Model-Driven Engineering

Radosław Klimek

2015-22

http://home.agh.edu.pl/rklimek

Język OCL

(Object Constraint Language)

Krótka charakterystyka

- Język formalny używany do opisywania wyrażeń w języku UML
- Wyrażenia określają niezmienne warunki, które musi spełnić modelowany system
- Wyrażenia określają także operacje albo akcje w taki sposób, że po wykonaniu zmień stan systemu Służy do określania ograniczeń specyficznych dla aplikacji
- Został opracowany w celu uniknięcia niejednoznaczności podczas opisu dodatkowych ograniczeń modelu
- Wyrażenia OCL nie mają wpływu na model
- Nie jest to język programowanie z związku z tym nie jest możliwe napisanie logiki programu lub kontrolować przepływ
- Każde wyrażenie musi mieć określony typ oraz muszą być zgodne z regułami
- Na przykład nie jest dozwolone porównanie zmiennej typu Integer ze zmienną typu String

Ograniczenie

Ograniczenie – to restrykcja nałożona na jeden lub więcej elementów modelu lub system.

- Ograniczenie jest sformułowane na poziomie klas, ale jego semantyka jest stosowana na poziomie obiektów.
- Relacje między elementami, jak i same elementy, mogą mieć sprawdzalną prawidłowość za pomocą ograniczeń.
- Ograniczenia umożliwiają zapisywanie wyrażeń, które muszą być spełnione, aby model był prawidłowy.

Składnia

Ograniczenia mają następującą składnię:

{ ograniczenie tekstowe }

Ograniczenie tekstowe może być prostym wyrażeniem, pełnym zdaniem, lub może być zapisane za pomocą formalnej składni ograniczenia.

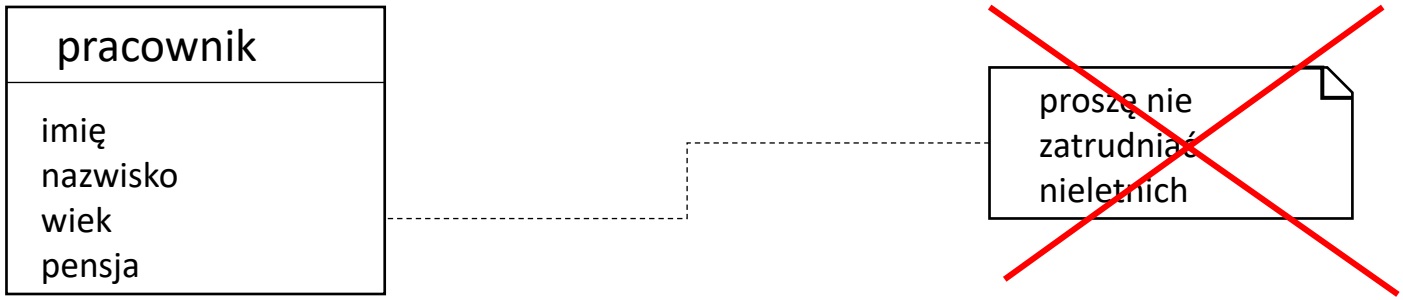
UML definiuje formalną gramatykę **OCL** (*Object Constraint Language*).
OCL jest to formalny język służący do wyrażania ograniczeń w **UML** .

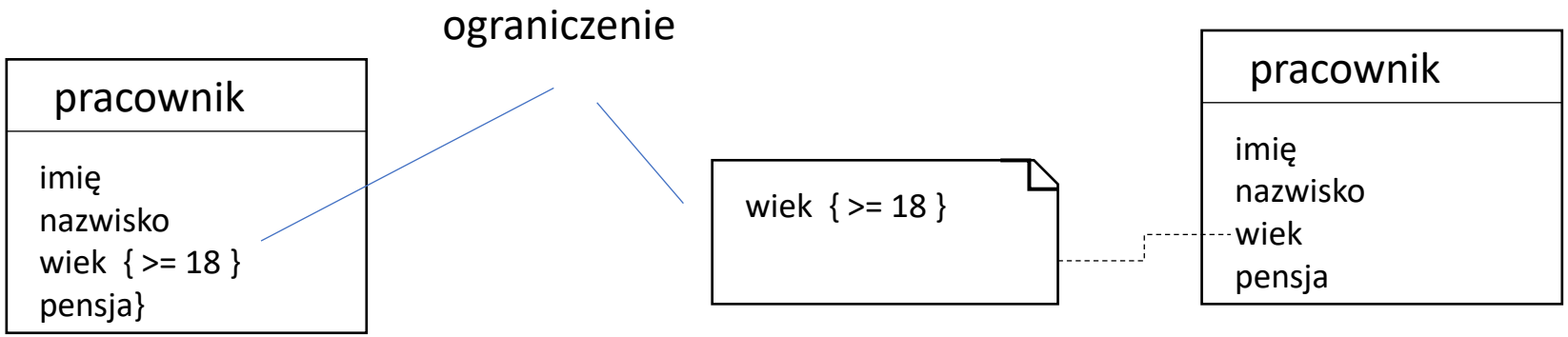
Ograniczenia są zawarte wewnątrz nawiasów **{ }** i umieszczane za elementem w klasie, lub poza klasą. Z reguły są umieszczane w komentarzu.
Natomiast symbole

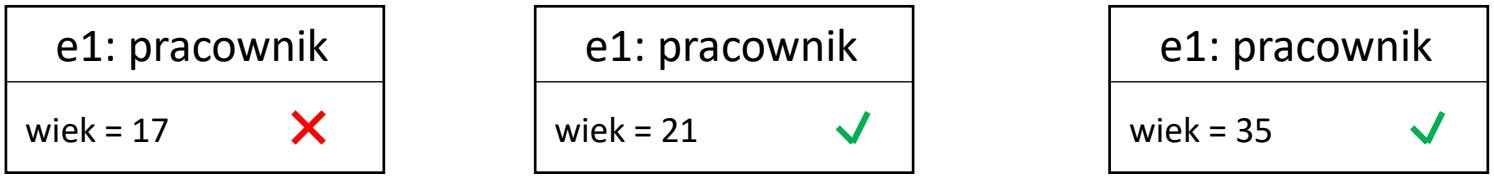
- - - - - - - - >

są używane do wskazywania elementów, na które zostały nałożone ograniczenia.

