Chapter 12

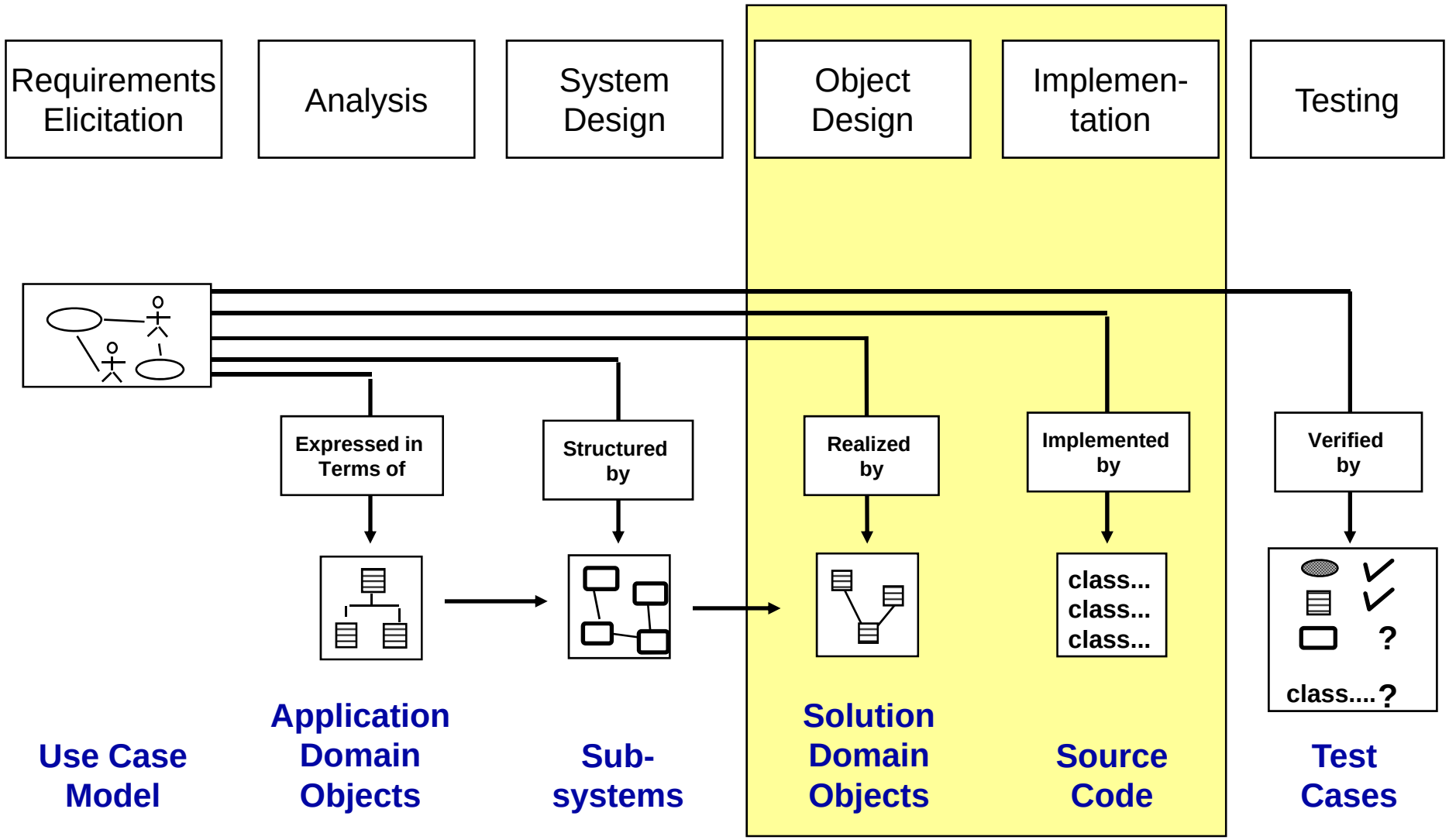
Object Design
Mapping Models to Code

Object-**O**riented
Software **C**onstruction

Armin B. Cremers, Tobias Rho,
Daniel Speicher & Holger Mügge
(based on Bruegge & Dutoit)

Software Lifecycle Activities

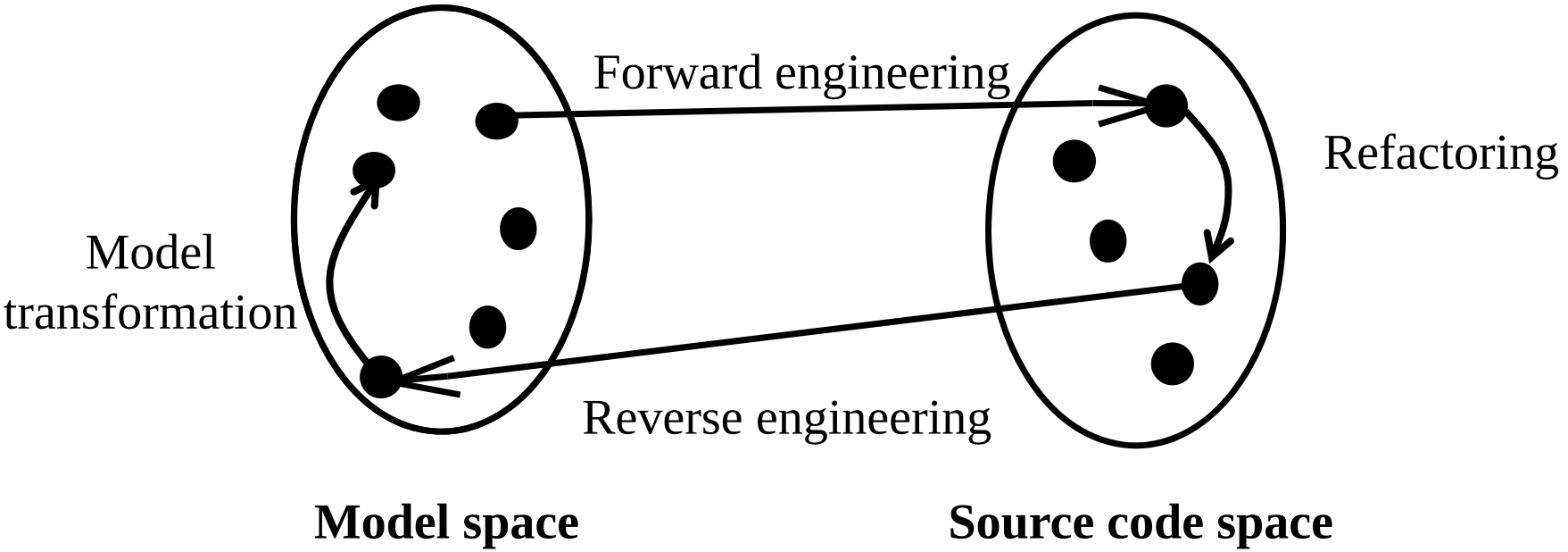
...and their models



Lecture Overview

- Model and Program Transformations
- Operations on the object model
  - Optimizations to address performance requirements
- Implementation of class model components
  - Realization of associations
  - Realization of operation contracts
  - Refactorings
- Realizing entity objects based on selected storage strategy
  - Mapping the class model to a storage schema

