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.

Solution

- 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 space

Source code space
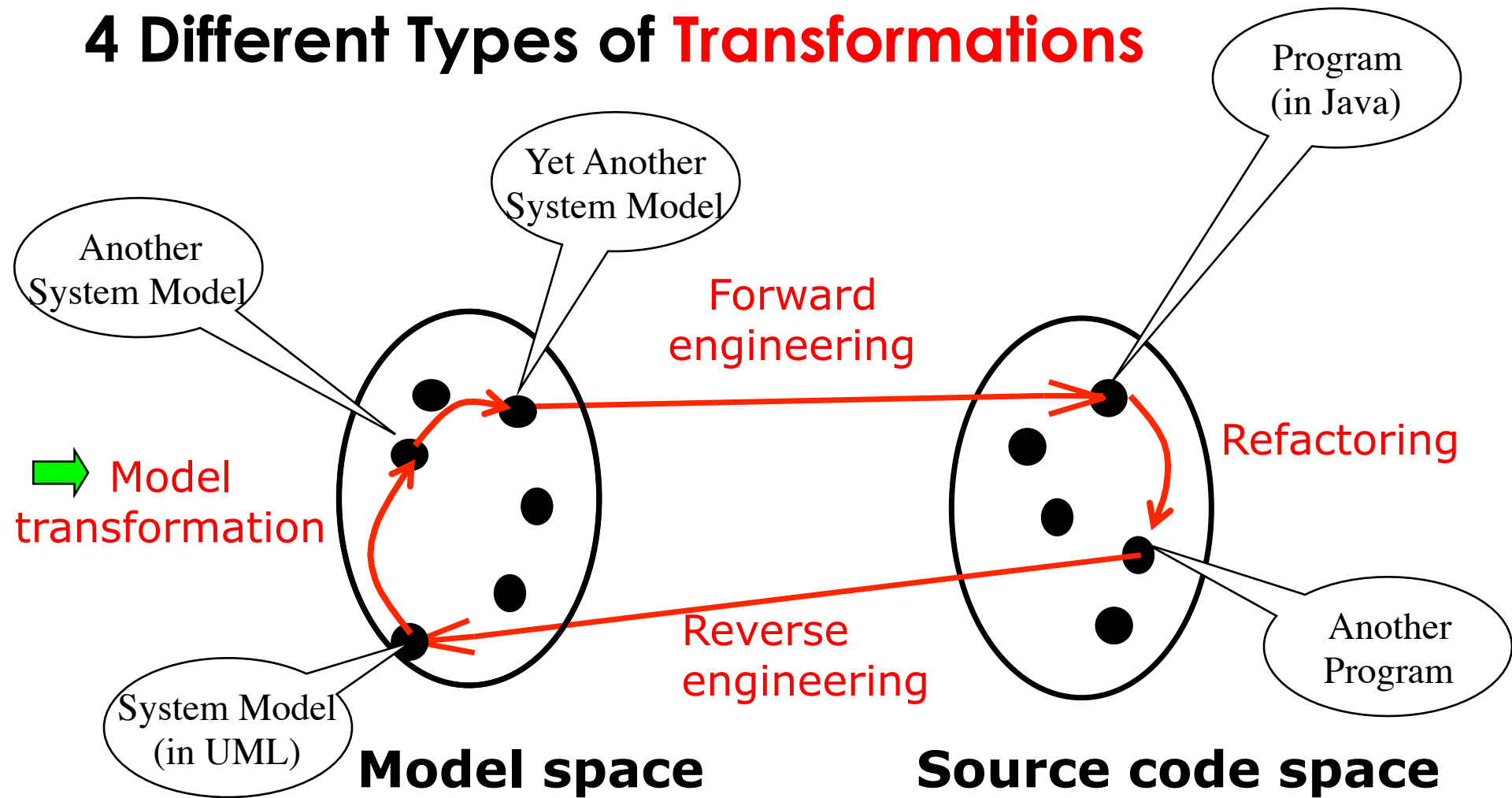
Model Transformation Example

Object design model before transformation:
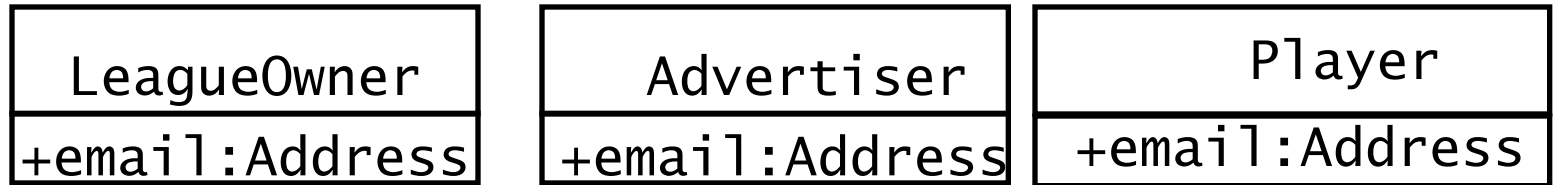
LeagueOwner
+email:Address

