# Inżynieria oprogramowania

Wykład 3:

Unified Modeling Language:

Diagramy klas i obiektów

Pakiety

Marek Krętowski
pokój 206
e-mail: mkret@ii.pb.bialystok.pl
http://aragorn.pb.bialystok.pl/~mkret


Wersja 1.31 st. zaoczne

## Diagramy klas

Zawierają:
- klasy, interfejsy i kooperacje
- związki: zależności, uogólnienia i powiązania

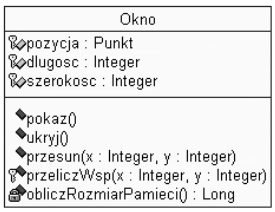
Służą do obrazowania **statycznych** aspektów perspektywy projektowej, w której bierze się pod uwagę wymagania funkcjonalne systemu (usługi, jakie system powinien udostępniać swoim użytkownikom)

Podstawowe zadania:
- **Modelowanie słownictwa systemu** - podejmowanie decyzji, które abstrakcje są częścią rozważanego systemu i określenie ich zobowiązań
- **Modelowanie prostych koopercji** - określenie zbioru kooperujących klas i związków między nimi; realizują one wspólnie zadania które są niemożliwe do wykonania w pojedynkę; pokazanie różnorakich sposobów współpracy elementów ze słownika)
- **Modelowanie schematu logicznej bazy danych** - w wielu systemach zachodzi konieczność przechowywania trwałych danych w relacyjnej lub obiektowej bazie danych; przez schemat rozumiemy plan projektu koncepcyjnego bazy danych, z tego punktu widzenia diagram klas jest nadzbiorem diagramów encja-związek (ang. ERD)



## Widoczność atrybutów i operacji

- `Public` - nieograniczony dostęp do takiego składnika
  - oznaczane "+"
- `Protected` - tylko potomkowie mają dostęp do takiego składnika
  - oznaczane "#"
- `Private` - dostęp ograniczny tylko do operacji składowych
  - oznaczane "-"

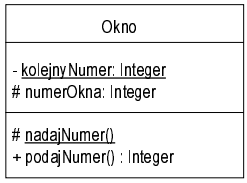



Widoczność w UML odpowiada analogicznym pojęciom zdefiniowanym w większości języków programowania (C++, Java, ...)



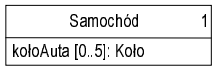
## Zasięg atrybutów i operacji

- `Instance` - każdy egzemplarz przechowuje oddzielną wartość tego składnika, zasięg domyślny
- `Classifier` - jest tylko jedna wartość tego składnika wspólna dla wszystkich egzemplarzy, oznaczane przez podkreślenie nazwy




Zasięg klasyfikatorowy odpowiada statycznym atrybutom i metodom zdefiniowanym w języku C++

- **Liczebność** - służy do ograniczenia liczby egzemplarzy konkretnej klasy lub atrybutów



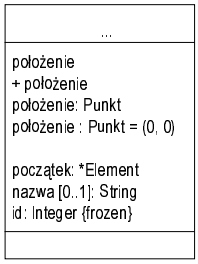



## Atrybuty

[widoczność] nazwa [liczebność] [:typ] [=wartość-początkowa] [{określenie-właściwości}]

Właściwości (ang. *properties*):
- `changeable` (nie ma ograniczeń co do modyfikacji wartości atrybutu, właściwość domyślna)
- `addOnly` (w przypadku atrybutów o liczebności >1 można dodawać nowe wartości, ale nie można ich zmieniać)
- `frozen` (wartość atrybutu nie może być zmieniona po zainicjowaniu obiektu)

Właściwość `frozen` odpowiada `const` w C++






## Operacje i metody

- **Operacja** to specyfikacja usługi, której można zażądać od obiektu natomiast **metoda** to implementacja operacji
- Każda nieabstrakcyjna operacja klasy musi być związana z metodą, której treść określa algorytm wykonywalny
- Może być wiele metod dla tej samej operacji (dzięki dziedziczeniu)

