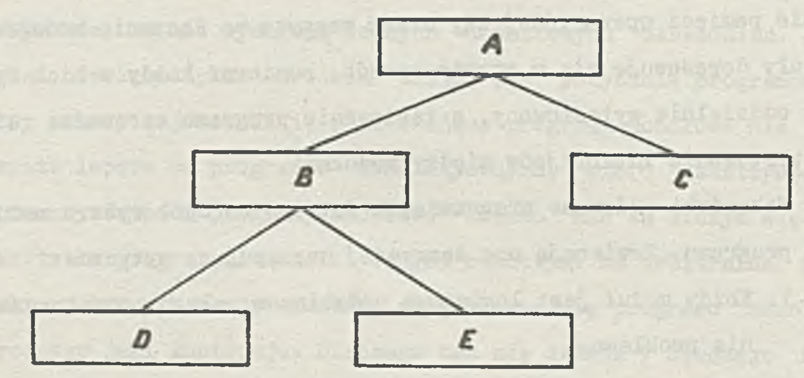
Celem programowania modułowego jest rozbicie programu na taką hierarchię modułów. Moduły są prostsze niż kompletny program, a więc łatwiejsze do zakodowania i wytestowania. I ta zaleta przewyższa wadę programowania modułowego, którą jest zapotrzebowanie pamięci operacyjnej czy czasu maszyna na łączenie modułów. Moduły dopasowuje się w prosty sposób, ponieważ każdy z nich był już oddzielnie wytestowany, a testowanie programu sprowadza się do testowania interfejsów między modułami.

Wskazówki podawane programistom dotyczą na ogół wyboru modułów programu. Zawierają one zazwyczaj następujące wytyczne:

1. Każdy moduł jest logicznym podzbiorem całego przetwarzania problemu.
2. Każdy moduł jest dla siebie całością i nie może być ani za duży ani za mały /powiedzmy jest odpowiednikiem 200 zdań Procedure Division COBOLu/.
3. Każdy moduł jest podprogramem programu.

Bardziej szczegółowe wskazówki mogą dodatkowo zawierać:

4. Każdy moduł powinien mieć jedno wejście i jedno wyjście.
5. Każdy moduł powinien być "czarną skrzynką" /t.j. wyniki jego wykonania powinny zależeć tylko od danych wejściowych a nie od zapamiętanych stanów modułu/.
6. Wszystkie wejścia/wyjścia powinny być ujęte w podprogramach.
7. Powinien istnieć główny logiczny nurt powołujący podprogramy przetwarzania.



RYS. 15. PROGRAM MODUŁOWY

Fakt, że dobrze zaprojektowany lub podzielony na moduły program rozpoznaje się od razu, jest dostatecznym dowodem, że modularyzacja nie powinna być tak trudna jak się wydaje.

Aby się o tym przekonać powróćmy do pierwszych zasad i przyjrzyjmy się programowi, który chcemy rozbić na moduły. Okaże się, że modularyzacja ma miejsce, ale tylko w specyficznym kontekście. Jest to przedstawione poniżej, w części poświęconej projektowaniu programu.

Program pobiera rekordy danych ze zbiorów wejściowych, przetwarza je i wpisuje do zbiorów wyjściowych rekordy zawierające przetworzone dane. Zbiory wejściowe są źródłem dwóch problemów:

1. Nie wszystkie rekordy zbioru mogą być przydatne.
2. Nie wszystkie kombinacje rekordów w zbiorach wejściowych /jeśli jest ich więcej niż jeden/ mogą być przydatne.

A zatem czytanie zbiorów wejściowych rozpada się na czytanie fizycznego zbioru, sprawdzenie czy czytany rekord jest przydatny dla programu oraz ustalenie korzystnych gotowych do przetwarzania kombinacji rekordów z rozmaitych zbiorów wejściowych.

Mówimy o tych funkcjach odpowiednio jako o wejściu/wyjściu. Wyjątek stanowi sprawdzanie i manipulowanie zbiorami.

Przetwarzanie kombinacji rekordów ustalonych przez manipulator zbiorami może wymagać korzystania ze zbiorów pomocniczych /np. bezpośredniego dostępu/ w celu dostarczenia dodatkowych informacji, a te mogą również wymagać działania w warunkach wyjątkowych /np. rekord nie znaleziony/. Przetwarzanie reorganizuje wówczas dane wejściowe i oblicza nowe dane w celu dostarczenia informacji wymaganych dla zbiorów wyjściowych lub raportów.

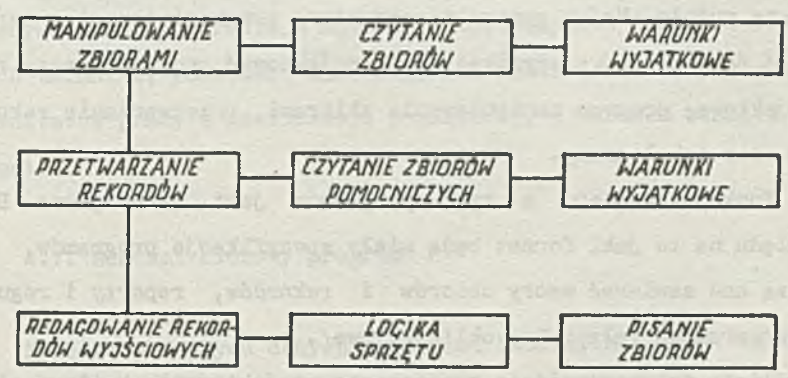
Z wpisywaniem do zbiorów wyjściowych wiążą się nowe problemy, ponieważ dane potrzebne do zbudowania rekordów wyjściowych są rozproszone /mogą pochodzić z jakiegokolwiek rekordu wejściowego lub mogą być wyliczone przez podprogramy przetwarzania/. A więc na kolejność w której zostały napisane rekordy mogła wpłynąć logika uzależniona od hardware'u.