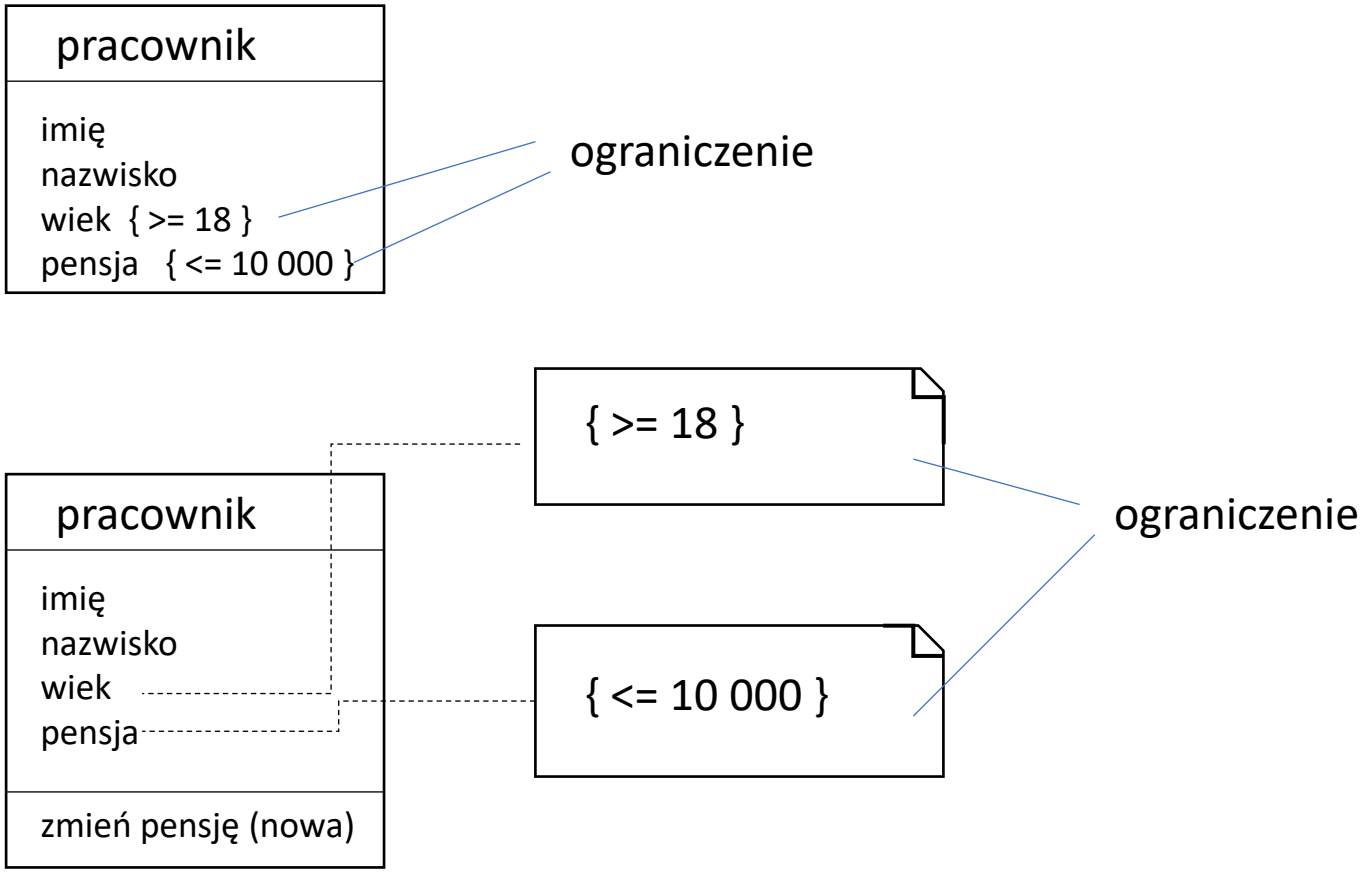
Przykłady ograniczeń dla atrybutów klas



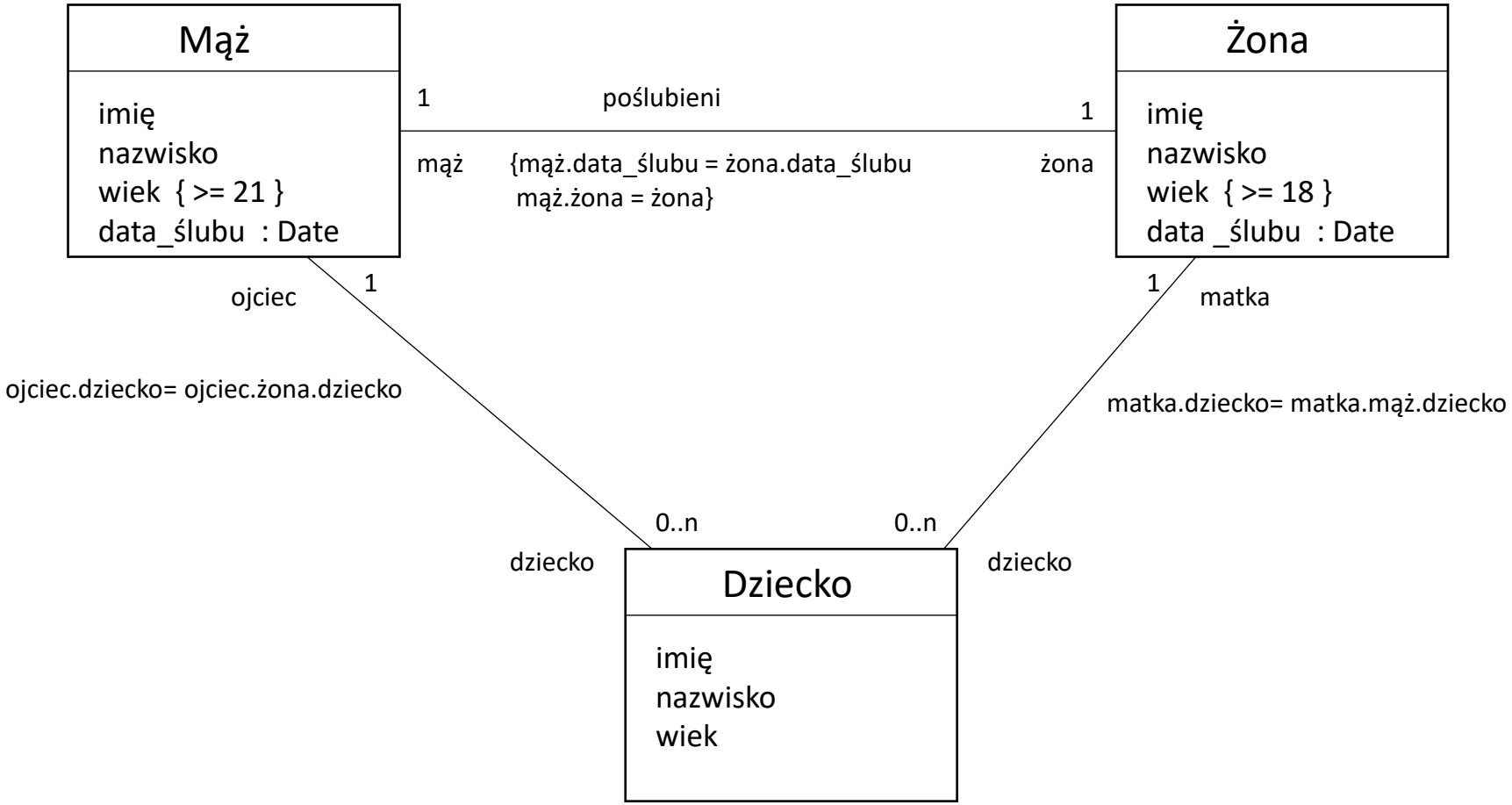




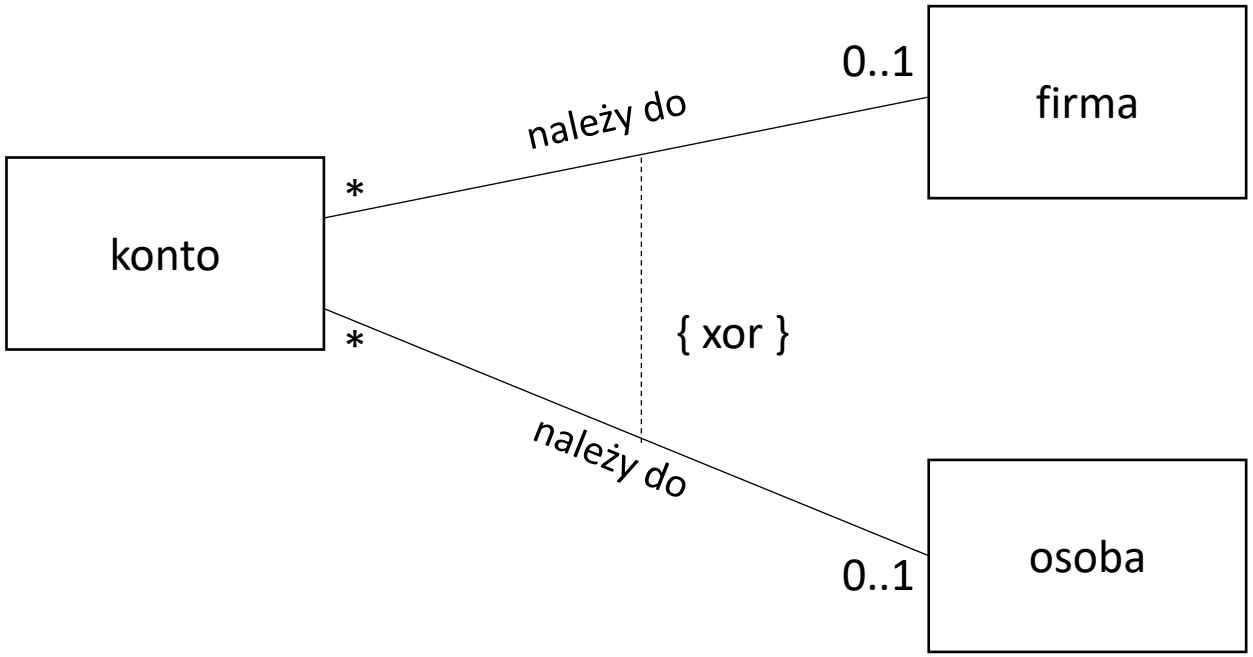
Przykłady ograniczeń dla atrybutów klas



Przykłady ograniczeń



Przykład ograniczenia odnoszący się do asocjacji



Zalety ograniczeń

- **lepsza dokumentacja** – ograniczenia są doskonałą formą dokumentacji, ponieważ uzupełniają charakterystykę wizualnych modeli nowymi elementami i zależnościami między nimi.

- **większa precyzja** – niemożliwe jest odmienne interpretowanie ograniczeń przez różnych ludzi. Ograniczenia są jednoznaczne i zwiększają dokładność modelu lub systemu, do którego się odnoszą.

- **komunikacja bez nieporozumień** – komunikacja między użytkownikami, projektantami, programistami i innymi ludźmi odbywa się za pomocą modeli.

Historia OCL

- Opracowany w IBM w 1995 roku, pierwotnie jako business engineering language
- Przyjęty jako formalny język specyfikacji w UML
- Część oficjalnego standardu OMG dla UML (od wersji 1.1)
- Wykorzystywany do precyzyjnego definiowania zasad prawidłowego uformowania (WFR) dla UML i dalszych metamodeli związanych z OMG
- Obecna wersja to OCL 2.0

Własności OCL

OCL (*Object Constraint Language*) jest to formalny język o notacji tekstowej służący do specyfikowania warunków, ograniczeń, asercji i zapytań (zapisu wyrażeń ścieżkowych).

OCL zawiera pewien zestaw predefiniowanych operatorów do operowania na typach podstawowych i elementach kolekcji, ale nie jest przeznaczony do zapisywania kodu.

OCL ma możliwość definiowania własnych funkcji, warunków i niezmienników. Dzięki nim możliwe jest użycie go do prawie wszystkich elementów modelu UML (klasy, operacji, atrybutu, asocjacji etc.)

Korzystając z notacji **OCL** można również definiować niezmienniki stanów oraz warunki dozorów na przejściach w diagramach stanów i czynności, a także warunki przesyłania komunikatów na diagramach sekwencji i kooperacji.

Własności OCL

Ograniczenia są interpretowane w sposób deklaracyjny, tzn. określają co jest sytuacją poprawną.

Ewaluacja wyrażeń **OCL** następuje w sposób atomiczny (niepodzielny), nie powodując nigdy zmiany stanu jakiegokolwiek obiektu, nie mają efektów ubocznych.

Standard języka **OCL** nie definiuje działań podejmowanych w przypadku niespełnienia niezmienników klas lub warunków początkowych, czy końcowych operacji.

