Diagram klas - Class diagram

Widok logiczny w systemie 4+1 Kruchtena używany jest do modelowania części systemu oraz sposobów, w jaki one ze sobą współdziałają.

Ten widok tworzą zazwyczaj diagramy:

- **klas**
- obiektów
- maszyny stanowej
- interakcji

Diagram klas - cechy:

- Zawiera informacje o statycznych związkach między elementami (klasami)
- Klasy są ściśle powiązane z technikami programowania zorientowanego obiektowo
- Są jednymi z istotniejszych diagramów w UML

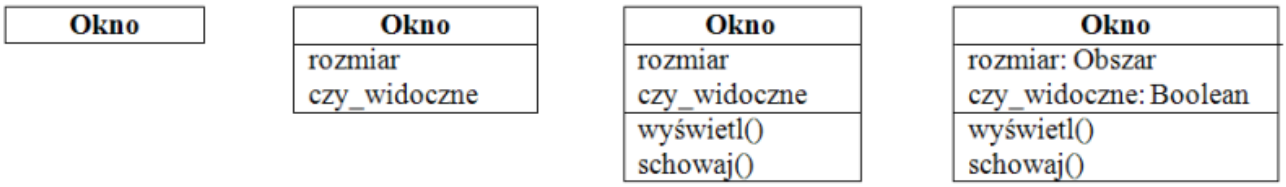
Symbol klasy

- Symbolem klasy jest prostokąt, zwykle podzielony poziomymi liniami na trzy sekcje:
  - nazwy
  - atrybutów
  - operacji
- W razie potrzeby może zostać uzupełniony dodatkowymi sekcjami (np. wyjątków)

Klasa - notacja

Podstawowe sposoby przedstawiania klas, różne poziomy szczegółowości

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



Pole nazwy klasy:

```
stereotyp nazwa_ klasy lista_wart_etyk
```

Pole atrybutów:

```
stereotyp dostępność nazwa_atrybutu : typ = wart_początkowa lista_wart_etyk
```

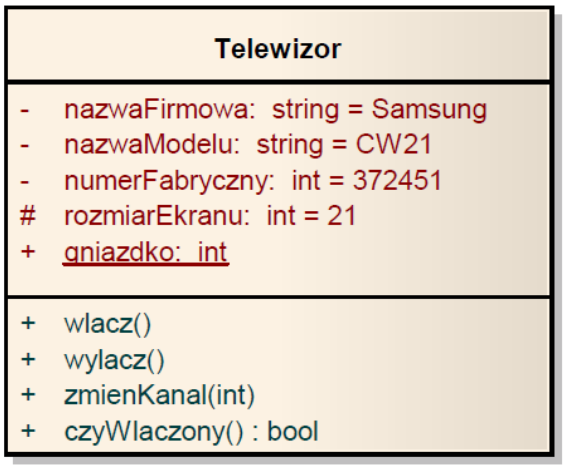
Pole metod:

```
stereotyp dostępność nazwa_metody (lista_arg) : typ_wart_zwracanej
lista_wart_etykt
```

Poziomy dostępu:

Dostępności:

+ publiczna
- prywatna
chroniona
~ zakres pakietu




```
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.

Składniki statyczne

- Składniki mozna zadeklarować jako statyczne ? działające na rzecz klasy, nie obiektu
- Koncepcja identyczna jak w językach zorientowanych obiektowo
- Reprezentacją graficzną jest **podkreślenie**

Krotność

Krotność pozwala określić minimalną i maksymalną liczbę obiektów, jakie można powiązać z daną cechą:

- **dolna granica..górna granica** - przedział od-do
- **1** - dokładnie jeden obiekt
- **0..1** - opcjonalnie jeden obiekt
- **1..*** - przynajmniej jeden obiekt
- **1, 3, 5** - konkretne liczby obiektów
- * - dowolna liczba obiektów

Krotność cechy wskazuje, ile obiektów można, a ile trzeba w niej zawrzeć. Krotność można określać jako ograniczenie dolne i górne, jednak oczywiste lub powtarzające się wartości graniczne można pomijać, np.. zapis 0..* jest skracany do *, a zapis 1..1 do 1. W praktyce programowania istotna jest krotność 0, 1 i dowolna, natomiast wartości dyskretne są mniej ważne, jako szczególne przypadki wymienionych trzech. W UMLu 2.0 dlatego formalnie

usunięto możliwość podawania dowolnych liczb będących ograniczeniami, np. 2..4, jednak z uwagi na czytelność tego zapisu użycie go nie stanowi wielkiego błędu.

Atrybuty

Atrybuty wpisane
Atrybuty zadeklarowane poprzez asocjację

Ogólna deklaracja atrybutu wpisanego

```
[widoczność] nazwa [liczebność] [:typ] [=wartość początkowa]] [właściwość]
```

- właściwość changeable - bez ograniczeń
- addOnly - gdy liczebność jest większa niż jeden, można dodawać nowe wartości, ale raz dodane nie moga byc zmieniane
- readOnly, frozen - nie ma możliwości zmiany (jak const w C++)
- Przykład: `+ wysokosc : double = 10 frozen`

Ogólna deklaracja operacji

```
[widoczność] nazwa [(lista_parametrów)] [: typ_wyniku] [właściwość]
```

gdzie lista parametrów:

