Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 10, Mapping Models to Code

State of the Art: Model-based Software Engineering

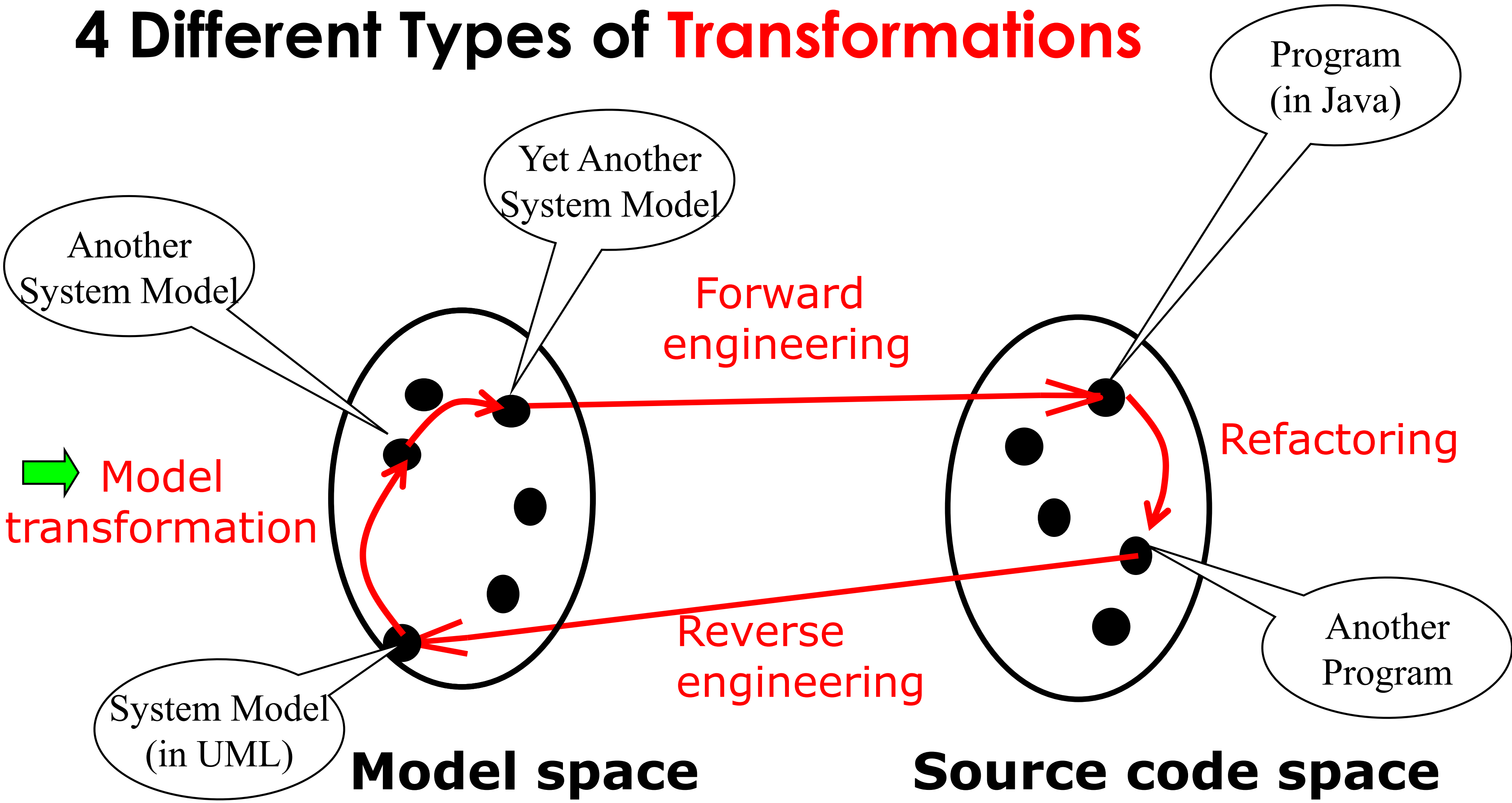
- The Vision
  - During object design we build an object design model that realizes the use case model and it is the basis for implementation (model-driven design)
- The Reality
  - Working on the object design model involves many activities that are error prone
  - Examples:
    - A new parameter must 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 database table is not updated (as a result, the new attributes are not persistent).

Other Object Design Activities

- Programming languages do not support the concept of a UML association
  - 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 interface specification of the involved classes
- All these object design activities cause problems, because they need to be done manually.

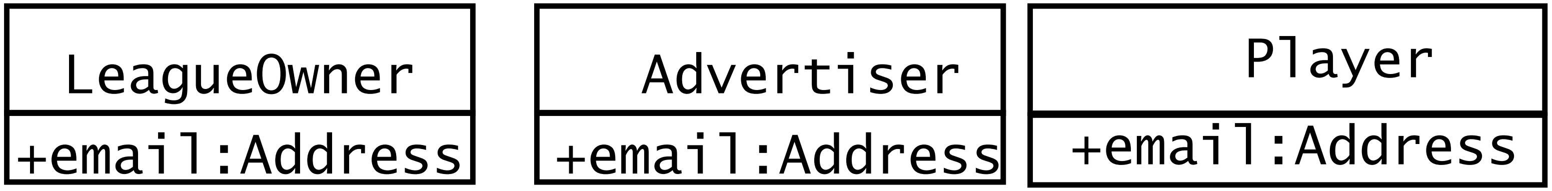
- Let us get a handle on these problems
- To do this we distinguish two kinds of spaces
  - the model space and the source code space
- and 4 different types of transformations
  - Model transformation,
  - Forward engineering,
  - Reverse engineering,
  - Refactoring.

