

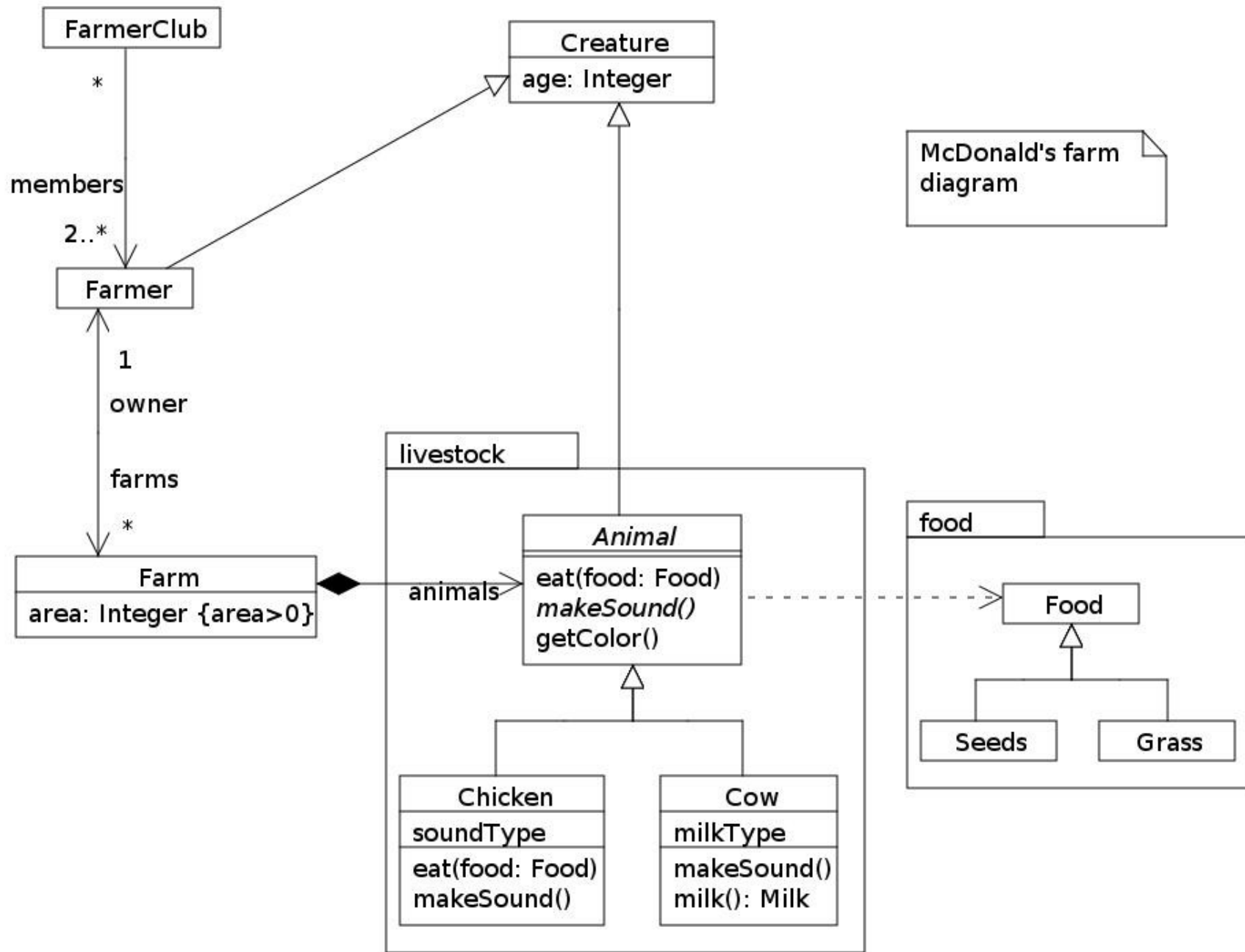
# Diagramy klas UML

Mateusz Kobos 17.03.2011

# UML

- UML to najpopularniejszy sposób graficznego opisywania budowy programu obiektowego.
- UML = Unified Modeling Language – powstał z połączenia kilku wcześniejszych języków.
- UML jest opisany w standardzie UML 2.3, jednak w praktyce korzysta się również z pewnych nie ujętych w standardzie (za to popularnych) konwencji.

