# More on Class Diagram Implementation

# State of the Art: Model-based Software Engineering

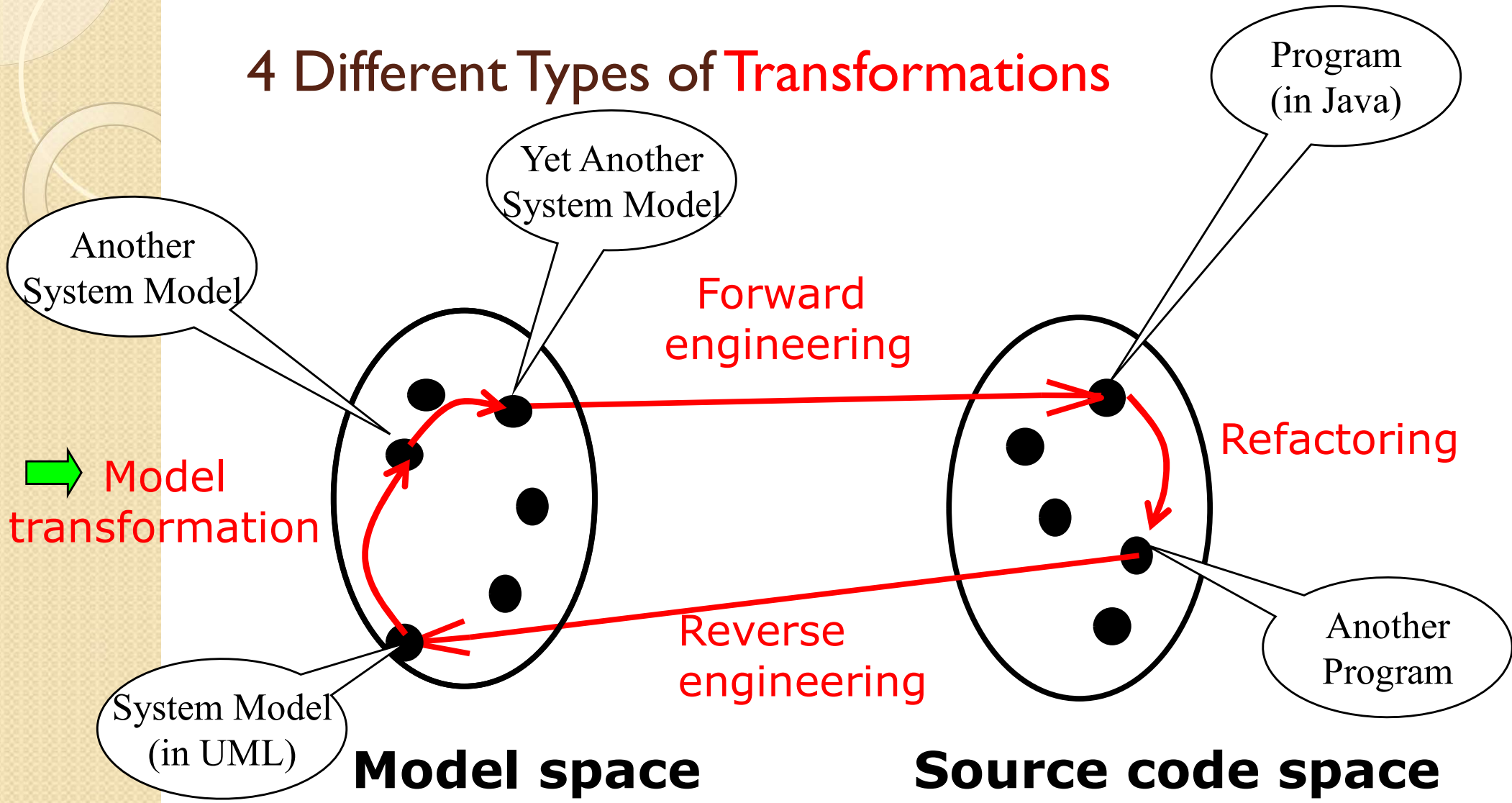
- The Vision
  - During object design we build an object design model that realizes the use case model and which is the basis for implementation (model-driven engineering)
- The Reality
  - Working on the object design model involves many activities that are error prone
  - Examples:
    - A new parameter must be added to an operation. Because of time pressure it is added to the source code, but not to the object model
    - Additional attributes are added to an entity object, but the data base table is not updated (as a result, the new attributes are not persistent).

