Class Design with OO Programing Languages

MDS-Ana Moreira

From analysis to design

- During the **analysis** phase we **didn't care about the implementation** language
  - This is one of the **main concerns in the design phase**
- The obvious candidate languages to support UML models are OO languages (e.g. C++, Java, C#)
  - However, several modeling abstractions used in UML are not available in those languages ☹

MDS-Ana Moreira

Implementing associations

- OO programming languages do not provide the concept of association
  - Instead, they provide references, in which one object stores a handle to another object
- In the design phase we have to generate references out of associations
  - However this poses a mapping problem

Mapping problem

- Associations …
  - can be bidirectional or unidirectional
    - this is specified through arrow heads (navigation)
  - take place among two or more objects
    - this is specified through cardinalities and type of association
- References…
  - are unidirectional
  - take place between two objects

The direction problem

- Unidirectional associations
  - are the most simple type of association
  - are much simpler to realize
- Bidirectional associations
  - are more complex
  - introduce mutual dependencies among classes
  - are sometimes necessary
    (e.g. in the cases of peer classes that need to work together closely)

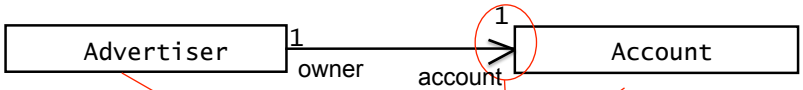
Mapping associations

1. Unidirectional one-to-one association
2. Bidirectional one-to-one association
3. Bidirectional one-to-many association
4. Bidirectional many-to-many association

- Other cases are handled similarly

Unidirectional one-to-one association

Model before transformation:



Source code after transformation:

```
public class Advertiser {
      private Account account;
      public Advertiser() {
            account = new Account();
      }
      public Account getAccount() {
            return account;
      }
}
```

Operation to handle the reference

MDS-Ana Moreira

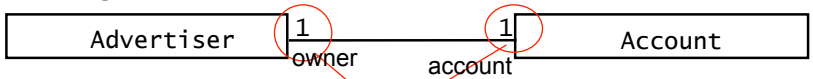
The direction problem!

- The **choice between a unidirectional or a bidirectional association is a trade-off** that we need to evaluate in the context of the problem and involved classes
- The direction of an association can often change during the development of the system

MDS-Ana Moreira

Bidirectional one-to-one association

Object design model before transformation:



Source code after transformation:

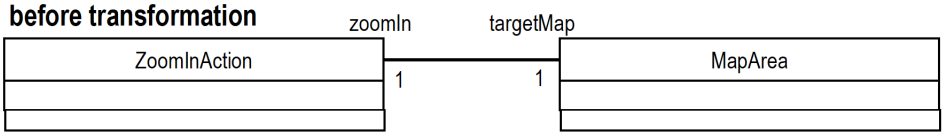
```
public class Advertiser {
/* account is initialized
 * in the constructor and never
 * modified. */
   private Account account;
   public Advertiser() {
    account = new Account(this);
   }
   public Account getAccount() {
    return account;
   }
}
```

```
public class Account {
   /* owner is initialized
    * in the constructor and
    * never modified. */
   private Advertiser owner;
   publicAccount(owner:Advertiser) {
    this.owner = owner;
   }
   public Advertiser getOwner() {
    return owner;
   }
}
```

MDS-Ana Moreira

Bidirectional one-to-one association

before transformation



after transformation

Reference + Added operations to handle the references

```
class ZoomInAction extends AbstractAction {
     private MapArea targetMap;
      /* Other methods omitted */
void setTargetMap(map: MapArea) {
                  if (targetMap != map) {
                  targetMap = map;
     targetMap.setZoomInAction(this);}}}
```

```
class MapArea extends JPanel {
   private ZoomInAction zoomIn;
/* Other methods omitted */
  void setZoomInAction (action:ZoomInAction)
  {if (zoomIn != action)
    {zoomIn = action;
     zoomIn.setTargetMap(this);}}}
```

MDS-Ana Moreira

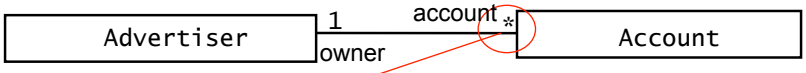
Realizing other cardinalities

- One-to-many associations
  - cannot be realized using a single reference or a pair of references.
  - Instead, the "many" part is realized by using a collection of references
- Many-to-many associations
  - In this case, **both end classes have attributes that are collections of references and operations** to keep these collections consistent

MDS-Ana Moreira

Bidirectional one-to-many association

Object design model before transformation:



Source code after transformation:

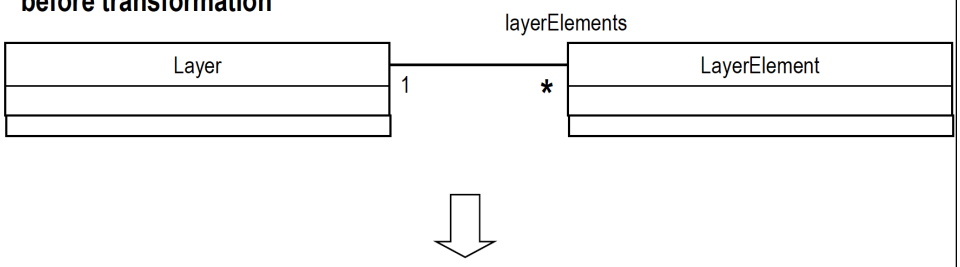
```
public class Advertiser {
   private Set accounts;
   public Advertiser() {
    accounts = new HashSet();
   }
   public void addAccount(Account a)
{
    accounts.add(a);
    a.setOwner(this);
   }
   public void removeAccount(Account a)
{
    accounts.remove(a);
    a.setOwner(null);
   }
}
```

```
public class Account {
   private Advertiser owner;
   public void setOwner(Advertiser
   newOwner) {
     if (owner != newOwner) {
          Advertiser old = owner;
          owner = newOwner;
          if (newOwner != null)
               newOwner.addAccount
   (this);
          if (oldOwner != null)
               old.removeAccount(this);
     }
   }
}
```

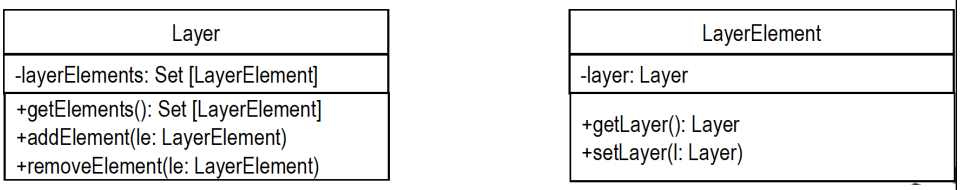
MDS-Ana Moreira

Bidirectional one-to-many association

before transformation



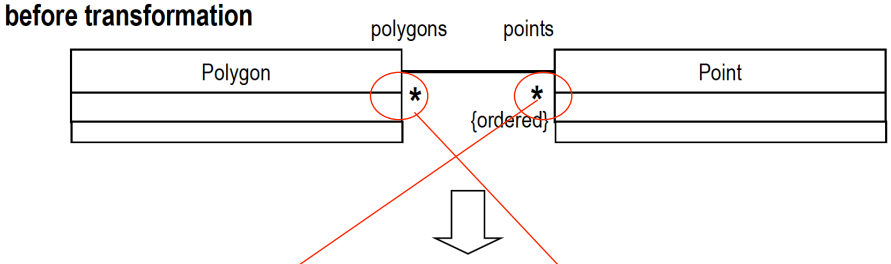
after transformation



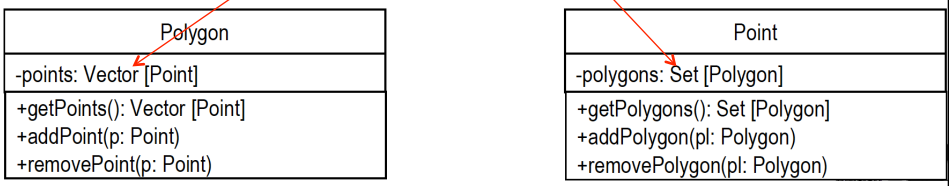
MDS-Ana Moreira

Realizing many-to-many associations

before transformation



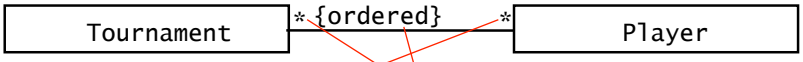
after transformation



MDS-Ana Moreira

Bidirectional many-to-many association

Object design model before transformation



Source code after transformation

```
public class Tournament {
   private List players;
   public Tournament() {
      players = new ArrayList();
   }
   public void addPlayer(Player p)
   {
      if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
      }
   }
}
```

```
public class Player {
   private List tournaments;
   public Player() {
      tournaments = new ArrayList
   ();
   }
   public void addTournament
   (Tournament t) {
      if (!tournaments.contains
   (t)) {
         tournaments.add(t);
         t.addPlayer(this);
      }
   }
}
```

- Decision made here: both ordered and implemented with a list

MDS-Ana Moreira

Transformation rules (1)

- The naming convention for references is similar to OCL
  - If defined, roles are used for naming references
  - Otherwise, we use the target class name starting in lowercase letter
- **Template classes** (parameterized types) available in the target OO language **can be used to implement the required collection types** that represent the "many" cardinality
  - e.g. remember the C++ STL (Standard Template Library)

