
Polsko-Japońska Wyższa Szkoła Technik Komputerowych
Modelowanie Systemów Informacyjnych (MSI)
dr inż. Mariusz Trzaska
mtrzaska@pjwstk.edu.pl
Wykład 9
Realizacja asocjacji w obiektowych językach programowania (2)
http://www.mtrzaska.com

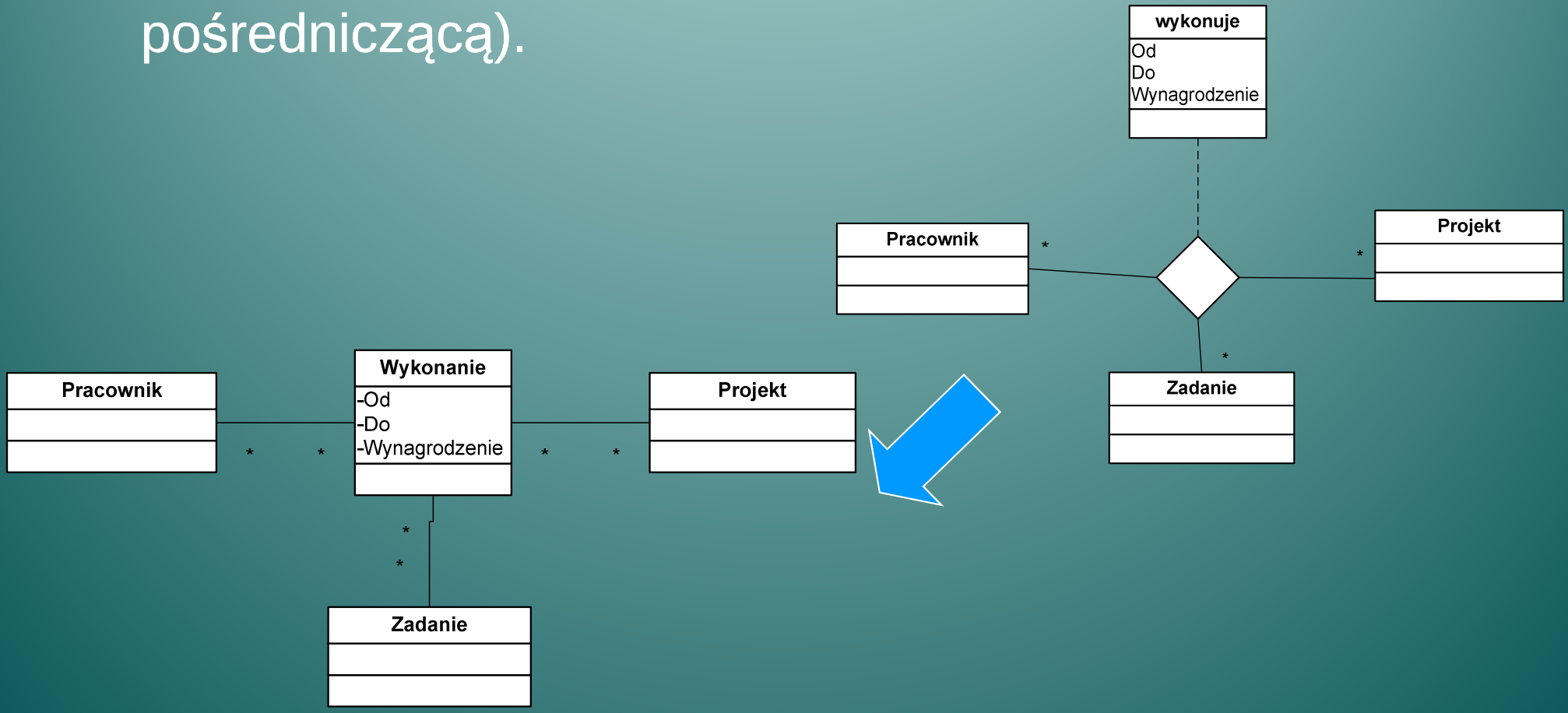
Ciąg dalszy poprzedniego wykładu

Zagadnienia

- *Wstęp teoretyczny*
- *Implementacja asocjacji:*
  - *Przy pomocy identyfikatorów,*
  - *Korzystając z natywnych referencji.*
- *Implementacja asocjacji:*
  - *ze względu na liczności,*
  - *binarnych,*
  - *z atrybutem,*
  - *kwalifikowanych,*
  - n-arnych,
- Implementacja agregacji,
- Implementacja kompozycji,
- Uniwersalne zarządzanie asocjacjami,
- Podsumowanie

