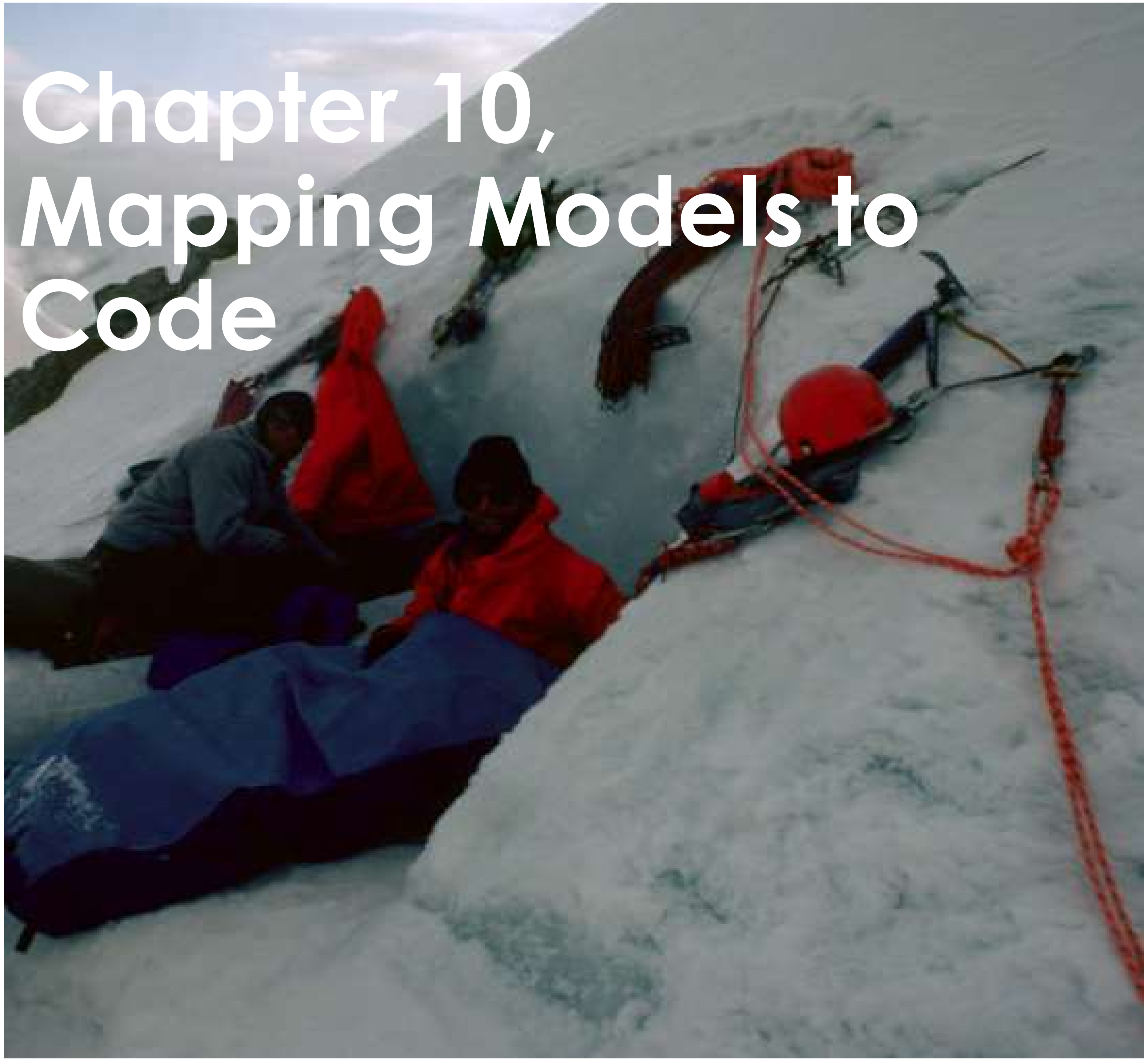
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 10, Mapping Models to Code



