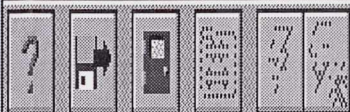


Borland Pascal

File Edit Search Run Compile Tools Options Window Help



Zofia Kruczkiewicz

Biblioteka Główna i OINT
Politechniki Wrocławskiej



100100382552



Metody programowania obiektowego





Politechnika Wrocławska



Zofia Kruczkiewicz

Metody programowania obiektowego

Wrocław 1996

Opiniodawcy

Zbigniew KIERZKOWSKI

Czesław KOŚCIELNY

Opracowanie redakcyjne i korekta

Barbara WACHOWSKA

Biblioteka Główna i OIM
Politechniki Wrocławskiej



00 1330952

© Copyright by Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 1996

OFICyna WYDAWNICZA POLITECHNIKI WROCLAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław



291061/23

ISBN 83-7085-228-9

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Zam. nr 536/96.

Alce. 809/k/96

Wprowadzenie

Idea języków zorientowanych obiektowo znana jest już od połowy lat sześćdziesiątych. Nie cieszyła się zbyt dużą popularnością w praktyce stosowania komputerów. Związane to było z małymi mocami obliczeniowymi maszyn, co uniemożliwiało wykonywanie złożonych odwzorowań i symulacji z wykorzystaniem konstrukcji języków obiektowych.

Obecnie obserwuje się przełom. Przyczyny tego są różne, a przejawiają się w dużym stopniu wzrostem zapotrzebowania na aplikacje powszechnego użytku z licznymi elementami m. in. wizualnego interfejsu (MS_WINDOWS, Motif, OS/2, Mac itd.). Metody programowania obiektowego, obiektowego opisu zasobów, a także obiektowo zorientowanej syntezy systemów są ważną składową nowoczesnej inżynierii informacyjnej.

W tradycyjnym podejściu strukturalnym podstawowym problemem przy konstruowaniu programu jest *wyбір właściwego algorytmu i odpowiednio dobranych struktur danych*. W projektowaniu programu najważniejsze decyzje dotyczą reprezentacji danych. Projektowanie danych powinno stać się wynikiem zrozumienia rozwiązywanego problemu. Istnieje wiele dziedzin problemowych, w których pojedyncza dana jest reprezentowana w postaci struktury zawierającej dane składowe o niejednorodnych wartościach. Te pojedyncze dane o pewnej strukturze zwanej rekordem są z kolei gromadzone w postaci ciągu o pewnym ustawieniu. Mogą być reprezentowane w postaci tablic rekordów, listy z powiązaniem lub pliku itd. Powszechne korzystanie z tych struktur danych powoduje, że programiści dużą wagę przywiązują do przetwarzania. Jednak umożliwienie realizacji projektowanych zadań wymaga użycia konkretnych mechanizmów językowych.

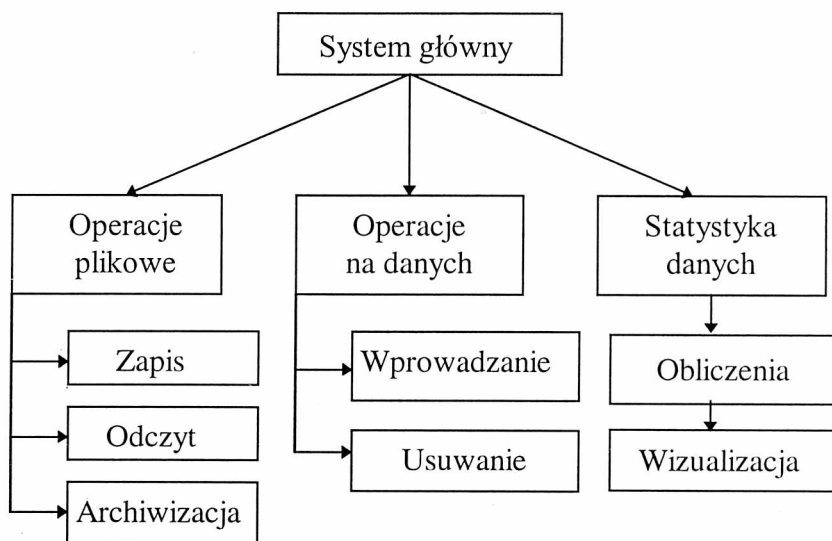
Programowanie *strukturalne* polega na wielopoziomowych podziałach systemu oprogramowania tak, aby każda uzyskana część stanowiła czytelny zapis wykonywanego zadania w postaci konkretnego podprogramu. Projektowanie kolejnych warstw programu można przedstawić w ujęciu *analitycznym*, czyli *zstępującym* lub *syntetycznym*, czyli *wstępującym*. W praktyce jednak są stosowane oba podejścia. W programowaniu analitycznym ([7]) program powstaje podobnie jak piramida budowana od wierzchołka, natomiast programowanie wstępujące przypomina budowę, która zaczyna się od podstawy. Ponieważ programowanie strukturalne jest utożsamiane z programowaniem zstępującym, zostaną przedstawione zasady takiego postępowania.

Algorytm działania jest następujący (na podstawie [7]):

1. W rozwiązywanym problemie poszukuje się takich części, które można byłoby wyrazić w zwięzły sposób za pomocą elementów podstawowych języka programowania. Jeśli udało się osiągnąć taki efekt, należy napisać program dla rozwiązywanego problemu, jeśli nie, należy przejść do kroku 3.
2. Należy rozwiązywany problem przedstawić w postaci części składowych takich, że każda z nich może być rozwiązana za pomocą pojęć podstawowych języka programowania. Jeśli udało się osiągnąć taki efekt, należy napisać program dla rozwiązywanego problemu, jeśli nie, należy przejść do kroku 3.
3. Każdą z części składowych z kroku poprzedniego należy potraktować jako nowy problem i przeprowadzić rozumowanie poczynając od kroku 2. Krok 3 należy tak długo powtarzać, aż osiągnie się taki poziom podziału problemu, który umożliwi napisanie programu.

Pojęciami podstawowymi języka mogą być wyrażenia, instrukcje, moduły. Przy powyższej specyfikacji programu należy pamiętać o zachowaniu zasady niezależności między częściami programu na kolejnych poziomach w celu uzyskania struktury drzewiastej programu. Dowód poprawności uzyskanego rozwiązania w postaci programu można przeprowadzić na poziomie naturalnych pojęć i ich specyfikacji. Przeprowadzenie dowodu polega na wyodrębnieniu wszystkich operacji niezdefiniowanych na poziomie języka i sprawdzenie ich poprawności udowadniając, że z osiągniętego rozwiązania w postaci programu wynika zamierzone rozwiązanie.

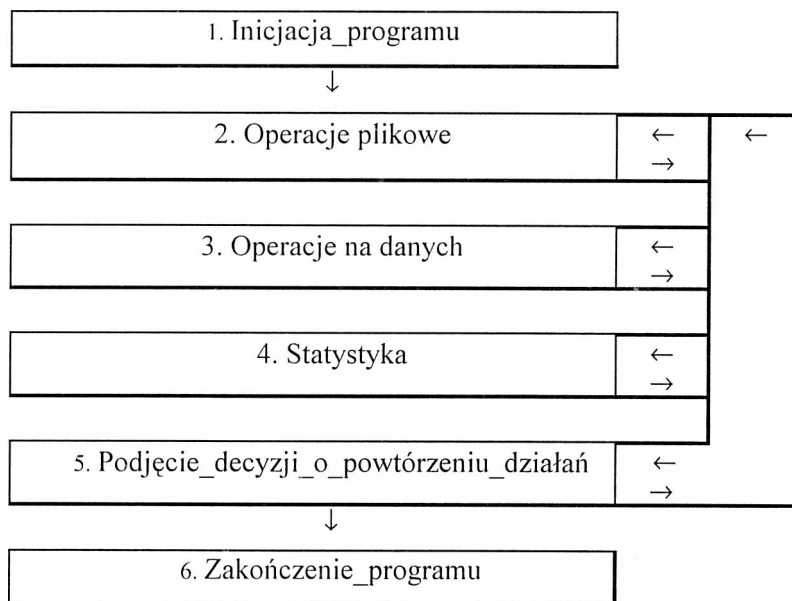
Na rys. 1 pokazano schemat dekompozycji programu.



Rys. 1. Przykład dekompozycji problemu w programowaniu strukturalnym

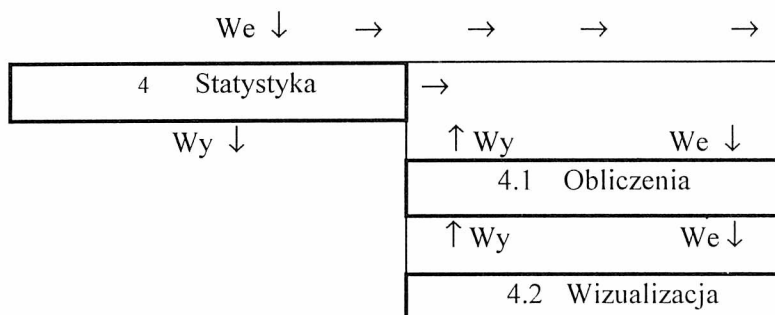
Podział na poziomy wynikał z pogrupowania wszystkich głównych operacji ('Operacje na danych', 'Operacje plikowe', 'Statystyka') i następnie wyodrębnienie w ramach tych grup wyspecjalizowanych czynności (np. 'Zapis', 'Odczyt', 'Archiwizacja'). Schemat takiego programu pokazano na rys. 1.

Na rys. 2 pokazano przebieg wykonania programu strukturalnego. Sterowanie w programie odpowiada schematowi dekompozycji z rys. 1.



Rys. 2. Przykładowy schemat przebiegu programu na pierwszym poziomie zagnieżdżenia

Działania 1, 2, 3, 4, 5, 6 mogą być wykonywane przez podprogramy (procedury lub funkcje). Z kolei każdy z tych podprogramów zawiera inne podprogramy o charakterze bardziej szczegółowym, skonstruowane w oparciu o elementy podstawowe języka (tzn. wyrażenia, instrukcje, moduły). Na przykład podprogram dla działania *Statystyka* mógłby wykorzystywać dwa wewnętrzne podprogramy: (a) *Obliczenia*, (b) *Wizualizacja* na rys. 3.



Rys. 3. Schemat wykonania fragmentu programu na drugim poziomie zagnieżdżenia

Ciągły rozwój wymagań i mocy obliczeniowych sprawił, że algorytm jako sposób specyfikacji programu okazuje się niewystarczający. Jest on odpowiedni jedynie dla niedużych, nawet złożonych programów, realizujących określone, ograniczone funkcje. Nowym sposobem tworzenia programów jest technika obiektowa.

Programowanie zorientowane obiektowo (OOP ang. *Object Oriented Programming*) jest techniką programowania wykorzystującą nowy typ danych zwany typem obiektowym, czyli *klasą*, w którym dane są łączone z podprogramami, zwanymi *metodami*. Programowanie obiektowe umożliwia abstrakcyjne podejście do danych, gdzie istotne są raczej operacje na nich niż ich reprezentacja w pamięci.

Konstruowanie programów upraszcza się, gdyż nowe typy danych można projektować w oparciu o już zdefiniowane klasy, wykorzystując przy tym mechanizmy: *hermetyzacji*, *dziedziczenia* oraz *polimorfizmu*. Pojęcia te zostaną przedstawione w następnych rozdziałach. Przy tworzeniu nowych klas powinien wystarczyć jedynie opis atrybutów oraz wykonywanych na nich operacji. Dzięki temu technika obiektowa realizuje wiele postulatów inżynierii programowania, wynikających ze wzrostu rozmiarów aplikacji oraz współpracy dużej ilości programistów jak i konieczności uwzględnienia cyklu rozwojowego oprogramowania. W programach wykorzystuje się obiekty czyli zmienne typu obiektowego.

Program obiektowy składa się z następujących części:

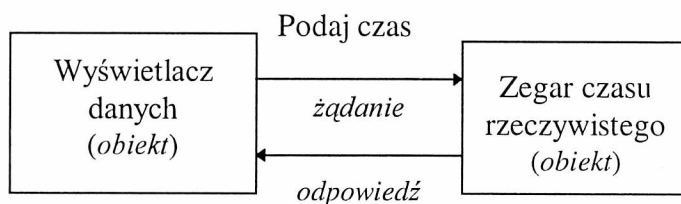
- inicjującej obiekty za pomocą operacji przypisanych danej klasie;
- głównej, składającej się z wywołań przez obiekty poszczególnych operacji działających na ich wewnętrznej reprezentacji. Sterowanie w programach obiektowych odbywa się poprzez wymianę informacji między obiektami za pośrednictwem operacji, dedykowanym konkretnym czynnościom i zwanymi

metodami. Oznacza to, że zamiast dekompozycji funkcjonalnej stosowanej w programowaniu strukturalnym, uruchamiane są obiekty wyspecjalizowane w konkretnych czynnościach i odpowiednio porozumiewają się między sobą;

- kończącej działanie programu, tzn. usuwającej obiekty i niepożądane skutki ich działania w systemie.

Schemat takiej interakcji pokazano na rys. 4., natomiast schemat porozumiewania się między obiektami zaprojektowanymi dla konkretnego problemu pokazano na rys. 6 i rys. 7.

Programy obiektowe są zapisem specyfikacji problemu z poziomu analizy i projektowania. Oznacza to, że aby napisać program obiektowo należy prawidłowo dokonać analizy obiektowej problemu (OOA ang. *Object Oriented Analysis*) i przełożyć tę analizę w fazie projektowania (OOD ang. *Object Oriented Design*) na program uwzględniając specyficzne cechy zastosowanego języka.



Rys. 4. Schemat porozumiewania się między obiektami

Celem tego opracowania jest przedstawienie pewnych elementów programowania obiektowego w języku Turbo Pascal 7.0 na przykładzie rozwiązania problemu dotyczącego sporządzania rachunków. Wybór problemu z dziedziny znanej z życia codziennego może pomóc w zrozumieniu przeprowadzonej analizy obiektowej i projektowania. Technika obiektowa powstała na gruncie komputerowej symulacji rzeczywistych procesów, toteż cały proces tworzenia programu opiera się na „naśladowaniu” czynności przy tworzeniu kolejnych elementów rachunku. W celu uproszczenia tych czynności wybrano taki problem, aby jego model nie wymagał skomplikowanego opisu formalnego. Dzięki temu można czytelnie zaprezentować możliwości techniki obiektowej.

Język Turbo Pascal 7.0 nie stanowi narzędzia sprzyjającego czystemu programowaniu obiektowemu. Powstał on na gruncie łączenia konwencjonalnych cech języka z cechami obiektowymi. W przykładach starano się dbać o zachowanie

zasad programowania obiektowego, jednak miejscami wykorzystano „hybrydowość” języka starając się nie nadużywać tych możliwości.

W dalszych rozdziałach opisano kolejne etapy projektowania i programowania obiektowego, przedstawiając sposób korzystania z różnych dostępnych środków tzn.:

- dziedziczenia metod, hermetyzacji, polimorfizmu,
- abstrakcyjnych typów danych.

Szczególnie istotne jest tworzenie klas w czasie analizy obiektowej, odzwierciedlające rozwiązywany problem i uzupełnianie klas podczas projektowania i programowania w celu zastosowania wzorcowych struktur danych typu:

- struktury dynamiczne, kolekcje (tablice wskaźników obiektów),
- strumienie i zasoby.

1. Przygotowanie programu

Obiektowe rozwiązanie problemu powinno składać się z następujących procesów:

- analizy obiektowej - OOA (ang. *Object-Oriented Analysis*),
- projektowania obiektowego - OOD (ang. *Object-Oriented Design*),
- programowania obiektowego - OOP (ang. *Object-Oriented Programming*).

Etapy OOA i OOD przedstawiono bardzo skrótowo, sygnalizując najważniejsze ich elementy na przykładzie rozwiązania problemu sporządzania rachunków.

1.1. Analiza obiektowa problemu

Analiza obiektowa jest niezależna od języka programowania i jest nowym sposobem specyfikacji rozwiązywanego problemu. W czasie omawiania tej fazy tworzenia programu zostały wykorzystane pewne pojęcia np.: *klasa*, *obiekty*, *metody*, *dziedziczenie*, *drzewa dziedziczenia* itp. Definicje tych pojęć zostaną przedstawione w dalszej części opracowania.

W analizie problemu można wyróżnić pięć następujących etapów:

- wybór klas;
- znajdowanie drzew dziedziczenia oraz klas z obiektami zagnieżdżonymi;
- definiowanie pól (atrybutów) z uwzględnieniem powiązań strukturalnych między obiektami;
- definiowanie usług (metod) z uwzględnieniem powiązań funkcjonalnych między obiektami;
- znajdowanie klas o uniwersalnym znaczeniu w rozwiązywanym problemie.

Przebieg analizy zaprezentowano na prostym przykładzie dotyczącym sporządzania rachunków. Po zrealizowaniu kolejnych etapów uzyskuje się schemat zależności strukturalnych i funkcjonalnych pomiędzy klasami i obiektami. Daje on obraz przebiegu porozumiewania się między obiektami, jaki można wykorzystać w przyszłym programie.

Wybór głównych klas

Wybrane klasy tworzą model danych w rozwiązywanym problemie. Wybór ich polega na poszukiwaniu opisu elementów tworzących cały model. Opis ten zawiera charakterystykę danych wraz z narzędziami do jej przetwarzania oraz charakterystykę narzędzi do wymiany informacji z innymi elementami modelu.

Poszukiwany opis może dotyczyć: struktur, systemów, urządzeń, rzeczy, zdarzeń, miejsc, jednostek organizacyjnych itp.

Klasa definiuje więc określone atrybuty i operacje i umożliwia powielanie identycznych elementów,

Obiekt jest programowym „pakunkiem”, stanowiącym realizację klasy, czyli jest jej „konkretem”.

W terminach języków programowania klasa oznacza typ danych obiektowych, natomiast obiekt jest zmienną obiektową.

W wyniku obiektowej analizy problemu sporządzania rachunków wytypowano trzy główne klasy, kierując się potocznym i umownym *opisem problemu* tzn.:

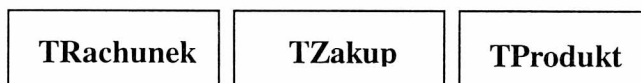
1. *Rachunek* zawiera elementy określane jako *Zakup* towaru oraz zna ich ilość.
2. Część *Rachunków* ma dodatkową informację o wyborze płatności (płatny przelewem lub gotówką).
3. *Rachunek* potrafi wykonać następujące operacje:
 - wstawiać nowy *Zakup* jedynie wtedy, kiedy zawiera on zupełnie nowy *Towar*. Jeżeli któryś z istniejących *Zakupów* ma ten sam *Towar*, w takim przypadku należy jedynie powiększyć w nim liczbę zakupionego już *Towaru*;
 - usuwać wybrany *Zakup*;
 - wyliczyć swoją wartość brutto itp..
4. *Zakup* składa się z informacji o *Towarze*, jego ilości oraz numeru zakupu.
5. *Zakup* potrafi wykonać następujące operacje:
 - wstawiać zakupiony *Towar*, jego ilość oraz numer dokonanego zakupu;
 - obliczyć swoją wartość netto (liczba towaru * cena towaru);
 - obliczyć swój podatek (wartość netto * podatek);
 - sprawdzić inny *Zakup* czy nie zawiera tego samego *Towaru*.
6. *Towar* ma podstawowe dane zawierające informację o nazwie, cenie oraz:
 - część *Towarów* ma dodatkowe informacje o gatunku,
 - część *Towarów* ma dodatkowe informacje o podatku.
7. *Towar* potrafi wykonywać następujące czynności:
 - porównać swoje dane z danymi innego *Towaru*,
 - wyliczyć swoją cenę w oparciu o gatunek (cena * 0.8, jeśli *gatunek*=2 itp).

Podsumowanie opisu **klas**:

- Informacje przypisane klasie stanowią jej *atrybuty* i są przechowywane w polach.
- Operacje lub czynności, podane w opisie problemu, mają charakter algorytmicznie złożony czyli realizują główny cel, do którego przeznaczono klasę. Są to *metody problemu*.

- Każda klasa na żądanie innego obiektu podaje informację o wartości swoich atrybutów lub potrafi ustawiać wartości swoich atrybutów. Jest to grupa *metod dostępu do pól*. Jedynie za ich pośrednictwem należy żądać dostępu do pól obiektu.

Na rys. 5. przedstawiono wytypowane główne klasy w wyniku wstępnej analizy problemu. Nazwy mają charakter mnemotechniczny.



Rys. 5. Klasy główne

Znajdowanie drzew dziedziczenia oraz klas z obiektami zagnieżdżonymi

W tej fazie analizy obiektowej (OOA) należy poszukać powiązań strukturalnych pomiędzy wytypowanymi klasami głównymi oraz wyodrębnić klasy różniące się pewnymi cechami, tzn. stanowiące kolejne specjalizacje.

Każda *klasa-specjalizacja* dziedziczy atrybuty oraz operacje po swoim poprzedniku (klasie bazowej), przy czym:

- atrybuty dziedziczy wszystkie i uzupełnia własnymi,
- operacje dziedziczone może zmieniać (zakrywać) oraz tworzyć nowe.

Poszukiwanie specjalizacji prowadzi do wyszukania tzw. drzew dziedziczenia, natomiast ustalenie struktur typu część-całość typuje klasy z obiektami zagnieżdżonymi. Klasy te korzystają z usług obiektów, z którymi są związane strukturalnie.

Na podstawie opisu problemu można wyodrębnić trzy rodzaje produktów, mających różne zestawy cech (punkt 6 w rozdz. 1.1). Ponieważ niektóre cechy się powtarzają (*nazwa i cena*), natomiast inne atrybuty rozszerzają zestaw cech nowego produktu tzn. *nazwa, cena, gatunek* lub *nazwa, cena, podatek* lub *nazwa, cena, gatunek, podatek* - stąd każda z poszerzonych klas korzysta przez dziedziczenie z atrybutów i operacji działających na nich od poprzednika. Część operacji przybywa przy rozszerzaniu liczby atrybutów np. obsługa podatku lub gatunku (punkt 7 w rozdz. 1.1). Inne metody ulegają modyfikacji np. ustalanie ceny w zależności od gatunku. Taki projekt klas prowadzi do skonstruowania drzewa dziedziczenia typu *TProdukt..* (rys. 6). Wyodrębniono również drugie drzewo *TRachunek..* na podstawie opisu rachunków. Podstawowy rachunek ma nazwę oraz liczbę zakupów, natomiast

poszerzony rachunek dodatkowo jest opisany atrybutem określającym rodzaj płatności. Stąd klasa, opisująca rachunek o dodatkowym atrybucie, może dziedziczyć operacje od klasy podstawowej (bazowej) uzupełniając je o operację informującą o rodzaju płatności.

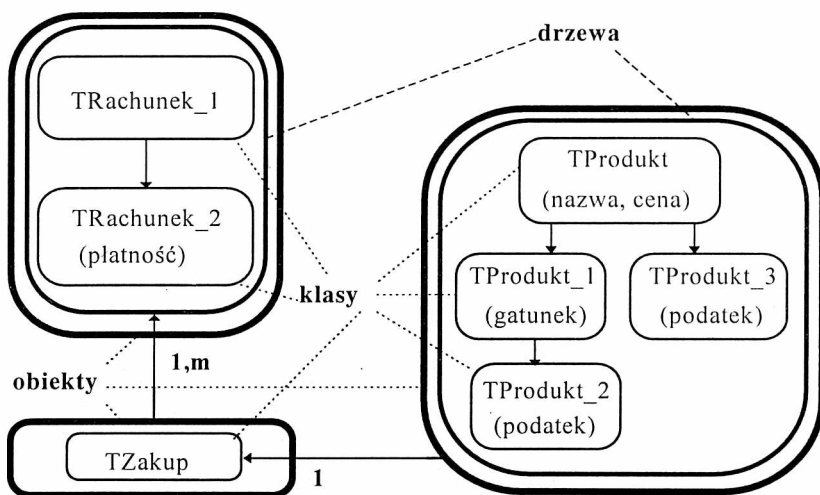
Na podstawie tych ustaleń, w tej fazie analizy obiektowej (OOA), wytypowano następujące klasy-specjalizacje, stanowiące drzewa klas:

- dla towarów (*TProdukt*, *TProdukt_1*, *TProdukt_2*, *TProdukt_3*)
- oraz rachunków (*TRachunek_1* oraz *TRachunek_2*).

Na podstawie opisu problemu, przedstawionego w rozdziale 1.1 (punkty 1-7) wytypowano również klasy z obiektami zagnieżdżonymi. W punkcie 1 opisano zależność strukturalną między klasami z drzewa *TRachunek*.. a obiektami klasy *TZakup*; podobnie w punkcie 4 określono zależność między klasą *TZakup* a obiektami z drzewa *TProdukt*...Takie związki wskazują na możliwość utworzenia klasy, która bezpośrednio współpracuje z pewną liczbą obiektów, jednocześnie zarządzając nimi w kontaktach z innymi klasami. Oznacza to możliwość odwołania do każdego z nich za pośrednictwem klasy zarządzającej. Tak sformułowane zależności spowodowały następujący wybór klas z obiektami zagnieżdżonymi :

- klasę *TZakup* z jednym obiektem z drzewa klas *TProdukt*...
- klasy z drzewa rachunków (*TRachunek*...) zawierające 1..m obiektów *TZakup*.

Schemat takich podziałów dla rozwiązywanego problemu sporządzania rachunków przedstawiono na rys. 6.



Rys. 6. Schemat drzew klas oraz struktur całość-część

Obramowania wykonane grubszą linią symbolizują obiekty, natomiast cieńsze ramki oznaczają drzewa klas, a najcieńsze klasy. Kierunek strzałki wyznacza przynależność poszczególnych obiektów zagnieżdżonych do klas.

Wybór atrybutów, czyli pól w klasach z uwzględnieniem powiązań strukturalnych

W tej fazie analizy obiektowej (OOA) określono szczegółowo atrybuty występujące w poszczególnych klasach bez następców oraz tworzących drzewo dziedziczenia.

Zastosowano przy tym następujące zasady:

1. Ogólniejsze atrybuty wstawiano do klasy wyższej w drzewie.
2. Wyspecjalizowane atrybuty wstawiano niżej w drzewie.
3. Atrybuty w danej klasie, nie będące obiektami zagnieżdżonymi, powinny określać pojęcia 'końcowe'.
4. Atrybuty powinny zawierać pełną informację o wyodrębnionej klasie jako całości funkcjonalnej i strukturalnej.
5. Wybór atrybutów realizujących całość-część, tzn. zagnieżdżenia obiektów powinien odzwierciedlać powiązania strukturalne wynikające z przyjętego modelu danych w rozwiązywanym problemie.
6. Atrybuty powinny zawierać jedynie informację, której nie można podać na żądanie, w przeciwnym wypadku zamiast atrybutu należy zdefiniować odpowiednią metodę.

W wyniku tych ustaleń określono następujące atrybuty w poszczególnych klasach:

- w klasie *TProdukt* : nazwa, cena,
- w klasie *TProdukt_1*: nazwa, cena, gatunek,
- w klasie *TProdukt_2*: nazwa, cena, gatunek, podatek,
- w klasie *TProdukt_3*: nazwa, cena, podatek,
- w klasie *TZakup* : produkt (jeden obiekt z drzewa *TProdukt.*), liczba produktu, numer pozycji,
- w klasie *TRachunek_1*: zakupy(zbiór obiektów *TZakup*), liczba zakupów,
- w klasie *TRachunek_2*: zakupy(zbiór obiektów *TZakup*), liczba zakupów, rodzaj płatności.

Określenie usług, czyli metod w klasach z uwzględnieniem powiązań funkcjonalnych

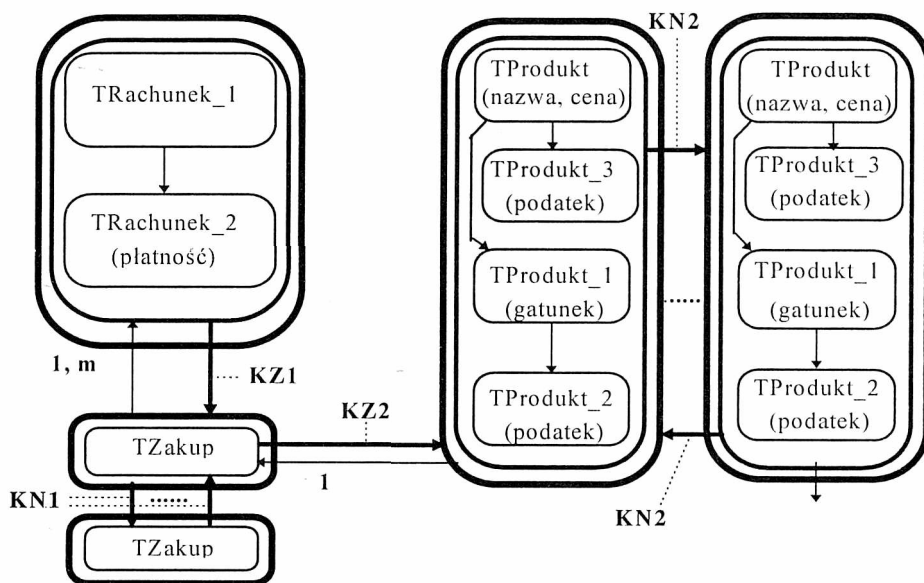
W tej fazie analizy obiektowej (OOA) ustalono następujące rodzaje metod w każdej klasie:

1. metody inicjujące i likwidujące obiekt,

2. metody obsługujące pola ("ustaw" oraz "podaj"), czyli metody dostępu do pól,
3. metody algorytmicznie złożone (wykonujące zadanie klasy), czyli metody problemu,
4. metody wykorzystujące powiązania z innymi obiektami (aby za ich pośrednictwem wykonać swoje zadanie), przy czym te metody mogą realizować jedną z wyżej wymienionych metod. Powiązania między obiektami mogą mieć charakter strukturalny (obiekty zagnieżdżone) lub komunikacja może odbywać się między obiektami niezależnymi.

Metody dostępu do pól wyznacza się na podstawie atrybutów: ich liczby i typu przechowywanej informacji. Oprócz pewnych wyjątków każdy atrybut wymaga dwóch metod: odbierających dane oraz przekazujących kopię swoich danych.

Na rys. 7 przedstawiono powiązania strukturalne i funkcjonalne między klasami i obiektami.



Rys. 7. Schemat powiązań strukturalnych i funkcjonalnych między klasami i obiektami

Połączenia oznaczone literą *KZ* i numerem określają kierunek żądań, skierowanych od klasy-właściciela do obiektów zagnieżdżonych.

Wyznaczają one metody klasy realizujące jej zadanie przy wywołaniu metod obiektów zagnieżdżonych.

Połączenia oznaczone literą *KN* i numerem określają kierunek żądań, skierowanych od obiektu do innego obiektu. Wyznaczają one metody klasy realizujące swoje zadanie przy wywołaniu metod innych obiektów niezależnych.

Przedstawiony schemat na rys. 7 jest jednocześnie obrazem sterowania przebiegiem porozumiewania się obiektów, jaki można wykorzystać w programie.

Znajdowanie klas o kluczowym znaczeniu w rozwiązaniu problemu

W czasie analizy obiektowej należy wyodrębniać klasy wielokrotnego użytku. Oznacza to uwzględnianie w czasie opracowania klas pewnych uniwersalnych rozwiązań, które można byłoby wykorzystać w innych problemach.

Podczas analizy stwierdzono, że można wielokrotnie wykorzystać drzewo produktów *TProdukt..*, rachunków *TRachunek..* lub klasę *TZakup*, ponieważ w życiu codziennym istnieje wiele dziedzin, gdzie można zastosować te klasy np.: stany magazynowe, finanse przedsiębiorstw itp..

Przedstawione wyniki analizy wykorzystano do projektu i programowania poszczególnych klas problemu i klas pomocniczych.

1.2. Projektowanie obiektowe

Projektowanie obiektowe jest uzależnione od języka i środowiska oraz istniejących zasobów programowych. Przebieg projektowania można podzielić na dwa etapy:

1. projekt części problemowej - ściśle korzystanie z wyników analizy;
2. projekt części dotyczącej kontaktu z użytkownikiem;

Punkt 1 dotyczy fazy projektowania poszczególnych klas, dobierając reprezentację w pamięci poszczególnych atrybutów oraz specyfikację każdej z metod. Projektowanie należy rozpocząć od obiektów najprostszych ze schematu funkcjonalno-strukturalnego.

W tej fazie projektowania często wyodrębnia się dodatkowe klasy o charakterze struktur danych np.:

- kolekcje,
- struktury dynamiczne,
- strumienie,
- zasoby itp.

Przykłady wyodrębniania takich klas oraz przesłanki, jakimi należy się kierować przy ich projektowaniu przedstawiono w dalszych rozdziałach.

Punkt 2 dotyczy projektowania platformy (interfejsu) kontaktu z użytkownikiem, przy czym należy uwzględnić możliwość wykorzystania gotowych bibliotek.

W następnych rozdziałach przedstawiono kolejne etapy projektu. Każdy z nich kończy się wykonaniem programu, realizującym stan bieżący. Jednocześnie te same wyniki analizy obiektowej zostały wykorzystane do kolejnych wersji projektów, prezentowanych w kolejnych podrozdziałach. Spowodowało to opracowanie kilku wersji programów wykorzystujących elementy programowania obiektowego, tzn.: hermetyzację i dziedziczenie statyczne oraz różne sposoby zastosowania polimorfizmu.

Celem takiego działania było przedstawienie różnorodnych aspektów programowania obiektowego i możliwość doboru właściwych dla rozwiązywanego problemu środków programowych.

2. Hermetyzacja danych, dziedziczenie metod statycznych

Kolejne podrozdziały są realizacją projektu poszczególnych klas przedstawionych na rys. 3. wykorzystując podstawowe elementy programowania obiektowego: hermetyzację oraz dziedziczenie metod statycznych.

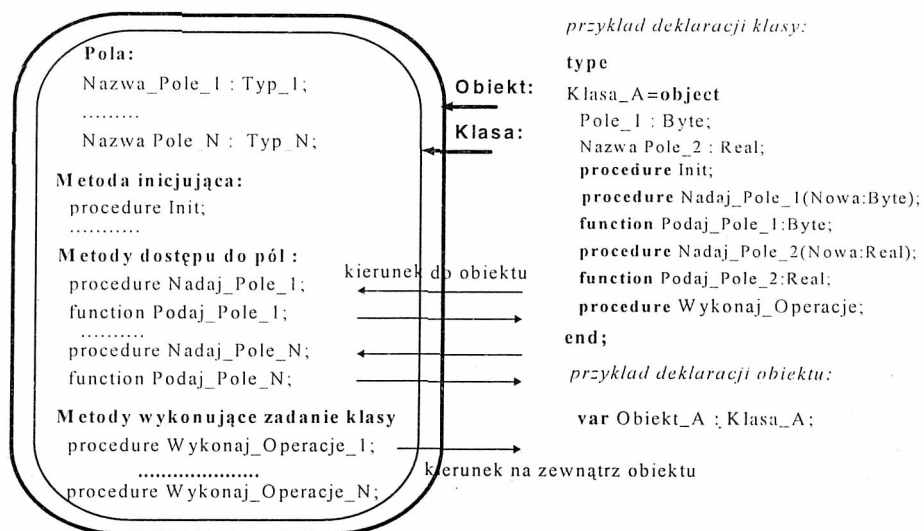
2.1. Podstawowe zagadnienia

Programowanie obiektowe opiera się na typie obiektowym, zdefiniowanym przez użytkownika. Typ obiektowy, czyli *klasa*, wiąże dane oraz funkcje i procedury, czyli tzw. metody. Zmienne obiektowe, zwane *obiektami*, tego samego typu należą do tej samej klasy.

Oznacza to, że:

- mają identyczne pola oraz te same metody,
- znajdują się w różnych obszarach pamięci.

Na rys. 8 pokazano graficzną postać definicji klasy i obiektu. Deklaracja klasy przypomina deklarację typu **record**, tzn. tak samo deklaruje się pola i dodatkowo są wyszczególniane metody przez umieszczenie wykazu ich nagłówków typu **procedure** lub **function**. Zamiast słowa kluczowego **record** używa się **object**.



Rys. 8. Postać graficzna i formalna deklaracji klasy i obiektu

Poniżej przedstawiono moduł **UCW1_1** zawierający deklarację klasy *TProdukt* oraz definicje jej metod. Klasa reprezentuje produkt handlowy. Program **CW1_1** wykorzystuje tę klasę poprzez zadeklarowanie obiektu *Produkt_1* i wywoływanie jego metod. Nagłówek definiowanej metody składa się z nazwy klasy oddzielonej kropką od nazwy metody. W obrębie klasy nazwy parametrów formalnych lub lokalnych w metodach muszą różnić się od nazw pól.

```

unit UCW1_1;
interface
type
    Lan_1=string[20];
const
    Max_Produkt=3;    Max_Ceny =3;
    Nazwy:array[1..Max_Produkt]of Lan_1=
        ('Produkt_A', 'Produkt_B', 'Produkt_C');
    Ceny :array [1..Max_Ceny] of Integer = (25, 22, 20);

type
    TProdukt = object
        private
            Nazwa : Lan_1;
            Cena : Real;
        public procedure Inicjuj;

        procedure Nadaj_Nazwe (Nowa_Nazwa:Lan_1);
        function Podaj_Nazwe : Lan_1;
        procedure Nadaj_Cene (Nowa_Cena : Real);
        function Podaj_Cene : Real;
        procedure Wydruk_Produktu;

end;

implementation

```

```
procedure TProdukt.Inicjuj;
begin
  Nazwa := '';
  Cena := 0;
end;

procedure TProdukt.Nadaj_Nazwe (Nowa_Nazwa : Lan_1);
begin
  Nazwa := Nowa_Nazwa;
end;

function TProdukt.Podaj_Nazwe : Lan_1;
begin
  Podaj_Nazwe := Nazwa;
end;

procedure TProdukt.Nadaj_Cene (Nowa_Cena : Real);
begin
  Cena := Nowa_Cena;
end;

function TProdukt.Podaj_Cene : Real;
begin
  Podaj_Cene := Cena;
end;

procedure TProdukt.Wydruk_Produktu;
begin
  Write(#13#10,'Nazwa: ',Nazwa:20,' Cena_netto: ',Cena:8:2);
end;
end.                                     {koniec modułu UCW1_1}
```

Wywołanie metody składa się z nazwy obiektu, kropki i nazwy metody (linie 1).

```
program CW1_1;
uses UCW1_1;
var Produkt_1 : TProdukt;
begin
  Produkt_1.Inicjuj
  Produkt_1.Wydruk_Produktu; Readln;
  Produkt_1.Nadaj_Nazwe(Nazwy[1]);      {prawidłowa obsługa pól}{1}
```

```

Produkt_1.Nadaj_Cene(Ceny[1]); {1}
Produkt_1.Wydruk_Produktu; Readln;
    {lub zastosowanie do wydruku metod pobierających wartości pól -
    podejście zapewniające swobodę w wykorzystaniu obiektu}
Writeln('Nazwa: ', Produkt_1.Podaj_Nazwe:20,
    ' Cena: ', Produkt_1.Podaj_Cene: 8:2); {1}
{Nieprawidłowa obsługa pól - uniemożliwiona dyrektywą private
    Produkt_1.Nazwa := Nazwy[1];
    Produkt_1.Cena := Ceny[1];}
end. {koniec programu CW1_1}

```

Zachowanie zasady *hermetyzacji* danych umożliwia dokonanie dowolnych zmian w definicji pól bez potrzeby dokonywania zmian w pozostałych częściach modułu oraz programu, o ile nie zmieniają się nagłówki metod. Przedstawiony niżej moduł **UCW1_2** nie wymaga dokonania żadnych zmian w programie **CW1_2** (kopia programu **CW1_1**), mimo że pola są wskaźnikami. Oczywiście, w wyszczególnieniu modułów należało zmienić jedynie nazwę modułu na **UCW1_2**.

