



Modelowanie i Analiza Systemów informacyjnych (MAS)

dr inż. Mariusz Trzaska
mtrzaska@pjwstk.edu.pl

Wykład 6

Realizacja asocjacji w obiektowych językach programowania (2)

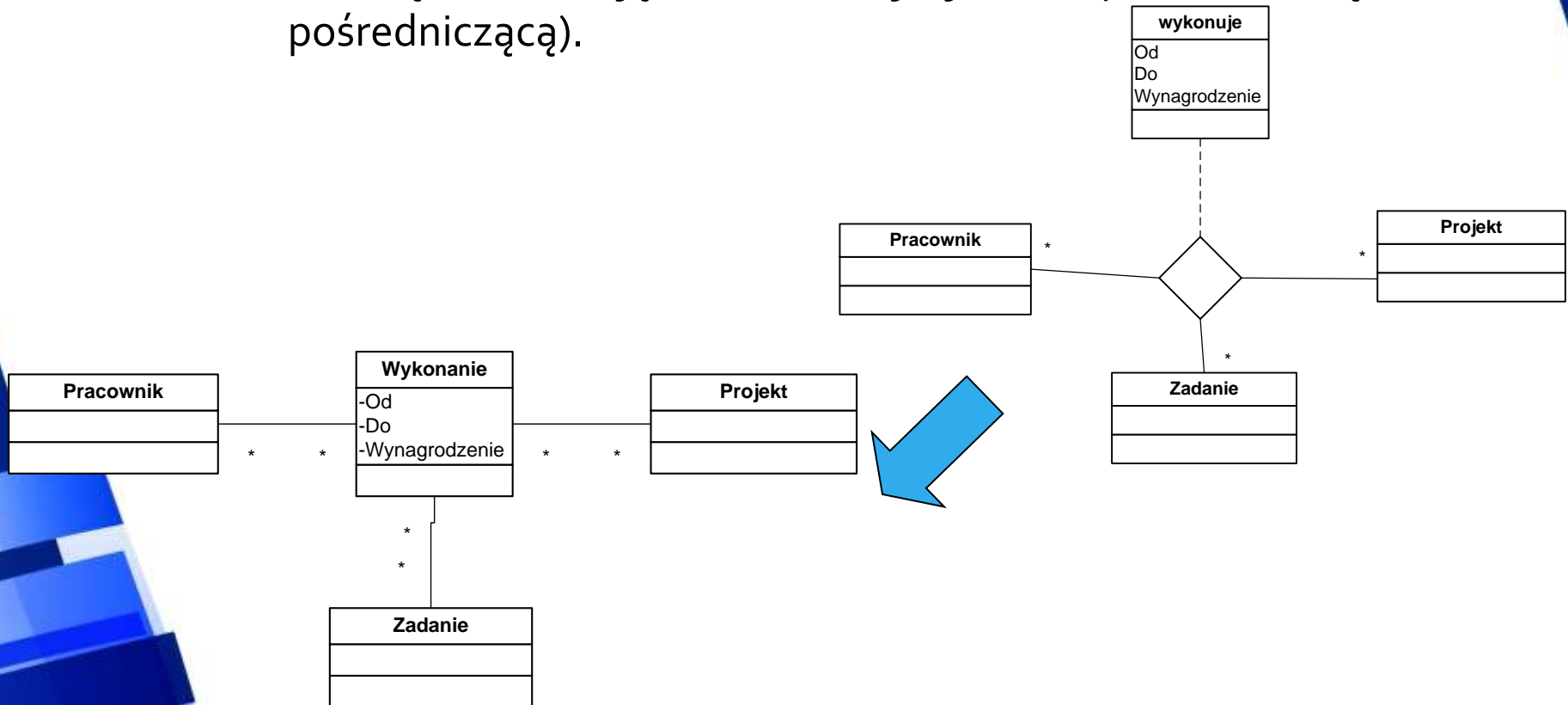
- Ciąg dalszy poprzedniego wykładu

Zagadnienia

- *Wstęp teoretyczny*
- *Implementacja asocjacji:*
 - *Przy pomocy identyfikatorów,*
 - *Korzystając z natywnych referencji.*
- *Implementacja asocjacji:*
 - *ze względu na licznosci,*
 - *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-arna)
 - 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:
 - Blokowanie samodzielnego tworzenia części,
 - Zakazanie współdzielenia części,
 - 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 Part {
    public String name;    // public for simplicity
    private Whole whole;

    private Part(Whole whole, String name) {
        this.name = name;
        this.whole = whole;
    }

    public static Part createPart(Whole whole, String name) throws Exception {
        if(whole == null) {
            throw new Exception("The given whole does not exist!");
        }