MDS-Ana Moreira

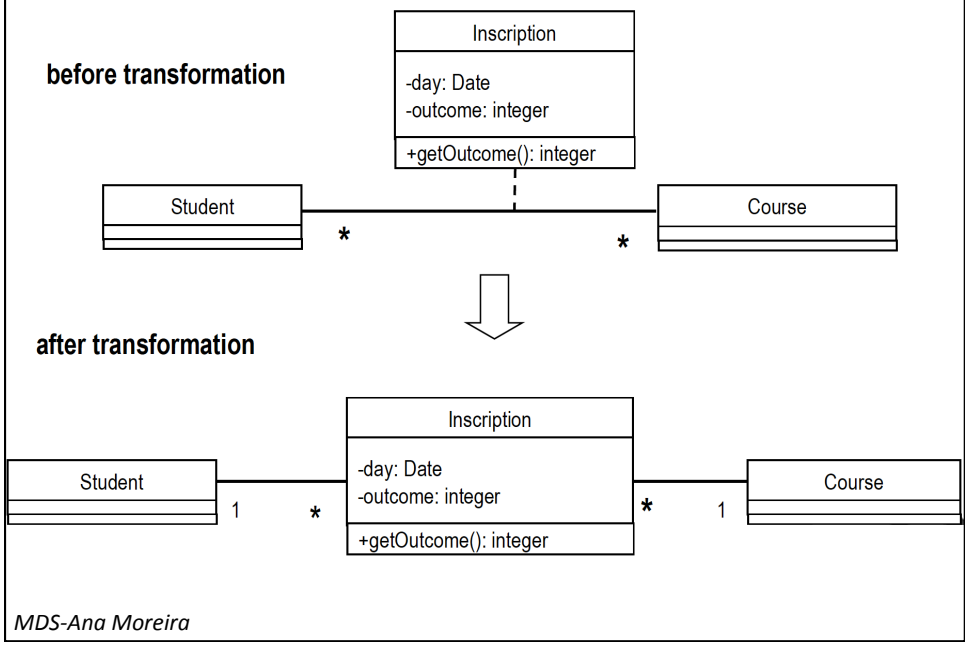
Transformation rules (2)

- When we have a "many" cardinality we use:
  - a Set or other non-sequential parameterized collection type, if the association is not ordered
    - This corresponds to the Set type in OCL
  - a Vector, a List or other sequential collection type available in the target language, if the association is ordered
    - This corresponds to the Sequence type in OCL

Realizing Associative Classes

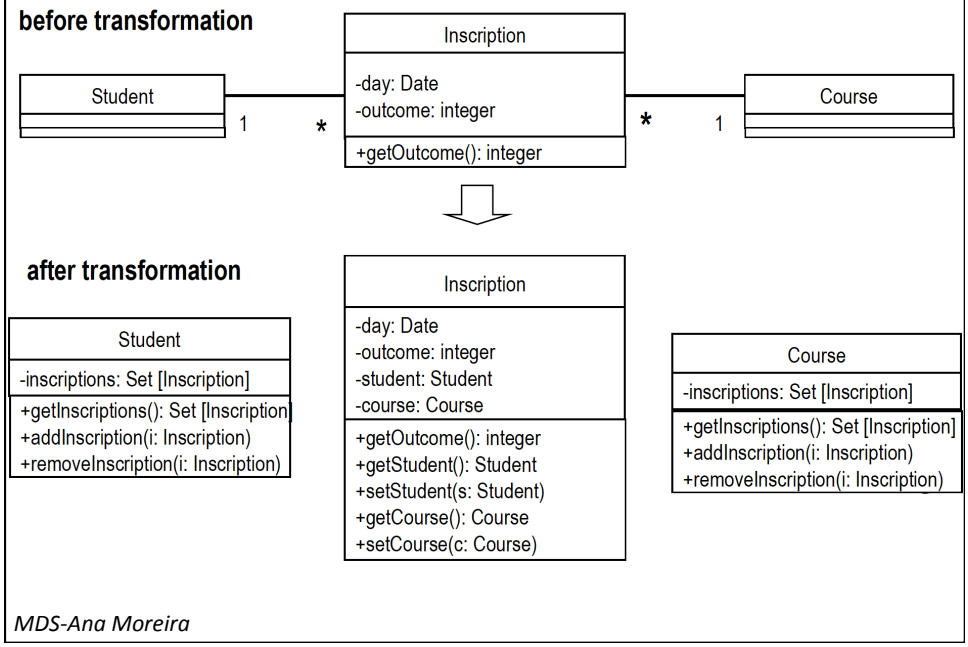
- Remember that an association class holds attributes and operations of the association
- The association class is transformed into a separate class and a number of binary associations

Realizing Associative Classes (1st step)



MDS-Ana Moreira

Realizing Associative Classes (2nd step)



MDS-Ana Moreira