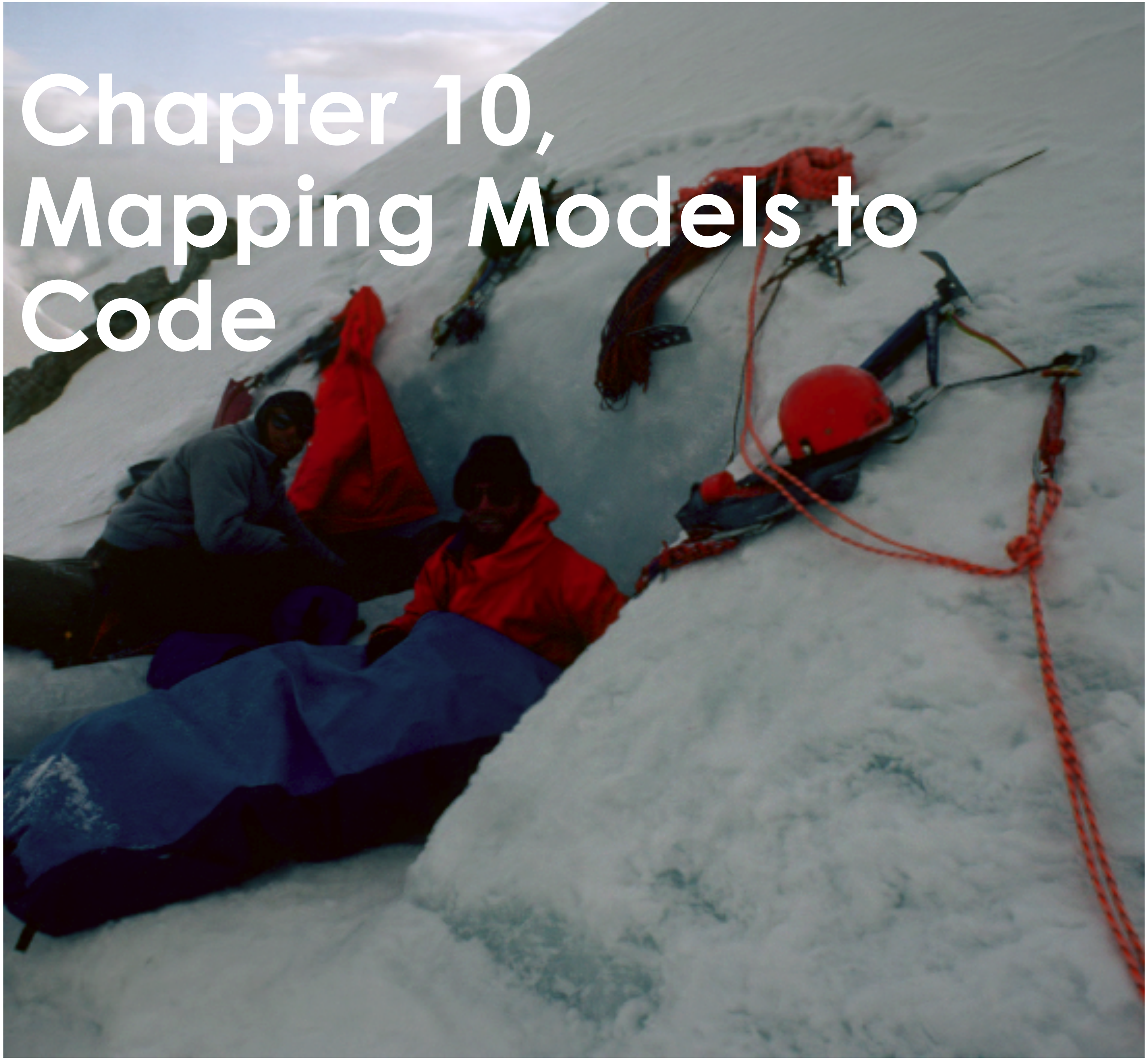
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 10, Mapping Models to Code



