

Modelowanie Systemów Informatycznych (MSI)

dr inż. Mariusz Trzaska
mtrzaska@pjawstk.edu.pl

Wykład 3

Model obiektowy – cz. 1

<http://www.mtrzaska.com>

Zagadnienia

Klasa; notacja w UML

Dziedziczenie:

- jednoaspektowe
- wieloaspektowe
- wielokrotne
- dynamiczne

Klasa parametryzowana

Rozszerzenia i ograniczenia w podklasie

Wystąpienie klasy

Klasa abstrakcyjna a klasa konkretna

Metoda abstrakcyjna

Interfejs, zależność, realizacja

Ekstensja klasy

Własności klas: atrybuty, metody

Przesłanianie a przeciążanie

Typ; kontrola typów

Własność zamienialności



*Wykorzystano materiały z wykładu PRI
autorstwa dr inż. Ewy Stemposz
oraz prof. Kazimierza Subiety*

Klasa; notacja (1)

Klasa jest nazwanym opisem grupy obiektów, które współdzielą ten sam zbiór własności (inwariantów). Klasa **nie jest** zbiorem obiektów, lecz jest używana do opisywania (deklarowania) obiektów.

Cztery pola: nazwy, atrybutów, metod i użytkownika (np. w celu specyfikowania odpowiedzialności klasy). Możliwe są różne poziomy szczegółowości.

Okno

Okno
rozmiar czy widoczne

Okno
rozmiar czy widoczne
wyświetl() schowaj()

Okno
Rozmiar : Obszar czy widoczne : Boolean
wyświetl() schowaj()

Pole nazwy klasy:

stereotyp dostępność nazwa_ klasy lista_wart_etyk

Pole atrybutów:

stereotyp dostępność nazwa_atrybutu : typ = wart_początkowa lista_wart_etyk

Pole metod:

stereotyp dostępność nazwa_metody (lista_arg) : typ_wart_zwracanej
lista_wart_etykt

Klasa; notacja (2)

gdzie:

dostępność jest określana przez trzy symbole:

- + publiczna
- prywatna
- # chroniona

lista_arg: rodzaj nazwa_arg : typ = wart_początkowa

rodzaj: definiuje sposób, w jaki metoda korzysta z danego argumentu

in: metoda może czytać argument, ale nie może go zmieniać

out: może zmieniać, nie może czytać

inout: może czytać i zmieniać

Wszystkie elementy specyfikacji klasy za wyjątkiem nazwy klasy są opcjonalne.

Nazwa klasy to z reguły rzeczownik w liczbie pojedynczej. Klasa zazwyczaj nazywana jest tak, jak nazywany jest pojedynczy jej obiekt, np. klasa *Pracownik* – jeden obiekt opisuje jednego pracownika;

Przykłady klas

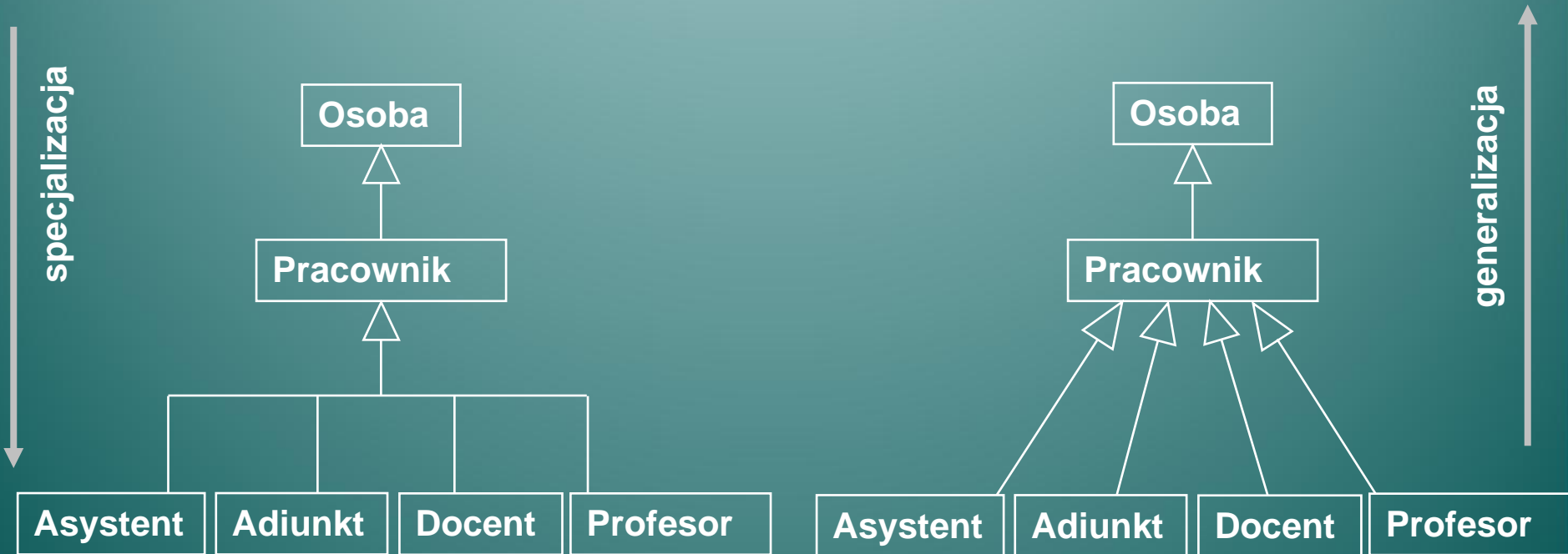
Okno
{abstrakcyjna, autor= "Kowalski" status="przetestowane"}
+rozmiar: Obszar = (100,100) #czy_widoczne: Boolean = false +rozmiar_domyślny: Prostokąt #rozmiar_maksymalny: Prostokąt -xwskaźnik: XWindow*
+wyświetl() +schowaj() +utwórz() -dołączXWindow(xwin: XWindow*)

«trwała» Prostokąt
punkt1: Punkt punkt2: Punkt
«konstruktor» Prostokąt (p1: Punkt, p2: Punkt)
«zapytania» obszar () : Real aspekt() : Real . .
«aktualizacje» przesuń(delta: Punkt) przeskaluj(współczynnik: Real)

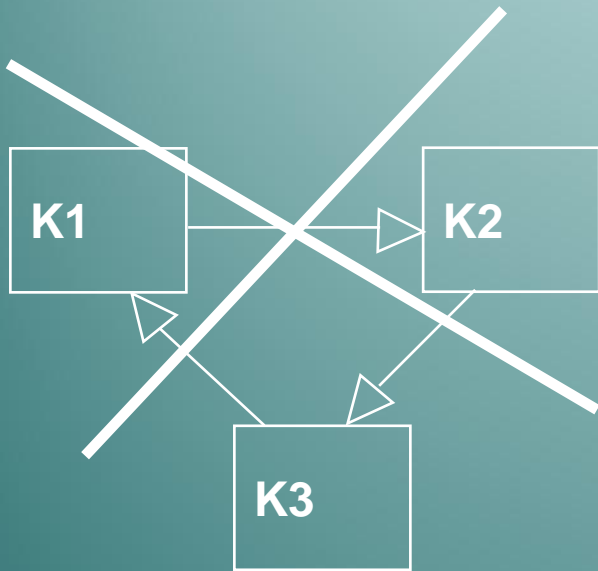
Stereotypy zostały tu wykorzystane do metaklasyfikacji metod.

Dziedziczenie (1)

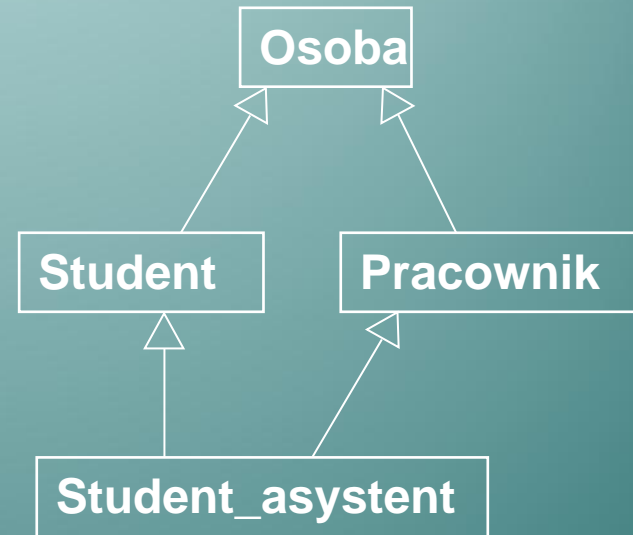
Dziedziczenie pozwala na tworzenie drzewa klas lub innych struktur bez pętli.