Przykład 2.

```

unit UCW1_2;
interface
type
    Lan_1 = string[20];
const
    Max_Produkt = 3; Max_Ceny = 3;
    Nazwy:array [1..Max_Produkt] of Lan_1=
        ('Produkt_A', 'Produkt_B', 'Produkt_C',);
    Ceny : array [1..Max_Ceny] of Integer = (25, 22, 20);
type
    Produkt = object
        private Nazwa : ^Lan_1;
                Cena : ^Real;
        public procedure Inicjuj;
        procedure Nadaj_Nazwe (Nowa_Nazwa : Lan_1); {metody dostępu do pól}
        function Podaj_Nazwe : Lan_1;
        procedure Nadaj_Cene (Nowa_Cena : Real);
        function Podaj_Cene : Real;
        procedure Wydruk_Produktu; {metoda dodatkowa}
end;
implementation

```

```
procedure TProdukt.Inicjuj;
begin
  if MaxAvail < SizeOf(Nazwa^) then Halt;
  New(Nazwa); Nazwa^ := '';
  if MaxAvail < SizeOf(Cena^) then Halt;
  New(Cena); Cena^ := 0;
end;

procedure TProdukt.Nadaj_Nazwe (Nowa_Nazwa : Lan_1);
begin
  Nazwa^ := Nowa_Nazwa;
end;

function TProdukt.Podaj_Nazwe : Lan_1;
begin
  Podaj_Nazwe := Nazwa^;
end;

procedure TProdukt.Nadaj_Cene (Nowa_Cena : Real);
begin
  Cena^ := Nowa_Cena;
end;

function TProdukt.Podaj_Cene : Real;
begin
  Podaj_Cene := Cena^;
end;

procedure TProdukt.Wydruk_Produktu;
begin
  Write(#13#10, 'Nazwa: ', Nazwa^:20, 'Cena_netto: ', Cena^:8:2);
end;
end. {koniec modulu UCWI_2}
```

Drugim mechanizmem, wykorzystywanym w programowaniu obiektowym, jest **dziedziczenie metod i pól** w klasach zdefiniowanych jako następcy klasy podstawowej, tzn. bazowej (tab. 1). Schemat powiązań między klasami, wynikającymi z dziedziczenia, będziemy nazywać *drzewem dziedziczenia*. Klasy dziedziczące będziemy nazywać *specjalizacjami* lub *następcami* klasy będącej na szczycie drzewa, którą z kolei będziemy nazywać *korzeniem klas*.

Podprogramy, czyli procedury i funkcje, zadeklarowane w danej klasie bez dodatkowych dyrektyw języka np. **virtual**, nazywamy *statycznymi*. Metody te, wywoływane w programie, w czasie kompilacji mają już ustalony konkretny adres. Kompilator dobiera adres metody na podstawie tzw. drzewa dziedziczenia, tworzonego w czasie deklaracji klas, tzn.: najpierw szuka metody w klasie wywołującej, jeżeli tam nie znajdzie, poszukuje jej u kolejnych poprzedników aż do korzenia. W przypadku, kiedy nie znajdzie metody, zgłasza błąd.

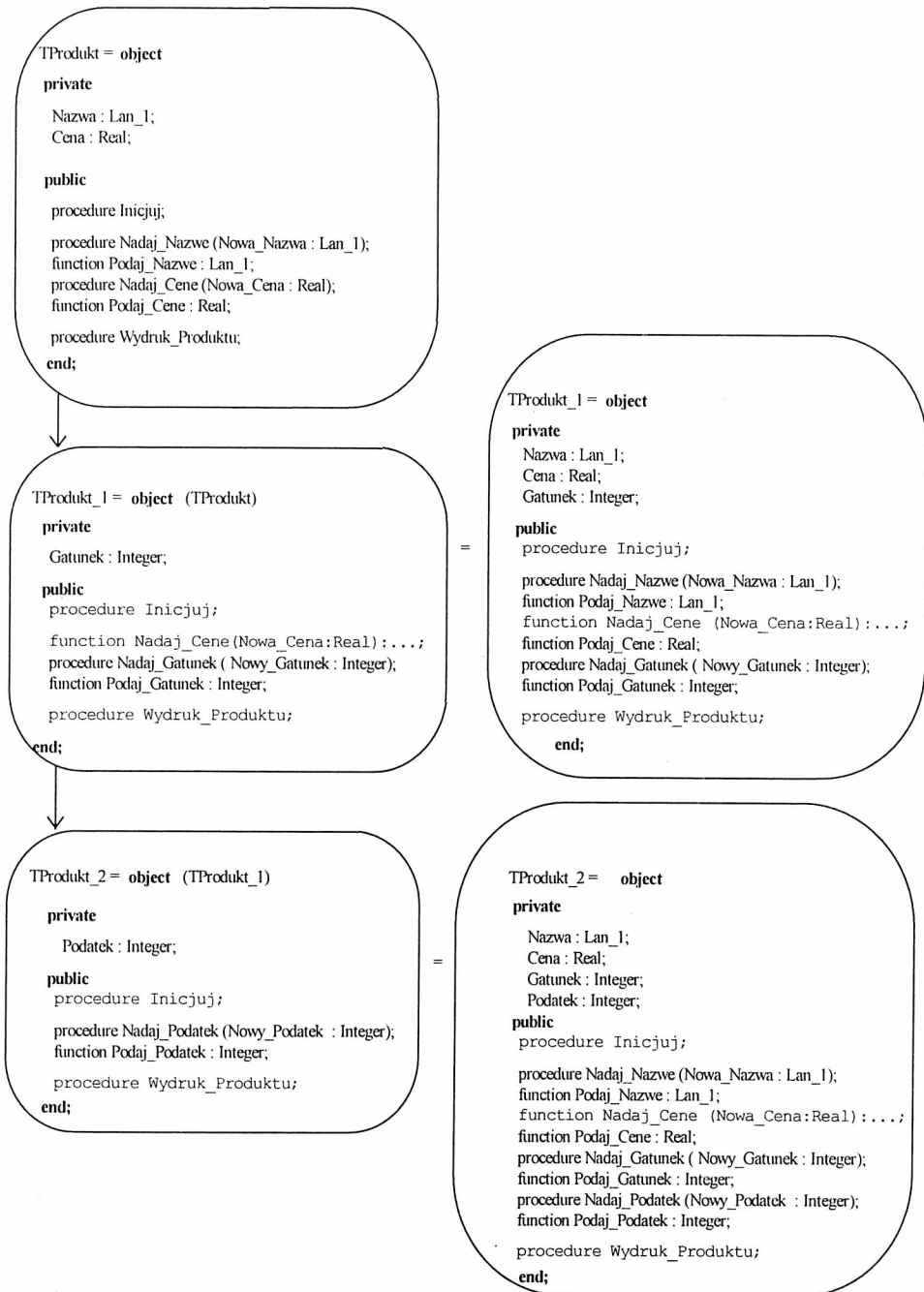
Na rys. 9 pokazano drzewo klas, które dziedziczą pola i metody po swoich poprzednikach. Z prawej strony przedstawiono schemat deklaracji faktycznie posiadanych pól i metod przez obiekty *TProdukt_1* i *TProdukt_2*.

Metody *Inicjuj*, *Nadaj_Cene* oraz *Wydruk_Projektu* zostały zakryte w klasie *TProdukt_1* przez inne metody, w których uwzględniono nowe pole *Gatunek*. W metodzie *Nadaj_Cene* uwzględniono inny scenariusz naliczenia ceny po wprowadzeniu pola *Gatunek*. Dodatkowo przy zakrywaniu metod zmieniono nagłówki metody *Nadaj_Cene* z procedury na funkcję (możliwa jest również zmiana listy parametrów formalnych). W deklaracji klasy *TProdukt_1* doszły dwie metody: *Nadaj_Gatunek* oraz *Podaj_Gatunek*. W klasie *TProdukt_2* zadeklarowano nowe pole *Podatek*, stąd zostały przykryte metody *Inicjuj* oraz *Wydruk_Projektu* oraz dołożono metody *Nadaj_Podatek* i *Podaj_Podatek*.

W tabeli 1 przedstawiono efekty rozbudowy kolejnych klas.

Klasa	<i>TProdukt</i>	<i>TProdukt_1</i>	<i>TProdukt_2</i>
Nowe pole	<i>Nazwa, Cena</i>	<i>Gatunek</i>	<i>Podatek</i>
Metody zakryte		<i>Inicjuj</i> , <i>Wydruk_Projektu</i>	<i>Inicjuj</i> , <i>Wydruk_Projektu</i>
Metody zakryte ze zmianą nagłówka		<i>Nadaj_Cene</i>	
Metody nowe	<i>Inicjuj</i> <i>Nadaj_Nazwe</i> , <i>Podaj_Nazwe</i> , <i>Nadaj_Cene</i> , <i>Podaj_Cene</i> , <i>Wydruk_Projektu</i>	<i>Nadaj_Gatunek</i> <i>Podaj_Gatunek</i>	<i>Nadaj_Podatek</i> <i>Podaj_Podatek</i>
Typ klasy	korzeń drzewa podstawowa czyli bazowa dla klasy <i>TProdukt_1</i>	podstawowa czyli bazowa dla klasy <i>TProdukt_2</i>	

Tabela 1. Dziedziczenie w drzewie *TProdukt...*

Rys. 9. Drzewo dziedziczenia obiektów *TProdukt*, *TProdukt_1* oraz *TProdukt_2*

W tekście metod wykorzystano słowo kluczowe **inherited** oznaczające, że pobrano metodę od bezpośredniego poprzednika w drzewie dziedziczenia (np. metoda *Inicjuj*). Użycie tego słowa lub nazwy klasy (*Nazwa_Klasy.Nazwa_Metody*), z której jest pobrana metoda (z dowolnej klasy idąc w kierunku korzenia) jest konieczne wtedy, gdy dana metoda jest *zakryta* w klasie wywołującej. W innych przypadkach wskazujemy jedynie, od którego poprzednika należy pobrać metodę, jeżeli metody są *zakryte* w klasach poprzedników (tzw. *dziedziczenie z przeskokiem*).

W przykładzie 3 pokazano, jakie zmiany należy wykonać w części publicznej modułu oraz implementacyjnej modułu **UCW1_1** przy rozbudowie drzewa *TProdukt...*, aby wykonać moduł **UCW1_3**. Deklaracje klasy *TProdukt* rozbudowano o metodę *Porownaj_Towary*, która zostanie omówiona w dalszej części tego rozdziału.

Przykład 3.

```
unit UCW1_3;
interface
    {deklaracje z interfejsu modułu UCW1_1}

type TProdukt= object
    {deklaracje z modułu UCW1_1}

    function Porownaj_Towary(Towar:TProdukt):Boolean;
end;

TProdukt_1 =object (TProdukt)
    private Gatunek : Integer;
    public procedure Inicjuj;
    {metody dostępu do pól}

    procedure Nadaj_Gatunek(Nowy_Gatunek:Integer);
    function Podaj_Gatunek : Integer;
    function Nadaj_Cena(Nowa_Cena : Real) : Boolean;
    procedure Wydruk_Produktu;
    {metoda dodatkowa}
end;

TProdukt_2 = object (TProdukt_1)
    private Podatek : Integer;
    public procedure Inicjuj;
    {metody dostępu do pól}

    procedure Nadaj_Podatek(Nowy_Podatek:Integer);
    function Podaj_Podatek : Integer;
    procedure Wydruk_Produktu;
    {metoda dodatkowa}
end;
```

implementation*{pozostale definicje metod klasy TProdukt z modulu UCW1_1}*

```
function TProdukt.Porownaj_Towary(Towar:TProdukt):Boolean;
begin
    Porownaj_Towary := (Nazwa=Towar.Podaj_Nazwe)
                        and (Cena = Towar.Podaj_Cene);
end;
{-----TProdukt_1-----}
procedure TProdukt_1.Inicjuj;
begin
    inherited Inicjuj;                                {lub TProdukt.Inicjuj;}
    Gatunek := 1;
end;

procedure TProdukt_1.Nadaj_Gatunek(Nowyy_Gatunek:Integer);
begin
    Gatunek := Nowyy_Gatunek;
end;

function TProdukt_1.Podaj_Gatunek : Integer;
begin
    Podaj_Gatunek := Gatunek;
end;

function TProdukt_1.Nadaj_Cene(Nowa_Cena:Real):Boolean;
begin
    inherited Nadaj_Cene(Nowa_Cena);                    {lub TProdukt.Nadaj_Cene;}
    Nadaj_Cene := True;
    case Gatunek of
        1 : ;
        2 : Cena := Cena * 0.8;
        3 : Cena := Cena * 0.5;
    else Nadaj_Cene := False;
    end;
end;

procedure TProdukt_1.Wydruk_Produktu;
begin
    inherited Wydruk_Produktu;                            {lub TProdukt.Wydruk_Produktu}
    Write(' Gatunek: ' : 11, Gatunek : 2);
end;
```

```

{-----TProdukt_2-----}
procedure TProdukt_2.Inicjuj;
begin
    inherited Inicjuj;                                {lub TProdukt_1.Inicjuj;}
    Podatek := 1;
end;

procedure TProdukt_2.Nadaj_Podatek (Nowy_Podatek: Integer);
begin
    Podatek := Nowy_Podatek;
end;

function TProdukt_2.Podaj_Podatek : Integer;
begin
    Podaj_Podatek := Podatek;
end;

procedure TProdukt_2.Wydruk_Produktu;
begin
    inherited Wydruk_Produktu;                        {lub TProdukt_1.Wydruk_Produktu}
    Write(' Podatek: ' : 11, Podatek : 2, '%');
end;
end.                                                {koniec modułu UCW1_3}

```

Stosując *instrukcję przypisania* należy przestrzegać zasady, że z lewej strony musi być podstawiony obiekt klasy identycznej lub poprzedzającej z drzewa dziedziczenia w odniesieniu do obiektu z prawej strony. W wyniku działania instrukcji przypisania kopiowane są jedynie pola obiektu z prawej strony do pól obiektu z lewej strony bez zmiany klasy. Działanie instrukcji przypisania przedstawiono w przykładzie 4.

Przykład 4.

```

program CW1_3;
uses Crt, UCW1_3;
var Produkt_1, Produkt_2 : TProdukt;
    Produkt_3             : TProdukt_1;
    Produkt_4             : TProdukt_2;
begin
    Produkt_4.Inicjuj;
    Produkt_4.Nadaj_Nazwe (Nazwy[1]);
    Produkt_4.Nadaj_Gatunek (2);
    Produkt_4.Nadaj_Cene (Ceny[1]);

```

```

Produkt_4.Nadaj_Podatek(22);
Produkt_4.Wydruk_Produktu; Readln;
                        {kopiowane są pola Nazwa, Cena, Gatunek; pomijany Podatek}
Produkt_3 := Produkt_4;
Produkt_3.Wydruk_Produktu; Readln;
                        {kopiowane są pola Nazwa, Cena; pomijany Gatunek}
Produkt_2 := Produkt_3;
Produkt_2.Wydruk_Produktu; Readln;
                                                {kopiowane są pola Nazwa, Cena}

Produkt_1 := Produkt_2;
Produkt_1.Wydruk_Produktu;
end.
                                                {koniec programu CWI_3}

```

Komunikacja między obiektami może odbywać się poprzez przekazywanie w liście parametrów formalnych metody zmiennej obiektowej.

Dla przykładu zdefiniowano metodę *Porownaj_Towary* w klasie *TProdukt*. Będzie ona dziedziczona we wszystkich następcach (*TProdukt_1*, *TProdukt_2*). Nagłówek metody zawiera w liście parametrów formalnych obiekt *Towar* klasy *TProdukt*, przekazywany przez wartość. Przy wywołaniu metody można podstawić w liście parametrów metody dowolny obiekt z drzewa dziedziczenia, ponieważ podano w definicji parametru formalnego obiektu *Towar* korzeń drzewa.

Przy przekazywaniu obiektów przez wartość w liście parametrów metody lub dowolnego podprogramu działają te same zasady, co przy instrukcji przypisania, tzn.: kopiowane są tylko pola zawarte w obiekcie podstawianym, natomiast klasa obiektu jest identyczna z klasą parametru formalnego. Wykorzystanie tej metody przedstawiono w programie **CWI_4**.

```

function TProdukt.Porownaj_Towary(Towar:TProdukt):Boolean;
begin
    Porownaj_Towary := (Nazwa=Towar.Podaj_Nazwe)
                        and (Cena = Towar.Podaj_Cene);
end;

```

Przy takim przekazaniu (bez **var**, czyli przez wartość) na stos kopiowane są jedynie pola *Nazwa* i *Cena* dla obiektów każdej klasy. Jeżeli zastosujemy **var** (przekazanie przez zmienną), w metodzie po podstawieniu obiektu aktualnego będzie dostęp do wszystkich jego pól i metod, ponieważ może nastąpić zmiana klasy obiektu. Przekazywanie obiektów przez zmienną w liście parametrów formalnych będzie omówione w rozdziale 3.1.

W przykładzie 5 przedstawiono wydruk programu **CW1_4** wykorzystującego metodę umożliwiającą komunikację między dwoma obiektami z dowolnego poziomu dziedziczenia.

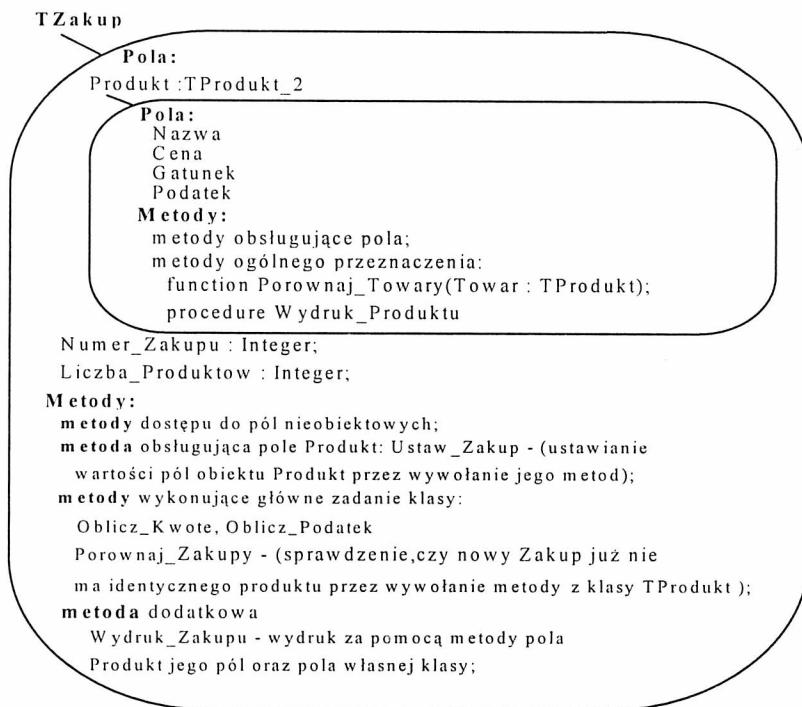
Przykład 5.

```
program CW1_4;
uses Crt, UCW1_3;
var Produkt_1 : TProdukt;
    Produkt_2 : TProdukt_1;
    Produkt_3 : TProdukt_2;
begin
  ClrScr;
  Produkt_3.Inicjuj;
  Produkt_3.Nadaj_Nazwe(Nazwy[1]);
  Produkt_3.Nadaj_Gatunek(2);
  Produkt_3.Nadaj_Cene(Ceny[1]);
  Produkt_3.Nadaj_Podatek(22);
  Produkt_3.Wydruk_Produktu;
  Readln;
  Writeln(#13#10, 'Wydruk poprzednika nr 2');           {True}
  Produkt_2 := Produkt_3;
  Produkt_2.Wydruk_Produktu;
  Writeln(#13#10, Produkt_3.Porownaj_Towary(Produkt_2)); {True}
  Readln;
  Writeln(#13#10, 'Wydruk poprzednika nr 1');           {True}
  Produkt_1 := Produkt_2;
  Produkt_1.Wydruk_Produktu;
  Writeln(#13#10, Produkt_2.Porownaj_Towary(Produkt_1)); {True}
  Readln;
end.                                                     {koniec programu CW1_4}
```

2.2. Obiekty zagnieżdżone

Dalsza droga do realizacji programowej kolejnych klas wyodrębnionych w czasie analizy obiektowej (OOA) polega na wykorzystaniu dotychczasowych doświadczeń, wynikających z zastosowania obiektów z drzewa dziedziczenia *TProdukt...* w programach **CW1_3** i **CW1_4**. Z analizy wynika, że klasa *TZakup* jest właścicielem jednego obiektu z tego drzewa. Wybieramy obiekt typu *TProdukt_2*. Klasa *TZakup* służy do wykonania zakupu danego produktu, czyli między innymi przechowuje liczbę produktów oraz potrafi obliczyć całkowitą kwotę na podstawie ceny i podatku,

pobieranych od obiektu *Produkt* oraz liczby produktów (rozdział 1.1). Schemat deklaracji klasy *TZakup* przedstawiono na rys. 10.



Rys. 10. Schemat klasy *TZakup* z zagnieżdżonym obiektem klasy *TProdukt_2*

Moduł **UCW1_4** powstaje przez dodanie deklaracji klasy *TZakup* i definicji jej metod do modułu **UCW1_3**.

Przykład 6.

```

unit UCW1_4;
interface
const Max_Ilosc = 3;
      Max_Zakupow = 10;
      Ilosci : array[1..Max_Ilosc] of Integer= (1,2,3);
                                                {deklaracje z modulu UCW1_3}
type TProdukt_2 = object
                                                {deklaracje klasy z modulu UCW1_3}
      function Porownaj_Towary (Towar:TProdukt_2):Boolean;
end;
  
```

```

TZakup = object
  private Produkt          : TProdukt_2;
          Numer_Zakupu     : Integer;
          Liczba_Produktow : Integer;
  public  procedure Inicjuj;

                                     {metody dostępu do pól}
  procedure Podaj_Produkt(var Towar:TProdukt_2);
  function Podaj_Liczbe_Produktow : Integer;
  procedure Ustaw_Liczbe_Produktow(Nowa_Liczba:Integer);
  function Podaj_Numer_Zakupu : Integer;
  procedure Ustaw_Numer_Zakupu (Nowy_Numer : Integer);

                                     {metody wykonujące główne zadanie klasy}
  procedure Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                          N_Gatunek, N_Podatek:Integer);

  function Oblicz_Kwote : Real;
  function Oblicz_Podatek : Real;
  function Porownaj_Zakupy(var Zakup:TZakup):Boolean;
  procedure Wydruk_Zakupu;                                     {metoda dodatkowa}
end;
Tab_Zakupow = array [1..Max_Zakupow] of TZakup;

```

implementation

```

                                     {definicje metod z modułu UCWI_3}
function TProdukt_2.Porownaj_Towary(Towar : TProdukt_2);
begin
  Podaj_Towary = inherited Porownaj_Towary(Towar)
    and (Podatek = Towar.Podaj_Podatek);
end;

                                     {definicje metod klasy TZakup omówione dalej w tym rozdziale}
end.                                     {koniec modułu UCWI_4}

```

Poniżej pokazano definicje metod klasy *TZakup*. Metody dostępu do pól zawierają proste przypisania:

```

procedure TZakup.Inicjuj;
begin
  Numer_Zakupu :=0;
  Liczba_Produktow := 0;
end;

```



```
procedure TZakup.Podaj_Produkt(var Towar:TProdukt_2);  
begin  
    Towar := Produkt;  
end;  
  
function TZakup.Podaj_Liczbe_Produktow : Integer;  
begin  
    Podaj_Liczbe_Produktow := Liczba_Produktow;  
end;  
  
procedure TZakup.Ustaw_Liczbe_Produktow(Nowa_Liczba:Integer);  
begin  
    Liczba_Produktow := Nowa_Liczba;  
end;  
  
function TZakup.Podaj_Numer_Zakupu : Integer;  
begin  
    Podaj_Numer_Zakupu := Numer_Zakupu;  
end;  
  
procedure TZakup.Ustaw_Numer_Zakupu(Nowy_Numer:Integer);  
begin  
    Numer_Zakupu := Nowy_Numer;  
end;
```

W metodzie problemu *Ustaw_Zakup* występują żądania skierowane do klasy *TProdukt_2*, aby odebrała dane i ustawiła swoje pola własnymi metodami. Klasa *TZakup* nie musi znać wewnętrznej reprezentacji pól obiektu *TProdukt_2*, a jedynie zleca mu za pomocą jego metod konkretne działania.

```
procedure TZakup.Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny,  
                               Indeks_Ilosci, N_Gatunek, N_Podatek: Integer);  
begin  
    with Produkt do begin  
        Nadaj_Nazwe (Nazwy[Indeks_Nazwy]);  
        Nadaj_Gatunek(N_Gatunek);  
        Nadaj_Podatek(N_Podatek);  
        Nadaj_Cene (Ceny[Indeks_Ceny]);  
    end;  
    Liczba_Produktow := Ilosci[Indeks_Ilosci];  
end;
```

W metodach problemu *Oblicz_Podatek* i *Oblicz_Kwote* żąda się od obiektu *TProdukt_2*, aby podał *Cene* oraz *Podatek* w celu wyliczenia wymaganych kwot. W metodzie problemu *Porownaj_Zakupy* po pobraniu *Produktu* od innego obiektu *TZakup* zleca się własnemu obiektowi *Produkt*, aby porównał za pomocą swojej metody *Porownaj_Towary* własne atrybuty z atrybutami komunikującego się obiektu.

```
function TZakup.Oblicz_Kwote : Real;
begin
  Oblicz_Kwote:=Liczba_Produktow*Produkt.Podaj_Cene;
end;

function TZakup.Oblicz_Podatek : Real;
begin
  Oblicz_Podatek:=Oblicz_Kwote*(Produkt.Podaj_Podatek/100);
end;

function TZakup.Porownaj_Zakupy(var Zakup:TZakup):Boolean;
var Pomoc : TProdukt_2;
begin
  Zakup.Podaj_Produkt (Pomoc);
  Porownaj_Zakupy:= Produkt.Porownaj_Towary(Pomoc);
end;
```

Metoda *Wydruk_Zakupu* ma charakter tymczasowy, ponieważ tylko w czasie wdrażania kolejnych faz programowania jest wykorzystywana do prezentowania wyników działania programu, natomiast nie wykonuje zadań przydzielonych klasie *TZakup*. Po zakończeniu fazy programowania klasy należy zaprojektować obiektowy interfejs kontaktu z użytkownikiem, odrzucając we wszystkich prezentowanych w tym opracowaniu klasach metody tymczasowe *Wydruk...*

```
procedure TZakup.Wydruk_Zakupu;
begin
  Write(#13#10,Numer_Zakupu:3); Produkt.Wydruk_Produktu;
  Write(#13#10,'Liczba produktu: ', Liczba_Produktow:3,
        ' Kwota_Netto: ',Oblicz_Kwote:8:2,
        ' Podatek: ',Oblicz_Podatek:8:2);
end;
```

Tak zdefiniowane metody umożliwiają wykonanie programu wykorzystującego klasę *TZakup*. W przykładzie 6 przedstawiono program **CW1_5**, który zapowiada nową klasę *TRachunek*. Umożliwia on przetestowanie pewnych operacji przed zdefiniowaniem przyszłych metod. W programie zadeklarowano tablicę obiektów *TZakup*. Obiekty te wstawiane są do tablicy (linie 1) i jednocześnie aktualizowana jest

jej zawartość (linie 2) w przypadku wstawienia wcześniej obiektu *TZakup* z identycznymi atrybutami nadanymi obiektowi *TProdukt_2* (zwiększana jest jedynie liczba produktów już w istniejącym obiekcie). Operację tę wykonuje procedura *Aktualizacja_Tablicy_Zakupow*.

```

program CW1_5;
uses Crt, UCW1_4;
var Zakupy           : Tab_Zakupow;
    Numer, I, Liczba  : Integer;

procedure Aktualizacja_Tablicy_Zakupow;
begin
  for I :=1 to Numer-1 do
    if Zakupy[I].Porownaj_Zakupy(Zakupy[Numer]) then
      begin
        Liczba := Zakupy[I].Podaj_Liczbe_Produktow+
          Zakupy[Numer].Podaj_Liczbe_Produktow;
        Zakupy[I].Ustaw_Liczbe_Produktow(Liczba);
        Dec(Numer);
        Break;
      end;
  end;

begin
  ClrScr;
  Numer := 1; Zakupy[Numer].Ustaw_Zakup(1,2,2,1,7);           {1}
  Zakupy[Numer].Ustaw_Numer_Zakupu(Numer);
  Inc(Numer); Zakupy[Numer].Ustaw_Zakup(1,2,2,1,22);          {1}
  Zakupy[Numer].Ustaw_Numer_Zakupu(Numer);
  Aktualizacja_Tablicy_Zakupow;                                 {2}
  Inc(Numer); Zakupy[Numer]:= Zakupy[1];                       {1}
  Aktualizacja_Tablicy_Zakupow;                                 {2}
  for I:=1 to Numer do Zakupy[I].Wydruk_Zakupu;
  Readln;
end.                                                           {koniec programu CW1_5}

```

Na podstawie programu z przykładu 6 zaprojektowano deklarację klasy *TRachunek* oraz zdefiniowano jej metody rozbudowując moduł *UCW1_4* do *UCW1_5*. Klasa ta zawiera tablicę zagnieżdżonych obiektów klasy *TZakup* (rys. 11).

Przykład 7.

unit UCW1_5;

interface

{ deklaracje z modułu UCW1_4 }

type TRachunek = **object**

private Opis_Rachunku : Lan_1;

 Ostatni_Zakup : Integer;

 Zakupy : Tab_Zakupow;

procedure Przenumeruj_Zakupy; *{ metody pomocnicze }*

function Zakres_Rachunku (Numer:Integer):Boolean;

public **procedure** Inicjuj; *{ metoda inicjująca wartości pól }*

{ metody dostępu do pól }

function Podaj_Opis_Rachunku : Lan_1;

procedure Ustaw_Opis_Rachunku (Nowy_Opis : Lan_1);

function Podaj_Ostatni_Zakup : Integer;

procedure Podaj_Zakupy (**var** Zakup:Tab_Zakupow);

function Podaj_Zakup (Numer_Zakupu:Integer;

var Zakup:TZakup):Boolean;

{ metody wykonujące zadanie klasy }

function Oblicz_Rachunek : Real;

procedure Aktualizacja_Rachunku;

procedure Dopisz_Zakup (Indeks_Nazwy, Indeks_Ceny,

 Indeks_Ilosci, N_Gatunek, N_Podatek:Integer);

procedure Wykresl_Zakup (Numer : Integer);

procedure Wydruk_Rachunku; *{ metoda dodatkowa }*

end;

implementation

{ definicje z modułu UCW1_4 }

{ definicje metod klasy TRachunek opisane dalej w tym rozdziale }

end.

{ koniec modułu UCW1_5 }

Metoda pomocnicza *Zakres_Rachunku*, służąca do sprawdzania zakresu indeksów tablicy i zadeklarowana jako prywatna, jest zdefiniowana następująco:

function TRachunek.Zakres_Rachunku (Numer:Integer) :Boolean;

var Log : Boolean;

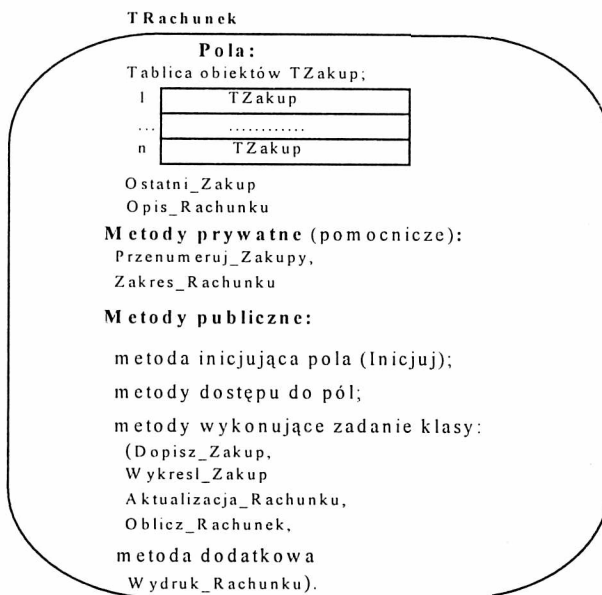
begin

 Log := (Numer > Max_Zakupow) **or** (Numer > Ostatni_Zakup) **or** (Numer <= 0)

if Log **then** Writeln('Dane sa niepoprawne');

 Zakres_Rachunku := Log;

end;



Rys. 11. Schemat klasy *TRachunek*

Metoda inicjująca pola oraz metody dostępu do są następujące:

```

procedure TRachunek.Inicjuj;
begin
    Ostatni_Zakup :=0;
end;

procedure TRachunek.Podaj_Zakupy(var Zakup:Tab_Zakupow) ;
begin
    Zakup := Zakupy;
end;

function TRachunek.Podaj_Zakup(Numer_Zakupu:Integer;
                                var Zakup:TZakup):Boolean;
begin
    if not Zakres_Rachunku(Numer_Zakupu) then
        begin
            Zakup:=Zakupy[Numer_Zakupu] ;
            Podaj_Zakup:=True;
        end
    end

```

```

    else Podaj_Zakup := False;
end;

```

```

function TRachunek.Podaj_Opis_Rachunku : Lan_1;
begin
    Podaj_Opis_Rachunku := Opis_Rachunku;
end;

```

```

procedure TRachunek.Ustaw_Opis_Rachunku (Nowy_Opis:Lan_1);
begin
    Opis_Rachunku := Nowy_Opis;
end;

```

```

function TRachunek.Podaj_Ostatni_Zakup : Integer;
begin
    Podaj_Ostatni_Zakup := Ostatni_Zakup;
end;

```

Przedstawione dalej definicje metod problemu klasy *TRachunek* korzystają z odwołań do obiektów zagnieżdżonych *TZakup*.

Mając do dyspozycji szereg metod tych obiektów, żądają od nich wykonania niezbędnych operacji:

- przy tworzeniu rachunku metodami *Dopisz_Zakup*, *Aktualizacja_Rachunku* korzystają z metod obiektów *Zakupy[I]^*,
- przy naliczaniu wartości rachunku w metodzie *Oblicz_Rachunek* pobierają od każdego obiektu umieszczonego w tablicy *Zakupy* obliczone przez nie własne wartości:
 - netto: metodą *Oblicz_Kwote*,
 - podatku: metodą *Oblicz_Podatek*.

```

function TRachunek.Oblicz_Rachunek : Real;
var I : Integer; Suma : Real;
begin
    Suma := 0;
    for I := 1 to Ostatni_Zakup do
        Suma:=Suma+Zakupy[I].Oblicz_Kwote+Zakupy[I].Oblicz_Podatek;
    Oblicz_Rachunek := Suma;
end;

```

```
procedure TRachunek.Dopisz_Zakup(Indeks_Nazwy,  
    Indeks_Ceny, Indeks_Ilosci, N_Gatunek, N_Podatek: Integer);  
begin  
    Inc(Ostatni_Zakup);  
    if Zakres_Rachunku(Ostatni_Zakup) then Dec(Ostatni_Zakup)  
    else  
        begin  
            Zakupy[Ostatni_Zakup].Ustaw_Zakup(Indeks_Nazwy,  
                Indeks_Ceny, Indeks_Ilosci, N_Gatunek, N_Podatek);  
            Zakupy[Ostatni_Zakup].Ustaw_Numer_Zakupu(Ostatni_Zakup);  
            Aktualizacja_Rachunku;  
        end;  
    end;
```

```
procedure TRachunek.Aktualizacja_Rachunku;  
var I, Liczba : Integer;  
begin  
    for I := 1 to Ostatni_Zakup-1 do  
        if Zakupy[I].Porownaj_Zakupy(Zakupy[Ostatni_Zakup]) then  
            begin  
                Liczba := Zakupy[I].Podaj_Liczbe_Produktow +  
                    Zakupy[Ostatni_Zakup].Podaj_Liczbe_Produktow;  
                Zakupy[I].Ustaw_Liczbe_Produktow(Liczba);  
                Dec(Ostatni_Zakup);  
                Break;  
            end;  
    end;
```

Metoda *Przenumeruj_Zakupy* jest metodą pomocniczą dla metody *Wykresl_Zakup*, dlatego zadeklarowana jest jako metoda prywatna.

```
procedure TRachunek.Przenumeruj_Zakupy;  
var I : Integer;  
begin  
    for I := 1 to Ostatni_Zakup do  
        Zakupy[I].Ustaw_Numer_Zakupu(I);  
    end;
```

Metoda problemu *Wykresl_Zakup* usuwa z tablicy element typu *TZakup* o rozmiarze *SizeOf(TZakup)* i indeksie *Numer* za pomocą procedury *Move* (linia 1). Po usunięciu aktualizuje numerację we wszystkich obiektach *TZakup* (linia 2).

```
procedure TRachunek.Wykresl_Zakup (Numer:Integer);  
begin  
    if not Zakres_Rachunku(Numer) then  
        begin  
            Move(Zakupy[Numer+1],Zakupy[Numer],  
                (Ostatni_Zakup-Numer)*SizeOf(TZakup));           {1}  
            Dec(Ostatni_Zakup);  
            Przenumeruj_Zakupy;                                     {2}  
        end;  
    end;
```

Podobnie jak w poprzednich klasach, metoda *Wydruk_Rachunku* nie realizuje zadań klasy.

```

procedure TRachunek.Wydruk_Rachunku;
var I : Integer;
begin
    Writeln (Opis_Rachunku);
    for I:=1 to Ostatni_Zakup do
        Zakupy[I].Wydruk_Zakupu;
    Writeln(#13#10,'Suma calkowita:',Oblicz_Rachunek:10:2);
end;

```

Program **CW1_6** przedstawia wykorzystanie klasy *TRachunek* do wykonania rachunku zawierającego trzy pozycje zakupu (linie 1) oraz proste operacje: wykreślanie zakupów i wydruki (linie 2).

```
program CW1_6  
uses Crt, UCW1_5;  
var Rachunek : TRachunek;  
    I         : Integer;  
begin  
    Rachunek.Inicjuj;  
    Rachunek.Ustaw_Opis_Rachunku('2-08-1995');  
    Rachunek.Dopisz_Zakup(1,2,2,2,7); {1}  
    Rachunek.Dopisz_Zakup(2,1,2,1,22); {1}  
    Rachunek.Dopisz_Zakup(1,2,2,2,7); {1}  
    Rachunek.Wydruk_Rachunku; Readln; {2}  
    Rachunek.Wykresl_Zakup(2); {2}  
    Rachunek.Wydruk_Rachunku; Readln; {2}  
end. {koniec programu CW1_6}
```


3. Polimorfizm, obiekty dynamiczne

W tym rozdziale zostaną przedstawione podstawowe cechy polimorfizmu oraz wpływ tych cech na postać programu dotyczącego wyników analizy obiektowej (OOA) przedstawionej w rozdziale 2.

3.1. Podstawowe zagadnienia

Jeżeli w programie powtarzają się pewne czynności wykonywane na obiektach różnych klas, pochodzących z tego samego drzewa dziedziczenia, to możemy je zastąpić procedurą, która wywołuje po kolei te czynności i wykonuje je na parametrze formalnym. Typem parametru formalnego jest klasa stanowiąca *korzeń drzewa*. Wtedy można podstawić, jako parametr aktualny, dowolny obiekt z tego drzewa. Poniżej przedstawiono fragment programu, w którym dokonano takiego uproszczenia.

Przykład 8.

```
program WARIANT_1
uses UCW1_3;
var Produkt_1 : TProdukt_1;
    Produkt_2 : TProdukt_2;
begin
Produkt_1.Inicjuj;
Produkt_2.Inicjuj;
Produkt_1.Nadaj_Cene(Ceny[1]);
Produkt_1.Nadaj_Nazwe(Nazwy[1]);
Produkt_1.Wydruk_Produktu;
Produkt_2.Nadaj_Nazwe(Nazwy[2]);
Produkt_2.Nadaj_Cene(Ceny[2]);
Produkt_2.Nadaj_Gatunek(2);
Produkt_2.Wydruk_Produktu;
end.
```

```
program WARIANT_2;
uses UCW1_3;
var Produkt_1 : TProdukt_1;
    Produkt_2 : TProdukt_2;
procedure Wydruk(Nr: Integer;
    var Produkt: TProdukt);
begin
Produkt.Nadaj_Cene(Ceny[Nr]);
Produkt.Nadaj_Nazwe(Nazwy[Nr]);
{Produkt.Nadaj_Gatunek(2);
tutaj kompilator zgłosiłby błąd, ponieważ tej
metody nie ma w klasie TProdukt} {1}
Produkt.Wydruk_Produktu;
end;
begin
Produkt_1.Inicjuj;
Produkt_2.Inicjuj;
Wydruk(1, Produkt_1);
Wydruk(2, Produkt_2); {2}
end.
```

Obiekty aktualne są przekazywane przez zmienną za pośrednictwem listy parametrów formalnych metod lub podprogramów. W takiej sytuacji obiekt aktualny

stanowi parametr nielokalny, tzn. może nastąpić zamiana klasy i tym samym można uzyskać dostęp do różnych pól i metod obiektu aktualnego wewnątrz tych podprogramów (lub metod). Aby jednak można było operować na polach parametru aktualnego we właściwy mu sposób, a więc za pośrednictwem jego metod, trzeba spełnić kilka warunków. Wyjaśnienie tego problemu przedstawiono dalej.

