Charakterystyka oprogramowania obiektowego

1. Definicja systemu informatycznego
2. Model procesu wytwarzania oprogramowania - model cyklu życia oprogramowania
3. Wymagania
4. Problemy z podejściem nieobiektowym
5. Podejście obiektowe – rozwiązanie pewnych problemów podejścia nieobiektowego

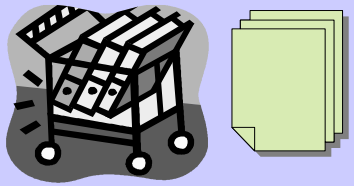
Charakterystyka oprogramowania obiektowego

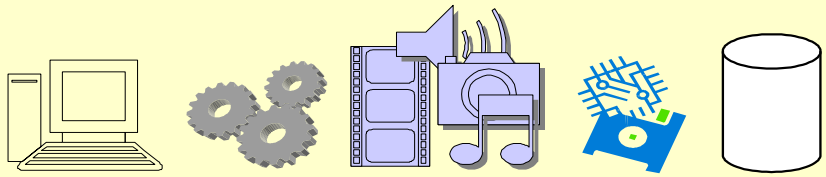
1. Definicja systemu informatycznego

System informatyczny

Nieformalny system informacyjny:
zasoby osobowe - ludzie

Formalny system informacyjny:
procedury zarządzania, bazy wiedzy





Techniczny system informacyjny:

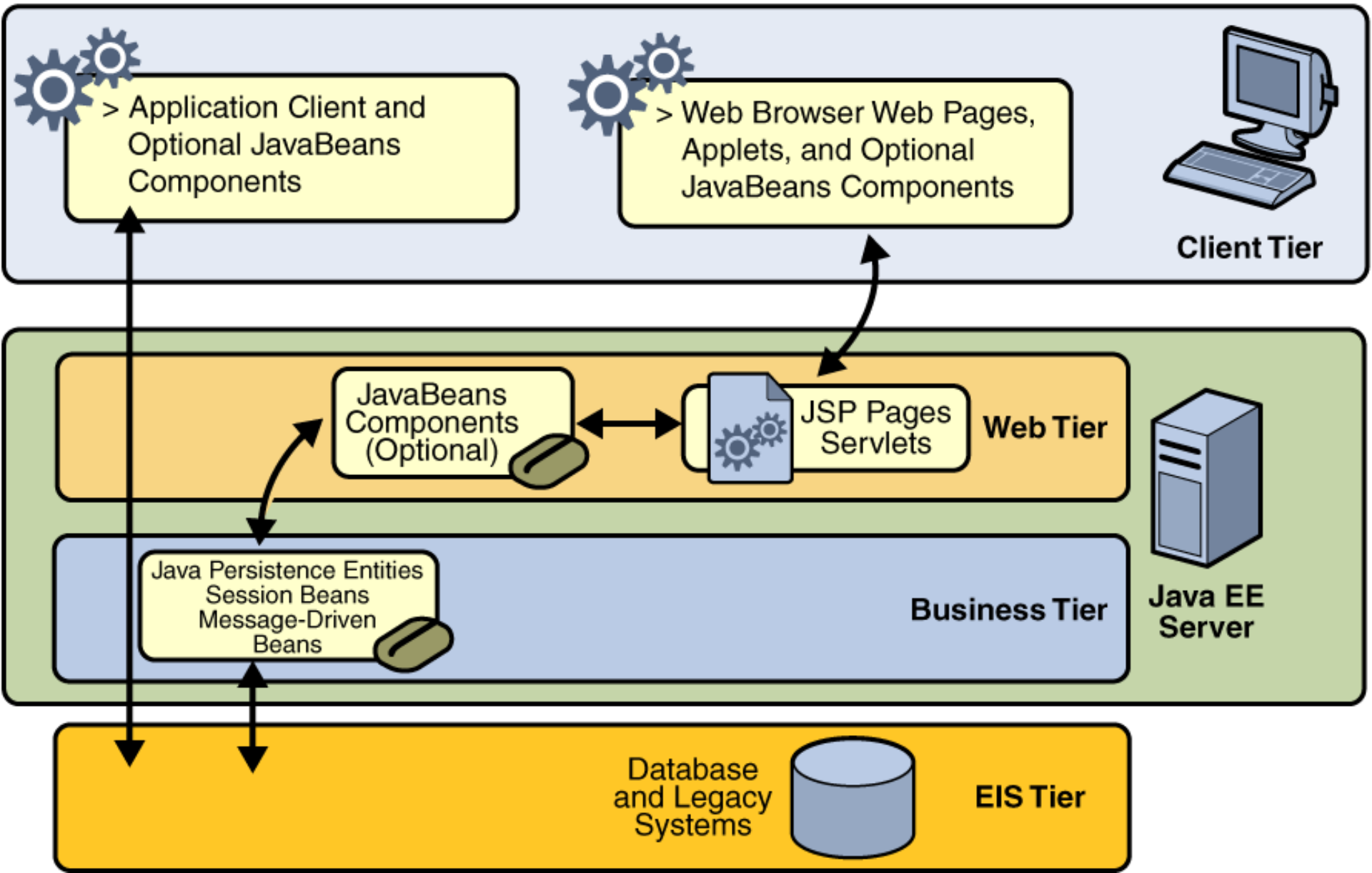
- **Sprzęt**
- **Oprogramowanie**
- **Bazy danych, bazy wiedzy**