        // Create a new part
        Part part = new Part(whole, name);

        // Add to the whole
        whole.addPart(part);

        return part;
    }
}
```

Implementacja kompozycji (4)

```
public class Whole {
    private List<Part> parts = new ArrayList<>();

    private String name;

    public Whole(String name) {
        this.name = name;
    }

    public void addPart(Part part) throws Exception {
        if(!parts.contains(part)) {
            parts.add(part);
        }
    }

    @Override
    public String toString() {
        String info = "Whole: " + name + "\n";
        for(Part part : parts) {
            info += "    " + part.name + "\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 Whole {
    private List<Part> parts = new ArrayList<>();
    private static Set<Part> allParts = new HashSet<>();
    // [...]

    public void addPart(Part part) throws Exception {
        if(!parts.contains(part)) {
            // Check if the part has been already added to any wholes
            if(allParts.contains(part)) {
                throw new Exception("The part is already connected with a whole!");
            }

            parts.add(part);

            // Store on the list of all parts
            allParts.add(part);
        }
    }

    @Override
    public String toString() {
        String info = "Whole: " + name + "\n";
        for(Part part : parts) {
            info += "    " + part.name + "\n";
        }

        return info;
    }
}
```

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.

```
// Create a new whole
Whole whole = new Whole("Whole 01");
// Create a new part
Whole.Part part = whole.new Part("Part 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 Whole {
    private String wholeName;
    private List<Part> parts = new ArrayList<>();

    public Whole(String wholeName) {
        this.wholeName = wholeName;
    }

    public Part createPart(String partName) {
        Part part = new Part(partName);
        parts.add(part);

        return part;
    }

    @Override
    public String toString() { return wholeName; }

    // The inner class (the part)
    public class Part {
        private String partName;
        // Because of Java inner class properties, we do not need a reference pointing at the whole.