Dziedziczenie (2)

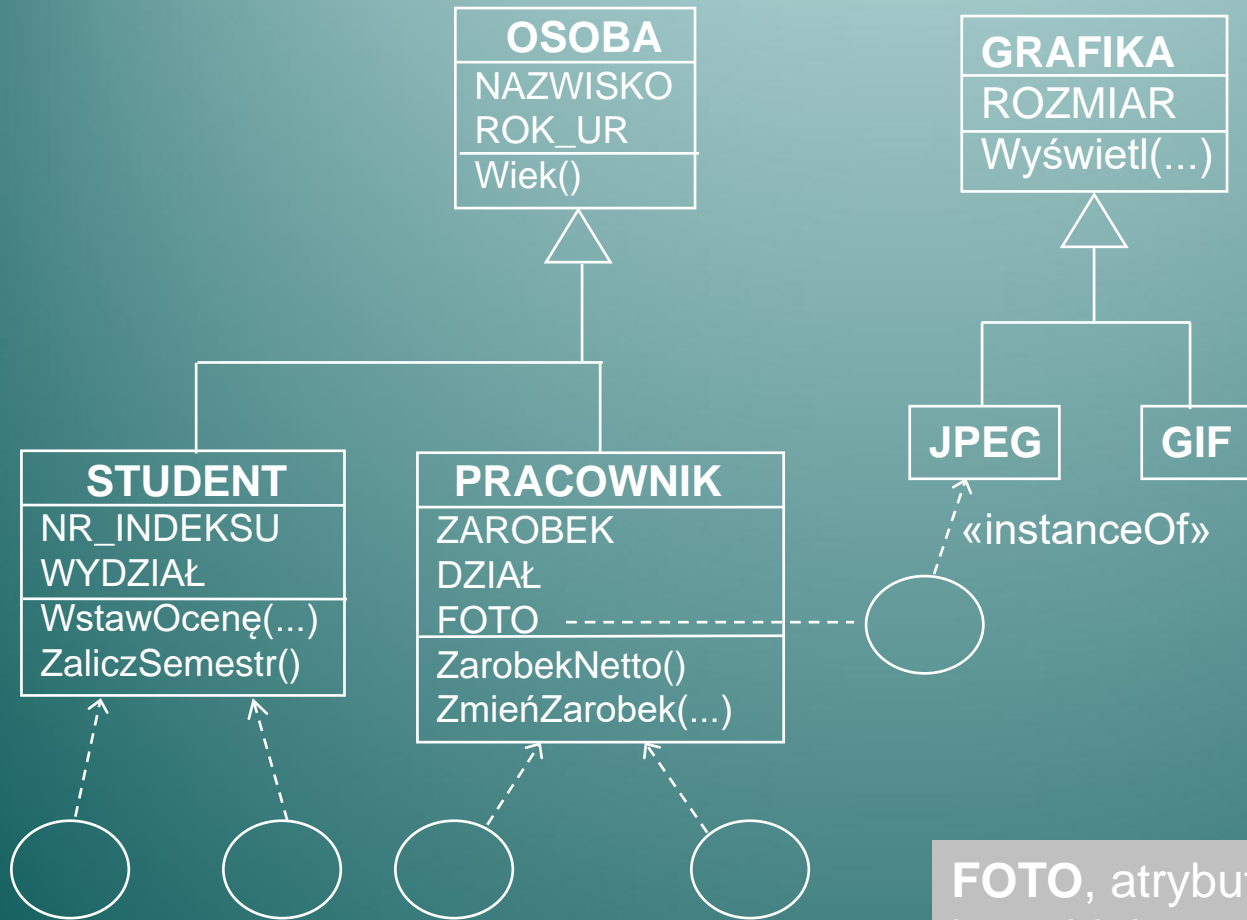


**Struktura typu pętla
jest zabroniona**



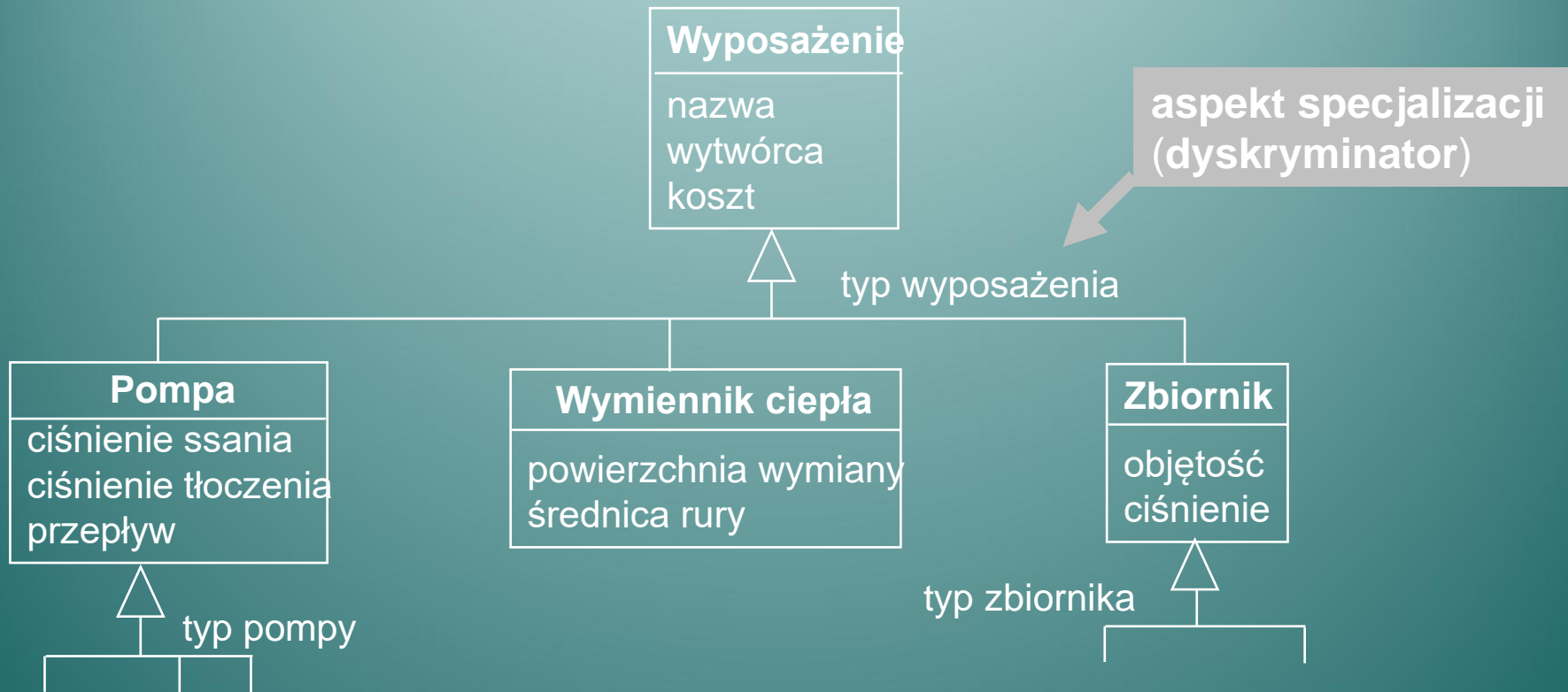
**Struktura typu krata
jest dopuszczalna**

Dziedziczenie (3)



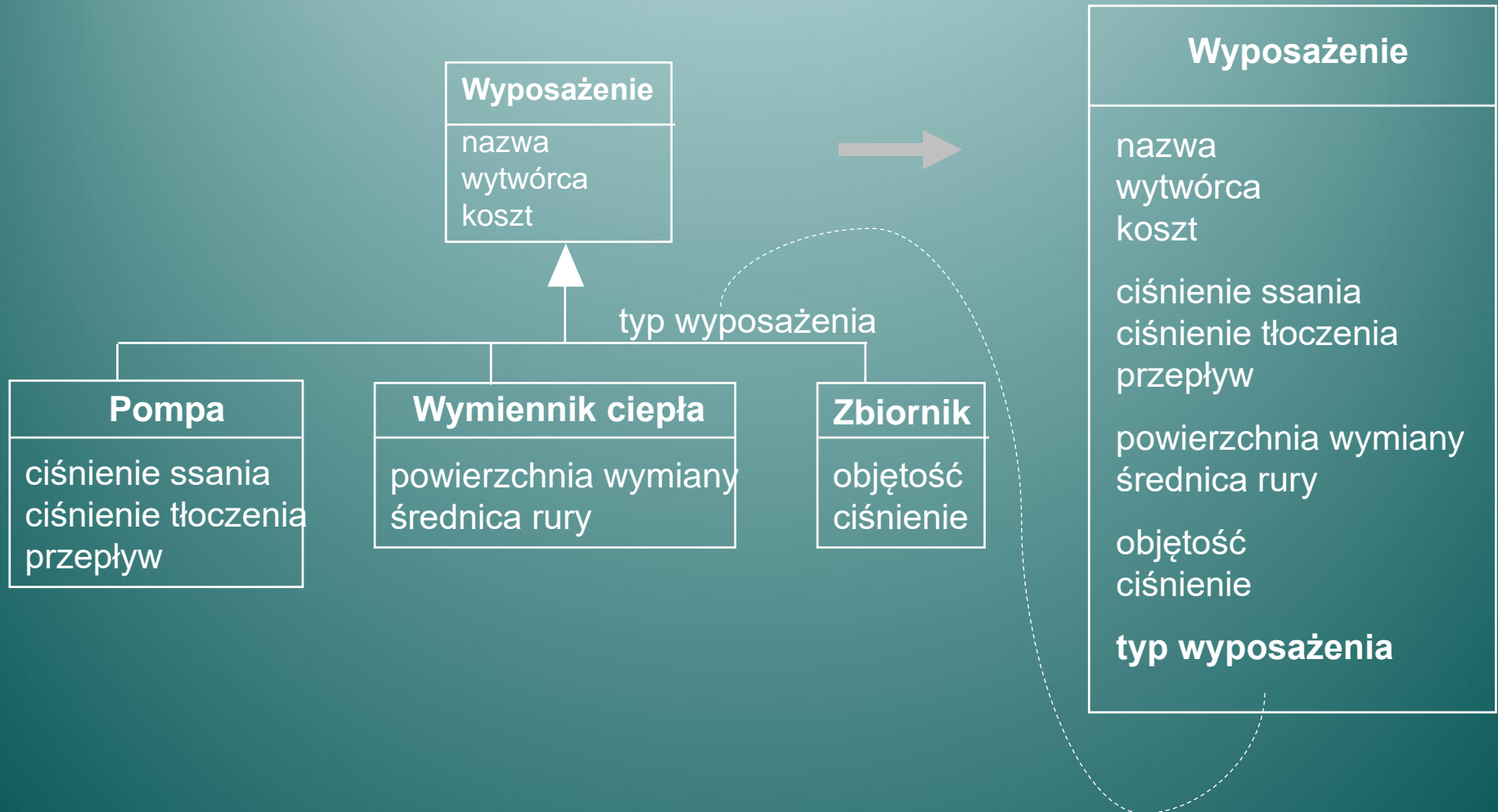
FOTO, atrybut klasy **PRACOWNIK**,
jest obiektem, członkiem klasy **JPEG**.