System informatyczny
jest to zbiór powiązanych ze sobą elementów nieformalnych, formalnych i technicznych, którego funkcją jest przetwarzanie danych przy użyciu techniki komputerowej

Techniczny system informacyjny

- zorganizowany zespół środków technicznych (komputerów, oprogramowania, urządzeń teletransmisyjnych itp.)
- służący do gromadzenia, przetwarzania i przesyłania informacji

Warstwy aplikacji (Java EE)



Charakterystyka oprogramowania obiektowego

1. Definicja systemu informatycznego
2. Model procesu wytwarzania oprogramowania - model cyklu życia oprogramowania

Model procesu wytwarzania oprogramowania - czyli model cyklu życia oprogramowania

Tworzenie technicznego systemu informacyjnego jest powiązane z:

- budową oprogramowania: co i jak wykonać?
- zarządzaniem procesem tworzenia oprogramowania: kiedy wykonać?
- wdrażaniem oprogramowania

Modelowanie struktury i dynamiki systemu	**Implementacja systemu,**	**struktury i dynamiki generowanie kodu**
Perspektywa koncepcji *co należy wykonać?*	Perspektywa specyfikacji *jak należy używać?*	Perspektywa implementacji *jak należy wykonać?*
• model problemu np. przedsiębiorstwa • wymagania • analiza (model konceptualny) • testy modelu	• projektowanie (model projektowy: architektura sprzętu i oprogramowania; dostęp użytkownika; przechowywanie danych) • testy projektu	• programowanie (specyfikacja programu : deklaracje, definicje; dodatkowe struktury danych: struktury „pojemnikowe”, pliki, bazy danych) • testy oprogramowania • wdrażanie • testy wdrażania

Charakterystyka oprogramowania obiektowego

1. Definicja systemu informatycznego
2. Model procesu wytwarzania oprogramowania - model cyklu życia oprogramowania
3. Wymagania

Wymagania określają, co program ma robić

- Wymagania są najczęściej niekompletne
- Wymagania wprowadzają w błąd
- Wymagania nie są wyczerpująco określone
- Wymagania zawsze się zmieniają, gdyż:
  - Pogłębia się zrozumienie swoich potrzeb przez klienta
  - Pogłębia się zrozumienie dziedziny zastosowań przez programistę
  - Zmieniają się technologie

Wniosek

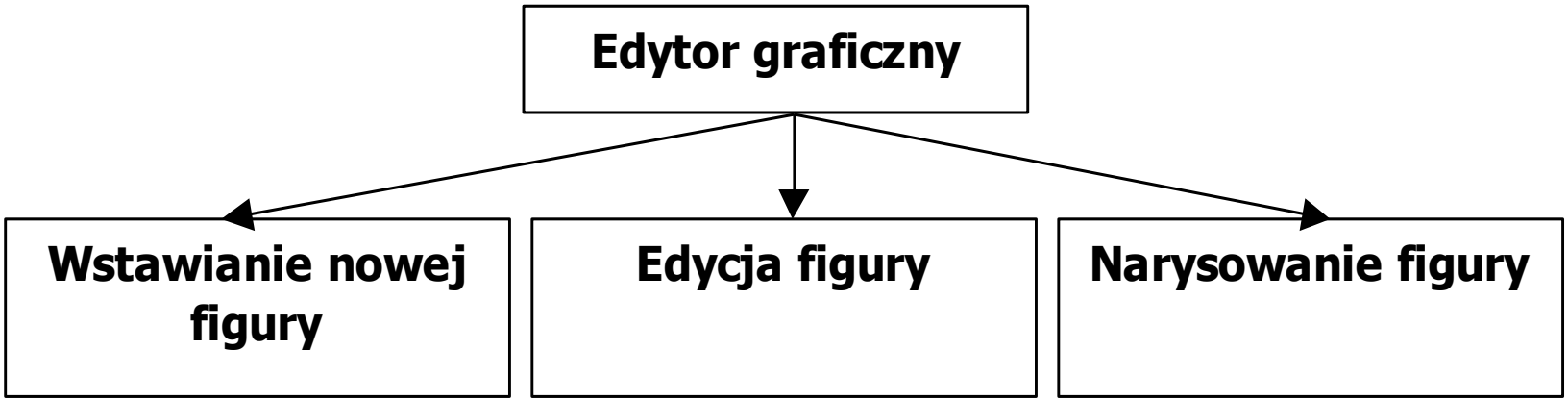
Skoro zmiany wymagań są nieuniknione, to należy zmniejszyć wrażliwość tworzonego oprogramowania na te zmiany.