OCL jest niezbędnym uzupełnieniem notacji UML (lub innych języków graficznych), aby móc precyzyjnie określić wszystkie szczegółowe aspekty projektu systemu

Użycie wyrażeń OCL

- specyfikacja zapytań
- specyfikacja niezmienników klas i typów danych w modelu klas
- specyfikacja niezmienników typu dla stereotypów
- specyfikacja warunków wstępnych (*preconditions*) i warunków końcowych (*postconditions*) dla operacji i metod
- specyfikacja warunków dozoru (*guards*)
- specyfikacja celu komunikatów i akcji
- specyfikacja ograniczeń dla operacji
- specyfikacja reguł wyprowadzenia atrybutów pochodnych

Przegląd konstrukcji

Podstawowymi jednostkami, z których składa się wyrażenie OCL, są obiekty i ich właściwości. O każdym obiekcie można powiedzieć, że jest pewnego typu, definiującego wykonywane na nim operacje.

zestaw typów + operacje na nich

Typy podzielono na następujące grupy:

predefiniowane typy podstawowe
predefiniowane typy kolekcyjne
typy modelowe

Przegląd konstrukcji

predefiniowane typy podstawowe

- Integer { Z}
- Real { R }
- Boolean { true, boolean }
- String { ASCII, Unicode }

predefiniowane typy kolekcyjne – służące do precyzyjnego opisywania rezultatu nawigacji po powiązaniach w modelu klas.

- CollectionSet
- Bag
- OrderedSet
- Sequence

typy modelowe – definiowane przez użytkownika (typy zdefiniowane na diagramach UML)

Typ logiczny

Operacja	Notacja	Typ wyniku
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean

Typ całkowity i rzeczywisty

Operacja	Notacja	Typ wyniku
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a+ b	Integer or Real
minus	a - b	Integer or Real
multiplication	a* b	Integer or Real
division	a / b	Real
modulus	a.mod (b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs( )	Integer or Real
max	a.max(b)	Integer or Real
min	a.min(b)	Integer or Real
round	a.round()	Integer
flor	a.flor()	Integer

Typ łańcuchowy

Operacja	Notacja	Typ wyniku
concatenation	s1.concat(s2)	String
size	s.size ()	Integer
to lower case	s.toLower ()	String
to upper case	s.toUpper ()	String
substring	s.substring(i, j)	String
equals	s1 = s2	Boolean
not equals	s1 <> s2	Boolean

Typ modelowy

Typ modelowy – typ zdefiniowany przez użytkownika, to klasyfikator określony w danym modelu **UML** . Klasy, interfejsy, typy danych, wszystkie te elementy modelu UML są typami **OCL**.

Wszystkie klasyfikatory zdefiniowane w danym modelu **UML** są typami dostępnymi w wyrażeniach **OCL** dołączonych do tego modelu

Wyrażenie **OCL** może odnosić się do właściwości typu modelowego: atrybutów, operacji, atrybutów i operacji klas ,nawigacji, wyliczenia zdefiniowane jako typy atrybutów.

Atrybuty i operacje typu modelowego mogą być używany w wyrażeniach **OCL**. Są przywoływane za pomocą notacji kropkowej.

Notacja kropkowa

element.selektor

- selektor może być nazwą atrybutu – wtedy zwracana jest albo wartość albo zbiór wartości atrybutu
- selektor może być nazwą roli – wtedy zwracany jest zbiór powiązanych obiektów

element.selektor(lista_arg)

- selektor może być nazwą operacji wywoływanej dla elementu – wtedy wartością wyrażenia jest wynik zwracany przez tę operację

element.selektor(kwalifikator)

- selektor specyfikuje asocjację kwalifikowaną – element wraz z wartością kwalifikatora jednoznacznie identyfikują zbiór obiektów powiązanych z obiektem specyfikowanym przez element

Nawigacja

OA oznacza obiekt klasy A wówczas:

- wyrażenie OA.aA zwróci wartość atrybutu aA
- wyrażenie OA.rB zwróci zbiór obiektów klasy B powiązanych z OA

Nawigacja

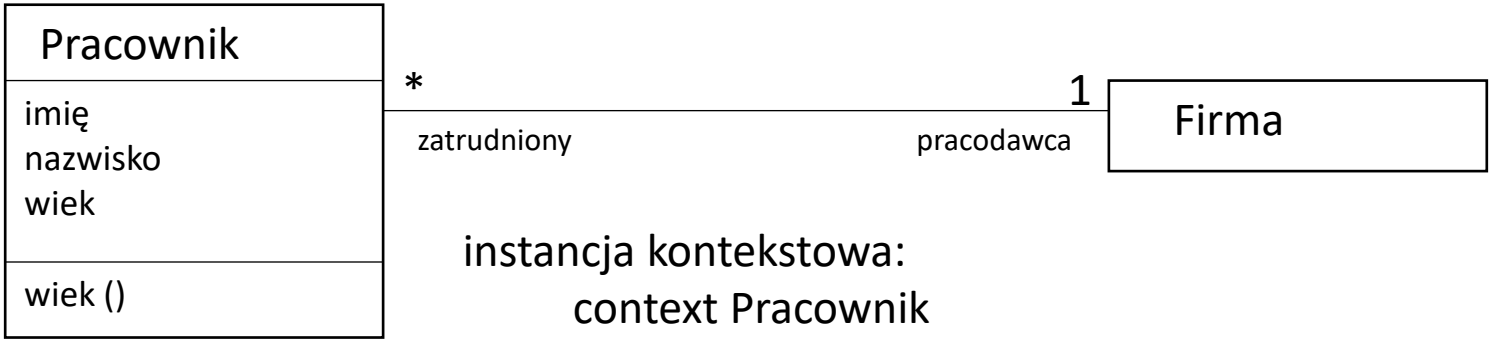
Zakończeń asocjacji można używać do nawigacji z jednego obiektu modelu do drugiego obiektu. Nawigacje są traktowane jako atrybuty. Nazwa nawigacji to nazwa roli lub nazwa typu powiązanego.

Typ nawigacji jest to typ modelowy lub zbiór typów zdefiniowanych przez użytkownika, w zależności od krotności.

Jeśli krotnością zakończenia jest co najwyżej 1 to wartością wyrażenia jest obiekt odpowiedniego typu .

Jeśli zakończenie ma górne ograniczenie krotności większe niż 1 wartością jest kolekcja. Standardowo Set(T), a dla właściwości uporządkowanych OrederedSet(T).

Nawigacja



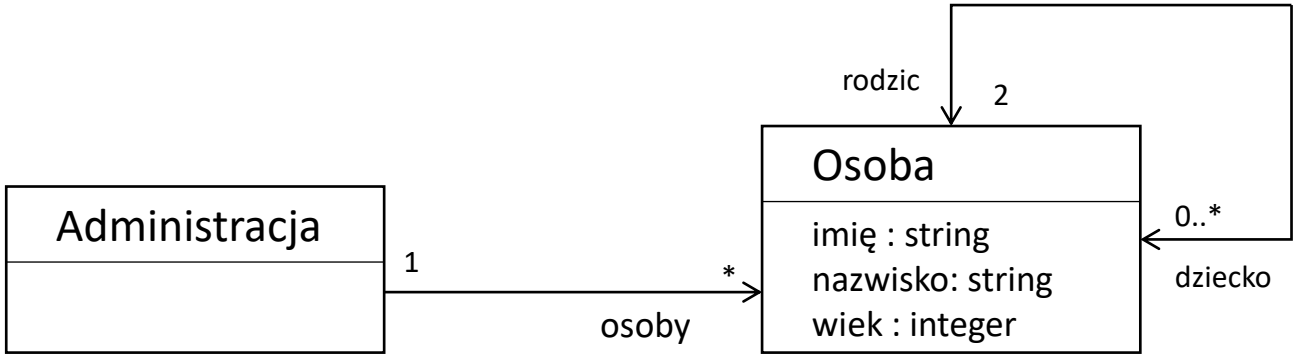
instancja kontekstowa:
context Pracownik

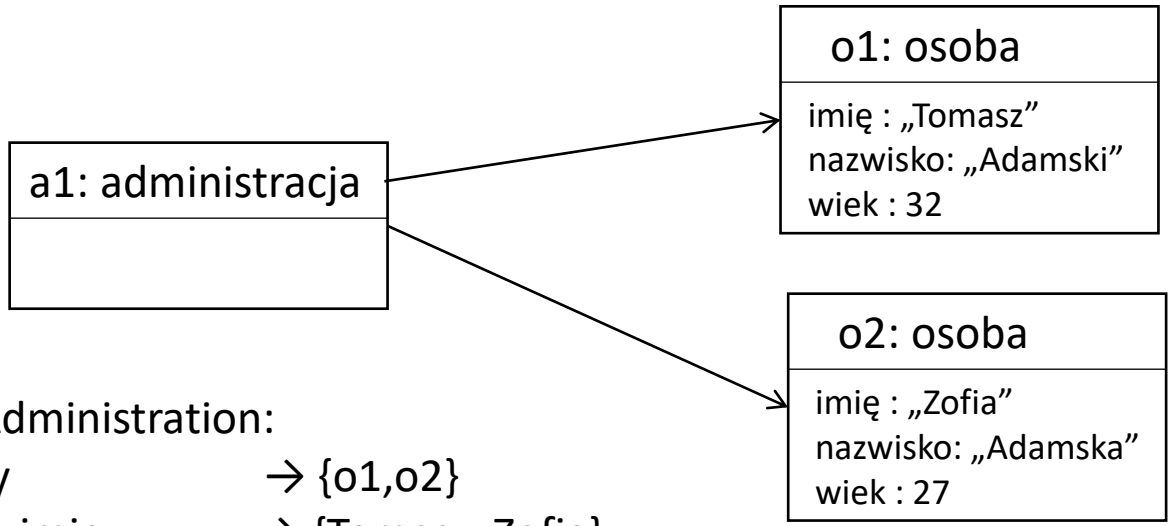
dostęp do atrybutów:
self. wiek

dostęp do operacji:
self. wiek()

Nawigacja : używane są nazwy ról
context Firma
self.pracodawca – zwracaną wartością jest typ Firma
self. zatrudniony – nawigacja zwraca zbiór obiektów, zwracany typ Set of (Pracownik)