# Problems with implementing an Object Design Model

- Programming languages do not support the concept of UML associations
  - The associations of the object model must be transformed into collections of object references
- Many programming languages do not support contracts (invariants, pre and post conditions)
  - Developers must therefore manually transform contract specification into source code for detecting and handling contract violations
- The client changes the requirements during object design
  - The developer must change the contracts in which the classes are involved
- All these object design activities cause problems, because they need to be done manually.

- Let us get a handle on these problems
- To do this we distinguish two kinds of spaces
  - the model space and the source code space
- and 4 different types of transformations
  - Model transformation
  - Forward engineering
  - Reverse engineering
  - Refactoring.

# 4 Different Types of Transformations



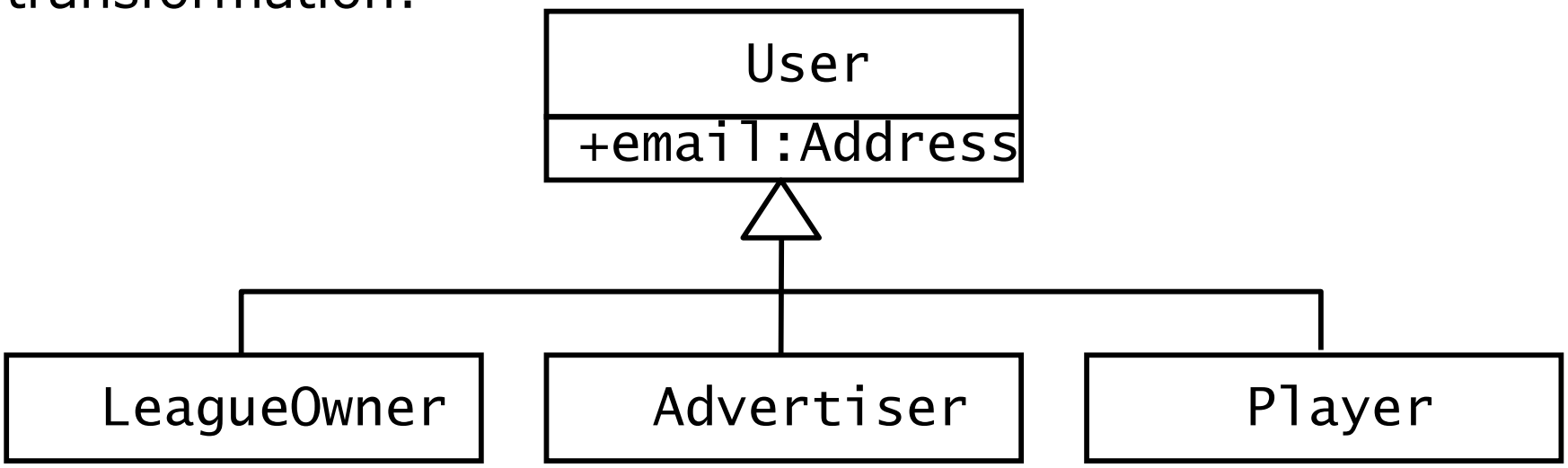
# Model Transformation

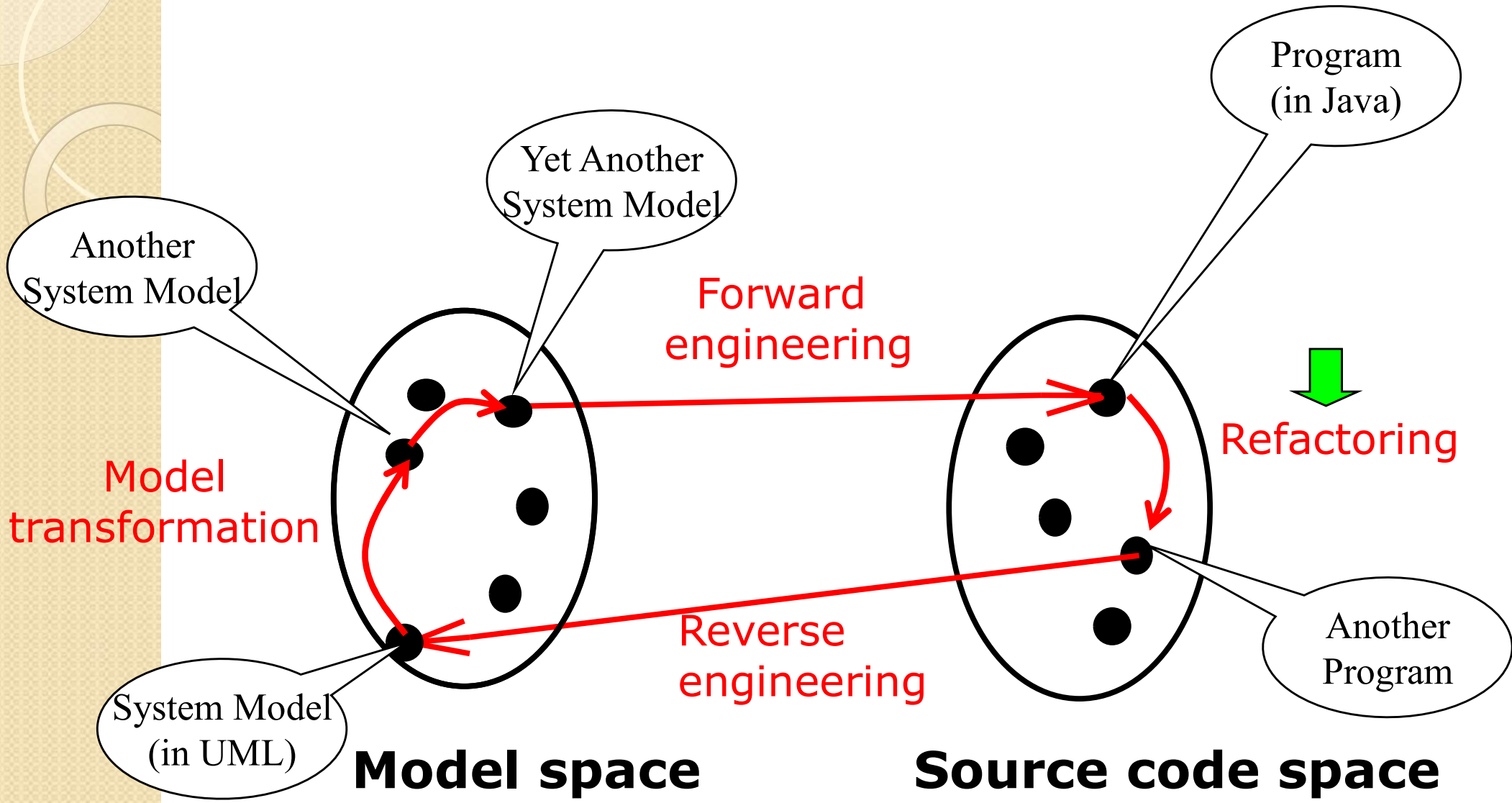
- Takes as input a model conforming to a meta model (for example the MOF metamodel) and produces as output another model conforming to the metamodel
- Model transformations are used in MDA (Model Driven Architecture).

# Model Transformation Example

Object design model before transformation:


Object design model after transformation:




Program
(in Java)
Yet Another
System Model
Another
System Model
Forward
engineering
Refactoring
Model
transformation
Reverse
engineering
Another
Program
System Model
(in UML)
Model space
Source code space

# Refactoring : Pull Up Field

```
public class Player {
   private String email;
   //...
}
public class LeagueOwner {
   private String eMail;
   //...
}
public class Advertiser {
   private String
   email_address;
   //...
}
```