Model / Program Transformations



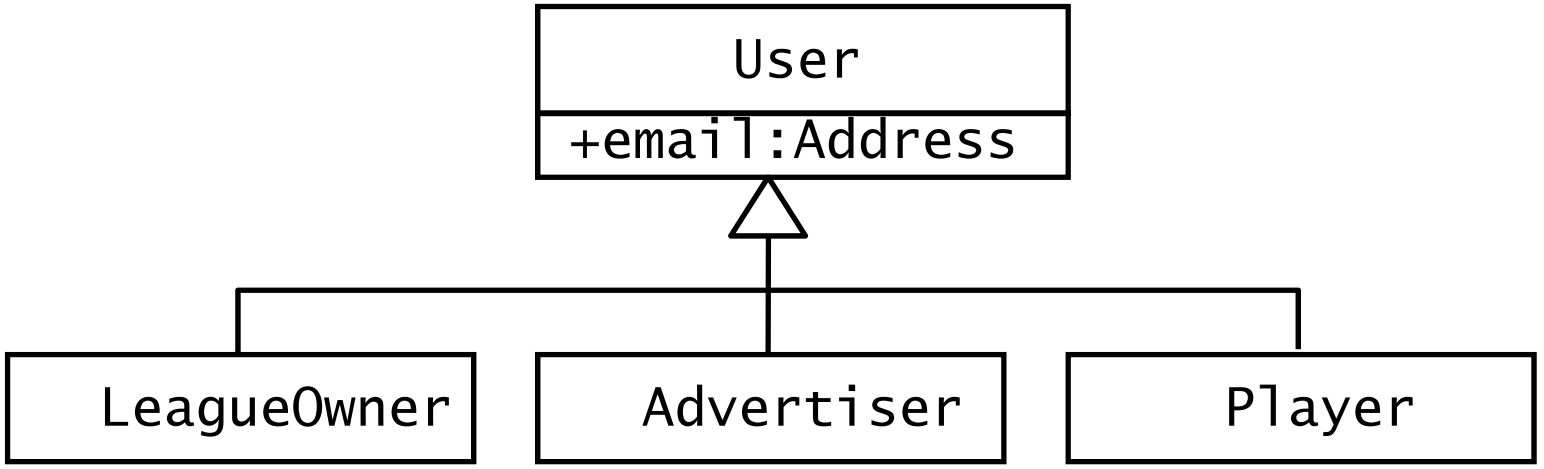
More Terminology

- Roundtrip Engineering
  - Forward Engineering + Reverse Engineering
  - Among other vendors Together, Omondo and IBM Rational provide tools for reverse engineering
- Reengineering
  - Used in the context of project management
  - Providing new functionality (customer dreams up new stuff) in the context of new technology (technology enablers)
  - Extract requirements / design from legacy code without available design documents

Model Transformation Example

Object design model before transformation

Object design model after transformation:



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 {
  protected 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 {
   protected 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 {
protected String email;

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

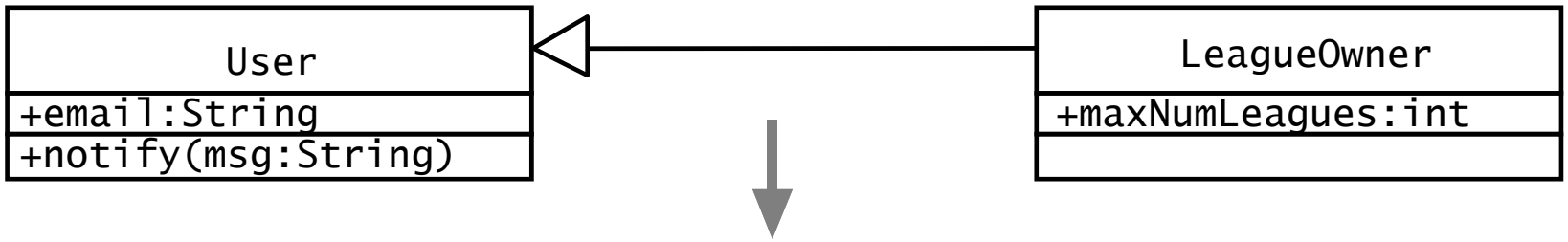
```
public class Player extends User {
   public Player(String email) {
         super(email);
   }
}
```

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

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

Forward Engineering Example

Object design model



Generated Source code

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

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

Mapping Activities

- Optimizing the Object Design Model
- Mapping Associations
- Mapping Contracts to Exceptions
- Mapping Object Models to Tables

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. Store derived attributes to save computation time
  3. Rearrange computations for greater efficiency
- As an object designer you must strike a balance between efficiency and clarity.
  - Optimizations will make your models more obscure

Increase Inheritance

- Rearrange and adjust classes and operations to prepare for inheritance
- Abstract common behavior out of groups of classes
  - If a set of operations or attributes are repeated in 2 classes the classes might be special instances of a more general class.
- Reuse classes by implementation inheritance where possible
  - E.g. Eiffel's non-conforming inheritance