Charakterystyka oprogramowania obiektowego

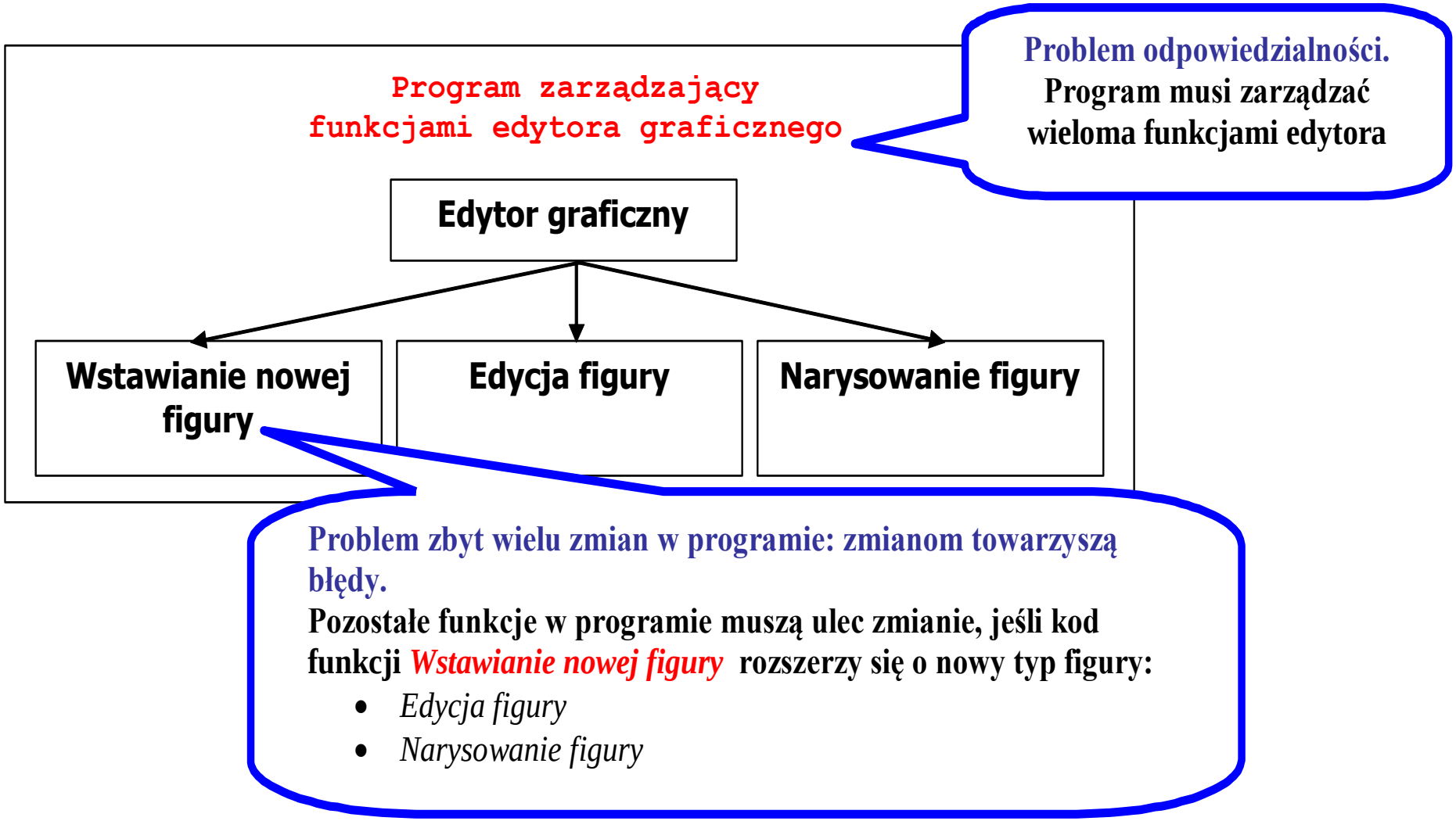
1. Definicja systemu informatycznego
2. Model procesu wytwarzania oprogramowania - model cyklu życia oprogramowania
3. Wymagania
4. Problemy z podejściem nieobiektowym

Podejście nieobiektowe przy tworzeniu oprogramowania – podczas analizy dekompozycja funkcjonalna

Przykład 1: Edycja figur



Konsekwencje podejścia nieobiektowego opartego na dekompozycji funkcjonalnej przy zmianach wymagań – (1)

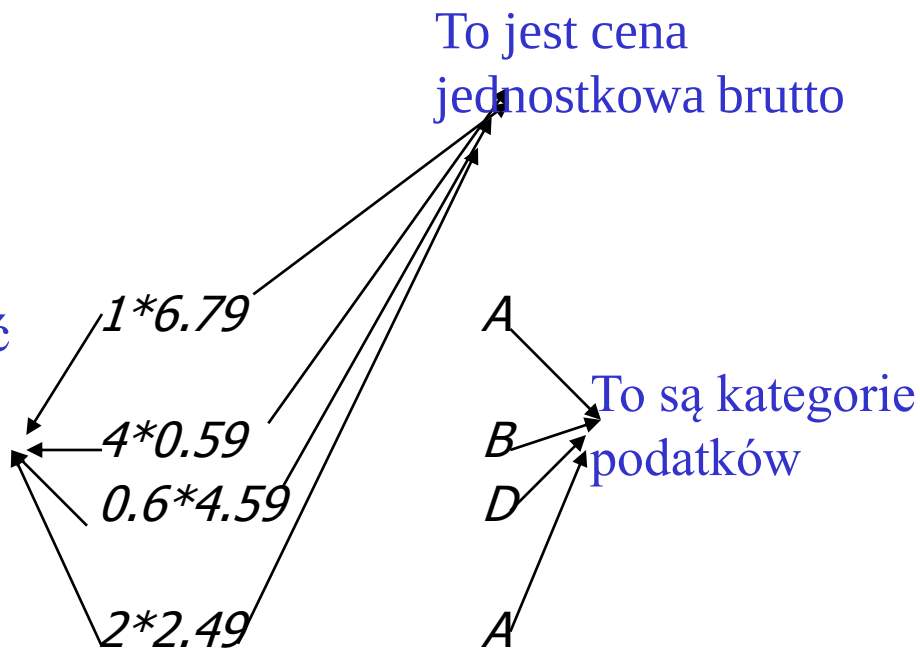


Przykład 2: Obliczanie wartości rachunku

Sklep Spożywczo – Przemysłowy „ABC”
Jan Kowalski
ul. Leśna 1, xx-xxx Jakieś miasto
NIP xxx-xxx-xx-xx
Dn. 07r-09-24 nr wydr.8212

