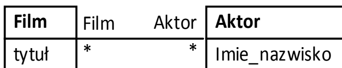
1. Asocjacja



Nie występują bezpośrednio. Mogą być zaimplementowane za pomocą:

- **identyfikatorów**, np. liczbowe [116]

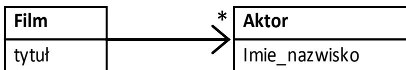
Do każdej klasy dodawany atrybut typu `int` jednoznacznie identyfikujący dany obiekt.
Info. o powiązaniach przechowujemy pamiętając id-ry. Dla asocjacji 2-kierunkowych pamiętamy pary id-rów.
Film: /Id = 1 Aktor=[3,4]/ **Aktor**: /Id = 3, Film=[1]/

- **natywnych referencji** Javy [122]

Ref-cja wskazuję bezpośrednio na obiekt, mamy natychmiastowy dostęp do niego. W zależności od liczności (1 lub *) stosujemy: pojedynczą referencje (1) lub kontener (*) (np. `ArrayList`) umieszczone w odpowiedniej z powiązanych klas.

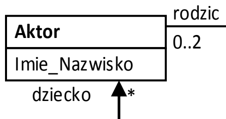
2. Asocjacja skierowana [128]

Analogicznie jak asocjacja 2-kierunkowa tylko powiązania pamiętamy dla jednej z klas. Wątpliwe jest jej zastosowanie biznesowe.



3. Asocjacja rekurencyjna [129]

W klasie umieszczane dwa kontenery po jednym dla każdej z ról, przechowujące informacje informacje z punktu widzenia każdej z ról (UML: Obowiązkowe nazwy ról.)

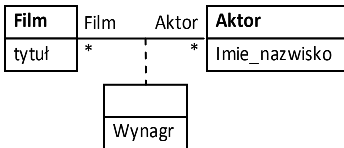



```
public class aktor {
String Imie_Nazwisko;
private ArrayList<Aktor> rodzic = new…()
private ArrayList<Aktor> dziecko = new…()
…}
```

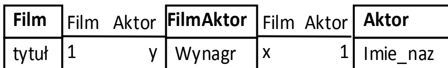
4. Asocjacja z atrybutem [130]

Inaczej „Klasa asocjacji”.

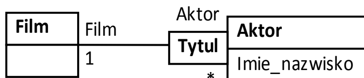
Jedną konstrukcję UML – asocjacje z atrybutem zamieniamy na inną – asocjacje z klasą pośredniczącą. Otrzymaliśmy 2 asocjacje które implementujemy na jeden ze znanych sposobów.



Po zamianie:



5. Asocjacja kwalifikowana [131]



Dostęp do obiektu docelowego odbywa się na podstawie unikatowego id-ra.

Implementacja prosta:

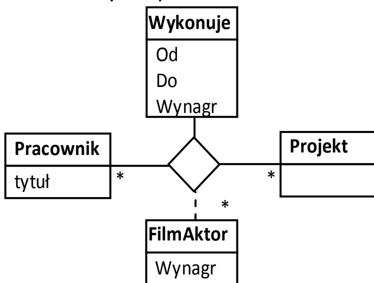
- Realizujemy jak prostą asocjacje,
- dodajemy metodę która zwraca obiekt na podstawie kwalifikatora.

Implementacja z mapującym kontenerem:

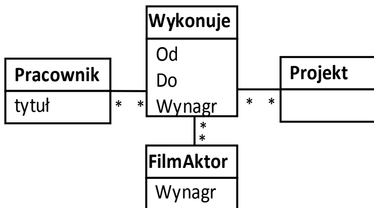
- Zamiast stand. kontenera (np. `ArrayList`) stosujemy kontener mapujący np. `TreeMap` `/TreeMap<String,Film>FilmKwalif/` gdzie kluczem jest kwalifikator, a wartością referencja do obiektu docelowego,
- Informacja zwrotna przechowywana w dotychczasowy sposób.

6. Asocjacja n-arna [134]

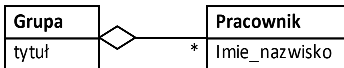
Jedną konstrukcję UML – asocjacje n-arną zamieniamy na inną – n asocjacji binarnych oraz klasę pośredniczącą. Nowe asocjacje Impl-my na jeden ze znanych sposobów.



Po zamianie:

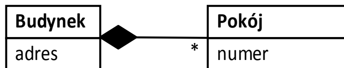


7. Agregacja [41, 135]



Implementujemy tak samo jak zwykłą asocjacje.