Dziedziczenie (4)

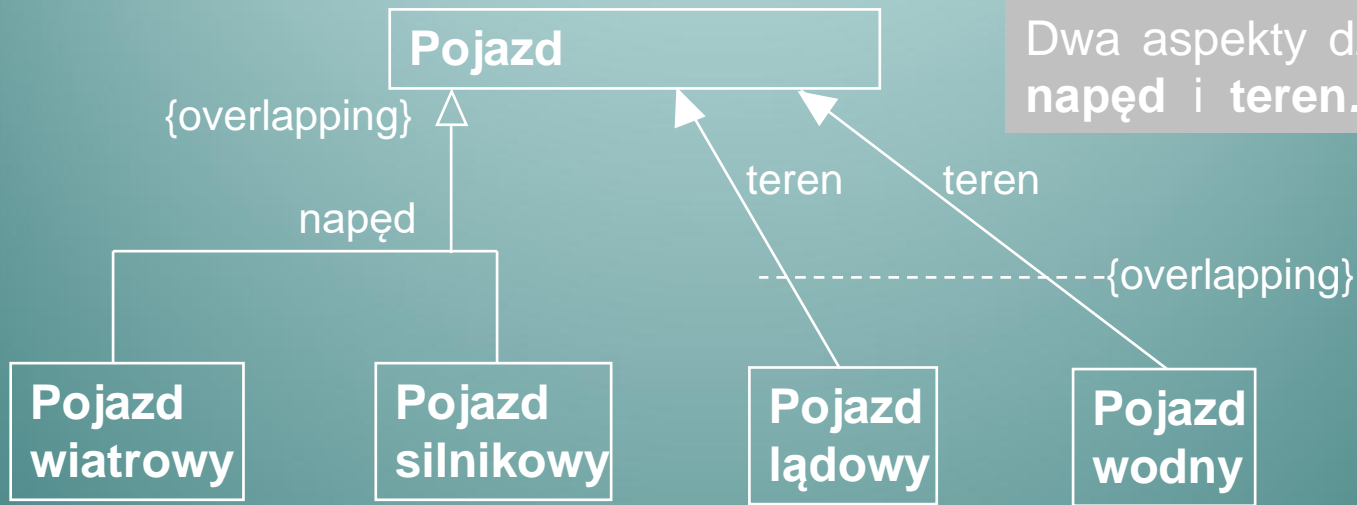


Dziedziczenie jednoaspektowe – aspekt specjalizacji (nazywany czasami dyskryminatorem) jest atrybutem opcjonalnym.

Dziedziczenie (5)



Dziedziczenie wieloaspektowe



Dwa aspekty dziedziczenia:
napęd i teren.

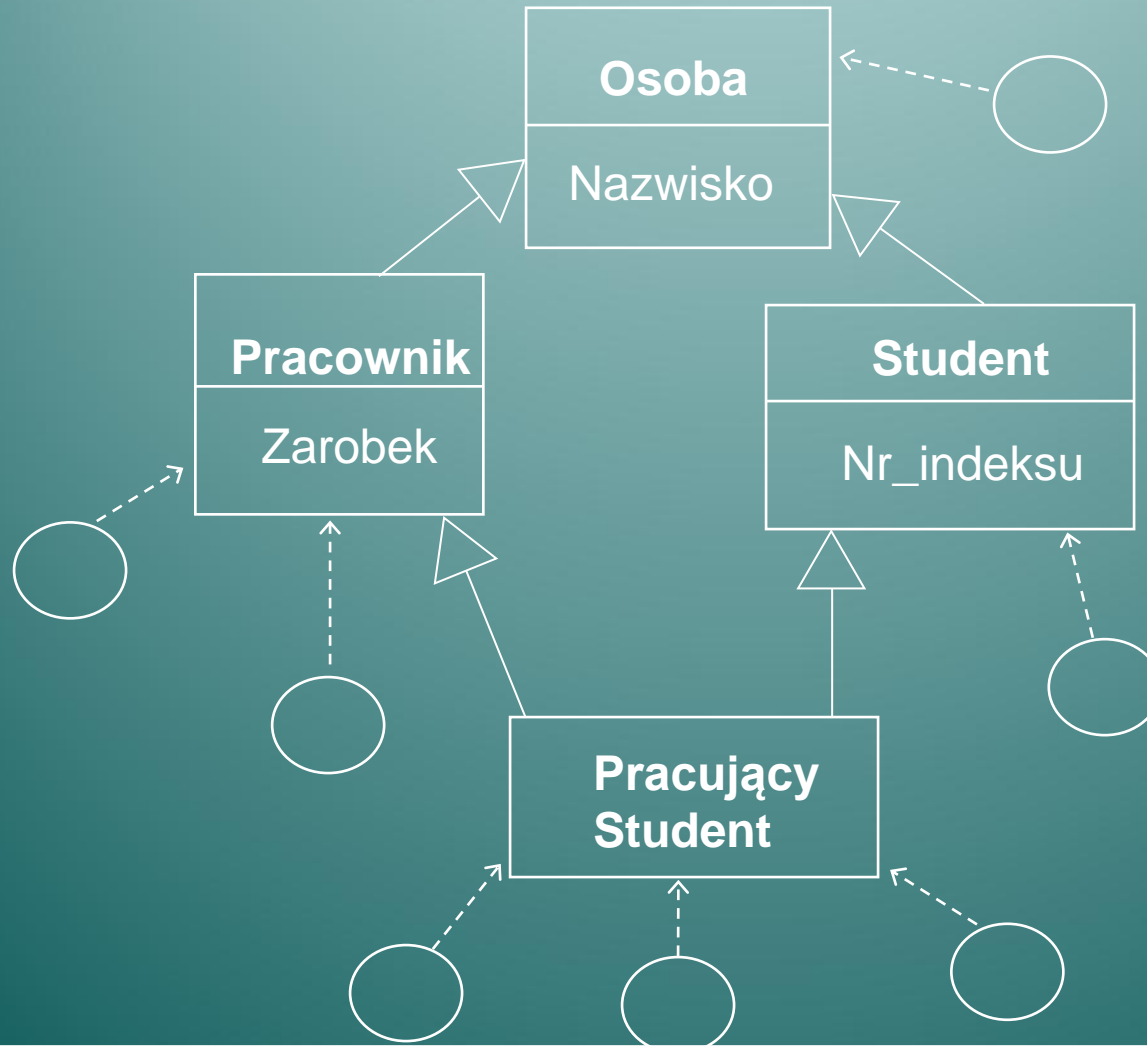
Dla dziedziczenia wieloaspektowego aspekty dziedziczenia nie mogą być opuszczone.



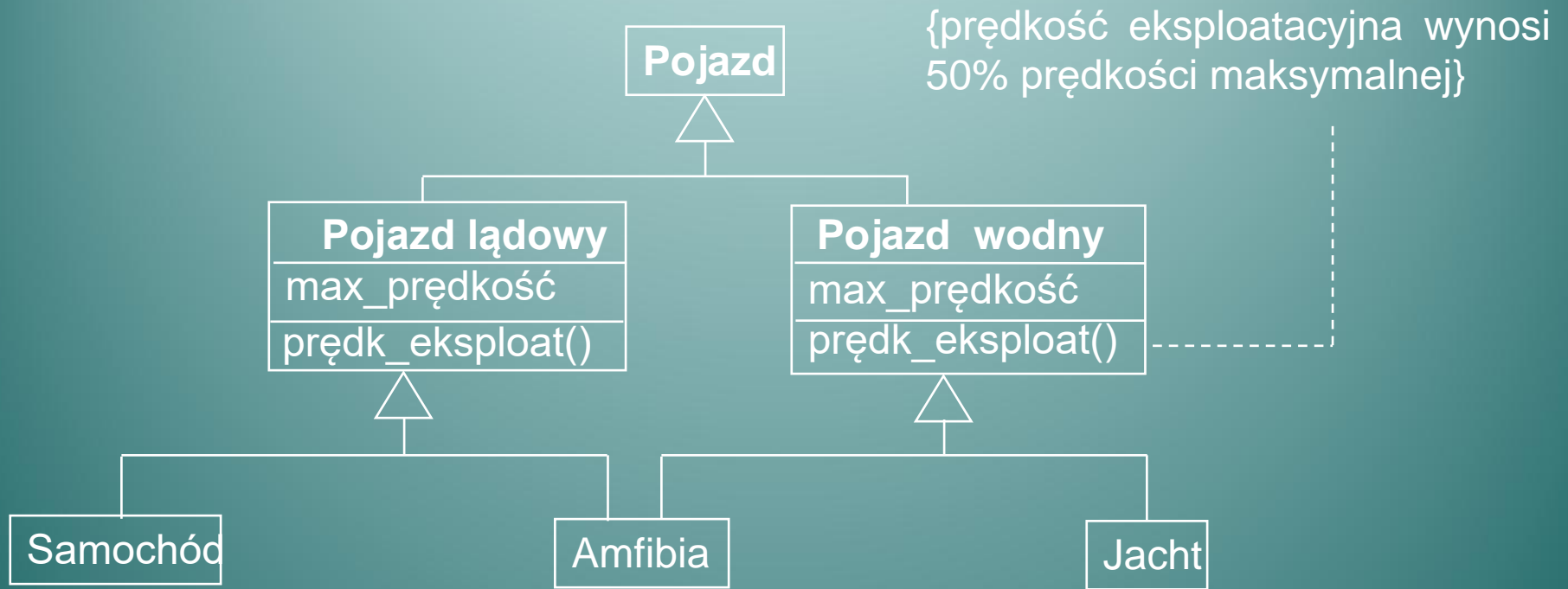
- disjont (domyślne): podział rozłączny
- overlapping: podział nierozłączny; przecięcie zbiorów obiektów klas, np. *Pojazd lądowy* i *Pojazd wodny*, nie jest zbiorem pustym;
- complete (domyślne): wszystkie planowane podklasy zostały zdefiniowane
- incomplete: nie wszystkie planowane podklasy zostały już zdefiniowane,
- ... (ellipsis): niektóre ze zdefiniowanych klas, np. te nieistotne dla aktualnie rozważanego problemu, zostały pominięte

Dziedziczenie wielokrotne

Dziedziczenie wielokrotne (wielodziedziczenie) ma miejsce, gdy klasa dziedziczy inwarianty z więcej niż jednej klasy (bezpośrednio).