# UML – przykładowy diagram klas

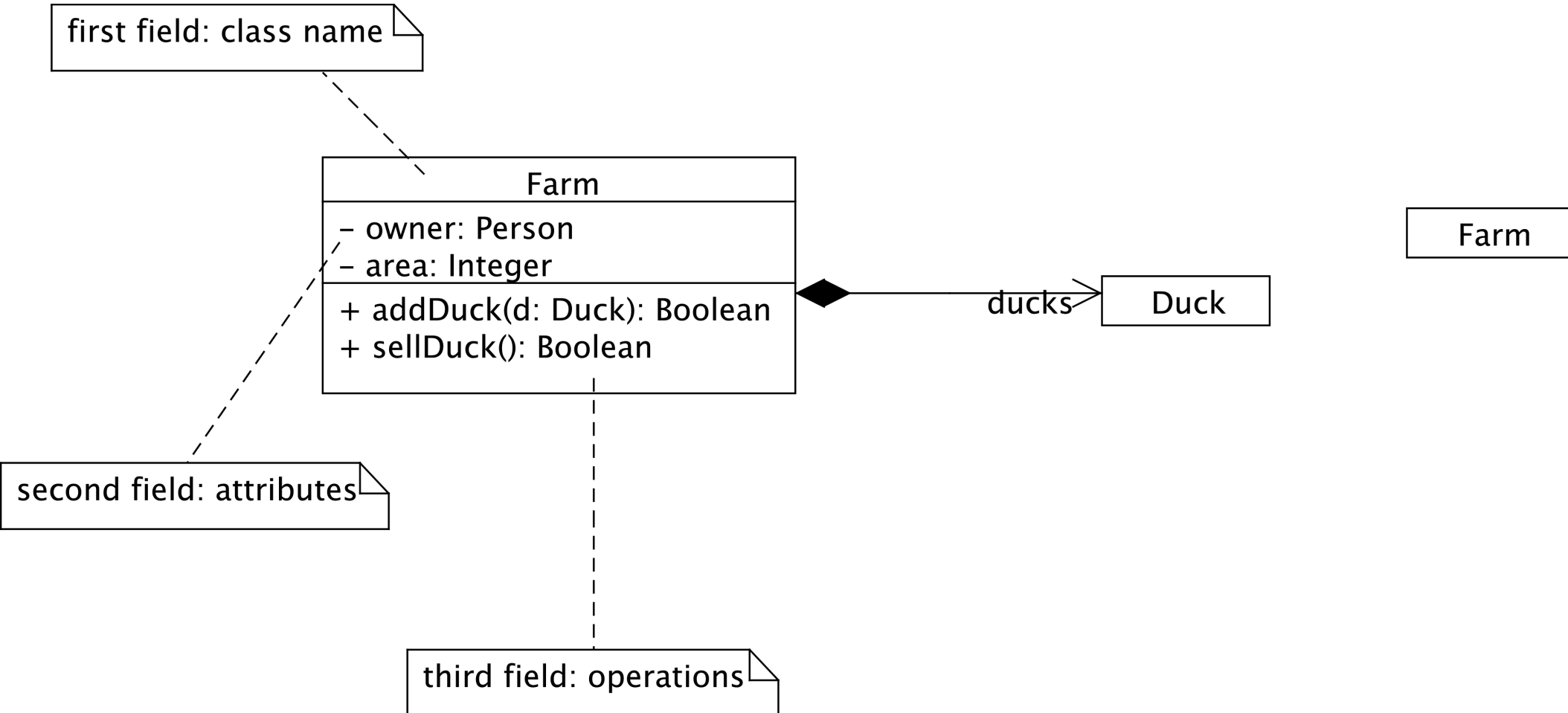


# Podstawowe informacje

- Nie jest zdefiniowane, jak diagram UML przekłada się na określony język programowania – przy implementacji zawsze zachodzi potrzeba interpretacji diagramu przez człowieka.
- Szczegółowość diagramów zależy od fazy tworzenia oprogramowania
  - przy analizie problemu – diagramy ogólne,
  - przy tworzeniu dokumentacji technicznej – diagramy bardziej szczegółowe.
- Na jednym diagramie można używać elementów należących do różnych typów diagramów.
- Najczęściej używane diagramy UML to diagramy klas, które służą do reprezentowania statycznej struktury programu.

# Podstawowe informacje – ukrywanie informacji

- Podstawowa zasada UML: każda informacja na diagramie może zostać pominięta
  - Np. klasa **Farm** w wersji rozszerzonej i w wersji z ukrytą większością informacji

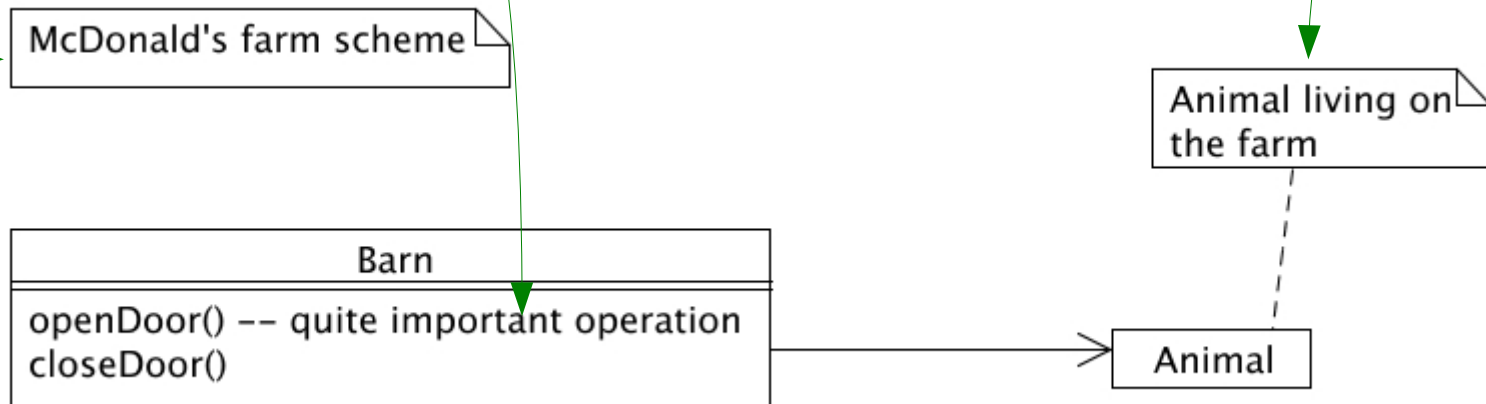


# Podstawowe informacje – ukrywanie informacji

- Wniosek: na podstawie nieobecności jakiegoś elementu na diagramie nie można wyciągać wniosku o obecności lub nieobecności tego elementu w modelowanym systemie
  - np. jeśli brakuje oznaczenia krotności danej relacji, to nie można wywnioskować jaką krotność może mieć dana relacja.
- Nawet jeśli w standardzie UML jest określona wartość domyślna (jak krotność =1 dla atrybutów), a danej informacji nie ma na diagramie, to może być tak dlatego, że prawdziwa wartość jest ukryta (a nie równa wartości domyślnej).
- Mimo to istnieją pewne konwencje jak ta, że atrybuty wielowartościowe to zbiory.

# Uwagi i komentarze

- Uwaga/komentarz mogą być:
  - oddzielnym obiektem
  - oddzielnym obiektem połączonym przerywaną linią z komentowanym elementem
  - komentarzem „in-line”



# Zależności (ang. *dependencies*)

- Jeśli klasa **A** zależy od klasy **B**, oznacza to, że **A** wykorzystuje **B** lub wie o istnieniu **B**. Wynika stąd, że zmiana **B** pociągnie za sobą prawdopodobnie zmianę **A**.
- Zależność oznacza się za pomocą strzałki rysowanej przerywaną linią
- Np. **Farmer** zależy od **Animal**, **Animal** zależy od **Food**



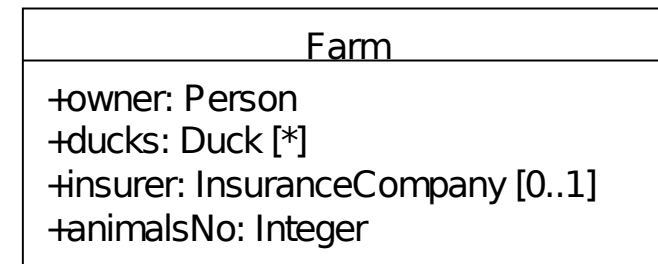
- W praktyce to, że klasa **Animal** zależy od klasy **Food** może np. po prostu oznaczać, że klasa **Animal** zawiera metodę **eat()**, której jednym z parametrów jest parametr typu **Food**. Zgodnie z definicją, możemy zauważyć, że w tym przypadku, jeśli zmieni się np. jakaś publiczna metoda klasy **Food**, to pociągnie to za sobą prawdopodobnie zmianę w metodzie **eat()** klasy **Animal**.