Implementacja asocjacji n-arnej

- Najpierw musimy zamienić:
  - jedną konstrukcję UML (asocjacja n-arną)
  - na inną konstrukcję UML (n asocjacji binarnych oraz klasę pośredniczącą).



Implementacja asocjacji n-arnej (2)

- Dzięki zastąpieniu asocjacji n-arnej, asocjacjami binarnymi oraz klasą pośredniczącą otrzymaliśmy ***n*** zwykłych asocjacji.
- „Nowe” asocjacje implementujemy na jeden ze znanych sposobów.
- Problemy z semantyką
  - Nazwa nowej klasy,
  - Nazwy ról asocjacji: „starych” oraz „nowych”,
  - Liczności asocjacji.
- Utrudniony dostęp do obiektów docelowych (poprzez obiekt klasy pośredniczącej)

Implementacja agregacji

- Czy zastosowanie agregacji niesie jakieś konsekwencje dla zaangażowanych obiektów?
- **Nie**!
- W związku z powyższym agregacje implementujemy dokładnie tak samo jak klasyczne asocjacje.

Implementacja kompozycji

- Część „asocjacyjna” realizowana na dotychczasowych zasadach.
- Problemy do rozwiązania:
  1. Blokowanie samodzielnego tworzenia części,
  2. Zakazanie współdzielenia części,
  3. Usuwanie części przy usuwaniu całości.
- Możliwe dwa podejścia:
  - Zmodyfikowanie istniejącego rozwiązania,
  - Wykorzystanie klas wewnętrznych.

Implementacja kompozycji (2)

1. Blokowanie samodzielnego tworzenia części (istnienia części bez całości),
   - Prywatny konstruktor,
   - Dedykowana metoda (klasowa):
     - pobierająca referencję do całości (i sprawdzająca czy jest ona prawidłowa),
     - tworząca obiekt części,
     - dodające informacje o powiązaniu zwrotnym.

Implementacja kompozycji (3)

```
public class Czesc {
	public String nazwa;// public - dla uproszczenia
	private Calosc calosc;
	private Czesc(Calosc calosc, String nazwa) {
		this.nazwa = nazwa;
		this.calosc =calosc;
	}

	public static Czesc utworzCzesc(Calosc calosc, String nazwa) throws Exception
	{
		if(calosc == null) {
			throw new Exception("Calosc nie istnieje!");
		}

		// Utwocz nowa czesc
		Czesc cz = new Czesc(calosc, nazwa);

		// Dodaj do calosci
		calosc.dodajCzesc(cz);

		return cz;
	}
}
```

Implementacja kompozycji (4)

```
public class Calosc {
     private Vector<Czesc> czesci = new Vector<Czesc>();
     private String nazwa;

     public Calosc(String nazwa) {
          this.nazwa = nazwa;
     }

     public void dodajCzesc(Czesc czesc) {
          if(!czesci.contains(czesc)) {
               czesci.add(czesc);
          }
     }

     public String toString() {
          String info = "Calosc: " + nazwa + "\n";
          for(Czesc cz : czesci) {
               info += "   " + cz.nazwa + "\n";
          }

          return info;
     }
}
```

Implementacja kompozycji (5)

2. Zakazanie współdzielenia części

- Zmodyfikowana wersja metody dodającej część
  - Sprawdzająca czy dana część nie jest już gdzieś dodana,
  - Oprócz dodawania informacji o powiązaniu z podaną częścią, zapamiętuje (globalnie) fakt, że dana część jest już powiązana z całością.
- Atrybut klasowy przechowujący informacje o wszystkich częściach powiązanych z całościami.