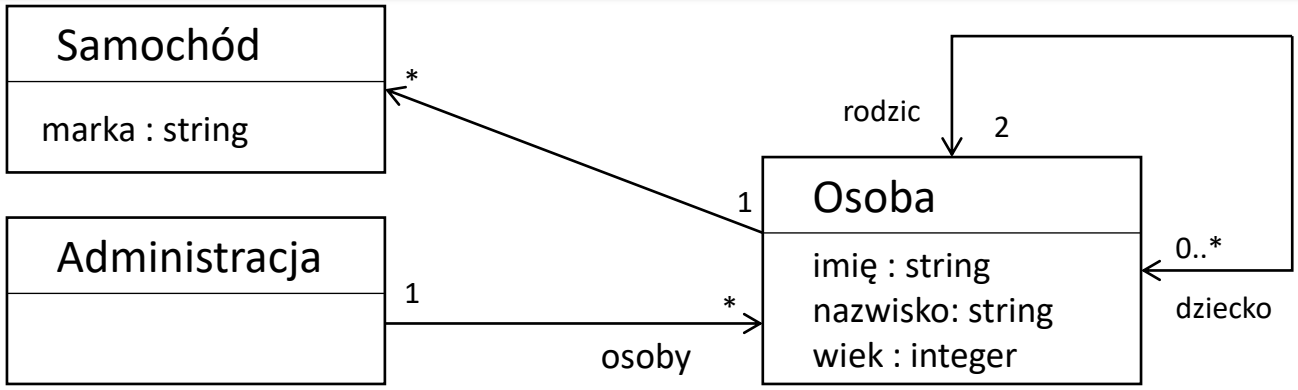
Nawigacja

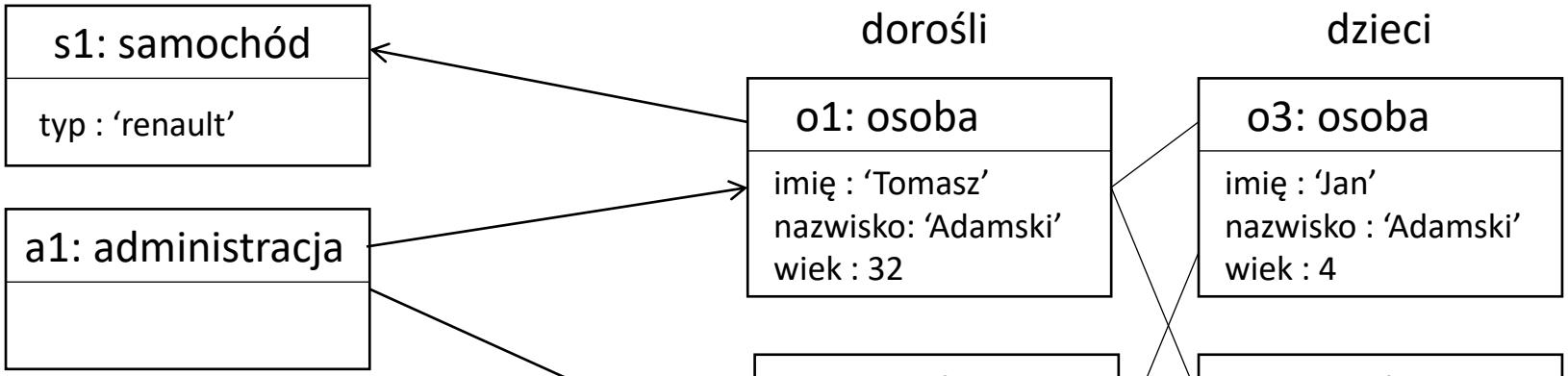




context Administration:
self.osoby → {o1,o2}
self.osoby.imię → {Tomasz, Zofia}
self.osoby.wiek → {32, 27}

Nawigacja





context Administration:
self.osoby.dzieci → {o3,o4,o3,o4}
self.osoby.dziecko.rodzic → {o1,o2,o1,o2}
self.osoby.samochód.marka →{‘Renault’}

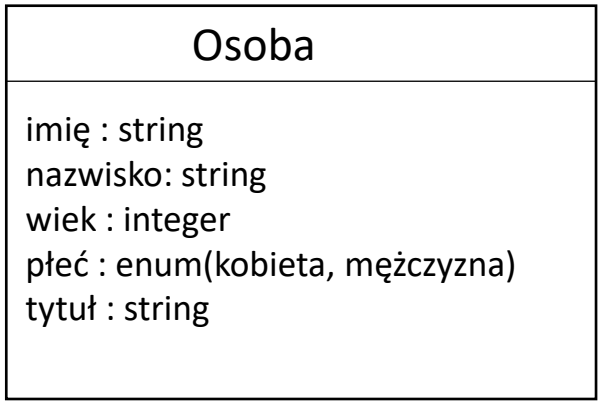
Typ wyliczeniowy

Typ wyliczenowy jest specyficznym typem modelowym, często używanym jako typ atrybutu. Ma następującą składnię:

enum (wartość1, wartość2, wartość3 …)

W wyrażeniach OCL do elementów wyliczenia jako wartości w wyrażeniach odwołujemy się w następujący sposób #wartość1.

Operacje na typie wyliczeniowym: równy = i różny **<>**



context osoba:
self.płeć = #kobieta implies tytuł = ‘ Pani’

Typ kolekcyjny

Collection jest abstrakcyjnym nadtypem dla każdego typu zestawu.

Specjalizujące go typy konkretne

Set (T) – zbiór elementów, nie zawiera duplilatów
OrderedSet (T) – zbiór uporządkowany,), elementy mają przypisane numery
Bag (T) – wielozbiór (zbiór z powtórzeniami)
Sequence (T) – ciąg (wielozbiór uporządkowany), elementy mają przypisane numery

Typ kolekcyjny

Kolekcje są wynikami powiązań:

- **Set (T)** – dla powiązań prostych (bez dodatkowych atrybutów)
- **OrderedSet (T)** – dla powiązań prostych uporządkowanych
- **Bag (T)** – wynik kilku nawigacji, dla powiązań złożonych oraz prostych z powtórzeniami
- **Sequence (T)** – dla powiązań prostych uporządkowanych z powtórzeniami

Typ kolekcyjny

Typy kolekcyjne mogą być określone poprzez wyliczenie ich elementów:

Set { 1, 2, 5, 3, 4 }

Set { ‘kwadrat’, ‘trójkąt’, ‘okrąg’, ‘prostokąt’}

OrderedSet {‘niebieski’, ‘biały’, ‘czerwony’, ‘zielony’}

Bag { 1, 1, 5, 3, 1, 2, 2 }

Sequence {5, 2, 2, 1, 3}

Dla Sequence istnieją równoważne notacje skrócone

Sequence { 1 .. 6} = Sequence { 1 .. (2 + 4)} = Sequence {1, 2, 3, 4, 5, 6}

Typ kolekcyjny

Czasami typ elementu modelu musi być wyraźnie określony, np. podczas definiowania nowego atrybutu.

Set (Customer)

Sequence (Set (ProgramPartner))

OrderedSet (ServiceLevel)

Bag (Burning)

W przypadku zagnieżdżonych kolekcji, w większości przypadków są one automatycznie spłaszczone.

Set { Set { 1, 2 }, Set { 3, 4 }, Set { 5, 6 } }

Set { 1, 2, 3, 4, 5, 6}

Operacje na kolekcjach

Wiele standardowych operacji typu Collection(T) i jego podtypów jest zdefiniowanych w standardowej bibliotece OCL.

Są one oznaczane w wyrażeniach OCL za pomocą strzałki (operacje zdefiniowane przez użytkownika są oznaczone za pomocą kropek).

Składnia

collection -> operacja(...)

{1, 2, 3} -> size()

self.zatrudniony -> size() < 1000

Operacje standardowe nigdy nie zmieniają stanu kolekcji (są zapytaniami).

Standardowe operacje na kolekcjach

Operacja	Opis
count (element)	Liczba wystąpień obiektu w kolekcji
excludes (element)	True, jeśli element nie jest elementem kolekcji
excludesAll (collection)	True, jeśli nie wszystkie elementy kolekcji parametrów są w aktualnej kolekcji
includes (element)	True, jeśli element jest elementem kolekcji
includesAll (collection)	True, jeśli wszystkie elementy kolekcji parametrów są w aktualnej kolekcji
isEmpty()	True, jeśli kolekcja nie zawiera żadnych elementów
notEmpty()	True, jeśli kolekcja zawiera jeden albo więcej elementów
size()	Ilość elementów w kolekcji
sum()	Suma wszystkich elementów kolekcji. Elementy muszą być typu Real lub Integer

Operacje na kolekcjach

Niektóre operacje zdefiniowane dla wszystkich typów kolekcyjnych mają inne znaczenie w zależności od tego na jakim typie zostały zastosowane:

równy (**=**) and różny (**<>**)

- dwa **Set** (zbiory) są sobie równe, jeżeli wszystkie elementy są takie same.
- dwa **OrderedSets** (uporządkowane zbiory) są sobie równe , jeżeli wszystkie elementy są takie same i w takim samym porządku.
- dwa **Bags** (wielozbiory) są sobie równe, jeżeli wszystkie elementy pojawiają się w nich tyle samo razy.
- dwie **Sequences** (sekwencje) są sobie równe, jeżeli wszystkie elementy pojawiają się w nich tyle samo razy i takiej samej kolejności

Operacje na kolekcjach

including(element**)**

- rezultatem jest powstanie nowej kolekcji z dodanym elementem
- przypadku **Set** lub **OrderedSet** element jest dodawany tylko wtedy, jeżeli takiego elementu wcześniej nie było
- w przypadku **Sequence** lub **OrderedSet** element dodawany jest na koniec.

excluding(element**)**

- rezultatem jest powstaniem nowej kolekcji bez usuwanego elementu
- w przypadku **Bag** lub **Sequence** usuwane są wszystkie wystąpienia danego elementu.

Operacje na kolekcjach

flatten - (spłaszczenie) zamienia kolekcję kolekcji na kolekcję pojedynczych elementów.