[widoczność] nazwa [(lista-parametrów)] [:typ-wyniku] [{określenie-właściwości}]

gdzie deklaracja każdego parametru jest następująca:

[tryb] nazwa : typ [=wartość-domyślna]

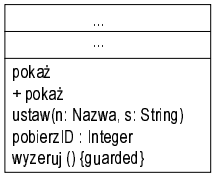
## Operacje i metody (2)

Właściwości:

- `isQuery` (wykonanie funkcji nie zmienia stanu systemu)
- `sequential` (wywołujące obiekty muszą zadbać, aby w danej chwili był tylko jeden przepływ sterowania, w przeciwnym razie nie jest zagwarantowane poprawne działanie)
- `guarded` (dzięki szeregowemu wywołaniu wszystkich chronionych operacji zapewnia poprawne działanie przy wielu przepływach sterowania)
- `concurrent` (operacja jest traktowana jako niepodzielna i musi być tak zaprojektowana, aby przebiegać poprawnie nawet gdy równolegle wywołano inne operacje (np. sequential) na tym samym obiekcie)

Tryby

- `in` (parametr wejściowy, nie może być modyfikowany)
- `out` (parametr wyjściowy; może być modyfikowany w celu przekazania informacji wywołującemu)
- `inout` (parametr wejściowy, może być modyfikowany)

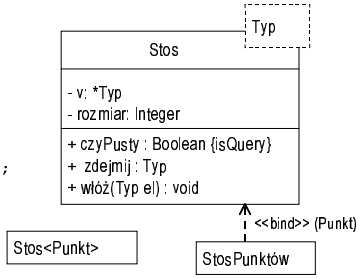





## Wzorce klas

- Wzorzec to byt sparametryzowany; nie można go używać bezpośrednio, trzeba najpierw utworzyć egzemplarz, poprzez dowiązanie parametru aktualnego do parametru formalnego

```
templete<class Typ>
class stos {
  Typ* v;
  Typ* p;
  int rozmiar;
public:
  bool czyPusty() const;
  Typ zdejmij();
  void włóż(Typ el);
};
```

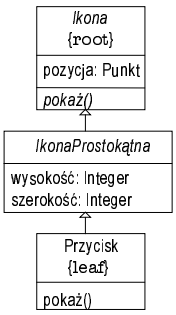





## Dziedziczenie

- **Klasy abstrakcyjne** - nie mogą mieć bezpośrednich egzemplarzy, oznaczone przez podanie nazwy *kursywą*
- Ograniczenie dziedziczenia w liściach (oznaczone przez `leaf`)
- Klasa bez przodków - korzeń (oznaczona przez `root`)
- **Operacje abstrakcyjne** - wymagamy, aby potomkowie zapewnili ich implementacje, nazwa - *kursywą*

Operacje abstrakcyjne odpowiadają czystym funkcjom wirtualnym w C++






## Związki: Uogólnienia

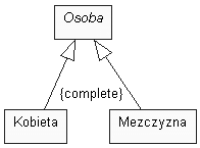
**Uogólnienie** - związek pomiędzy elementem ogólnym a pewnym specyficznym jego rodzajem ("jest-rodzajem"); potomek może prawie zawsze zastąpić przodka, ale nie odwrotnie

`implementation`, stereotyp standardowy, potomek dziedziczy całą implementację przodka, ale nie udostępnia jako publicznych jego interfejsów, ani ich nie realizuje (nie można zastąpić przodka takim potomkiem)

Stereotyp `implementation` służy do modelowania dziedziczenia prywatnego (np. z C++)

Ograniczenia standardowe:

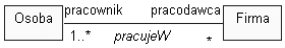
- `complete` - wszyscy potomkowie już zostali uwzględnieni, nie wolno dodawać żadnych nowych potomków
- `incomplete` - nie wszyscy potomkowie w uogólnieniu zostali w modelu uwzględnieni, można dodawać nowych