Implementacja kompozycji (6)

- Specjalna wersja metody dodającej część

```
public class Calosc {
     private Vector<Czesc> czesci = new Vector<Czesc>();
     private static HashSet<Czesc> wszystkieCzesci = new HashSet<Czesc>();
     // [...]
     public void dodajCzesc(Czesc czesc) throws Exception {
          if(!czesci.contains(czesc)) {
               // Sprawdz czy ta czesc nie zostala dodana do jakiejs calosci
               if(wszystkieCzesci.contains(czesc)) {
                    throw new Exception("Ta czesc jest juz powiazana z caloscia!");
               }

               czesci.add(czesc);

               // Zapamietaj na liscie wszystkich czesci (przeciwdziala wspoldzielniu czesci)
               wszystkieCzesci.add(czesc);
          }
     }
     // [...]
}
```

Implementacja kompozycji (7)

3. Usuwanie części przy usuwaniu całości.

- W językach typu Java oraz C#
  - nie ma możliwości ręcznego usuwania obiektu,
  - obiekt jest usuwany przez VM gdy nie jest osiągalny (nie ma do niego żadnych referencji).
- W C++ mamy możliwość ręcznego usunięcia obiektu. Polecenia usunięcia części warto umieścić w destruktorze. Dzięki temu całość jest w miarę zautomatyzowana.

Implementacja kompozycji (8)

3. Usuwanie części przy usuwaniu całości – c. d.

- W przypadku naszej implementacji, warto:
  - Stworzyć metodę (klasową) usuwającą całość z ekstensji,
  - Powyższa metoda powinna również zadbać o usunięcie informacji z globalnej listy części (przeciwdziałającej współdzieleniu).

Implementacja kompozycji przy pomocy klas wewnętrznych

- Obiekt klasy wewnętrznej nie może istnieć bez (otaczającego go) obiektu klasy zewnętrznej.
- Obiekt klasy wewnętrznej ma bezpośredni dostęp do inwariantów obiektu klasy zewnętrznej.
- Poniższy kod nie wywołuje błędu, ale jego efekt nie jest taki jaki byśmy chcieli.
  - Obiekt klasy wewnętrznej (Czesc) ma dostęp do obiektu klasy zewnętrznej (Calosc) – to dobrze,
  - Obiekt klasy zewnętrznej (Calosc) nic nie wie, że utworzono obiekt klasy wewnętrznej (Czesc) – a to już bardzo źle.

```
// Bledny efekt: utworzenie nowej czesci w kontekscie istniejacej calosci,
// ale bez informowania o tym calosci.
Calosc.Czesc cz = c1.new Czesc("Czesc 02");
```

Implementacja kompozycji przy pomocy klas wewnętrznych (2)

- Aby temu przeciwdziałać:
  - klasa wewnętrzna powinna mieć prywatny konstruktor,
  - klasa zewnętrzna musi dostarczać dedykowaną metodą zapewniającą właściwe utworzenie obiektów-części.
- Należy ręcznie zadbać o:
  - Blokowanie współdzielenia części. Część zawsze jest połączona tylko z jednym obiektem-całością. Niemniej, może się zdarzyć, że różne całości byłyby połączone (pokazywałyby na) z tą samą częścią.
  - Usuwanie części przy usuwaniu całości (podobnie jak w przypadku poprzedniego sposobu implementacji kompozycji).

Implementacja kompozycji przy pomocy klas wewnętrznych (3)

```
public class Calosc {
     private String nazwaCalosci;
     private Vector<Czesc> czesci = new Vector<Czesc>();
     public Calosc(String nazwa) {
          nazwaCalosci = nazwa;
     }
     public Czesc utworzCzesc(String nazwa) {
          Czesc czesc = new Czesc(nazwa);
          czesci.add(czesc);
          return czesc;
     }

     // Klasa wewnetrzna – czesc.
     public class Czesc {
          private String nazwaCzesci;
          // Ze wzgledu na specyfike klas wewnetrznych, nie potrzebujemy referencji
          pokazujacej na calosc.

          public Czesc(String nazwa) {
               nazwaCzesci = nazwa;
          }
     }
}
```