Wszystkie opisane powyżej funkcje programu można przedstawić w postaci schematu struktury programu - jak na rys. 16. Każdy prostokąt schematu przedstawia funkcję programu - nie czyni się tu założeń co do wyboru modułów programu.

W szczególnych przypadkach może okazać się, że program wykonuje dodatkowe funkcje. Ma to miejsce, kiedy w celu optymalizacji dostępu do zbiorów lub wykorzystania pamięci połączy się programy o oddzielnej logice. Przykładem tego może być uaktualnianie zbioru pomocniczego w programie, który korzysta z tego zbioru również dla innego przetwarzania. W takim przypadku poprawki wprowadzane do zbioru pomocniczego byłyby wprowadzane do kolejnego zbioru i oddzielny program przeprowadzałby tę aktualizację. Przypadki takie rzadko mają miejsce, a optymalizacja ma tendencję do zamaskowania podlegających jej funkcji, do głównej struktury można zawsze wprowadzić dodatkowe funkcje.

Aby upewnić się czy program działa, należy go wytestować. Celem zagwarantowania możliwie dokładnego przetestowania należy testować jego poszczególne części. Aby program nadawał się do testowania:

1/ Musi być możliwość symulacji fizycznego wejścia/wyjścia z wyjątkiem wyjścia bezpośrednio na drukarkę wierszową



RYS.16. SCHEMAT STRUKTURY PROGRAMU

/aby uniknąć uzycia aktualnych zbiorów/.

2/ Potrzebna jest pewna jednostka kodu dla włączenia jej w system testujący /aby wytestować oddzielnie każdą funkcję programu/.

Musimy teraz powrócić do programisty, stojącego wobec konieczności sporządzania specyfikacji programu, który ma podzielić na moduły. Mając generalną strukturę programista może teraz podać ją w swojej specyfikacji. Zaprojektować program znaczy zaprojektować program manipulowania zbiorami, przetwarzania rekordów oraz tabulogramy.

Funkcja każdego z tych programów jest teraz jasna. Bez względu na to jaki format będą miały specyfikacje programów, to muszą one zawierać wzory zbiorów i rekordów, raporty i reguły przetwarzania /algorytmy obliczeniowe/.
Umożliwia to programiście rozpoczęcie projektowania każdego elementu programu.

Podzielenie programu na moduły pozwala przekazać je wraz z niezbędną dokumentacją programistom do równoległego wykonania i wytestowania przez co skracamy łączny czas pisania i testowania programów. Moduły składające się na program posiadają różną skalę trudności co umożliwia optymalnie wykorzystać posiadane kwalifikacje członków zespołu programującego.

4. Strukturogramy

Właściwie stosowana inżynieria oprogramowania to nie tylko strukturalne programowanie i planowanie, ale także odpowiednia

dokumentacja programów umożliwiająca innym łatwe i szybkie, a więc efektywne wpracowanie się w program, czy to w przypadku korekty błędów czy aktualizacji oraz dostosowania go do nowych wymogów.

Postacie reprezentacji i sposoby opisywania, z których korzysta się przy rozwoju oprogramowania są to metody pracy nieporównywalne z tego rodzaju metodami stosowanymi w innych dziedzinach techniki, ponieważ nie dają one możliwości dokumentowania rezultatów pracy w graficznej, poglądowej, a zarazem zwartej postaci.

4.1. Schemat blokowy programu

Schemat przepływu danych i schemat blokowy nie mają takiego znaczenia, jakie mają rysunki techniczne w budowie maszyn czy układ połączeń w elektrotechnice, bowiem nie unaoczniają one wyników pracy w tak poglądowej i przejrzystej postaci. W praktyce schemat blokowy często opracowuje się dopiero po zakończeniu pracy nad programem i uważa się go nie jako środek pomocniczy, ale za uciążliwy przepis 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 programu nie jest przejrzysty i zwykle

zajmuje kilka stron papieru przez co utrudnia praktyczne wykorzystanie schematu przy analizie poprawności struktury logicznej programu i jego elementy pokrywają się najczęściej z instrukcjami programowymi języków wyższego rzędu. Programista zamiast kreślić skomplikowane figury geometryczne woli bezpośrednio operować instrukcjami programowymi. Mając gotowy program w postaci ciągu instrukcji nie widzi potrzeby kreślenia adekwatnego schematu blokowego, który musiał by uaktualniać wraz z programem w miarę postępów w testowaniu programu. Dublowanie prac na etapie testowania nie przynosi żadnych efektów, gdyż materiałem do analizy jest zawsze wydruk programu a nie schemat blokowy. Schemat blokowy powstaje dopiero po pełnym wytestowaniu programu na podstawie wydruku instrukcji programu co stanowi dokładne odwrócenie problemu.

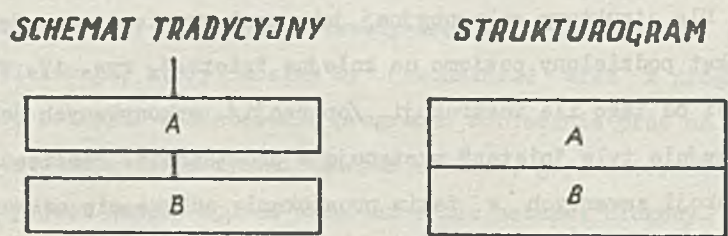
Wraz z wprowadzeniem nowych metod programowania zaczęto nadawać schematowi blokowemu programu duże znaczenie, nie tylko jako materiałowo dokumentującemu wykonaną pracę ale jako narzędziu wykorzystywanemu w procesie pisania i testowania programów. Nowe podejście do tworzenia schematów blokowych programu uwzględnia niedoskonałości dotychczas stosowanej techniki jak również zmiany w technologii tworzenia programu czyli programowanie z góry na dół i podział programu na moduły.

