Podstawy programowania komputerów dla inżynierów

Podręczniki – Politechnika Lubelska

Politechnika Lubelska
Wydział Mechaniczny
ul. Nadbystrzycka 36
20-618 LUBLIN

Grzegorz Samołyk

Podstawy programowania komputerów dla inżynierów

Politechnika Lubelska
Lublin 2011

Recenzent:
dr hab. inż. Andrzej Gontarz, prof. Politechniki Lubelskiej

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-62596-46-1

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
www.esus.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakład: 100 egz.

SPIS TREŚCI

OD AUTORA

Rozwój technologii informacyjnej, jaki obserwowany jest od wielu lat, stawia nowe wymagania dla inżynierów mechaników. Oprócz podstawowych umiejętności z zakresu projektowania i eksploatacji maszyn, inżynier XXI wieku musi również posiadać wiedzę z zakresu informatyki stosowanej, w szczególności programowania komputerów. Jest to dział informatyki, którego elementy są coraz częściej włączane w zakres prawie wszystkich dyscyplin wiedzy. Obecnie, dla każdego inżyniera technologie informacyjne, w tym również projektowanie i programowanie programów komputerów, są źródłem cennych narzędzi technicznych, metodologicznych lub formalnych.

Oczywiście powstaje pytanie o następującej treści: W jakim stopniu i w jakim zakresie inżynier mechanik powinien poznać techniki programowania komputerów? Jednak zanim zostanie udzielona odpowiedź na to pytanie, należy zwrócić uwagę na ważną kwestię. Mianowicie, praca inżyniera mechanika, który nie jest informatykiem w pełnym znaczeniu tego określenia, polega głównie na projektowaniu części maszyn oraz opracowywaniu technologii ich wykonania. W obecnych czasach jego podstawowym narzędziem są specjalistyczne oprogramowania komputerowe wspomagające szereg jego czynności. W większości przypadków są to uniwersalne lub wyspecjalizowane aplikacje komercyjne. Przykładem mogą być systemy typu CAx, które są używane podczas projektowania lub obliczeń inżynierskich. Większość z tych programów posiada wbudowane zaawansowane technicznie edytory służące to tworzenia tzw. makr. Dzięki temu można zautomatyzować te czynności, które są wielokrotnie powtarzane oraz charakteryzują się czasochłonnością i monotonicznością.

Niestety, wadą specjalistycznego oprogramowania komercyjnego jest jego stosunkowo duża cena, często zaporowa w przypadku małych zakładów produkcyjnych lub biur konstrukcyjnych. Wiedząc o tym, że wyspecjalizowane programy komputerowe implementują metody projektowe, dobrze znane inżynierom, powstaje kolejne istotne pytanie: Czy inżynier może opracować taką implementację w własnym zakresie? Odpowiedź jest jedna – „tak”. Ponadto, w sytuacji, gdy nie można zakupić odpowiedniego oprogramowania (z powodu ceny lub jego braku na rynku), można śmiało odpowiedzieć – „powinien”, szczególnie jeżeli takie oprogramowanie w ogóle nie istnieje.

Autor chciałby zaznaczyć, że inżynier XXI wieku powinien posługiwać się biegle dowolnym językiem programowania, który w sposób prosty i szybki

umożliwi mu opracowanie dowolnej komputerowej implementacji postawionego zadania inżynierskiego. Z punktu widzenia osoby nie będącej informatykiem, najlepszym narzędziem jest taki edytor programowania (język programowania), w którym pewna grupa czynności programistycznych jest automatycznie obsługiwana przez ten edytor (lub/i kompilator). Takimi czynnościami, o których jest mowa, z pewnością są przydzielanie pamięci dla zmiennych, rzutowanie danych oraz generowanie interfejsu graficznego programu. Rola mechanika-programisty powinna sprowadzać się jedynie do napisania odpowiednich procedur implementujących rozwiązanie danego zadania. Jednym z języków programowania, które spełnia kryteria narzędzia prostego w użyciu jest Visual Basic. Jednak dotychczasowe doświadczenie autora wykazało, że opracowanie w pełni funkcjonalnej aplikacji specjalistycznej (nawet z użyciem bardziej zaawansowanego języka programowania) jest możliwe tylko wtedy, gdy programista dodatkowo (oprócz znajomości składni języka) posiada elementarną wiedzę z zakresu technik programowania komputerów.

Niniejszy skrypt jest dedykowany studentom studiującym na kierunkach technicznych, których profil związany jest z dziedziną mechaniki, budowy i eksploatacji maszyn. Zawarty w nim materiał dydaktyczny może zostać wykorzystany w takich przedmiotach jak „języki programowania”, „projektowanie i programowanie obiektowe” oraz „komputerowo wspomagane przetwarzanie danych doświadczalnych”. Autor celowo pomija zagadnienia związane z programowaniem maszyn sterowanych numerycznie, ze względu na dużą dostępność literatury omawiającej to zagadnienie. Głównym celem tej książki jest przybliżenie istoty oraz elementarnej wiedzy z zakresu technik tworzenia programów wspomagających obliczenia inżynierskie. Zakłada się, że czytelnik zna składnię języka Visual Basic, zarówno w wersji starszej oznaczonej numerem 6, jak i wersji tzw. VBA oraz wersji obiektowej oznaczonej jako VB .NET.

W tym miejscu autor chciałby wyrazić podziękowania wszystkim osobom, które przyczyniły się do powstania niniejszego opracowania. Odrębne podziękowania kieruje do recenzenta, Pana dr hab. inż. Andrzeja Gontarza, profesora Politechniki Lubelskiej, którego cenne uwagi i spostrzeżenia wpłynęły na obecną postać skryptu.

1. WSTĘP DO PROGRAMOWANIA KOMPUTERÓW

1.1. Definicja komputera

Pojęcie „**komputer**” można zdefiniować jako elektroniczną maszynę cyfrową przeznaczoną do automatycznego przetwarzania danych, które są przedstawione w postaci zaszyfrowanej. Tłumacząc z łaciny, słowo „*compurere*” oznacza czynność polegającą na rozważaniu lub obliczaniu.

W praktyce, komputer jest to zespół urządzeń elektronicznych, które można pogrupować w bloki funkcjonalne. Zgodnie z zdefiniowaną architekturą komputera przez J. von Neumanna [2, 9, 19, 20, 24, 28], można wyróżnić:

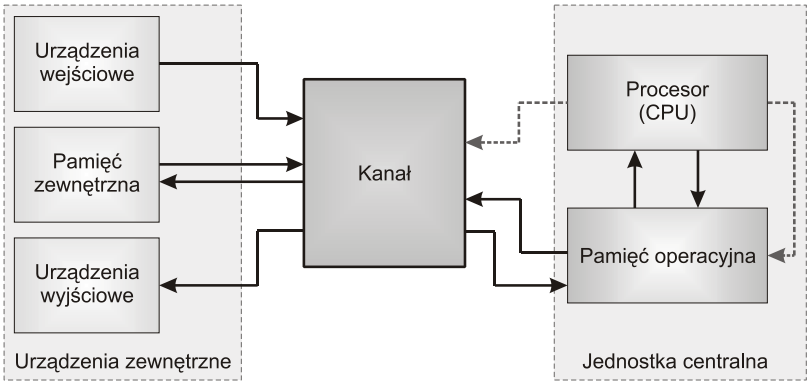
- **urządzenia wejścia**, które służą do wprowadzania do komputera danych do przetwarzania oraz programów;
- **urządzenia wyjścia**, za pomocą których wyprowadzane są z komputera wyniki (informacja) przetwarzania danych;
- **pamięć operacyjna** służąca do przechowywania danych i programów, które przetwarzają te dane;
- **procesor** (CPU), który wykonuje operacje arytmetyczne i logiczne na danych pobieranych z pamięci operacyjnej oraz steruje (synchronizuje) i kontroluje pracą wszystkich elementów komputera.

Procesor oraz pamięć operacyjna stanowi jednostkę centralną komputera (rys. 1.1). Natomiast urządzenia wejścia i wyjścia należą do urządzeń zewnętrznych, w skład których zalicza się również inne urządzenia współpracujące z komputerem, między innymi: pamięć zewnętrzna (nośniki danych i programów), klawiatura, drukarki, plotery, monitory, urządzenia pomiarowe oraz urządzenia sterujące maszynami przemysłowymi.

Ostatnią grupą urządzeń, które są niezbędne do funkcjonowania komputera są urządzenia przesyłu (wymiany) danych pomiędzy urządzeniami zewnętrznymi a jednostką centralną. Urządzenia te nazywa się kanałami.

Głównymi zaletami obecnych komputerów, wynikającymi między innymi z ich budowy, są:

- automatyczne podejmowanie decyzji, zgodnie z treścią programu;
- duża pojemność pamięci, która pozwala przechowywać jednocześnie wiele programów i dużych zbiorów danych;



Rys. 1.1. Schemat blokowy komputera, gdzie: → strumień przepływu danych, ⇢ sygnały sterujące i synchronizujące pracą komputera

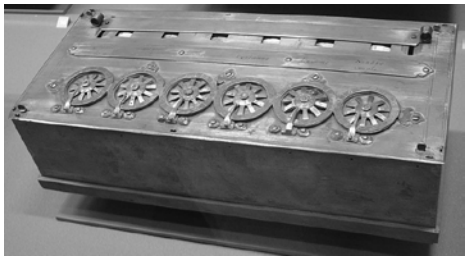
- duża szybkość wykonywanych operacji;
- duża pewność działania.

W obecnych czasach, komputery mają zastosowanie we wszystkich dziedzinach nauki, techniki i gospodarki. Wykorzystywane są one do obliczeń naukowych i inżynierskich, stosowane są jako urządzenia pomocnicze przy projektowaniu konstrukcji i procesów technologicznych oraz do sterowania procesami technologicznymi.

1.2. Zarys historii rozwoju komputerów

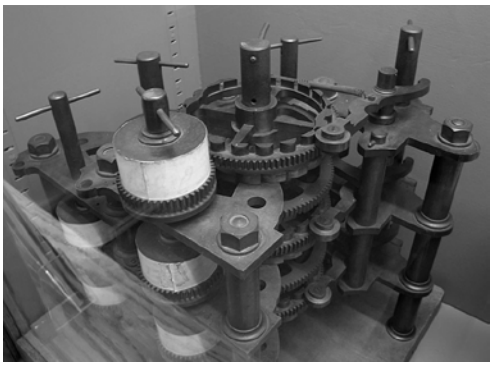
W obecnych czasach, komputer jest uważany za elektroniczną maszyną cyfrową. Jednak nie zawsze był on utożsamiany z elektroniką. Pierwszy komputer wykonany przez człowieka został zbudowany ponad pięć tysięcy lat temu. W roku około 3 000 p.n.e., starożytni Grecy używali liczydła do szybkiego ręcznego dodawania, które nazywali **abakiem** (abakusem). Liczydło to jest uważane za pierwszy nieformalny komputer zerowej generacji. Abak również był używany później przez Rzymian, a także w Europie Zachodniej aż do XVIII wieku. Dopiero po odkryciu logarytmów (suwaka logarytmicznego, pałeczki Nepara) liczydła jako „komputer” mogły zostać zastąpione przez pierwsze maszyny liczące. W roku 1645, B. Pascal zbudował maszynę mechaniczną do wykonywania operacji dodawania [8]. Fotografię przedstawiającą taką maszynę – zwaną potocznie **paskaliną** – umieszczono na rys. 1.2. Jak wynika z projektów

W. Schickarda, S. Morlanda oraz G.W. Leibniza, mechaniczne maszyny tej generacji mogły wykonywać również operacje mnożenia i dzielenia, jednak nie były to automaty zdolne do automatycznego przetwarzania danych.



Rys. 1.2. Mechaniczna maszyna licząca projektu B. Pascala [7]

Prawdziwych początków informatyki można doszukiwać się dopiero w XIX wieku, gdy Ch. Babbage stworzył koncepcję (1822 r.) automatycznej i uniwersalnej maszyny liczącej [7]. Co ciekawe, skonstruowana maszyna „różnicowa" wyprzedzała technicznie swoją epokę. Dopiero po dziesięciu latach prób udało mu się technicznie zrealizować tylko fragment tej maszyny. Niepowodzenie Ch. Babbage'a przerodziło się w opracowanie projektu (1833) nowej maszyny, nazwanej maszyną „analityczną". W pełni funkcjonalny model maszyny „różnicowej" wytworzono dopiero w latach dziewięćdziesiątych ubiegłego wieku. Obecnie znajduje się on w Londyńskim Muzeum Nauki (rys. 1.3). Maszyna Ch. Babbage'a [7, 20] odpowiada swoją strukturą współczesnym komputerom, w szczególności architekturze J. von Neumann'a. Posiada ona magazyn (odpowiednik pamięci) oraz młyn (jednostkę liczącą). Sterowana była programem zapisanym na kartach perforowanych.



Rys. 1.3. Maszyna różnicowa projektu Ch. Babbage'a wykonana przez Londyńskie Muzeum Nauki [7]

Idea Ch. Babbage'a została wykorzystana przy budowie pierwszych formalnych komputerów zerowej generacji. Była to elektromechaniczna maszyna licząca (z wbudowanym algorytmem pracy), o nazwie ASCC, zbudowana z przekaźników przez H. Aikena w roku 1944 oraz kalkulator **MARK I** (IBM, 1939 r.) [7, 8, 20].

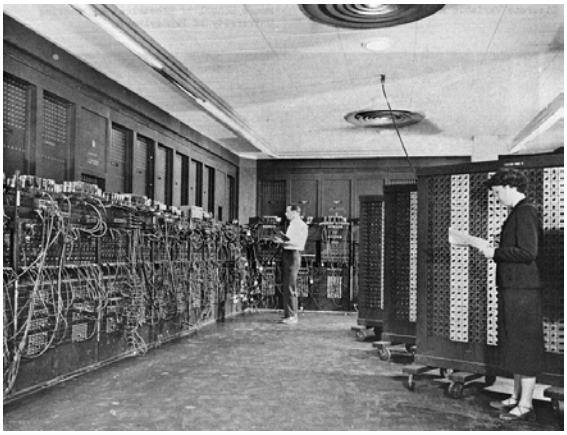
Wybuch II Wojny Światowej w znaczący sposób przyczynił się do przyspieszenia

prac rozwojowych nad komputerami. Już w roku 1946, J. Mauchly i J.P. Eckert zbudowali komputer pierwszej generacji w oparciu o lampy próżniowe (elektronowe), który nazwali **ENIAC**. Była to pierwsza maszyna cyfrowa, w pełni elektroniczna, która zawierała 18 tys. lamp elektronowych i była ponad tysiąc razy szybsza od kalkulatora MARK I. Niestety, nie była to maszyna ekonomiczna (rys. 1.4). Zajmowała klimatyzowane pomieszczenie o powierzchni 140 m^2, ważyła ponad 30 ton i wymagała obsługi kilkudziesięciu ludzi [8, 20].

Zastosowanie tranzystorów do budowy maszyn cyfrowych (od roku 1959) przyczyniło się znacząco do zmniejszenia rozmiarów komputerów oraz polepszenia ich sprawności i możliwości obliczeniowej. Dlatego też okres panowania komputerów drugiej generacji to okres, kiedy rozpoczyna się intensywny rozwój języków programowania. W tym czasie powstaje szereg odmian języków, a ich rozwojowi przyświeca konieczność usystematyzowania efektywnego sposobu oprogramowania komputerów.

Rozpoczęcie realizacji projektów badań kosmicznych przez NASA wymusiło dalszy rozwój komputerów. W roku 1965 rozpoczęto budować komputery trzeciej generacji w oparciu o układy scalone. Dzięki temu uzyskano maszyny cyfrowe charakteryzujące się małymi rozmiarami (można było je zabrać na pokład statku kosmicznego), pobierały znacząco mniej energii niż komputery poprzedniej generacji oraz były bardziej niezawodne. Obecnie (od 1975 r.) komputery budowane w oparciu o układy scalone o dużym stopniu integracji (VLSI) zaliczane są do generacji czwartej. Przewiduje się, że w najbliższej przyszłości powstaną komputery piątej generacji, które będą wytwarzane bazując na technologii optycznej [17, 20].

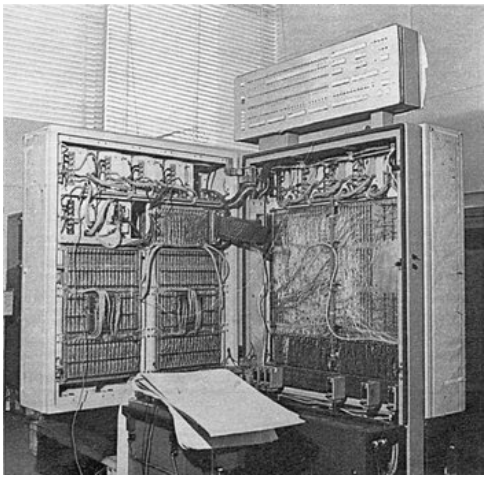
Na terenie Polski pierwszym uruchomionym komputerem była maszyna cyfrowa XYZ, wykonana technologią lampową. Powstała ona w roku 1958 pod kierunkiem L. Łukaszewicza [8]. Na szczególną uwagę zasługuje pierwszy polski komputer III generacji – **ODRA** [5, 9], począwszy od serii 1000, a kończąc na serii



Rys. 1.4. ENIAC – komputer pierwszej generacji (opis w tekście) [8, 9]

1300. Historia tej maszyny zaczyna się w latach 70 ubiegłego wieku. Przez wiele lat był to komputer wykorzystywany do wielu zadań. Charakteryzował on się wyjątkową bezawaryjnością, możliwością oprogramowania językiem zewnętrznym oraz podłączenia urządzeń peryferyjnych. Ostatni model ODRY (z serii 1300 – rys. 1.5) został ostatecznie wyłączony w 2003 r. Komputer ODRA 1305 był maszyną bardzo szybką jak na ówczesne czasy – lata 70 ubiegłego wieku. Cechował się rozbudowanym systemem urządzeń peryferyjnych i był przeznaczony do wykonywania obliczeń naukowo-technicznych. Komputer ten posiadał możliwość działania na liczbach stało- i zmienno-przecinkowych. Jednostką informacji było słowo, składające się z 24 bitów, które mogło przechowywać liczbę, znak czteroliterowy oraz rozkaz (kod w języku wewnętrznym). Istniała również możliwość sterowania działaniami ODRY za pomocą języka wyższego rzędu. Dołączony interpreter umożliwiał programowanie tej maszyny w takich językach jak COBOL, ALGOR lub FORTRAN. Jak na tamte czasy, komputer ODRA z serii 1300 był maszyną nowoczesną o dużych możliwościach obliczeniowych. Gdy systemem operacyjnym był GEORGE 3, maszyna ta cechowała się możliwością podłączenia nawet 63 urządzeń peryferyjnych pracujących w trybie *on-line*, wieloprogramowością oraz możliwością pracy w układzie wieloprocesorowym (dwu- lub czteroprocesorowym).

W latach 80 ubiegłego wieku rozpoczął się intensywny rozwój komputerów, na skalę nie spotykaną w innych dziedzinach techniki. W połowie roku 1981 amerykańska firma IBM wprowadza na rynek komputer osobisty, który wyznaczył kierunek dalszej ewolucji maszyn liczących. Pierwszy model komputera IBM PC (z ang. *Personal Computer*) wyposażony był w procesor Intel 8088 o zegarze taktowanym z częstotliwością 4,77 MHz, pamięć operacyjną o pojemności 64 KB, pierwszy BIOS oraz system operacyjny MS-DOS. Jego architektura była modułowa składająca się z ogólnie dostępnych podzespołów elektronicznych. Najważniejszą cechą tej budowy było to, że była ona nadzwyczaj nowoczesna i nie posiadała zastrzeżeń patentowych.



Rys. 1.5. ODRA 1305 – komputer trzeciej generacji produkcji polskiej [6, 7]

Oczywiście należy zaznaczyć, że przed skonstruowaniem komputera IBM PC na rynku były już dostępne komputery osobiste (np. Apple, Atari, Commodore). Niestety, komputery te (a raczej mikrokomputery) wyposażone były w mikroprocesory (częstotliwość zegara nie przekraczała 1 MHz), a ich architektura nie była modułowa. Zatem, popularne ówcześnie mikrokomputery nie były przystosowane do szybkiej ewolucji determinowanej rozwojem technologii informacyjnej.

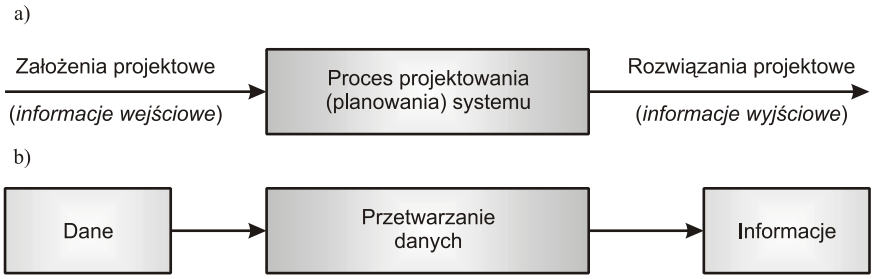
Kończąc ten podrozdział, jako ciekawostkę warto przytoczyć sceptyczne wypowiedzi osób, które odegrały ważną rolę w rozwoju komputerów. Na początku lat 40 XX wieku, T. Watson (prezes firmy IMB) stwierdził, że na świecie znajdzie się być może pięć osób, które kupią komputer. Natomiast 35 lat później, K. Olsen (założyciel firmy Digital Equipment Corp – DEC) uznał, że nie ma żadnego powodu, aby ktoś normalny chciał mieć komputer w domu [3]. Zapewne, kiedyś w podobny sposób o swoim zakresie umiejętności myśleli ówcześni inżynierowie mechanicy. Nie przypuszczali oni, że współczesny rynek pracy wymaga od inżynierów dodatkowo znajomości programowania, zarówno komputerów jak i maszyn sterowanych numerycznie.

1.3. Przetwarzanie danych

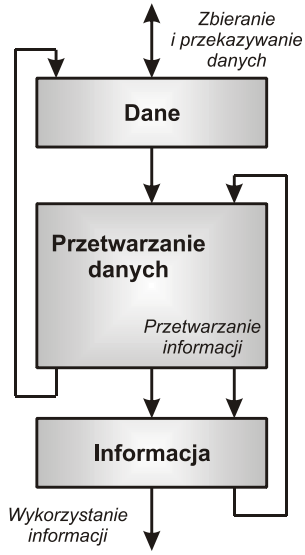
Zasadnicze dziedziny twórczej pracy inżynierskiej to projektowanie oraz budowa i eksploatacja systemów. Pod pojęciem „system” należy rozumieć zbiór określonych obiektów wraz z relacjami istniejącymi między tymi obiektami. Systemem może być zarówno maszyna (konstrukcja) przemysłowa jak i proces technologiczny. Dzięki temu, model pracy inżyniera mechanika zbudowany na bazie pojęcia systemu (rys. 1.6a), można z łatwością porównać do informacyjnego modelu danych i ich przetwarzania w informację (tj. dane wykorzystywane do celowego działania) – rys. 1.6b. W efekcie uzyskuje się podstawowy schemat budowy komputerowych implementacji czynności projektowych inżyniera, który jest równocześnie przykładem płynnego przejścia z płaszczyzny dziedziny technicznej (np. mechaniki i budowy maszyn) na płaszczyznę informatyki stosowanej.

Schemat blokowy przetwarzania danych w informatyce stosowanej (przedstawiony na rys. 1.6b oraz rys. 1.7) składa się z trzech ważnych pojęć [5, 9]:

- **dane** – reprezentują określoną treść (również informację), nadającą się do przesyłania, magazynowania oraz wykonywania na nich działań logicznych i matematycznych;
- **informacja** – utożsamiana jest z treścią posiadającą zrozumienie, jakie przy odpowiedniej konwencji przyporządkowuje się danym; informacja to również dane, które mogą być subiektywnie wykorzystane do celowego działania;



Rys. 1.6. Model pracy inżyniera w interpretacji informacyjnej (a) w porównaniu z blokowym modelem przetwarzania danych (b)



Rys. 1.7. Powiązania logiczne pomiędzy pojęciami modelu przetwarzania danych [9]

- **przetwarzanie danych** jest to proces przekształcania treści i postaci danych poprzez wykonywanie działań operacji matematycznych i logicznych.

Dodatkowo, na rys 1.7 umieszczono schematycznie powiązania logiczne oraz współzależności występujące pomiędzy wyżej opisanymi pojęciami. Zrozumienie tych powiązań jest istotne z punktu widzenia poprawnego i optymalnego programowania komputerów. Ponadto, należy pamiętać, że każda informacja przedstawiona w odpowiedniej formie może być danymi, ale dane bez przetworzenia nie mogą być traktowane jako informacją.

Podsumowując można zaznaczyć, że procesor (CPU) jest sekwencyjnym urządzeniem służącym do przetwarzania danych. Jego architektura jest ścisłym odzwierciedleniem filozofii modelu przedstawionego na rys. 1.6b.

1.4. Przechowywanie danych

Istotną rolę w funkcjonowaniu cyfrowych urządzeń elektrycznych, w szczególności komputerów, odgrywa **pamięć**. Można ją zdefiniować jako urządzenie do przechowywania danych zapisanych w postaci cyfrowej. Pamięć

komputerowa jest podzielona na komórki, którym są przypisane określone i kolejno po sobie następujące numery nazywane **adresami**.

Podstawową jednostką pojemności pamięci jest jeden **bajt** (z ang. *bite*) oznaczany dużą literą „**B**”, który wyraża ilość komórek pamięci. Ze względów praktycznych, pojemność pamięci wyraża się w jednostkach pochodnych będących wielokrotnością jednego bajta – tj. kB (równy 2^{10} B), MB (2^{20} B), GB (2^{30} B) oraz TB (2^{40} B). Drugą jednostką stosowaną w informatyce do określenia ilości danych jest **słowo** (z ang. *word*). Jeżeli dla jednostkowej porcji (bloku) danych przydzielono określoną ilość komórek pamięci (tj. wielokrotność bajtów), to taką ilość danych można wyrazić za pomocą **słowa maszynowego**.

Podano, że pamięć komputera jest podzielona na komórki. Zatem powstaje pytanie: W jaki sposób w komórkach są przechowywane dane? W praktyce, komórki składają się z ośmiu miejsc mogących przyjmować dwa skrajne stany – np. włączony lub wyłączony. Zatem, wartość przechowywana w komórkach wyrażona jest za pomocą sekwencji ośmiu znaków cyfrowych – bitów (z ang. *binary digit*). Zgodnie z teorią Johna Turkey’a [2], **bit** jest najmniejszą, niepodzielną porcją danych reprezentowaną w sposób symboliczny przez znak przyjmujący jedną z dwóch skrajnych wartości. Przyjmując binarny system liczbowy, wartość bitu może wynosić albo 0 lub 1. Zatem, jedna komórka pamięci składająca się z ośmiu bitów może przyjmować aż 256 różnych stanów, które są wykorzystywane do zakodowania wartość przechowywanej danej. Bit oznacza się małą literą „**b**”, a jego ilość można wyrażać w jednostkach pochodnych na tej samej zasadzie jak ilość bajtów.

Fizyczna forma przechowywania danych w pamięci zależy od technologii jej wytworzenia (jako urządzenia elektronicznego) oraz przeznaczenia danych. Ze względu na funkcjonalność można wyróżnić pamięć [2, 3, 5, 16]:

- **operacyjną**, w której przechowuje się program wykonywany przez procesor oraz dane przeznaczone dla tego programu; dostęp do tej pamięci ma procesor za pośrednictwem magistrali komputera, a czas dostępu wynosi zwykle kilka cykli zegara; każdy bajt lub słowo w tej pamięci można adresować; pamięć operacyjną komputera nazywa się również pamięcią wewnętrzną, główną lub pamięcią typu RAM (z ang. *Random Access Memory*);
- **podręczną**, która jest częścią procesora i zawiera kopię tych komórek pamięci operacyjnej, z których procesor będzie korzystał w najbliższym czasie; dostęp do jednej komórki pamięci podręcznej wynosi dokładnie jeden cykl zegara;
- **stałą**, gdzie przechowuje się dane, które są niezbędne tuż po uruchomieniu komputera lub muszą pozostać niezmienne w trakcie jego pracy; przykładem danych przechowywanych w tej pamięci są: mikroprogram procesora, program ładujący system operacyjny z pamięci zewnętrznej do pamięci ope-

racyjnej; program specjalizowanych sterowników oraz dane przedstawiające opis kształtu znaków wyświetlanych na monitorze;
- **zewnętrzną**, służącą do przechowywania danych w sposób trwały (najczęściej w postaci plików) oraz do przenoszenia danych pomiędzy różnymi cyfrowymi urządzeniami elektronicznymi; dane w tej pamięci zapisywane są na nośniku danych w formie bloków o wielkości od kilkudziesięciu do kilku tysięcy bajtów; dostęp do danych w pamięci zewnętrznej wymaga skopiowania całego bloku do pamięci operacyjnej – tzw. bufora.

Druga klasyfikacja pamięci oparta jest na technologii jej wykonania. Zatem, wyróżnić można pamięć:
- **półprzewodnikową**, którą ze względu na trwałość zapisu danych można podzielić na pamięć **ulotną** – gdzie zawartość pamięci jest kasowana po wyłączeniu zasilania komputera oraz pamięć **nieulotną**, gdzie jej zawartość jest zachowywana;
- **magnetyczną**, przykładem są dyski twarde;
- **optyczną** – np. płyty typu CD, DVD itp.

W półprzewodnikowych pamięciach ulotnych stan każdego bitu jest określany przez aktualny poziom napięcia lub prądu w określonym punkcie układu elektrycznego. W przypadku pamięci półprzewodnikowej statycznej (SRAM, z ang. *Static RAM*), każdy półprzewodnik reprezentuje jeden bit dostępny dopóty, dopóki zasilanie prądowe nie zostanie wyłączone lub nie zostanie nadpisany innym bitem. Zwykle pamięć statyczna jest stosowana jako pamięć podręczna, jako rejestry procesora oraz jako bufor w urządzeniach peryferyjnych.

Natomiast pamięć półprzewodnikowa dynamiczna (DRAM, z ang. *Dynamic RAM*) jest zbudowana z układów scalonych zbliżonych w zasadzie działania do kondensatorów, w których poziom naładowania określa stan bitu. Pamięć ta jest wolniejsza od pamięci statycznej i wymaga cyklicznego odświeżania informacji z powodu ciągłego rozładowywania się „kondensatorów". Z drugiej strony, w tego typu pamięci uzyskuje się znaczną gęstość zapisu danych na jednostkę powierzchni układu scalonego, dzięki czemu pamięć dynamiczna jest tańsza w produkcji oraz jest wykorzystywana jako pamięć operacyjna.

Półprzewodnikowe pamięci nieulotne są stosowane w komputerze jako pamięci stałe lub jako pamięć zewnętrzna w formie kart pamięci. Ten typ pamięci w zależności od sposobu zapisu danych dzieli się na [5]:
- **ROM** (z ang. *Read-Only Memory*), gdzie jej zawartość jest ustalana w czasie produkcji układu scalonego i już nigdy nie jest zmieniana;
- **PROM** (z ang. *Programmable ROM*) – zawartość takiej pamięci może być nagrywana jednorazowo za pomocą tzw. wypalarki;
- **EPROM** (z ang. *Erasable* PROM) – zawartość pamięci może zostać wielokrotnie kasowana, najczęściej poprzez naświetlanie promieniami ultrafiole-

towymi, a następnie ponownie nagrana tak jak w przypadku pamięci typu PROM;
- **EEPROM** (z ang. *Electrically Erasable PROM*) lub **Flash**, gdzie jej zawartość może być wielokrotnie kasowana poprzez zastosowanie odpowiedniego napięcia elektrycznego.

Większość obecnie używanych pamięci, np. typu RAM, ROM, dyski magnetyczne, dyski optyczne itp. charakteryzują się swobodnym dostępem do danych. Oznacza to, że każda komórka jest jednakowo dostępna w tym samym czasie. W przypadku pamięci w postaci taśmy np. magnetycznej, dostęp do danych jest realizowany tylko w sposób **sekwencyjny**. Wyróżnia się również pamięci o dostępie **asocjacyjnym**. W takiej pamięci komórki nie posiadają własnych adresów, a część każdej komórki jest przeznaczana na zapamiętanie etykiety. Pamięć o takim dostępie jest używana jako pamięć podręczna procesora oraz jako pamięć wirtualna.

1.5. Kodowanie liczb w komputerze

1.5.1. Pozycyjne systemy liczbowe

W ramach wstępu zostanie przybliżona elementarna wiedza z teorii liczb. Słowo „liczba” jest spostrzegane jako pojęcie abstrakcyjne, co idealnie wkomponowuje się w ideę programowania. Pierwotnie liczby służyły tylko do określania zbiorów przedmiotów, a do tego celu używano tylko liczb naturalnych. Gdy w XVII wieku p.n.e. w Egipcie wprowadzono liczby wymierne, a ok. tysiąc lat później w Grecji zaczęto używać liczb niewymiernych, liczby zaczęły służyć również do wyrażania wielkości zbiorów ciągłych, takich jak: długość, pole powierzchni, objętość oraz ciężar [5].

W rozwoju metod obliczeniowych oraz technik komputerowych kluczowe znaczenie mają zasady zapisu (odczytu) liczb całkowitych. Do tego celu używa się zastawu znaków, nazywanych cyframi. Ze względu na sposób interpretacji uszeregowania cyfr, wyróżnia się następujące dwa układy zapisu liczby [5]:
- **addytywny**, gdzie zapisana wartość jest sumą arytmetyczną poszczególnych znaków; przykładem jest układ jedynkowy, w którym pionowa kreska oznacza jedynkę, a jej wielokrotne powtórzenie daje odpowiednią wartość liczbową; w bardziej zaawansowanych układach addytywnych (np. w staroegipskim zapisie hieroglificznym lub numeracji greckiej) występuje bardziej zróżnicowana symbolika, w której osobne znaki służą do przedstawiania różnych jednostek numeracji;
- **pozycyjny**, sposób zapisu liczb charakteryzuje się tym, że wartość poszczególnych cyfr zależy od ich położenia w zapisie; położenie znaku może być rozpatrywane względem albo znaków sąsiadujących (przykładem takiego

układu jest numeracja rzymska, w której cyfra większa poprzedzająca mniejszą na wartość dodatnią, a cyfra mniejsza znajdująca się przed większą – wartość ujemną) albo znaku końcowego (taki układ liczb jest nazywany **pozycyjnym systemem liczbowym**).

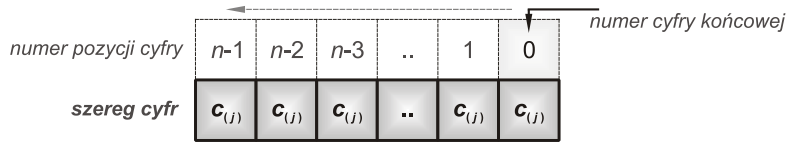
Opiszmy teraz pozycyjne systemy liczbowe, które charakteryzują się zwartym zapisem, a wykonywanie na nich działań rachunkowych jest bardzo proste. Schematyczne przedstawienie uszeregowania cyfr tworzących omawiany układ liczby przedstawiono na rys. 1.8. Liczby zapisuje się cyframi $c_{(j)}$ należącymi do pewnego niewielkiego zbioru o wielkości p, gdzie indeks $j \in [1, p]$. Natomiast wartość takiej liczby jest sumą iloczynów liczb cząstkowych reprezentowanych przez poszczególne cyfry i potęg liczby naturalnej p, nazywanej **podstawą systemu liczbowego**, o wykładnikach równych numerowi pozycji cyfry w ciągu. Można to zapisać w następujący sposób:

$$(wartość\_liczby)_{10} = c_{(j)} \cdot p^{n-1} + c_{(j)} \cdot p^{n-2} + c_{(j)} \cdot p^{n-3} + \ldots + c_{(j)} \cdot p^{1} + c_{(j)} \cdot p^{0}, \qquad (1.1)$$

gdzie kolejność występowania cyfr $c_{(j)}$ we wzorze jest taka sama jak w diagramie przedstawionym na rys. 1.8, a wykładnik liczby p jest jednocześnie numerem pozycji danej cyfry w szeregu.

Zgodnie z podstawową klasyfikacją, tj. ze względu na wielkość zbioru cyfr $c_{(j)}$, można wyróżnić trzy podstawowe pozycyjne systemy liczbowe. Są to systemy:

- **dziesiętny** ($p = 10$), który obecnie jest naturalny dla człowieka i powszechnie stosowany we wszystkich dziedzinach;
- **dwójkowy** ($p = 2$), który ma szczególne znaczenie dla informatyki, ponieważ jest naturalny dla maszyn liczących;
- **szesnastkowy** ($p = 16$), który ze względu na swoje cechy jest stosowany powszechnie w procesie programowania komputerów.



Rys. 1.8. Schemat uszeregowania cyfr tworzących **pozycyjny układ liczby**, w którym wartości znaków określa się względem cyfry końcowej; $c_{(j)}$ – cyfra z pewnego zbioru o wielkości p, którą identyfikuje indeks j należący do przedziału domkniętego $[1, p]$, n – ilość cyfr tworzących daną liczbę; numer pozycji danej cyfry w szeregu podano w wierszu pierwszym, gdzie strzałka ◄--- oznacza kierunek wyliczania pozycji cyfr

Dwójkowy system liczbowy

System dwójkowy (binarny, z ang. *binary*) umożliwia zapis liczb za pomocą dwóch umownych znaków, tj. zera (0) oraz jedynki (1). Wspomniana umowność zapisu pozwala przyjąć, że znaki te są w stanie wyrazić dwa skrajne stany dowolnego układu elektronicznego bez wnikania w szczegóły, np. jakie jest napięcie na danym obwodzie układu. Z praktycznego punktu widzenia, brak napięcia jest reprezentowany przez zero, natomiast każde napięcie o wartości większej od zera jest traktowane jednakowo – tj. reprezentowane przez jedynkę. Ta własność tego systemu liczbowego sprawia, że jest on naturalny dla urządzeń elektronicznych (komputerów) [5, 9, 31, 33].

Poniższe wyrażenie (1.2) przedstawia przykład przeliczenia liczby binarnej na liczbę dziesiętną. Jest to wykonywane zgodnie z zasadą zamieszczoną we wzorze (1.1).

$$\mathbf{(1101)_2} = 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = \mathbf{(13)_{10}}. \tag{1.2}$$

Kolejny przykład pokazuje sposób przekształcenia liczby dziesiętnej na jej odpowiednik binarny. Jak wynika z wyrażenia (1.3), jest to zadanie o charakterze algorytmicznym. Najpierw liczbę dzieli się przez dwa, a otrzymany wynik ilorazu jest obcinany do liczby całkowitej. Czynność tą powtarza się rekurencyjnie aż do uzyskania wartości mniejszej od jedności. Każdy wynik ilorazu poddaje się analizie. Jeśli wynik ten jest liczbą wymierną (czyli powstaje reszta z dzielenia), to uzyskuje się liczbę binarną poprzez wstawienie do niej, zaczynając od końca, znak „1”. W przeciwnym razie – cyfrę „0”.

$$\mathbf{(13)_{10}} = \left|\begin{array}{ll} 13 : 2 = 6,... & \textit{reszta} \Rightarrow 1 \\ 6 : 2 = 3 & \sim\textit{reszta} \Rightarrow 0 \\ 3 : 2 = 1,... & \textit{reszta} \Rightarrow 1 \\ 1 : 2 = 0,... & \textit{reszta} \Rightarrow 1 \end{array}\right| = \mathbf{(1101)_2}. \tag{1.3}$$

gdzie znak „~” oznacza operator negacji.

Szesnastkowy system liczbowy

System szesnastkowy (heksadecymalny, z ang. *hexadecimal*) umożliwia zapis liczb za pomocą aż szesnastu znaków. Zbiór tych znaków składa się z dziesięciu cyfr arabskich (0, 1, …, 8, 9) oraz sześciu znaków alfabetu łacińskiego (A, B, C, D, E, F) reprezentujących wartości w systemie dziesiętnym od 10 do 15. Cechą charakterystyczną tego systemu liczbowego jest to, że za jego pomocą można zapisać długie liczby binarne w zwięzłej postaci. Jedna cyfra liczby heksadecymalnej koduje wartość binarną składającą się co najwyżej z czterech znaków [5, 33].