        public Part(String partName) {
            this.partName = partName;
        }

        @Override
        public String toString() {
            return "Part: " + partName;
        }
    }
}
```

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 nową 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

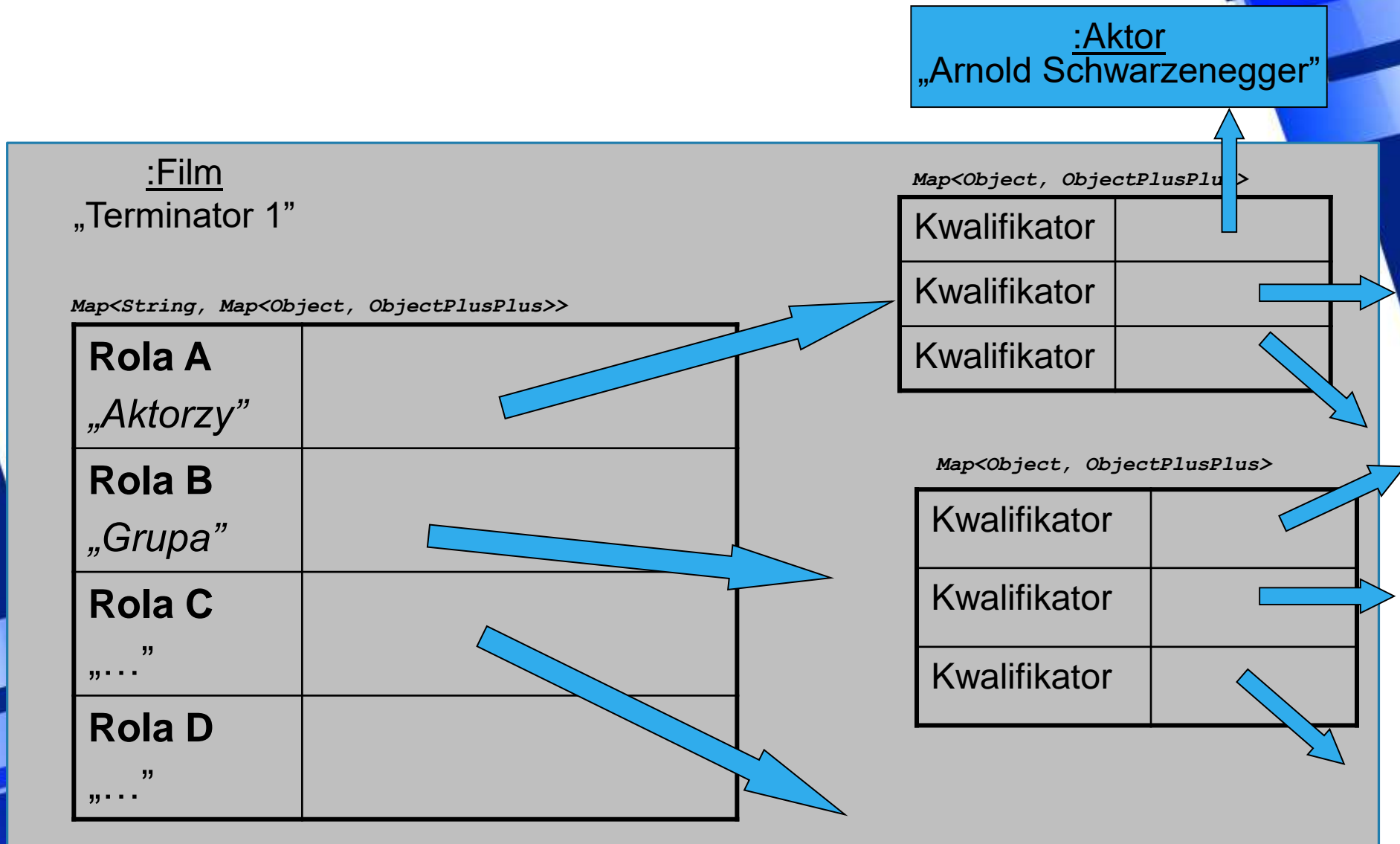
(2)

- Stworzymy klasę z której będą dziedziczyć wszystkie biznesowe klasy w naszej aplikacji.
- Nazwijmy ją *ObjectPlusPlus* i wyposażymy 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 abstract class ObjectPlusPlus extends ObjectPlus implements Serializable {
    /**
     * Stores information about all connections of this object.
     */
    private Map<String, Map<Object, ObjectPlusPlus>> links = new Hashtable<>();

    /**
     * Stores information about all parts connected with any objects.
     */
    private static Set<ObjectPlusPlus> allParts = new HashSet<>();

    /**
     * The constructor.
     *
     */
    public ObjectPlusPlus() {
        super();
    }

    // [...]
}
```

Klasa ObjectPlusPlus (2)

```
public abstract class ObjectPlusPlus extends ObjectPlus implements Serializable {
    private Map<String, Map<Object, ObjectPlusPlus>> links = new Hashtable<>();
    // [...]

    private void addLink(String roleName, String reverseRoleName, ObjectPlusPlus targetObject, Object qualifier, int counter) {
        Map<Object, ObjectPlusPlus> objectLinks;

        // Protection for the reverse connection
        if(counter < 1) {
            return;
        }

        // Find a collection of links for the role
        if(links.containsKey(roleName)) {
            // Get the links
            objectLinks = links.get(roleName);
        }
        else {
            // No links ==> create them
            objectLinks = new HashMap<>();
            links.put(roleName, objectLinks);
        }

        // Check if there is already the connection
        // If yes, then ignore the creation
        if(!objectLinks.containsKey(qualifier)) {
            // Add a link for the target object
            objectLinks.put(qualifier, targetObject);

            // Add the reverse connection
            targetObject.addLink(reverseRoleName, roleName, this, this, counter - 1);
        }
    }
    // [...]
```


Klasa ObjectPlusPlus (3)

```
public abstract class ObjectPlusPlus extends ObjectPlus implements Serializable {
    /**
     * Stores information about all connections of this object.
     */
    private Map<String, Map<Object, ObjectPlusPlus>> links = new Hashtable<>();

    /**
     * Stores information about all parts connected with any objects.
     */
    private static Set<ObjectPlusPlus> allParts = new HashSet<>();

    public void addLink(String roleName, String reverseRoleName, ObjectPlusPlus targetObject, Object qualifier) {
        addLink(roleName, reverseRoleName, targetObject, qualifier, 2);
    }

    public void addLink(String roleName, String reverseRoleName, ObjectPlusPlus targetObject) {
        addLink(roleName, reverseRoleName, targetObject, targetObject);
    }

    public void addPart(String roleName, String reverseRoleName, ObjectPlusPlus partObject) throws Exception {
        // Check if the part exist somewhere
        if(allParts.contains(partObject)) {
            throw new Exception("The part is already connected to a whole!");
        }

        addLink(roleName, reverseRoleName, partObject);

        // Store adding the object as a part
        allParts.add(partObject);
    }
    // [...]
}
```