Wcześniej zostało już wykazane, że każdy program można zbudować, korzystając jedynie z trzech podstawowych przebiegów: sekwencji, wyboru i powtarzania. Poprzez odpowiednie zanurzanie w sobie, jak na rys. 3, i łączenie z tych trzech przebiegów można skonstruować dowolny program. Wykorzystując to przyjęto dla

kazdego z przebiegów podstawowych jeden odpowiadający mu znak graficzny. Z tak określonych znaków graficznych można budować schemat blokowy programu. W zależności od przyjętej szczegółowości schematu blokowego można osiągać ogólne schematy blokowe lub wnikając w coraz głębsze poziomy zanurzenia, jak na rys. 3, uszczegóławiać schemat blokowy.

Dla struktury sekwencyjnej jako znak graficzny przyjęto prostokąt podzielony poziomo na kolejne "piętra", rys. 17. 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 poprzez wszystkie pośrednie. Tak określona struktura posiada jedno wejście, którym jest górna krawędź prostokąta, oraz jedno wyjście, którym jest dolna krawędź prostokąta czyli może być modułem indywidualnie programowanym o ile nie ma innych przeciwwskazań.

Dla struktury wyboru określono dwa symbole, rys. 18. Jeden rozważa wybór dwuwartościowy, a drugi wielowartościowy. W górnej części prostokąta /wejście do prostokąta realizowane jest również poprzez górną krawędź/ opisany jest warunek, decydujący o dokonaniu wyboru stanu tak /T/ lub nie /N/ dla wyboru dwuwartościowego lub kolejne możliwe do osiągnięcia stany. W dalszej części prostokąta opisana jest akcja programu dla realizacji konkretnej sytuacji. Wyjście realizowane jest poprzez dolną krawędź prostokąta po zrealizowaniu ciągu instrukcji wybranych po zbadaniu warunku.



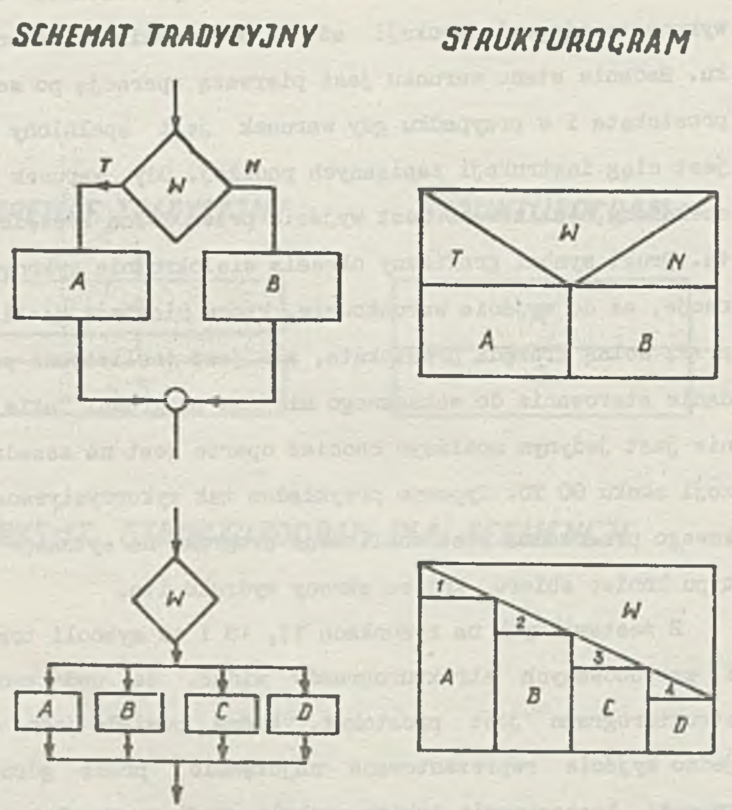
RYS. 17. STRUKTUROGRAM DLA SEKWENCJI

Struktura powtarzania, rys. 19, jest również reprezentowana przez dwa znaki graficzne. Jeden znak przedstawia wielokrotne wykonanie ciągu instrukcji aż do spełnienia określonego warunku. Badanie stanu warunku jest pierwszą operacją po wejściu do prostokąta i w przypadku gdy warunek jest spełniony wykonywany jest ciąg instrukcji zapisanych poniżej. Gdy warunek nie jest spełniony realizowane jest wyjście przez dolną krawędź prostokąta. Drugi symbol graficzny określa wielokrotnie wykonywane operacje, aż do wyjścia warunkowego, które nie jest realizowane poprzez dolną krawędź prostokąta, ale jest realizowane poprzez oddanie sterowania do wskazanego miejsca programu. Takie rozwiązanie jest jedynym możliwym chociaż oparte jest na zasadzie instrukcji skoku GO TO. Typowym przykładem tak wykorzystywanego warunkowego przerwania jest realizacja programu na sytuacje wyjątkowe typu koniec zbioru, koniec strony wydruku itp.