Poniższe wyrażenie (1.4) przedstawia przykład, ukazujący sposób przeliczania liczby szesnastkowej na dziesiętną. Obliczenia te bazują na zależności (1.1).

$$\mathbf{(12A)_{16}} = 1\cdot 16^2 + 2\cdot 16^1 + (10)_{10}\cdot 16^0 = \mathbf{(298)_{10}}. \tag{1.4}$$

Natomiast kolejne wyrażenie (1.5) pokazuje metodę zmiany liczby dziesiętnej na szesnastkową. Idea jest w zasadzie podobna jak w przypadku wyrażenia (1.3), z tą różnicą, że powstała reszta ilorazu jest mnożona przez podstawę systemu szesnastkowego. Pozwala to określić cyfrę przekształconej liczby.

$$\mathbf{(298)_{10}} = \left|\begin{array}{lll} 298 : 16 = 18{,}625 & \Rightarrow & 0{,}625\cdot 16 = (10)_{10} \Rightarrow (A)_{16} \\ 18 : 2 = 1{,}125 & \Rightarrow & 0{,}125\cdot 16 = (2)_{10} \Rightarrow (2)_{16} \\ 1 : 2 = 0{,}0625 & \Rightarrow & 0{,}0625\cdot 16 = (1)_{10} \Rightarrow (1)_{16} \end{array}\right| = \mathbf{(12A)_{16}}. \tag{1.5}$$

Następny przykład wyjaśnia zasadę przekształcania liczby szesnastkowej na jej odpowiednik binarny. Rozpatrzmy jeszcze raz liczbę $(12A)_{16}$, którą zamienić można na liczbę binarną zgodnie z poniższym zapisem:

$$\mathbf{(12A)_{16}} = [(1)_{16} \cup (2)_{16} \cup (A)_{16}]_2, \tag{1.6}$$

gdzie poszczególne liczby cząstkowe, wyrażone w systemie szesnastkowym i znajdujące się po prawej stronie równości, zostają zamienione na postać binarną zgodnie z poniższymi obliczeniami:

$$\begin{aligned}
\mathbf{(1)_{16}} &= \left|1 : 2 = 0{,}... \quad \textit{reszta} \Rightarrow 1\right| = \mathbf{(0001)_2}\ , \\
\mathbf{(2)_{16}} &= \left|\begin{array}{ll} 2 : 2 = 1 & \sim\textit{reszta} \Rightarrow 0 \\ 1 : 2 = 0{,}... & \textit{reszta} \Rightarrow 1 \end{array}\right| = \mathbf{(0010)_2}\ , \\
\mathbf{(A)_{16}} &= \left|\begin{array}{ll} 10 : 2 = 5 & \sim\textit{reszta} \Rightarrow 0 \\ 5 : 2 = 2{,}... & \textit{reszta} \Rightarrow 1 \\ 2 : 2 = 1 & \sim\textit{reszta} \Rightarrow 0 \\ 1 : 2 = 0{,}... & \textit{reszta} \Rightarrow 1 \end{array}\right| = \mathbf{(1010)_2}\ .
\end{aligned} \tag{1.7}$$

Uwzględniając zależność (1.6) oraz otrzymane wyniki obliczeń (1.7) można ostatecznie zapisać:

$$\mathbf{(12A)_{16}} = (0001)_2 \cup (0010)_2 \cup (1010)_2 = (100101010)_2, \tag{1.8}$$

przy czym, w końcowym zapisie liczby binarnej wszystkie zera początkowe są usuwane. Stąd wynika, że każda cyfra liczby zapisanej w systemie szesnastkowym może zostać przekształcona do postaci binarnej, która jest jednocześnie cząstkowym szeregiem znaków ostatecznej liczby binarnej. Jest to ważna cecha,

która determinuje przydatność systemu szesnastkowego w procesie programowania komputerów.

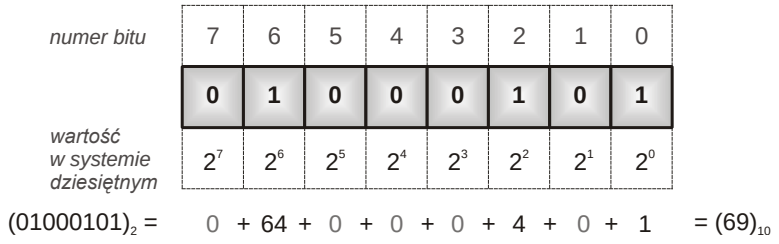
1.5.2. Przechowywanie liczb w pamięci komputera

W technice komputerowej, liczby są przedstawiane za pomocą systemu binarnego. Jest to naturalne i zarazem wygodne ze względu na budowę pamięci komputera. Kodowanie liczb całkowitych jest zagadnieniem trywialnym, jednak już w przypadku liczb rzeczywistych, zapis ich jest procesem złożonym. Jednocześnie należy pamiętać, że liczby zapisywane w komputerze stanowią jedynie określony podzbiór liczb całkowitych lub rzeczywistych [3, 5, 9, 30].

Liczby całkowite

Rozpatrzmy sposób zakodowania liczby całkowitej przy użyciu ośmiu bitów. Budowę komórki ośmiobitowej (tj. jednego bajtu), w kontekście przechowywania wartości liczby całkowitej, przedstawiono na rys. 1.9. Poszczególne bity reprezentuje jeden znak użyty do zapisu liczby w systemie binarnym. Ponadto, na tym rysunku przedstawiono przykład przeliczenia liczby $(01000101)_2$ zapisanej w systemie binarnym na liczbę $(69)_{10}$ w zapisie dziesiętnym. W tym przypadku zer początkowych w zapisie binarnym nie usuwa się, ponieważ reprezentują one bity komórki pamięci.

Analizując sposób zapisu w komputerze liczby całkowitej można stwierdzić, że przy użyciu kodowania ośmiobitowego można przedstawić jedynie 256 wartości. Najmniejszą wartością jest 0, natomiast największą 255. Jednak powstaje pytanie: w jaki sposób zakodować liczby całkowite ujemne? Rozwiązanie tego problemu jest takie, że do reprezentowania bezwzględnej wartości liczby używa się bitów o numerze od 0 to 6, natomiast bit 7 przechowuje informację o znaku,



Rys. 1.9. Reprezentacja komórki ośmiobitowej przechowującej wartość liczby całkowitej oraz przykład przeliczenia jej wartości zapisanej w systemie binarnym na zapis w systemie dziesiętnym

tj. 0 oznacza liczbę dodatnią, a 1 – liczbę ujemną. Oczywiście w takiej sytuacji nadal mamy możliwość zakodowania 256 wartości, ale z przedziału od -128 do +127. Aby zwiększyć ilość wartości dla liczby całkowitej, w praktyce stosuje się reprezentację 16-bitową, 32-bitową, 64-bitową oraz 128-bitową, gdzie ostatni bit zarezerwowany jest do przechowywania informacji o znaku liczby [2, 9, 33].

Liczby rzeczywiste

Liczby rzeczywiste są przechowywane w komputerze za pomocą reprezentacji zmiennoprzecinkowej (z ang. *floating point*) zapisanej w formie wykładniczej [5]:

$$liczba = \pm m \cdot 2^e, \tag{1.9}$$

gdzie m oznacza mantysę o wartości z zakresu $1 \le m < 2$, natomiast e jest wykładnikiem. Podstawa jest ustalona i wynosi 2, co jest zdeterminowane dwójkowym systemem liczbowym.

Mantysa, wykładnik oraz znak liczby jest pamiętany osobno w poszczególnych polach komórki pamięci. Przykładową reprezentację 16-bitowej liczby zmiennopozycyjnej przedstawiono na rys. 1.10. Znak z liczby jest zawsze jednobitowy i ma wartość 0 jeśli liczba jest dodatnia lub 1 gdy jest ujemna. Na zapis mantysy M oraz wykładnika E przeznaczone są określone, ustalone liczby bitów. Oznacza to, że za pomocą takiej reprezentacji można przedstawić skończoną ilość wartości. Dokładność zapisanej liczby jest tym większa, im dłuższa jest mantysa. Natomiast przedział wartości tej liczby jest zdeterminowany długością wykładnika.

W zasadzie każdą liczbę można zapisać w postaci zmiennopozycyjnej na wiele sposobów. Aby ujednolicić formę kodowania liczb rzeczywistych, przyjęto ograniczenia zakresu mantysy i wykładnika, które reguluje standard IEEE 754. Ponadto, pierwsza cyfra znacząca mantysy ma zawsze wartość 1 i nie jest jawnie zapamiętywana. Zatem można zapisać, że:

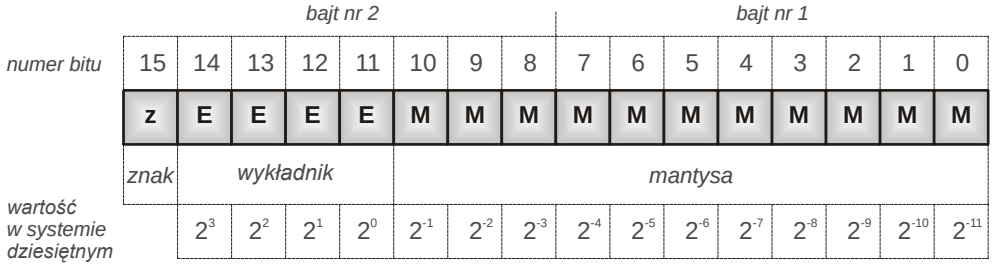
$$(mmmm)_2 = (\mathbf{1},MMMM)_2, \tag{1.10}$$

gdzie: *mmmm* – jest wartością mantysy w kodzie dwójowym, którą używa się we wzorze (1.9); *MMMM* – jest szeregiem bitów mantysy przechowywanej w pamięci komputera (rys. 1.10) i reprezentuje część ułamkową.

Z kolei wartość wykładnika liczby jest zależna od długości pola bitów i wynosi:

$$(e)_2 = (E)_2 - (2^{p-1} - 1)_{10}, \tag{1.11}$$

gdzie: e – wykładnik w kodzie dwójkowym wykorzystywany w zależności (1.9), E – wykładnik (cecha) zapisany w pamięci komputera, p – ilość bitów zarezerwowanych na wykładnik. Wyrażenie 2^{p-1} we wzorze (1.11) nazywane jest **przesunięciem**.



Rys. 1.10. Reprezentacja liczby zmiennopozycyjnej 16-bitowej; opis w tekście

W ramach przykładu zapisana zostanie liczba 2,05 przyjmując kodowanie zmiennopozycyjne 16-bitowe zgodnie z rys. 1.10. Ponieważ pole wykładnika składa się z czterech bitów, to zgodnie ze wzorem (1.11) przesunięcie wynosi -6. Bazując na zależności (1.9) zapiszmy rozwinięcie rozpatrywanej liczby, czyli:

$$2{,}05 = +1{,}025 \cdot 2^{(7-6)}. \tag{1.12}$$

W przypadku wykładnika, w pamięci komputera należy zapisać liczbę 7, zatem cecha w dwójkowym kodzie wynosi:

$$E = (0111)_2. \tag{1.13}$$

Z kolei w przypadku mantysy, w pamięci powinno się zapisać liczbę 0,025. Okazuje się, że jeżeli dysponuje się polem składającym się z 11 bitów, dokładne zapisanie tej liczby w znormalizowanym kodzie binarnym jest niemożliwe. Liczbę tą należy zaokrąglić do wartości 0,025390625, tj.:

$$M = (00000110100)_2, \tag{1.14}$$

albo uciąć do wartości 0,0249025344, co można zapisać w postaci:

$$M = (00000110011)_2. \tag{1.15}$$

Zatem, ze względu na określoną dokładność zapisu liczby za pomocą zmiennopozycyjnej reprezentacji 16-bitowej, zamiast liczby 2,05 otrzymuje się jedną z dwóch poniższych liczb:

$$\begin{aligned} 2{,}05078125 &\rightarrow \mathbf{0}\ 0111\ 00000110\mathbf{100} \\ 2{,}04980688 &\rightarrow \mathbf{0}\ 0111\ 00000110\mathbf{011} \end{aligned} \tag{1.16}$$

gdzie bity są przedstawione zgodnie z przyjętą reprezentacją liczby (rys. 1.10). Przedstawiony przykład wyjaśnia jednocześnie powód pojawiania się błędów w obliczeniach wykonywanych z użyciem liczb zmiennoprzecinkowych.

Powróćmy teraz do wspomnianej normy IEEE 754, która jest powszechnie obowiązującym standardem określającym zasadę przechowywania liczb zmiennopozycyjnych we współczesnych komputerach. Norma ta również określa zasady wykonywania obliczeń arytmetycznych na tych liczbach, co zapewnia niezmienność działania programów uruchamianych na różnych komputerach. Jednocześnie warto zaznaczyć, że za operacje na liczbach zmiennopozycyjnych odpowiadają zupełnie inne obwody procesora niż za operację na liczbach całkowitych [2, 9].

Zgodnie z zaleceniami wspomnianego standardu, liczby zmiennopozycyjne należy przechowywać w następujących formatach [10, 21, 25, 26]:

- format **pojedynczej precyzji** (single) – jest to 32-bitowy format, w którym na mantysę (znormalizowaną z ukrytą jedynką przed przecinkiem) przeznaczono 23 bity, a na wykładnik – 8 bitów;
- format **podwójnej precyzji** (double) – 64-bitowy format, gdzie na mantysę zarezerwowano 52 bity, a na wykładnik przeznaczono 11 bitów;
- format o **rozszerzonej precyzji** do 128 bitów, gdzie na mantysę przeznaczono 111 bitów, a na wykładnik – 16 bitów.

Cechą charakterystyczną reprezentacji zmiennopozycyjnej jest stosunkowo duży odstęp pomiędzy pierwszą dodatnią i ujemną wartością kodowaną (tj. -2^{Emin}, $+2^{Emax}$). Przedział ten nazywa się **niedomiarem**. Co ciekawe, za pomocą wyżej omówionej reprezentacji liczb zmiennopozycyjnym, gdy mantysa jest znormalizowana i ma wartości z przedziału $1 \le m < 2$, zakodowanie zera jest niemożliwe – patrz zależność (1.10). Dlatego też, norma IEEE 754 określa pewne kody do specjalnych zastosowań. Są to kody, w których wykładnik ma wartość minimalną lub maksymalną, mianowicie [10, 16]:

- **zero** jest reprezentowane przez kod, w którym wszystkie bity wykładnika i mantysy są wyzerowane, przy czym wyróżnia się **zero dodatnie** (gdy bit znaku ma wartość 0) oraz **zero ujemne** (bit znaku – 1);
- **nieskończoność** jest zakodowana w ten sposób, że wszystkie bity wykładnika mają wartość 1, a bity mantysy – 0;

- **NaN** (z ang. *Not A Number*) – tzw. „nie-liczby”, np. wynik pierwiastka kwadratowego z liczby nieujemnej, są kodowane w ten sposób, że wszystkie bity wykładnika mają wartość 1, a mantysa jest różna od zera;
- **liczby bliskie zeru** koduje się w ten sposób, że bity wykładnika są wyzerowane, a mantysa posiada ukryte zero, jako cyfrę znaczącą przed przecinkiem. Przyjmuje się, że wartość mantysy we wzorze (1.9) zawarta jest w przedziale $0 \le m < 1$, a wykładnik e wynosi:

$$(e)_2 = -(2^{p-1} + 2)_{10}. \tag{1.17}$$

1.6. Definicja programowania

Pod pojęciem „**programowanie**” należy rozumieć zespół działań mających na celu rozwiązanie pewnej klasy problemów programistycznych. Działania te składają się na sformalizowany proces tworzenia programu. Jest on procesem logicznym, który powinien składać się z następujących podstawowych czynności [3, 9, 15÷17, 24, 28, 29]:

- **projektowanie** programu, które składa się z dwóch głównych etapów – najpierw planuje się program poprzez określenie jego ogólnej koncepcji i przeznaczenia, zdefiniowanie informacji wejściowych i wynikowych oraz opracowywanie logiki działań, a następnie w kolejnym etapie rozpoczyna się kompletowanie dokumentacji technicznej programu;
- **zapis** programu – na podstawie wcześniej sporządzonego projektu wybierany jest odpowiedni język programowania oraz jego edytor, po czym rozpoczyna się właściwy proces programowania, w którym powstaje kod źródłowy programu;
- **kompilacja** i **konsolidacja** programu;
- uruchomienie i testowanie programu, które ma na celu wyeliminowanie błędów powstałych w fazie zapisu programu;
- **dystrybucja** programu wraz z utworzoną dokumentacją użytkownika;
- **konserwacja** programu, która polega na wprowadzeniu poprawek lub eliminacji wykrytych błędów programu.

Podczas projektowania programu szczególną uwagę należy zwrócić na poprawność wykonania analizy sformułowanego problemu programistycznego. Powinna ona być ukierunkowana na określenie i wybór metod rozwiązania każdego elementarnego zadania. Należy zawsze dążyć do sprowadzenia danego problemu do już znanego wzorca projektowego oraz do zastosowania znanych algorytmów.

Pod pojęciem „**algorytm**” należy rozumieć metodę rozwiązania określonej klasy problemu w skończonej liczbie działań elementarnych. Jest on zapisywany

za pomocą języka programowania. Każdy algorytm powinien charakteryzować się następującymi cechami [1, 19, 28, 33]:

- jeśli algorytm posiada dane wejściowe, to pochodzą one z dobrze zdefiniowanego zbioru;
- efektem działania algorytmu zawsze jest wynik;
- algorytm jest precyzyjnie zdefiniowany, a działanie elementarne jest określone w sposób jednoznaczny;
- algorytm jest skończony, a wynik jest uzyskiwany w możliwie w najkrótszym czasie przy wykorzystaniu zasobów komputera w możliwie najmniejszej ilości.

Pod pojęciem „**uruchomienie**” należy rozumieć zespół czynności mających na celu lokalizację i usunięcie błędów w zapisie programu, które zaburzają prawidłowe jego działanie. Natomiast **testowanie** jest procesem badania programu zmierzającego do stwierdzenia, czy program spełnia określone kryteria. Najczęściej celem testowania programu jest określenie:

- zachowania się programu w różnych przewidywanych sytuacjach;
- poprawności uzyskiwanych wyników dla typowego, reprezentatywnego zbioru danych wejściowych;
- szybkości z jaką działa program.

Uruchomienie i testowanie programu komputerowego, są procesami wzajemnie uzupełniającymi się. W fazie zapisu programu, te dwie grupy czynności często przeplatają się.

Konserwacja programu wiąże się przede wszystkim z serwisem gwarancyjnym oraz ewentualnymi reedycjami programu komputerowego. W trakcie fazy testowania programu, programista nie jest w stanie przewidzieć wszystkich, szczególnie nietypowych sytuacji użytkowania oprogramowania. Jeżeli, pierwotna wersja programu nie jest w stanie obsłużyć nowej zaistniałej sytuacji, obowiązkiem programisty jest wykonanie odpowiedniej jego edycji. Łatwość wykonania tej edycji zależy w głównej mierze od sposobu zapisu algorytmu, tj. użytego języka programowania, stylu programowania i sposobu jego udokumentowania [9, 16, 33].

2. JĘZYKI PROGRAMOWANIA

2.1. Wprowadzenie

Język programowania jest to ustalony zestaw instrukcji, symboli, operatorów i wyrażeń wraz z regułami ich stosowania. W informatyce wyróżnia się wiele języków, które różnią się między sobą w zależności od ich przeznaczenia. Jednak wszystkie języki programowania charakteryzują się tym, że ich składnia jest ścisła i pozbawiona wszelkiej dowolności w jej stosowaniu. Jest to determinowane tym, że pisanie programów dla komputerów, które są tylko maszynami liczącymi, jest procesem złożonym i wymagającym ścisłych sformułowań.

Język programowania jest specjalistycznym **narzędziem** programisty. Służy on jedynie do zapisu programów komputerowych w formie zrozumiałej, zarówno dla programisty, jak i dla komputera.

W rozdziale tym przedstawiane są najważniejsze zagadnienia dotyczące języków programowania w kontekście programowania komputerów przez inżynierów mechaników. Celem tego rozdziału jest jedynie usystematyzowanie wiedzy teoretycznej na temat struktury i funkcjonowania języków programowania. Zakłada się, że czytelnik jest osobą, która już zna składnię dowolnego języka programowania w stopniu przynajmniej podstawowym.

2.2. Klasyfikacja języków programowania

W zależności od stopnia zaawansowania składni, wyróżnia się dwie podstawowe grupy języków programowania, mianowicie [8, 10, 11, 16]:

- **języki niskiego poziomu**, których składnia jest zbliżona (lub taka sama) do języka wewnętrznego (poleceń) maszyny cyfrowej (ściślej: procesora); jednej instrukcji elementarnej takiego języka odpowiada najczęściej jedna operacja elementarna procesora;
- **języki wysokiego poziomu**, które charakteryzują się rozbudowaną składnią zbliżoną w dużym stopniu do języka naturalnego człowieka, dzięki temu są one w pełni zrozumiałe dla programisty; wadą ich jest to, że jedna instrukcja elementarna zapisana w takim języku jest realizowana zazwyczaj przez bardzo dużą ilość operacji elementarnych procesora.

Języki niskiego poziomu

W informatyce funkcjonują tylko dwa podstawowe języki niskiego poziomu. Pierwszym z nich jest **język maszynowy** (rys. 2.1), w którym zapis programu wymaga użycia instrukcji w postaci liczb. Liczby te (zapisane zazwyczaj w systemie binarnym) są zarazem rozkazami, jak i danymi bezpośrednio pobieranymi przez procesor wykonujący program. Jak wynika z rys. 2.1, zapis programu według takiej składni jest trudnym zadaniem, a treść jest zrozumiała tylko i wyłącznie przez maszynę cyfrową (ściślej: tylko przez określony typ procesora). Programiści piszący program w języku wewnętrznym używają edytorów, w którym liczby są zapisywane w systemie szesnastkowym. W efekcie tego, składnia języka maszynowego przyjmuje formę symboliczną.

Drugim językiem z grupy języków niskiego poziomu jest **język asemblerowy** (asembler). Jego składnia jest łatwiejsza do opanowania przez człowieka. Każda instrukcja zapisana w asemblerze odpowiada jednemu rozkazowi wyrażonemu w kodzie maszynowym. Program napisany w asemblerze musi być skompilowany, czyli przetłumaczony do poziomu kodu maszynowego.

Składnia języka asemblerowego, w porównaniu z językiem maszynowym, jest bogatsza. Składa się ona z takich elementów jak: zbioru znaków alfanumerycznych (z podstawowego zakresu kodu ASCII), słów kluczowych, stałych (wyrażonych w dowolnym systemie liczbowym lub w postaci ciągu znaków alfanumerycznych), nazw symbolicznych oraz atrybutów symboli. Dzięki temu, programowanie w tym języku nie sprawia większych trudności, nawet inżynierowi mechanikowi.

Instrukcje, rozkazy oraz dyrektywy asemblera są zapisywane w jednej linii (rys. 2.2). Instrukcja składa się z etykiety, przedrostka, słowa kluczowego określającego typ instrukcji, argumentu oraz komentarza. Przy czym etykieta oraz

```
0000:  CD 28 00 A0 00 9A EE FE  1D F0 4F 03 06 1C 8A 03  .....
0010:  06 1C 17 03 06 1C BF 06  01 01 01 00 02 FF FF FF  .....
0020:  FF FF FF FF FF FF FF FF  FF FF FF FF E1 1B 4E 01  .....
0030:  C6 20 A4 00 10 00 A2 21  FF FF FF FF 00 00 00 00  .....
0040:  05 00 00 00 00 00 00 00  00 00 4F 06 1C 00 00 00  .....
0050:  CD 21 CB 00 00 00 20 20  20 20 20 EE 10 35 C1 B0  .....
0060:  00 00 00 00 00 20 20 20  20 20 FF FF FF FF FF FF  .....
0070:  CA A1 45 20 A1 C0 CA 00  00 00 00 B2 B2 F0 32 00  .....
```

Rys. 2.1. Zrzut ekranu zawierający fragment kodu napisanego w języku maszynowym, gdzie liczby podano w szesnastkowym systemie liczbowym – pierwsze pięć znaków w linii (np. 0010:) wyraża adres pierwszej komórki pamięci, natomiast pozostałe liczby reprezentują wartości, jakie są przechowywane w poszczególnych komórkach pamięci [11, 16]

```
 1:   .model small     ;komentarz
 2:   .stack
 3:   .data
 4:   a db 3ah
 5:   b db 19
 6:   c db ?
 7:   .code
 8:   start:
 9:   mov ax, @data    ;komentarz
10:   mov ds.., ax
11:   mov al.., a
12:   mov ah, b
13:   add ah, al..
14:   mov c, ah
15:   end
```

Rys. 2.2. Zrzut ekranu przedstawiający fragment kodu napisanego w asemblerze [11, 16]

komentarz są opcjonalne. Przykładem instrukcji może być pierwsza linia wydruku umieszczonego na rys. 2.2. Przykładowe rozkazy asemblera są umieszczone w liniach o etykietach 9÷14 omawianego wydruku. Każdy rozkaz musi składać się z kodu mnemonicznego (trzyliterowego symbolu) oraz z argumentów. Natomiast przykładem dyrektywy może być ostatnia linia wydruku (tj. nr 15).

Posługiwanie się językiem niskiego poziomu wymaga od programisty wiedzy oraz doświadczenia. Programowanie komputerów przy użyciu np. asemblera jest zadaniem czasochłonnym. Języki z tej grupy są adresowane głównie do informatyków zawodowo zajmujących się programowaniem komputerów. W praktyce bardzo rzadko do pisania programów wykorzystuje się tego typu języki. Jednak są przypadki, gdy napisanie fragmentu programu za pomocą języka niskiego poziomu jest wskazane. Przypadkiem takim jest sytuacja, kiedy szybkość działania programu jest priorytetem.

Języki wysokiego poziomu

Wady języków niskiego poziomu (np. trudność nauki składni, wydłużony czas programowania, trudność edycji napisanego programu itd.) wymusiły opracowanie języków wysokiego poziomu. Dodatkowym impulsem do rozwoju języków programowania była ewolucja komputerów. Począwszy od III generacji maszyn cyfrowych, gdy stopień wykorzystania komputerów w przedsiębiorstwach stawał się znaczący, pojawiła się potrzeba usystematyzowania technik programowania oraz stworzenia języka, który byłby zbliżony składnią do języka naturalnego człowiekowi. Głównym wykładnikiem języków wyższego poziomu jest ich efektywność w procesie pisania programów oraz łatwość edycji (rozbudowy) tak napisanej aplikacji w późniejszym czasie. Niestety, wspomniane zalety języków wysokiego poziomu wpływają na zmniejszenie szybkości działania tak napisanych programu w porównaniu do szybkości działania programów napisanych bezpośrednio w kodzie maszynowym.

Jedną z podstawowych klasyfikacji języków wysokiego poziomu jest podział ze względu na ich możliwości programistycznie. W myśl tego podziału wyróżnia się języki programowania wysokiego poziomu, które mają główne zastosowanie w procesie programowania typu [11, 16, 19, 24, 29, 30, 33]:

- proceduralnego i/lub strukturalnego – ogólnie nazywanego jako **imperatywnego**;
- zorientowanego obiektowo (w skrócie: **obiektowego**);
- **specjalizowanego**, np. funkcyjnego, logicznego, sterowania przepływem danych, sterowania zdarzeniami, współbieżnego itp.

Kolejną klasyfikacją języków programowania, jaką można przytoczyć, jest podział języków ze względu na stopień samodokumentowania kodu źródłowego. Można wyróżnić języki programowania [10, 16, 24]:

- **symboliczne**, nazwy instrukcji mają postać skrótów, a do opisu zasady działania stosuje się dodatkowe komentarze;
- **opisowe**, nazwy instrukcji są wyrazami wyjaśniającymi jednocześnie jej przeznaczenie i zasadę działania.

Z tą drugą grupą języków wiąże się idea języków samodokumentujących się. Niestety praktyka wykazała, że nie jest możliwe stworzenie języka programowania, który byłby w stu procentach samodokumentującym. Obecnie wszystkie języki programowania umożliwiają stosowanie komentarzy, które pełnią rolę szczegółowego opisu programu.

2.3. Zarys historii rozwoju języków wysokiego poziomu

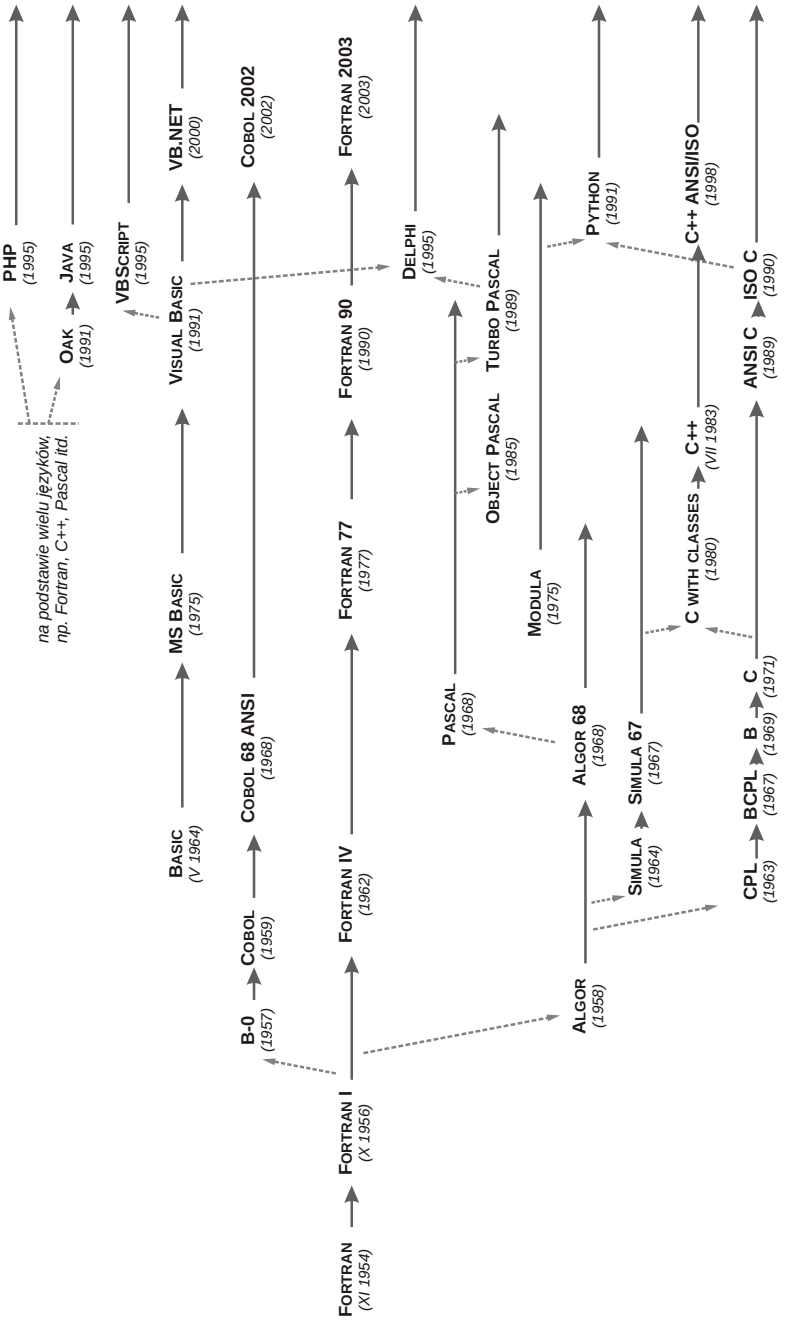
Historia programowania komputerów przy użyciu języka wysokiego poziomu rozpoczyna się w 1954 roku. Zespół pod kierunkiem Johna Backus’a, pracujący ówcześnie pod szyldem firmy IBM®, opracował pierwszy historyczny kompilator dla języka programowania o nazwie **FORTRAN** (z ang. *Formula Translator*). Kompilator ten jest używany (i rozwijany) po dzień dzisiejszy z uwagi na jego szczególne cechy. Kod maszynowy wygenerowany przez ten kompilator odznacza się nadzwyczaj wysoką jakością, stabilnością i szybkością obliczeń numerycznych. W szczególności dotyczy to operacji arytmetycznych wykonywanych na liczbach zmiennoprzecinkowych o dużej precyzji [7, 8, 10].

Już pierwsza wersja kompilatora była zoptymalizowana w wysokim stopniu. Powodem tak starannego opracowania kompilatora FORTRAN były obawy Johna Backus’a, że język ten nie znajdzie użytkowników, jeżeli efektywność napisanego programu w Fortranie nie będzie porównywalna z programami napisanymi w asemblerze, który w tamtych czasach był używany powszechnie.

Formalny standard tego języka programowania opracowano dopiero w roku 1962 (rys. 2.3). Regulacja składni dotyczyła wersji języka o nazwie Fortran IV. Kolejną wersję standardu tego języka, niestety nadal o ubogiej składni, opubli-

kowano w roku 1966 (Fortran 66). Dopiero w roku 1977, amerykańska organizacja standaryzująca ANSI opracowała znormalizowaną składnię języka pod nazwą Fortran 77 (norma ANSI X3.9-1978), która była stosowana od roku 1980 na całym świecie jako jednolita forma języka programowania. Kolejnym znaczącym etapem rozwoju tego języka jest powstanie nowej wersji Fortran 90, którego składnia zostaje dostosowana do powszechnie używanych ówczesnych języków programowania. Co ciekawe, dopiero od 2003 roku w składni tego języka pojawiają się wbudowane funkcje graficzne, służące m.in. do prezentacji wyników obliczeń. Obecnie, programy pisane w Fortranie są przeznaczenie do wykonywania zaawansowanych i dokładnych obliczeń inżynierskich – głównie z zakresu aerodynamiki, wytrzymałości materiałów itp. Programy napisane w tym języku są fragmentami aplikacji inżynierskich i stanowią jej tzw. *solver* (podprogram obliczeniowy). Natomiast interfejs graficzny, obsługa strumieni oraz ogólne procedury takiej aplikacji są tworzone przy użyciu innego, uniwersalnego języka programowania wysokiego poziomu.

Pierwsza wersja języka Fortran była przykładem **języka symbolicznego imperatywnego** [10, 16, 19], przeznaczonego głównie do tworzenia programów wspomagających pracę naukową. Jednak, żadna gospodarka (przemysł) nie może funkcjonować optymalnie, jeżeli jest pozbawiona programów wspomagających obliczenia i analizy z zakresu ekonomii. Zastosowanie Fortranu do programowania w obszarze ekonomi sprawiało trudności – napisanie programu w tym języku nie było efektywne. Taka sytuacja sprawiła, że pod koniec lat pięćdziesiątych ubiegłego wieku rozpoczęto prace projektowe nad nowym językiem programowania wysokiego poziomu, tzw. „dla zastosowań biznesowych" (B-0 – rys. 2.3). Efektem prac prowadzonych w tym kierunku było sformułowanie składni języka programowania o nazwie **COBOL** (*z ang. COmmon Business Oriented Language*). Twórcą tego języka był zespół programistów kierowany przez Grace'a Hopper'a, Admirała Marynarki Wojennej USA. Ideą towarzyszącą przy projektowaniu COBOL'a była chęć stworzenia języka samodokumentującego się (niestety nie osiągnięto tego celu w zamierzonym stopniu), który przeznaczony byłby do programowania pod kątem przetwarzania danych handlowo-biznesowych, transakcji towarowo-pieniężnych oraz danych personalnych. Składnia tego języka została znormalizowana przez organizację ANSI dopiero w 1968 roku. Cechą charakterystyczną standardu COBOL'a jest rozbudowany kod źródłowy, który jest podzielony na cztery części. Kolejno występuje część identyfikacyjna, konfiguracyjna, zawierająca dane oraz zawierająca procedury (część proceduralna). Składnia tego języka pozbawiona jest skrótów, co jest konsekwencją próby wdrożenia idei samodokumentowania zapisu programu. Język COBOL jest przykładem **języka opisowego proceduralnego**. Opisowość tego języka sprawia, że kod źródłowy jest w pełni zrozumiały również dla „nie-programisty" [7, 8].



Rys. 2.3. Schemat przedstawiający historię rozwoju wybranych języków wysokiego poziomu [7],
gdzie: → oś czasu, ⇢ transfer wybranych cech

Trzecim ważnym językiem programowania (z punktu widzenia historii i rozwoju informatyki) jest ALGOR. Został on opracowany w 1958 roku przez międzynarodowy zespół informatyków, w skład którego wchodził Peter Naur oraz John Bachus (twórca języka Fortran). Język ALGOR powstał bazując na wybranych regułach notacji języka Fortran, które P. Naur zmodyfikował w ten sposób, aby składnia nowopowstałego języka mogła być użyta do wspomagania opisu algorytmów w pracach naukowych. W ramach składni ALGOR'a, po raz pierwszy wprowadzono do języków programowania takie mechanizmy oraz elementy programistyczne jak: rekurencja, tablice dynamiczne, typy danych użytkownika, instrukcje blokowe oraz przekazywanie parametrów do procedury.

Od momentu pojawienia się pierwszych języków programowania, nie było już odwrotu od dalszego rozwoju zasad pisania oprogramowania specjalistycznego, systemów operacyjnych oraz rozwoju techniki komputerowej. Prawie każda gałąź przemysłu wyrażała chęć rozwijania specjalistycznego języka programowania, który będzie stosowany tylko do rozwiązywania zadań typowych dla danej dziedziny przemysłu. Tym szczególnym okresem były lata 60 i 70 ubiegłego wieku, w którym również zaczęto zastanawiać się nad optymalizacją zasad programowania, szczególnie programów służących do rozwiązywania złożonych problemów inżynierskich. W tym czasie pojawił się pierwszy język służący do programowania obiektowego – **Simula** (1964). Język ten stanowił rozszerzenie języka Algor 60 i został opracowany przez Norweski Ośrodek Obliczeniowy w Oslo. W 1967 roku powstał język Simula 67, który był użytkowany do lat osiemdziesiątych XX wieku. Simula był językiem pionierskim, stosowanym do programowania symulacyjnego. Do jego składni wprowadzono po raz pierwszy pojęcia klasy oraz obiektu, co zapoczątkowano programowanie zorientowane obiektowo. Było to związane z koniecznością wykonywania skomplikowanych obliczeń inżynierskich w trakcie projektowania statków morskich. Forma strukturalna i proceduralna ówczesnych języków powodowała niemałe kłopoty implementacji zaawansowanych obliczeń inżynierskich. Natomiast obiekt (tj. struktura zgrupowanych danych i procedur) okazał się idealnym sposobem rozwiązywania skomplikowanych problemów, dla których podejście obiektowe było naturalne [7, 8].

Równolegle do stworzenia pierwszego języka obiektowego, w Dartmouth College opracowano (przez wykładowców Johna Kemmeny'ego i Thomasa Kurtz'a, 1964 r.) pierwszy prosty i uniwersalny język programowania o nazwie **Basic** (*z ang. Beginners All-purpose Symbolic Instruction Code*) [14, 15, 17]. Język ten był przeznaczony do szybkiego (w czasie studiów) nauczania podstaw programowania. W procesie dydaktycznym nie można było skutecznie zastosować ówczesnych języków takich jak np. Fortran, ponieważ ich składnia była trudna do opanowania w krótkim czasie.

Początkowo, język Basic działał tylko w środowisku interpretera. Dzięki takiemu uproszczeniu, student mógł się skoncentrować tylko na samym procesie programowania pomijając kwestie kompilacji kodu źródłowego, które były nieistotne w początkowym etapie edukacji. Ponieważ program interpretowany działa wolniej od skompilowanego, zdecydowano się również na opracowanie kompilatora dla języka Basic. Począwszy od 1975 roku, w rozwój tego języka zaangażowała się firma Microsoft®. Firma ta, najpierw wprowadziła na rynek standard MS Basic [17], a następnie Visual Basic (1991, w skrócie: VB), który umożliwia programowanie w środowisku graficznym systemu Windows®. Obecnie, Visual Basic funkcjonuje jako język kompilowany (programowanie programów dla systemu operacyjnego Windows) oraz język interpretowany, który jest używany do programowania makr w aplikacjach (Visual Basic for Application, w skrócie: VBA) [14, 25, 26].