Problemy dziedziczenia wielokrotnego



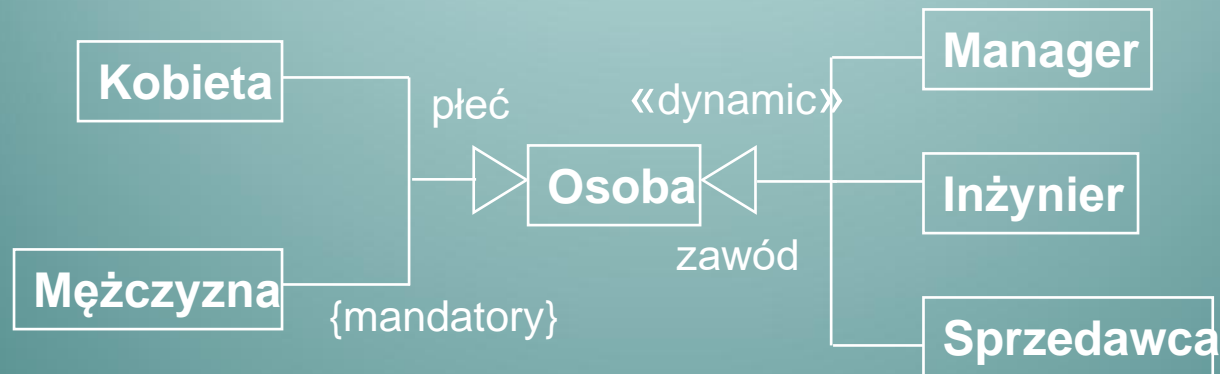
Konflikt nazw:

Który z atrybutów *max_prędkość* ma odziedziczyć amfibia?

Czy znaczenie metody *prędk_eksploat()* zależy od ścieżki dziedziczenia?

(np. C++: należy określić ścieżkę dziedziczenia)

Dziedziczenie dynamiczne



Dziedziczenie zostało tu opatrzone stereotypem «dynamic».

Osoba może zmieniać zawód, co może być modelowane poprzez tzw. dziedziczenie dynamiczne. Przydatne dla modelowania koncepcyjnego, trudne w implementacji.

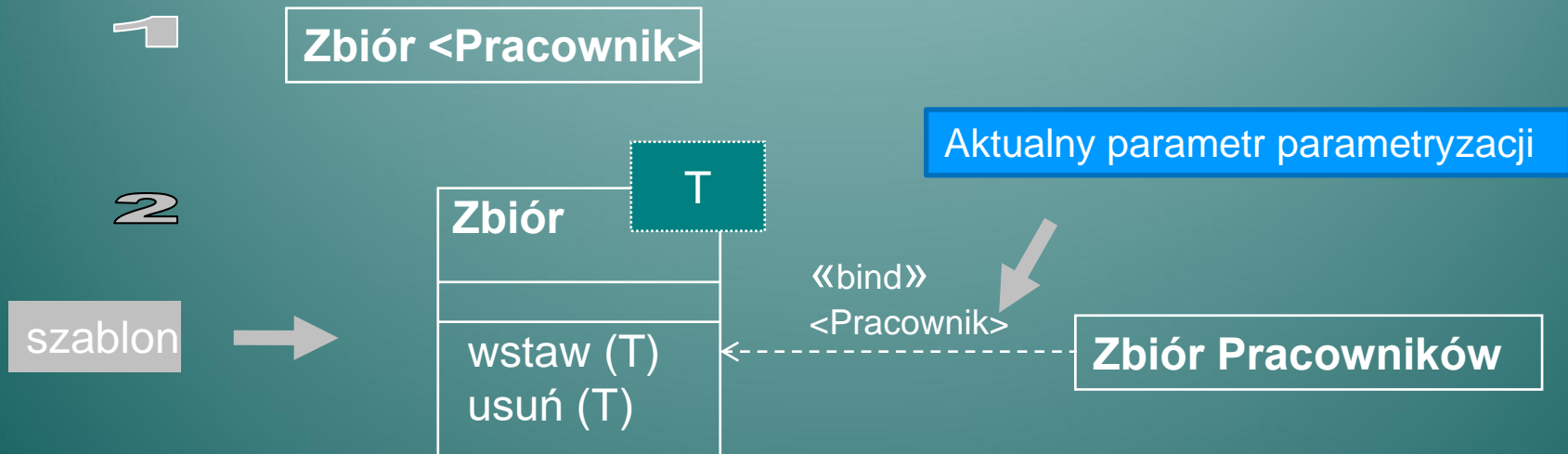
mandatory – obowiązujący, obowiązkowy

Klasa parametryzowana

Klasy parametryzowane są użyteczne z dwóch zasadniczych powodów: podnoszą poziom abstrakcji i wpływają na zmniejszenie długości kodu źródłowego programu.

Klasy parametryzowane posiadają duży potencjał ponownego użycia.

Klasa parametryzowana może być wstawiana do diagramów UML na dwa sposoby



Podstawowe zastosowanie klas parametryzowanych polega na wykorzystaniu ich do definiowania zbiorów (szerzej – kolekcji). Każdy obiekt klasy **Zbiór <Pracownik>**, czy analogicznie **Zbiór Pracowników**, jest zbiorem.

Rozszerzenia i ograniczenia w podklasie

- ✓ Podklasa nie może omijać lub zmieniać atrybutów nadklasy.
- ✓ Podklasa może zmienić ciało metody z nadklasy, ale bez zmiany jej specyfikacji.
- ✓ Podklasa może dowolnie dodawać nowe atrybuty i metody (rozszerzać zbiór własności nadklasy).
- ✓ Podklasa może ograniczać wartości atrybutów. Np. *Koło* jest podklasą klasy *Elipsa*, gdzie obie średnice elipsy są sobie równe. Ograniczenia mogą spowodować, że część metod przestanie być poprawna. Np. zmiana jednej ze średnic obiektu – dozwolona dla obiektu klasy *Elipsa* – jest niedopuszczalna w obiekcie podklasy *Koło*, gdyż muszą tam być zmieniane obie średnice jednocześnie.

Czy *Koło* powinno być podklasą klasy *Elipsa* czy też powinno być odwrotnie?

Wystąpienie klasy

Pojęcie **wystąpienie klasy** (instancja klasy) oznacza obiekt, który jest “podłączony” do danej klasy, jest jej członkiem.

Wystąpienia mogą być: **bezpośrednie** i **pośrednie**.

Obiekt jest wystąpieniem bezpośrednim swojej klasy i wystąpieniem pośrednim wszystkich jej nadklas.

W zależności od poziomu szczegółowości możliwe są następujące oznaczenia obiektu:

<u>nazwa obiektu</u> : nazwa klasy
nazwa_atrybutu = wart_atrybutu
...

: <u>nazwa klasy</u>
nazwa_atrybutu = wart_atrybutu
...