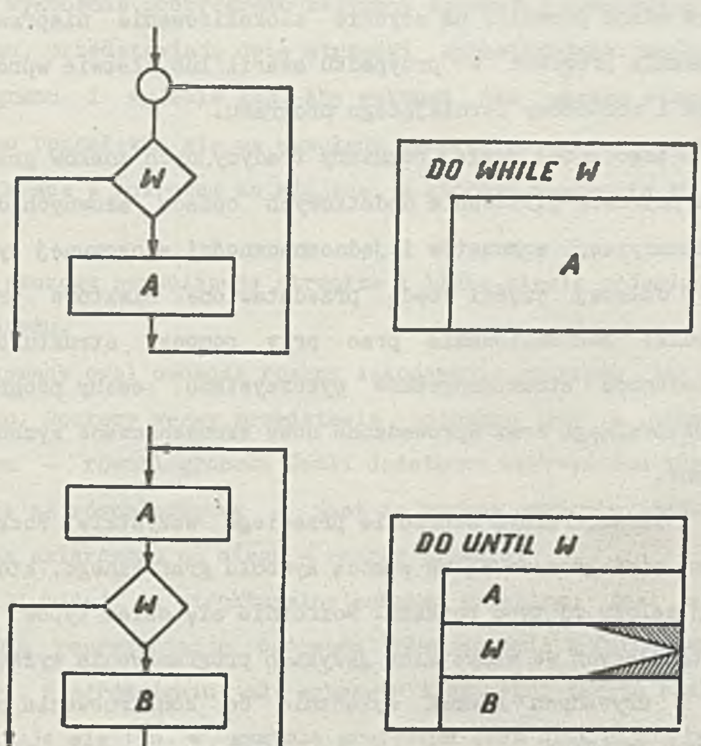
Z zestawionych na rysunkach 17, 18 i 19 symboli tradycyjnych i proponowanych strukturogramów widać, że podstawową figurą strukturogramu jest prostokąt, który posiada jedno wejście i jedno wyjście reprezentowane najczęściej przez górną i dolną krawędź. Zastosowanie takich symboli graficznych ułatwia stosowanie programowania z góry na dół jak również rozbiciu na moduły funkcjonalne, gdyż każdy prostokąt może reprezentować moduł programowy.

4.2. Techniki dokumentowania prac

Schemat blokowy jest elementem dokumentującym pewną pracę



RYS. 18. STRUKTUROGRAM DLA WYBORU



RYS. 19. STRUKTUROGRAMY DLA POWTARZANIA

koncepcyjną jaką jest stworzenie struktury hierarchicznej programu. Schemat blokowy prezentuje sieć działań logicznych w programie i winien stanowić materiał roboczy do kolejnych faz tworzenia programu czyli kodowania i testowania. Po skończeniu testowania programu schemat blokowy stanowi dokumentację programu, która winna pozwolić na szybkie zlokalizowanie nieprawidłowego działania programu w przypadku awarii lub ułatwia wprowadzenie zmian i rozbudowy istniejącego programu.

Biorąc pod uwagę rozmiary tradycyjnych znaków graficznych oraz potrzebę stosowania dodatkowych opisów słownych dążono do miniaturyzacji schematów i jednoznaczności stosowanej symboliki.

W dalszej części będą przedstawione niektóre propozycje techniki dokumentowania prac przy pomocy 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 rozkazu. Rozróżnia się osiem typów rozkazów występujących we wszystkich językach programowania wyższego rzędu i używanych niemal wyłącznie do konstruowania programów w tych językach.

Na rys. 20 podane są te typy rozkazów wraz z odpowiadającymi im symbolami graficznymi i przykładem rozkazu w PL/1. Rozkaz przyporządkowujący wartość zmienionej przedstawiony jest za pomocą koła. Rozkaz decyzyjny ustalający, który rozkaz ma być wykonany jako następny, symbolizowany jest tak jak w dotychczas

znanych schematach blokowych - za pomocą rombu. Rozkaz pętli, nie oznaczany dotychczas w schematach blokowych żadnym symbolem, reprezentują dwa trójkąty znajdujące się jeden nad drugim; górny trójkąt oznacza początek pętli, zaś dolny - jej koniec.

Rozkaz wykonania podprogramu nazywany zazwyczaj wywołaniem podprogramu, przedstawiają dwie strzałki symbolizujące wejście do podprogramu i wyjście zeń. Aby wykonać ten rozkaz sterowanie programu rozgałęzia się na wywołany podprogram, a po jego zakończeniu wraca z powrotem na miejsce, z którego nastąpiło to wywołanie.

Rozkaz skokowy symbolizuje strzałka i linia ciągła prowadząca do jego adresu.

Zakreskowany owal oznacza rozkaz zakończeniu programu lub podprogramu. Rozkazy we/wy przedstawia podobnie jak w schemacie blokowym - równoległobok. Jeśli dodatkowo wprowadzona strzałka wskazuje na równoległobok - jest to rozkaz czytania, jeśli zaś jest ona skierowana od niego - rozkaz pisania.

Z rys. 21 widać, że strukturalny schemat przebiegu jest czysto graficzną reprezentacją podawaną równocześnie z formułowaniem rozkazów. W odróżnieniu od schematu blokowego, tekstu rozkazów nie wpisuje się tu w symbole graficzne lecz obok nich. Bezpośredni związek pomiędzy symbolem graficznym a sformułowaniem słownym powstaje przez zapis w tym samym wierszu.

Do słownego sformułowania rozkazów pokazanego na rys. 21 nie użyto żadnego ze zwykłych języków programowania, ale przeprowadzono je w języku formalnym z gramatycznymi elementami językowymi. Język ten nie ma jednak żadnego specjalnego znaczenia dla

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 ...

