ZJEDNOCZENIE INFORMATYKI
OŚRODEK BADAWCZO-ROZWOJOWY INFORMATYKI

PROGRAMOWANIE STRUKTURALNE

Europejski Program Badawczy Diebolda

86

Warszawa 1977

ZJEDNOCZENIE INFORMATYKI
OŚRODEK BADAWCZO-ROZWOJOWY INFORMATYKI

PROGRAMOWANIE STRUKTURALNE

Europejski Program Badawczy Diebolda

Wyłącznie do użytku na terenie PRL

86

Warszawa 1977

Tytuł oryginału: Structured programming
Document No E 135
May 1976

Tłumaczenie: Danuta Żelawska

Redakcja: Adam Rzymowski

Komitet Redakcyjny

Andrzej Idźkiewicz, Janina Jerzykowska /sekretarz/, Stanisław Nelken, Witold Staniszkis, Ryszard Terebus /przewodniczący/

Wydawca

OBRI-Dział Wydawnictw, 02-021 Warszawa, ul.Grójecka 17

Warszawa 1977. Nakład: 900 + 98 egz. Objętość: ark. wyd. 4; ark. druk. 9,7. Format A4. Papier offsetowy kl.III, 80g, 61x86

Zam.89/77 DN.444-13/73 Cena zł 92.-

SPIS TREŚCI

STRESZCZENIE 5

I. WSTĘP 9

II. METODY I KRYTERIA PROJEKTOWANIA PROGRAMU 11

A. Struktura hierarchiczna 11

B. Abstrakcja 17

C. Metodologia projektowania 20

D. Zasada jasności 34

E. Postępowanie w przypadku pojawiania się błędów . 35

III. POJĘCIA SKŁADNI WE WDRAŻANIU MODUŁÓW 39

A. Modułowe zastosowania w ALGOLu 39

B. Modułowe zastosowania w PL/1 46

C. Modułowe zastosowania w FORTRANie 50

D. Modułowe zastosowania w COBOLu 51

E. Nadawanie struktury danym 53

F. Struktury danych w modularnych programach PL/1 . 54

G. Struktury danych w modularnych programach ALGOLu 68 56

H. Kodowanie strukturalne 57

IV. OPROGRAMOWANIE WSPIERAJĄCE PROGRAMOWANIE STRUKTURALNE 62

V. TECHNIKI DOKUMENTACYJNE 64

A. Wykresy Nassi-Schneidermanna 64

B. Wykresy Hipo 67

VI. PROBLEMY WIODĄCEGO ZESPOŁU PROGRAMOWANIA 71

VII. WPROWADZENIE PROGRAMOWANIA STRUKTURALNEGO 77

STRESZCZENIE

Dyskusja na temat programowania strukturalnego nie pozwoliła jeszcze wypracować ostatecznej definicji tego pojęcia. Różne jego rozumienie i interpretacja nie wykluczają jednak oech wspólnych kilku podstawowych koncepcji. Są to: hierarchiczna struktura systemów programowych, operacje myślowe: abstrakcji i dekompozycji oraz stosowanie znormalizowanych bloków strukturalnych dla uproszczenia prezentacji cyklu programu.

W sprawozdaniu dokonano przeglądu tych koncepcji jak następuje:

Struktura hierarchiczna

Koncepcja ta doprowadziła w programowaniu do pojęcia modułowości i dzielenia złożonego zadania opracowania systemu programowego na proste zadania częściowe.

Abstrakcja i dekompozycja

Abstrakcja jest operacją myślową precyzowania złożonych systemów, opierającą się na pospolitej zdolności umysłu ludzkiego odnajdywania podobieństw /wg jakiegoś kryterium/ w zbiorze różnorodnych elementów i budowania z nich /wg tych podobieństw/ większych całości przy jednoczesnym ignorowaniu cech różnicujących te elementy, dzięki czemu w myśleniu ludzkim nie dochodzi do "przepełniania" pamięci.

Metodologia decyzji projektowej

Dwie intelektualne metody: abstrakcji i dekompozycji odpowiadają dwóm metodom projektowania "od góry-w dół" i "od dołu-w górę". Zwolennicy programowania strukturalnego podkreślają za-

lety projektowania "od góry-w dół" w powiązaniu z metodą precyzowania "krok-po-kroku".

Projektowanie systemu programowego "od góry-w dół" rozpoczyna się od sformułowania modelu użytkownika i jego wyobrażeniu o opracowywanym programie, drogą odpowiedzi na następujące główne pytania:

. Co program ma realizować?

. Jakiego wymaga wejścia?

. Jakie ma mieć wyjścia?

. Jakiej obsługi wymaga?

. Jak się ma zachować w przypadku błędów i awarii?

Technika projektowania "od góry-w dół" jest skuteczną metodą rozwiązywania wielu problemów projektowania, chociaż nie może być stosowana powszechnie bez ryzyka określonych trudności.

Postępowanie w przypadku błędów

Programowanie strukturalne wymusza podział wiedzy i krępuje działania żądając znajomości kolejnych poziomów abstrakcji. Wymaganiom tym można sprostać podejmując w odpowiednim momencie środki ostrożności w planowaniu i wykonywaniu systemu programowego.

Koncepcje stosowania modułów

Istniejące w językach programowych koncepcje modułowości stanowią najistotniejszą problematykę analizy możliwości lingwistycznych w odniesieniu do programowania strukturalnego.

Koncepcje tworzenia struktur danych

Dla zapewnienia modułowości systemów programowych na poziomie lingwistycznym muszą istnieć nie tylko środki tworzenia struktur tekstów programów, lecz również i środki nadawania danym struktur używanych zgodnie z modularną strukturą.

Oprogramowanie wspierające programowanie strukturalne

Dyskusja na temat programowania strukturalnego wykazała, że klasyczne zasady inżynierii programowania są pomocne przy tworzeniu doskonalszych form oprogramowania.

Techniki dokumentacyjne

Jakość systemu programowego jest zasadniczo określona jakością jego projektu. W programowaniu konwencjonalnym standardową metodą projektowania jest schemat blokowy. Jednak w praktyce, przy stosowaniu schematów blokowych pojawiają się z reguły problemy spowodowane trudnościami rozumienia schematów.

Wykresy Nassi - Schneidermanna

Stosując się do zasad programowania strukturalnego, Nassi i Schneidermann opracowali technikę projektowania i reprezentacji programów, która dając możność zaprojektowania programu uczy także opracowania rozwiązania funkcjonalnego złożonego z małych, znormalizowanych części, drogą:

- ustawiania bloków strukturalnych w szeregi,
- umieszczania ich obok siebie jako wyróżnienie przypadku,
- tworzenia gniazd, przez umieszczanie jednego bloku w drugim.

Wykresy Hipo

Wykresy Hipo reprezentują cykl systemu programowego jako hierarchicznie uporządkowany system faz: wejścia, przetwarzania i wyjścia. Mogą one być używane do projektowania, kodowania i aktualizacji systemu programowego.

Wiodący zespół opracowania programu

Wiodący zespół opracowania programu składa się z trzech osób: głównego programisty, programisty wspierającego /odpowied-

nik starszego programisty/ i sekretarki współdziałającej przy opracowaniu programu od etapu projektowania do ostatecznego testowania. Zespół ten jest okresowo zasilany, w zależności od opracowywanego zagadnienia, przez odpowiednich programistów i specjalistów.

Praktyczne doświadczenia wykazały, że na rozwiązanie czeka jeszcze wiele problemów psychologicznych i organizacyjnych.

I. WSTĘP

Problemy oprogramowania komputerów są tak stare jak same komputery. Regularnie też co jakiś czas pojawiają się metody roszczące sobie prawo do rozwiązania tych problemów.

Czy programowanie strukturalne /Structured Programming/ jest ostatecznym rozwiązaniem wszystkich trudności programowania można będzie zawyrokować dopiero po wielu latach jego stosowania, tymczasem zaś w sprawie tej przez przemysł przetacza się fala gwałtownej argumentacji za i przeciw. Wciąż jeszcze mało jest uznanych zalet programowania strukturalnego i z pewnością za mało czasu poświęcono analizie tych metod i technik, które zakończyły się sukcesem lub niepowodzeniem projektów rozwijania oprogramowania. Dlatego nie dziwi, że jedynym bezspornym faktem, jakim operuje się w dyskusjach nad domniemanymi zaletami i wadami programowania strukturalnego jest to, że stanowi ono "pierwszą metodę ogólnego zastosowania i znormalizowanego sporządzania schematów systemów programowych i ich wdrażania".

Ostatnio dużo mówiono o kryzysie w oprogramowaniu[x/]. Dyskusje na ten temat ujawniły, że mówić o trudnościach oprogramowania jest łatwo, natomiast ustalić podstawowe tego przyczyny dużo trudniej. Nasza znajomość rzeczy jest niekompletna na skutek małej liczby danych o kosztach rozwijania oprogramowania w przeszłości /co umożliwiałoby dokonywanie oceny kosztów przyszłego rozwoju oprogramowania drogą ekstrapolacji/ oraz braku wypróbowanych metod tworzenia dobrego oprogramowania. Dlatego w sprawozdaniu niniejszym możemy dać tylko zarys problematyki programowania strukturalnego oraz służyć taktyczną radą przy prze-

---

x/ Patrz: DRP-E Conference Proceedings for Munich, marzec 1976, sesja K.

chodzeniu od programowania konwencjonalnego do programowania strukturalnego; nie jesteśmy natomiast w stanie sformułować żadnych ostatecznych wniosków co do jego użytkowania.

Czytając niektóre pełne euforii artykuły w specjalistycznych czasopismach można było odnieść wrażenie, że wprowadzenie i stosowanie programowania strukturalnego nie nastręcza w ogóle żadnych problemów, a szablonowe przykłady wprowadzające przerabiane na kursach i seminariach mogą odwracać uwagę od rzeczywistych zagadnień tematu. Z tych powodów pewne części sprawozdania ujmują sprawę bardziej szczegółowo niż mogliby się tego spodziewać zainteresowani tematem kierownicy-organizatorzy. Stoimy bowiem na stanowisku, że złożoność zagadnienia wymaga wnikliwej oceny i ostrożności, zwłaszcza gdy waży się sprawa decyzji wprowadzenia programowania strukturalnego.

II. METODY I KRYTERIA PROJEKTOWANIA PROGRAMU

Dyskusja nad programowaniem strukturalnym ciągle jeszcze jest odległa od ustalenia ostatecznej jego definicji. Różne interpretacje tego pojęcia wykazują jednak pewne wspólne elementy. Do nich należą: hierarchiczna struktura systemów programowych, sposób logicznego abstrahowania i dekompozycji /rozkładu/ oraz stosowanie standardowych bloków strukturalnych w celu uproszczenia procesu sterowania. Dwa pierwsze, dla lepszego zrozumienia zagadnienia, pokrótce omówimy.

A. STRUKTURA HIERARCHICZNA

Podstawową przyczyną trudności oprogramowania był brak metod umożliwiających budowanie złożonych systemów programowych w sposób bezpośrednio zrozumiały dla człowieka. Stwierdzono, że po przekroczeniu pewnej wielkości systemów programowych następuje zjawisko zmiany charakteru trudności ich rozumienia: przeobrażenie się trudności natury ilościowej w trudności natury jakościowej. Obecnie jednym z najbardziej skutecznych sposobów uczynienia systemów programowych zrozumiałymi dla człowieka jest nadawanie im tzw. struktury hierarchicznej.

Znalazłszy się na gruncie dyscyplin inżynierskich wobec zadań budowania jakiegoś złożonego systemu, często stosuje się starą zasadę "dziel i rządź". Mianowicie dokonuje się podziału systemu na indywidualne składniki oraz podaje określony sposób przepływu informacji i sterowania, decydujący o ich wzajemnym oddziaływaniu w ramach systemu. Technika ta doprowadziła do tzw. modułowości w programowaniu, tzn. dzielenia całego zadania zbudowania programu na zadania częściowe w postaci względnie niezależnych członów i dokładnego określenia tzw. punktów interakcji, poprzez które ma być realizowana wymiana danych między tymi członami.

Technika modułowa daje dwie poważne korzyści:

- podział systemu na kilka składników umożliwia ich niezależne wdrażanie i testowanie. To zaś umożliwia równoległe prowadzenie prac nad systemem, co stanowi podstawowy wymóg przy opracowywaniu dużych systemów programowych w rozsądnym czasie,
- poszczególne funkcje systemu programowego są wykonywane przez pojedyncze moduły. Tworzą się więc wydzielone obszary, dogodne w przypadku wprowadzania jakichś zmian i tylko jeden raz wdrażane.

Niewątpliwie tego rodzaju zbiór "nieuporządkowanych modułów" niesie określone ryzyko uzyskania pożądanej jakości całości systemu programowego. Na przykład:

- moduł może wykonywać zbyt wiele funkcji, na skutek czego jego logika staje się trudna do zrozumienia,
- wspólne części funkcji poszczególnych modułów mogą być zbyt późno rozpoznane i skutkiem tego wdrażane wiele razy,
- moduły mogą oddziaływać, w niezamierzony sposób, na wspólne pola danych.

Złożoność powiązań między poszczególnymi modułami jest czynnikiem krytycznym dla zrozumiałości dużych systemów programowych. Powiązania mogą występować w postaci przepływu sterowania, użycia wspólnych pól danych i konwencji przypisanych modułom. Złożoność powiązań między modułami może przedstawiać [illegible] osiągnięcia następujących podstawowych celów:

- łatwego sprawdzenia dotychczas opracowanych programów,
- możliwie największej niezależności poszczególnych składników programu,
- ograniczenia do minimum samych powiązań między modułami, a to z myślą o uchronieniu się przed niewygodami komunikacji między personelem opracowującym system a personelem obsługującym system.

Dotychczas brak jest kryterium oceny jakości struktury systemów programowych. Ostatnie wyniki badań sugerują, że próba stworzenia takiego kryterium dla istniejących programów jest całkowicie nierealna. Pewne jest jednak, że duża rozbieżność charakteru modułów jest równoznaczna z niewłaściwą strukturą programu. Jednym z najbardziej skutecznych sposobów tworzenia właściwej struktury systemu programowego jest uporządkowanie modułów wg ich specyfiki i wzajemnych, hierarchicznych powiązań.

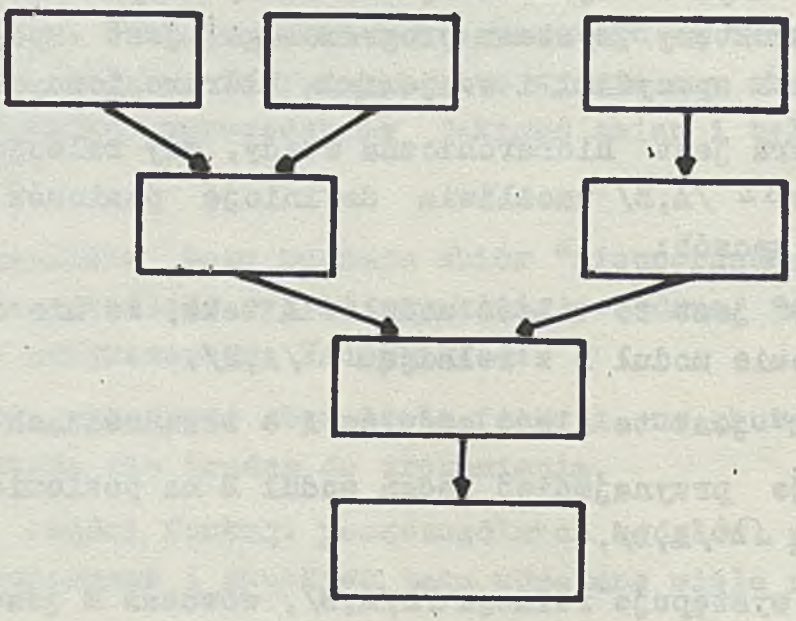
Struktura jest hierarchiczna wtedy, gdy relacja używająca pary modułów Ω /A,B/ umożliwia definicję poziomów programu w następujący sposób:

1. Poziom "0" jest to ilość modułów A taka, że nie istnieje na tym poziomie moduł B z relacją Ω /A,B/.
2. Poziom "i" jest to ilość modułów A o własnościach takich, że:
   - istnieje przynajmniej jeden moduł B na poziomie /i - 1/ z relacją Ω /A,B/,
   - jeżeli występuje relacja Ω/A,B/, wówczas B jest na poziomie /i - 1/ lub niżej.

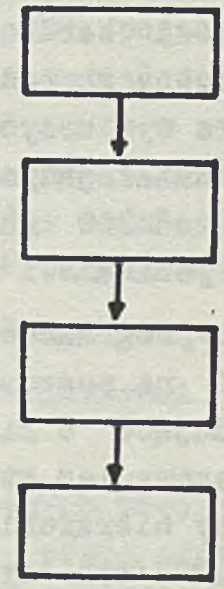
Jednak hierarchiczna struktura relacji modułowych oparta o podaną definicję stanowi bardzo mglistą teorię. W praktyce trudno jest dokonywać oceny na podstawie tej definicji, ponieważ nie da się całkowicie wykluczyć cykli relacji modułowych w znaczeniu procesów współdziałających. Należy starać się sprowadzić do minimum ilość modułów należących do cyklu relacji i umieścić je na wspólnym poziomie.

Dla każdego systemu programowego można znaleźć dekompozycję jego ogólnej funkcji na poszczególne moduły i relacje między tymi modułami, decydujące o hierarchicznej strukturze systemu. W związku z programowaniem strukturalnym często wymienia się dwa rodzaje struktury hierarchicznej: tzw. strukturę "drzewa" i strukturę "liniową" /zob. rys.1 i 2/.

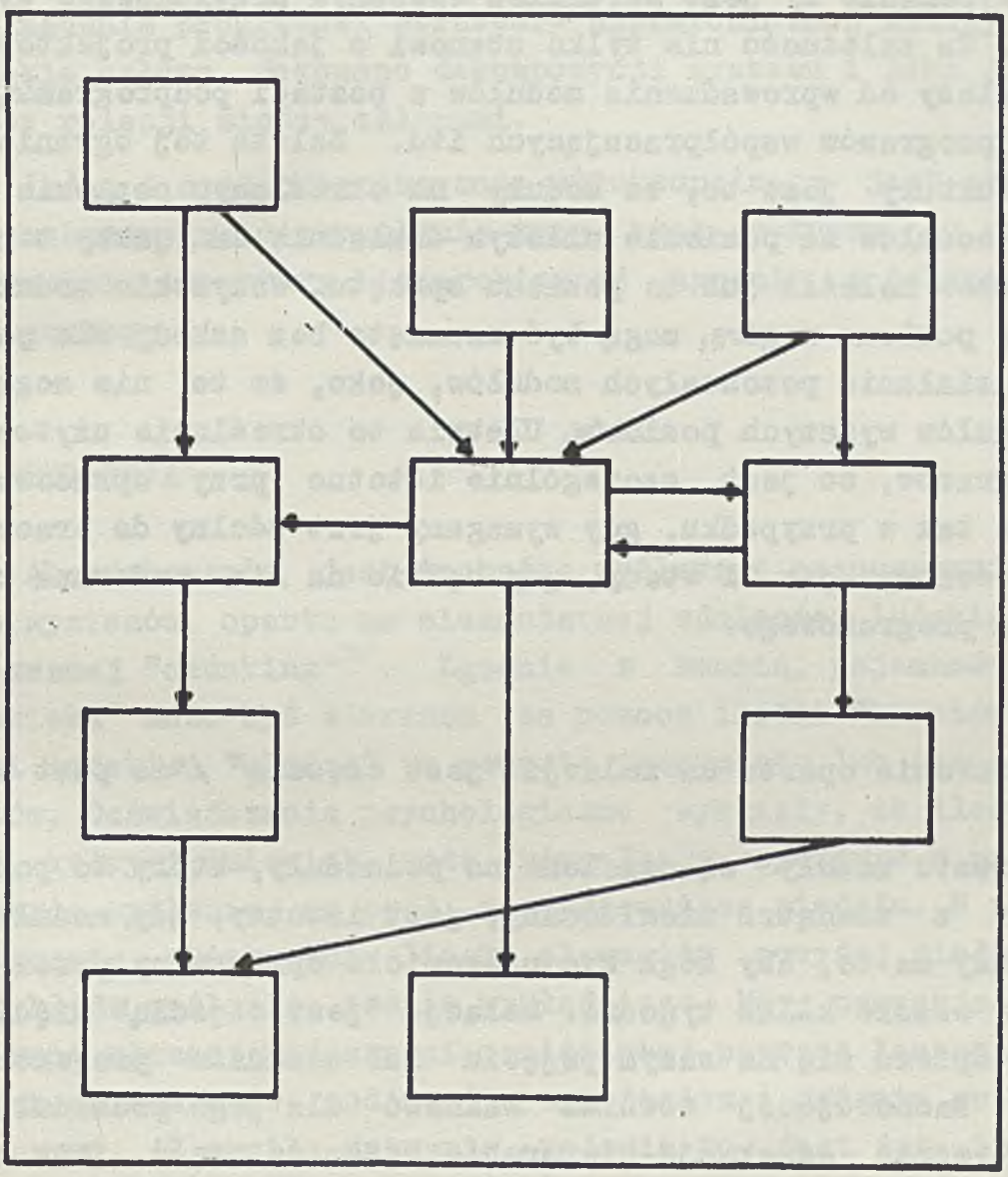
STRUKTURA DRZEWA



STRUKTURA LINIOWA



Rys. 1 PRZYKŁADY HIERARCHICZNYCH STRUKTUR PROGRAMÓW



Rys. 2 STRUKTURA HIERARCHICZNA
Z CYKLEM RELACJI

1. Hierarchia oparta na relacji "użyty" /"used"/

Między modułem A/ i modułem B/ występuje relacja "użyty" dokładnie wtedy, kiedy moduł A/ woła moduł B/ i kiedy prawidłowe wykonanie B/ jest warunkiem wstępnym prawidłowego wykonania A/. Ta zależność nie tylko stanowi o jakości projektowania, lecz zależy od wprowadzenia modułów w postaci podprogramów, procedur, programów współpracujących itd. Zaletą tej ograniczającej struktury jest to, że moduły na określonym poziomie mogą używać modułów na poziomie niższym dokładnie tak, jakby ten niższy poziom należał już do poziomu sprzętu. Wszystkie moduły, od pewnego poziomu w górę, mogą być usunięte bez szkody dla poprawności działania pozostałych modułów, jako, że te nie mogą używać modułów wyższych poziomów. Ułatwia to określenie użytecznych podprogramów, co jest szczególnie istotne przy opracowywaniu systemu tak w przypadku, gdy wymagany jest zdolny do pracy system częściowy, jak i wtedy, gdy będzie na nim budowana reszta systemu programowego.