<u>nazwa obiektu</u> : nazwa klasy

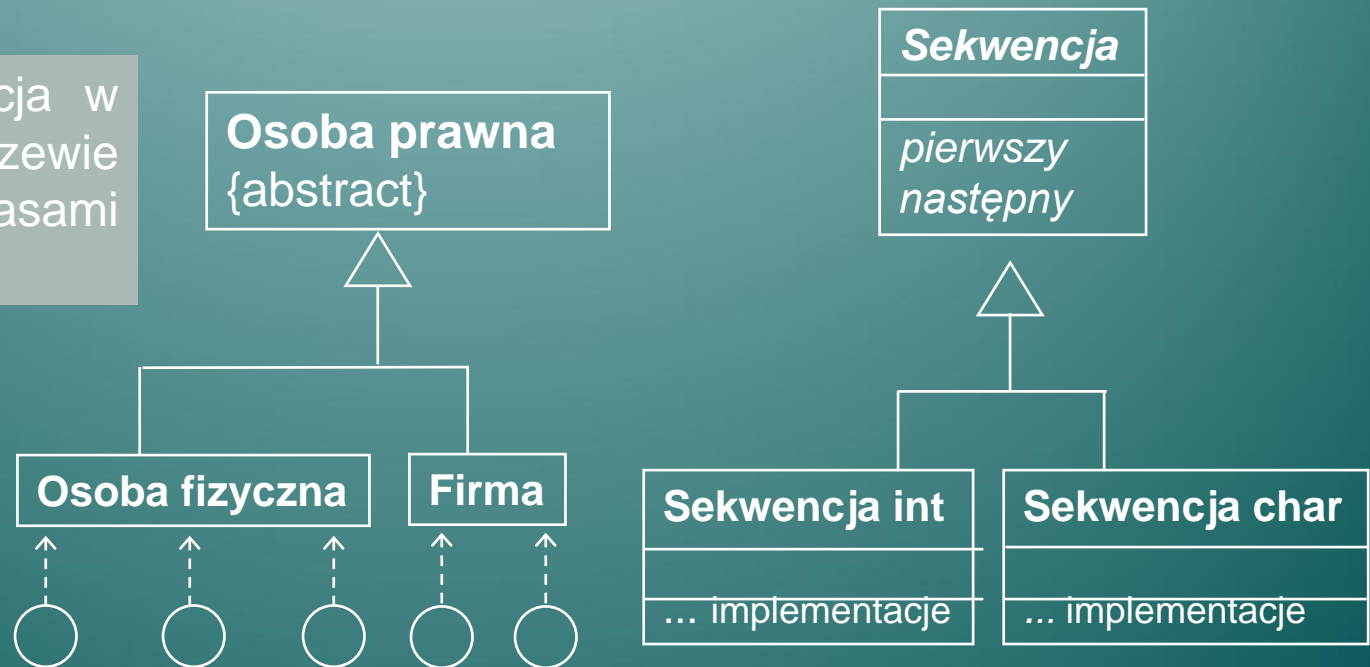
: <u>nazwa klasy</u>

Klasa abstrakcyjna a konkretna (1)

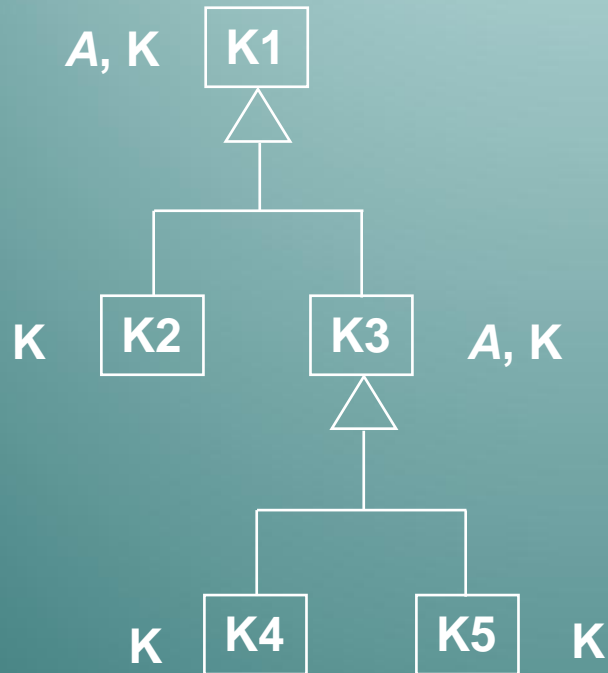
Klasa abstrakcyjna nie ma (nie może mieć) bezpośrednich wystąpień i służy wyłącznie jako nadklasa dla innych klas. Stanowi jakby wspólną część definicji grupy klas o podobnej semantyce. UML pozwala na oznaczenie bytu abstrakcyjnego za pomocą wartości etykietowanej **{abstract = TRUE}** (TRUE można opuścić) lub napisanie nazwy bytu abstrakcyjnego *italikami* (nazwy klasy czy metody abstrakcyjnej).

Klasa konkretna może mieć (ma prawo mieć) wystąpienia bezpośrednie.

Klasyczna klasyfikacja w biologii: liście w drzewie klas muszą być klasami konkretnymi.



Klasa abstrakcyjna a konkretna (2)



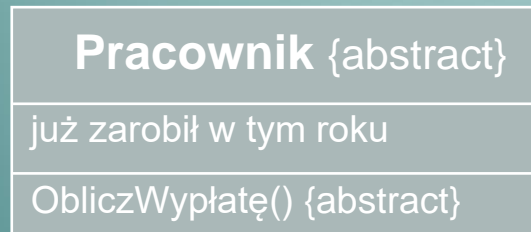
A – klasa abstrakcyjna

K – klasa konkretna

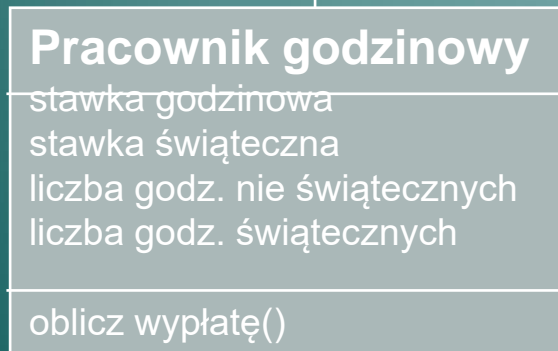
**Klasa abstrakcyjna nie może znaleźć się w liściu drzewa.
Klasa konkretna może zająć każde położenie w drzewie.**

Metoda abstrakcyjna

Metoda abstrakcyjna jest to metoda wyspecyfikowana w nadklasie, której implementacja musi znaleźć się w którejś z podklas.

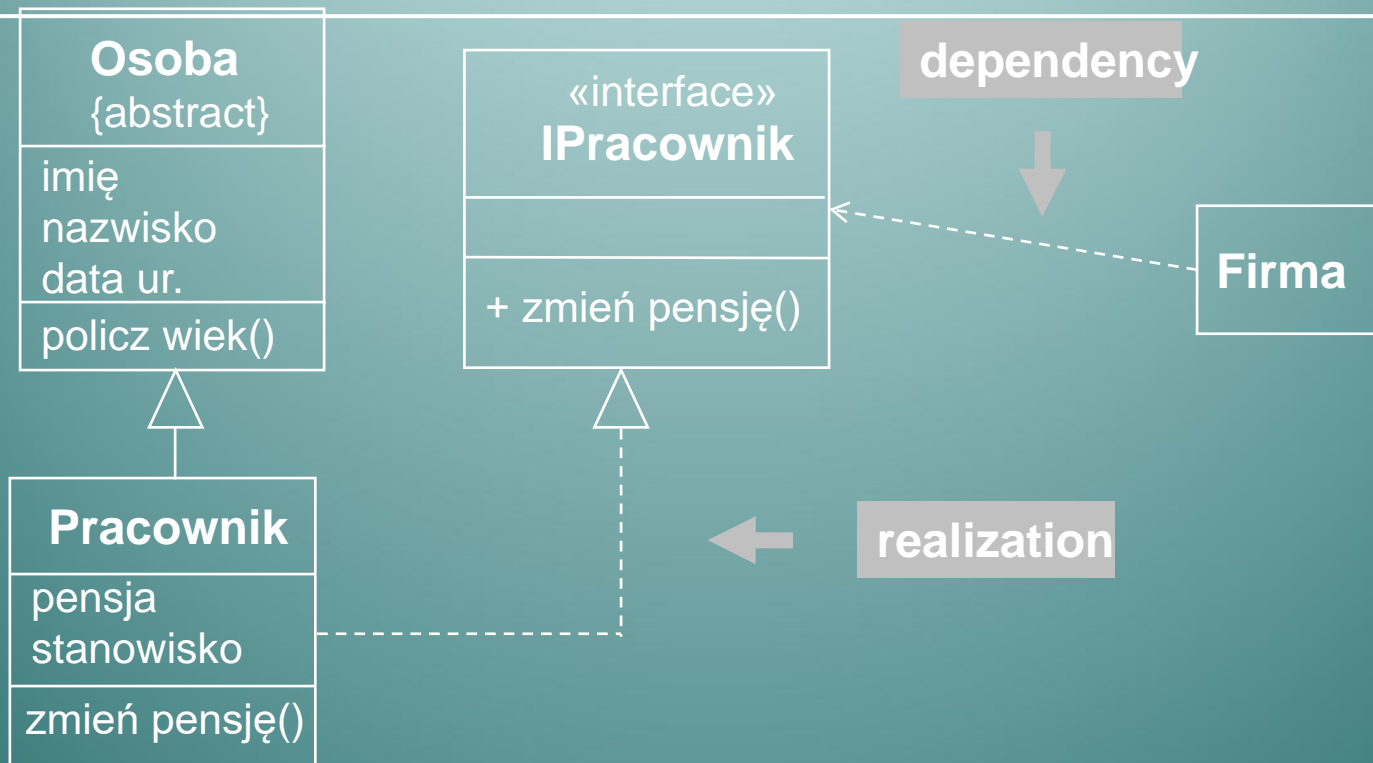


Specyfikacja operacji *ObliczWypłatę* znajduje się w klasie abstrakcyjnej *Pracownik*. Każda z klas konkretnych zawiera właściwą dla siebie implementację tej operacji.



Klasa abstrakcyjna **może** zawierać abstrakcyjne metody, ale **nie musi**, co oznacza, że klasa abstrakcyjna może zawierać wyłącznie metody zaimplementowane. Klasa konkretna **musi** zawierać implementacje tych metod abstrakcyjnych, które nie zostały zaimplementowane w żadnej z nadklas danej klasy konkretnej. Klasa konkretna nie może zawierać metod abstrakcyjnych.

Interfejs, zależność, realizacja (1)



Stereotyp «interface» poprzedza nazwę klasy, która zawiera jedynie specyfikacje metod, bez implementacji. W UML interfejs nie zawiera atrybutów, a wszystkie metody są publiczne. Implementacje metod wyspecyfikowanych w interfejsie **Ipracownik** zawiera klasa **Pracownik**, na co wskazuje symbol realizacji (ang. realization) o notacji podobnej do notacji dziedziczenia.