```
[tryb] nazwa : typ [=wartość_domyślna]
```

tryb:

- **in** - wejściowy, niemoże być modyfikowany
- **out** - wyjściowy, może być modyfikowany
- **inout** - wejściowy, wyjściowy, może być modyfikowany

właściwość:

- **leaf** - funkcja niepolimorficzna (nie może być nadpisana)
- **isQuerry** - funkcja nie zmieniająca obiektu
- **sequental, guarded, concurent** (mają zastosowanie przy operacjach współbieżnych)

Nazwa funkcji pisana *kursywą* - **funkcja wirtualna**

Związki między klasami

Wymienione od najsłabszych:

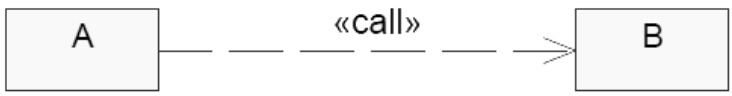
- Zależność
- Asocjacja
- Agregacja częściowa
- Agregacja całkowita
- Dziedziczenie

Zależność - *dependency*

Zależność pomiędzy dwiema klasami informuje, że jedna z nich, aby używać obiektów innej, musi mieć o niej informacje. Zależność występuje, gdy zmiana specyfikacji jednej klasy, może powodować konieczność wprowadzania zmiany w innej klasie. Najczęściej używa sie zależności do pokazania, że jedna klasa używa innej jako parametru jakiejś operacji. Obie klasy są zależne od siebie nawzajem w czemu zapewnienia poprawnej pracy!

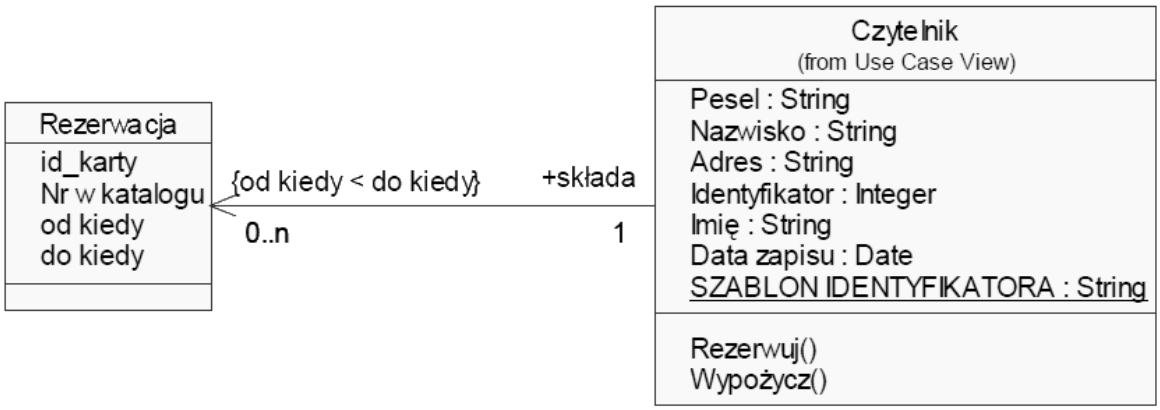
Zależności są najprostszym i najsłabszym rodzajem relacji łączących klasy. Oznaczają, że zmiana jednej z nich w pewien sposób wpływa na drugą, np.

- << call >> - operacje w klasie A wywołują operacje w klasie B