# Cechy (ang. *properties*)

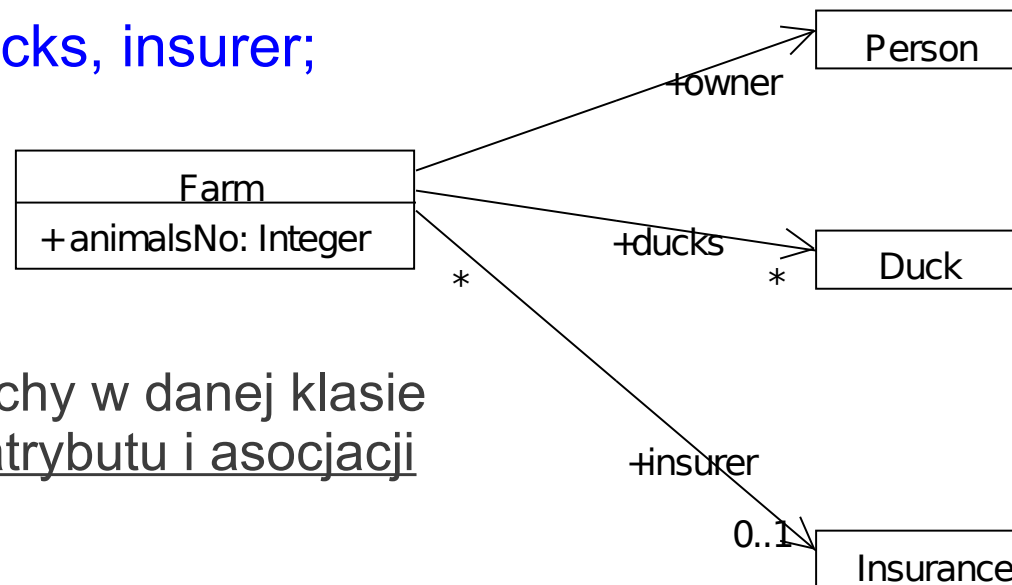
- Cecha odpowiada polu klasy (lub właściwości (ang. *property*) z języka C#) i ew. prostym metodom służącym do obsługi tego pola (typu get..., set..., add..., remove...)
- Są 2 alternatywne sposoby reprezentacji cechy:

- atrybut (ang. *attribute*) – umieszczamy w 2. od góry prostokącie diagramu klasy, np.  
atrybuty: **owner, ducks, insurer, animalsNo**



- asocjacja/powiązanie (ang. *association*) – reprezentowane przez ciągłe strzałki, np.

- asocjacje: **owner, ducks, insurer;**
- atrybuty: **animalsNo**



Uwaga: do reprezentacji danej cechy w danej klasie nie należy używać jednocześnie atrybutu i asocjacji (tj. należy wybrać jedno z nich).

# Cechy - atrybuty

Farm
+owner: Person
+ducks: Duck [*]
+insurer: InsuranceCompany [0..1]
+animalsNo: Integer

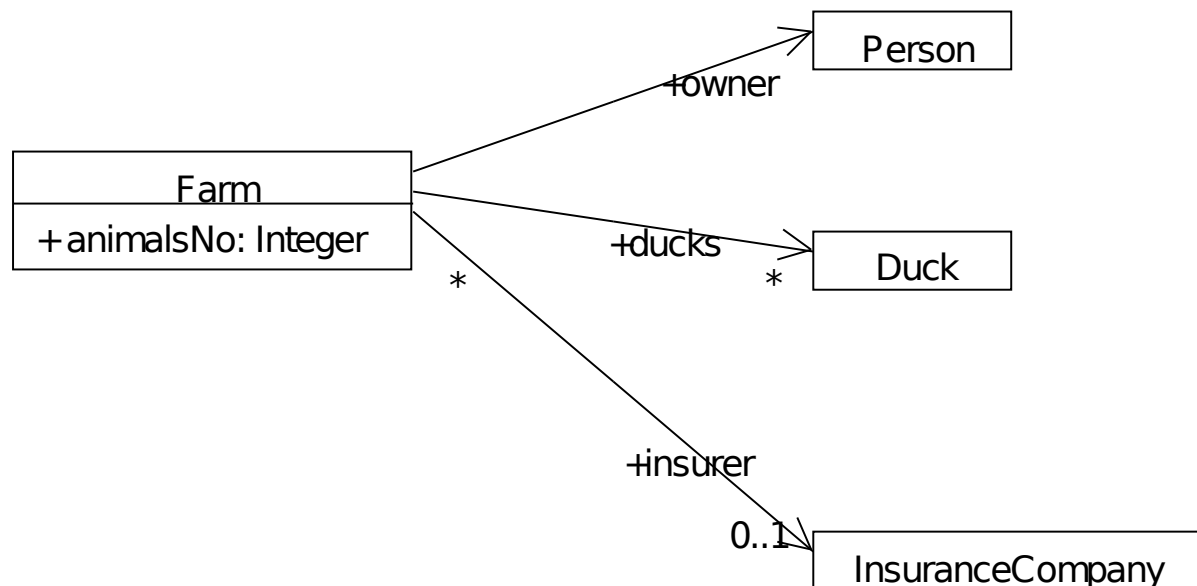
- Ogólna postać atrybutu (umieszczamy w 2. od góry prostokącie diagramu klasy):

visibility name: type [multiplicity] = default {property-string}

- np. - *surname: String [1] = "Doe" {readOnly}*
- wszystkie elementy oprócz **name** są opcjonalne
- elementy:
  - **visibility**: public (+), package (~), protected (#), private (-)
  - **name**: odpowiada nazwie pola klasy
  - **type**: odpowiada typowi pola klasy
  - **multiplicity**: krotność - liczba elementów np. 1, 3..5, \*
  - **default**: wartość domyślna, którą przyjmuje nowy obiekt
  - **property-string**: pozwala określić dodatkowe właściwości atrybutu i ograniczenia nakładane na atrybut
    - np. {readOnly} - atrybut nie może być modyfikowany po ustawieniu początkowej wartości

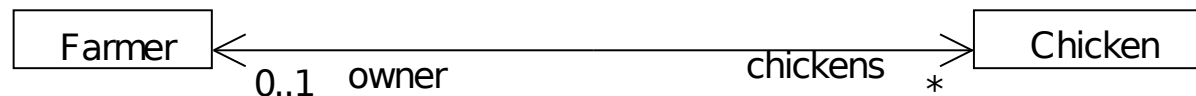
# Cechy – asocjacje

- Do oznaczenia asocjacji wykorzystuje się strzałki rysowane ciągłą linią
- Nazwa zawieranego elementu i jego krotność znajduje się na końcu z grotem
- Ważniejsze spostrzeżenia:
  - analogiczne oznaczenia jak przy atrybutach stosuje się w asocjacjach;
  - w przeciwieństwie do atrybutu, krotność może występować po dwóch stronach relacji.

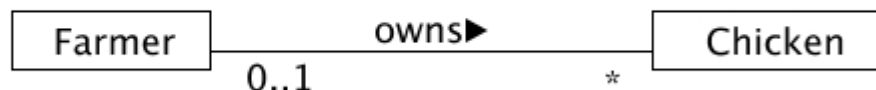


# Cechy – asocjacje

- Asocjacje mogą być dwukierunkowe – oznacza to parę cech połączonych ze sobą na zasadzie odwrotności. Np.:



- Poruszając się po strukturze danych: zaczynając od określonego **kurczaka**, można znaleźć jego **właściciela**. Następnie, spośród **kurczaków** należących do danego **farmera** można znaleźć określonego **kurczaka** tym samym wracając do punktu wyjścia.
  - W praktyce oznacza to, że klasa **Farmer** ma pole **chickens** a klasa **Chicken** ma pole **owner**
- Asocjacje mogą mieć przypisany czasownik wraz z kierunkiem asocjacji. Czasownik opisuje relację. Np.:




- **farmer** jest właścicielem **kurczaków**

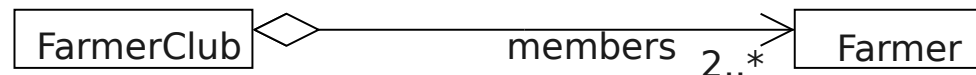
# Cechy - krotność (ang. *multiplicity*)