Zarządzanie asocjacjami

- Przedstawione sposoby implementacji zarządzania asocjacjami będą (prawie) takie same dla każdej biznesowej klasy w systemie.
- Co więcej, będą (prawie) takie same dla każdej asocjacji nawet w tej samej klasie.
- Czy da się to jakoś zunifikować? Aby nie pisać wiele razy (prawie) tego samego kodu?
- Oczywiście – podobnie jak przy okazji zarządzania ekstensją, wykorzystamy dziedziczenie istniejące w języku Java.

Uniwersalne zarządzanie asocjacjami

- Przy okazji zarządzania ekstensją stworzyliśmy nowa klasę `ObjectPlus`.
- Aby nie stracić zawartej tam funkcjonalności, stworzymy nową klasę `ObjectPlusPlus` dziedziczącą z `ObjectPlus`.
- Dzięki temu, istniejącą funkcjonalność w zakresie ekstensji, uzupełnimy o ułatwienia w zarządzaniu asocjacjami.

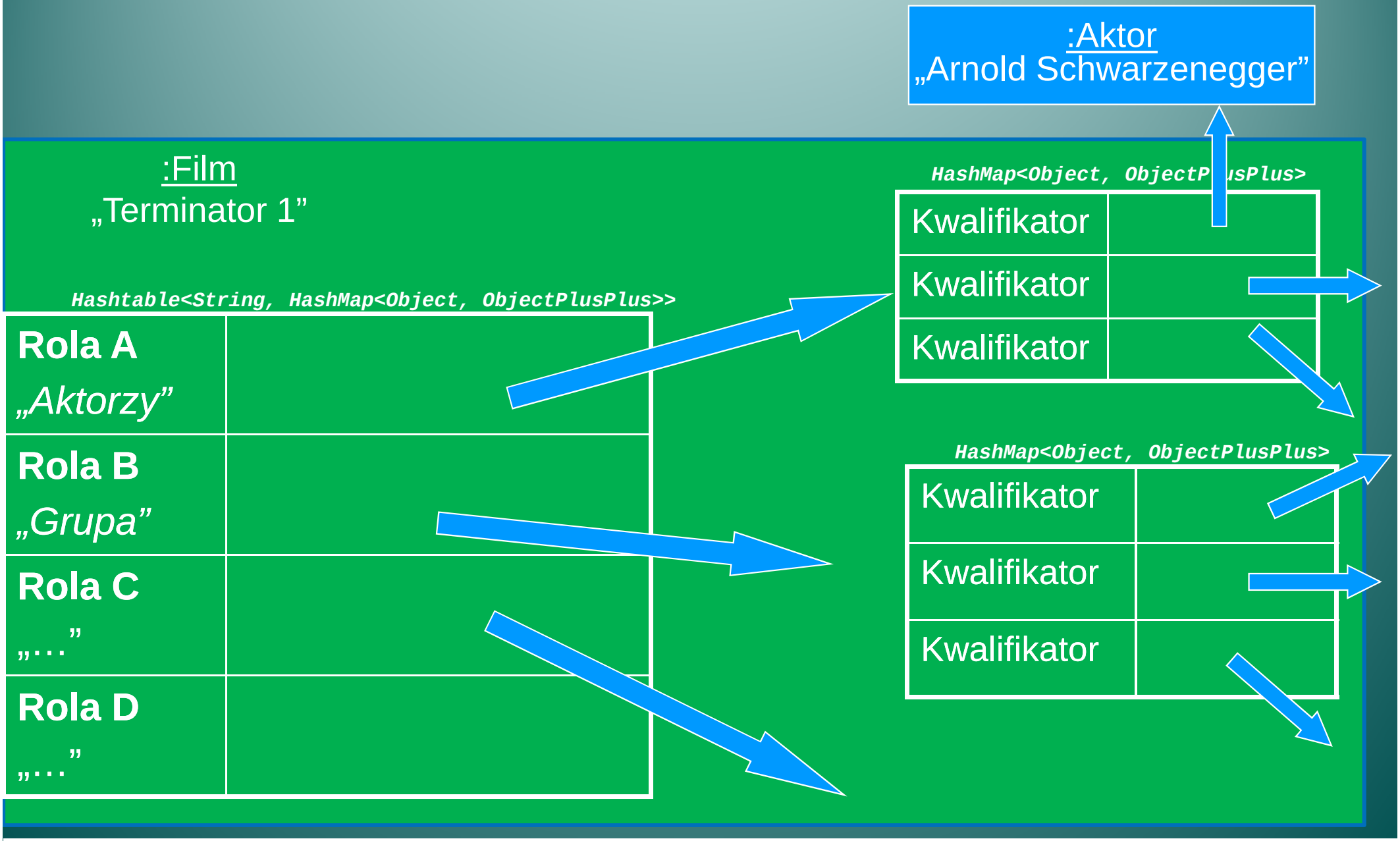
Uniwersalne zarządzanie asocjacjami (2)