- << create >> - klasa A tworzy instancje klasy B
  << instantiate >> - obiekt A jest instancją klasy B
- << use >> - do zaimplementowania klasy A wymagana jest klasa B



Asocjacja

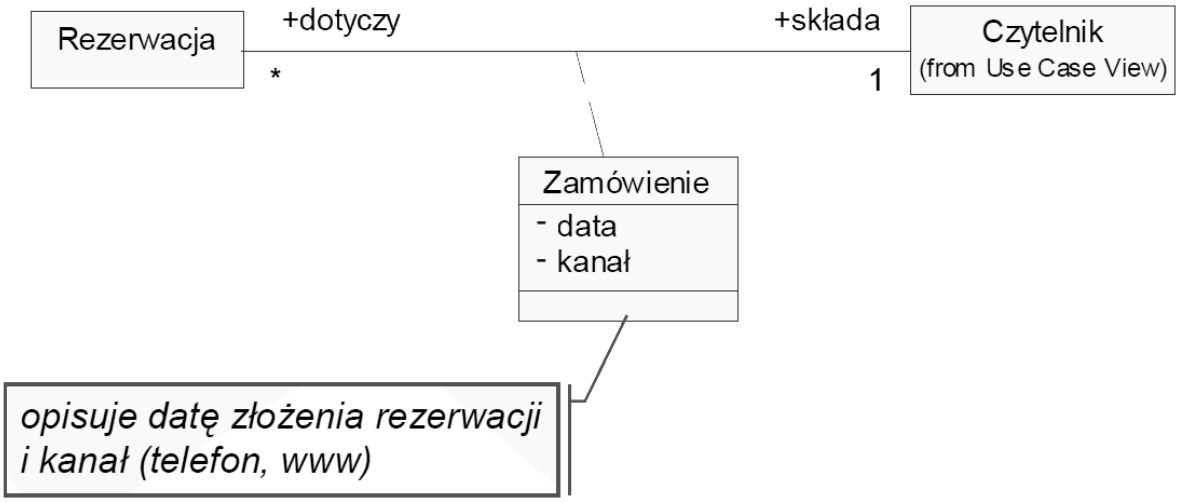
Asocjacja reprezentuje czasowe powiązanie pomiędzy obiektami dwóch klas. Obiekty związane asocjacją są od siebie niezależne. Asocjacja jest też używana jako alternatywny (obok atrybutu) i równorzędny sposób zapisu cech klasy.



Asocjacje są silniejszymi relacjami niż zależności. Wskazują, że jeden obiekt jest związany z innym przez pewien okres czasu. Jednak czas życia obu obiektów nie jest od siebie zależny: usunięcie jednego nie powoduje usunięcia drugiego. Relacje asocjacji są zwykle opisywane frazami "...posiada...", "...jest właścicielem...", jednak ich znaczenie często jest mylone z inną relacją - agregacją. W przypadku asocjacji żaden obiekt nie jest właścicielem drugiego: nie tworzy go, nie zarządza nim, a moment usunięcia drugiego obiektu nie jest z nim związany. Z drugiej strony, obiekt powiązany asocjacją z drugim posiada referencję do niego, może się do niego odwołać etc. Asocjacje mogą posiadać nazwy, zwykle w postaci czasownika, który pozwala przeczytać w języku naturalnym jej znaczenie, np. ?A posiada B?. Często pomija się jedną z nazw asocjacji dwukierunkowej, jeżeli jest ona jedynie stroną bierną drugiej nazwy, np. ?przechowuje? ? ?jest