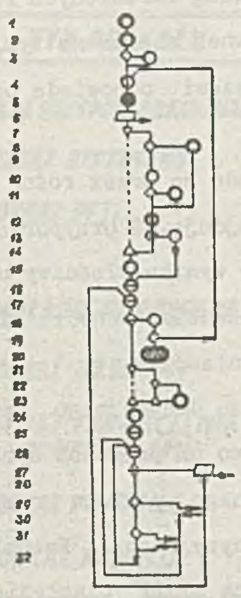
RYS. 20. TYPY ROZKAZÓW WRAZ Z ODPOWIADAJĄCYMI IM SYMBOLAMI GRAFICZNYMI.

dalszych rozwiązań i dlatego też nie objaśniono go szczegółowo. W strukturalnym schemacie przebiegu wyżej opisanych symboli do graficznej interpretacji przebiegu programu nie rozmieszcza się jak w schemacie blokowym dowolnie, ale według określonych reguł. Podstawowa zasada tych reguł polega w głównej mierze na tym, że w graficznej reprezentacji każdemu rozkazowi odpowiada pewien określony poziom rozkazu.

Poszczególne poziomy rozkazów reprezentowane są przez różne linie pionowe, na których te symbole się znajdują. Z przyporządkowania symboli do poszczególnych poziomów wynika właściwy układ strukturalnego schematu przebiegu i w zasadzie dochodzi się do niego stosując się do trzech reguł reprezentacji.

Pierwsza reguła mówi, że wyjście TAK rozkazu decyzyjnego zawsze reprezentuje linia odgałęziająca się poziomo na prawo od symbolu graficznego i prowadząca do podporządkowanego poziomu rozkazu /leżącego na prawo od symbolu rozkazu decyzyjnego/. Natomiast wyjście NIE zawsze przedstawia się za pomocą linii odgałęziającej się pionowo od symbolu i nie prowadzącej do poziomu podporządkowanego. Jeśli na przykład na rys. 21 odpada pierwszy rozkaz decyzyjny z /Mm = 1/ to zostaje wykonany rozkaz przyporządkowujący przedstawiony na poziomie podporządkowanym - w przeciwnym razie nie jest on wykonywany.

Druga reguła określa, że symbole początku i końca pętli na tym samym poziomie i wszystkie rozkazy w obrębie pętli przedstawia się na poziomie podporządkowanym /a więc na prawo stąd/. Trójkąty odpowiadające początkowi i końcowi pętli łączy się przerywaną linią ilustrującą skok powrotny przy wykonywaniu rozkazu



```
1.  SZA = 1.
    SMENDE = SPAR (RSE) - SPAR (RSA)
2.  JEŚLI MM = 1 TO SMENDE = SMENDE + 1
3.  ZDS = 0,
    BAHN = BAHN + 1,
    NAPISZ REKORD 1 (BAHN, KROPKI)
    UTWÓRZ PĘTLE : 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
                          I WYKONAJ 5.
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 PETLE: DZ = 1 DO KH
    8.1 UTWÓRZ PĘTLE: DS = 1 DO 132
        8.1.1  DRUCK (DZ,DS) = ☺
9.  CDS = 1
    A SM = SMA - 8
10. SM = SM + 1
11. JEŚLI SM > SME, NAPISZ REKORD 2 ( DRUCK /DZ,DS/
                    I WYKONAJ 6.
12. JEŚLI MA /1,SM/ = 1, WYKONAJ PODROGRAM KNOTEN
                         WYKONAJ 10.
13. JEŚLI MA /1.SM/ = 0, WYKONAJ PODROGRAM STELM
14. WYKONAJ 10
```

RYS. 21. STRUKTURALNY SCHEMAT PRZEBIEGU Z SEKWENCJĄ ROZKAZÓW

pętli. Na rys. 21 rozkaz pętli po pierwszym rozkazie czytania obejmuje np. wszystkie rozkazy przedstawione pomiędzy dwoma odpowiednimi trójkątami na prawo od przerywanej linii łączącej. Trzecia reguła polega na tym, że po wykonaniu jednego z pozostałych rozkazów nie może nastąpić żadne przejście na podporządkowany poziom rozkazu - z wyjątkiem rozkazu skokowego. W pokazanym przykładzie reguła ta odnosi się również do rozkazu skokowego, jednakże przy ograniczonym stosowaniu na ogół nie daje się ona przestrzegać. Jak widać rozkaz skokowy zajmuje w strukturalnym schemacie przebiegu szczególne stanowisko, gdyż nie ma dla niego żadnych ograniczeń ze względu na poziom następnego rozkazu. Na rys. 21 dokonano pewnych modyfikacji symboli podanych na rys. 20. Modyfikacje te przyczyniają się do zwiększenia graficznej treści informacji w strukturalnym schemacie przebiegu. Tak więc zamiast kółka podstawowego symbolu rozkazu przyporządkowującego wprowadza się równie duży punkt, kiedy zmiennej wynikowej /zmienna zawierająca wynik z wykonanego podprogramu/ przydzielona zostaje pewna wartość. Dzięki temu, w podprogramie od razu widoczne są tylko te nieliczne miejsca, w których zmiennym wynikowym nadana została pewna wartość.

Z kolei modyfikuje się rozkaz przyporządkowujący i rozkaz decyzyjny w przypadkach, gdy powołują się one na jeden licznik. Przez jeden licznik rozumie się tu zmienną, za pomocą której w czasie przebiegu podprogramu wyliczona zostaje pewna określona wielkość. Ta modyfikacja rozkazu przyporządkowującego i rozkazu decyzyjnego mająca na celu użycie jednego licznika - to wprowadzenie pionowej kreski. W przykładzie podanym na rys. 21 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 jeden indeks. Przez jeden indeks rozumie się tu zmienną służącą do wskazania innej wskazywanej zmiennej.