Advertiser
+email:Address

Player
+email:Address

Object design model
after transformation:

User
+email:Address

```
        LeagueOwner        Advertiser        Player
```

4 Different Types of **Transformations**

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
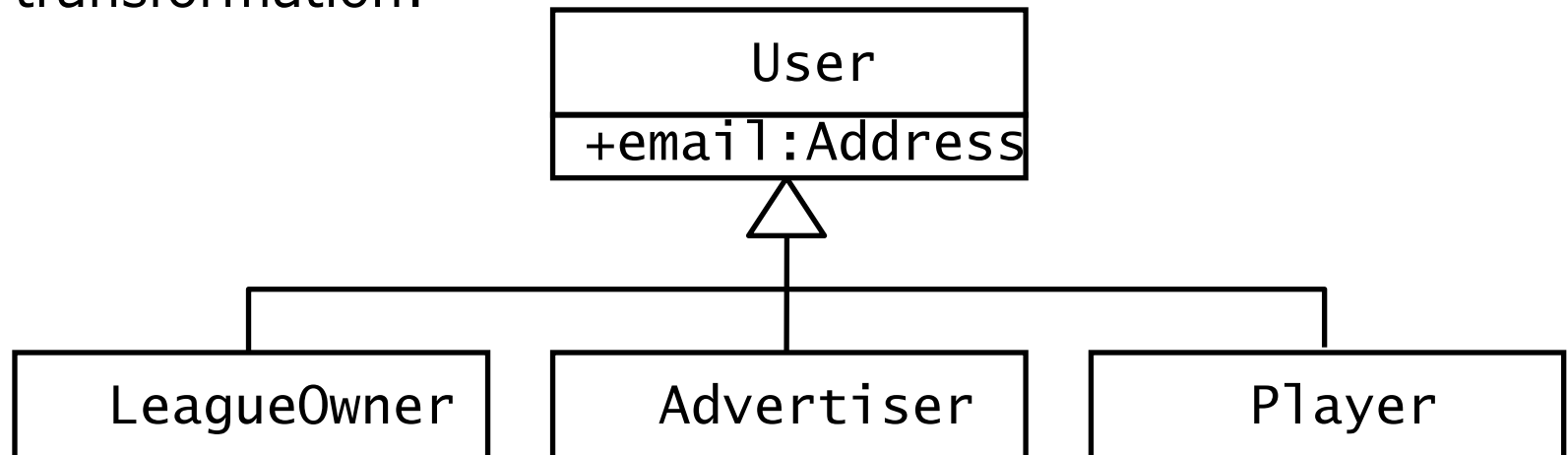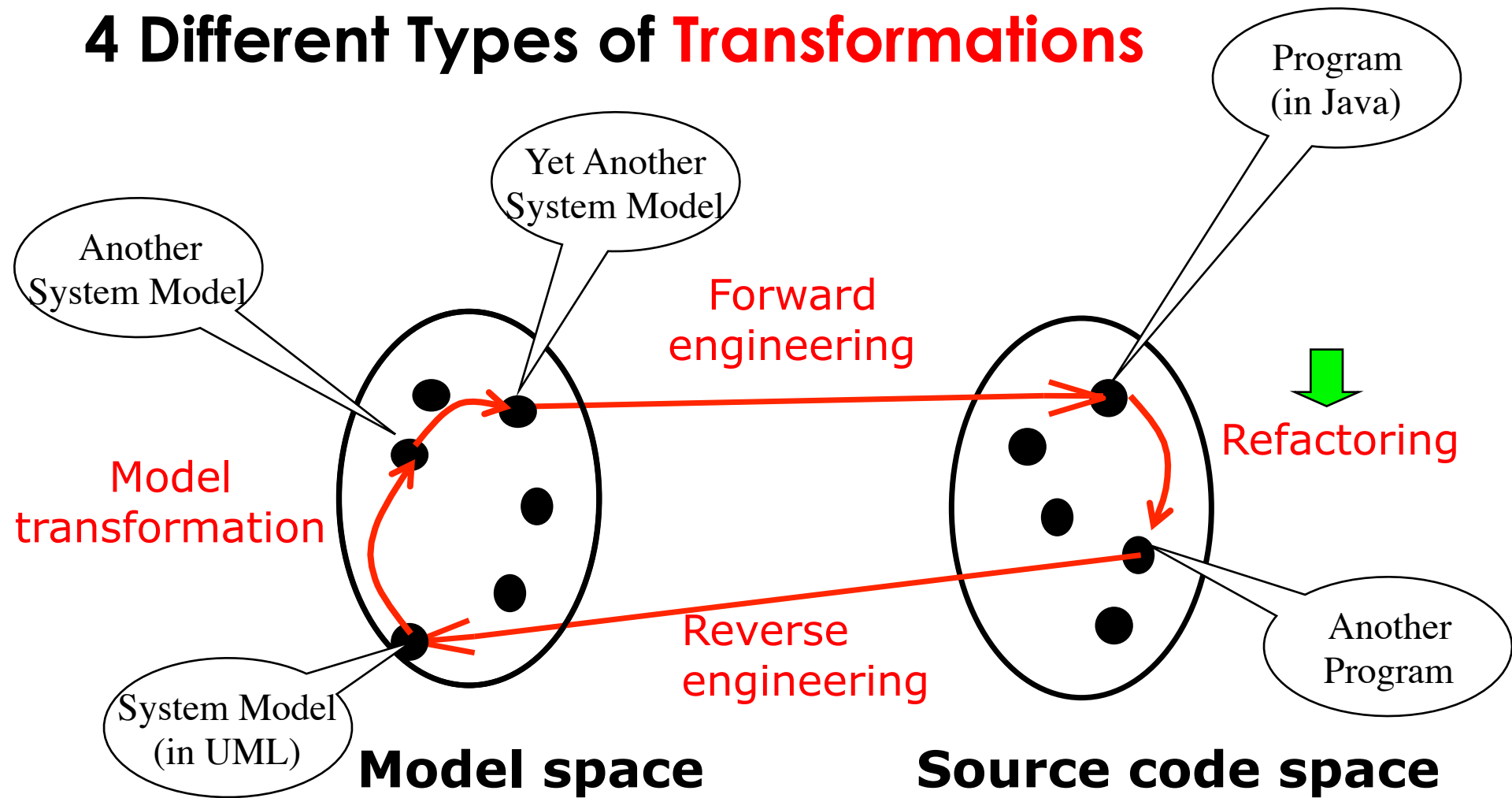
Refactoring Example: Pull Up Field