przechowywany?. Asocjacja jest równoważna atrybutowi: UML nie rozróżnia obiektu, który jest polem klasy od obiektu i jest z nią związany asocjacją. Warto jednak przyjąć konwencję, w której obiekty reprezentujące wartości (np. daty) oraz typy proste (liczby, napisy, znaki) są modelowane jako atrybuty, natomiast obiekty dostępne poprzez referencje ? są przedstawiane poprzez asocjacje.

Klasa asocjacyjna

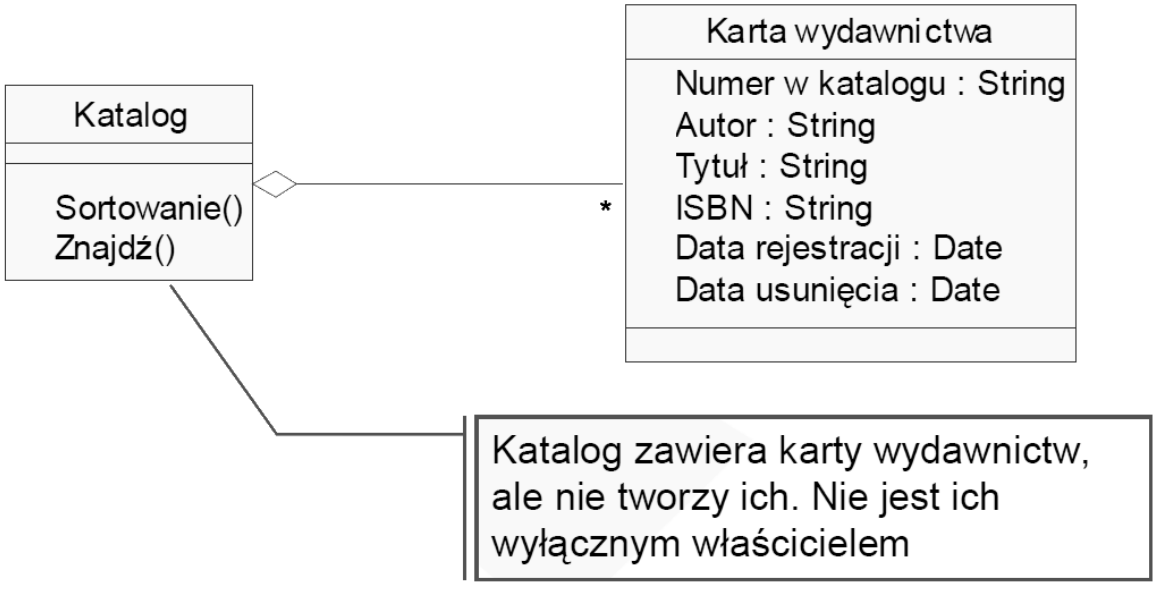
Klasy asocjacyjne są związane z relacją asocjacji i opisują jej właściwości. Mają dostęp do innych obiektów uczestniczących w relacji



Klasa asocjacyjna umożliwia opisanie za pomocą atrybutów i operacji nie obiektu, ale właśnie samej asocjacji pomiędzy klasami. Informacje przechowywane w klasie asocjacyjnej nie są związane z żadną z klas uczestniczących w asocjacji, dlatego wygodnie jest stworzyć dodatkową klasę i powiązać ją z relacją. Klasy asocjacyjne są reprezentowane graficznie jako klasy połączone linią przerywaną z relacją asocjacji, której dotyczą.

Agregacja częściowa

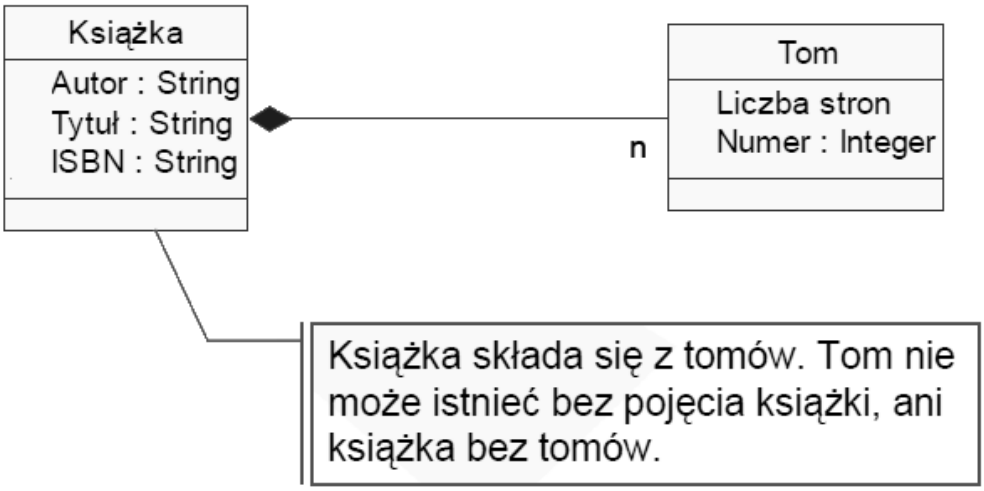
Agregacja reprezentuje relację typu całość-część, w której część może należeć do kilku całości, a całość nie zarządza czasem istnienia części.