Na rys. 21 zmienne ZM i SM są wskaźnikami użytymi do oznaczenia wiersza i kolumny macierzy. Toteż w pierwszym rzędzie odpowiednio modyfikowane są symbole rozkazów w wierszach 16, 17, 18 oraz 25, 26 i 27, powołujące się na te zmienne.

Za pomocą ukośnej kreski modyfikuje się rozkaz przyporządkowujący oraz rozkaz decyzyjny, przede wszystkim wtedy, gdy powołują się one na jedną cechę /zmienna Boola/. Przez cechę rozumie się tu pewną zmienną, która może przyjąć tylko dwie wartości i służy jedynie do zbadania czy jakiś określony symbol jest prawdą czy też nie. Tę ukośną kreskę pochyla się w lewo lub w prawo w zależności od tego, która z tych dwóch wartości zostaje przyporządkowana lub zapytana. Na rys. 21 nie zachodzi taki przypadek.

Poza tymi zmodyfikowanymi symbolami określone zmienne przebiegu programu można wykazywać za pomocą różnych barw, co ułatwia jeszcze bardziej orientację w strukturalnym schemacie przebiegu. Przyczyny drukarsko-techniczne nie pozwoliły na pokazanie tego efektu na rysunku.

Zmiany symboli przeprowadzone zgodnie z uprzednio skomentowanymi aspektami nie są głównym elementem strukturalnego schematu przebiegu, ale sprawdziły się one w praktyce i okazały się trafne.

W innych zastosowaniach modyfikacje były możliwe według całkiem odmiennych punktów widzenia.

Zrobione w związku z tym uwagi mają jedynie podkreślić, że pomysłowa modyfikacja symboli w strukturalnym schemacie przebiegu może zwiększyć przejrzystość graficznej reprezentacji. Czysto graficzną interpretacją podprogramów, jaką jest strukturalny schemat przebiegu, czyta się niezależnie od podanego obok niej tekstu rozkazów. W połączeniu z tym tekstem pozwala ona na bardzo zwartą reprezentację przebiegu programu oraz na znacznie szybszą orientację w nim - aniżeli zwykły schemat blokowy, w którym informacje graficzne pomieszane są z informacjami alfanumerycznymi. Ze względu na swoje rozprzestrzenienie wzajemna zależność tekstu rozkazów i graficznych symboli zabiera dużo wolnej powierzchni. Równoległa reprezentacja strukturalnego schematu przebiega i słowne sformułowanie rozkazów zapewniają bezpośrednie przejście z liniowej /jednowymiarowej/ reprezentacji przebiegu programu. Dwuwymiarowa reprezentacja jako wylistowanie rozkazów jest lepiej dostosowana do możliwości spostrzegania ludzkiego oka.

Często ze względu na potrzebną powierzchnię schemat blokowy dzieli się na reprezentacje cząstkowe i za pomocą tzw. łączników tworzy się znowu powiązania pomiędzy reprezentacjami cząstkowymi; ale graficzna interpretacja przebiegu programu redukuje częściowo te łączniki. Przepada przy tym możliwość zapisania w postaci graficznej całego przebiegu programu. Dalsza istotna zaleta strukturalnego schematu przebiegu, w porównaniu ze schematem blokowym, polega na możliwości graficznej reprezentacji rozkazu

pętli.

Główna zaleta strukturalnego schematu przebiegu polega na tym, że w graficznie poglądowej postaci można oddać nadrzędność względnie podrzędność rozkazów, a więc ich wzajemną zależność. W tradycyjnym schemacie blokowym nie jest to możliwe ze względu na brak miejsca i odpowiedniego symbolu graficznego. W schemacie blokowym zarówno nadrzędność jak i podrzędność rozkazów 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:

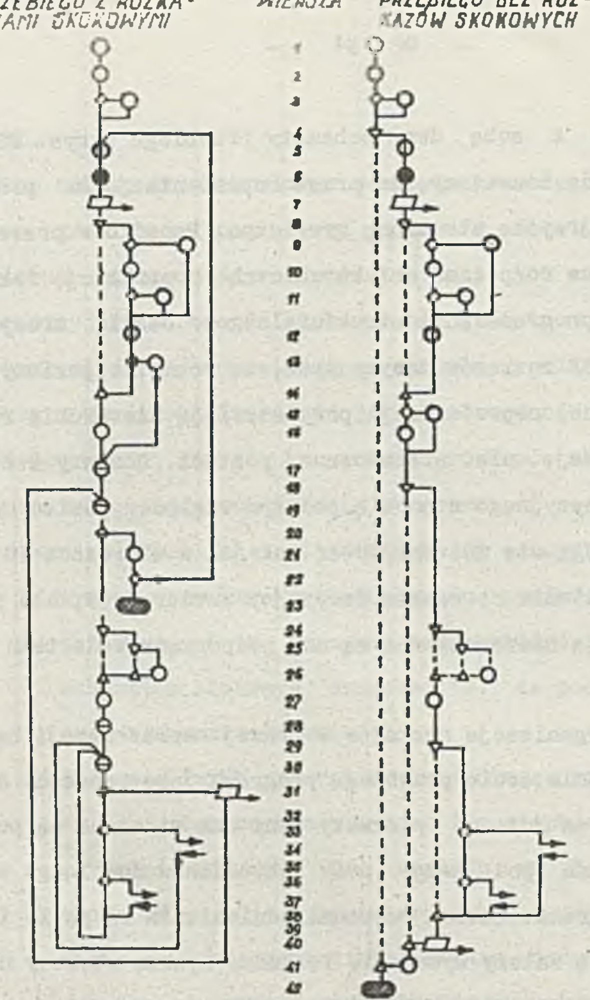
1/ można go łatwo rysować bez szablonów; w porównaniu ze schematem blokowym oznacza to, że poszczególne symbole mogą być znacznie mniejsze,

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, podczas gdy w schemacie blokowym nie zawsze tak jest.