Zależność (ang. **dependency**) wskazuje na klasę, która korzysta z danego interfejsu.

Interfejs, zależność, uszczegółowienie (2)

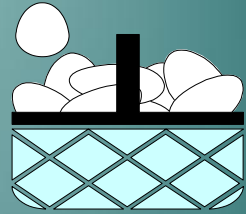
Dla poprzedniego diagramu można zastosować inną, bardziej zwięzłą notację.



Klasa abstrakcyjna i interfejs zostały tu potraktowane w podobny sposób – jako definicje interfejsów do klasy **Pracownik**. Jedyna różnica: klasa abstrakcyjna, w przeciwieństwie do interfejsu, może zawierać atrybuty i implementacje metod.

Ekstensja klasy (1)

Ekstensja klasy (class extent) = aktualny (zmienny w czasie) zestaw wszystkich wystąpień tej klasy. Ekstensja klasy w implementacji oznacza specjalną strukturę danych, konkretny byt programistyczny dołączony do klasy. Ta struktura stanowi skład obiektów, przechowując wszystkie obiekty będące członkami danej klasy.



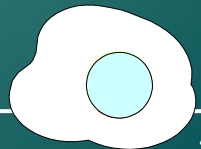
Niektóre metody zawarte w ramach klasy odnoszą się do jej **wystąpień**:

`oPracownik.wiek` `oPracownik.zwolnij` `oKonto.obliczProcent`

Niektóre metody zawarte w ramach klasy odnoszą się do jej **ekstensji**:

`KIPracownik.nowy` `KIPracownik.zlicz` `KIKonto.obliczSum`
`e`

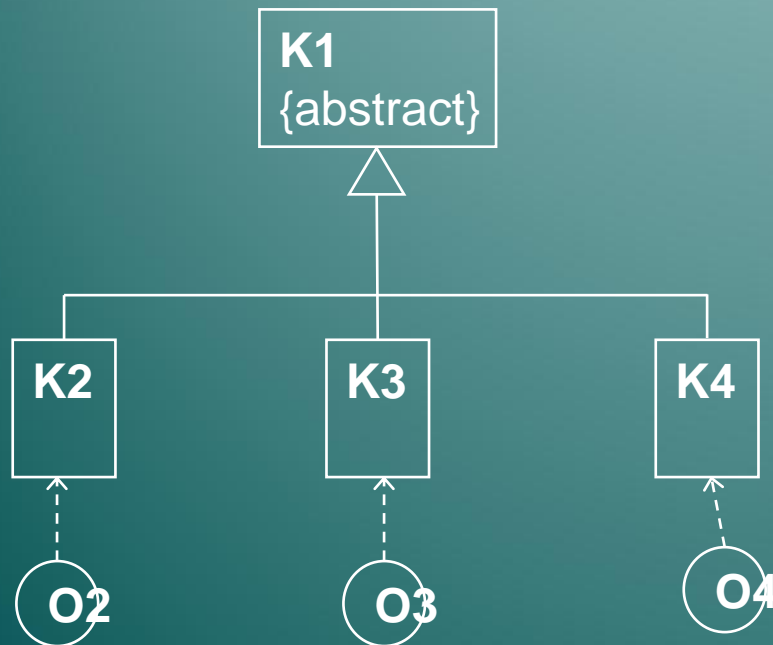
Klasa może mieć nie jedną, ale wiele ekstensji.



Ekstensja klasy (2)

Istnieje kilka definicji ekstensji klasy:

- I jest to zbiór jedynie bezpośrednich wystąpień danej klasy,
- ✓ II jest to zbiór wszystkich wystąpień danej klasy (bezpośrednich i pośrednich), ale obcięty do atrybutów wyspecyfikowanych w tej klasie,
- III jest to, jak poprzednio, zbiór wszystkich wystąpień danej klasy, ale bez obcinania atrybutów, które zostały wyspecyfikowane w podklasach tej klasy.

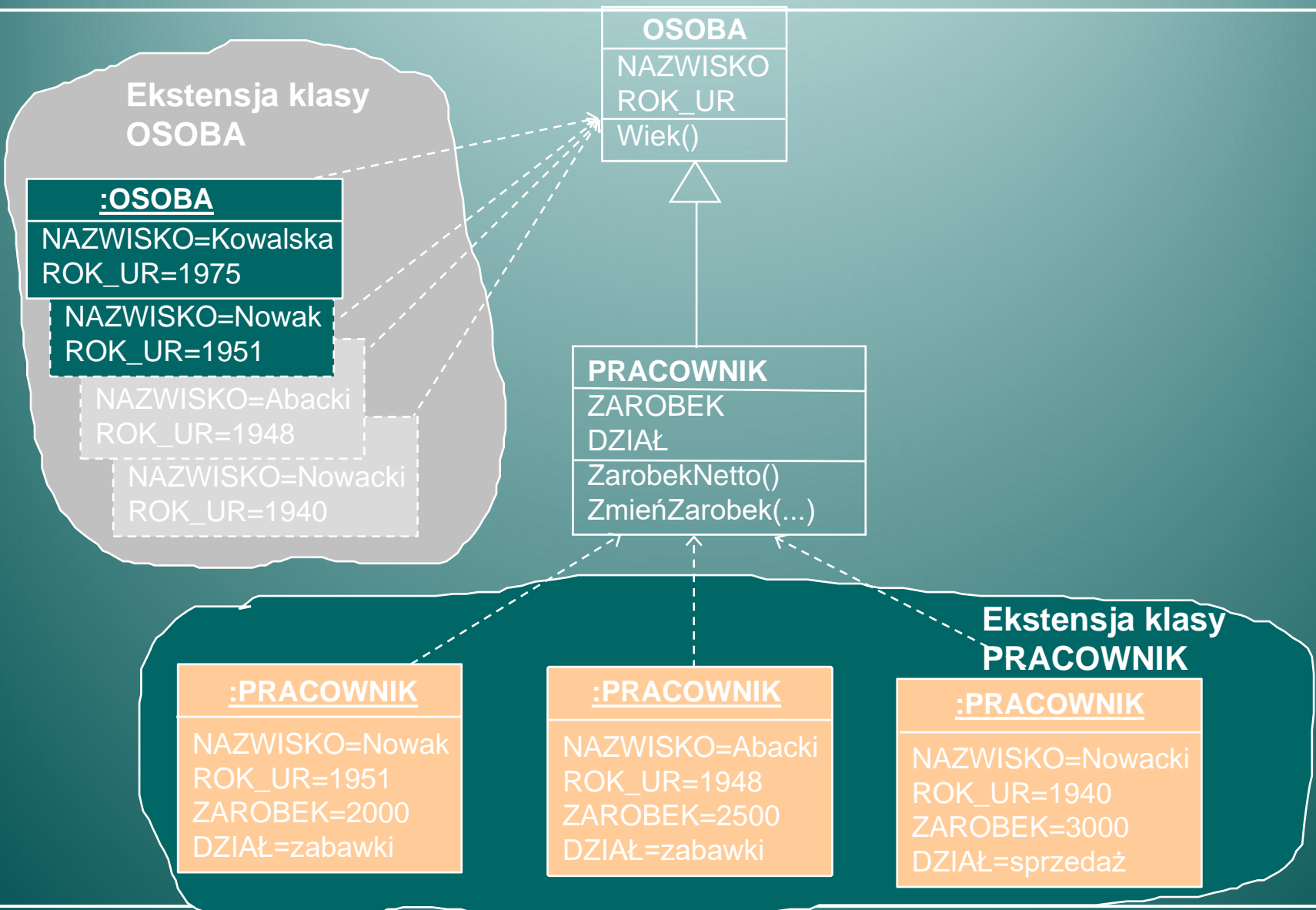


I $E_{K1} = \{\}, E_{K2} = \{O2\}, E_{K3} = \{O3\}, E_{K4} = \{O4\}$

II $E_{K1} = \{O2, O3, O4\}, E_{K2} = \{O2\}, E_{K3} = \{O3\}$
 $E_{K4} = \{O4\}$

III $E_{K1} = \{O2, O3, O4\}, E_{K2} = \{O2\}, E_{K3} = \{O3\}$
 $E_{K4} = \{O4\}$

Ekstensja klasy; przykład



Atrybuty (1)

Atrybut może być **nazwaną wartością** lub **obiektem** (podobiekt). Atrybut, będący **wartością**, nie posiada tożsamości. Wartości atrybutów są przechowywane przez obiekty, ponieważ nie należą do inwariantów klasy.

Uwaga! Sformułowanie “**wartość atrybutu**” w przypadku, gdy atrybut jest podobiektem jest uproszczeniem.

nazwisko
wiek



atrybuty obiektów klasy **Osoba**

Klasa z atrybutami

Osoba
nazwisko : string /wiek : integer

Obiekty (wystąpienia klasy) z wartościami atrybutów

<u>O-Nowak : Osoba</u>
nazwisko = " Nowak" /wiek = 53

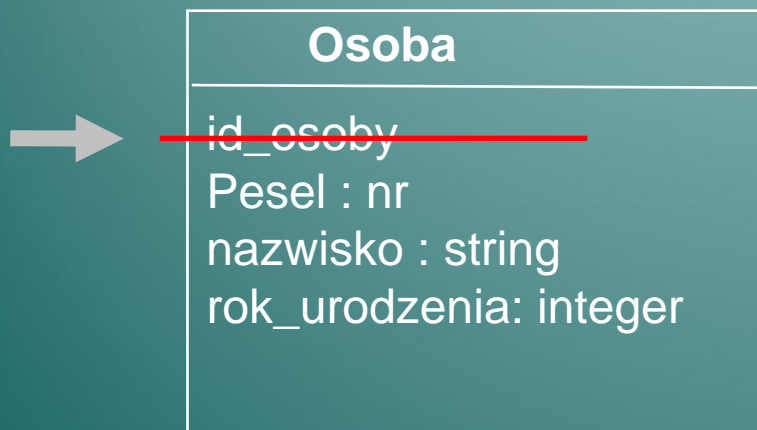
<u>O-Stycz : Osoba</u>
nazwisko = " Stycz " /wiek = 24

Atrybuty (2)

Identyfikator obiektu

Atrybut unikalnie identyfikujący obiekt (**klucz**) nie jest wymagany, ponieważ każdy obiekt posiada tożsamość, implementowaną poprzez wewnętrzny unikalny identyfikator obiektu, automatycznie generowany przez system w momencie powoływania obiektu do życia i niewidoczny dla użytkownika.