PARAGON FISKALNY

xxxxxxxxxxxx
Nazwa produktu1 xxxxx
xxxxxxxxxxxx
Nazwa produktu2 xxxx
Nazwa produktu3 xxx
xxxxxxxxxxxx
Nazwa produktu1 xxxxx



Sp.op.A	11.77	PTU A = 22.00%	2.12
Sp.op.B	2.36	PTU B = 7.00%	0.15
Sp.op.D	2.75	PTU D = 3.00%	0.08
		Razem PTU	2.35

To jest cena brutto towarów z danej kategorii podatku

To są kwoty tara wynikające z istniejących kategorii podatków

RAZEM ZŁ 16.88

Konsekwencje podejścia nieobiektowego opartego na dekompozycji funkcjonalnej przy zmianach wymagań (2)

Centralizacja odpowiedzialności

Rozwiązanie 1. Sporządzanie rachunków – podejście nieobiektowe

1. Rachunek:
   - oblicza cenę brutto zakupionego produktu (wg atrybutów: cena netto, podatek, promocja)
   - zna liczbę zakupionych produktów
2. Rachunek podaje swoją wartość: sumuje wartość zakupu każdego produktu obliczając cenę brutto na podstawie atrybutów danego produktu i mnożąc cenę brutto przez liczbę zakupionego produktu.

Wnioski:

- Duża odpowiedzialność rachunku w obliczaniu wartości rachunku oraz duża wrażliwość na zmiany algorytmów obliczania wartości zakupu produktu
- Zmiany działania rachunku w odniesieniu jednego produktu mogą wpływać w przypadkowy sposób na obsługę innych produktów i powodować błędy.

Charakterystyka oprogramowania obiektowego

1. Definicja systemu informatycznego
2. Model procesu wytwarzania oprogramowania - model cyklu życia oprogramowania
3. Wymagania
4. Problemy z podejściem nieobiektowym
5. Podejście obiektowe – rozwiązanie pewnych problemów podejścia nieobiektowego

Konsekwencje podejścia obiektowego przy zmianach wymagań - decentralizacja odpowiedzialności

Rozwiązanie 2. Sporządzanie rachunków –podejście obiektowe

1. **Rachunek:**
   - posiada kolekcję zakupów
   - oblicza swoją wartość: sumuje kolejno wartości zakupionych produktów otrzymane od zakupów umieszczonych w kolekcji
2. **Zakup:**
   - posiada zakupiony produkt oraz ilość zakupionego towaru
   - potrafi obliczyć swoją wartość: pobiera od każdego produktu cenę brutto i mnoży otrzymaną wartość przez liczbę tego produktu
3. **Produkt:**
   - każdy produkt posiada atrybuty opisujące: nazwę, ceną netto oraz dodatkowe opcjonalnie atrybuty: podatek, promocję
   - każdy produkt potrafi sam obliczyć tę cenę brutto uwzględniając indywidualnie cenę netto oraz dodatkowe atrybuty (podatek, promocja)

Wnioski:

- Mała odpowiedzialność rachunku w zakresie obliczania wartości rachunku, mała wrażliwość na zmiany algorytmów obliczania wartości zakupu produktu dotyczącego np.. zmiany ceny brutto. Czynności te realizują różne typy produktów.
- Zmiany w funkcjach produktów nie wywołują efektów ubocznych (błędów)

Obiektowość

Obiekt posiada zestaw własnych danych z operacjami (metodami) wspólnymi z innymi podobnymi obiektami – są to składowe klasy.

Obiekt jest odpowiedzialny za siebie.

Przykłady obiektów

Produkt:

- zna swoje atrybuty
- potrafi obliczyć swoją cenę brutto na podstawie ceny netto oraz dodatkowych atrybutów jak podatek oraz promocja.

Zakup:

- wie, jaki i ile produktu zakupiono
- potrafi obliczyć swoją wartość na podstawie otrzymanej ceny brutto i liczby zakupionego produktu

Rachunek:

- wie, ilu dokonano zakupów
- sumując wartości wszystkich zakupów uzyskuje swoją wartość.

Perspektywy rozumienia obiektów

- **Perspektywa koncepcji** (modelu konceptualnego)

  - obiekt jest zbiorem różnego rodzaju odpowiedzialności

- **Perspektywa specyfikacji** (modelu projektowego)

  - obiekt jest zbiorem metod (zachowań), które mogą być wywoływane przez metody tego obiektu lub innych obiektów

- **Perspektywa implementacji** (kodu źródłowego)

  - obiekt składa się z kodu metod i danych oraz interakcji między nimi

Sposób identyfikacji obiektów oraz ich odpowiedzialności w prostych projektach

- Wyszukanie bytów w dziedzinie problemu
  - Rzeczowniki w opisie problemu pozwalają określić obiekty np. *Rachunek*, *Zakup*, *Produkt*
  - Czasowniki zawarte w opisie problemu są metodami tych obiektów:

    np.