Struktura i składnia pierwszych wersji języka Visual Basic była nieskomplikowana, charakteryzująca się prostotą – mimo to, jego możliwości na ówczesne czasy oceniano wysoko. Początkowo, był to język typowo symboliczny imperatywny. Dopiero od wersji oznaczonej cyfrą 4, składnię tego języka wzbogacono w klasy – niestety ich budowa była uproszczona i nie zapewniała pełnego obiektowego podejścia w programowaniu. Ale za to, istniała możliwość zaawansowanego programowania strukturalnego. Wprowadzenie klas do Visual Basic oraz zwiększanie ilości wbudowanych funkcji znacząco komplikowało składnię tego języka. Z drugiej strony, dodane nowe elementy funkcyjne do standardu języka zwiększały jego funkcjonalność, szczególnie w obrębie operacji na ciągach znaków, obsługi baz danych, strumieni danych oraz budowy niestandardowych elementów interfejsu użytkownika (tzw. kontrolek – obiektów graficznych). Rozwój tego języka, w którym przeważały elementy strukturalne, zakończono na wersji o numerze 6. Najnowsza odsłona tego języka, która miała premierę w lipcu 2000 roku, otrzymała nowy standard bazujący na platformie .NET [4, 15, 25]. Wymiernym efektem wykonanej ewolucji było powstanie Visual Basic’a w pełni obiektowego, mającego bardzo szerokie pole zastosowania – przy czym, nadal jest językiem uniwersalnym. Niestety, przekształcenie Visiual Basic’a w zaawansowane narzędzie programistyczne zostało okupione utratą dotychczasowej cechy charakterystycznej tego języka – tj. prostoty jego składni.

2.4. Języki wysokiego poziomu

2.4.1. Charakterystyka ogólna

Program komputerowy w postaci kodu źródłowego ma postać tekstu napisanego przez programistę. Forma zapisu tego tekstu ma istotne znaczenie, a reguły poprawności składni są ściśle i jednoznacznie określone. Każdy najdrobniejszy

element programu ma określone znaczenie i sens. Do zapisu treści programu najczęściej używa się liter z alfabetu języka angielskiego, cyfr arabskich oraz znaków specjalnych. Znaki specjalne mogą być dopełnieniem formy zapisu –np. mają za zadanie uzupełnić lub uwypuklić treść poszczególnych słów. Mogą one również pełnić rolę samodzielnych elementów składni języka, np. uczestniczą w zapisie operacji arytmetycznych lub logicznych. W zapisie programu również używa się spacji. Jednak ich występowanie ma na celu jedynie sprawienie, aby treść programu była czytelna dla człowieka. Przykładowo, grupa spacji na początku linii tworzy jej wcięcie, co zapewnia lepszą czytelność programu. Jednocześnie należy pamiętać, że spacje są tzw. „znakami białymi" i są pomijane podczas translacji kodu źródłowego.

Przeprowadzając syntezę budowy składni dowolnego języka wysokiego poziomu, można wyodrębnić następujące pojęcia charakterystyczne [9, 10, 16, 25, 26, 31, 32]:

- **nazwa** – jest to unikalny identyfikator dowolnego obiektu występującego w programie i składa się z ciągu znaków alfanumerycznych; nazwa nie może zawierać żadnych znaków specjalnych oprócz znaku „podkreślenia" (_) i nie może zaczynać się od cyfry;
- **słowo kluczowe** – jest to wyodrębniona jednostka leksykalna danego języka programowania mająca określone znaczenie; jest ono używane do formułowania instrukcji (np. deklaracji zmiennych, iteracji itp.);
- **instrukcja** – jest to złożona jednostka składni stanowiąca podstawę tekstu programu, która jest poleceniem wykonania określonych operacji przez komputer; można przyjąć, że jest ona analogią do zdania w języku naturalnym; instrukcja ma określoną postać i znaczenie i może zajmować jeden lub kilka wierszy kodu;
- **segment** (blok) – jest to fragment programu zawierający kilka instrukcji i jest tworzony głównie w celu uporządkowania treści programu; przykładem może być blok instrukcji tworzących procedurę lub funkcję;
- **zmienna** – jest abstrakcyjnym obiektem programu reprezentującym miejsce w pamięci komputera, gdzie przechowywane są dane; instancja zmiennej posiada nazwę, za pomocą której dokonuje się odwołania do niej, oraz wartość należąca do określonej dziedziny – tj. typu; ze względu na konstrukcję komputera, wartości zmiennej mogą pochodzić tylko z ograniczonego zbioru wartości określonego przez dany typ;
- **stała** – jest to szczególny rodzaj zmiennej, której wartość ustalona w fazie pisania kodu źródłowego nie ulega zmianie w trakcie wykonywania programu;
- **tablica** – jest to uporządkowana i poindeksowana grupa zmiennych jednakowego typu; wyróżnia się tablice jedno- i wielowymiarowe (więcej informacji na temat tablicy, jako struktury danych zawiera rozdział 5);

- **wyrażenie** – jest to zapis operacji, np. algebraicznych lub logicznych, które są wykonywane na zmiennych określonego typu; wyrażenie może zawierać zmienne, stałe, operatory działania, wywołania funkcji oraz znaki grupujące, np. nawiasy; wyróżnia się **wyrażenia arytmetyczne** – złożone tylko z elementów typu liczbowego, oraz **wyrażenia znakowe**, które zawierają tylko elementy typu znakowego;
- **funkcja** – jest to fragment programu uporządkowany w segment; funkcja posiada nazwę, listę argumentów określonego typu, a wynikiem jej działania jest zwracana wartość określonego typu; funkcja jest wywołana poprzez podanie jej nazwy oraz wartości argumentów, które są ujęte w nawiasy tuż po nazwie;
- **funkcja standardowa** (wbudowana) – jest to funkcja predefiniowana w ramach składni języka programowania i automatycznie dostępna dla programisty; w nowszych wersjach języka VB .NET [25] całkowicie zrezygnowano z takiego rozwiązania na rzecz metod zawartych w predefiniowanych klasach;
- **procedura** – jest to szczególny rodzaj funkcji, której wywołanie nie powoduje wygenerowania zwracanej wartości; celem procedury jest jedynie uporządkowanie fragmentu kodu, który jest wielokrotnie wykorzystywany w różnych fragmentach programu.

Nieodzowną cechą algorytmów, szczególnie numerycznych, jest wykonywanie obliczeń w sposób iteracyjny lub w oparciu o określony warunek. Aby programista mógł zapisać taki algorytm w dowolnym języku, to składnia tego języka programowania powinna umożliwiać łatwą budowę odpowiedniej struktury (w sensie funkcyjnego układu instrukcji, słów kluczowych i wyrażeń traktowanych jako integralną jednostkę składni języka). Zatem, wyróżnić można następujące najważniejsze konstrukcje składni języków wysokiego poziomu [17, 30]:

- **struktura decyzyjna** – jest to taka konstrukcja składni języka, która umożliwia wykonanie jednego z dwóch alternatywnych segmentów kodu źródłowego w oparciu o instrukcję warunkową;
- **struktura cyklu** – jest to taka struktura, która umożliwia wykonywanie w sposób iteracyjny określonego segmentu kodu źródłowego; struktura ta zawiera instrukcję warunkową zakończenia iteracji oraz instrukcję modyfikującą element testowany w warunku; instrukcje te muszą być tak napisane, aby ilość iteracji była skończona, przy czym elementem testowanym najczęściej jest zmienna w wyrażeniu boolean’owskim;
- **struktura obsługi wyjątku** – jest to taka konstrukcja, która składa się z instrukcji przechwytującej tzw. wyjątek (tj. błąd wykonania programu) oraz instrukcji przekierowania wykonywania programu do segmentu obsługującego dany wyjątek; brak takiej struktury kończy się zazwyczaj niekontrolowanym zakończeniem działania programu.

Ważnym elementem składowym każdego programu jest operacja. Obliczenia wykonywane przez procesor sprowadzają się do algorytmicznego przetwarzania danych wejściowych w informację wyjściową. Algorytm ten jest przedstawiony w postaci programu, a więc segmentów instrukcji, określających operację. Zatem **operacją** komputera nazywa się zbiór elementarnych czynności, wykonywanych przez procesor według ściśle określonych zasad i według ściśle określonego porządku [5, 9, 28].

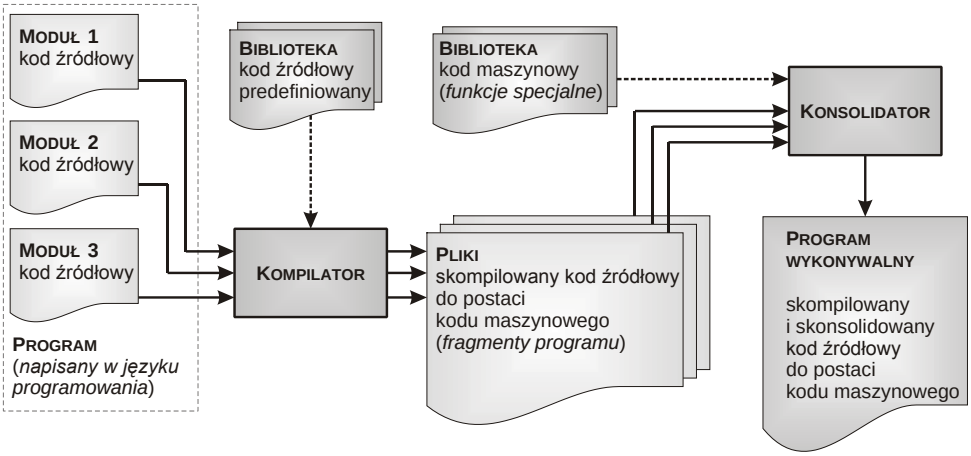
2.4.2. Translacja

Ponieważ składnia języków programowania wyższego poziomu jest niezrozumiała dla maszyn cyfrowych, program napisany w języku tego typu musi zostać przekształcony do postaci kodu maszynowego. Do tego celu używany jest **kompilator** lub **interpreter** (interpretator), a taki proces przekształcenia zapisu programu nazywa się ogólnie **translacją** kodu źródłowego na kod maszynowy [16, 17, 28].

Kompilator jest programem komputerowym stanowiącym integralną część każdego języka programowania. Służy on do kompilacji, czyli translacji całego kodu źródłowego programu, napisanego w języku wyższego poziomu (lub w asemblerze), do postaci zrozumiałej przez dany typ procesora – tj. właściwego kodu maszynowego. Wynik pracy kompilatora jest zawsze zapisywany na nośniku danych w formie pliku wykonywalnego. Dzięki temu, zapisany program może zostać w dowolnym momencie uruchomiony w środowisku operacyjnym – a ściślej: wykonany bezpośrednio przez procesor. W przypadku systemu operacyjnego z rodziny Windows®, plik wykonywalny jest plikiem o rozszerzeniu EXE. Natomiast w systemie operacyjnym Unix®, program jest zapisany w pliku typu X, dla którego ustawiono tzw. „tryb wykonywalności".

Programy komputerowe pisane w języku wyższego poziomu posiadają zazwyczaj budowę modułową (z racji swojej dużej objętości). Kod źródłowy programu jest podzielony na moduły i umieszczony w osobnych plikach tekstowych. W osobnych modułach są umieszczane również dodatkowe fragmenty programu, np. procedury predefiniowane, które są dołączane do programu w trakcie kompilacji. Pliki tekstowe przechowujące wspomniane predefiniowane fragmenty programu nazywane są **bibliotekami** (rys. 2.4).

Kompilacji poddaje się każdy moduł z osobna, a uzyskany kod maszynowy jest umieszczany w oddzielnych plikach tzw. „przejściowych". Aby uzyskać program komputerowy w postaci jednego pliku wykonywalnego, proces kompilacji musi zostać dopełniony konsolidacją plików przejściowych. Do tego celu jest stosowany **konsolidator** – dodatkowy program służący do połączenia ze sobą wszystkich fragmentów programu, które uprzednio przekształcono do postaci kodu maszynowego, a następnie zapisania efektu tej operacji do jednego



Rys. 2.4. Schemat procesu kompilacji i konsolidacji programu – opis w tekście

pliku wykonywalnego. Konsolidator może również dołączać z bibliotek dodatkowe, wcześniej skompilowane funkcje specjalne (tj. fragmenty kodu maszynowego). Zazwyczaj są to funkcje przeznaczone do zarządzania bazami danych, szybkiej obsługi grafiki, przesyłania danych pomiędzy urządzeniami peryferyjnymi a komputerem, obsługą urządzeń peryferyjnych itd.

Jednak nie zawsze program, napisany w języku wyższego poziomu, jest w całości kompilowany do poziomu kodu maszynowego i następnie zapisywany jako plik wykonywalny. Program może również mieć postać kodu źródłowego umieszczonego w pliku nazywanym **skryptem powłoki** lub może być podprogramem (makrem) osadzonym w dokumencie dowolnej aplikacji np. MS Excel, MS Word, CorelDRAW, Solid Edge, MathCAD, AutoCAD itd. W takim przypadku uruchomienie i wykonanie takiego programu wymaga użycia interpretera.

Interpreter jest programem, który przekształca kod źródłowy na kod maszynowy w sposób sekwencyjny. Proces ten polega na tym, że każda instrukcja programu jest tłumaczona z osobna, a powstały kod maszynowy reprezentujący daną instrukcję jest umieszczany bezpośrednio w pamięci podręcznej komputera (RAM). Następnie jest on bezzwłocznie wykonywany przez procesor. Niestety, ponowne wykonanie całego programu wymaga ponownego użycia interpretera. W efekcie tego program wykonywany w taki sposób jest znacząco wolniejszy, niż jego odpowiednik wcześniej skompilowany.

Translatory potrafią również same wykrywać niektóre z błędów popełnionych przez programistę. Są to błędy [17]:

- **translacji**, którymi są nieprawidłowe postacie składni kodu źródłowego; często ich wagę porównuje się do błędów ortograficznych dla języka naturalnego;
- **wykonania**, które powstają podczas wykonywania programu, gdy nie można wykonać wymaganej operacji; przykładem jest wyrażenie, w którym próbuje się wykonać działanie dzielenia liczby przez zero.

Każdy zapis programu może posiadać dodatkowo **błędy logiczne**, które nie są wykrywane przez translator. Są to zazwyczaj błędy popełnione przez programistę i objawiają się tym, że pomimo pozornie poprawnego działania programu, wygenerowane wyniki są niepoprawne. Często błędy ukryte są ujawniane dopiero w fazie użytkowania programu i mogą zostać usunięte tylko i wyłącznie przez programistę.

3. TECHNIKI DOKUMENTOWANIA PROGRAMU

3.1. Wprowadzenie

Pod pojęciem „**dokumentowanie programu**” należy rozumieć zespół czynności mających na celu utworzenie kompletnego opisu tworzonego programu. Do tego celu używa się zarówno prostych metod opisowych jak i zaawansowanych technik dokumentowania. Efektem wszystkich czynności jest **dokumentacja**, która składa się z dwóch odmiennych części. Są to:

- dokumentacja (podręcznik) użytkownika,
- dokumentacja techniczna.

W ramach wstępu do rozdziału zostaną podane ogólne informacje o zawartości pełnej dokumentacji spostrzeganej jako dokument. Natomiast w kolejnych podrozdziałach omówiono jedynie ważniejsze techniki sporządzania opisu funkcjonalnego programu (podprogramu, modułu, algorytmu). Opis ten wykonuje się w ramach dokumentacji technicznej.

Dokumentacja użytkownika

Dokumentacja użytkownika jest dokumentem przeznaczonym dla przyszłych użytkowników programu. Powinna ona zawierać podstawowe informacje o programie komputerowym włącznie ze szczegółowym opisem sposobu jego użytkowania. Dokument ten ma formę opisową, a jego typowy układ (w formie rozdziałów i podrozdziałów) jest następujący [17, 26, 28]:

- **strona tytułowa**, która powinna zawierać informacje ogólne, takie jak: nazwa programu, wersja programu, autor dokumentu, jego data opracowania oraz dane kontaktowe z twórcą programu;
- **spis treści** dokumentu;
- krótki **opis programu** w formie streszczenia;
- **charakterystyka problemu**, który jest rozwiązywany przez program; treść tego rozdziału obejmuje: teoretyczne wprowadzenie, opis notacji stosowanej w dokumentacji, dokładne sformułowanie problemu oraz obszar zastosowania programu;
- **środowisko programu**, opisane za pomocą pojęć: dane wejściowe i uzyskiwane wyniki (opis danych i wyników sprowadza się do podania ich znaczenia, formatu, sposobu wprowadzania lub odczytu, warunków poprawno-

ści oraz reakcji na błędy), sposób komunikacji programu z użytkownikiem, oraz przykłady;
- **sytuacje niepoprawne**, w szczególności wykaz komunikatów diagnostycznych oraz opis błędów wykonania programu wraz z podaniem warunków, w jakich mogą powstać błędy;
- **literatura**.

Dokumentację użytkownika można również wykonać w formacie elektronicznym. Efektem wymiernym takiego rozwiązania jest to, że do każdego z tematów dokumentacji można przypisać odwołanie z poziomu programu. Jednak takie podejście należy uwzględnić już na etapie projektowania i zapisu programu. Często wymaga to użycia tzw. edytorów plików pomocy [26].

Dokumentacja techniczna

Dokumentacja techniczna jest najważniejszym dokumentem, który jest przeznaczony dla programisty. Powstaje ona w trakcie całego procesu programowania. Na podstawie dokumentacji programista wykonuje zapis programu (przy użyciu języka programowania) oraz późniejszą jego edycję (w ramach konserwacji). Zatem, powinna ona zawierać pełny zestaw szczegółowych informacji dotyczących budowy i działania programu.

Dokument ten sporządza się w formie papierowej, a jego układ (uporządkowany w rozdziały i podrozdziały) powinien składać się z następujących elementów [4, 11, 16, 17, 23, 28, 33]:
- **strona tytułowa**, która powinna zawierać informacje ogólne, takie jak: nazwa programu, autor dokumentu oraz data opracowania dokumentacji;
- **spis treści**;
- **charakterystyka programu** zawierająca: krótki opis programu, jego przeznaczenie oraz obszar jego zastosowania;
- **struktura programu** – rozdział ten powinien zawierać: opis plików zewnętrznych (ich organizacja i użycie), globalne struktury danych (opis, przeznaczenie, użytkownicy), podział programu na moduły wraz ze schematami komunikacji między modułami, wykaz używanych modułów systemu operacyjnego;
- **opis modułu** (każdego z osobna), podprogramu lub programu – powinien składać się z następujących podrozdziałów:
  - **informacje ogólne** (zwięzła charakterystyka danego modułu);
  - **opis funkcjonalny modułu**, w szczególności: przeznaczenie tego modułu, sposób jego wykorzystania (wykaz procedur i ich przeznaczenie, przekazywane parametry do modułu, sekwencja wywoływania procedur oraz lista innych modułów, które korzystają z zasobów opisywane-

go modułu), korzystanie z zasobów innych modułów oraz globalnych struktur danych;
  - **charakterystyka działania modułu**, która obejmuje: szczegółowy opis algorytmów, lokalne struktury danych, budowa modułu (procedury pomocnicze oraz ich przeznaczenie, parametry, korzystanie z nielokalnych struktur danych), wymagania czasowe i pamięciowe modułu;
  - **sytuacje niepoprawne**, takie jak: kontrola poprawności danych wejściowych, wykaz komunikatów diagnostycznych, błędy wykonania (opis błędu, warunki jego powstania);
- **literatura**;
- **załączniki**, które tworzy się po zakończeniu etapu uruchomienia i testowania programu – są to:
  - informacje o przebiegu testowania modułów (programu),
  - przykładowe testy programu (dane testowe oraz omówienie testu),
  - wydruki programu (kod źródłowy ostatecznej wersji programu).

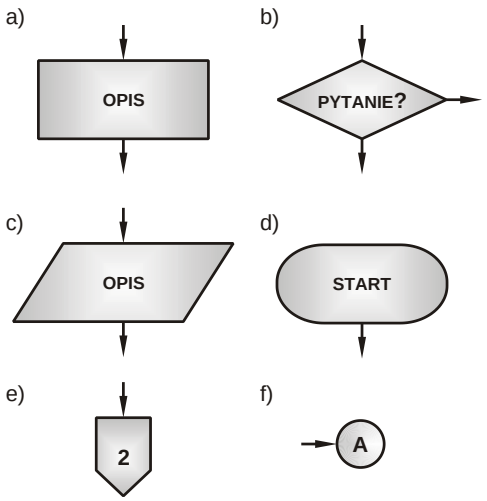
Do tworzenia dokumentacji technicznej programu, szczególnie w fazie projektowania, wykorzystuje się trzy zasadnicze techniki dokumentowania (projektowania) programu komputerowego, które zostaną scharakteryzowane w kolejnych podrozdziałach. Są to [5, 9, 13, 16, 23]:
- sieć działań (schemat blokowy),
- pseudo-kod,
- język modelowania wizualnego (UML).

Dokumentacja techniczna programu składa się zarówno z elementów utworzonych wyżej wymienionymi metodami dokumentowania, jak i z szeregu opisów w formie tekstowej. Warto podkreślić, że dokumentowanie programu odbywa się również na etapie pisania kodu źródłowego. Jest to realizowane poprzez umieszczanie w kodzie źródłowym **komentarzy** (tj. opisów tekstowych, które są formalnym elementem składni języka) oraz stosuje się odpowiednie konwencje notacji i formatowania (np. wcięcia linii, odstępy itd.). Typowymi miejscami kodu źródłowego, w których powinno się zamieszczać komentarze, są [16, 25, 26]:
- początek programu lub modułu (autor, data oraz krótki opis programu);
- miejsce deklaracji ważniejszych zmiennych i obiektów (zwięzły opis);
- deklaracje procedur, funkcji i klas (przeznaczenie, opis parametrów, wykaz używanych zmiennych globalnych z zaznaczeniem sposobu korzystania z nich, założenia dotyczące poprawności wywołania);
- wyróżnione fragmenty programu, np. wczytywanie i zapis danych, znalezienie elementu optymalnego, zmiana stanu programu (objaśnienia);
- ważniejsze struktury (instrukcje) iteracyjne i decyzyjne (wyjaśnienie znaczenia danej instrukcji).

3.2. Sieć działań

Technika dokumentowania za pomocą **sieci działań** jest metodą graficzną, polegającą na opisie działania programu (algorytmu) za pomocą grafu złożonego z odpowiednich bloków, łączników oraz linii zakończonych grotami. Poszczególne bloki oznaczają działania wykonywane przez program, linie wskazują kolejność wykonywanych działań lub kierunek przepływu strumieni danych, natomiast łączniki stanowią punkt przeniesienia przepływu logicznego reprezentowanego przez linię. Typowe symbole graficzne, stosowane w tej metodzie, mają następuje znaczenie i postać [5, 16]:

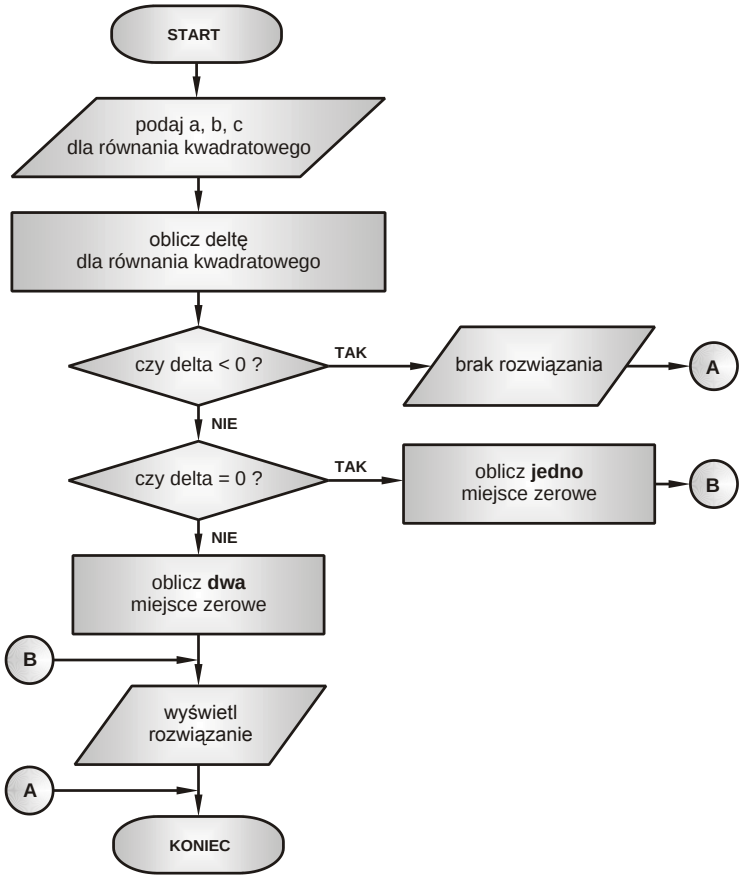
- **akcja do wykonania** – jest to blok prostokątny zawierający opis działania (reprezentowany przez segment kodu źródłowego), gdzie opis jest najczęściej w formie symbolicznej (rys. 3.1a);
- **warunek do zbadania** – ma postać bloku romboidalnego służącego do opisu struktury decyzji, dlatego też jest to jedyny symbol mający więcej niż jedno wyjście (rys. 3.1b); opis zawarty wewnątrz tego bloku ma najczęściej formę pytania, na które odpowiedzią jest „tak” albo „nie”;
- **operacja I/O** (z ang. *Input/Output*), jest przedstawiana za pomocą bloku w postaci równoległoboku opisującego działanie składające się z instrukcji wprowadzenia lub wyprowadzenia danych (rys. 3.1c);
- **miejsce terminalne** – blok owalny przedstawiający miejsce rozpoczęcia, zakończenia lub przerwania działania programu (rys. 3.1d), opis tego bloku sprowadza się do jednego słowa, odpowiednio: „start”, „koniec” lub „stop”;
- **łącznik międzystronicowy** (rys. 3.1e) – jest to symbol stanowiący początek lub zakończenie linii przepływu danych i jest stosowany, gdy dokumentacja programu zawarta jest na co najmniej dwóch stronach, wtedy opis jego ma formę etykiety wskazującej na numer odpowiedniej strony dokumentacji;
- **łącznik przepływów** (rys. 3.1f) – jest to symbol o kształ-



Rys. 3.1. Podstawowe symbole używane do budowy sieci działań – opis w tekście

cie okręgu, który umieszcza się w miejscu przerwania linii przepływu danych; symbol ten zawiera etykietę w postaci litery.

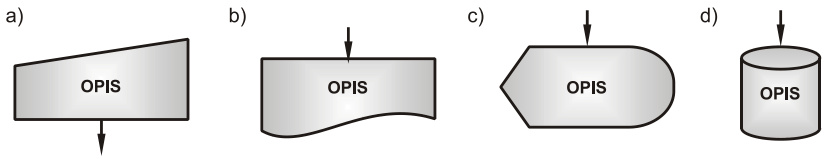
Na rys. 3.2 umieszczono przykładowy schemat blokowy (sieć działań), który wyjaśnia działanie algorytmu wyznaczenia rzeczywistych miejsc zerowych dla równania kwadratowego o postaci $ax^2+bx+c=0$. Podczas budowy schematu blokowego należy kierować się następującymi wytycznymi:



Rys. 3.2. Schemat blokowy utworzony w oparciu o sieć działań, który przedstawia graficzny zapis algorytmu wyznaczenia rzeczywistych miejsc zerowych dla równania kwadratowego; łączniki przepływu oznaczone etykietami „A" i „B" wyjaśniają ideę ich stosowania (tj. poprawiają czytelność schematu)

- sieć działań jest budowana w oparciu o standardowe symbole graficzne,
- ciąg logiczny działań powinien przebiegać z góry na dół oraz od lewej do prawej,
- opisy działań opisywane są przejrzystym i potocznym językiem.

Oczywiście są dopuszczalne odstępstwa od tych reguł, jednak przy zachowaniu zasady, aby sieć działań była możliwie prosta, przejrzysta i zrozumiała. Dopuszczalne są również alternatywne symbole dla operacji I/O (rys. 3.1c). Symbole te przedstawiono na rys. 3.3 i reprezentują określone rodzaje instrukcji wprowadzania i wyprowadzania danych.



Rys. 3.3. Odmiany symbolu operacji I/O: a) ręczne wprowadzanie danych, b) dokument, c) wyprowadzenie danych na ekran, d) dysk magnetyczny (nośnik danych)

3.3. Pseudo-kod

Kolejną techniką dokumentowania programu (podprogramu, modułu itd.) jest sposób zapisu treści programu (algorytmu) przy użyciu tzw. „pseudo-kodu”. Sposób ten charakteryzuje się tym, że uzyskany opis programu zapewnia dobrą czytelność i zrozumiałość przekazu przy jednoczesnym zachowaniu docelowej struktury programu (algorytmu). Forma pseudo-kodu jest pośrednia pomiędzy formalnym językiem programowania a językiem naturalnym. Jest on sformalizowany tylko częściowo, ponieważ obok pewnych, ustalonych struktur składni stosuje się również opis czynności wyrażony za pomocą języka naturalnego.

Zatem, pseudo-kod w swojej budowie jest zbliżony do języka programowania. Notacja używana do dokumentowania programu (algorytmu) zawiera następujące elementy składniowe [10, 16]:

- **zdanie imperatywne**, które określa czynność do wykonania – np. „*oblicz pierwiastki równania kwadratowego*”; jest ono zapisywane w języku naturalnym w postaci zdania albo równoważnika zdania i określa elementarny krok algorytmu; jeśli opisywana czynność jest złożona, to w toku uszczegółowiania dokumentacji, zdanie ogólne zostanie zastąpione całą sekwencją prostszych zdań;

- **zdanie grupujące** ma postać sekwencji zdań imperatywnych pojedynczych, które zastępują jedno zdanie ogólne; zdanie to stosowane jest w sytuacji, kiedy jedno zdanie imperatywne jest zbyt ogólne; sekwencję zdań ujętych w nawiasy klamrowe należy taktować jako jedno zdanie (listing 3.1);

  Listing 3.1.

```
{
   zdanie imperatywne 1
   ...
   zdanie imperatywne n
}
```

- **zdanie decyzyjne** – zawiera ono strukturę opisującą decyzje podejmowane w algorytmie i ich skutki; do zapisu tej struktury używa się wyróżnionych słów i zwrotów języka naturalnego pisanych drukiem pogrubionym, których znaczenie jest takie, jak w języku potocznym; zdanie to musi zawierać kryterium lub warunek wyboru oraz zdanie opisujące możliwości wyboru; zdanie decyzyjne zapisuje się za pomocą albo struktury prostej – listing 3.2, lub struktury z alternatywą – listing 3.3;

  Listing 3.2.

```
Jeśli warunek to
   zdanie imperatywne
```

  Listing 3.3.

```
Jeśli warunek to
   zdanie imperatywne nr 1
w przeciwnym przypadku
   zdanie imperatywne nr 2
```

- **zdanie iteracyjne** jest stosowane do zapisu sytuacji, w której pewną czynność należy powtórzyć dla kolejnych wartości z danego zakresu albo określoną liczbę razy; strukturę tego zdana przedstawia listing 3.4;

  Listing 3.4.

```
Dla lista przypadków wykonuj
   zdanie imperatywne
```

- **zdanie iteracyjne** warunkowe opisuje sytuację, w której określone czynności należy powtarzać dopóty, dopóki jest spełniony podany warunek; postać zapisu tego zdania umieszczono w listingu 3.5. Zapis w tym listingu należy interpretować w ten sposób, że iteracja kończy się w momencie, kiedy „*warunek*” przestaje być prawdziwy. Zatem ważne jest, aby „*zdanie imperatywne*” miało wpływ na ten warunek.

Listing 3.5.

```
Podczas gdy warunek wykonuj
   zdanie imperatywne
```

Reguły zapisu algorytmu, również za pomocą języka programowania, są uzupełnione określonymi konwencjami notacji, które poprawiają czytelność zapisu algorytmu. Mianowicie, zdania podporządkowane innym są wcięte o kilka znaków, tak aby zależności te były widoczne graficznie. Zdania równorzędne są wcięte o tyle samo znaków, a nawiasy grupujące zapisane są tak, aby tworzyły parę – tj. początek i koniec danego zdania grupującego muszą być widoczne graficznie. Każde nowe zdanie jest zapisywane w nowej linii.

Konwencje powyższe nie stanowią formalnej części składni zapisu programu, ale ich przestrzeganie świadczy o dobrym stylu programowania przez programistę. Celem stosowania danej notacji jest przede wszystkim zapewnienie maksymalnej przejrzystości i czytelności programu dla programisty.

Powróćmy do przykładu algorytmu obliczania miejsc zerowych dla równania kwadratowego, który został przedstawiony na rys. 3.2 za pomocą schematu blokowego. Algorytm ten, zapisany przy użyciu pseudo-kodu, umieszczono w listing 3.6. Proszę zwrócić uwagę na wcięcia, które nie tylko polepszają czytelność zapisu, ale również pozwalają ustalić hierarchę zdań.

Listing 3.6.

```
podaj wartości współczynników a, b, c dla równania
oblicz deltę dla równania kwadratowego
jeśli delta < 0 to
   wyświetl, że nie ma rozwiązania rzeczywistego
w przeciwnym przypadku
   jeśli delta = 0 to
      {
        oblicz jedno miejsce zerowe x
        przyjmij, że x₁ = x₂ = x
      }
   w przeciwnym przypadku
      oblicz dwa miejsca zerowe x1, x2
wyświetl rozwiązanie w postaci x1, x2
```

3.4. Język modelowania wizualnego (UML)

3.4.1. Charakterystyka ogólna

Sieć działań i pseudo-kod są narzędziami wykorzystywanymi głównie w procesie projektowania i dokumentowania algorytmów, które są wykorzystywane w programie tworzonym głównie w oparciu o paradygmat imperatywny. Natomiast UML (z ang. *Unified Modeling Language* – zunifikowany język mo-

delowania) jest wykorzystywany do graficznego przedstawiania relacji i powiązań pomiędzy klasami i obiektami na etapie projektowania programu komputerowego zorientowanego obiektowo [18, 19, 23, 29].

Początki modelowania wizualnego datuje się na przełom lat osiemdziesiątych i dziewięćdziesiątych ubiegłego wieku. W tym okresie, G. Booch, J. Rumbaugha oraz I. Jacobson [22, 29] niezależnie rozpoczęli pracę nad opracowaniem własnej metodyki obiektowej analizy i projektowania programów komputerowych. Z czasem, zaczęli współpracować ze sobą, a efektem tego było wspólne stworzenie zunifikowanego języka modelowania wizualnego – w skrócie UML. Obecnie, ta technika dokumentowania programu jest standardem w przemyśle informatycznym i wciąż jest rozwijana.

UML składa się z elementów graficznych, które są grupowane w postaci **diagramów**. Ponieważ pomiędzy tymi diagramami są zdefiniowane zasady łączenia i ustalania relacji (**związków**), technika UML ma statut języka modelowania wizualnego, stosowanego do szeroko pojętego planowania programów i systemów komputerowych. Ciekawostką jest to, że ideę UML można również adaptować do procesu projektowania technologii maszyn.

Celem UML jest czytelne przedstawienie modelu projektowanego programu komputerowego. Utworzone diagramy i zdefiniowane związki pomiędzy nimi mają za zadanie jedynie opisać zakres działań wykonywanych przez program, ale pomijając jednocześnie sposób ich zaimplementowania. W rozdziale tym zostaną omówione ważniejsze cechy UML, w zakresie wystarczającym do opanowania poprawnego projektowania programów komputerowych stosowanych do obliczeń inżynierskich. Należy pamiętać, że zrozumienie istoty UML wymaga uprzedniego poznania zasad programowania obiektowego, które omówiono w rozdziale 4.5.

W notacji języka modelowania wizualnego można wyróżnić wiele komponentów tworzących diagramy, które charakteryzują się różnym stopniem złożoności. Najprostsze diagramy, które reprezentują np. obiekty, mają formę prostych figur geometrycznych i najczęściej opisują statyczny stan programu – często zamiennie nazywane są komponentami UML. Istnieją również diagramy, które zawierają dodatkowo szereg informacji o związkach pomiędzy jego elementami, zawierają opisy oraz grupują te elementy w złożone pakiety. Diagramy te również opisują dynamikę programu w czasie. Typowymi elementami języka modelowania UML są [22, 29]:

- diagram klasy,
- diagram obiektu,
- diagram przypadków użycia,
- diagram stanów,
- diagram przebiegu,

- diagram czynności,
- diagram kooperacji.

3.4.2. Podstawowe komponenty UML

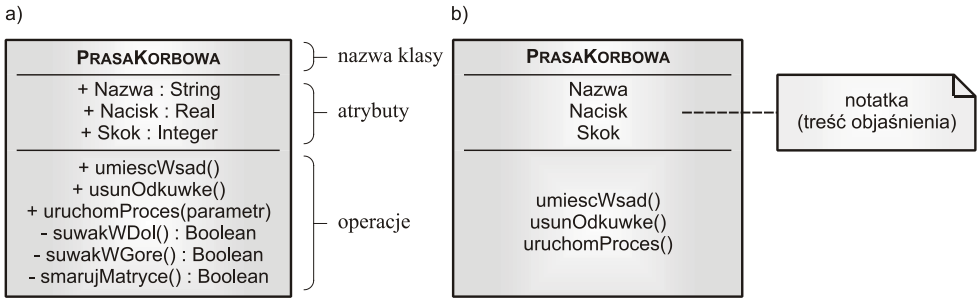
Poniżej zostaną przedstawione jedynie wybrane komponenty UML [15, 23, 25, 29], które opisują stan statyczny programu i są wykorzystywane do budowy złożonych diagramów. W celu lepszego zrozumienia istoty modelowania UML, komponenty te reprezentują w sposób abstrakcyjny wirtualny model zakładu produkcyjnego z przemysłu maszynowego.

Diagram klasy

Podstawowym komponentem, stosowanym w trakcie modelowania techniką UML, jest diagram klasy. Jego przykład przedstawiono na rys. 3.4. Diagram ten ma postać prostokąta podzielonego poziomymi liniami na trzy sekcje zawierające:

- nazwę klasy,
- listę atrybutów, które reprezentują zmienne klasy – dlatego też oprócz ich nazwy, po dwukropku podaje się typ atrybutu;
- listę operacji, które są odpowiednikiem metod klasy – do opisu tej sekcji stosuje się tę samą logikę jak w przypadku atrybutów, z tym, że po znaku „dwukropek” podawany jest typ zwrotny operacji.

W sekcji atrybutów i operacji (rys. 3.4a) przed nazwami mogą być umieszczone znaki specjalne określające dostępność do zasobów klasy. Typowe znaki takie jak „*plus*” i „*minus*” określają odpowiednio typ dostępu „*public*” – atrybut



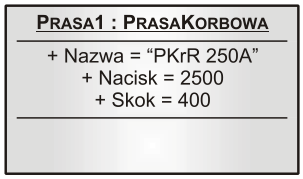
Rys. 3.4. Przykład diagram klasy: a) postać pełna (z opisem wyjaśniającym znaczenie poszczególnych sekcji komponentu), b) postać uproszczona, do której dołączono notatkę (komponent UML)

lub operacja jest widoczna na zewnątrz klasy oraz „*private*" – element jest widoczny tylko wewnątrz klasy.

Na rys. 3.4b przedstawiono przykład diagramu klasy w formie uproszonej wraz z dołączoną notatką. Stopień uproszczenia jest w zasadzie dowolny. Natomiast notatka pozwala uzupełnić diagram o dodatkowe informacje na temat klasy lub wybranych jej atrybutów i operacji.

Diagram obiektu

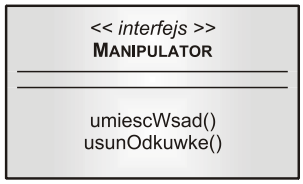
Kolejnym istotnym komponentem UML jest diagram obiektu. Przykład takiego komponentu umieszczono na rys. 3.5. Reprezentuje on instancję klasy, która została przedstawiona na wcześniejszym rys. 3.4a. W odróżnieniu od diagramu klasy, diagram obiektu najczęściej zawiera tylko nazwę komponentu, która składa się z oddzielonych dwukropkiem nazwy obiektu i nazwy klasy. Nazwa komponentu zawsze jest podkreślona. Diagram ten może zawierać również atrybuty z podanymi ich wartościami oraz notatkę.



Rys. 3.5. Przykład diagramu obiektu o nazwie „*Prasa1*", który jest instancją klasy „*PrasaKorbowa*"

Diagram interfejsu

Przykład diagramu interfejsu przedstawiono na rys. 3.6. Nazwa komponentu składa się z dwóch części, mianowicie nad nazwą interfejsu znajduje się napis informujący, że diagram ten przedstawia interfejs. Zgodnie z zasadami programowania obiektowego, interfejs jedynie definiuje zestaw własności, metod i zdarzeń, które powinny być zaimplementowane i udostępnione w dowolnej klasie. W pewnym sensie, interfejs przypomina klasę abstrakcyjną, w której są zdefiniowane jedynie operacje pomijając ich implementację.



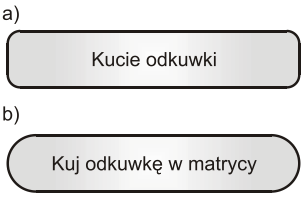
Rys. 3.6. Przykład diagramu interfejsu o nazwie „*Manipulator*" (opis w tekście)

Omawiany diagram jest przykładem stereotypu stosowanego w notacji UML. Ideą stereotypu jest przekształcenie już istniejącego diagramu w komponent pełniący inną funkcję. Celem wprowadzenia stereotypów do UML jest uelastycznienie zasad notacji języka modelowania i jednocześnie zapobieżenia sytu-

acji zmuszającej do definiowania nowych znaków graficznych. W przypadku diagramu interfejsu, powstał on na bazie diagramu klasy.

