Design and Analysis of Information Systems (MAS)

Mariusz Trzaska, Ph. D.
mtrzaska@pjwstk.edu.pl

Lecture 06

Associations in Object-Oriented Programming Languages (2)

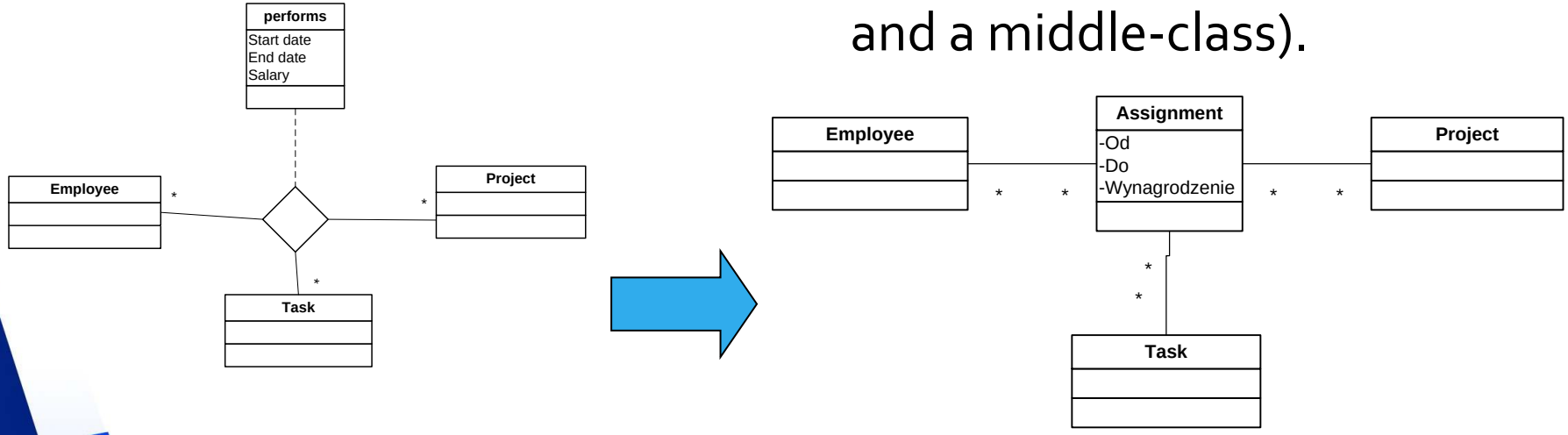
Continuation of the previous lecture

Outline

- *Introduction*
- *Implementation of the associations using:*
  - *identifiers,*
  - *references.*
- *Implementation of the associations:*
  - *In relation to cardinalities,*
  - *binary,*
  - *attribute association,*
  - *qualified,*
  - n-ary,
- Implementation of an aggregation,
- Implementation of a composition,
- Generic management of associations,
- Summary

Implementation of a N-ary Association

- At the beginning we need to transform:
  - One UML construct (n-ary association)
  - Into another UML construct (n binary associations and a middle-class).



Implementation of a N-ary Association (2)

- Thanks to the transformation we got ***n*** „ordinary" associations.
- The „new" associations can be implemented using one of the discussed approaches.
- Possible semantics problems
  - The name of the new class,
  - Roles' names: „old" and „new",
  - Cardinalities.
- More difficult access to the target objects (through the middle-class).

Implementation of an Aggregation

- Does an aggregation mean some consequences to the linked objects?
- **No, it does not**!
- Hence, an aggregation is implemented in the same way like an association.

Implementation of a Composition

- Association part is implemented in the same way like an „ordinary” association.
- Problems to solve:
  1. Preventing from creating parts without the whole,
  2. Forbidding of sharing the parts,
  3. Removing parts during removing the whole.
- Two approaches:
  - Modification of the existing solution,
  - Utilization of the inner classes.

Implementation of a Composition (2)

1. Preventing from creating parts without the whole,
   - The private constructor,
   - A dedicated class method (`static`):
     - Taking a reference to the whole (and validating it); we can potentially use the `@NotNull` annotation, but it is only informative and does not check/enforce anything,
     - Creating the part,
     - Adding a reverse connection.