- Stworzymy klasę z której będą dziedziczyć wszystkie biznesowe klasy w naszej aplikacji.
- Nazwijmy ją `ObjectPlusPlus` i wyposażmy w funkcjonalność ułatwiającą zarządzanie:
  - zwykłymi asocjacjami binarnymi,
  - asocjacjami kwalifikowanymi,
  - kompozycjami (częściowo – tylko warunek nr 2).
- Zastosujemy drugie z omawianych podejść do implementacji asocjacji: w oparciu o referencje.

Uniwersalne asocjacje

- Ponieważ wszystkie asocjacje w ramach jednego obiektu będą przechowywane w jednej kolekcji, nie możemy zastosować zwykłego pojemnika typu `Vector` czy `ArrayList`.
- Użyjemy kontenera przechowującego klucze i wartości:
  - Kluczem będzie nazwa roli asocjacji,
  - Wartością mapa zawierająca:
    - Klucz będący kwalifikatorem. Gdy nie chcemy użyć asocjacji kwalifikowanej, kwalifikator będzie tożsamy z obiektem docelowym.
    - Wartość - referencje do konkretnego powiązania.
- Atrybut klasowy przechowujący referencje do wszystkich obiektów dodanych jako części, umożliwi pilnowanie warunku nr 2 dotyczącego kompozycji (brak współdzielenia),
- Innymi słowy, ten nowy kontener będzie zawierał powiązania istniejące w ramach wielu asocjacji.

Uniwersalne asocjacje (2)



Klasa ObjectPlusPlus

```
public class ObjectPlusPlus extends ObjectPlus implements Serializable {
     /**
      * Przechowuje informacje o wszystkich powiazaniach tego obiektu.
      */
     private Hashtable<String, HashMap<Object, ObjectPlusPlus>> powiazania = new
          Hashtable<String, HashMap<Object, ObjectPlusPlus>>();


     /**
      * Przechowuje informacje o wszystkich czesciach powiazanych z ktorymkolwiek z
     obiektow.
      */
     private static HashSet<ObjectPlusPlus> wszystkieCzesci = new
          HashSet<ObjectPlusPlus>();


     /**
      * Konstruktor.
      *
      */
     public ObjectPlusPlus() {
          super();
     }

     // [...]
```

Klasa ObjectPlusPlus (2)

```
public class ObjectPlusPlus extends ObjectPlus implements Serializable {
     // [...]
     private void dodajPowiazanie(String nazwaRoli, String odwrotnaNazwaRoli,
          ObjectPlusPlus obiektDocelowy, Object kwalifikator, int licznik) {
          HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;

          if(licznik < 1) {
               return;
          }
          if(powiazania.containsKey(nazwaRoli)) {
               // Pobierz te powiazania
               powiazaniaObiektu = powiazania.get(nazwaRoli);
          }
          else {
               // Brak powiazan dla takiej roli ==> utworz
               powiazaniaObiektu = new HashMap<Object, ObjectPlusPlus>();
               powiazania.put(nazwaRoli, powiazaniaObiektu);
          }
          if(!powiazaniaObiektu.containsKey(kwalifikator)) {
               // Dodaj powiazanie dla tego obiektu
               powiazaniaObiektu.put(kwalifikator, obiektDocelowy);
               // Dodaj powiazanie zwrotne
               obiektDocelowy.dodajPowiazanie(odwrotnaNazwaRoli, nazwaRoli, this, this,
               licznik - 1);
          }}
```