Komponent stanu i czynności

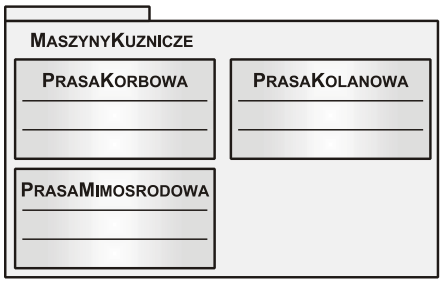
Komponenty stanu i czynności są elementami zachowania się programu lub wybranego obiektu. Przykład **komponentu stanu** obiektu przedstawiono na rys. 3.7a. Reprezentowany jest on przez symbol graficzny w formie prostokąta, w którym naroża są zaokrąglone. Opisem jest równoważnik zdania, np. „*kucie odkuwki*”. Natomiast na rys. 3.7b umieszczono **komponent czynności** obiektu. Jest on reprezentowany przez symbol bloku owalnego. Opisem jest zdanie w trybie rozkazującym, np. „*kuj odkuwkę*”.



Rys. 3.7. Przykład komponentu stanu (a) oraz komponentu czynności (b) – opis w tekście

Pakiet

Pakiet jest to komponent składowy UML służący do grupowania wybranych elementów diagramów. Ma on kształt figury przedstawiającej „folder z zakładką”. Komponent ten najczęściej służy do utworzenia podsystemu składającego się np. z kilku klas. Przykład zgrupowania trzech klas w pakiecie o nazwie „*MaszynyKuznicze*” przedstawiono na rys. 3.8.



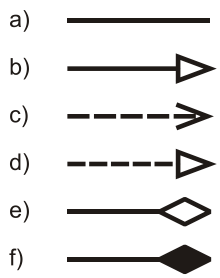
Rys. 3.8. Przykład pakietu o nazwie „*MaszynyKuznicze*” (opis w tekście)

3.4.3. Związki pomiędzy komponentami UML

Budowa modelu wizualnego programu komputerowego za pomocą techniki UML polega na zestawieniu komponentów i diagramów oraz zdefiniowaniu związków pomiędzy nimi. Najczęściej definiuje się związki pomiędzy klasami i obiektami, a przedstawia się je na diagramie UML za pomocą linii łączących odpowiednie komponenty. Na rys. 3.9 umieszczono podstawowe symbole graficzne reprezentujące takie związki jak [22, 29]:

- powiązanie (rys. 3.9a),
- uogólnienie (rys. 3.9b),
- zależność (rys. 3.9c),
- realizacja (rys. 3.9d),
- agregacja z ograniczeniem (rys. 3.9e),
- agregacja częściowa (rys. 3.9f).

W dalszej części podrozdziału zostaną omówione szczegółowo poszczególne typy związków [23, 25, 29].

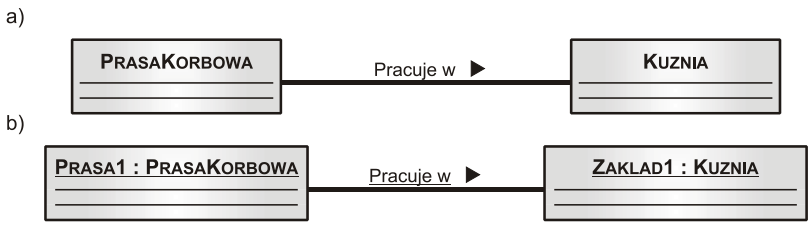


Rys. 3.9. Symbole graficzne związków – opis w tekście

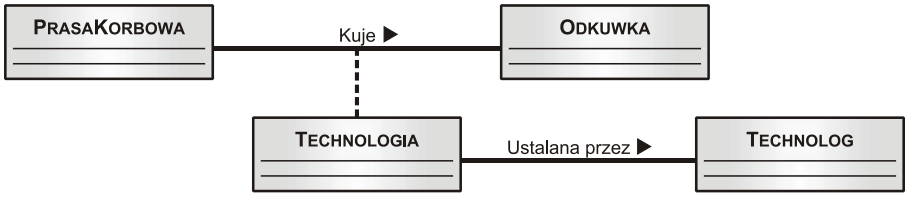
Powiązanie

Powiązaniem nazywamy związek znaczeniowy istniejący pomiędzy klasami lub ich obiektami. W tym drugim przypadku mówi się o **wiązaniu**, czyli instancji powiązania odnoszącego się do instancji klasy (tj. obiektu). Przykład powiązania i wiązania w formie podstawowej przedstawiono na rys. 3.10. Powiązanie przedstawiane jest za pomocą linii ciągłej (rys. 3.9a) łączącej ze sobą dwa komponenty lub łączące kilka komponentów z jednym komponentem. Nazwę rodzaju powiązania oraz zwrot związku (wypełniony trójkąt) umieszcza się nad linią.

Powiązanie może mieć zdefiniowane atrybuty i operacje – w takim przypadku powiązanie jest nazywane **klasą powiązania**, którą na diagramie przedstawia się w taki sam sposób jak zwykłą klasę. Na rys. 3.11 umieszczono przykład ideowy diagramu, gdzie klasa powiązania o nazwie „*Technologia*" jest połączona linią przerywaną z linią powiązania „*Kucie*". Dodatkowo, klasa powiązania może być powiązana z inną klasą – na omawianym przykładzie, jest to klasa o nazwie „*Technolog*".

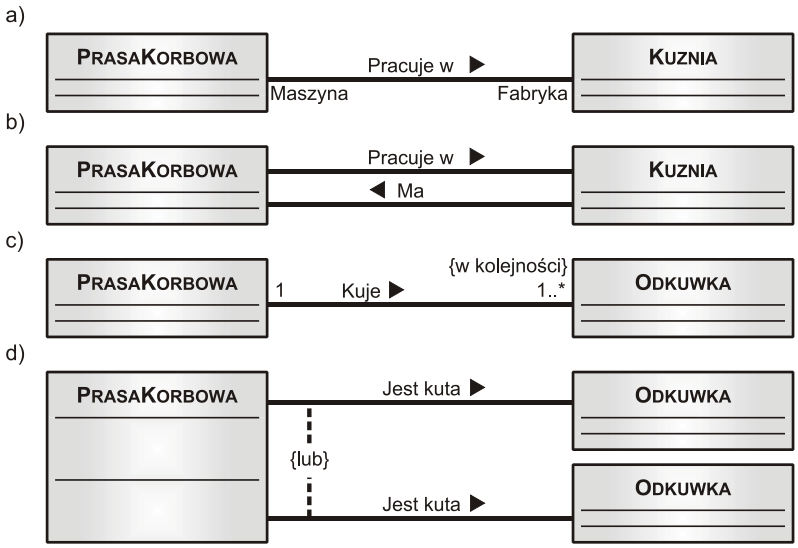


Rys. 3.10. Przykład podstawowej formy przedstawiania powiązania pomiędzy klasami (a) oraz wiązania (instancji powiązania) pomiędzy dwoma obiektami (b)



Rys. 3.11. Przykład klasy powiązania o nazwie „*Technologia*” – opis w tekście

W trakcie dokumentowania struktury programu komputerowego dość często na diagramie umieszcza się szereg dodatkowych informacji o sposobie realizacji powiązania (lub wiązania). Najczęściej spotykane rozbudowane formy prezentacji sposobu powiązania umieszczono na rys. 3.12. Informacjami, o których jest mowa, są:



Rys. 3.12. Przykłady umieszczania dodatkowych informacji o powiązaniu pomiędzy klasami: a) wskazanie roli klas w powiązaniu; b) podanie dwóch powiązań na jednym diagramie; c) podanie liczebności; c), d) sformułowanie ograniczeń powiązania

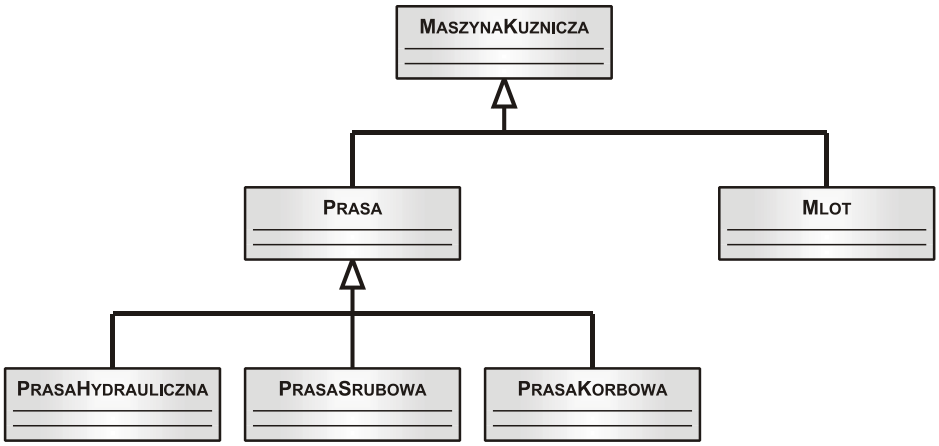
- rola klasy (lub obiektu) jaką pełni w powiązaniu (wiązaniu) – objaśnia się to poprzez podanie opisów umieszczanych w pobliżu linii powiązania, np. w sposób przedstawiony na rys. 3.12a;
- zestawienie dwóch, przeciwbieżnych powiązań pomiędzy dwoma klasami (rys. 3.12b);
- liczebność określającą ilość obiektów jednej klasy, które są w związku z jednym obiektem klasy powiązanej; liczebność określa się poprzez podanie liczby nad linią powiązania obok klasy; wyróżnia się następujące typy związków: „*jedne do jednego*" (brak liczb na diagramie określa domyślnie ten typ powiązania), „*jeden do wielu*", „*jeden do jednego lub więcej*" (przykład na rys. 3.12c, gdzie znak gwiazdki oznacza „więcej" lub „wiele"), „*jeden do »określonego przedziału«*" oraz „*jeden do »zestawu możliwości do wyboru«*";
- ograniczenia związku – na rys. 3.12c podano warunek realizacji powiązania poprzez umieszczenie opisu ograniczenia w nawiasie klamrowym w pobliżu linii powiązania i klasy, której to ograniczenie dotyczy; natomiast na rys. 3.12d przedstawiono ograniczenie określające możliwość wyboru powiązania – odpowiedni opis ograniczenia w nawiasach klamrowych umieszcza się na linii przerywanej łączącej linie dwóch alternatywnych związków.

Uogólnienie

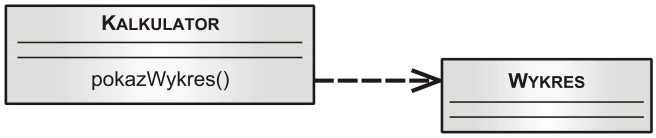
Związek uogólnienia znajdujący się w diagramie UML przedstawia mechanizm dziedziczenia, który jest cechą programowania zorientowanego obiektowo. Związek ten jest szczególnym przypadkiem relacji pomiędzy dwoma klasami. Linia uogólnienia łączy klasę nazywaną **podklasą** (klasą potomną) z klasą określaną jako **nadklasą** (przodkiem klasy potomnej). Na linii tej umieszcza się niewypełniony trójkąt wskazujący na nadklasę. Ten rodzaj połączenia klas określany jest sformułowaniem „jest rodzajem". Zgodnie z przykładem hierarchii uogólnienia przedstawionym na rys. 3.13, stwierdza się, że podklasa „*Prasa*" jest rodzajem nadklasy „*MaszynaKuznicza*", a podklasa „*PrasaKorbowa*" jest rodzajem nadklasy „*Prasa*".

Zależność

Zależność jest to rodzaj związku występującego w przypadku, gdy jedna klasa używa innej klasy. Najczęściej objawia się to sytuacją, gdy w sygnaturze klasy zależnej występuje nazwa innej klasy. W notacji UML zależność ta jest przedstawiana za pomocą linii przerywanej (rys. 3.9c), której grot strzałki wskazuje na klasę niezależną. Na rys. 3.14 przedstawiono przykład zależności, w którym klasa zależna o nazwie „*Kalkulator*" posiada operację „*pokazWykres()*" powodującą skorzystanie z klasy o nazwie „*Wykres*".



Rys. 3.13. Przykład uogólnienia – opis w tekście

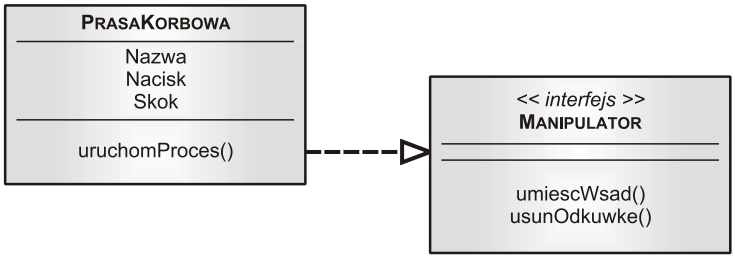


Rys. 3.14. Przykład zależności – opis w tekście

Realizacja

Realizacja jest to związek występujący między klasą a interfejsem. W notacji UML przedstawiany jest on za pomocą linii przerywanej (rys. 3.9d) zakończonej trójkątem wskazującym na interfejs. Na rys. 3.15 przedstawiono przykład omawianego związku, w którym interfejs „*Manipulator*” realizuje część zachowań klasy o nazwie „*PrasaKorbowa*”. Innym przykładem klasycznego interfejsu jest np. abstrakcja klawiatury komputera. Należy podkreślić, że wydzielenie czynności klasy w formie interfejsu ma na celu:

- określenie wspólnych zachowań dla grupy podobnych klas, np. przedstawionych na rys. 3.13, co znacząco ułatwia projektowanie programu;



Rys. 3.15. Przykład realizacji – opis w tekście

- umożliwienie późniejszej konserwacji programu, ponieważ zgodnie z zasadą projektowania implementacja zachowań klasy może być zmienna, ale jej interfejs musi być stały.

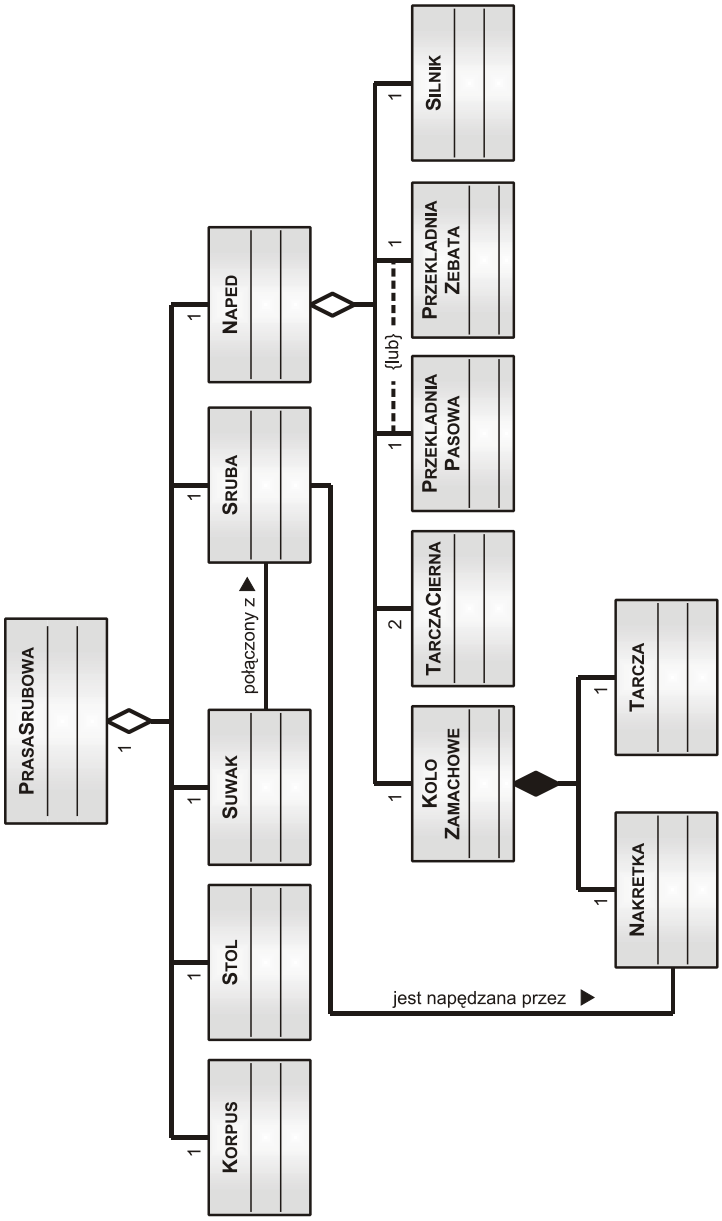
Agregacja

Związek agregacji przedstawiany w diagramie UML reprezentuje pewną formę logicznej kompozycji klas i obiektów. Programując obiektowo często ma się do czynienia z klasą składającą się z wielu innych klas, gdzie jednocześnie określona jest ich hierarchia i wzajemne powiązanie. Na rys. 3.16 przedstawiono przykład takiego związku bazując na abstrakcji prasy śrubowej. Klasa „*PrasaSrubowa*" jest tzw. klasą całkowitą, którą jest kompozycją klas reprezentujących poszczególne podzespoły maszyny.

Szczególną odmianą omawianego związku jest agregacja całkowita, która nazywana jest również kompozytem. Jest to tzw. „silny" typ kompozycji, w której każda klasa składowa może należeć tylko do jednej całości.

3.4.4. Złożone diagramy UML

Omówione wcześniej podstawowe diagramy (komponenty) i związki używane w notacji UML, umożliwiają modelowanie programów komputerowych pod kątem przedstawienia jego budowy. Jednak pełna dokumentacja aplikacji powinna również zawierać informacje o jej działaniu. Do modelowania zachowania się programu komputerowego w trakcie jego działania służą trzy typy diagramów [23, 29], które są opisane w dalszej części rozdziału.



Rys. 3.16. Przykład agregacji z podaniem liczebności oraz ograniczenia typu {lub} – opis w tekście

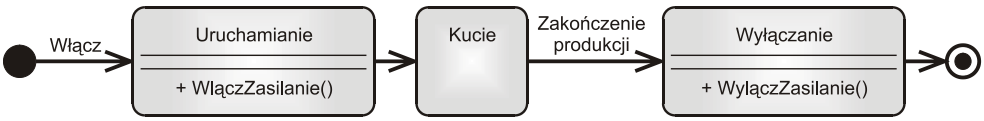
Diagram stanów

Jednym ze sposobów opisania zmian zachodzących w trakcie działania programu jest stwierdzenie, że jego obiekty zmieniają swój stan w odpowiedzi na zaistniałe zdarzenia. Tego rodzaju zmiany są przedstawiane za pomocą diagramu stanów. Składa się on z komponentów, reprezentujących aktualny stan danego obiektu, oraz połączeń (linii ciągłych zakończonych grotem otwartym) reprezentujących przejścia pomiędzy kolejnymi stanami. W komponencie stanu przedstawionym na rys. 3.7a można również umieszczać informacje o zmiennych i czynnościach podobnie jak w diagramie klasy. Jednak są to informacje ściśle powiązane z charakterystyką stanu. Na przykład zmiennymi mogą być liczniki, natomiast czynnościami – zdarzenia i akcje.

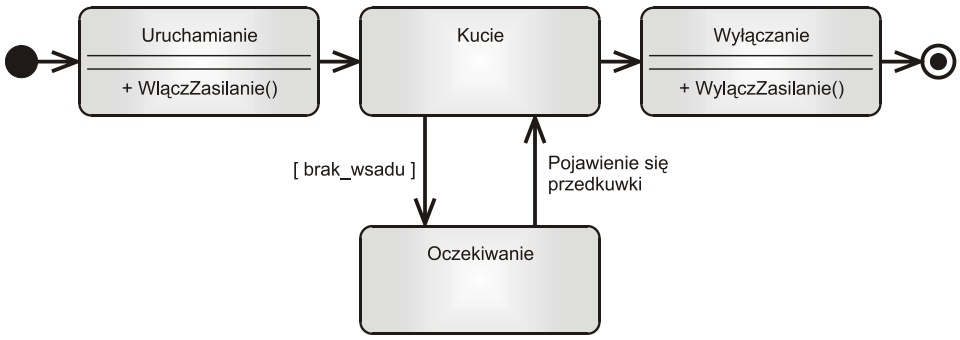
Na rys. 3.17 przedstawiono przykładowy diagram stanów bazując na abstrakcji maszyny kuźniczej, np. młocie do kucia matrycowego odkuwek. Stan początkowy i końcowy obiektu – tzw. „*wejście*" i „*wyjście*" oznacza się specjalnymi znakami graficznymi w postaci zaczernionych kół. Przejścia do kolejnych stanów mogą być realizowane w wyniku zaistnienia akcji, zdarzenia lub samoczynnie. Przykładem akcji jest „*włączenie*" maszyny, natomiast przykładem zdarzenia – „*zakończenie produkcji*" serii odkuwek.

Diagram stanów może również zawierać informacje określające warunki przejścia w inny stan. W terminologii UML jest to nazywane **warunkiem dozoru**. Na rys. 3.18 przedstawiono zmodyfikowaną wersję diagramu z rys. 3.17, w którym pokazano sposób modelowana wspomnianego warunku dozoru. Załóżmy, że po upływie określonego czasu, w którym kowal nie pobrał przedkuwki (wsadu) z pojemnika, młot przechodzi w stan oczekiwania. W tym przypadku warunkiem dozoru jest zmienna „*brak_wsadu*" ujęta w nawiasy kwadratowe. W ogólnym przypadku, parametr ten często wyraża się w wymiarze czasu.

Przedstawione na rys. 3.17 i rys. 3.18 diagramy, a w szczególności stany obiektu o nazwie „*Kucie*" i „*Oczekiwanie*", zawierają informacje zbyt uogólnio-



Rys. 3.17. Diagram stanów dla obiektu „*MłotMatrycowy*" – przedstawiono stany, przejścia z akcjami („*Włącz*") i zdarzeniami („*Zakończenie produkcji*") uruchamiającymi oraz przejścia bez zdarzeń uruchamiających stany



Rys. 3.18. Diagram stanów dla obiektu „*MłotMatrycowy*”, gdzie przedstawiono stan „*Oczekiwanie*” wraz z warunkiem dozoru – opis w tekście

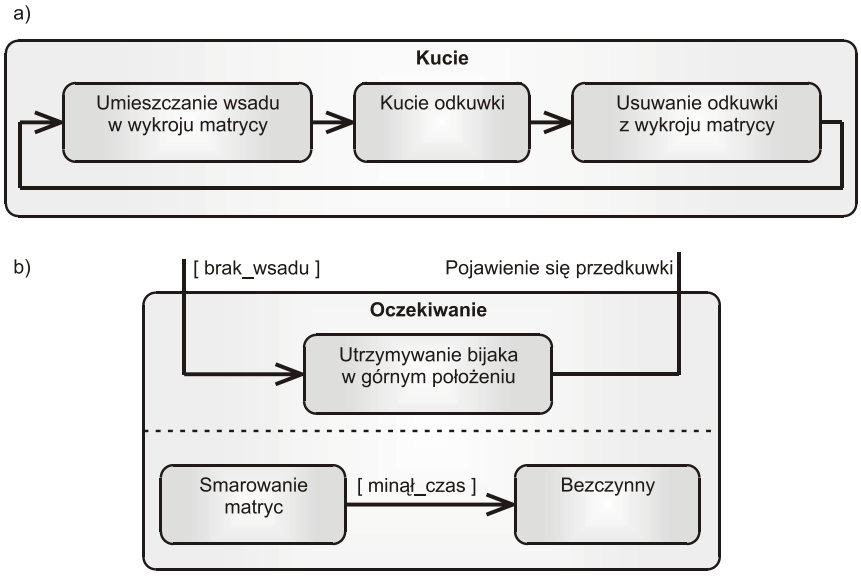
ne. W zasadzie można je uszczegółowić – do tego celu służą **diagramy podstanów**. Wyróżnia się dwa rodzaje podstanów, mianowicie podstany:

- **sekwencyjne** – przykład umieszczono na rys. 3.19a, który jest dopełnieniem informacji na temat stanu „*Kucie*”;
- **współbieżne** (rys. 3.19b – dopełnienie informacji o stanie „*Oczekiwanie*”), które są diagramami stanów złożonych z co najmniej dwóch części (podstanów), pomiędzy którymi występuje taki sam związek jak w przypadku agregacji całkowitej. Podstany te występują w tym samym czasie, a w reprezentacji graficznej są one oddzielone poziomą linią przerywaną.

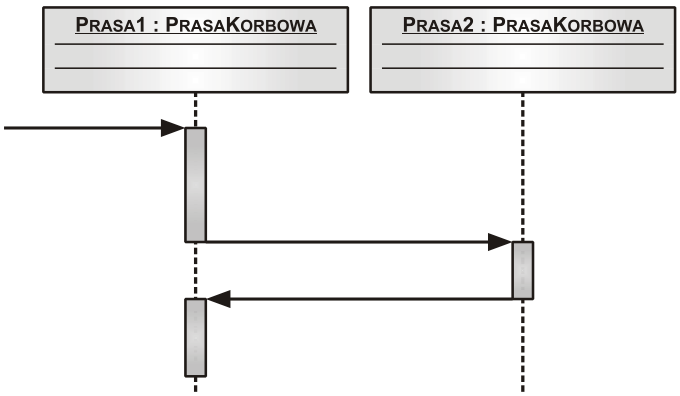
Diagram przebiegów

Diagram przebiegów jest dwuwymiarowym obrazem interakcji zachodzących pomiędzy obiektami w trakcie działania programu. Dodatkowym parametrem determinującym dwuwymiarowość diagramu jest czas. Typowy diagram, który przedstawiono na rys. 3.20, składa się z następujących elementów:

- **obiektów**, które są reprezentowane przez uproszczone diagramy obiektów umieszczanych w górnej części diagramu przebiegu od lewej do prawej strony;
- **osi czasu** – od każdego obiektu w dół przebiega linia przerywana reprezentująca jego „linię życia” (tłumaczenie dosłowne z ang. [23]);
- **aktywacji** – na osi czasu danego obiektu mogą znajdować się wąskie prostokąty, które reprezentują wykonywane operacje przez obiekt; długość prostokąta aktywacji określa w sposób podglądowy czas jej trwania;



Rys. 3.19. Diagram podstanów sekwencyjnych (a) oraz współbieżnych (b) – opis w tekście



Rys. 3.20. Podstawowa postać diagramu przebiegów – opis w tekście

- **komunikatów** – linie poziome zakończone grotem pomiędzy osiami czasu, które przedstawiają kierunek przekazania sterowania z obiektu do innego obiektu.

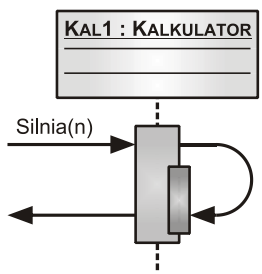
Na rys. 3.21 przedstawiono symbole graficzne komunikatów jakie są używane na diagramie przebiegów. Wyróżnia się trzy rodzaje komunikatów:

- **prosty** – następuje przekazanie sterowania z jednego obiektu do innego;
- **synchroniczny** – obiekt wysyłający ten rodzaj komunikatu do innego obiektu oczekuje na odpowiedź, po czym otrzymawszy ją przechodzi do dalszych własnych działań;
- **asynchroniczny** – obiekt po wysłaniu komunikatu do innego obiektu kontynuuje własne działania nie oczekując na odpowiedź.

Rys. 3.21. Symbole komunikatów na diagramie przebiegów: a) prosty, b) synchroniczny, c) asynchroniczny

Komunikaty mogą zawierać etykiety umieszczane nad linią. Najczęściej zawierają one nazwę metody (operacji) towarzyszącej wysłaniu danego komunikatu. Etykieta również może przedstawiać warunek przypadku wystąpienia danego komunikatu – w takim przypadku jest on ujęty w nawiasach kwadratowych.

Programy komputerowe z reguły charakteryzują się wielowątkowością. Występują w nim instrukcje warunkowe (np. instrukcja typu *„If … Then”* [25]) oraz bloki iteracyjne (tzw. pętle typu *„While … Do”* [25]). Te dwie zasadnicze konstrukcje programu są możliwe do przedstawienia za pomocą diagramu przebiegów za pomocą rozgałęzienia i pętli. Ta druga konstrukcja może być stosowana również do przedstawienia rekurencji. Nawiązując do algorytmu obliczania silni z liczby *n* (rozdział 5.2), na rys. 3.22 przedstawiono przykładowy diagram przebiegów, który modeluje rekurencyjny charakter metody „*Silnia(n)*” zdefiniowanej w klasie o nazwie „*Kalkulator*”.



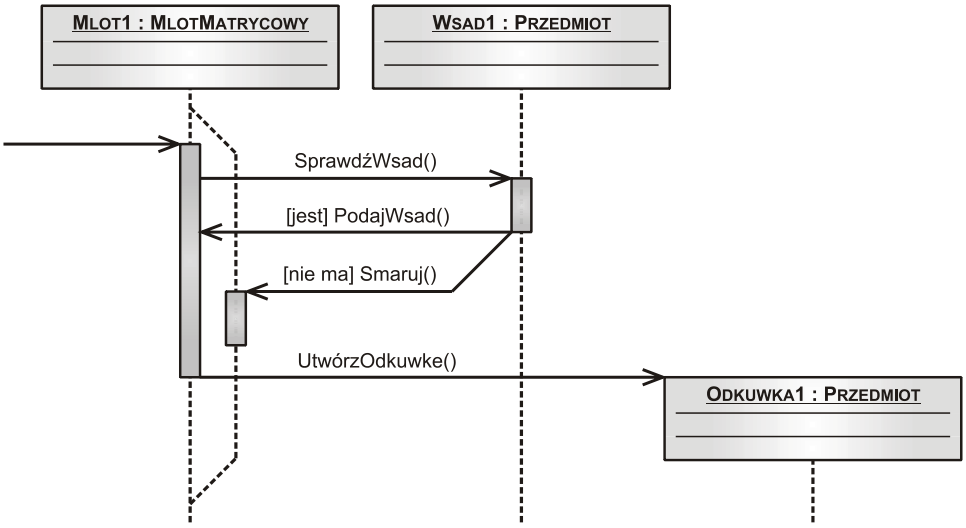
Rys. 3.22. Diagram przebiegów dla rekurencji (opis w tekście)

Jeżeli za pomocą diagramu przebiegów przedstawia się jedynie jeden „scenariusz” przypadku użycia programu, tak jak to pokazano na rys. 3.20, to taki diagram jest nazywany **egzemplarzowym**. Natomiast, jeżeli diagram przebiegów opisuje wiele

„scenariuszy" przypadków użycia programu, wykorzystując przy tym rozgałęzienia i pętle, to diagram taki jest nazywany **ogólnym**.

Przykład ogólnego diagramu przebiegów umieszczono na rys. 3.23. Reprezentuje on interakcje pomiędzy obiektami, które uczestniczą w abstrakcyjnym procesie kucia odkuwki na młocie matrycowym. Umieszczony na rys. 3.18 i rys. 3.19 diagram stanów dla obiektu typu „*MlotMatrycowy*" informuje, że np. obiekt „*Mlot1*" (rys. 3.23) może posiadać dwa stany – „*kucie odkuwki*" oraz „*smarowanie matryc*". Zatem w zależności od spełnienia warunku dostępności wsadu (reprezentowanego przez obiekt „*Wsad1*") w danym czasie, obiekt „*Mlot1*" wykona jedną z dwóch możliwych operacji – odpowiednio „*PodajWsad()*" gdy warunek jest spełniony oraz „*Smaruj()*" gdy warunek nie jest spełniony.

Po zakończeniu przez obiekt akcji związanych z jego stanem „*kucie*" następuje utworzenie nowego obiektu o nazwie „*Odkuwka1*". Dla podkreślenia, że obiekt powstaje w trakcie trwania programu, diagram danego obiektu nie umieszcza się obok innych obiektów, ale niżej, aby jego położenie odpowiadało momentowi utworzenia.



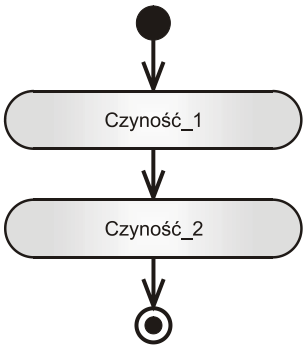
Rys. 3.23. Ogólny diagram przebiegów – przykład abstrakcji procesu kucia odkuwki, gdzie na diagramie umieszczono rozgałęzienie wraz z warunkiem dostępności wsadu w danym czasie oraz przypadek utworzenia nowego obiektu w trakcie przebiegu

Diagram czynności

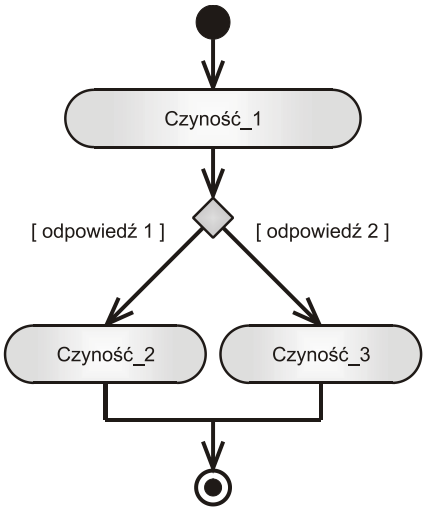
Diagram czynności jest uważany za rozszerzenie diagramu stanów, gdzie następuje wyeksponowanie stanów i połączeń. Zatem, diagram ten służy do przedstawienia przepływu sterowania od czynności do czynności. W sposób efektywny obrazuje to, co dzieje się w programie w trakcie wykonywania operacji przez obiekty. Jest on integralną częścią dokumentacji programu komputerowego, a swoją budową i zasadami funkcjonowania przypomina „sieć działań” używaną w programowaniu imperatywnym.

Diagram czynności jest budowany w oparciu o następujące konstrukcje:

- **przejście od czynności do czynności** (rys. 3.24) – komponenty czynności (rys. 3.7b) są połączone liniami zakończonymi strzałkami, które reprezentują przejścia (przepływy sterowania) pomiędzy czynnościami, przy czym diagram czynności posiada dodatkowo punkt startowy i końcowy (tak samo jak diagram stanów);
- **decyzja** (rys. 3.25) – rozgałęzienie przepływu sterowania pomiędzy czynnościami jest pokazywane za pomocą dwóch linii wychodzących z symbolu graficznego w postaci małego rombu, przy których umieszcza się warunki dozoru ujęte w nawiasy kwadratowe;

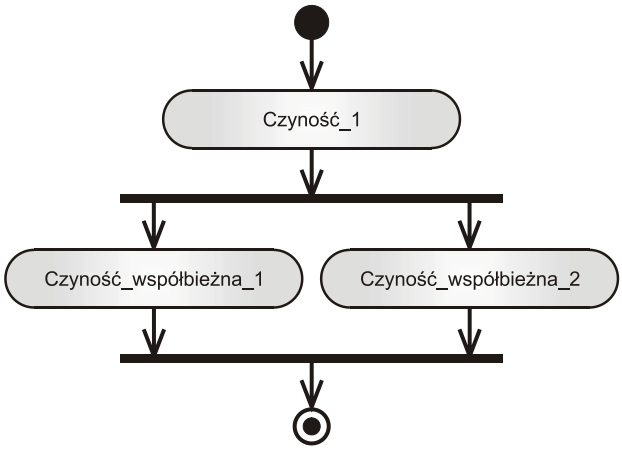


Rys. 3.24. Diagram czynności – punkt startowy i końcowy oraz przejścia (przepływy sterowania) od czynności do czynności

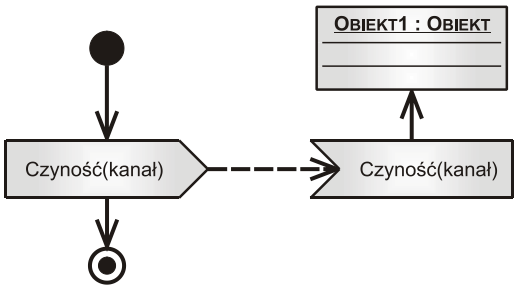


Rys. 3.25. Diagram czynności – decyzja (rozgałęzienie przepływu sterowania) wraz z warunkami dozoru

- **rozwidlenie i przejście współbieżne** (rys. 3.26) – służy do przedstawienia sytuacji często spotykanej w programowaniu obiektowym, która polega na tym, że w tym samym czasie są wykonywane współbieżnie dwie czynności przez jeden obiekt;
- **wysyłanie sygnału** (rys. 3.27) – reprezentuje sytuację, w której jeden obiekt wysyła sygnał do drugiego obiektu, a przyjęcie sygnału skutkuje wykonaniem określonej czynności.

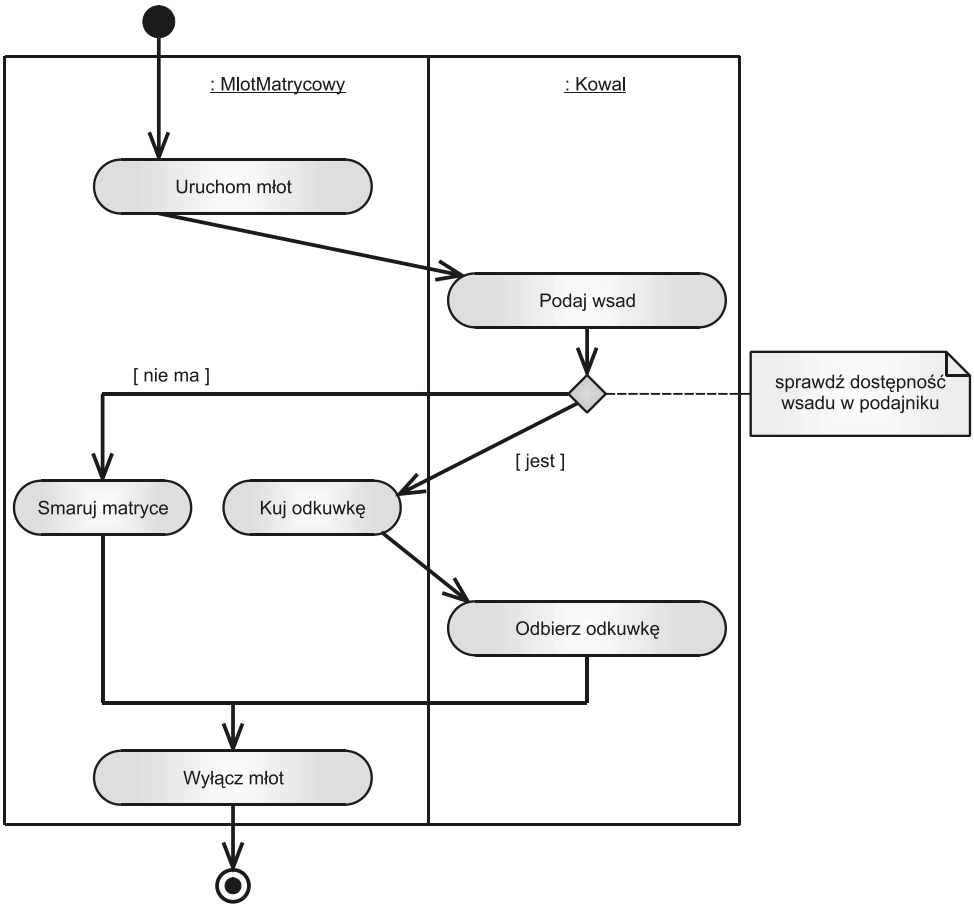


Rys. 3.26. Diagram czynności – rozwidlenie i współbieżne przejście pomiędzy czynnościami



Rys. 3.27. Diagram czynności – wysłanie sygnału z jednego do drugiego obiektu z podaniem kanału

W języku UML komponenty w postaci pięciokąta wypukłego i wklęsłego (rys. 3.27) symbolizują **zdarzenie** odpowiednio wyjściowe i wejściowe. Umieszczenie na tym rysunku dodatkowo komponentu obiektu jest przykładem hybrydowej postaci diagramu, ponieważ komponent ten w zasadzie nie należy do typowych elementów diagramu czynności. W odróżnieniu od „sieci działań” (rozdział 3.2), diagram czynności można budować w spobób swobodny, bez przestrzegania ścisłych zasad jakie np. obowiązują przy tworzeniu wspomianych sieci działań. Dopuszcza się nawet sytację, w której linie przejścia przecinają się



Rys. 3.28. Przykład przedstawiający torową wersję diagramu czynności, który rozszerza diagram stanu umieszczono na rys. 3.18

wzajemnie. Dopuszczalne jest to dlatego, że podczas modelowania w języku UML najważniejsza jest informacyjność przekazu [23, 25].