4 Different Types of Transformations

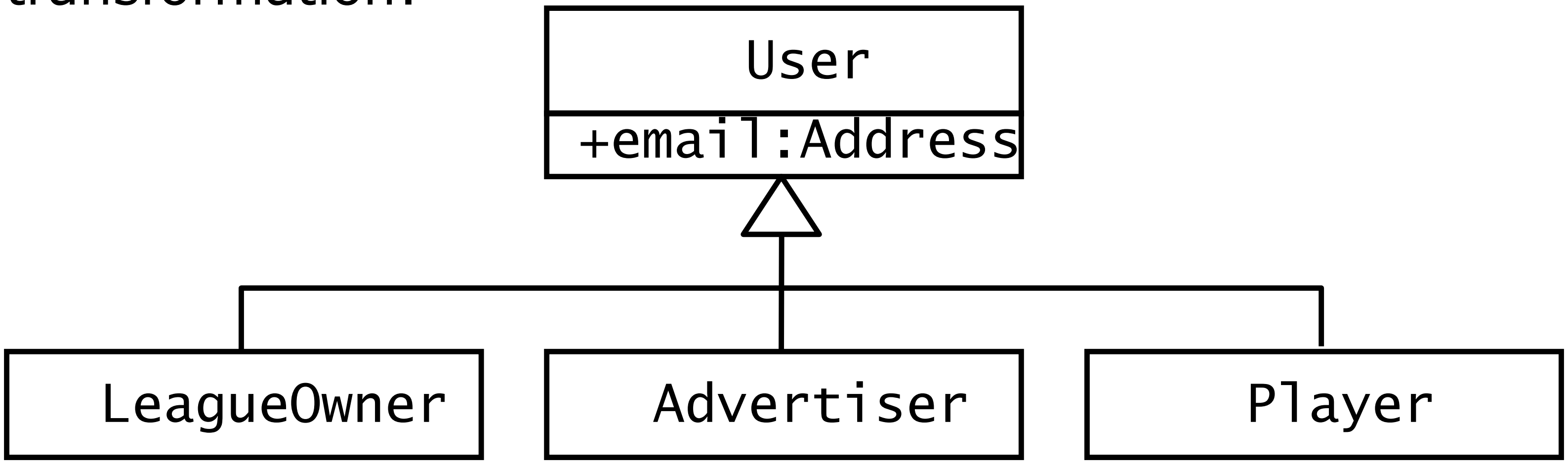


Model Transformation Example

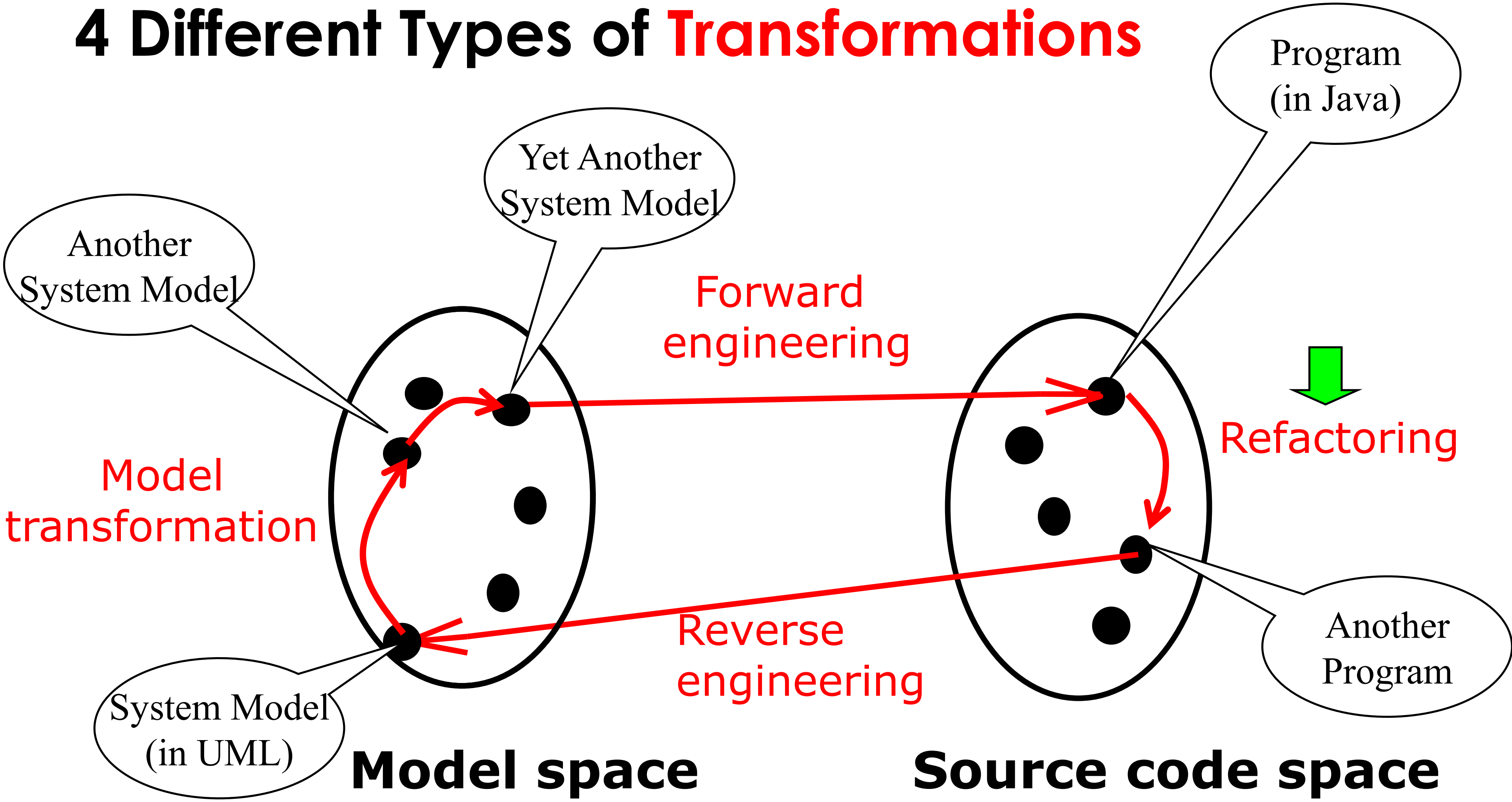
Object design model before transformation:



Object design model after transformation:



4 Different Types of Transformations



Refactoring Example: Pull Up Field

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

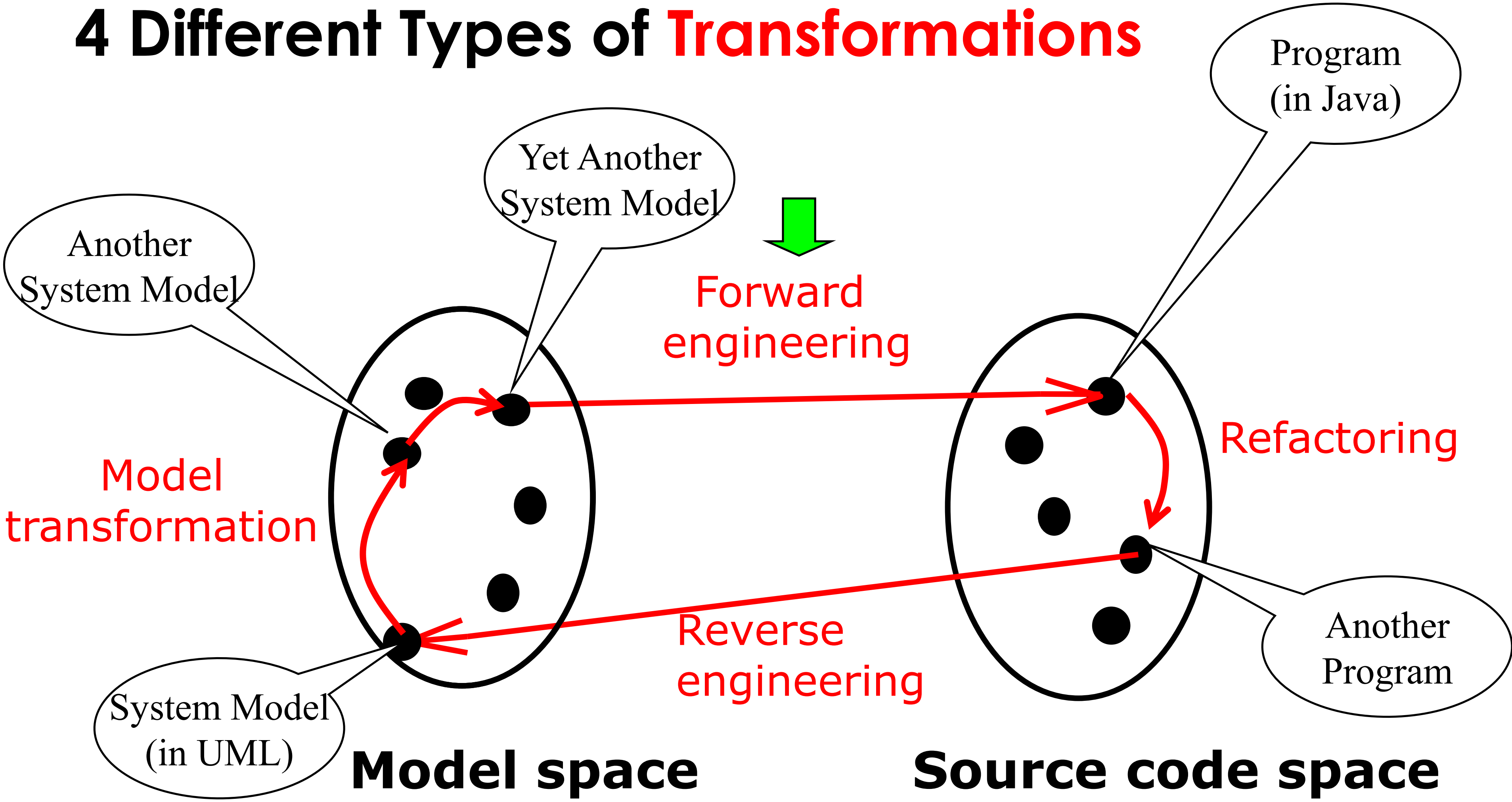
```
public class User {
  private String email;
}

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 extendsUser{
   public Advertiser(String email) {
        this.email = email;
   }
}
```