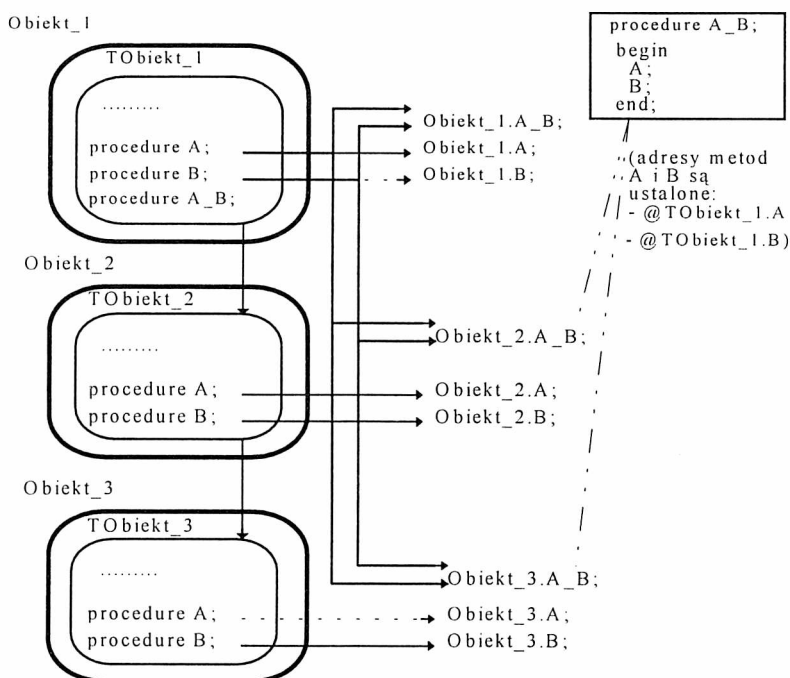
W czasie kompilacji ustalane są adresy metod lub podprogramów. Stąd każdy obiekt wywołuje z góry określoną metodę własną lub dziedziczną. Oznacza to, że mimo iż parametr aktualny ma zasięg nielokalny i może pochodzić z różnych klas, to zawsze będzie używał *metod klasy parametru formalnego*. Stąd w linii 2 programu **WARIANT_2** nie zostanie wydrukowana wartość gatunku dla obiektu *Produkt_2*, gdyż zostanie wywołana metoda *Wydruk_Produktu* z klasy *TProdukt*.

Jednocześnie, jeżeli obiekt aktualny ma więcej pól, to metody obsługujące te pola nie są znane na poziomie klasy korzenia. Oczywiście, nie można zastosować w tekście metody lub podprogramu żadnego identyfikatora metody, który nie został zadeklarowany w klasie użytej do zdefiniowania parametru formalnego lub dowolnego poprzednika (ograniczenie ze strony kompilatora np. linia 1 programu **WARIANT_2**).

W takiej sytuacji potrzebne metody można zdefiniować z wyprzedzeniem, w klasie parametru formalnego, jako puste i zakryć odpowiednimi metodami u następców, którzy będą podstawiani jako parametry aktualne. *Jednak kompilator zawsze przypisze adres metody z klasy parametru formalnego lub jego poprzednika*. Prowadzi to do tego, że parametr aktualny nie wywoła swojej metody, lecz zawsze sztywno przypisaną metodę parametru formalnego, która w sposób ‘abstrakcyjny’ zna obiekt. Gdyby np. została zdefiniowana metoda *Nadaj_Gatunek* w klasie *TProdukt*, wtedy nie wystąpiłby błąd w linii 1 programu **WARIANT_2**, ale tylko ta metoda byłaby tam wywołana. Nie rozwiązałoby to problemu, gdyż ona symuluje obsługę nieistniejącego w jej klasie pola *Gatunek* (jest metodą pustą) i dla obiektów z klasy *TProdukt_1* nie uzyskamy prawidłowych rezultatów.

Podobny mechanizm występuje przy dziedziczeniu metod. Jeżeli w metodzie dziedziczonej użyte są metody pokrywane u następcy, to następca wywołujący dziedziczną metodę będzie musiał korzystać w niej nie z własnych metod, lecz z metod tych klas, które były "widziane" w czasie kompilacji metody dziedziczonej (wszystkie metody tej klasy i jej poprzedników).

Na rys. 12 przedstawiono schemat pobierania dziedziczonych metod statycznych.



Rys. 12. Pobieranie metod przy dziedziczeniu statycznym

Dalej zostanie przedstawiony przykład 9, zawierający następujące przypadki:

- przekazywanie obiektów przez zmienną ze zmianą klasy parametru aktualnego w liście parametrów formalnych procedury,
- wywołania metody dziedziczonej, zawierającej wywołania metod zakrywanych u następców.

Przykład 9 nawiązuje do przypadku, omówionego na początku tego rozdziału. Przedstawia wszystkie środki, jakie wykonano, aby program **CW2_1** mógł zostać skompilowany. Zadeklarowano w każdej z klas *TProdukt..* metody symulujące obsługę brakujących pól. Celem tego była próba uzyskania prawidłowej obsługi obiektów z tego drzewa.

Przykład 9.

```
unit UCW2_1;
interface
```

```

type Lan_1 = string[20];
const
Max_Produkt = 3; Max_Ceny = 3;
Nazwy : array[1..Max_Produkt]of Lan_1=
        ('Produkt_A', 'Produkt_B', 'Produkt_C');
Ceny : array [1..Max_Ceny] of Integer = (25, 22, 20);

type
TProdukt =object
    private   Nazwa : Lan_1;
              Cena  : Real;
    public    procedure Inicjuj;
                                     {metody dostępu do pól}

procedure Nadaj_Nazwe(Nowa_Nazwa:Lan_1);
function Podaj_Nazwe:Lan_1;
procedure Nadaj_Cene (Nowa_Cena:Real);
function Podaj_Cene:Real;
                                     {metody 'pseudoabstrakcyjne'}

procedure Nadaj_Gatunek(Nowy_Gatunek:Integer);
function Podaj_Gatunek:Integer;
procedure Nadaj_Podatek (Nowy_Podatek:Integer);
function Podaj_Podatek:Integer;
                                     {metody dodatkowe}

procedure Wydruk_Atrybutow_Produktow(N_Naz,N_Cen,
                                     N_Gat,N_Podatek:Integer);

procedure Wydruk_Produktu;
end;

TProdukt_1 = object (TProdukt)
    private   Gatunek : Integer;
    public    procedure Inicjuj;
                                     {metody dostępu do pól}

procedure Nadaj_Gatunek (Nowy_Gatunek:Integer);
function Podaj_Gatunek:Integer;
procedure Nadaj_Cene(Nowa_Cena:Real);
procedure Wydruk_Produktu;
                                     {metoda dodatkowa}
end;

TProdukt_2 = object (TProdukt_1)
    private   Podatek : Integer;
    public    procedure Inicjuj;

```

{metody dostępu do pól}

```
procedure Nadaj_Podatek (Nowy_Podatek:Integer);
function Podaj_Podatek:Integer;
procedure Wydruk_Produktu;                                {metoda dodatkowa}
end;
implementation
{-----TProdukt-----}
procedure TProdukt.Inicjuj;
begin
  Nazwa:= '';
  Cena:= 0;
end;

procedure TProdukt.Nadaj_Nazwe (Nowa_Nazwa:Lan_1);
begin
  Nazwa:= Nowa_Nazwa;
end;

function TProdukt.Podaj_Nazwe:Lan_1;
begin
  Podaj_Nazwe:= Nazwa;
end;

procedure TProdukt.Nadaj_Cene (Nowa_Cena:Real);
begin
  Cena:= Nowa_Cena;
end;

function TProdukt.Podaj_Cene:Real;
begin
  Podaj_Cene:= Cena;
end;

procedure TProdukt.Nadaj_Gatunek(Newy_Gatunek:Integer);
begin
end;

function TProdukt.Podaj_Gatunek:Integer;
begin
  Podaj_Gatunek:= -1;
end;
```

```
procedure TProdukt.Nadaj_Podatek(Newy_Podatek:Integer);
begin
  end;

function TProdukt.Podaj_Podatek:Integer;
begin
  Podaj_Podatek:= -1;
end;

procedure TProdukt.Wydruk_Produktu;
begin
  Write(#13#10,'Nazwa: ',Nazwa:20,'Cena_netto: ':13,Cena:8:2);
end;

procedure TProdukt.Wydruk_Atrybutow_Produktow
  (N_Naz, N_Cen, N_Gat, N_Podatek:Integer);
begin
  Nadaj_Nazwe(Nazwy[N_Naz]);  Nadaj_Gatunek(N_Gat);
  Nadaj_Cene(Ceny[N_Cen]);    Nadaj_Podatek(N_Podatek);
  Wydruk_Produktu;           Readln;
end;
{-----TProdukt_1-----}
procedure TProdukt_1.Inicjuj;
begin
  inherited Inicjuj;
  Gatunek:= 1;
end;

procedure TProdukt_1.Nadaj_Gatunek(Newy_Gatunek:Integer);
begin
  Gatunek:= Nowy_Gatunek;
end;

function TProdukt_1.Podaj_Gatunek:Integer;
begin
  Podaj_Gatunek:= Gatunek;
end;

procedure TProdukt_1.Nadaj_Cene(Nowa_Cena:Real);
begin
  inherited Nadaj_Cene(Nowa_Cena);
```

```
case Gatunek of
  2 : Cena := Cena * 0.8;
  3 : Cena := Cena * 0.5;
end;
end;

procedure TProdukt_1.Wydruk_Produktu;
begin
  inherited Wydruk_Produktu;
  Write ('Gatunek : ':11,Gatunek:2);
end;
{-----TProdukt_2-----}
procedure TProdukt_2.Inicjuj;
begin
  inherited Inicjuj;
  Podatek:= 0;
end;

procedure TProdukt_2.Nadaj_Podatek(Nowy_Podatek:Integer);
begin
  Podatek:= Nowy_Podatek;
end;

function TProdukt_2.Podaj_Podatek:Integer;
begin
  Podaj_Podatek:= Podatek;
end;

procedure TProdukt_2.Wydruk_Produktu;
begin
  inherited Wydruk_Produktu;
  Write(' Podatek : ',Podatek:3, ' %');
end;
end.                                     {koniec modułu UCW2_1}
```

W programie **CW2_1** do procedury *Wydruk_Atrybutow_Produktow* przekazywane są przez zmienną obiekty różnych klas lub wywoływana jest przez obiekty różnych klas dziedziczona metoda *Wydruk_Atrybutow_Produktow*. Jednak w programie wywoływane są jedynie metody klasy *TProdukt*. Przyczyna niepowodzenia jest wyjaśniona na rys. 12.

```

program CW2_1;
uses Crt, UCW2_1;
var Produkt_1 : TProdukt;
    Produkt_2 : TProdukt_1;
    Produkt_3 : TProdukt_2;
    I          : Integer;
procedure Wydruk_Atrybutow_Produktow(var Produkt:TProdukt;
                                       Numer:Integer);

begin
    Produkt.Nadaj_Nazwe(Nazwy[Numer]);
    Produkt.Nadaj_Gatunek(2);
    Produkt.Nadaj_Cene(Ceny[Numer]);
    Produkt.Nadaj_Podatek(22);
    Produkt.Wydruk_Produktu;  Readln;
end;

begin
    ClrScr;
    Produkt_1.Inicjuj;
    Produkt_2.Inicjuj;
    Produkt_3.Inicjuj;

    Wydruk_Atrybutow_Produktow(Produkt_1,1);
    Produkt_1.Wydruk_Atrybutow_Produktow(1,1,1,22);
    {wydruk prawidłowy}
    {brak na wydruku pola Gatunek}
    Wydruk_Atrybutow_Produktow(Produkt_2,2);
    Produkt_2.Wydruk_Atrybutow_Produktow(2,2,2,22);
    {brak na wydruku pól Gatunek i Podatek}
    Wydruk_Atrybutow_Produktow(Produkt_3,3);
    Produkt_3.Wydruk_Atrybutow_Produktow(3,3,3,22);
end.
    {koniec programu CW2_1}

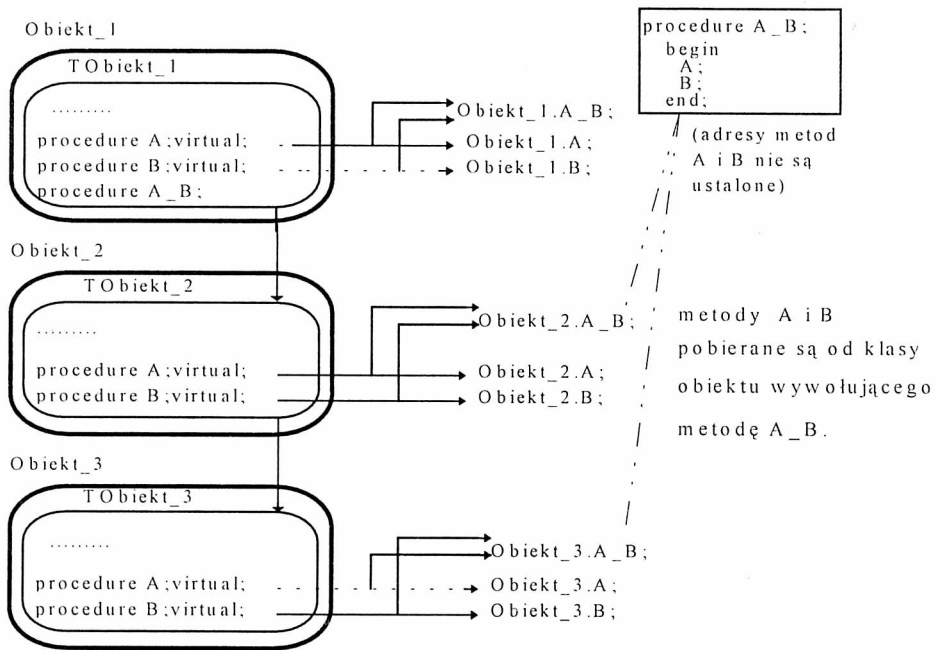
```

Sytuacja przedstawiona w przykładzie 8 ulega całkowitej zmianie w przypadku zastosowania mechanizmu **polimorfizmu**. Umożliwia on ustalanie adresów wybranych metod dopiero w czasie wykonania programu. Dzięki temu obiekt może wykonać tę specjalnie przydzielaną metodę, która we właściwy sposób obsługuje jego pola. Taki sposób ustalania adresu metody jest możliwy po dodaniu przy jej deklaracji dyrektywy **virtual**.

W czasie kompilacji kompilator zawsze sprawdza, czy identyfikator każdej metody jest w zasięgu danej klasy (od klasy w górę drzewa dziedziczenia). Następnie, w przypadku, gdy metoda jest zadeklarowana bez słowa **virtual**, adres jej ustalany

jest w czasie kompilacji. Jeżeli jednak metoda jest zadeklarowana z dyrektywą **virtual**, kompilator zostawia możliwość dynamicznego wywołania metody obiektu aktualnego w czasie działania programu.

Na rys. 13 przedstawiono schemat pobierania metod wirtualnych w czasie działania programu.



Rys. 13. Pobieranie metod przy dziedziczeniu polimorficznym

Przykład 10 przedstawia efekt wprowadzenia polimorfizmu w module UCW2_1 z przykładu 9. Dodanie słowa **virtual** we wszystkich klasach, w deklaracji tych metod, które występują w tekście metody lub procedury *Wydruk_Atrybutow_Produktow*, a zawierają różny scenariusz wykonania, oraz zamiana w deklaracji metod *Inicjuj* słowa kluczowego **procedure** na **constructor** pozwoli na przydzielenie dopiero w czasie wykonania programu adresu metod odpowiadających wywołującemu je obiektowi.

Następujące metody wirtualne zadeklarowane w klasie *TProdukt* (*Podaj_Podatek*, *Nadaj_Podatek*, *Podaj_Gatunek*, *Nadaj_Gatunek*) mają charakter **pseudo-abstrakcyjny**, gdyż w ich definicji występują instrukcje puste, natomiast zostały zadeklarowane w celu umożliwienia zaprogramowania odpowiednich operacji

w typach potomnych. Stąd klasa *TProdukt* może być wykorzystana do deklaracji zmiennych obiektowych.

W odróżnieniu od wirtualnych metod pseudoabstrakcyjnych deklaruje się **wirtualne metody abstrakcyjne**, które muszą być pokryte w klasach potomnych. Typy obiektowe z metodami abstrakcyjnymi są **klasami abstrakcyjnymi**. Oznacza to, że typ ten służy jedynie do wyprowadzania typów potomnych z pokryciem metod abstrakcyjnych i nie może być wykorzystany do deklarowania zmiennych obiektowych. W definicji tych metod występuje albo instrukcja pusta albo wywołanie procedury *Abstract*. Wykorzystanie klasy abstrakcyjnej przedstawiono na przykładzie klasy *TMenu* w rozdziale 3.2 oraz typu *TObiekt* w rozdziale 5.

W przykładzie 10 przedstawiono zmiany, jakie zrobiono w deklarowanych nagłówkach metod, które zamieszczono w module **UCW2_2**, aby uzyskać prawidłowy przebieg programu. Nie pociągnęło to za sobą żadnych zmian w treści tych metod.

Program **CW2_2**, oprócz zmiany w wyszczególnieniu modułów z **UCW2_1** na **UCW2_2**, również nie różni się od programu **CW2_1**. Ze względu na te minimalne różnice nie zamieszczono tekstu programu.

Przykład 10.

{deklaracje klas z drzewa TProdukt.. uzupełnione dyrektywą virtual}

TProdukt = object

private Nazwa : Lan_1;

Cena : Real;

public constructor Inicjuj;

{metody dostępu do pól}

procedure NadaJ_Nazwe (Nowa_Nazwa:Lan_1);

function Podaj_Nazwe:Lan_1;

function Podaj_Cene : Real;

procedure NadaJ_Cene (Nowa_Cena:Real) ;**virtual**;

{metody wirtualne pseudoabstrakcyjne}

procedure NadaJ_Gatunek (Nowy_Gatunek:Integer) ;**virtual**;

function Podaj_Gatunek:Integer;**virtual**;

procedure NadaJ_Podatek (Nowy_Podatek:Integer) ;**virtual**;

function Podaj_Podatek:Integer;**virtual**;

{metody dodatkowe}

procedure Wydruk_Produktu;**virtual**;

procedure Wydruk_Atrybutow_Produktow (N_Naz,N_Cen, N_Gat,
N_Podatek:Integer);

end;

```
TProdukt_1 = object (TProdukt)
    private Gatunek : Integer;
    public constructor Inicjuj;

    {metody dostępu do pól}
    procedure Nadaj_Gatunek (Nowy_Gatunek:Integer);virtual;
    function Podaj_Gatunek:Integer;virtual;
    procedure Nadaj_Cene(Nowa_Cena:Real);virtual;
    procedure Wydruk_Produktu;virtual; {metoda dodatkowa}
end;

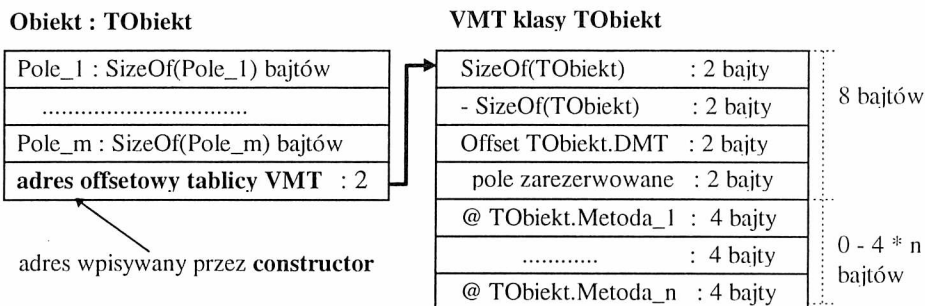
TProdukt_2 = object (TProdukt_1)
    private Podatek:Integer;
    public constructor Inicjuj;

    {metody dostępu do pól}
    procedure Nadaj_Podatek (Nowy_Podatek:Integer);virtual;
    function Podaj_Podatek:Integer;virtual;
    procedure Wydruk_Produktu;virtual; {metoda dodatkowa}
end;
```

Istotą wewnętrznej realizacji polimorfizmu jest specjalna tablica metod wirtualnych (ang. VIRTUAL METHOD TABLE) przypisana klasie zadeklarowanej przy wykorzystaniu pojęć polimorfizmu, tzn. metody typu **constructor** oraz dyrektywy **virtual**.

Wprowadzenie nowej metody **constructor** umożliwia tworzenie tablicy metod wirtualnych (VMT), umieszczanej po uruchomieniu programu w segmencie danych. Każda klasa może mieć jedną tablicę metod wirtualnych. Na rys. 14 pokazano budowę tablicy VMT dla klasy zawierającej metody wirtualne oraz reprezentację w pamięci obiektu należącego do tej klasy.

W VMT przechowywane są między innymi adresy metod, zaznaczonych w czasie deklaracji dyrektywą **virtual**. Jednak zadeklarowanie samej metody **constructor** powoduje dla danej klasy utworzenie tablicy równej zawsze 8 bajtów, natomiast każda metoda wirtualna powoduje zwiększenie jej rozmiarów o 4 bajty na przechowanie swojego adresu.



Rys. 14. Tablica metod wirtualnych (VMT) i reprezentacja w pamięci odpowiadającego jej obiektu

Zanim zaczniemy w programie korzystać z obiektów posiadających VMT, należy koniecznie wcześniej je zainicjować. W tym celu wywołuje się **constructor** (może być nawet metodą pustą), który zapisuje na 2 bajtach dołączonych teraz do pól obiektu adres offsetowy tablicy metod wirtualnych. Jeżeli tego nie zrobimy, program bez właściwego adresu offsetowego, nie będzie miał dostępu do właściwych adresów metod wirtualnych, które właśnie są zapisane w VMT. Obiekty posiadające tablice metod wirtualnych możemy rozpoznać, do której należą klasy, porównując adresy ich VMT za pomocą funkcji *TypeOf(Parametr)*. Funkcja ta zwraca wskaźnik do tej tablicy, gdy parametrem jej wywołania jest nazwa obiektu lub nazwa klasy (przykład 11).

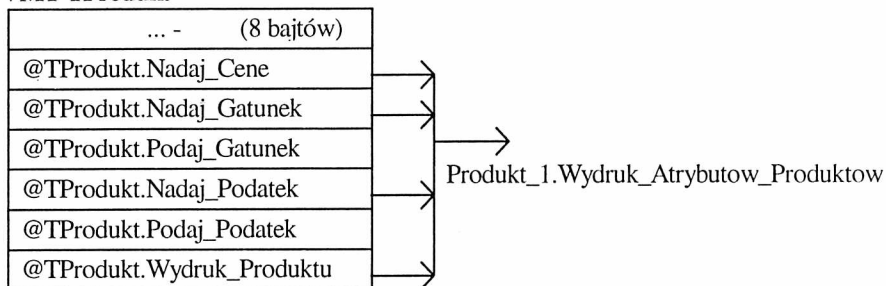
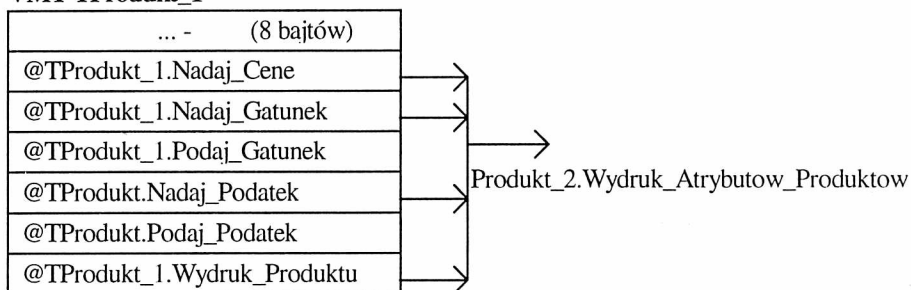
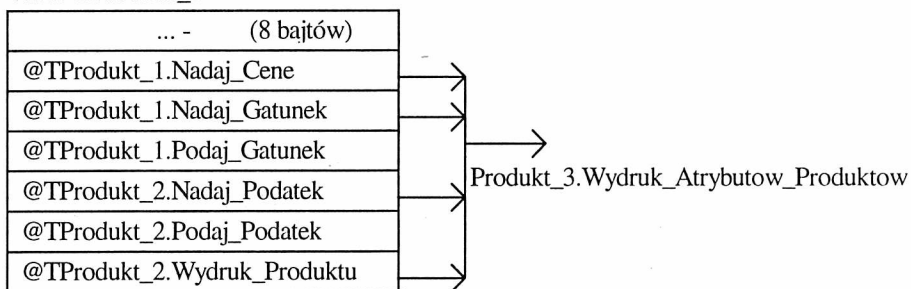
Na rys. 15 przedstawiono tablice metod wirtualnych klas obiektów użytych w programie **CW2_2** oraz położenie adresów offsetowych w każdym z tych obiektów.

Na rys. 16 pokazano mechanizm przydzielania metody wirtualnej danemu obiektowi.

Na rys. 15 pokazano zasady dziedziczenia i pokrywania metod wirtualnych. Adresy dziedziczonych metod wirtualnych są zawsze przepisywane do tablicy VMT klasy dziedziczącej. Adresy metod zakrywanych pojawiają się w miejscu adresu poprzedniej metody, przy czym nie wolno zmieniać nagłówka metody, co było możliwe w przypadku metod statycznych. Można natomiast zmienić deklarację metody statycznej na wirtualną w klasie dziedziczącej.

Pamięć przeznaczona na dane	Pamięć - Segment danych
Produkt_1 : TProdukt; Nazwa : Lan_1 (21 bajtów) Cena : Real (6 bajtów) adres offsetowy (2 bajty)	VMT TProdukt SizeOf (TProdukt) - (2 bajty) SizeOf (TProdukt) - (2 bajty) nil - (2 bajty) zarezerwowane - (2 bajty) @ TProdukt.Nadaj_Cene - (4 bajty) @ TProdukt.Nadaj_Gatunek @ TProdukt.Podaj_Gatunek @ TProdukt.Nadaj_Podatek @ TProdukt.Podaj_Podatek @ TProdukt.Wydruk_Produktu
Produkt_2 : TProdukt_1; Nazwa : Lan_1 (21 bajtów) Cena : Real (6 bajtów) adres offsetowy (2 bajty) Gatunek : Integer (2 bajty)	VMT - TProdukt_1 SizeOf (TProdukt_1) - (2 bajty) SizeOf (TProdukt_1) - (2 bajty) nil - (2 bajty) zarezerwowane - (2 bajty) @ TProdukt_1.Nadaj_Cene - (4 bajty) @ TProdukt_1.Nadaj_Gatunek @ TProdukt_1.Podaj_Gatunek @ TProdukt.Nadaj_Podatek @ TProdukt.Podaj_Podatek @ TProdukt_1.Wydruk_Produktu
Produkt_3 : TProdukt_2 Nazwa : Lan_1 (21 bajtów) Cena : Real (6 bajtów) adres offsetowy (2 bajty) Gatunek : Integer (2 baty) Podatek : Integer (2 bajty)	VMT - TProdukt_2 SizeOf (TProdukt_2) - (2 bajty) SizeOf (TProdukt_2) - (2 bajty) nil - (2 bajty) zarezerwowane - (2 bajty) @ TProdukt_1.Nadaj_Cene - (4 bajty) @ TProdukt_1.Nadaj_Gatunek @ TProdukt_1.Podaj_Gatunek @ TProdukt_2.Nadaj_Podatek @ TProdukt_2.Podaj_Podatek @ TProdukt_2.Wydruk_Produktu

Rys. 15. Tablice metod wirtualnych klas z drzewa dziedziczenia *TProdukt*., oraz reprezentacje odpowiadajacych im obiektów w pamięci (moduł UCW2_2)

VMT TProdukt**VMT TProdukt_1****VMT TProdukt_2**

Rys. 16. Pobieranie adresów metod wirtualnych z tablicy klasy, do której należy obiekt, wykonujący metodę *Wydruk_Atrybutow_Projektow*

Program **TABLICA_VMT** pokazany w przykładzie 11 umożliwia wyświetlenie na ekranie zawartości tablicy metod wirtualnych klas zadeklarowanych w module **UCW2_2** oraz obiektów zadeklarowanych w programie **TABLICA_VMT**. W linii (1) pobrany został adres VMT na podstawie faktu, że dla wszystkich klas

zadeklarowanych w programie lub w modułach istnieją w segmencie danych ich VMT, niezależnie czy zostały zadeklarowane zmienne obiektowe tych klas. W linii (2) możliwe jest pobranie adresu VMT tylko w przypadku zadeklarowania zmiennej obiektowej i po wywołaniu metody **constructor**, który wpisze do obiektu adres offsetowy VMT, należącej do jego klasy.

Przykład 11.

```
program TABLICA_VMT;
uses UCW2_2, Crt;
type Rek = record
    SizePoz, SizeNeg, Pom1, Pom2           : Word;
    Adr1, Adr2, Adr3, Adr4, Adr5, Adr6 : Pointer;
end;
var Produkt : TProdukt_1;
    Pom      : Pointer;

procedure Wydruk_VMT(Adres:Pointer);
begin
    with Rek(Adres^) do
    begin
        Writeln('Rozmiar Obiektu: ', SizePoz);
        Writeln('Rozmiar zanegowany obiektu: ', SizeNeg);
        Writeln('Pom1: ', Pom1, ' Pom2: ', Pom2);
        Writeln('Adres Nadaj_Cene :', Seg(Adr1^), ': ', Ofs(Adr1^));
        Writeln('Adres Nadaj_Gatunek:', Seg(Adr2^), ': ', Ofs(Adr2^));
        Writeln('Adres Podaj_Gatunek:', Seg(Adr3^), ': ', Ofs(Adr3^));
        Writeln('Adres Nadaj_Podatek:', Seg(Adr4^), ': ', Ofs(Adr4^));
        Writeln('Adres Podaj_Podatek:', Seg(Adr5^), ': ', Ofs(Adr5^));
        Writeln('Adres Wydruk_Produktu:', Seg(Adr6^), ': ', Ofs(Adr6^));
    end;
    end;

begin
    Pom:=TypeOf(TProdukt);
    Wydruk_VMT(Pom);
    Produkt.Inicjuj;
    Pom:=TypeOf(Produkt);
    Wydruk_VMT(Pom);
    Readln;
end.
```

{1}

{2}

W przykładzie 12 pokazano skutki użycia niezainicjowanych obiektów.

Przykład 12.

```

program OBIEKTY_STATYCZNE;
uses UCW2_2;
var Produkt_1, Produkt_2 : PProdukt;
    Produkt_3             : PProdukt_1;
begin
    Produkt_1.Inicjuj;
    Produkt_1.Nadaj_Nazwe(Nazwy[1]);
    Produkt_1.Nadaj_Cene(Ceny[1]);
    Produkt_1.Wydruk_Produktu;
                                {kopiowane są tylko pola Nazwa i Cena bez adresu offsetowego;}
    {Produkt_2:=Produkt_1;
      Produkt_2.Wydruk;           po wywołaniu tej instrukcji komputer zawiesi swoją pracę,
                                gdyż obiekt Produkt_2 nie zna adresu dostępu do VMT, gdzie
                                znajduje się adres metody Wydruk }
                                {prawidłowy przebieg programu dzięki zainicjowaniu obiektu Produkt_2}
    Produkt_2.Inicjuj;
    Produkt_2:=Produkt_1;
    Produkt_2.Wydruk_Produktu;
end.                                {koniec programu OBIEKTY_STATYCZNE}

```

3.2. Obiekty dynamiczne

W rozdziałach 2 oraz 3.1 opisano obiekty stosowane w postaci zmiennych statycznych. Wraz z mechanizmem polimorfizmu bardzo wygodne jest operowanie w programach wskaźnikami obiektów. Wskaźnik może przechowywać adres obiektu klasy wynikającej z jego definicji lub adres dowolnego następcy. Należy podkreślić, że sposób tworzenia obiektów dynamicznych na stercie za pomocą procedury *New* oraz zwalnianie zajętej przez nie pamięci procedurą *Dispose* wynika ze specyfiki języka Turbo Pascal 7.0. *Wszystkie zasady tworzenia obiektów dynamicznych, opisane w przykładach, są charakterystyczne dla zastosowanego języka.* Opisy działania metod typu **constructor** oraz **destructor** mają charakter przedstawienia modelu działania tych operacji, gdzie istotne są skutki, a nie wewnętrzna realizacja ich działania.

Przykład definicji typu wskaźnika oraz samego wskaźnika jest następujący:

type

```
PProdukt = ^TProdukt;                                {typ wskaźnikowy}  
TProdukt = object                                    {typ bazowy typu wskaźnikowego}  
            {deklaracje pól i metod}  
            end;
```

```
var Produkt_1 : PProdukt;                            {wskaźnik obiektu}
```

W wyniku działania instrukcji przypisania lub w przypadku przekazywania przez wartość wskaźników w liście parametrów formalnych wskaźnik może wskazywać na obiekt tej klasy, która została mu przypisana, a nie jaka wynika z jego definicji i również dzięki polimorfizmowi można wywołać właściwą metodę. Taki mechanizm działa jedynie dla obiektów należących do tego samego drzewa dziedziczenia i przypisywane wskaźniki obiektów muszą mieć klasę bazową identyczną lub klasę następców.

W przypadku przekazywania wskaźników obiektów przez zmienną obowiązuje identyczność typu parametru formalnego oraz parametru aktualnego. W takim przypadku należy najpierw przypisać wskaźnik aktualny do zmiennej wskaźnikowej o typie identycznym z typem parametru formalnego, następnie tę zmienną podstawić za parametr formalny. Innym sposobem jest zmiana typu parametru aktualnego w momencie podstawiania (konwersja typu zmiennej). Wykorzystanie takiego sposobu pokazano w rozdziale 5.3 (w metodzie *Odczytaj* klasy *TStrumien*).

Dzięki wskaźnikom można wykorzystać obiekty dynamiczne umieszczane na sterku za pomocą procedury (lub funkcji) *New*, wywołanej z drugim parametrem typu **constructor**. Wówczas procedura (lub funkcja) *New* zachowuje się jak procedura *GetMem*. Po utworzeniu obiektu na sterku metoda **constructor** wykonuje następujące zadania:

- wpisuje adres offsetowy VMT do obiektu, co jest jej *głównym zadaniem*,
- następnie może działać jak zwykła metoda statyczna - najczęściej wykonuje inicjowanie wartości pól obiektu lub może być metodą pustą.

Metoda ta może być wykonana po uprzednim utworzeniu na sterku obiektu za pomocą procedury *New*, wywołanej z jednym parametrem.

Metoda **constructor** nie może być metodą wirtualną. Oznacza to, że jej adres jest ustalony w czasie kompilacji na podstawie typu bazowego wskaźnika. W dalszej części rozdziału zostanie wyjaśniona rola procedury *Fail*, użytej w definicji metody **constructor**.

Przy zwalnianiu pamięci wykorzystywana jest metoda **destructor**. Zaleca się używać jej jako drugiego parametru procedury *Dispose*. Tak wywołana procedura zachowuje się jak procedura *FreeMem*. Metoda **destructor** spełnia następujące zadania:

- może być wywołana jako zwykła metoda;
- wywołana w procedurze *Dispose* przed usunięciem obiektu jest wykonywana jak zwykła metoda - najczęściej jest wykorzystana do zwolnienia pamięci zajętej przez pola dynamiczne obiektu lub jest metodą pustą;
- podaje do procedury *Dispose* rozmiar obiektu, pobrany z aktualnej tablicy metod wirtualnych, co jest jej *głównym zadaniem*. Dzięki temu procedura *Dispose* działa jak procedura *FreeMem*.

Metoda **destructor** może być metodą wirtualną. W przeciwnym wypadku kompilator ustala jej adres na podstawie typu bazowego wskaźnika użytego w procedurze *Dispose*.

Poniżej przedstawiono moduł **UCW2_3** i program **CW2_3**, korzystające ze wskaźników obiektów. Połączenie wskaźników i polimorfizmu daje duże możliwości w programowaniu obiektowym i będzie to technika stosowana w kolejnych wersjach projektu.

Przykład 13.

```
unit UCW2_3;
interface
type Lan_1 = string[20];
const Max_Produkt = 3; Max_Ceny = 3;
      Nazwy:array[1..Max_Produkt]of Lan_1 =
          ('Produkt_A', 'Produkt_B', 'Produkt_C');
      Ceny : array [1..Max_Ceny] of Integer = (25, 22, 20);
type
PProdukt = ^TProdukt;
TProdukt = object
    private Nazwa : Lan_1;
           Cena : Real;
    public constructor Inicjuj;
           destructor Usun;

{metody dostępu do pól}

procedure Nadaj_Nazwe (Nowa_Nazwa:Lan_1);
function Podaj_Nazwe:Lan_1;
procedure Nadaj_Cene (Nowa_Cena:Real);virtual;
function Podaj_Cene:Real;
```

```

procedure Nadaj_Gatunek(Nowy_Gatunek:Integer);virtual;
function Podaj_Gatunek:Integer;virtual;
procedure Nadaj_Podatek (Nowy_Podatek:Integer);virtual;
function Podaj_Podatek:Integer;virtual;
                                     {metoda wykonująca zadanie klasy}
function Porownaj_Towary(Towar:PProdukt):Boolean;virtual;
                                     {metody dodatkowe}
procedure Wydruk_Atrybutow_Produktow(N_Naz, N_Cen,N_Gat,
                                     N_Podatek:Integer);

procedure Wydruk_Produktu;virtual;
end;
PProdukt_1 = ^TProdukt_1;
TProdukt_1 = object (TProdukt)
    private Gatunek : Integer;
    public constructor Inicjuj;
                                     {metody dostępu do pól}
procedure Nadaj_Gatunek(Nowy_Gatunek:Integer);virtual;
function Podaj_Gatunek:Integer;virtual;
procedure Nadaj_Cene(Nowa_Cena:Real);virtual;
procedure Wydruk_Produktu;virtual;           {metoda dodatkowa}
end;
PProdukt_2 = ^TProdukt_2;
TProdukt_2 = object (TProdukt_1)
    private Podatek : Integer;
    public constructor Inicjuj;
                                     {metody dostępu do pól}
procedure Nadaj_Podatek(Nowy_Podatek:Integer);virtual;
function Podaj_Podatek:Integer;virtual;
                                     {metoda wykonująca zadanie klasy}
function Porownaj_Towary(Towar:PProdukt):Boolean;virtual;
procedure Wydruk_Produktu;virtual;           {metoda dodatkowa}
end;
implementation
{-----TProdukt-----}
    {pozostałe definicje metod klasy TPProdukt z modułu UCW2_2}
function TPProdukt.Porownaj_Towary (Towar:PProdukt):Boolean;
begin
    Porownaj_Towary:=
    (Nazwa=Towar^.Podaj_Nazwe) and (Cena=Towar^.Podaj_Cene);
end;

```

```

destructor TProdukt.Unsun;
begin
end;

{-----TProdukt_1-----}
{definicje metod klasy TProdukt_1 z modułu UCW2_2}
{-----TProdukt_2-----}
{definicje metod klasy TProdukt_2 z modułu UCW2_2}
function TProdukt_2.Porownaj_Towary (Towar:PProdukt):Boolean;
begin
    Porownaj_Towary:= inherited Porownaj_Towary (Towar)
                    and (Podatek=Towar^.Podaj_Podatek);

end;
end.                                     {koniec modułu UCW2_3}

program CW2_3;
uses Crt, UCW2_3;
var Produkt_1, Produkt_2 : PProdukt;
    Produkt_3              : PProdukt_1;
    Produkt_4              : PProdukt_2;
begin
    ClrScr;
    Writeln('Sterta przed: ',MemAvail);
    if MaxAvail < SizeOf(TProdukt_2) then Halt;
    {pamięć jest zajmowana podobnie jak przez GetMem(Produkt_4, SizeOf(TProdukt_2))}
    New( Produkt_4, Inicjuj);                                     [1]
    Produkt_4^.Wydruk_Atrybutów_Produktow(1,1,1,22);
    Readln;
    Produkt_3:= Produkt_4;                                       [2]
    Produkt_3^.Wydruk_Produktu;
    Writeln (Produkt_3^.Porownaj_Towary (Produkt_4));
    Readln;
    Produkt_2:=Produkt_3;                                       [3]
    Produkt_2^.Wydruk_Produktu;
    Writeln (Produkt_2^.Porownaj_Towary (Produkt_4));
    Readln;
    Produkt_1:= Produkt_2;                                       [4]
    Produkt_1^.Wydruk_Produktu;
    Writeln (Produkt_1^.Porownaj_Towary (Produkt_4));
    {zwalnianie pamięci}
    {Dispose(Produkt_1) - wystąpi błąd, gdyż zwolni się pamięć o rozmiarze równym
    SizeOf(TProdukt), ale zastosowanie metody destructor pozwoli na odczyt w VMT aktualnej
  
```

klasy rozmiaru faktycznie zajmowanej pamięci, wskazywanej przez *Produkt_1*, czyli *SizeOf(TProdukt_2)*. Wywołanie procedury *Dispose(Produkt_1, Usun)* z metodą **destructor** 'zamienia' ją na procedurę *FreeMem(Produkt_1, SizeOf(TProdukt_2))*. Należy więc wykonać następującą instrukcję:}

Dispose(Produkt_1, Usun) {5}

{*Dispose(Produkt_4)* - tak byłoby też dobrze, gdyż właśnie zwolniona zostałaby pamięć o rozmiarze *SizeOf(TProdukt_2)*.}

```
Writeln('Sterta po: ', MemAvail);
Readln
end. {koniec programu CW2_3}
```

Na rys. 17 przedstawiono efekty działań na wskaźnikach w programie **CW2_3**.