Innym przykładem diagramu hybrydowego jest diagram czynności umieszczony wewnątrz komponentu obiektu. Jednak częściej w dokumentacji programu stosuje się diagramy czynności w wersi torowej. Pod pojęciem „**tor**" należy rozumieć równoległe segmenty przypisane do podmiotu odpowiedzialnego za realizację danej czynności.

Na rys. 3.28 przedstawiono przykład torowej wersji diagramu, który rozszerza diagram stanu umieszczony na rys. 3.18. W tym przypadku występują dwa tory, należące do obiektów klasy „*MlotMatrycowy*" oraz klasy „*Kowal*". Symbole rozpoczęcia i zakończenia diagramu oraz notatki umieszcza się poza siatką torów. W omawianym przykładzie, notatka precyzuje warunek decyzji. Dzięki temu umieszczone przy liniach przepływu warunki dozoru są bardziej czytelne dla programisty.

4. PARADYGMATY PROGRAMOWANIA

4.1. Wprowadzenie

Programowanie programów komputerowych na przełomie lat pięćdziesiątych i sześćdziesiątych ubiegłego wieku było pozbawione w zasadzie jakichkolwiek reguł. Ówcześni programiści, którzy oprócz języka maszynowego i asemblera wykorzystywali również języki wysokiego poziomu, nie stosowali żadnej formalnej metodyki programowania mającej na celu optymalizacje efektywności ich pracy. Powód takiego stanu rzeczy był banalny – nie istniały usystematyzowane zasady programowania. Oczywiście ten stan rzeczy pył zdeterminowany tym, że ówczesne programy nie były skomplikowane pod względem zarówno budowy jak i przeznaczenia [2, 19, 24, 28÷30].

Dopiero rosnące wymagania rynku wymusiły konieczność podjęcia odpowiednich kroków w kierunku opracowania spójnych reguł projektowania, programowania oraz konserwacji programów komputerowych. Myślą przewodnią rodzącej się filozofii programowania było pojęcie „funkcji obliczanej", której wartość można wyznaczyć w sposób efektywny dla dowolnych wskazanych argumentów.

Efektem podjętych i prowadzonych po dzień dzisiejszy prac rozwojowych są paradygmaty i techniki programowania oraz wzorce projektowe, które dotyczą głównie tworzenia programów zorientowanych obiektowo.

Wyjaśnijmy znaczenie pojęcia „paradygmat programowania". Z języka greckiego, słowo „paradygmat" oznacza w ogólnym znaczeniu wzorzec lub przykład. W przypadku programowania komputerów, pojęcie to określa zbiór mechanizmów, jakich programista używa pisząc program oraz sposób wykonywania tego programu przez komputer. W zasadzie wyróżnia się cztery podstawowe paradygmaty, które nazywa się programowaniem [16, 19, 24]:

- **imperatywnym** – jest to najbardziej pierwotny sposób pisania programu postrzeganego jako ciąg poleceń dla procesora;
- **obiektowym** – gdzie program jest zbiorem komunikujących się ze sobą obiektów zawierających zarówno dane, jak i operacje, które są wykonywane na tych danych;
- **funkcyjnym** – program jest złożeniem funkcji niższego i wyższego rzędu (w sensie matematycznym), w którym nie ma zmiennych (w rozumieniu in-

nych paradygmatów), a wynik jest generowany dopiero w momencie otrzymania danych wejściowych, przy czym wynikiem funkcji mogą być nie tylko dane (informacja) a również może być inna funkcja;
- **logicznym** – program składa się ze zbioru zależności (przesłanek) i stwierdzeń (celów), gdzie obliczenia wykonywane są poprzez dowodzenie celu – wykonywanie programu to nic innego jak próba udowodnienia celu w oparciu o podane przesłanki; podobnie jak w programowaniu funkcyjnym, składnia opisuje to co jest wiadome i to co należy uzyskać.

Podsumowując, paradygmat programowania jest swoistą filozofią pisania programów, która wymaga od programisty „spojrzenia” na zadanie programistyczne na określonym, odmiennym poziomie rozumowania. Oznacza to, że nie można napisać efektywnego i poprawnego programu np. zorientowanego obiektowo myśląc tylko na poziomie np. imperatywnym.

Oczywiście, w technikach programowania istnieje wiele różnych paradygmatów, jednak w informatyce wiele z nich stosuje się sporadycznie, często tylko w szczególnych sytuacjach. Istnieją również przypadki, że w obrębie jednego paradygmatu wytworzono nadparadygmaty w wyniku ewolucji. Przykładem może być **programowanie proceduralne** i **strukturalne**, które są praktycznie jedynie rozszerzeniem paradygmatu imperatywnego. Również spotykane są style programowania, które stanowią połączenie wybranych cech spośród różnych, często odmiennych paradygmatów pisania programów.

W rozdziale tym zostaną przybliżone ogólne zasady pisania programów komputerowych w oparciu o paradygmaty stosowane najczęściej do rozwiązywaniu zadań inżynierskich.

4.2. Programowanie proceduralne

Programowanie proceduralne jest zasadniczym stylem pisania programów opartym o paradygmat imperatywny. Idea omawianego paradygmatu polega na tym, że program komputerowy jest postrzegany jako model pewnego fragmentu świata rzeczywistego. W świecie tym występują określone obiekty (np. zjawiska, przedmioty itp.), które z upływem czasu zmieniają swój stan. Oznacza to, że poprzez programowanie imperatywne programista modeluje [2, 28, 30]:
- obiekty rzeczywiste – za pomocą instancji zmiennych, które są abstrakcją komórek pamięci komputera;
- zmiany stanu obiektów – wykonując operacje przypisania wartości do zmiennych;
- czas – w efekcie sekwencyjnego wykonywania kolejno następujących instrukcji programu, które odpowiadają za zmianę stanu.

Podsumowując, istotą omawianego stylu programowania są zmienne oraz instrukcje wykonywane krok po kroku. Efektem działania programu, powstałego

w ten sposób, jest ciągła zmiana stanu komputera (tj. w funkcji czasu), aż do uzyskania oczekiwanego wyniku. Taki styl programowania jest naturalny dla budowy sprzętu komputerowego o architekturze J. von Neuman'a [19, 28].

W wyniku rozwoju zarówno komputerów, jak i języków programowania, następowała równoległa ewolucja programowania imperatywnego. Powstanie nadparadygmatu programowania proceduralnego wzbogaciło omawiany styl tworzenia programów o nowe elementy służące do sterowania wykonywania instrukcji, w szczególności takie jak:

- procedury i funkcje, które są zasadniczą abstrakcją programowania;
- instrukcje skoku warunkowego;
- instrukcje powtórzenia (tzw. pętle).

Jak wiadomo, procedurą nazywa się uporządkowany blok sekwencji instrukcji programu. W zasadzie procedura jest abstrakcją instrukcji elementarnej, ponieważ zadeklarowaną procedurę można używać w dowolnym miejscu programu. Procedura posiada swoją nazwę, za pomocą której jest wywoływana, a za pośrednictwem parametrów do procedury przekazuje się wartości dowolnych zmiennych. Jeśli działanie procedury kończy się wygenerowaniem wyniku w postaci wartości zwracanej, to procedurę taką nazywa się funkcją. Nadrzędną procedurą w każdym programie jest procedura o nazwie „*main*". Uruchomienie programu powoduje automatyczne wywołanie tej procedury.

Poniższy listing 4.1 przedstawia przykładową postać programu proceduralnego, który został zapisany za pomocą języka Visual Basic.

Listing 4.1.

```
01 Sub Main ()
02    Dim zmienna1 As Single
03    Dim zmienna2 As Single
04    zmienna1 = Input()
05    zmienna2 = Pierwiastek(zmienna2)
06    Print(zmienna2)
07 End Sub

08 Function Pierwiastek(parametr As Single) As Single
09    Pierwiastek = parametr ^ 0.5
10 End Function
```

Linia pierwsza listingu 4.1 rozpoczyna procedurę główną programu (tj. „*Main*"), która kończy się słowem kluczowym umieszczonym w linii 7. Linia druga i trzecia zawiera instrukcję deklaracji zmiennych programu, które będą przechowywały wartości liczb zmiennoprzecinkowych pojedynczej precyzji. Deklaracja zmiennej polega na zdefiniowaniu zasięgu (lokalnego, w obrębie procedury lub w obrębie programu), nazwy oraz sposobu przechowywania wartości w pamięci danych (poprzez określenie typu danych).

Kolejna linia 4 przykładowego programu zawiera instrukcję przypisu. Zmiennej o nazwie „*zmienna1*” zostanie przypisana wartość wygenerowana przez funkcję wbudowaną „*Input*”. Funkcja ta realizuje operację wejścia danych do programu (np. wprowadzonych z klawiatury). Kolejna linia programu (nr 5) zawiera instrukcję przypisania, w której wartość wstawiana do zmiennej „*zmienna2*” jest wartością zwracaną przez wywołaną funkcję użytkownika o nazwie „*Pierwiastek*”. Funkcja ta jest zdefiniowana poza procedurą główną – są to linie 8÷10 listingu 4.1. Definicja funkcji polega na deklaracji zasięgu, nazwy, parametrów i typu danych zwracanej wartości oraz podania sekwencji instrukcji (tzw. implementacja procedury), przy czym jedna instrukcja musi generować wynik zwracany (linia 9).

Ostatnia linia procedury „*Main*” (nr 6) realizuje operację wyjścia danych (np. na ekran). Wywołanie procedury wbudowanej „*Print*” powoduje, że wynik działania programu zostanie wyświetlony w tym przypadku na ekranie.

Cechą charakterystyczną programowania proceduralnego jest to, że parametry procedury stanowią jej zmienne wewnętrzne. Podczas wywoływania procedury, parametrom tym przypisuje się wartości – tak jak np. w linii 5 listingu 4.1. Inną cechą tego stylu programowania jest przyzwolenie na nieograniczone stosowanie instrukcji skoku (w Visual Basic są to instrukcja „*Goto*” oraz konstrukcja „*GoSub…Return*”). Modyfikację listingu 4.1, uwzględniającą instrukcję skoku, przedstawiono w listingu 4.2.

Listing 4.2.

```
01 Sub Main ()
02    Dim zmienna1 As Single
03    Dim zmienna2 As Single
04    zmienna1 = Input()
05    GoSub Etykieta
06    Print(zmienna2)
07    Exit Sub
08 Etykieta:
09    zmienna2 = zmienna1 ^ 0.5
10    Return
11 End Sub
```

Linia piąta wymusza sok do miejsca oznaczonego w programie przez etykietę (linia 8) – w tym przypadku do wyrażenia umieszczonego w linii 9. Następnie, napotkane słowo kluczowe „*Return*” (linia 10) powoduje powrót do linii 6. Kolejne napotkane słowo kluczowe „*Exit Sub*” odpowiada za przerwanie wykonywania programu.

Analizując powyższe przykłady można wykazać, że programowanie proceduralne charakteryzuje się następującymi cechami:

- zmienne nie są porządkowane i grupowane;
- działanie programu opera się na instrukcjach, które przetwarzają zmienne;

- instrukcje są umieszczane w kolejnych liniach kodu, ale ich kolejność nie zawsze wykazuje logiczną ciągłość;
- stosowanie w nadmiernej ilości instrukcji skoku dodatkowo zaburza logikę programu.

Większość przytoczonych cech uważa się za wady. Konsekwencją takiego stylu programowania jest utrudnienie późniejszej konserwacji programu. W skrajnych przypadkach jakakolwiek modyfikacja oprogramowania jest wręcz niemożliwa – szczególnie, gdy kod źródłowy nie zawiera komentarzy.

4.3. Programowanie strukturalne

4.3.1. Charakterystyka ogólna

Programowanie strukturalne to filozofia tworzenia programów oparta na założeniu, że programy komputerowe powinny mieć ściśle ustaloną formę. Efektem programowania opartego na tej technice jest „dobrze" napisany i czytelny program, który bazuje na trzech zasadniczych konstrukcjach: **sekwencji**, **selekcji** oraz **iteracji**. Dodatkowo kod programu jest uzupełniony o wyczerpujący komentarz. Oczywiście programowanie strukturalne jest kolejnym nadparadygmatem imperatywnym, lecz pozbawionym zasadniczych wad programowania proceduralnego [5, 28, 30].

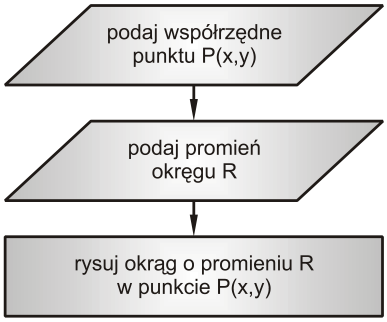
Obecnie paradygmat programowania strukturalnego wykorzystuje się do tworzenia małych, nieskomplikowanych programów i procedur, np. w postaci makr dołączanych do aplikacji. W oparciu o tą technikę programowania zaleca się również implementować algorytmy (w szczególności numeryczne) oraz procedury (metody) w obrębie klasy. Natomiast już przy programowaniu rozbudowanych i zaawansowanych aplikacji specjalistycznych jest konieczne zastosowanie innego paradygmatu – najczęściej obiektowego. Warto podkreślić, że wszystkie wymienione wyżej style programowania należy traktować jako efekt ewolucji zasad projektowania i programowania komputerów.

4.3.2. Zasadnicze konstrukcje programowania

Sekwencja

Sekwencją nazywa się co najmniej dwie instrukcje następujące jedna po drugiej tworzące logiczny ciąg. Konstrukcja ta jest uważana za podstawowy i zarazem najłatwiejszy do uzyskania składnik programu spośród trzech kluczowych konstrukcji programowania strukturalnego. Zaleca się, aby stosować sekwencję zawsze, kiedy to tylko jest możliwe. Celem stosowania sekwencji w programowaniu jest uzyskanie logicznego i przejrzystego programu.

Przykład omawianej konstrukcji, którą zapisano przy pomocy „sieci działań”, przedstawiono na rys. 4.1. Kolejność operacji wykonanych w ramach tego zadania jest zdeterminowana logiką jego rozwiązania. Zgodnie z zasadami programowania strukturalnego, wskazane jest aby dane oraz procedury były w sposób logiczny pogrupowane. Oznacza to, że instancja punktu „*P*(*x*, *y*)” powinna być zadeklarowana jako zmienna typu użytkownika, a operację „*rysuj*” należy uprzednio zadeklarować jako procedurę z możliwością przekazywania wartości do jej parametrów. Przykładową implementację przykładu z rysunku 4.1, zapisaną za pomocą języka Visual Basic, przedstawiono w listingu 4.3. Analiza tego zapisu wykazuje, że program posiada cechy konstrukcji sekwencji. Po pierwsze, zapis charakteryzuje się logicznym następstwem instrukcji, wyrażeń itd. Po drugie, program cechuje organizacja danych i procedur, a kod dopełniony jest komentarzami. Wszystkie te cechy, które są zgodne z paradygmatem programowania strukturalnego, sprawiają, że konserwacja tego programu jest ułatwiona.



Rys. 4.1. Przykład konstrukcji sekwencji; opis w tekście

Listing 4.3.

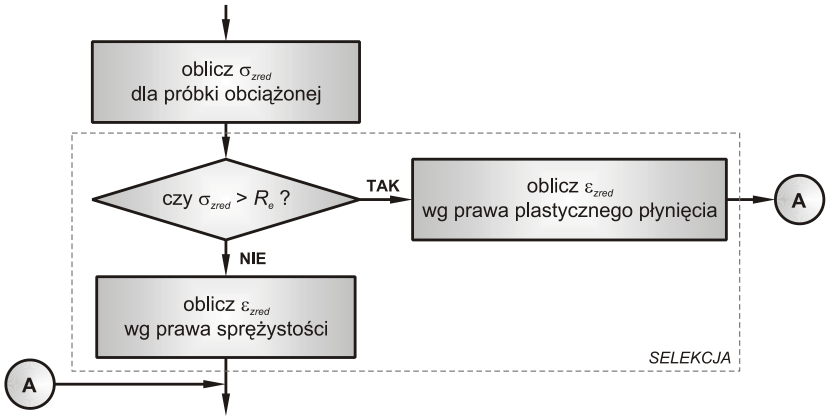
```
Type Punkt          'definicja struktury, abstrakcyjny typ danych
   X As Single
   Y As Single
End Type

Sub Main ()
   Dim P As Punkt        'deklaracja zmiennych programu
   Dim R As Single
     P.X = Input()       'wartości współrzędnych X i Y podane przez
     P.Y = Input()       'użytkownika programu
     R = Input()         'pobranie wartości promień R
     Rysuj(P, R)         'wywołanie procedury użytkownika
End Sub

Sub Rysuj (P As Punkt, R As Single)  'definicja procedury użytkownika
...Dim i As Integer                  'zmienna określająca grubość linii
   Dim k As Long                     'zmienna określająca kolor linii
   i = Input()                       'pobranie wartości podanych przez
   k = Input()                       'użytkownika programu
   Ekran.DrawWidth = i               'przygotowanie do rysowania na
   Ekran.ForeColor = k               'ekranie monitora
   Circle(P.X, P.Y, R)               'rysowanie okręgu
End Sub
```

Selekcja

Selekcją nazywa się taką konstrukcję programu, która składa się przynajmniej z jednego zdania decyzyjnego (instrukcji warunkowej). Podjęcie decyzji następuje na podstawie zadanego pytania, na które są tylko dwie możliwe odpowiedzi – „tak" (prawda) lub „nie" (fałsz). Zastosowanie w tym przypadku logiki klasycznej jest zgodne z binarną naturą procesora. Skutkiem podjęcia decyzji jest wykonanie określonej instrukcji (lub sekwencji instrukcji) z jednoczesnym pominięciem instrukcji alternatywnej. Dla implementacji zadań inżynierskich, selekcja odgrywa istotną rolę w budowie algorytmów [5, 16, 33].

Podstawową konstrukcję selekcji przedstawiono na rys. 4.2, w formie „sieci działań", na przykładzie wyznaczania odkształcenia zredukowanego ε_{zred} w oparciu o naprężenie zredukowane σ_{zred}. Sprawdzając, czy wartość σ_{zred} jest większa od granicy plastyczności R_e, wybiera się jedną z dwóch metod obliczeń. Konstrukcję selekcji zapisaną w języku Visual Basic, za pomocą instrukcji warunkowej, umieszczono w listingu 4.4.

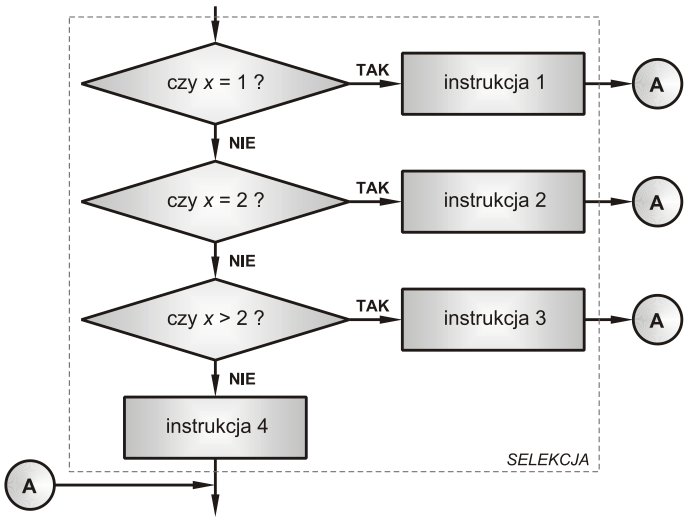
W obliczeniach inżynierskich dość często ma się do czynienia z sytuacją, gdzie należy podjąć decyzję w oparciu o parametr, który może przyjmować różne wartości. Wynikiem podjęcia decyzji jest wykonanie jednej spośród wielu możliwych operacji. W takim przypadku należy stosować złożoną konstrukcję selekcji, której przykład przedstawiono w formie ogólnej na rys. 4.3. Instrukcje oznaczone cyframi od 1 do 3 reprezentują bloki programu, które są wykonywane, gdy odpowiedź na dane pytanie jest pozytywna. Instrukcja numer 4 jest blokiem programu wykonywanym jeśli żaden z warunków nie zostanie spełniony.



Rys. 4.2. Przykład konstrukcji selekcji; opis w tekście

Listing 4.4.

```
Sub Main ()
   Dim Sigma   As Single                        'deklaracja zmiennych programu
   Dim Epsilon As Single
   Const Re As Single = 300                     'instrukcja deklaracji stałej
   If Sigma > Re Then                           'początek instrukcji warunkowej
       Epsilon = PrawoPlastPlyniecia(Sigma)     'wywołanie funkcji pierwszej
   Else
       Epsilon = PrawoHooka(Sigma)              'wywołanie funkcji alternatywnej
   End If                                       'koniec instrukcji warunkowej
End Sub
```



Rys. 4.3. Przykład złożonej konstrukcji selekcji; opis w tekście

Złożoną konstrukcję selekcji, na przykładzie „sieci działań” z rys. 4.3, można zapisać używając języka Visual Basic na dwa sposoby. Podstawy sposób polega na zastosowaniu instrukcji warunkowej typu „*If*” , którą przedstawiono w listingu 4.5. Drugim sposobem realizacji złożonej konstrukcji selekcji jest użycie instrukcji warunkowej typu „*Select Case*” (listing 4.6), która jest mniej uniwersalna niż instrukcja typu „*If*”.

Listing 4.5.

```
Dim X As Integer          'deklaracja zmiennej (parametru)
X = PrzypiszWartosc()     'wywołanie określonej funkcji
  If X = 1 Then
     instr_1                'wywołanie instrukcji nr 1
  ElseIf X = 2 Then
     instr_2                'wywołanie instrukcji nr 2
  ElseIf X > 2 Then
     instr_3                'wywołanie instrukcji nr 3
  Else
     instr_4                'wywołanie instrukcji nr 4
  End If
```

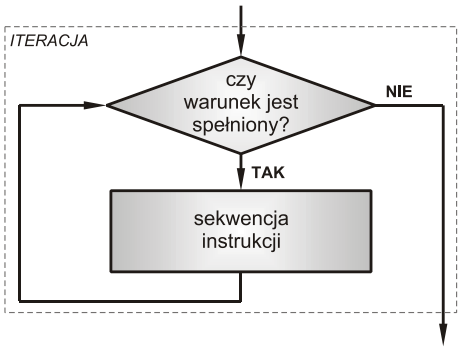
Listing 4.6.

```
Dim X As Integer           'deklaracja zmiennej (parametru)
X = PrzypiszWartosc()      'wywołanie określonej funkcji np. przypisu
  Select Case X
  Case 1: instr_1             'wywołanie instrukcji nr 1
  Case 2: instr_2             'wywołanie instrukcji nr 2
  Case Is > 2: instr_3        'wywołanie instrukcji nr 3
  Case Else: instr_4          'wywołanie instrukcji nr 4
  End Select
```

Iteracja

Iteracją nazywa się taką konstrukcję programu, która składa się z instrukcji powtórzenia – popularnie nazywaną pętlą. Instrukcja ta realizuje cykliczne wykonywanie pewnych czynności komputera (sekwencji instrukcji), przy czym ilość cykli jest określana na podstawie warunku kontynuacji (lub zakończenia) iteracji. Omawiana konstrukcja jest stosowana zawsze w przypadkach wykonywania dużej liczby jednakowych operacji. Ponadto, umożliwia ona skrócenie zapisu programu, w efekcie czego program jest bardziej czytelny.

Podstawową konstrukcję iteracji przedstawiono na rys. 4.4 w postaci „sieci działań". Konstrukcja ta jest jedyną, w której stosuje się wyjątek od reguły mówiący, że ciąg logiczny działań powinien przebiegać z góry na dół oraz od lewej do prawej. Instrukcja



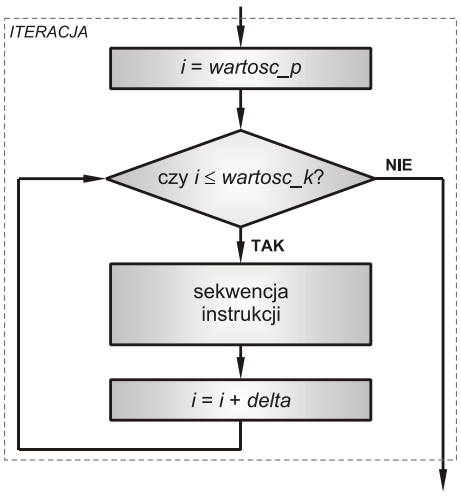
Rys. 4.4. Przykład konstrukcji iteracji; schemat typu: „dopóki warunek spełniony wykonuj powtórzenie"

powtórzenia według omawianego schematu polega na wielokrotnym wykonywaniu bloku instrukcji, aż do momentu, kiedy warunek będzie spełniony. Jest to warunek kontynuacji powtórzeń, gdzie wartość testowanej zmiennej (o dowolnym typie) jest modyfikowana przez odpowiednią instrukcję zawartą wewnątrz iteracji. Najczęściej jednak zmienna ta jest modyfikowana przy okazji wykonywania bloku instrukcji głównych – np. podczas wczytywania danych z pliku do pamięci komputera sprawdza się czy wskaźnik pozycji odczytu danych nie osiągnął końca pliku. Ogólną postać omawianej instrukcji iteracji, zapisaną w języku Visual Basic, przedstawiono w listingu 4.7.

Listing 4.7.

```
01 Dim X As Boolean                ‘deklaracja zmiennej testowanej
02   Do While X = True             ‘sprawdzenie warunku kontynuacji powtórzenia
03      ‘sekwencja instrukcji
04      ‘instrukcja modyfikująca zmienną
05   Loop
```

Szczególną wersją wyżej omówionego schematu konstrukcji iteracji jest instrukcja powtórzenia przedstawiona na rys. 4.5 (w postaci sieci działań). Jest to schemat najczęściej stosowany podczas pisania programu. Charakteryzuje się on tym, że do konstrukcji iteracji wprowadza się zmienną kontrolną „*i*”, która jest typu liczbowego. Zmiennej tej jest nadawana wartość początkowa „*wartosc_p*”, która musi być mniejsza od wartości końcowej „*wartosc_k*”. Po każdorazowym wykonaniu sekwencji instrukcji, wartość zmiennej kontrolnej „*i*” zostaje powiększona o wartość parametru kroku „*delta*”. Jeżeli nowa wartość zmiennej kontrolnej jest mniejsza od wartości końcowej, sekwencja instrukcji jest powtarzana do momentu uzyskania przez zmienną kontrolną „*i*” wartości równej lub większej od „*wartosc_k*”. Ogólną postać tej instrukcji przedstawiono w listingu 4.8.

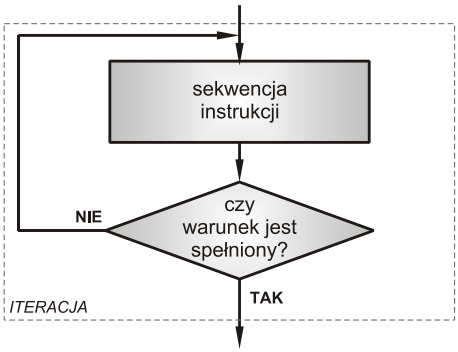


Rys. 4.5. Przykład konstrukcji iteracji; schemat typu: „dla zmiennej kontrolnej rozpoczynając od wartości początkowej zwiększając o przyrost aż do wartości końcowej wykonuj powtórzenie”

Listing 4.8.

```
Dim i As Integer     'deklaracja zmiennej kontrolnej
  For X = wartosc_p To wartosc_k Step delta
     'sekwencja instrukcji
  Next i
```

Innym schematem iteracji jest instrukcja powtórzenia, w której komputer wykonuje sekwencję instrukcji do momentu spełnienia określonego warunku. Konstrukcja ta, w formie sieci działań, przedstawiona jest na rys. 4.6. W odróżnieniu od konstrukcji iteracji omówionej wcześniej (na podstawie rys. 4.4), w tym przypadku sekwencja instrukcji jest wykonywana co najmniej jeden raz. Przykładem wykorzystania omawianego schematu iteracji, jest odczytywanie danych z pliku, gdzie warunkiem sprawdzanym jest osiągnięcie znacznika końca pliku. Ogólną postać instrukcji iteracji omawianego typu, zapisanej za pomocą języka Visual Basic, przedstawiono w listingu 4.9.



Rys. 4.6. Przykład konstrukcji iteracji; schemat typu: „powtarzaj powtórzenie aż do spełnienia warunku"

Listing 4.9.

```
Dim X As Boolean            'deklaracja zmiennej testowanej
  Do Until X = True         'sprawdzenie warunku zakończenia powtórzenia
     'sekwencja instrukcji
     'instrukcja modyfikująca zmienną
  Loop
```

4.4. Programowanie sterowane przepływem danych

Zgodnie z paradygmatem programowania „sterowanym przepływem danych", program jest spostrzegany jako graf, którego wierzchołki reprezentują moduły, natomiast jego krawędzie przedstawiają przepływ danych. Pod pojęciem „moduł" należy rozumieć zgrupowaną sekwencję instrukcji. W ogólnym przypadku, wyróżnia się dwa typy modułów:

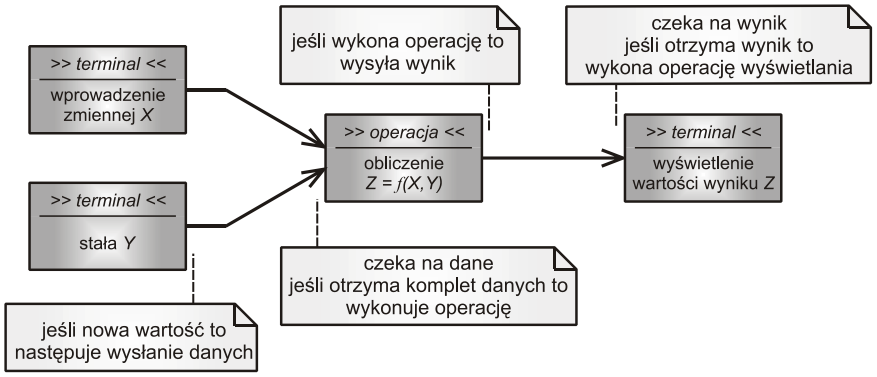
- **terminal**, który jest tzw. punktem wprowadzenia (wyprowadzenia) danych do programu (z programu);
- **operacja**, która realizuje określony proces przetwarzania danych.

Na rys. 4.7 umieszczono przykładowy schemat ideowy, wyjaśniający istotę tego paradygmatu programowania. Pokazuje on, że terminal może zarówno realizować proces komunikacji między użytkownikiem a programem, jak i pełnić rolę zasobnika danych, które są stałe w czasie wykonywania programu. Natomiast moduł „*operacja*” uaktywnia się tylko w przypadku, jeżeli otrzyma komplet danych. Po zakończeniu obliczeń, moduł ten przekazuje wynik do odpowiedniej ścieżki przepływu danych oraz przechodzi w stan oczekiwania na nowe dane wejściowe. Podsumowując, moment wykonania dowolnej operacji nie zależy od liniowej sekwencji instrukcji (np. jak w programowaniu imperatywnym), ale od dostępności danych. Wygenerowanie danych przez dowolny moduł powoduje uruchomienie kolejnych, powiązanych modułów [12, 13, 27, 28].

Paradygmat ten opisuje pewien abstrakcyjny model programowania, który można zobrazować np. poprzez analogię do linii produkcyjnej. Określona operacja (czynność, zabieg) na danym stanowisku roboczym jest wykonywana wtedy, gdy przy tym stanowisku pojawi się obrabiany przedmiot. Zatem można przyjąć, że dane są abstrakcją „przedmiotu obrabianego”, a moduły – abstrakcją „stanowisk roboczych”.

Programowanie za pomocą systemu LabVIEW®

Przykładem praktycznego zastosowania omawianego podejścia do programowania jest tworzenie programów za pomocą graficznego języka G w środo-



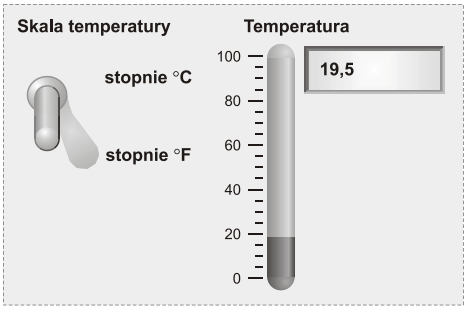
Rys. 4.7. Przykład w postaci grafu (wraz z komentarzami) obrazujący istotę programowania sterowanego przepływem danych

wisku systemu LabVIEW® (**Lab**oratory **V**irtual **I**nstrumentation **E**ngineering **W**orkbench). Jest to aplikacja niezwykle obszerna i rozbudowana, której podstawowym przeznaczeniem jest tworzenie programów służących do wykonywania pomiarów, sterowania urządzeniami oraz przetwarzania sygnałów. Aby realizacja takich zadań była możliwa, LabVIEW posiada biblioteki narzędzi służących do współpracy z szerokim zakresem sprzętu kontrolno-pomiarowego, takiego jak: karty przetworników A/C i C/A, moduły akwizycji danych, sterowniki programowalne, sieci czujników bezprzewodowych, kamery itd. Ponadto, aplikacja ta umożliwia programowanie komunikacji w standardach takich jak np. RS232, GPIB, USB oraz wykorzystując sieć internetową poprzez korzystanie np. z protokołu TCP-IP [12, 27].

Idea programowania z użyciem języka G polega na zaprojektowaniu „panelu czołowego", który jest interfejsem użytkownika – oraz stworzeniu „diagramu blokowego", który reprezentuje właściwy kod programu. Wszystkie komponenty tego diagramu są przedstawiane w LabVIEW za pomocą symboli graficznych. Typowy diagram składa się z [12]:

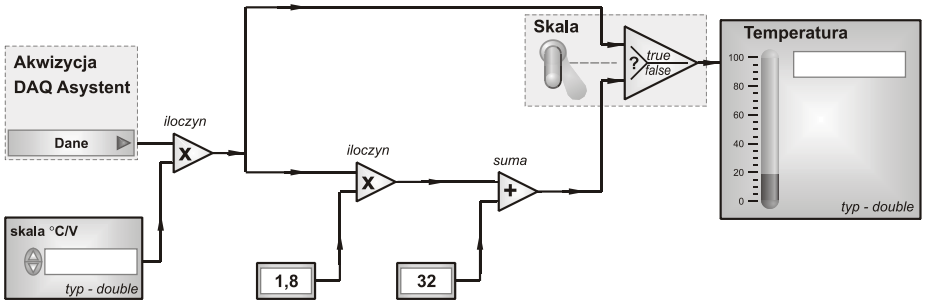
- modułów, które odpowiadają za wykonywanie określonych operacji lub są terminalami dla wprowadzanych lub wyprowadzanych danych;
- przewodów, które reprezentują ścieżkę przepływu danych pomiędzy modułami;
- instrumentów wirtualnych, które są z reguły uporządkowanym układem modułów i przewodów.

Poniżej zostanie omówiony przykładowy program, stworzony w systemie LabVIEW. Program ten służy do pomiaru i wyświetlenia wartości temperatury w wybranej skali. Na rys 4.8 przedstawiono wygląd panelu czołowego, natomiast diagram blokowy przedstawiono schematycznie na rys. 4.9. Na schemacie tym, oprócz typowych komponentów diagramu, umieszczono również elementy graficzne zwiększające czytelność przykładu. Do budowy przykładowego programu użyto następujące, typowe moduły:



Rys. 4.8. Zrzut ekranu zawierający panel czołowy wykonany w systemie LabVIEW; opis w tekście [26]

- „DAQ Asystent" – moduł reprezentujący wirtualny obiekt służący do akwizycji pomiarów odczytywanych z karty pomiarowej, do której podłączony jest czujnik



Rys. 4.9. Schematyczne przedstawienie przykładowego programu utworzonego za pomocą diagramu blokowego w systemie LabView; opis w tekście

temperatury; moduł ten jest specjalistycznym narzędziem dołączanym do biblioteki systemu LabVIEW;

- moduł terminalny „***skala °C/V***”, do którego jest wprowadzana wartość liczbowa służąca do przeliczenia pomiaru, odczytanego przez moduł „DAQ Asystent”, na wartość wyrażającą temperaturę w skali Celsjusza – moduł ten nie jest widoczny na przykładowym panelu czołowym;
- moduł terminalny „***temperatura***”, który służy do wyświetlania wyniku obliczeń – moduł ten jest widoczny na panelu czołowym jako pole tekstowe; komponent ten jest połączony z dodatkowym obiekt graficznym, który jest abstrakcją osi liczbowej;
- dwa moduły terminalne, w których są wprowadzone numeryczne stałe liczbowe – te moduły nie są widoczne na panelu czołowym;
- trzy moduły operacji arytmetycznych realizujących mnożenie lub dodawanie danych przychodzących z podłączonych przewodów – operacja jest wykonywana w momencie, kiedy na wejściu pojawi się komplet danych; uzyskany wynik zostaje wysłany przewodem podłączonym na jego wyjściu;
- moduł „***skala***”, który steruje przepływem danych; realizuje on typową konstrukcję selekcji, jaka jest stosowana np. w paradygmacie imperatywnym.

Cenną zaletą systemu LabView jest to, że utworzony program można zapisać do pliku (o określonym rozszerzeniu), a następnie zaimportować go do nowego projektu jako integralny moduł użytkownika. Więcej szczegółów na temat tego systemu można znaleźć np. w literaturze specjalistycznej [27] lub na stronie internetowej firmy *National Instruments* [12].

4.5. Programowanie zorientowane obiektowo

4.5.1. Charakterystyka ogólna

Programowanie komputerów zorientowane obiektowo jest techniką tworzenia programów, których struktura bazuje na zbiorze obiektów. Zakłada się, że dane wraz z procedurami należy grupować w specjalne bloki, a czynność tą należy wykonywać w sposób przemyślany, tak aby uzyskać pełną kontrolę nad danymi. Istotne jest to szczególnie podczas opracowywania dużych, rozbudowanych aplikacji.

Pod pojęciem „**obiekt**" należy rozumieć wydzielony blok programu, który składa się z danych (atrybutów, własności – z ang. *properties*) oraz z procedur lub funkcji (operacji, metod – z ang. *methods*). Zadaniem metod jest przetwarzanie danych obiektu w pełni kontrolowany sposób oraz nadzorowanie dostępu do obiektu. Przyjmuje się, że dane znajdujące się w obiekcie reprezentują jego stan. Natomiast wywołanie dowolnej metody realizowanie jest poprzez tzw. wysłanie do obiektu komunikatu (z ang. *message*). Komunikat ten jest wysyłany albo przez użytkownika danego programu, albo przez system operacyjny (lub inną aplikację). Najczęściej do tego celu wykorzystuje się tzw. zdarzenia (z ang. *events*), które służą do powiadomienia o wystąpieniu określonej zmiany stanu obiektu.

Początki obiektowego podejścia do pisania programów komputerowych sięgają lat sześćdziesiątych ubiegłego wieku. Nie od razu ta technika programowania zdobyła duże uznanie wśród programistów. Powodem tego było między innymi przekonanie, że nie ma potrzeby przebudowywać programu, który dobrze działa pomimo, że został on stworzony inną nieobiektową techniką. Dodatkowym argumentem jest fakt, że zmiana paradygmatu programowania wymusza od programisty również zmianę jego sposobu myślenia [6, 7, 29].

Wraz z postępem rewolucji informacyjnej [3], przypadającej na drugą połowę XX wielu oraz początek XXI wieku, wzrastało znaczenie omawianej techniki tworzenia programów. Konieczność budowy złożonych aplikacji, które musiały poradzić sobie z efektywnym przetwarzaniem i archiwizowaniem dużej ilości danych, przesyłaniem informacji itd. spowodowało, że programiści zaczęli na dużą skalę używać paradygmatu programowania obiektowego [13, 28].

Początkowo, większość nowo powstałych programów komputerowych miały jedynie znamiona obiektowości. Aplikacje te zawierały zarówno typowe obiekty jak i również tzw. obiekty osłonowe. Pod pojęciem „**obiekt osłonowy**" należy rozumieć taki kod obiektowy programu, który zawiera w swoim wnętrzu inny kod, najczęściej w postaci modułu strukturalnego (lub proceduralnego) [29]. Celem stosowania takiego zabiegu jest to, aby taki moduł programu wyglądał i zachowywał się jak obiekt. Przykładem obiektów osłonowych są instancje

modułu klasy utworzone za pomocą języka Visual Basic w wersji oznaczonej numerem 6 lub w wersji VBA. Innym przykładem takiego zabiegu jest tzw. „opakowanie” starych systemów komputerowych, napisanych językiem nieobiektowym, w specjalne obiekty osłonowe (z ang. *wrapper*) [29].