Zaleca się, by identyfikator nie miał znaczenia w dziedzinie problemowej.



Oczywiście, jeżeli identyfikator jest wymagany ze względu na potrzeby biznesowe, to można go umieścić, np. numer faktury.

Atrybuty (3)

Atrybuty mogą być:

- **proste:** *imię, nazwisko, nazwisko panieńskie, wiek, płeć, stosunek do służby wojsk.*
- **złożone:** *data ur., adres, lista poprz. miejsc pracy, dane firmy, zdjęcie*
- **opcjonalne:** *nazwisko panieńskie, stosunek do służby wojsk, lista poprzednich miejsc pracy*
- **powtarzalne:** *lista poprz. miejsc pracy*
- **pochodne:** *wiek*
- **klasowe:** *adres firmy*
- **atrybut będący obiektem:** *zdjęcie*

Pracownik

imię
nazwisko
nazwisko panieńskie : [0..1]
data ur.
/wiek
adres
płeć
stosunek do służby wojsk. : [0..1]
lista poprz. miejsc pracy : [0..*]
dane firmy
zdjęcie

Atrybuty klasowe należą do inwariantów danej klasy.

Kiedy z atrybutu warto zrobić klasę?

W jakiej sytuacji atrybut *adres firmy* przestanie być atrybutem klasowym?

Specyfikacja metod

Metoda może mieć argumenty (oprócz obiektu, który jest argumentem *implicite* dla metod obiektu). **Sygnatura (specyfikacja) metody** włącza liczbę i typ argumentów plus typ wyniku metody.

Osoba
nazwisko
wiek
dane firmy
adres
zmień firmę (nowaFirma)
zmień_adres

Plik
nazwa_pliku
długość w bajtach
ostatniaZmiana
drukuj

Obiekt geometryczny
kolor
pozycja : Punkt
lista wierzchołków
przesuń (delta : Wektor)
wewnątrz (p : Punkt) : Boolean
obróć (kąť)

Jeżeli argumenty nie są specyfikowane, to może ich być dowolnie dużo, również w ogóle. Brak specyfikacji argumentów na wczesnych etapach analizy może oznaczać zarówno, że metoda ich nie posiada, jak i to że w danym momencie nie interesujemy się jeszcze nimi. To samo dotyczy wartości zwracanej przez metode.

Rodzaje metod

Metody mogą być:

- **abstrakcyjne**
- **objektu:** *policz wiek, czy pracował w (nazwa f)*
- **klasowe:** *policz wiek (imię, nazwisko),
znajdź najstarszego*

Klasa **Pracownik** nie może posiadać metod abstrakcyjnych, gdyż jako jedyna klasa na diagramie musi być klasą konkretną.

Metoda obiektu operuje na atrybutach jednego obiektu, tego dla którego została wywołana. Obiekt stanowi argument domyślny (implicite) dla metody obiektu.

Metoda klasowa operuje na ekstensji klasy, czyli posiada dostęp do atrybutów wszystkich obiektów członków danej klasy.

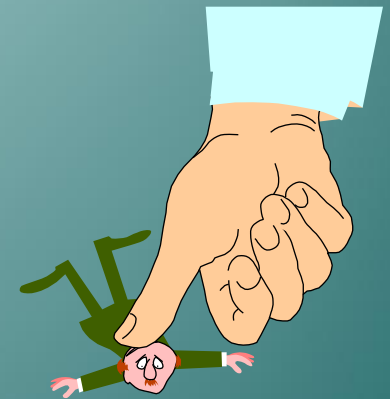
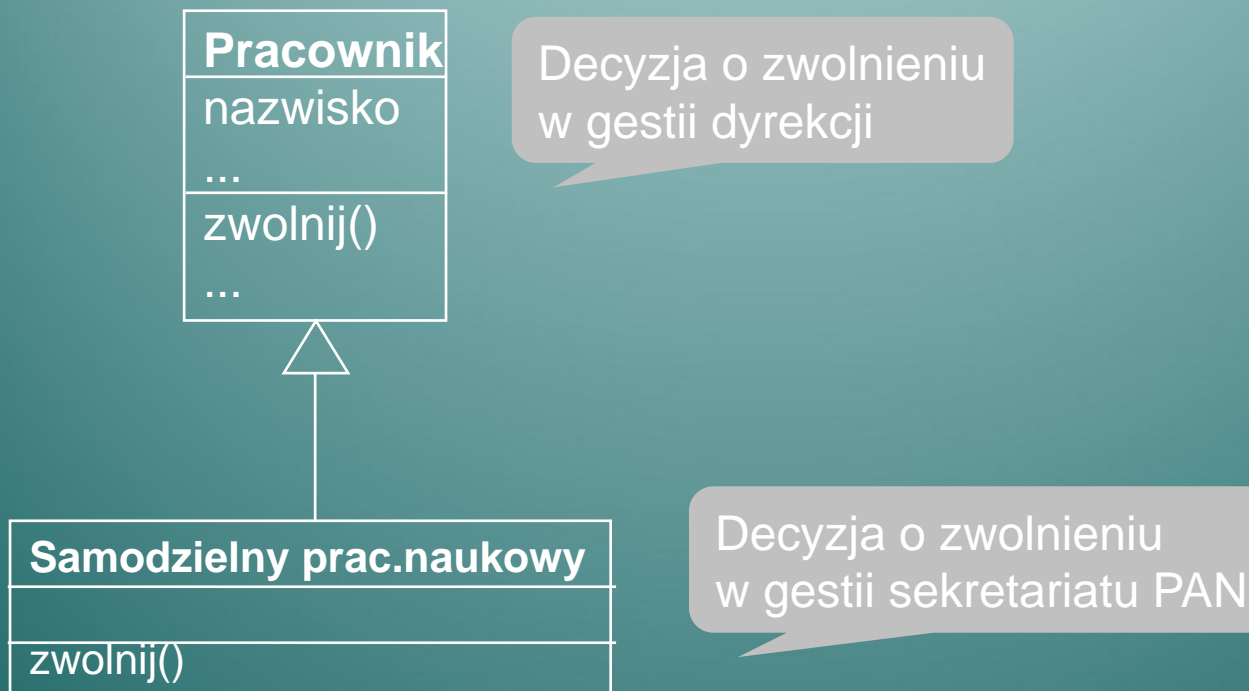
Pracownik

imię
nazwisko
data ur.
/wiek
adres zamieszkania
płeć
stosunek do służby wojsk. : [0..1]
lista poprz. miejsc pracy : [0..*]
adres firmy

~~policz wiek (imię, nazwisko)~~
policz wiek
policz wiek (imię, nazwisko)
czy pracował w (nazwa firmy)
znajdź najstarszego

Przesłanianie metod (1)

Przesłanianie (overriding): metoda z klasy bardziej wyspecjalizowanej może przesłonić metodę z klasy bardziej ogólnej. Wybierana jest metoda znajdująca się najbliżej obiektu, w sensie hierarchii dziedziczenia.



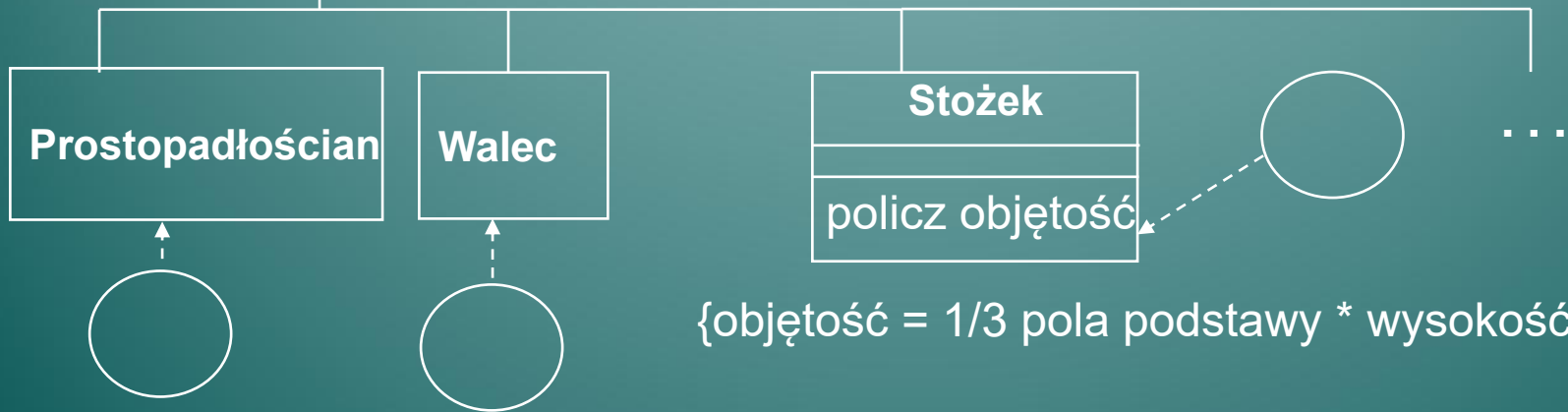
Metody mają tu identyczną sygnaturę ale różne implementacje (ciała).