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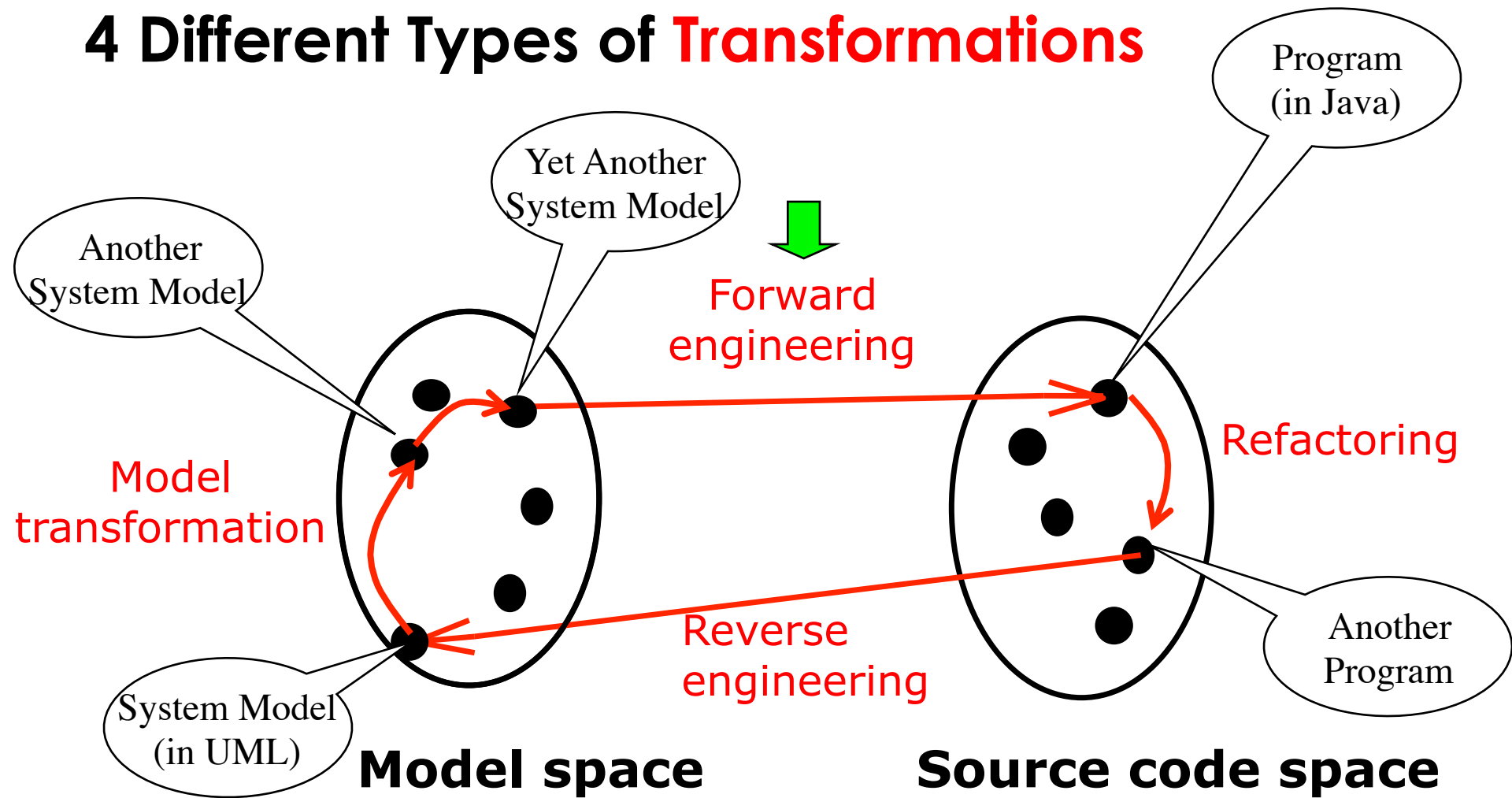
```java
public class User {
   private String email;
}
public class Player extends User {
   //...
}

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 extendsUser{
    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**

Program (in Java)

Yet Another System Model

Another System Model

Forward engineering

Model transformation

Refactoring

System Model (in UML)

Reverse engineering

Another Program

Model space

Source code space

Forward Engineering Example

Object design model before transformation:

```
┌─────────────────────────────┐          ┌─────────────────────────────┐
│            User             │◁─────────│         LeagueOwner         │
├─────────────────────────────┤          ├─────────────────────────────┤
│ -email:String               │          │ -maxNumLeagues:int          │
├─────────────────────────────┤          ├─────────────────────────────┤
│ +getEmail():String          │          │ +getMaxNumLeagues():int     │
│ +setEmail(e:String)         │          │ +setMaxNumLeagues(n:int)    │
│ +notify(msg:String)         │          └─────────────────────────────┘
└─────────────────────────────┘
```
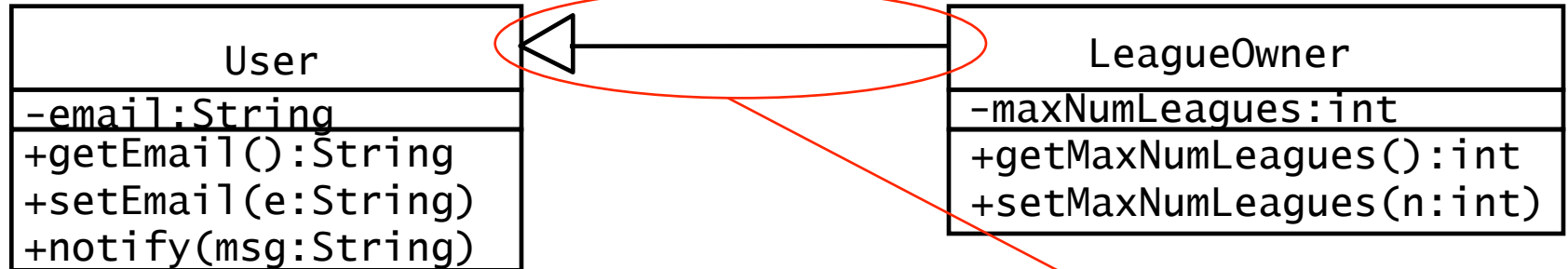
Source code after transformation: ✖

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

```java
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:

Person

SocialSecurity
number:String

Object design model after transformation:

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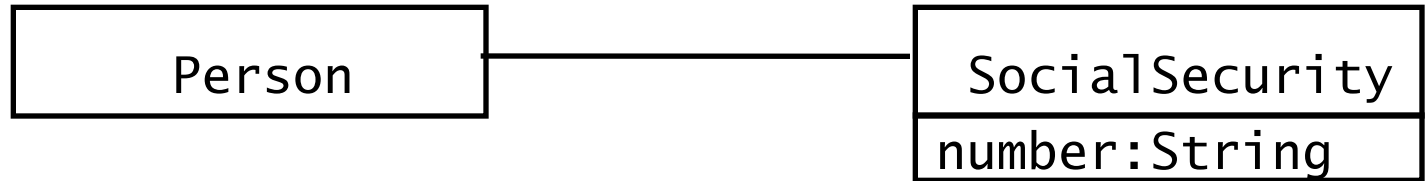
Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