    Rachunek ma metodę: *obliczwartośćrachunku,*

    Zakup ma metodę: *obliczwartośćzakupu*,

    Produkt ma metodę: *obliczcenębrutto*
- Rodzaj odpowiedzialności obiektów można określić wg akcji wykonywanych przez te obiekty

Sposób identyfikacji klas w prostych projektach

- Klasa zawiera metody używane przez wiele obiektów tej klasy np. klasa *TProdukt* zawiera metody używane przez wiele obiektów *Produkt* o różnych wartościach tych samych typów atrybutów
- Klasa zawiera opis danych (atrybutów) używanych przez obiekt
- Klasa określa sposób dostępu do danych i metod

Wniosek:

Obiekt jest instancją klasy

Program obiektowy – obliczanie wartości rachunku

1. Start programu
2. Utworzenie instancji klasy *TRachunek*
3. Utworzenie w instancji klasy *TRachunek* instancji klasy *TKolekcja* zawierającej instancje klasy *TZakup*; każda instancja klasy *TZakup* zawiera instancję klasy *TProdukt* oraz liczbę zakupionego produktu
4. Wywołanie metody ***obliczwartośćrachunku*** instancji klasy *TRachunek*, w której:

   4.1. instancja klasy *TRachunek* pobiera instancję klasy *TZakup* i wywołuje jej metodę *obliczwartośćzakupu*

      4.1.1. w metodzie ***obliczwartośćzakupu*** instancja klasy *TZakup* wywołuje metodę ***obliczcenębrutto*** instancji klasy *TProdukt*

         4.1.1.1.w metodzie ***obliczcenębrutto*** instancja klasy *TProdukt* oblicza cenę brutto na podstawie własnych atrybutów i zwraca wartość do metody ***obliczwartośćzakupu*** instancji klasy *TZakup*

      4.1.2. Metoda ***obliczwartośćzakupu*** oblicza wartość zakupu (mnoży otrzymaną ceną brutto od instancji TProdukt razy liczbę produktów) i zwraca wartość metody ***obliczwartośćrachunku***

   4.2. Metoda ***obliczwartośćrachunku*** dodaje otrzymaną wartość do wartości rachunku

   4.3. Metoda ***obliczwartośćrachunku*** powtarza krok 4.1 tak długo, aż wyczerpie instancje klasy *TZakup* w instancji kolekcji. Po jej wyczerpaniu przechodzi do p.4.4