2. Hierarchia oparta na relacji "jest częścią" /"is part of"/

Często moduły są dzielone na podmoduły, który to podział, chociaż z zewnątrz niewidoczny, jest istotny, gdy moduł jest zbyt duży na to, aby mógł być całkowicie opracowany przez jedną osobę w czasie kilku tygodni. Relacja "jest częścią" między modułami opiera się na samym pojęciu lub zasadzie projektowania modułu zachowującej również ważność dla jego podmodułów, co automatycznie egzekwuje hierarchię, jako że część jest zawsze mniejsza od całości.

Nie ma prostego związku między hierarchią "użyty" i hierarchią "jest częścią", gdyż z faktu, że dwa moduły wspólnie korzystają z jakiegoś szczególnego założenia /np. struktury tablicy/ nie wynika, że wymagają one tych samych modułów dla wykonania swoich funkcji. W hierarchii typu "użyty" mogą one nawet być odległe od siebie.

System programowy o strukturze hierarchicznej ma ograniczony zakres możliwych wariantów wdrożenia. Może to być wadą jak i zaletą. Podstawową zaletą struktury hierarchicznej jest to, że daje ona możność tworzenia złożonych systemów z prostszych składników programowych w łatwo zrozumiały sposób. Jednak celowość nadania programowi struktury hierarchicznej zależy od tego na jakie człony dokonano dekompozycji systemu i jaki jest charakter relacji między członami.

Jedną z cech programowania strukturalnego jest stosowanie systematycznego doskonalenia typu krok-po-kroku, co umożliwia opracowania struktury hierarchicznej zgodnie z celami systemu programowego.

B. ABSTRAKCJA

Abstrahowanie jest techniką myślowego opanowywania złożonych systemów, opartą na elementarnej zdolności ludzkiej pamięci zwanej "chunking"[x]. Zgodnie z teorią, pojemność pamięci człowieka może być mierzona za pomocą ilości "klooków". W pamięci ludzkiej "klocek" to pojęcie przedmiotu lub zespołu przedmiotów. Doświadczenia psychologiczne wykazały, że ilość "klocków", którymi człowiek może równolegle operować w pamięci, w procesie myślenia, na ogół nie przewyższa siedmiu. W przypadku usiłowania zwiększenia liczby elementów powyżej siedmiu, procent błędu myślenia rośnie wykładniczo. Mózg pokonuje to ograniczenie własnej pojemności pamięciowej poprzez łączenie określonych przedmiotów pod wspólnym pojęciem i nadanie mu oddzielnej nazwy. Z chwilą nazwania, pojęcie to jest już traktowane jako jeden "klocek".

Abstrakcja w naszym przypadku jest sposobem tworzenia "klocków" przez programistę. Jest to proces uogólniania, pole-

---

x/ Chunk - klocek, chunking - składanie klocków

gający na koncentrowaniu się na podobieństwach lub wspólnych właściwościach przedmiotów i składaniu ich w nowy przedmiot według tych ich wspólnych cech. Poprzez ograniczenie procesu uogólniania do właściwości istotnych dla składającego, abstrahuje się od wszystkich pozostałych właściwości przedmiotów.

Dekompozycja jest procesem przeciwnym do abstrakcji. Polega na rozkładaniu dużych "klocków" na mniejsze i wreszcie na pojedyncze elementy. Tutaj wspólne cechy są ignorowane, a pod uwagę bierze się zasadnicze różnice. Abstrakcja i dekompozycja są istotne dla pracy programisty nad funkcjami i strukturami danych. W odniesieniu do funkcji, dekompozycja oznacza m.in. rozłożenie funkcji na podfunkcje, współdziałające z sobą dla spełnienia zadań funkcji pierwotnych. Hierarchia logiczna przy tworzeniu struktury złożonych systemów programowych może być budowana iteracyjnie, "od dołu-w górę" /bottom-up/ drogą abstrakcji i "od góry-w dół" /top-down/ drogą dekompozycji. Hierarchia wypracowana w ten sposób składa się z poziomów będących wytworem procesów abstrakcji lub dekompozycji i stanowiących poszczególne poziomy abstrakcji. Pojęcie to można wyjaśnić na przykładzie dekompozycji.

Projektowanie systemu programowego rozpoczyna się od zgrubnego opisu rozwiązania problemu przetworzenia danych oraz opisu systemu, na którym ma się opierać projektowany system programowy. Opis rozwiązania problemu tworzy pierwszy, górny, abstrakcyjny poziom budowanej hierarchii, podczas gdy system bazowy tworzy dolny poziom abstrakcji. Pojęcia przetwarzania danych, funkcje lub operacje - i struktury danych, wykorzystuje się do opisu rozwiązania problemu. W złożonych rozwiązaniach występuje znaczny dystans między pojęciami systemu bazowego, a więc i duży stopień abstrakcji. W konsekwencji uciekamy się do kolejnych procesów dekompozycji i precyzowania zgrubnych pojęć najwyższego - określonego opisem rozwiązania - poziomu abstrakcji metodą "krok-po-kroku" i przetransformowania tych zgrubnych pojęć na pojęcia systemu bazowego. Iteracyjne stosowanie dekompozycji powoduje wzrost ilości poziomów abstrakcji, ale ze zmniejszającym się stopniem abstrakcji w każdym z nich. Każdy poziom abstrak-

cji powoduje uprecyzyjnienie wyższego poziomu abstrakcji i w wyniku uściślenie pierwotnego problemu, a idąc w dół, każdy poziom abstrakcji wydobywa szczegóły poziomu niższego i w konsekwencji, systemu bazowego. Wykorzystuje swoje pojęcia dla określenia nowych pojęć, które lepiej odpowiadają poszukiwanym rozwiązaniom.

Dijkstra opisał idąc "od dołu-do góry" właściwości poziomów abstrakcji A_0, A_1,... A_n, rozpoczynając od systemu bazowego A_0, jak następuje:

- Pojęcia, struktury danych i funkcje poziomu A_i tworzą kompletną podstawę, tak że może być na niej budowany lub z niej określony poziom A_{i+1}. Nie można natomiast użyć pojęć z poziomu A_{i-1} dla określenia poziomu A_{i+1}. Tak więc każdy poziom A_i tworzy pełny opis punktów intersekcji w hierarchii.
- Pojęcia poziomu A_{i-1}, których użyto dla zdefiniowania nowych pojęć poziomu A_i, nie mogą występować na poziomie A_i.
- Poprawność rozwiązania problemu pierwotnego może być ustalona metodą "krok-po kroku", poprzez sprawdzanie poprawności definicji każdego poziomu abstrakcji A_{i+1}, na podstawie poziomu abstrakcji A_i.

Porównując ten opis z definicją struktury hierarchicznej rozpoznaje się właściwość hierarchicznej struktury poziomów abstrakcji. W praktyce trudno jest stosować się do drugiej z trzech wyżej wymienionych właściwości. Często na poziomie A_i występują pojęcia identyczne z pojęciami na poziomie A_{i-1}.

Poprzez wprowadzenie pojęć każdy poziom abstrakcji logicznie określa nowy język programowania. Szkielet języka jest dany przez zestaw dopuszczalnych operacji, które można traktować jako ilość instrukcji dla maszyny, na której dokonuje się wdrożenia powyższego poziomu abstrakcji. Dlatego pojęcie "poziom abstrakcji" często jest utożsamiane z pojęciem "maszyny abstrakcyjnej". Jednak w publikacjach na temat programowania strukturalnego termin: poziom abstrakcji nie ma ujednoliconej definicji. Używaliśmy pojęcia poziom abstrakcji w odniesieniu do hierarchii modularnej na podstawie relacji "jest częścią", ponieważ

te moduły dopuszczają abstrakcję poprzez przemilczenie szczegółów wdrożenia. Niemniej pojęcie poziomu abstrakcji jest używane od czasu do czasu w powiązaniu z hierarchią modularną w oparciu o relację "użyty". Dowiedziono właśnie, że obydwie te hierarchie nie są identyczne.

C. METODOLOGIA PROJEKTOWANIA

W poprzedniej części pokazano, że model maszyny abstrakcyjnej może być tworzony zarówno przez dekompozycję, jak i abstrakcję. Te dwie techniki myślowe odpowiadają technikom projektowania "od góry-w dół" i "od dołu-do góry". Zwolennicy programowania strukturalnego podkreślają zalety projektowania "od góry-w dół" w powiązaniu z systematycznym uściślaniem metodą "krok-po kroku".

1. Projektowanie "od góry-w dół"

Projektowanie systemu programowego "od góry-w dół" rozpoczyna się właściwie od stworzenia modelu użytkownika. Model taki zawiera dane charakteryzujące przyszłego użytkownika oraz opisuje jego funkcje i wzajemne oddziaływania między użytkownikiem a systemem programowym. Ponieważ technik mający opracować program buduje swoje pojęcie o nim na podstawie modelu, ten ostatni winien być wiernym obrazem użytkownika. Korzystne mogą okazać się tu odpowiedzi na następujące pytania:

- co ma robić program?
- jakiego wymaga wejścia?
- co daje na wyjściu?
- jakiej wymaga obsługi?
- jak będzie reagować na błędy i braki?

Przy wypracowaniu koncepcji użytkownika, należy pamiętać o "zasadzie najmniejszych udziwnień": program powinien, w miarę

możności wykonywać zawsze tylko to czego oczekuje po nim użytkownik i co jest potrzebne i zrozumiałe. Jasność stanowi kryterium efektywności programu - niejasności zwiększają prawdopodobieństwo błędów u użytkownika.

Jako przeciwieństwo tego bazowego modelu, formułuje się rozwiązanie problemu przetwarzania informacji jako wstępny, zgrubny zarys oparty o model maszyny abstrakcyjnej. Abstrakcyjna maszyna reprezentuje najwyższy poziom abstrakcji i odtwarza w uproszczony sposób podstawowy algorytm rozwiązania problemu. Dla określenia tego algorytmu używa się bardzo ogólnych i właściwie dobranych pojęć uwzględniających strukturę danych i funkcje. Na podstawie użytych pojęć maszyna abstrakcyjna ma dać pełny opis poszukiwanego cyklu przetwarzania informacji oraz umożliwić przeprowadzenie przybliżonych ustaleń ilościowych w oparciu o jakość rozwiązania problemu, podobnie jak z jego struktury określa się zgrubnie strukturę całego systemu programowego.

Maszynę abstrakcyjną formułuje się w dwu etapach. W pierwszym etapie staramy się intuicyjnie znaleźć abstrakcje odpowiadające problemowi, tak aby można je było opisać w drugim etapie.

Stosując wystarczająco pełny zgrubny opis systemu programowego, precyzuje się, krok-po-kroku, funkcje i struktury danych w kierunku maszyny bazowej. Poszczególne kroki tego procesu precyzowania składają się z intuicyjnych odkryć dekompozycji i dokładnego jej opisu. Ze składników najwyższej, abstrakcyjnej maszyny wybiera się do dekompozycji te składniki, których uszczegóławianie najmniej narusza inne składniki. Precyzowanie może się tyczyć funkcji /precyzowanie dynamiczne/ lub struktury danych /precyzowanie statyczne/. Przyjmowane są te decyzje projektowe, które ze wzrostem przybliżenia do maszyny bazowej zakładają specyficzne ograniczenia do ostatecznego rozwiązania problemu, które ma być wdrożone na maszynie bazowej. W konsekwencji pozostaje kierować się wyrosłą na gruncie doświadczenia zasadą "odłożenia decyzji". Decyzje projektowe tyczące się funkcji specjalnych lub struktur danych należy odkładać aż do momentu, gdy decyzja o dalszym uszczegółowieniu jest nie do uniknięcia.

Rozważmy dekompozycję zawierającą odpowiednie uszczegółowienie pojęć maszyny abstrakcyjnej zgodnie z określonymi celami systemu programowego. Taka dekompozycja może spowodować powstanie maszyny podporządkowanej, której pojęcia, funkcje i struktury danych są bardziej zbliżone do koncepcji maszyny bazowej niż te, które są związane z samą maszyną abstrakcyjną.

Te kroki na drodze precyzowania mogą być powtarzane aż do czasu, kiedy funkcje najwyższej maszyny abstrakcyjnej zostaną rozbite na takie podfunkcje, które będą łatwo zrozumiałe i które będzie można opisać posługując się funkcjami i strukturami danych maszyny bazowej.

Dijkstra wyjaśnił pojęcie maszyny abstrakcyjnej na przykładzie drukarki liniowej, która może być sterowana rozkazami.

NLCR /new line carriage return - karetka z powrotem do nowego wiersza/ i PRSYM/n/:

- NLCR — określa najbardziej na lewo wysuniętą pozycję drukowania w następnym wierszu jako faktyczną pozycję drukowania
- PRSYM/n/ — drukuje w faktycznej pozycji drukowania znak określony przez wartość parametru n, wyrażony liczbą całkowitą oraz określa następną, wysuniętą na prawo, pozycję drukowania jako faktyczną pozycję drukowania.

Rozważmy przykład, w którym ma być opracowany program używający dwu rozkazów do sterowania wydrukiem funkcji całkowej danej w postaci parametrycznej z parametrem dyskretnym, wykorzystujący przy tym drukarkę wierszową w taki sam sposób jak ploter cyfrowy.

Dijkstra następująco zdefiniował maszynę najwyższej abstrakcji:

```
COMPFIRST
begin
draw: /build; print/;
```

var image

instr. build /image/, print /image/

end

Ta maszyna abstrakcyjna używa następujących składników:

- obraz /image/ struktury danych zastępujący obraz reprezentacji, która ma być drukowana,
- rozkaz buduj /build/, który tworzy ten obraz /image/,
- rozkaz drukuj /print/, który powoduje wydruk tego obrazu /image/.

Obraz struktury danych oraz zestaw dopuszczalnych operacji buduj i drukuj tworzą lingwistyczną podstawę dla sformułowania programu kreśl, który rozwiąże określony problem. Jednak operacje buduj i drukuj, tak jak obraz struktury danych, są abstrakcjami niezrozumiałymi dla maszyny bazowej z rozkazami NLCR i PRSYM /n/. Dlatego muszą one być dokładnie zanalizowane za pomocą dekompozycji.

Dekompozycja, której uszczegółowienie w najmniejszym stopniu narusza pozostałe składniki jest uprecyzyjnieniem "buduj". Prowadzi to do maszyny abstrakcyjnej, której operacja "wyczyść" /clear/ przypisuje wartość "blank" /= puste miejsce, brak znaku/ do obrazu struktury danych i której operacja staw znak /Setmark/ ustala dla wydruku obraz funkcji na obrazie struktury danych wg znaków. Program buduj zdefiniowany na podstawie tych dwu rozkazów symuluje operację buduj ww. abstrakcyjnej maszyny COMPFIRST.

Ale operacje wyczyść i staw znak też nie są rozumiane przez maszynę bazową. Podobnie jak operacja drukuj maszyny abstrakcyjnej COMPFIRST, wymagają one dalszego sprecyzowania.

Dekompozycja, która najmniej ogranicza uszczegółowienie innych składników prowadzi do maszyny abstrakcyjnej:

ISCANNER

begin i nteger i;

setmarks: i: = 0, while 1000 do add mark; i plus 1;

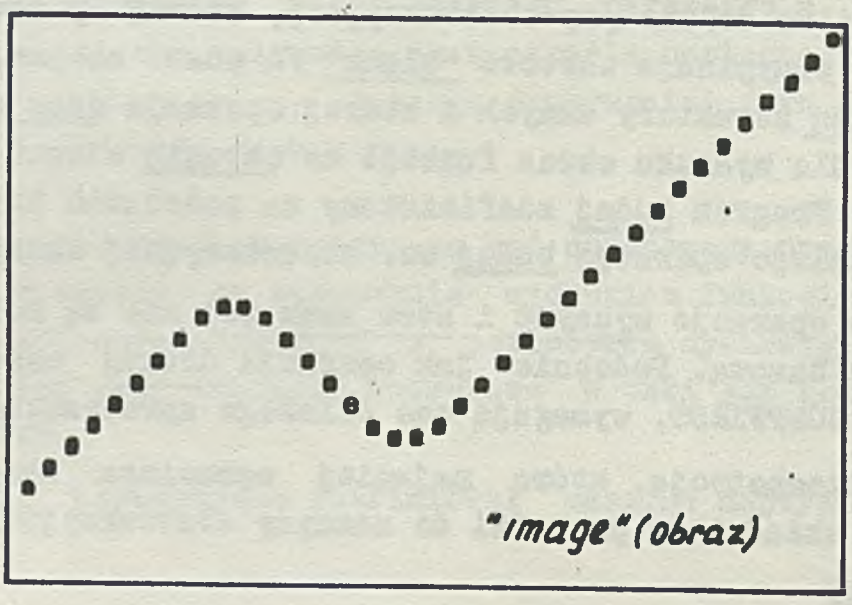
```
instr add mark /i, image/
end
```

. ... której operacja dodaj znak /add mark/ ustawia i-te oznakowanie w obrazie /image/. Program tej maszyny staw znak /set-mark/ określa kolejność znaków /the character sequence/ w jakiej dokonywane są oznakowania i symuluje operację staw znak abstrakcyjnej maszyny CLEARFIRST.

Dalszym krokiem jest szczegółowa analiza operacji dodaj znak /add mark/, która prowadzi do abstrakcyjnej maszyny

```
COMPOS
begin integer x,y;
add mark: x: = f x /i/; y = f y /i/; mark pos:
instr mark pos /x,y, image/
end
```

.... której operacja mark pos /pozycja znaku/ ustawia oznakowanie o współrzędnych x i y w obrazie /image/.



W tym momencie został osiągnięty punkt krytyczny szczegółowej, "krok-po-kroku", analizy, ponieważ dalsza dynamiczna, szczegółowa analiza różnych funkcji lub operacji jest niemożliwa bez statycznej, dokładnej analizy obrazu struktury danych.