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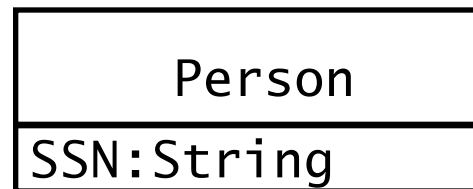
- Model Transformations
    - Goal: Optimizing the object design model
        - Collapsing objects
    - ➡ Delaying expensive computations
- Forward Engineering
    - Goal: Implementing the object design model in a programming language
    - Mapping inheritance
    - Mapping associations
    - Mapping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

Delaying expensive computations

Object design model before transformation:
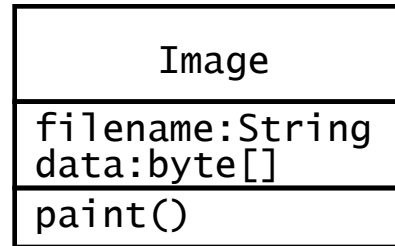
```
┌─────────────────────┐
│       Image         │
├─────────────────────┤
│ filename:String     │
│ data:byte[]         │
├─────────────────────┤
│ paint()             │
└─────────────────────┘
```

Object design model after transformation:

```
              ┌─────────────────────┐
              │       Image         │
              ├─────────────────────┤        Proxy Pattern!
              │ filename:String     │
              ├─────────────────────┤
              │ paint()             │
              └─────────────────────┘
                        △
          ┌─────────────┴─────────────┐
┌──────────────────┐  image  ┌──────────────────┐
│   ImageProxy     │         │    RealImage     │
├──────────────────┤  1  0..1├──────────────────┤
│ filename:String  │         │ data:byte[]      │
├──────────────────┤         ├──────────────────┤
│ paint()          │         │ paint()          │
└──────────────────┘         └──────────────────┘
```

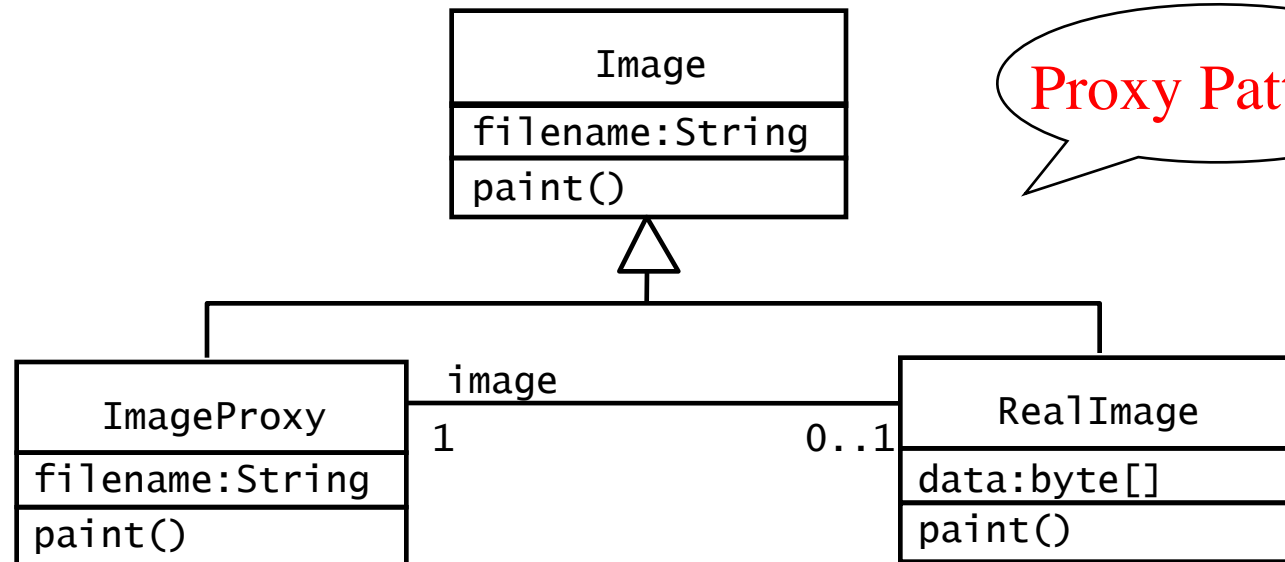# Examples of Model Transformations and Forward Engineering