4.5.2. Projektowanie klasy

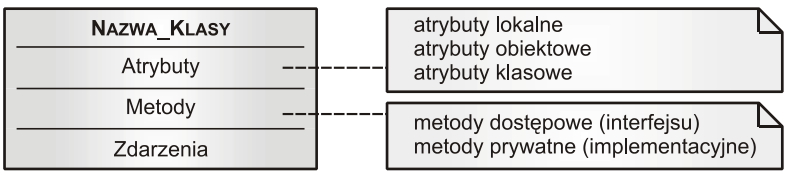
Definicja klasy

Klasa jest to kod programu (tzw. byt programistyczny [25]) definiujący budowę obiektu. Pełni ona rolę szablonu służącego do utworzenia dowolnej ilości obiektów. Zgodnie z [13, 16, 18, 29, 30] klasę można spostrzegać jako abstrakcyjny typ danych wyższego rzędu. Z kolei obiekt jest instancją (egzemplarzem – z ang. *instance*) typu danej klasy. W nomenklaturze programowania obiektowego przyjęto, że proces tworzenia obiektu (egzemplarza) nazywa się wystąpieniem klasy (z ang. *instantiation*).

Pojęcie „**abstrakcja**” jest nierozłącznym elementem każdego paradygmatu programowania. W przypadku programowania obiektowego, obiekty w programie komputerowym reprezentują zazwyczaj rzeczywiste obiekty, spotykane w życiu codziennym. Przykładem takiego obiektu może być maszyna, np. prasa, obrabiarka lub nawet pojazd mechaniczny. Obiekt taki posiada odpowiednie cechy i może wykonywać różne działania. Stosując proces abstrakcji podczas programowania, można zbudować taką klasę, której cechy obiektu oraz jego zadania są zredukowane do minimum zdeterminowanego przez dane zadanie programistyczne. Zatem, definiowanie klasy jest w istocie podobnym zadaniem jak budowa modelu w procesie modelowania numerycznego.

Anatomia klasy

Na rys. 4.10 przedstawiono schematycznie najważniejsze elementy składowe klasy, które mogą w niej występować. Tymi elementami są unikatowa nazwa klasy oraz definicje atrybutów, metod oraz zdarzeń. Przykładowa definicja klasy



Rys. 4.10. Schematyczne przedstawienie najważniejszych elementów składowych klasy za pomocą diagramu UML; opis w tekście

o nazwie „*Prasa*" została przedstawiona w listingu 4.10. Do zapisu przykładu użyto języka obiektowego Visual Basic .NET. Dla uproszczenia przekazu, celowo pominięto deklaracje ewentualnych zdarzeń.

Nazwa klasy pełni rolę identyfikacyjną i jest wykorzystywana podczas tworzenia obiektu (listing 4.11). Atrybuty zdefiniowane wewnątrz klasy, w zależności od ich zasięgu, mogą być (listing 4.10):

- **lokalne**, które są definiowane wewnątrz metody; często nazywa się je własnościami metody [25];
- **obiektowe** – są one definiowane na początku klasy, a każde kolejne wystąpienie klasy powoduje przydzielenie oddzielnego miejsca w pamięci komputera (tj. dla każdego obiektu osobno);
- **klasowe** – są zadeklarowane jako zmienne statyczne, co oznacza, że każdy utworzony obiekt odwołuje się do jednej zmiennej (tj. tego samego miejsca w pamięci).

Listing 4.10.

```
Public Class Prasa                    ‘ <- nazwa klasy / początek klasy
  Private strNazwa As String          ‘ <- atrybut obiektowy
..Static iDKuźni   As Integer         ‘ <- atrybut klasowy
  ‘--- poniżej: metody dostępne (interfes klasy) -------------------------
   Public Sub Kucie()
      Dim iDOdkuwki As Integer     ‘ <- atrybut lokalny
      ‘sekwencja instrukcji
   End Sub
   Public Property Nazwa() As String
      Get                          ‘przekazywanie wartości atrybutu
        Return strNazwa
      End Get
      Set (Wartosc As String)      ‘pobieranie wartości atrybutu
        strNazwa = Wartosc
      End Set
   End Property
  ‘--- poniżej: metody prywatne (tzw. implementacja klasy) ---------------
  Private Sub PobierzOdkuwke()
     ‘sekwencja instrukcji
  End Sub
End Class
```

Listing 4.11.

```
Sub Main()
   Private PrasaPXW As New Prasa     ‘utworzenie obiektu / wystąpienie klasy
   PrasaPXW.Nazwa = “PXW-100A”       ‘przypisanie wartości
End Sub
```

Atrybuty obiektowe lub klasowe (własności), które pełnią rolę interfejsu publicznego klasy, należy udostępniać jedynie za pośrednictwem instrukcji dostępowej. Przykład zapisu takiej instrukcji w języku Visual Basic .NET zamieszczono w listingu 4.10 (linie 9÷16).

Natomiast metody deklarowane w klasie, w zależności od pełnionej roli, dzieli się na metody:

- **dostępowe** – są to procedury (funkcje) widoczne przez inne obiekty, dzięki temu mogą tworzyć tzw. interfejs publiczny klasy (obiektu) służący do bezpiecznej, kontrolowanej komunikacji między obiektami;
- **prywatne** – są one zadeklarowane jako procedury (funkcje) o zasięgu lokalnym i stanowią implementację klasy; dzięki temu inne obiekty nie mają do nich bezpośredniego dostępu.

Wytyczne projektowania klasy

Jednym z najważniejszych celów programowania zorientowanego obiektowo jest modelowanie rzeczywistych systemów poprzez zastosowanie takiego sposobu myślenia, które jest naturalne dla ludzi. Podstawową metodą osiągnięcia takiego celu jest **projektowanie klas** z zachowaniem elementarnych zasad paradygmatu obiektowości [13, 28, 29]. Istotą obiektowości jest to, że klasy stanowią model obiektów rzeczywistych oraz ich wzajemnych zachowań. Zatem, program składający się z egzemplarzy klas nie jest wykonywany w sposób sekwencyjny, tak jak w przypadku programów imperatywnych, ale oczekuje na zaistnienie zdarzenia, po czym wykonuje określone działanie zdefiniowane w klasie.

Podczas projektowania klasy należy stosować następujące wytyczne:

- klasa powinna reprezentować prawdziwe zachowania obiektu świata rzeczywistego, co spowoduje, że interakcje pomiędzy wystąpieniami klasy będą przypominały naturalną formę komunikowania się;
- modelując postać klasy należy stosować abstrakcję;
- implementację klasy należy ukrywać, co jest realizowane poprzez deklarowanie atrybutów i metod o zasięgu lokalnym (tzw. mechanizm hermetyzacji);
- interfejs publiczny (tj. atrybuty i metody dostępowe) powinien być zminimalizowany;
- klasa powinna zawierać mechanizm obsługi błędów;
- należy projektować niezawodne konstruktory i destruktory (tj. specjalne metody inicjujące i kasujące wystąpienie klasy);
- należy stosować mechanizmy takie jak dziedziczenie, polimorfizm, kompozycja;

- kod klasy powinien być czytelny, poszczególne sekcje powinny być wyodrębnione oraz należy klasę dokumentować stosując komentarze.

Proces projektowania programu komputerowego zorientowanego obiektowo można podzielić na następujące ogólne etapy [29]:

- wykonanie analizy zadania projektowego;
- opracowanie zakresu planowanych prac projektowych;
- gromadzenie wymagań dotyczących zaplanowanych prac;
- opracowanie prototypu interfejsu publicznego;
- identyfikacja klas;
- określenie zakresu funkcjonalnego każdej klasy;
- określenie wzajemnych interakcji pomiędzy poszczególnymi klasami;
- utworzenie kompletnego modelu programu komputerowego, który ma zostać zbudowany – najlepiej stosując język UML.

Szczegóły dotyczące zasad projektowania systemów komputerowych, stosując opisywany paradygmat programowania, są zawarte w literaturze specjalistycznej np. [25, 29].

4.5.3. Zasadnicze mechanizmy programowania obiektowego

Hermetyzacja

Hermetyzacją danych (z ang. *encapsulation*) nazywa się mechanizm (proces), który polega na celowym ukrywaniu wybranych atrybutów i zachowań zdefiniowanych wewnątrz klasy. Dzięki temu programista może zadecydować, który zestaw danych jest niewidoczny dla kodu programu znajdującego się poza klasą. Z hermetyzacją danych są ściśle powiązane dwa pojęcia, o których już wcześniej wspomniano. Są to:

- **interfejs publiczny** – zestaw atrybutów, metod i zdarzeń, które są zadeklarowane jako dane publiczne; dzięki temu są one widoczne (dostępne) na zewnątrz klasy;
- **implementacja klasy** – zestaw danych i zachowań ukryty wewnątrz klasy.

Celem ukrywania wewnętrznej implementacji klasy jest:

- niedopuszczenie do niepotrzebnego, celowego lub przypadkowego manipulowania wewnętrznymi danymi i zawartością metod przez zewnętrzny kod programu;
- łatwiejsze zarządzanie kodem programu, ponieważ poszczególne obiekty są od siebie niezależne;
- wydajniejsza praca programisty, dzięki podzieleniu programu na mniejsze i niezależne części.

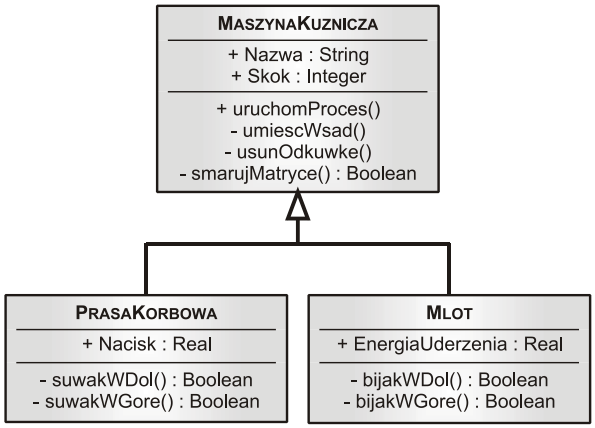
Aby w pełni uzyskać zalety płynące ze stosowania hermetyzacji danych, program komputerowy należy projektować według **modelu „interfejs – implementacja”**. W myśl tego modelu, raz zdefiniowany interfejs publiczny klasy nie może być zmodyfikowany. Natomiast wszelkie zmiany w implementacji klasy są dozwolone. Dzięki temu istnieje możliwość łatwej konserwacji programu komputerowego z zachowaniem jego pierwotnego szkieletu.

Dziedziczenie

Dziedziczeniem (z ang. *inheritance*) nazywa się mechanizm polegający na utworzeniu klasy na bazie innej klasy. Zdefiniowanie powiązań między klasami umożliwia wielokrotne wykorzystywanie kodu bez konieczności jego kopiowania. Takie projektowanie programu, opartego na klasach posiadających pewne wspólne cechy wielu obiektów, jest bardzo efektywne.

Dzięki temu mechanizmowi, projektowana podklasa (potocznie: klasa potomna, z ang. *children*) dziedziczy wszystkie atrybuty, metody i zdarzenia po wcześniej zdefiniowanej klasie nadrzędnej (nadklasie, potocznie również nazywana przodkiem lub rodzicem – z ang. *parent*). Podklasa może zmodyfikować to co odziedziczyła, dodać własne elementy lub usunąć niektóre odziedziczone.

Na rys. 4.11 umieszczono diagram UML przedstawiający relacje pomiędzy trzema przykładowymi klasami reprezentującymi rzeczywiste obiekty typu maszyny kuźnicze. Gdy spojrzy się na tak przedstawioną hierarchię dziedziczenia od dołu, będzie można zauważyć, że klasa nadrzędna (o nazwie „*MaszynaKuznicza*”) zawiera listę wspólnych cech swoich klas potomnych. Zatem, mechanizm dziedziczenia polega na tym, że atrybuty i metody, które są wspólne dla klas potomnych, zostają umieszczone w jednej klasie nadrzędnej. Ponieważ nadklasa jest bardziej ogólna niż jej klasy potomne, ten szczególny rodzaj abstrakcji jest nazywany również **uogólnieniem**.



Rys. 4.11. Diagram UML klas przykładowych maszyn kuźniczych

Implementacja omawianego związku pomię-

dzy wspomnianymi trzema klasami została umieszczona w listingu 4.12 (klasa nadrzędna) oraz w listingach 4.13 i 4.14 (klasy potomne). Należy zwrócić uwagę, że w listingu 4.12 w linii 4, jedna z instrukcji deklaracji metody zawiera dodatkowe słowo kluczowe „*overridable*". Ten sposób deklaracji przyzwala na późniejszą modyfikację tej metody przez klasę potomną. Zatem, w klasie potomnej (listing 4.13 i 4.14) występująca ponowna deklaracja tej metody musi posiadać słowo kluczowe „*overrides*", co w konsekwencji powoduje, że metoda ta przesłania wersję pierwotną, umieszczoną w klasie nadrzędnej.

Listing 4.12.

```
01 Public Class MaszynaKuznicza                ‘klasa nadrzędna (przodek)
02   Public Nazwa As String
03   Public Skok As  Integer
04   Public Overridable Sub UruchomProces()    ‘zezwolenie na modyfikację
05   End Sub
06   Private Sub umiescWsad()
07   End Sub
08   Private Sub usunOdkuwke()
09   End Sub
10   Private Function smarujMatryce() As Boolean
11   End Function
12 End Class
```

Listing 4.13.

```
01 Public Class PrasaKorbowa               ‘klasa podrzędna
02   Inherits MaszynaKuznicza              ‘dziedziczenie (związek typu „ma”)
03   Public Nacisk As Double
04   Public Overrides Sub UruchomProces()  ‘modyfikacja metody odziedziczonej
05   End Sub
06   Private Function suwakWDol() As Boolean
07   End Function
08   Private Function suwakWGore() As Boolean
09   End Function
10 End Class
```

Listing 4.14.

```
01 Public Class Mlot                        ‘klasa podrzędna
02   Inherits MaszynaKuznicza               ‘dziedziczenie (związek typu „ma”)
03   Public EnergiaUderzenia As Double
04   Public Overrides Sub UruchomProces() ‘modyfikacja metody odziedziczonej
05   End Sub
06   Private Function bijakWDol() As Boolean
07   End Function
08   Private Function bijakWGore() As Boolean
09   End Function
10 End Class
```

Kolejnym ważnym zagadnieniem dziedziczenia jest związek typu „**ma**” oraz „**jest**”. Utworzenie klasy potomnej poprzez uszczegółowienie (dziedziczenie właściwe) przedstawiono w listingach 4.13 i 2.14. W linii 2 znajduje się instrukcja „*inherits*” zawierająca nazwę klasy nadrzędnej. Informuje ona, że do klasy potomnej zostaną dołączone wszystkie atrybuty, metody i zdarzenia, jakie zadeklarowano w klasie nadrzędnej. W tym przypadku mówi się, że klasa potomna „ma” elementy klasy nadrzędnej. Natomiast związek typu „jest” to forma abstrakcji polegającą na tym, że w klasie potomnej deklaruje się nowy atrybut, który „jest” typu klasy nadrzędnej. Ten sposób budowy klasy potomnej zamieszczono w kolejnym listingu 4.15. Proszę zwrócić uwagę na linię 2 oraz linię 4. Sytuację tą wyjaśnia listing 4.16. W linii 3 następuje wywołanie metody, która jest zdefiniowana w klasie „*Mlot*”. Natomiast w linii 4 zostaje wywołana metoda wewnętrzna atrybutu „*Cecha*”, a jej implementacja jest zdefiniowana w klasie nadrzędnej „*MaszynaKuznicza*”.

Listing 4.15.

```
01 Public Class Mlot                        'klasa podrzędna
02   Public Cecha As MaszynaKuznicza        'dziedziczenie (związek typu „jest”)
03   Public EnergiaUderzenia As Double
04   Public Sub UruchomProces()             'nowa metoda, inna niż w nadklasie
05   End Sub
06   Private Function bijakWDol() As Boolean
07   End Function
08   Private Function bijakWGore() As Boolean
09   End Function
10 End Class
```

Listing 4.16.

```
01 Sub Main()
02    Private Mlot1 As New Mlot      'obiekt typu klasy zdef. w listingu 4.15
03    Mlot1.UruchomProces()          'metoda zdef. w klasie 'Mlot' (linia nr 4)
04    Mlot1.Cecha.UruchomProces()    'metoda zdef. W klasie 'MaszynaKuznicza'
05 End Sub
```

Polimorfizm

Polimorfizm (z ang. *polymorphism*) jest mechanizmem ściśle powiązanym z dziedziczeniem. Ze względu na swoje cechy, jest on jedną z najważniejszych zalet programowania zorientowanego obiektowo. Mechanizm ten polega na tym, że obiekt danej klasy może być traktowany tak, jakby tył obiektem klasy nadrzędnej. Dzięki temu istnieje możliwość bezpośredniego przypisania wartości z klasy potomnej do zmiennej będącej typem klasy nadrzędnej. W praktyce programistycznej, polimorfizm jest często wykorzystywany do jednolitego traktowania całych zbiorów obiektów (tzw. kolekcji).

Przykład obrazujący wykorzystanie polimorfizmu umieszczono na listingu 4.17. W liniach 1 oraz 2 umieszczono instrukcje powodujące utworzenie dwóch obiektów, będących instancjami różnych klas. Jednak zgodnie z treścią listingów 4.13 i 4.14 można stwierdzić, że utworzone obiekty mogą być uznane za obiekty jednej klasy nadrzędnej o nazwie „*MaszynaKuznicza*". Ta klasa nadrzędna jest jednocześnie typem zmiennej, którą zadeklarowano w linii 3. W takiej sytuacji, zgodnie z wyrażeniami umieszczonymi w liniach 5 i 6, konwersja obiektu do zmiennej jest możliwa, ponieważ jest to zgodne z mechanizmem polimorfizmu. Ale już próba konwersji zmiennej do obiektu, który jest typu klasy podrzędnej (linia 8) jest możliwa. Również nie można wykonać konwersji dwóch obiektów będących instancją różnych klas (linia 9), pomimo tego, że klasy te są potomkami jednej klasy nadrzędnej.

Listing 4.17.

```
01 Dim Prasa1   As New PrasaKorbowa       'utworzenie obiektu
02 Dim Mlot1    As New Mlot               'utworzenie obiektu
03 Dim Maszyna1 As MaszynaKuznicza        'deklaracja zmiennej
04 '--- próby przypisania wartości do zmiennej ----------------------
05   Maszyna1 = Prasa1     'Dobrze
06   Maszyna1 = Mlot1      'Dobrze
07 '--- nieudane próby konwersji (rzutowania) zmiennych i obiektów --
08   Prasa1 = Maszyna1     'Źle
09   Prasa1 = Mlot1        'Źle
```

Przeciążenie

Uzupełnieniem polimorfizmu jest mechanizm przeciążania metod i operatorów klas. Dzięki temu można tworzyć wiele metod o tej samej nazwie, które różnią się tylko parametrami. W listingu 4.18 zamieszczono przykład przeciążenia konstruktora klasy – tj. metody o nazwie „*New*", która zawsze jest wywoływana podczas tworzenia obiektu danej klasy. Pierwszy konstruktor nie przyjmuje żadnych parametrów, a jedynie inicjuje zmienną „*Nazwa*" wartością domyślną. Natomiast drugi, przeciążony konstruktor pobiera jeden łańcuch znaków jako parametr („*strNazwa*") i inicjuje nim wspomnianą zmienną. Przypadki wykorzystania odpowiednich konstruktorów pokazano w listingu 4.19.

Listing 4.18.

```
01 Public Class Prasa
02   Private Nazwa As String
03   Public Sub New()                          'konstruktor nr 1
04     Nazwa = "Prasa"
05   End Sub
06   Public Sub New(ByVal strNazwa As String)   'konstruktor nr 2 (przeciążony)
07     Nazwa = strNazwa
08   End Sub
09 End Class
```

Listing 4.19.

```
Dim Prasa1 As New Prasa()              ‘użycie konstruktora nr 1
Dim Prasa2 As New Prasa(„PXW-100”)     ‘użycie konstruktora nr 2
```

Innym sposobem wykorzystania mechanizmu przeciążenia jest nadanie nowego znaczenia operatorowi. Będzie on używany tylko w wyrażeniach, gdzie wyrazami są obiekty danej klasy. W celu zobrazowania tego mechanizmu zostanie zdefiniowana klasa o nazwie „*Complex*”, która reprezentuje liczby zespolone (listing 4.20). Linie 10÷12 zawierają instrukcję przeciążenia operatora – w tym przypadku operatora arytmetycznego sumy. Instrukcja ta nadaje nowe znaczenie dla tego operatora, a sposób wykonania obliczeń zdefiniowano w linii 11. Poszczególne wartości atrybutów obiektów „*c1*” i „*c2*” są dodawane do siebie, a wynik zostaje wstawiony do właściwego atrybutu nowo utworzonego obiektu, poprzez użycie konstruktora mającego zdefiniowane parametry. Przykład użycia przeciążonego operatora dodawania liczb zespolonych zamieszczono w kolejnym listingu 4.21.

Listing 4.20.

```
01 Public Class Complex      ‘klasa reprezentująca liczby zespolone
02   Public Re As Double     ‘część rzeczywista
03   Public Im As Double     ‘część urojona
04   Public Sub New()
05   End Sub
06   Public Sub New(ByVal iRe As Double, ByVal iIm As Double)
07     Re = iRe
08     Im = iIm
09   End Sub
10   Public Shared Operator + (c1 As Complex, c2 As Complex) As Complex
11     Return New Complex(c1.Re + c2.Re, c1.Im + c2.Im)
12   End Operator
13 End Class
```

Listing 4.21.

```
01 Dim Liczba1 As New Complex(1,5)
02 Dim Liczba2 As New Complex(2,0)
03 Dim Liczba3 As New Complex()
04 Liczba3 = Liczba1 + Liczba2       ‘użycie przeciążonego operatora sumy
```

5. ELEMENTY TEORII PROGRAMOWANIA

5.1. Struktury danych

5.1.1. Charakterystyka ogólna

Struktury danych stosowane w programowaniu komputerów są ważnym narzędziem programistycznym, które umożliwia uporządkowanie danych przechowywanych zarówno w pamięci komputera, jak i na dowolnym nośniku. Są one używane również podczas rozwiązywania skomplikowanych zadań algorytmicznych, ponieważ umiejętna organizacja danych zapewnia znacząco poprawę efektywności i wydajności ich przetwarzania [1, 17, 30, 33].

Organizacja danych w odpowiednie struktury pozwala osiągnąć następujące korzystne efekty [3, 13, 18, 31, 33]:

- możliwość łatwego tworzenia elastycznych i efektywnie działających baz danych, bez konieczności stosowania zaawansowanych, komercyjnych aplikacji specjalistycznych;
- efektywne rozwiązywanie złożonych zagadnień obliczeniowych i optymalizacyjnych;
- możliwość wykonywania analizy symbolicznej złożonych wyrażeń algebraicznych i logicznych;
- rozwiązywanie problemów z dziedziny sztucznej inteligencji;
- budowa algorytmów działających według modelu „sieci neuronowej".

Ze względu na postać, struktury danych można podzielić na struktury o budowie:

- prostej, np. tablice, rekordy;
- złożonej (zaawansowanej), np. listy, grafy, drzewa.

Struktury danych o budowie prostej mają postać zbioru zmiennych, który jest definiowany albo jako typ podstawowy, albo jako typ innej struktury. W ogólnym przypadku, taka postać organizacji danych jest nazywana **rekordem**, a poszczególne jego zmienne – **polami**.

Natomiast struktury danych o zaawansowanej budowie charakteryzują się tym, że dodatkowo zawierają one wewnętrzne funkcje (procedury), które służą do zarządzania zawartością struktury. Dane w takich strukturach są przechowywane w wewnętrznej strukturze prostej, którą nazywa się rekordem lub **pojem-**

nikiem (kontenerem, z ang. *container*), a poszczególne pola tego pojemnika są określane mianem elementu struktury. Natomiast wspomniane funkcje (procedury) są nazywane metodami dostępu, a typowymi ich operacjami są: dodawanie, usuwanie, wyszukiwanie i porządkowanie elementu (elementów).

Poszczególne typy struktur danych różnią się między sobą przede wszystkim organizacją zawartości struktury oraz sposobem dostępu do poszczególnych elementów. Wspólną ich cechą jest abstrakcja.

5.1.2. Proste struktury danych

Tablica

Tablica jest najprostszym pojemnikiem danych, w którym poszczególne elementy (komórki) są indeksowane unikatowym numerem. Każdy element przechowywany w tablicy musi być tego samego typu – może być to zarówno typ podstawowy, jak i również typ struktury prostej (złożonej). W języku Visual Basic tablica powstaje w wyniku deklaracji zmiennej przy użyciu nawiasów okrągłych, w których podaje się rozmiar tablicy (indeks ostatniej komórki). Jeżeli rozmiar tablicy jest niezmienny w czasie wykonywania programu, to tablica taka jest nazywana **statyczną**. Natomiast, jeżeli rozmiar ten może ulegać zmianie – to tablica jest nazywana **dynamiczną**. Obecnie większość języków programowania udostępnia jedynie tablice dynamiczne [26, 33].

Ze względu na wymiar, wyróżnia się tablice jednowymiarowe oraz wielowymiarowe (najczęściej dwu- lub trzywymiarowe). Tablica jednowymiarowa jest abstrakcją ciągu matematycznego, natomiast tablica dwuwymiarowa – macierzy. Tablice można użyć również do implementacji innych struktur danych, np. listy oraz drzewa. Przykłady deklaracji i użycia tablicy w języku Visual Basic .NET przedstawiono w listingu 5.1. Kombinacja słów kluczowych „*Redim*” i „*Preserve*” powoduje zmianę rozmiaru tablicy z jednoczesnym zachowaniem wcześniej wprowadzonych wartości do komórek.

Listing 5.1.

```
01 Dim Tablica1(10) As Integer    'tablica jednowymiarowa o 11 elementach (0÷10)
02 Redim Preserve Tablica1(20)    'zmiana rozmiaru tablicy w sposób dynamiczny
03 Dim Tablica2(2, 2) As Single   'tablica dwuwymiarowa (macierz o rozmiarze 3x3)
04 Tablica1(1) = 10   'wstawienie wartości do komórki o indeksie 1 (tj. drugiej)
```

W języku obiektowym VB .NET, istnieje również możliwość utworzenia tablicy jako obiektu typu klasy „*Array*”. Jednak taka postać tablicy jest strukturą bardziej złożoną, ponieważ klasa ta ma zaimplementowane metody służące do zarządzania danymi znajdującymi się w takiej strukturze. Wadą takiego podejścia (w porównaniu do tablic tradycyjnych) jest to, że tablica obiektowa jest

statyczna, zajmuje większą ilość miejsca w pamięci, a prędkość wykonywania obliczeń z użyciem takiej struktury jest zdecydowanie mniejsza [15, 18, 25].

Rekord

Rekord jest strukturą spostrzeganą jako abstrakcja zbioru różnych elementów, które mogą mieć różny rozmiar oraz mogą być odmiennego typu. Podstawowym, praktycznym zastosowaniem tej struktury w programowaniu jest użycie jej do zbudowania pliku lub bazy danych.

Rekord może również służyć do zgrupowania kilku zmiennych, które mają sens tylko w przypadku, gdy są rozpatrywane łącznie. W ten sposób zbudowana struktura reprezentuje abstrakcję wyższego rzędu – np. może posłużyć do utworzenia zmiennej przedstawiającej punkt w przyjętym układzie odniesienia (listing 5.2). W nomenklaturze programowania, czynność definiowania rekordu, jako struktury grupującej zmienne elementarne, nazywa się utworzeniem nowego typu użytkownika (z ang. *user-defined type of data*).

Przykład deklaracji rekordu, jako nowego typu użytkownika, za pomocą języka Visual Basic (wersja oznaczona numerem 6, VBA) przedstawiono w listingu 5.2. Poszczególne pola tej struktury są zadeklarowane w liniach 2÷4 jako lokalne zmienne typu podstawowego (jednakowy typ wszystkich pól jest przypadkowy). Natomiast kolejny listing 5.3 zawiera przykład utworzenia zmiennej typu użytkownika oraz sposób uzyskania dostępu do jej pól.

Listing 5.2.

```
01 Public Type Punkt        'Rekord / typ użytkownika o nazwie 'Punkt'
02   Dim R As Double        'współrzędna R w walcowym układzie współrzędnych
03   Dim Z As Double        'współrzędna Z
04   Dim Teta As Double     'współrzędna Teta (kąt)
05 End Typ
```

Listing 5.3.

```
01 Dim Punkt1 As Punkt          'deklaracja zmiennej typu użytkownika
02 Punkt1.R = 5                 'przypisanie wartości do pola o nazwie 'R'
03 Punkt1.Teta = 15.5           'przypisanie wartości do pola o nazwie 'Teta'
```

Kolejny listing 5.4 zawiera deklarację tego samego rekordu, który został omówiony powyżej, ale wykonaną za pomocą języka obiektowego Visual Basic .NET. W tym przypadku, zamiast słowa kluczowego „*Type*" używa się słowa kluczowego „*Structure*". Składnia tego języka umożliwia zadeklarowane procedury (funkcje) wewnątrz struktury, które są stosowane powszechnie jako konstruktory danego rekordu. W listingu 5.5 umieszczono sposób utworzenia zmiennej (linia 1 i 2), która jest zmienną typu użytkownika, oraz zmiennej (linia 3) traktowanej jako instancja rekordu (tj. obiekt). Należy pamiętać, że konstruktor danego rekordu jest wywoływany tylko podczas tworzenia obiektu.

Listing 5.4.

```
Public Structure PunktNET    'Rekord / stuktura o nazwie 'Punkt'
  Public R As Double         'współrzędna R w walcowym układzie współrzędnych
  Public Z As Double         'współrzędna Z
  Public Teta As Double      'współrzędna Teta (kąt)
  '--- definicja konstruktora rekordu ---
  Public Sub New(iR As Double, iZ As Double, iTeta As Double)
    R = iR          'zainicjowanie pola wartością pobraną za pomocą parametru
    Z = iZ          ' j.w.
    Teta = iTeta    ' j.w.
  End Sub
End Structure
```

Listing 5.5.

```
Dim Punkt3 As PunktNET                  'deklaracja zmiennej typu użytkownika
Dim Punkt(10) As PunktNET               'deklaracja tablicy rekordów
Dim Punkt4 As New PunktNET(5, 0, 12.5)  'deklaracja obiektu z użyciem konstr.
```

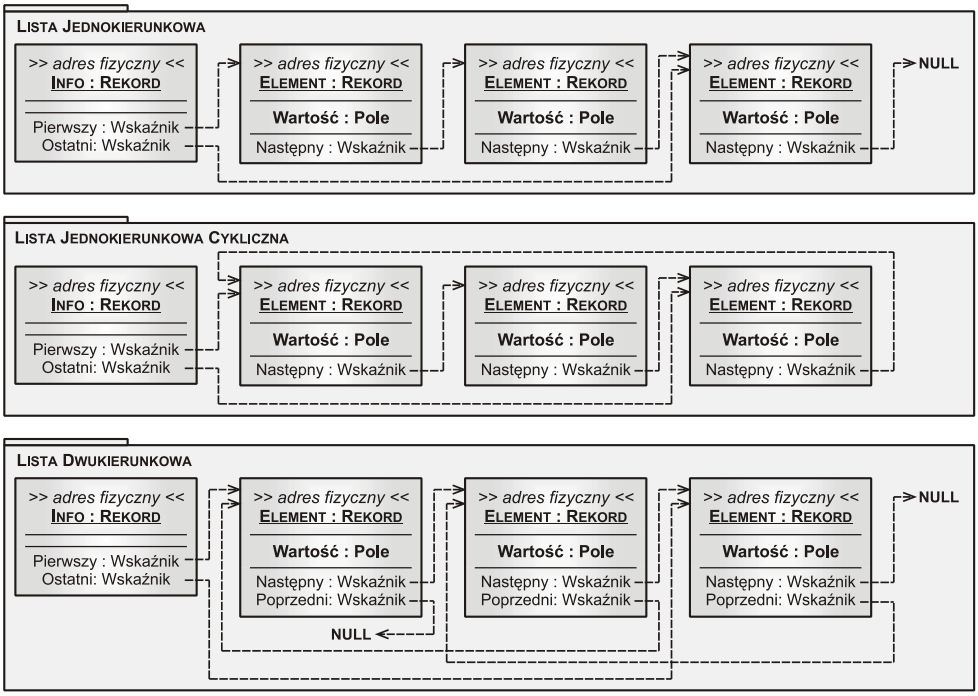
5.1.3. Złożone struktury danych

Lista

Lista jest dynamicznym pojemnikiem składającym się z ciągu połączonych ze sobą elementów, najczęściej tego samego typu oraz metod dostępu do jej danych. Elementem listy jest rekord, który posiada co najmniej dwa pola. Pierwsze pole przechowuje dane właściwe, natomiast drugie pole jest zmienną wskazującą na kolejny element listy. Takie rozwiązanie powoduje, że dostęp do poszczególnych danych w liście jest sekwencyjny, co jest zgodne z istotą funkcjonowania listy [1, 5, 33].

Typowa lista jest implementowana w sposób tzw. **dołączeniowy** (wskaźnikowy). Taka metoda budowy listy wymaga użycia zmiennej wskaźnikowej, która umożliwia bezpośredni dostęp do pamięci komputera, gdyż wartość tej zmiennej jest fizycznym adresem danego rekordu. Niestety, tylko nieliczne języki programowania posiadają taki typ zmiennej, np. C++. Obejściem tej niedogodności jest zaimplementowanie listy w sposób tzw. **tablicowy** (indeksowy) – w tym celu tworzona jest tablica rekordów, które zawierają tylko pola danych. Taka implementacja, w porównaniu z implementacją dołączeniową, jest odporniejsza na błędy, a nawigacja wewnątrz listy jest łatwiejsza (dzięki bezpośredniemu dostępowi do poszczególnych elementów listy). Natomiast wadą implementacji tablicowej jest niska elastyczność listy oraz minimalizacja jej dynamicznego charakteru [1, 33].

Na rys. 5.1 przedstawiono schematycznie, za pomocą diagramu graficznego, strukturę rekordów tworzących listę dołączeniową. Do budowy takiej struktury danych używane są dwa rodzaje rekordów, mianowicie [33]:



Rys. 5.1. Schematyczne przedstawienie budowy trzech różnych rodzajów list dołączeniowych, gdzie: linia przerywana (-➢) reprezentuje wskazanie zmiennej wskaźnikowej na adres fizyczny danego rekordu; NULL – specjalna wartość zmiennej wskaźnikowej, która informuje o tym, że zmienna nie wskazuje na żaden adres fizyczny pamięci komputera; opis w tekście

- rekord natury informacyjnej (o nazwie „*Info*") pełniący rolę „strażnika" listy; zawiera on tylko dwie zmienne wskaźnikowe, które wskazują na początkowy i końcowy element listy;
- rekord o charakterze roboczym (nazwany jako „*Element*"), który oprócz zmiennych wskaźnikowych (wskazanie na kolejny i/lub poprzedzający element listy) zawiera również pole „*wartość*", które w tym przypadku reprezentuje w sposób umowny przechowywane dane rekordu.

Podstawową klasyfikacją list dołączeniowych jest ich podział ze względu na sposób połączenia ze sobą rekordów. Zgodnie z rys. 5.1, wyróżnia się następujące listy [33]:

- jednokierunkowe – rekordy robocze zawierają tylko jedną zmienną wskaźnikową, która przechowuje adres fizyczny następnego elementu w liście; je-

żeli element listy jest rekordem ostatnim, wskaźnik „*następny*” przechowuje wartość „ NULL”;
- dwukierunkowe – rekordy zawierają dwie zmienne wskaźnikowe, które wskazują na rekord poprzedzający oraz rekord następny;
- cykliczne – są to listy jedno- lub dwukierunkowe, gdzie zmienna wskaźnikowa o nazwie „*następny*”, należąca do ostatniego rekordu w liście, wskazuje na rekord znajdujący się na początku listy; w przypadku listy dwukierunkowej cyklicznej, dodatkowo wskaźnik „*poprzedni*” w rekordzie pierwszym wskazuje na rekord ostatni.

Inną spotykaną klasyfikacją omawianych struktur danych jest podział list ze względu na ich przeznaczenie (specjalne zastosowanie). Jest ono zdeterminowane przez zaimplementowane metody dostępu do danych, które realizują określone czynności bezpośrednio dotyczące rekordów. Tymi czynnościami są [1, 33]:
- pobieranie początkowego lub końcowego elementu z listy,
- wstawianie nowego elementu na początek lub koniec listy,
- usunięcie pierwszego lub ostatniego elementu z listy.

Zatem, nawiązując do wyżej wspomnianej klasyfikacji, można wyróżnić dwie listy o specjalnym zastosowaniu, które są najczęściej używane w praktyce. Tymi strukturami danych są [33]:
- **stos** (z ang. *stack*), określany również skrótem LIFO (z ang. *Last-In, First-Out*), gdzie dostęp do listy jest możliwy tylko z jednej strony – oznacza to, że nowy element jest wstawiany zawsze na tzw. wierzchołek (początek) listy, z którego również jest pobierany tzw. rekord dostępny;
- **kolejka** (z ang. *queue*), nazywana w skrócie FIFO (z ang. *First-In, First-Out*), w której nowy element jest wstawiany na początek listy, a dostępny rekord jest zawsze pobierany z jej końca.

Stos i kolejka są listami, w których są stosowane tylko dwie zasadnicze metody. W przypadku stosu są to dwie operacje, które realizują tzw. „*wrzucanie*” (z ang. *push*) nowego elementu na początek stosu oraz tzw. „*wyrzucanie*” (z ang. *pop*) elementu, który aktualnie znajduje się na wierzchołku listy. Natomiast w przypadku kolejki, są używane operacje wykonujące tzw.„*wstawianie*” (z ang. *enqueue*) elementu na początek kolejki oraz „*obsłużenie*” (z ang. *dequeue*) elementu znajdującego się na końcu kolejki. Należy pamiętać, że operacjom „*wyrzucanie*” i „*obsłużenie*” rekordu towarzyszy zawsze czynność jego usuwania z danej listy. Struktury te mają szczególne znaczenie w efektywnym przetwarzaniu danych oraz są stosowane w większości znanych algorytmów.

Najprostszym sposobem utworzenia listy jest zbudowanie jej jako implementacji tablicowej. Charakteryzuje się ona tym, że rekord o nazwie „*Element*” nie zawiera zmiennej wskaźnikowej, a jego adres fizyczny (za pomocą którego w implementacji dołączeniowej uzyskuje się dostęp do elementu) nie musi być

znany. Rekord roboczy jest częścią tablicy i ma przypisany indeks porządkowy, natomiast rekord o nazwie „*Info*" jest zastępowany zazwyczaj zmienną liczbową, która przechowuje informację o aktualnym rozmiarze listy.

Pisanie programu komputerowego zorientowanego obiektowo wymaga, aby lista została utworzona w oparciu o odpowiednią klasę. W większości przypadków, języki programowania posiadają już zdefiniowane takie klasy, np. Visual Basic .NET posiada klasę o nazwie „*ArrayList*". Jedyną wadą tzw. „gotowych" wzorców jest ich ogólność, która w przypadku programowania zaawansowanego niejednokrotnie zmniejsza efektywność działania tak utworzonej listy. Zatem w przypadkach szczególnych zaleca się definiować własne klasy.

Przykład budowy własnej klasy (listy „*ListaPunkt*"), przy użyciu języka Visual Basic .NET, umieszczono w listingu 5.6. Kod ten przedstawia jedynie szkielet listy o implementacji tablicowej, która ma za zadanie przechowywać punkty (ściślej: jego współrzędne, zgodnie z definicją rekordu w listingu 5.4).