## Związki: Powiązania
(ang. association)

- Powiązanie (asocjacja) - **związek strukturalny**, który wskazuje, że obiekty jednego elementu są połączone z obiektami innego
- Istnieją 4 podstawowe dodatki do powiązań: nazwa, rola i liczebność przy każdym końcu oraz agregacja
- Rola - klasa biorąca udział w powiązaniu odgrywa w nim pewną określoną rolę
  - rola to swego rodzaju „oblicze"
  - klasa może odgrywać tą samą lub różne role w powiązaniach
- Liczebność - określa ile obiektów klasy ma być połączonych z każdym obiektem klasy znajdującej się po drugiej stronie powiązania; najczęstsze występujące liczebności:
  - **0..1** - opcjonalnie (zero lub jeden)
  - **1** - dokładnie jeden
  - **1..*** - co najmniej jeden
  - ***** - dowolnie wiele (zero lub więcej)
  - bardziej złożone liczebości określane są w postaci listy zakresów => **m..n, k..l**; np. 2..4, 6..8



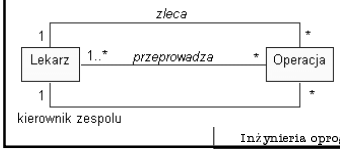



## Powiązania (2)

- Aby prawdłowo przedstawić liczebności powiązania należy "przeczytać" je w dwie strony:


Jeden obiekt Klasa_C jest powiązany z licz_D obiektów Klasa_D

Jeden obiekt Klasa_D jest powiązany z licz_C obiektów Klasa_C




- Inne dodatkowe właściwości to nawigacja, kwalifikacja i różne rodzaje agregacji

## Powiązania (3)

Zwykłe powiązanie (bez dodatków) umożliwia przechodzenie od obiektów jednej klasy do drugiej (w obie strony)

Można to ograniczyć poprzez określenie **kierunku nawigacji**

Wyspecyfikowanie kierunku nawigacji jest deklaracją efektywności przechodzenia wzdłuż powiązania, nie musi wykluczać ewentualnego przejścia przez powiązanie dotyczące innych (nie pokazanych) klas

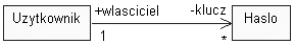
Możliwe zwykle dzięki przechowywaniu przez obiekt źródłowy odniesienia do obiektu docelowego

Można ograniczyć **widoczność** wzdłuż powiązania dla obiektów nie biorących w nim udziału

Poziomy widoczności końca powiązania (`public` - wartość domyślna, `protected` i `private`) są analogiczne jak w wypadku składników klas

`Private` - oznacza, że obiekty na tym końcu nie są dostępne poza obiektami biorącymi udział w powiązaniu

`Protected` - dodatkowo obejmuje to potomków klas





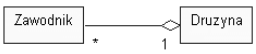

## Powiązania: agregacje

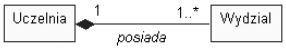
### Agregacja

- Związek "całość-część", w którym klasa reprezentuje większy element składający się z części; wyraża się zależnością "ma": obiekt-całość ma obiekty-części
- Agregacja w tej postaci ma jedynie znaczenie pojęciowe, nie zmienia znaczenia nawigacji, nie wyznacza zależności między czasem życia poszczególnych obiektów

### Agregacja całkowita

- Inne nazwy: zawieranie, kompozycja (ang. composition)
- Charakteryzuje się relacją wyłącznej własności oraz jednością czasu życia całości i części
- Części o nieustalonej liczebności mogą powstawać po utworzeniu całości, ale potem żyją i umierają razem z całością
- Oznacza to, iż część biorąca udział w agregacji całkowitej może należeć tylko do jednej całości
- Całość odpowiada za istnienie (tworzenie i niszczenie) części






## Powiązania (4)

**Kwalifikacja** - służy do rozwiązania problemu wyszukiwania, identyfikacji obiektów z drugiego końca powiązania;

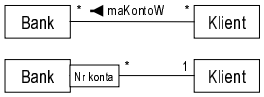
Wykorzystujemy kwalifikator, który jest argumentem powiązania (jego wartości charakteryzują jednoznacznie obiekty osiągalne przez powiązania) i służy do zmniejszenia liczności relacji w jej opisie