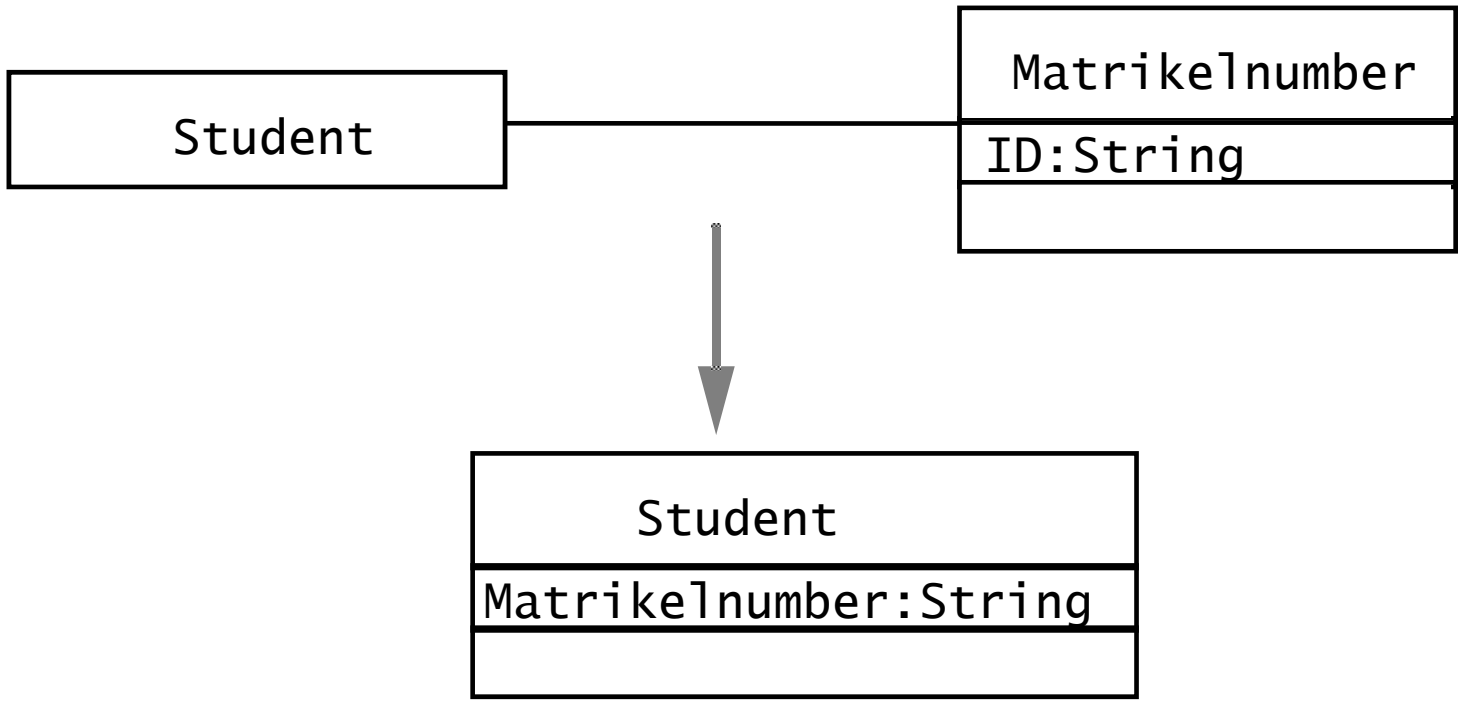
Design Optimization Activities

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
  - “Every student receives a matrikel number at the first day in the university. “

Optimization Activities: Collapsing Objects



- Collapse a class into an attribute if the only operations (at least the most) defined on the attributes are setters and getters.

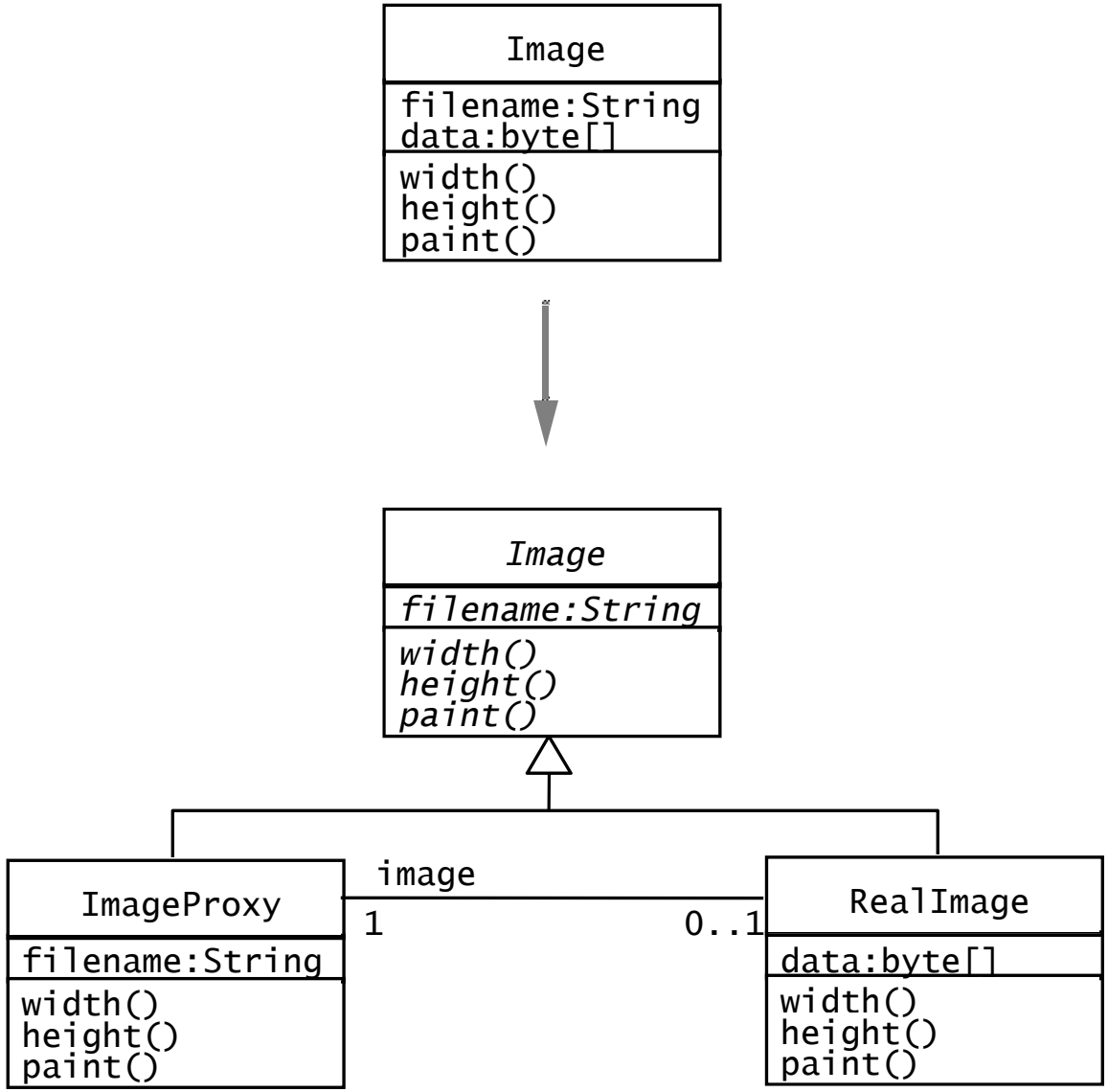
Design Optimizations (continued)

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 / on demand when dirty (pull)
    - Active value: An attribute can designate set of dependent values which are automatically updated when active value is changed (notification, data trigger)

Optimization Activities:
Delaying Complex Computations



Mapping Activities

- ✓ Optimizing the Object Design Model
- ♦ Mapping Associations
- ♦ Mapping Contracts to Exceptions
- ♦ Mapping Object Models to Tables

Mapping Activities
Handle Language Limitations

- Programming languages do not support the concept of association:
  - transform the associations of the object model into collections of object references
- If the programming language does not support contracts:
  - write code for detecting and handling contract violations
- These activities are intellectually not challenging
  - However, they have a repetitive and mechanical flavor that makes them error prone