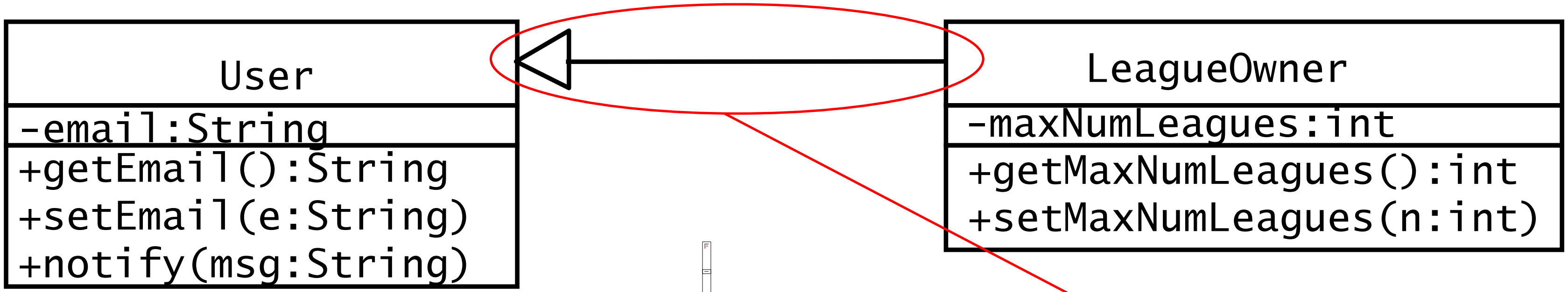
```
 public class User {
    public User(String email) {
          this.email = email;
    }
 }

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 Use
{
         public Advertiser(String
email) {
                 super(email);
         }
}
```

4 Different Types of Transformations



Forward Engineering Example

Object design model before transformation:



User

-email:String

+getEmail():String
+setEmail(e:String)
+notify(msg:String)

LeagueOwner

-maxNumLeagues:int

+getMaxNumLeagues():int
+setMaxNumLeagues(n:int)

Source code after transformation:

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

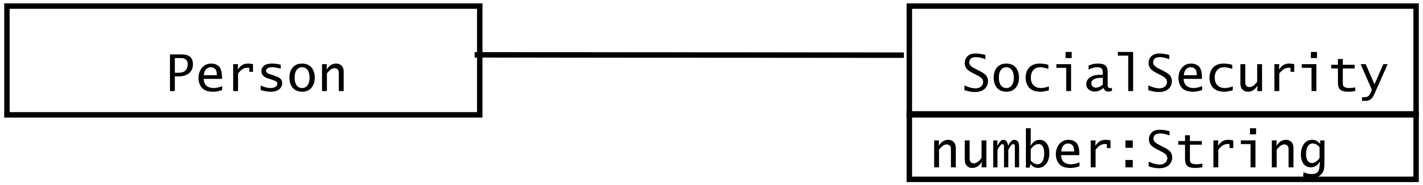
```
public class LeagueOwner extends User {
   private int maxNumLeagues;
   public int getMaxNumLeagues() {
         return maxNumLeagues;
   }
   public void setMaxNumLeagues
                  (int value) {
         maxNumLeagues = value;
   }
}
```

More Examples of Model Transformations and Forward Engineering

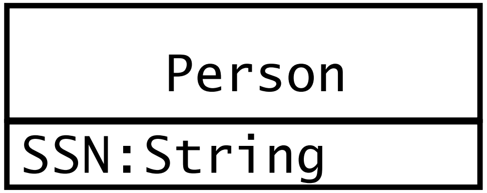
- Model Transformations
  - Goal: Optimizing the object design model
    - ➡ Collapsing objects
    - Delaying expensive computations
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - Mapping inheritance
  - Mapping associations
  - Mapping contracts to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

Collapsing Objects

Object design model before transformation:



Object design model after transformation:



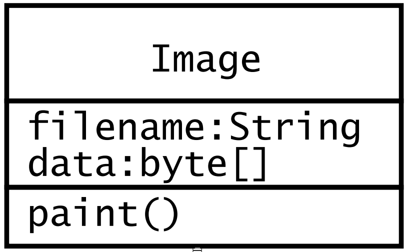
Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

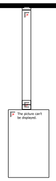
Examples of Model Transformations and Forward Engineering

- Model Transformations
  - Goal: Optimizing the object design model
    - Collapsing objects
    - ➡ 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

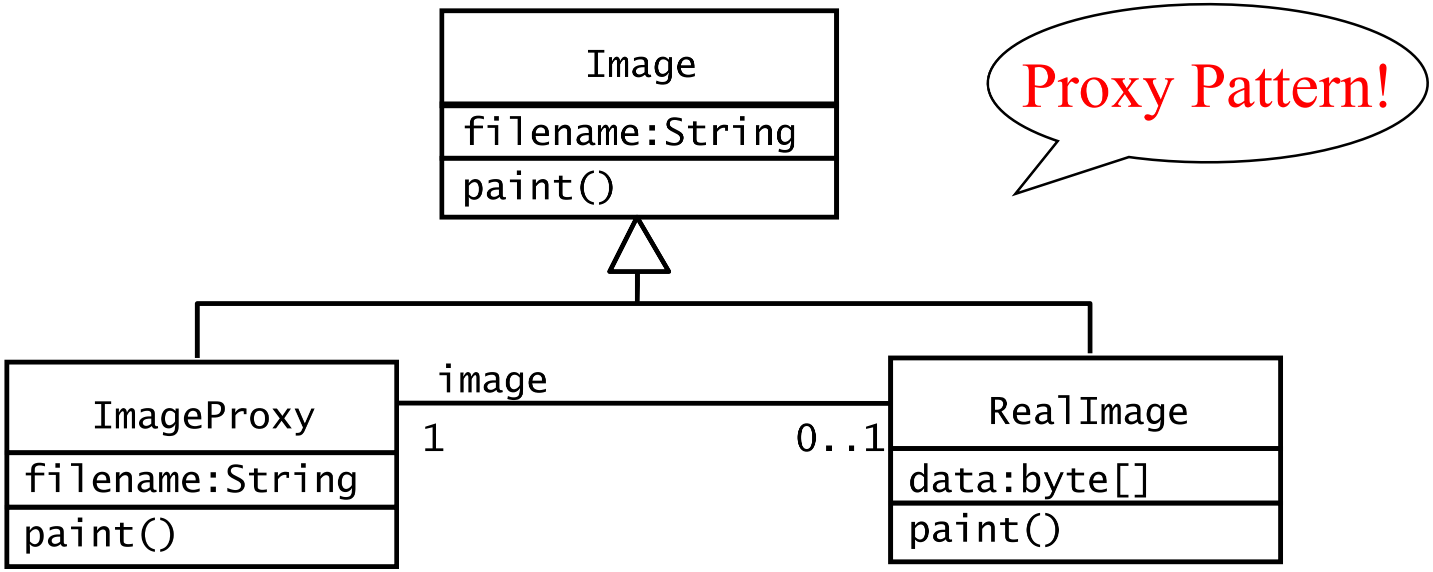
Delaying expensive computations

Object design model before transformation:





Object design model after transformation:



Proxy Pattern!