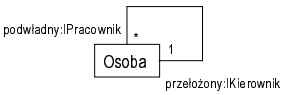
Kwalifikator może zawierać wiele atrybutów

Interfejs to zestaw operacji, które wyznaczają usługi oferowane przez klasę lub komponent, przy czym klasa może oferować wiele interfejsów

Przez powiązanie z klasą docelową klasa źródłowa może prezentować tylko część swego oblicza

Osiąga się to przez zastosowanie **specyfikatorów interfejsu**

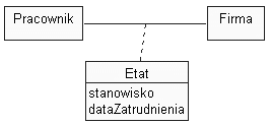





## Klasy powiązania

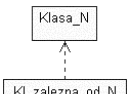
- **Klasa powiązania** - powiązanie między klasami może samo mieć właściwości
- Klasa powiązania jest bytem o własnościach zarówno klasy jak i powiązania
- Nie można jej dołączyć do więcej niż jednego powiązania, ale może ona dziedziczyć po innych klasach
- Klasa powiązania może być powiązana z innymi klasami






## Związki: Zależności

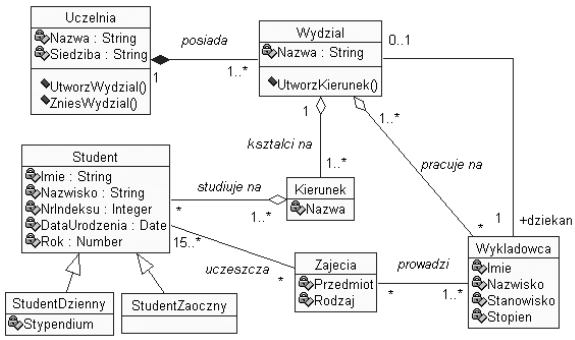
- Zależność - związek użycia; aby pokazać, że jedna klasa używa drugiej jako argumentu w sygnaturze operacji; zmiany w definicji el. mogą mieć wpływ na el. który go używa;




- Stereotypy dotyczące zależności między klasami a obiektami:
  - bind - źródło tworzy egzemplarz wzorca docelowego z użyciem danych parametrów strukturalnych
  - derive - źródło można wyznaczyć na podstawie celu (np. do atrybutów wyliczalnych na podstawie pozostałych)
  - **friend** - źródło ma szczególny dostęp do wnętrza celu (klasy zaprzyjaźnione w C++)
  - instanceOf - obiekt źródłowy jest egzemplarzem klasyfikatora docelowego
  - instantiate - źródło tworzy egzemplarze celu
  - powertype - cel jest typem potęgowym źródła. Typ potęgowy jest klasyfikatorem, którego wszystkie obiekty mają danego przodka
  - refine - źródło jest na doskonalszym poziomie abstrakcji (bardziej szczegółowym) niż cel
  - use - znaczenie bytu źródłowego zależy od znaczenia części publicznej celu



## Przykład diagramu klas - uczelnia

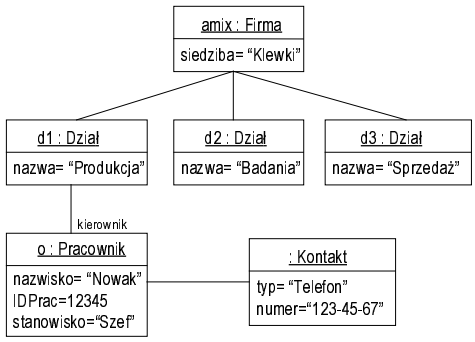


## Diagram obiektów

- Diagram klas obrazuje zbiór obiektów i ich związków w ustalonej chwili, zawiera:
  - obiekty
  - wiązania
- Diagram obiektów to w zasadzie egzemplarz diagramu klas lub statyczna część diagramu interakcji;
- Kładzie nacisk na konkretne (lub prototypowe) egzemplarze
- Celem jest głównie modelowanie struktur obiektowych - polega ono na robieniu rzutów obiektów systemu w ustalonych momentach czasu
- Szczególnie przydatne przy modelowaniu złożonych struktur danych