- rezultatem **flatten** **Bag** lub **Set** jest odpowiednio **Bag** lub **Set**, kolejność elementów nie może być dokładnie określona.
- w przypadku **Sequence** lub **OrderedSet** jest odpowiednio **Sequence** lub **OrderedSet**

Set { Set { 1, 2 }, Set { 2, 3, 4 }, Set { 4, 5, 6 } } → Set { 1, 2, 3, 4, 5, 6 }

Bag { Set { 1, 2 }, Set { 1, 2 }, Set { 4, 5, 6 } } → Bag { 1, 1, 2, 2, 4, 5, 6 }

Sequence { Set { 1, 2 }, Set { 2, 3}, Set { 4, 5, 6 } }
→ Sequence { 1, 2, 2, 3, 4, 5, 6 }

Operacje na kolekcjach

asSet, **asSequence**, **asBag**, **asOrderedSet**

- operacje te przekształcają instancje jednego typu kolekcji w kolekcję innego typu
- zastosowanie **asSet** na **Bag** lub **asOrderedSet** na **Sequence** usunie zduplikowane elementy.
- zastosowanie **asSet** na **OrderedSet** lub **asBag** na **Sequence** spowoduje utratę informacji o uporządkowaniu elementów.
- zastosowanie **asOrderedSet** lub **asSequence** na **Set** lub **Bag** spowoduje losowe rozmieszczenie elementów.

Operacje na kolekcjach

union – łączy dwie kolekcje w jedną

- uporządkowane kolekcje mogą być łączone tylko z uporządkowanymi kolekcjami
- rezultatem połączenia **Set** z **Bag** będzie **Bag**.

intersection – przecięcie dwóch kolekcji

- nie może być stosowane do kolekcji uporządkowanych

minus – różnica dwóch kolekcji

- zastosowane do uporządkowanych kolekcji nie zmienia kolejności.

Operacje na kolekcjach

Operacje na uporządkowanych kolekcjach (**OrderedSet** i **Sequence**):

first/last – zwraca first/last element

at – zwraca element o zadanym indeksie

indexOf – zwraca indeks danego elementu

insertAt – wstawia element na zadaną pozycję

subSequence – zwraca podciąg (subsequence) o określonych indeksach

subOrderedSet – zwraca uporządkowany podzbiór o określonych indeksach

append/prepend – dodaje na koniec albo na początek kolekcji dany element

Operacje iteracyjne po elementach kolekcji

W OCL zdefiniowano kilka operacji kolekcyjnych, które służą do wychwytywania pewnych elementów kolekcji.

Obliczają one wartość zadanego wyrażenie dla wszystkich elementów kolekcji.

możliwa składnia tych operacji:

collection -> operacja (element: Typ |wyrażenie(element))

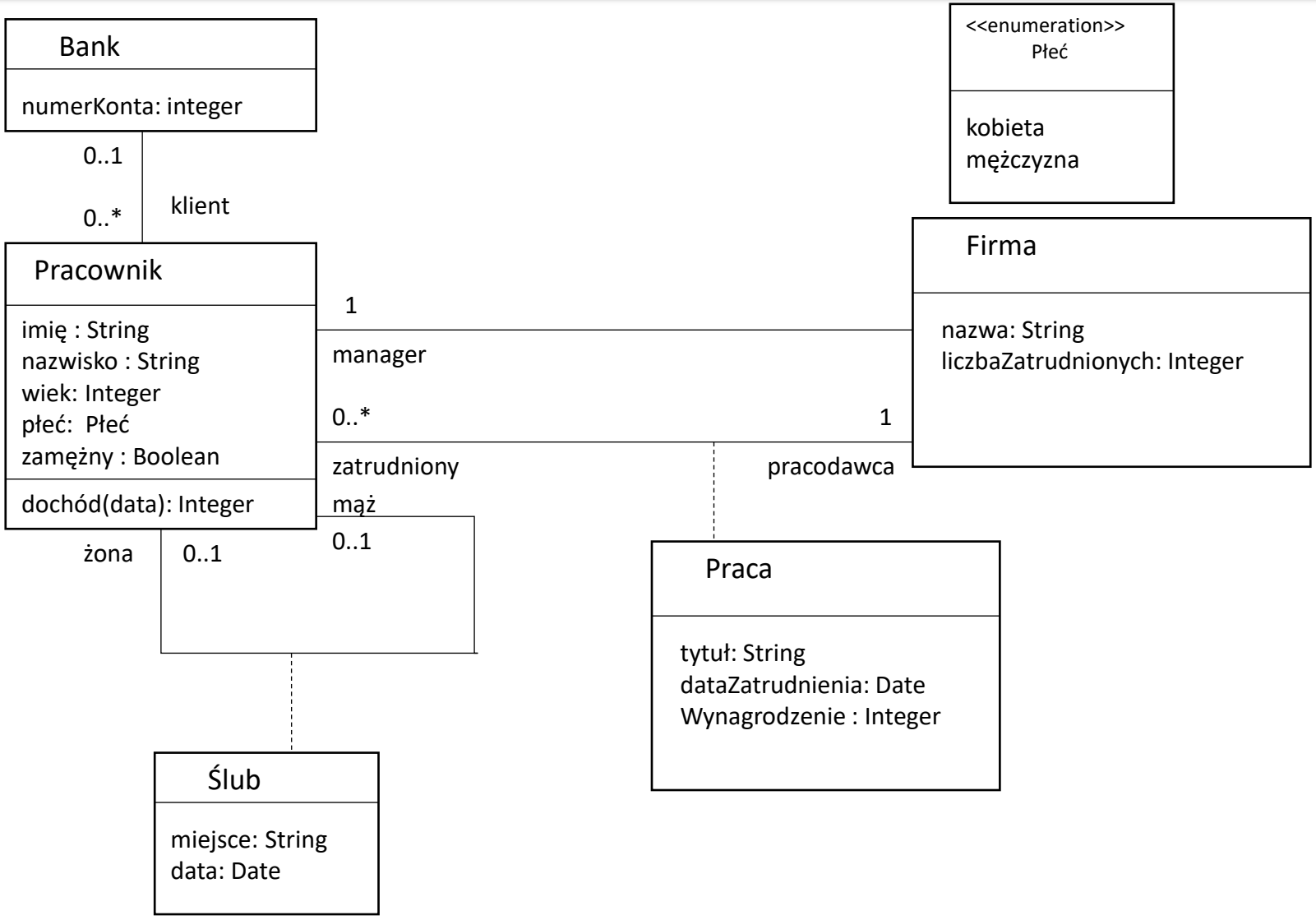
collection -> operacja (element | wyrażenie (element))

collection -> operacja (wyrażenie)

Operacje iteracyjne po elementach kolekcji

Operation	Opis
any (wyrażenie)	Zwraca losowy element kolekcji źródłowych, dla których wyrażenie podane w nawiasach jest prawdziwe
collect(wyrażenie)	Zwraca kolekcję elementów, które spełniają wyrażenie w nawiasach dla każdego elementu kolekcji źródłowej
exist(wyrażenie)	Zwraca true, jeśli co najmniej jeden element kolekcji źródłowej dla którego wyrażenie jest prawdziwe
forAll(wyrażenie)	Zwraca true, jeśli wyrażenie jest prawdziwe dla wszystkich elementów kolekcji źródłowej
isUnique(wyrażenie)	Zwraca true, jeśli wyrażenie ma unikalną wartość dla wszystkich elementów kolekcji źródłowej
iterate(…)	Iteruje po wszystkich elementach kolekcji źródłowej
one(wyrażenie)	Zwraca true, jeśli kolekcja źródłowa zawiera dokładnie jeden element, dla której wyrażenie jest prawdziwe
reject(wyrażenie)	Zwraca podkolekcję zawierającą wszystkie elementy, dla których wyrażenie jest nieprawdziwe
select(wyrażenie)	Zwraca podkolekcję zawierającą wszystkie elementy, dla których wyrażenie jest prawdziwe
sortedBy(wyrażenie)	Zwraca kolekcję zawierającą wszystkie elementy kolekcji źródłowej, uporządkowanych przez wartość wyrażenia

Przykładowy diagram klas



Operacja select

select - ta operacja przyjmuje jako dane wejściowe wyrażenie logiczne, i zwraca kolekcję zawierającą wszystkie elementy dla których wyrażenie logiczne jest prawdziwe.

Składnia

collection -> select(wyrażenieLogiczne)

collection -> select(element | wyrażenieLogiczne(element))

collection -> select(element: Typ | wyrażenieLogiczne(element))

równoważne

```
context Firma
inv: zatrudniony -> select(wiek >55) -> notEmpty()
inv: zatrudniony -> select(p | p.wiek >55) -> notEmpty()
inv: zatrudniony -> select(p:Pracownik |p.wiek >55) -> notEmpty()
inv: zatrudniony -> select (dochód > 10 000) -> isEmpty()
```

Operacja reject

reject - pozwala wyspecyfikować podzbiór wejściowej kolekcji przez podanie warunku odrzucenia

Składnia (analogiczna jak dla Select)

collection->reject(wyrażenieLogiczne)

collection->reject(element | wyrażenieLogiczne(element))

collection->reject(element: Typ | wyrażenieLogiczne(element))

```
context Company
inv: zatrudniony -> reject(zamężny) -> isEmpty()
inv: zatrudniony -> reject(p | p.zamężny) -> isEmpty()
inv: zatrudniony -> reject(p: Pracownik | p.zamężny) -> isEmpty()
```

Operacja collect

collect – zwraca nową kolekcję wywiedzioną z danej kolekcji, ale zawierającą inne obiekty

- Set(T) przekształca w Bag(T)
- OrderedSet(T) przekształca w Sequence(T)

składnia

collection -> collect(wyrażenie)

zatrudniony -> collect(element|wyrażenie(element))

zatrudniony -> collect(element: Typ | wyrażenie(element))

Operacja collect

zatrudniony -> collect(imię)

zatrudniony -> collect(pracownik | pracownik.imię)

zatrudniony -> collect(p: Pracownik | p.imię)

równoważne

zatrudniony -> collect(imię) -> asSet()

zwraca Bag
z powtórzeniami

zwraca Set
bez powtórzeń

w notacji skróconej

zatrudniony -> collect(imię) = zatrudniony.collect(imię)