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

Forward Engineering: Mapping a UML Model into Source Code

- **Goal**: We have a UML-Model with inheritance. We want to translate it into source code
- **Question**: Which mechanisms in the programming language can be used?
  - Let’s focus on Java
- Java provides the following mechanisms:
  - Overriding of methods (default in Java)
  - Final classes
  - Final methods
  - Abstract methods
  - Abstract classes
  - Interfaces

Realizing Inheritance in Java

- Realization of specialization and generalization
  - Definition of subclasses
  - Java keyword: **`extends`**
- Realization of strict inheritance
  - Overriding of methods is not allowed
  - Java keyword: **`final`**
- Realization of implementation inheritance
  - No keyword necessary:
    - Overriding of methods is default in Java
- Realization of specification inheritance
  - Specification of an interface
  - Java keywords: **`abstract`**, **`interface`**

Examples of Model Transformations and Forward Engineering

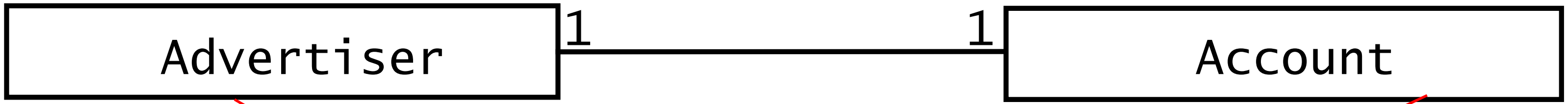
- Model Transformations
  - Goal: Optimizing the object design model
    - ✓Collapsing objects
    - ✓Delaying expensive computations
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - ✓Mapping inheritance
  - ➡Mapping associations
  - Mapping contracts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

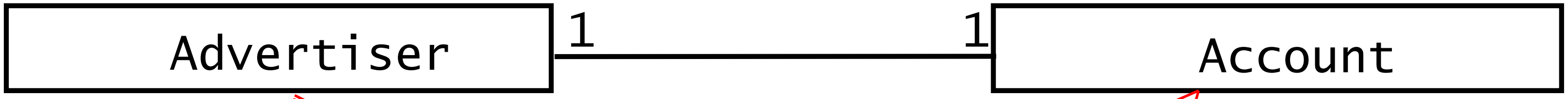


Source code after transformation:

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

Bidirectional one-to-one association

Object design model before transformation:



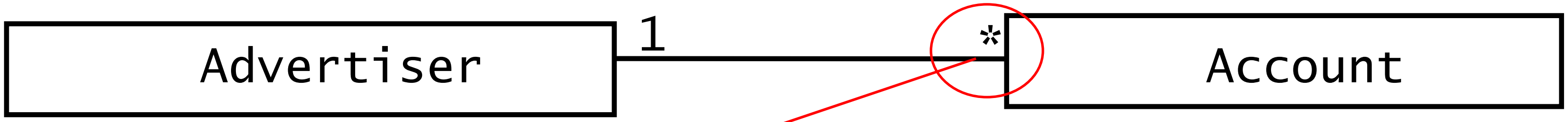
Source code after transformation:

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

Bidirectional one-to-many association

Object design model before transformation:



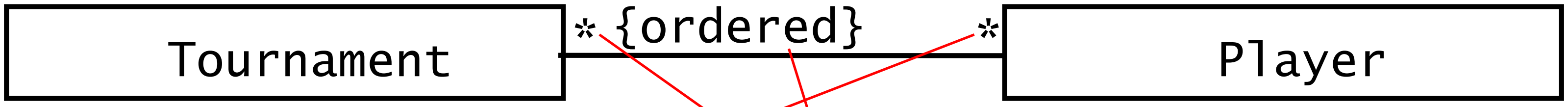
Source code after transformation:

```
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);
       }
   }
}
```

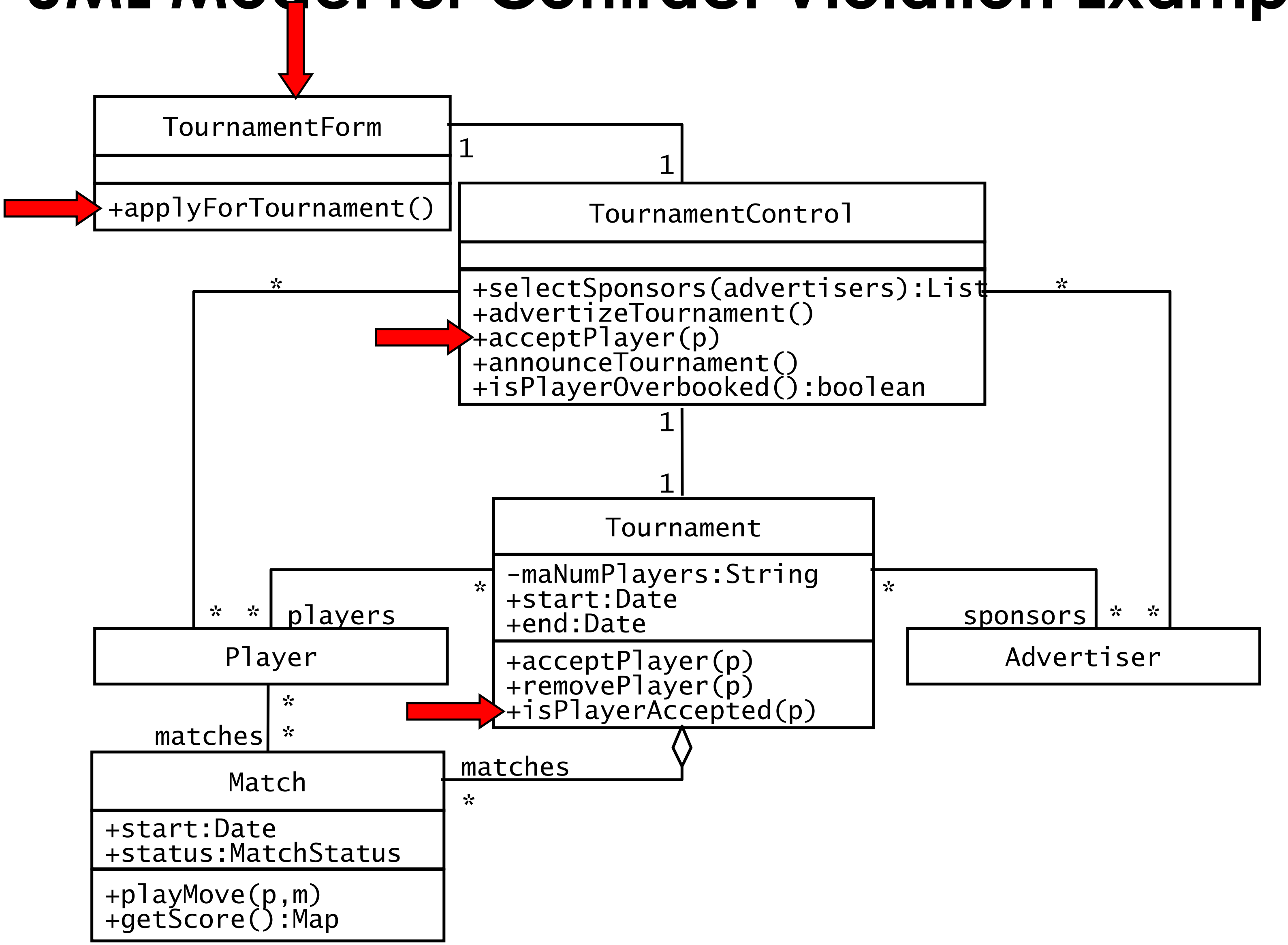
Examples of Model Transformations and Forward Engineering