- 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 simple 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:

```
┌─────────────────────┐ 1        1 ┌─────────────────────┐
│     Advertiser      │────────────│      Account        │
└─────────────────────┘            └─────────────────────┘
```
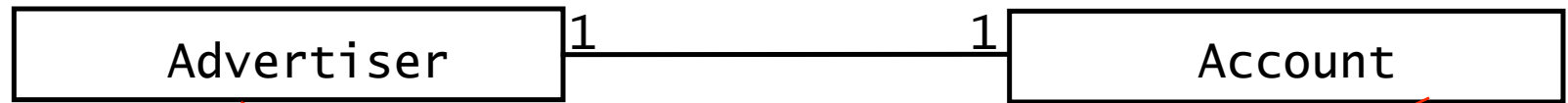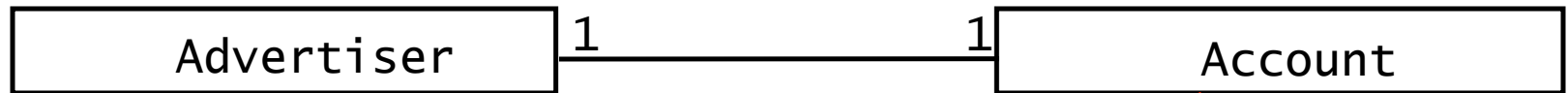
Source code after transformation:

```
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:

```
┌─────────────────┐ 1                1 ┌─────────────────┐
│   Advertiser    │────────────────────│     Account     │
└─────────────────┘                    └─────────────────┘
```

Source code after transformation:

```java
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;
  }
}
```

```java
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:

| Advertiser | 1 ————— * | Account |

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

```
                            *{ordered}    *
┌─────────────────┐                         ┌─────────────────┐
│   Tournament    │─────────────────────────│     Player      │
└─────────────────┘                         └─────────────────┘
```
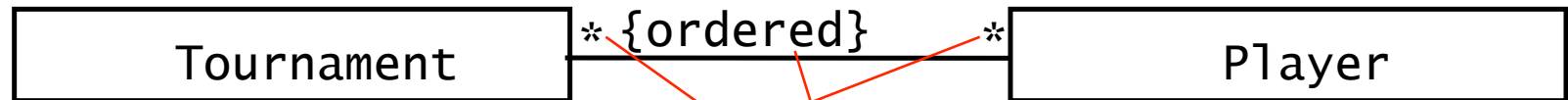
Source code after transformation

```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);
       }
   }
}
```

Bidirectional qualified association

Object design model before model transformation

League		*	*	Player

nickName
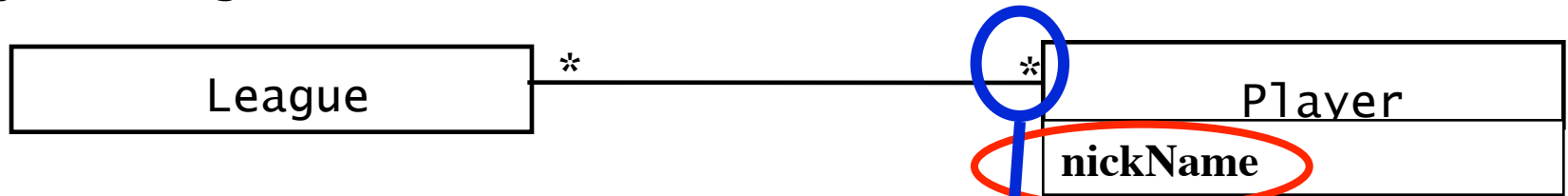
Object design model after model transformation

League		nickName	*	0..1	Player

Source code after forward engineering (see next slide)

Bidirectional qualified association cntd.

Object design model before forward engineering

League	nickName	* 0..1	Player
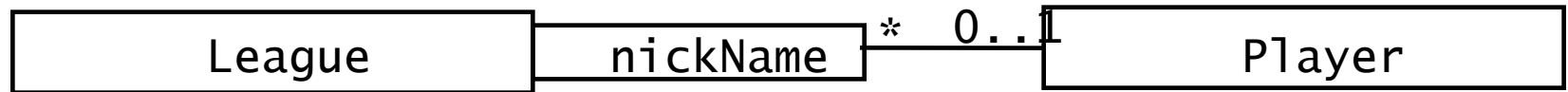
Source code after forward engineering