Implementation of a Composition (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;
   }
}
```

Implementation of a Composition (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;
   }
}
```

Implementation of a Composition (5)

2. Forbidding of sharing the parts

- A modified version of the method adding the part:
  - Checking if the part already exist with other whole,
  - Aside of adding a link, stores globally an information about being a part.
- A class attribute storing info about all parts connected with wholes.

Implementation of a Composition (6)

- A special version of the method adding the part.

```
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;
   }
}
```

Implementation of a Composition (7)

3. Removing parts during removing the whole.

- In languages like Java or C#
  - There is no way to manually remove an object,
  - The object is deleted by the VM when it is not reachable (no references).
- In the C++ there is a way to manually remove an object. The code which will remove parts should be located in the destructor.

Implementation of a Composition (8)

3. Removing parts during removing the whole - *cont*.

- In case of our implementation it is worth:
  - To create a (class) method removing the whole from the extent,
  - The method should also remove information about the part from the global list of parts.

Implementation of the composition using inner classes

- An object of the inner class cannot exist without (surrounding) object of the outer class.
- An object of the inner class has direct access to the invariants of the outer class object.
- The following code is correct however does not have a proper semantics.
  - An inner class object (the part) has access to an outer class object - correct,
  - An outer class object is not aware of inner class object – that is bad.

```
// Create a new whole
Whole whole = new Whole("Whole 01");
// Create a new part
Whole.Part part = whole.new Part("Part 02");
```

Implementation of the composition using inner classes (2)

- To prevent (correct) the behaviour:
  - An inner class should have a private constructor,
  - The inner class must provide a dedicated method for a proper creating „part" objects.
- Manually:
  - Preventing sharing of the parts. A part has to be connected with only whole-object. However it may happen that different wholes would be connected with the same part.
  - Removing parts in case of removing the whole.