State of the Art of Model-based Software Engineering

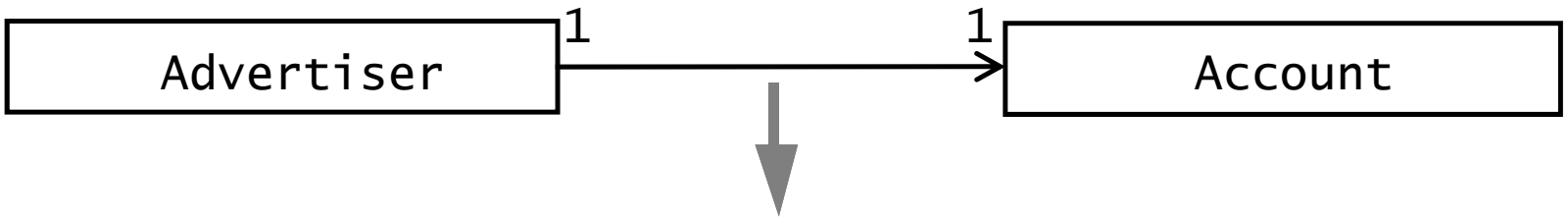
- The Vision
  - During object design we would like to implement a system that realizes the use cases specified during requirements elicitation and system design.
- The Reality
  - Different developers usually handle contract violations differently.
  - Undocumented parameters are often added to the API to address a requirement change.
  - Additional attributes are usually added to the object model, but are not handled by the persistent data management system, possibly because of a miscommunication.
  - Many improvised code changes and workarounds that eventually yield to the degradation of the system.

Realizing Associations

- Strategy for implementing associations
  - Be as uniform as possible
  - Individual decision for each association
- Example of uniform implementation (often used by CASE tools)
  - 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 Hashtable

Realization of a unidirectional, one-to-one association

Object design model

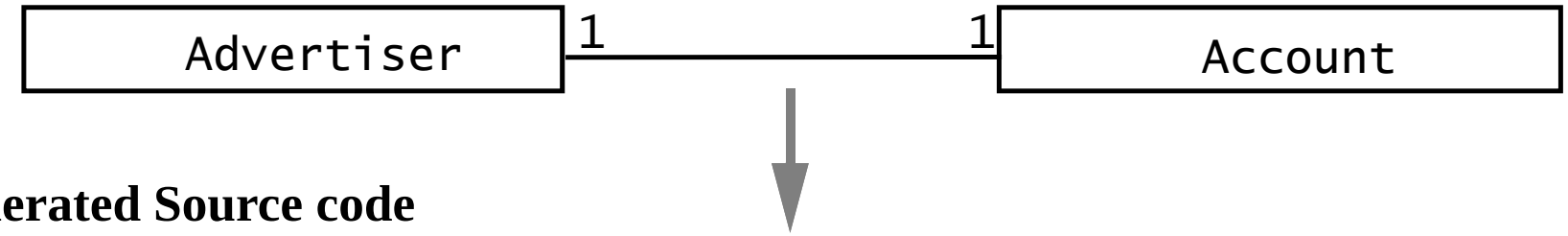


Generated Source code

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

Bidirectional one-to-one association

Object design model



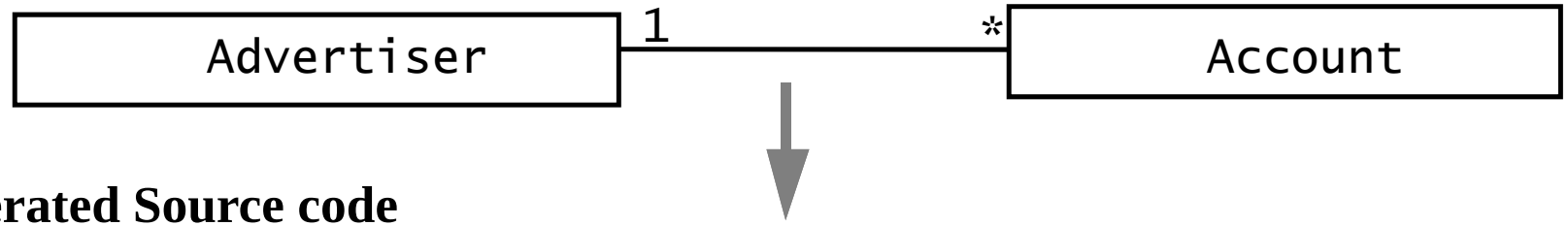
Generated Source code

```
public class Advertiser {
   /* The account field 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 {
   /* The owner field is
   initialized
    * during 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



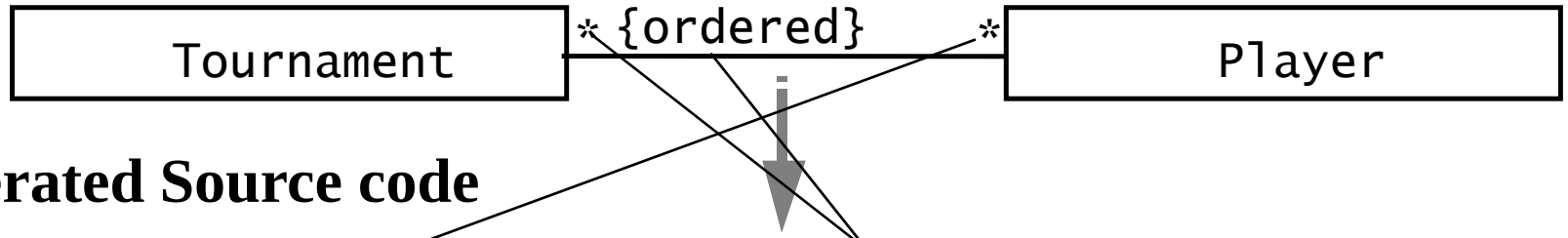
Generated Source code

```
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



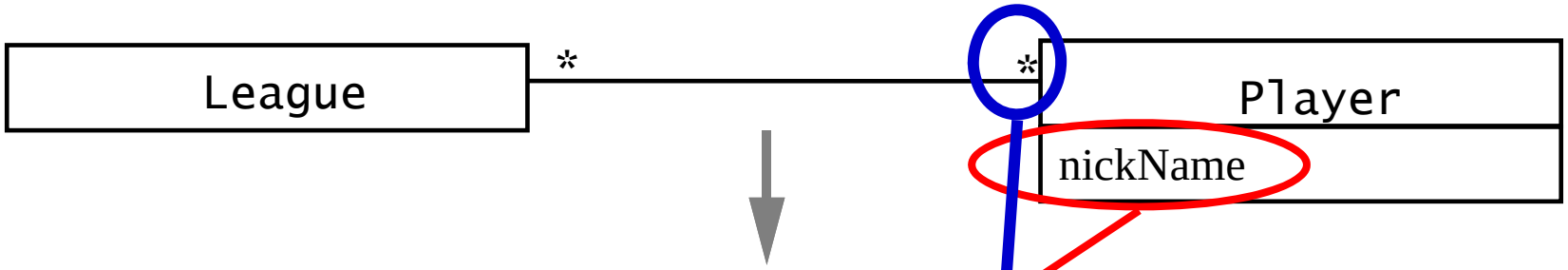
Generated Source code

```
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);
        }
   }
}
```