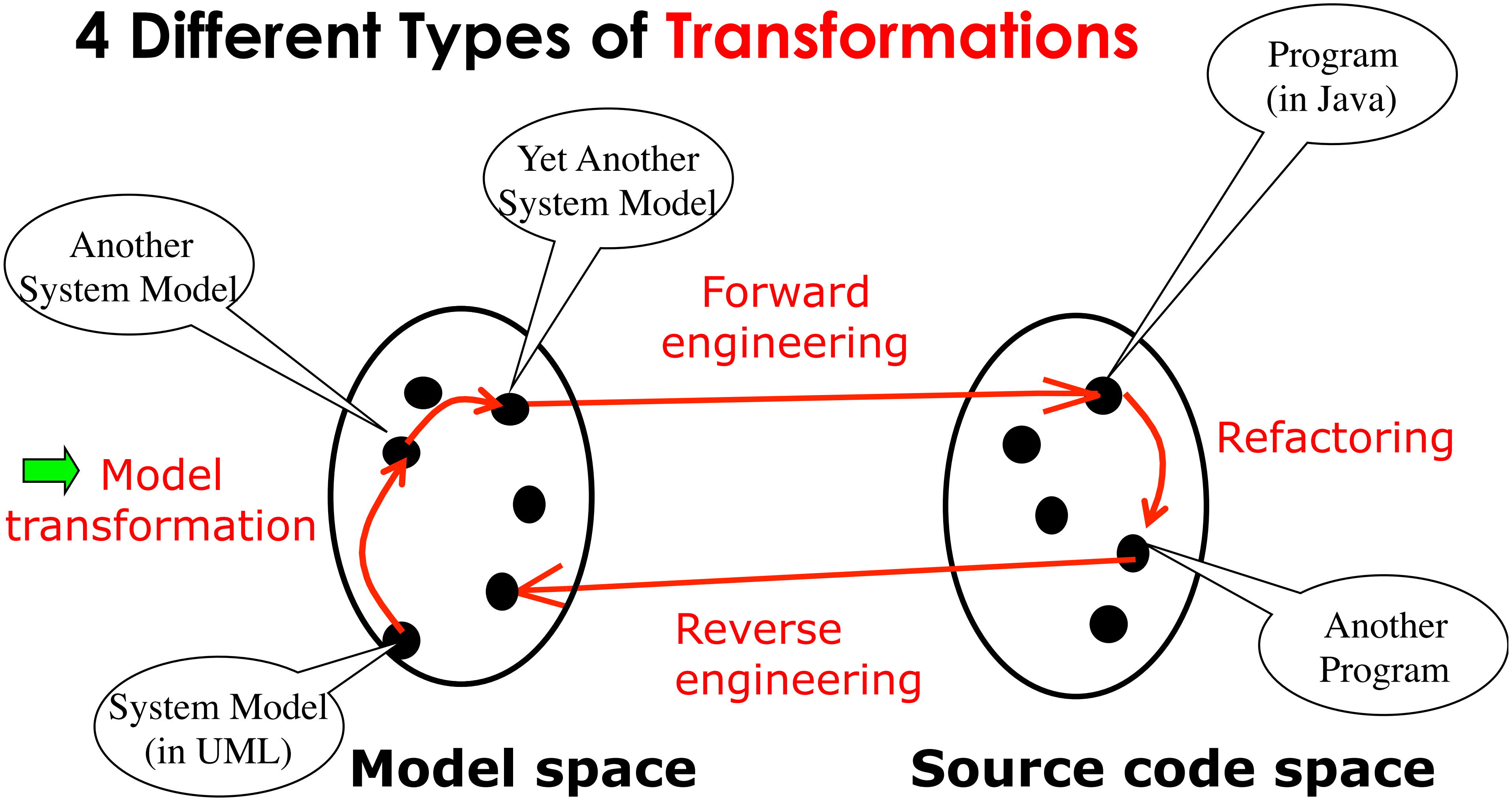
Lecture Plan

- Part 1
  - Operations on the object model:
    - Optimizations to address performance requirements
  - Implementation of class model components:
    - Realization of associations
    - Realization of operation contracts
- Part 2
  - Realizing entity objects based on selected storage strategy
  - Mapping the object model to a storage schema
  - Mapping class diagrams to tables

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.

4 Different Types of Transformations



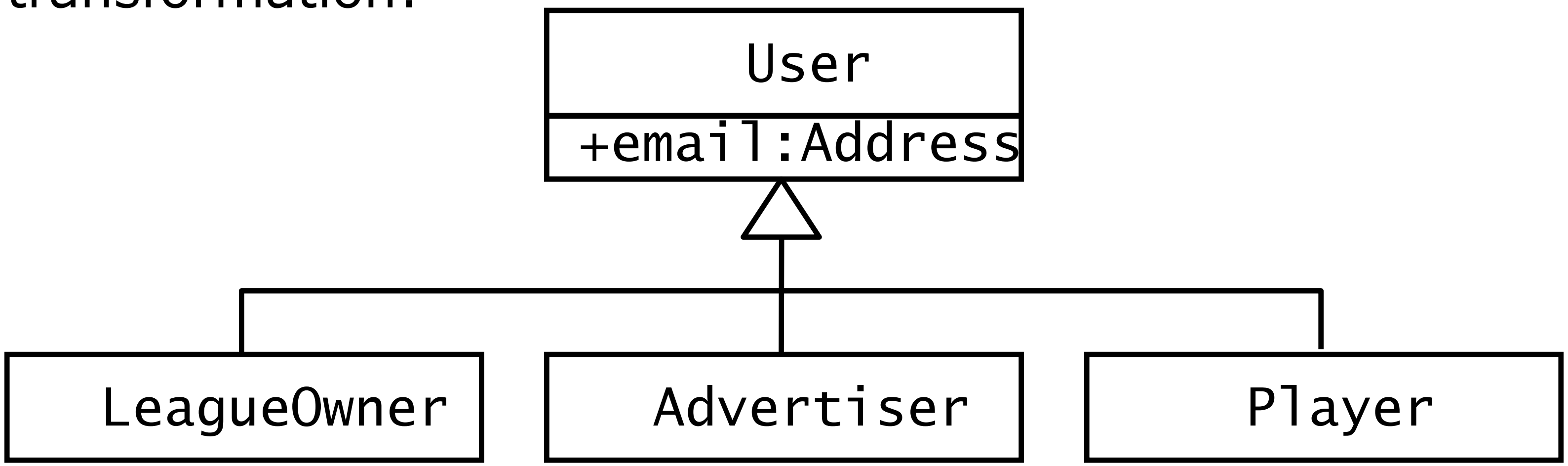
Model Transformation

- 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

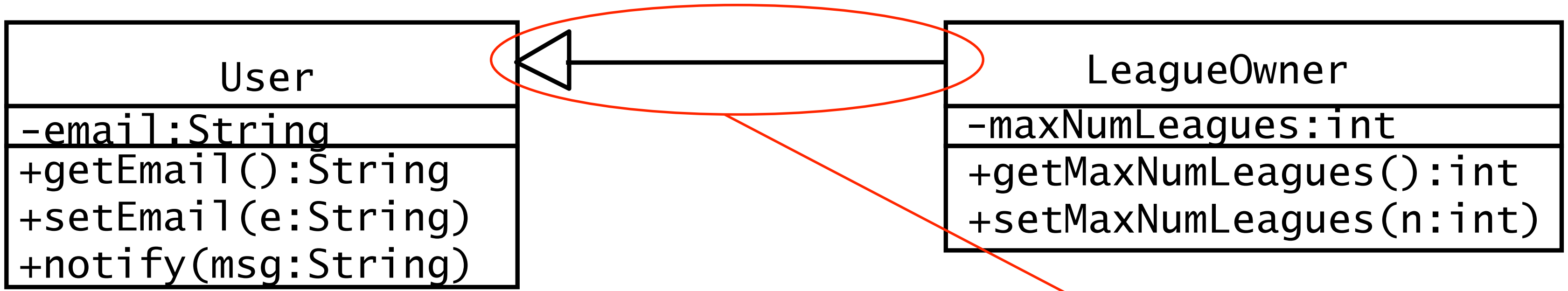
Object design model before transformation:

Object design model
after transformation:



Forward Engineering Example

Object design model before transformation:



Source code after transformation:

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

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

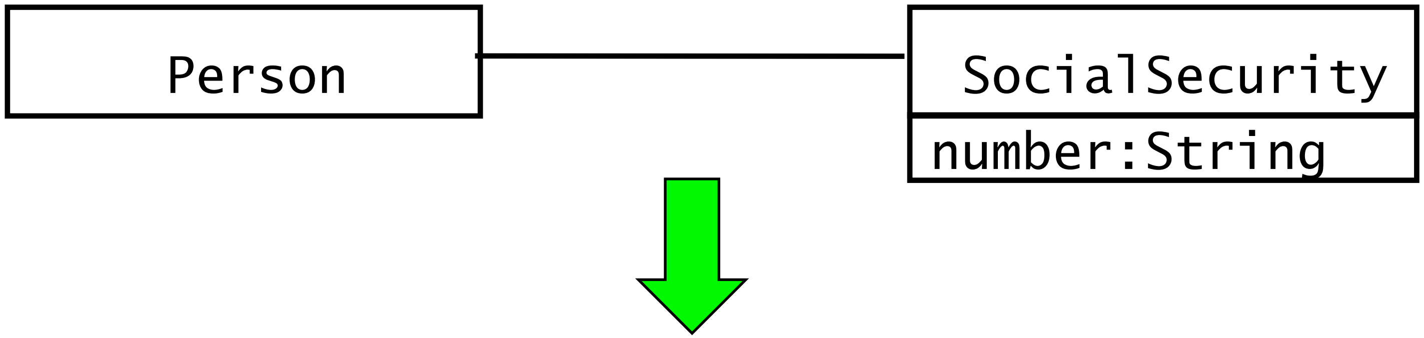
More Forward Engineering Examples

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

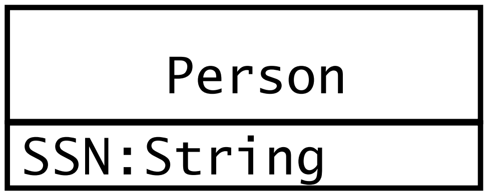
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - Mapping inheritance
  - Mapping associations
  - Mapping contracts to exceptions
  - Mapping object models to tables

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

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:
  - Overwriting of methods (default in Java)
  - Final classes
  - Final methods
  - Abstract methods
  - Abstract classes
  - Interfaces.

Inheritance: Let's recall something

Implementation Inheritance and Specification Inheritance

There are two different types of inheritance:

- Implementation inheritance
  - Also called class inheritance
  - Goal:
    - Extend an applications' functionality by reusing functionality from the super class
    - Inherit from an existing (concrete) class with **some or all operations already implemented**
- Specification Inheritance
  - Also called subtyping
  - Goal:
    - Inherit from a specification
    - The specification is an abstract class with **all the operations** specified but **not yet implemented**.

Implementation Inheritance vs. Specification Inheritance

	Interface (of superclass)	Implementations of methods (of superclass)
Implementation Inheritance	Inherited	Inherited
Specification Inheritance	Inherited	NOT inherited

The Liskov Substitution Principle for specification inheritance

- The Liskov Substitution Principle [Liskov, 1988] provides a formal definition for **specification inheritance**.
- It essentially states that, if a client code uses the methods provided by a superclass, then developers should be able to add new subclasses without having to change the client code.

- Liskov Substitution Principle
  - If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

- Interpretation
  - In other words, a method written in terms of a superclass T must be able to use instances of any subclass of T without knowing whether the instances are of a subclass.
- An inheritance relationship that complies with the Liskov Substitution Principle is called **strict inheritance**.

Now let's go back to realizing inheritance

Realizing Inheritance in Java

- Realisation of specialization and generalization
  - Definition of subclasses
  - Java keyword: **`extends`**
- Realisation of strict inheritance
  - Overwriting of methods is not allowed
  - Java keyword: **`final`**
- Realisation of implementation inheritance
  - Overwriting of methods
  - No keyword necessary:
    - Overwriting of methods is default in Java
- Realisation 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

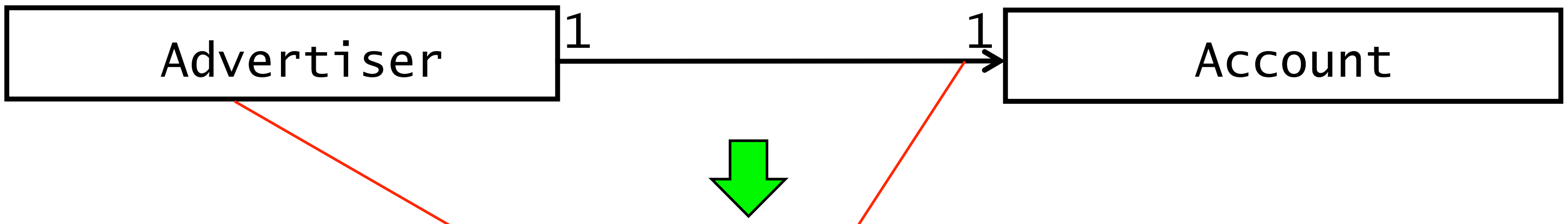
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - ✓ Mapping inheritance
  - ➡ Mapping associations
  - Mapping contracts to exceptions
  - Mapping object models to tables