Klasa ObjectPlusPlus (3)

```
public class ObjectPlusPlus extends ObjectPlus implements Serializable {
    // [...]
    public void dodajPowiazanie(String nazwaRoli, String odwrotnaNazwaRoli,
        ObjectPlusPlus obiektDocelowy, Object kwalifikator) {
        dodajPowiazanie(nazwaRoli, odwrotnaNazwaRoli, obiektDocelowy, kwalifikator, 2);
    }
    public void dodajPowiazanie(String nazwaRoli, String odwrotnaNazwaRoli,
        ObjectPlusPlus obiektDocelowy) {
        dodajPowiazanie(nazwaRoli, odwrotnaNazwaRoli, obiektDocelowy, obiektDocelowy);
    }
    public void dodajCzesc(String nazwaRoli, String odwrotnaNazwaRoli, ObjectPlusPlus
        obiektCzesc) throws Exception {
        // Sprawdz czy ta czesc juz gdzies nie wystepuje
        if(wszystkieCzesci.contains(obiektCzesc)) {
            throw new Exception("Ta czesc jest już powiazana z jakas caloscia!");
        }

        dodajPowiazanie(nazwaRoli, odwrotnaNazwaRoli, obiektCzesc);

        // Zapamietaj dodanie obiektu jako czesci
        wszystkieCzesci.add(obiektCzesc);
    }
    // [...]
```

Klasa ObjectPlusPlus (4)

```
public class ObjectPlusPlus extends ObjectPlus implements Serializable {
     // [...]
     public ObjectPlusPlus[] dajPowiazania(String nazwaRoli) throws Exception {
          HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;
          if(!powiazania.containsKey(nazwaRoli)) {
               // Brak powiazan dla tej roli
               throw new Exception("Brak powiazan dla roli: " + nazwaRoli);
          }
          powiazaniaObiektu = powiazania.get(nazwaRoli);
          return (ObjectPlusPlus[]) powiazaniaObiektu.values().toArray(new ObjectPlusPlus[0]);
     }

     public void wyswietlPowiazania(String nazwaRoli, PrintStream stream) throws Exception {
          HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;
          if(!powiazania.containsKey(nazwaRoli)) {
               // Brak powiazan dla tej roli
               throw new Exception("Brak powiazan dla roli: " + nazwaRoli);
          }
          powiazaniaObiektu = powiazania.get(nazwaRoli);
          Collection col = powiazaniaObiektu.values();
          stream.println(this.getClass().getSimpleName() + " powiazania w roli " +
                    nazwaRoli + ":");
          for(Object obj : col) {
               stream.println("   " + obj);
          }
     }
     // [...]
```

Klasa ObjectPlusPlus (5)