- Przykłady określeń krotności, które są używane do opisu własności:
  - 1 – dokładnie jeden obiekt
  - 3..5 – od 3 do 5 obiektów
  - \* - dowolna liczba obiektów (zero lub więcej)
  - 2..\* - 2 i więcej obiektów
- Domyślna wartość krotności dla atrybutu: 1
- Domyślnie, elementy związane z cechą o krotności większej niż 1 tworzą zbiór (tzn. nie są uporządkowane, elementy nie powtarzają się).
  - By pokazać, że elementy są uporządkowane, można użyć property-string: {ordered}
  - By pokazać, że elementy mogą się powtarzać, można użyć property-string: {nonunique}


# Agregacja i zawieranie

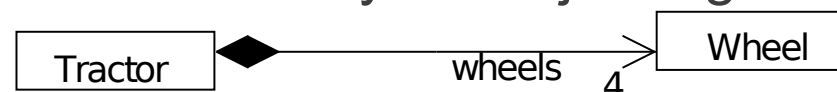
- Agregacja (ang. *aggregation*) – służy do pokazania relacji „A jest właścicielem B”

- Oznaczana strzałką z białym rombem 
- Ta sama instancja klasy B może być składnikiem wielu innych instancji klasy A
- Np. jeden farmer może należeć do wielu klubów



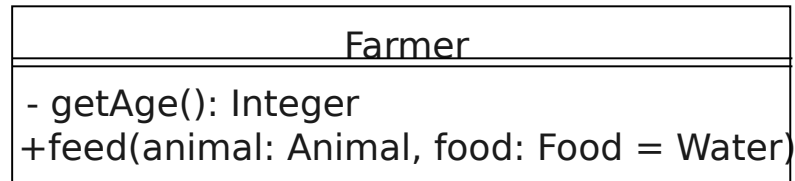
- Uwaga: agregacja jest pojęciem b. zbliżonym do asocjacji, dlatego w [D] zaleca się nie stosować agregacji
- Zawieranie (ang. *composition*) – służy do pokazania relacji „B jest częścią A”

- Oznaczana strzałką z czarnym rombem 
- Jedna instancja klasy B może być składnikiem tylko jednej instancji klasy A
- Np. jedno koło może należeć tylko do jednego ciągnika.



# Operacje (ang. *operations*)

- Operacja odpowiada metodzie klasy
- Operacje umieszczamy w 3. od góry prostokącie diagramu klasy, np:



# Operacje

Farmer
- getAge(): Integer +feed(animal: Animal, food: Food = Water)

- Ogólna postać operacji:

visibility name (parameter-list) : return-type {property-string}

– np. **+ feed(animal: Animal, quantity: int): Boolean**

- wszystkie elementy oprócz **name** są opcjonalne

- elementy:

- **visibility**: public (**+**), package (**~**), protected (**#**), private (**-**)
- **name**: odpowiada nazwie metody klasy
- **parameter-list**: lista parametrów. Każdy z parametrów ma postać:

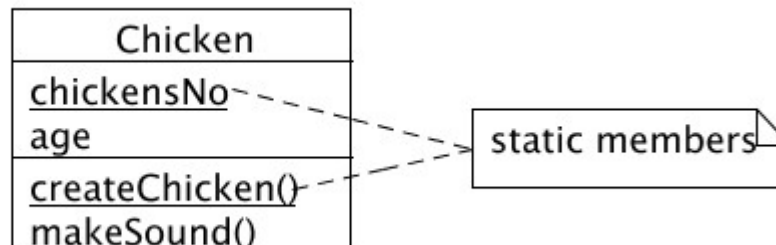
direction name : type = default , gdzie

- wszystkie elementy oprócz **name** są opcjonalne
- **name, type, default**: analogicznie jak przy atrybutach
- **direction**: mówi, czy parametr jest typu input (**in**), output (**out**), input&output (**inout**). Domyślna wartość to **in**.
- **return-type**: typ zwracanej wartości
- **property-string**: pozwala określić dodatkowe właściwości operacji



# Własności operacji i atrybutów

- Operacje i atrybuty, które są statyczne są oznaczone przez podkreślenie
  - np. atrybut `chickensNo` i operacja `createChicken` są statyczne



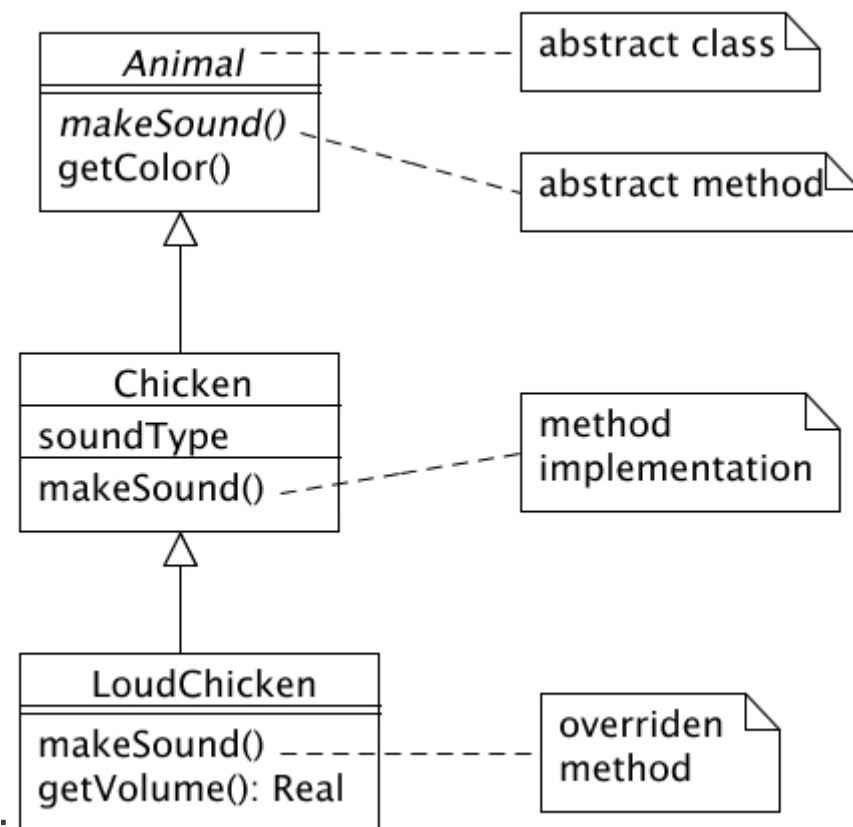
- Uwaga: typy prymitywne/podstawowe (np. `int`, `double`, `String`) nie są zdefiniowane w standardzie UML – standardowo przyjmuje się że pochodzą one z języka, w którym będzie implementowany diagram.
- Nie ma oddzielnej notacji do określania, że dana cecha jest tablicą. Jednakże, gdy cecha ma stałą liczbę obiektów np. atrybut postaci `ducks: Duck[7]`, lub asocjacja z przypisaną krotnością 7, to stanowi sugestię, że dana cecha ma zostać zaimplementowana jako tablica 7-elementowa. Analogicznie przy oznaczeniu `duck: Duck[3..*] {ordered}` mamy sugestię, że dana cecha powinna zostać zaimplementowana jako lista (lub np. `ArrayList`) o minimum 3 elementach.