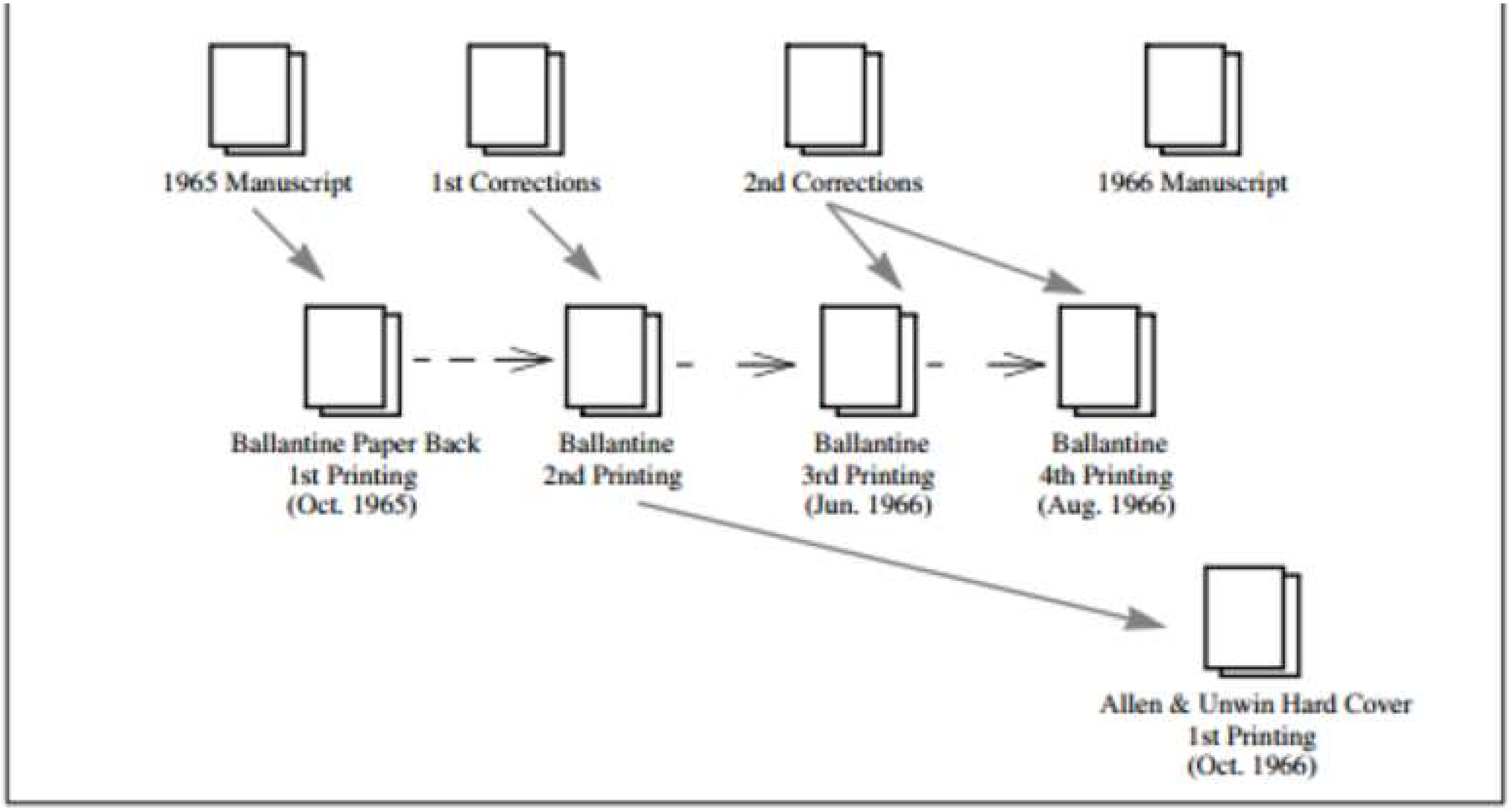
10 . Mapping Models to Code

- a selection of transformations to illustrate a disciplined approach to implementation to avoid such a system degradation
  - optimizing the class model
  - mapping associations to collections
  - mapping operation contracts to exceptions
  - mapping the class model to a storage schema.