Klasa ObjectPlusPlus (4)

```
public abstract class ObjectPlusPlus extends ObjectPlus implements Serializable {
    private Map<String, Map<Object, ObjectPlusPlus>> links = new Hashtable<>();

    // [...]

    public ObjectPlusPlus[] getLinks(String roleName) throws Exception {
        Map<Object, ObjectPlusPlus> objectLinks;

        if(!links.containsKey(roleName)) {
            // No links for the role
            throw new Exception("No links for the role: " + roleName);
        }

        objectLinks = links.get(roleName);
        return (ObjectPlusPlus[]) objectLinks.values().toArray(new ObjectPlusPlus[0]);
    }

    public void showLinks(String roleName, PrintStream stream) throws Exception {
        Map<Object, ObjectPlusPlus> objectLinks;

        if(!links.containsKey(roleName)) {
            // No links
            throw new Exception("No links for the role: " + roleName);
        }

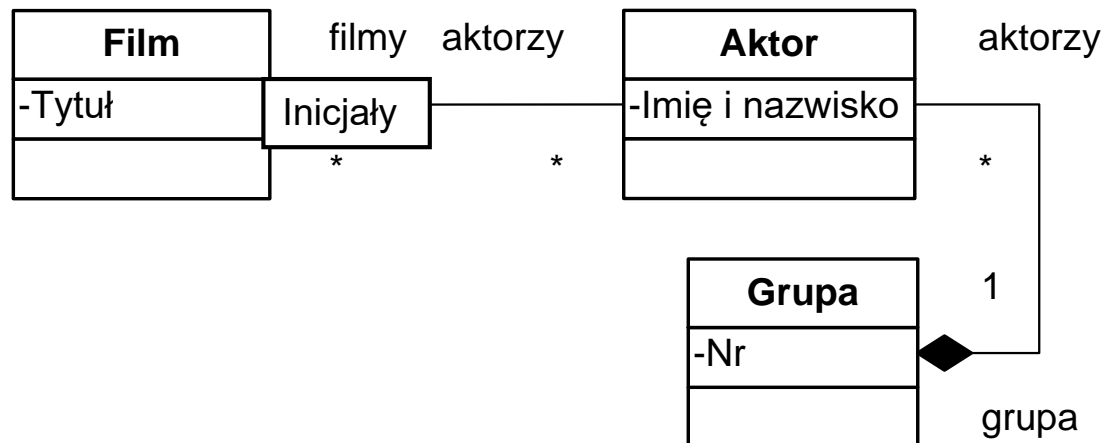
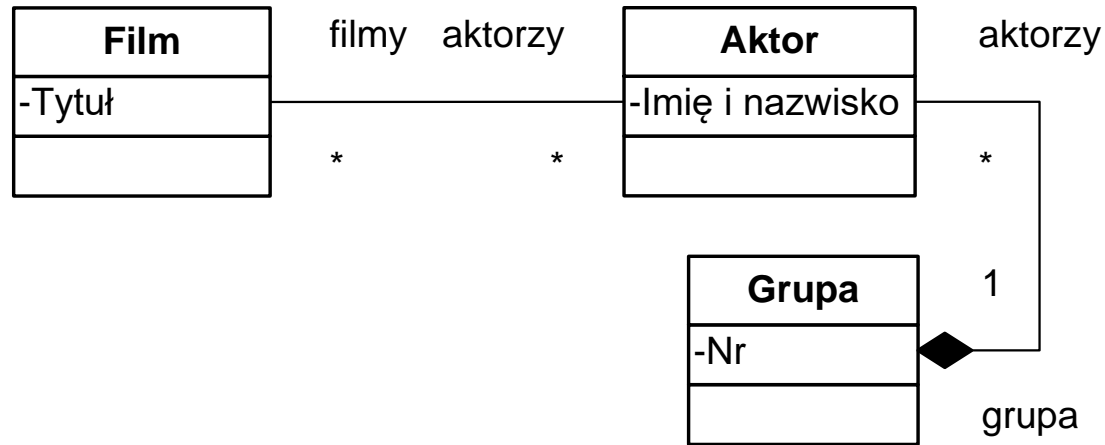
        objectLinks = links.get(roleName);
        Collection col = objectLinks.values();
        stream.println(this.getClass().getSimpleName() + " links, role '" + roleName + "':");

        for(Object obj : col) {
            stream.println("    " + obj);
        }
    }
}
```