Operacja forAll

forAll – kwantyfikator ogólny, pewien warunek musi obowiązywać dla wszystkich elementów kolekcji.

składnia

collection -> forAll(wyrażenieLogiczne)

collection -> forAll(element | wyrażenieLogiczne(element))

collection -> forAll(element : Typ| wyrażenieLogiczne(element))

Operacja forAll

forAll – kwantyfikator ogólny, pewien warunek musi obowiązywać dla wszystkich elementów kolekcji.

```
context Firma
inv: zatrudniony -> forAll(wiek <= 65)
inv: zatrudniony -> forAll(p | p.wiek <= 65)
inv: zatrudniony -> forAll(p: Pracownik | p.wiek <= 65)
```

równoważne

W operacji tej może być zdeklarowany wiele zmiennych iteracyjnych.

```
context Firma
inv: zatrudniony -> forAll(p1, p2 |
                    p1 <> p2 implies p1.nazwisko <> p2.nazwisko)
inv: zatrudniony -> forAll( p1 | zatrudniony -> forAll (p2 |
                    p1 <> p2 implies p1.nazwisko <> p2.nazwisko))
```

równoważne

Operacja exists

exists – kwantyfikator szczegółowy, pewien warunek musi obowiązywać dla co najmniej jednego elementu kolekcji.

składnia

collection -> exists(wyrażenieLogiczne)

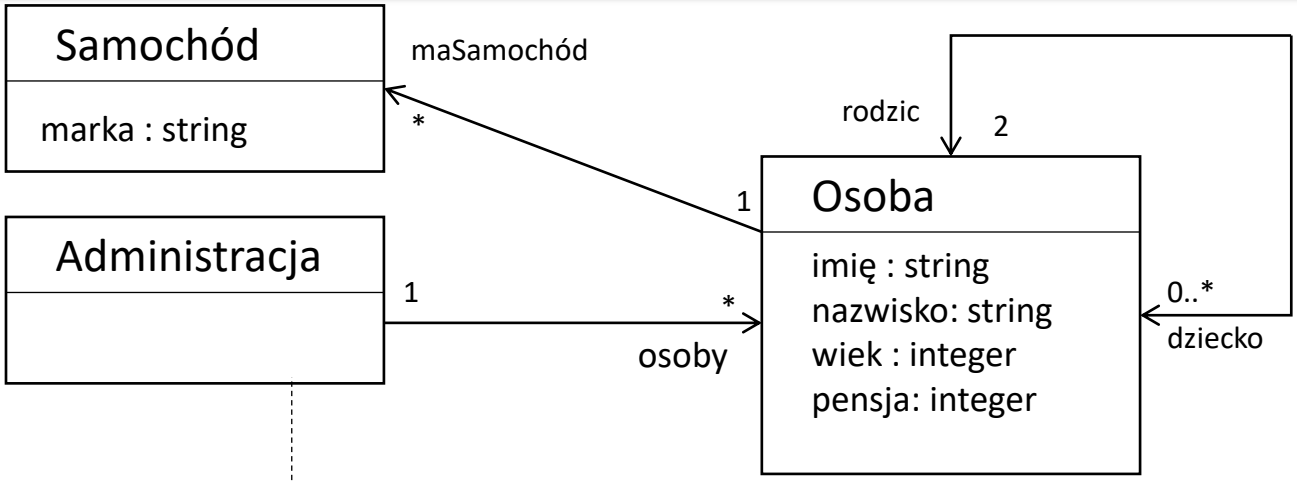
collection -> exists(element | wyrażenieLogiczne(element))

collection -> exists(element : Typ| wyrażenieLogiczne(element))

```
context Firma
inv: zatrudniony -> exists(płeć = #kobieta)
inv: zatrudniony -> exists(p | p.płeć = #kobieta)
inv: zatrudniony -> exists(p: Pracownik | p. płeć = #kobieta)
```

równoważne

Operacja iteracyjne na elementach kolekcji




```
context Administracja
inv:self.osoby.wiek –> collect (wiek >= 65) –> count() < 10
self.osoby .pensja –> select (pensja > 10 000) –> sum() < 10
self.osoby.imię    –> exist(imię = ‘Tomasz’)
```

```
context Osoba
self.osoby.maSamochód -> exist( marka = ‘ mercedes’)
```

Operacja iterate

iterate – Najogólniejsza z operacji iteracyjnych, pozwala stworzyć nową kolekcję na podstawie danej kolekcji i jawnie określonego wyrażenia.

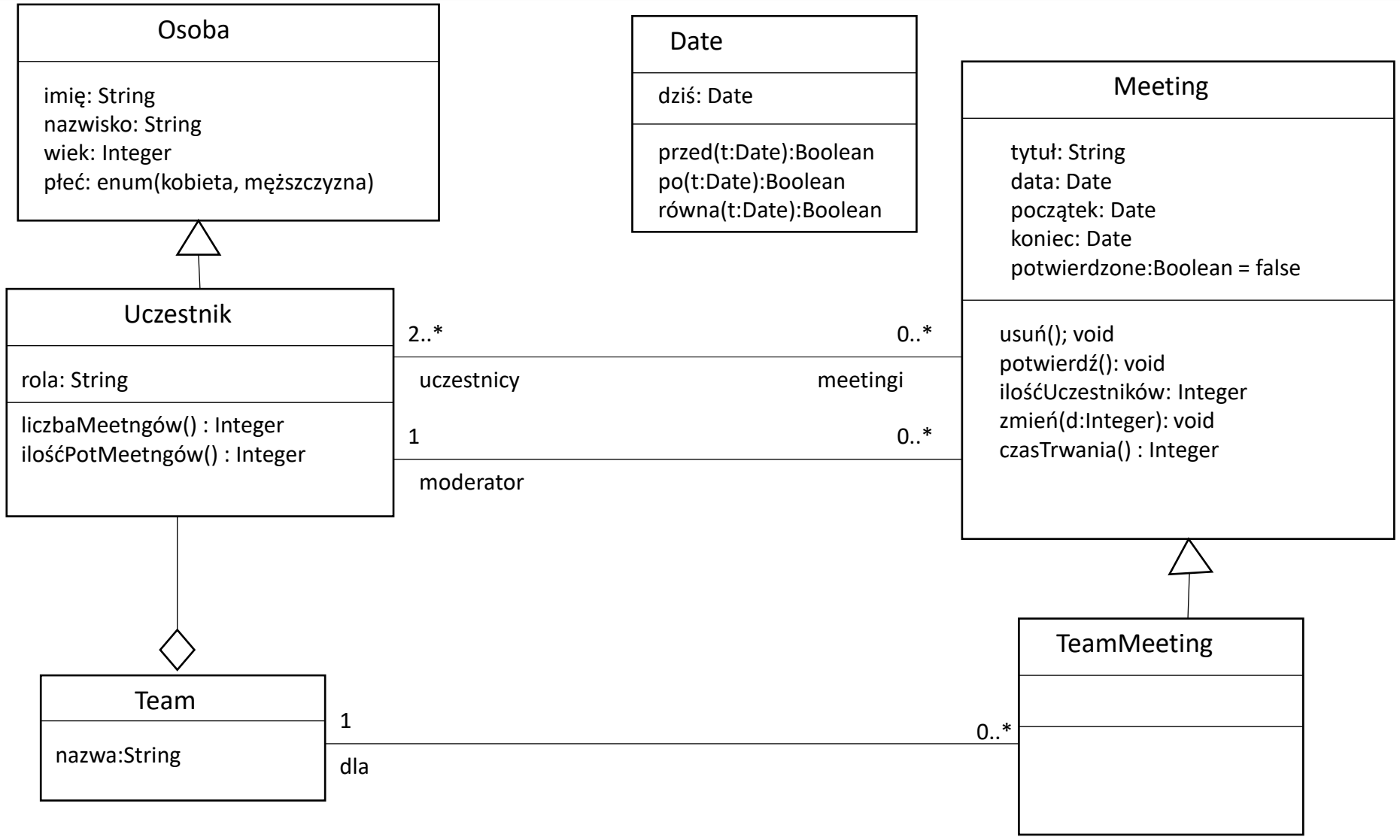
Składnia

collection -> iterate(element : Type1; result : Type2 = wyrażenie|
wyrażenie(element,result))

Set {1, 2, 3 } -> iterate (i :integer, sum : Integer = 0 | sum +i)

Set {1, 2, 3 } -> sum( )

Przykład


Osoba
imię: String
nazwisko: String
wiek: Integer
płeć: enum(kobieta, męższczyzna)
Date
dziś: Date
przed(t:Date):Boolean
po(t:Date):Boolean
równa(t:Date):Boolean
Meeting
tytuł: String
data: Date
początek: Date
koniec: Date
potwierdzone:Boolean = false
usuń(); void
potwierdź(): void
ilośćUczestników: Integer
zmień(d:Integer): void
czasTrwania() : Integer
Uczestnik
rola: String
liczbaMeetngów() : Integer
ilośćPotMeetngów() : Integer
2..*
uczestnicy
0..*
meetingi
1
moderator
0..*
Team
nazwa:String
1
dla
0..*
TeamMeeting

Podstawowe elementy składni – context

Kontekst wyrażenia **OCL** określa encja w modelu UML, dla której jest wyrażenie zdefiniowane.

Każde wyrażenie **OCL** jest napisane w kontekście pewnej instancji konkretnego typu.

Zwykle kontekstem jest klasa, interfejs, typ danych, składnik lub metoda.

Typ kontekstowy to typ obiektu dla które wyrażenie będzie weryfikowane.

Jeśli sam kontekst jest typem, kontekst jest równy typowi kontekstowemu.

Jeśli kontekst to operacja lub atrybut, typ kontekstowy to typ, dla którego jest zdefiniowany kontekst.