- use Java and Java-based technologies in this chapter

10.1 Introduction: A Book Example

- Although many steps of the publishing process are now automated, mistakes still creep in,introduced by the author, the publisher, or the printer, resulting in a series of corrections that can introduce new errors.
- Working on an object design model similarly involves many transformations that are error prone


1965 Manuscript
1st Corrections
2nd Corrections
1966 Manuscript
Ballantine Paper Back
1st Printing
(Oct. 1965)
Ballantine
2nd Printing
Ballantine
3rd Printing
(Jun. 1966)
Ballantine
4th Printing
(Aug. 1966)
Allen & Unwin Hard Cover
1st Printing
(Oct. 1966)

10.2 An Overview of Mapping

- A transformation aims at improving one aspect of the model (e.g., its modularity) while preserving all of its other properties
-
- We focus in detail on the following activities:
  - Optimization (Section 10.4.1)
    - addresses the performance requirements of the system model. Ex) reducing the multiplicities of associations to speed up queries, adding redundant associations for efficiency, and adding derived attributes to improve the access time to objects.
  - Realizing associations (Section 10.4.2)
    - map associations to source code constructs, such as references and collections of references.

- Mapping contracts to exceptions (Section 10.4.3)
  - describe the behavior of operations when contracts are broken
  - includes raising exceptions when violations are detected and handling exceptions in higher level layers of the system.

- Mapping class models to a storage schema (Section 10.4.4)
  - map the class model to a storage schema, such as a relational database schema.

10.3 Mapping Concepts

- four types of transformations (Figure 10-1):
  - Model transformations operate on object models (Section 10.3.1).
  - Refactorings are transformations that operate on source code (Section 10.3.2)
  - Forward engineering produces a source code template that corresponds to an object model (Section 10.3.3).
  - Reverse engineering produces a model that corresponds to source code(Section 10.3.4).

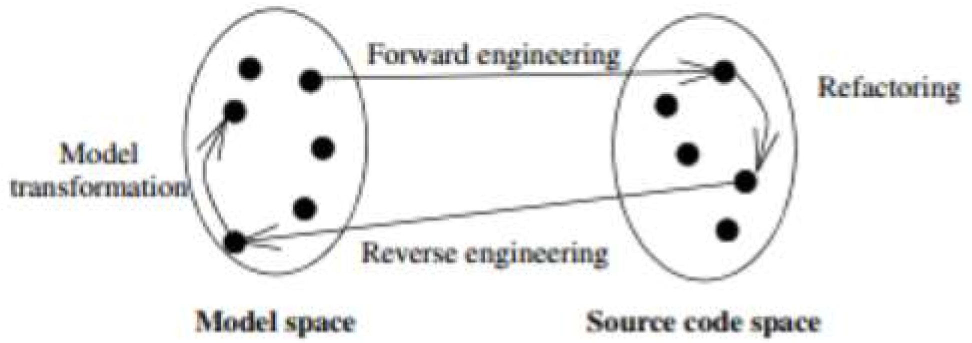


Figure 10-1 The four types of transformations described in this chapter: model transformations, refactorings, forward engineering, and reverse engineering.

10.3.1 Model Transformation

- A model transformation
  - applied to an object model and results in another object model
- The purpose of object model transformation
  - to simplify or optimize the original model, bringing it into closer compliance with all requirements in the specification
- A transformation
  - may add, remove, or rename classes, operations, associations, or attributes.
  - add information to the model or remove information from it.