Klasa ObjectPlusPlus (5)

```
public abstract class ObjectPlusPlus extends ObjectPlus implements Serializable {  
    private Map<String, Map<Object, ObjectPlusPlus>> links = new Hashtable<>();  
  
    // [...]  
  
    public ObjectPlusPlus getLinkedObject(String roleName, Object qualifier) throws Exception {  
        Map<Object, ObjectPlusPlus> objectLinks;  
  
        if(!links.containsKey(roleName)) {  
            // No links  
            throw new Exception("No links for the role: " + roleName);  
        }  
  
        objectLinks = links.get(roleName);  
        if(!objectLinks.containsKey(qualifier)) {  
            // No link for the qualifer  
            throw new Exception("No link for the qualifer: " + qualifier);  
        }  
  
        return objectLinks.get(qualifier);  
    }  
}
```

Klasy biznesowe do zaimplementowania



Implementacja klas biznesowych korzystająca z ObjectPlusPlus

```
public class Actor extends ObjectPlusPlus {
    private String name;

    public Actor(String name) {
        super(); // call the super constructor
        this.name = name;
    }

    @Override
    public String toString() {
        return "Actor: " + name;
    }
}
```

```
public class Movie extends ObjectPlusPlus {
    private String title;

    public Movie(String title) {
        super(); // call the super constructor
        this.title = title;
    }

    @Override
    public String toString() {
        return "Movie: " + title;
    }
}
```

Implementacja klas biznesowych korzystająca z ObjectPlusPlus (2)

```
public class Group extends ObjectPlusPlus {
    private int number;

    public Group(int number) {
        super(); // call the super constructor
        this.number = number;
    }

    @Override
    public String toString() {
        return "Group: " + number;
    }
}
```

Przykład wykorzystania klas biznesowych korzystających z ObjectPlusPlus

```
public static void testAssociationsObjectPlus() throws Exception {
    // Create new objects (no links)
    Actor a1 = new Actor("Arnold Schwarzenegger");
    Actor a2 = new Actor("Michael Biehn");
    Actor a3 = new Actor("Kristanna Loken");

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

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

    // Add info about links
    f1.addLink("actor", "movie", a1);
    // f1.addLink("actor", "movie", a2);
    f1.addLink("actor", "movie", a2, "MB"); // use the qualified association
    f3.addLink("actor", "movie", a1);
    f3.addLink("actor", "movie", a3);

    g1.addPart("part", "whole", a1);
    g1.addPart("part", "whole", a2);
    g2.addPart("part", "whole", a3);
    // g2.addPart("part", "whole", a1); // an exception because the part already belongs to another
    whole (group)

    // [...]
}
```

Przykład wykorzystania klas biznesowych korzystających z ObjectPlusPlus (2)

Zadanie

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

```
public static void testAssociationsObjectPlus() throws Exception
// [...]

// Show infos
f1.showLinks("actor", System.out);
f3.showLinks("actor", System.out);

a1.showLinks("movie", System.out);

g1.showLinks("part", System.out);

// Test the qualified association
System.out.println(f1.getLinkedObject("actor", "MB"));
}
```

Movie links, role 'actor':

Actor: Arnold Schwarzenegger

Actor: Michael Biehn

Movie links, role 'actor':

Actor: Arnold Schwarzenegger

Actor: Kristanna Loken

Actor links, role 'movie':

Movie: Terminator 1

Movie: Terminator 3

Group links, role 'part':

Actor: Arnold Schwarzenegger

Actor: Michael Biehn

Actor: Michael Biehn

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.

Pliki źródłowe

- Pobierz pliki źródłowe do wszystkich wykładów MAS



<http://www.mtrzaska.com/plik/mas/mas-source-files-lectures>