Decyzja dekompozycji obrazu struktury danych definiuje obraz jako obszar 50 wierszy, będących podmiotem rozkazów NLCR i PRSYM/n/. Ta szczegółowa analiza prowadzi do maszyny abstrakcyjnej

```
LINER
begin integer j;
image: arrey line 0 : 49 :
print: j: = 49; while j 0 do lineprint /line j, j minus 1/:
clear: j: = 49; while j 0 do line clear /line j; j, minus 1/:
```

której pojęciami są: wiersz struktury danych oraz operacje drukuj wiersz /line print/; wyczyść wiersz /line clear/ oraz oznakuj wiersz /line mark/. Programy print, clear i mark pos symulują odpowiednie maszyny abstrakcyjne COMPFIRST, CLEARFIRST i COMPOS w oparciu o wiersz struktury danych.

Ponieważ operacje line print, line clear i line mark nie są zdefiniowane w maszynie bazowej, należy przejść do następnej analizy szczegółowej przez zdefiniowanie wiersza jako obszaru 100 elementów, które zawierają, zgodnie z pozycją drukowania i, parametry dla operacji PRSYM. Parametrami dla PRSYM są spacje /miejsca puste/, oznaczające, że nie można drukować w pozycji drukowania oraz mark /znak/ oznaczający, że w bieżącej pozycji drukowania ma być wydrukowany znak. Dekompozycja "line" /wiersza/ prowadzi do maszyny abstrakcyjnej

```
LONGREP
begin integer k
line: integer array sym 0 : 99;
lineprint : k: = 0; while k 100 do PRSYM /symk/; k plus 1;
                NLCR;
```

```
lineclear : k: = 0; while k 100 do sym k: = space; k plus 1
linemark: symx : = mark
end
```

Tu programy line print, line clear i line mark mogą być wykonane za pomocą rozkazów maszyny bazowej, PRSYM i NLOR.

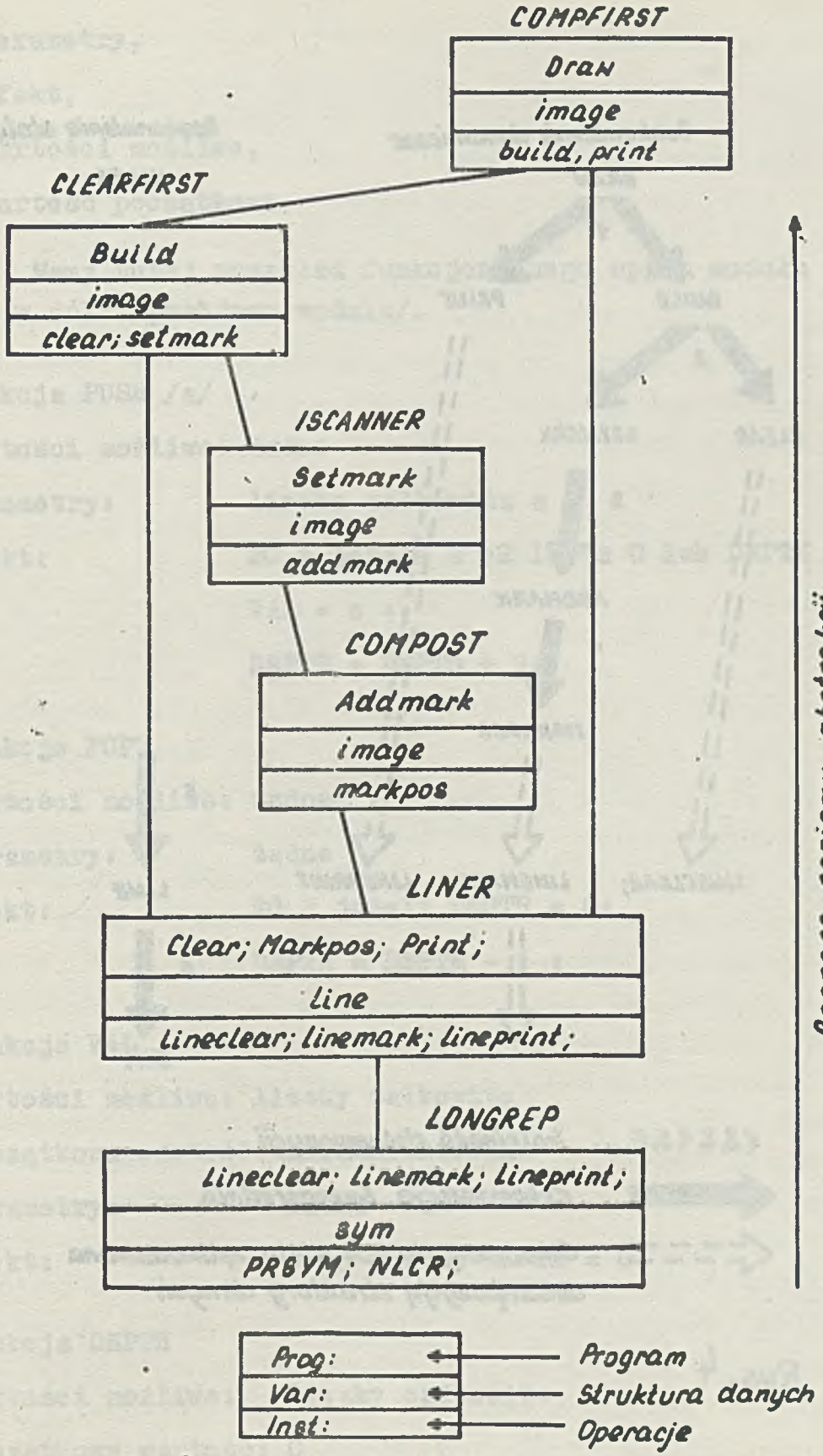
Hierarchię maszyn abstrakcyjnych opracowanych przez systematyczne uściślanie "krok po kroku", pokazano na rysunkach 3 i 4.

Podfunkcje, które można opisać za pomocą pojęć maszyny bazowej i które nie mogą być dalej dekomponowane, tworzą najniższy poziom modułów. Tworzą one bazę dla wdrożenia wyżej ustawionych modułów w następnej, wyższej maszynie abstrakcyjnej, która jest podstawą dla maszyny abstrakcyjnej znajdującej się jeszcze wyżej.

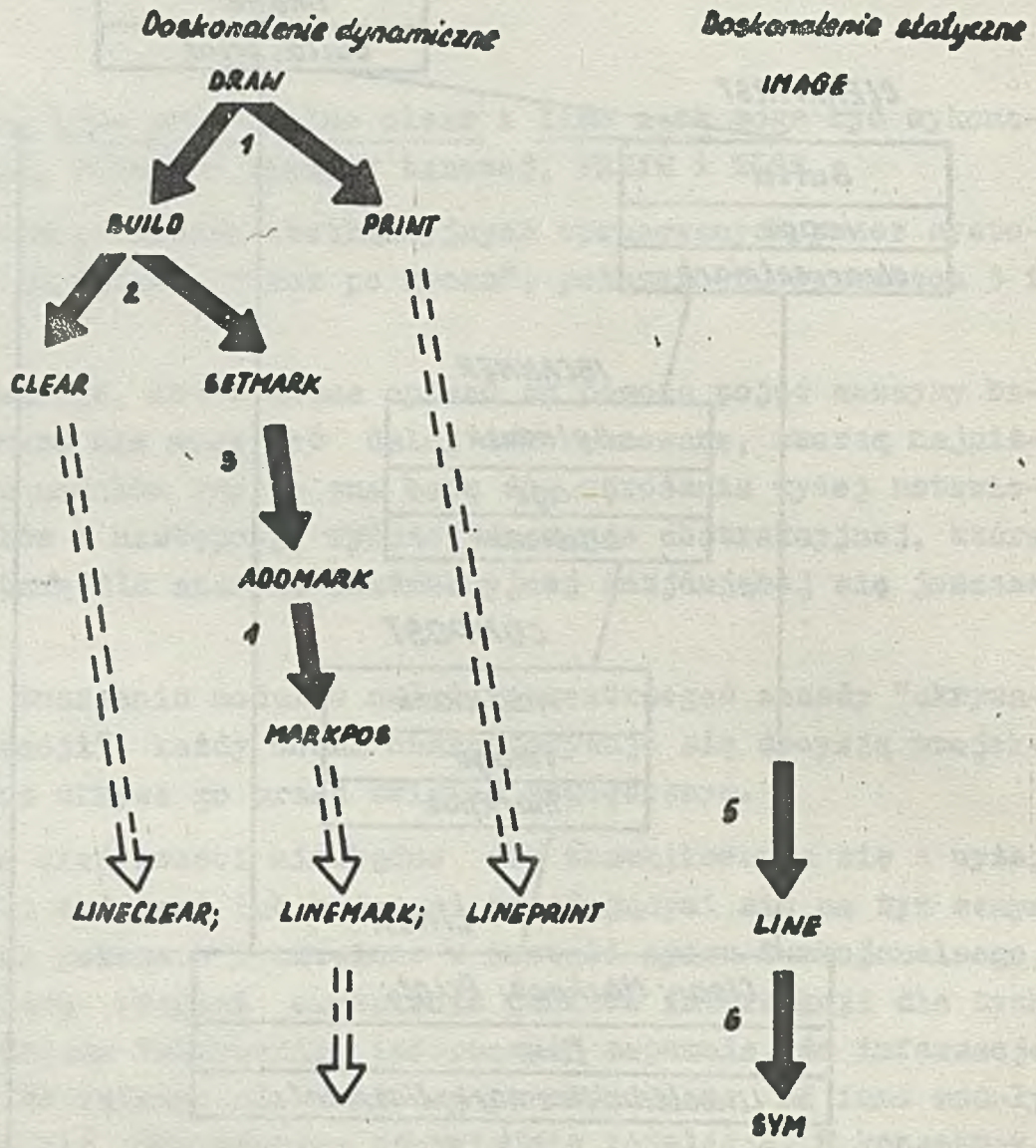
Przy tworzeniu modułów należy przestrzegać zasady "ukrywania informacji": każdy moduł charakteryzuje się decyzją projektową, która ukrywa go przed światem zewnętrznym.

Tylko właściwości niezbędne dla komunikowania się z wyżej ustawionymi modułami lub modułami znajdującymi się na tym samym poziomie, są pokazane i ustalone w postaci opisu funkcjonalnego, który zawiera również określenie punktów intersekcji dla tych modułów. Zasada "ukrywania informacji" zapewnia, że informacje niezbędne do zmiany modułu nie są porozdzielane na inne moduły i że muszą być wypracowane odpowiednie korelacje. W konsekwencji moduł nie może funkcjonować w zakresie odbiegającym od tego, który określono w opisie funkcjonalnym.

Struktury opracowanej na podstawie decyzji projektowej i charakteryzującej się relacją między modułami typu "jest częścią" nie można mylić ze strukturą powstającą w oparciu o relację "użyty" poprzez programowe wdrożenie modułów. Ściślej mówiąc funkcjonalny opis modułu tworzy podstawę dla opisu jego funkcjonalnej realizacji w postaci projektu programu. Efektywną metodą funkcjonalnego opisu modułu jest specyfikacja następujących zmiennych dla jego poszczególnych funkcji:


COMPFIRST
Oran
image
build, print
CLEARFIRST
Build
image
clear; setmark
ISCANNER
Setmark
image
addmark
COMPOST
Addmark
image
markpos
LINER
Clear; Markpos; Print;
line
lineclear; linemark; lineprint;
LONGREP
lineclear; linemark; lineprint;
sym
PRSYM; NLCR;
Rosnące poziomy abstrakcji
Prog:
Var:
Inst:
Program
Struktura danych
Operacje

Rys. 3


Doskonalenie dynamiczne
Doskonalenie statyczne
DRAW
IMAGE
1
BUILD
PRINT
2
CLEAR
SETMARK
3
ADDMARK
4
MARKPOS
5
LINE
6
SYM
LINECLEAR;
LINEMARK;
LINEPRINT
1,2,3,4,5,6.... kolejność dekompozycji
dekompozycja bezpośrednia
dekompozycja pośrednia spowodowana dekompozycją struktury danych

Rys. 4

. parametry,

. efekt,

. wartości możliwe,

. wartość początkowa,

Mamy tutaj przykład funkcjonalnego opisu modułu "zepchnięcia w dół" /pushdown module/.

Funkcja PUSH /a/

wartości możliwe: żadne

parametry: liczba całkowita a

efekt: EC 1 jeżeli a p2 lub a 0 lub DEPTH = p1,

VAL = a ;

DEPTH = DEPTH + 1;

Funkcje POP

wartości możliwe: żadne

parametry: żadne

efekt: EC 2 jeżeli DEPTH = 0;

DEPTH = DEPTH - 1 ;

Funkcja VAL

wartości możliwe: liczby całkowite

początkowa wartość: niezdefiniowana

parametry: żadne

efekt: EC 3 jeżeli DEPTH = 0;

Funkcja DEPTH

wartości możliwe: liczby całkowite

początkowa wartość: 0

parametry: żadne

efekt: żaden

Własności modułu:

p1: maksymalna ilość elementów w module zepchnięcia w dół

p2: maksymalna wartość elementu

Ciągi PUSH /a/; POP nie dają efektów o ile nie ma wołania błędu.

W tej metodzie efekt modułu widoczny z zewnątrz po rozwiązaniu, jest zdefiniowany za pomocą zachowania się mechanizmu modułu, chociaż bark jest informacji o szczegółach wdrożenia lub chronologicznej kolejności funkcji.

2. Krytyka projektowania "od góry-w dół"

Technika projektowania "od góry-w dół" jest użyteczna przy rozwiązywaniu wielu problemów projektowych, ale powoduje konieczność pokonania wielu trudności. Tworzenie wewnętrznej struktury systemu programowego w oparciu o zasadę dokładnego opisu celów rozwiązania problemu ma określone zalety. Stwarza możliwość zabezpieczenia niezbędnych właściwości systemu przed zniszczeniem lub uczynieniem go niezrealizowanym. Zmniejsza ryzyko stworzenia systemu o niespodziewanych i niepożądanych właściwościach. Zalety te nie sygnalizują podstawowej wady projektowania "od góry-w dół". Jeżeli rozważamy maszynę abstrakcyjną A_j, z punktu widzenia projektowania "od dołu-do góry", istnieje możliwość zastosowania wielu różnych maszyn abstrakcyjnych A_{j+1}. Projektowanie "od góry-w dół" opracowuje tylko jedną maszynę abstrakcyjną na określonym poziomie abstrakcji, ponieważ zakłada się, że istnieje tylko jeden problem. Ale jeżeli problem, który ma być rozwiązany, może być podzielony na pojedyncze podproblemy, które muszą być rozwiązywane jednocześnie, wówczas należy opracować moduły wspólnie wykorzystujące zasoby, a ich praca musi być zsynchronizowana. W tym przypadku projektu systemu programowego nie można otrzymać na drodze kolejnych kroków dekompozycji i lepsze jest projektowanie "od dołu-do góry".

Generalnie, projektowanie oznacza przejście przez poszczególne poziomy abstrakcji więcej niż jeden raz oraz rewizję wcześniejszych decyzji projektowych, ponieważ często nie sposób jest rozpoczynać opracowania od udowodnionych założeń. Dla każdego poziomu abstrakcji niezbędne jest zapewnienie kompletności oraz upewnienie się, że na poziomie wyższym nie pominięto rzeczy istotnych. W toku dokonywania korekty hierarchii należy brać pod uwagę sprawę wydajności programu. Każda maszyna abstrakcyjna A_i, generalnie jest wolniejsza niż wszystkie maszyny poniżej jej, ponieważ wykonanie funkcji na maszynie abstrakcyjnej A_i powoduje konieczność wołania modułów z niżej stojącej maszyny abstrakcyjnej A_{i-1}. Oznacza to, że ze względu na wydajność, funkcje często używane są wdrażane na poziomie możliwie najbliższym systemu bazowego, po to, aby przyspieszyć ich wykonanie. Pociąga to za sobą okrężne dojście do struktury hierarchicznej otrzymywanej przez szczegółowe precyzowanie "krok-po-kroku".

Ostatnio często krytykowane w projektowaniu "od góry-w dół" jest to, że technika doskonalenia "krok-po-kroku" jest zbyt trudna dla przeciętnego programisty, ponieważ ma on do czynienia ze zbyt dużą ilością możliwości, nie dysponując jednocześnie jakimś obiektywnym kryterium poprawności swoich decyzji projektowych.

Jackson zaleca tu podejście pragmatyczne, oferujące programiście schemat nadawania struktury programom, ale jest ono wystarczające tylko przy prostych programach typu zastosowaniowego. Polega ono na wykorzystaniu struktury danych występujących w programie jako szkieletu dla szczegółowej, "krok-po-kroku", analizy i projektowania struktury programu w sposób schematyczny, zgodnie ze strukturą danych.

Jackson podkreśla, że struktura rozwiązania problemu najlepiej może być wyprowadzana ze struktury używanych danych, ponieważ dane oddają model istoty problemu i charakter kroków przetwarzania jest określony właśnie przez elementy struktury danych. Jest to właśnie myśl, którą opracował Dijkstra w przykładzie programu drukowania dla drukarki wierszowej. Zgodnie z tym projekt programu jest dzielony na następujące kroki:

1. Definicja i analiza struktur danych pojawiających się w programie; reprezentacja w postaci wykresów strukturalnych.

2. Próba łączenia struktur danych pojawiających się w programie w jedną strukturę danych poprzez pokrewieństwo lub dopasowanie składników strukturalnych; analiza sprzeczności strukturalnych, tj. niezgodności między logicznymi strukturami danych; projektowanie struktury programu zgodnie ze schematem struktury danych przy zastosowaniu bloków strukturalnych.

3. Przydzielenie niezbędnych funkcji przetwarzania do poszczególnych bloków strukturalnych; analiza częstotliwości wykonania; kolejności wykonywania i warunków związanych z wykonaniem; odpowiednie zrewidowanie kroku 2.

4. Analiza zastosowania struktur danych lub funkcji udostępnionych do ogólnego używania przez system do wdrożenia projektowanego programu; redefinicja możliwości, przydział tymczasowej pamięci lokalnej.

5. Analiza nagłówków przetwarzania w każdym bloku, możliwości weryfikacji ogólnej używalności bloku, przydatność obecnie istniejącego kodu i konieczność optymalizacji; odpowiednie zrewidowanie kroku 2 i ustalenie wdrażania.

6. Definicja typu każdego bloku; parametry; sposób działania; ograniczenia dotyczące poprawności wejścia.

Ten schemat programowania strukturalnego jest bardzo efektywny w programowaniu typu zastosowaniowego, szczególnie w problemach ekonomicznych. Jest to jedno z pierwszych, pragmatycznych i praktycznych podejść rozszerzenia zastosowania techniki programowania strukturalnego.

Parnas wysunął więcej problemów związanych z projektowaniem "od góry-w dół" a mianowicie:

- Bardzo trudno jest sformułować dokładnie decyzje projektowe bez włączenia wewnętrznych właściwości poszczególnych modułów.
- Często projektowania nie da się wyprowadzić bezpośrednio ze specyfikacji na podstawie modelu maszyny abstrakcyjnej. Dla

podanej z zewnątrz specyfikacji istnieje wiele struktur wewnętrznych, tak że potrzebne są pewne dodatkowe ograniczenia zanim mogą być rozpoczęte badania dla celów udoskonalenia. Ograniczenia te są zwykle określone przez maszynę bazową.

- Przy projektowaniu "od góry-w dół" proste jest projektowanie mechanizmów wewnętrznych, które, chociaż reprezentują idealne wdrożenie pożądanych właściwości, są bardzo trudne do zrealizowania.
- Projektowanie "od góry-w dół" jest trudne w zastosowaniu do grupy systemów objętych ogólnym pojęciem "systemów wielocelowych".
- Projekt oprogramowania często jest budowany w sposób ogólny, w oparciu o ustaloną maszynę bazową /sprzęt dla systemu operacyjnego, system operacyjny dla programów użytkownika/. W tych przypadkach podejście "od góry-w dół" jest bezużyteczne.

W wielu przypadkach - na przykład w projektowaniu systemów operacyjnych - trudno jest zdefiniować "górę" systemu programowego. Czasem jako "górę" definiuje się świat zewnętrzny systemu programowego. Ta metoda może prowadzić do niepotrzebnie usztywnionego systemu. Często obserwuje się, że pierwsze decyzje projektowe są najtrudniejsze do zmiany po zakończeniu systemu i w wielu przypadkach właśnie świat zewnętrzny zmienia się najszybciej. W konsekwencji, projektowanie "z zewnątrz-w" /outside-in/ może prowadzić do powstania systemów trudnych do zrealizowania ze względu na to, że cały system jest robiony pod kątem świata zewnętrznego.