8. Kompozycja [135]

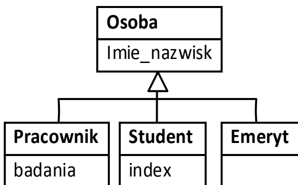


Część „asocjacyjna" Implementujemy na dotychczasowych zasadach. Do zrealizowania:

- **blokowanie samodzielnego tworzenia części** (konstuktor Części – `private`, metoda klasowa `Utworz`),
- **zakazane współdzielenie części** (atrybut klasowy /`Kontener`/ przechow. info. o wsz. częściach powiązanych z całościami),
- **usuwanie części przy usuwaniu całości** (usunięcie wszystkich części z ekstensji i wszystkich powiązań na usuwane części).

Można zmodyfikować realizacje asocjacji lub skorzystać z klas wewnętrznych.

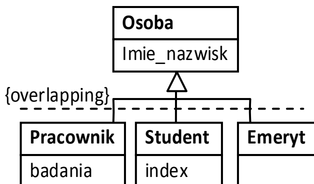
9. Dziedziczenie Disjoint [42, 155]



Występuję bezpośrednio w Java. Składnia dla dziedziczenia w Javie używa się słowa `extends`.

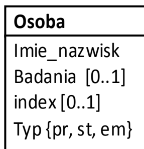
Zgodnie z zasadami hermetyzacji ukrywamy z pomocą `private` wszystkie elementy których programista nie potrzebuje do pracy.

10. Overlapping [160]



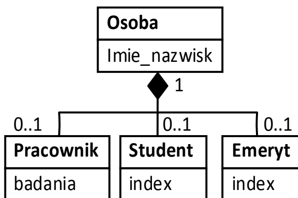
Nie występuję bezpośrednio w Java. Obejście:

- **Grupowanie — zastąpienie całej hierarchii dziedziczenia jedną klasą.** [160]



1. Wszystkie inwarianty umieszczamy w jednej klasie.
2. Dodajemy deskryminator informujący o rodzaju obiektu (typ wyliczen. `EnumSet`).

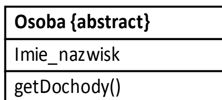
- **Wykorzystanie agregacji i kompozycji.**

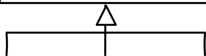


Zamiast dziedziczenia wstawiamy kompozycje, gdzie nadklasa staje się całością, a podklasy częściami. Brak bezpośredniego dostępu do metod nadklasy uzupełniamy metodami w Klasie-Części propagujące zapytanie do Klasy-Całości. Przy impl-cji kompozycji Objekty-Części nie mogą być ukryte, musimy zachować do nich dostęp.

- **Rozwiązania łączące powyższe metody.**

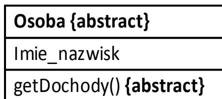
11. Klasa abstrakcyjna [43]

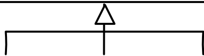




Klasa która nie może mieć bezpośrednich wystąpień (nie mogą istnieć obiekty należące do tej klasy). Może zawierać metody konkretne i abstrakcyjne. Klasę abstrakcyjną możemy stworzyć za pomocą kwalifikatora `abstract`.

12. Metoda abstrakcyjna [44]





Metoda która posiada tylko deklaracje, ale nie posiada funkcji (ciała). W Java oznaczana jest za pomocą kwalifikatora `abstract` i może być deklarowana tylko w klasie Abstrakcyjnej. Jeżeli wszystkie metody klasy są abstrakcyjne, zaleca się, aby taką klasę zadeklarować jako `Interface`. Metoda abstrakcyjna nie może być klasową `static`.

13. Atrybut Prosty [92]

Występuję bezpośrednio w Java. Składnia dla atrybutu w Javie:

```
private float cena;
```

14. Atrybut Złożony [93]

Opisywany za pomocą dedykowanej klasy, która może być dostarczana przez Javę jako biblioteka wewnętrzną lub stworzoną przez użytkownika. W klasie biznesowej przechowujemy referencje do jego wystąpienia, a nie wartość. Więc można współdzielić z innymi obiektami. Składnia dla Złożonego atrybutu w Javie identyczna jak dla Prostego, różnica jest w typie:

```
private Date dataZakupu;
```

Atrybut Wymagany/ Opcjonalny [93]

Pracownik
Badania [0..1]

Może być prosty lub złożony.

- **Atrybut Prosty** — przechowuje zawsze jakąś wartość (Wymagany). Żeby przechowywać `null` (Opcjonalny) można zaimplementować jako klasę przechowującą prostą wartość.
- **Atrybut Złożony** — przechowuje referencję do obiektu „będącą jego wartością”. Możliwa wartość `null` (Opcjonalny). Dla Wymagany – sprawdzać czy nie ma `null`.