- 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

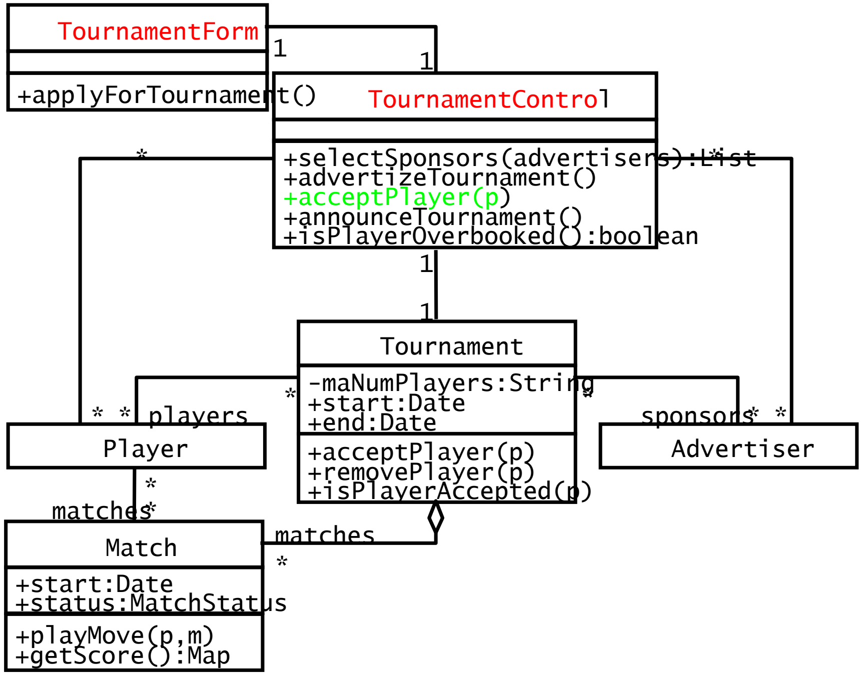
Implementing Contract Violations

- Many object-oriented languages do not have built-in support for contracts
- However, if they support exceptions, we can use their exception mechanisms for signaling and handling contract violations
- In Java we use the try-throw-catch mechanism
- Example:
  - Let us assume the acceptPlayer() operation of TournamentControl is invoked with a player who is already part of the Tournament
    - UML model (see slide 34)
  - In this case acceptPlayer() in TournamentControl should throw an exception of type KnownPlayer
    - Java Source code (see slide 35).

UML Model for Contract Violation Example



Implementation in Java



```
public class TournamentForm {
  private TournamentControl control;
  private ArrayList players;
  public void processPlayerApplications() {
   for (Iteration i = players.iterator(); i.hasNext();) {
      try {
          control.acceptPlayer((Player)i.next());
       }
      catch (KnownPlayerException e) {
        // If exception was caught, log it to console
        ErrorConsole.log(e.getMessage());
        }
       }
     }
}
```

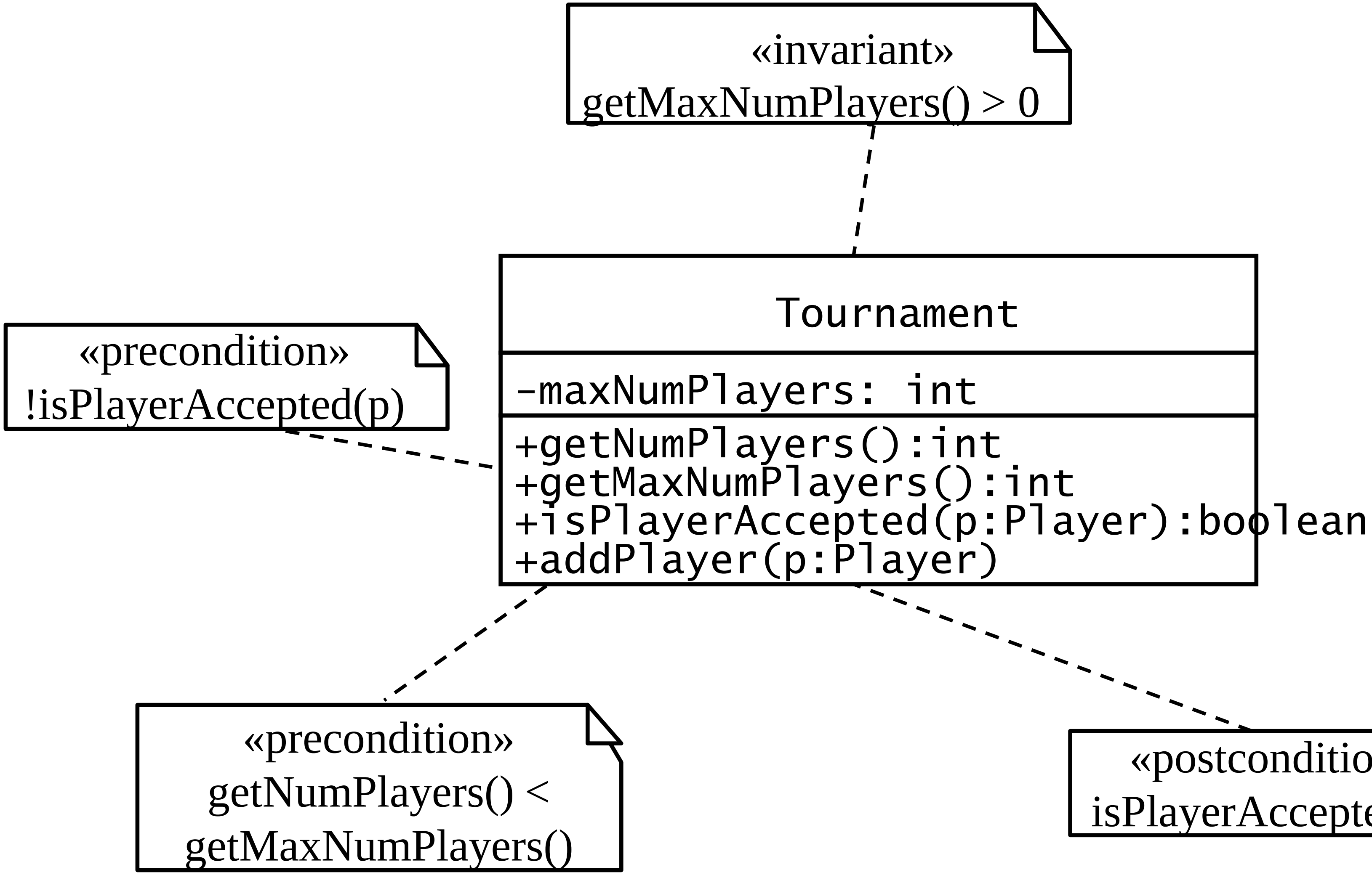
The try-throw-catch Mechanism in Java

```
public class TournamentControl {
  private Tournament tournament;
  public void acceptPlayer(Player p) throws KnownPlayerException {
      if (tournament.isPlayerAccepted(p)) {
             throw new KnownPlayerException(p);
      }
      //. . Normal addPlayer behavior
  }
}

public class TournamentForm {
  private TournamentControl control;
  private ArrayList players;
  public void processPlayerApplications() {
   for (Iteration i = players.iterator(); i.hasNext();) {
      try {
           control.acceptPlayer((Player)i.next());
       }
      catch (KnownPlayerException e) {
        // If exception was caught, log it to console
        ErrorConsole.log(e.getMessage());
        }
       }
     }
}
```