   4.4. Metoda ***obliczwartośćrachunku,*** jeśli zostaną wyczerpane te instancje, zwraca wartość bieżącego rachunku

5. Koniec programu

Schemat blokowy – algorytm obliczania wartości rachunku

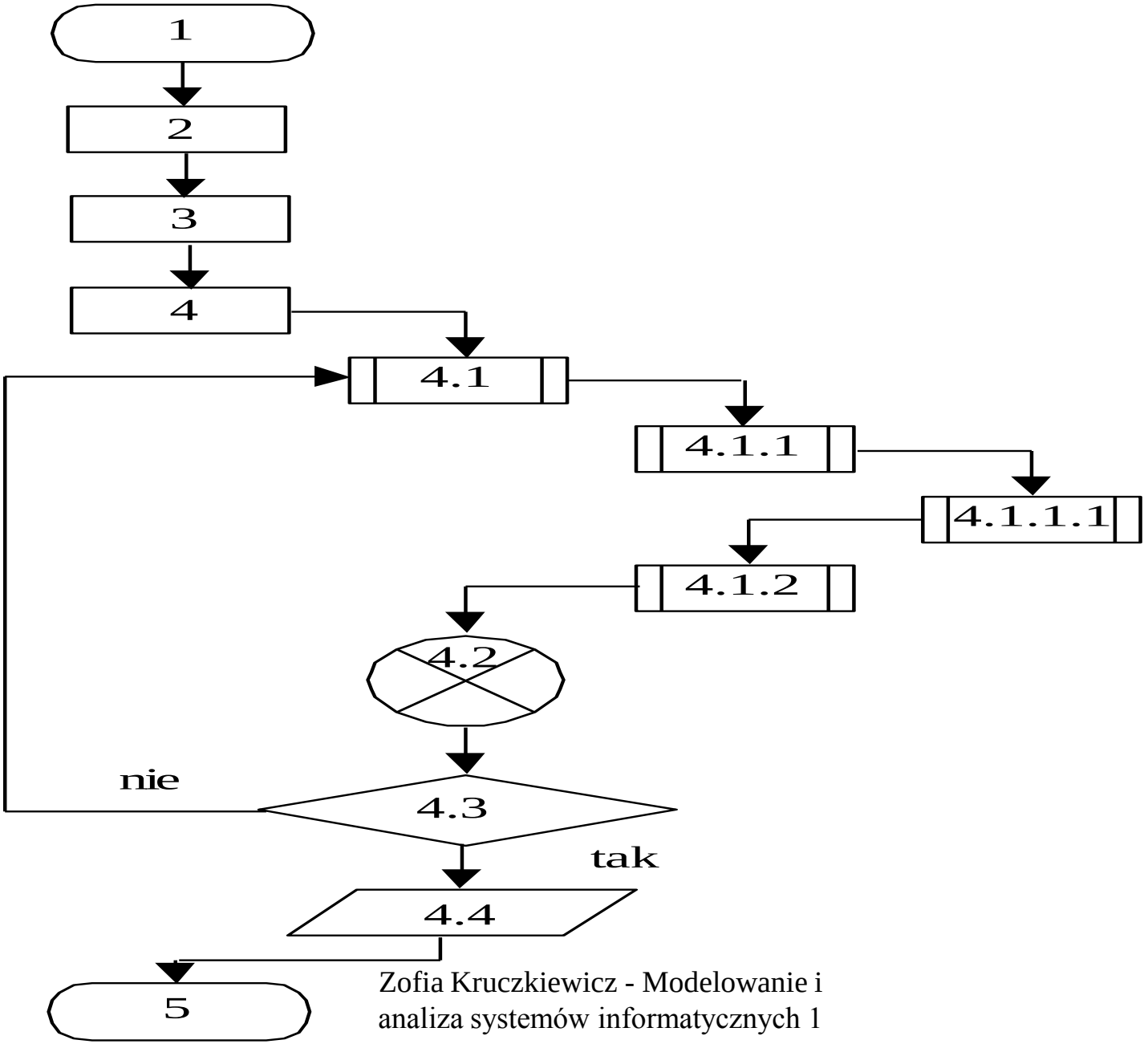
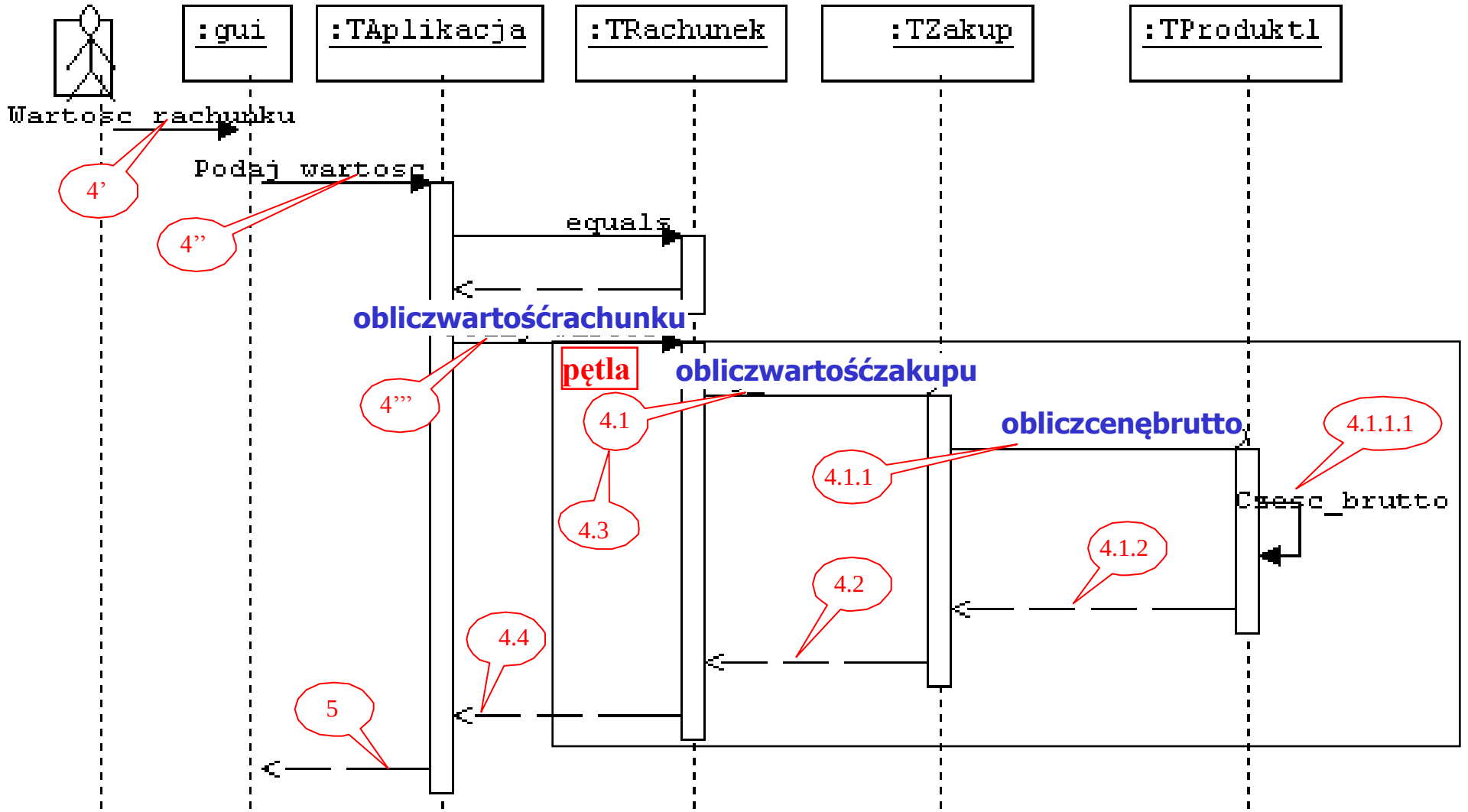


Diagram sekwencji UML– obiektowy sposób przedstawienia scenariusza obliczania rachunku