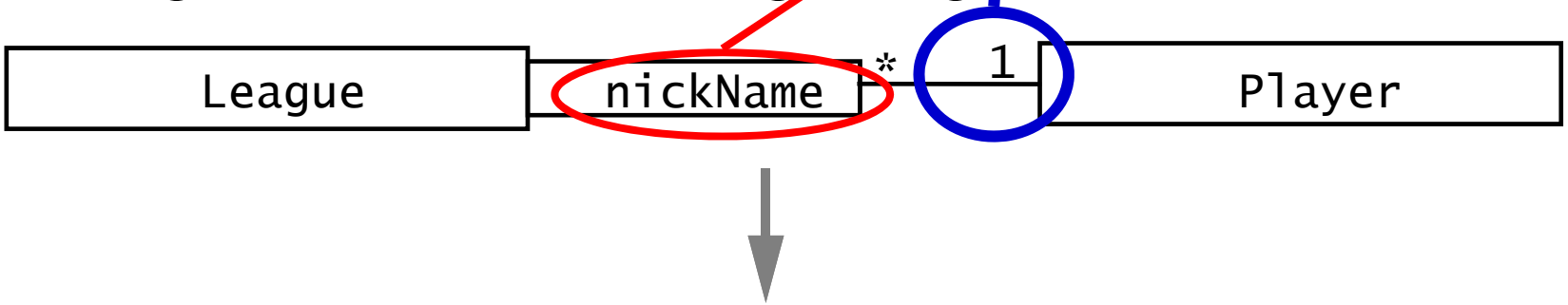
```
public class Player {
 private List tournaments;
 public Player() {
        tournaments = new ArrayList();
 }
 public void addTournament(Tournament t) {
        if (!tournaments.contains(t)) {
         tournaments.add(t);
         t.addPlayer(this);
        }
 }
}
```

Bidirectional qualified association

Object design model before transformtion



Object design model before forward engineering



Source code after forward engineering

Bidirectional qualified association (continued)

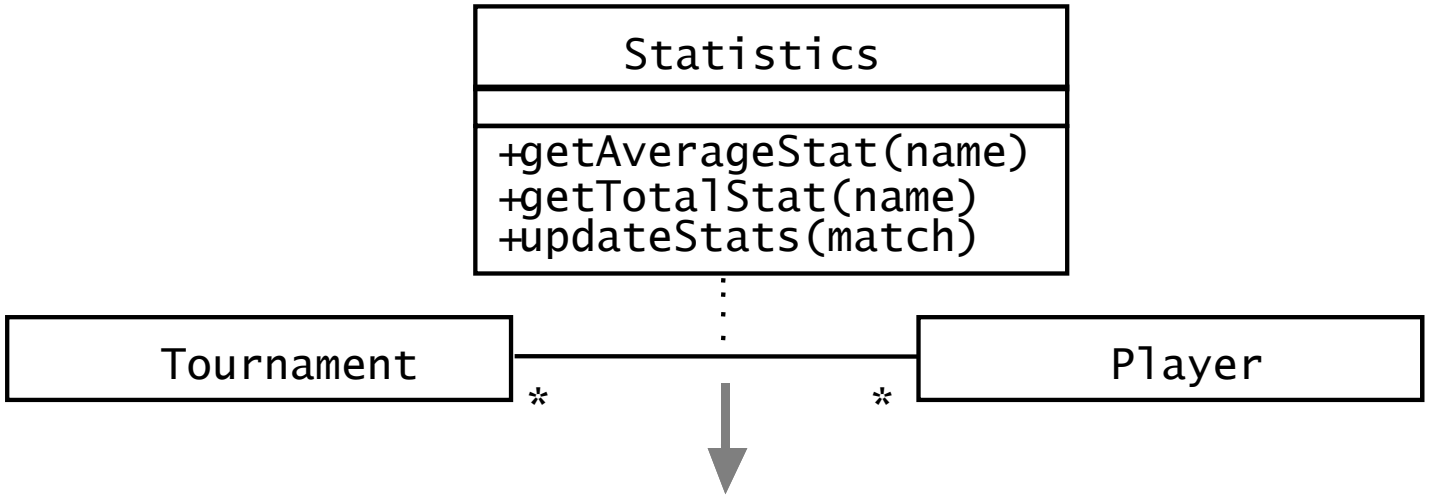
Source code after forward engineering

```
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);
   }
}
```