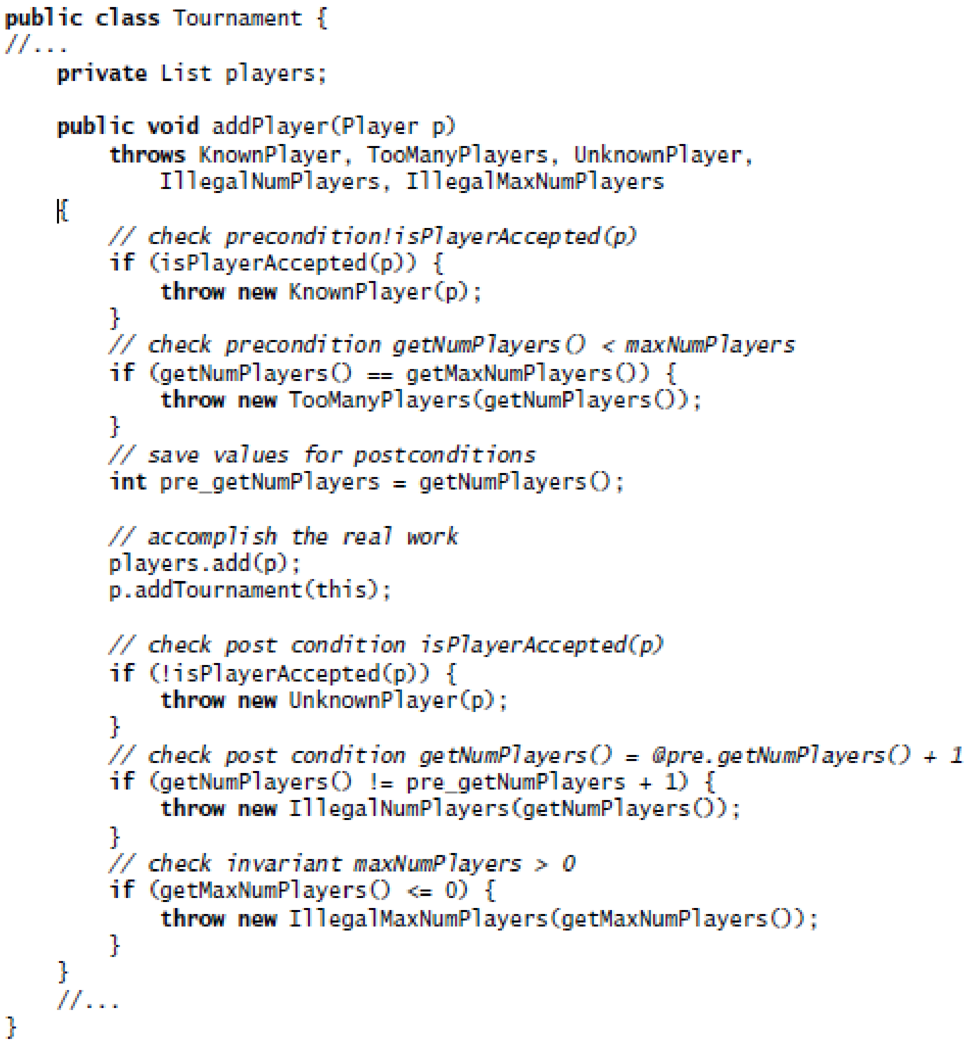
Implementing a Contract

- **Check each precondition**:
  - Before the beginning of the method with a test to check the precondition for that method
    - Raise an exception if the precondition evaluates to false
- **Check each postcondition:**
  - At the end of the method write a test to check the postcondition
    - Raise an exception if the postcondition evaluates to false. If more than one postcondition is not satisfied, raise an exception only for the first violation.
- **Check each invariant:**
  - Check invariants at the same time when checking preconditions and when checking postconditions
- **Deal with inheritance:**
  - Add the checking code for preconditions and postconditions also into methods that can be called from the class.

A complete implementation of the Tournament.addPlayer() contract



Adding constraints



```java
public class Tournament {
//...
    private List players;

    public void addPlayer(Player p)
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,
            IllegalNumPlayers, IllegalMaxNumPlayers
    {
        // check precondition!isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();

        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // check post condition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() + 1
        if (getNumPlayers() != pre_getNumPlayers + 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
    }
    //...
}
```

Heuristics: Mapping Contracts to Exceptions

- Executing checking code slows down your program
  - If it is too slow, omit the checking code for private and protected methods
  - If it is still too slow, focus on components with the longest life
    - Omit checking code for postconditions and invariants for all other components.

Heuristics for Transformations

- For any given transformation always use the same tool
- Keep the contracts in the source code, not in the object design model
- Use the same names for the same objects
- Have a style guide for transformations (Martin Fowler)

Object Design Areas

1. Service specification
   - Describes precisely each class interface
2. Component selection
   - Identify off-the-shelf components and additional solution objects
3. Object model restructuring
   - Transforms the object design model to improve its understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Design Optimizations

- Design optimizations are an important part of the object design phase:
  - The requirements analysis model is semantically correct but often too inefficient if directly implemented.
- Optimization activities during object design:
  1. Add redundant associations to minimize access cost
  2. Rearrange computations for greater efficiency
  3. Store derived attributes to save computation time
- As an object designer you must strike a balance between efficiency and clarity.
  - Optimizations will make your models more obscure

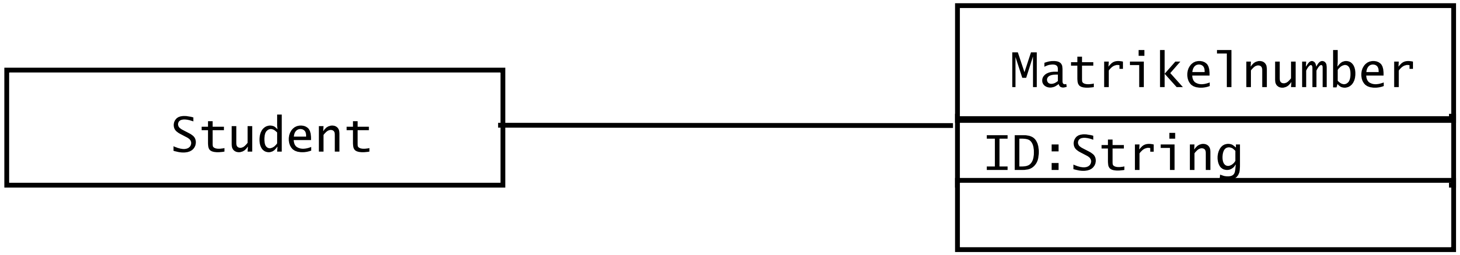
Design Optimization Activities

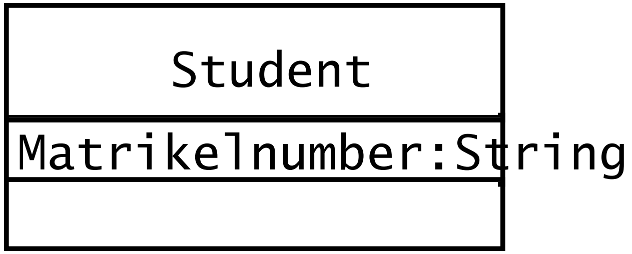
1. Add redundant associations:
   - What are the most frequent operations? (Sensor data lookup?)
   - How often is the operation called? (30 times a month, every 50 milliseconds)
2. Rearrange execution order
   - Eliminate dead paths as early as possible (Use knowledge of distributions, frequency of path traversals)
   - Narrow search as soon as possible
   - Check if execution order of loop should be reversed
3. Turn classes into attributes

Implement application domain classes

- To collapse or not collapse: Attribute or association?
- Object design choices:
    - Implement entity as embedded attribute
    - Implement entity as separate class with associations to other classes
- Associations are more flexible than attributes but often introduce unnecessary indirection
- Abbott's textual analysis rules.