Paradygmaty obiektowości

- Abstrakcja klas
- Dziedziczenie
- Hermetyzacja
- Polimorfizm
- Agregacja

Paradygmaty obiektowości

- **Abstrakcja klas**

W programie ***Obliczanie wartości rachunku*** wystąpiła klasa typu *TKolekcja* zawierająca instancje klasy typ *TZakup*. W przypadku wprowadzenia nowej klasy typu *TZakup1* np. z powodu zmiany strategii obliczania wartości zakupu towaru mogłaby ulec zmianie również klasa *TKolekcja*. Aby uniezależnić się od tego zjawiska umieszcza się w kolekcji taki typ klasy, który pozwala na umieszczanie instancji klas wyspecjalizowanych w odniesieniu do tej klasy np. klasy *TZakup* i *TZakup1*. Sama ***klasa uogólniona*** np. ***TObiekt*** nie może posiadać instancji – służy jedynie do uogólnienia wybranych właściwości wielu klas obsługiwanych przez jedną klasę *TKolekcja*.

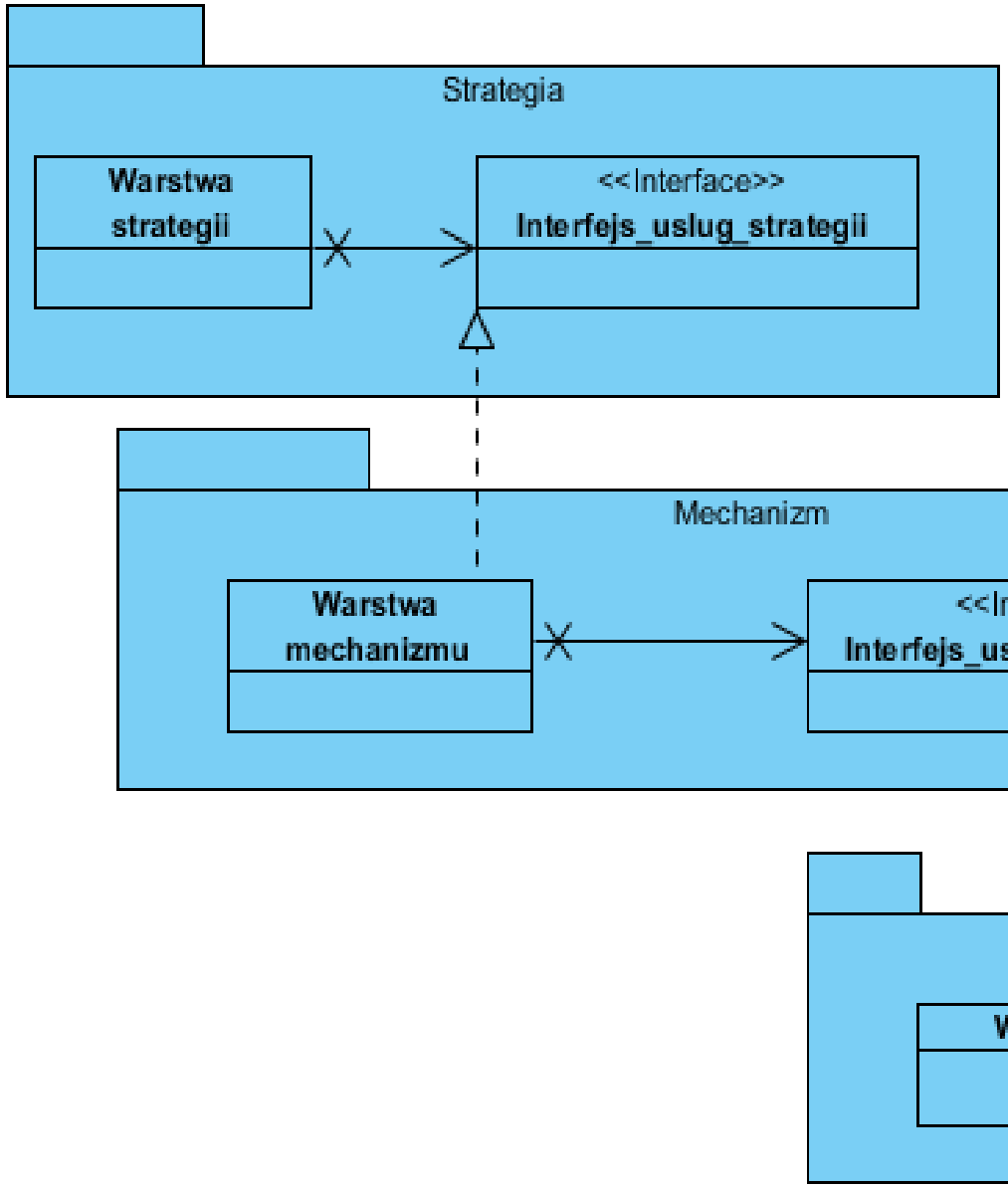
Taka uogólniona klasa nazywa się klasą abstrakcyjną.

Klasa abstrakcyjna może wystąpić w perspektywach koncepcji, specyfikacji lub implementacji.