# Dziedziczenie i klasy abstrakcyjne

- Dziedziczenie jest oznaczone za pomocą strzałki zakończonej białym grotem.
- Konwencja: jeśli klasa **B** dziedziczy po klasie **A**, to **B** powinna znajdować się poniżej klasy **A** na diagramie (jeśli to możliwe)
- Klasa abstrakcyjna jest oznaczona przez wyróżnienie nazwy kursywą. Abstrakcyjny atrybut i abstrakcyjna operacja również są oznaczone przez wyróżnienie nazwy kursywą

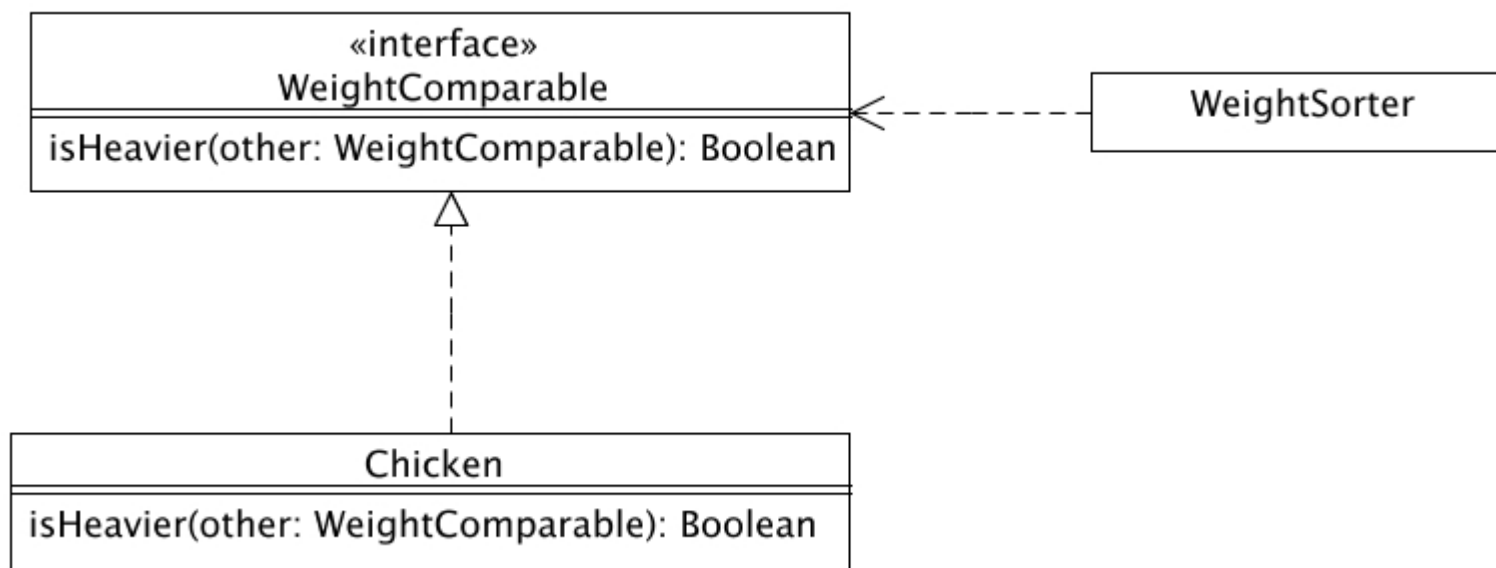
Np. **Animal** jest klasą abstrakcyjną. **makeSound()** w klasie **Animal** jest operacją abstrakcyjną. **makeSound()** w klasie **Chicken** jest zaimplementowaną wersją operacji **makeSound()** z klasy **Animal**. **makeSound()** w klasie **LoudChicken** jest nadpisaną (ang. *override*) wersją operacji **makeSound()** z klasy **Chicken**.

Uwaga: w podklasie nie należy powtarzać nazwy pola, które zostało zdefiniowane w nadklasie. Analogicznie, w podklasie nie należy powtarzać nazwy operacji, która została zdefiniowana w nadklasie, o ile podklasa nie nadpisuje tej operacji.



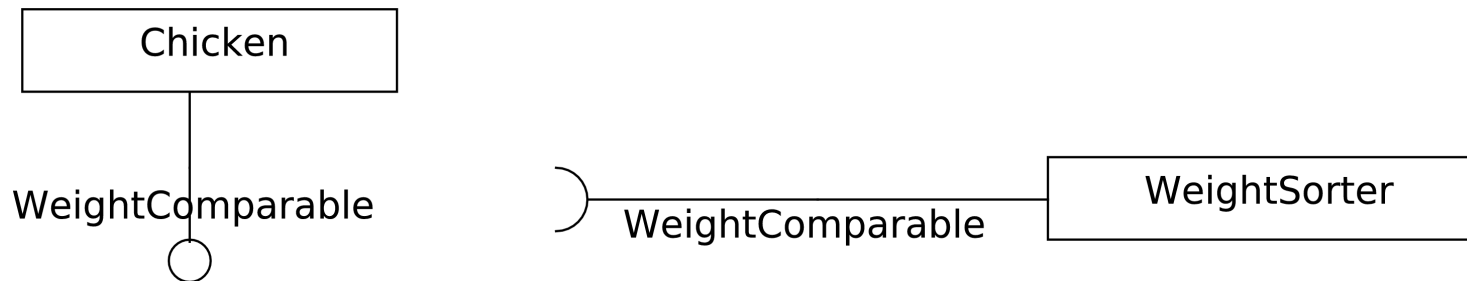
# Interfejsy (ang. *interfaces*)

- Interfejs oznacza się przez użycie słowa kluczowego `<<interface>>`
- Implementację/rozszerzenie interfejsu oznacza się za pomocą strzałki rysowanej przerywaną linią. Strzałka ta ma biały grot. W klasie implementującej interfejs znajdują się nazwy operacji z interfejsu, ponieważ klasa ta implementuje (czyli jakby nadpisuje) te operacje.
- Np. klasa `Chicken` implementuje interfejs `WeightComparable`. Klasa `WeightSorter` wykorzystuje interfejs `WeightComparable`.