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:

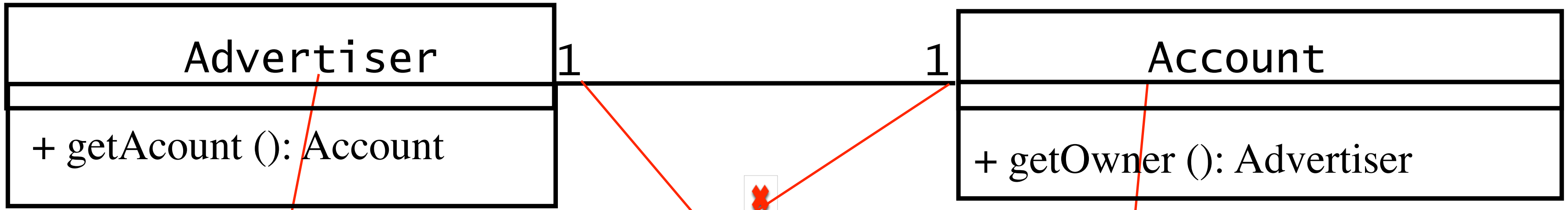


Source code after transformation:

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

Bidirectional one-to-one association

Object design model before transformation:



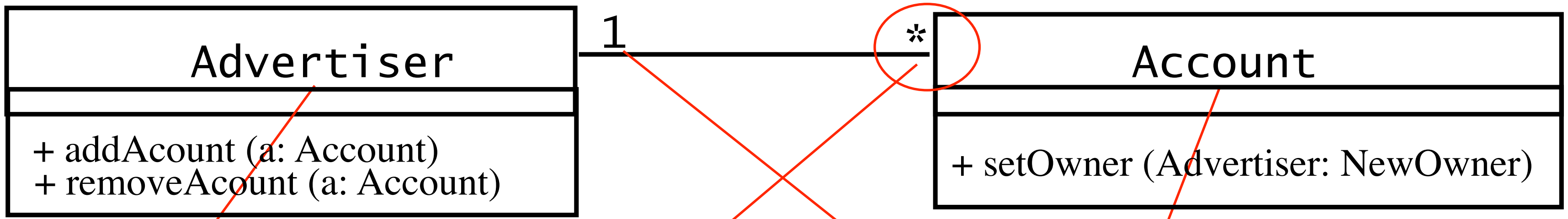
Source code after transformation:

```
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:

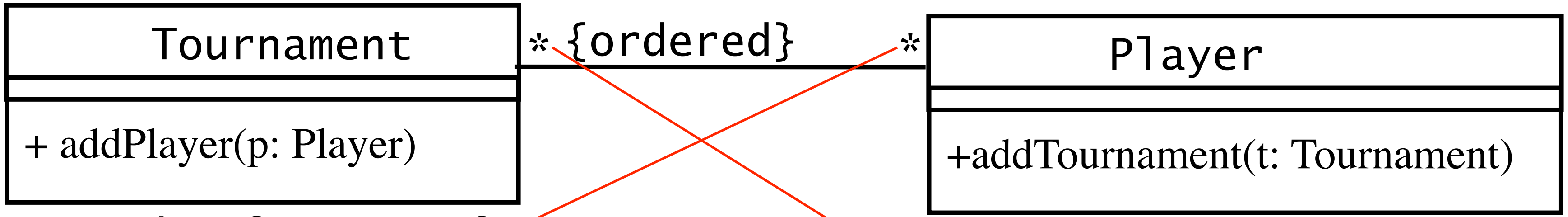
```
public class Advertiser {
   private Set accounts;
   public Advertiser() {
         accounts = new HashSet();
   }
   public void addAccount(Account a) {
         accounts.add(a);
         a.setOwner(this);
   }
   public void removeAccount(Account a) {
         accounts.remove(a);
         a.setOwner(null);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

```
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);
   }
```

A List of Account would be a more common choice:
List<Account> account= new ArrayList<Account>();

Bidirectional many-to-many association

Object design model before transformation



Source code after transformation

```java
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);
     }
  }
}
```

```java
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
  - Next! ➡ 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
  - In this case acceptPlayer() in TournamentControl should throw an exception of type KnownPlayer
    - Java Source code.

The Try-throw-catch mechanism

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following:

```
try {
    code
}
catch (ExceptionType name) {
…
}
catch (ExceptionType name) {
…
}
```

From:
Catching and Handling Exceptions
http://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html

The Try-throw-catch mechanism/2

- Each catch block is an exception handler and handles the type of exception indicated by its argument.
  - The argument type, ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the **Throwable** class. The handler can refer to the exception with name.

```
try {
    code
}
catch (ExceptionType name) {
…
}
catch (ExceptionType name) {
…
}
```

The Try-throw-catch mechanism/3

- The catch block contains code that is executed if and when the exception handler is invoked.

```
try {
      code
}
catch (ExceptionType name) {
…
}
catch (ExceptionType name) {
…
}
```

The Try-throw-catch mechanism/4

- The following are examples of exception handlers
- The first handler, in addition to printing a message, throws a user-defined exception: SampleException(e).