Optimization Activities: Collapsing Objects





To Collapse or not to Collapse?

- Collapse a class into an attribute if the only operations defined on the attributes are Set() and Get().

Design Optimizations (continued)

Store derived attributes

- Example: Define new classes to store information locally (database cache)

- Problem with derived attributes:
  - Derived attributes must be updated when base values change.
  - There are 3 ways to deal with the update problem:
    - Explicit code: Implementor determines affected derived attributes (push)
    - Periodic computation: Recompute derived attribute occasionally (pull)
    - Active value: An attribute can designate set of dependent values which are automatically updated when active value is changed (notification, data trigger)

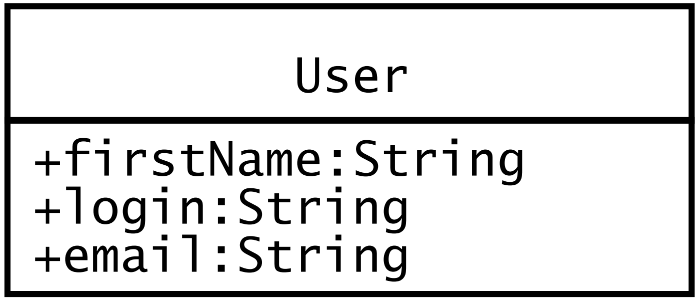
Summary

- Four mapping concepts:
  - Model transformation
  - Forward engineering
  - Refactoring
  - Reverse engineering
- Model transformation and forward engineering techniques:
  - Optiziming the class model
  - Mapping associations to collections
  - Mapping contracts to exceptions
  - Mapping class model to storage schemas

Mapping an Object Model to a Database

- UML object models can be mapped to relational databases:
  - Some degradation occurs because all UML constructs must be mapped to a single relational database construct - the table
- Mapping of classes, attributes and associations
  - Each *class* is mapped to a table
  - Each class *attribute* is mapped onto a column in the table
  - An *instance* of a class represents a row in the table
  - *A many-to-many association* is mapped into its own table
  - A *one-to-many association* is implemented as buried foreign key
- Methods are not mapped.

Mapping a Class to a Table



User

+firstName:String
+login:String
+email:String

User table

id:long	firstName:text[25]	login:text[8]	email:text[32]

Primary and Foreign Keys

- Any set of attributes that could be used to uniquely identify any data record in a relational table is called a **candidate key**
- The actual candidate key that is used in the application to identify the records is called the **primary key**
  - The primary key of a table is a set of attributes whose values uniquely identify the data records in the table
- A **foreign key** is an attribute (or a set of attributes) that references the primary key of another table.

Example for Primary and Foreign Keys

User table

Primary key

firstName	login	email
“alice”	“am384”	“am384@mail.org”
“john”	“js289”	“john@mail.de”
“bob”	“bd”	“bobd@mail.ch”

Candidate key

Candidate key

League table

name	login
“tictactoeNovice”	“am384”
“tictactoeExpert”	“bd”
“chessNovice”	“js289”

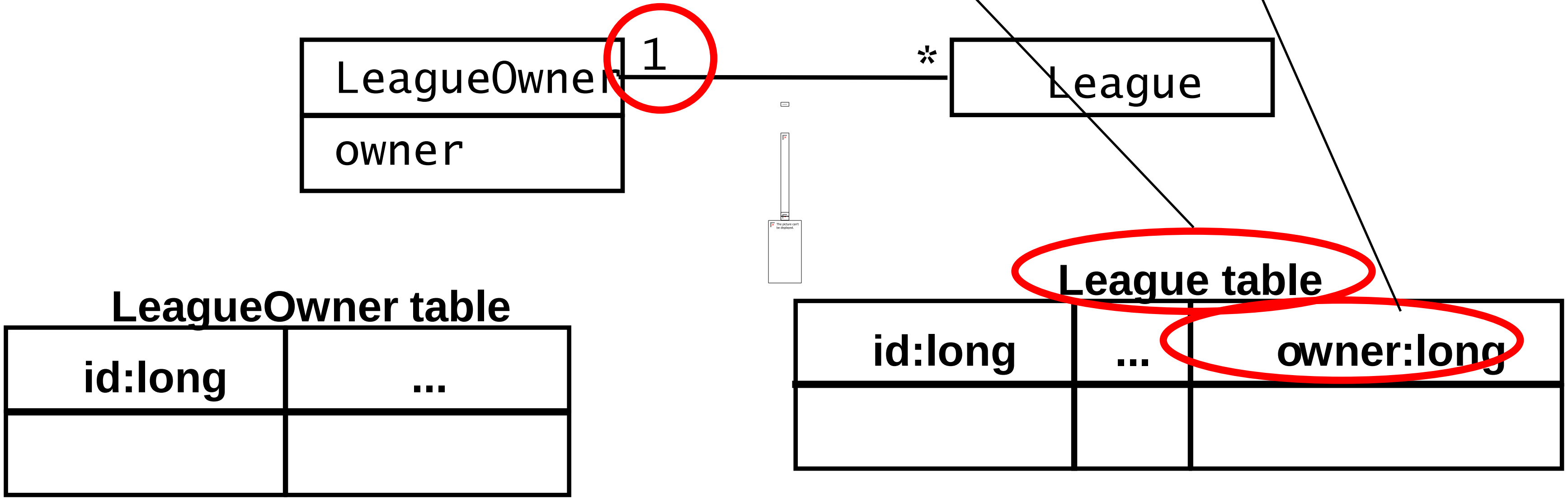
Foreign key referencing **User table**

Buried Association

- Associations with multiplicity “one” can be implemented using a foreign key

For one-to-many associations we add the foreign key to the table representing the class on the “many” end.

For all other associations we can select either class at the end of the association.



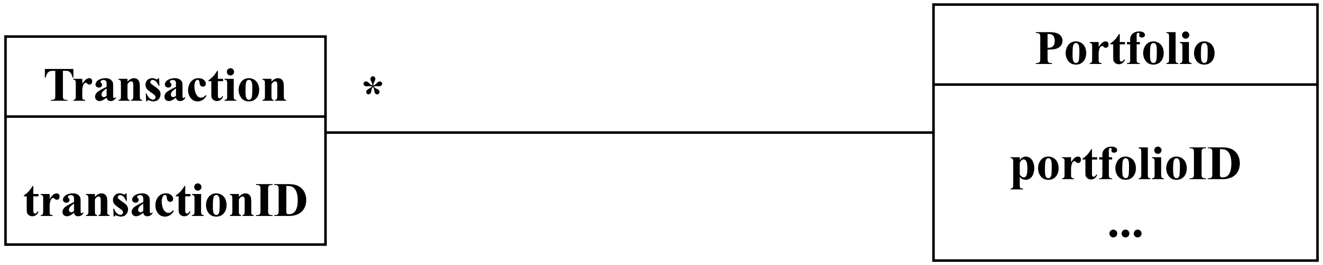
LeagueOwner table

id:long	...