Wymienione problemy wskazują na to, że w praktyce czyste projektowanie "od góry-w dół" jest trudne i problematyczne do zrealizowania. Ważne jest zrozumienie jak projektowanie ogranicza wdrażanie i jak możliwości informacyjne ograniczają projektowanie.

Dlatego Parnas sugeruje, że czyste projektowanie "od góry-w dół" należy uzupełnić zrewidowaniem poziomów abstrakcji zgodnie z zasadą jasności abstrakcji w procedurze "od dołu-do góry". W typowym etapie projektowania "od dołu-do góry" występują dwa poziomy: dobrze zdefiniowana, niższa maszyna abstrakcyjna A_i i znajdująca się nad nią maszyna abstrakcyjna A_{i+1}, opracowana przez abstrakcję. Przez dobrze zdefiniowaną rozumiemy, że wszystkie możliwe stany, które są osiągalne za pomocą funkcji maszyny mogą być wykazane. Dla każdego wdrożenia maszyny abstrakcyjnej A_{i+1} na maszynie abstrakcyjnej A_i można ustalić, jakie stany A_i można osiągnąć przez wykonanie programów w języku maszyny abstrakcyjnej A_{i+1}.

Maszyny abstrakcyjne A_i i A_{i+1} są zupełnie przejrzyste gdy

- każdy stan maszyny bazowej A_i posiada odpowiedni stan w maszynie A_{i+1},
- wszystkie szeregi stanów maszyny bazowej A_i, które są generowane przez jej operacje także będą generowane przez operacje maszyny A_{i+1}.

Jeżeli jakieś stany lub szeregi stanów maszyny bazowej A_i nie mogą być osiągnięte przez programowanie maszyny abstrakcyjnej A_{i+1}, uważa się to za utratę przejrzystości.

Gdy każdy poziom jest oceniany przez utratę przejrzystości i mamy pewność, że nie zostały zgubione pożądane właściwości, możemy zagwarantować, że wyższe poziomy abstrakcji także mają żądane właściwości. Rozpoczynamy od założenia, że elementarne mechanizmy najniższej maszyny abstrakcyjnej mogą robić wszystko czego ostatecznie żąda się od systemu na najwyższym poziomie.

We wszystkich rozsądnych wdrożeniach, w których liczy się kryterium wydajności, będzie występować zgubienie przejrzystości. Ludzie biorący udział w projektowaniu powinni mieć to na uwadze i przygotować się na trudną analizę, poprzedzającą decyzję projektową.

E. POSTĘPOWANIE W PRZYPADKU POJAWIANIA SIĘ BŁĘDÓW

Wszystkie poprzednie rozważania dotyczące metodologii projektowania odnosiły się do projektowania poprawnie działających systemów. W praktyce jednak pojawiają się błędy w systemach programów, niezależnie od wysiłku, jaki zrobiono, aby się przed nimi uchronić. Błędna sytuacja jest dalej komplikowana przez hierarchiczną strukturę systemu programowego. Funkcje niższych poziomów abstrakcji muszą nadawać się do pracy bez poziomów abstrakcji stojących powyżej nich i muszą nadawać się do użycia przez moduły na różnych, wyższych poziomach. W konsekwencji niższe poziomy abstrakcji nie powinny mieć żadnej wiedzy o wyższych poziomach abstrakcji. Dla zajęcia się błędami zwykle wymagane jest zespolone działanie kilku poziomów abstrakcji i modułów. Dzieje się tak dlatego, że zwykle błąd jest wykrywany przez niższy poziom abstrakcji, jednak dopiero informacje na wyższym poziomie abstrakcji umożliwiają podjęcie odpowiednich działań korygujących.

Trudności jakie pociąga za sobą programowanie strukturalne wiążą się z podziałem wiedzy i hamują działania wymagające znajomości wielu poziomów abstrakcji. Teoretycznie istnieją dwa sposoby postępowania w przypadku wykrycia błędów:

- prześlij komunikat o pojawieniu się błędu na wyższy poziom abstrakcji,
- przygotuj informacje potrzebne do zajęcia się komunikatem o pojawieniu się błędu na poziomie abstrakcji, na którym błąd może się pojawić.

Drugi sposób stawia pod znakiem zapytania zalety struktury hierarchicznej, pierwszy - stwarza trudności wynikające z faktu, że zgodnie z prawdopodobieństwem występowania błędu, kodowanie i postępowanie w przypadku jego pojawienia się ogromnie komplikuje sprawę kodowania programu. Najlepiej wyjaśnia to traktowanie wejścia i wyjścia, gdyż tutaj prawdopodobieństwo wystąpienia błędu jest bardzo duże. Dla oddzielenia kodowania programu dla przypadku standardowego, od postępowania w przy-

padku pojawienia się błędu, Parnas uważa za korzystne zastosowanie mechanizmu "pułapki" /trap/ w modułach hierarchicznie zbudowanego systemu, w taki sam sposób jak dla sprzętu. "Pułapki" są to programy, które są wołane dla często pojawiających się i łatwo wykrywalnych błędów po to, aby przejąć postępowanie w przypadku wystąpienia błędów. Zapewnia to dosłowne oddzielenie programu standardowego i postępowania w przypadku błędów. Odpowiedzialność za rozpoznanie kiedy następuje przekroczenie specyfikacji poziomu abstrakcji przejmuje sama maszyna abstrakcyjna, Gdy błąd jest rozpoznany następuje wejście programu "pułapki", który zajmuje się błędem oraz przesyła do wyższej maszyny abstrakcyjnej komunikat błędu o uszkodzeniu maszyny abstrakcyjnej. Postępowanie w przypadku błędu powinno być zapisane przez użytkownika maszyny abstrakcyjnej i powinno mieć dostęp do tych samych danych co pozostałe moduły maszyny abstrakcyjnej. Tak więc maszyna abstrakcyjna jest odpowiedzialna za wszystkie skutki programu "pułapki". Umożliwia to kontynuację przetwarzania kodu dla standardowego przypadku po zajęciu się błędem.

W praktyce taka sytuacja pojawia się często tam, gdzie konieczność odpowiedniej reakcji na błędy narusza przyjęte abstrakcje. Dlatego należy pamiętać o następującej zasadzie: nie należy specyfikować abstrakcji, które są często obalane przez błędy. Często właściwsze będzie zaprojektowanie modułu raczej mniej abstrakcyjnego oraz umożliwienie w samym projektowaniu potraktowania najczęściej pojawiających się błędów jako błędów abstrakcyjnych.

Błąd wykryty na jednym poziomie systemu może być przekazany w dół jako sposób w jaki zostaną naruszone określone ograniczenia maszyny abstrakcyjnej emitującej błąd, lub może być przesłany wyżej jako uszkodzenie lub zawiadomienie o błędzie, który poprzednio był przesłany w dół. Po wykryciu błędu, błąd przesyłany w dół zawsze musi wrócić na wyższy poziom. Odpowiedzialność za diagnozę i określone postępowanie musi spoczywać na poziomie wyższym, ponieważ niższy poziom nie ma wystarczających informacji o specyfikacji wyższego poziomu.

W mechaniźmie "pułapki" komunikat o błędzie jest tworzony przez wołanie programu będącego częścią modułu, który jako ostatni będzie używać maszyny abstrakcyjnej. Poziom dowiaduje się p błędzie na podstawie wołania programu "pułapki". W każdym komunikacie o błędzie pierwszą rzeczą jest sprawdzenie czy błąd powstał na tym samym poziomie. Jeżeli autorem błędu jest poziom wyższy, wołany jest program "pułapka" na wyższym poziomie i wszystkie informacje na jego poziomie abstrakcji stają się dla niego dostępne. Gdy dany poziom sam jest autorem błędu, program "pułapka" sam podejmie odpowiednie działania lub prześle błąd, jako uszkodzenie maszyny abstrakcyjnej, do wyższego poziomu, poprzez wołanie programu "pułapki".

Niższe poziomy systemu nie powinny przerywać wykonania w przypadku pojawienia się uszkodzenia, ponieważ właściwe postępowanie lub zredukowanie skutków, może być możliwe na wyższych poziomach. W konsekwencji wykonanie powinno być przerwane tylko na najwyższym poziomie.

Mechanizm "pułapki" ma trzy zalety:

1. Każda procedura postępowania w przypadku błędu może być zapisana na tym poziomie, gdzie dostępne są wszystkie potrzebne informacje. Oznacza to możliwość tworzenia postępowania w przypadku błędów za pomocą abstrakcyjnych operacji tego poziomu.
2. Mechanizm "pułapki" daje możliwość poprawienia niezawodności systemu programowego bez potrzeby szerokich zmian. Przy pierwszej translacji systemu programowego programy "pułapki" będą tak proste, że ograniczą się tylko do wydruku nazwy. W miarę wzrostu doświadczeń i większego zrozumienia cykli wewnętrznych, programy mogą być udoskonalane przez bardziej wyszukane podejścia do postępowania w przypadku błędów.
3. W programach "pułapkach" można zapewnić wskazanie błędnego modułu w toku zajmowania się komunikatem o wystąpieniu błędu. W przeciwnym przypadku, gdy program powstaje przy udziale wielu programistów, błąd może się pojawić i nikt nie będzie wiedzieć do kogo należy zajęcie się nim.

Mechanizm "pułapki" jest logiczną i dobrze pomyślaną metodą postępowania w przypadku pojawienia się błędów w hierarchicznie zbudowanych systemach programowych, chociaż może mieć pewne wady gdy chodzi o efektywność programu. Na przykład wartość przechodząca przez szereg modułów może być stale badana, w każdym module dla zachowania tego samego zakresu wartości, gdy ten mechanizm jest konsekwentnie wdrażany. Ten nadmiar z pewnością jest istotny we wczesnym stadium testowania, ale gdy system jest ogólnie pewny obniżenie efektywności spowodowane redundancją zyskuje na znaczeniu. Istnieją dwa sposoby pozbycia się redundantnych testów w przypadku gdy błędy pojawiają się tylko rzadko:

1. Przy założeniu, że błędy nie pojawiają się na drodze do niższych poziomów abstrakcji, testy są trzymane na wyższych poziomach i usunięte z niższych poziomów.
2. Utrzymuje się testy na poziomach niższych, a programy "pułapki", na poziomach pośrednich, są używane po to, aby "zwrócić" błędy na poziomy, które je spowodowały.

Z wyjątkiem pewnych przypadków preferowany jest drugi sposób. W praktyce jednak, kiedy używany jest podprogram wołania dla zastosowania mechanizmu "pułapki", nadal istnieją problemy do rozwiązania:

- tożsamość programu wołającego nie jest znana podprogramowi,
- dynamiczny przydział programów "pułapek" jest niepotrzebnie skomplikowany,
- przebieg sterowania między programami standardowymi i postępowaniem w przypadku błędów jest bardzo ograniczony.

III. POJĘCIA SKŁADNI WE WDRAŻANIU MODUŁÓW

Podczas gdy w rozdziale II zajmowaliśmy się decyzjami projektowymi dla stworzenia struktury systemu programowego w postaci zrozumiałych modułów, w tej części przedyskutujemy koncepcje wdrożenia modułów w FORTRANie, ALGOLu 68, PL/1 i COBOLu. Aby to zrobić musimy odnieść pojęcie "modułowości" do poziomu lingwistycznego systemów APD. Modułowość oznacza, że na poziomie lingwistycznym systemu APD istnieje klasa obiektów składniowych, które jako jednostki reprezentujące systemy programowe, umożliwiają wdrożenie modułów. Poziom lingwistyczny wspiera połączenie kilku modułów dla stworzenia wyższego, większego modułu bez konieczności ich zmiany. Znaczenie modułu pozostaje niezmienione, niezależnie od kontekstu jego użycia.

Istnieją pewne podstawowe koncepcje modułowości w wymienionych językach programowania, które w naszym pojęciu reprezentują najważniejsze obszary problemowe w analizie możliwości lingwistycznych związanej z programowaniem strukturalnym.

A. MODUŁOWE ZASTOSOWANIA W ALGOLU

ALGOL posiada dwie istotne koncepcje w stosowaniu modułów - koncepcję bloku i koncepcję procedury. Chociaż dwa przetłumaczone programy ALGOLu nie mogą być połączone razem w pojedynczy moduł, to koncepcja procedury dostarcza możliwości połączenia wielu modułów dla stworzenia modułu ogólnego - który z kolei, może być przyłączony do innych modułów w tym samym programie ALGOLu. Podstawą logiczną dla koncepcji procedury jest koncepcja bloku.

Blok jest komplikacją logicznie powiązanych instrukcji w połączeniu z definicją danych, która jest ważna tylko w ramach tej złożonej instrukcji. Stąd, blok składa się z dwu składników

logicznych: z "nagłówka bloku" /"block head"/, w którym są zadeklarowane zmienne lokalne, ważne dla tego bloku i z "korpusu bloku" /"block body"/ składającego się z jednej lub więcej, instrukcji przetwarzania.

Blok jest osadzony statycznie w sekwencyjne sterowanie przebiegiem programu i nie może być sterowany z jakiegoś dowolnego punktu programu. Posiada on wejście i wyjście, a jego cykl dynamiczny odpowiada sekwencyjnym szeregom rozkazów w nim zawartych. Blok jest istotny z punktu widzenia programowania strukturalnego w tym sensie, że szereg bloków może być łączonych razem dla stworzenia większego, ogólnego bloku - w konsekwencji tekst programu może być klasyfikowany hierarchicznie.

Koncepcja procedury jest dynamiczną wersją koncepcji bloku. Ze statycznego punktu widzenia, procedura jest blokiem posiadającym właściwości hierarchicznej klasyfikacji tekstu programu oraz ograniczającym zakres deklaracji nazw. Ale ten blok określający procedurę jest tylko reprezentantem dynamicznej aktywacji procedur. Procedura może być przywołana z wielu punktów w tekście programu, a także z punktów w tekście samej procedury /przywołanie rekursywne/. Jednak, niezależnie od punktu, w którym w ramach tekstu całego programu jest umieszczony jej tekst, procedura może być aktywowana tylko za pomocą procedury przywołania, a nigdy przez sterowanie sekwencyjne.

Komunikacja danych w ramach otoczenia procedury zachodzi w dwojaki sposób:

- przez zmienne ogólne należące do części programu zawierającego procedurę,
- przez bieżące parametry punktu wołania lub przez parametry wynikowe, które są do tego punktu dostarczone.

Wymiana danych w ramach statycznego otoczenia procedury jest uzupełniana przez zmienne zewnętrzne. Procedury, które zmieniają zmienne zewnętrzne reprezentują bezpośrednią dekompozycję częściową ogólnego zadania programu. Przydział wartości zmiennym zewnętrznym podczas cyklu procedury jest określony jako "efekt uboczny" i z powodu superpozycji połączeń statycznych

i dynamicznych powoduje, że trudniej jest otrzymać jasny obraz dynamicznego zachowywania się programu. Co więcej, efekty uboczne nie są zgodne z zasadami programowania strukturalnego, gdy procedura tworzy moduł na poziomie A_i i zmienia dane na poziomie A_{i+1} poprzez mechanizm zmiennej zewnętrznej /taka zmiana nie jest dopuszczalna z racji funkcjonalnego opisu modułu/.

Dalej może wystąpić konflikt nazw jeśli używane są dwie procedury, które są zdefiniowane w ramach tej samej procedury lub w ramach tego samego bloku i które używają tej samej nazwy dla przywoływania zmiennej zewnętrznej, która sama została zdefiniowana w ramach procedury lub bloku zawierającym te nazwy. Aby nie dopuścić do tego rodzaju komplikacji należy przestrzegać zasady modułowości w ALGOLu, w wyniku której dane do i z procedury powinny być przesyłane tylko za pomocą mechanizmu formalnych parametrów.

Komunikacja danych w ramach otoczenia dynamicznego zachodzi poprzez połączenie bieżących parametrów punktu wołania z formalnymi parametrami procedury. Połączenia tego można dokonać w postaci statycznej substytucji na wejściu do procedury. Alternatywnie można to zrobić w postaci dynamicznej substytucji podczas wykonywania procedury przy dowolnym dostępie do formalnego parametru. W przypadku sybstytucji statycznej jest przesyłana jedna z następujących właściwości postaci danych:

- nazwa bieżącego parametru /wołana przez literał/,
- miejsce, gdzie jest zapamiętany bieżący parametr, w jakim przypadku pojawia się uboczny efekt w ramach procedury dzięki przydzieleniu wartości parametrowi towarzyszącemu /wołany przez odsyłacz/. Substytucja dynamiczna jest formą wołania przez odsyłacz, gdzie parametr bieżący jest przeliczany każdorazowo, ilekroć następuje odwołanie się do towarzyszącego, formalnego parametru. W tych okolicznościach należy podkreślić niebezpieczeństwo właściwe tej koncepcji, gdyż prowadzi to do trudnych do rozumienia struktur programowych.

Semantykę procedury wołania można opisać za pomocą reguły kopiowania: w każdym punkcie procedury "wołaj" należy sobie wy-

obrazić tekst procedury jako podlegający skopiowaniu i modyfikacji, w dowolnym, określonym czasie, zgodnie z bieżącymi parametrami punktu wołania. Tak rozwinięty program jest normalnie przetwarzany sekwencyjnie. Tak więc mechanizm procedury polega na utrzymywaniu tekstu programu w postaci krótkiej. Kiedy następuje przywołanie procedury przebieg sterowania opuszcza punkty wołania i zatrzymuje się na starcie wołanej procedury. Następuje przejście przez procedurę - w czasie którego mogą być przywołane inne procedury - i ponownie opuszcza ją przy jej dynamicznym zakończeniu. Procedura wołana zazwyczaj wraca do procedury przywołującej przeciwnie do kolejności wołania w wołaniach procedury gniazdowej, a w kierunku niższego poziomu tworzenia gniazd w wołaniach rekursywnych. W programowaniu strukturalnym odpowiada to poprawnej strukturze zachowania się dynamicznego, a przebieg sterowania można przedstawić w postaci drzewa.

W praktyce jednak, ze względu na wydajność, toleruje się dwa przypadki specjalne:

- nadzwyczajny powrót do najdalej od środka znajdującej się procedury,
- bardziej elastyczny, nadzwyczajny powrót do określonej procedury w kolejności wołania.

Z powodu niezgodności z zasadami programowania strukturalnego, obydwa przypadki powinny zachować charakter wyjątku o ograniczonych możliwościach stosowania dla zabezpieczenia się przed niezrozumiałym, dynamicznym zachowaniem się w programie. Dokładnie sekwencyjne zachowanie systemu programowego, dzięki właściwości synchronizacyjnej wołania procedury w powiązaniu z normalnym mechanizmem skoku, stanowi ograniczenie modelu maszyny abstrakcyjnej. Nie należy to całkowicie do sfery programowania strukturalnego, jednak wynika z określonych właściwości technicznego zastosowania koncepcji procedury.

Jedną z podstaw sekwencyjnego zachowywania się systemu programowego zmodularyzowanego w oparciu o procedury jest ograniczone życie kopii procedur, dynamiczne aktywowanie procedury. Kopie, które są aktywowane jako ostatnie przestają istnieć jako

pierwsze. Ta właściwość ograniczenia umożliwia zastosowanie języka do dynamicznego przydziału pamięci pojedynczym kopiom procedury w postaci stosu.

Rozpatrywana jako uproszczony model procedura składa się z części programu niezależnej od jej dynamicznej aktywacji i szeregu czynnych rekordów odpowiadających ilości dynamicznych aktywacji.