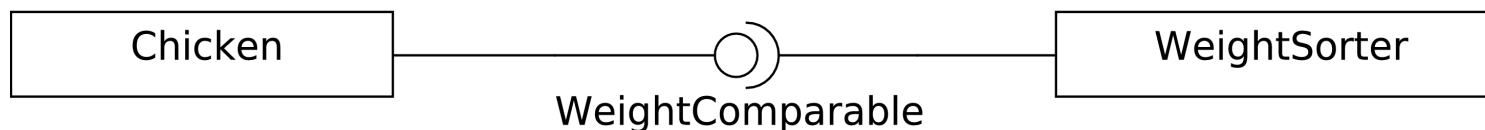


# Interfejsy – notacja gniazd

- Do opisu interfejsów implementowanych przez klasę można użyć też notacji gniazd (zwaną też notacją „z lizakami”). Notacja ta ukrywa szczegółową budowę interfejsu, za to podkreśla informację na temat udostępnianych i wymaganych przez klasę interfejsów.
- Np. odpowiednik diagramu z poprzedniego slajdu – klasa **Chicken** implementuje (udostępnia) interfejs **WeightComparable** a klasa **WeightSorter** wykorzystuje interfejs (wymaga interfejsu) **WeightComparable**



- Co można również przedstawić tak:

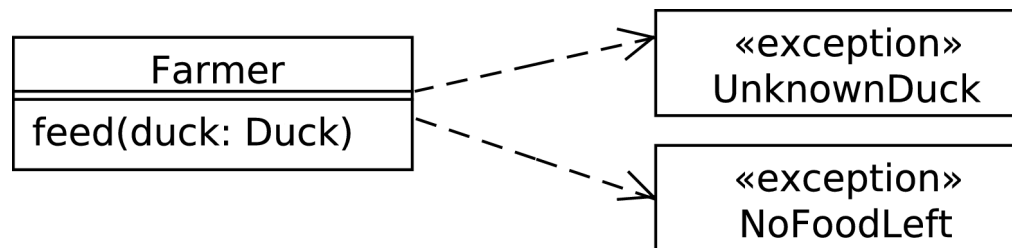


# Mechanizmy rozszerzające UML

- Elementy zdefiniowane w UML nie zawsze wystarczają do przedstawienia wszystkich ważnych niuansów rozważanego systemu. Dlatego wprowadzono mechanizmy kontrolowanego rozszerzania języka przez użytkownika, a wśród nich:
  - stereotypy (ang. *stereotypes*) – czasami stosuje się też nazwę „słowa kluczowe” (ang. *keywords*),
  - ograniczenia (ang. *constraints*).

# Stereotypy (ang. *stereotypes*)

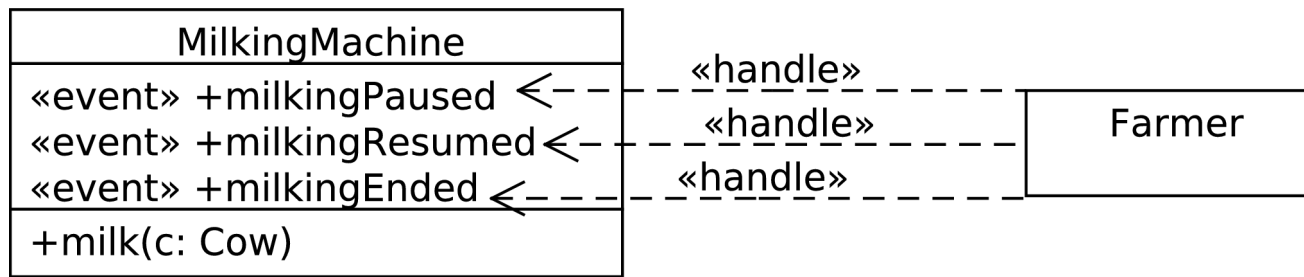
- Stereotyp służy do stworzenia nowego rodzaju obiektu ma podstawie obiektu zdefiniowanego już w standardzie UML.
- Najprostszym sposobem dodania nowego stereotypu jest dodanie nazwy stereotypu **<<nazwa stereotypu>>** przy nazwie obiektu.
- W standardzie UML jest zdefiniowanych kilkadziesiąt stereotypów np. **<<interface>>**.
- Np. gdy chcemy wśród klas wyróżnić wyjątki (bo mają one zupełnie inne zastosowanie niż reszta klas), można to zrobić dodając nowy stereotyp **<<exception>>**:



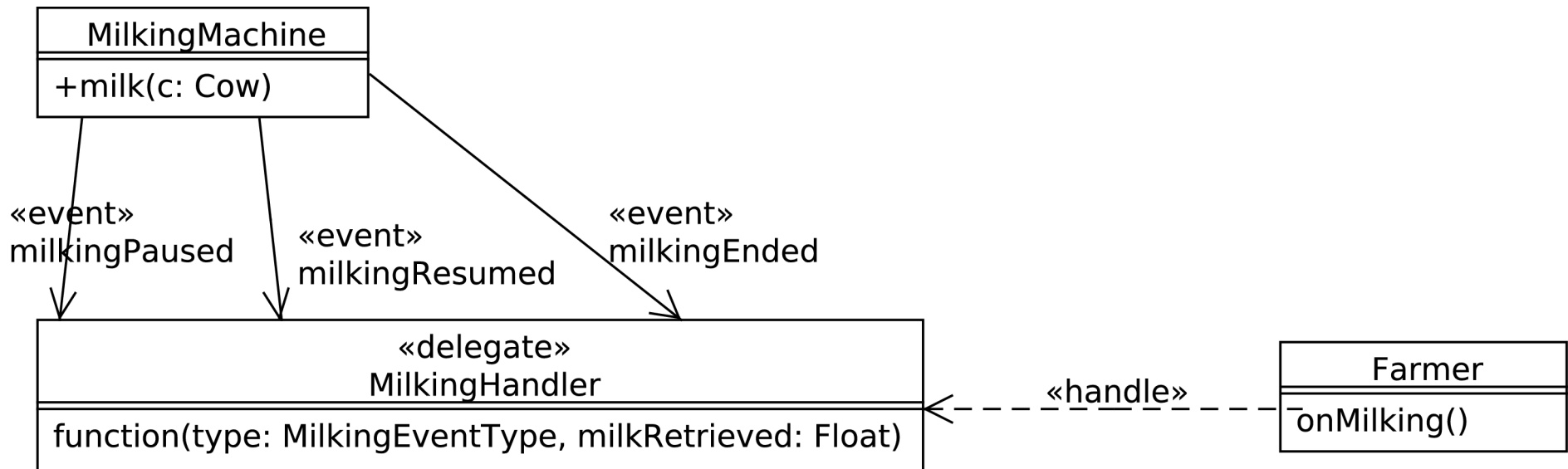
- Uwaga: wprowadzanie własnych oznaczeń do diagramów zmniejsza „natychmiastową” ich czytelność. Dlatego z możliwości tworzenia nowych stereotypów należy korzystać tylko wtedy, gdy rzeczywiście jest taka potrzeba.

# Stereotypy – przykład z event

- Za pomocą zdefiniowanych przez użytkownika stereotypów można też reprezentować elementy, które nie są zdefiniowane w standardzie UML. Takim elementem jest np. zdarzenie (ang. *event*) występujące w C#.
- Zdarzenia możemy reprezentować np. wprowadzając stereotypy `<<event>>`, `<<handle>>` i używając ich w sposób następujący:

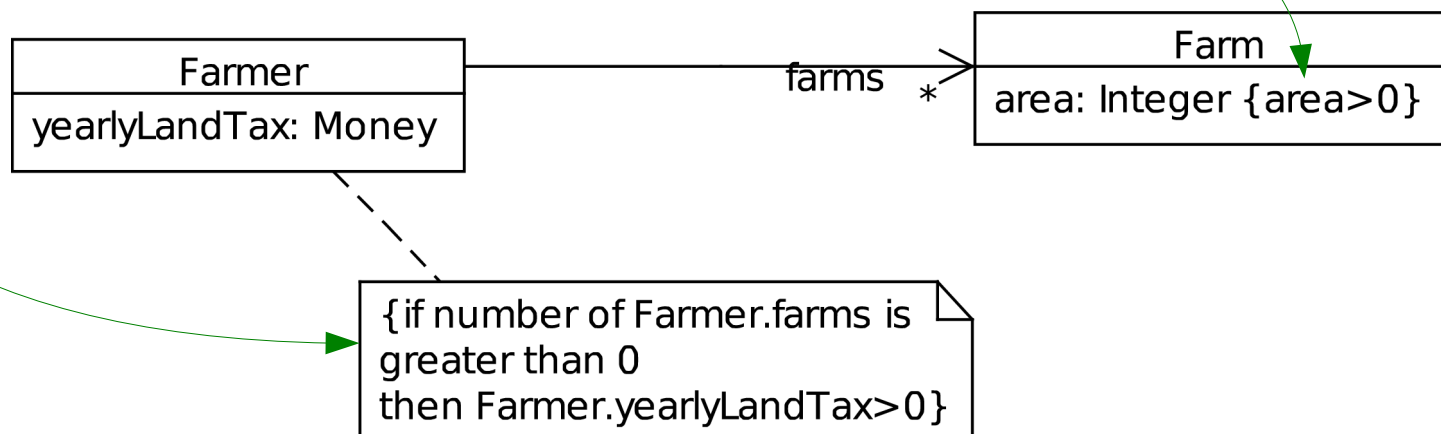


- Zdarzenie można reprezentować też w sposób bardziej rozbudowany np. wprowadzając stereotypy `<<event>>`, `<<handle>>`, `<<delegate>>`:



# Ograniczenia (ang. *constraints*)

- Tworzenie asocjacji, ustalanie hierarchii dziedziczenia, ustalanie krotności itd. może być interpretowane jako dodawanie ograniczeń w konstrukcji programu. Jednak nie wszystkie ograniczenia można wyrazić za pomocą symboli UML, dlatego ograniczenia można przedstawiać również w sposób opisowy, przedstawiony poniżej.
- Ograniczenia można umieszczać:
  - w uwadze
  - jako „in-line”



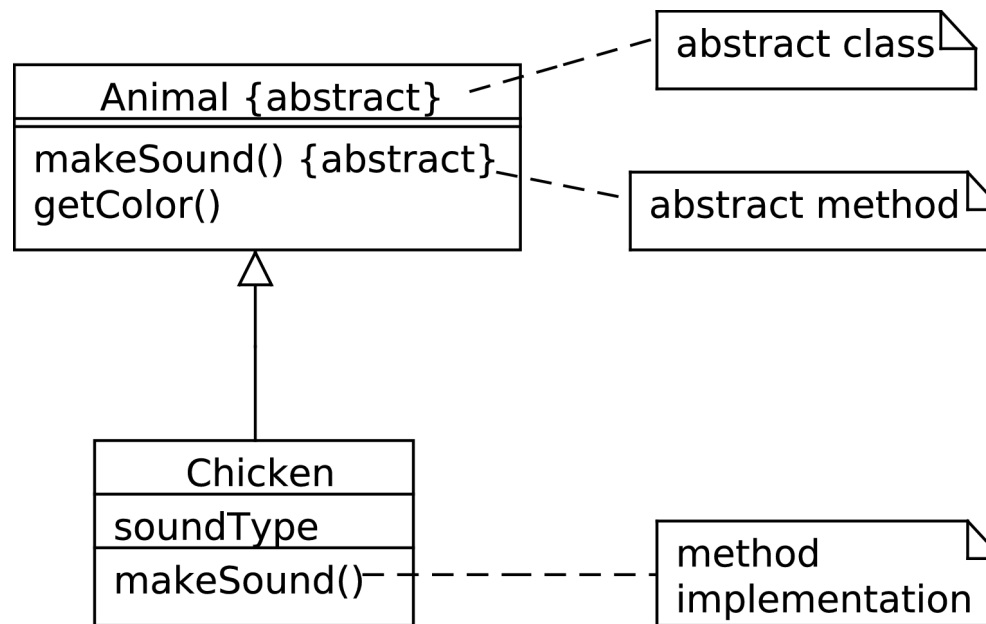


# Ograniczenia (ang. *constraints*) cd.

- Ogólna postać ograniczenia: `{name: description}`
  - np. `{disallow multilocation: person must not be in more than one place at once}`
- elementy:
  - **name** (opcjonalne) – nazwa ograniczenia
  - **description** – opis ograniczenia w języku naturalnym (opcja zalecana) lub w języku programowania lub w UML-owym języku OCL
- UML zawiera pewne zdefiniowane ograniczenia np. dla cech: **readOnly**, **nonunique**, **ordered**
- W języku programowania ograniczeniom odpowiadałyby asercje (w języku C#: metoda ***Debug.assert***, w języku Java i C++: ***assert***) lub warunki, których niespełnienie skutkowałoby wyrzuceniem wyjątku

# Ograniczenia – klasy abstrakcyjne

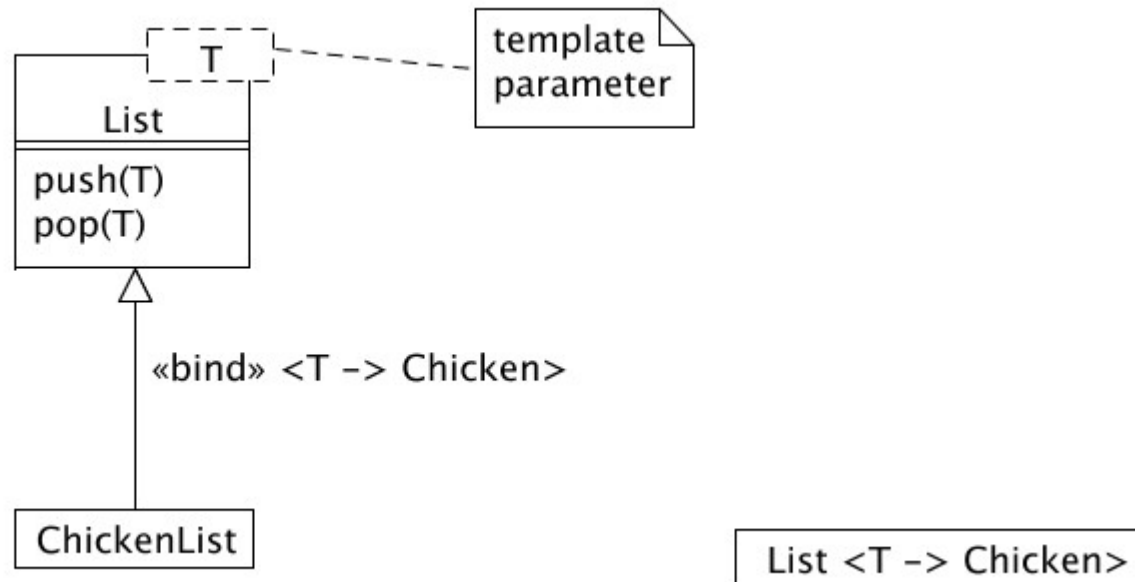
- Ograniczeń można też użyć do oznaczenia klas, operacji i atrybutów abstrakcyjnych:



- Dodajemy tutaj ograniczenie **{abstract}**, które jest skróconą wersją **{abstract=True}**.
- Ta konwencja jest jednak o wiele mniej popularna niż używanie kursywy do oznaczania nazw elementów abstrakcyjnych.