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

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

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

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

* *

Player

*

matches

*

Match

+start:Date
+status:MatchStatus

+playMove(p,m)
+getScore():Map

matches

*

*

sponsors

* *

Advertiser

Implementation in Java

```
TournamentForm                  1              1
─────────────────────────
+applyForTournament()        TournamentControl
                             ────────────────────────────
                *            +selectSponsors(advertisers):List
                             +advertizeTournament()
                             +acceptPlayer(p)
                             +announceTournament()
                             +isPlayerOverbooked():boolean
                                          1

                                          1
                              Tournament
                              ─────────────────────
                              -maNumPlayers:String
              *  * players    +start:Date              sponsors *  *
              ─────────       +end:Date                ───────────────
              Player          ─────────────────────    Advertiser
                              +acceptPlayer(p)
                              +removePlayer(p)
                    *         +isPlayerAccepted(p)
              matches*
              ─────────────────        matches
              Match                                  *
              ─────────────────
              +start:Date
              +status:MatchStatus
              ─────────────────
              +playMove(p,m)
              +getScore():Map
```

```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());
        }
      }
    }
}
```
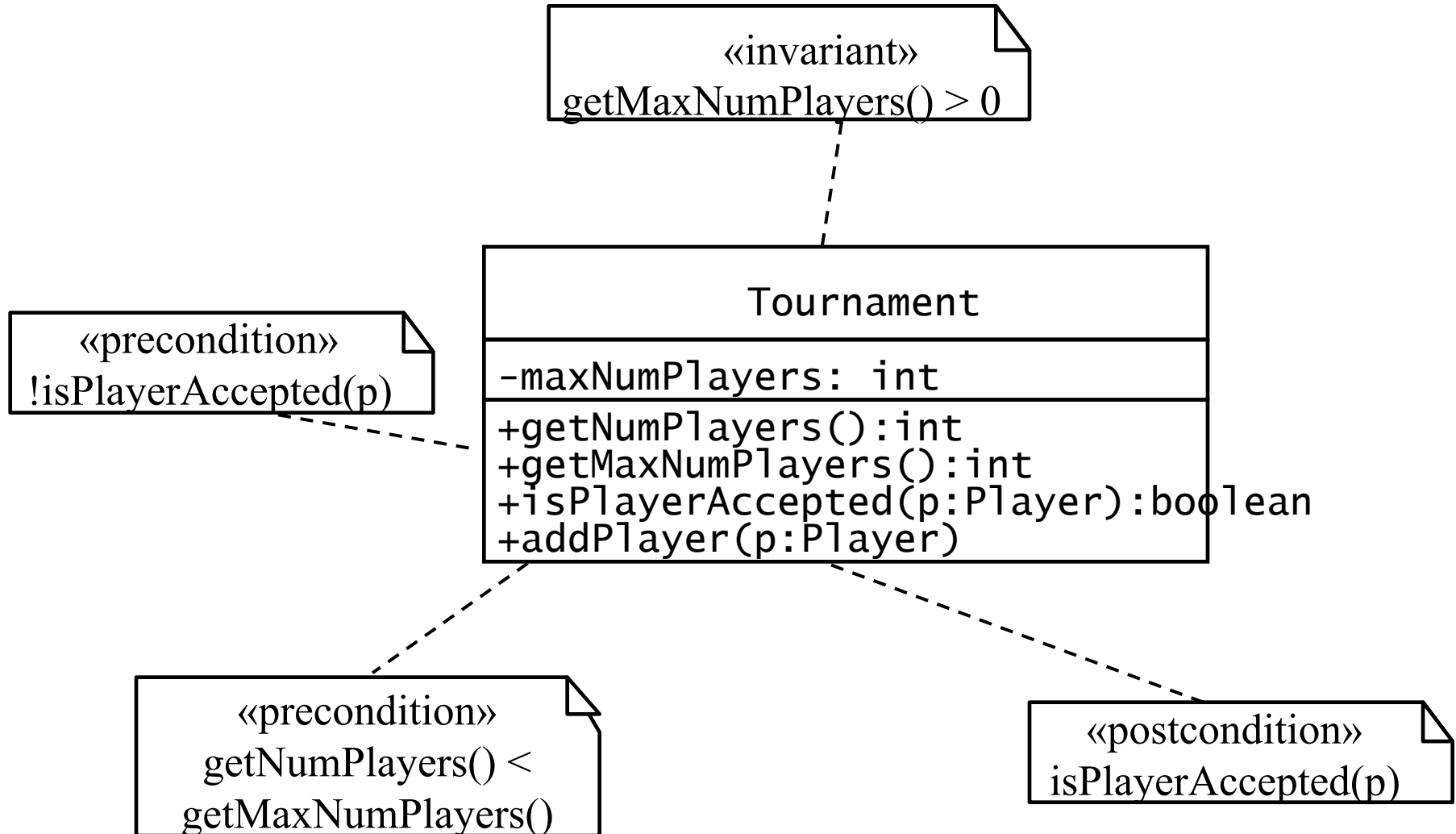
The try-throw-catch Mechanism in Java

```java
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(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());
            }
        }
    }
}
```

TournamentForm

+applyForTournament()

TournamentControl

+selectSponsors(advertisers):List
+advertizeTournament()
+acceptPlayer(p)
+announceTournament()
+isPlayerOverbooked():boolean

1   1

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
*

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

«invariant»
getMaxNumPlayers() > 0

«precondition»
!isPlayerAccepted(p)

Tournament

-maxNumPlayers: int

```
+getNumPlayers():int
+getMaxNumPlayers():int
+isPlayerAccepted(p:Player):boolean
+addPlayer(p:Player)
```