Implementation of the composition using inner classes (3)

```java
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;
   }

   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;
      }

      public Whole getWhole() {
         return Whole.this; // Could be useful for accessing the whole
      }

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

Associations Management

- The presented ways of implementing associations are almost the same for each business class in the system.
- Moreover, they will be (almost) the same for each association in the class.
- Is it possible to unified them? To prevent writing the same code all the time?
- Of course – similarly to the extent management, we will use the inheritance.

The Universal Associations Management

- We created the `ObjectPlus` class which was useful for the extent management.
- To keep the functionality, we will create another class called `ObjectPlusPlus` which inherits from the `ObjectPlus`.
- Such an approach guarantees that the existing extent functionality will be complemented with associations.

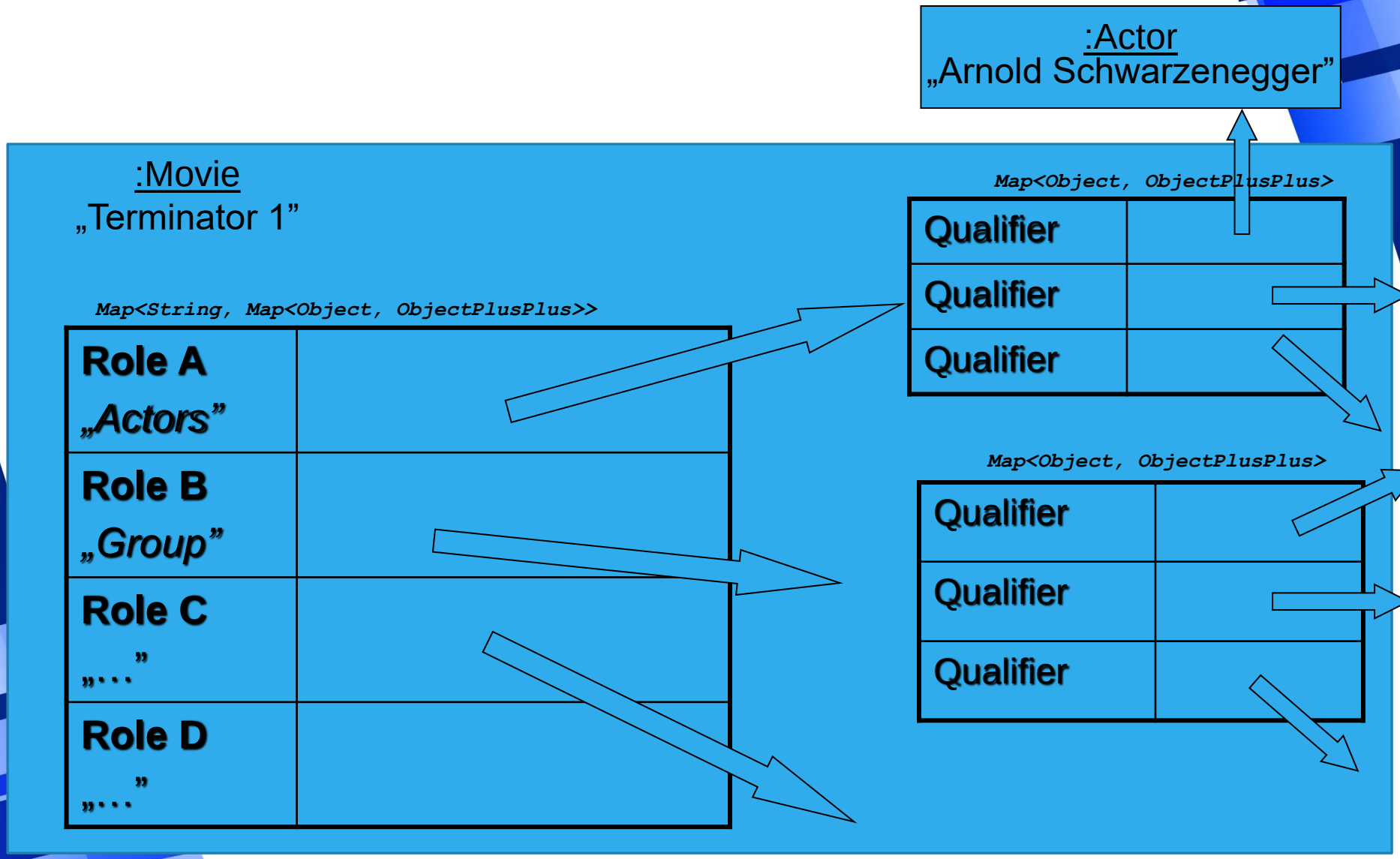
The Universal Associations Management (2)

- Let's create a class which will be a super class for all business classes in our system.
- Let's call it `ObjectPlusPlus` and add the functionalities for management of:
  - Binary associations,
  - Qualified associations,
  - Compositions (partial – only req. no 2).
- We will utilize the second approach (using references rather then ids).

The Universal Associations

- Because all associations in one object will be stored in the same collection, it is not possible to use ordinary container like `Vector` or `ArrayList`.
- We will use a map storing key and values:
  - The key will be a role's name,
  - The value will be a map containing:
    - A key for a qualified association. In case of ordinary associations both key and value will store the same value.
    - A value will store a reference to the target object.
- A class attribute storing references to all objects being parts in compositions. This will allow fulfilment of the requirement no 2 (no sharing),
- In the other words, the new container will store links for all associations of the particular objects.

The Universal Associations (2)



The `ObjectPlusPlus` Class

```
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();
   }

   // [...]
```

The `ObjectPlusPlus` Class (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);
      }
   }
   // [...]
```