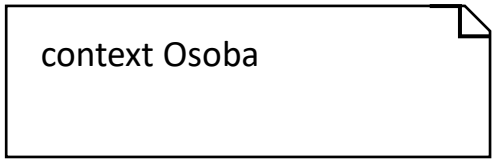
Podstawowe elementy składni – context

Wyrażenie **OCL** jest zawsze weryfikowane dla pojedynczego wystąpienia typu kontekstowego, zwanego instancją kontekstową.

Kontekst jest określany przez słowo kluczowe **context**, po którym następuje nazwa elementu modelu (głównie nazwy klas).

Kontekst wewnątrz wyrażenia można określić przy pomocy deklaracji kontekstu:

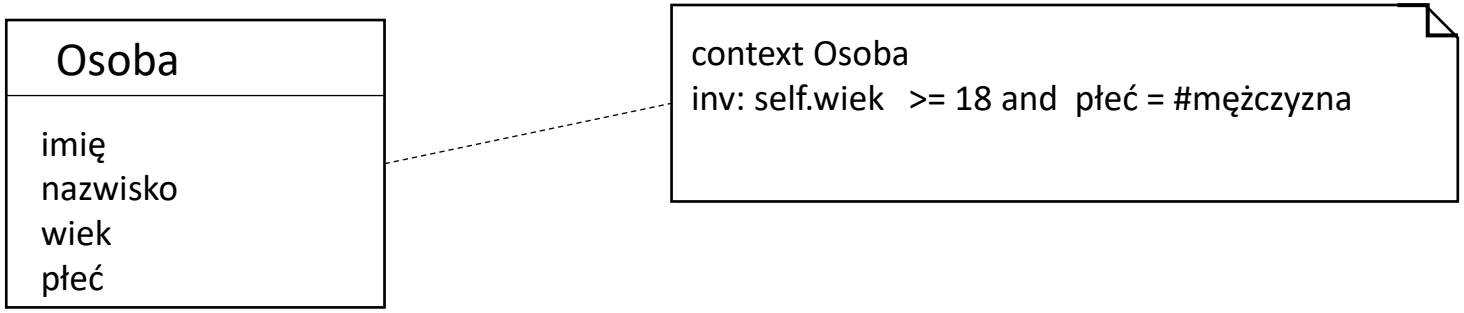
context nazwa elementu modelu



Podstawowe elementy składni – context

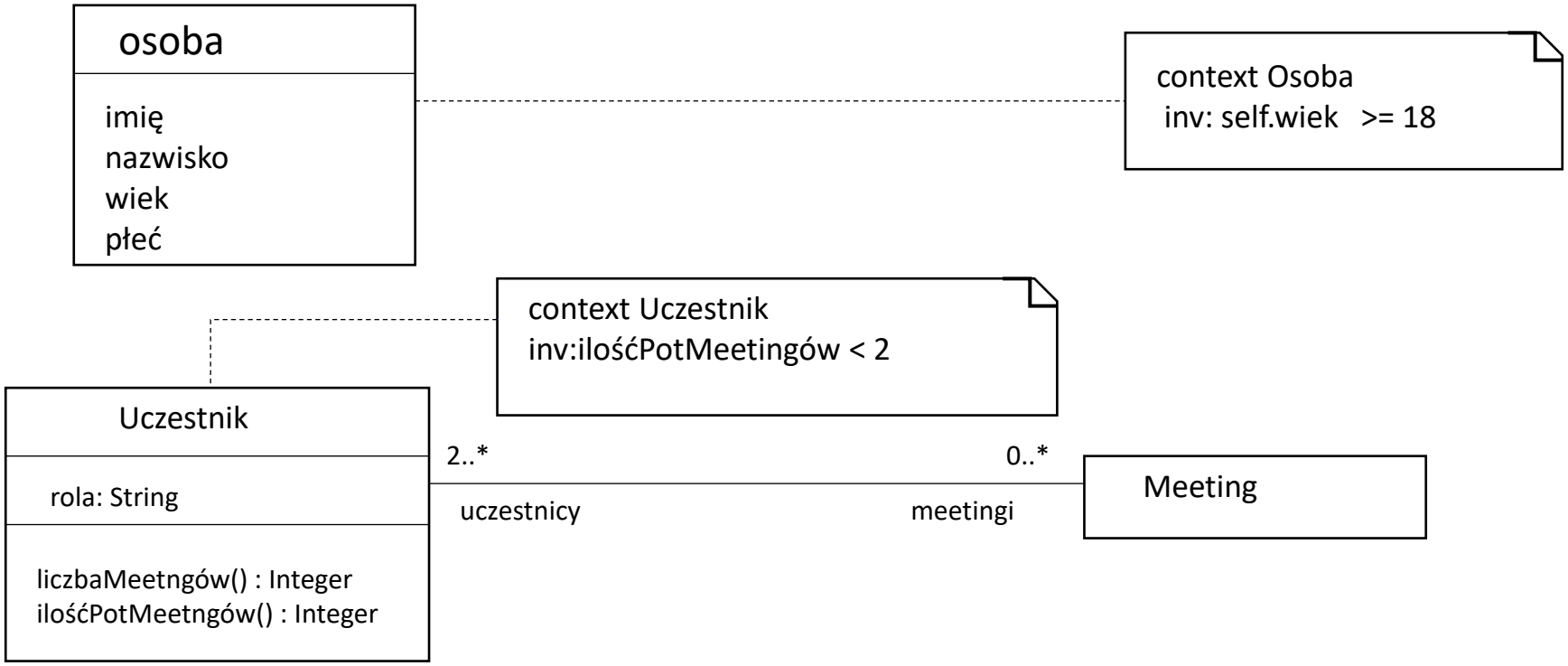
Kontekst – musi być przypisany do każdej instrukcji **OCL**

- adres początkowy – dla którego elementu modelu zdefiniowana jest instrukcja **OCL**
- określa, które elementy modelu mogą używać ścieżki wyrażenia



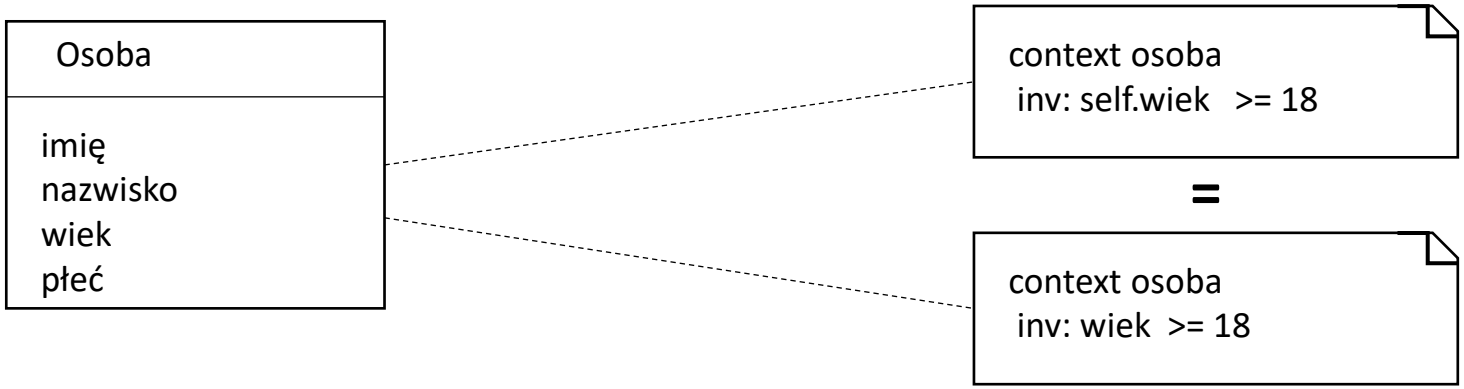
Podstawowe elementy składni – self

Słowo kluczowe **self** określa bieżącą instancję, która będzie sprawdzana przez niezmiennik (instancja kontekstowa).



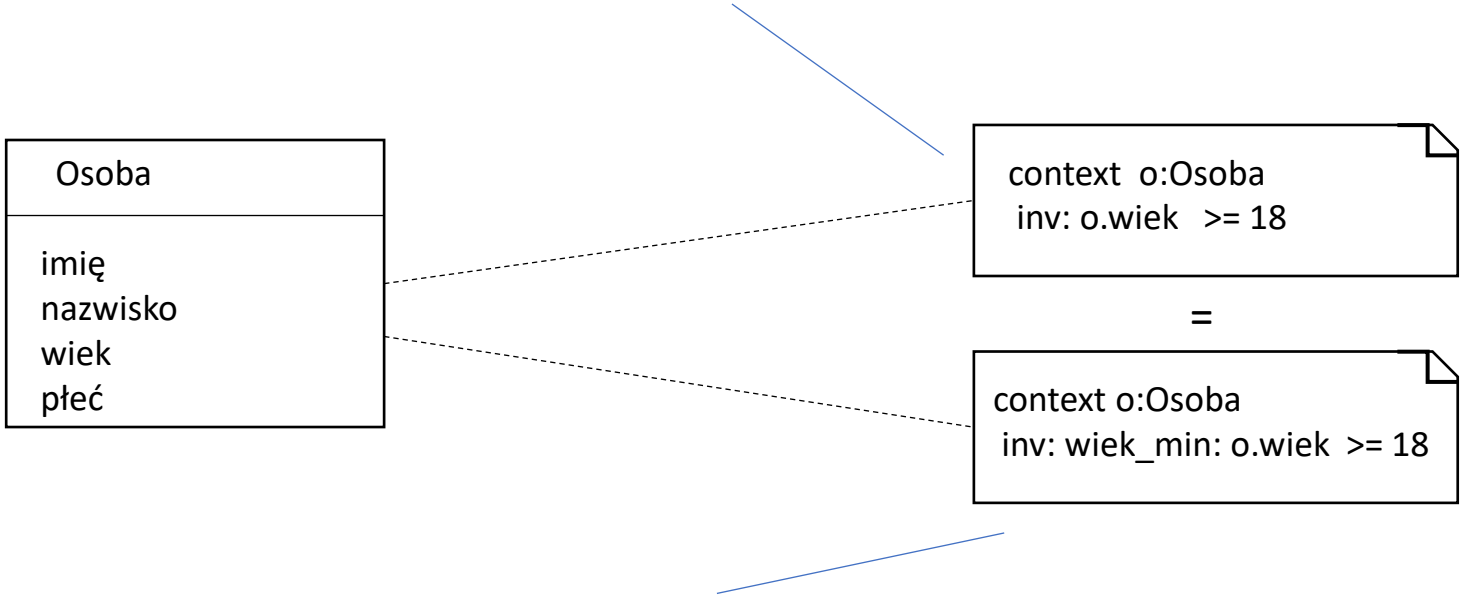
Podstawowe elementy składni – self

self można pominąć, jeśli instancja kontekstu jest unikalna



Podstawowe elementy składni – self

Można zdefiniować symbol zastępujący **self**



Ograniczeniu można nadać nazwę

context – klasy

Gdy kontekstem jest klasa (lub klasyfikator), można używać następujących typów wyrażeń OCL:

- niezmienniki
- definicje nowych atrybutów
- definicje nowych operacji

Niezmiennik

Niezmiennik (ang. *invariant*) – to ograniczenie, które obowiązuje dla każdej instancji danego klasyfikatora i musi być zawsze spełnione.