```
try {
      code

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " +
                                             e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: " +
                                             e.getMessage());
}
```

The Try-throw-catch mechanism/5

- The **finally** block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

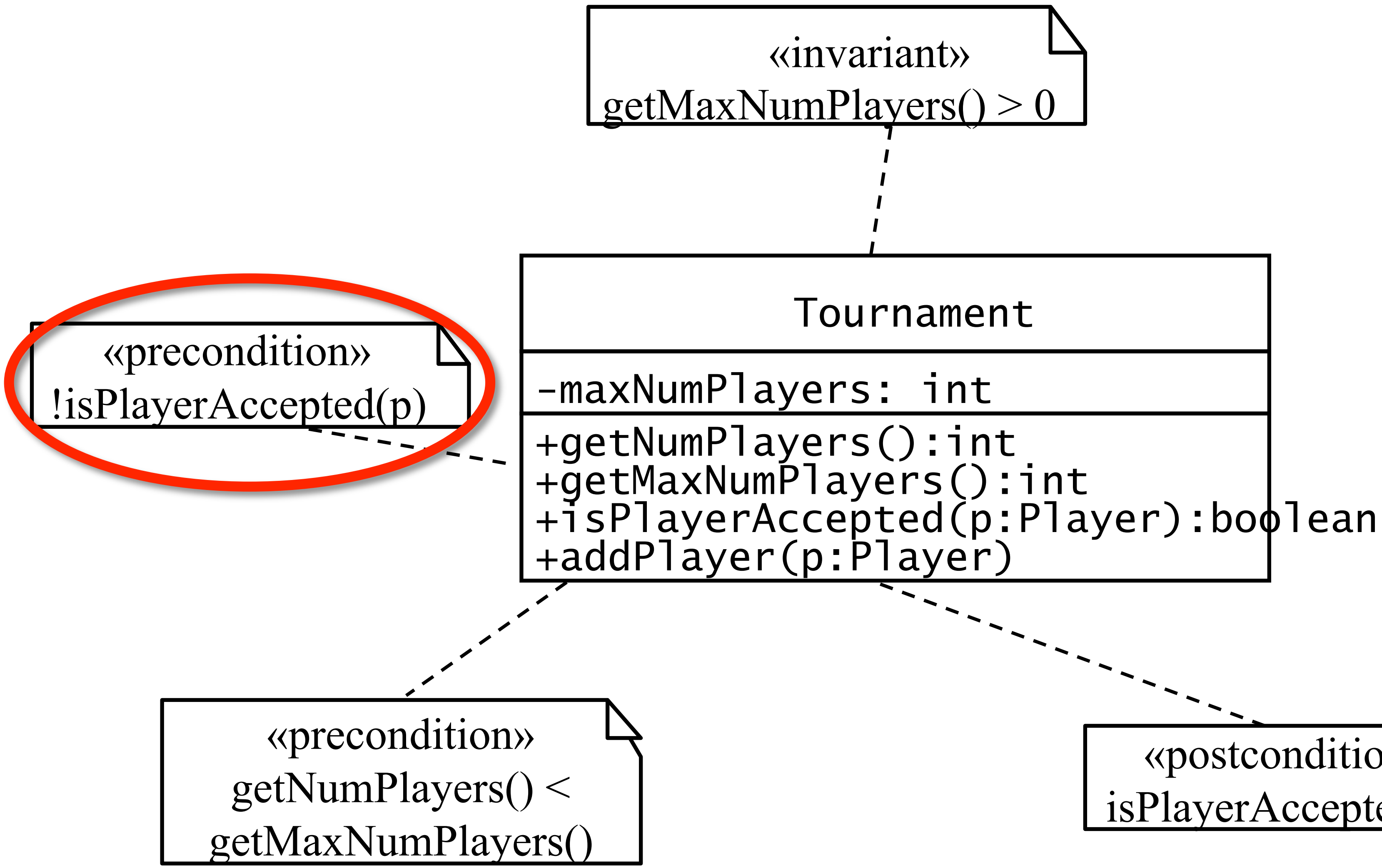
```
try {

} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " +
                                        e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: " +
                                        e.getMessage());
}
finally {
     if (out != null) {
          System.out.println("Closing PrintWriter");
          out.close();
     } else {
          System.out.println("PrintWriter not open");
     }
}
```

Implementing a Contract

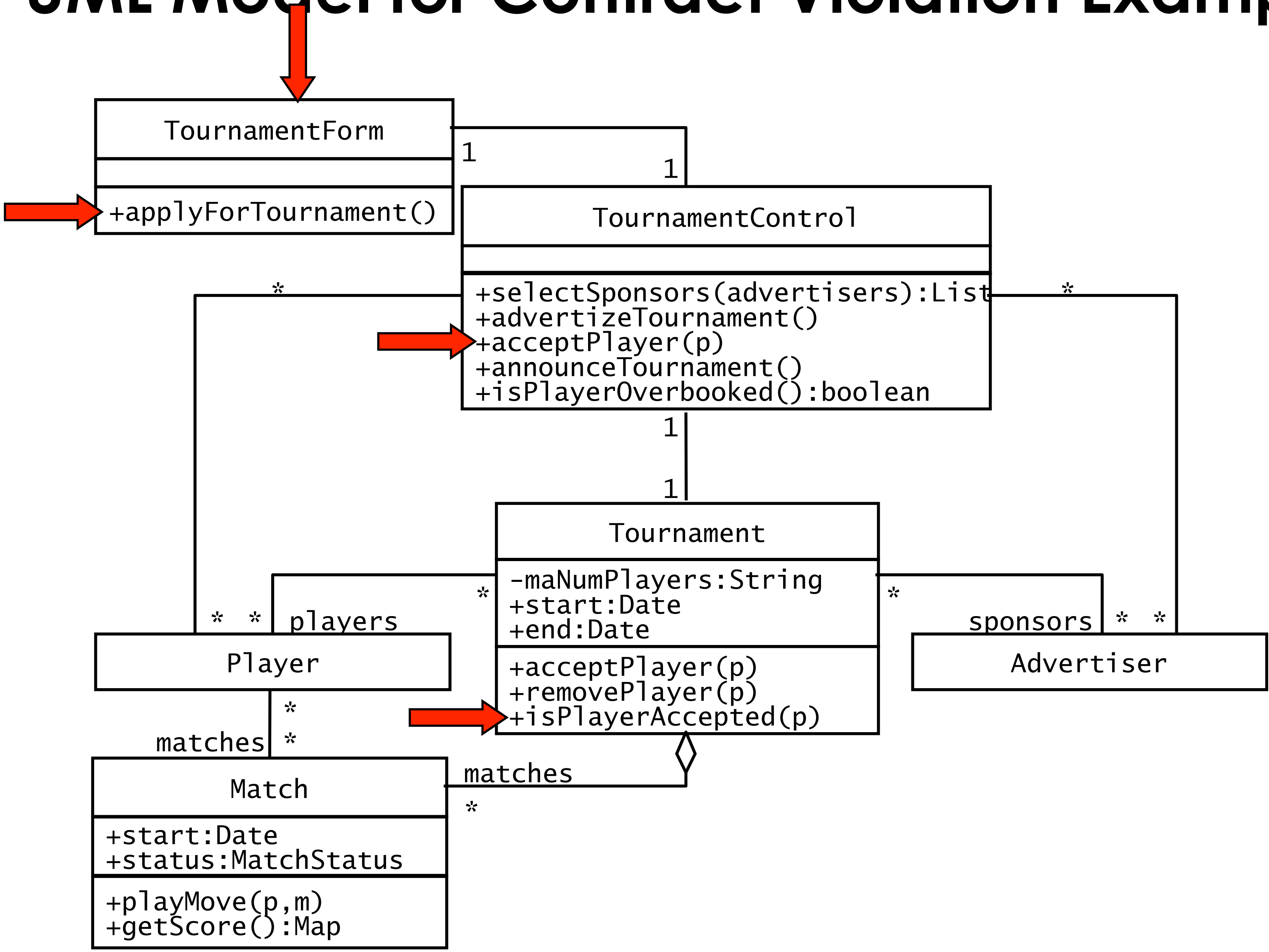
- **Check each precondition**:
  - Before the beginning of the method with a test to check the precondition for that method
    - Raise an exception if the precondition evaluates to false
- **Check each postcondition:**
  - At the end of the method write a test to check the postcondition
    - Raise 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 subclass (protected methods).

An example of contract specification in Java

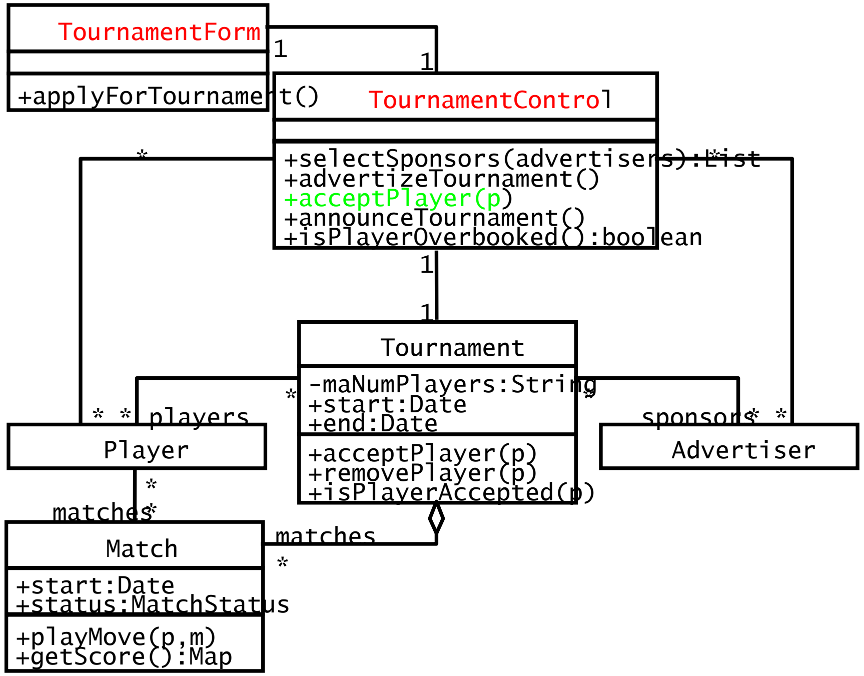
A complete implementation of the Tournament.addPlayer() contract



UML Model for Contract Violation Example



Implementation in Java