15. Atrybut Pojedynczy [94]

Pracownik
Pensja

Jeden atrybut przechowuje jedna wartość.

16. Atrybut Powtarzalny [94]

Pracownik
Adres [1..*]

Implementujemy w jakimś kontenerze lub zwykłej tablicy. Kontener jest preferowany jeżeli nie znamy ilości elementów. Rodzaj kontenera zależy od sposobu pracy z atrybutem.

Możliwe przechowywanie w postaci `Stringu`,

Pracownik
Adresy

oddzielając poszczególne atrybuty przecinkiem – niezalecane.

17. Atrybut Klasowy [94]

Pracownik
MinPensja

Realizacja zależy od podejścia do ekstensji:

- **W ramach tej samej klasy** — stosujemy atrybuty klasowe w tej samej klasie z `static`.
- **W ramach klasy dodatkowej** — Impel-jemy w klasie dodatkowej bez ~~`static`~~.

18. Atrybut Wyliczalny [95]

Pracownik
Data urodz. / Wiek

Symulujemy w oparciu o metody. Dostęp do atrybutów i tak odbywa się za pomocą setterów i getterów. Specjalne traktowanie atrybutu implementowane jest w ciele metody udostępniającej/zmieniającej jej wartość.

19. Atrybut Unikalny

Pracownik
PESEL {unique}

- Konstruktor `private`
- metoda `utwórz(PESEL)` sprawdzająca czy w ekstensji nie ma już osoby z identycznym PESEL-em.

20. Metoda obiektowa [96]

Pracownik
Ustal pensje (kwota)

Ma taką samą semantykę jak w obiektowości. Operuję na obiekcie na rzecz którego została wywołana. Ma dostęp do wszystkich jego elementów: atrybutów i innych metod. W ciele metody możemy używać słowa kluczowego `this`, które jest referencją na obiekt na rzecz którego została wywołana metoda.

21. Metoda klasowa [96]

Pracownik
Znajdź najstarszego ()

Nie występują w Javie. Częściowo jest realizowana przez metodę static, ponieważ ekstensja nie jest realizowana bezpośrednio w Javie metoda statyczna nie ma

do niej automatycznego dostępu. W zależności od sposobu realizacji ekstensji:

- **Ekstensja w tej samej klasie** – metoda `static`, ma dostęp do kontenera z ekstensją klasy
- **Ekstensja jako klasa dodatkowa** – metoda w klasie dodatkowej dodatkowej bez ~~`static`~~.

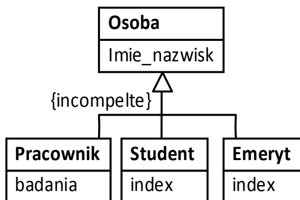
22. Asocjacja z licznością X..* [???]

1. Zamiast kontenera zwykłą tablicę z zadaną ilością „komórek” do przechowywania referencji (ID) – wtedy nie mamy fizycznej możliwości przechowywać więcej niż X referencji (ID) w ramach asocjacji.
2. Używamy za pomocą asocjacji * – * a metodzie tworzącej nowe połączenie sprawdzamy czy ilość już istniejących połączeń nie równa się maksymalnej liczności.

W różnych sytuacjach biznesowych może być wymagane stworzenie dla jednej asocjacji w ramach jednej metody połączenie z X obiektami jednocześnie:

```
klient.utworz(imie, matka, ojciec).
```

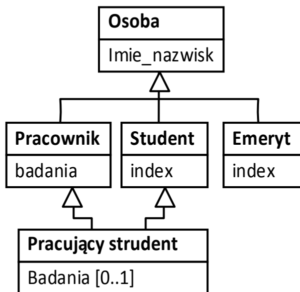
23. Dziedziczenie incomplete [167]



Raczej jest ignorowane.

Jest impl-wane jak zwykłe dziedziczenie. Jedynie trzeba pamiętać że ze względu na dalsze dziedziczenie nie możemy używać `final` – nie pozwala na dziedziczenie z takiej klasy.