Listing 5.6.

```
Public Class ListaPunkt
  Private Pojemnik() As PunktNET      'definicja tablicy rekordów roboczych
  Private iLP As Integer              'tzw. strażnik (ilość elementów)
  Public Sub New()                    'konstruktor klasy (listy)
    iLP = 0                           'brak elementów w liście
    ReDim Pojemnik(iLp)               'lista zawiera tylko rekord „zerowy"
  End Sub
  Public Sub Dodaj(ByRef pkt As PunktNET)
    iLP = iLP + 1
    ReDim Preserve Pojemnik(iLP)      'dynamiczna zmiana rozmiaru tablicy
    Pojemnik(iLP) = pkt               'wstawienie nowego elementu do listy
  End Sub
  Public Function Pobierz(ByRef pkt As PunktNET) As Integer
    If iLp > 0 Then                    'obsługa ewentualnego wyjątku
      pkt = Pojemnik (iLP)             'pobranie ostatniego elementu z listy
      iLP = iLP - 1
      ReDim Preserve Pojemnik (iLP)    'usuwanie ostatniego elementu z listy
      Return 0                         'operacja wykonana poprawnie
    Else
      Return -1                        'wyjątek (błąd): operacja nieudana
    End If
  End Function
End Class
```

Struktura przykładowej klasy składa się z lokalnej tablicy rekordów (elementów listy) o nazwie „*Pojemnik*", lokalnej zmiennej liczbowej o nazwie „*iLp*" (zwaną również strażnikiem listy) oraz dwóch metod (interfejsu klasy), które realizują operację dodawania nowego elementu na koniec listy oraz pobierania ostatniego elementu z listy (zgodnie ze strukturą LIFO). Rozmiar tablicy jest zmieniany w sposób dynamiczny (za pomocą instrukcji „*ReDim*"), gdzie aktualny jej rozmiar jest przechowywany w zmiennej liczbowej „*iLp*".

Utworzenie dowolnego obiektu typu „*ListaPunkt*” (listing 5.7, linia 3) spowoduje, że wartość zmiennej lokalnej „*iLP*” zostanie wyzerowana oraz zostanie dostosowany rozmiar tablicy „*Pojemnik*” (zadanie konstruktora). Początkowo, tablica ta będzie posiadała tylko jedną komórkę o indeksie „0”, którą nazywa się **elementem zerowym** (będzie on zawsze pusty). Dodatkowy listing 5.7 przedstawia przykład utworzenia instancji listy oraz sposoby jej użycia. Proszę zwrócić uwagę na linię 6 oraz 9, w której wykonuje się operację pobrania elementu, który jest ostatni na liście (w tym przypadku – z wierzchołka). Przykład ten implementuje działanie struktury o nazwie „stos”.

Listing 5.7.

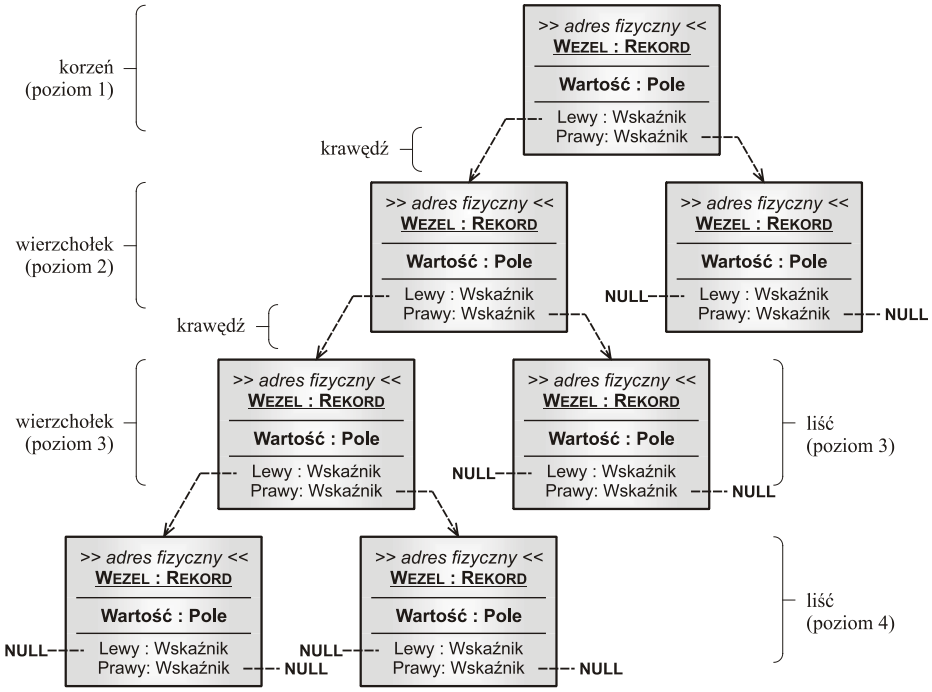
```
01 Dim punkt1 As New PunktNET(5, 0, 12.5)  'deklaracja obiektu pierwszego
02 Dim punkt2 As New PunktNET(0, 0, 0)     'deklaracja obiektu drugiego
03 Dim Lista As New ListaPunkt             'deklaracja listy (jako obiekt)
04 Lista.Dodaj(punkt1)                    'operacja dodania elementu do listy
05 Lista.Dodaj(New PunktNET(10, 2, 6))    'j.w., sposób drugi
06 Lista.Pobierz(punkt2)   'operacja pobrania elementu z listy i „wstawienie” go
07                         'do obiektu „punkt2”. Po tej operacji pola tego
08                         'obiektu mają wartości równe odpowiednio {10, 2, 6}
09 Lista.Pobierz(punkt2)   'j.w. z tym, że teraz wartość pól wynosi {5, 0, 12.5}
```

Drzewo

Struktura danych o nazwie „drzewo” (z ang. *tree*) jest abstrakcją matematycznego spójnego grafu niezorientowanego, pozbawionego cykli [5]. Cechą charakterystyczną tej struktury jest to, że w naturalny sposób reprezentuje hierarchię danych. Znaczenie drzewa w programowaniu komputerów jest bardzo duże, a ze względu na swoje własności jest ono stosowane praktycznie w każdej implementacji komputerowej, szczególnie w bazach danych, grafice komputerowej, przetwarzaniu tekstu, modelowaniu numerycznym itd. Drzewa ułatwiają oraz przyśpieszają operacje wykonywane na uporządkowanych danych, zwłaszcza takie jak wyszukiwanie lub sortowanie danych [1, 33].

Przykładową architekturę drzewa przedstawiono schematycznie na rys. 5.2. Strukturę tą, podobnie jak w przypadku listy, można zaimplementować w sposób dołączeniowy (wskaźnikowy) lub tablicowy. Jednak sposób implementacji drzewa nie ma większego znaczenia na efektywność wykonywanych operacji na jego elementach (rekordach).

Omawiana struktura danych składa się z **wierzchołków** (węzłów) oraz łączących je **krawędzi**. Węzłem jest rekord, którego budowa jest podobna do rekordu listy, natomiast rolę krawędzi drzewa pełni zmienna wskaźnikowa, która przechowuje adres fizyczny do następnego węzła. W implementacji tablicowej, abstrakcją krawędzi jest procedura (funkcja) określająca indeks komórki, w której jest przechowywany następny węzeł drzewa.



Rys. 5.2. Schematyczne przedstawienie budowy drzewa w formie implementacji dołączeniowej, gdzie: linia przerywana (−≫) reprezentuje krawędzie drzewa, które wskazują na adres fizyczny rekordu (danego węzła drzewa); opis w tekście

W nomenklaturze programowania komputerów przyjęto, że wszystkie wierzchołki połączone z jednym węzłem, a leżące na następnym poziomie (względem poziomu rozpatrywanego węzła) są nazywane **następnikami** (potomkami) tego węzła. Każdy wierzchołek drzewa ma tylko jednego **poprzednika** (rodzica), przy czym wyjątkiem jest węzeł na poziomie pierwszym, który jest nazywany **korzeniem**. W ogólnym przypadku wierzchołek może mieć dowolną liczbę następników, ale jeśli nie ma ich wcale, to węzeł taki jest nazywany **liściem** drzewa. Szczególnym przypadkiem omawianej struktury danych jest **drzewo binarne** (z ang. *binary tree*), w którym liczba następników jest ograniczona tylko do dwóch. Takie drzewo ma specjalne znaczenie dla informatyki.

Kolejnym ważnym pojęciem, tym razem związanym z operacjami wykonywanymi na drzewach, jest tzw. **przechodzenie** drzewa. Jest to proces „odwie-

dzania” kolejnych wierzchołków z zachowaniem zależności pomiędzy poprzednikiem i następnikiem. Ze względu na sposób przechodzenia drzewa, podczas którego są wykonywane dodatkowe działania na jego wierzchołkach, wyróżnia się przechodzenie:

- **preorder**, gdy działanie jest wykonywane najpierw na poprzedniku, a następnie na następniku;
- **postorder**, kiedy operacje są wykonywane najpierw na wszystkich następnikach, a dopiero na końcu na poprzedniku;
- **inorger**, gdy w drzewie binarnym operacja jest wykonywana na jednym następniku, następnie na poprzedniku i dopiero na koniec na drugim następniku.

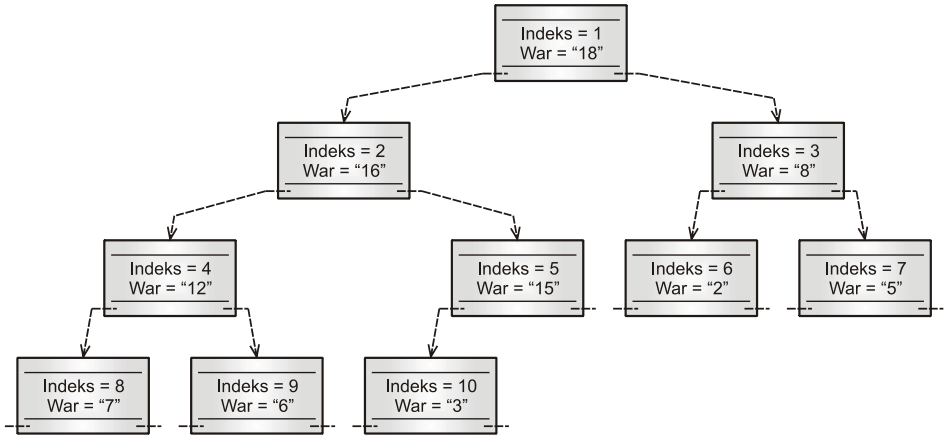
Podstawowymi operacjami, które wykonuje się z udziałem drzew, są:

- wyliczenie wszystkich elementów drzewa,
- wyszukanie elementu spełniającego określone kryterium,
- dodanie nowego elementu w określone miejsce drzewa,
- usunięcie określonego elementu z drzewa.

Drzewa odgrywają ważną rolę w budowaniu efektywnych algorytmów. Ze względu na zastosowanie drzew w programowaniu komputerów, można wyróżnić następujące główne odmiany drzew:

- kopiec,
- drzewo typu BST,
- drzewo typu AVL.

Struktura danych o nazwie „**kopiec**” (sterta, z ang. *heap*) jest drzewem zupełnym, najczęściej binarnym. Jego liście występują na ostatnim, ewentualnie przedostatnim poziomie i są spójne, ułożone od strony lewej do prawej. Cechą charakterystyczną kopca jest to, że wartość zmiennej kluczowej (klucza) każdego węzła jest nie mniejsza od zmiennej przechowywanej przez jego następników. Budowę drzewa oraz sposób uporządkowania jego elementów przedstawiono schematycznie na rys. 5.3. Rekord zawiera dwie zmienne (wybrano je w sposób przykładowy). Zmienna „*War*” jest traktowana jako „klucz” struktury, natomiast zmienna o nazwie „*Indeks*” przechowuje numer porządkowy elementu kopca. Można zauważyć, że rekordy są uszeregowane liniowo, co sugeruje sposób ich składowania w implementacji tablicowej. Korzeń kopca będzie przechowywany w komórce o numerze „1”, lewy potomek każdego i-tego wierzchołka będzie składowany w komórce o indeksie równym $2 \cdot i$, natomiast prawy potomek będzie ulokowany w komórce o numerze wynoszącym $2 \cdot i+1$. Cechy tej struktury danych powodują, że jest ona stosowana do implementacji zbiorów danych o ustalonej wzajemnej relacji oraz w algorytmach sortowania.

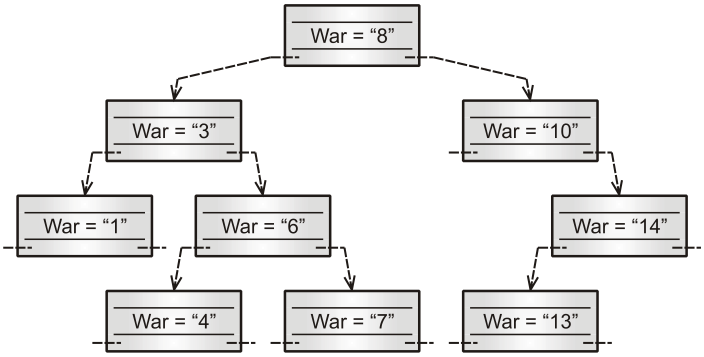


Rys. 5.3. Schematyczne przedstawienie budowy drzewa typu „kopiec”, gdzie zmienna „*Indeks*” zawiera numer porządkowy rekordów, a zmienna „*War*” jest „kluczem”, według której są szeregowane rekordy

Binarne drzewo przeszukiwań, w skrócie **drzewo BST** (z ang. *Binary Search Tree*), jest tzw. „samoorganizującą się” strukturą danych, spełniającą określony warunek relacji pomiędzy swoimi rekordami. Wartość zmiennej kluczowej danego wierzchołka jest nie mniejsza od odpowiedniej zmiennej wszystkich węzłów potomnych znajdujących się po jego lewej stronie. Jednocześnie wartość ta jest nie większa od zmiennych kluczowych wszystkich węzłów potomnych, znajdujących się z prawej strony tego wierzchołka. Przykład takiej struktury przedstawiono schematycznie na kolejnym rys. 5.4. Drzewo BST jest wykorzystywane głównie w algorytmach przeszukiwania. Odmianą omawianej struktury jest **drzewo AVL**, w którym różnica pomiędzy ilością poziomów lewego i prawego poddrzewa każdego wierzchołka jest nie większa niż jeden. Zrównoważenie drzewa gwarantuje, że pesymistyczny czas wyszukiwania określonego rekordu jest możliwie najmniejszy.

Graf

Pod pojęciem „graf” należy rozumieć strukturę danych, która jest abstrakcją matematycznego obiektu o tej samej nazwie [5]. Ze względu na własności grafu, struktura ta odgrywa bardzo ważną rolę w procesie programowania komputerów. Wszystkie najbardziej efektywne algorytmy, szczególnie rozwiązujące



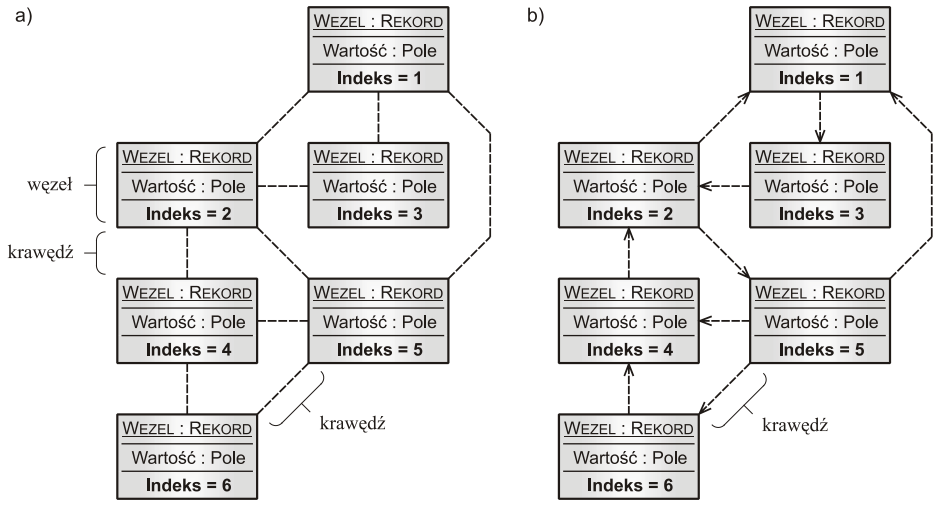
Rys. 5.4. Schematyczne przedstawienie budowy drzewa BST, gdzie zmienna „*War*” jest „kluczem”, według której są uporządkowane rekordy

problemy złożone, bazują na tej strukturze. Grafy są również nierozłącznym elementem sztucznej inteligencji. Zarówno systemy eksperckie, jak i sieci neuronowe, są budowane w oparciu o grafy oraz algorytmy wykorzystujące ich własności. Tematyka grafów oraz algorytmów grafowych jest bardzo obszerna. Celem tego podrozdziału jest jedynie przybliżenie elementarnej wiedzy – pełną informację o grafach zawiera literatura specjalistyczna, np. [1, 5, 33].

Z punktu widzenia informatyki, struktura ta składa się z pary dwóch skończonych zbiorów, które implementuje się za pomocą tablicy, listy lub innej struktury danych. Pierwszy wspomniany zbiór zawiera rekordy z danymi, które reprezentują abstrakcyjnie węzły w grafie. Natomiast drugim zbiorem jest zespół elementów tworzących krawędzie grafu, czyli pary niektórych węzłów. Na rys. 5.5 przedstawiono przykład grafu niezorientowanego oraz zorientowanego (krawędzie są ukierunkowane). Węzły tego grafu są reprezentowane przez rekordy – dla uproszczenia opisu, ograniczono się do dwóch pól reprezentujących dane właściwe oraz zmienną „*Indeks*”, która identyfikuje dany węzeł.

W praktyce, węzły są przechowywane w liście, natomiast krawędzie grafu są implementowane na jeden z dwóch sposobów. Do tego celu używa się:

- listy sąsiedztwa, która jest strukturą dynamiczną; uzyskuje się ją poprzez zdefiniowanie tablicy rekordów, gdzie indeks tej tablicy odpowiada indeksowi rozpatrywanego węzła w grafie; polem tego rekordu jest kolejna tablica zmiennych, tym razem przechowująca numery węzłów sąsiednich (tj. tworzących parę z rozpatrywanym węzłem); w przypadku grafu zoriento-



Rys. 5.5. Przykład przedstawiający graf niezorientowany (a) oraz zorientowany (b); gdzie węzłami są rekordy tworzące zbiór skończony, natomiast krawędziami są pary węzłów tworzące osobny zbiór; opis w tekście

wanego, w liście tej podaje się tylko numery tych węzłów sąsiednich, do których jest skierowana krawędź;

- macierzy sąsiedztwa, która jest utworzona przez tablicę dwuwymiarową zmiennych liczbowych lub boolean'owskich; jej indeksy reprezentują numery węzłów, natomiast jej komórki przechowują informację na temat istnienia krawędzi pomiędzy danymi węzłami (zazwyczaj wartość jeden lub „*True*"), w przeciwnym razie wartością komórki jest zero lub „*False*".

Poniżej umieszczone dwie tabele 5.1 oraz 5.2 przedstawiają schematycznie postać listy sąsiedztwa dla grafu przedstawionego na rys. 5.5. W przypadku grafu zorientowanego, lista sąsiedztwa zawiera tylko te indeksy węzłów, które zgodnie z orientacją krawędzi są węzłami końcowymi (tj. wskazywanymi przez tą krawędź).

Natomiast kolejne dwie tabele 5.3 oraz 5.4 przedstawiają macierzową postać zbioru krawędzi grafu znajdującego się na rys. 5.5. Przyjmuje się, że wiersze macierzy oznaczają węzły początkowe krawędzi, natomiast kolumny – węzły końcowe.

Tabela 5.1.
Lista sąsiedztwa dla grafu z rys. 5.5a

Indeks węzła	Lista indeksów węzłów sąsiednich
1	2, 3, 5
2	1, 3, 4, 5
3	1, 2
4	2, 5, 6
5	1, 2, 4, 6
6	4, 5

Tabela 5.2.
Lista sąsiedztwa dla grafu z rys. 5.5b

Indeks węzła	Lista indeksów węzłów sąsiednich
1	3
2	1, 5
3	2
4	2
5	1, 4, 6
6	4

Tabela 5.3.
Macierz sąsiedztwa dla grafu z rys. 5.5a

	1	2	3	4	5	6
1	0	1	1	0	1	0
2	1	0	1	1	1	0
3	1	1	0	0	0	0
4	0	1	0	0	1	0
5	1	1	0	1	0	1
6	0	0	0	1	1	0

Tabela 5.4.
Macierz sąsiedztwa dla grafu z rys. 5.5b

	1	2	3	4	5	6
1	0	0	1	0	0	0
2	1	0	0	0	1	0
3	0	1	0	0	0	0
4	0	1	0	0	0	0
5	1	0	0	1	0	1
6	0	0	0	1	0	0

Na zakończenie omawiania tej struktury danych, warto wspomnieć, że najważniejszą operacją (algorytmem) wykonywaną z udziałem grafu jest tzw. **przechodzenie**. Jego celem jest synteza informacji zawartych w tej strukturze danych, a efektem – rozwiązanie zadania, najczęściej o charakterze optymalizacyjnym. Klasycznymi problemami, które rozwiązuje się z użyciem grafów, są [1, 5, 33]:

- wyznaczenie cyklu Eulera;
- określenie najkrótszej ścieżki;
- zbudowanie minimalnego drzewa rozpinającego;
- tzw. „kolorowanie grafu” - rozwiązanie zadania optymalnego doboru, które polega na minimalizowaniu konfliktów;

5.2. Techniki programowania

5.2.1. Wprowadzenie

Walory użytkowe programu komputerowego zależą głównie od zastosowanej abstrakcji opisu danych oraz szybkości ich przetwarzania. Zależą one również, w sposób pośredni, od zastosowanego paradygmatu programowania i języka

jego zapisu. Szybkość przetwarzania danych oraz rozwiązywania zadań obliczeniowych jest zdeterminowana przez użytą do tego celu implementację algorytmów.

Poniżej zostaną omówione najważniejsze metody układania algorytmów (procedur obliczeniowych). Ponieważ zadanie to ma charakter twórczy, przedstawione metody (techniki programowania) są ogólne i w pewnych sytuacjach można je stosować jako wzorce o dużej elastyczności.

Należy również podać, że w każdej technice programowania – zarówno podczas układania algorytmu, jaki i projektowania programu jako całości zadania – obowiązuje jedna podstawowa zasada. Można ją wyrazić następującym hasłem: „*od ogółu, do szczegółu*”. Zatem, pierwszym etapem programowania zawsze jest utworzenie „szkieletu” algorytmu (programu), a dopiero w kolejnych fazach, sukcesywnie uszczegóławia się poszczególne jego fragmenty funkcyjne.

5.2.2. Metoda typu „*dziel i zwyciężaj*”

Programowanie metodą typu „*dziel i zwyciężaj*” (z ang. *divide and conquer*) wywodzi się z znanej idei, propagowanej przez wielu wielkich strategów wojennych (np. Napoleona Bonaparte, Juliusza Cezara), którą wyraża się zdaniem „*Divide ut Regnes*”. W przypadku informatyki, technika ta sprowadza się do wykorzystania procesu dekompozycji złożonego problemu na pewną, skończoną ilość podproblemów tego samego typu. Następnie, otrzymane cząstkowe rozwiązania tych podproblemów są łączone w określony sposób, a efektem tego jest uzyskanie rozwiązania globalnego (ostatecznego). Zatem można stwierdzić, że każdy trudny problem można w łatwy i szybki sposób rozwiązać, jeżeli zostanie on podzielony na mniejsze problemy, *de facto* łatwiejsze do rozwiązania.

Formalny zapis tej metody za pomocą pseudojęzyka programowania zamieszczono w listingu 5.8. Przedstawia on pewną funkcję o nazwie „*DiZ*” zwracającą wynik rozwiązania, a jej parametrem jest pewna zmienna reprezentująca rząd problemu. Z reguły jest to rozmiar (czyli: zmienna liczbowa) danych, które są przetwarzane w ramach rozwiązywania problemu algorytmicznego.

Listing 5.8.

```
Funkcja DiZ (parametr N jako rząd problemu) zwracająca wynik w
{
  Jeśli N jest wystarczająco małe to
       zwróć wynik przypadku elementarnego w = w_elementarny
  w przeciwnym przypadku
        podziel problem na k mniejszych podproblemów rzędu N_1 .. N_k
        Dla i z przedziału od 1 do k wykonuj
              oblicz wynik cząstkowy o wartości w_i = DiZ(N_i)
        zwróć wynik z połączenia wyników cząstkowych w = f(w_1 .. w_k)
}
```

W ogólnym przypadku, metoda typu „*dziel i zwyciężaj*” pozwala na zmianę klasy algorytmu np. z liniowego na logarytmiczny. Oznacza to, że efektem wymiernym budowy algorytmów tą techniką jest zwiększenie ich szybkości działania. Załóżmy następujący problem: należy odszukać określony rekord znajdujący się w liście jednokierunkowej o rozmiarze 10 elementów. Zastosowanie algorytmu przeszukiwania sekwencyjnego (klasa złożoności liniowa) spowoduje, że w pesymistycznym przypadku czas przeszukiwania będzie wynosił np. 10 sekund. Natomiast, użycie algorytmu utworzonego w oparciu o omawianą technikę programowania spowoduje, że ten czas wyniesie tylko 3,3 sekundy. Ale już dla listy, w której jest np. 1000 elementów, czas ten zostanie zmniejszony aż stukrotnie.

Rekursja (rekurencja)

Rekursja (rekurencja, z ang. *recursion*) jest mechanizmem ściśle związanym z programowaniem typu „*dziel i zwyciężaj*”. W ogólnym rozumieniu, podejście rekurencyjne do rozwiązania problemu sprowadza się w zasadzie do jego podziału na elementarne zadania. Trywialnym przykładem obrazującym istotę tego mechanizmu może być zadanie w postaci polecenia o treści „*wywierć otwory w płycie*”. Rekurencyjne rozwiązanie tego zadania będzie wyglądało w ten sposób, że ślusarz wywierci każdy otwór z osobna, jeden za drugim.

Rekurencja w informatyce jest realizowana poprzez sposób definiowania procedur (funkcji), polegający na umieszczeniu w treści tej procedury (funkcji) odwołań do samej siebie. Przykład takiego odwołania zawiera linia 8 listingu 5.8. Klasycznym algorytmem stosującym ten mechanizm jest funkcja obliczająca silnię z podanej liczby *X*. Algorytm taki, zapisany w języku Visual Basic .NET, zamieszczono w poniższym listingu 5.9. Jednocześnie jest to przykład typu **rekursji (rekurencji) naturalnej**.

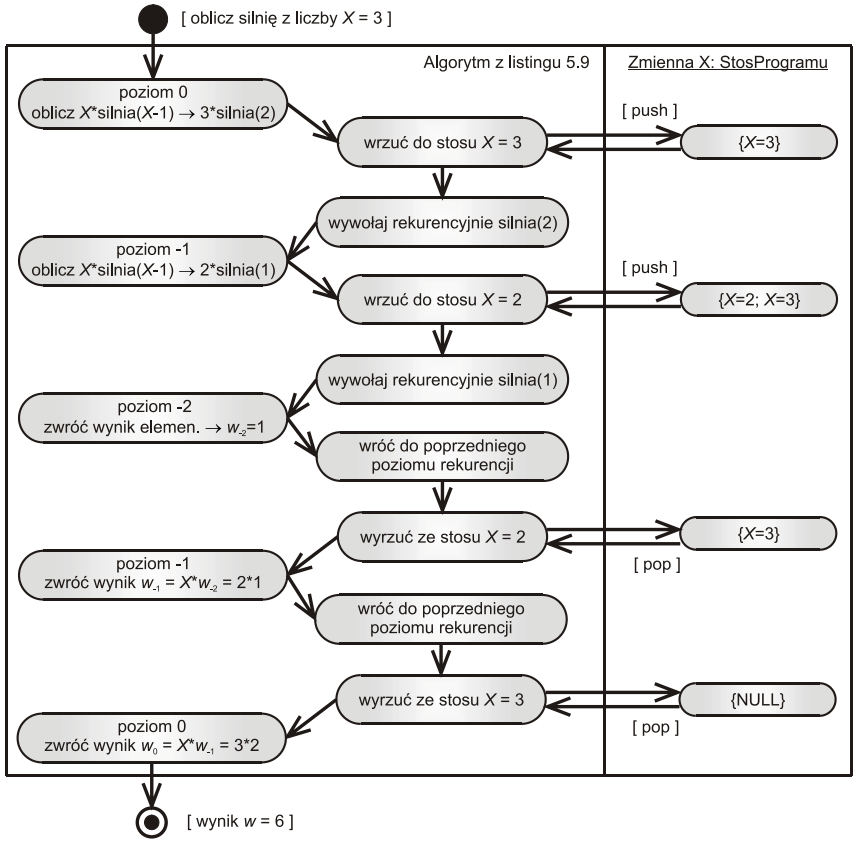
Listing 5.9.

```
01 Function Silnia(ByVal X As Double) As Double
02   If X = 0 Then
03     Return 1                 'wynik elementarny
04   Else
05     Return X * Silnia(X-1)   'wywołanie rekurencyjne z jednoczesnym łączeniem
06                              'wyników cząstkowych
07   End If
08 End Function
```

Główną zaletą stosowania rekursji do budowy procedur (funkcji) jest uzyskiwanie prostej i czytelnej składni zapisu algorytmu. Niestety, cechą charakterystyczną takich procedur jest duże zapotrzebowanie na zasoby pamięciowe. Z każdym wywołaniem rekursyjnym procedury wiążą się dwie zasadnicze czynności, które wykorzystują stos przydzielony programowi przez system operacyj-

ny. Struktura ta zapamiętuje pewne informacje niezbędne do odtworzenia stanu sprzed wywołania procedury oraz pośredniczy w przekazywaniu parametrów. Zatem, powstaje pewne ograniczenie algorytmów rekursyjnych związane z możliwym np. tzw. przepełnieniem stosu lub nawet zawieszenia działania programu.

Na rys. 5.6 przedstawiono diagram, który schematycznie obrazuje wykonywanie poszczególnych czynności podczas działania algorytmu z listingu 5.9. W tym przypadku, stos programu jest używany jako pojemnik do przechowywania kolejnych wartości zmiennej funkcji. Dla uproszczenia, w trakcie omawiania algorytmu pomija się pozostałe dane przechowane w takiej liście, a które są



Rys. 5.6. Diagram czynności, który obrazuje działanie algorytmu z listingu 5.9, na przykładzie obliczania wartości 3!; opis w tekście

niezbędne do poprawnego przechodzenia pomiędzy poziomami wywołań rekurencyjnych. Po osiągnięciu najniższego poziomu, w którym jest wykonywana operacja elementarna (linia 3 w listingu 5.9), uzyskuje się pierwszy wynik cząstkowy (elementarny). W trakcie powrotu do poziomu zerowego, algorytm ponownie wykorzystuje stos programu. Tym razem pobierane są wartości zmiennej lokalnej ze stosu (co wymaga wykonania dodatkowych, czasochłonnych operacji na tej strukturze danych – na rys. 5.6 jest to reprezentowane przez operację „*pop*”), a następnie wykonywany jest proces łączenia wyników cząstkowych.

Sposobem odciążenia stosu oraz skrócenia czasu działania algorytmu jest zbudowanie procedury (funkcji) z zastosowaniem tzw. **parametru dodatkowego**. Służy on do przechowywania połączenia wyników cząstkowych, a czynność ich łączenia jest wykonywana z każdym kolejnym wywołaniem rekursyjnym procedury (funkcji). W kolejnym listingu 5.10 umieszczono zmodyfikowaną funkcję „*Silnia*”, która zawiera wspomniany parametr dodatkowy.

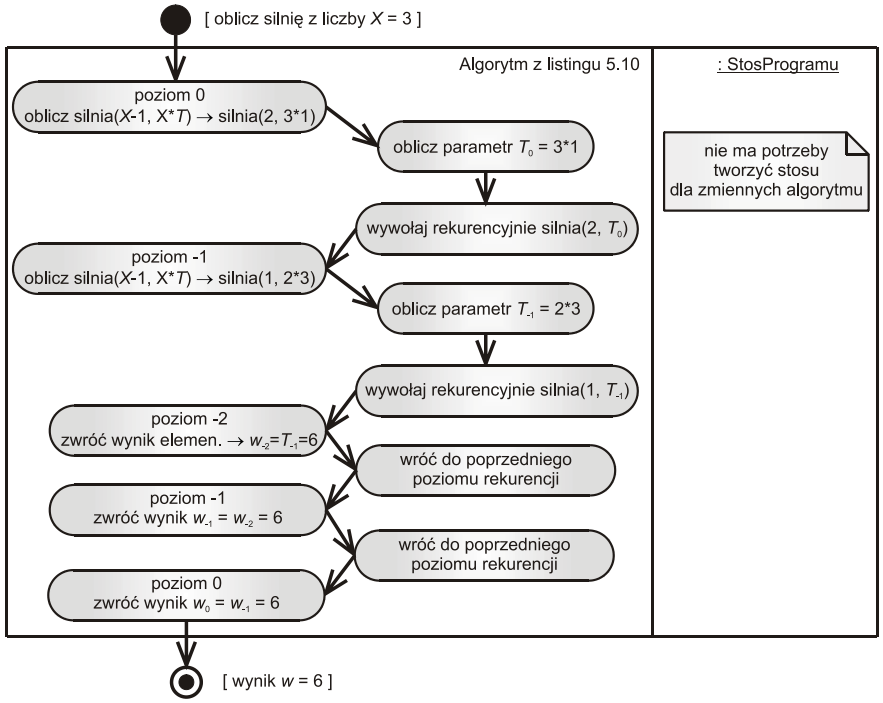
Listing 5.10.

```
01 Function Silnia(ByVal X As Double, ByVal T As Double = 1) As Double
02   If X = 0 Then
03     Return T                  'wynik, który jest rozwiązaniem zadania
04   Else
05     Return Silnia(X-1, X*T)   'wywołanie rekurencyjne z jednoczesnym łączeniem
06                               'wyników cząstkowych, ale za pośrednictwem „T”
07   End If
08 End Function
```

Przykład w formie diagramu, który wyjaśnia zasadę wykonywania algorytmu z powyższego listingu 5.10, przedstawiono na rys. 5.7. Znaczenie tego diagramu jest identyczne jak w przykładzie na rys. 5.6. Wprowadzenie dodatkowego parametru „*T*” do omawianego algorytmu powoduje, że rezygnuje się z utworzenia stosu dla zmiennych lokalnych funkcji. Efektem tego jest to, że wśród czynności algorytmu nie ma operacji obsługujących stos (porównanie z rys. 5.6). Ponadto, rozwiązanie zadania uzyskuje się już w momencie osiągnięcia najniższego poziomu wywołań rekurencyjnych. Podczas powrotu do poziomu zerowego, jedyną czynnością wykonywaną przez algorytm jest tylko przekazywanie już otrzymanego rozwiązania. Zatem można uznać, że omawiany algorytm jest dwukrotnie szybszy od algorytmu zamieszczonego w listingu 5.9. Jednocześnie, przykład ten wyjaśnia powód, dlaczego warto unikać używania stosu programu.

Derekursywacja algorytmów

Derekursywacja algorytmów jest procesem przekształcania procedur (funkcji), zapisanych z użyciem rekurencji, na jej odpowiednią postać iteracyjną z zachowaniem pełniej równoważności funkcjonalnej. Derekursywacja procedur



Rys. 5.7. Diagram czynności, który obrazuje działanie algorytmu z listingu 5.10, na przykładzie obliczania wartości 3!; opis w tekście

(funkcji) jest realizowana szczególnie w sytuacji, kiedy jest to konieczne z uwagi na parametry czasowe programu komputerowego lub ograniczone zasoby pamięciowe – np. aby uniknąć użycia stosu programu.

Początkowo, proces ten był stosowany do algorytmów zapisywanych w języku programowania, który nie był wyposażony w mechanizm wywołań rekurencyjnych. Obecnie, prawie wszystkie języki, a zwłaszcza obiektowe, umożliwiają programowanie rekursyjne. Jednak, przyjęło się stosować zwyczaj programowania, który nakazuje używać rekursywnego zapisu procedur (funkcji) dla pierwotnej wersji algorytmów, natomiast w momencie zapisu jego wersji końcowej wszystkie wystąpienia rekurencyjne należy przetransformować na analogiczną postać iteracyjną. Takie postępowanie pozwala wykorzystać zalety rekursji we wstępnej fazie programowania (tj. zwięzły i prosty zapis), oraz ominąć ewentualne ujawnienie się wad rekurencji podczas działania finalnej wersji programu.

Oczywiście, te zalecenie nie jest aksjomatem i w praktyce są dopuszczalne odstępstwa od tych zaleceń.

Należy pamiętać, że nie każdy algorytm rekursyjny można przekształcić na jej odpowiednik iteracyjny. Podstawowym warunkiem do spełnienia jest to, aby wywołanie rekurencyjne procedury (funkcji) było **terminalne**, a dany algorytm zawierał tylko jedno takie wywołanie. Wywołaniem terminalnym procedury (funkcji) nazywa się takie wywołanie rekursyjne, po którym nie następuje żadna instrukcja tej procedury (funkcji). Należy pamiętać, że umieszczenie wywołania rekursyjnego wewnątrz instrukcji powtórzeniowej (pętli) powoduje, że takie wywołanie nigdy nie jest terminalne.

Jedna z najpopularniejszych metod derekursywacji polega na zastąpieniu wystąpienia rekursyjnego procedury (funkcji) instrukcjami nierekurencyjnymi, które wykonują operacje na strukturze danych (np. stosie) utworzonej przez użytkownika. Dzięki temu, program jest pod pełną kontrolą przez cały czas jego działania, a struktura danych nigdy nie zostanie przepełniona. Ponadto, pamięć dla takiej struktury przydzielana jest w sposób dynamiczny podczas wykonywania programu. Natomiast pamięć dla stosu programu, który jest wykorzystywany przez mechanizm rekurencji, jest przydzielana jednorazowo przez system operacyjny w momencie uruchomienia programu.

Dodatkowym zabiegiem wykonywanym podczas derekursywacji algorytmów jest eliminacja zmiennych lokalnych. Polega to na zastąpieniu zmiennych lokalnych w procedurze (funkcji) zmiennymi globalnymi. Jest to wykonywane jeszcze przed transformacją algorytmu na postać iteracyjną. Powód takiego zabiegu wyjaśnia rys. 5.6 – tj. zmienne lokalnych procedur (funkcji) są zawsze przechowywane w stosie programu.

Na zakończenie zostanie przedstawiona iteracyjna wersja funkcji „*Silnia*”, którą umieszczono w listingu 5.11. Wywołanie rekurencyjne funkcji zamieniono na instrukcję powtórzeniową typu „*Do .. Loop*”. Stos programu oraz działania na nim zostały zastąpione jedną zmienną liczbową „*T*” oraz instrukcją znajdującą się w linii 3 (tj. wewnątrz pętli). Pomimo tych zabiegów, algorytm nadal cechuje się rekurencyjnym podejściem rozwiązania zadania.

Listing 5.11.

```
01 Function Silnia(ByVal X As Double, ByVal T As Double = 1) As Double
02   Do While (X <> 0)
03     T = X * T
04     X = X - 1   'instrukcja gwarantująca spełnienie warunku zakończenia pętli
05   Loop
06   Return T
07 End Function
```

5.2.3. Programowanie dynamiczne

Programowanie dynamiczne jest techniką projektowania algorytmów, które głównie mają zastosowanie do rozwiązywania zagadnień o naturze optymalizacyjnej lub podobnej. Zostało ono opracowane w drugiej połowy XX wieku przez matematyka R.E. Bellman'a [33], który po raz pierwszy zastosował je do rozwiązania problemu sterowania optymalnego układów dyskretnych.

Programowanie przy użyciu tej metody sprowadza się do rekurencyjnego sformułowania rozwiązania problemu. Jednak w tym przypadku nie używa się rekursyjnego wywoływania procedury (funkcji), a proces poszukiwania wyniku zadania rozpoczyna się od uzyskania rozwiązania problemu elementarnego, nazywanego również warunkiem brzegowym. Następnie, na jego podstawie wylicza się kolejno rozwiązania problemu wyższego rzędu w sposób iteracyjny tak długo, aż otrzyma się rozwiązanie problemu najwyższego rzędu. Algorytmy tego typu najczęściej przechowują wszystkie rozwiązania w tablicy. Dzięki temu nie ma potrzeby powtarzać tych czynności obliczeniowych, dla których już wcześniej wyznaczono wynik. Wskazane jest, aby ta struktura danych nie była zmienną lokalną procedury (funkcji), która realizuje ten algorytm.

Formułowanie algorytmu, który wykorzystuje zasady programowania dynamicznego, sprowadza się do trzech zasadniczych etapów (listing 5.12):

- **koncepcji** – dla danego problemu tworzy się rekurencyjny model jego rozwiązania z jednoczesnym ścisłym określeniem przypadków elementarnych (warunków brzegowych);
- **inicjacji** – tworzy się strukturę danych, najczęściej tablicę (linia 3 w listingu 5.12), do której wprowadza się wartości brzegowe (rozwiązania przypadków elementarnych – linia 4);
- **progresji** – oblicza się rozwiązania problemów wyższego rzędu na podstawie wartości już wpisanych do tablicy, a uzyskany nowy wynik wprowadza się do odpowiedniej komórki (linie 7÷10).