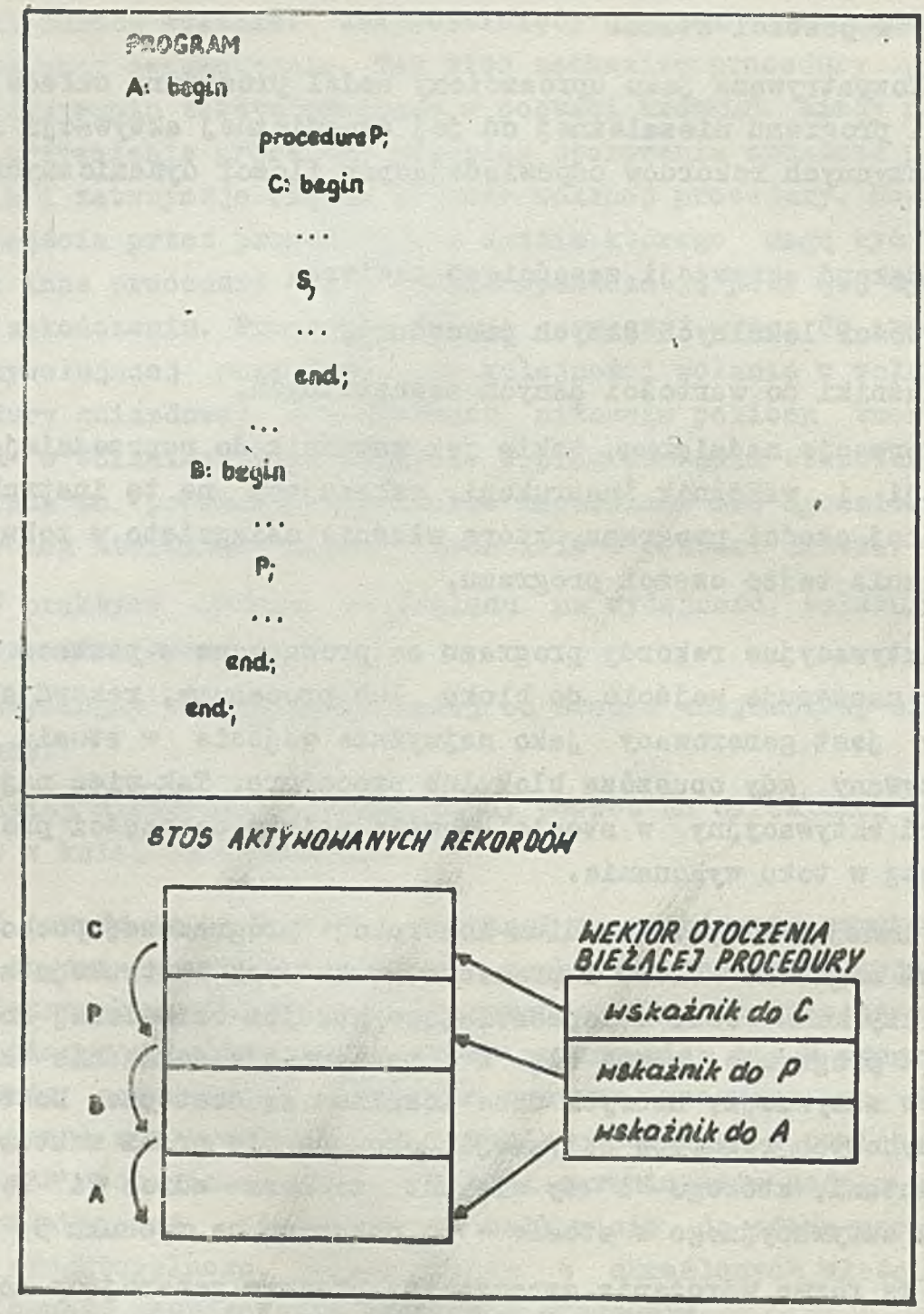
Rekord aktywacji zasadniczo zawiera:

- wartości lokalnych danych procedury,
- wskaźniki do wartości danych zewnętrznych,
- informacje nadmiarowe, takie jak wskaźnik do poprzedniej aktywacji i wskaźnik instrukcji, wskazujący na tę instrukcję w stałej części programu, którą właśnie osiągnięto w toku wykonywania tejże części programu.

Aktywacyjne rekordy programu są prowadzone w postaci stosu. Kiedy następuje wejście do bloku lub procedury, rekord aktywacyjny jest generowany jako najwyższe wejście w stosie i jest wymazywany gdy opuszcza blok lub procedurę. Tak więc najwyższy rekord aktywacyjny w stosie zawsze należy do części programu, będącej w toku wykonania.

Umiejscawiające stadium instrukcji programowej pochodzi od ilości nagłówków bloku i procedur, w których instrukcja zawiera się fizycznie. Jeśli umiejscawiające stadium określonej instrukcji w programie równa się "k", to istnieje dokładnie "k" rekordów aktywacji, których dane lokalne są dostępne. Dostępu do pojedynczych rekordów aktywacji dokonuje się przez wektor z "k" elementami, którego "i"-ty element zawiera adres "i"-tego rekordu aktywacyjnego w stosie - co pokazano na rysunku 5.

Ta forma wdrożenia oznacza, że program generujący nową kopię procedury nie oddziałuje na nią tak, jak byłoby to w przypadku programu. faktycznie istniejącego. Po ponownym przejęciu sterowania przez program wywołujący, uaktywniona przez niego kopia procedury jest wymazana podobnie jak i rekord aktywacyjny w stosie.



Rys. 5 RELACJA MIĘDZY WEKTOREM OKREŚLAJĄCYM OTOCZENIE BIEŻĄCEJ PROCEDURY AKTYWOWANYCH REKORDÓW

Ta forma wdrażania uniemożliwia równoległe istnienie na tym samym poziomie programu kilku aktywacji procedury, która komunikuje się z wyższym poziomem abstrakcji. Wynikiem tego jest ograniczenie modelu maszyny abstrakcyjnej. Koncepcje lepiej spełniające wymogi programowania strukturalnego, zostały wprowadzone do języka symulacyjnego SIMULA 67, zdefiniowanego na bazie ALGOLu 60. Można je zreasumować następująco:

1. Klasy postaci

Procedura, którą może aktywować kopię istniejącą nawet wówczas, gdy sterowanie wraca do programu wywołującego nosi nazwę klasy; kopie aktywowane przez klasę są nazywane postaciami. Klasę deklaruje się w taki sam sposób jak procedurę.

deklaracja klasy : : = class identyfikator

klasy części formalnego parametru; część specyfikacji; ciało klasy

Wywołanie klasy generuje nową postać klasy

generator postaci : : = new identyfikator klasy

część bieżącego parametru

Kiedy następuje generowanie postaci, zmiennej będącej rodzajem odsyłacza, przydzielana jest wartość, tak że może ona być odniesiona do postaci.

2. Konkatenacja /łańcuchowanie/ programu

To pojęcie umożliwia połączenie dwu klas A i B w nową klasę. Połączona klasa B jest generowana przez

A class B /b1, b2, .../; .... Specyfikacja b;

begin ..... "atrybuty" B....; .... "czynności" B

... end

jeśli A jest składnikiem kombinacji.

Za pomocą tej kombinacji złożone pojęcia mogą być budowane z pojęć prostych i może być generowana hierarchia pojęć.

3. Hierarchie pojęć

Odpowiednio do poszczególnych kroków abstrakcji, warstwy programu mogą być budowane jako szeregi deklaracji klas

```
        class  C1; .... ;
C1      class  C2; .....;
        ..........
Cn - 1  class  Cn; .......;
```

W tym przykładzie warstwa programu jest zawsze budowana na innej, bezpośrednio ją poprzedzającej. Klasa C_1 definiuje maszynę bazową.

Sam program jest opisany przez

```
Cn begin ..... end
```

Pojęcia klas postaci, konkatenacji programu i hierarchii pojęć Dijkstra kojarzy całkowicie w koncepcji maszyny wirtualnej.

B. MODUŁOWE ZASTOSOWANIA W PL/1

Jako język zorientowany na procedury PL/1 używa pojęć ALGOLu - pojęcia bloku i pojęcia procedury - jednak w nieco zmodyfikowanej postaci. W pojęciu bloku zasadnicza różnica polega na tym, że blok "begin" w PL/1 posiada oznaczenie, tak że skok do oznaczonego bloku można wykonać z punktu programu, gdzie oznaczenie jest znane. PL/1 umożliwia również zdefiniowanie struktur danych, które nie są związane z ograniczonym życiem bloku.

W PL/1 odróżnia się atrybuty pola i atrybuty pamięci nazwy. Atrybuty pola określają pole instrukcji gdzie nazwa jest ważna, podczas gdy atrybuty pamięci określają dynamiczny okres wykonywania programu, podczas którego ta nazwa istnieje. Podstawową cechą jest to, że atrybuty pola i atrybuty pamięci są od siebie niezależne.

Atrybut pola nazwy może być wewnętrzny lub zewnętrzny. Nazwa z atrybutem wewnętrznym jest, tak jak i w ALGOLu, znana tylko w ramach bloku, w którym została wyraźnie zdefiniowana. Nazwa z zewnętrznym atrybutem posiada to samo znaczenie we wszystkich blokach, gdzie została zdefiniowana jako zewnętrzna i może być używana do komunikacji między blokami /w ten sposób jak wspólne pole w FORTRANie/. Wynikiem tego jest pojawienie się tych samych problemów jak w przypadku zewnętrznych odsyłaczy w blokach ALGOLu. Uboczne skutki, niezgodne z zasadami programowania strukturalnego, mogą być spowodowane przez nazwy z atrybutem zewnętrznym.

Atrybut pamięci nazwy może być: statyczny /static/, automatyczny /automatic/ lub sterowany /controlled/. Miejsce zapamiętania nazwy z atrybutem "statyczny" pozostaje w użyciu, aż wykonanie programu, w którym to zostało zadeklarowane zostanie zakończone. Tak jak w ALGOLu, miejsce w pamięci na nazwy z atrybutem "automatyczny" jest generowane, gdy następuje wejście do bloku, w którym dokonano deklaracji i jest niszczone, gdy następuje opuszczenie bloku. Atrybut "sterowany" umożliwia dynamiczny przydział miejsca w pamięci dla struktur danych, na czas wykonania. Nazwa z atrybutem "sterowany" jest znana tylko wtedy, gdy następuje wejście do bloku, w którym nastąpiła deklaracja; miejsce w pamięci jest jej przydzielane tylko przez rozkaz "allocate" /przydziel/. Rozkaz "free" /wolny/ niweluje przydział miejsca w pamięci. Dynamiczny przydział miejsca w pamięci może też być uzupełniony przez mechanizm wskaźnika /pointer mechanism/.

W odniesieniu do problemu atrybutów pola i pamięci nazwy, kiedy ma się do czynienia z prostym przypadkiem tworzenia struktury przepływu sterowania, programista w PL/1 staje przed zbyt

dużą ilością możliwości przydziału danych zadeklarowanym postaciom danych. Dla zapewnienia modułowości systemu programowego, konieczne jest przestrzeganie zasady, że:

, dane między modułami powinny być wymieniane tylko przez mechanizm formalnego parametru procedury,

. wszystkie postacie danych użytych w ramach procedury powinny być deklarowane jako "internal automatic".

Pojęcie procedury stosowane w PL/1 różni się zasadniczo od ALGOLu, dlatego można tutaj zdefiniować szereg wejść dla procedury i wołać procedurę asynchronicznie. Oznaczenie /marker/ związane z każdym pojawieniem się procedury identyfikuje podstawowe wejście do procedury. Jednak mogą być definiowane inne wejścia przez oznaczoną instrukcję, a bieżące parametry dla różnorodnych wejść, mogą być definiowane w każdym punkcie procedury. Do procedury można wejść przez którekolwiek z jej wejść poprzez wołanie oznaczenia zdefiniowane przez instrukcję wejścia; mechanizm ten oznacza, że wykonanie określonej części procedury może być uzależnione od określonych warunków modułu wołającego. Jednak pociąga to za sobą konieczność przyjęcia przez moduł wołający założeń dotyczących wewnętrznej struktury modułu wołanego, co może prowadzić do niepotrzebnego rozdzielania założeń. Co więcej, w przypadku wielu wejść procedury, konieczne jest zapewnienie, że wymagane dane są przesyłane tylko przez bieżące parametry.

Dla wołania procedury w sposób asynchroniczny PL/1 również przedstawia dodatkową koncepcję w modularnym wdrażaniu - koncepcję "TASK". W dyskusji dotyczącej tworzenia hierarchicznej struktury systemu programowego podkreślono, że chociaż cykliczne powiązania informacji na tym samym poziomie hierarchicznym są niezgodne z zasadą tworzenia struktur hierarchicznych, to jednak nie zawsze można tego uniknąć. W każdym razie, bardzo istotne jest utrzymywanie ilości modułów w cyklu na poziomie minimalnym. Cykliczne powiązania między modułami pojawiają się, kiedy dwa moduły mają przygotować wynik we wzajemnej współpracy. PL/1 oferuje możliwość specyfikacji równoległego przetwa-

rzania dwu modułów. Jednak takie równoległe przetwarzanie z góry zakłada istnienie dwu interpretatorów lub dwu jednostek centralnych. W przeciwnym razie wykonanie dwu modułów jest wzajemnie sekwencyjne. Moduł PL/1, który może być wykonywany równolegle z innym modułem nosi nazwę zadania /task/. Szereg instrukcji, w których zawarte jest zadanie wołania nosi nazwę zadania przydzielającego /assigning task/, podczas gdy szereg instrukcji zadania wołanego nazywa się zadaniem przydzielonym /assigned task/. Zadanie jest traktowane jak procedura standardowa, gdy nie został podany jeden z następujących atrybutów: TASK /nazwa zadania/, zdarzenie /nazwa zdarzenia/ i priorytet /M/. Nazwa zadania wskazuje specjalną nazwę zadania. Priorytet wskazuje pierwszeństwo zadania w stosunku do innych zadań. Przydzielone zadanie może posiadać nazwę zdarzenia w powiązaniu z jego zakończeniem. Ta nazwa zdarzenia może być używana w zadaniu przydzielonym dla synchronizacji dalszego wykonania rozkazów z zakończeniem zadania przydzielonego za pomocą instrukcji "ozekaj" /wait/. Jednak o ile kolejność wykonywania rozkazów w programach współpracujących jest ustalona, o tyle kolejność wykonywania rozkazów dla dwu zadań jest zupełnie nieokreślona. To zachowanie się opiera się na założeniu, że wynik przetwarzania dwu zadań "sklejonych" nie zależy od kolejności wykonywania ich rozkazów.

Jeśli zadanie generuje przydzielone zadanie B, to otoczenie zadania A, które jest ważne w generowaniu B, jest związane z obydwoma zadaniami. Zadanie, które posiada dostęp do nielokalnego otoczenia tylko na zasadzie czytania nie posiada żadnych efektów ubocznych. Jeżeli zarówno zadanie przydzielające jak i zadanie przydzielone nie mają "skutków ubocznych", wynik równoległego wykonywania obydwu zadań zawsze jest niezależny od względnej, chronologicznej kolejności, w jakiej rozkazy zadań są wykonywane. Jeżeli jedno lub obydwa zadania równoległe, mają "skutki uboczne", to organizacja wykonania może mieć wpływ na ostateczny stan otoczenia lub wynik współdziałania. Skutków ubocznych tego rodzaju należy unikać za wszelką cenę.

C. MODUŁOWE ZASTOSOWANIA W FORTRANIE

Prosta struktura składni FORTRANu oferuje zalety w zastosowaniu języka, a kompajlery FORTRANu zwykle tworzą kod bardziej wydajny niż pozostałe, problemowo zorientowane języki. Jednak FORTRAN nie jest odpowiedni dla przygotowania programów strukturalnych:

- FORTRAN nie posiada struktury blokowej; jedyny sposób połączenia szeregu modułów dla stworzenia modułu wyższego, jest w kategoriach podprogramów. Podprogramy są sztywno związane z programem głównym i tworzą wraz z nim kompletny program FORTRANu, który nie może być połączony z innym programem FORTRANu.
- Nie ma możliwości powtórnych wejść do podprogramów; nie można opracować jednego podprogramu, który sam się przywołuje, lub dwu podprogramów przywołujących się nawzajem.
- Brak struktury blokowej zmusza do częstszego używania instrukcji GO-TO /idź - do/ dla pominięcia sekcji programu i w ten sposób symuluje bloki. Instrukcje GO-TO przeszkadzają w zrozumieniu dynamicznego zachowania się programu.

Prostota struktury składniowej umożliwia łatwe wdrożenie FORTRANu. Program FORTRANu składa się zwykle z trzech składników logicznych:

- programu głównego,
- pewnej ilości podprogramów,
- pewnej ilości WSPÓLNYCH/COMMON/ pól danych.

Każdy program główny i podprogram tworzą niezależną jednostkę programową składającą się z trzech pól: pola tekstu programu, pola danych lokalnych jednostki programowej i pola roboczego.

Niezależnie od jednostek programowych występuje jedno lub więcej wspólnych pól danych. W polu dla danych lokalnych znajdują się postaci danych, które należą do omawianej jednostki programowej. Dostęp do nich może być osiągnięty podczas wykonywania jednostki programowej i są one przesyłane jedynie za pomocą mechanizmu parametru programu wołania. W polu wspólnych

danych znajdują się dane, do których dostęp jest możliwy podczas wykonywania instrukcji dla szeregu podprogramów.

Wewnętrzna organizacja programu umożliwia ustalenie wymogu miejsc w pamięci i czasu przemieszczenia oraz połączenia jednostek programowych i wspólnych pól danych, w łatwy sposób, po prostu przez podstawowe rejestry adresowe. Posiada to jednak tę zasadniczą wadę, że tekst programu i dane lokalne są sztywno połączone, co powoduje niemożliwość ponownego wejścia do nich.

Niezgodność nazw może się pojawić, gdy luźno zdefiniowane pole wspólnych danych jest używane dla różnych celów w dwu podprogramach, lub gdy postaci danych w podprogramach, które zostały zdefiniowane jako wspólne, posiadają tę samą nazwę. Odnosi się to również do nazw podprogramów. Pole wspólnych danych umożliwia również komunikację między modułami; tu również "skutki uboczne" mogą prowadzić do niepożądanych wyników.

D. MODUŁOWE ZASTOSOWANIA W COBOLU

Środkami lingwistycznymi w COBOLu dla stosowania modułów są głównie paragrafy i rozdziały, które mogą być połączone razem w następujące trzy sposoby:

- sekwencyjne uporządkowanie w szeregi,
- skok i wyskok z GO-TO,
- wołanie przez "wykonaj" /PERFORM/.

W pierwszym rzędzie, paragraf i rozdział po prostu umożliwiają statyczny podział programu na sekcje, na jednym poziomie. Ich związek z innymi sekcjami programu może być rozpoznany tylko w przypadku, jeśli programista przyjął wyraźne mierniki, aby połączenia uczynić przejrzystymi.

Instrukcja "wykonaj" jest jedynym sposobem wbudowania składnika programu w bardziej ogólny składnik. To wbudowanie osiąga się na zasadzie połączenia dynamicznego, nawet jeżeli składniki są używane tylko w jednym miejscu. W miejscu, gdzie to jest używane, procedura opracowywana w ten sposób jest zastąpiona przez

przywołanie; tekst procedury jest usunięty. Procedura nie posiada żadnego rodzaju danych lokalnych, wszystkie używane elementy danych są zadeklarowane w sekcji danych, dla całego programu cobolowego. W konsekwencji, jakikolwiek przydział wartości postaci danych w procedurze powoduje "efekt uboczny".

Istnieje możliwość używania danych lokalnych i przesyłania parametrów bieżących tylko wtedy, gdy moduł COBOLu jest stosowany jako oddzielnie skompilowany podprogram; jednak tę symulację koncepcji procedury osiąga się kosztem znacznego czasu wykonania i zajęcia dużo miejsca w pamięci operacyjnej.

Przesyłanie parametrów może być symulowane przez przydzielenie obszarów wejścia i wyjścia podprogramom oraz przesyłania danych do, i z, tych obszarów.

Podobnie jak w FORTRANie, prostota wdrożenia w COBOLu również powoduje pojawienie się kilku właściwości, które nie są zgodne z zasadami programowania strukturalnego. Na przykład instrukcja "zmień" /ALTER/ powinna być w związku z programowaniem strukturalnym całkowicie zabroniona. Jeśli instrukcja odgałęzienia zmienia się dynamicznie, przestaje istnieć zapewnienie czytelności programu.

Szkodliwy charakter instrukcji "idź do" /GO-TO/ był punktem wyjściowym dyskusji na temat programowania strukturalnego. W odniesieniu do COBOLu nie wydaje się dobrym pomysłem uznanie tej instrukcji za nieodpowiednią; należy raczej oddzielić jej stronę użyteczną od szkodliwej: jako instrukcje skoku w programie generalnie maskują związki logiczne, wyraźnie zabroniony jest skok "idź do", z jednego składnika programu do innego. W konsekwencji, użycie rozkazu "idź do" jest dozwolone tylko w ramach składników programu, a nawet i wówczas, tylko gdy jest to konieczne. Floyd uważa za pożyteczne następujące instrukcje "idź do":

. służące stworzeniu warunkowej pętli,

. służące sformułowaniu różnicy zdarzenia,

. prowadzące ze struktury do końca składnika programu,

. zastosowane do zajęcia się sytuacją wyjątkową.