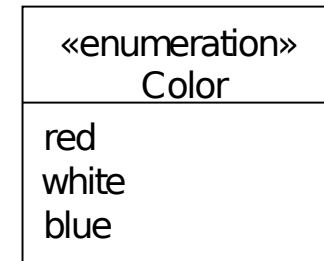
# Wzorce (ang. *templates*)

- Wzorzec (szablon/klasa generyczna/template) jest przedstawiany poprzez umieszczenie parametrów wzorca w prostokącie w prawym górnym rogu klasy. Prostokąt jest rysowany przerywaną linią.
- konkretyzacja (wiązananie wzorca) może być przedstawiana na 2 sposoby: jawny i niejawnym
  - Np. **ChickenList** jest jawną konkretyzacją wzorca **List**, gdzie **Chicken** jest parametrem wzorca. Klasa znajdująca się po prawej stronie jest konkretyzacją niejawną.

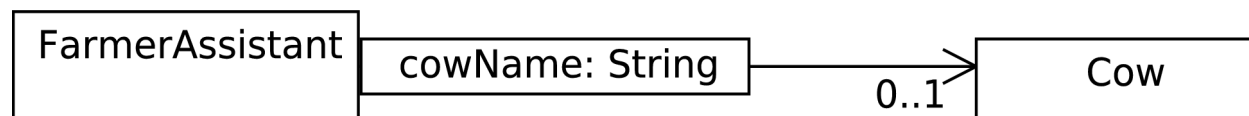


# Inne

- Typy wyliczeniowe (ang. *enumerations*):

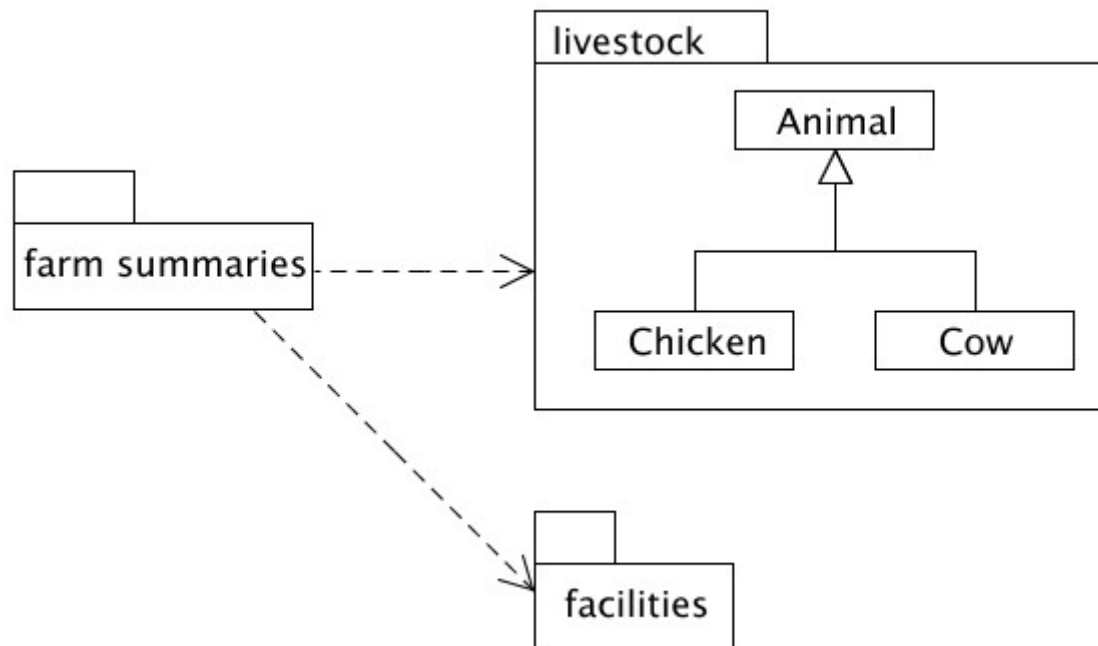


- Asocjacje kwalifikowane (ang. *qualified associations*) – odpowiednik struktur słownikowych, hashmap. W tych strukturach określonemu kluczowi przypisujemy pewną wartość/pewne wartości. Krotność podana na diagramie odnosi się do liczby wartości związanych z danym kluczem.
  - Np. po podaniu **imienia krowy** (klucz), **asystent** zwraca **krowę** (wartość), która jest do przypisana do imienia (ew. zwraca pustą referencję, jeśli **krowa** o takim **imieniu** nie istnieje). Dane **imię** może być przypisane co najwyżej jednej **krowie**.



# Diagramy pakietów (ang. *package diagrams*)

- Diagramy pakietów reprezentują pakiety/przestrzenie nazw (ang. *namespaces*) i zależności między nimi
- Np. pakiet **farm summaries** zależy od pakietów **livestock** i **facilities**. Pakiet **livestock** zawiera co najmniej 3 klasy: **Animal**, **Chicken**, **Cow**.



# Uwagi końcowe

- UML jest obszernym standardem i większość ludzi używa tylko małego podzbioru UML
- Diagram przede wszystkim powinien być czytelny i powinien przekazywać główny zamysł projektowy autora. Wynikają z tego następujące zalecenia.
  - Przy tworzeniu diagramów nie należy starać się używać wszystkich możliwych oznaczeń i pokazywać wszystkich możliwych zależności.
  - Nie należy się starać na jednym diagramie zawrzeć wszystkich klas i zależności występujących w programie – najlepiej przedstawiać budowę programu „od ogółu do szczegółu”, czyli np. stworzyć jeden diagram ogólny programu o małej szczegółowości a potem kolejne diagramy, bardziej szczegółowo opisujące poszczególne elementy programu.
- Na diagramie powinno być b. mało obiektów, które nie są połączone z innymi (np. strzałką) – diagram powinien przedstawiać spójną strukturę całego programu.

# Polecane programy i literatura

- Polecane programy do tworzenia diagramów UML:
  - UMLet (posłużył do stworzenia diagramów umieszczonych na tych slajdach)
  - BOUML
  - Dia
  - MS Visio
  - programy do tworzenia grafiki wektorowej (np. Inkscape)
- Literatura:
  - [Dpl] Fowler, *UML w kropelce, wersja 2.0*, 2005 (opis UML 2.0)
  - [D] Fowler, *UML distilled, 3<sup>rd</sup> ed.*, 2003 (opis UML 2.0)
  - [N] Pilone, Pitman, *UML 2.0 in a Nutshell*, 2005 (opis UML 2.0)
  - [G] Booch, Rumbaugh, Jacobson, *The Unified Modeling Language user guide 2<sup>nd</sup> ed*, 2005 (opis UML 2.0)
  - [Gpl] Booch, Rumbaugh, Jacobson, *UML – przewodnik użytkownika*, 2001 (opis UML 1.0)
  - [R] Rumbaugh, Jacobson, Booch, *The Unified Modeling Language reference manual*, 1999 (opis UML 1.0)