```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());
          }
        }
      }
}
```

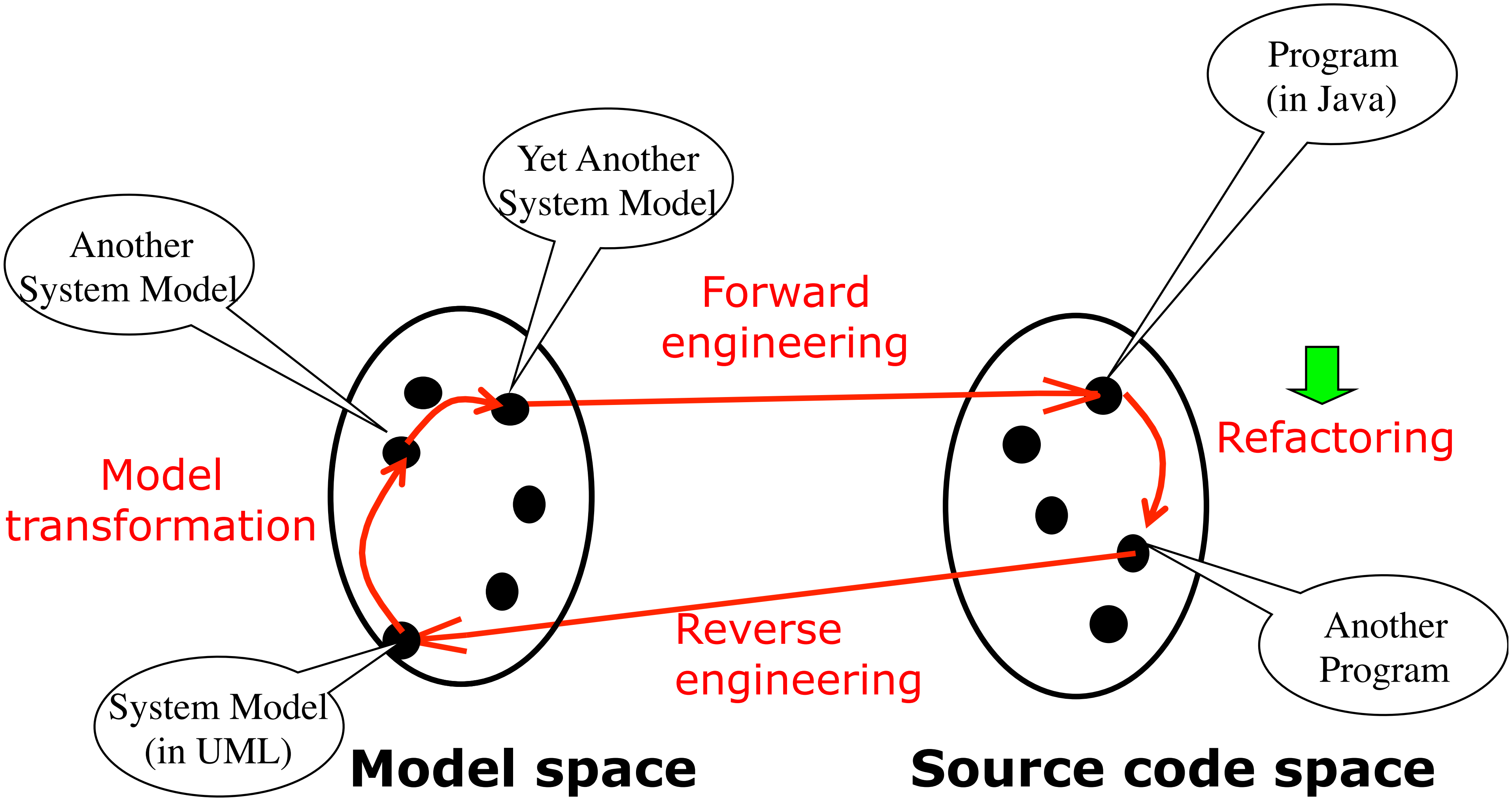
The try-throw-catch Mechanism in Java

```
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());
          }
         }
       }
 }
```

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

Additional Slides


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

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

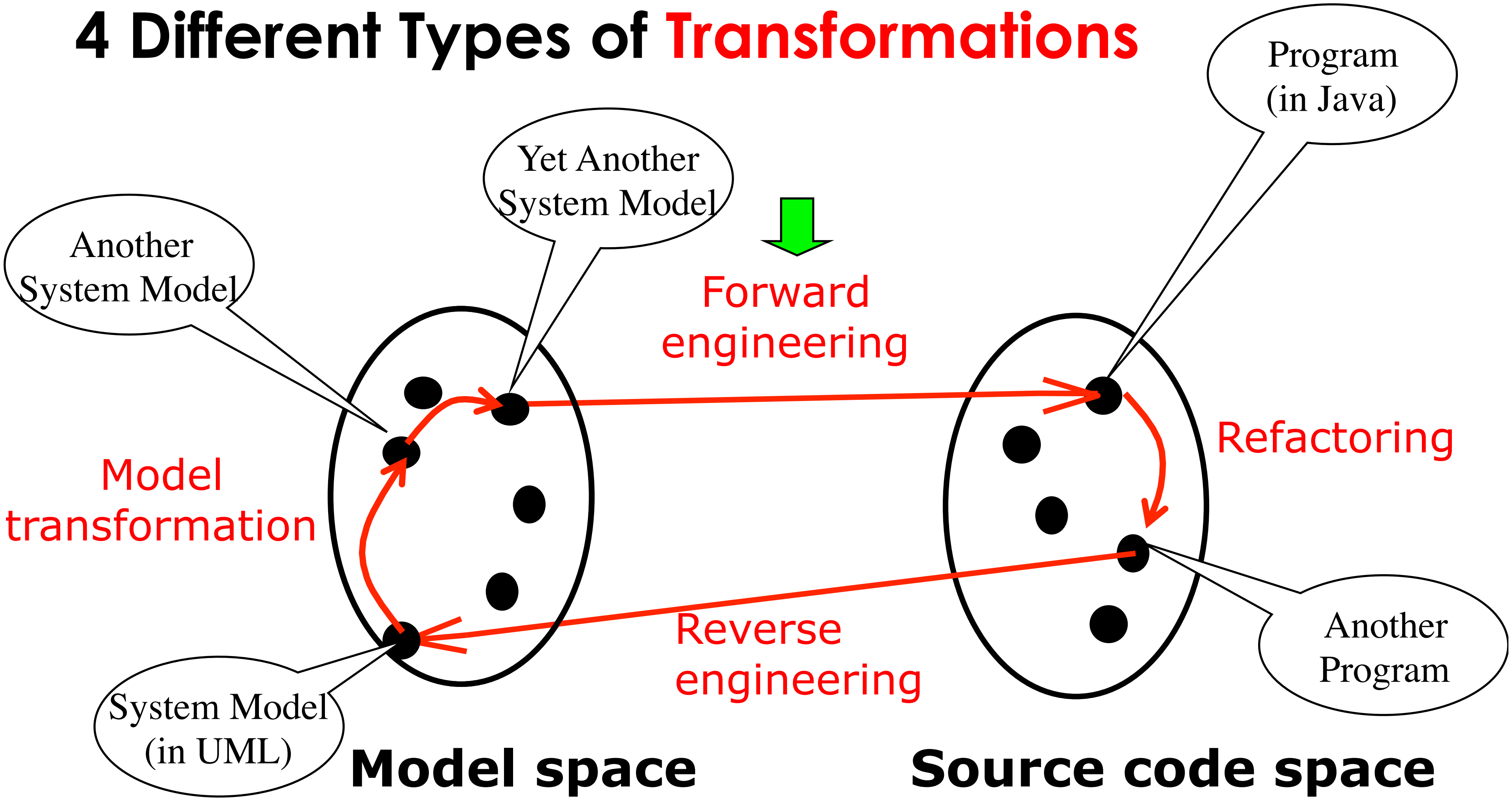
```java
public class User {
  private String email;
}