The `ObjectPlusPlus` Class (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);
   }
   // [...]
```

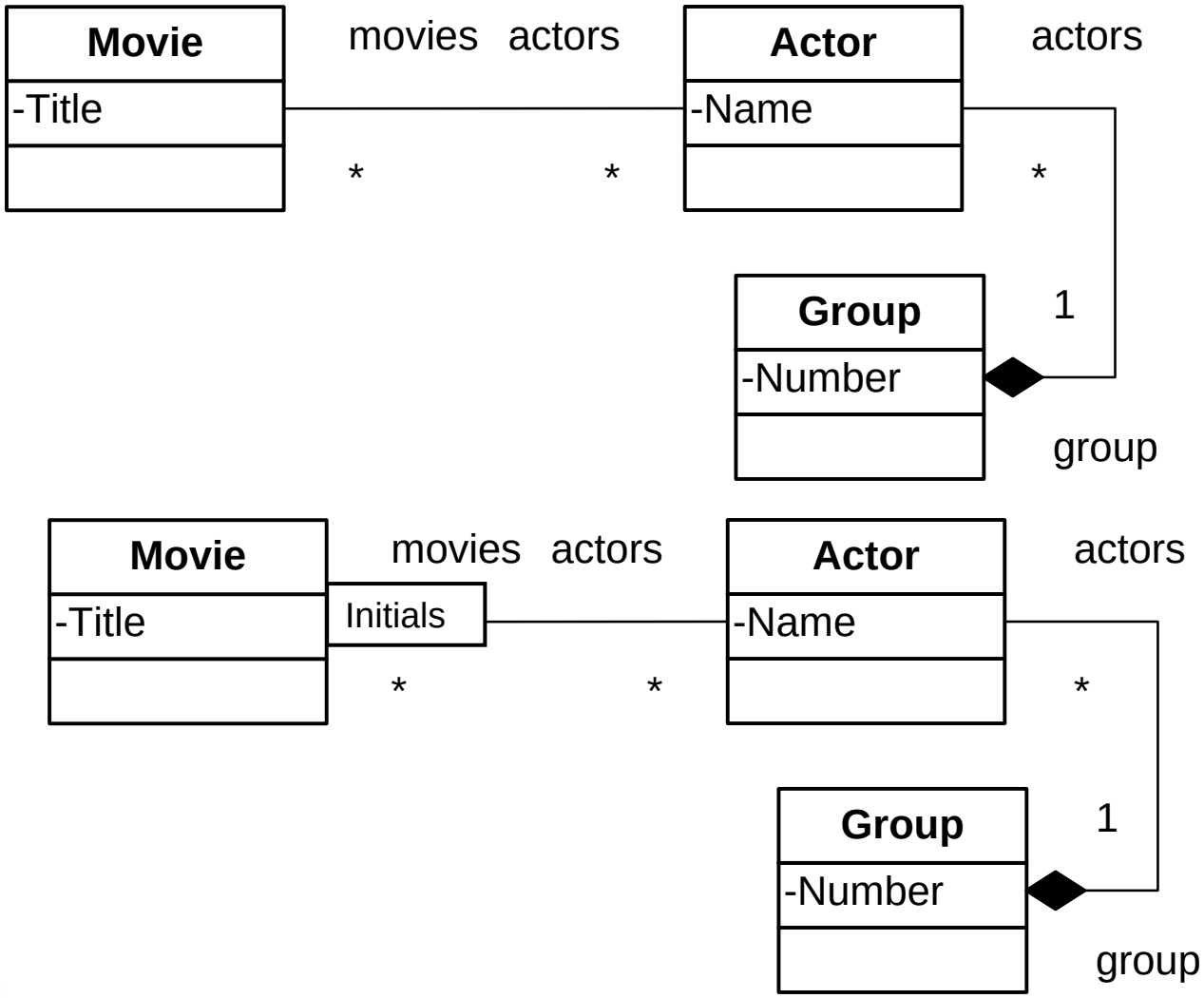
The `ObjectPlusPlus` Class (4)

```
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);
      }
   }
```

The `ObjectPlusPlus` Class (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);
   }
```

Business classes to implement



Implementation of the business classes using `ObjectPlusPlus`

```java
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;
   }
}
```

```java
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;
   }
}
```

Implementation of the business classes using `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;
   }
}
```

Utilization of the businnes classes based on the `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)

   // [...]
}
```

Utilization of the businnes classes based on the `ObjectPlusPlus` (2)

Homework

- What's possibly wrong with the approach?
- How it can be improved?

```
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
```

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

   // 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"));
}
```

The Summary

- We can define two general approaches to implementation of the associations:
  - Identifiers,
  - Native references.
- Some of the associations before the implementation should be transformed into other UML constructs.
- Thanks to that we can implement them using one of the already known ways.
- Entire functionality for the association management should be encapsulated in the super class (`ObjectPlusPlus`).
- Such a solution guarantees that the new class (`ObjectPlusPlus`) also manages the extents.

Source files

Download source files for all MAS lectures

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