E. NADAWANIE STRUKTURY DANYM

Dla zapewnienia modułowości systemów programowych, na poziomie lingwistycznym muszą być spełnione dwa wymogi:

- środki dla nadania dynamicznej struktury tekstowi programu,
- środki do stworzenia struktury użytych danych, zgodnie z modularną strukturą programu.

Cel wymagań modułowości polega na tym, że każda postać danych stanowiących integralną część komunikacji między modułami, może być traktowana jako bieżący wejściowy lub wyjściowy parametr modułów. Istnieją trzy podstawowe warunki, aby dany poziom lingwistyczny spełniał wymogi modułowości:

- każda struktura danych powinna mieć możliwość pojawienia się jako składnik innej struktury danych,
- ponieważ każda postać danych, do której moduł ma dostęp, powinna stanowić integralną część samego modułu, należy stworzyć odpowiednie warunki na poziomie lingwistycznym, aby umożliwić przeniesienie każdej struktury danych, jako parametru bieżącego do, lub z, modułu programu,
- moduł programu powinien być w stanie budować struktury danych o dowolnym stopniu złożoności.

Zgodnie z hierarchiczną strukturą systemu programowego, opracowywanego przez analizę prowadzoną metodą "krok-po-kroku", powinna istnieć możliwość wdrożenia struktur danych odpowiadających pojedyńczym maszynom abstrakcyjnym i określenia wielopoziomowych struktur danych, w których punkty węzłowe struktury danych na poziomie A_i służą jako nagłówki struktury danych na poziomie A_{i+1}. Powinien też istnieć mechanizm sprawdzający, który w ten sam sposób jak rozkaz "idź do", zapobiega nieograniczonym odgałęzieniom, tak że zachowana zostaje prostota struktury używanych danych.

Obecnie żaden z szeroko używanych języków programowania nie oferuje takich możliwości. Jest to zasadniczo wynik przeplatania semantycznych koncepcji struktury danych z pojęciami

wdrażania. Wynika to z sekwencyjnej struktury pamięci operacyjnej systemu komputerowego /na której, w ostatecznej analizie, opisywane są wszystkie struktury danych/ i jego ograniczeń odnośnie rozmiaru tych struktur danych. Struktura sekwencyjna pamięci operacyjnej ukrywa ograniczenia programowania modułowego tak dalece, że proste rozkazy organizacji pamięci nie pozwalają na żadne zmiany w strukturze danych bez objęcia reprezentacji innych struktur danych, na przykład, gdy struktura danych jest rozszerzona. Jeśli w obszarze pamięci występuje więcej niż jedna struktura danych, z powodu ograniczeń pamięci operacyjnej, znaczenie adresów będzie się zmieniać w toku wykonywania programu. Dla ustalenia, które postacie danych mają zajmować wspólne pole, moduł musi posiadać informacje o wewnętrznej strukturze innego modułu co oznacza naruszenie zasady modułowości. Pomimo że zastosowanie pamięci wirtualnej posiada określone zalety, programiście trudno jest opisać strukturę danych w pamięci w taki sposób, aby były one przesyłane razem między poziomami pamięci.

Dla zapewnienia właściwego podejścia do tworzenia struktury danych dla programowania strukturalnego, istniejące języki programowania, będą musiały być rozszerzone lub będzie musiał być wprowadzony nowy język, przez opracowanie stosownego translatora lub interpretatora dla obszaru specjalnych problemów.

Omówimy teraz pokrótce możliwości oferowane przez najszerzej używane języki programowania dla tworzenia struktur danych, w kontekście programowania strukturalnego. /Nie ma potrzeby głębszego wchodzenia w szczegóły odnośnie FORTRANu, ponieważ "szeregi", których granice, po alokacji miejsca w pamięci, są stałe i których wymiary są ustalone przez tekst programu, dają jedynie możliwości dla tworzenia struktur danych/.

F. STRUKTURY DANYCH W MODULARNYCH PROGRAMACH PL/1

PL/1 dysponuje następującymi rodzajami danych dla reprezentacji i przetwarzania danych strukturalnych: szereg, /array/;

struktura /structure/; zmienna umiejscowiona i wskaźnik /based variable and pointer/.

1. Szereg

W stosunku do szeregów odnoszą się te same ograniczenia PL/1 co i w FORTRANie: rozmiar szeregu musi być zdefiniowany, a granice szeregu są niezmienne po przydzielaniu mu miejsca w pamięci. Wszystkie elementy w szeregu muszą stanowić tenssam rodzaj danych. Te ograniczenia umożliwiają przydział szeregu do pola o stałym adresie i z tego powodu pozostawienie niezmiennej struktury danych w czasie cyklu programu.

2. Struktura

Dane typu "struktura" też nie spełniają wymogów programowania strukturalnego. Podczas wykonywania programu nie można zrobić arbitralnej postaci danych typu "struktura" składnikiem innej postaci danych tego samego typu, ponieważ ostateczna postać struktury musi być ustalona zanim wartość może być przydzielona jednemu z jej składników.

3. Zmienna umiejscowiona i wskaźnik

Chociaż wskaźnik typu "dana" w połączeniu z szeregami i strukturami, które zostały zadeklarowane jako umiejscowione /based/, oferuje bardzo ogólny sposób budowania struktur danych, to jednak wymagania programowania strukturalnego nie są spełnione. Wartość wskaźnika nie może być uważana za odsyłacz do struktury danych, ponieważ PL/1 nie umożliwia identyfikacji ilości elementów w strukturze wyznaczonej przez wartość wskaźnika. Nie ma też żadnej możliwości, aby struktura której składniki są połączone przez wskaźniki, mogła być składnikiem innej struktury. Co więcej nie ma żadnej gwarancji, że element wyznaczony jako wskaźnik reprezentuje typ danych, do którego programista chciałby się odwołać. Elementy w strukturze są po prostu wymazywane przez wolną instrukcję i element oddzielony od połączonej struk-

tury przez zmianę wartości wskaźnika istnieje do czasu odebrania mu miejsca w pamięci.

Stąd programista jest zmuszony do opracowywania własnych konwencji dla tworzenia struktur danych. Dlatego zastosowanie połączonych struktur dla komunikacji między niezależnymi modułami nie posiada istotnych zalet.

G. STRUKTURY DANYCH W MODULARNYCH PROGRAMACH ALGOLU 68

W ALGOLu 68 występują dwa podstawowe typy danych tworzące strukturę postaci danych: wartość wielokrotna /multiple value/ i struktura /structure/. Podobnie jak szeregi w PL/1, wartości wielokrotne nie dają żadnej właściwej podstawy dla programowania strukturalnego. Deklaracja struktury typu danych zakłada, że każda wartość typu danych stanowi postać danych posiadającą stałą liczbę składników stanowiących postacie typu danych i może być identyfikowana za pomocą selektorów. Przez wprowadzenie szeregu deklaracji typu danych może być zdefiniowana klasa postaci danych charakteryzujących się strukturą drzewa. Każdy węzeł tego drzewa posiada odpowiedni typ danych i stanowi początek ustalonej ilości brzegów /edges/, z których każdy ukrywa selektor określony w deklaracjach typu danych. Ponieważ dopuszczalne są rekursywne deklaracje typu danych, postacie specjalnego typu danych mogą posiadać nieograniczoną głębokość. Jednak, ponieważ każda struktura tego rodzaju musi być zadeklarowana, nie ma możliwości substytucji struktury arbitralnej. Nie można napisać procedury ALGOLu, która otrzymuje strukturę danych z modułu programu i przesyła ją do innego modułu nie wiedząc wystarczająco dużo o strukturze danych, aby określić jej rodzaj w wskazanym module. Moduł programowy nie może też łączyć razem arbitralnych struktur danych, ponieważ skończona ilość deklaracji typów jest niewystarczająca do opisania pełnej klasy postaci ALGOLu.

Konwencje dla opisania struktur danych w ALGOLu 68 są odpowiednie dla budowania złożonych struktur danych. Rozkazy w ALGOLu

68 zajmujące się sprawą danych, generalnie są lepsze od PL/1, chociaż warunek, że typ każdej zmiennej musi być wyraźnie określony, stanowi przykre ograniczenie.

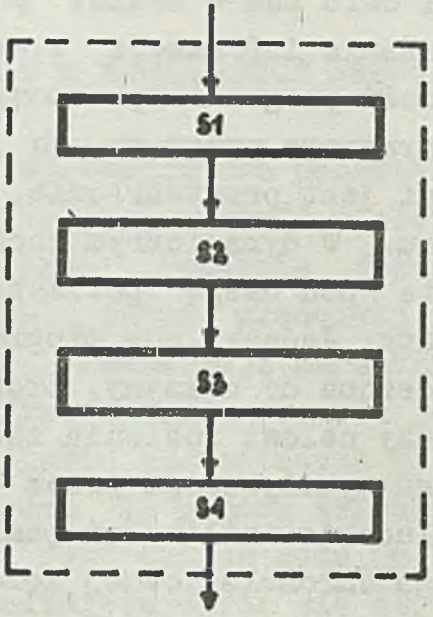
H. KODOWANIE STRUKTURALNE

Program nie stanowi celu sam w sobie; pociąga on za sobą szereg operacji przetwarzania informacji z myślą o uzyskaniu wyniku. Chociaż działalność programisty kończy się z momentem uzyskania przez niego poprawnego programu, to jednak rzeczywistym sensem jego czynności jest przetwarzanie, które odbywa się pod kontrolą jego programu. W dynamicznym zachowaniu się tego procesu muszą być wzięte pod uwagę potrzebne specyfikacje i osiągnięcie żądanego efektu. Jednak, gdy program jest zakończony, wykonanie jest przenoszone do maszyny. Zrozumienie statycznego tekstu programu służy celowi robienia założeń dotyczących dynamicznego cyklu procesu wykonywanego przez maszynę. Dijkstra uważa, że intelektualne ludzkie możliwości pamiętania dynamicznego cyklu procesu w jego chronologicznej kolejności są raczej nierozwinięte. W konsekwencji dystans między statycznym tekstem programu i dynamicznym procesem powinien być jak najmniejszy; łatwość i niezawodność założeń tyczących się dynamicznych cykli procesu w dużej mierze zależy od jakości przepływu sterowania w programie. Dla osiągnięcia prostego przepływu sterowania sugeruje się stosowanie struktur pokazanych na rysunkach 6, 7 i 8.

Wszystkie pokazane struktury posiadają jedną właściwość istotną dla jasnego rozumienia programu - występowanie pojedynczego, logicznego wejścia i pojedynczego logicznego wyjścia.

Bohm i Jacopini udowodnili, że w zasadzie każdy, dowolny program może być napisany tylko przy użyciu tych struktur. Wówczas kod programu wynikowego jasno i prosto odzwierciedla związek między statycznym i dynamicznym zachowaniem się programu. Zasoby umysłowe potrzebne do zrozumienia takiego programu muszą być proporcjonalne do jego długości. Podstawową konsekwencją wynikającą z pracy Bohma i Jacopiniego jest to, że zasadniczo

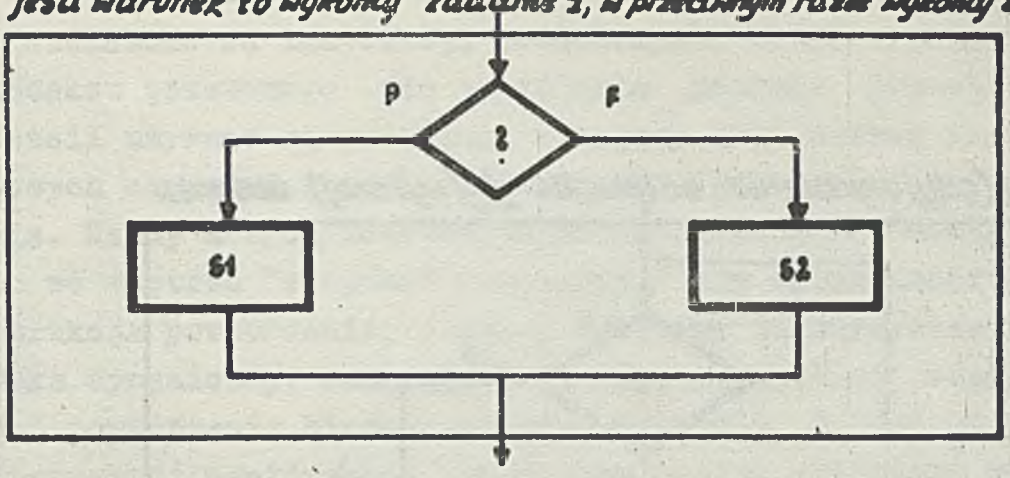
można uniknąć polecenia "idź do". Użycie "idź do", oznacza, że program nie może być czytany w naturalny, sekwencyjny sposób i że z tekstu programu nie można wyciągnąć wniosków co do postępu przebiegu biegnącego pod kontrolą programu.



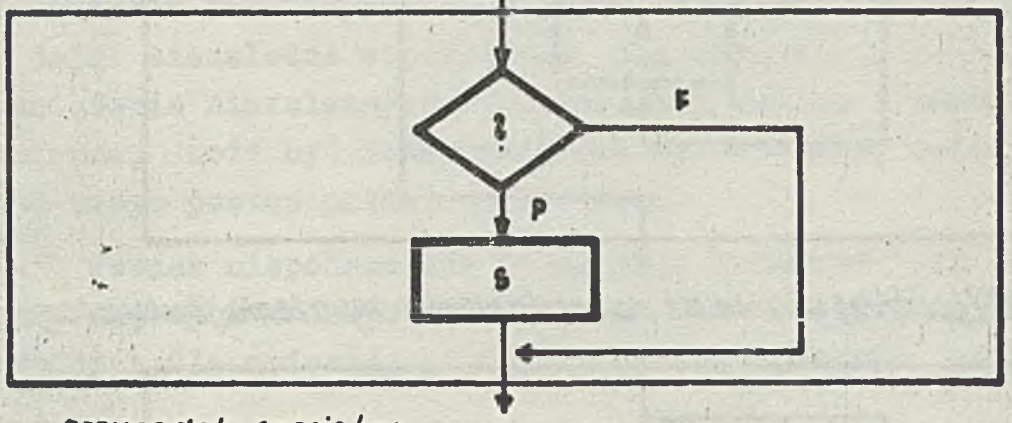
Rys. 6 INSTRUKCJA POŁĄCZONA (LICZENIE SEKWENCYJNE)

Jeśli program składa się wyłącznie z instrukcji połączonych wystarczy wskazać miejsce między dwoma kolejnymi zdaniami, aby opisać postęp przebiegu procesu dynamicznego. Chronologiczna kolejność podczynności w procesie rzeczywistym odpowiada kolejności instrukcji w tekście programu. Dijkstra mówi o tym wskaźniku - który wskazuje na miejsce w tekście programu określające postęp w przebiegu procesu - jako o indeksie tekstowym /textual index/.

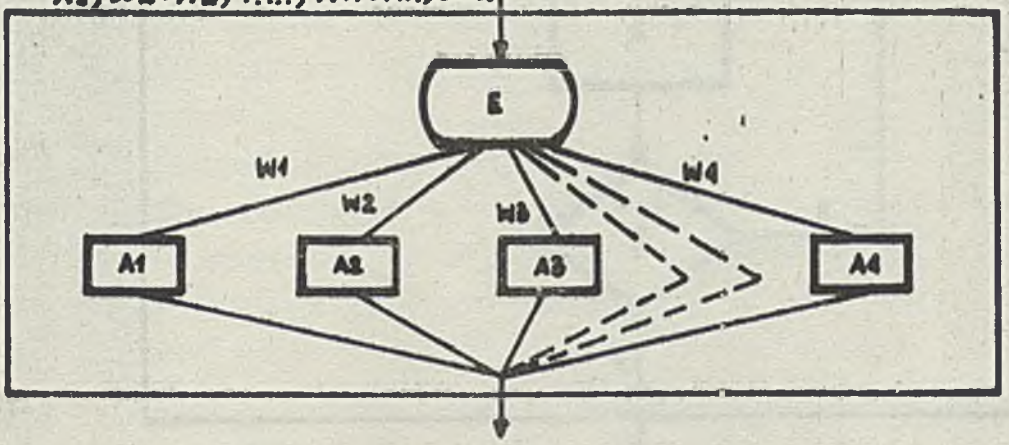
jeśli warunek to wykonuj zadanie 1, w przeciwnym razie wykonaj zadanie 2



przypadek specjalny
jeśli (if) warunek wówczas (then) zdanie



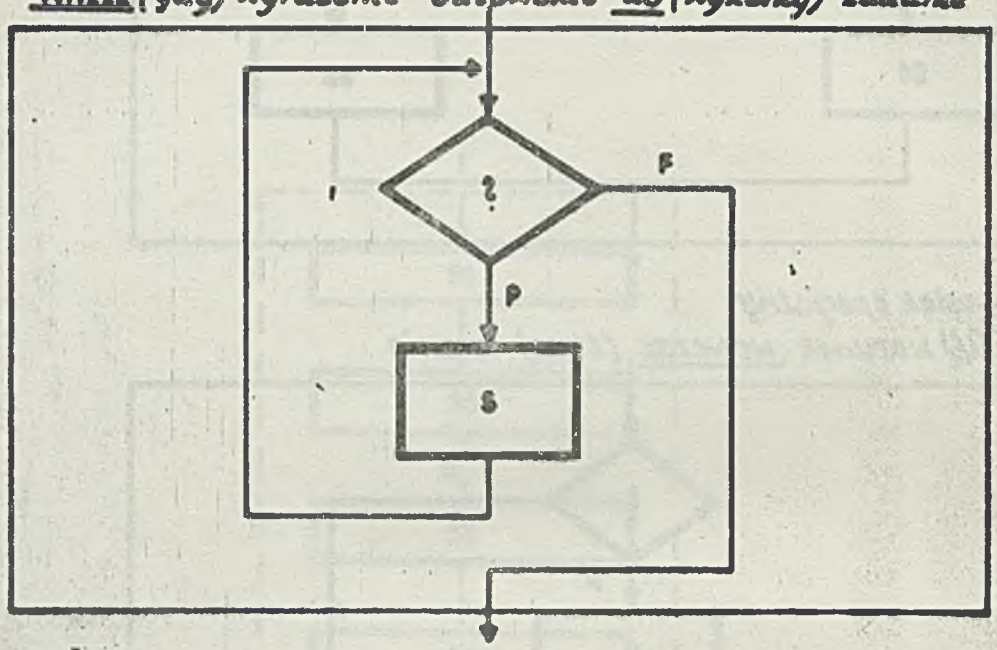
przypadek specjalny
wyrażenie przypadku (case expression of) W1 :
A1; W2 : A2; .....; Wn : An; end



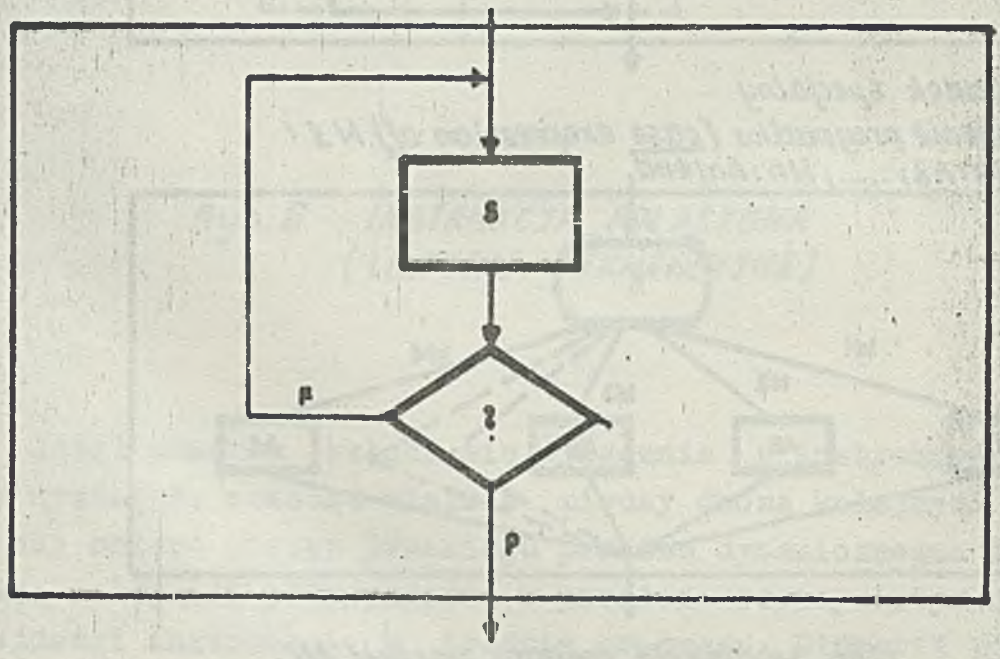
jeśli wyrażenie posiada wartość Wi,
(then) wówczas wykonywane jest Ai

Rys. 7 INSTRUKCJA WARUNKOWA

while (gdy) wyrażenie bulowskie **do** (wykonaj) zadanie



repeat (powtórz) zdanie aż do wystąpienia wyrażenia bulowskiego



Rys. 8 INSTRUKCJE POWTÓRZENIA

Jeżeli instrukcje warunkowe są włączone do tekstu programu, niezależnie od instrukcji połączonych, nadal obowiązuje ważność indeksu tekstowego dla określenia postępu przebiegu procesu. Jeżeli używane są procedury, wymagany jest szereg indeksów tekstowych odpowiadających dynamicznej głębokości procedury wołania. Każdy z tych tekstowych indeksów wskazuje dokładnie miejsce we wnętrzu "korpusu" procedury. Gdy wprowadzona zostaje instrukcja powtórzenia, indeksy tekstowe są niewystarczające. Indeks dynamiczny, wskazujący liczbę porządkową odpowiadającego jej powtórzenia bieżącego, jest związany z każdym wejściem do instrukcji powtórzenia. Stąd więc postęp przebiegu procesu zawsze może być jasno opisany przez szereg kombinacji dynamicznych i indeksów tekstowych. Indeksy te są poza kontrolą programisty i dają niezależne współrzędne dla opisania procesu. Potrzeba posiadania niezależnych współrzędnych wynika z tego, że wartość zmiennej może być interpretowana tylko wtedy, jeśli weźmie się pod uwagę postęp przebiegu procesu.