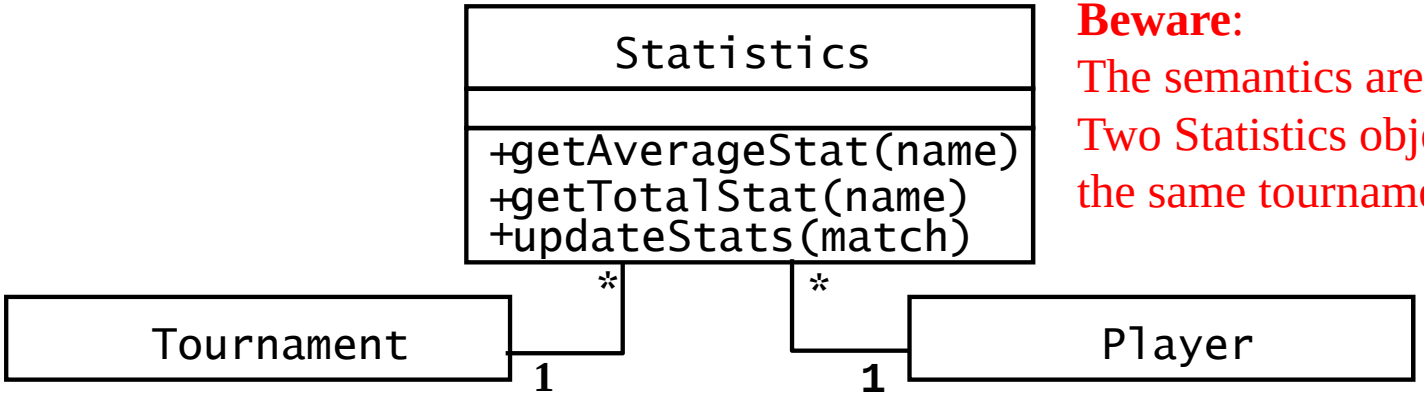
```
public class Player {
   private Map leagues;

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

Transformation of an association class

Object design model before transformation



Object design model after transformation: 1 class and two binary associations



Beware:
The semantics are not equivalent.
Two Statistics objects may reference the same tournament and player!

Mapping Activities

- ✓ Optimizing the Object Design Model
- ✓ Mapping Associations
- ➢ Mapping Contracts to Exceptions
- ♦ Mapping Object Models to Tables

Exceptions as building blocks for notifying contract violations

- Many object-oriented languages, including Java (which only provides assertions) do not include built-in support for contracts.
- However, we can use their exception mechanisms as building blocks for signaling and handling contract violations
- In Java we use the try-throw-catch mechanism, optionally assertions
- Example:
  - Let us assume the addPlayer() operation of TournamentControl is invoked with a player who is already part of the Tournament.
  - In this case addPlayer() should throw an exception of type KnownPlayer.
  - See source code on next slide

The try-throw-catch Mechanism in Java

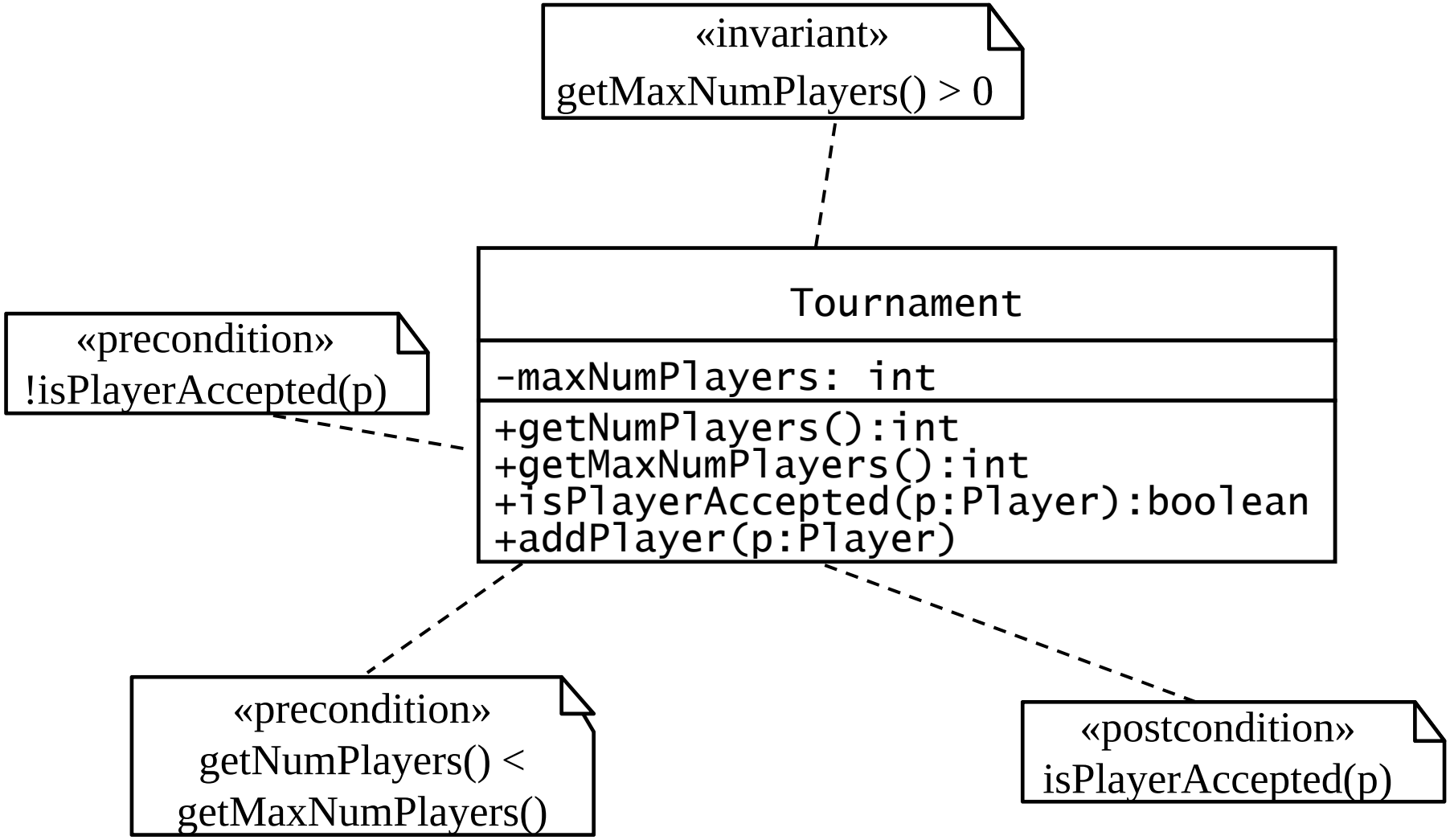

```
public class TournamentControl {
   private Tournament tournament;
   public void addPlayer(Player p) throws KnownPlayerException {
         if (tournament.isPlayerAccepted(p)) {
               throw new KnownPlayerException(p);
         }
         //... Normal addPlayer behavior
   }
}
public class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

   public void processPlayerApplications() { // Go through all the players
               for (Iteration i = players.iterator(); i.hasNext();) {
               try {          // Delegate to the control object.
                     control.addPlayer((Player)i.next());

               } catch (KnownPlayerException e) {
                     // If an exception was caught, log it to the console
                     ErrorConsole.log(e.getMessage());
               }
         }
   }
}
```

Implementing a contract

For each operation in the contract, do the following

- **Check precondition**: Check the precondition before the beginning of the method with a test that raises an exception if the precondition is false.
- **Check postcondition:** Check the postcondition at the end of the method and raise an exception if the contract is violoated. If more than one postcondition is not satisfied, raise an exception only for the first violation.
- **Check invariant:** Check invariants at the same time as postconditions.
- **Deal with inheritance:** Encapsulate the checking code for preconditions and postconditions into separate methods that can be called from subclasses.

Implementation of the Tournament.addPlayer() contract



Implementation of the Tournament.addPlayer() contract


```
public void addPlayer(Player p) throws KnownPlayerException {
        boolean exceptionThrown = false;
        try {
                // pre conditions
      if (tournament.isPlayerAccepted(p)) {
              throw new KnownPlayerException(p);
      }
      //... Normal addPlayer behavior

        } catch(Throwable t) {
       exceptionThrown = true;
               throw t;
        } finally {
              if(!exceptionThrown) {
                // post conditions
                if(isPlayerAccepted(p)) {
           throw new TournamentException(…);
                }
                // invariants
                if(!(getMaxNumPlayers() > 0))
                      throw new TournamentException("max players
                                        must be greater than 0");
            }
        }