League table

id:long	...	owner:long

Another Example for Buried Association



Transaction Table

transactionID	portfolioID

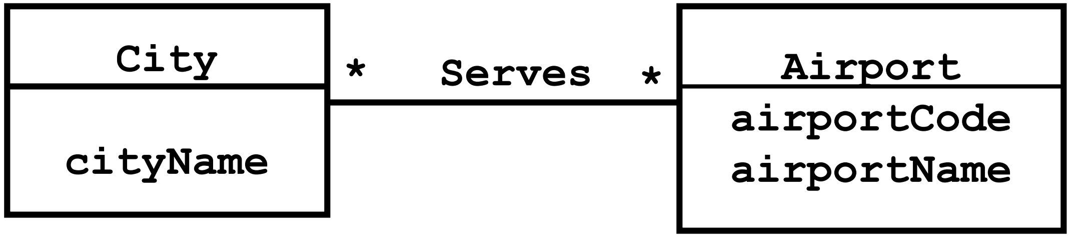
Foreign Key

Portfolio Table

portfolioID	...

Mapping Many-To-Many Associations

In this case we need a separate table for the association



Separate table for
the association “Serves”

Primary Key

City Table

cityName
Houston
Albany
Munich
Hamburg

Airport Table

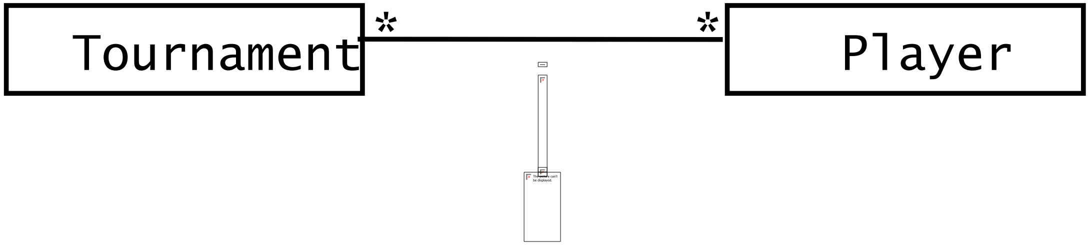
airportCode	airportName
IAH	Intercontinental
HOU	Hobby
ALB	Albany County
MUC	Munich Airport
HAM	Hamburg Airport

Serves Table

cityName	**airportCode**
Houston	IAH
Houston	HOU
Albany	ALB
Munich	MUC
Hamburg	HAM

Another Many-to-Many Association Mapping

We need the Tournament/Player association as a separate table



Tournament table

id	name	...
23	novice	
24	expert	

TournamentPlayerAssociation table

tournament	player
23	56
23	79

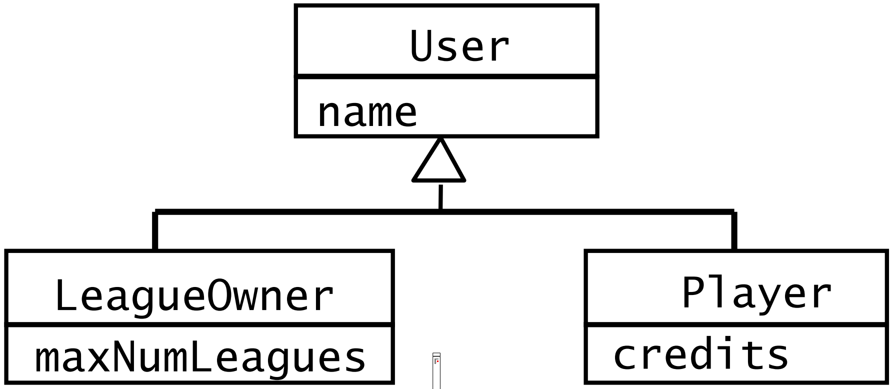
Player table

id	name	...
56	alice	
79	john	

Realizing Inheritance

- Relational databases do not support inheritance
- Two possibilities to map an inheritance association to a database schema
  - ➡ With a separate table ("vertical mapping")
    - The attributes of the superclass and the subclasses are mapped to different tables
  - By duplicating columns ("horizontal mapping")
    - There is no table for the superclass
    - Each subclass is mapped to a table containing the attributes of the subclass and the attributes of the superclass

Realizing inheritance with a separate table (Vertical mapping)



User table

id	name	...	role
56	zoe		LeagueOwner
79	john		Player

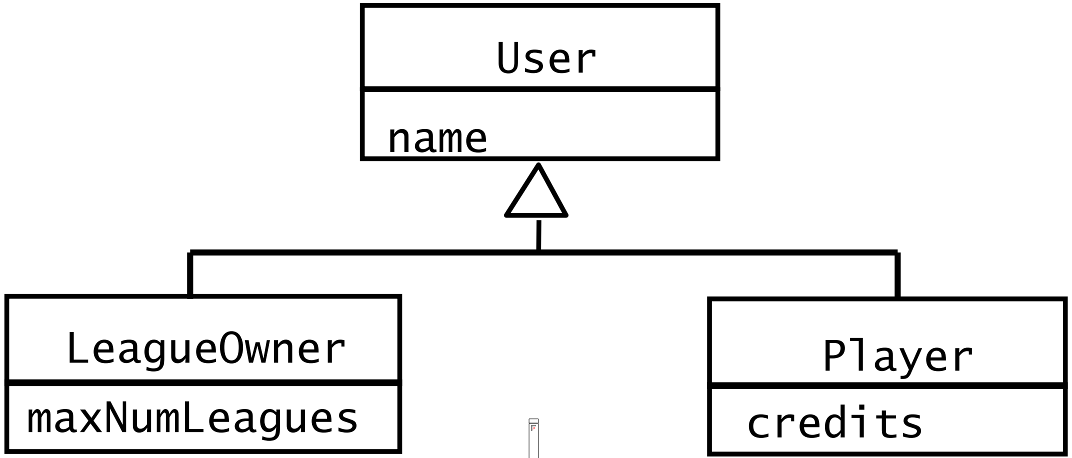
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Realizing inheritance by duplicating columns (Horizontal Mapping)



LeagueOwner table

id	name	maxNumLeagues	...
56	zoe	12	

Player table

id	name	credits	...
79	john	126	

Comparison: Separate Tables vs Duplicated Columns

- The trade-off is between modifiability and response time
  - How likely is a change of the superclass?
  - What are the performance requirements for queries?
- Separate table mapping (Vertical mapping)
  - ☺We can add attributes to the superclass easily by adding a column to the superclass table
  - ☹Searching for the attributes of an object requires a join operation.
- Duplicated columns (Horizontal Mapping)
  - ☹Modifying the database schema is more complex and error-prone
  - ☺Individual objects are not fragmented across a number of tables, resulting in faster queries

Summary

- Four mapping concepts:
  - Model transformation improves the compliance of the object design model with a design goal
  - Forward engineering improves the consistency of the code with respect to the object design model
  - Refactoring improves code readability/modifiability
  - Reverse engineering discovers the design from the code.
- Model transformations and forward engineering techniques:
  - Optimizing the class model
  - Mapping associations to collections
  - Mapping contracts to exceptions
  - Mapping class model to storage schemas.