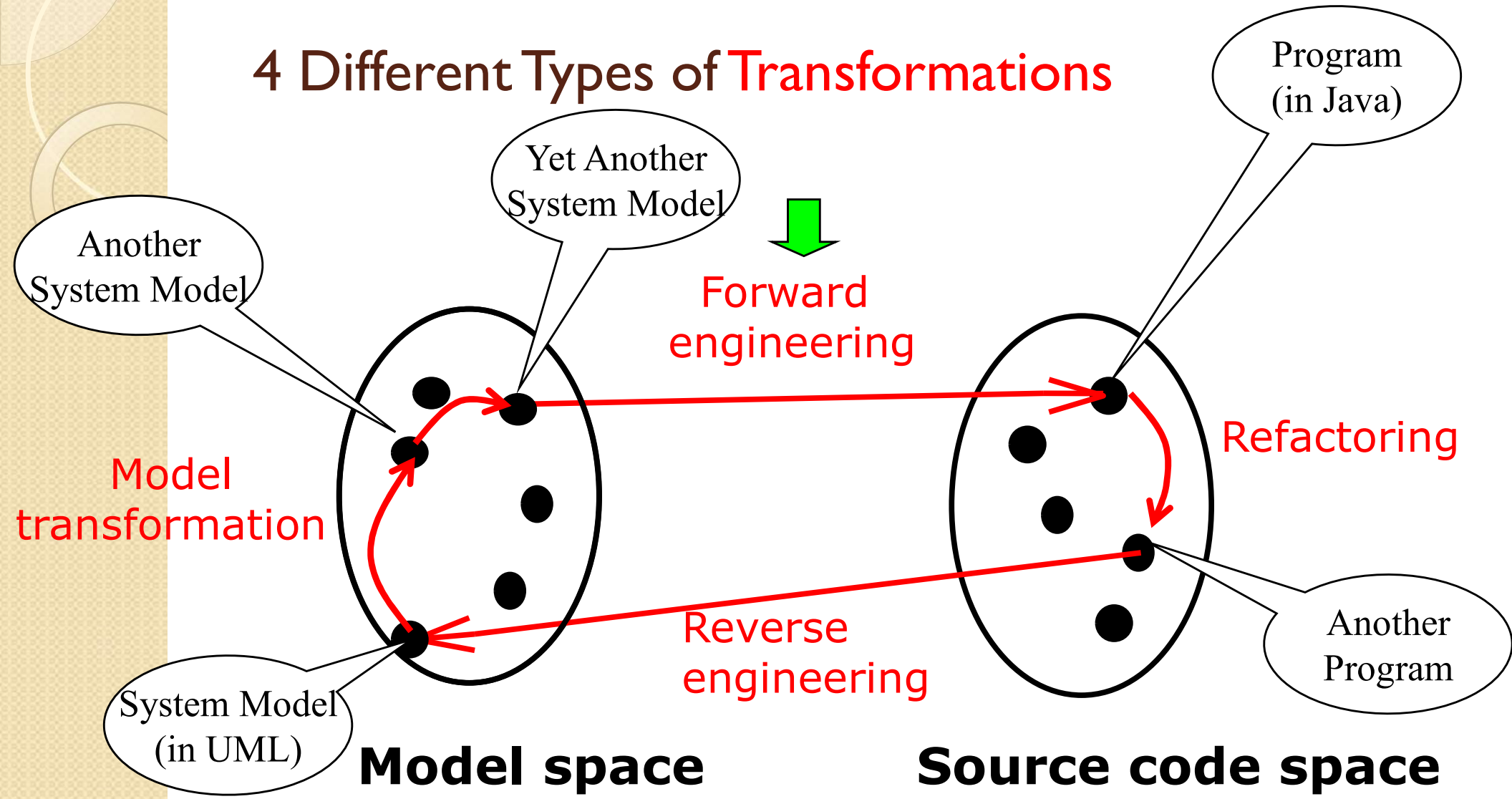
```
public class User {
   private String email;
}
public class Player extends User {
   //...
}

public class LeagueOwner extends
   User {
   //...
}

public class Advertiser extends User
   {
   //...
}.
```

# Refactoring Example: Pull Up Constructor Body

```
public class User {
   private String email;
}


public class Player extends User {
   public Player(String email) {
       this.email = email;
   }
}
public class LeagueOwner extends
   User{
   public LeagueOwner(String email)
   {
       this.email = email;
   }
}
public class Advertiser extends
   User{
   public Advertiser(String email) {
       this.email = email;
   }
}
```