Kroki programu	Zmienne programu	Sterta					
	Produkt_1 = ?, Produkt_2 = ?, Produkt_3 = ?, Produkt_4 = ?.	?					
1.New(Produkt_4,Init);	Produkt_1, Produkt_2, Produkt_3 = ? Produkt_4 →	Produkt_4^ - TProdukt_2 → <table><tr><td>Nazwa (21 bajtów)</td></tr><tr><td>Cena (2 bajty)</td></tr><tr><td>adres offsetowy (2 bajty)</td></tr><tr><td>Gatunek (2 bajty)</td></tr><tr><td>Podatek (2 bajty)</td></tr></table>	Nazwa (21 bajtów)	Cena (2 bajty)	adres offsetowy (2 bajty)	Gatunek (2 bajty)	Podatek (2 bajty)
Nazwa (21 bajtów)							
Cena (2 bajty)							
adres offsetowy (2 bajty)							
Gatunek (2 bajty)							
Podatek (2 bajty)							
2.Produkt_3:= Produkt_4;	Produkt_1, Produkt_2 = ?, Produkt_3 →	"					
3.Produkt_2:= Produkt_3;	Produkt_1 = ?, Produkt_2 →	"					
4.Produkt_1:= Produkt_2;	Produkt_1 →	"					
5.Dispose(Produkt_1,Us);	FreeMem (Produkt_1,SizeOf(TProdukt_2)) Produkt_1 = ?, Produkt_2 = ?, Produkt_3 = ?, Produkt_4 = ?.	?					

Rys. 17. Schemat działań programu CW2_3

Zastosowanie procedury *Fail* wpływa na przebieg działania metody **constructor**. Poniżej przedstawiono przykład definicji metody **constructor**, w której w przypadku braku pamięci wywołana jest procedura *Fail* i następuje wyjście z metody. We fragmencie programu przedstawiono dwa następujące przypadki:

- kiedy **constructor** jest wywołany w procedurze *New*, zostanie przypisana wartość **nil** wskaźnikowi obiektu (linia 1) w przypadku stwierdzenia braku pamięci; procedura *Fail* działa jednocześnie jak procedura *Exit*.
- jeżeli **constructor** zostanie wywołany poza procedurą *New*, zastosowanie procedury *Fail* spowoduje, że stanie się funkcją zwracającą wartość typu Boolean (linia 2), tzn.: *False*, gdy procedura wykona się, w przeciwnym wypadku *True*.

```
constructor TProdukt.Inicjuj ( Rozmiar : Word);
begin
  if MaxAvail< Rozmiar then
    begin
      Writeln('Brak pamieci');
      Fail;
    end;
    Nazwa := '';
    Cena := 0;
end;
```

W programie, którego fragment pokazano poniżej, wywołano **constructor** dwoma sposobami:

```
var Produkt_1 : PProdukt;
    Produkt_2 : TProdukt;
{.....}
Writeln('Serta przed: ', MemAvail);
New(Produkt_1, Inicjuj(SizeOf(TProdukt)));
if Produkt_1=nil then Halt; {1}
if not Produkt_2.Inicjuj(1000) then Halt; {2}
{.....}
```

Problem tłumaczenia programu zawierającego wskaźniki przedstawia przykład 14. Komentarze, umieszczone w programie wyjaśniają działanie kolejno wykonywanych instrukcji programu **METODY**.

W czasie tłumaczenia programu kompilator *sprawdza obecność identyfikatorów* zawsze w zasięgu klasy typu bazowego wskaźnika, niezależnie od zmiany klasy obiektu wskazywanego przez niego.

Adresy metod statycznych przypisane są w czasie tłumaczenia programu z zasięgu klasy typu bazowego wskaźnika obiektu. W czasie wykonania programu metody te będą działały w obszarze obiektu aktualnego. Metody wirtualne w klasie typu bazowego wskaźnika nie są przypisane w czasie tłumaczenia, natomiast w czasie wykonania programu są pobierane dynamicznie z aktualnej tablicy metod wirtualnych (VMT) obiektu wskazywanego przez wskaźnik.

Przykład 14.

```

program METODY;
  type
    PTyp_1 = ^Typ_1;
    Typ_1 = object
      Pole_1 : Integer;
      constructor Inicjuj;
      destructor Usun;
      procedure Metoda_1;
      procedure Metoda_2; virtual;
    end;

    PTyp_2 = ^Typ_2;
    Typ_2 = object (Typ_1)
      Pole_2 : Longint;
      procedure Metoda_1;
      procedure Metoda_2; virtual;
      procedure Metoda_3; virtual;
    end;

    PTyp_3 = ^Typ_3;
    Typ_3 = object (Typ_2)
      Pole_3: string;
      procedure Metoda_1; virtual;
      procedure Metoda_3; virtual;
    end;

  {-----Typ_1-----}
constructor Typ_1.Inicjuj;
begin
end;

```

```
procedure Typ_1.Metoda_1;
begin
  Writeln('Metoda_1 statyczna klasy Typ_1');
end;

procedure Typ_1.Metoda_2;
begin
  Writeln('Metoda_2 virtualna klasy Typ_1');
end;

destructor Typ_1.Usun;
begin
end;
{-----Typ_2-----}
procedure Typ_2.Metoda_1;
begin
  Writeln('Metoda_1 statyczna klasy Typ_2');
end;

procedure Typ_2.Metoda_2;
begin
  Writeln('Metoda_2 virtualna klasy Typ_2');
end;

procedure Typ_2.Metoda_3;
begin
  Writeln('Metoda_3 virtualna klasy Typ_2');
end;
{-----Typ_3-----}
procedure Typ_3.Metoda_1;
begin
  Writeln('Metoda_1, teraz virtualna, klasy Typ_3');
end;

procedure Typ_3.Metoda_3;
begin
  Writeln('Metoda_3 virtualna klasy Typ_3');
end;

var PO1 : PTyp_1;
    PO2 : PTyp_2;
    PO3 : PTyp_3;
```


begin

New (PO3, Inicjuj) ; {było 262 +2 = 264 bajtów pamięci, ponieważ utworzono na
stercie obiekt klasy Typ 3}

PO3^.Metoda_1; {metoda wirtualna w klasie Typ_3, adres przydzielany z VMT -
Typ_3 w czasie działania programu}

PO3^.Metoda_2; {metoda wirtualna w klasie Typ_3, dziedziczona od klasy Typ_2,
adres pobierany z VMT-Typ_3 w czasie działania programu}

```
PO3^.Metoda_3;      {metoda wirtualna w klasie Typ_3, adres pobierany z VMT-Typ_3
                     w czasie działania programu}
```

`PO2 := PO3 ;` {wskaźnik *PO2* wskazuje na obiekt klasy Typ 3}

PO2^.Metoda_1; {w czasie kompilacji przydzielony adres metody statycznej z klasy
Typ 2}

P02^.Metoda_2; {metoda wirtualna w klasie Typ_2, adres przydzielony z VMT-
Typ_3 metody dziedziczonej Typ_2.Metoda_2, w czasie działania
programu }

PO2^.Metoda_3; {metoda wirtualna w klasie Typ_2, adres przydzielany z VMT -
Typ_3 metody pokrywającej w klasie Typ_3, w czasie działania programu}

$PO1 := PO2;$ {wskaźnik *PO1* wskazuje na obiekt klasy Typ 3}

PO1^.Metoda_1; {w czasie kompilacji przydzielony adres metody statycznej z klasy
Typ 1}

P01^.Metoda_2; {metoda wirtualna w klasie Typ_1, adres przydzielany z VMT-
Typ_3 metody dziedziczonej Typ_2.Metoda_2, w czasie działania
programu}

```
{ P01^.Metoda_3; brak metody w klasie Typ 1-wystąpi błąd kompilatora}
```

Dispose (PO1, Usun) ; {przybyło 262 + 2 = 264 bajtów, rozmiar pobrany z VMT-Typ_3}
 {lub Dispose(PO2, Usun)} {za pomocą dziedziczonej, statycznej metody **destructor**
 Typ 1. Usun}

*{ lub tylko Dispose(PO3)} {bez metody **destructor**, gdyż typ bazowy wskaźnika odpowiada klasie obiektu zwalnianego ze sterty oraz pola są statyczne}*

end.

```
{koniec programu METODY}
```

W przykładzie 15 przedstawiono proces zwalniania pamięci za pośrednictwem metody **destructor**. Przykład ten pokazuje, kiedy **destructor** powinien być wirtualny. Wyjaśnienia zostały podane w komentarzach.

Podstawowa zasada, dotycząca zwalniania pamięci zajętej przez obiekt polimorficzny polega na tym, że informacja o rozmiarze obiektów z polami statycznymi jest zapisana w ich VMT. Taką informację może pobrać metoda **destructor** (pusta) dziedziczona od korzenia, jak to przedstawiono w przykładzie 14.

Jeśli obiekty w danym drzewie mają pola dynamiczne, wtedy w ich VMT zapisany jest rozmiar uwzględniający wielkość pól wskaźnikowych. Aby prawidłowo zwolnić pamięć, należy przed usunięciem obiektu zwolnić pamięć zajętą przez jego pola. Wówczas w każdej klasie powinien być zdefiniowany **destructor**, zawierający odpowiednie instrukcje zwalnijące te pola. *Taki destructor powinien być wirtualny.*

Gdyby było inaczej, w linii 1 programu **DESTRUCTORY** zostałaby zwolniona pamięć równa: $8(2)+8(2)+16(14)$, co oznaczałoby, że pamięć przeznaczona na sam obiekt zostałaby zwolniona prawidłowo, ponieważ rozmiar zostałby pobrany z VMT. Typ_3, natomiast pola byłyby zwolnione metodą *Typ_1.Usun* (w linii 2 metodą *Typ_2.Usun*), a więc na sterce zostałyby niezwolnione pole *Pole_3*[^] równe 256 bajtów.

Przykład 15.

```
program DESTRUCTORY;
```

```
type PTyp_1 = ^Typ_1;
```

```
    Typ_1 = object
```

```
        Pole_1 : ^Integer;
```

```
        constructor Inicjuj;
```

```
        destructor Usun;virtual;
```

```
    end;
```

```
PTyp_2 = ^Typ_2;
```

```
Typ_2 = object (Typ_1)
```

```
    Pole_2 : ^Longint;
```

```
    constructor Inicjuj;
```

```
    destructor Usun;virtual;
```

```
    end;
```

```
PTyp_3 = ^Typ_3;
```

```
Typ_3 = object (Typ_2)
```

```
    Pole_3 : ^string;
```

```
    constructor Inicjuj;
```

```
    destructor Usun;virtual;
```

```
    end;
```

```
{-----Typ_1-----}
constructor Typ_1.Inicjuj;
begin
    if MaxAvail< SizeOf(Integer) then
        Halt;
        New(Pole_1);
    end;

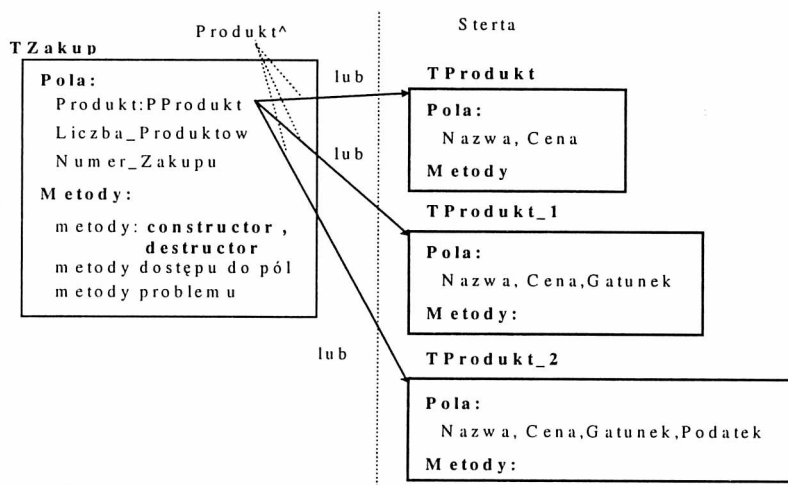
destructor Typ_1.Usun;
begin
    Dispose(Pole_1)
end;
{-----Typ_2-----}
constructor Typ_2.Inicjuj;
begin
    inherited Inicjuj;
    if MaxAvail<SizeOf(Longint) then
        Halt;
        New(Pole_2);
    end;

destructor Typ_2.Usun;
begin
    inherited Usun;
    Dispose(Pole_2);
end;
{-----Typ_3-----}
constructor Typ_3.Inicjuj;
begin
    inherited Inicjuj;
    if MaxAvail< SizeOf(string) then
        Halt;
        New(Pole_3);
    end;

destructor Typ_3.Usun;
begin
    inherited Usun;
    Dispose(Pole_3);
end;
```


3.3. Dynamiczne obiekty zagnieżdżone

Kolejny etap projektu wykorzystuje rezultaty uzyskane ze stosowania klas z drzewa *TProdukt..* po wprowadzeniu polimorfizmu i obiektów dynamicznych. Teraz klasa *TZakup* może przechowywać wskaźnik na dowolny obiekt z tego drzewa. Oznacza to realizację obiektu o zmiennej strukturze danych. Na rys. 18 przedstawiono schemat zmian, uzyskanych dzięki zastosowaniu zagnieżdżonych obiektów dynamicznych oraz polimorfizmowi.



Rys. 18. Klasa *TZakup* reprezentuje typ danych o zmiennej strukturze

Deklarację klasy *TZakup* wykonano w module **UCW2_4** wykorzystując moduł **UCW2_3**.

Przykład 16.

```
unit UCW2_4;
```

```
interface
```

{deklaracje z modułu UCW2_3}

```
const Max_Ilosc = 3;
```

```
Max_Zakupow = 10;
```

```
Ilosci : array[1..Max_Ilosc] of Integer = (1,2,3);
```

```
type PZakup = ^TZakup;
```

```
TZakup = object
```

```
private Produkt : PProdukt;
```

```
Numer_Zakupu : Integer;
```

```
Liczba_Produktow : Integer;
```

```

constructor Inicjuj (Nowy_Produkt:PProdukt);
destructor Usun;
                                                    {metody dostępu do pól}

function Podaj_Produkt:PProdukt;
function Podaj_Liczbe_Produktow:Integer;
procedure Ustaw_Liczbe_Produktow(Nowa_Liczba:Integer);
function Podaj_Numer_Zakupu:Integer;
procedure Ustaw_Numer_Zakupu (Nowy_Numer:Integer);
                                                    {metody wykonujące zadania klasy}
procedure Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                        N_Gatunek, N_Podatek:Integer);

function Oblicz_Kwote:Real;
function Oblicz_Podatek:Real;
function Porownaj_Zakupy(Zakup:PZakup):Boolean;
procedure Wydruk_Zakupu;
                                                    {metoda dodatkowa}
end;
  Tab_Zakupow = array [1..Max_Zakupow] of PZakup;
implementation
                                                    {definicje z modułu UCW2_3}
                                                    {definicje metod klasy TZakup}
                                                    {koniec modułu UCW2_4}
end.

```

Poniżej pokazano definicje metod klasy *TZakup*. Metoda **constructor** *Inicjuj* inicjuje pola klasy i przekazuje adres obiektu dynamicznego z drzewa *TProdukt...*, natomiast metoda **destructor** usuwa ten obiekt ze sterty.

```

constructor TZakup.Inicjuj (Nowy_Produkt:PProdukt);
begin
  Produkt:=Nowy_Produkt;
  Numer_Zakupu:=0;
  Liczba_Produktow:=0;
end;
destructor TZakup.Usun;
begin
  Dispose(Produkt, Usun);
end;

```

Pozostałe metody mają takie samo znaczenie jak w przypadku statycznej klasy *TZakup*, natomiast w definicjach występują odwołania do obiektów dynamicznych z drzewa *TProdukt...* oraz *TZakup*.

Metody dostępu do pól: *Podaj_Liczbe_Produktow*, *Ustaw_Liczbe_Produktow*, *Podaj_Numer_Zakupu*, *Ustaw_Numer_Zakupu* są identyczne, jak metody w module UCW1_4 (przykład 6 w rozdz. 2.2). Metoda *Podaj_Produkt* zwraca teraz wskaźnik pola *Produkt* klasy bazowej z drzewa *TProdukt*...

```
function TZakup.Podaj_Produkt:PProdukt;  
begin  
    Podaj_Produkt:= Produkt;  
end;
```

Metoda problemu *Ustaw_Zakup* może odwołać się do dowolnego dynamicznego obiektu zagnieżdżonego z drzewa *TProdukt*..dzięki wirtualnym metodom tych klas: *Nadaj_Gatunek*, *Nadaj_Podatek*, *Nadaj_Cene*. Podobnie jest przy obliczaniu podatku metodą *Oblicz_Podatek*, gdzie wywołuje wirtualną metodę *Podaj_Podatek*.

```
procedure TZakup.Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny,  
                               Indeks_Ilosci, N_Gatunek, N_Podatek:Integer);
```

```
begin  
    with Produkt^ do  
        begin  
            Nadaj_Nazwe(Nazwy[Indeks_Nazwy]);  
            Nadaj_Gatunek(N_Gatunek);  
            Nadaj_Podatek(N_Podatek);  
            Nadaj_Cene (Ceny[Indeks_Ceny]);  
        end;  
    Liczba_Produktow:= Ilosci[Indeks_Ilosci];  
end;
```

```
function TZakup.Oblicz_Kwote:Real;  
begin  
    Oblicz_Kwote:= Liczba_Produktow * Produkt^.Podaj_Cene;  
end;
```

```
function TZakup.Oblicz_Podatek:Real;  
var Pom:Real;  
begin  
    Pom:=Oblicz_Kwote*(Produkt^.Podaj_Podatek/100);  
    if Pom<0 then Oblicz_Podatek:=0  
    else Oblicz_Podatek:=Pom;  
end;
```

W metodzie problemu *Porownaj_Zakupy* po pobraniu *Produktu* od innego obiektu *TZakup* zleca się własnemu obiektowi *Produkt*, aby porównał za pomocą swojej metody *Porownaj_Towary* własne atrybuty z atrybutami komunikującego się obiektu. Zastosowano funkcję *TypeOf* w celu zbadania klasy obiektu *Produkt* oraz klasy tego obiektu w komunikującym się obiekcie *Zakup*, aby wykluczyć opisany dalej przypadek. Gdy wskaźnik obiektu *Produkt* posiada klasę bazową *TProdukt* lub *TProdukt_1*, po wywołaniu wirtualnej metody *Porownaj_Towary* nie porówna pola *Podatek* w przypadku, gdy komunikujący się obiekt *Zakup* posiada *Produkt* klasy *TProdukt_2*. W linii {1} podejmowana jest decyzja, czy obiekty powinny się porównywać

```
function TZakup.Porownaj_Zakupy (Zakup:PZakup):Boolean;
var Pomoc:PProdukt;
begin
  Pomoc:= Zakup^.Podaj_Produkt;
  if TypeOf(Pomoc^) = TypeOf(Produkt^) then                                     {1}
    Porownaj_Zakupy:=Produkt^.Porownaj_Towary(Pomoc)
  else   Porownaj_Zakupy:=False;
end;
```

Problem ten można rozwiązać również zmieniając w klasie *TProdukt* definicję metody *Porownaj_Towary*: likwidując metodę *TProdukt_2.Porownaj_Towary* oraz usuwając dyrektywę **virtual**. Atrybut *Podatek* obsługiwany jest przez wirtualną metodę *Podaj_Podatek*, a więc można było porównać te pola na poziomie klasy *TProdukt*. Po tej zmianie każdy obiekt, dziedziczący metodę *Porownaj_Towary*, wywołując ją wykona własną metodę *Podaj_Podatek*.

```
function TProdukt.Porownaj_Towary (Towar:PProdukt):Boolean;
begin
  Porownaj_Towary:=
    (Nazwa=Towar^.Podaj_Nazwe) and (Cena= Towar^.Podaj_Cene)
    and (Podaj_Podatek = Towar^.Podaj_Podatek);
end;
```

Wówczas metoda *Porownaj_Zakupy* miałyby postać:

```
function TZakup.Porownaj_Zakupy (Zakup:PZakup):Boolean;
var Pomoc:PProdukt;
begin
  Pomoc:= Zakup^.Podaj_Produkt;
  Porownaj_Zakupy:=Produkt^.Porownaj_Towary(Pomoc)
end;
```


Dodatkowa metoda *Wydruk_Zakupu* drukuje przy udziale zagnieżdżonego dynamicznego obiektu *Produkt* aktualne wartości atrybutów.

```
procedure TZakup.Wydruk_Zakupu;  
begin  
  Write(#13#10,Numer_Zakupu:3);  Produkt^.Wydruk_Produktu;  
  Write(#13#10,'Liczba produktu: ',Liczba_Produktow:3,  
    ' Kwota_Netto: ',          Oblicz_Kwote:8:2,  
    ' Podatek: ',Oblicz_Podatek:8:2);  
end;
```

Definicje metod zostały wykorzystane w programie **CW2_41** (przykład 16) wykorzystującego obiekty dynamiczne klasy *PZakup*. W częściach (1) i (2) programu tworzone są obiekty dynamiczne z drzewa *TProdukt..* oraz *TZakup* i przypisywane są konkretne wartości do ich atrybutów. Wskaźniki obiektów dynamicznych klasy *TZakup* przechowywane są w tablicy *Zakupy*. W linii (3) programu obiekty z tablicy *Zakupy* drukują się na ekranie, natomiast w linii (4) następuje usuwanie obiektów z pamięci. W każdej iteracji najpierw usuwany jest z pamięci obiekt z drzewa *TProdukt..*(każdy obiekt zagnieżdżony, usuwany destruktor *TProdukt.Usun*) oraz obiekt klasy *TZakup*.

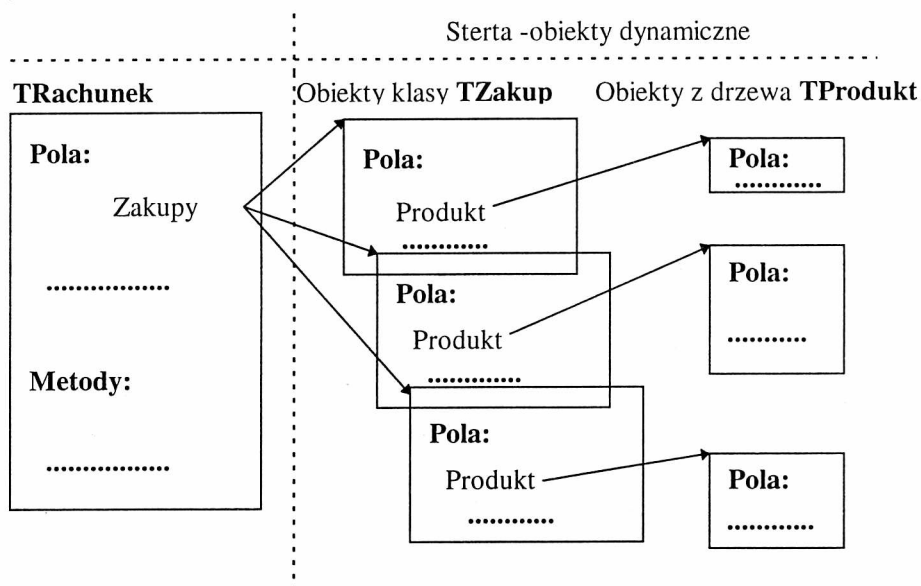
```
program CW2_41;  
uses Crt, UCW2_4;  
var Zakupy          : Tab_Zakupow;  
    Numer, I, Liczba : Integer;  
procedure Aktualizacja_Tablicy_Zakupow;  
begin  
  for I:=1 to Numer-1 do  
    if Zakupy[I]^.Porownaj_Zakupy(Zakupy[Numer]) then  
      begin  
        Liczba:= Zakupy[I]^.Podaj_Liczbe_Produktow+  
          Zakupy[Numer]^.Podaj_Liczbe_Produktow;  
        Zakupy[I]^.Ustaw_Liczbe_Produktow(Liczba);  
        Dispose(Zakupy[Numer],Usun);  
        Dec(Numer); Break;  
      end;  
  end;  
begin  
  ClrScr;  
  Numer:= 1;
```

```

Pamiec (SizeOf (TProdukt_1)+SizeOf (TZakup) +16); {1}
New (Zakupy[Numer], Inicjuj (New (PProdukt_1, Inicjuj)));
Zakupy[Numer]^..Ustaw_Zakup (1,2,2,2,22);
Zakupy[Numer]^..Ustaw_Numer_Zakupu (Numer);
Inc (Numer); Pamiec (SizeOf (TProdukt_2)+SizeOf (TZakup) +16); {2}
New (Zakupy[Numer], Inicjuj (New (PProdukt_2, Inicjuj)));
Zakupy[Numer]^..Ustaw_Zakup (1,2,2,2,22);
Zakupy[Numer]^..Ustaw_Numer_Zakupu (Numer);
Aktualizacja_Tablicy_Zakupow;
for I:= 1 to Numer do Zakupy[I]^..Wydruk_Zakupu; {3}
for I:= 1 to Numer do Dispose (Zakupy[I], Usun); {4}
end. {koniec programu CW2_41}

```

Zmianie uległa deklaracja i definicja klasy *TRachunek*. Na rys. 19 przedstawiono położenie w pamięci kolejno zagnieżdżonych obiektów: dynamicznych obiektów klasy *TZakup*, których wskaźniki przechowywane są w tablicy *Zakupy* oraz obiekty dynamiczne z drzewa *TProdukt*., których wskaźniki przechowują obiekty dynamiczne z klasy *TZakup*.



Przykład 17.

```

unit UCW2_5;
interface

                                                                 {deklaracje z modułu UCW2_4}

type
PRachunek = ^TRachunek;
TRachunek = object
  private Opis_Rachunku : Lan_1;
           Ostatni_Zakup : Integer;
           Zakupy        : Tab_Zakupow;
                                                                 {metody pomocnicze}
           procedure Przenumeruj_Zakupy;
           function Zakres_Rachunku (Numer:Integer):Boolean;
  public   constructor Inicjuj;
           destructor Usun;
                                                                 {metody dostępu do pól}

function Podaj_Opis_Rachunku:Lan_1;
procedure Ustaw_Opis_Rachunku(Nowy_Opis:Lan_1);
function Podaj_Ostatni_Zakup:Integer;
procedure Podaj_Zakupy (var Zakup:Tab_Zakupow);
function Podaj_Zakup(Numer_Zakupu:Integer):PZakup;
                                                                 {metody wykonujące zadanie klasy}

function Oblicz_Rachunek:Real;
procedure Aktualizacja_Rachunku;
procedure Dopisz_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                       N_Gatunek, N_Podatek:Integer; N_Produkt:PProdukt);
procedure Wykresl_Zakup (Numer:Integer);
procedure Wydruk_Rachunku;
                                                                 {metoda dodatkowa}
end;

implementation
                                                                 {definicje z modułu UCW2_4}
                                                                 {definicje metod klasy TRachunek}
end.
                                                                 {koniec modułu UCW2_5}

```

Metoda *Inicjuj* jest typu **constructor** i inicjuje pole obiektu. Metoda **destructor** *Usun* usuwa wszystkie obiekty umieszczone na stercie, tzn. w każdej iteracji najpierw usuwa obiekt dynamiczny z drzewa *TProdukt..*, zagnieżdżony w obiekcie *Zakupy[I]^* klasy *TZakup*, a następnie ten obiekt.

```

constructor TRachunek.Inicjuj;
begin
    Ostatni_Zakup:=0;
end;

destructor TRachunek.Usun;
var I:Integer;
begin
    for I:= 1 to Ostatni_Zakup do Dispose(Zakupy[I],Usun);
end;

```

Metody dostępu do pól oraz metody problemu mają takie samo znaczenie jak przy definiowaniu metod w module UCW1_5. Metoda prywatna *Zakres_Rachunku* oraz następujące metody dostępu do pól: *Podaj_Opis_Rachunku*, *Ustaw_Opis_Rachunku*, *Podaj_Ostatni_Zakup*, *Podaj_Zakupy* są identyczne jak w module UCW1_5 (przykład 7 w rozdziale 2.2).

Metoda *Podaj_Zakup* zwraca wskaźnik obiektu klasy *TZakup*, pobrany z tablicy *Zakupy*.

```

function TRachunek.Podaj_Zakup (Numer_Zakupu:Integer) :PZakup;
begin
    if not Zakres_Rachunku (Numer_Zakupu) then
        Podaj_Zakup:=Zakupy[Numer_Zakupu]
    else Podaj_Zakup:= nil;
end;

```

Metoda problemu *Wykresl_Zakup* usuwa wskaźnik wybranego obiektu dynamicznego typu *TZakup* z tablicy *Zakupy* po wcześniejszym usunięciu go z pamięci. Następnie porządkuje numerację kolejnych zakupów metodą *Przenumeruj_Zakupy*.

```

procedure TRachunek.Wykresl_Zakup (Numer:Integer);
begin
    if not Zakres_Rachunku (Numer) then
        begin
            Dispose(Zakupy[Numer], Usun);
            Move(Zakupy[Numer+1], Zakupy[Numer],
                (Ostatni_Zakup-Numer)*SizeOf(PZakup));
            Dec(Ostatni_Zakup); Przenumeruj_Zakupy;
        end;
    end;

```

```
procedure TRachunek.Przenumeruj_Zakupy;  
var I:Integer;  
begin  
  for I:=1 to Ostatni_Zakup do  
    Zakupy[I]^Ustaw_Numer_Zakupu(I);  
end;
```

Metoda *Aktualizacja_Rachunku* zwiększa liczbę produktu o wartość *Liczba* w *Zakupy[I]* (linia 2}, jeżeli nowa pozycja *Zakupy[Ostatni_Zakup]* posiada ten sam *Produkt* (linia 1) i usuwa nową pozycję (linia 3).}

```
procedure TRachunek.Aktualizacja_Rachunku;  
var I, Liczba:Integer;  
begin  
  for I:=1 to Ostatni_Zakup-1 do  
    if Zakupy[I]^Porownaj_Zakupy(Zakupy[Ostatni_Zakup]) then {1}  
      begin  
        Liczba:= Zakupy[I]^Podaj_Liczbe_Produktow +  
          Zakupy[Ostatni_Zakup]^Podaj_Liczbe_Produktow; {2}  
        Zakupy[I]^Ustaw_Liczbe_Produktow(Liczba); {2}  
        Dispose(Zakupy[Ostatni_Zakup], Usun); {3}  
        Dec(Ostatni_Zakup);  
        Break;  
      end;  
end;
```

Metoda problemu *Dopisz_Zakup* generuje nowy dynamiczny obiekt klasy *TZakup* (linia 1), wysyła do niego wartości atrybutów (linia 2), ustawia numer zakupu (linia 3) oraz podejmuje decyzję, czy ten zakup ma stanowić nową pozycję (linia 4).

```
procedure TRachunek.Dopisz_Zakup(Indeks_Nazwy, Indeks_Ceny,  
  Indeks_Ilosci, N_Gatunek, N_Podatek:Integer; N_Produkt:PProdukt);  
begin  
  Inc(Ostatni_Zakup);  
  if Zakres_Rachunku(Ostatni_Zakup) then  
    Dec(Ostatni_Zakup)  
  else begin  
    Pamiec(SizeOf(TZakup));  
    New(Zakupy[Ostatni_Zakup], Inicjuj(N_Produkt)); {1}  
    Zakupy[Ostatni_Zakup]^Ustaw_Zakup(Indeks_Nazwy,  
      Indeks_Ceny, Indeks_Ilosci, N_Gatunek, N_Podatek); {2}
```

```

        Ostatni_Zakup);           {3}
        Aktualizacja_Rachunku;    {4}
    end;
end;

```

Metoda problemu *Oblicz_Rachunek* oblicza wartość brutto wszystkich zakupów pobierając od każdego jego kwotę netto (*Policz_Kwote*) oraz wartość podatku (*Oblicz_Podatek*).

```

function TRachunek.Oblicz_Rachunek:Real;
var I:Integer; Suma:Real;
begin
    Suma:= 0;
    for I:= 1 to Ostatni_Zakup do
        Suma:=Suma+Zakupy[I]^Oblicz_Kwote+Zakupy[I]^Oblicz_Podatek;
    Oblicz_Rachunek:= Suma;
end;

```

Metoda dodatkowa *Wydruk_Rachunku* drukuje przy udziale swoich dynamicznych obiektów zagnieżdżonych wszystkie atrybuty oraz kwotę brutto.

```

procedure TRachunek.Wydruk_Rachunku;
var I:Integer;
begin
    Writeln (Opis_Rachunku);
    for I:=1 to Ostatni_Zakup do
        Zakupy[I]^Wydruk_Zakupu;
    Writeln(#13#10,'      Suma calkowita: ',Oblicz_Rachunek:10:2);
end;

```

Program **CW2_51** przedstawiony w przykładzie 17 wykorzystuje obiekt klasy *TRachunek*, wykonując następujące operacje:

- tworzenie pozycji rachunku (linia 1),
- wydruki (linia 2),
- usuwanie wybranej pozycji w rachunku (linia 3)
- usuwanie rachunku z pamięci (linia 4).

```

program CW2_51;
uses Crt, UCW2_5;
var Rachunek : PRachunek;
    I          : Integer;

```

```

begin
  ClrScr;
  Pamiec(SizeOf(TRachunek));
  New(Rachunek, Inicjuj);
  with Rachunek^ do
    begin
      Ustaw_Opis_Rachunku('20-08-1995');
      Pamiec(SizeOf(TProdukt));
      Dopisz_Zakup(1,2,2,2,22,New(PProdukt, Inicjuj));           {1}
      Pamiec(SizeOf(TProdukt_1));
      Dopisz_Zakup(2,1,2,3,7,New(PProdukt_1, Inicjuj));         {1}
      Pamiec(SizeOf(TProdukt_2));
      Dopisz_Zakup(2,2,2,1,22,New(PProdukt_2, Inicjuj));         {1}
      Pamiec(SizeOf(TProdukt));
      Dopisz_Zakup(1,2,3,2,22,New(PProdukt, Inicjuj));           {1}
      Wydruk_Rachunku;                                           {2}
      Readln;
      ClrScr;
      Wykresl_Zakup(2);                                           {3}
      Wydruk_Rachunku;                                           {2}
      Readln;
    end;
  Dispose(Rachunek, Usun);                                       {4}
end.                                                             {koniec programu CW2_51}

```

3.4. Prosty interfejs kontaktu z użytkownikiem

Strategia projektowania specjalizowanych obiektów, służących do kontaktów z człowiekiem powinna uwzględniać:

1. wymagania użytkownika;
2. hierarchię poleceń;
3. szczegóły interakcji;
4. tworzenie prototypów na kolejnych etapach projektowania;
5. projekt klas: okien głównych i składowych, wyboru poleceń, grafiki;
6. wykorzystanie standardowych interfejsów.

Opracowano dwie klasy, które mogą być wykorzystane do konstruowania prostego interfejsu:

- abstrakcyjną klasę *TMenu* umożliwiającą dokonanie czynności wyboru

- uniwersalną klasę *TWyświetlacz*, wykorzystywaną do wyświetlania na ekranie danych po przekształceniu na postać łańcuchową.

Abstrakcja klasy *TMenu* polega na tym, że zawsze wymagana jest deklaracja następcy, który zna opcje wyboru, tzn. potrafi je zapisać do tablicy *Opcje* i musi pokryć abstrakcyjną metodę *Utworz_Menu* własną metodą, która zna swoje 'menu'. Klasa *TWyświetlacz* jest dlatego klasą *uniwersalną*, ponieważ jest gotowa do wykorzystania.

Poniżej przedstawiono fragment modułu **UMENU_1** zawierającego podstawowe deklaracje i definicje tych klas oraz kilka pomocniczych procedur i funkcji.