24. Dziedziczenie wielokrotne [47, 168]



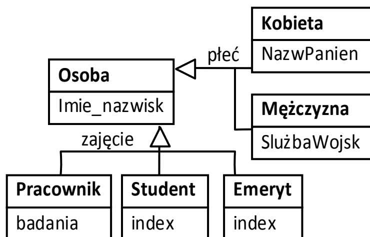
Nie występuje w Java. Implementacja taka jak dla overlapping-u:

Osoba
Imie_nazwisk Badania [0..1] index [0..1] Typ {pr, st, em}

- **Grupowanie** — Zostawiamy jedną hierarchie dziedziczenia – resztę grup-my w nadklasie. [160]
- **Grupowanie i wykorzystanie kompozycji lub agregacji.** Zostawiamy jedną hierarchie dziedziczenia – resztę impl-jemy za pomocą komp. lub agreg.
- **Wykorzystanie `interface`.** Klasa może Impel-wać dowolną ilość `interface` – co daje możliwość definiowania metod, atrybutów (tylko w postaci metod: setterów/ getterów). Wady: dot. specyficznych metod i nie pomaga w implementacji „normalnych" metod, w przypadku modyfikacji musimy ręcznie poprawiać każdą impl-cje. Wielokrotną impl-cje można obejść za pomocą **wzorców projektowych – delegacji lub polecenia** – impl-ując taką samą metodę w różnych klasach, zamiast powielać kod bezpośrednio w jej ciele, delegujemy jej

wykonanie (z wnętrza w wielu klasach) do jakiegoś (tego samego) obiektu.

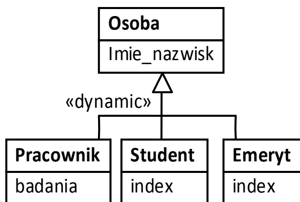
25. Dziedziczenie wieloaspektowe [49,173]



Nie występuję w żadnym języku.

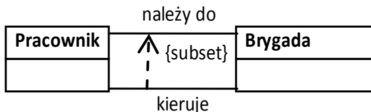
- **Grupowanie** — Zostawiamy jedną hierarchie dziedziczenia – resztę grupujemy w nadklasie i dodajemy dyskryminator. Lub grupujemy całość w nadklasie.
- **Grupowanie i wykorzystanie kompozycji lub agregacji.** Zostawiamy jedną hierarchie dziedziczenia – resztę impl-jemy za pomocą kompozycji lub agregacji.

26. Dziedziczenie dynamiczne [50, 176]



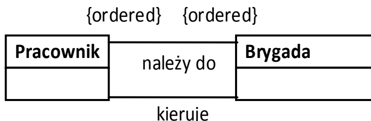
- **Grupowanie** — całość grupujemy w nadklasie i dodajemy dyskryminator.
- **Agregacja/kompozycja z XOR** — podobna implementacja jak dla overlapping-u, dodatkowo metoda ułatwiająca „zmianę klas"
- **Sprytne kopiowanie klas** — stary obiekt zastępowany nowym. W każdej z klas tworzymy konstruktorzy otrzymujące jako parametr referencje do obiektu nadklasy, z niego kopiujemy wspólne atr. do nowego obiektu. Problem z update referencji „do starego" obiektu. Również problem stanowią referencje „do" których nie ma w nowej klasie.

27. Ograniczenie subset [185]



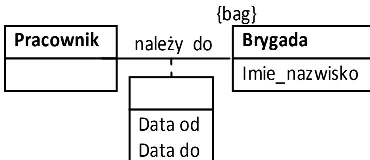
Powiązania w ramach podstawowej asocjacji „składa się" implementujemy standardowo.
A w metodzie tworzącej powiązanie z ograniczeniem, „kieruje" sprawdzamy czy istnieje podstawowe powiązanie w ramach „nadrzędnej" asocjacji.

28. Ograniczenie ordered [189]



Do przechow. elementów stosujemy odpowiedni kontener gwarantujący kolejność.

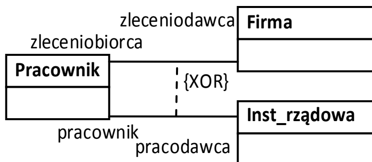
29. Ograniczenia history, bag [189]



`{bag}` – pozwala na przechowywanie więcej niż jednego powiązania z tymi samymi obiektami. Zwykle to ograniczenie pojawia się dla asocjacji z atrybutem, którą implementujemy przez klasę pośrednicząca. Ew. stosujemy kontener pozwalający pamiętać wiele razy tą samą ref. np. `vector`.

`{history}` – postępowanie identyczne, tylko że samo `history` podkreśla aspekt czasowy informacji.

30. Ograniczenie XOR [190]



Metoda dodająca powiązanie w grupie asocjacji objętych XOR sprawdza czy nie występuje już powiązanie w tej grupie.

Szczegółowo:

- Sprawdzamy czy dodawana rola jest objęta XOR.
- Sprawdzamy czy dla ról objętych ograniczeniem istnieje już powiązanie, jeżeli tak – wyjątek.
- Stosujemy zwykłą metodę tworzącą powiązanie pomiędzy obiektami.