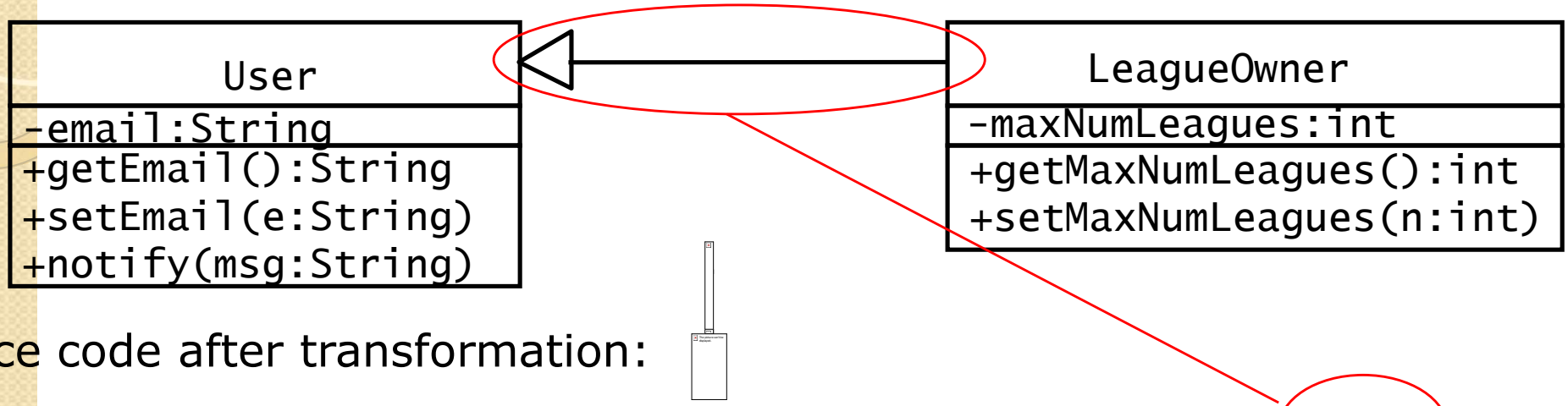
```
   public class User {
      public User(String email) {
           this.email = email;
      }
   }

public class Player extends User {
 public Player(String email) {
      super(email);
       }
}

public class LeagueOwner extends
User {
 public LeagueOwner(String email) {
       super(email);
       }
}

public class Advertiser extends
User {
  public Advertiser(String email) {
       super(email);
       }
}.
```

# 4 Different Types of Transformations



# Forward Engineering Example

Object design model before transformation:




Source code after transformation:

```
public class User {
   private String email;
   public String getEmail() {
         return email;
   }
   public void setEmail(String e){
         email = e;
   }
   public void notify(String msg) {
         // ....
   }
}
```

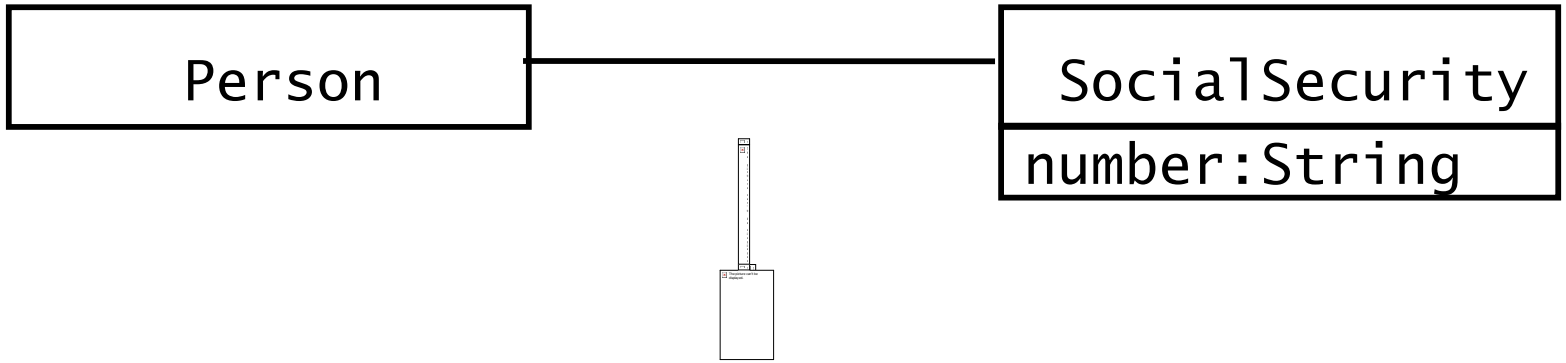
```
public class LeagueOwner extends User {
   private int maxNumLeagues;
   public int getMaxNumLeagues() {
         return maxNumLeagues;
   }
  public void setMaxNumLeagues(int n) {
         maxNumLeagues = n;
   }
}
```