Przykład 18.

```
unit UMENU_1;
interface
uses Crt;
const
  Maks_Opcji = 10;
type
  Lan_2 = string[20];
  Lan_3 = string[70];

PMenu := ^TMenu
TMenu = object
  Opcje           : array [1..Maks_Opcji] of Lan_2;
  Etykieta_Wyboru           : Lan_2;
  Ilosc, Wybor           : Integer;
  Menu_X1, Menu_Y1, Menu_X2, Menu_Y2           : Integer;
  Etykieta_X1, Etykieta_Y1, Etykieta_X2, Etykieta_Y2 : Integer;
  constructor Inicjuj (MenuX_1, MenuY_1, MenuX_2, MenuY_2,
                      EtX_1, EtY_1, EtX_2, EtY_2: Integer);
  destructor Usun;

                                {abstrakcyjna metoda wirtualna}
  procedure Utworz_Menu (var Rachunek); virtual;
  procedure Dodaj_Opcje (Nowa_Opcja: Lan_2);
  procedure Dodaj_Etykieta_Wyboru (Etykieta: Lan_2);
  procedure Wyświetl_Menu;
  function Podaj_Wybor: Integer;
  function Zakres_Wyboru (Wartosc: Integer): Boolean;
end;
```



```

PWyswietlacz = ^TWyswietlacz;
TWyswietlacz = object
    Komunikat                      : Lan_3;
    Naglowek                      : Lan_3;
    R_KomX1,R_KomY1,R_KomX2,R_KomY2 : Integer;
    constructor Inicjuj (KmX1,KmY1,KmX2,KmY2:Integer);
    destructor Usun;
    procedure Ustaw_Komunikat (Nowy_Komunikat:Lan_3);
    function Podaj_Komunikat:Lan_3;
    procedure Ustaw_Naglowek (Nowy_Naglowek:Lan_3);
    function Podaj_Naglowek:Lan_3;
    procedure Wyszwietl_Komunikat;
    procedure Wyszwietl_Naglowek;
end;

                                {deklaracja pomocniczych procedur i funkcji}
    function Nowy_Str(Wartosc:Integer):Lan_2;
    function Wyrownaj_Lancuch(Lan:Lan_2;Dlugosc:Byte):Lan_2;
    function Pauza:Char;
    procedure Pamiec(Rozmiar:Longint);

implementation

function Pauza:Char;
{Funkcja zwraca kod klawisza po wyczyszczeniu bufora klawiatury}
begin
    while Keypressed do Readkey;
    Pauza:= Readkey;
end;

function Nowy_Str(Wartosc:Integer):Lan_2;
{Funkcja zwraca łańcuch '-', jeżeli Wartosc jest ujemna, w przeciwnym wypadku zamienia ją na łańcuch cyfr reprezentujących tę Wartość.}
var Lan:Lan_2;
begin
    if Wartosc < 0 then Nowy_Str:= '-'
    else
        begin
            Str(Wartosc,Lan);
            Nowy_Str:= Lan;
        end;
    end;
end;

```

```
function Wyrownaj_Lancuch(Lan:Lan_2;Dlugosc:Byte):Lan_2;  
{Funkcja dodaje spacje z prawej strony łańcucha Lan, uzupełniając jego długość do wartości  
podanej w parametrze Dlugosc.}  
var Pom:string;  
begin  
  FillChar(Pom[1],Dlugosc,' ');  
  Pom[0]:=Char(Dlugosc-Length(Lan));  
  Wyrownaj_Lancuch:= Pom + Lan;  
end;  
  
procedure Pamiec(Rozmiar:Longint);  
{Procedura przerywa program, jeżeli obszar wolnej pamięci jest mniejszy od Rozmiar.}  
begin  
  if MaxAvail<Rozmiar then  
    begin  
      ClrScr;  
      Writeln('Brak Pamieci');  
      Pauza;  
      Halt;  
    end;  
end;  
  
procedure Ramka(X1,Y1,X2,Y2:Integer);  
{Procedura rysuje ramkę o podanych współrzędnych lewego górnego punktu X1,Y1 oraz  
prawego dolnego punktu: X2,Y2.}  
var I:Integer;  
begin  
  GotoXY(X1,Y1); Write(#201); GotoXY(X1,Y2); Write(#200);  
  GotoXY(X2,Y1); Write(#187); GotoXY(X2,Y2); Write(#188);  
  for I:= 1 to X2-X1-1 do  
    begin  
      GotoXY(X1+I,Y1);Write(#205);GotoXY(X1+I,Y2);Write(#205);  
    end;  
  for I:= 1 to Y2-Y1-1 do  
    begin  
      GotoXY(X1,Y1+I);Write(#186);GotoXY(X2,Y1+I);Write(#186);  
    end;  
end;
```

```
{-----TMenu-----}
constructor TMenu.Inicjuj (MenuX_1, MenuY_1, MenuX_2, MenuY_2,
                          EtX_1, EtY_1, EtX_2, EtY_2: Integer);
{Constructor przypisuje współrzędne ramki, w której wyświetlane są opcje wyboru (linie 1) oraz łańcuch wprowadzania wybranej opcji (linia 2).}
begin
  Ilosc:= 0;
  Menu_X1:= MenuX_1; Menu_Y1:= MenuY_1;                                {1}
  Menu_X2:= MenuX_2; Menu_Y2:= MenuY_2;                                {1}
  Etykieta_X1:= EtX_1; Etykieta_Y1:= EtY_1;                            {2}
  Etykieta_X2:= EtX_2; Etykieta_Y2:= EtY_2;                            {2}
end;

destructor TMenu.Usun;
begin
end;

procedure TMenu.Utworz_Menu (var Rachunek);
begin
end;

function TMenu.Zakres_Wyboru (Wartosc: Integer): Boolean;
{Metoda sprawdza poprawność wybranego indeksu tablicy opcji.}
begin
  Zakres_Wyboru:= (Wartosc>Maks_Opcji) or (Wartosc>Ilosc)
                  or (Wartosc<=0);
end;

procedure TMenu.Dodaj_Opcje (Nowa_Opcja: Lan_2);
{Metoda zwiększa Ilosc opcji i przypisuje odpowiadający jej łańcuch znaków do tablicy Opcje.}
begin
  Inc(Ilosc);
  if Zakres_Wyboru(Ilosc) then
    Dec(Ilosc)
  else
    Opcje[Ilosc]:= Nowa_Opcja;
end;
```

```

procedure TMenu.Dodaj_Etykieta_Wyboru (Etykieta:Lan_2);
{Metoda przypisuje łańcuch zachęcający do wyboru opcji.}
begin
    Etykieta_Wyboru:= Etykieta;
end;

procedure TMenu.Wyswietl_Menu;
{Metoda rysuje ramkę, w której drukuje na ekranie łańcuchy opcji oraz ramkę, w której drukuje na ekranie etykietę wyboru.}
var I:Integer;
begin
    Ramka(Menu_X1,Menu_Y1,Menu_X2,Menu_Y2);
    Ramka(Etykieta_X1,Etykieta_Y1,Etykieta_X2,Etykieta_Y2);
    GotoXY(Menu_X1+1,Menu_Y1+1);
    for I:= 1 to Ilosc do Write(I,' ',Opcje [I],' ');
    GotoXY(Etykieta_X1+1,Etykieta_Y1+1);
    Write(Etykieta_Wyboru);
end;

function TMenu.Podaj_Wybor:Integer;
{Metoda pobiera z klawiatury numer opcji (linia 1), następnie sprawdza poprawność i zakres wprowadzonego znaku (linie 2). Jeżeli znak jest poprawnie wprowadzony, zapamięta Wybór i zwraca jego wartość (linie 3).}
var Spr:Boolean; Pom:Char; Blad:Integer;
begin
    GotoXY(Length(Etykieta_Wyboru)+1, WhereY);
    repeat
        Pom:= Pauza;
        Val(Pom,Wybor,Blad); {1}
        Spr:= not(Blad=0) or Zakres_Wyboru(Wybor); {2}
        if Spr then
            GotoXY (Length(Etykieta_Wyboru)+1, WhereY);
    until not Spr;
    Write(Wybor);
    Pauza;
    if Wybor = Ilosc then {3}
        Podaj_Wybor:= 0
    else {3}
        Podaj_Wybor:= Wybor;
end;

```

```
{-----TWyswietlacz-----}
constructor TWyswietlacz.Inicjuj (KmX1, KmY1, KmX2, KmY2: Integer
{Constructor inicjuje współrzędne ramki wyświetlacza.}
begin
    Komunikat:= '';
    R_KomX1:= KmX1; R_KomY1:= KmY1;
    R_KomX2:= KmX2; R_KomY2:= KmY2;
end;

destructor TWyswietlacz.Usun;
begin
end;

procedure TWyswietlacz.Ustaw_Komunikat (Nowy_Komunikat: Lan_3);
{Metoda przypisuje łańcuch zawierający dane do wyświetlenia na ekranie.}
begin
    Komunikat:= Nowy_Komunikat;
end;

function TWyswietlacz.Podaj_Komunikat: Lan_3;
begin
    Podaj_Komunikat:= Komunikat;
end;

procedure TWyswietlacz.Ustaw_Naglowek (Nowy_Naglowek: Lan_3);
{Metoda przypisuje łańcuch z informacją dodatkową o wyświetlanych łańcuchach w komunikacie. Informacja ta będzie drukowana nad Komunikatem w tej samej ramce.}
begin
    Naglowek:= Nowy_Naglowek;
end;

function TWyswietlacz.Podaj_Naglowek: Lan_3;
begin
    Podaj_Naglowek:= Naglowek;
end;

procedure TWyswietlacz.Wyswietl_Naglowek;
begin
    Ramka (R_KomX1, R_KomY1, R_KomX2, R_KomY2);
    GotoXY (R_KomX1+1, R_KomY1+1); Write (Naglowek);
end;
```

```

procedure TWyswietlacz.Wyswietl_Komunikat;
begin
  Ramka (R_KomX1, R_KomY1, R_KomX2, R_KomY2);
  GotoXY(R_KomX1+1, R_KomY1+3);
  Write(Komunikat);
end;
end.

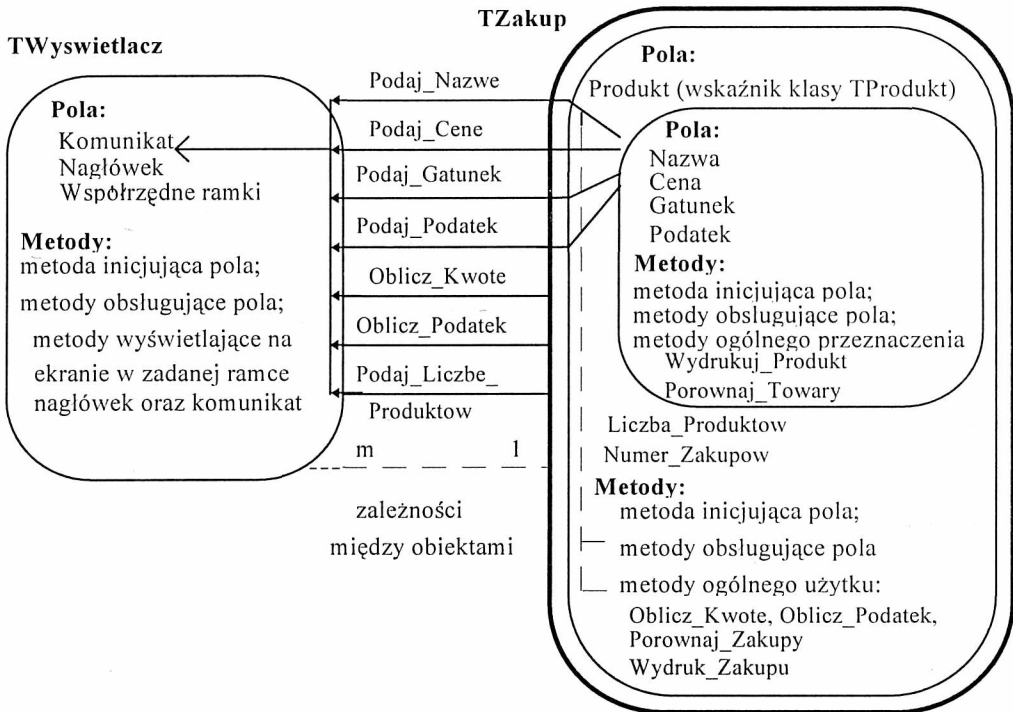
```

{koniec modułu **UMENU_1**}

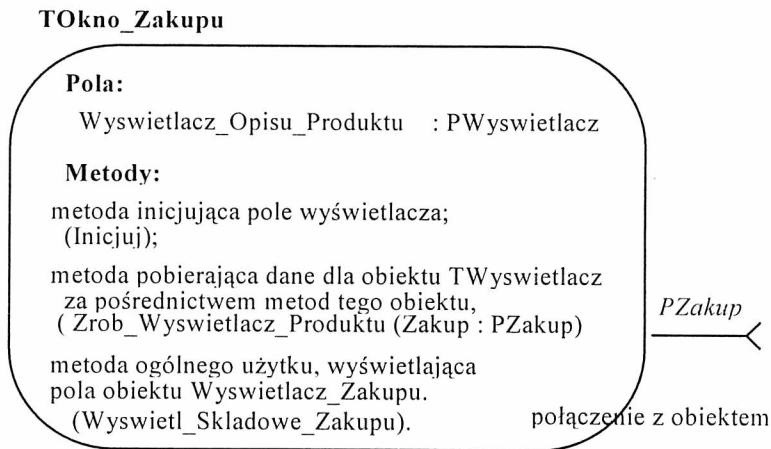
Na rys. 20 pokazano, jakie czynności powinien wykonać prosty interfejs, który nazwiemy oknem.

Czynności wykonywane przez okno:

1. pobieranie od obiektu *TZakup* danych: wartości jego atrybutów oraz wartości netto oraz podatku, wyliczonych przez obiekt na żądanie okna;
2. przekształcanie danych na postać znakową;
3. wydruk na ekranie danych wraz z nagłówkiem w wyznaczonych polach



Rys. 20. Komunikacja między obiektami klasy *TZakup* a klasą *TWyswietlacz*



Rys. 21. Schemat klasy TOkno_Zakupu

Na rys. 21 pokazano projekt okna, które ‘potrafi’ wyświetlać na ekranie dane dowolnego obiektu *TZakup*, komunikując się z nim poprzez listę parametrów formalnych metod klasy *TOkno_Zakupu*. Aby spełnić ten cel, klasa posiada dynamiczny obiekt zagnieżdżony klasy *TWyswietlacz*.

Przykład 19.

```
unit UMENU1_1;
interface
uses UMENU_1,UCW2_4, Crt;
type
  POkno_Zakupu = ^TOkno_Zakupu;
  TOkno_Zakupu = object
    Wyswietlacz_Opisu_Produktu : PWyswietlacz;
    constructor Inicjuj;
    destructor Usun;
    procedure Zrob_Wyswietlacz_Produktu(Zakup:PZakup);
    procedure Wyswietl_Skladowe_Zakupu;
  end;
implementation
constructor TOkno_Zakupu.Inicjuj;
begin
  Pamiec(SizeOf(TWyswietlacz));
  New(Wyswietlacz_Opisu_Produktu, Inicjuj(1,1,79,10));
end;
```

```

destructor TOkno_Zakupu.Usun;
begin
    Dispose(Wyswietlacz_Opisu_Produktu, Usun);
end;

procedure TOkno_Zakupu.Zrob_Wyswietlacz_Produktu(Zakup:PZakup);
{Metoda pobiera atrybuty od obiektu Zakup^ i zamienia je na łańcuchy znaków: A,, B, C, D,
E, F, G tworząc z nich komunikat (linia 2). Ustala stały nagłówek przypisując mu łańcuch H
(linia 2).}
var A,B,C,D,E,F,G,H : Lan_3;
    Pom                : Lan_2;
    Produkt            : PProdukt;
begin
    Produkt:= Zakup^.Podaj_Produkt;
    A:= Wyrownaj_Lancuch(Produkt^.Podaj_Nazwe,20);
    Str(Produkt^.Podaj_Cene:8:2, B);
    C:=Wyrownaj_Lancuch(Nowy_Str(Produkt^.Podaj_Gatunek),4);
    D:=Wyrownaj_Lancuch(Nowy_Str(Produkt^.Podaj_Podatek),4);
    Str(Zakup^.Podaj_Liczbe_Produktow:10, E);
    Str(Zakup^.Oblicz_Kwote:12:2, F);
    Str(Zakup^.Oblicz_Podatek:12:2,G);
    H:= '          Nazwa      Cena Gat Pod Licz_Prod ' +
        'Kwota_Netto  Kwota_Pod';
    Wyswietlacz_Opisu_Produktu^.Ustaw_Naglowek(H);
    Wyswietlacz_Opisu_Produktu^.Ustaw_Komunikat(A+B+C+D+E+F+G)
end;

procedure TOkno_Zakupu.Wyswietl_Skladowe_Zakupu;
{Metoda zleca polu Wyswietlacz_Opisu_Produktu wydrukowanie na ekranie komunikatu i
nagłówka w ramce.}
begin
    ClrScr;
    Wyswietlacz_Opisu_Produktu^.Wyswietl_Komunikat;
    Wyswietlacz_Opisu_Produktu^.Wyswietl_Naglowek; Pauza;
end;
end.

```

{koniec modułu UMENU1_1}

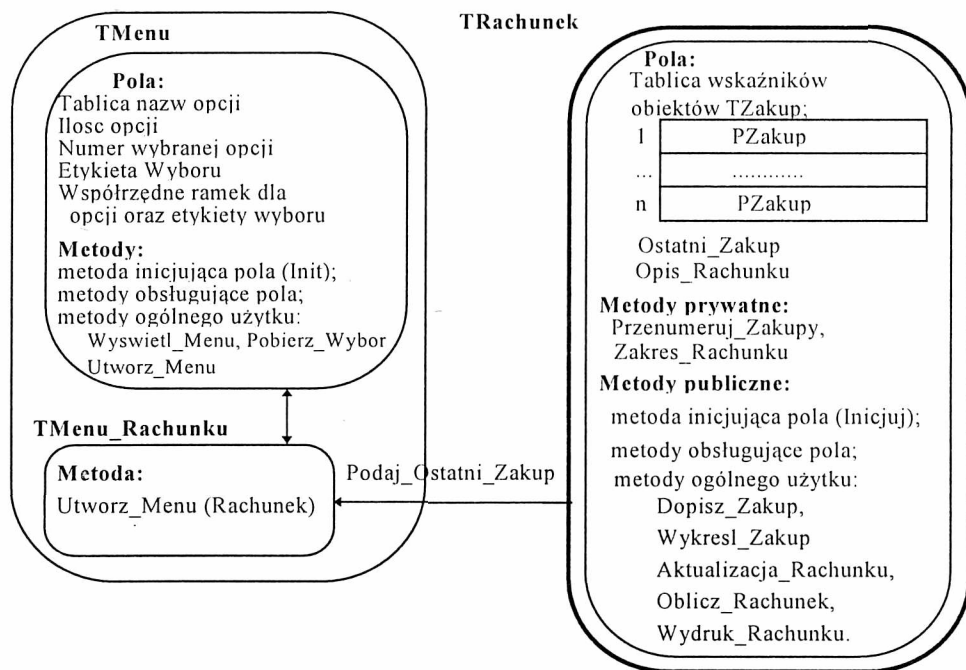
W przykładzie 19 program **CW2_42** nawiązuje do programu z przykładu 16, lecz wydruk zakupów odbywa się za pomocą interfejsu *Okno_Zakupu* w ostatnim fragmencie programu - linia {1 i 2}.

Kolejnym przykładem interfejsu jest okno zapewniające kontakt człowieka z obiektem klasy *TRachunek*. Na rys. 22 pokazano główne czynności, jakie powinno wykonać okno.

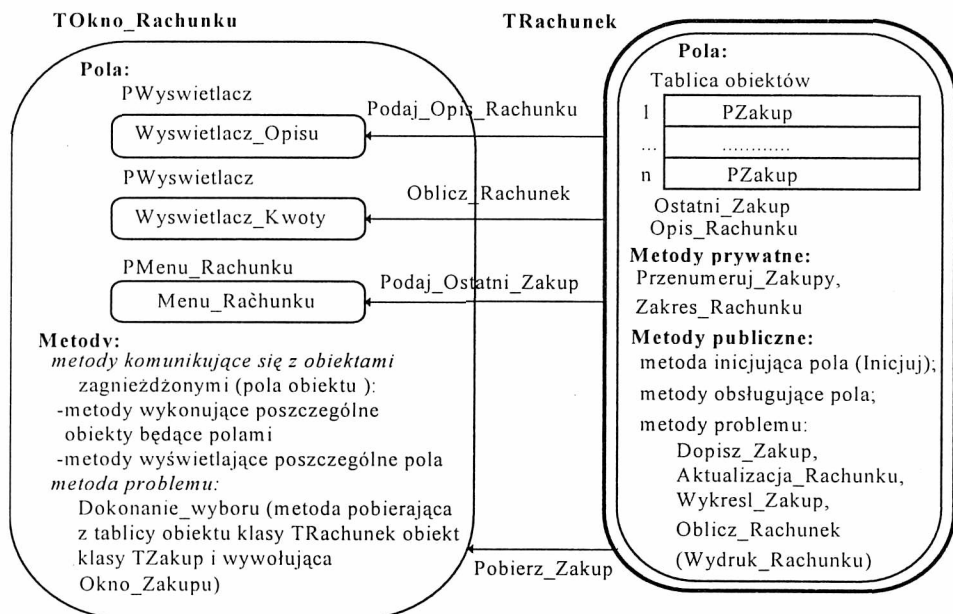
Sformułowano następujące zadania okna klasy *TOkno_Rachunku*:

1. pobieranie od obiektu *TRachunek* danych: wartości jego pól oraz wartości kwot obliczonych przez obiekt na żądanie okna;
2. przekształcanie danych na postać znakową;
3. wydruk danych wraz z nagłówkiem w wyznaczonych polach.
4. wybór pozycji rachunku (czyli obiektu klasy *TZakup*);
5. utworzenie okna klasy *TOkno_Zakupu* i przekazanie wybranego obiektu klasy *TZakup* (dalsza akcja wykonywana przez *Okno_Zakupu*);

Na rys. 23 przedstawiono projekt klasy *TOkno_Rachunku*.



Rys. 22. Komunikacja między obiektami klasy *TRachunek* a klasą *TMenu_Rachunku*



Rys. 23. Komunikacja między obiektami klasy *TRachunek* i klasą *TWyswietlacz* oraz klasą *TMenu_Rachunku*

Poniżej przedstawiono deklaracje klasy *TOkno_Rachunku*, które zawiera dwa obiekty typu *TWyswietlacz* oraz obiekt służący do wyboru opcji typu *TMenu_Rachunku* (następca klasy *TMenu*, tworzący tablicę opcji dedykowanej operacjom na obiektach klasy *TRachunek*). Jest to opis rozbudowy modułu **UMENU1_1** do modułu **UMENU1_2**.

Przykład 20.

```
unit UMENU1_2;
interface
uses UMENU_1, UCW2_5, Crt;

type
PMenu_Rachunku = ^TMenu_Rachunku;
TMenu_Rachunku = object(TMenu)
    procedure Utworz_Menu(var Rachunek); virtual;
end;
```

```

Pokno_Rachunku = ^TOkno_Rachunku;
TOkno_Rachunku = object
  Wyświetlacz_Opisu : PWyświetlacz;
  Wyświetlacz_Kwoty : PWyświetlacz;
  Menu_Rachunku      : PMenu_Rachunku;
  constructor Inicjuj;
  destructor Usun;
  procedure Zrob_Wysw_Opisu (Rachunek: PRachunek);
  procedure Zrob_Wysw_Kwoty (Rachunek: PRachunek);
  procedure Zrob_Menu (Rachunek: PRachunek);
  procedure Zrob_Skladowe (Rachunek: PRachunek);
  procedure Wyświetl_Skladowe;
  procedure Dokonanie_Wyboru (Rachunek: PRachunek);
end;

                                     {deklaracja klasy TOkno_Zakupu z modułu UMENU1_1}
implementation
{-----TOkno_Zakupu-----}
                                     {definicje metod klasy TOkno_Zakupu z modułu UMENU1_1}

{-----TMenu_Rachunku-----}
procedure TMenu_Rachunku.Utworz_Menu (var Rachunek);
{Metoda pobiera od dynamicznego obiektu klasy TRachunek liczbę opcji (1) i nadaje im
konkretne nazwy (linia 2).}
  var I, J: Integer;
  begin
    Ilosc:=0;
    J:=PRachunek (Rachunek) ^.Podaj_Ostatni_Zakup;           {1}
    for I:=1 to J do
      Dodaj_Opcje ('Zakup_' + Chr (I+48));                     {2}
      Dodaj_Opcje ('Koniec');
      Etykieta_Wyboru:='Wybierz:'
    end;

{-----TOkno_Rachunku-----}
constructor TOkno_Rachunku.Inicjuj;
{Metoda tworzy na stercie dynamiczne obiekty zagnieżdżone TWyświetlacz służące do
wyświetlania danych obiektu klasy TRachunek (linie 1) oraz TMenu_Rachunku tworzący pole
wyboru (linia 2).}

```

```
begin
  Pamiec (SizeOf (TWyswietlacz)) ;
  New (Wyswietlacz_Opisu, Inicjuj (1, 21, 40, 25)) ;           {1}
  Pamiec (SizeOf (TWyswietlacz)) ;
  New (Wyswietlacz_Kwoty, Inicjuj (41, 21, 79, 25)) ;         {1}
  Pamiec (SizeOf (TMenu_Rachunku)) ;
  New (Menu_Rachunku, Inicjuj (1, 1, 79, 3, 1, 4, 79, 6)) ;    {2}
end;

destructor TOkno_Rachunku.Usun;
begin
  Dispose (Wyswietlacz_Opisu, Usun) ;
  Dispose (Wyswietlacz_Kwoty, Usun) ;
  Dispose (Menu_Rachunku, Usun) ;
end;

procedure TOkno_Rachunku.Zrob_Wysw_Kwoty (Rachunek: PRachunek) ;
{Metoda pobiera od obiektu Rachunek^ jego kwotę brutto (linia 1) i przekazuje tę wartość do
Wyswietlacza_Kwoty wraz z ustalonym nagłówkiem 'Suma: '(linie 2,3)}
  var Lan : Lan_1;
begin
  Str (Rachunek^.Oblicz_Rachunek:10:2, Lan) ;                 {1}
  Wyswietlacz_Kwoty^.Ustaw_Naglowek ('Suma: ') ;             {2}
  Wyswietlacz_Kwoty^.Ustaw_Komunikat (Lan) ;                 {3}
end;

procedure TOkno_Rachunku.Zrob_Wysw_Opisu (Rachunek: PRachunek) ;
{Metoda pobiera od obiektu Rachunek^ jego opis (linia 2), przekazuje do Wyswietlacza_Opisu
(linia 3) oraz ustala jego nagłówek jako 'Opis Rachunku' (linia 1). }
  var Pom : string;
begin
  Wyswietlacz_Opisu^.Ustaw_Naglowek ('Opis Rachunku: ') ;    {1}
  Pom:= Rachunek^.Podaj_Opis_Rachunku;                        {2}
  Wyswietlacz_Opisu^.Ustaw_Komunikat (Pom) ;                 {3}
end;

procedure TOkno_Rachunku.Zrob_Menu (Rachunek: PRachunek) ;
{Metoda 'każe' obiektowi Menu_Rachunku utworzyć dane do wyświetlania Menu.}
begin
  Menu_Rachunku^.Utworz_Menu (Rachunek) ;
end;
```

```
procedure TOkno_Rachunku.Zrob_Skladowe(Rachunek:PRachunek);
{Metoda 'każe' obiektom zagnieżdżonym ustawić swoje atrybuty na podstawie danych pobranych od obiektu Rachunek^.}
```

```
begin
```

```
    Zrob_Wysw_Opisu(Rachunek);
```

```
    Zrob_Wysw_Kwoty(Rachunek);
```

```
    Zrob_Menu(Rachunek);
```

```
end;
```

```
procedure TOkno_Rachunku.Wyświetl_Skladowe;
{Metoda 'każe' wydrukować swoje dane obiektom zagnieżdżonym.}
```

```
begin
```

```
    ClrScr;
```

```
    Wyświetlacz_Opisu^.Wyświetl_Komunikat;
```

```
    Wyświetlacz_Opisu^.Wyświetl_Naglowek;
```

```
    Wyświetlacz_Kwoty^.Wyświetl_Komunikat;
```

```
    Wyświetlacz_Kwoty^.Wyświetl_Naglowek;
```

```
    Menu_Rachunku^.Wyświetl_Menu;
```

```
end;
```

```
procedure TOkno_Rachunku.Dokonanie_Wyboru(Rachunek:PRachunek);
{Metoda realizuje interakcję z użytkownikiem programu. Obiekt Menu_Rachunku pobiera numer opcji od użytkownika (linia 2) i przekazuje związany z tym numerem obiekt Zakup^ (linia 2) do Okna_Zakupu^ (linia 3). Następnie wyświetla dane na ekranie (linia 4). Działanie metody kończy się w chwili, gdy użytkownik wybierze ostatnią opcję (Pom=0) (linia 1).}.
```

```
var OKno_Zakupu : POkno_Zakupu;
```

```
    Pom          : Integer;
```

```
    Zakup        : PZakup;
```

```
begin
```

```
    Pamiec(SizeOf(TOkno_Zakupu));
```

```
    OKno_Zakupu:=New(POkno_Zakupu, Inicjuj);
```

```
    Zrob_Skladowe(Rachunek);
```

```
repeat
```

```
    Wyświetl_Skladowe;
```

```
    Pom:= Menu_Rachunku^.Podaj_Wybor; {1}
```

```
    Zakup:= Rachunek^.Podaj_Zakup(Pom); {2}
```

```
if Zakup <> nil then
```

```
    begin
```

```
        Okno_Zakupu^.Zrob_Wyświetlacz_Produktu(Zakup); {3}
```

```
        Okno_Zakupu^.Wyświetl_Skladowe_Produktu; {4}
```

```
    end;
```

```

until Pom = 0;
Dispose(Okno_Zakupu, Usun);
ClrScr;
end;
end.                                     {koniec modułu UMENU1_2}

```

Program **CW2_52** w przykładzie 20 nawiązuje do programu **CW2_51** z przykładu 17, umożliwia jednak prostą interakcję z użytkownikiem programu. Cały program wykonano za pomocą metod obiektu klasy *TOkno*, stanowiącego rodzaj ‘pojemnika’ obiektów problemu: klasy *TRachunek* oraz klasy *TOkno_Rachunku*.

```

program CW2_52;
uses Crt, UCW2_5, UMENU1_2, UMENU_1;
type
TOkno = object
    Rachunek          : PRachunek;
    Okno_Rachunku     : POkno_Rachunku;
    procedure Inicjuj;
    procedure Wykonaj;
    procedure Usun;
end;

procedure TOkno.Inicjuj;
{Metoda tworzy dynamiczny obiekt klasy TRachunek wraz z danymi (linia 1) oraz obiekt klasy
TOkno_Rachunku przeznaczone do kontaktu użytkownika z rachunkiem (linia 2).}
begin
    Pamiec(SizeOf(TRachunek));
    {New(Rachunek, Inicjuj(3)); postać instrukcji dla programu CW3_2 ze str. 120}
    New(Rachunek, Inicjuj);                                     {1}
    Rachunek^.Ustaw_Opis_Rachunku('20-08-1995');             {1}
    Pamiec(SizeOf(TProdukt_1));
    Rachunek^.Dopisz_Zakup(1, 2, 2, 2, 22, New(PProdukt_1, Inicjuj)); {1}
    Pamiec(SizeOf(TProdukt));
    Rachunek^.Dopisz_Zakup(2, 2, 2, 1, 22, New(PProdukt, Inicjuj)); {1}
    Pamiec(SizeOf(TProdukt_2));
    Rachunek^.Dopisz_Zakup(2, 2, 2, 1, 22, New(PProdukt_2, Inicjuj)); {1}
    Pamiec(SizeOf(TOkno_Rachunku));
    New(Okno_Rachunku, Inicjuj);                               {2}
end;

```

```
procedure TOkno.Wykonaj;  
{Metoda wykonuje główną interakcję z użytkownikiem za pośrednictwem obiektu  
Okno_Rachunku.}  
  begin  
    Okno_Rachunku^.Dokonanie_Wyboru(Rachunek) ;  
  end;  
  
procedure TOkno.Usun;  
  begin  
    Dispose(Rachunek, Usun) ;  
    Dispose(Okno_Rachunku, Usun) ;  
  end;  
  
var Okno_Glowne : TOkno;  
begin  
  Okno_Glowne.Inicjuj;  
  Okno_Glowne.Wykonaj;  
  Okno_Glowne.Usun;  
end. {koniec programu CW2_52}
```

W module **UMENU1_2** przedstawiono uniwersalne znaczenie klas *TWyswietlacz* oraz *TMenu*, z których utworzono różne interfejsy kontaktu z człowiekiem:

- *TOkno_Zakupu*,
- *TOkno_Rachunku*.

4. Uniwersalne struktury danych - listy, kolekcje

Ten etap projektowania jest poświęcony definiowaniu klas o charakterze struktur danych. W poprzednich rozdziałach obiekty z klasy *TZakup* były przechowywane w klasie *TRachunek* w postaci tablicy wskaźników. Aktualizacja tych danych wymaga wykonania następujących operacji na zawartości tablicy: wstawiania lub usuwania wskaźników itp. Te czynności powodują konieczność zdefiniowania w każdej klasie wykorzystującej tablicę wskaźników takich samych metod. Korzystniejszym rozwiązaniem jest stosowanie klas przeznaczonych do wykonania tych zadań. Takimi klasami mogą być następujące wzorcowe struktury danych:

- struktury dynamiczne;
- kolekcje.

4.1. Struktury dynamiczne obiektów - listy jednokierunkowe

W przypadku, gdy należy przechowywać w pamięci nie znaną liczbę obiektów z tego samego lub różnych drzew dziedziczenia, można wykorzystać struktury dynamiczne np. stosy, kolejki, listy, drzewa itd. W rozdziale 4.1 opisano obiektową listę nieuporządkowaną oraz przedstawiono przykład programowy.

Podstawowe zagadnienia

Klasa realizująca listę jednokierunkową nieuporządkowaną dostarcza podstawowe narzędzia do tworzenia i aktualizacji struktury. Natomiast czynności mające charakter problemowy, będą zawierać klasy - specjalizacje, dziedziczące po klasie *TLista* (rys. 24). Deklarację klasy oraz definicję metod zawiera moduł **STRUK3_1**.

Deklaracja uniwersalnej klasy typu *TLista*, umożliwiającą tworzenie listy złożonej z różnych obiektów, dziedziczących po *TLista*, jest następująca:

Przykład 21.

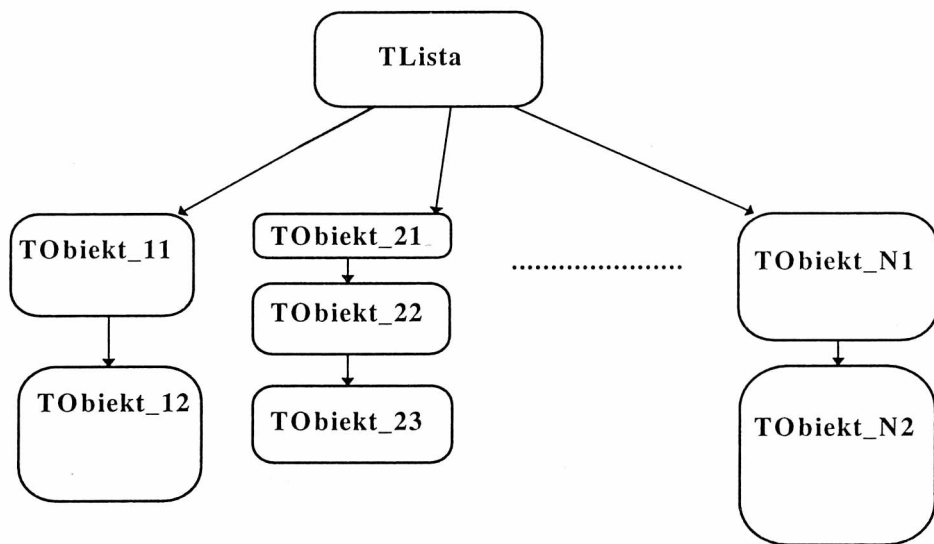
```
unit STRUK3_1;  
interface  
uses Crt, UMENU_1;  
PLista = ^TLista;
```

```

TLista =
  object
    Nowy : PLista;
    constructor Inicjuj;
    destructor Usun;virtual;
                                     {metody podstawowe obsługujące strukturę}
    procedure Dodaj (Nastepny:PLista);
    function Usun_Element (Numer : Integer) : Boolean;
    function Podaj_Element (Numer_Elementu:Integer):PLista;
                                     {metody dodatkowe }

    procedure Wydruk_Elementu;virtual;
    procedure Wydrukuj_liste;
  end;
implementation
                                     {definicje metod klasy TLista;
                                     {koniec modulu STRUK3_1}
end.

```

Rys. 24. Schemat dziedziczenia klas w drzewie *TLista*...

Na rys. 25 przedstawiono przykład struktury dynamicznej typu lista, realizowanej na bazie klasy *TLista*. Na rys. 26 pokazano przebieg usuwania z listy elementu o zadanym numerze.

W klasie *TLista* zadeklarowano dwie metody wirtualne: *Wydrukuj_Element* oraz **destructor** *Usun*. Obie metody przewidują u swoich następców inny typ pól do wyświetlenia lub usunięcia z pamięci, a więc muszą dostosować się do aktualnego obiektu za pomocą polimorfizmu.

Pozostałe metody zakładają, że wskaźniki następców można przypisać wskaźnikowi korzenia, którym właśnie jest typ bazowy *PLista*. Metody wynikające ze specjalizacji obiektu aktualnego, podstawianego do listy, definiowane są w klasie dziedziczącej po *TLista*.

W tekście metody *Usun_Element* użyto parametru **self**. Parametr **self** zawsze niejawnie przekazywany jest na końcu listy parametrów formalnych każdej metody przez zmienną nieokreślonego typu. Za pomocą tego parametru przekazywany jest obiekt aktualnie wywołujący tę metodę. Oznacza to, że typ nagłówka każdej metody można przedstawić następująco:

```
metoda_procedura = procedure (lista_parametrow; var self);
metoda_funkcja = function (lista_parametrow; var self) : typ_funkcji.
```

Jeżeli wykonujemy metodę:

```
Lista_Objektow^.Usun_Element(2),
```

to zostanie wykonana instrukcja:

```
Poprzedni := @self;
```

co oznacza, że zmienna *Poprzedni* po tym przypisaniu będzie zawierać adres obiektu aktualnie wywołującego metodę, czyli obiektu *Lista_Objektow^*.

Definicje metod klasy *TLista* są następujące

implementation

{moduł *STRUK3_1*}

constructor *TLista*.Inicjuj;

begin

Nowy := nil;

end;

destructor TLista.Usun;

{Metoda usuwa wszystkie elementy z listy (linia 2), zaczynając się od pola Nowy. Procedura Dispose usuwa każdy element listy prawidłowo, ponieważ równocześnie wywoływany jest ich destruktor zadeklarowany w każdej klasie dziedziczącej po TLista jako wirtualny. W przykładzie 24 pokazano definicje destruktorów TZakup.Usun oraz TRachunek.Usun dla elementów dziedziczących po klasie TLista.}

begin

if Nowy <> **nil** **then** Dispose(Nowy, Usun); {2}

end;

procedure TLista.Dodaj(Nastepny:PLista);

{Metoda wstawia nowy element Nastepny do pola Nowy obiektu wywołującego metodę (linia 1) , jeśli lista jest pusta. Jeśli nie, szuka końca struktury (linia 2) i wstawia wskaźnik Nastepny do pola Nowy ostatniego elementu listy.}

var Pom : PLista;

begin

if Nowy = **nil** **then** Nowy := Nastepny {1}

else

begin

Pom:=Nowy;

while Pom^.Nowy <> **nil** **do** Pom:=Pom^.Nowy; {2}

Pom^.Nowy:=Nastepny; {3}

end;

end;

procedure TLista.Wydruk_Elementu;

{Metoda abstrakcyjna, pokrywana w klasach dziedziczących. Umożliwia, na poziomie klasy TLista, wydrukowanie dowolnego następcy w metodzie Wydrukuj_Liste.}

begin

end;

procedure TLista.Wydrukuj_liste;

var Pom : PLista;

begin

Pom := Nowy;

while Pom <> **nil** **do**

begin

Pom^.Wydruk_Elementu;

Pom := Pom^.Nowy;

end;

end;

```
function TLista.Podaj_Element(Numer_Elementu:Integer):PLista;  
{Metoda zwraca wskaźnik elementu listy o numerze Numer_Elementu lub wartość nil, jeżeli  
takiego elementu nie ma na liście.}
```

```
var I          : Integer;  
    Biezacy    : PLista;  
begin  
    I := 1;  
    Biezacy := Nowy;  
    while (Biezacy <> nil) and (I <> Numer_Elementu) do  
        begin  
            Biezacy := Biezacy^.Nowy;  
            Inc(I);  
        end;  
    Podaj_Element := Biezacy;  
end;
```

```
function TLista.Usun_Element(Numer:Integer):Boolean;  
{Metoda usuwa element o numerze Numer (linie 2 i 3), jeśli element o takim numerze istnieje  
(linia 1). Zwraca wartość True po usunięciu elementu lub False w przeciwnym wypadku.}
```

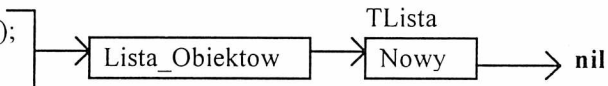
```
var Poprzedni, Biezacy : PLista;  
    I                  : Integer;  
begin  
    Usun_Element:=True;  
    I:=1;  
    Poprzedni:=@self;  
    Biezacy:=Nowy;  
    while (Biezacy <> nil) and (I <> Numer) do {1}  
        begin  
            Poprzedni:=Biezacy;  
            Biezacy:=Biezacy^.Nowy;  
            Inc(I);  
        end;  
    if Biezacy <> nil then  
        begin  
            Poprzedni^.Nowy := Biezacy^.Nowy; {2}  
            Dispose(Biezacy,Usun); {3}  
        end  
    else Usun_Element := False;  
end;
```

Podstawowa deklaracja:

var Lista_Objektow : PLista; {wskaźnik obiektu rozpoczynającego listę, bez danych}

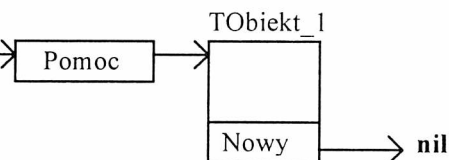
1. Założenie listy.