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

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

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

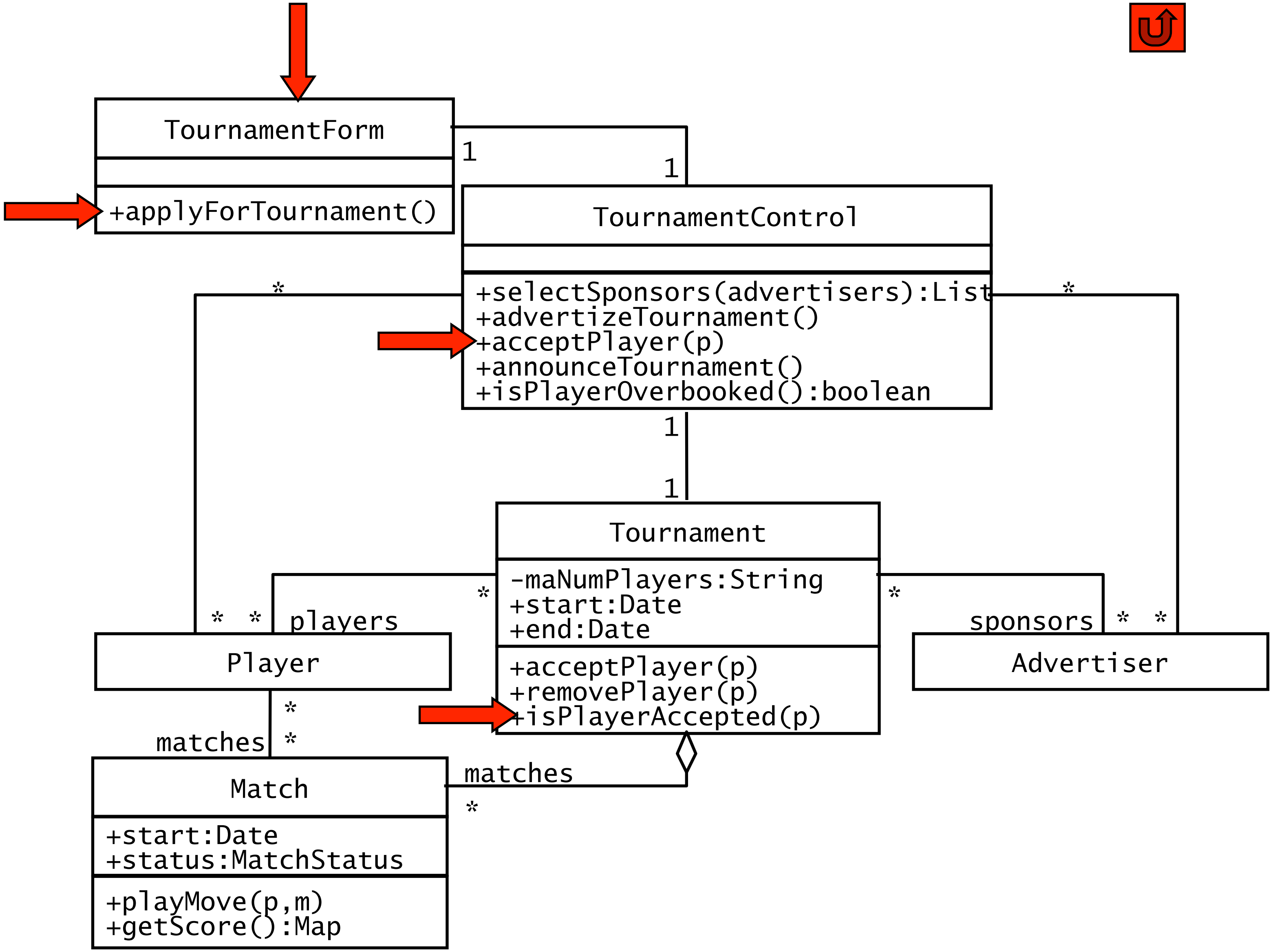
Refactoring Example: Pull Up Constructor Body

```java
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;
   }
}
```

```java
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





TournamentForm
+applyForTournament()
1
1
TournamentControl
+selectSponsors(advertisers):List
+advertizeTournament()
+acceptPlayer(p)
+announceTournament()
+isPlayerOverbooked():boolean
*
*
1
1
Tournament
-maNumPlayers:String
+start:Date
+end:Date
+acceptPlayer(p)
+removePlayer(p)
+isPlayerAccepted(p)
players
*
*
*
sponsors
*
*
*
Player
Advertiser
matches
*
*
Match
+start:Date
+status:MatchStatus
+playMove(p,m)
+getScore():Map
matches
*