Przesłanianie metod (2)



- ✓ Przesłanianie jest ściśle powiązane z polimorfizmem metod
- ✓ Przesłanianie wymaga dynamicznego wiązania.
- ✓ Przesłanianie jest ważnym elementem wspomagającym ponowne użycie.

{objętość = pole podstawy * wysokość}



{objętość = 1/3 pola podstawy * wysokość}

Dwie metody implementujące operację *policz objętość*. Metoda *policz objętość* w klasie abstrakcyjnej *Bryła* nie może być metodą abstrakcyjną, ponieważ jest

dziedziczona

Dynamiczne (poźne) wiązanie

Wiązanie (*binding*): zamiana identyfikatora symbolicznego (nazwy) występującego w programie na: wartość, adres lub wewnętrzny identyfikator bytu programistycznego (stałej, zmiennej, obiektu, procedury, ...)

Wczesne (statyczne) wiązanie: przed uruchomieniem programu, podczas kompilacji i konsolidacji.

Zalety: większa szybkość działania programu, możliwość pełnej statycznej kontroli typów.

Wady: brak możliwości rozbudowy aplikacji podczas jej działania.

Późne (dynamiczne) wiązanie: w czasie wykonania programu.

Zalety: możliwość przesłaniania w trakcie działania aplikacji, możliwość komponowania programu w trakcie jego działania, szybkie przechodzenie od pomysłu do realizacji.

Wady: wolniejsze działanie programu, utrudniona kontrola typów

Późne wiązanie jest nieodzownym warunkiem dla:

- implementacji komunikatów (polimorfizmu)
- dynamicznie tworzonych perspektyw
- dynamicznie tworzonych procedur bazy danych
- języków zapytań
- migracji obiektów
- ewolucji schematu BD

Przeciążanie metod

Przeciążanie (overloading): oznacza, że jakiś symbol (np. operatora czy funkcji) ma znaczenie zależne od kontekstu jego użycia, np. od ilości/typu argumentów.

Np. `przesuń(x, y)`, `przesuń(x, y, z)` – mają różną ilość argumentów
`przesuń(int, int)`, `przesuń(float, float)` – mają różne typy argumentów

Powszechne jest przeciążanie operatora równości `=` który służy do porównania liczb całkowitych, liczb rzeczywistych, stringów, identyfikatorów, struktur, itd.

Podobnie, operator `+` może oznaczać dodawanie lub konkatencję.

Przeciążanie nie wymaga dynamicznego wiązania: znaczenie operatora można wydedukować na podstawie **statycznej** analizy tekstu programu. W odróżnieniu od przeciążania, przesłanianie jest własnością **dynamiczną**, nie zawsze da się wydedukować z tekstu programu.

Niektórzy autorzy (np. Cardelli – propagator teorii typów polimorficznych) uważają, że przeciążanie nie jest polimorfizmem. Stwierdzenie “Wszystkie metody implementujące daną operację muszą mieć tę samą sygnaturę”, leżące u podstaw idei polimorfizmu, jest sprzeczne z definicją przeciążania.

Typ

Typ bytu programistycznego nakłada ograniczenia na jego budowę (lub argumenty i wynik) oraz ogranicza kontekst, w którym odwołanie do tego bytu może być użyte w programie.

W wielu opracowaniach i językach (C++, Eiffel) typ jest utożsamiany z klasą. Wielu autorów uważa jednak te dwa pojęcia za różne.

Klasa: przechowalnia inwariantów, implementacja metod.

Typ: specyfikacja budowy obiektu, specyfikacja metod.

Podstawowe zastosowanie klasy: modelowanie pojęciowe.

Podstawowe zastosowanie typu: wspomaganie kontroli formalnej poprawności programów.

Generalnie, na linii rozróżnień definicyjnych pomiędzy pojęciami:

- klasa
- typ
- abstrakcyjny typ danych (ADT)
- ekstensja

panuje spore zamieszanie.

Mocna kontrola typów

Mocna kontrola typów oznacza, że każdy byt programistyczny (stała, zmienna, obiekt, procedura, funkcja, metoda, klasa, moduł, ...) podlega obowiązkowej specyfikacji typu. Każde odwołanie do tego bytu w programie jest sprawdzane na zgodność ze specyfikacją jego typu.

Statyczna kontrola typu: kontrola tekstu programu (podczas kompilacji).

Dynamiczna kontrola typu: kontrola typów podczas czasu wykonania.

Zwykle mocna kontrola typów oznacza kontrolę statyczną.

Kontrola dynamiczna jest znacznie mniej skuteczna, z dwóch powodów:

- jest istotnym obciążeniem czasu wykonania,
- błąd typu podczas wykonania jest takim samym błędem jak każdy inny, a rakietę przecież jest już w locie...

Z drugiej strony, mocna statyczna kontrola typów powoduje znaczne zmniejszenie mocy języka programowania i jego elastyczności. Np. jak napisać procedurę w Pascal'u, która mnoży dwie macierze o dowolnych rozmiarach?

Własności takie jak: późne wiązanie, wartości zerowe, warianty, perspektywy, procedury bazy danych, dynamiczne klasy, etc. wymagają kontroli dynamicznej.

Podtyp

Dwie definicje:

➡ Ekstensja podtypu jest podzbiorem ekstensji typu

Np. zbiór liczb naturalnych jest podtypem zbioru liczb całkowitych.

➡

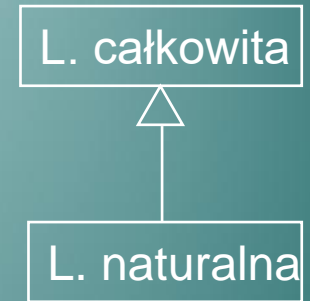
Typ B jest podtypem typu A, jeżeli B posiada więcej własności (atrybutów, metod, ...) niż A, innymi słowy B jest bardziej wyspecjalizowane niż A.



```
struct Osoba {string Nazwisko; integer Rok_urodz;};  
struct Pracownik {string Nazwisko; integer Rok_urodz; integer Zarobek; }
```

Pracownik jest podtypem Osoba

Innym (równoważnym) punktem widzenia na kwestię podtypowania jest założenie, że każdy obiekt może mieć wiele typów: swojej klasy podstawowej i wszystkich jej nadklas.



Własność zamienialności

Definiowanie relacji podtypu między typami posiada konkretny cel, określany przez zasadę **zamienialności (substitutability)**:

Jeżeli w jakimś miejscu programu (zapytania, ...) może być użyty byt typu *A*, to może tam być także użyty byt, którego typ jest podtypem typu *A*.

Np., jeżeli w jakimś miejscu programu może być użyty obiekt *Osoba*, to w tym samym miejscu może być użyty obiekt *Pracownik*. Wszędzie, gdzie może być użyta *Elipsa*, można też użyć obiektu klasy *Koło*, wszędzie gdzie może być użyta liczba całkowita można użyć liczby naturalnej. Zamiana odwrotna nie jest możliwa.

Zasada zamienialności ma duże znaczenie dla przyrostowego rozwoju oprogramowania: obiekty nowych, bardziej wyspecjalizowanych klas mogą być wykorzystywane w tym samym środowisku, co mniej wyspecjalizowane, bez potrzeby zmiany środowiska przy każdej zmianie związanej z rozszerzeniami wynikłymi ze specjalizacji.

Typy masowe

Typy masowe to typy, dla których rozmiar bytu nie da się ani przewidzieć ani sensownie ograniczyć.

Kolekcje (termin ODMG-93 przyjęty dla określenia typów masowych):

Zbiory (sets): nie uporządkowane kolekcje elementów dowolnego ustalonego typu, bez powtórzeń.

Wielozbiory (multisets, bags): nie uporządkowane kolekcje elementów dowolnego ustalonego typu, elementy mogą się powtarzać.

Sekwencje (sequences): uporządkowane kolekcje elementów dowolnego ustalonego typu; porządek ma znaczenie informacyjne, elementy mogą się powtarzać.

Tablice dynamiczne (dynamic arrays): sekwencje, ale z dostępem poprzez indeks.

Ortogonalność konstruktorów typu:

typy masowe mogą być dowolnie kombinowane, w tym również z typami indywidualnymi, np. zbiór sekwencji czy obiekt, którego atrybutami są wielozbiory.

Rozszerzalność systemu typów

- ✓ Projektant ma do wyboru wiele konstruktorów typu.
- ✓ Nowy typ można zdefiniować na podstawie typu już istniejącego (ortogonalna kombinacja)

Konstruktory typów:

- **typy atomowe:** character, integer, float, string, boolean, bitmap, ...
- **typy zapisów (records):** struct {nazwa:string; waga:float;}
- **kolekcje:**
 - ⊙ **zbiory (sets):** set of bitmap, set of struct {nazwa:string; waga:float;}
 - ⊙ **wielozbiory (bags):** zbiory z powtórzeniami
 - ⊙ **sekwencje (sequences):** wielozbiory uporządkowane
 - ⊙ **tablice (arrays):** array of integer, array[5..30] of set of bitmap

Definicja nowego typu na podstawie typu już zdefiniowanego:

```
TypCzęści = struct {string nazwa; float waga;};  
TypRelacjiCzęści = set of TypCzęści;
```

Rozszerzalność systemu typów znacząco wspomaga ponowne użycie.