New(Lista_Objektow, Inicjuj);
Nowy := nil;

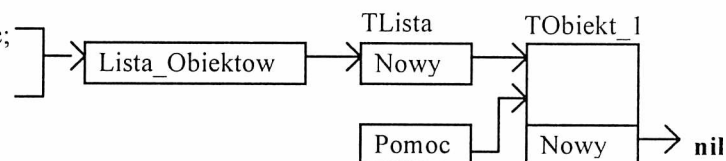


2. Dodanie pierwszego elementu do listy.

Pomoc := New(PObiekt_1, Inicjuj);

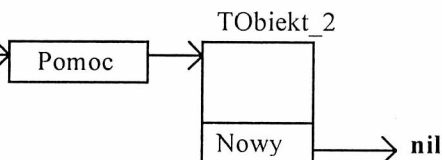


Nowy := Pomoc;

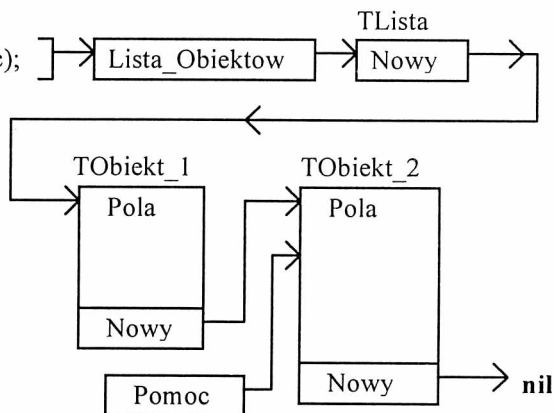


2. Dodanie elementu na końcu listy.

Pomoc := New(PObiekt_2, Inicjuj);

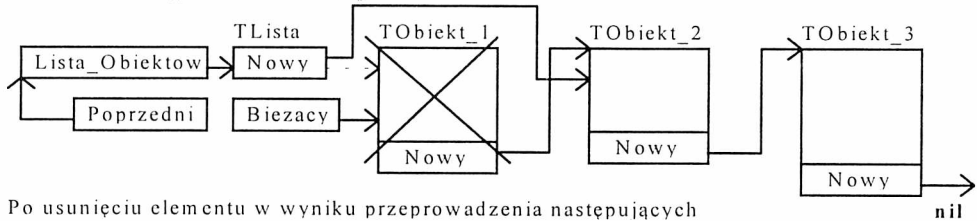


Lista_Objektow.Dodaj(Pomoc);



Rys. 25. Dodawanie elementów do listy obiektów

1. Usuwanie 1 - go elementu z listy.



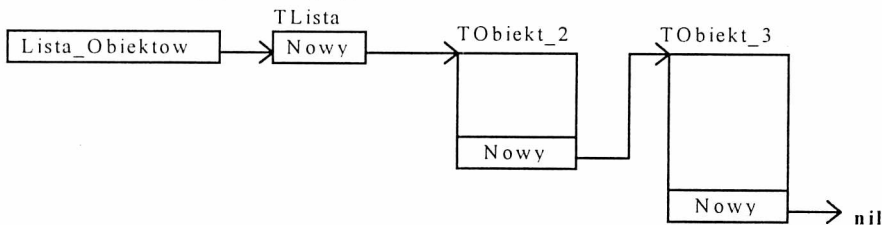
Po usunięciu elementu w wyniku przeprowadzenia następujących

```

Poprzedni := @ self;
Biezacy := Nowy;
Poprzedni^.Nowy := Biezacy^.Nowy;
Dispose(Biezacy, Usun);

```

lista obiektów wygląda następująco:



2. Usuwanie jednego z pozostałych elementów listy - np. obiektu 2 typu

Po wykonaniu następujących

```

I := 1;
Poprzedni := @ self;
Biezacy := Nowy;

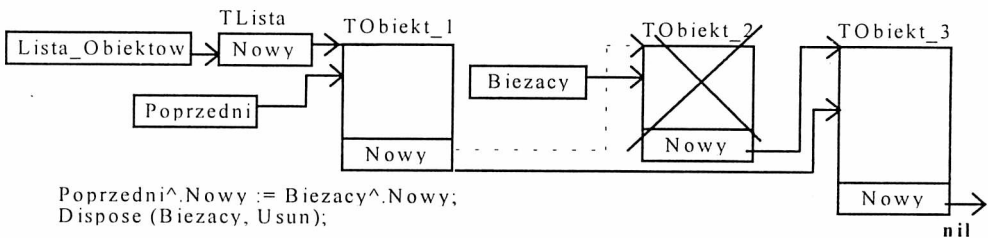
```

dopóki Biezacy <> nil i I <> 2 zmien:

```

- Poprzedni := Biezacy;
- Biezacy := Biezacy^.Nowy

```

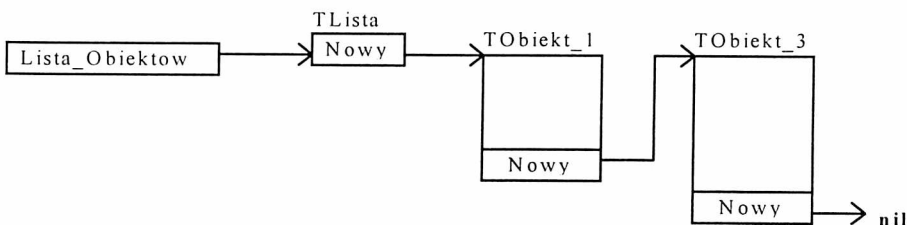


```

Poprzedni^.Nowy := Biezacy^.Nowy;
Dispose (Biezacy, Usun);

```

lista wygląda następująco :



Rys. 26. Usuwanie elementów z listy nieuporządkowanej obiektów

4.2. Przykład programu z zastosowaniem list obiektów

W module **UCW3_1** umieszczono deklaracje klas i definicje metod wykorzystujących model danych *TLista*. Moduł ten zawiera część deklaracji i definicji klas z modułu **UCW2_5**. Praktyczne wykorzystanie obiektu *TLista* można przedstawić na obiekcie typu *TRachunek*, który przechowuje pewną ilość obiektów *TZakup*. To pole możemy teraz zaprojektować jako listę, zmieniając jedynie deklarację klasy *TZakup* (rozdz. 3.3., przykład 17) następująco:

Przykład 22.

```
unit UCW3_1;
interface
uses Crt, STRUK3_1, UMENU_1;

Zakup = ^TZakup;
TZakup = object (TLista)
private
    Produkt          : PProdukt;
    Numer_Zakupu     : Integer;
    Liczba_Produktow : Integer;
public
    constructor Inicjuj (Nowy_Produkt : PProdukt);
    destructor Usun; virtual;

    function Podaj_Produkt: PProdukt;
    function Podaj_Liczbe_Produktow : Integer;
    procedure Ustaw_Liczbe_Produktow (Nowa_Liczba: Integer);
    function Podaj_Numer_Zakupu : Integer;
    procedure Ustaw_Numer_Zakupu (Nowy_Numer : Integer);

    function Oblicz_Kwote : Real;
    function Oblicz_Podatek : Real;
    function Porownaj_Zakupy (Zakup : PZakup): Boolean;
    procedure Ustaw_Zakup (Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                           N_Gatunek, N_Podatek : Integer);
    procedure Wydruk_Elementu; virtual;
end;

{Natomiast deklarację klasy TRachunek (rozdz. 3.3, przykład 17) należy zmienić
następująco:}
```



```

PRachunek = ^TRachunek;
TRachunek = object (TLista)
private   Opis_Rachunku : Lan_1;
           Ostatni_Zakup : Integer;
           Zakupy        : PLista;
           procedure Przenumeruj_Zakupy;
public    constructor Inicjuj;
           destructor Usun;virtual;

                                           {metody dostępu do pól}
function Podaj_Opis_Rachunku : Lan_1;
procedure Ustaw_Opis_Rachunku(Newy_Opis : Lan_1);
function Podaj_Ostatni_Zakup : Integer;
procedure Ustaw_Liczbe_Zakupow(N_Zakup : Integer);
function Podaj_Zakupy : PZakup;
function Podaj_Zakup(Numer_Zakupu : Integer): PZakup;
                                           {metody wykonujące zadanie klasy}
function Oblicz_Rachunek : Real;
procedure Aktualizacja_Rachunku(N_Nowy : PZakup);
procedure Dopisz_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                       N_Gatunek, N_Podatek: Integer; N_PProdukt: PProdukt);
procedure Wykresl_Zakup (Numer : Integer);
                                           {metody dodatkowe}
procedure Wydruk_Elementu;virtual;
procedure Wydruk_Rachunku;virtual;
end;
implementation
                                           {definicje z modułu UCW2_5}
                                           {definicje zmienionych metod klasy TZakup i TRachunek}
end.
                                           {koniec modułu UCW3_1}

```

W części publicznej modułu **UCW3_1** zmianie w klasie *TZakup* uległy jedynie definicja metody **constructor** oraz metody **destructor**.

```

constructor TZakup.Inicjuj(Newy_Produkt:PProdukt);
begin
  inherited Inicjuj;
  Produkt:=Newy_Produkt;
  Numer_Zakupu:=0;
  Liczba_Produktow:=0;
end;

```

```

destructor TZakup.Usun;
begin
    TLista.Usun;
    {uruchamia łańcuch wywołań destructorów elementów listy, aż Nowy
    będzie równy nil. Wtedy wykonywana jest instrukcja z linii 1 i kończy się instrukcja
    TLista.Usun elementu, zawierającego w polu Nowy wskaźnik bieżącego obiektu (self) itd.. }
    Dispose(Produkt, Usun);
end;

```

Dodatkowa metoda *Wydruk_Elementu* jest zdefiniowana identycznie jak *Wydruk_Zakupu* w module **UCW2_4** (rozdz. 3.3) i jest wykorzystana przez metodę *Wydrukuj_liste* do wydrukowania umieszczonych na liście obiektów klasy *TZakup*.

Definicje metod klasy *TRachunek* uległy zasadniczej zmianie. Metoda **constructor** *Inicjuj* oprócz inicjowania pól, tworzy zagnieżdżony obiekt dynamiczny klasy *TLista*, natomiast metoda **destructor** usuwa ten obiekt. W momencie usuwania na stercie może istnieć już lista obiektów, dziedziczących po klasie *TLista*. Wywołanie metody **destructor** *Usun* obiektu *Zakupy* powoduje usunięcie wszystkich umieszczonych na liście obiektów wraz z ich zagnieżdżonymi obiektami dynamicznymi.

```

constructor TRachunek.Inicjuj;
begin
    inherited Inicjuj;
    PamiEC(SizeOf(TLista));
    New(Zakupy, Inicjuj);
    Ostatni_Zakup :=0;
end;

destructor TRachunek.Usun;
begin
    TLista.Usun;
    Dispose(Zakupy, Usun);
end

```

Metody dostępu do pól: *Podaj_Opis_Rachunku*, *Ustaw_Opis_Rachunku*, *Podaj_Ostatni_Zakup*, *Ustaw_Liczbe_Zakupow* są identyczne jak w przykładzie 17 (rozdz. 3.3, moduł **UCW2_5**). Pozostałe metody dostępu pól zawierają operacje na liście *Zakupy*:

```
function TRachunek.Podaj_Zakupy : PZakup;  
begin  
  Podaj_Zakupy := Pointer(Zakupy);  
end;  
  
function TRachunek.Podaj_Zakup(Numer_Zakupu:Integer):PZakup;  
begin  
  Podaj_Zakup:=Pointer(Zakupy^.Podaj_Element(Numer_Zakupu));  
end;
```

Metoda *Przenumeruj_Zakupy* ma charakter pomocniczy dla metody głównej *Aktualizacja_Zakupow*, dlatego jest metodą prywatną. Konwersja typu wskaźnika *Zakupy^.Nowy* do typu *Pointer* jest spowodowana tym, że typ bazowy wskaźnika po prawej stronie instrukcji przypisania (*TLista*) jest przodkiem typu bazowego wskaźnika po lewej stronie (*TZakup*).

```
procedure TRachunek.Przenumeruj_Zakupy;  
var I : Integer;  
    Pom : PZakup;  
begin  
  I := 1;  
  Pom := Pointer(Zakupy^.Nowy);  
while Pom<> nil do  
  begin  
    Pom^.Ustaw_Numer_Zakupu(I);  
    Inc(I);  
    Pom:=Pointer(Pom^.Nowy);  
  end;  
end;
```

Definicje metod problemu również uległy zmianie, gdyż muszą pobierać z listy *Zakupy* wskaźniki do poszczególnych obiektów klasy *TZakup*:

```
function TRachunek.Oblicz_Rachunek : Real;  
var I : Integer;  
    Suma : Real;  
    Pom : PZakup;  
begin  
  Pom := Pointer(Zakupy^.Nowy);  
  Suma := 0;
```

```

while Pom <> nil do
begin
    Suma := Suma + Pom^.Oblicz_Kwote+ Pom^.Oblicz_Podatek;
    Pom := Pointer(Pom^.Nowy);
end;
Oblicz_Rachunek:=Suma;
end;

procedure TRachunek.Aktualizacja_Rachunku(N_Nowy:PZakup);
var Pom : PZakup;
begin
    Pom := Pointer(Zakupy^.Nowy);
    while Pom <> nil do
        if Pom^.Porownaj_Zakupy(N_Nowy) then
            begin
                Pom^.Ustaw_Liczbe_Produktow (Pom^.Podaj_Liczbe_Produktow
                                                + N_Nowy^.Podaj_Liczbe_Produktow);
                Dispose(N_Nowy, Usun);
                Exit;
            end
        else
            Pom := Pointer(Pom^.Nowy);
            Inc(Ostatni_Zakup);
            N_Nowy^.Ustaw_Numer_Zakupu(Ostatni_Zakup);
            Zakupy^.Dodaj (N_Nowy);
        end;
    end;

procedure TRachunek.Dopisz_Zakup(Indeks_Nazwy, Indeks_Ceny,
Indeks_Ilosci, N_Gatunek, N_Podatek: Integer; N_PProdukt: PProdukt);
var Pom : PZakup;
begin
    Pamiec(SizeOf(TZakup));
    Pom := New(PZakup, Inicjuj(N_PProdukt));
    Pom^.Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                      N_Gatunek, N_Podatek);
    Aktualizacja_Rachunku(Pom);
end;

```

Metoda *Wykresl_Zakup* zleca liście *Zakupy* usunięcie zakupu o podanym numerze (linia 1) i sprawdza wynik operacji. Jeśli wynik jest równy True, wtedy ustawia pola *Ostatni_Zakup* oraz porządkuje numerację zakupów umieszczonych w liście *Zakupy*.

Ta operacja wyraźnie świadczy o podziale zadań między klasami, gdyż jest to operacja na strukturze danych i klasa *TRachunek* jedynie korzysta z takiej możliwości.

```
procedure TRachunek.Wykresl_Zakup (Numer:Integer);  
begin  
  if Zakupy^.Usun_Element(Numer) then                                {1}  
    begin  
      Dec(Ostatni_Zakup);  
      Przenumeruj_Zakupy;  
    end;  
end;
```

Metody *Wydruk_Elementu* oraz *Wydruk_Rachunku* są metodami dodatkowymi. Metoda wirtualna *Wydruk_Elementu* jest wywoływana przez metodę listy *Wydrukuj_Liste* (gdę elementami listy są obiekty klasy *TRachunek*):

```
procedure TRachunek.Wydruk_Elementu;  
begin  
  Wydruk_Rachunku;  
end;
```

Natomiast metoda *Wydruk_Rachunku* każe się wydrukować liście *Zakupy* (linia 1).

```
procedure TRachunek.Wydruk_Rachunku;  
begin  
  ClrScr;  
  Writeln (#13#10,Opis_Rachunku);  
  Zakupy^.Wydrukuj_Liste;                                             {1}  
  Writeln(#13#10,' Suma całkowita: ',Oblicz_Rachunek:10:2);  
  Readln;  
end;
```

W przykładzie 22 przedstawiono zastosowanie obiektów klasy *TRachunek* w zaprojektowanym prostym interfejsie kontaktu z użytkownikiem. Program **CW3_1** zawiera deklarację klasy *TOkno*, która stanowi „pojemnik”, w którym zadeklarowano wskaźniki obiektów: klasy *TRachunek* oraz klasy *TOkno_Rachunku*. Umożliwia to wykonanie programu poprzez wywołanie metod tego obiektu, który pełni rolę

głównego właściciela wszystkich obiektów. Tekst programu **CW3_1**, nie różni się niczym poza wyszczególnieniem modułów od programu **CW2_52** z przykładu 20 (rozdz. 3.4). Dodatkowo moduł **UMENU2_1** niczym nie różni się od modułu **UMENU1_2** oprócz wyszczególnienia modułów. Dalej podano zmienione części tekstu programu i modułu.

```
unit UMENU2_1;
interface
uses Crt, UMENU_1, UCW3_1;

                                     {tekst modułu UMENU1_2}
end.                                {koniec modułu UMENU2_1}

program CW3_1;
uses Crt, UCW3_1, UMENU2_1, UMENU_1;

                                     {tekst programu CW2_52}
end.                                {koniec programu CW3_1}
```

4.3. Przechowywanie wskaźników obiektów w tablicy - kolekcje

Bardzo często stosowaną strukturą do przechowania pewnej liczby obiektów są tablice wskaźników obiektów. Do zarządzania taką tablicą służy klasa typu *TKolekcja*, która posiada podstawowe narzędzia do :

- tworzenia tablicy mieszczącej zadeklarowaną liczbę wskaźników;
- zapamiętania wskaźnika w tablicy, pobrania wskaźnika o podanym numerze, usunięcia wyznaczonego wskaźnika itp.

Kolekcja nie rozróżnia typu wskaźników, gdyż przechowuje je w tablicy zadeklarowanej jako: **array [1..m] of Pointer**. W *Turbo Pascalu* jest to wygodny sposób przy przekazywaniu parametrów przez wartość oraz w instrukcjach przypisania, gdyż:

- w instrukcji przypisania typ *Pointer* może wystąpić z lewej lub z prawej strony i nie wystąpi błąd, jeżeli drugim typem będzie wskaźnik dowolnej klasy obiektowej;
- przy przekazywaniu parametrów przez wartość również nie wystąpi błąd, gdyż obowiązuje zgodność w sensie przypisania dla wskaźnika dowolnego typu i parametru typu *Pointer*.

Podstawowe zagadnienia

Poniżej przedstawiono deklarację klasy *TKolekcja* oraz definicje metod, które zawiera moduł **STRUK3_2**.

Przykład 23.

```

unit STRUK3_2;
interface
uses Crt, UMENU_1;
const
Max_Kolekcji = 65519 div SizeOf(Pointer);
type
PTab_Kolekcji = ^Tab_Kolekcji;
Tab_Kolekcji = array[1..Max_Kolekcji] of Pointer;

PKolekcja = ^TKolekcja;
TKolekcja = object
  private Tablica : PTab_Kolekcji;
           Ilosc   : Integer;
           Ostatni : Integer;
  public   constructor Inicjuj(N_Ilosc : Integer);
           destructor Usun;

                                     {metody podstawowe obsługujące strukturę}
  function Podaj_Liczbe_Elementow : Integer;
  procedure Ustaw_Liczbe_Elementow (Nowa_Liczba : Integer);
  function Dodaj(Nastepny : Pointer) : Boolean;
  function Usun_Element(Numer : Integer) : Boolean;
  function Zakres (Wartosc : Integer) : Boolean;
  procedure Usun_Kolekcje;
  function Podaj_Element(Numer_Elementu : Integer) : Pointer;
end;
implemetation

constructor TKolekcja.Inicjuj(N_Ilosc : Integer);
{Metoda constructor tworzy na sterce dynamiczną tablicę (linia 2) do przechowania N_Ilosc
wskaźników, jeżeli jest dostępna pamięć (linia 1). Tę maksymalną liczbę wskaźników
przechowuje w polu Ilosc.}
begin
  if MaxAvail< N_Ilosc * SizeOf(Pointer) then Halt;           {1}
  Ilosc := N_Ilosc;                                           {3}
  if Zakres(N_Ilosc) then Halt;

```

```
    GetMem(Tablica, N_Ilosc*SizeOf(Pointer));                                {2}
    Ostatni := 0;
end;

destructor TKolekcja.Usun;
begin
    if Tablica <> nil then Usun_Kolekcje;
end;

function TKolekcja.Zakres (Wartosc : Integer) : Boolean;
{Metoda służy do sprawdzania zakresu indeksów tablicy wskaźników.}
var Log : Boolean;
begin
    Log:=(Wartosc>Max_Kolekcji)or(Wartosc>Ilosc)or(Wartosc<=0);
    Zakres:=Log;
end;

function TKolekcja.Podaj_Liczbe_Elementow : Integer;
{Metoda zwraca indeks ostatniego elementu w tablicy wskaźników.}
begin
    Podaj_Liczbe_Elementow := Ostatni;
end;

procedure TKolekcja.Ustaw_Liczbe_Elementow(Nowa_Liczba:Integer);
{Metoda ustawia liczbę wskaźników przechowywanych w polu Tablica.}
begin
    Ostatni := Nowa_Liczba;
end;

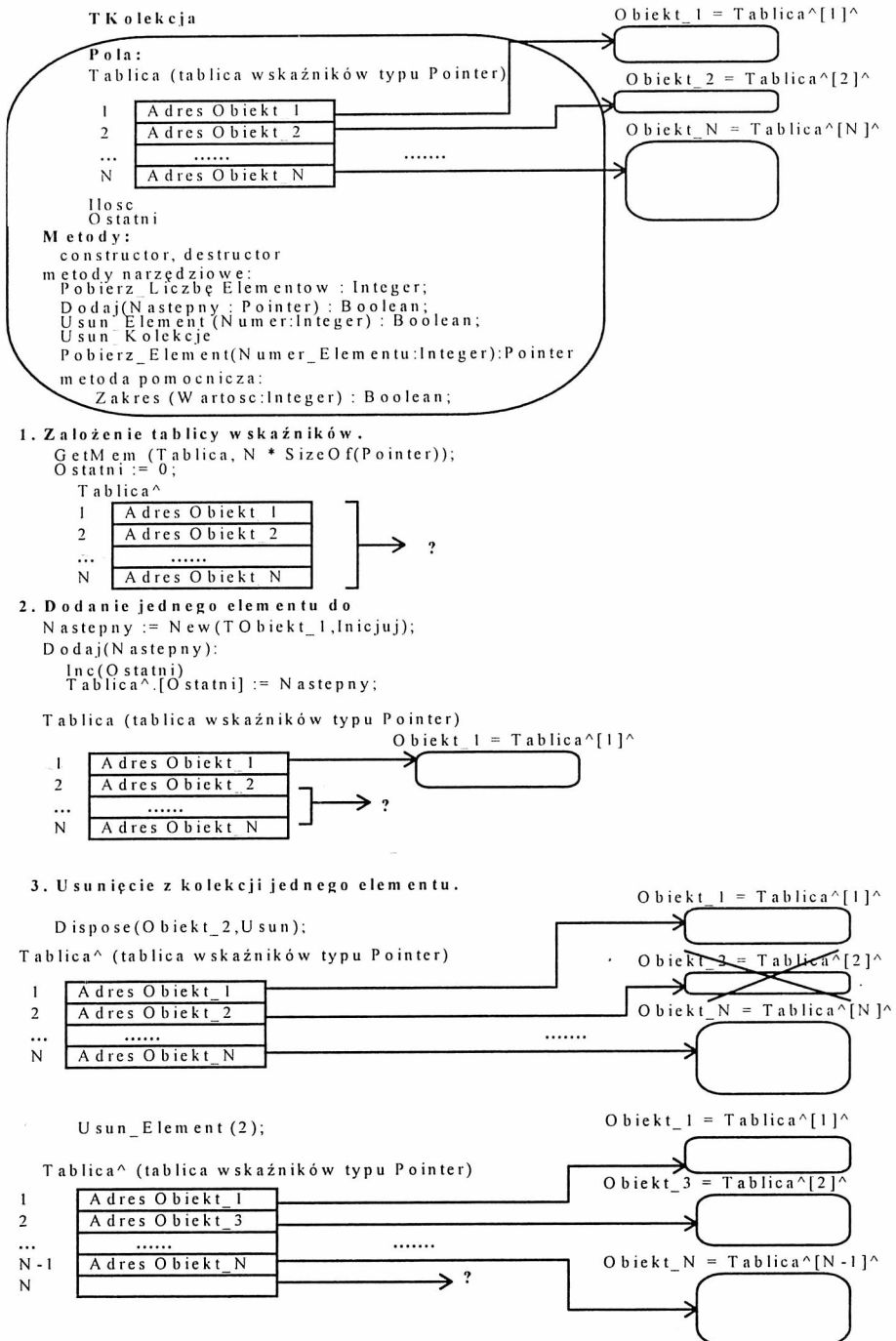
function TKolekcja.Dodaj(Nastepny:Pointer) : Boolean;
{Metoda wstawia do tablicy na koniec wskaźnik dowolnego obszaru pamięci podany
w parametrze Nastepny.}
begin
    if Ostatni<Ilosc then
        begin
            Inc(Ostatni);
            Tablica^[Ostatni]:=Nastepny;
            Dodaj:=True;
        end
    else Dodaj := False;
end;
```



```
function TKolekcja.Usun_Element(Numer : Integer) : Boolean;  
{Metoda usuwa wskaźnik o indeksie Numer z tablicy wskaźników. Obszar pamięci wskazywany  
przez element o danym indeksie musi być już zwolniony.}  
begin  
  if Zakres(Numer)or(Numer>Ostatni) then  
    Usun_Element:=False  
  else  
    begin  
      Usun_Element :=True;  
      if Numer <> Ostatni then  
        Move(Tablica^[Numer+1],Tablica^[Numer],  
              (Ostatni-Numer)*SizeOf(Pointer));  
        Dec(Ostatni);  
      end;  
    end;  
end;  
  
procedure TKolekcja.Usun_Kolekcje;  
{Metoda usuwa tablicę wskaźników z pamięci.}  
begin  
  FreeMem(Tablica,Ilosc * SizeOf(Pointer));  
  Tablica := nil;  
end;  
  
function TKolekcja.Podaj_Element(Numer_Elementu:Integer) :  
                                                                           Pointer;  
{Metoda zwraca wskaźnik elementu o indeksie Numer.}  
var I : Integer;  
begin  
  if Zakres(Numer_Elementu)or(Numer_Elementu>Ostatni) then  
    Podaj_Element := nil  
  else  
    Podaj_Element:=Tablica^[Numer_Elementu];  
end;  
end.  
                                                                           {koniec modułu STRUK3_2}
```

Na rys. 27. przedstawiono sposób wykorzystania kolekcji do przechowania obiektów.

W rozdziale 4.4 skonstruowano program **CW3_2**, w którym zastosowano klasę *TKolekcja_1*. Dzięki temu uproszczono destruktory klas posiadających kolekcję jako dynamiczny obiekt zagnieżdżony.



Rys. 27. Podstawowe działania wykonywane przez klasę TKolekcja

4.4. Przykład programu z zastosowaniem kolekcji obiektów

Praktyczne wykorzystanie obiektu *TKolekcja* można przedstawić na obiekcie typu *TRachunek*, który przechowuje pewną ilość obiektów *TZakup*. Można utworzyć moduł **UCW3_2**, uzyskany po zmianach, jakie należy wykonać w module **UCW2_5**.

Pole służące do przechowania zakupów możemy teraz zaprojektować jako kolekcję, zmieniając deklarację klasy *TRachunek* następująco:

Przykład 24.

```
unit UCW3_2;
interface
uses Crt, STRUK3_2, UMENU_1;

PRachunek = ^TRachunek;
TRachunek = object
  private
    Opis_Rachunku : Lan_1;
    Zakupy : PKolekcja;
    procedure Przenumeruj_Zakupy;
  public
    constructor Inicjuj(Liczba_Zakupow : Integer);
    destructor Usun;virtual;

    {metody dostępu do pól}

  function Podaj_Opis_Rachunku : Lan_1;
  procedure Ustaw_Opis_Rachunku(Nowy_Opis : Lan_1);
  function Podaj_Ostatni_Zakup : Integer;
  function Podaj_Zakupy : PKolekcja;
  function Podaj_Zakup(Numer_Zakupu : Integer) : PZakup;

    {metody problemu}

  function Oblicz_Rachunek : Real;
  procedure Aktualizacja_Rachunku(N_Nowyy : PZakup);
  procedure Dopisz_Zakup(Indeks_Nazwy,Indeks_Ceny,Indeks_Ilosci,
    N_Gatunek,N_Podatek :Integer; N_PProdukt : PProdukt);
  procedure Wykresl_Zakup (Numer : Integer);
  procedure Wydruk_Rachunku;virtual;

    {metoda dodatkowa}
end;
implementation

    {definicje metod z modułu UCW2_5}
    {definicje metod klasy TRachunek}
    {koniec modułu UCW3_2}
end.
```

Metody klasy wykorzystują fakt, że zastosowano kolekcję do przechowywania obiektów klasy *TZakup*, przy czym klasa ta nie wymaga dokonania żadnych zmian w deklaracji i definicji metod. Część **implementation** modułu **UCW3_2** zawiera następujące zmiany w definicjach klasy *TRachunek*:

```
constructor TRachunek.Inicjuj(Liczba_Zakupow : Integer);
{Metoda constructor tworzy na stercie kolekcję o tablicy zdolnej pomieścić Liczba_Zakupow
wskaźników.}
begin
    Opis_Rachunku := '';
    Pamiec(SizeOf(TKolekcja));
    New(Zakupy, Inicjuj(Liczba_Zakupow));
end;

destructor TRachunek.Usun;
{Metoda usuwa z pamięci (linia 1) obiekty umieszczone w kolekcji, pobierając od niej kolejne
wskaźniki (linia 2). Na końcu usuwa kolekcję (linia 3).}
var I, J:Integer; Pom:PZakup;
begin
    if Zakupy <> nil then
        begin
            J := Zakupy^.Podaj_Liczbe_Elementow;
            for I := 1 to J do
                begin
                    Pom := Zakupy^.Podaj_Element(I);           {2}
                    Dispose(Pom, Usun);                         {1}
                end;
            end;
            Dispose(Zakupy, Usun);                               {3}
        end;
    end;
```

Metody dostępu do pola *Opis_Rachunku*: *Podaj_Opis_Rachunku*, *Ustaw_Opis_Rachunku* są identyczne, jak w przykładzie 17 (rozdz. 3.3, moduł **UCW2_5**). Natomiast metody odwołujące się do kolekcji „zlecają” jej wykonanie określonych zadań.

```
function TRachunek.Podaj_Zakupy : PKolekcja;
begin
    Podaj_Zakupy := Zakupy;
end;
```

```
function TRachunek.Podaj_Zakup(Numer_Zakupu: Integer) : PZakup;  
begin  
    Podaj_Zakup := Zakupy^.Podaj_Element(Numer_Zakupu);  
end;
```

```
function TRachunek.Podaj_Ostatni_Zakup : Integer;  
begin  
    Podaj_Ostatni_Zakup := Zakupy^.Podaj_Liczbe_Elementow;  
end;
```

Metody problemu również korzystają z usług kolekcji przy wstawianiu i usuwaniu elementów, natomiast same realizują główny algorytm: przy wstawianiu elementu do kolekcji sprawdzają, jak dotąd, czy istnieje już pozycja rachunku z takim samym produktem itp.

```
procedure TRachunek.Dopisz_Zakup(Indeks_Nazwy, Indeks_Ceny,  
Indeks_Ilosci, N_Gatunek, N_Podatek: Integer; N_PProdukt: PProdukt);  
var Pom : PZakup;  
begin  
    Pamiec(SizeOf(TZakup));  
    Pom := New(PZakup, Inicjuj(N_PProdukt));  
    Pom^.Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,  
                      N_Gatunek, N_Podatek);  
    Aktualizacja_Rachunku(Pom);  
end;
```

```
procedure TRachunek.Przenumeruj_Zakupy;  
var I, J : Integer;  
    Pom : PZakup;  
begin  
    J := Zakupy^.Podaj_Liczbe_Elementow;  
    for I := 1 to J do  
        begin  
            Pom := Zakupy^.Podaj_Element(I);  
            Pom^.Ustaw_Numer_Zakupu(I);  
        end;  
    end;
```

```

procedure TRachunek.Aktualizacja_Rachunku(N_Nowyy : PZakup);
var I      : Integer;
      Pom   : PZakup;
begin
  I := Zakupy^.Podaj_Liczbe_Elementow;
  N_Nowyy^.Ustaw_Numer_Zakupu(I+1);
  while not Zakupy^.Zakres(I) do
    begin
      Pom := Zakupy^.Podaj_Element(I);
      if Pom^.Porownaj_Zakupy(N_Nowyy) then
        begin
          Pom^.Ustaw_Liczbe_Produktow(Pom^.Podaj_Liczbe_Produktow+
                                         N_Nowyy^.Podaj_Liczbe_Produktow);
          Dispose(N_Nowyy, Usun);
          Exit;
        end
      else Dec(I);
    end;
  Zakupy^.Dodaj(N_Nowyy);
end;

```

Metoda *Wykresl_Zakup* zleca kolekcji (linia 2) usunięcie wskaźnika zakupu, który wcześniej został usunięty z pamięci (linia 1).

```

procedure TRachunek.Wykresl_Zakup (Numer : Integer);
var Pom : PZakup;
begin
  Pom := Zakupy^.Podaj_Element(Numer);
  if Pom <> nil then
    begin
      Dispose(Pom, Usun);                                     {1}
      if Zakupy^.Usun_Element(Numer) then
        Przenumeruj_Zakupy;                                   {2}
      end;
    end;
end;

```

```

function TRachunek.Oblicz_Rachunek : Real;
var I, J : Integer;
      Suma : Real;
      Pom  : PZakup;

```

```
begin
  Suma := 0;
  J := Zakupy^.Podaj_Liczbe_Elementow;
  for I := 1 to J do
    begin
      Pom := Zakupy^.Podaj_Element(I);
      Suma := Suma + Pom^.Oblicz_Kwote+ Pom^.Oblicz_Podatek;
    end;
  Oblicz_Rachunek := Suma;
end;
```

Metoda *Wydruk_Rachunku* ma charakter tymczasowy. Do interakcji z użytkownikiem programu można wykonać prosty interfejs, podobnie jak przedstawiony w rozdziale 3.4.

```
procedure TRachunek.Wydruk_Rachunku;
var I, J : Integer;
    Pom : PZakup;
begin
  ClrScr; Writeln (#13#10,Opis_Rachunku);
  J := Zakupy^.Podaj_Liczbe_Elementow;
  for I := 1 to J do
    begin
      Pom := Zakupy^.Podaj_Element(I);
      Pom^.Wydruk_Zakupu;
    end;
  Writeln(#13#10, 'Suma calkowita: ',Oblicz_Rachunek:10:2);
  Pauza;
end;
```

Niewygodną cechą kolekcji przedstawionej w tym rozdziale jest iteracyjne pobieranie wskaźnika obiektu. W metodach klas posiadających obiekt zagnieżdżony typu *TKolekcja* te iteracje można zaakceptować, gdyż metody wyrażają specjalizację danej klasy. Jednak w każdej klasie wykorzystującej obiekty klasy *TKolekcja* pojawia się problem zwalniania pamięci przez elementy. Postać metody **destructor** będzie identyczna w każdej klasie, jak to pokazano wyżej. Aby uzyskać **destructor** podobny do destructora klasy *TLista* (jedynie linia 1), należy w takiej kolekcji założyć, że wszystkie obiekty mają wspólnego przodka, po którym dziedziczą „abstrakcyjny” wirtualny **destructor**, przewidujący konieczność zwalniania różnych pól dynamicznych u następców. Podobnie można uprościć metodę *Wykresl_Zakup* w

klasie *TRachunek*, zlecając kolekcji usuwanie obiektu typu *TZakup* jak i jego wskaźnika z pamięci.

Jeżeli zadeklarujemy abstrakcyjną klasę *TObiekt_1*, posiadającą wirtualny **destructor**, wówczas każdy obiekt klasy dziedziczącej po tej klasie, wstawiony do kolekcji może być zwolniony na jej poziomie. Poniżej podano deklaracje klas:

- *TObiekt_1*.
- *TKolekcja_1*

oraz definicje ich metod, które należy dołączyć do modułu **STRUK3_2**, aby go rozbudować do modułu **STRUK322**.

```
unit STRUK322;
interface
    {deklaracje z modułu STRUK3_2}

PObiekt_1 = ^TObiekt_1;
TObiekt_1 = object
    constructor Inicjuj;
    destructor Usun;virtual;
end;

PKolekcja = ^TKolekcja
TKolekcja = object
    {deklaracje klasy z modułu STRUK3_2}
    procedure Usun_Kolekcje;virtual;
end;

PKolekcja_1 = ^TKolekcja_1;
TKolekcja_1 =
    object(TKolekcja)
        procedure Usun_Jeden(I:Integer);
        procedure Usun_Kolekcje;virtual;
        function Usun_Element(Numer : Integer) : Boolean;
    end;
implementation
    {definicje brakujących metod z modułu STRUK3_2}
    {-----TObiekt1-----}
    constructor TObiekt_1.Inicjuj;
    begin
    end;

    destructor TObiekt_1.Usun;
    begin
    end;
```



```

{-----TKolekcja_1-----}
procedure TKolekcja_1.Usun_Jeden(I : Integer);
{Przy usuwaniu każdego elementu Tablica^[I] dokonana jest konwersja do typu
wskaźnikowego klasy abstrakcyjnej TObiekt_1, co umożliwia wywołanie wirtualnej metody
destructor aktualnego obiektu.}
begin
  Dispose(PObiekt_1(Tablica^[I]), Usun);
end;

function TKolekcja_1.Usun_Element(Numer : Integer) : Boolean;
{Metoda usuwa z pamięci obiekt (linia 1) oraz jego wskaźnik (linia 2).}
begin
if Zakres(Numer) or (Numer>Ostatni) then
  Usun_Element := False
else
begin
  Usun_Jeden(Numer);                                {1}
  Usun_Element := True;
if Numer <> Ostatni then
    Move(Tablica^[Numer+1], Tablica^[Numer],
          (Ostatni-Numer)*SizeOf(Pointer));          {2}
  Dec(Ostatni);
end;
end;

procedure TKolekcja_1.Usun_Kolekcje;
{Metoda jest zadeklarowana jako wirtualna w klasie TKolekcja. Destrutor jest
zadeklarowany w korzeniu drzewa TKolekcja., stąd po jego wywołaniu zostanie wykonana
metoda Usun_Kolekcje obiektu aktualnego.}
var I : Integer;
begin
for I := 1 to Ostatni do
  Usun_Jeden(I);
inherited Usun_Kolekcje;
end;
end.

```

{koniec modułu STRUK322}

Dalej pokazano fragmenty deklaracji klas, które należy zmienić oraz zmienione definicje ich metod w module UCW3_22, aby można było je zwalniać wywołując jedynie **destructor** klasy TObiekt_1.

5. Strumienie i zasoby

5.1. Podstawowe zagadnienia

W *Turbo Pascalu* typ plikowy nie może być bezpośrednio wykorzystany do przechowania obiektów. W przypadku, gdy zadeklarujemy w programie:

```
type TObiekt =  
    object  
        ....  
    end;  
var Plik : file of TObiekt;
```

kompilator zgłosi błąd (*Error 24: File components may not be files of objects*). Można wykorzystać zmienną plikową amorficzną typu **file** do zapisu zawartości pól obiektu, lecz będą to dane zupełnie nie związane z konkretnym obiektem. Takie podejście w pewnych wypadkach jest wystarczające.

Jeżeli utworzenie danych obiektu wymaga skomplikowanych, długotrwałych obliczeń itp. co opóźnia działanie programu, wtedy generuje się go jednorazowo i zapisuje w strumieniu następujące informacje:

- specjalne dane identyfikujące klasę obiektu,
- dane zawarte w polach.

Następnie można zainicjować ten obiekt na podstawie tych danych odczytanych ze strumienia.

Turbo Pascal dostarcza w pakiecie modułów *Turbo Vision* typy obiektowe dziedziczące po *TStream*, które umożliwiają sekwencyjny zapis i odczyt obiektów w strumieniach. Strumienie mogą być związane z:

- plikiem dyskowym,
- pamięcią EMS,
- stertą.

W *Turbo Vision* dostępny jest również obiekt *TResourceFile* zwany plikiem zasobów. Umożliwia on niesekwencyjny zapis lub odczyt obiektów w pliku dyskowym za pomocą niepowtarzalnego klucza, składającego się z łańcucha znaków (do 255). Przetwarzane obiekty muszą dziedziczyć po klasie *TObject*.

W tym rozdziale przedstawiono przygotowane uproszczone narzędzia do sekwencyjnego i niesekwencyjnego zapisu i odczytu danych obiektowych w plikach dyskowych, korzystając częściowo z pomysłu zaprezentowanego w [2]. Głównym celem przedstawienia tego problemu jest zaawansowane korzystanie z polimorfizmu z wykorzystaniem abstrakcji klas.

Istotne jest przy analizie tych rozwiązań, aby użytkownik potrafił w każdej klasie zdefiniować dodatkowe metody służące do:

- zapisu obiektu, przy założeniu, że obiekt potrafi ‘sam się zapisać’- swoje dane nieobiekto-owe oraz obiekty zagnieżdżone,
- odczytu obiektu na podstawie informacji zapisanej w strumieniu.

Na rys. 28 i 32 przedstawiono poszczególne kroki, jakie trzeba wykonać, aby procesy te przebiegały pomyślnie.

5.2. Sekwencyjny zapis i odczyt obiektów - strumienie



Rys. 28. Etapy przetwarzania przy korzystaniu ze strumienia

Do zapisu obiektów w strumieniu przeznaczono obiekt klasy *TStrumien*, który potrafi wykonać następujące zadania:

- ♦ zapisać obiekty w strumieniu,
- ♦ odczytać obiekty ze strumienia, tzn.:
 - zainicjować obiekt właściwej klasy na sterce,
 - zapisać dane do pól tego obiektu.

Aby obiekt mógł przewidzieć, jakie obiekty ma przetwarzać, musi „abstrakcyjnie” znać ich sposób zapisu i odtwarzania. Z tej przyczyny wszystkie obiekty zapisywane i odczytywane ze strumienia powinny dziedziczyć po takiej *abstrakcyjnej klasie*, która za pomocą metod wirtualnych wykonuje polecenia strumienia. Klasę taką nazwiemy *TObiekt*. Obie klasy zadeklarowano w module **PLIKI_I**. Część publiczna modułu jest następująca:

```
unit PLIKI_1;
interface
type   Buf_Plik = array[1..100] of Word;
       PBuf_Plik = ^Buf_Plik;

PObiekt = ^TObiekt;
TObiekt = object
  constructor Inicjuj;
  function Odtworz_Objekt(Tab:PBuf_Plik):PObiekt;virtual;
  procedure Zapisz_Na_Dysku (var Plik:file);virtual;
  function Podaj_Numer_Typu:Integer;virtual;
  function Zachowaj_Objekt(Tab:PBuf_Plik):Integer;virtual;
  procedure Czytaj_Dalej(var Plik:file) ;virtual;
end;

PStrumien = ^TStrumien;
TStrumien = object
  private Plik : file;
  public  constructor Inicjuj (Nazwa:string; Sposob:Boolean);
         destructor Usun;
         function Podaj_Plik:Pointer;
         procedure Zapisz (Objekt:PObiekt);
         procedure Odczytaj(var Objekt:PObiekt);
         procedure Koniec;
end;
procedure Rejestracja_W_Strumieniu(Vmtp:Pointer;Numer:Integer) ;
procedure Odczyt(var Objekt:PObiekt;var Plik:file);
procedure Usun_Rejestr;
```

Abstrakcyjna klasa *TObiekt* zawiera wirtualne metody niezbędne przy zapisie i odczycie, jednak nie nadająca się do bezpośredniego wykorzystania. Obiekty dziedziczące muszą część tych metod pokryć, dostosowując je do własnych potrzeb, wykorzystując polimorfizm. W rozdziale 5.2 przedstawiono definicje metod klasy *TObiekt*.

Podstawą zapisu i odczytu obiektu jest rozpoznanie, do jakiej klasy ten obiekt należy. W związku z tym każda klasa powinna mieć *niepowtarzalny numer*, umieszczany wraz z danymi w strumieniu, a jednocześnie znany w programie. Numer ten jest łącznikiem między informacją zapisaną i odczytaną. Umożliwia on rozpoznanie klasy obiektu przy odczycie i poprzez adres jej VMT pobranie adresu właściwej wirtualnej metody, odtwarzającej obiekt. Zakłada się, że w czasie działania programu adresy metod nie ulegną zmianie.