Agregacja jest silniejszą formą asocjacji. W przypadku tej relacji równowaga między powiązanymi klasami jest zaburzona: istnieje właściciel i obiekt podrzędny, które są ze sobą powiązane czasem swojego życia. Właściciel jednak nie jest wyłącznym właścicielem obiektu podrzędnego, zwykle też nie tworzy i nie usuwa go. Relacja agregacji jest zaznaczana linią łączącą klasy/obiekty, zakończoną białym rombem po stronie właściciela

Agregacja całkowita (kompozycja)

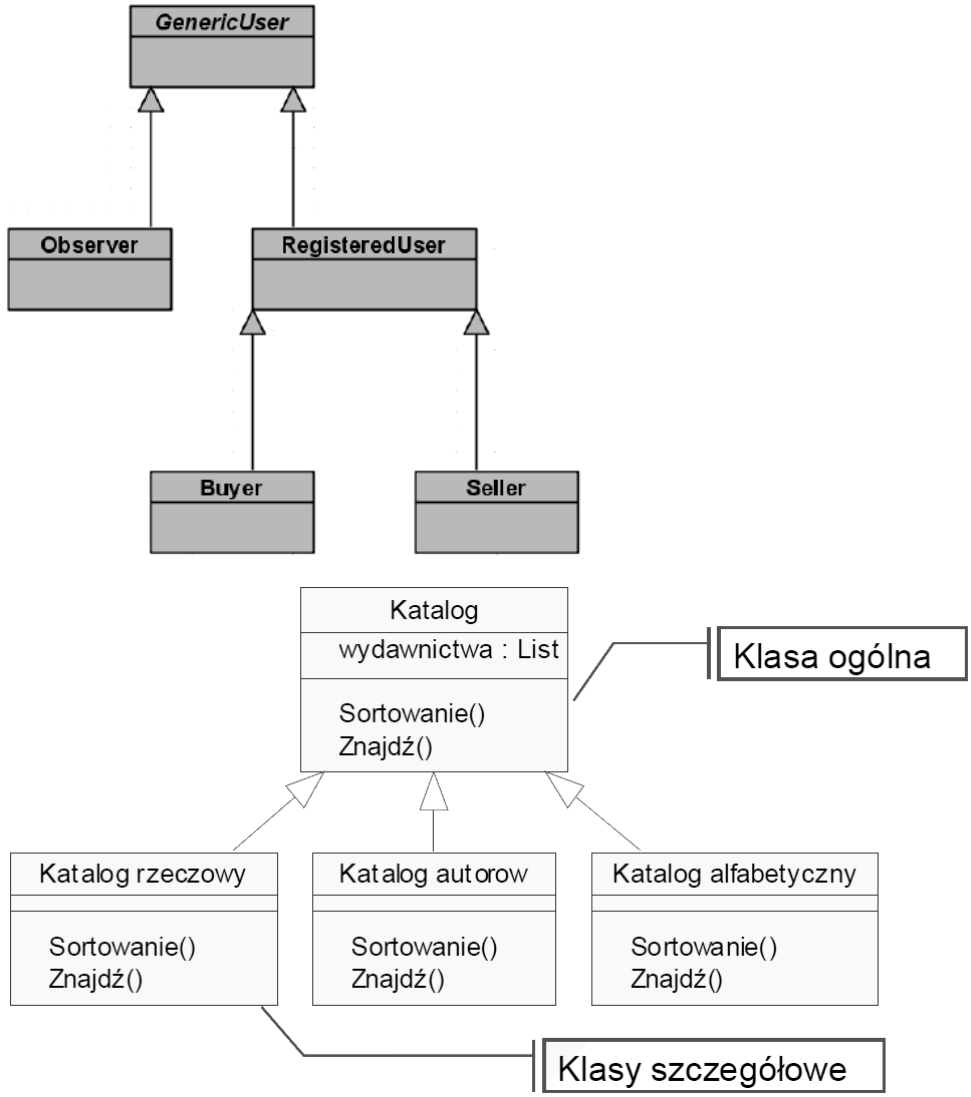
Agregacja całkowita jest relacją typu całość-część, w której całość jest wyłącznym właścicielem części, tworzy je i zarządza nimi.



Agregacja całkowita jest najsilniejszą relacją łączącą klasy. Reprezentuje relacje całość-część, w których części są tworzone i zarządzane przez obiekt reprezentujący całość. Ani całość, ani części nie mogą istnieć bez siebie, dlatego czasy ich istnienia są bardzo ściśle ze sobą związane i pokrywają się: w momencie usunięcie obiektu całości obiekty części są również usuwane. Typowa fraza związana z taką relacją to "...jest częścią...". Agregacja całkowita jest przedstawiana na diagramie podobnie jak agregacja częściowa, przy czym romb jest wypełniony.

Uogólnienie - Dziedziczenie

Uogólnienie tworzy hierarchię klas, od ogólnych do bardziej szczegółowych. Pozwala wyłączyć części wspólne klas.



Uogólnienie posiada różne interpretacje. Na przykład, w modelu pojęciowym Katalog jest uogólnieniem Katalogu rzeczowego, jeżeli każda instancja Katalogu rzeczowego jest także instancją Katalogu. Uogólnienie