- dla dowolnej klasy można podać jej niezmiennik w postaci zbioru wyrażeń.
- każde wyrażenie niezmiennika jest wyrażeniem boolowskim, tzn. może być wartościowane jako true lub false.
- dla więcej niż jednego wyrażenia przyjmuje się, że niezmiennik jest koniunkcją tych wyrażeń.
- niezmiennik danej klasy oznacza, że wszystkie obiekty tej klasy spełniają zadany niezmiennik.

Niezmiennik

Niezmiennik oznaczamy słowem kluczowym **inv**

Składnia:

context nazwa klasy

inv : [nazwa ograniczenia]: wyrażenie logiczne

```
context Meeting
 inv : self.end > self.start
```

```
context Meeting inv  startEndConstraint:
self.end > self.start
```

ograniczeniu można nadać nazwę

```
context Osoba  inv:
self.nazwisko <> ‘ ’  and  self.wiek  >= 18
self.wiek  <= 65
```

Nowe atrybuty i operacje

Za pomocą wyrażeń **OCL** do definicji klasy można dodać nowe atrybuty

```
context Osoba
def: nickname: String = „J23”
```

Podobnie, nowe operacje można dodać do definicji klasy. Wszystkie operacje zdefiniowane przez **OCL** muszą być operacjami zapytania.

```
context Osoba
def: jestDorosły (): Boolean= self.wiek >= 18
```

Kontekst – operacje

Gdy kontekst jest operacją lub inną cechą czynnościową, można wówczas stosować następujące typy wyrażeń OCL:

- warunki wstępne (*pre-conditions*)
- warunki końcowe (*post-conditions*)

Składnia

context Type::operation(par1: T1, ...): ReturnType

pre: ...

post: …

self jest instacją typu, do którego należy dana cecha czynnościowa

słowo **result** określa wynik operacji

Warunek wstępny/końcowy

Warunek wstępny/ warunek końcowy – to ograniczenia, które określają zastosowanie i efekt działania operacji bez podania algorytmu lub implementacji.

- są one dołączone do operacji na diagramie klas
- pozwalają na pełniejszą specyfikację systemu

Warunek wstępny

Warunek wstępny – jest to ograniczenie, które musi być prawdziwe tuż przed rozpoczęciem wykonywania danej operacji, aby ta operacja wykonała się poprawnie.

Składnia

context klasyfikator :: operacja (parametry)

pre [nazwa ograniczenia]: wyrażenie logiczne

```
context Meeting :: zmień(d : integer)
pre:  self.potwierdzone = false
```

Warunek końcowy

Warunek końcowy – jest to ograniczenie, które musi być prawdziwe tuż po wykonywania danej operacji.

Warunki końcowe to sposób, w jaki rzeczywisty wynik operacji jest opisany w OCL.

Składnia

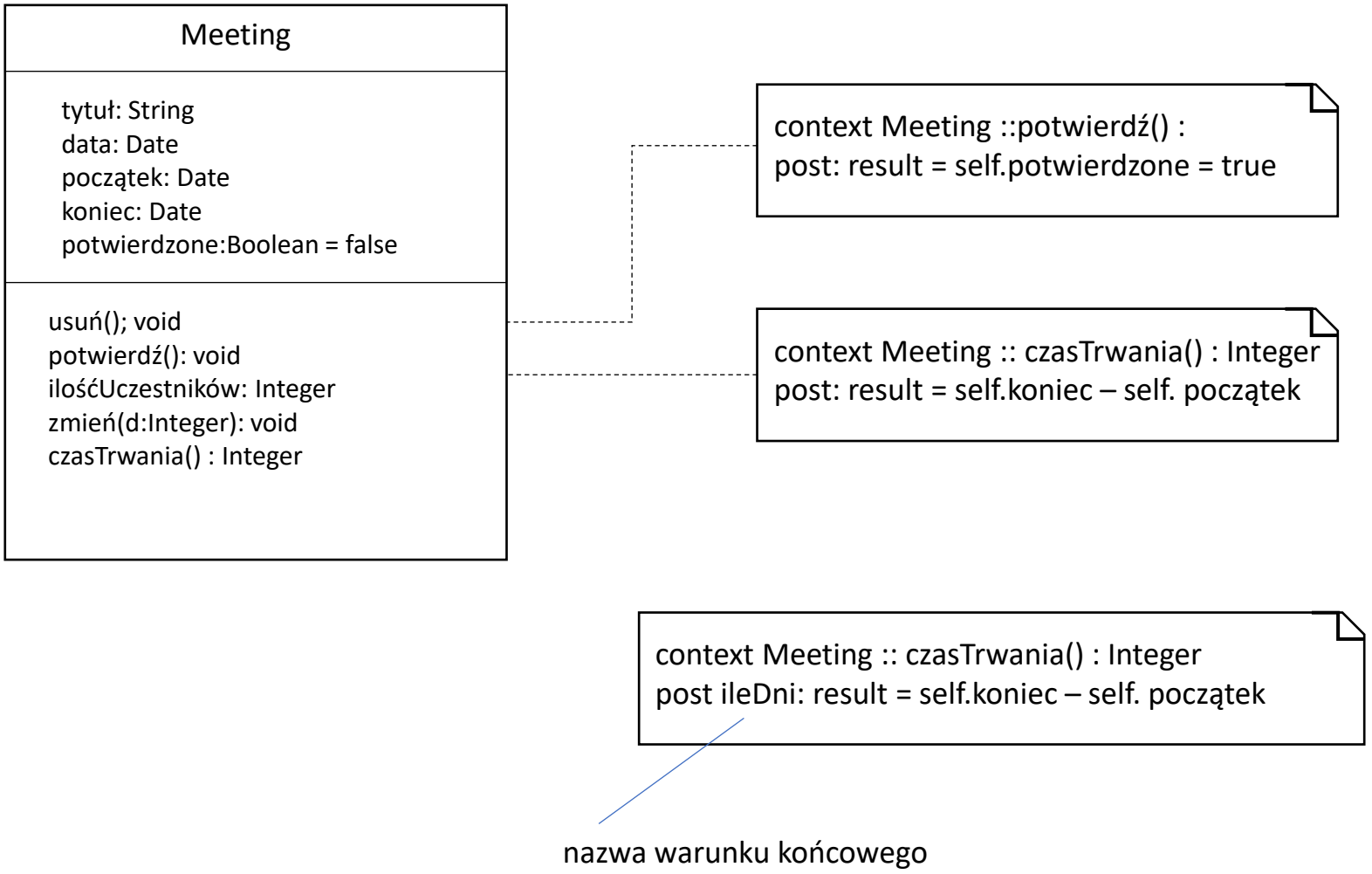
context klasyfikator :: operacja (parametry): Typ

post [nazwa ograniczenia]: wyrażenie logiczne

context Meeting ::potwierdź() :
post: result = self.potwierdzone = true

context Pracownik::dochód(d: Date) : Integer
post: result = 5 000

Warunek końcowy



Jak pisać wyrażenia OCL

Aby ograniczenia powinny były łatwe do odczytania i zapisania należy:

- unikać skomplikowanych wyrażeń nawigacyjnych
- znaleźć odpowiedni kontekst
- unikać **allInstances()**
- zmniejszać ilość **and** dzieląc ograniczenie na wiele ograniczeń
- używać **collect**
- używać nazw zakończeń asocjacji (nazw ról) zamiast nazw asocjacji w modelu

Język ODCL

Język **OCL** umożliwia specyfikację statycznych ograniczeń, jest jednak niewystarczający przy specyfikacji ograniczeń dynamicznych. Rozszerzenie języka **OCL** o operatory logiki temporalnej umożliwia specyfikację ograniczeń dynamicznych to **ODCL** (*Object Dynamic Constraint Language*).

Połączenie języka OCL i logiki temporalnej umożliwia tworzenie formalnych opisów zachowania systemu przez określenie poprawnych sekwencji stanów, co jest szczególnie użyteczne przy modelowaniu zachowania systemu.

Język ODCL

W języku **ODCL**, oprócz standardowych konstrukcji języka OCL, wykorzystywać można także operatory temporalne czasu przeszłego interpretowane następująco:

- **prev A** – jest spełnione w stanie aktualnym, jeżeli A jest spełnione w poprzednim stanie.
- **A since B** – jest spełnione w aktualnym stanie, jeżeli istniał stan wcześniejszy, w którym spełnione było B.
- **sometime A** – kiedyś w przeszłości spełnione było A.
- **always A** – zawsze w przeszłości spełnione było A.

gdzie: A, B są wyrażeniami logicznymi języka ODCL.