Jednak niepohamowane stosowanie rozkazów "idź do" ma bezpośredni wpływ - bardzo trudne staje się określenie liczby współrzędnych dla opisania postępu przebiegu procesu, o którym mowa.

IV. OPROGRAMOWANIE WSPIERAJACE PROGRAMOWANIE STRUKTURALNE

Dyskusja na temat programowania strukturalnego pokazała, że zasady inżynierii oprogramowania mogą być pomocne w tworzeniu nowego oprogramowania. Czego jeszcze nie można użyć to maszyny dla programowania strukturalnego, ogólnego modelu i strukturalnej koncepcji, które postawią programistę i innych ludzi związanych z przetwarzaniem danych, w pozycji umożliwiającej wypowiedzenie się w sposób zgodny z pojęciami programowania strukturalnego. Zrobiono już kilka małych kroków w kierunku machiny programowania strukturalnego. Interesującym osiągnięciem w tym temacie jest np. pakiet oprogramowania COLUMBUS.

Metoda programowania wspierana przez ten pakiet oprogramowania, za punkt wyjścia przyjmuje wykresy Nassi i Schneidermanna, które dopuszczają dwuwymiarową reprezentację przepływu sterowania w programie. Zaletę tej techniki stanowi to, że może ona być używana w opracowywaniu programu przez jego precyzowanie sposobem "krok-po-kroku" oraz to, że nie dopuszcza ona do niewłaściwej struktury logiki sterowania w programach. Programiści piszą programy stosując tę technikę i dostarczają program do systemu komputerowego w sposób liniowy, przez urządzenie końcowe /terminal/. Ten program wejściowy może być wówczas przetworzony przez jakąś ilość programów użytkowych /utility/, zasadniczo niezależnych od wybranego języka programowania. Te programy użytkowe przygotowują różne odsyłacze informacji i odtwarzają wykres programu Nassi - Schneidermanna na ekranopisie lub drukarce. Dla osiągnięcia wysokiego stopnia niezależności od języka programowania, programista używa szeregu prymitywnych rozkazów dla opisania logiki sterowania. Rozkazy te są wspólne dla wszystkich języków. Liniowa reprezentacja tekstu programu jest teraz transferowana przez zależny od języka preprocesor na czysty język programowy. Zapewniona jest cała gama metod dla opracowania lub śledzenia dynamicznego wykonania programu. Plan na najbliższą przyszłość to dalsze rozszerzenie koncepcji do-

starczonych przez COLUMBUSa. Podstawowym zamierzeniem jest poprawienie tworzenia odsyłaczy tak,aby można było wskazać, gdzie w ckreślonym programie, określona postać danych, jednostka danych lub określona procedura, są używane.

Inną koncepcją pozostającą do opracowania jest "porządek hezoniczny" /"hezonical order"/ programu. Wykorzystuje się tu fakt, że istniejące języki programowania dają programiście jakąś elastyczność w tym gdzie i w jakiej kolejności umieszcza on dane i deklaracje procedur. W dużych systemach programowych ta elastyczność może stanowić wadę, ponieważ brak jest środków zapewniających znajomość, gdzie program musi być kontrolowany dla sprawdzenia, czy występuje, lub nie odpowiednia deklaracja."Porządek kanoniczny" /"canonical order"/ zapewnia, że deklaracja danych może być znaleziona na poziomie, gdzie logicznie powinna być umiejscowiona. W wyniku tego poprawia się czytelność programu i łatwiejsze jest oddzielenie w nim poszczególnych bloków.

V. TECHNIKI DOKUMENTACYJNE

A. WYKRESY NASSI - SCHNEIDERMANNA

Jakość systemu programowego w dużej mierze zależy od jakości projektowania. Standardową metodą projektowania w programowaniu konwencjonalnym jest dobrze znany schemat blokowy. W praktyce jednak, praca w oparciu o schematy blokowe napotyka stale na trudności związane z ich zrozumieniem, głównie z powodu, że:

- cel pojedynczych składników nie jest natychmiast oczywisty,
- przeplatają się elementy rozwiązania funkcjonalnego i technicznego,
- nie ma oczywistego podziału na składniki strukturalne, lub wyższe jednostki logiczne.

Co do dalszego rozwoju techniki budowania programów, to szczególnie ostatnia sprawa ma drastyczne wady.

Zgodnie z zasadami programowania strukturalnego, Nassi i Schneidermann opracowali technikę projektowania i reprezentacji dla programów, która skłania projektującego program do opracowywania rozwiązania funkcjonalnego rozpoczynając od małych, standardowych elementów strukturalnych. Są one używane dla określenia logiki sterowania bądź to przez ustawienie w szeregi, bądź jeden obok drugiego /dla odróżnienia zdarzeń/ lub też umieszczając je w gniazdach. W konsekwencji statyczne ustawienie bloków strukturalnych odzwierciedla ich funkcję w dynamicznym cyklu programu. Umieszczając bloki strukturalne jeden w drugim spełnia się podstawową zasadę programowania strukturalnego - projektowanie programu "od góry-w dół" poprzez szczegółową analizę "krok-po-kroku". Funkcja logiczna jest najpierw reprezentowana w postaci bloku, do którego dodawane są poprzez tę szczegółową krokową analizę, "wewnętrzne" bloki strukturalne, odczia-

ływujące na siebie przy wykonywaniu zadania. Gdy bloki strukturalne, do których się kolejno wchodzi,stają się zbyt małe, najbardziej wewnętrzny blok strukturalny otrzymuje nazwę i jest dalej opracowywany w postaci wykresu - patrz rysunek 9.

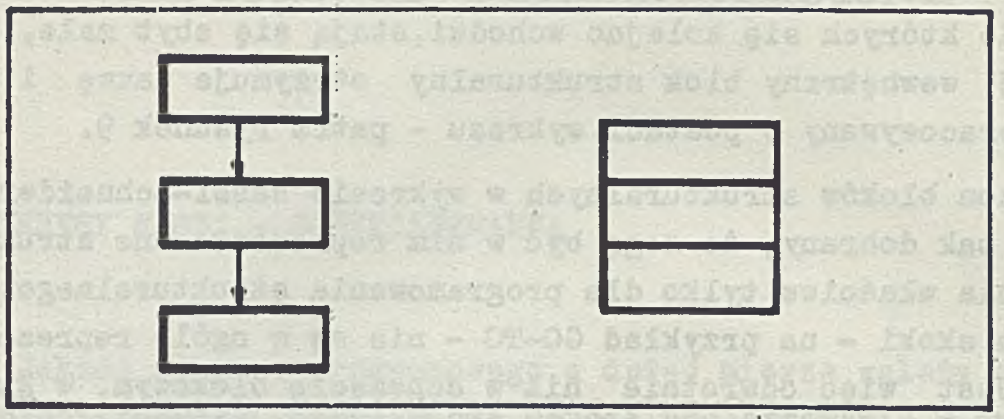
Trzon bloków strukturalnych w wykresie Nassi-Schneidermanna jest tak dobrany, że mogą być w nim reprezentowane struktury sterowania właściwe tylko dla programowania strukturalnego. Zabronione skoki - na przykład GO-TO - nie są w ogóle reprezentowane. Jest więc odwrotnie niż w schemacię blokowym, w którym strzałki wyznaczające przepływ sterowania mogą być prowadzone, bez żadnych ograniczeń, z jednego punktu na wykresie, do jakiegokolwiek innego punktu. Wszystkie bloki strukturalne, których można używać, posiadają następujące właściwości:

- każdy blok strukturalny ma wejście /górna krawędź/ i wyjście /dolna krawędź/,
- logika sterowania dynamicznego przebiega przez bloki strukturalne, a w konsekwencji przez cały program, od góry do dołu,
- każdy blok strukturalny określa wnętrze i to co jest na zewnątrz; inny blok strukturalny albo jest w nim zawarty, albo umieszczony na zewnątrz,
- każdy blok strukturalny stanowi więc zamkniętą jednostkę funkcjonalną, niezależnie od logicznego poziomu projektu programu, na którym jest on ustalony,
- blok strukturalny odpowiada wyłącznie blokom bezpośrednio sąsiadującym z jego górną i dolną krawędzią. Przesyła on sterowanie otrzymane od swego górnego sąsiada, tylko do sąsiada dolnego; nie ma żadnych związków między nim a blokami strukturalnymi po prawej i lewej stronie.

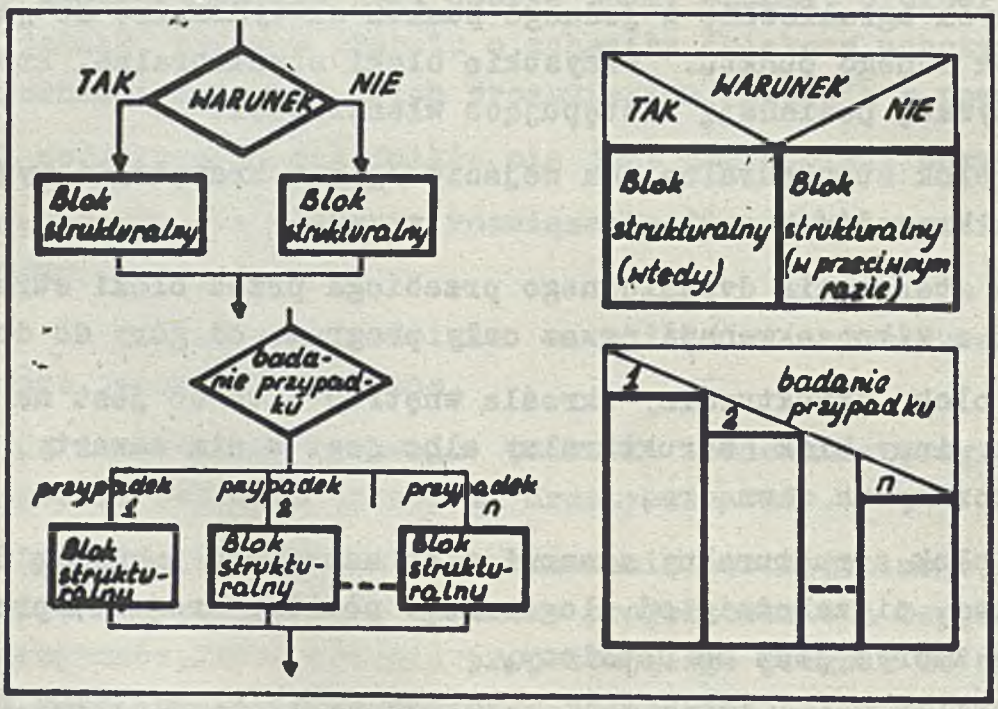
Planowanie programu wymaga żeby:

- bloki strukturalne, które mają być wykonywane sekwencyjnie były kolejno opisywane,
- bloki strukturalne wewnętrzne były wprowadzane bezpośrednio do bloku strukturalnego otaczającego. Alternatywnie blok wewnętrzny otrzymuje nazwę i będzie opracowywany dalej jako nowy, podporządkowany wykres.

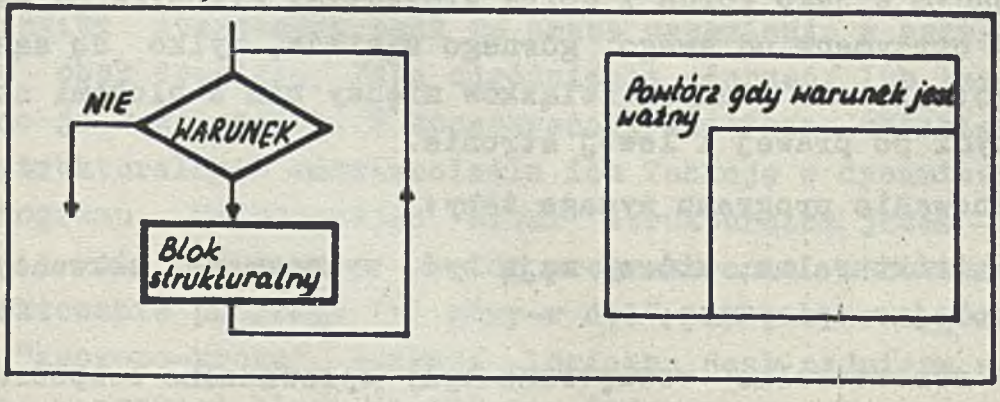
1. Instrukcja połączona



2. Instrukcja warunkowa



3. Instrukcja powtórzenia



Rys. 9

Wykresy Nassi-Schneidermanna są bardzo przydatne do reprezentowania przepływu sterowania opracowanego na zasadach programowania strukturalnego, chociaż nadal nie łatwo jest rozpoznać wzajemne oddziaływanie między poszczególnymi funkcjami systemu programowego i przepływem danych.

Z powodu tego ograniczenia organizacja funkcjonalna systemu programowego powinna być opisana przez wykresy Hipo[x/].Na tej podstawie struktura logiki sterowania jest opracowywana w postaci wykresów Nassi-Schneidermanna.

B. WYKRESY HIPO

Każda operacja przetwarzania danych może być opisana przez wejście, przetwarzanie i wyjście.Wykresy Hipo reprezentują cykl systemu programowego jako hierarchicznie uporządkowany system faz wejścia, przetwarzania i wyjścia. Można je stosować do projektowania, kodowania i utrzymywania systemu programowego.

Wykresy Hipo składają się z czterech logicznych części: części wejścia, części przetwarzania, części wyjścia i części dla rozszerzonego opisu kroków przetwarzania stworzonych z części przetwarzania. Wykres Hipo opisuje wyłącznie funkcję składnika programu i dynamiczny cykl składnika programu otrzymującego najwyższy priorytet. Cechą specjalną techniki Hipo jest hierarchiczna struktura reprezentacji.Na najwyższym poziomie abstrakcji występuje pojedynczy wykres Hipo, który opisuje funkcję całego programu. Podfunkcje są opisane w oddzielnych wykresach Hipo w następnym, niższym poziomie.

Przewaga wykresów Hipo nad schematami blokowymi wyraża się w tym, że:

- widoczny jest dokładny przepływ danych przez system programowy,

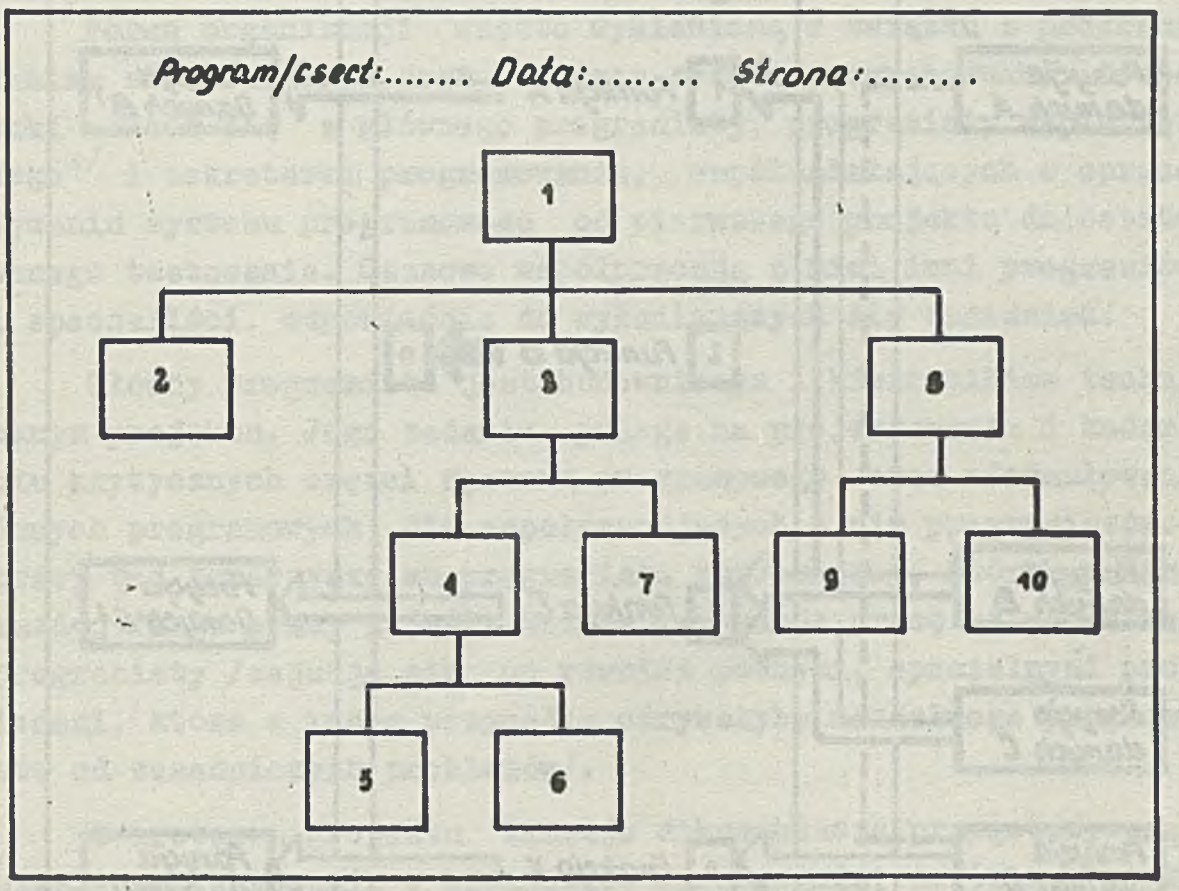
---

x/ Hipo - skrót od ang. Hierarchy and Input, Process, Output /Hierarchia plus Wejście, Przetwarzanie, Wyjście/ - przyp. tłum.

- każdy wykres Hipo reprezentuje funkcję, której podfunkcje muszą być opisane w części przetwarzania wykresu Hipo i których nie trzeba poszukiwać w przewlekłej analizie,
- zasada uporządkowania hierarchicznego umożliwia systematyczne i natychmiastowe umiejscowienie określonego składnika programu.

W praktyce okazało się korzystne uzupełnienie wykresów Hipo systemu programowego, wykresem hierarchii, dającym ogólny przegląd wzajemnego oddziaływania między poszczególnymi modułami programu. Przykład wykresu hierarchii pokazano na rysunku 10.