w przypadku klas często jest traktowane jako synonim dziedziczenia, podczas gdy **dziedziczenie** jest tylko możliwą techniką uogólniania. Inną jest np. **wykorzystanie interfejsów**, które pozwalają utworzyć relację uogólnienia/uszczegółowienia pomiędzy typami (dziedziczenie interfejsu) lub klasą i interfejsem (implementacja interfejsu).

Interfejsy i klasy abstrakcyjne

Klasa abstrakcyjna

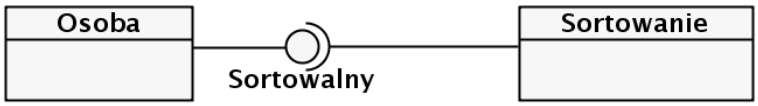
Klasa abstrakcyjna deklaruje wspólną funkcjonalność grupy klas. Nie może posiadać obiektów i musi definiować podklasy.

Interfejs

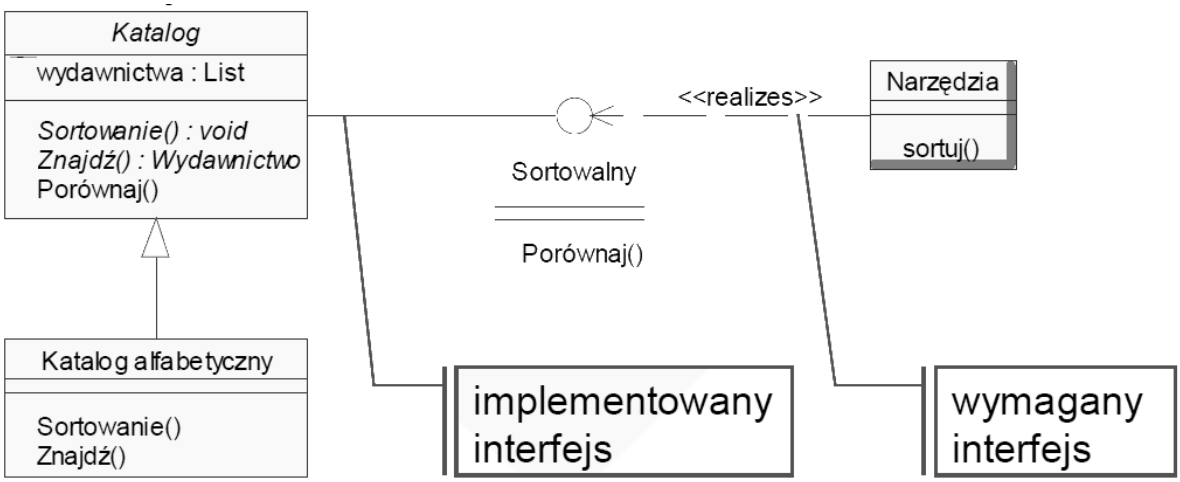
Interfejs - klasyfikator zawierający deklaracje właściwości i metod ale nie implementuje ich (definicja UML != np. Java)
Dwie reprezentacje graficzne:

- jak klasa

- jako kółko



Interfejs deklaruje grupę operacji bez podawania ich implementacji.



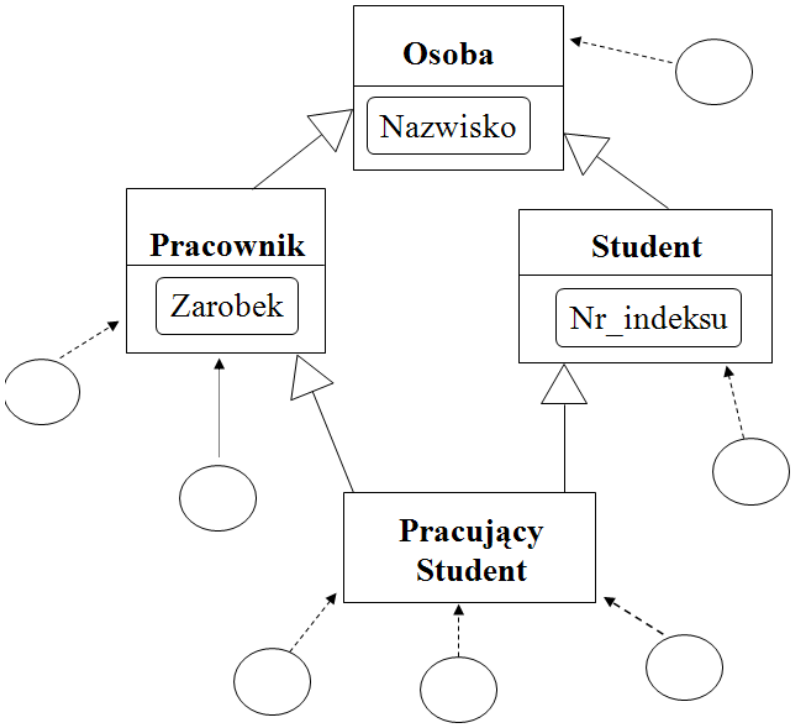
- Celem tworzenia klas abstrakcyjnych i interfejsów jest identyfikacja wspólnych zachowań różnych klas, które są realizowane w różny od siebie sposób. Użycie tych mechanizmów pozwala na wykorzystanie relacji uogólniania do ukrywania (hermetyzacji) szczegółów implementacji poszczególnych klas.
- Klasa abstrakcyjna reprezentuje wirtualny byt grupujący wspólną funkcjonalność kilku klas. Posiada ona sygnatury operacji (czyli deklaracje, że klasy tego typu będą akceptować takie komunikaty), ale nie definiuje ich implementacji.