# More Forward Engineering Examples

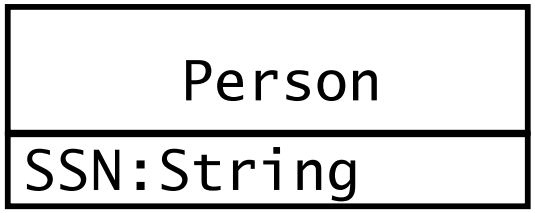
- Model Transformations
  - Goal: Optimizing the object design model
    - Collapsing objects
    - Delaying expensive computations
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - Mapping inheritance
  - Mapping associations
  - Mapping contracts to exceptions
  - Mapping object models to tables

# Collapsing Objects

Object design model before transformation:




Object design model after transformation:




Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

# Examples of Model Transformations and Forward Engineering

- Model Transformations
  - Goal: Optimizing the object design model
    - Collapsing objects
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - Mapping inheritance
  - Mapping associations
  - Mapping contracts to exceptions
  - Mapping object models to tables

# Examples of Model Transformations and Forward Engineering

- Model Transformations
  - Goal: Optimizing the object design model
    - Collapsing objects
    - Delaying expensive computations
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - Mapping inheritance
  - Mapping associations
  - Mapping contracts to exceptions
  - Mapping object models to tables

# Examples of Model Transformations and Forward Engineering

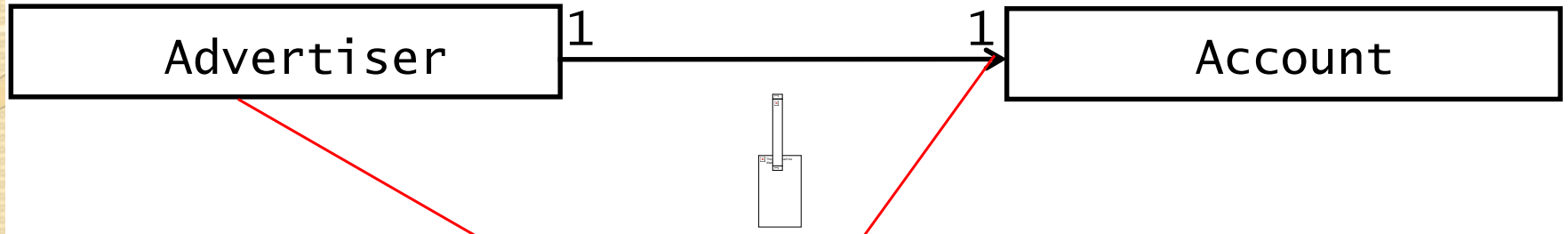
- Model Transformations
  - Goal: Optimizing the object design model
    - ✓Collapsing objects
    - ✓Delaying expensive computations
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - ✓Mapping inheritance
  - Mapping associations
  - Mapping contracts to exceptions
  - Mapping object models to tables

# 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
5. Bidirectional qualified association.

# Unidirectional one-to-one association

Object design model before transformation:

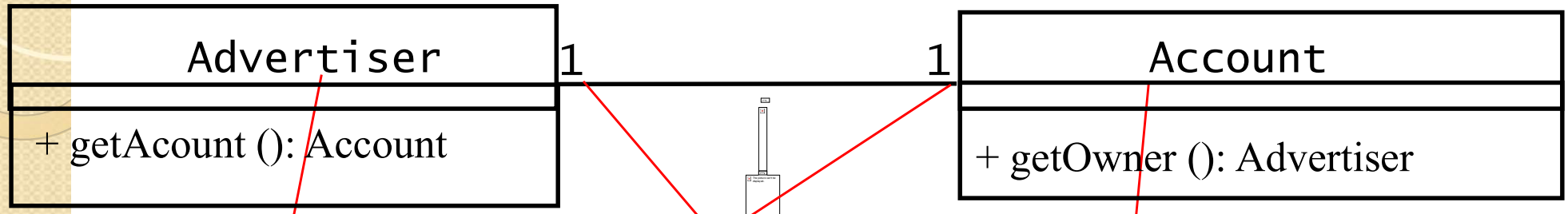



Source code after transformation:

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

# Bidirectional one-to-one association

Object design model before transformation:



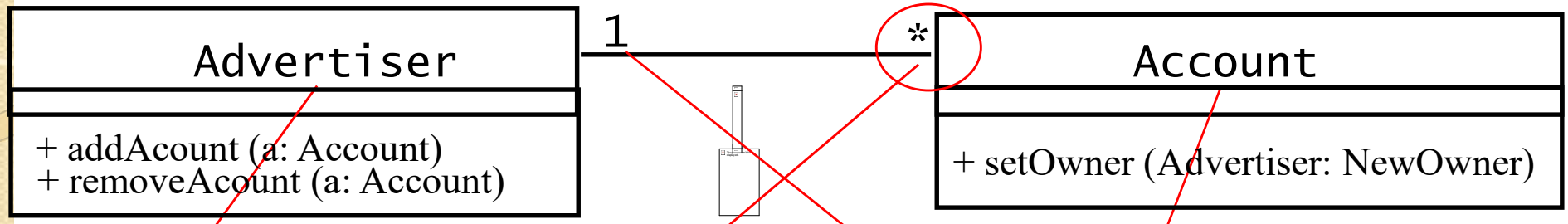

Source code after transformation:

```
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;
  public Account(owner:Advertiser)
  {
       this.owner = owner;
  }
  public Advertiser getOwner() {
       return owner;
  }
}
```

# Bidirectional one-to-many association

Object design model before transformation:



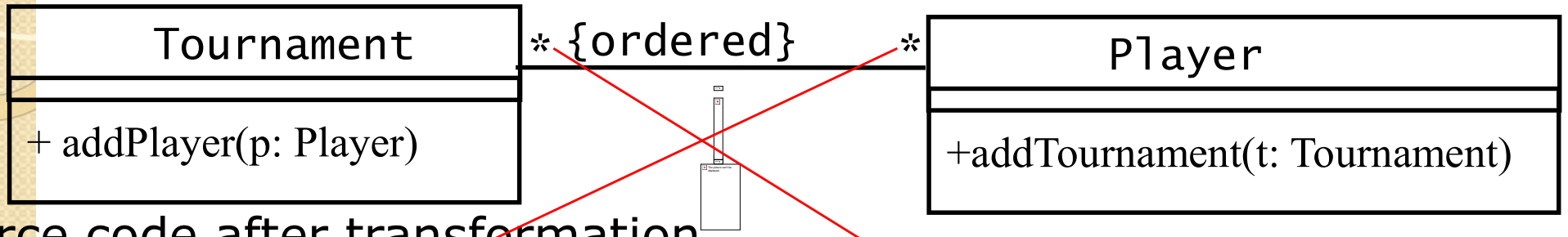

Source code after transformation:

```
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);
     }
   }
}
```

# Bidirectional many-to-many association

Object design model before transformation



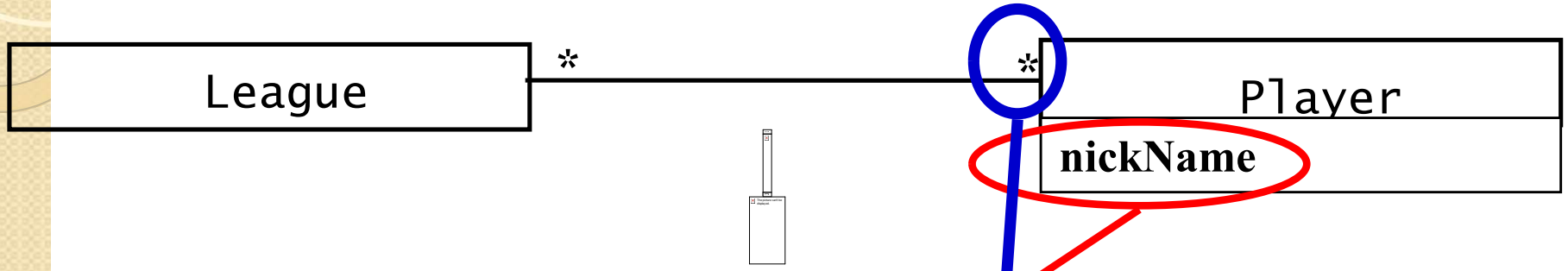

Source code after transformation

```
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);
      }
   }
}
```