W kolejnych punktach przetwarzania strumieni zostaną omówione zadania spełniane przez te metody. Definicje metod zawiera część implementacyjna modułu **PLIKI_1**.

Aby wykorzystać gotowe narzędzia do zapisu i odczytu obiektów w strumieniu, należy *zdefiniować w każdej klasie właściwe metody*. Strumień wywołuje te metody i zleca im wykonanie szczegółowych zadań. W dalszej części rozdziału zostanie opisany sposób definiowania takich metod na tle opisu kolejnych etapów przetwarzania strumieni.

Przed szczegółowym przedstawieniem poszczególnych etapów przetwarzania jest pokazany program **CW4_1** wykorzystujący strumienie. Program korzysta z modułu **UCW4_1**, zawierającego deklaracje drzewa *TProdukt..* z modułu **UCW2_3** (rozdz. 3.2, przykład 13) uzupełnionych o metody służące do zapisu i odczytu obiektów w strumieniu.

Przykład 25.

```

unit UCW4_1;
interface
{deklaracje z modułu UCW2_3}

PProdukt = ^TProdukt;
TProdukt =
object (TObiekt)
  private   Nazwa : Lan_1;
            Cena  : Real;
            { deklaracja klasy z modułu UCW2_3 }
            { metody do zapisu i odczytu w strumieniu }
  function Odtworz_Obiekt (Tab: PBuf_Plik) : PObiekt; virtual;
  function Zachowaj_Obiekt (Tab: PBuf_Plik) : Integer; virtual;
end;

```

```

PProdukt_1 = ^TProdukt_1;
TProdukt_1 =
object (TProdukt)
  private Gatunek : Integer;
                                     { deklaracje klasy z modułu UCW2_3 }
                                     { metody do zapisu i odczytu w strumieniu }
  function Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;virtual;
  function Zachowaj_Objekt (Tab:PBuf_Plik):Integer;virtual;
end;

PProdukt_2 = ^TProdukt_2;
TProdukt_2 =
object (TProdukt_1)
  private Podatek : Integer;
                                     { deklaracje klasy z modułu UCW2_3 }
                                     { metody do zapisu i odczytu w strumieniu }
  function Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;virtual;
  function Zachowaj_Objekt (Tab:PBuf_Plik):Integer;virtual;
end;
implementation
  {metody zdefiniowane w module UCW2_3 oraz definicje metod omówionych w rozdz. 5.2}.
end.
                                     {koniec modułu UCW4_1}

```

Program **CW4_1** zapisuje obiekty do strumienia realizując cztery etapy przetwarzania pokazane na rys. 28.

```

program CW4_1;
uses Crt, PLIKI_1, UCW4_1, UMENU_1
var Produkt_1 : PProdukt;
    Produkt_2 : PProdukt_1;
    Produkt_3 : PProdukt_2;
    Strumien : PStrumien;
begin
  ClrScr;
                                     {etap 1 - rejestracja klas}
  Rejestracja_W_Strumieniu (TypeOf (TProdukt), 1);
  Rejestracja_W_Strumieniu (TypeOf (TProdukt_1), 2);
  Rejestracja_W_Strumieniu (TypeOf (TProdukt_2), 3);

```

```

                                                                    {przygotowanie danych}
Pamiec(SizeOf(TProdukt));      New(Produkt_1, Inicjuj);
Pamiec(SizeOf(TProdukt_1));    New(Produkt_2, Inicjuj);
Pamiec(SizeOf(TProdukt_2));    New(Produkt_3, Inicjuj);
Produkt_1^.Wydruk_Atrybutow_Produktow(1,1,1,0);
Produkt_2^.Wydruk_Atrybutow_Produktow(2,2,2,0);
Produkt_3^.Wydruk_Atrybutow_Produktow(3,3,3,0);

                                                                    {etap 2 - utworzenie strumienia}
Pamiec(SizeOf(TStrumien));
New(Strumien, Inicjuj('Plik_Ob', False));

                                                                    {etap 3 - zapis do strumienia}
Strumien^.Zapisz(Produkt_1);
Strumien^.Zapisz(Produkt_2);
Strumien^.Zapisz(Produkt_3);

                                                                    {etap 4 - zamknięcie strumienia i usunięcie z pamięci}
Dispose(Strumien, Usun);
Dispose(Produkt_1, Usun);
Dispose(Produkt_2, Usun);
Dispose(Produkt_3, Usun);
Usun_Rejestr;
end.                                                                    {koniec programu CW4_1}

```

Program **CW4_2** odczytuje obiekty do strumienia realizując cztery etapy przetwarzania pokazane na rys. 28

```

program CW4_2;
uses Crt, PLIKI_1, UCW4_1, UMENU_1;
var Produkt_1 : PProdukt;
    Produkt_2 : PProdukt_1;
    Produkt_3 : PProdukt_2;
    Strumien : PStrumien;
begin
    ClrScr;

                                                                    {etap 1 - rejestracja klas}
    Rejestracja_W_Strumieniu(TypeOf(TProdukt), 1);
    Rejestracja_W_Strumieniu(TypeOf(TProdukt_1), 2);
    Rejestracja_W_Strumieniu(TypeOf(TProdukt_2), 3);

                                                                    {etap 2 - utworzenie strumienia}
    Pamiec(SizeOf(TStrumien));
    New(Strumien, Inicjuj('Plik_Ob', True));

```



```

                                                    {etap 3 - odczyt ze strumienia}
Strumien^.Odczytaj (PObiekt (Produkt_1));
Strumien^.Odczytaj (PObiekt (Produkt_2));
Strumien^.Odczytaj (PObiekt (Produkt_3));
                                                    {etap 4 - zamknięcie strumienia}
Dispose(Strumien, Usun);

                                                    {przetwarzanie odczytanych danych}
Produkt_1^.Wydruk_Produktu; Dispose(Produkt_1, Usun);
Produkt_2^.Wydruk_Produktu; Dispose(Produkt_2, Usun);
Produkt_3^.Wydruk_Produktu; Dispose(Produkt_3, Usun);
Usun_Rejestr;
end.                                                    {koniec programu CW4_2}

```

Rejestracja klas obiektów

Rejestracja klasy zapisywanego lub odczytywanego obiektu polega na zapisaniu w elementach specjalnej tablicy, znajdującej się na sterce przez cały czas trwania przetwarzania strumieni:

- ♦ numeru przydzielonego klasie (niepowtarzalnego),
- ♦ adresu offsetowego tablicy VMT,
- ♦ adresów następujących metod wirtualnych:
 - *Odtworz_Objekt* (służącej do utworzenia obiektu na sterce oraz przypisania odczytanych z dysku polom tego obiektu)
 - *Zapisz_Na_Dysku* (służącej do zapisania na dysku danych obiektu).

Definicje niezbędne do zadeklarowania tablicy rejestracji danych *Rejestr*^, znajdującej się w części implementacyjnej modułu **PLIKI_1** zostały przedstawione poniżej:

```

implementation                                                    {moduł PLIKI_1}
uses UMENU_1;
type Funkcja_1=function(Buf:PBuf_Plik;var self):PObiekt;          {1}
    Obiekt_Rek = record
        Numer      : Integer;
        VMT        : Word;
        Met_Wczyt  : Funkcja_1;
    end;
    Tab_Rek = array[1..10] of Obiekt_Rek;
    PTab_Rek = ^Tab_Rek;

```

```

var Rejestr : PTab_Rek;
      Licznik_Rejestracji : Integer;
{ definicje procedur: Rejestracja_W_Strumieniu, Odczyt, Usun_Rejestr oraz definicje metod
klasy TObiekt i TStrumien}
end.

```

{koniec modułu **PLIKI_1**}

W linii (1) zdefiniowano typ proceduralny reprezentujący nagłówek metody *Odtworz_Obiekt*, stąd dodatkowy parametr **self** przekazywany przez zmienną. Typ wskaźnikowy *PTab_Rek*, zdefiniowany w części interfejsowej modułu, jest wykorzystywany na dynamiczny bufor danych przetwarzanych obiektów.

```

Tab_Rek   = array[1..10] of Obiekt_Rek;
PTab_Rek  = ^Tab_Rek;

```

Zakłada się, że w czasie działania programu adresy metod i położenie tablicy metod wirtualnych VMT danej klasy nie zmieniają się. Te parametry są zapisane w tablicy w pamięci i przechowywane przez cały czas działania programu. Jednocześnie taki sam numer klasy jest zapisany na początku danych obiektu w pliku (patrz Zapis).

Na rys. 29 przedstawiono czynności wykonywane przy rejestracji.

1. Pobranie adresu offsetowego klasy za pomocą funkcji *TypeOf* i przygotowanie danych rejestracji obiektu : adres offsetowy w polu *Vmt*, numer klasy w polu *Numer*, oraz adres metody wirtualnej *Odtworz_Obiekt* w polu *Met_Wczyt*.

2. Zapis do tablicy rejestracji *Rejestr*[^] danych przygotowanych w pkt.1:

Po sprawdzeniu, czy w tablicy nie przydzielono już innej klasie wybranego numeru, wykonuje się poniższe czynności:

Inc(Licznik_Rejestracji);

Dostosowanie rozmiaru tablicy do liczby rejestrowanych klas;

Zapis danych do tablicy rejestracji *Rejestr*[^]

Element tablicy *Rejestr*[^] z danymi dotyczącymi rejestrowanego obiektu należącego np. do klasy *TObiekt_1*, dziedziczącej po klasie *TObiekt*.

Ofs(TypeOf(TObiekt_1)^)	adres offsetowy tablicy VMT
Numer_TObiekt_1	niepowtarzalny numer klasy
@ TObiekt_1.Odtworz_Obiekt	adres metody inicjującej obiekt danej klasy

Rys. 29. Algorytm rejestracji klasy dziedziczącej po *TObiekt*

Dalej przedstawiono procedurę rejestrującą *Rejestracja_W_Strumieniu* i podprogramy pomocnicze: *Dodaj_Objekt_W_Tablicy*, *Szukaj_Objekt_W_Tablicy*.

```

procedure Dodaj_Objekt_Do_Tablicy (N_Rek:Obiekt_Rek);
{Procedura powiększa na stercie tablicę rejestracji Rejestr (2) o nowy rekord za pomocą
pomocniczej tablicy PPom (1)}
  var PPom      : PTab_Rek;
      Rozmiar   : Integer;
begin
  Rozmiar:=(Licznik_Rejestracji +1)*SizeOf(Obiekt_Rek);
  Pamięc(Rozmiar);
  GetMem(PPom,Rozmiar);
  if Rejestr<>nil then
    begin
      Move(Rejestr^,PPom^,Rozmiar-SizeOf(Obiekt_Rek));
      FreeMem(Rejestr,Rozmiar- SizeOf(Obiekt_Rek));
    end;
  Inc(Licznik_Rejestracji;
  Rejestr := PPom;
  Rejestr^[Licznik_Rejestracji]:=N_Rek;
end;

```



```

procedure Rejestracja_W_Strumieniu(Vmtp:Pointer;Numer:Integer);
{Procedura kompletuje dane rejestracji obiektu (1) i zapisuje do tablicy rejestracji (2), jeśli
Numer nie został już wykorzystany do rejestracji innej klasy.}
  var Rek : Obiekt_Rek;
      Pom : Pointer;
begin
  Pom := Vmtp;
  Rek.Numer := Numer;
  Rek.Vmt := Ofs(Pom^);
  Inc(Word(Pom),8);
  Rek.Met_Wczyt := Funkcja_1(Pom^);
  if Szukaj_Objekt_W_Tablicy(Numer) = 0 then
    Dodaj_Objekt_Do_Tablicy(Rek);
end;

```

```

function Szukaj_Objekt_W_Tablicy(N_Numer:Integer):Integer;
{Funkcja zwraca indeks elementu z tablicy rejestracji, zawierającego dany numer klasy}
var I : Integer;
begin
    I:= Licznik_Rejestracji;
    while (I>0) and (Rejestr^[I].Numer<>N_Numer) do Dec(I);
    Szukaj_Objekt_W_Tablicy := I;
end;

```

Zainicjowanie strumienia

W programie należy zadeklarować wskaźnik typu *PStrumien* i zainicjować go na stercie:

```

Pamiec(SizeOf(TStrumien));
New(Strumien, Inicjuj('Nazwa_1', Sposob);

```

Parametr *Sposob* - jest równy *True*, jeżeli plik jest otwarty do odczytu oraz *False*, gdy jest zakładany. Definicja **constructora** *Inicjuj* jest przedstawiona niżej:

```

constructor TStrumien.Inicjuj(Nazwa : string;Sposob : Boolean);
begin
    Assign(Plik, Nazwa)
    if Sposob then
{$I-} begin
        Reset(Plik, 1)
{$I+}
        if IOResult <> 0 then Halt;
    end
    else Rewrite(Plik, 1);
end;

```

Przetwarzanie obiektów w strumieniu

Zapis

Zasady dotyczące przygotowania obiektu do zapisu:

1. Przodkiem wszystkich obiektów zapisywanych musi być klasa *TObiekt*.

2. Klasy zawsze muszą mieć zdefiniowaną metodę wirtualną (typu funkcja) *Zachowaj_Objekt*, która zapisuje do bufora przekazanego do metody dane pól nieobiektywych tworząc blok o rozmiarze równym sumie rozmiarów pól oraz zwraca tę wartość.

W przypadku pól wskaźnikowych zapisywana jest wartość zmiennej wskazywanej przez to pole, a do rozmiaru danych wliczana jest jej wielkość. Klasy bez obiektów zagnieżdżonych korzystają z metody *Zapisz_Na_Dysku*, dziedziczonej od klasy *TObiekt*, w której wywoływana jest metoda *Zachowaj_Objekt*.

3. Klasy posiadające pola obiektywne muszą mieć własną metodę *Zapisz_Na_Dysku*, w której najpierw zapisywane są dane nieobiektywne klasy główną metodą *TObiekt.Zapisz_Na_Dysku* (za pomocą metody *Zachowaj_Objekt*), a następnie 'zapisują się' są pola obiektywne.

Na rys. 30. przedstawiono czynności wykonywane przy zapisie obiektu.

Wywołanie przez *Strumien* metody *Zapisz_Na_Dysku* danego obiektu:

1. Wywołanie przez obiekt metody *Zachowaj_Objekt*, zwracającą liczbę bajtów, jakie zajmują pola obiektu:

$$\text{Rozmiar_W_Pliku} := \text{SizeOf}(\text{Pole_1}) + \text{SizeOf}(\text{Pole_2}) +$$
2. Utworzenie bufora o rozmiarze równym zwróconej przez metodę liczbie bajtów równą *Rozmiar*, powiększonej o 4:

$$\text{Inc}(\text{Rozmiar_W_Pliku}, 4);$$

$$\text{GetMem}(\text{Bufor}, \text{Rozmiar_W_Pliku})$$
3. Zapis do bufora następujących danych charakteryzujących zapisywany obiekt:

Bufor[^]

Rozmiar_W_Pliku -	(2 bajty)
Numer klasy -	(2 bajty)
Dane zapisane w polach obiektu ((Rozmiar_W_Pliku - 4) bajty)	

Uwaga: Numer klasy pobrany jest z tablicy rejestracji metodą *Pobierz_Numer_Typu*.

4. Zapis zawartości Bufora do pliku i zwolnienie pamięci zajmowanej przez Bufor[^].

Rys. 30. Algorytm zapisu obiektu do strumienia

Poniżej przedstawiono wspomniane wyżej metody dla obiektów klas *TProdukt...*. Ponieważ nie mają obiektów zagnieżdżonych, bezpośrednio dziedziczą metodę *Zapisz_Na_Dysku* oraz *Podaj_Numer_Typu* od klasy *TObiekt*, natomiast mają własne metody *Zachowaj_Obiekt*. Bezpośredni zapis do pliku wykonuje metoda strumienia *Zapisz*, która inicjuje cały proces.

```
procedure TStrumien.Zapisz (Obiekt:PObiekt);
{W metodzie wywołana jest wirtualna metoda Zapisz_Na_Dysku zapisywanego obiektu Obiekt,
dziedziczona od TObiekt lub zdefiniowana w klasie zapisywanego obiektu.}
begin
  Obiekt^.Zapisz_Na_Dysku (Plik);
end;
```

```
procedure TObiekt.Zapisz_Na_Dysku (var Plik : file);
var Pom                : array[1..2]of Integer;
    Rozmiar_W_Pliku,I,Ile : Word;
    Bufor                : PBuf_Plik;
begin
  Rozmiar_W_Pliku := Zachowaj_Obiekt(nil) + 4;           {krok 1 z rys. 30}
  Pamiec (Rozmiar_W_Pliku);
  GetMem(Bufor,Rozmiar_W_Pliku);                         {krok 2 z rys. 30}
  I:=1;
  Bufor^[I] := Rozmiar_W_Pliku;                          {krok 3 z rys. 30}
  Inc(I);
  Bufor^[I] := Podaj_Numer_Typu;
  Inc(I);
{tutaj wywoływana jest zawsze metoda wirtualna Zachowaj_Obiekt obiektu wywołującego
metodę Zapisz_Na_Dysku}
  Zachowaj_Obiekt(@(Bufor^[I]));
  BlockWrite(Plik, Bufor^,Rozmiar_W_Pliku,Ile);          {krok 4 z rys. 30}
  if Ile < > Rozmiar_W_Pliku then
    begin
      Rewrite(Plik);
      Halt;
    end;
  FreeMem(Bufor, Rozmiar_W_Pliku);
end;
```

```

function TObiekt.Podaj_Numer_Typu : Integer;
{Metoda zwraca numer rejestracji klasy (linia 2) obiektu wywołującego tę metodę, jeżeli klasa
jest już zarejestrowana (linia 1)- krok 3 z rys. 30}
var J : Integer;
begin
  J := Licznik_Rejestracji;
  while (J>0) and (Rejestr^[J].Vmt<>Ofs (TypeOf (self) ^) ) do
    Dec (J);
  if J=0 then
    Halt;
  Podaj_Numer_Typu:=Rejestr^[J].Numer;
end;

```

{1}

{2}

```

function TObiekt.Zachowaj_Objekt (Tab:PBuf_Plik):Integer;
{Metoda zwraca wartość równą 0 dla klasy abstrakcyjnej.}
begin
  Zachowaj_Objekt:=0;
end;

```

Metody zapisu obiektów w strumieniu, pokrywające metodę *Zachowaj_Objekt* abstrakcyjnej klasy *TObiekt*, są następujące (moduł *UCW4_1*):

```

function TProdukt.Zachowaj_Objekt (Tab:PBuf_Plik):Integer;
{Funkcja oblicza rozmiar pól (linia 1) i zwraca go (linia 3) oraz zapisuje (linia 2) ich
zawartość do bufora Tab^ - krok 1 i 3 z rys. 30.}
var I : Integer;
begin
  I := SizeOf (Nazwa) + SizeOf (Cena);
  if Tab<> nil then
    Move (Nazwa, Tab^, I);
    Zachowaj_Objekt := I;
end;

```

{1}

{pola tworzą spójny blok-2}

{3}

```

function TProdukt_1.Zachowaj_Objekt (Tab:PBuf_Plik):Integer;
var I : Integer;
begin
  I := 29;
  if Tab<> nil then
    Move (Nazwa, Tab^, I);
    Zachowaj_Objekt := I;
end;

```

```

function TProdukt_2.Zachowaj_Objekt (Tab:PBuf_Plik):Integer;
var I : Integer;
begin
  I := 31;
  if Tab<> nil then
    Move (Nazwa, Tab^,I);
    Zachowaj_Objekt :=I
  end;

```

Dla obiektów mających pola typu obiektowego (obiekty zagnieżdżone) należy pokryć metodę *TObiekt.Zapisz_Na_Dysku* np. dla klasy *TZakup* mamy:

```

procedure TZakup.Zapisz_Na_Dysku(var Plik:file);
begin
  TObiekt.Zapisz_Na_Dysku(Plik);           {zapis głównego obiektu self}
  Produkt^.Zapisz_Na_Dysku(Plik);         {zapis pola obiektowego}
end;

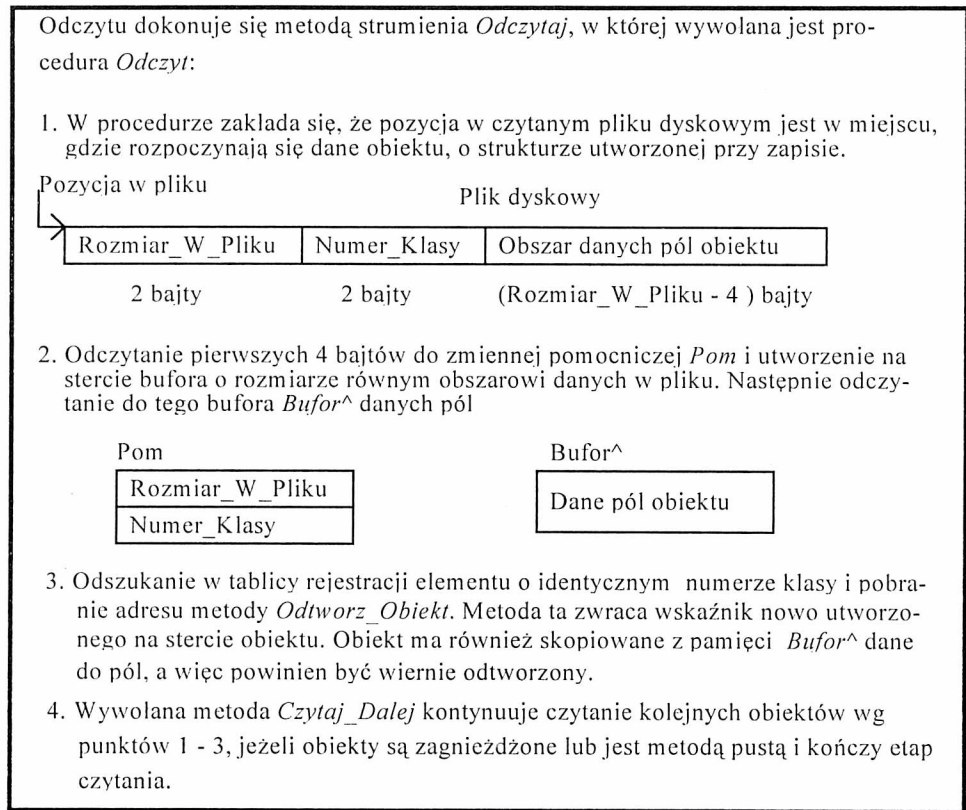
```

Odczyt

Zasady dotyczące przygotowania metod obiektu do odczytu:

1. Przodkiem wszystkich obiektów musi być klasa *TObiekt*.
2. Każda klasa musi mieć zdefiniowaną metodę (typu funkcja) *Odtworz_Objekt*. W tej metodzie tworzony jest obiekt danej klasy za pomocą procedury *New*. Następnie z bufora przekazanego do metody pobierane są dane obiektu tworzące blok o strukturze ustalonej w czasie zapisu i przypisywane są do odpowiednich pól.
3. Klasy mające pola obiektowe, oprócz metody *Odtworz_Objekt* muszą mieć zdefiniowaną metodę *Czytaj_Dalej*, w której odtwarzane są pola obiektowe. W przypadku braku pól obiektowych metoda *Czytaj_Dalej* jest dziedziczona od klasy *TObiekt* i jest metodą pustą.

Na rys. 31. przedstawiono opisane czynności wykonywane przy odczycie obiektu.



Rys. 31. Algorytm odczytu ze strumienia

Poniżej przedstawiono wspomniane wyżej metody dla obiektów klas z drzewa *TProdukt...* Ponieważ nie mają obiektów zagnieżdżonych, bezpośrednio dziedziczą metodę *Czytaj_Dalej* od klasy *TObiekt*, natomiast mają własne metody *Odtworz_Objekt*. Bezpośredni odczyt wykonuje metoda strumienia *Odczytaj*, która inicjuje cały proces odczytu.

```

procedure TStrumien.Odczytaj(var Obiekt:PObiekt);
begin
    Odczyt(Obiekt, Plik);
end;

```

```

procedure Odczyt(var Obiekt:PObiekt var Plik:file);
{Procedura wykorzystywana przy definiowaniu metod odczytu Czytaj_Dalej w obiektach
dziedziczacych po TObiekt.}
var Pom      : array[1..2]of Integer;   {tablica dla rozmiaru i numeru obiektu}
      Bufor    : PBuf_Plik;
      Rozmiar_W_Pliku,J,Ile : Word;
begin
  BlockRead(Plik, Pom, 4, Ile);                                {krok 2 z rys. 31}
  if Ile<>4 then
    Halt;
  Rozmiar_W_Pliku := Pom[1] - 4;
  Pamiec (Rozmiar_W_Pliku);
  GetMem(Bufor,Rozmiar_W_Pliku);                               {krok 2 z rys. 31}
  BlockRead(Plik,Bufor^,Rozmiar_W_Pliku, Ile);
  if Ile <> Rozmiar_W_Pliku then
    Halt;
  J := Szukaj_Obiekt_W_Tablicy(Pom[2]);                         {krok 3 z rys. 31}
  if J = 0 then
    begin
      Writeln('Brak obiektu o numerze ',Pom[2]);
      Readln;
      Halt;
    end;
  Obiekt:=Rejestr^[J].Met_Wczyt (Bufor,J);                     {krok 3 z rys. 31}
  { metoda wirtualna Odtworz_Obiekt wywołana jest za pomocą parametru proceduralnego;}
  FreeMem(Bufor,Rozmiar_W_Pliku);
  if Obiekt <> nil then
    Obiekt^.Czytaj_Dalej (Plik);
    {krok 4 z rys. 31 - dalsze czytanie, jeżeli obiekt zawiera obiekty zagnieżdżone}
end;

function TObiekt.Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;
{Metoda zwraca wskaźnik równy nil, ponieważ dotyczy klasy abstrakcyjnej.}
begin
  Odtworz_Obiekt:= nil;
end.

```

Metody odczytu obiektów z drzewa *TProdukt..* zdefiniowano w module **UCW4_1**. Pokrywają one wirtualną metodę *Odtworz_Obiekt* abstrakcyjnej klasy *TObiekt*.

function TProdukt.Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;
{Metoda wykorzystana do wykonania kroku 3 algorytmu z rys. 27. W linii (1) tworzony jest obiekt dynamiczny klasy TProdukt, w linii (2) przypisywane są odczytane ze strumienia wartości pól; w linii (3) zwracany jest wskaźnik utworzonego obiektu.}

```
type T = record
    A : Lan_1;
    B : Real;
end;
PT = ^T;
var Pom : PProdukt;
begin
    Pamiec (SizeOf (TProdukt_1)); New (Pom, Inicjuj);
    Pom^.Nadaj_Nazwe (PT (Tab) ^.A);
    Pom^.Nadaj_Cene (PT (Tab) ^.B);
    Odtworz_Objekt := Pom;
end;
```

```
function TProdukt_1.Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;
type T = record
    A : Lan_1;
    B : Real;
    C : Integer;
end;
Pt = ^T;
var Pom : PProdukt_1;
begin
    Pamiec (SizeOf (TProdukt_1)); New (Pom, Inicjuj);
    Pom^.Nadaj_Nazwe (Pt (Tab) ^.A);
    Pom^.Nadaj_Cene (Pt (Tab) ^.B);
    Pom^.Nadaj_Gatunek (Pt (Tab) ^.C);
    Odtworz_Objekt := Pom;
end;
```

```
function TProdukt_2.Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;
type T = record
    A : Lan_1;
    B : Real;
    C,D : Integer;
end;
Pt = ^T;
var Pom : PProdukt_2;
```

begin

```
Pamiec (SizeOf (TProdukt_2)) ;
New(Pom, Inicjuj) ;
Pom^.Nadaj_Nazwe (Pt (Tab) ^.A) ;
Pom^.Nadaj_Cene (Pt (Tab) ^.B) ;
Pom^.Nadaj_Gatunek (Pt (Tab) ^.C) ;
Pom^.Nadaj_Podatek (Pt (Tab) ^.D) ;
Odtworz_Objekt := Pom;
```

end;

W przypadku obiektów zagnieżdżonych należy pokryć metodę *TObiekt.Czytaj_Dalej*, w której należy odtworzyć strukturę pól obiektu np. dla klasy *TZakup* mamy:

```
procedure TZakup.Czytaj_Dalej (var Plik:file) ;
```

{Metoda odczytuje obiekt zagnieżdżony Produkt. Pola nieobiektywne zostały już odczytane przez strumień jego metodą Odczytaj.}

begin

```
Odczyt (PObiekt (Produkt), Plik) ;
```

end;

Zamknięcie strumienia

Etap ten zamyka plik dyskowy skojarzony ze strumieniem i powoduje usunięcie ze sterty obiektu *TStrumien* tzn:

```
Dispose (Strumien, Usun) ;
```

gdzie metoda **destructor** klasy *TStrumien* oraz metoda *Koniec* zostały zdefiniowane następująco:

```
procedure TStrumien.Koniec;
```

begin

```
Close(Plik) ;
```

end;

```
destructor TStrumien. Usun;
```

begin

```
Koniec;
```

end;

W przypadku, gdy nastąpi koniec programu, można usunąć z pamięci tablicę rejestracji klas za pomocą procedury *Usun_Rejestr*.

```
procedure Usun_Rejestr;
begin
  if Rejestr <> nil then
    FreeMem(Rejestr, Licznik_Rejestracji*SizeOf(Obiekt_Rek));
    Rejestr := nil;
end;
```

Definicja metody *TStrumien.Podaj_Plik* jest podana w rozdz. 5.4 (str. 151).

5.3. Przykład programu przetwarzającego obiekty w strumieniach

Program **CW4_21** wykonuje zapis obiektów do strumienia, natomiast program **CW4_22** dokonuje sekwencyjnego odczytu tych obiektów. Wykorzystano dodatkowo następujące moduły: **UCW4_2**, **UMENU_1** oraz moduł **STRUK4_1**. Zawartość tych modułów jest opisana dalej.

Przykład 26.

```
unit STRUK4_1;
interface
uses Crt, PLIKI_1, UMENU_1;

PObiekt_1 = ^TObiekt_1;
TObiekt_1 = object (TObiekt);
    constructor Inicjuj;
    destructor Usun;virtual;
end;

PKolekcja = ^TKolekcja;
TKolekcja = object (TObiekt)
    Tablica          : PTab_Kolekcji;
    Ilosc, Ostatni   : Integer;
    { deklaracje z modulu STRUK322, rozdz. 4.4}
function Odtworz_Obiekt (Tab: PBuf_Plik) : PObiekt;virtual;
function Zachowaj_Obiekt (Tab: PBuf_Plik) : Integer;virtual;
end;
```

```

PKolekcja_1 = ^TKolekcja_1;
TKolekcja_1 = object (TKolekcja)
    { deklaracje z modulu STRUK322, rozdz. 4.4}
    function Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;virtual;
    procedure Czytaj_Dalej(var Plik:file);virtual;
    procedure Zapisz_Na_Dysku(var Plik:file);virtual;
end;

implementation
{-----TKolekcja-----}
    {definicje pozostalych metod klasy TKolekcja z modulu STRUK322, rozdz. 4.4}
function TKolekcja.Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;
type T = record
    A,B : Integer;
    end;
    Pt = ^T;
var Pom : PKolekcja;
begin
    Pamiec (SizeOf(TKolekcja));
    New (Pom, Inicjuj (Pt (Tab)^.A));
    Move (Tab^, Pom^.Ilosc, 4);
    Odtworz_Objekt := Pom;
end;

function TKolekcja.Zachowaj_Objekt (Tab:PBuf_Plik):Integer;
var I : Integer;
begin
    I := 4;
    if Tab<> nil then
        Move (Ilosc, Tab^, I);
        Zachowaj_Objekt := I
    end;
{-----TKolekcja_1-----}
    {definicje pozostalych metod z modulu STRUK322, rozdz. 4.4}
function TKolekcja_1.Odtworz_Objekt (Tab:PBuf_Plik):PObiekt;
type T = record
    A,B : Integer;
    end;
    Pt = ^T;
var Pom : PKolekcja_1;

```

```

begin
  Pamiec (SizeOf (TKolekcja_1));
  New (Pom, Inicjuj (Pt (Tab) ^ .A));
  Move (Tab ^, Pom ^ .Ilosc, 4);
  Odtworz_Objekt := Pom;
end;

procedure TKolekcja_1.Zapisz_Na_Dysku (var Plik : file);
{Metoda zapisuje elementy kolekcji do strumienia, zakładając, że są obiektami i dziedziczą po
klasie TObiekt wirtualną metodę Zapisz_Na_Dysku.}
  var I : Integer;
  begin
    TObiekt.Zapisz_Na_Dysku (Plik); {zapisane pola Ilosc i Ostatni}
    for I:=1 to Ostatni do {zapis obiektów dziedziczących po TObiekt}
      PObiekt (Tablica ^ [I]) ^ .Zapisz_Na_Dysku (Plik);
    end;

procedure TKolekcja_1.Czytaj_Dalej (var Plik: file);
{Metoda odczytuje ze strumienia obiekty dziedziczące po klasie TObiekt i wstawia je do
kolekcji.}
  var I, J : Integer;
      Pom : Pointer;
  begin
    J := Ostatni; Ostatni := 0;
    for I := 1 to J do
      begin
        Odczyt (PObiekt (Pom), Plik);
        Dodaj (Pom);
      end;
    end;
  end.
{koniec modułu STRUK4_1}

```

Klasy *TZakup* i *TRachunek* zadeklarowane w module **UCW4_2** dziedziczą przez klasę *TObiekt_1* od klasy *TObiekt* wirtualne metody zapisu i odczytu w strumieniu.

```

unit UCW4_2;
interface
uses Crt, STRUK4_1, PLIKI_1, UMENU_1;
      {deklaracje klas TProdukt.. z modułu UCW4_1, rozdz. 5.2, przykład 25}
const Max_Ilosc = 4      Max_Zakupow = 10;
      Ilosci : array [1..Max_Ilosc] of Integer = (1, 2, 3);

```

```

PZakup = ^TZakup;
TZakup = object (TObiekt_1)
    private      Produkt           : PProdukt;
                Numer_Zakupu, Liczba_Produktow : Integer;
                                (deklaracja klasy z modulu UCW3_22)
function Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;virtual;
procedure Czytaj_Dalej (var Plik:file);virtual;
function Zachowaj_Obiekt (Tab:PBuf_Plik):Integer;virtual;
procedure Zapisz_Na_Dysku (var Plik:file);virtual;
end;

PRachunek = ^TRachunek;
TRachunek = object (TObiekt_1)
    private  Zakupy           : PKolekcja_1;
            Opis_Rachunku   : Lan_1;
                                (deklaracja klasy z modulu UCW3_22)
constructor Inicjuj_Obiekt;
function Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;virtual;
procedure Czytaj_Dalej (var Plik : file);virtual;
function Zachowaj_Dane_Obiektu (Tab:PBuf_Plik):Integer;virtual;
procedure Zapisz_Na_Dysku (var Plik:file);virtual;
end;

implementation
                                {definicje metod dla klas z drzewa TProdukt.z modulu UCW4_1}
                                { definicje pozostalych metod dla klas TZakup oraz TRachunek z modulu UCW3_22}
function TZakup.Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;
    type T = record
        A,B : Integer;
    end;
                                {struktura danych nieobiektowych}
    Pt := ^T;
    var Pom : PZakup;
    begin
        Pamiec (SizeOf (TZakup));
        New (Pom, Inicjuj (nil));
        Pom^.Ustaw_Liczbe_Produktow (Pt (Tab) ^.B);
        Pom^.Ustaw_Numer_Zakupu (Pt (Tab) ^.A);
        Odtworz_Obiekt := Pom;
    end;

```



```
procedure TZakup.Czytaj_Dalej(var Plik : file);  
  begin  
    Odczyt(PObiekt(Produkt), Plik);  
  end;  
  
function TZakup.Zachowaj_Dane_Objektu(Tab:PBuf_Plik):Integer;  
  var I : Integer;  
  begin  
    I := 4;  
    if Tab<> nil then  
      Move (Numer_Zakupu, Tab^, I);  
      Zachowaj_Dane_Objektu := I  
    end;  
  
procedure TZakup.Zapisz_Na_Dysku(var Plik:file);  
  begin  
    TObiekt.Zapisz_Na_Dysku(Plik);  
    Produkt^.Zapisz_Na_Dysku(Plik);  
  end;  
{-----TRachunek-----}  
constructor TRachunek.Inicjuj_Objekt;  
{ Konstruktor jest używany tylko przy odczycie obiektu ze strumienia. Obiekt zagnieżdżony  
Zakupy zostanie utworzony po wywołaniu metody Czytaj_Dalej}  
  begin  
  end;  
  
function TRachunek.Odtworz_Objekt(Tab:PBuf_Plik):PObiekt;  
  type T = record  
    A : Lan_1; (struktura pól klasy)  
  end;  
  Pt = ^T;  
  var Pom : PRachunek;  
  begin  
    Pamiec(SizeOf(TRachunek));  
    New(Pom, Inicjuj_Objekt);  
    Pom^.Ustaw_Opis_Rachunku(Pt(Tab)^.A);  
    Odtworz_Objekt := Pom;  
  end;
```

```

procedure TRachunek.Czytaj_Dalej(var Plik : file);
begin
    Odczyt(PObiekt(Zakupy), Plik);
end;

function TRachunek.Zachowaj_Obiekt(Tab:PBuf_Plik):Integer;
    var I : Integer;
begin
    I := 21;
    if Tab<> nil then
        Move (Opis_Rachunku, Tab^, I);
        Zachowaj_Dane_Obiektu := I
    end;

procedure TRachunek.Zapisz_Na_Dysku(var Plik:file);
{Metoda najpierw zapisuje swoje pole nieobiektove Opis_Rachunku (linia 1), a potem 'każde'
zapisac się kolekcji (linia 2).}
begin
    TObiekt.Zapisz_Na_Dysku(Plik);                                {1}
    Zakupy^.Zapisz_Na_Dysku(Plik);                                {2}
end;
end.                                                                {koniec modulu UCW4_2}

```

Program CW4_21 zapisuje obiekty do strumienia.

```

program CW4_21;
uses Crt, PLIKI_1, UCW4_2, UMENU_1, STRUK4_1;
    var Strumien          : PStrumien;
        Lista_Rachunkow  : PKolekcja_1;

procedure Zarejestruj_Obiekty;
begin
    Rejestracja_W_Strumieniu(EOF(TKolekcja_1), 1);
    Rejestracja_W_Strumieniu(EOF(TRachunek), 2);
    Rejestracja_W_Strumieniu(EOF(TZakup), 4);
    Rejestracja_W_Strumieniu(EOF(TProdukt), 5);
    Rejestracja_W_Strumieniu(EOF(TProdukt_1), 6);
    Rejestracja_W_Strumieniu(EOF(TProdukt_2), 7);
end;                                                                {koniec procedury Zarejestruj_Obiekty}

```

```

procedure Dodaj_Rachunek (Liczba : Integer);
    {Liczba - dana określająca liczbę zakupów na rachunku}

var Pom : PRachunek;

begin
    Pamięć(SizeOf(TRachunek));
    case Liczba of
        1: begin
            Pamięć(SizeOf(TRachunek)); Pom:=New(PRachunek, Inicjuj(1));
            Pom^.Ustaw_Opis_Rachunku('Pierwszy Rachunek');
            Pamięć(SizeOf(TProdukt));
            Pom^.Dopisz_Zakup(1,1,1,1,0,New(PProdukt, Inicjuj));
            end;
        2: begin
            Pamięć(SizeOf(TRachunek)); Pom:=New(PRachunek, Inicjuj(2));
            Pom^.Ustaw_Opis_Rachunku('Drugi Rachunek');
            Pamięć(SizeOf(TProdukt));
            Pom^.Dopisz_Zakup(1,1,1,1,0,New(PProdukt, Inicjuj));
            Pamięć(SizeOf(TProdukt_1));
            Pom^.Dopisz_Zakup(2,2,2,2,22,New(PProdukt_1, Inicjuj));
            end;
        3: begin
            Pamięć(SizeOf(TRachunek)); Pom:=New(PRachunek, Inicjuj(3));
            Pom^.Ustaw_Opis_Rachunku('Trzeci Rachunek');
            Pamięć(SizeOf(TProdukt));
            Pom^.Dopisz_Do_Rachunku(1,1,1,1,0,New(PProdukt, Inicjuj));
            Pamięć(SizeOf(TProdukt_1));
            Pom^.Dopisz_Zakup(2,2,2,2,0,New(PProdukt_1, Inicjuj));
            Pamięć(SizeOf(TProdukt_2));
            Pom^.Dopisz_Zakup(3,3,3,3,22,New(PProdukt_2, Inicjuj));
            end;
    end;
    Lista_Rachunkow^.Dodaj(Pom);
end;
    {koniec procedury Dodaj_Rachunek}

procedure Demo_Rachunki;

var I, J:Integer;

begin
    Pamięć(SizeOf(TKolekcja_1));
    Lista_Rachunkow := New(PKolekcja_1, Inicjuj(3));
    for I:=1 to 3 do    Dodaj_Rachunek(I);
end;
    {koniec procedury Demo_Rachunki}

```

```

procedure Zapisz_Objekty;
  var I, J : Integer;
      Pom  : PRachunek;
begin
  Strumien^.Zapisz(Lista_Rachunkow);
end;                                     {koniec procedury Zapisz_Objekty}

```

```

procedure Usun_Objekty_Demo;
  var I, J : Integer;
      Pom  : PRachunek;
begin
  Dispose(Lista_Rachunkow, Usun);
  Usun_Rejestr;
end;                                     {koniec procedury Usun_Objekty}

```

```

begin                                     {program główny CW4_21}
  ClrScr;
  Pauza;
  Demo_Rachunki;
  Zarejestruj_Objekty;                     {krok 1 z rys. 30}
  Pamiec(SizeOf(TStrumien));               {krok 2 z rys. 30}
  New(Strumien, Inicjuj('Plik_2', False));
  Zapisz_Objekty;                           {krok 3 z rys. 30}
  Dispose(Strumien, Usun);                  {krok 4 z rys. 30}
  Usun_Objekty_Demo;
  Pauza;
end.                                     {koniec programu CW4_21}

```

Program CW4_22 odczytuje obiekty ze strumienia.

```

program CW4_22;
uses Crt, UCW4_2, Struk4_1, PLIKI_1, UMENU_1;
var Strumien      : PStrumien;
    Lista_Rachunkow : PKolekcja_1;

procedure Zarejestruj_Objekty;             {definicja procedury z programu CW4_21}

procedure Usun_Objekty_Demo;               {definicja procedury z programu CW4_21}

```

```
procedure Odczytaj_Objekty;
begin
  Strumien^.Odczytaj (PObiekt(Lista_Rachunkow));
end;

procedure Wydruk_Rachunkow;
var I, J : Integer;
    Pom : PRachunek;
begin
  J:=Lista_Rachunkow^.Podaj_Liczbe_Elementow;
  for I := 1 to J do
    begin
      Pom:=Lista_Rachunkow^.Podaj_Element(I);
      Pom^.Wydruk_Rachunku;
    end;
end;

begin
  ClrScr;
  Pauza;
  Zarejestruj_Objekty;
  Pamiec (SizeOf (TStrumien));
  New (Strumien, Inicjuj ('Plik_2', True));
  Odczytaj_Objekty;
  Dispose (Strumien, Usun);
  Wydruk_Rachunkow;
  Usun_Objekty_Demo;
  Pauza;
end.
```

{początek programu CW4_22}

{krok 1 z rys. 31}

{krok 2 z rys. 31}

{krok 3 z rys. 31}

{krok 4 z rys. 31}

{koniec programu CW4_22}

5.4. Niesekwencyjny zapis i odczyt obiektów - zasoby

Narzędzia do sekwencyjnego zapisu i odczytu obiektów w strumieniu nie są wygodne w użyciu. W przypadku, gdy należy odczytywać wybrane obiekty, wygodniej stosować metodę niesekwencyjnego przetwarzania strumienia.