Listing 5.12.

```
01 Funkcja PD (parametr N jako rząd problemu) zwracająca wynik w
02 {
03    utwórz tablicę N-wymiarową W(0..N)
04    wstaw do tablicy wynik rozwiązania przypadku elementarnego W(0)
05    Dopóki nie uzyskano rozwiązania globalnego W(N) wykonuj
06    {
07       Na podstawie wyniku rozwiązania problemu niższego rzędu W(i-1) wykonuj
08           rozwiąż problem wyższego rzędu W(i)
09       wstaw do tablicy wynik rozwiązania wyższego rzędu W(i)
10       zwiększ wartość zmiennej i określający rząd problemu
11    }
12    zwróć wynik rozwiązania globalnego w = W(N)
13 }
```

Przedstawiona jeszcze raz funkcja „*Silnia*” w listingu 5.13, została opracowana według zasad programowania dynamicznego. Rekurencyjnym modelem rozwiązania tego problemu jest instrukcja iteracji, która znajduje się w liniach 5÷7. Strukturą danych służącą do przechowywania wyników jest zmienna „*Tab*”, która jest zadeklarowana poza funkcją (linia 1), a jej rozmiar jest dostosowywany dynamicznie w linii 3. Warunek brzegowy realizuje operacja znajdująca się w linii 4 – następuje wstawienie wartości rozwiązania elementarnego do pierwszej komórki w tablicy „*Tab*”.

Listing 5.13.

```
01 Public Tab() As Double       'zmienna globalna / „pojemnik” na dane
02 Function Silnia(ByVal X As Double) As Double
03   ReDim Tab(X)               'ustawienie rozmiaru tablicy oraz jej wyzerowanie
04   Tab(0) = 1                 'rozwiązanie problemu elementarnego
05   For i = 1 To X             'rozwiązanie rekurencyjne z użyciem „pojemnika”
06     Tab(i) = Tab(i-1) * i
07   Next i
08   Return Tab(X)              'zwrócenie wyniku rozwiązania
09 End Function
```

5.2.4. Metoda *„zachłanna”*

Jest to technika projektowania algorytmów, które są przeznaczone do wykonywania obliczeń natury optymalizacyjnej. Ideą programowania metodą „*zachłanną*” (z ang. *greedy method*) jest sformułowanie modelu rozwiązania problemu, który zapewni uzyskanie wyniku w możliwie krótkim czasie, ale przy założeniu pewnego kompromisu. Mianowicie, w ogólnym przypadku uzyskiwany wynik obliczeń jest rozwiązaniem jedynie przybliżonym do wyniku optymalnego, aczkolwiek nie wyklucza się, że będzie on rozwiązaniem oczekiwanym. Oznacza to, że do modelu rozwiązania należy wprowadzić pewne założenia (ograniczenia), pozwalające na uznanie rozwiązania problemu za zadawalające [1, 33]. W tym celu przyjmuje się, że:

- dane wejściowe muszą tworzyć zbiór, który spełnia określone kryteria;
- na każdym etapie rozwiązywania problemu, odszukuje się takie rozwiązanie cząstkowe, które można uznać za wynik rokujący w uzyskaniu co najmniej jednego rozwiązania globalnego;
- po uzyskaniu rezultatu zadawalającego, następuje zaniechanie dalszego rozwiązywania problemu.

Formalny zapis algorytmu, działającego zgodnie z ideą metody „*zachłannej*”, za pomocą pseudo-kodu programowania umieszczono w listingu 5.14. Przedstawia on pewną funkcję o nazwie „*MZ*” zwracającą wynik rozwiązania problemu w postaci pewnego zbioru „*C*”, który z założenia jest rozwiązaniem w przybliżeniu optymalnym. Parametrem tej funkcji jest zbiór „*P*”, w którym umiesz-

czone są dane wejściowe. Działanie tego algorytmu sprowadza się do pobrania elementu „optymalnego" ze zbioru danych wejściowych, na podstawie którego sprawdza się czy jest możliwe wygenerowanie rozwiązania cząstkowego – jeśli tak, to powstały wynik jest dołączany do zbioru „*C*". Postępowanie to realizuje się dopóki, dopóty nie jest spełniony warunek zakończenia poszukiwań wyniku końcowego.

Listing 5.14.

```
Funkcja MZ (parametr P jako zbiór danych) zwracająca wynik jako zbiór C
{
   wyzeruj zbiór C
   Dopóki zbiór C nie jest rozwiązaniem i zbiór P nie jest pusty wykonuj
   {
      odszukaj i pobierz optymalny element pi ze zbioru P
      Jeśli na podstawie pi można uzyskać rozwiązanie cząstkowe ci to
          dodaj ci do zbioru C
   }
   Jeśli zbiór C może być rozwiązaniem to
      zwróć wynik w postaci zbioru C
   w przeciwnym przypadku
      zwróć wynik w postaci zbioru pustego co oznacza „brak rozwiązania"
}
```

Przykład

Rozpatrzmy sposób opracowania optymalnego planu produkcji dla przykładowej obrabiarki. Zakładamy, że zbiorem danych są przedmioty, które różnią się czasem wykonania. Zbiór ten można zapisać schematycznie:

$$P = \{p_1; p_2; p_3; p_4; p_5; p_6\} = \{2; 1{,}5; 0{,}5; 2; 1; 3\}, \qquad (5.1)$$

gdzie podane wyżej uszeregowane wartości reprezentują poszczególne elementy zbioru i wyrażają wymagany czas (np. w godzinach) ich wykonania na rozpatrywanej obrabiarce.

Celem zadania jest wykonanie jak największej ilości przedmiotów w określonym czasie pracy obrabiarki – np. ośmiu godzin. Problem polega na tym, że łączny czas wykonania wszystkich przedmiotów ze zbioru (5.1) jest większy od planowanego czasu pracy maszyny. Zatem powstaje pytanie: które przedmioty wybrać, ale w możliwie najszybszy sposób? Rozwiązanie tego problemu, stosując omawianą metodę programowania, wymaga wykonania następujących czynności:

- odszukać i pobrać element (przedmiot) ze zbioru (5.1), dla którego czas jego wykonania jest najmniejszy;
- sprawdzić, czy można wykonać ten przedmiot, wraz z innymi już wcześniej wybranymi przedmiotami, w planowanym czasie pracy obrabiarki;

- jeżeli wynik sprawdzania powyższego kryterium jest pomyślny, to aktualnie rozpatrywany przedmiot umieścić w zbiorze rozwiązania zadania;
- powyższe czynności powtórzyć aż do momentu, w którym zabraknie elementów w zbiorze danych wejściowych lub wynik warunku sprawdzania kryterium (czynność druga) będzie negatywny.

Efektem tak działającego algorytmu jest uzyskanie następującego zbioru rozwiązania:

$$C = \{p_3, p_5, p_2, p_1, p_4\}. \tag{5.2}$$

Porównując zbiory (5.1) i (5.2) można stwierdzić, że w planowanym czasie działania obrabiarki (8 godzin) wykona się tylko pięć przedmiotów, których łączny czas obróbki będzie wynosił 7 godzin. Z porównania tych zbiorów wynika również, że jeżeli przedmiot p_4 zastąpi się elementem p_6, to czas ten będzie równy 8 godzin. Zatem, uzyskany wynik nie jest optymalny. Ale został on osiągnięty w możliwie najprostszy sposób, bez stosowania dodatkowych czynności modyfikujących pierwotną postać rozwiązania. Jest to szczególnie ważne, jeżeli zbiór danych wejściowych będzie składał się ze znacznie większej ilości elementów – w takim przypadku dodatkowy nakład pracy algorytmu jest nieopłacalny.

Załóżmy teraz, że elementy zbioru (5.1) są reprezentowane dodatkowo przez wartości reprezentujące np. koszt ich wykonania. W takiej sytuacji celem zadania może być wykonanie jak największej ilości przedmiotów przy możliwie najniższych poniesionych kosztach. Jest to problem natury polioptymalizacyjnej, który wymaga dodatkowo zbudowania pewnej funkcji celu oraz określenia tzw. współczynników wagowych. Taki przypadek uznaje się za „typowy” do rozwiązania metodą „*zachłanną*”.

5.3. Wzorce programowania komputerów

5.3.1. Wzorce projektowe

Pojęcie „wzorce projektowe” (z ang. *design patterns*) jest ściśle związane z programowaniem zorientowanym obiektowo. Należy je postrzegać jako sposób na wielokrotne wykorzystanie wcześniej opracowanego szkieletu, architektury, motywu lub/i kodu programu w projektach nowych programów. Ideą definiowania wzorców jest opisanie oraz skatalogowanie typowych schematów oraz reguł programowania, które mogą być wykorzystane do rozwiązywania grupy problemów o podobnej naturze [4, 23, 29].

Pierwszym, znaczącym wzorcem projektowym był wzorzec Krasnera-Pope’a (MVC), który pojawił się pod koniec lat osiemdziesiątych ubiegłego wieku. Przedstawił on schemat szkieletu aplikacji, która tworzona była za pomocą języka obiektowego Smaltalk [4]. Opisywał on reguły tworzenia interfejsu użytkow-

nika, zakładając że powinien on składać się z trzech zasadniczych części – **modelu** danych aplikacji, **widoku** reprezentującego interfejs użytkownika na ekranie oraz **kontrolera** interakcji pomiędzy użytkownikiem i widokiem (ściślej: sposób reakcji interfejsu na dane wejściowe podawane przez użytkownika).

W latach dziewięćdziesiątych XX wieku, rozpoczęto systematyzować już istniejące i nowo odkrywane wzorce. Warto w tym miejscu podkreślić, że wzorce nigdy nie były wymyślane – były one tzw. „efektem ubocznym" procesu projektowania nowych (nowatorskich) aplikacji. Dlatego przyjmuje się, że wzorce projektowe są **odkrywane** [4, 29], a proces ich poszukiwania jest nazywany **eksploracją wzorców**.

Zatem, opis sposobu projektowania i programowania zorientowanego obiektowo ma na celu ułatwienie opracowania programistycznego rozwiązania określonego problemu. Jednak, aby móc taki opis nazwać formalnie wzorcem projektowym, powinien on składać się z czterech elementów składowych [29]:

- **nazwy** – jest to etykieta wzorca (najczęściej wyrażona w języku angielskim), która jednoznacznie określa reprezentowany problem projektowy;
- **definicji problemu**, w której podaje się objaśnienie istoty problemu projektowego oraz listę warunków, jakie muszą być spełnione, aby zastosowanie danego wzorca było zasadne;
- **rozwiązania problemu** – jest to uogólniony opis, w formie abstrakcyjnej, wszystkich składowych projektu tak, aby mógł on być zastosowany jako szablon do wielu, tematycznie podobnych sytuacji danego problemu;
- **kosztów użycia wzorca** – jest to opis efektów korzystnych i negatywnych wynikających z zastosowania danego wzorca projektowego do opracowania rozwiązania problemu.

Ponadto, wzorce projektowe mogą mieć postać wyrażoną na różnych poziomach abstrakcji – od rozwiązań szczegółowych do bardzo ogólnych. W literaturze specjalistycznej, np. [4, 23, 25] opisano wiele wzorców. Niektóre z nich są powszechnie stosowane w szerokim zakresie, inne są używane do projektowania jedynie pojedynczych problemów.

Obecnie, podstawową klasyfikacją wzorców projektowych jest ich podział ze względu na funkcyjność. Wyróżnia się następujące kategorie wzorców [25]:

- konstrukcyjne,
- strukturalne,
- czynnościowe.

Autor skryptu ograniczył się jedynie do podania w niej ogólnej charakterystyki wybranych wzorców. Celem rozdziału jest jedynie zasygnalizowanie tematu, ponieważ jest on bardzo obszerny i stanowi przedmiot wielu badań naukowych w dziedzinie informatyki. Przyjmuje się, że przedstawiony poniżej opis

zagadnienia jest wystarczający dla inżyniera-mechanika. Osoby, które chcą uzyskać pełną wiedzę na omawiany temat, powinny zapoznać się np. z treścią pozycji [4].

Wzorce konstrukcyjne

Wzorce konstrukcyjne opisują sposoby tworzenia klas oraz ich instancji. Cechą charakterystyczną jest to, że kod programu ma możliwość samodzielnego decydowania o tym, jakiego rodzaju obiekt należy utworzyć w danej sytuacji. Zastosowanie wzorców tej kategorii ma na celu zwiększenie elastyczności programu komputerowego.

Podstawowymi wzorcami projektowymi tego rodzaju są [4, 29]:

- „*Factory Method*” – proponuje stworzenie takiej klasy, która ma za zadanie zwracanie instancji różnych klas pochodnych (obiektów) bazując na pewnej abstrakcyjnej klasie bazowej; o tym, jaki obiekt należy utworzyć decydują klasy pochodne na podstawie otrzymanych danych oraz w oparciu o pewien proces decyzyjny; wzorzec ten jest stosowany w sytuacjach, gdy trudno jest przewidzieć jakiego rodzaju obiekt będzie utworzony przez klasę, a decyzję o utworzeniu obiektu zamierza się powierzyć klasom pochodnym;
- „*Abstract Factory*” – podaje sposób dostarczania interfejsu umożliwiającego utworzenie i pobieranie obiektu jednej z kilku, hierarchicznie powiązanych klas; podstawowym celem stosowania tego wzorca jest odizolowanie klas, których obiekty są tworzone oraz uniemożliwienie przypadkowego użycia klasy innego rodzaju;
- „*Singleton*” – zapewnia stworzenie takiej klasy, która najczęściej posiada tylko jedną instancję (obiekt) oraz umożliwia dostęp do niej; celem stosowania tego wzorca jest ograniczenie ilości obiektów danej klasy, najczęściej tylko do jednego;
- „*Builder*” – podaje sposób dynamicznego tworzenia nowych obiektów z komponentów (innych obiektów); reprezentacja obiektów jest zmieniana w zależności od potrzeb programu przy jednoczesnym ukrywaniu szczegółów; wzorzec ten pozwala oddzielić konstrukcję złożonych obiektów od sposobu ich reprezentowania;
- „*Prototype*” – definiuje się klasy charakteryzujące się uogólnioną specyfikacją, a decyzja o ostatecznym ich kształcie jest podejmowana podczas działania programu; celem stosowania tego wzorca jest zastąpienie pracochłonnego tworzenia nowego obiektu przez inny proces – tj. przez klonowanie lub kopiowanie instancji klasy prototypowej, a następnie jej modyfikowanie w zależności od potrzeb.

Wzorce strukturalne

Wzorce strukturalne wskazują sposoby grupowania klas i obiektów w większe struktury (zespoły). Poszczególne wzorce klas (obiektów) różnią się między sobą przede wszystkim mechanizmem grupowania. Wzorce klas opisują sposób wykorzystania dziedziczenia, natomiast wzorce obiektów przedstawiają sposób użycia kompozycji obiektów. Celem stosowania wzorców strukturalnych jest uzyskanie złożonych interfejsów użytkownika (w przypadku klas) lub bardziej zaawansowanych struktur danych (w przypadku obiektów). Cechą charakterystyczną wzorców strukturalnych jest również to, że wiele z nich wykazuje podobieństwo w zakresie pewnych, pokrywających się cech.

Podstawowymi wzorcami projektowymi tego rodzaju są [4, 29]:

- „*Adapter*" – przedstawia metodę dopasowania interfejsu klasy do innego interfejsu w ten sposób, aby klasa dopasowywana zachowywała się tak jak klasa o innym (docelowym) interfejsie; mechanizm dopasowania bazuje albo na dziedziczeniu lub na kompozycji – w pierwszym przypadku tworzona jest nowa klasa, której interfejs uzupełnia się o brakujące metody, natomiast w drugiej sytuacji powstaje całkiem nowa klasa z właściwym interfejsem, która zawiera instancje (obiekty) klasy dopasowywanej; wzorca tego używa się w programach, gdy jest problem współdziałania klas niepowiązanych ze sobą relacją dziedziczenia;
- „*Bridge*" – podaje sposób przekształcenia interfejsu klasy w inny interfejs, ale poprzez oddzielenie go od implementacji tej klasy; efektem użycia tego wzorca jest możliwość dowolnej, niezależnej zmiany zarówno implementacji, jak i interfejsu klasy oraz ukrycie rzeczywistej reprezentacji danych; podstawowym celem wzorca jest uzyskanie możliwości dowolnej zmiany klasy oraz prezentowania jej używając nadal tego samego kodu programu;
- „*Composite*" – umożliwia utworzenie złożonej struktury danych, najczęściej o hierarchii typu „całość-część" (tj. agregacji), gdzie każdy element tej struktury może być zarówno pojedynczym obiektem, jak i kolekcją obiektów; celem wzorca jest ujednolicenie sposobu tworzenia i obsługi struktur danych oraz rozwiązanie problemu budowy interfejsu struktury, szczególnie dla drzew;
- „*Decorator*" – pokazuje sposób dynamicznego dodawania nowej funkcjonalności do obiektu, przy czym modyfikacja jego zachowań następuje poprzez kompozycję i nie wymaga tworzenia klasy pochodnej;
- „*Facade*" – umożliwia obudowanie zbioru złożonych klas jednym interfejsem, który upraszcza sposób posługiwania się nimi; celem stosowania wzorca jest reprezentacja złożonego podsystemu za pomocą jednej klasy; wzorzec ten jest najczęściej używany do tworzenia struktury dostępu do bazy danych, która może być zbudowana z podsystemów baz różnego typu;

- „*Flyweight*” – podaje sposób ograniczenia podobnych obiektów współdzielonych, które różnią się między sobą tylko niewielką liczbą składowych, a informacja o stanie obiektu jest przechowywana poza obiektem; celem wzorca jest uniknięcie stosowania w programie nadmiernej ilości podobnych obiektów, co jest realizowane poprzez usunięcie różnic (tj. skasowanie pewnych, unikatowych składowych obiektów) oraz oddzielenie wewnętrznych danych od zewnętrznych; dane zewnętrzne oraz usunięte składowe są przekazywane do obiektu jako parametry metod;
- „*Proxy*” – proponuje utworzenie prostego obiektu i wykorzystanie jego do tymczasowego reprezentowania bardziej złożonego obiektu docelowego, który będzie używany w programie znacznie później, a jego uzyskanie jest czasochłonne; prosty obiekt posiada zwykle ten sam zestaw metod (interfejs) co obiekt docelowy, a w odpowiednim czasie wywoływania metod są przekazywane do metod obiektu docelowego.

Wzorce czynnościowe

Wzorce czynnościowe wspomagają definiowanie ścieżek komunikacyjnych między obiektami oraz ułatwiają kontrolowanie przepływu sterowania w skomplikowanych programach. Wiele z nich jest powszechnie stosowanych w programach komputerowych posiadających interfejs graficzny.

Podstawowymi wzorcami projektowymi tego rodzaju są [4, 25, 26, 29]:

- „*Chain of Responsibility*” – służy do ograniczenia powiązania pomiędzy obiektami poprzez przekazywanie pomiędzy kolejnymi klasami tzw. żądania obsługi, aż do momentu jego wykonania (obsłużenia); wzorzec ten konstruuje strukturę najczęściej liniową (rzadziej w formie drzewa) złożoną z szeregu klas, które podejmują kolejno próbę obsługi pewnego żądania; poszczególne klasy nie posiadają wiedzy o możliwościach innych klas i nie są ze sobą w żaden sposób powiązane;
- „*Command*” – pokazuje sposób przekazania żądania obsługi do określonego obiektu posiadającego interfejs publiczny; wzorzec używa prostych obiektów do reprezentacji operacji wykonywanych przez program oraz tworzy dziennik tych operacji, których wykonanie można wycofać;
- „*Interpreter*” – definiuje sposób, w jaki można zdefiniować gramatykę języka programu i używać jej do interpretacji instrukcji wprowadzonych przez użytkownika; język ten służy do budowy makr programu, w których automatyzuje się często wykonywane operacje;
- „*Iterator*” – formalizuje sposób poruszania się po elementach dowolnej kolekcji danych; wzorzec ten pozwala przeglądać elementy struktury danych (listy, kolekcji) za pomocą standardowego interfejsu bez konieczności znajomości reprezentacji tych danych; wzorzec umożliwia również zdefinio-

wanie specjalnych sposobów wykonywania dodatkowych operacji na elementach przeglądanej struktury;
- „*Mediator*" – podaje sposób osłabienia powiązań pomiędzy klasami programu, poprzez wprowadzenie osobnej tzw. „klasy mediatora"; wzorzec upraszcza komunikację obiektów przez likwidację bezpośrednich odwołań pomiędzy obiektami oraz centralizuje tę komunikację w obiekcie (klasie) mediatora; wszystkie klasy programu informują klasę mediatora o zmianie w zestawieniu swoich metod, bądź komunikują się z tą klasą w celu uzyskania informacji o metodach innych klas;
- „*Memento*" – definiuje sposób przechowywania danych o obiekcie w celu późniejszego odtworzenia jego stanu; zastosowanie wzorca umożliwia zapisywanie i odtwarzanie stanu każdego obiektu bez konieczności implementacji tych operacji w klasie obiektu, przy czym zasada hermetyzacji obiektu zostaje zachowana;
- „*Observer*" – przedstawia pewien sposób powiadamiania kilku obiektów o pewnych danych; wzorzec ten zakłada, że pewna grupa obiektów (lub jeden obiekt) reprezentuje dane programu, które są obserwowane i jednocześnie prezentowane w różnej formie przez inną grupę różnych obiektów;
- „*State*" – podaje sposób zmiany zachowania obiektu na skutek zmiany jego stanu wewnętrznego; wzorzec tworzy grupę (agregację) obiektów częściowych reprezentujących różne stany obiektu całkowitego; zmiana stanu obiektu całkowitego następuje poprzez pobranie odpowiedniego jego obiektu składowego.

5.3.2. Wzorzec programowania systemu inżynierskiego

Ogólna charakterystyka

Programowanie komputerów, jako sposób budowy narzędzi wspomagających obliczenia inżynierskie, posiada niezaprzeczalne znaczenie dla dziedziny mechaniki, budowy i eksploatacji maszyn. Obecnie każda gałąź tej dziedziny nie może w pełni funkcjonować jeżeli inżynierowie nie używają specjalistycznych programów komputerowych. Charakteryzują się one pewnymi wspólnymi cechami, które składają się na swoisty wzorzec programowania. Podstawowymi cechami każdego takiego systemu komputerowego są jego modułowość oraz wykonywanie obliczeń, z reguły o naturze optymalizacyjnej. Poniżej zostaną one krótko opisane.

System komputerowy stosowany do obliczeń inżynierskich powinien składać się z trzech modułów (wydzielonych części składowych). Są to [22]:
- **preprocesor** – pełni rolę interfejsu programu; podstawowym jego zadaniem jest umożliwienie wprowadzenia danych wyjściowych do systemu, włącznie

z ich uporządkowaniem i zgrupowaniem w odpowiednie struktury; preprocesor pozwala również przygotować proces obliczeniowy, który będzie realizowany w kolejnym module programu; większość czynności i operacji wykonywanych przez preprocesor są zautomatyzowane;

- **solver** – jest to najważniejszy moduł każdego systemu komputerowego; wykonuje on wszystkie obliczenia, które są wymagane do rozwiązania problemu inżynierskiego; zawiera on implementację metody rozwiązania zadania, na którą składają się algorytmy i struktury danych reprezentujące zarówno dane wyjściowe, jak i dane wynikowe (wynik rozwiązania);
- **postprocesor** – jest to interfejs programu, którego zadaniem jest przetwarzanie danych wynikowych w informację prezentowaną w formie czytelnej dla użytkownika programu;

Obliczenia wykonywane w module „*solver*”, w ramach implementacji metody rozwiązania zadania inżynierskiego, są wykonywane w oparciu o algorytmy numeryczne. Są one budowane według ogólnych zasad programowania komputerów. Ale oprócz typowych technik programowania, w tych algorytmach wykorzystuje się również metody optymalizacji (polioptymalizacji).

Definicja problemu i sposób jego rozwiązania

Załóżmy, że istnieje pewien zbiór danych, który reprezentuje problem inżynierski. Zbiór ten zapisujemy w postaci:

$$D = \{d_1, d_2, d_3, \ldots, d_m\}, \tag{5.3}$$

gdzie każdy element tego zbioru jest rekordem składającym się z pewnej ilości pól kluczowych, przy czym liczebność tych rekordów wynosi *m*. Zadaniem jest odszukanie optymalnego rozwiązania problemu inżynierskiego w oparciu o analizę zbioru danych *D*. Załóżmy, że rozwiązanie to można sprowadzić do pewnej reprezentacji zapisanej w postaci rekordu:

$$r_{optm} = \{p_1, p_2, p_3, \ldots, p_k\}, \tag{5.4}$$

gdzie jego polami są poszukiwane optymalne parametry liczbowe. Wiemy jednocześnie, że według określonych kryteriów, rozwiązanie (5.4) należy do pewnego zbioru wszystkich możliwych rozwiązań *R* problemu inżynierskiego:

$$r_{optm} \in R, \tag{5.5}$$

gdzie zbiór ten można zapisać w podobny sposób jak zbiór (5.3), mianowicie:

$$R = \{r_1, r_2, r_3, \ldots, r_n\}. \tag{5.6}$$

Liczebność *n* powyższego zbioru jest zdeterminowana przez ilość pól rekordu *r* (która wynosi *k*) oraz ilość możliwych wartości, jakie mogą przyjąć poszczególne pola. Zakładając, że każde pole rekordu może przyjmować *x* różnych wartości z zakresu (p_{min}, p_{max}), łatwo określić, że liczebność *n* wynosi:

$$n = \prod_{i=1}^{k} (x_i). \tag{5.7}$$

Powyższa wartość ma znaczenie zasadnicze, ponieważ w sposób pośredni określa wymagany czas rozwiązania problemu inżynierskiego.

Zatem, rozwiązanie problemu inżynierskiego sprowadza się w zasadzie do wykonania dwóch grup czynności, mianowicie:

- zbudowania zbioru możliwych rozwiązań *R* bazując na pewnych założeniach brzegowych oraz wprowadzonych ograniczeniach;
- odszukania w zbiorze *R* rekordu optymalnego r_{optm}, który spełnia warunek (5.8) taki, że pewna funkcja celu przyjmuje najmniejszą możliwą wartość.

$$\Phi = f(D, r_{optm}) \rightarrow min. \tag{5.8}$$

W praktyce, te dwie grupy czynności „przeplatają się" – tzn. łączy się je w jeden algorytm zwany „procesem optymalizacji (polioptymalizacją) rozwiązania". Algorytm ten zakłada, że elementy zbioru (5.6) są generowane w sposób dynamiczny. Za każdym razem, gdy jest tworzony nowy rekord zbioru *R* sprawdza się jednocześnie, czy może on być rekordem optymalnym. W przypadku stwierdzenia, że już odszukano r_{optm} dalsze generowanie zbioru *R* jest przerywane.

Algorytm rozwiązania tak przedstawionego problemu inżynierskiego może być budowany w oparciu o dwie znane metody optymalizacyjne [22], mianowicie:

- „*systematycznego przeszukiwania*",
- „*Monte-Carlo*".

Formalny zapis algorytmu poszukiwania rozwiązania, bazujący na procesie optymalizacji według metody „*systematycznego przeszukiwania*", zamieszczono w listingu 5.15. Algorytm ten wymaga zastosowania zagęszczonej konstrukcji iteracji, gdzie ilość poziomów wynosi *k* – tj. jest równa liczebności pól rekordu reprezentującego rozwiązanie zadania. Istotą tej metody jest to, że w trakcie obliczeń pola $p_1, \ldots, p_k$ przyjmują kolejno wszystkie możliwe wartości. Przykładowo, dla pola p_1 czynność ta jest realizowana przez instrukcje zamieszczone w liniach 4, 7 oraz 27 w listingu 5.15. Dodatkowo, instrukcja w linii 27 determinuje dokładność przeszukiwania (tj. ilość wartości dla danego parametru).

Po ustaleniu chwilowych wartości parametrów $p_1, \ldots, p_k$ kolejno tworzy się rekord r_i (linia 12), oblicza się pewną funkcję celu Φ_i (linia 13), a następnie sprawdza się, czy jej wartość jest mniejsza od ostatnio zapamiętanej wartości najmniejszej Φ_C. Jeśli tak, to przyjmuje się, że ten rekord będzie rozwiązaniem (linia 17).

Listing 5.15.

```
01 Funkcja OPT1 (parametr D jako zbiór danych) zwracająca wynik jako rekord r
02 {
03    ustaw Φc na możliwie największą wartość
04    ustaw początkową wartość pola p1 = p1min
05    ...
06    ustaw początkową wartość pola pk = pkmin
07    Dopóki p1 < p1max wykonuj
08    {
09       ...
10           Dopóki pk < pkmax wykonuj
11           {
12              przypisz wartości p1 … pk do rekordu ri
13              oblicz Φi na podstawie zbioru D oraz rekordu ri
14              Jeśli Φi < Φc to
15              {
16                    zapamiętaj, że teraz Φc = Φi
17                    załóż, że teraz roptm = ri
18                    Jeśli funkcja Φc > Φmin to
19                          zwróć roptm jako wynik r
20                          zakończ algorytm poszukiwania rozwiązania
21                    w przeciwnym przypadku
22                          kontynuuj poszukiwanie rozwiązania
23              }
24              ustaw kolejną wartość pola pk z zakresu (pkmin, pkmax)
25           }
26       ...
27       ustaw kolejną wartość pola p1 z zakresu (p1min, p1max)
28    }
29    zwróć roptm jako wynik r
30 }
```

Zaletą opisanego algorytmu jest duża dokładność przeszukiwania zbioru możliwych rozwiązań *R*, która w zasadzie gwarantuje odszukanie rozwiązania optymalnego. Niestety, wadą tej metody jest duży czas obliczeń. Zgodnie z zależnością (5.7) jest to algorytm o teoretycznej złożoności czasowej typu wielomianowej (zakładając, że parametr *k* jest stały) lub wykładniczej (zakładając, że parametr *k* jest zmienny). Sposobem skrócenia czasu – ale tylko przy sprzyjającym układzie rekordów w zbiorze *R*, jest wprowadzenie dodatkowej instrukcji decyzji (linie 18÷22), która określa warunek przedwczesnego zakończenia algorytmu. Zakłada się, że jeżeli funkcja celu Φ_C osiągnie wartość zadawalającą Φ_{min}, dalsze poszukiwanie rozwiązania jest niezasadne.

Bardziej efektywnym sposobem poszukiwania rozwiązania jest zastosowanie algorytmu bazującego na procesie optymalizacji według metody „*Monte-Carlo*”. Zakłada się, że sprawdzany rekord r_i jest pobierany ze zbioru *R* w sposób losowy. Formalny zapis tego algorytmu umieszczono w kolejnych listingach 5.16 oraz 5.17, przy czym listing drugi przedstawia procedurę losowania rekordu *r*.

Procedura ta jest wywoływana w listingu pierwszym w linii 3 (w celu zainicjowania funkcji celu Φ_C oraz struktury r_{optm}) oraz w linii 8.

Listing 5.16.

```
01 Funkcja OPT2 (parametr D jako zbiór danych) zwracająca wynik jako rekord r
02 {
03    pobierz rekord r1
04    oblicz Φc na podstawie zbioru D oraz rekordu r1
05    załóż, że ropt = r1
06    Dopóki funkcja Φc > Φmin oraz rekord r nie jest pusty wykonuj
07    {
08       pobierz rekord ri
09       oblicz Φi na podstawie zbioru D oraz rekordu ri
10       Jeśli Φi < Φc to
11            {
12            zapamiętaj, że teraz Φc = Φi
13            załóż, że teraz roptm = ri
14            }
15    }
16    zwróć roptm jako wynik r
17 }
```

Listing 5.17.

```
01 Funkcja pobierz() zwracająca wynik jako rekord r
02 {
03    zwiększ wartość zmiennej statycznej Losowanie o jeden
04    jeżeli wartość Losowanie jest mniejsza od maksymalnej to
05         wylosuj wartość pola p1 z zakresu (p1min, p1max)
06         ...
07         wylosuj wartość pola pk z zakresu (pkmin, pkmax)
08         przypisz wartości p1 … pk do rekordu r
09         zwróć rekord r
10    w przeciwnym przypadku
11         zwróć rekord pusty r = NULL
12    }
```

Algorytm ten ma złożoność liniową, a czas obliczeń zależy tylko od liczby losowań L rekordu r ze zbioru dopuszczalnych rozwiązań R. Ilość losowań zawsze jest mniejsza od wartości wyrażenia (5.7) o kilka rzędów wielkości. Niestety, wadą tego algorytmu jest to, że nie gwarantuje on odnalezienia rozwiązania optymalnego, szczególnie w sytuacji, jeżeli ilość losowań jest niedostatecznie mała. Z doświadczenia autora [22] wynika, że liczba losowań rzędu $10^4 \div 10^6$ pozwala uzyskać zadawalający wynik rozwiązania w bardzo krótkim czasie.

Przykład systemu komputerowego, który zbudowano na bazie omówionego wzorca, został szczegółowo omówiony w pracy własnej [22]. Służy on do analizy danych uzyskanych w próbach plastometrycznych, a efektem obliczeń jest uzyskanie optymalnej postaci równania konstytutywnego.

Literatura

1. Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, Wydanie IV, Warszawa, Wydawnictwo Naukowo-Techniczne, 2003
2. Bell G., Newell A., *Computer Structures: reading and Examples*, New York, McGraw-Hill Book Company, 1971
3. Cieciura M. *Podstawy technologii informacyjnych z przykładami zastosowań*, Warszawa, Wydawnictwo Vizja Press&IT Sp. z o.o., 2006
4. Cooper J.W., *Visual Basic wzorce projektowe*, Gliwice, Wydawnictwo Helion, 2002
5. *Encyklopedia szkolna. Matematyka*, red. Pańkowska H., Warszawa, Wydawnictwo Szkolne i Pedagogiczne, 1988
6. Gamma E., *Design Patterns: Elements of Resable Object-Oriented Software*, Boston, Addison-Wesley, 1995
7. *History of Programming Languages*, O'Reilly Media Inc., 2004 [dostęp styczeń 2005]. Dostęp w Word Wide Web: http://www.oreilly/go/languageposter
8. *Historia maszyn liczących*, 2009 [dostęp 12 grudnia 2009]. Dostęp w Word Wide Web: http://www.infor.webb.pl/
9. Kierzkowski Z., *Elementy informatyki*, Państwowe Wydawnictwo Naukowe, Poznań 1976
10. Kott R.K., Walczak K., *Programowanie w języku Fortran 77*, Warszawa, Wydawnictwo Naukowo-Techniczne, 1991
11. Kruk S., *Asembler: podręcznik użytkownika*, Warszawa, Wydawnictwo MIKOM, 1999
12. *LabView Environment Basics*, National Instruments, 2011 [dostęp marzec 2011]. Dostęp w Word Wide Web: http://www.ni.com/gettingstarted/
13. Lausen G., Vossen G., *Obiektowe bazy danych. Modele danych i języki*, Warszawa, Wydawnictwo Naukowo-Techniczne, 2000
14. Lewandowski M., *VBA dla Excela 2003/2007. Leksykon kieszonkowy*, Gliwice, Wydawnictwo Helion, 2007
15. Mackenzie D., Sharkey K, *Visual Basic.NET dla każdego*, Gliwice, Wydawnictwo Helion, 2002
16. Neibauer A.R., *Your First C/C++ Program*, Wydanie IV, Alameda, Sybex Inc., 1994

17. Perry G., *Programowanie. Przewodnik dla zupełnych nowicjuszy*, Warszawa, Wydawnictwo Intersoftland, 1993
18. Powers L., Snell M., *Microsoft Visual Studio 2008. Księga eksperta*, Gliwice, Wydawnictwo Helion, 2009
19. Reynolds J.C., *Theories of Programming languages*, Cambridge, Cambridge University Press, 1998
20. Rojas R., Hashagen U., *The First Computers: History and Architectures*, Cambridge, The MIT Press, 2000
21. Roselman B., Peasley R., Pruchniak W., *Poznaj Visual Basic 6*, Warszawa, Wydawnictwo MIKOM, 1999
22. Samołyk G., Gontarz A., Pater Z., Komputerowe wspomaganie wyznaczania krzywej umocnienia, W: *Przeprowadzone badania procesów plastycznego kształtowania w latach 2005-2007*, Lublin, Wyd. Pol. Lubelskiej, 2008, str. 165-186
23. Schmuller J., *UML dla każdego*, Gliwice, Wydawnictwo Helion, 2003
24. Sebesta R.W., *Concepts of Programming Languages*, 9th Edition, Boston, Addison Wesley, 2010
25. Stephens R., *Visual Basic 2008. Warsztat programisty*, Gliwice, Wydawnictwo Helion, 2009
26. Thayer R.: *Visual Basic 6. Księga eksperta*, Gliwice, Wydawnictwo Helion, 1999
27. Tłaczała W. *Środowisko LabView w eksperymencie wspomaganym komputerowo*, Warszawa, Wydawnictwo Naukowo-Techniczne, 2002
28. Van Roy P., Haridi S., *Concepts, Techniques, and Models of Computer Programming*, Cambridge, The MIT Press, 2004
29. Weisfeld M., *Myślenie obiektowe w programowaniu*, Wydanie III, Gliwice, Wydawnictwo Helion, 2010
30. Wojtuszkiewicz K., *Programowanie strukturalne i obiektowe*, Tom I, Warszawa, Państwowe Wydawnictwo Naukowe, 2009
31. Wróbel J., *Technika komputerowa dla mechaników*, Warszawa, Państwowe Wydawnictwo-Naukowe, 1994
32. Wróbel J., *Technika komputerowa dla mechaników. Laboratorium*, Warszawa, Oficyna Wydawnicza Politechniki Warszawskiej, 2004
33. Wróblewski P., *Algorytmy, struktury danych i techniki programowania*, Wydanie III, Gliwice, Wydawnictwo Helion, 2003

STRESZCZENIE

W skrypcie zamieszczono podstawowe informacje dydaktyczne na temat programowania komputerów. Adresatem tej książki są przede wszystkim studenci kierunków technicznych, w szczególności mechaniki, budowy i eksploatacji maszyn, gdzie informatyka nie jest przedmiotem istotnym. Celem skryptu jest przekazanie inżynierom-mechanikom, którzy nie są informatykami (w pełnym rozumieniu tego słowa), elementarnej wiedzy o zasadach tworzenia programów komputerowych używanych jako narzędzie wspomagające obliczenia inżynierskie. Opisane zagadnienia w tej książce mogą być wykorzystane jako materiał dydaktyczny w ramach takich przedmiotów jak: „języki programowania", projektowanie i programowanie obiektowe", „komputerowe przetwarzanie danych" lub podobnych.

Skrypt podzielono na pięć tematycznych części. Rozdział pierwszy jest szerokim wprowadzeniem do problematyki programowania komputerów przez inżynierów-mechaników. Podano podstawowe definicje komputera, przetwarzania danych i programowania. Wyjaśniono zagadnienia reprezentacji i przetwarzania liczb w komputerze oraz przedstawiono krótki zarys historii rozwoju techniki komputerowej z perspektywy inżyniera.

Kolejny rozdział zawiera zagadnienia związane z językami programowania. Zamieszczono klasyfikację języków, omówiono ich rozwój oraz przedstawiono ich charakterystykę. Główną uwagę skupiono na omówieniu języków wysokiego poziomu. Wyjaśniono również na czym polega translacja kodu źródłowego.

W rozdziale trzecim omówiono trzy podstawowe techniki dokumentowania programów komputerów. Zwrócono uwagę na opisywanie (schematyczne i graficzne) algorytmów oraz przedstawianie struktury programu komputerowego za pomocą języka modelowania wirtualnego. Opisy poruszanych zagadnień uzupełniono o liczne przykłady, które abstrakcyjnie reprezentują zagadnienia z zakresu technologii maszyn. Natomiast rozdział czwarty zawiera kompletny opis i charakterystykę najważniejszych paradygmatów programowania, między innymi: programowania imperatywnego, sterowanego przepływem danych oraz zorientowanego obiektowo.

Ostatni rozdział przedstawia wybrane zagadnienia z teorii programowania. Skupiono się na scharakteryzowaniu podstawowych struktur danych, które można użyć do rozwiązywania zadań inżynierskich, najważniejszych technik pro-

gramowania oraz streszczono problematykę wykorzystania wzorców projektowych do tworzenia programów komputerowych.

Wszystkie przykłady, w postaci kodu źródłowego, zostały zapisane za pomocą języka Visual Basic. W skrypcie autor wyjaśnia, dlaczego ten język jest „odpowiedni” dla inżynierów-mechaników.