## Diagram obiektów - przykład






# Pakiety



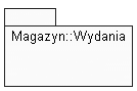
## Pakiety - wprowadzenie

- Złożone systemy mogą składać się z dużej liczby bytów (klas, interfejsów, ...)
- Aby móc efektywnie nimi zarządzać powstaje potrzeba grupowania tych bytów w pewne porcje
- W UML do realizacji tego zadania wykorzystywane są pakiety
- Umieszczenie bytów w pakietach pozwala na zarządzanie nimi jak grupami
- Dostęp do zawartości pakietów jest kontrolowany - kontrola widoczności (część składników może być widoczna, a część ukryta)
- Dobrze zaprojektowane pakiety składają się z podobnych znaczeniowo i razem się zmieniających bytów. Pakiety powinny być zatem powiązane ze sobą w sposób luźny, natomiast bardzo spójne wewnętrznie

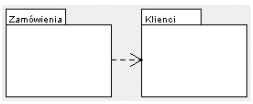


## Definicje

- Pakiet to uniwersalny mechanizm podziału składników na grupy w dowolnym celu; na diagramach przedstawiany jest jako skoroszyt z fiszką
- Każdy pakiet musi mieć przypisaną nazwę, która wyróżnia go spośród innnych pakietów (nazwa jest napisem)
- Jeżeli nazwa poprzedzona jest nazwą pakietu otaczającego mówimy o *nazwie ścieżkowej*, w przeciwnym przypadku mamy nazwę prostą








- Między dwoma elementami istnieje zależność, jeżeli zmiany definicji jednego elementu mogą spowodować zmiany drugiego; w odniesieniu do klas zależności obserwuje, gdy np. jedna klasa wysyła do drugiej komunikaty, jedna klasa zawiera drugą jako część danych, klasa jest parametrem operacji
- Zależność między pakietami (zawierającymi klasy) istnieje, gdy istnieje jakakolwiek zależność między dowolnymi klasami tych pakietów



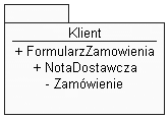
## Składniki pakietów

- Składnikami pakietu mogą być klasy, interfejsy, komponenty, węzły, operacje, przypadki użycia, diagramy oraz inne pakiety (w praktyce należy unikać zbytniego zagłębiania pakietów)
- Relacja własności to agregacja całkowita (byty zadeklarowane są w pakiecie; gdy pakiet jest niszczony giną wszystkie jego składniki; każdy byt jest własnością dokładnie jednego pakietu)
- Pakiet jest obszarem nazw - byty tego samego rodzaju muszą mieć unikatowe nazwy w jego ramach
- Podział na pakiety ma szczególne znaczenie, gdy nad jednym projektem pracuje wiele zespołów, z których każdy niezależnie opracowuje własne klasy - wykorzystanie pakietów przeciwdziała konfliktom nazw

W UML przyjmuje się, że istnieje nienazwany pakiet nadrzędny. W konsekwencji byty zdefiniowane na górze modelu muszą mieć unikatowe nazwy.

## Widoczność

- Widoczność składników pakietu określana jest w taki sam sposób jak w wypadku atrybutów i operacji klasy
- Jeżeli byt będący własnością pakietu jest *publiczny* (+), wówczas jest dostępny dla zawartości dowolnego pakietu importującego dany pakiet
- Byt *chroniony* (#) może być widziany jedynie przez potomków, natomiast byt *prywatny* (-) nie może być w ogóle widziany na zewnątrz pakietu w którym jest zadeklarowany
- Łącznie publiczne składniki pakietu stanowią jego interfejs
- Pakiet zaprzyjaźniony z innym (`<<friend>>`) może korzystać ze wszystkich bytów, niezależnie od ich widoczności