}
```

Heuristics for Mapping Contracts to Exceptions

Be pragmatic, if you don't have enough time.

- Omit checking code for post conditions and invariants.
  - Usually redundant with the code accomplishing the functionality of the class
  - Not likely to detect many bugs unless written by a separate tester.
- Omit the checking code for private and protected methods.
- Focus on components with the longest life
  - Focus on Entity objects, not on boundary objects associated with the user interface.
- Reuse constraint checking code.
  - Many operations have similar preconditions.
  - Encapsulate constraint checking code into methods so that they can share the same exception classes.

(Native) Language for Design by Contract

- E.g. D and Eiffel provide native support
- For Java third-party support exists
  - Java Modeling Language
    - Annotations define invariants, pre- and postcondition
    - A range of verification tools exist
      - Runtime checkers and extended static checkers
  - Contract4J
    - Contracts are defined with Java 5 Annotations
    - Runtime verification

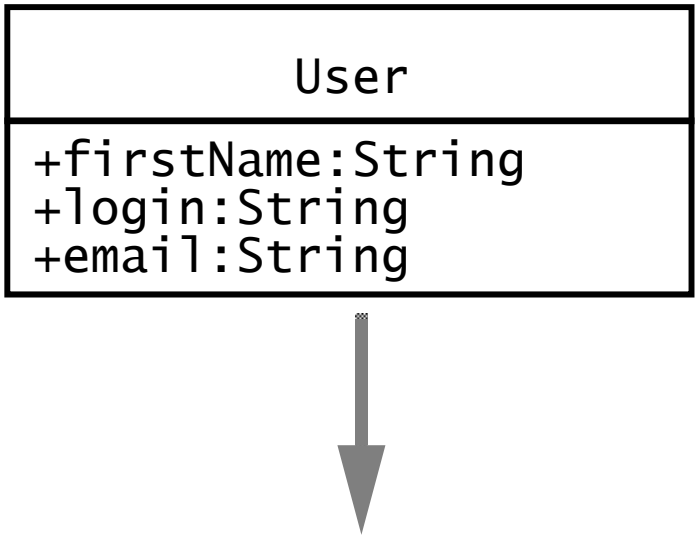
Mapping Activities

- ✓ Optimizing the Object Design Model
- ✓ Mapping Associations
- ✓ Mapping Contracts to Exceptions
- ➢ Mapping Object Models to Tables

Mapping an object model to a relational 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.
- UML mappings
  - 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 the User class to a database table



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”	“am384”
“chessNovice”	“js289”

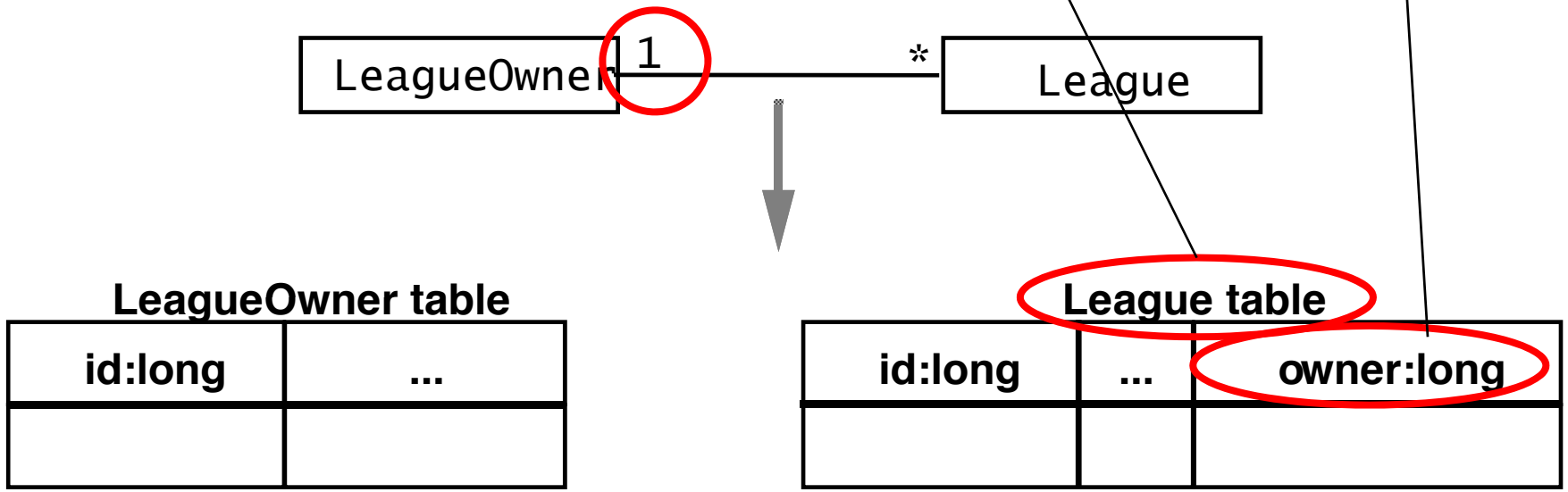
Foreign key referencing **User table**

Buried Association

- Associations with *multiplicity one* can be implemented using a foreign key.
- For *one-to-many* associations we add a 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.

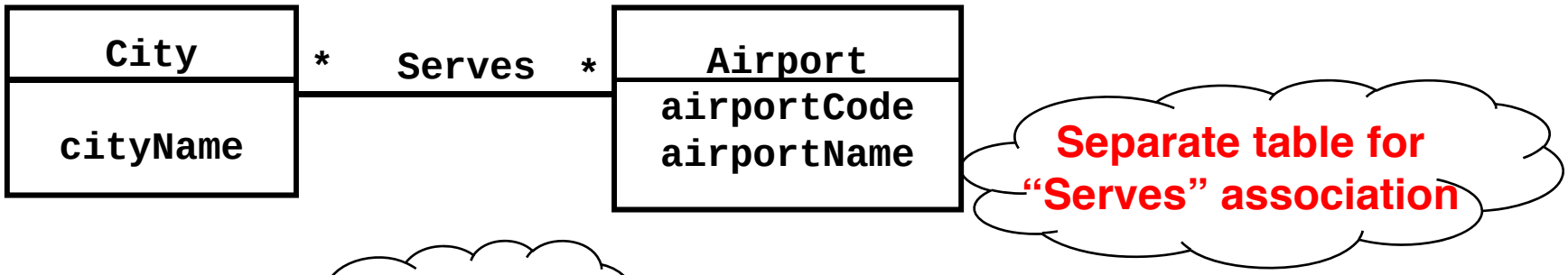
Buried Association

- Associations with multiplicity *one* can be implemented using a foreign key. Because the association vanishes in the table, we call this a buried association.
- 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.



Mapping Many-To-Many Associations

In this case we need a separate table for the association



Primary Key

City Table

cityName
Houston
Albany
Munich
Hamburg