W strukturalnym schemacie przebiegu rozkaz skokowy ma odrębne stanowisko, gdyż w przeciwieństwie do innych rozkazów nie ma dla niego żadnych ograniczeń ze względu na poziom następnego rozkazu. Poniżej przy pomocy przykładu objaśniono bliżej konsekwencje wynikające dla strukturalnego schematu przebiegu z unikania rozkazów skokowych.

Gdy porównujemy z sobą dwa schematy przebiegu z rys. 22 to w pierwszym rzędzie zauważamy, że prawa reprezentacja ma prostszą i bardziej przejrzystą strukturę graficzną. Ponadto w prawej reprezentacji można rozpoznać hierarchiczną organizację rozkazów tak typową dla programowania strukturalnego. Jeśli zrezygnuje się z części ELSE rozkazów decyzyjnych, to rozmaite poziomy rozkazów w graficznej reprezentacji przedstawiają hierarchię rozkazów w poglądowej, nie sfałszowanej postaci. Rozkazy w części ELSE rozkazu decyzyjnego stanowią pod tym względem pewien wyjątek, ponieważ mogą się one znajdować tak jak w wierszach 10 i 11 na tym samym poziomie co rozkaz decyzyjny w wierszu 9, ale zgodnie z organizacją hierarchiczną są one podporządkowane temu rozkazowi.
Hierarchiczna organizacja rozkazów w prawej reprezentacji bardzo ułatwia wyobrażenie sobie przebiegu programu jako procesu dynamicznego, ponieważ już na pierwszy rzut oka widoczne są 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 skokowy narusza tę prostą i jasną hierarchiczną organizację rozkazów - jak to można zobaczyć w lewej reprezentacji. 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 w przypadku wewnętrznego projektu programu.

Podejście strukturalne do rozwiązywanych problemów znalazło



○ ROZKAZ PRZYPORZĄDKOWUJĄCY DLA ZMIENNEJ WYNIKOWEJ

⦶ ROZKAZ PRZYPORZĄDKOWUJĄCY DLA LICZNIKA

◇ ROZKAZ DECYZYJNY DLA LICZNIKA

⊖ ROZKAZ PRZYPORZĄDKOWUJĄCY DLA INDEKSU

◇ ROZKAZ DECYZYJNY DLA INDEKSU

⊘⊘ ROZKAZ PRZYPORZĄDKOWUJĄCY DLA SYMBOLU (ZMIENNA BOOLA)

◇◇ ROZKAZ DECYZYJNY DLA SYMBOLU (ZMIENNA BOOLA)

RYS.22. PORÓWNANIE STRUKTURALNYCH SCHEMATÓW PRZEBIEGU DWÓCH EKWIWALENTNYCH SEKWENCJI ROZKAZÓW

zastosowanie nie tylko w programowaniu ale również w projektowaniu systemów informatycznych. Techniki dokumentowania projektowanego systemu oparte są na hierarchicznej strukturze systemu i 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/.

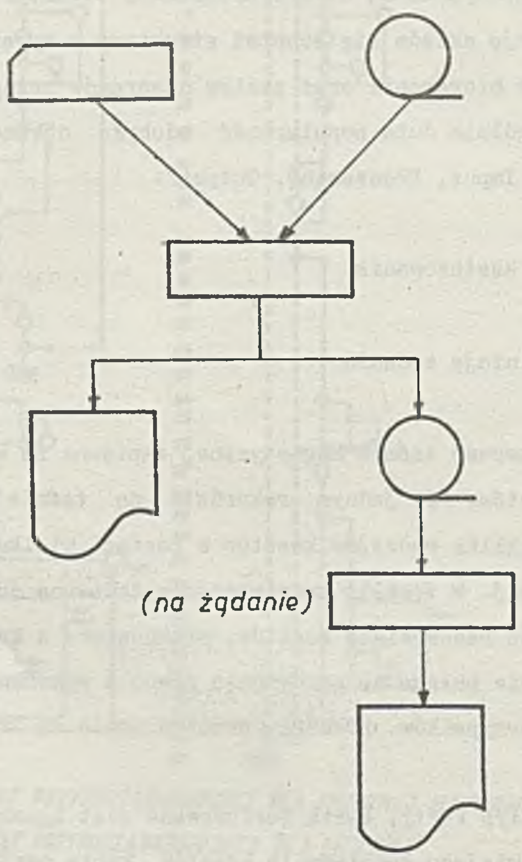
5. Przykład zastosowania

5.1. Definicja zadania

Na podstawowej taśmie magnetycznej zapisane są miejsca powstawania kosztów. W jednym rekordzie na taśmie magnetycznej znajduje się kilka rodzajów kosztów w postaci wielkości planowanej i wykonanej. W trakcie przetwarzania dodawana jest akumulacyjnie wartość rzeczywista kosztów, przenoszona z kart perforowanych. Zadanie polega na porównaniu planu z wykonaniem. Dla następujących przypadków podczas przetwarzania są drukowane meldunki:

- błędny typ karty, karta perforowana jest ignorowana,
- błędne miejsce powstawania kosztów, karta perforowana jest ignorowana,
- błędny rodzaj kosztów, karta perforowana jest ignorowana,
- przekroczenie kosztów planowanych,
- koniec taśmy magnetycznej, wydruk pozostałych kart perforowanych.