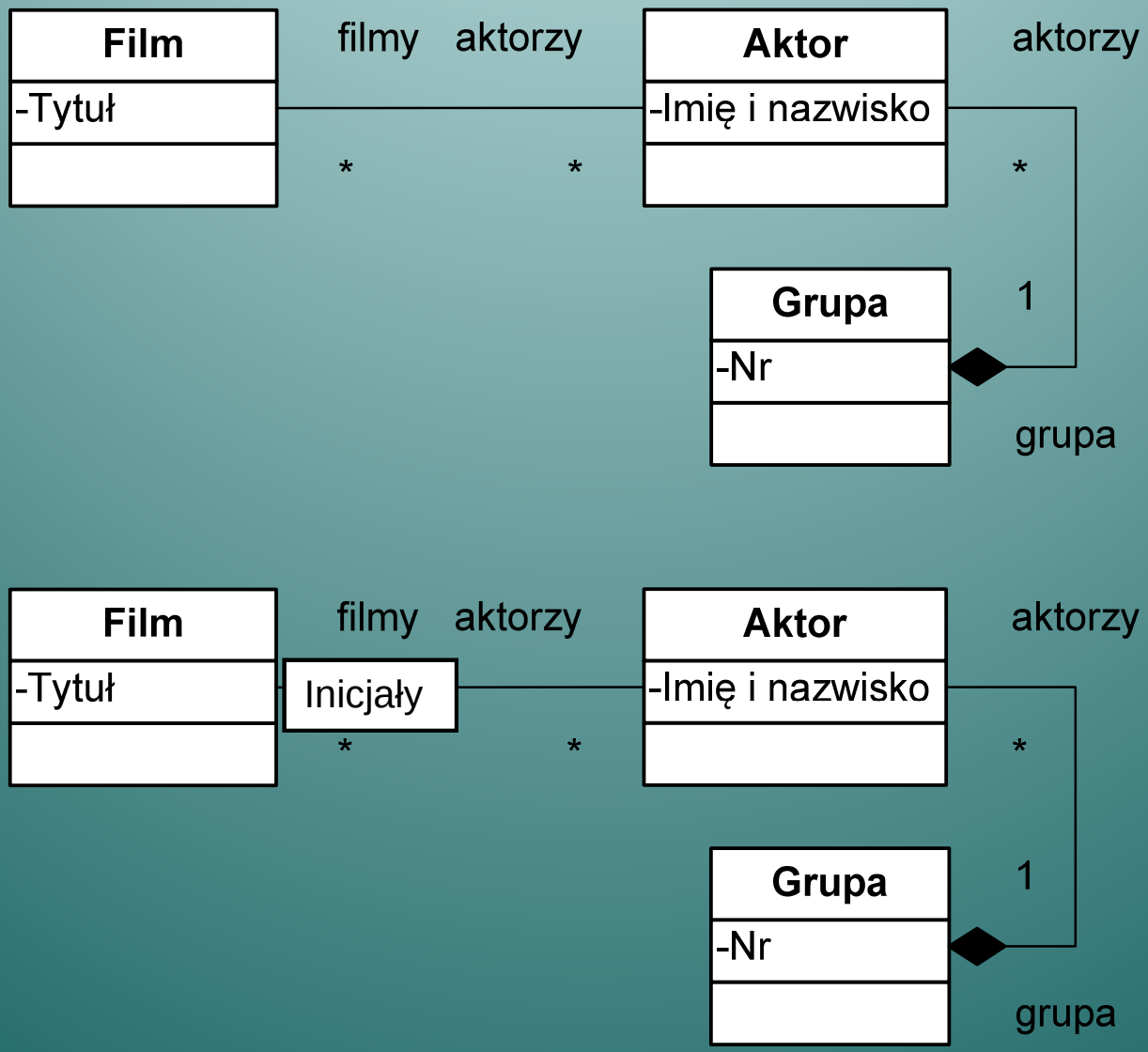
```
public class ObjectPlusPlus extends ObjectPlus implements Serializable {
     // [...]
     public ObjectPlusPlus dajPowiazanyObiekt(String nazwaRoli, Object kwalifikator) throws
          Exception {
          HashMap<Object, ObjectPlusPlus> powiazaniaObiektu;

          if(!powiazania.containsKey(nazwaRoli)) {
               // Brak powiazan dla tej roli
               throw new Exception("Brak powiazan dla roli: " + nazwaRoli);
          }

          powiazaniaObiektu = powiazania.get(nazwaRoli);
          if(!powiazaniaObiektu.containsKey(kwalifikator)) {
               // Brak powiazan dla tej roli
               throw new Exception("Brak powiazania dla kwalifikatora: " + kwalifikator);
          }

          return powiazaniaObiektu.get(kwalifikator);
     }
}
```