«precondition»
getNumPlayers() <
getMaxNumPlayers()

«postcondition»
isPlayerAccepted(p)

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)
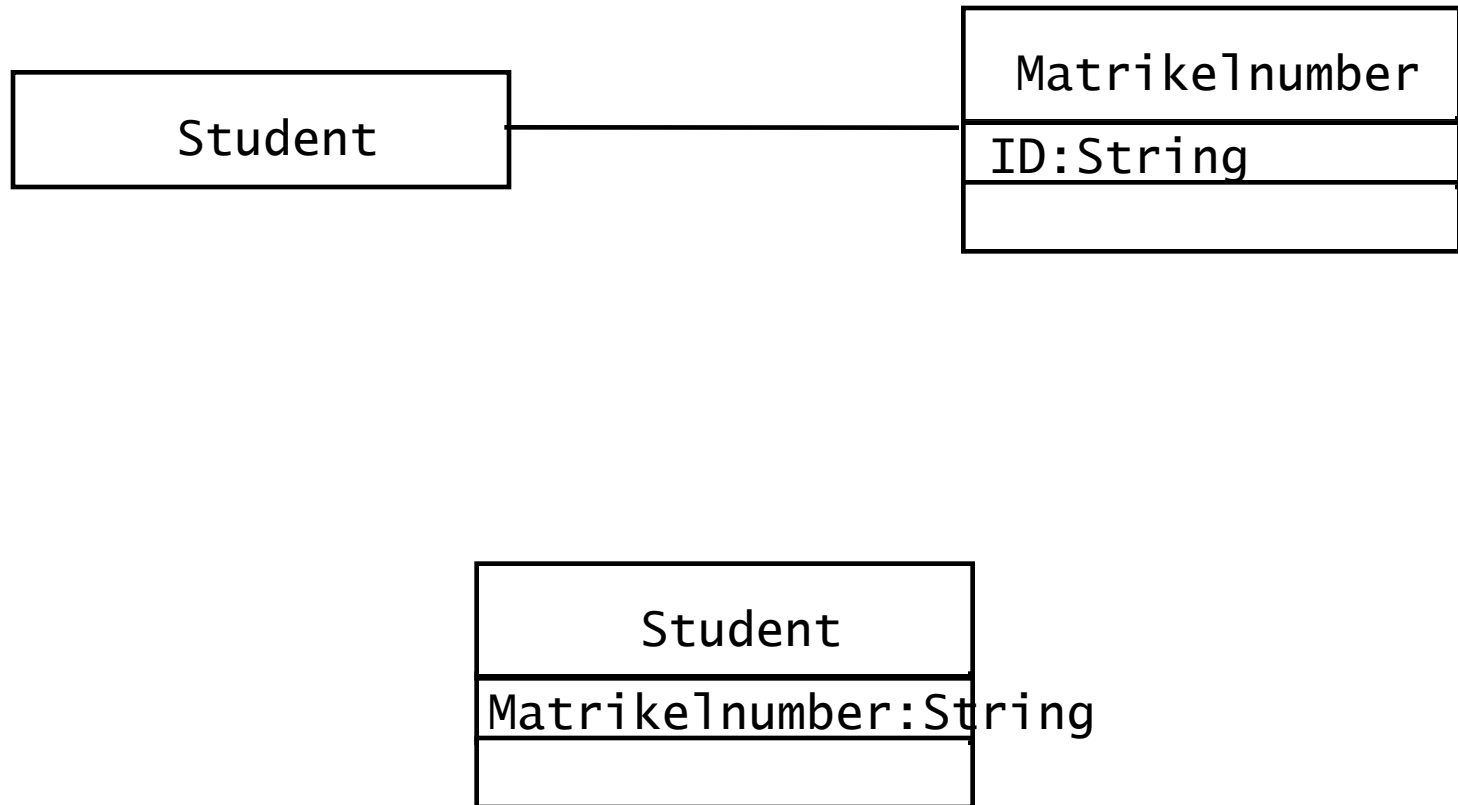
2. Rearrange execution order
   - Eliminate dead paths as early as possible (Use knowledge of distributions, frequency of path traversals)
   - Narrow search as soon as possible
   - Check if execution order of loop should be reversed

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

```
┌─────────────────────────┐         ┌─────────────────────────┐
│                         │         │      Matrikelnumber     │
│        Student          │─────────├─────────────────────────┤
│                         │         │      ID:String          │
│                         │         ├─────────────────────────┤
└─────────────────────────┘         │                         │
                                     └─────────────────────────┘
```

```
┌─────────────────────────┐
│        Student          │
├─────────────────────────┤
│  Matrikelnumber:String  │
├─────────────────────────┤
│                         │
└─────────────────────────┘
```

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