- Podobną rolę pełni interfejs. Różnica polega na tym, że klasa abstrakcyjna może posiadać implementacje niektórych operacji, natomiast interfejs jest czysto abstrakcyjny (choć, oczywiście interfejs i klasa w pełni abstrakcyjna są pojęciowo niemal identyczne).
- Ponieważ klasy abstrakcyjne nie mogą bezpośrednio tworzyć swoich instancji (podobnie jak interfejsy, które z definicji nie reprezentują klas, a jedynie ich typy)

dlatego konieczne jest tworzenie ich podklas, które zaimplementują odziedziczone abstrakcyjne metody. W przypadku interfejsu sytuacja jest identyczna.

- Przyjętym sposobem oznaczania klas i metod abstrakcyjnych jest zapisywanie ich pochyłą czcionką lub opatrywanie słowem kluczowym **{abstract}.**

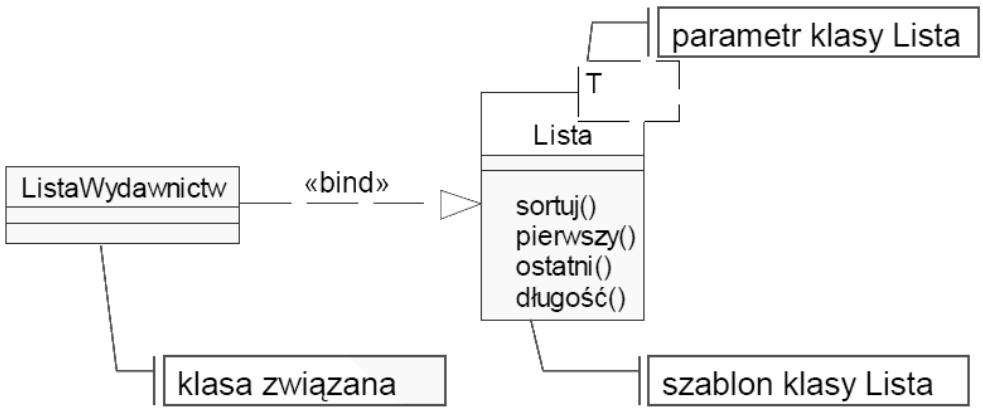
Dziedziczenie wielokrotne

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



Szablony klas

Szablon klasy określa, z jakimi innymi klasami (nie obiektami!) może współpracować podana klasa. Klasy te są przekazywane jako jej parametry. Klasa będąca uszczegółowieniem takiej klasy jest **klasą związaną.**



Szablony klas to pojęcie wywodzące się z języka C++. Oznaczają one klasy, których definicja wymaga podania argumentów będących innymi klasami. W ten sposób szablon klasy jest swego rodzaju niepełną klasą, która dopiero po ukonkretnieniu może zostać użyta.