5.2. Schemat przetwarzania



5.3. Opis zbiorów

Zbiór kart perforowanych postaci:

Kolumna	Zawartość
1 - 2	Typ karty, zawsze = 11
3 - 8	Miejsce powstawania kosztów
9 - 13	Rodzaj kosztów
14 - 19	Wartość rzeczywista kosztów

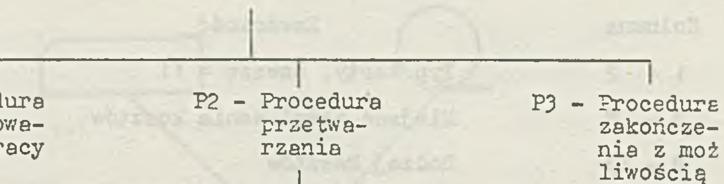
- Zbiór kart jest uporządkowany według miejsca powstawania kosztów
- Na jedno miejsce powstawania kosztów może być kilka kart perforowanych
- Karta z nowym miejscem powstawania kosztów ma być traktowana jako błędna.

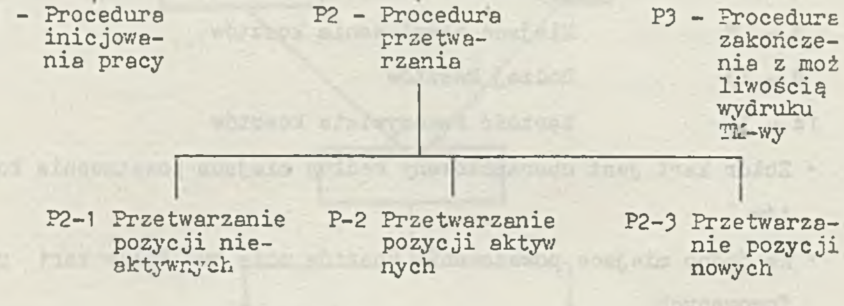
Konstrukcja zbioru we/wy na taśmie magnetycznej

Pole - zawartość	Rozmiar i typ
1. Miejsce powstawania kosztów	X/6/
2. Miesiąc	99
3. Rodzaj kosztów	9/5/
4. Plan na miesiąc	9/5/
5. Wykonanie w miesiącu	9/5/

6-14 jak pola 3-4 dla drugiego, trzeciego i czwartego rodzaju kosztów.

5.4. Schemat blokowy - strukturogram





W dalszej części podane zostaną strukturogramy dla poszczególnych procedur. Dla zwięzłości zapisu zastosowano oznaczenia:

KK - miejsce powstawania kosztów na karcie perforowanej

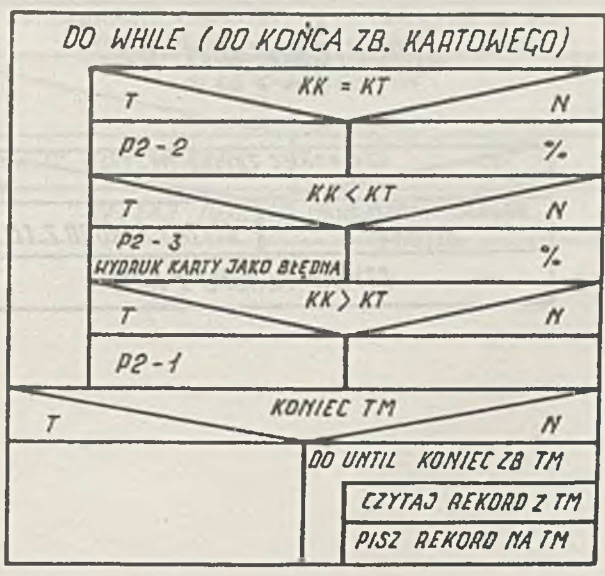
KT - miejsce powstawania kosztów na taśmie magnetycznej

% - oznacza brak operacji

STRUKTUROGRAM DLA P1

USTAWIENIE WARTOŚCI POCZĄTKOWYCH
OTWARCIE ZBIORÓW
WYDRUK NAGŁÓWKA NA DRUKARCE
CZYTAJ 1 REKORD Z TM
CZYTAJ 1 REKORD Z KART

STRUKTUROGRAM DLA P2



STRUKTUROGRAM P3

ZAMKNIĘCIE ZBIORÓW	
CZY ŻĄDANY WYDRUK TM-WY	
T	N
WYDRUK TM	%

STRUKTUROGRAM P2-1

CZY KONIEC ZBIORU TM	
T	N
DRUKUJ KARTĘ JAKO BŁĘDNĄ	CZYTAJ REKORD Z TM
CZYTAJ REKORD Z KART	

STRUKTUROGRAM P2-2

DO WHILE (DOPÓKI KK = KT I WYSTĘPUJĄ KARTY)

DO UNTIL (DLA WSZYSTKICH RODZAJÓW KOSZTÓW)

RODZAJ KOSZTÓW NA KARCIE = NATM

T	N
KUMULOWANIE WARTOŚCI WYKONANIA	WYDRUK KARTY JAKO BŁĘDNE

PLAN < WYKONANIE

T	N
WYDRUK KOMUNIKATU	%

CZYTAJ REKORD Z KART

L I T E R A T U R A

1. Baker F., Chief Programmer Team Management of Production Programming, IBM Systems Journal, Volume II No 1, 1972
2. Dahl O., Dijkstra E., Hoare C., Structured Programming, Academie Press, London and New York, 1972
3. Dijkstra E., Umiejętność programowania, WNT Warszawa 1978
4. Maynard J., Modular programming, London, 1972
5. Modular programming techniques T.P. 5091, publikacja techniczna ICL, 1971
6. Turski W., Metodologia programowania, WNT Warszawa 1978
7. Wirth N., Wstęp do programowania systematycznego, WNT Warszawa 1978