W metodzie tej obiekt jest odszukiwany w „strumieniu” za pomocą niepowtarzalnego klucza. Jest nim łańcuch znaków. Taki sposób jest możliwy tylko

wtedy, gdy przy zapisie obiektu do strumienia zapisuje się dodatkową informację wraz z kluczem.

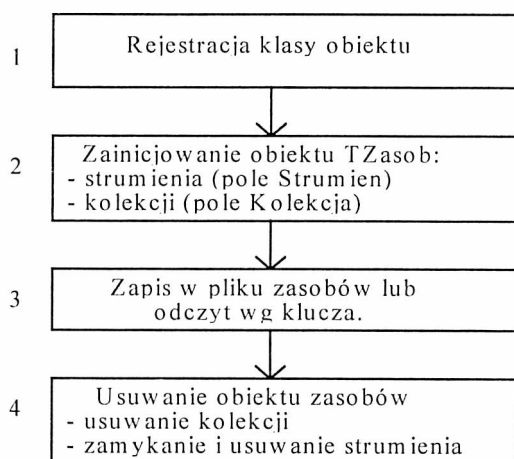
Obiekt umożliwiający taką operację nazywamy *obiektem zasobów* lub *plikiem zasobów*, natomiast obiekty lub grupy obiektów zapisywane lub odczytywane tą metodą nazywamy *zasobami*. Tak zapisywane i odczytywane obiekty najczęściej są elementami Menu, Pomocy itp. W przedstawionej metodzie założono, że w pliku zapisywane są tylko zasoby. W rozdziale 5.5 przedstawiono przykład programowy, w którym wykorzystano przetwarzanie pliku zasobów.

Aby wykorzystać gotowe narzędzia do zapisu i odczytu niesekwencyjnego obiektów, należy zdefiniować w ich klasach takie same metody, jakie opisano przy omawianiu strumieni. Natomiast w programie do przetwarzania używa się obiektu klasy

Klasa obiektu zasobów jest zadeklarowana następująco w module **PLIKI_2**:

```
unit PLIKI_2;
interface
uses STRUK3_2;                                {deklaracje z modułu PLIKI_1, rozdz. 5.2}
PZasob = ^TZasob;11
TZasob = object
  Strumien : PStrumien;
  Kolekcja : PKolekcja;
  constructor Inicjuj(N_Strumien:PStrumien);
  destructor Usun;
  procedure Utworz_Kolekcje;
  function Liczba_Zasobow:Integer;
  function Odczytaj_Zasob(Klucz:string):PObiekt;
  procedure Zapisz_Zasob(Element:PObiekt;Klucz:string);
end;
implementation
                                                    {definicje z modułu PLIKI_1}
                                                    {definicje klasy TZasob}
end.                                                    {koniec modułu PLIKI_2}
```

Na rys. 32 przedstawiono algorytm przetwarzania pliku zasobów przy korzystaniu z obiektu zasobu.



Rys. 32. Algorytm przetwarzania plików zasobów

W programach zapis lub odczyt obiektów za pomocą obiektu klasy *TZasob* odbywa się przy użyciu metod: *Zapisz_Zasob* oraz *Odczytaj_Zasob*. Proces ten polega na zapisie lub odczycie obiektu z jego obiektami zagnieżdżonymi, uzupełnionymi o dodatkową informację, w której między innymi jest zawarty klucz.

Rejestracja obiektów

Proces rejestracji obiektów, wchodzących w skład zasobów, został już przedstawiony przy omawianiu strumieni w rozdziale 5.2.

Zainicjowanie obiektu zasobów

Celem tej metody jest przygotowanie obiektu zasobów do zapisu lub odczytu zasobów w pliku dyskowym. Metoda spełnia następujące zadania:

- inicjuje pole *Strumien* wartością przekazanego w liście parametrów wskaźnika strumienia (parametr *N_Strumien*)
- przygotowuje wewnętrzną kolekcję *Kolekcja* do przechowywania wskaźników elementów typu *T_Zasob*.

W programie należy wywołać następującą instrukcję:

```
New(Zasob, Inicjuj(New(PStrumien, Inicjuj(Nazwa_Pliku, Sposob))));
```

gdzie *Sposob* określa tryb otwarcia pliku (*False* - do zapisu, *True* - do odczytu).

Poniżej przedstawiona jest definicja metody **constructor** inicjującej w pamięci obiekt *TZasob*:

```
constructor TZasob.Inicjuj (N_Strumien:PStrumien) ;
begin
  Pamiec (SizeOf (TKolekcja) ) ;
  New (Kolekcja, Inicjuj (10) ) ;
  Strumien := N_Strumien;
  Utworz_Kolekcje;
end;
```

Metoda *Utworz_Kolekcje* jest wykonana w całości tylko wtedy, gdy plik zasobów jest otwarty do odczytu. Podstawą tworzenia kolekcji są dane o strukturze *T_Zasob*, umieszczone dodatkowo w pliku razem z danymi zapisywanymi przez wewnętrzny strumień.

Element typu *T_Zasob* jest generowany w czasie zapisu zasobu i umieszczany w strumieniu przed danymi zasobu.

```
P_Zasob = ^T_Zasob;           {deklaracje z części implementation modułu PLIKI_2}
T_Zasob =
record
  Pos : Longint;           {wskaźnik położenia w pliku danych obiektowych zasobu}
  Klucz : string;         {niepowtarzalny klucz zasobu}
  Size : Word;             {rozmiar danych obiektowych zasobu}
end;
```

Po otwarciu pliku zasobów do odczytu, do kolekcji zapisywane są wszystkie elementy typu *T_Zasob*, pobrane z tego pliku. Liczba tych elementów jest równa liczbie wszystkich zasobów zapisanych w pliku zasobów.

Poniżej przedstawiono metodę *Utworz_Kolekcje*. Oczywiście, kolekcja powstanie tylko wtedy, gdy plik z zasobami istnieje i zawiera dane (tzn. jest spełniony warunek:

```
FileSize(Plik) <> 0 ).
```

Algorytm tworzenia kolekcji:

1. Warunek pętli: jeżeli plik nie jest pusty lub wyczerpany, powtarzaj;
2. Utwórz element kolekcji na sterzie (**record** typu *T_Zasob*);
3. Przeczytaj z pliku dane typu *T_Zasob*;
4. Dodaj element z danymi do kolekcji obiektu *TZasob*;

5. Ustaw wskaźnik położenia w pliku zasobów na początek następnego zasobu i przejdź do kroku 1, jeżeli warunek powtarzania pętli jest spełniony.

```
function TStrumien.Podaj_Plik : Pointer;
```

```
{Metoda udostępnia zmienną plikową klasy TStrumien.}
```

```
begin
```

```
    Podaj_Plik := @Plik;
```

```
end;
```

```
procedure TZasob.Utworz_Kolekcje;
```

```
{W liniach (1, 2, 3, 4) zaznaczono kroki algorytmu tworzenia kolekcji.}
```

```
var I      : Integer;
```

```
    Blad : Word;
```

```
    Pom  : P_Zasob;
```

```
    Plik : ^file;
```

```
begin
```

```
    Plik := Strumien^.Podaj_Plik;
```

```
if FileSize(Plik^) <> 0 then
```

```
    repeat
```

```
        Pamiec(SizeOf(T_Zasob));
```

```
        New(Pom);
```

```
{1}
```

```
        BlockRead(Plik^, Pom^, SizeOf(Pom^), Blad);
```

```
{2}
```

```
        if Blad = SizeOf(Pom^) then Kolekcja^.Dodaj(Pom)
```

```
{3}
```

```
    else
```

```
        begin
```

```
            Dispose(Pom);
```

```
            Break;
```

```
        end;
```

```
        Seek(Plik^, Pom^.Pos + Pom^.Size);
```

```
{4}
```

```
    until Blad = 0;
```

```
end;
```

Przetwarzanie obiektów w zasobach

Zapis

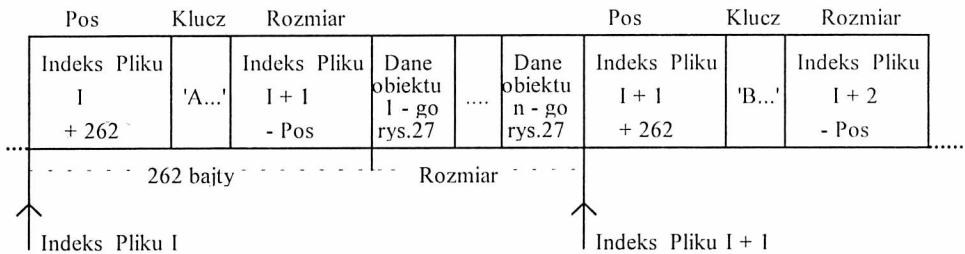
Zasady przygotowujące obiekt do zapisu są identyczne z zasadami zapisu do strumienia przedstawionymi w rozdziale 5.2.

Poniżej przedstawiono sposób zapisu danych do pliku dyskowego wraz z podanym kluczem wykonany przez obiekt zasobów.

Algorytm zapisu do pliku zasobów:

1. Przygotowanie danych typu *T_Zasob* dotyczących zasobu: pozycji w pliku, gdzie są zapisywane dane obiektu (pole *Pos*), klucza zasobu (pole *Klucz*);
2. Zapis obiektów, należących do zasobu, do strumienia (od położenia w pliku równym wartości pola *Pos*);
3. Obliczenie rozmiaru zapisanych obiektów w pliku i zapis tej danej do pola *Size*;
4. Zapis danych typu *T_Zasob* do pliku przed danymi obiektów należących do zasobu.
5. Ustawienie wskaźnika położenia pliku na pozycję zapisu następnego zasobu.

Na rys. 33. przedstawiono schematycznie fragment zawartości pliku zasobów.



Rys. 33. Fragment zawartości pliku zasobów na dysku

```

procedure TZasob.Zapisz_Zasob(Element:PObiekt; Klucz:string);
{W liniach (1, 2, 3, 4, 5) zaznaczono kroki algorytmu zapisu zasobów.}
var Pom : T_Zasob;
    I : Integer;
    Ile : Word;
    Plik : ^file;
begin
    Plik:=Strumien^.Podaj_Plik;
    I := SizeOf(Longint) + SizeOf(string) + SizeOf(Word); {1}
    Pom.Pos :=FilePos(Plik^) + I;
    Pom.Klucz := Klucz;
    Seek(Plik^, Pom.Pos);
    Strumien^.Zapisz(Element); {2}
    I := FilePos(Plik^); {3}
    Pom.Size := I - Pom.Pos;
    Seek(Plik^, Pom.Pos-SizeOf(Pom));
    BlockWrite (Plik^, Pom, SizeOf(Pom), Ile); {4}

```

```

if Ile <> SizeOf(Pom) then
  begin
    Rewrite(Plik^);
    Halt;
  end;
  Seek(Plik^, I);
end;

```

{5}

Odczyt

Zasady przygotowujące obiekt do odczytu są identyczne z zasadami przedstawionymi przy zapisie obiektów do strumienia w rozdziale 5.2.

Poniżej przedstawiono metodę odczytu zasobu z pliku zasobów *Odczytaj_Zasob* oraz pomocniczą metodę *Liczba_Zasobow*. Podstawą odczytu jest utworzona kolekcja po otwarciu pliku zasobów do odczytu.

Algorytm odczytu:

Powtarzaj w pętli przeszukiwanie kolekcji aż do jej wyczerpania:

1. Pobierz element kolekcji typu *T_Zasob*;
2. Sprawdź, czy klucz pobranego elementu jest równy szukanemu;
3. Jeżeli jest równy, ustaw wskaźnik położenia w pliku na pozycję *Pos*; jeżeli nie jest równy, skocz do kroku 1;
4. Odczytaj ze strumienia skojarzonego z plikiem zasobów obiekty należące do szukanego zasobu;
5. Zwróć wskaźnik do odczytanego zasobu.

```

function TZasob.Liczba_Zasobow : Integer;
begin
  Liczba_Zasobow := Kolekcja^.Podaj_Liczbe_Elementow;
end;

function TZasob.Odczytaj_Zasob(Klucz : string) : PObiekt;
{W liniach (1, 2, 3, 4, 5) zaznaczono kroki algorytmu odczytu zasobów}
var I      : Integer;
    Pom    : P_Zasob;
    Obiekt : PObiekt;
    Plik   : ^file;
begin
  Obiekt := nil;
  I := 1;
  Plik := Strumien^.Podaj_Plik;

```

```

while I <= Liczba_Zasobow do
begin
    Pom := Kolekcja^.Podaj_Element(I);           {1}
    if Pom^.Klucz = Klucz then                   {2}
    begin
        Seek(Plik^, Pom^.Pos);                   {3}
        Strumien^.Odczytaj(Obiekt); Break;       {4}
    end;
    Inc(I);
end;
Odczytaj_Zasob := Obiekt;                       {5}
end;

```

Zamknięcie obiektu zasobów

W tym etapie następuje:

- usuwanie wewnętrznej kolekcji;
- zamykanie strumienia oraz skojarzonego z nim dyskowego pliku zasobów
- usuwanie z pamięci obiektu zasobów.

W programie należy wywołać instrukcję:

Dispose(Zasob, Usun);

Poniżej przedstawiono **destructor** klasy *TZasob*.

```

destructor TZasob.Usun;
var I,J : Integer;
    Pom : P_Zasob;
begin
    J := Kolekcja^.Podaj_Liczbe_Elementow
    for I := 1 to J do
    begin
        Pom := Kolekcja^.Podaj_Element(I);
        Dispose(Pom);
    end;
    Dispose(Kolekcja, Usun);
    Dispose(Strumien, Usun);
end;

```

5.5. Przykład programu przetwarzającego obiekty w pliku zasobów

Programy **CW5_11** i **CW5_12** korzystają z zasobów zapisanych w pliku zasobów. Przedstawiono prosty przykład zapisu i odczytu obiektów klasy *TOkno_Zakupu*. W module **UMENU3_1** oraz **UMENU_2** przedstawiono sposób przygotowania metod przedstawionych w rozdziałach 5.2 oraz 5.4 dla klas, których obiekty mają być zapisane w pliku zasobów. Program **CW5_11** zapisuje obiekty do pliku zasobów.

Przykład 27.

```

program CW5_11;
uses Crt, UCW2_4, UMENU3_1, PLIKI_2, UMENU_2;
type TOkno = object
    Okno_Zakupu      : POkno_Zakupu;
    Zasob             : PZasob;
    procedure Inicjuj;
    procedure Wykonaj;
    procedure Usun;
end;

procedure TOkno.Inicjuj;
procedure Zarejestruj_Objekty_Zasobow;
begin
    Rejestracja_W_Strumieniu(Kind(TOkno_Zakupu), 1);
    Rejestracja_W_Strumieniu(Kind(TWyswietlacz), 2);
end;
begin
    Zarejestruj_Objekty_Zasobow;                                {krok 1 z rys. 32}
    Pamiec(SizeOf(TZasob)+SizeOf(TStrumien) + 16);
    New(Zasob, Inicjuj(New(PStrumien, Inicjuj('Plik.Res', False))));
                                                                {krok 2 z rys. 32}
end;                                                            {koniec metody Inicjuj}

procedure TOkno.Wykonaj;
var I, J      : Integer;
    Zakup       : PZakup;
procedure Zrob_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
    N_Gatunek, N_Podatek: Integer; N_Produkt: PProdukt);
begin
    Pamiec(SizeOf(TZakup));

```

```
New(Zakup, Inicjuj(N_Produkt));
Zakup^.Ustaw_Zakup(Indeks_Nazwy, Indeks_Ceny, Indeks_Ilosci,
                    N_Gatunek, N_Podatek);
```

```
end;
```

```
procedure Wykonanie_Zasobow_Zakupu(I:Integer);
```

```
begin
```

```
    {Przygotowanie i zapis kolejnych zasobów - realizacja kroku 3 algorytmu z rys. 32}
```

```
    Pamiec(SizeOf(TOkno_Zakupu));
```

```
    New(Okno_Zakupu, Inicjuj);
```

```
    Okno_Zakupu^.Zrob_Wyswietlacz_Produktu(Zakup);
```

```
    Zasob^.Zapisz_Zasob(Okno_Zakupu, 'Okno_Zakupu_'+Chr(48 + I)); {3}
```

```
    Dispose(Okno_Zakupu, Usun);
```

```
    Dispose(Zakup, Usun);
```

```
end;
```

```
begin
```

```
    Pamiec(SizeOf(TProdukt));
```

```
    Zrob_Zakup(1,1,1,1,1,New(PProdukt, Inicjuj));
```

```
    Wykonanie_Zasobow_Zakupu(1); {3}
```

```
    Pamiec(SizeOf(TProdukt_1));
```

```
    Zrob_Zakup(2,2,2,2,2,New(PProdukt_1, Inicjuj));
```

```
    Wykonanie_Zasobow_Zakupu(2); {3}
```

```
    Pamiec(SizeOf(TProdukt_2));
```

```
    Zrob_Zakup(3,3,3,3,22,New(PProdukt_2, Inicjuj));
```

```
    Wykonanie_Zasobow_Zakupu(3); {3}
```

```
end; {koniec metody Wykonaj}
```

```
procedure TOkno.Usun;
```

```
begin
```

```
    Dispose(Zasob, Usun);
```

```
    ClrScr;
```

```
    Usun_Rejestr;
```

```
    {krok 4 z rys. 32}
```

```
end; {koniec metody Usun}
```

```
var Okno_Glowne:TOkno;
```

```
begin
```

```
    {program główny CW5_11}
```

```
    Okno_Glowne.Inicjuj;
```

```
    Okno_Glowne.Wykonaj;
```

```
    Okno_Glowne.Usun;
```

```
end. {koniec programu CW5_11}
```

Program **CW5_12** odczytuje 10 razy z pliku zasobów jeden z trzech zasobów wyznaczając w sposób losowy jego klucz.

```

program CW5_12;
                                {deklaracja klasy TOkno jak w programie CW5_11}
procedure TOkno.Usun;          {definicja metody z programu CW5_11}

procedure TOkno.Inicjuj;
  procedure Zarejestruj_Objekty_Zasobow; {definicja z programu CW5_11}
  begin
    Zarejestruj_Objekty_Zasobow; {krok 1 z rys. 32}
    Pamiec (SizeOf (TZasob) + SizeOf (TStrumien) + 16); {krok 2 z rys. 32}
    New (Zasob, Inicjuj (New (PStrumien, Inicjuj ('Plik.Res', True))));
  end; {koniec metody Inicjuj}

procedure TOkno.Wykonaj;
  var I, J : Integer; {realizacja kroku 3 algorytmu z rys. 32.}
  begin
    for I:=1 to 10 do
      begin
        J := Random(3)+1;
        Okno_Zakupu:=Pointer (Zasob^.Odczytaj_Zasob ('Okno_Zakupu_'
                                                    +Chr (48+J) ));
        if Okno_Zakupu <>nil then
          begin
            Okno_Zakupu^.Wyswietl_Skladowe_Produktu;
            Dispose (Okno_Zakupu, Usun);
          end;
        end;
      end; {koniec metody Wykonaj}

end;

var Okno_Glowne : TOkno;
  begin {program główny CW5_12}
    Okno_Glowne.Inicjuj;
    Okno_Glowne.Wykonaj;
    Okno_Glowne. Usun;
  end. {koniec programu CW5_12}

```

Dalej przedstawiono moduł **UMENU3_1** oraz moduł **UMENU_2**. Moduły te powstały po rozbudowie modułów **UMENU1_2** oraz **UMENU_1**.

```

unit UMENU3_1;
interface
uses UMENU_2, UCW2_4, PLIKI_2, Crt;
type POkno_Zakupu = ^TOkno_Zakupu;
   Tokno_Zakupu = object (TObiekt)
       private   Wyszwieltacz_Opisu_Produktu : PWyszwieltacz;
               {deklaracje klasy TZakup z modulu UMENU1_2}

       constructor Inicjuj_1;
       function Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;virtual;
       procedure Czytaj_Dalej (var Plik:file);virtual;
       procedure Zapisz_Na_Dysku (var Plik:file);virtual;
end;
implementation
{-----Tokno_Zakupu-----}
               {definicje metod klasy Tokno_Zakupu z modulu UMENU1_2}
constructor Tokno_Zakupu.Inicjuj_1;
{Constructor do odczytu z pliku, gdyz obiekt zagniezdzony zostanie zainicjowany pozniej}
begin
end;

function Tokno_Zakupu.Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;
var Pom : POkno_Zakupu;
begin
    Pamiec (SizeOf (TOkno_Zakupu)) ;
    New (Pom, Inicjuj_1);      {obiekt bez zainicjowania obiektu klasy TWyszwieltacz }
    Odtworz_Obiekt := Pom;
end;

procedure Tokno_Zakupu.Czytaj_Dalej (var Plik:file);
{Metoda odtwarza strukture calosc-czesc}
begin
    Odczyt (PObiekt (Wyszwieltacz_Opisu_Produktu), Plik);
end;

procedure Tokno_Zakupu.Zapisz_Na_Dysku (var Plik : file);
{Metoda zapisuje strukture calosc-czesc}
begin
    TObiekt.Zapisz_Na_Dysku (Plik);
    Wyszwieltacz_Opisu_Produktu^.Zapisz_Na_Dysku (Plik);
end;
end.
               {koniec modulu UMENU3_1}

```



```

unit UMENU_2
interface
uses PLIKI_2, Crt;

PWyswietlacz = ^TWyswietlacz;
TWyswietlacz = object (TObiekt)
  Komunikat : Lan_3;
  Naglowek   : Lan_3;
  R_KomX1,R_KomY1,R_KomX2,R_KomY2 : Integer;
  {deklaracje z modulu UMENU_1}
  function Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;virtual;
  function Zachowaj_Obiekt (Tab:PBuf_Plik):Integer;virtual;
end;

implementation
  {definicje metod klasy TWyswietlacz z modulu UMENU_1}
function TWyswietlacz.Odtworz_Obiekt (Tab:PBuf_Plik):PObiekt;
  type T = record
    A, B:Lan_3;
    C, D, E, F:Integer;
  end;
  PT = ^T;
  var Pom:PWyswietlacz;
begin
  Pamiec (SizeOf (TWyswietlacz));
  New (Pom, Inicjuj (PT (Tab) ^ .C, PT (Tab) ^ .D, PT (Tab) ^ .E, PT (Tab) ^ .F));
  Pom ^ .Ustaw_Komunikat (PT (Tab) ^ .A;
  Pom ^ .Ustaw_Naglowek (PT (Tab) ^ .B);
  Odtworz_Obiekt := Pom;
end;

function TWyswietlacz.Zachowaj_Obiekt (Tab:PBuf_Plik):Integer;
  var I:Integer;
begin
  I := 2 *SizeOf (Lan_3) + 4 *SizeOf (Integer);
  if Tab<>nil then
    Move (Komunikat, Tab ^ ,I);
    Zachowaj_Obiekt:=I
  end;
end.
  {koniec modulu UMENU_2}

```

Zakończenie

Przedstawiono przykłady kilku rozwiązań programowych dotyczących zagadnienia sporządzania rachunków. Głównym celem było pokazanie różnych aspektów techniki obiektowej oraz możliwość ich porównania. Szczególnie atrakcyjne w programowaniu obiektowym jest elastyczne podejście do struktur danych dzięki zastosowaniu:

- hermetyzacji i dziedziczenia,
- polimorfizmu, obiektów dynamicznych,
- uniwersalnych struktur danych,
- abstrakcyjnych typów danych.

Stosując te elementy uzyskano programy zarządzające danymi o zmiennej strukturze (np. klasa *TZakup* zarządza różnymi obiektami z drzewa *TProdukt...*). Programy przedstawione w rozdziałach 4.2, 4.4, 5.3 oraz 5.5 były konstruowane w oparciu o następujące klasy dotyczące:

- struktur danych (moduły z serii **STRUK...** z rozdziału 4),
- zasobów i strumieni (moduły z serii **PLIK_** z rozdziału 5),
- klas problemu (moduły z serii **UCW...** z rozdziału 3)
- klas interfejsu (moduły z serii **UMENU..** z rozdziału 3).

Sposób korzystania z tych klas polegał na modernizacji pewnych metod w przypadku zastosowania uniwersalnych struktur danych (rozdz. 4.2 oraz 4.4) oraz na uzupełnianiu o metody wykonujące nowe zadania w klasie np. zapis lub odczyt w strumieniu lub zasobach (rozdz. 5.3 oraz 5.5).

Te niewielkie modyfikacje udowodniły ogromną elastyczność tworzenia oprogramowania obiektowego. Szczególnie ważne okazało się, iż wyniki analizy obiektowej są niezależne od realizacji programowej, w której klasy jedynie są wzbogacane o możliwości wykonania uniwersalnych czynności np. zapis lub odczyt w strumieniu itd..

Po umieszczeniu w gotowych modułach wyjściowych klas o charakterze uniwersalnym w zakresie pewnych zastosowań np. klasy interfejsu, strumieni, zasobów, dynamicznych struktur danych lub kolekcji, klasy potomne można konstruować poprzez wyszczególnianie w oprogramowaniu tych modułów i wykorzystanie mechanizmu dziedziczenia.

Przedstawione mechanizmy mają ułatwić czytelnikowi posługiwanie się profesjonalnymi bibliotekami obiektowymi, a jednocześnie konstruowanie własnego oprogramowania, dedykowanemu określonym potrzebom.

Literatura

- [1] Coad P, Nicola J. - *Programowanie obiektowe*, Wydawnictwo Oficyna Wydawnicza Read Me, Warszawa 1993.
- [2] Lachand T.-Robert - *Programowanie obiektowe w Turbo Pascalu*, Wydawnictwo Helion, Gliwice, 1993.
- [3] Kwaśnicka H. - *Programowanie obiektowe w Turbo Pascalu*, Wydawnictwo Oficyna Wydawnicza Politechniki Wrocławskiej, 1995.
- [4] Marciniak A. - *Borland Pascal 7.0*, Wydawnictwo Nakom, Poznań, 1994.
- [5] Sielicki A. (red.) - *Laboratorium programowania w języku Pascal*, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1994.
- [6] Taylor D.A. - *Technika obiektowa*, Wydawnictwo Helion, Gliwice 1994.
- [7] Turski W. - *Metodologia programowania*, Warszawa, WNT, 1989.
- [8] Wierzbicki M. - *Programowanie obiektowe w Turbo Pascalu 7.0 z wykorzystaniem biblioteki Turbo Vision*, Wydawnictwo Lupus, Warszawa, 1994.
- [9] Wirth N. - *Algorytmy + struktury danych = programy*, Wyd. 2. Warszawa, WNT, 1989.

Skorowidz

- abstrakcyjne, *Patrz* klasy, metody
- analiza obiektowa 7, 9, 160
- atrybut 9, 10, 11, 13, 14
- Borland Pascal 7.0
 - Abstract 48
 - constructor** 47, 49, 50, 53, 54, 55, 58, 60, 63
 - destructor** 54, 56, 58, 59, 64, 66
 - Dispose 54, 56, 58, 59
 - Fail 55, 60
 - inherited** 24
 - New 54, 55, 60
 - object** 17
 - Pointer 108
 - private** 18, 20
 - public** 18
 - self** 97, 99, 127, 128
 - TypeOf 50, 53, 70, 125, 128
 - virtual** 22, 46, 47, 48, 49
- definiowanie
 - metod 9, 18 *Patrz* klasy zdefiniowane w aplikacjach
 - pól 9, 13 *Patrz* klasy zdefiniowane w aplikacjach
 - usług 9, 13, 14
- deklaracje
 - klasy 17, 18
 - metod 17, 18, 46, 48
 - pól 17, 18
- dziedziczenie 6, 8, 17, 160
 - drzewo 9, 11, 12, 13, 21, 22, 23, 26, 27, 39, 55
 - metod 11, 12, 21
 - statycznych 17, 22, 23
 - wirtualnych 50, 51,
 - pól 11, 12, 22, 23
 - z przeskokiem 24
- hermetyzacja 6, 17, 18, 20, 160
- instrukcja przypisania 26, 27, 55
 - zasada przypisania obiektów statycznych 26
 - wskaźników obiektów 55
- język zorientowany obiektowo 3

klasy 6, 9, 10, 11, 12, 14, 15, 17

abstrakcyjne 48, 77, 78, 123

bazowe 11, 12, 21, 22, 55

korzeń klas 21, 22, 27, 39

podstawowe 12, 21, 22

specjalizacje 11, 21

uniwersalne 78, 94, 95, 160

wielokrotnego użytku 15

zadania 14

zdefiniowane w aplikacjach

interfejsu użytkownika 16, 77, 160

TMenu 77, 78, *Patrz* moduły **UMenu_1**, **UMenu_2**

TOkno 93, 155, *Patrz* programy **Cw2_52**, **CW5_11**, **CW5_12**

TMenu_Rachunku 88, *Patrz* moduły **UMENU1_2**, **UMENU2_1**,

TOkno_Rachunku 89, *Patrz* moduły **UMenu1_2**, **UMenu2_1**, **UMenu2_2**

TOkno_Zakupu 85, *Patrz* moduły **UMenu1_1**, **UMenu1_2**, **UMenu2_1**,
UMenu3_1

TWyswietlacz 78, *Patrz* moduły **UMenu_1**, **UMenu_2**

problemu 160

TProdukt, *TProdukt_1*, *TProdukt_2* 23, *Patrz* moduły **UCW1_1**,

UCW1_3, **UCW1_4**, **UCW1_5**, **UCW2_1**,

UCW2_2, **UCW2_3**, **UCW3_1**, **UCW3_2**,

UCW3_22, **UCW4_1**, **UCW4_2**

TRachunek, *TRachunek_1* 35, 72 *Patrz* moduły **UCW1_5**, **UCW2_5**,

UCW3_1, **UCW3_2**, **UCW3_22**, **UCW4_2**

TZakup 29, 67 *Patrz* moduły **UCW1_4**, **UCW1_5**, **UCW2_4**, **UCW2_5**,

UCW3_1, **UCW3_2**, **UCW3_22**, **UCW4_2**

struktur danych 160

TKolekcja 108, 112, *Patrz* moduły **STRUK3_2**, **STRUK3_22**, **STRUK4_1**

TKolekcja_1 111, 118, *Patrz* moduły **STRUK3_22**, **STRUK4_1**

TLista 96, 100, 101, *Patrz* moduły **STRUK3_1**

TObiekt 123 *Patrz* moduły **PLIKI_1**, **PLIKI_2**

TObiekt_1 118 *Patrz* moduły **STRUK322**, **STRUK4_1**

TStrumien 123, *Patrz* moduły **PLIKI_1**, **PLIKI_2**

TZasob 148, *Patrz* moduły **PLIKI_2**

kolekcje, *Patrz* struktura danych

komunikacja między obiektami 14, 27, 28, 84, 88, 89

konwersja typu zmiennej 55

lista

jednokierunkowa nieuporządkowana, *Patrz* struktura danych

parametrów formalnych, *Patrz* przekazanie parametrów

metody 6, 9, 17, 18, 19

abstrakcyjne 48

dodatkowe *Patrz* klasy zdefiniowane w aplikacjach

dostępu do pól 11, 14, *Patrz* klasy zdefiniowane w aplikacjach

dziedziczenie *Patrz* dziedziczenie metod

nagłówki 17, 18, 22, 27, 48, 50, 97, 128

pomocnicze 34, 37

problemu 10, 14, *Patrz* klasy zdefiniowane w aplikacjach

prywatna 18, 34, 37

przydzielanie metody

statycznej 22, 40, 41, 60-63

wirtualnej 46, 47, 50, 52, 60-63

pseudoabstrakcyjne 47

statyczne 22

wirtualne 46, 47

wykonujące zadanie klasy 14, *Patrz* metody problemu

zakrywanie 11, 22, 50

model danych 9, 13

sporządzania rachunków 7, 9, 10

moduły

PLIKI_1 123, 124, 127-130, 132-133, 135-136, 138-139

PLIKI_2 148, 150-154

STRUK3_1 95-96, 97-99

STRUK3_2 109-111

STRUK322 118-119

STRUK4_1 139-141

UCW1_1 18-19

UCW1_2 20-21

UCW1_3 24-26

UCW1_4 29-30

UCW1_5 34

UCW2_1 41-45

UCW2_2 48-49

UCW2_3 56-58

UCW2_4 67-68

UCW2_5 73

UCW3_1 102-103

UCW3_2 113

UCW3_22 119, 120

UCW4_1 124-125, 133-134, 136-138

UCW4_2 139, 141-144

- UMENU_1 78-84
- UMENU_2 157, 159
- UMENU1_1 85-86
- UMENU1_2 89-93
- UMENU2_1 108
- UMENU2_2 120
- UMENU3_1 157-158
- obiekty 6, 9, 10, 17
 - dynamiczne 54, 100
 - dynamiczne zagnieżdżone 67
 - statyczne 54
 - zagnieżdżone 11, 12, 13, 14, 28
- odczyt obiektów
 - niesekwencyjny, *Patrz* przetwarzanie obiektów w zasobach
 - sekwencyjny, *Patrz* przetwarzanie obiektów w strumieniu
- podprogramy 5
- pojęcia podstawowe języka 4
- polimorfizm 6, 39, 46, 49, 160
- programy 6
 - CW1_1 18, 19-20
 - CW1_2 20
 - CW1_3 26-27
 - CW1_4 28
 - CW1_5 33
 - CW1_6 38
 - CW2_1 41, 46
 - CW2_2 48, 50
 - CW2_3 56, 58
 - CW2_41 71-72
 - CW2_42 86, 87
 - CW2_51 76-77
 - CW2_52 93-94
 - CW3_1 107, 108
 - CW3_2 111, 120
 - CW4_1 125-126
 - CW4_2 126-127
 - CW4_21 139, 144-146
 - CW4_22 139, 146-147
 - CW5_11 155-156
 - CW5_12 157
- DESTRUCTORY 64-66

- METODY** 60, 61-63
- OBIEKTY_STATYCZNE** 54
- TABLICA_VMT** 52, 53
- WARIANT_1** 39, 40
- WARIANT_2** 39, 40
- programowanie 3
 - obiektywne 3, 6, 7, 9, 17, 160
 - strukturalne 3-6
 - wstępujące 3
 - zstępującym 3
- projektowanie obiektywne 9, 15
- prosty interfejs kontaktu z użytkownikiem 77
- przekazanie
 - obiektów przez
 - wartość 27
 - zmienną 39, 40
 - parametrów
 - przez wartość 27
 - przez zmienną 39, 40
 - wskaźników obiektów przez
 - wartość 55
 - zmienną 55
- przetwarzanie obiektów
 - w strumieniu 121, 122, 160
 - inicjowanie strumienia 130
 - odczyt 134
 - zapis 130
 - zamknięcie 138
 - w zasobach 121, 147, 152, 160
 - inicjowanie obiektu zasobu 149
 - odczyt 153
 - zapis 151
 - zamknięcie 154
- rejestracja klas obiektów 127, 149
- serta
 - zajmowanie *Patrz* Borland Pascal 7.0 New
 - zwalnianie *Patrz* Borland Pascal 7.0 Dispose, **destructor**
- struktura danych 3, 95
 - dynamiczna 15, 95
 - kolekcje 15, 108
 - listy 95

- strumień *Patrz* przetwarzanie obiektów w strumieniu
- uniwersalne 95, 160
- wzorcowe 95
- zasoby *Patrz* przetwarzanie obiektów w zasobach
- zmienna 67
- strumień, *Patrz* przetwarzanie obiektów w strumieniu
- symulacja 7
- tablica
 - metod wirtualnych (VMT) 49, 50, 51, 52, 53, 55, 61
 - wskaźników obiektów 95, *Patrz* struktura danych kolekcje
- technika obiektowa
 - OOA 7, 9
 - OOD 7, 9
 - OOP 6, 9
- typ danych
 - abstrakcyjny 48, 160
 - bazowy typu wskaźnikowego 55
 - obiekty *Patrz* klasy
 - wskaźnikowy 54, 55
- zainicjowanie obiektów
 - dynamicznych 55
 - statycznych 50, 54
- zapis obiektów
 - niesekwencyjny, *Patrz* przetwarzanie obiektów w zasobach
 - sekwencyjny, *Patrz* przetwarzanie obiektów w strumieniu
- zasoby, *Patrz* przetwarzanie obiektów w zasobach

SPIS TREŚCI

Wprowadzenie.....	3
1. Przygotowanie programu.....	9
1.1. Analiza obiektowa problemu.....	9
1.2. Projektowanie obiektowe.....	15
2. Hermetyzacja danych, dziedziczenie metod statycznych.....	17
2.1. Podstawowe zagadnienia.....	17
2.2. Obiekty zagnieżdżone.....	28
3. Polimorfizm, obiekty dynamiczne.....	39
3.1. Podstawowe zagadnienia.....	39
3.2. Obiekty dynamiczne.....	54
3.3. Dynamiczne obiekty zagnieżdżone.....	67
3.4. Prosty interfejs kontaktu z użytkownikiem.....	77
4. Uniwersalne struktury danych - listy, kolekcje.....	95
4.1. Struktury dynamiczne obiektów - listy jednokierunkowe.....	95
4.2. Przykład programu z zastosowaniem list obiektów.....	102
4.3. Przechowywanie wskaźników obiektów w tablicy - kolekcje.....	108
4.4. Przykład programu z zastosowaniem kolekcji obiektów.....	113
5. Strumień i zasoby.....	121
5.1. Podstawowe zagadnienia.....	121
5.2. Sekwencyjny zapis i odczyt obiektów - strumień.....	122
5.3. Przykład programu przetwarzającego obiekty w strumieniach.....	139
5.4. Niesekwencyjny zapis i odczyt obiektów - zasoby.....	147
5.5. Przykład programu przetwarzającego obiekty w pliku zasobów.....	155
Zakończenie.....	160
Literatura.....	161
Skorowidz.....	162



BIBLIOTEKA GŁÓWNA

291061/23

BI-28

Wydawnictwa Politechniki Wrocławskiej
są do nabycia w następujących księgarniach:
„Politechnika”

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
budynek A-1 PWr., tel. (0-71) 20 25 34;
„Tech”

plac Grunwaldzki 13, 50-377 Wrocław
budynek D-1 PWr., tel. (0-71) 20 32 52
Prowadzimy sprzedaż wysyłkową

ISBN 83-7085-228-9