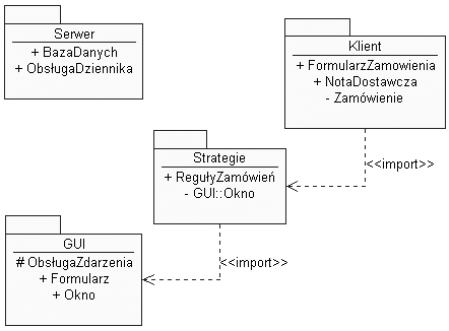

## Importowanie i eksportowanie

- Importowanie to jednostronne zezwolenie udzielone bytom jednego pakietu na dostęp do bytów drugiego pakietu
- W UML obrazowane jest za pomocą zależności uzupełnionej stereotypem `import` lub `access`, które wskazują, że pakiet źródłowy ma dostęp do zawartości pakietu docelowego
- `Import` rozszerza obszar nazw źródła o składniki celu (pojawia się zagrożenie sprzeczności nazw); `access` nie zmienia obszaru nazw i trzeba korzystać z nazw ścieżkowych
- Zależności `import` i `access` nie są przechodnie
- Część publiczną pakietu nazywamy jego eksportem

Pakiety zagnieżdżone pakietu mają dostęp do tego wszystkiego, co widzi otaczający je pakiet



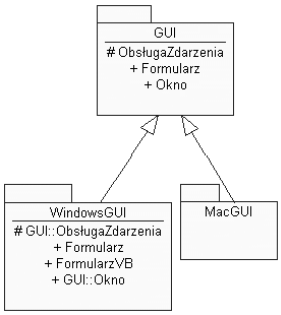
## Przykład importowania pakietów






## Uogólnianie pakietów

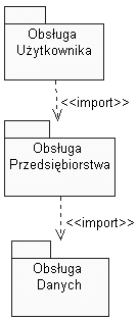
- Uogólnianie pakietów służy do budowania rodzin pakietów i jest bardzo podobne do uogólniania klas
- Pakiety, które dziedziczą po przodku mogą zastąpić byty bardziej ogólne (uszczegółowienie) lub dodawać nowe
- Pakiety biorące udział w uogólnieniach podlegają tej samej regule zastępstwa co klasy
- Nie wszystkie narzędzia (np. RR) umożliwiają uogólnianie pakietów






## Modelowanie grup bytów

- Organizowanie modelowanych bytów w grupy, które można nazwać i zarządzać nimi jak zbiorem
- Pakiety służą jedynie do systematyzowania składników modelu (nie mają tożsamości - nie mogą mieć egzemplarzy)
- Najczęściej zawierają elementy jednego rodzaju:
  - podział klas i ich powiązań uwzględnionych w perspektywie projektowej (zależność `import` posłużą do kontrolowania dostępu)
  - podobnie komponenty w perspektywie implementacyjnej
- Mogą obejmować byty różnego rodzaju:
  - w przypadku rozproszonego zespołu mogą być wykorzystane do zarządzania konfiguracjami (każdy z pakietów zawiera klasy i diagramy, które są pobierane i modyfikowane lokalnie)




Klasyczna architektura trójwarstwowa



## Modelowanie perspektyw architektonicznych

- Rozpatrując architekturę systemu informatycznego z różnych punktów widzenia potrzebujemy jeszcze większych porcji niż pakiety zawierające grupy pokrewnych bytów
- Perspektywa dotyczy organizacji i struktury systemu, przy czym kładzie się w niej nacisk na pewien szczególny aspekt systemu; najczęściej możliwy jest podział na niemal niezależne pakiety, które są właścicielami abstrakcji istotnych z danego punktu widzenia
- Kanoniczna dekompozycja najwyższego poziomu wyróżnia perspektywy: projektowa, procesowa, implementacyjna, wdrożeniowa i przyp. użycia
- Pakiety w roli perspektyw różnią się od fasad, gdyż są właścicielami swoich bytów, natomiast fasady zawierają jedynie odwołania do "cudzych" bytów. Każdy byt może należeć do dokładnie jednego pakietu, ale wiele fasad może się jednocześnie do niego odwoływać