Klasy biznesowe do zaimplementowania



Implementacja klas biznesowych korzystająca z `ObjectPlusPlus`

```
public class Aktor extends ObjectPlusPlus {
     private String imieNazwisko;

     public Aktor(String imieNazwisko) {
          super();// wolanie konstruktora z nadklasy - obowiazkowe!
          this.imieNazwisko = imieNazwisko;
     }

     public String toString() {
          return "Aktor: " + imieNazwisko;
     }
}
```

```
public class Film extends ObjectPlusPlus {
     private String tytul;

     public Film(String tytul) {
          super();// wolanie konstruktora z nadklasy - obowiazkowe!
          this.tytul = tytul;
     }

     public String toString() {
          return "Film: " + tytul;
     }
}
```

Implementacja klas biznesowych korzystająca z `ObjectPlusPlus` (2)

```
public class Grupa extends ObjectPlusPlus {
     private int nr;

     public Grupa(int nr) {
          super();// wolanie konstruktora z nadklasy - obowiazkowe!
          this.nr = nr;
     }

     public String toString() {
          return "Grupa: " + nr;
     }
}
```

Przykład wykorzystania klas biznesowych korzystających z `ObjectPlusPlus`

```
// Utworz nowe obiekty (na razie bez informacji o powiazaniach)
Aktor a1 = new Aktor("Arnold Schwarzenegger");
Aktor a2 = new Aktor("Michael Biehn");
Aktor a3 = new Aktor("Kristanna Loken");

Film f1 = new Film("Terminator 1");
Film f3 = new Film("Terminator 3");

Grupa g1 = new Grupa(1);
Grupa g2 = new Grupa(2);

// Dodaj informacje o powiazaniach
f1.dodajPowiazanie("aktorzy", "filmy", a1);
// f1.dodajPowiazanie("aktorzy", "filmy", a2);
f1.dodajPowiazanie("aktorzy", "filmy", a2, "MB");// wykorzystanie asocjacji kwalifikowanej
f3.dodajPowiazanie("aktorzy", "filmy", a1);
f3.dodajPowiazanie("aktorzy", "filmy", a3);

g1.dodajCzesc("czesc", "calosc", a1);
g1.dodajCzesc("czesc", "calosc", a2);
g2.dodajCzesc("czesc", "calosc", a3);
// g2.dodajCzesc("czesc", "calosc", a1); // wyjatek poniewaz dodawana czesc (aktor) nalezy
                                         juz do innej calosci (grupy)
```

Przykład wykorzystania klas biznesowych korzystających z `ObjectPlusPlus` (2)

Zadanie

- Co jest potencjalnie złego (niebezpiecznego) w tym podejściu?
- Jak można to poprawić?

```
Film powiazania w roli aktorzy:
   Aktor: Arnold Schwarzenegger
   Aktor: Michael Biehn
Film powiazania w roli aktorzy:
   Aktor: Arnold Schwarzenegger
   Aktor: Kristanna Loken
Aktor powiazania w roli filmy:
   Film: Terminator 1
   Film: Terminator 3
Grupa powiazania w roli czesc:
   Aktor: Arnold Schwarzenegger
   Aktor: Michael Biehn
Aktor: Michael Biehn
```

```
// Wyswietl informacje
f1.wyswietlPowiazania("aktorzy", System.out);
f3.wyswietlPowiazania("aktorzy", System.out);

a1.wyswietlPowiazania("filmy", System.out);

g1.wyswietlPowiazania("czesc", System.out);

// test asocjacji kwalifikowanej
System.out.println(f1.dajPowiazanyObiekt("aktorzy", "MB"));
```

Podsumowanie

- Mamy dwa generalne podejścia do implementacji asocjacji:
  - Identyfikatory,
  - Natywne referencje.
- Niektóre rodzaje asocjacji, przed ich implementacją należy zamienić na konstrukcje równoważne.
- Dzięki temu implementujemy je korzystając ze znanych już sposobów.
- Całą funkcjonalność związaną z zarządzaniem asocjacjami, warto zgromadzić w specjalnej nadklasie (`ObjectPlusPlus`).
- Dzięki temu, że nowa klasa (`ObjectPlusPlus`) dziedziczy z `ObjectPlus`, obsługuje również zarządzanie ekstensjami.