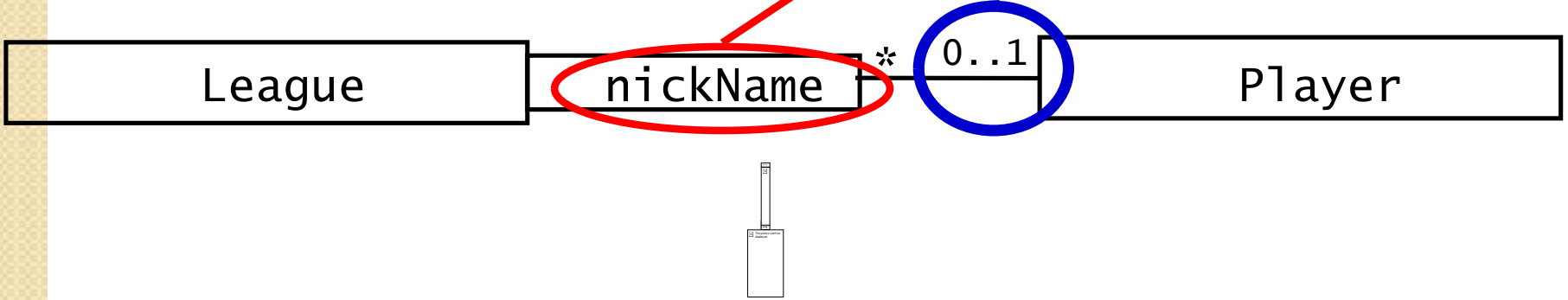
```
public class Player {
   private List tournaments;
   public Player() {
      tournaments = new
ArrayList();
   }
   public void
addTournament(Tournament t) {
     if (!tournaments.contains(t)) {
       tournaments.add(t);
       t.addPlayer(this);
     }
   }
}
```

# Bidirectional qualified association

Object design model before model transformation



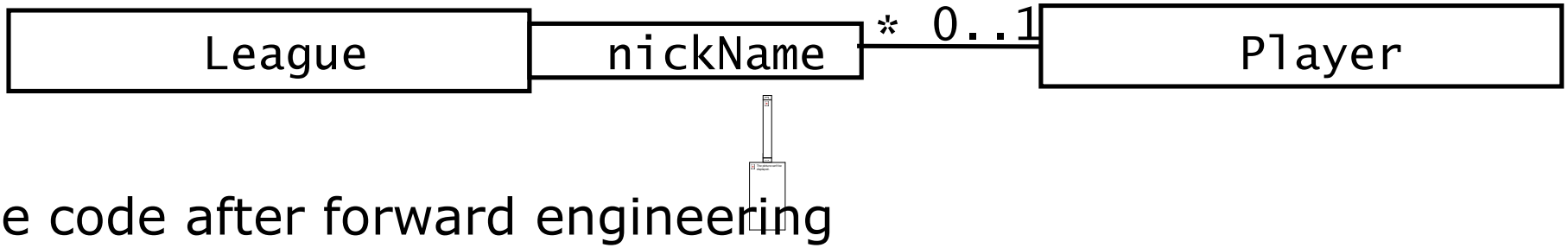

Object design model after model transformation




Source code after forward engineering (see next slide)

# Bidirectional qualified association (2)

Object design model before forward engineering




Source code after forward engineering

```
public class League {
  private Map players;

  public void addPlayer
    (String nickName, Player p) {
   if
   (!players.containsKey(nickName))
   {
     players.put(nickName, p);
     p.addLeague(nickName, this);
         }
   }
}
```

```
public class Player {
  private Map leagues;

  public void addLeague
     (String nickName, League l) {
     if (!leagues.containsKey(l)) {
      leagues.put(l, nickName);
      l.addPlayer(nickName, this);
     }
   }
}
```

# Summary

- Strategy for implementing associations:
  - Be as uniform as possible
  - Individual decision for each association
- Example of uniform implementation
  - 1-to-1 association:
    - Role names are treated like attributes in the classes and translate to references
  - 1-to-many association:
    - "Ordered many" :Translate to `Vector`
    - "Unordered many" : Translate to `Set`
  - Qualified association:
    - Translate to Hash table