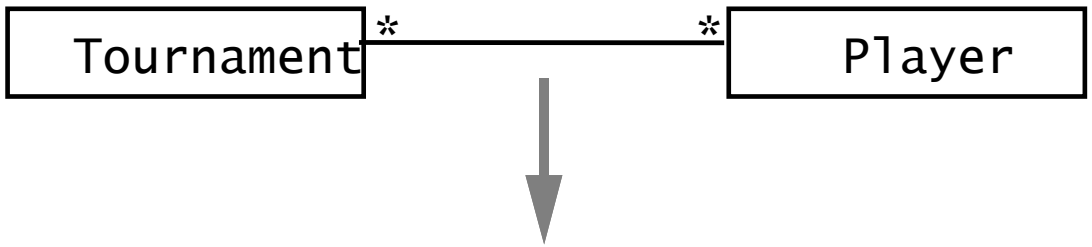
Airport Table

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

Mapping the Tournament/Player association as a separate table



Tournament table

id	name	...
23	novice	
24	expert	

TournamentPlayerAssociation table

tournament	player
23	56
23	79

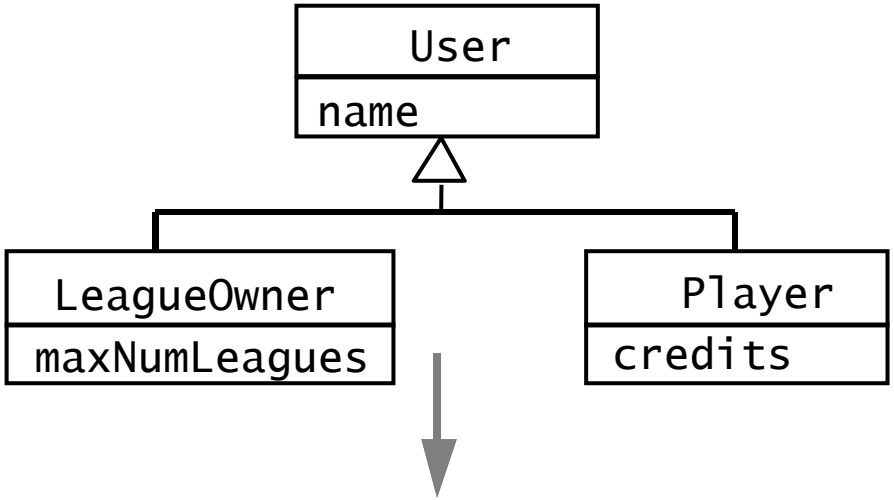
Player table

id	name	...
56	alice	
79	john	

Realizing Inheritance

- Relational databases do not support inheritance
- Two possibilities to map UML inheritance relationships 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



User table

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

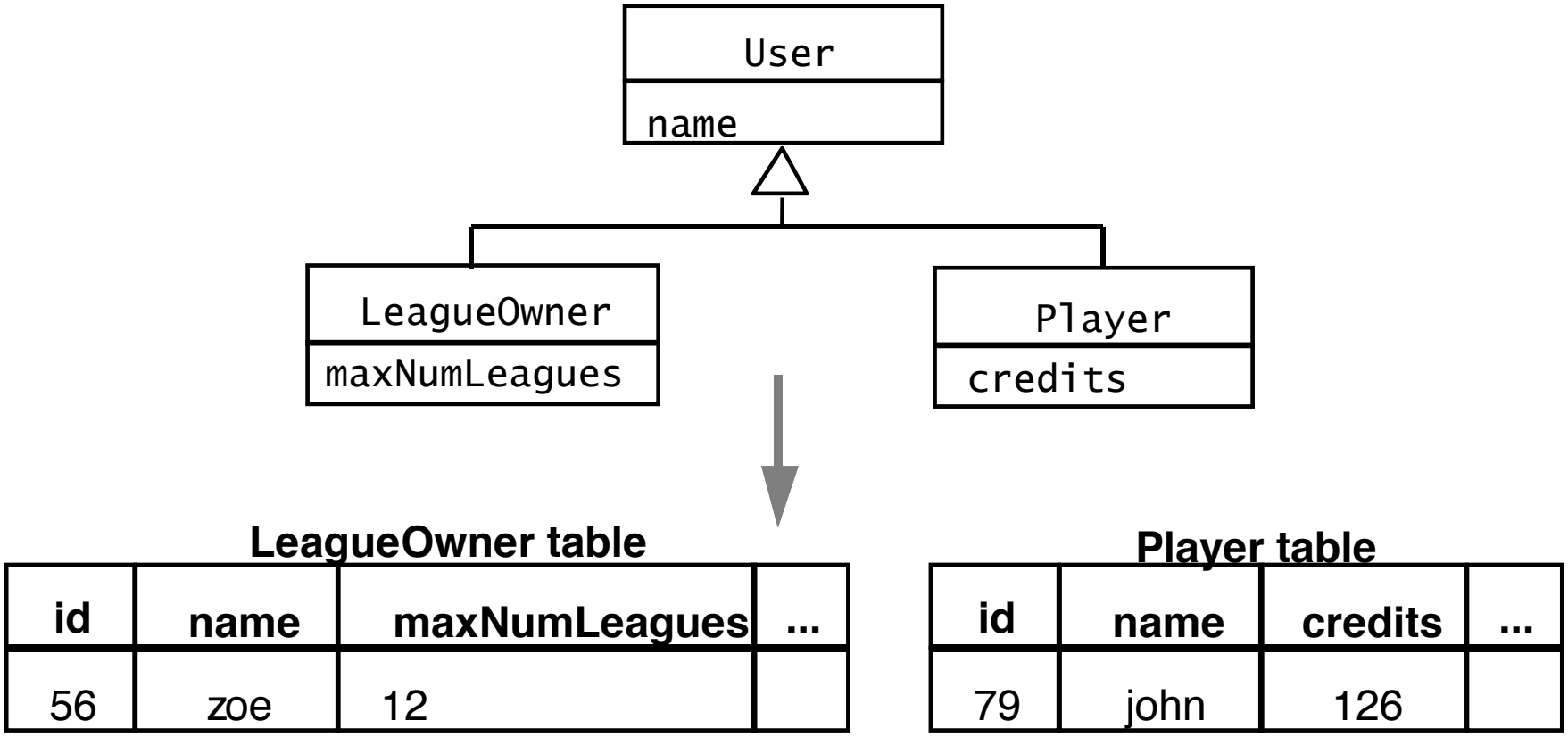
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Realizing inheritance by duplicating columns



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
  - ☺ 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
  - ☹ Modifying the database schema is more complex and error-prone
  - ☺ Individual objects are not fragmented across a number of tables, resulting in faster queries
- Information hiding

Heuristics for Transformations

- For a given transformation use the same tool
  - If you are using a CASE tool to map associations to code, use the tool to change association multiplicities.
- Keep the contracts in the source code, not only in the object design model
  - By keeping the specification as a source code comment, they are more likely to be updated when the source code changes.
- Use the same names for the same objects
  - If the name is changed in the model, change the name in the code and or in the database schema (generalized refactoring)
  - Provides traceability among the models
- Have a style guide for transformations
  - By making transformations explicit in a manual, all developers can apply the transformation in the same way.

Summary

- Undisciplined changes => degradation of the system model
- Four mapping concepts were introduced
  - 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 the readability or modifiability of the code
  - Reverse engineering attempts to discover the design from the code.
- We reviewed model transformation and forward engineering techniques:
  - Optimizing the class model
  - Mapping associations to collections
  - Mapping contracts to exceptions
  - Mapping class model to storage schemas