Abstrakcja klas

- Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji (np. klasy modułu strategii używają klas abstrakcyjnych/interfejsów, które są implementowane przez klasy modułu mechanizmu, które z kolei używają własnych klas abstrakcyjnych /interfejsów zaimplementowanych przez klasy modułu narzędzi. Zmiana implementacji klas warstwy mechanizmu nie wpływa na sposób ich użycia przez klasy z warstwy strategii oraz zmiana implementacji klas w warstwie narzędzi nie wpływa na sposób użycia tych klas w warstwie mechanizmu oraz nie wpływa na sposób użycia klas warstwy mechanizmu przez klasy warstwy strategii)
- Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji .
- Wsparcie zasady: otwarte dla rozbudowy, ale zamknięte dla modyfikacji

Abstrakcja klas



- Warstwy niższe zależą od wyższych
- Eliminacja zależności przechodniej: warstwa strategii – warstwa narzędzi
- Eliminacja zależności bezpośredniej: warstwa strategii – warstwa mechanizmu

Paradygmaty obiektowości

- Klasy abstrakcyjne
- **Dziedziczenie**

Klasy ***TZakup*** oraz ***TZakup1***, które są klasami wyspecjalizowanymi klasy abstrakcyjnej np. ***TObiekt***, powstają dzięki **dziedziczeniu** po klasie abstrakcyjnej i są nazywane **klasami pochodnymi.**

Można utworzyć wiele klas, które dziedziczą od klasy ***TObiekt*** **wspólne dane i metody** związane z przechowywaniem w kolekcji i mogą różnie implementować obliczanie wartości zakupionego produktu **dodając nowe atrybuty i nowe metody** np.:

- w klasie pochodnej ***TZakup*** obliczanie wartości zakupu polega na pomnożeniu ilości produktu przez cenę brutto otrzymaną od swojego produktu
- w klasie pochodnej ***TZakup1*** obliczanie wartości zakupu polega na obliczeniu tej wartości po dobraniu korekty ceny zakupu w zależności od ilości zakupionego produktu

Dziedziczenie

- specjalizacja klas
- służy do implementowania podtypów(część elementów podtypu zawiera się w klasie bazowej oraz część stanowi rozszerzenie)
- wspiera wielożywalność kodu

Paradygmaty obiektowości

- Klasy abstrakcyjne
- Dziedziczenie
- **Hermetyzacja**

W systemach obiektowych rozróżnia się następujące rodzaje dostępu:

- **publiczny** – dla obiektów dowolnej klasy
- **chroniony** – tylko dla obiektów danej klasy i obiektów pochodnych
- **prywatny** – tylko dla obiektów danej klasy.

Dostęp **prywatny** do składowych obiektów typu *TZakup* oznacza, że obiekt typu *TRachunek* nie zna jego atrybutów typu *TProdukt* oraz ilości produktu. To samo dotyczy nowych atrybutów w klasie *TZakup1*. Oznacza to brak wrażliwości kodu klasy *TRachunek* na zmiany w kodzie prywatnych składowych tych klas.

Natomiast klasa *TRachunek* jest zainteresowana wartością zakupu obliczaną przez obiekty typu *TZakup* i *TZakup1* za pomocą metody *obliczwartośćzakupu*, dlatego dostęp do tej metody musi być **publiczny**. Ewentualne błędy w klasie *TRachunek* nie powodują błędów w klasach *TZakup* i *TZakup1* oraz błędy w tych klasach nie powodują błędów w kodzie klasy *TRachunek*. Wynika to z separacji kodu wywoływanych metod publicznych.

Hermetyzacja

- określenie praw dostępu do składowych klasy
- ukrywanie logiki przetwarzania własnych atrybutów
- wsparcie dla uogólniania i specjalizacji

Paradygmaty obiektowości

- Klasy abstrakcyjne
- Dziedziczenie
- Hermetyzacja
- **Polimorfizm**

Obiekt typu *TKolekcja* w obiekcie *TRachunek* może przechowywać obiekty różnych typów klas pochodnych klasy *TObiekt* np. obiekty typu *TZakup* i *TZakup1*. Obiekty tych klas mogą mieć różny algorytm obliczania wartości zakupionego towaru w metodzie ***obliczwartośćzakupu***, wynikający z dziedziczenia. Dla obiektu typu *TRachunek* nie jest ważny ten sposób, tylko wynik obliczeń wartości zakupu po wywołaniu tej metody - dlatego nie musi on rozróżniać typów obiektów *TZakup* i *TZakup1*.
Taka właściwość klas pochodnych jest nazywana **polimorfizmem.**

Polimorfizm

- uogólnianie klas, czyli wynik wyróżnienia typu obiektowego, który całkowicie obejmuje inny typ obiektowy
- polimorfizm opiera się na przedefiniowaniu metod wirtualnych nadtypów w podtypach. Oznacza to, że musi istnieć możliwość zastępowania typów bazowych ich typami pochodnymi.
- wsparcie zasady: otwarte dla rozbudowy, ale zamknięte dla modyfikacji

Paradygmaty obiektowości

- Klasy abstrakcyjne
- Dziedziczenie
- Hermetyzacja
- Polimorfizm
- **Agregacja**

Obiekt klasy *TRachunek* składa się z instancji kolekcji obiektów klas pochodnych od klasy abstrakcyjnej typu *TObiekt*. Obiekty klasy *TZakup* lub *TZakup1* zawierają obiekty typu *TProdukt*.

Taka właściwość polegająca na tym, że obiekt zawiera inne obiekty, nazywa się **agregacją.** Jest to inny sposób budowy klasy niż budowa nowej klasy za pomocą dziedziczenia.

Agregacja

- składanie klas (kompozycja)
- agregacja silna (czasy życia obiektu agregowanego i agregującego są te same)
- agregacja słaba (czas życia obiektu agregowanego jest niezależny od czasu życia obiektu agregującego).