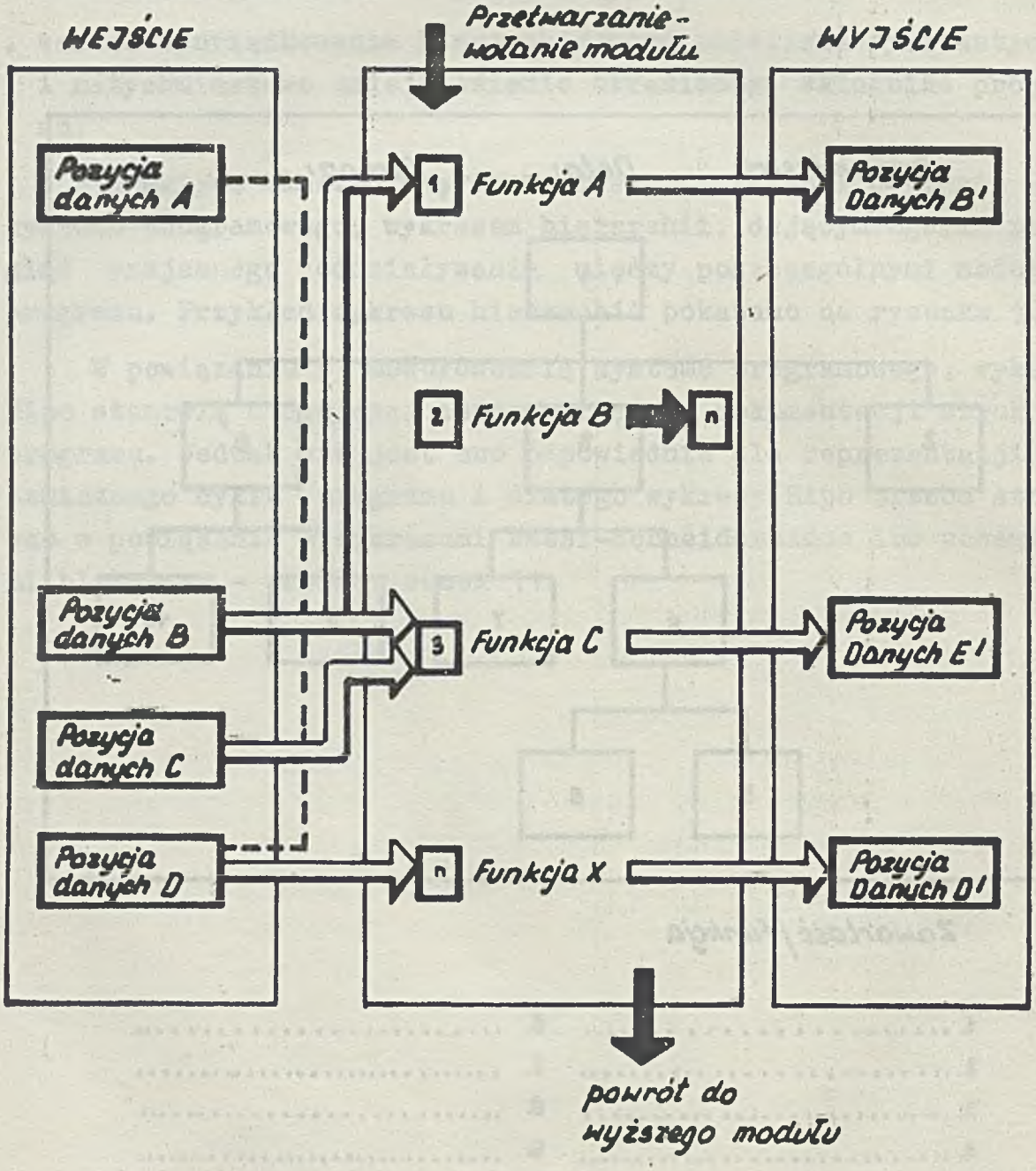
W powiązaniu z modułowością systemu programowego, wykresy Hipo stanowią nadzwyczaj cenne narzędzie dokumentacji struktury programu. Jednak nie jest ono odpowiednie dla reprezentacji dynamicznego cyklu programu i dlatego wykresy Hipo trzeba stosować w powiązaniu z wykresami Nassi-Schneidermanna lub schematami blokowymi - patrz rysunek 11.



Zawartość/funkcja

1. .....
2. .....
3. .....
4. .....
5. .....
6. .....
7. .....
8. .....
9. .....
10. .....

Rys. 10 WYKRES HIERARCHII


WEJŚCIE
Przetwarzanie - wołanie modułu
WYJŚCIE
Pozycja danych A
1
Funkcja A
Pozycja Danych B'
2
Funkcja B
n
Pozycja danych B
3
Funkcja C
Pozycja Danych E'
Pozycja danych C
Pozycja danych D
n
Funkcja X
Pozycja Danych D'
powrót do wyższego modułu
przepływ danych
odniesienie danych
przepływ sterowania

Rys. 11

VI. PROBLEMY WIODĄCEGO ZESPOŁU PROGRAMOWANIA

Formą organizacji często wymienioną w związku z programowaniem strukturalnym jest "wiodący zespół programowania". Zespół taki składa się z głównego programisty, programisty wspierającego[x/] i sekretarki programowania, współdziałających w opracowywaniu systemu programowego od pierwszego projektu do ostatecznego testowania. Czasowo współpracują z nimi inni programiści i specjaliści, odpowiednio do wyłaniających się zagadnień.

Główny programista jest budowniczym i kierownikiem technicznym projektu. Jego zadanie polega na projektowaniu i kodowaniu krytycznych części systemu programowego oraz sformułowaniu danych programowych dla współpracujących z nim programistów. W pracy tej towarzyszy mu programista wspierający, który pomaga w każdej fazie pracy i jeśli trzeba przejmuje pracę od naczelnego programisty /zajmuje się on również pewnymi, specjalnymi problemami, które w innym przypadku odrywałyby naczelnego programistę od zasadniczych problemów/.

Sekretarka projektu kieruje dokumentacją przygotowywaną w czasie projektowania w bibliotece opracowania, zarówno w postaci czytelnej dla maszyny jak i zwykłej, pisanej. Dokumentacja ta ma zwykle postać dziennika, w którym od decyzji projektowych do wyników testów, wszystko jest systematycznie zapisywane i aktualizowane, w postaci luźnych formularzy oraz biblioteki dokumentacji w postaci czytelnej dla maszyny do administrowania opracowanymi programami. Sekretarka projektu jest również odpowiedzialna za przygotowanie i wszystkie przebiegi programu oraz wypisanie i klasyfikację wyjścia.

---

x/ Back-up programmer — odpowiednik starszego programisty /przyp. tłumacza/

W porównaniu do zespołu konwencjonalnego, opracowanie programu przez wiodący zespół programowania pociągnęło za sobą dwie nowe jego cechy:

- dokładnie zorganizowane kierowanie biblioteką opracowania systemu programowego i prowadzenie testowania programu przez sekretarkę projektu,
- wprowadzenie programowania strukturalnego i podziału pracy zgodnie z precyzowaniem projektu programu metodą krok-po-kroku, oraz możliwościami i specjalizacją członków zespołu.

Forma organizacji wiodącego zespołu programowania zmierza do zmniejszenia czasu poświęconego diagnozie i rewizji programów, które stały się wadliwe na skutek ubogiej logiki lub niewłaściwej komunikacji między programistami.

Jakość systemu programowego ma być zagwarantowana przez doświadczenie i rozumienie przez głównego programistę, stosowanie zasad programowania strukturalnego, logiczny podział pracy, zgodnie ze specjalizacją oraz sukcesywnie tworzoną i wyczerpującą dokumentację projektu w postaci biblioteki opracowania.

Niezależnie od programowania strukturalnego biblioteka opracowania stanowi zasadniczy element w zespole. Umożliwia ona śledzenie postępu prac nad projektem i zmniejsza potrzebę komunikowania się między programistami. Zgodnie z prawem Brooka taka komunikacja rośnie w przybliżeniu wykładniczo wraz z liczbą członków zespołu. Choćby wiodący zespół programowania był idealny, w praktyce dają zwykle o sobie znać następujące problemy:

1. Wiele przedsiębiorstw ma tak małe działy przetwarzania danych, że projekty, aby prace mogły być prowadzone równolegle, trzeba dzielić między poszczególnych programistów.
2. Pozycja głównego programisty /i programisty wspierającego/ wymaga zatrudnienia doświadczonych, inteligentnych i wyszkolonych programistów. W większości przedsiębiorstw tacy ludzie od dawna znajdują się już w kierownictwie, a wśród programistów niewielu tylko ma doświadczenie w programowaniu strukturalnym.

3. Trudności pojawiają się też przy wyborze członków zespołu. W przedsiębiorstwach zwykle są ludzie, którzy chociaż są ekspertami w swojej dziedzinie, nie mają predyspozycji do pracy w zespole. Występuje też pytanie kto ma dobierać członków zespołu i według jakich kryteriów ma to czynić.

4. Bieżące metody oceny pracowników są dostosowane do środowiska konkurencji, gdy tymczasem wiodący zespół programowania powinien tworzyć środowisko współdziałania. Łatwiej jest oceniać osiągnięcia zespołu jako całości niż osiągnięcia poszczególnych członków w ramach tego zespołu. Jeśli ocenianie członków zespołu jest powierzone głównemu programiście, co podnosi go do rangi zwierzchnika, należy wytyczyć w ramach zespołu nową drogę awansu, aby pokazać w jaki sposób członek zespołu może stać się wiodącym programistą, a wiodący programista - kierownikiem.

5. Technika programowania strukturalnego ma to do siebie, że projektowaniu programu poświęca się więcej czasu. Kodowanie nie może być rozpoczęte zanim nie zostaną określone: logika sterowania i punkty intersekcji między modułami. Wynikiem tego jest konieczność ustalenia standardów procesu opracowywania systemu oraz określenie kompletności projektu, po to, aby móc ocenić postęp prac. Na tej podstawie można sporządzać wykazy kontrolne dla oceny postępu.

6. Dotąd stosowana technika testowania musi być zrewidowana pod kątem techniki wdrażania programów "od góry-w dół".

7. Chociaż w konwencjonalnej organizacji programowania nie ma wiarygodnego sposobu szacowania kosztów i czasu poświęconego projektowi, to jednak z niektórych przedsiębiorstw można uzyskać pewne empiryczne wartości. Natomiast brak jest empirycznych danych dotyczących wiodącego zespołu programowania.

8. Początkowe doświadczenia pracy wiodącego zespołu programowania ujawniły, że praca zespołowa może prowadzić do osobistej demotywacji działania. Indywidualnemu członkowi zespołu nie łatwo identyfikować się z autorami osiągnięć zespołu i to

tak dalece, że zatraca się osobista odpowiedzialność za system programowy jako całość.

Zasada braku "ja" jako czynnika emocjonalnego w programowaniu ma uniemożliwić indywidualnemu programiście zadzierzgnięcie psychicznych więzów z programem jako owocem jego pracy i niejako przedłużeniem własnej osobowości, a przez to chronić program od błędów mających źródło w zbyt wysokim mniemaniu programisty o sobie samym. Ale w takich warunkach unika on szukania rady i pomocy u kolegów, aby nie ujawnić swoich słabości i koniec końców pracuje nieefektywnie.

Brak czynnika "ja" w programowaniu stanowi o atmosferze pracy i zaletach tzw."otwartego" programowania. Przez "otwarte" posługiwanie się wstępnie przygotowanym kodem, indywidualny programista jest zmuszony do jasnego i zrozumiałego dla jego kolegów kodowania. Jednak niełatwo jest całkowicie wyrugować "ja" z programowania, ponieważ zawsze istnieje autorytet członków zespołu i kierownika.

Bez wątpienia biblioteka opracowań wiodącego zespołu programowania jest drogą upowszechnienia kodu. Rola głównego programisty jako wybitnego znawcy kodu odbiera pozostałym programistom okazję eksponowania swojego "ja" w tym zakresie.

Weinberg sądzi mimo to, że idea wiodącego zespołu programowego jest zbyt autorytatywnym podejściem do programowania bez indywidualnego "ja". Rozważa on organizację zespołu, w której wszyscy biorą jednakowy udział w pracy i wzajemnie się dopingują do osiągnięcia wspólnego celu w coraz to doskonalszy sposób. Niemniej jednak w ramach wiodącego zespołu programowania można egzekwować zmodyfikowaną postać braku "ja" w programowaniu drogą wspólnych spotkań użytkowników, analityków i programistów mających na celu przemyślenie poszczególnych części systemu programowego dla wykrycia jego błędów i niedociągnięć. Całkowite wyrugowanie "ja" z programowania jest nieosiągalne, ponieważ w grupie zawsze znajdzie się przynajmniej jeden nieformalny przywódca. Ale nawet gdy tak jest, Weinberg wymienia następujące zalety tej metody programowania:

1. Większe możliwości wykrycia błędów poprzez rewizję kodowania oraz gotowość członków zespołu do zwracania się o pomoc.
2. Mniejsza zależność od ludzkiego indywiduum, ponieważ więcej niż jedna osoba zna i może ocenić kodowanie.
3. Brak zahamowań pracy, ponieważ jej spiętrzenia są wygładzane przez ścisłą kooperację.
4. Duże możliwości uczenia się, ponieważ praca każdego z członków zespołu jest konfrontowana z pracą pozostałych kolegów.
5. Zmniejszenie kosztów opracowania systemu programowego.

Niezależnie od problemu wprowadzenia do działu przetwarzania danych wiodącego zespołu programowania, występują też problemy wynikające z prowadzenia programowania strukturalnego. Przeważająca część tych problemów jest natury psychologicznej. Z jednej strony myślenie w kategoriach bloków strukturalnych lub modułów jest obce programistom, ponieważ ich sposób myślenia wiąże się głównie z językiem programowania, którego używają i ponieważ ich szkolenie było nastawione na uczenie się języka, z drugiej zaś, co jest najważniejsze, wydają się akceptować zmianę. Tak więc, jak to zwięźle podsumował szef działu przetwarzania danych: "Istnieją dwie kategorie ludzi - tych, którzy wolą robić dalej to co zawsze robili i tych, którzy chcą robić coś innego".

Gdy wprowadza się programowanie strukturalne pierwsi zdecydowanie oponują, drudzy zaś to ci, którzy są mocniej zaangażowani we wstępnych etapach pracy. Starsi i bardziej doświadczeni programiści zazwyczaj podchodzą do sprawy programowania strukturalnego sceptycznie. Zawdzięczają oni swoje sukcesy znajomości i pracy w istniejącym porządku i nie mają pewności, czy w nowym nadal będą odnosić sukcesy. To samo tyczy się kierownika. Jakkolwiek istniejące metody opracowania systemu nie dają specjalnie dobrych wyników, to jednak są one bliższe personelowi, gdyż są po prostu znane. Wielu kierowników naraziło na szwank swoją pozycję, gdy odeszli od ustalonych i zaakceptowanych praktyk nie uzyskując przy tym poparcia podwładnych.

Odpowiedzialny kierownik powinien zdawać sobie sprawę z tego, że wszystkie jego próby związane z wprowadzeniem programowania strukturalnego będą dokładnie śledzone zarówno przez jego pomocników, jak i wrogów oraz z tego, że poparcia może się spodziewać nie wcześniej niż w momencie zapisania na swym koncie pierwszych sukcesów.W związku z tym często spotyka się ze strony kierowników przetwarzania danych ze stwierdzeniem,iż termin: "programowanie strukturalne" powinien być, w miarę możności, unikany, aby nie wzbudzać zbyt wielkich emocji.

Doświadczenie wskazuje, że zainteresowanie pracowników i dobre stosunki międzyludzkie wraz z odpowiednią organizacją prac i właściwym szkoleniem są przy wprowadzaniu programu strukturalnego czynnikami decydującymi o powodzeniu. Zróżnicowanie zarobków powinno tworzyć zachętę do szkolenia i podnoszenia własnego poziomu wiedzy pracowników. Fakt, że programiści starego typu czują się z reguły zagrożeni przy wprowadzaniu programowania strukturalnego daje dużo do myślenia.Przykłady wskazują jednak, że chociaż dobrzy programiści odnoszą się sceptycznie do programowania strukturalnego to jednak po krótkim zaznajomieniu się z nim, są za nim, gdyż jest ono instrumentem standaryzacji i uproszczenia procesu opracowania systemu.

Oprócz problemów natury emocjonalnej występują także problemy organizacyjne. Forma organizacji procesu opracowania systemu właściwa programowaniu strukturalnemu musi być rozpatrywana w świetle specyficznych warunków istniejącej organizacji. Organizacja strukturalną działów APD z jej względnie ścisłymi zasadami, generalnie nie nadaje się do przyjęcia za podstawę do wprowadzenia wiodących zespołów programowania, przed tym muszą tu być dokonane i zaaprobowane odpowiednie modyfikacje. Wszystko to pojawia się w Europie w czasie, gdy po wielu latach rozwoju nastąpiła już jakaś konsolidacja organizacji w działach APD. Konsekwencją tego stanu rzeczy jest duża niechęć do wprowadzania zmian.

VII. WPROWADZENIE PROGRAMOWANIA STRUKTURALNEGO

Wiele trudności wprowadzenia programowania strukturalnego wynika z charakteru używanych dziś języków programowania. Największe trudności adaptacji występują w najczęściej używanych językach i systemach operacyjnych. Poniżej przedstawiono zasady wprowadzania programowania strukturalnego w koncernie Boening:

1. Kierownik odpowiedzialny za wprowadzenie programowania strukturalnego potrzebuje pełnego poparcia ze strony kierownictwa najwyższego szczebla, a sam powinien doprowadzić to poparcie do kierowników i specjalistów zajmujących się opracowaniem oprogramowania.

2. Zapewnienie pełnego poparcia ze strony kierownictwa najwyższego szczebla musi zawierać gwarancję przejścia na programowanie strukturalne za jednym zamachem. Nie może być ono przerywane lub opóźniane odwrotami, odwołaniami lub innymi akcjami.

3. Należy dokładnie określić jakie metody programowania strukturalnego będą wprowadzane.

4. Dla obydwu grup: kierowników i programistów należy przygotować specjalny program szkolenia, zapewniający m.in. również wspólne zajęcia obydwu grup. Program szkolenia programistów powinien być zróżnicowany na szkolenie podstawowe, z niewielką ilością zajęć praktycznych i kurs dla bardziej zaawansowanych. Szkolenie podstawowe może składać się z czytania łatwego podręcznika programowania strukturalnego oraz prostego, o małym zakresie, pilotowego projektu - do wykonania w normalnym zakresie pracy programisty. Chodzi tu o stworzenie programiście warunków powodzenia w stosowaniu programowania strukturalnego i nabycia przez niego pewnego doświadczenia. Jednak nie wolno oszczędzić mu uświadomienia dużego jeszcze braku doświadczenia w programowaniu strukturalnym i istotnych trudności adaptowania się do tej techniki programowania.

5. Kurs bardziej intensywny pomyślany jest jako uczestnictwo w jednym z seminariów lub w zajęciach praktycznych na temat programowania strukturalnego. Oferuje je wiele znanych instytucji. Możliwe są również kontakty z innymi organizacjami pracującymi w tej dziedzinie zagadnień.

6. Główni programiści powinni być wyszukani spośród pracowników instytucji jako ci, którzy wyróżniają się zdolnościami technicznymi, społecznymi i kierowniczymi, i to nie tylko w ocenie kierownictwa, ale i reszty personelu. Na głównych programistów powinni być wybierani tylko ochotnicy.

7. Wprowadzenie wiodącego zespołu programowania stwarza konieczność zrewidowania poprzedniej struktury organizacji.

8. Należy powołać biuro kontroli oprogramowania, m.in. dla wypracowania norm postępowania w sprawach odpowiedzialności za doskonalenie, eksploatację i kontrolę standardów rozwoju oprogramowania, dla określenia procedur i harmonogramów testowania, metod kontroli konfiguracji itd.

9. W programowaniu strukturalnym jedną z podstawowych spraw jest analiza kosztów. Informacje uzyskane drogą kontroli kosztów powinny być wykorzystywane do oceniania wpływu programowania strukturalnego na zwrot poniesionych nakładów na system programowy.

10. Należy zrewidować istniejące metody oceny wzrostu i rozwoju personelu i kierownictwa.

Cena zł 82.-