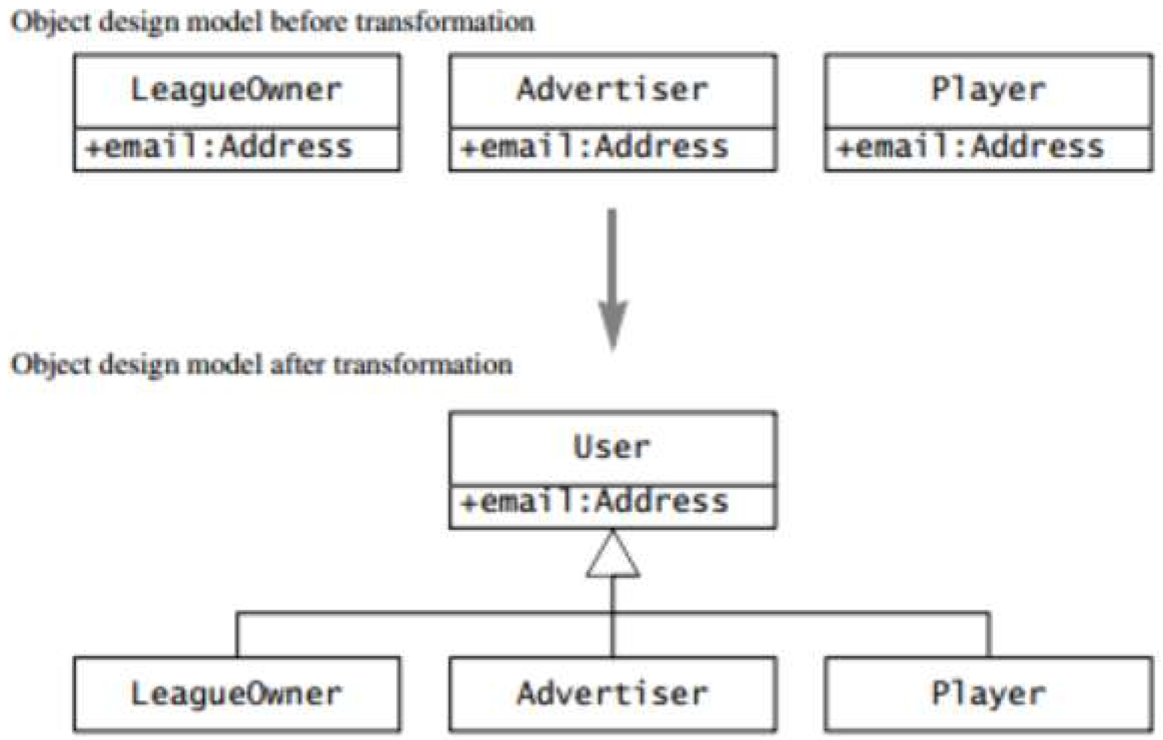
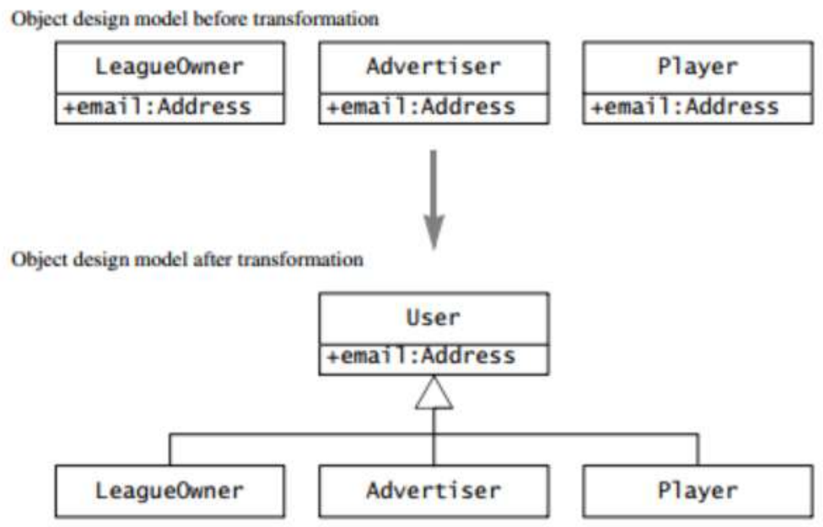


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

10.3.2 Refactoring

- A refactoring
  - a transformation of the source code that improves its readability or modifiability without changing the behavior of the system
  - aims at improving the design of a working system by focusing on a specific field or method of a class.
- To ensure that the refactoring does not change the behavior of the system, the refactoring is done in small incremental steps that are interleaved with tests.

Refactoring(#1)



Before refactoring

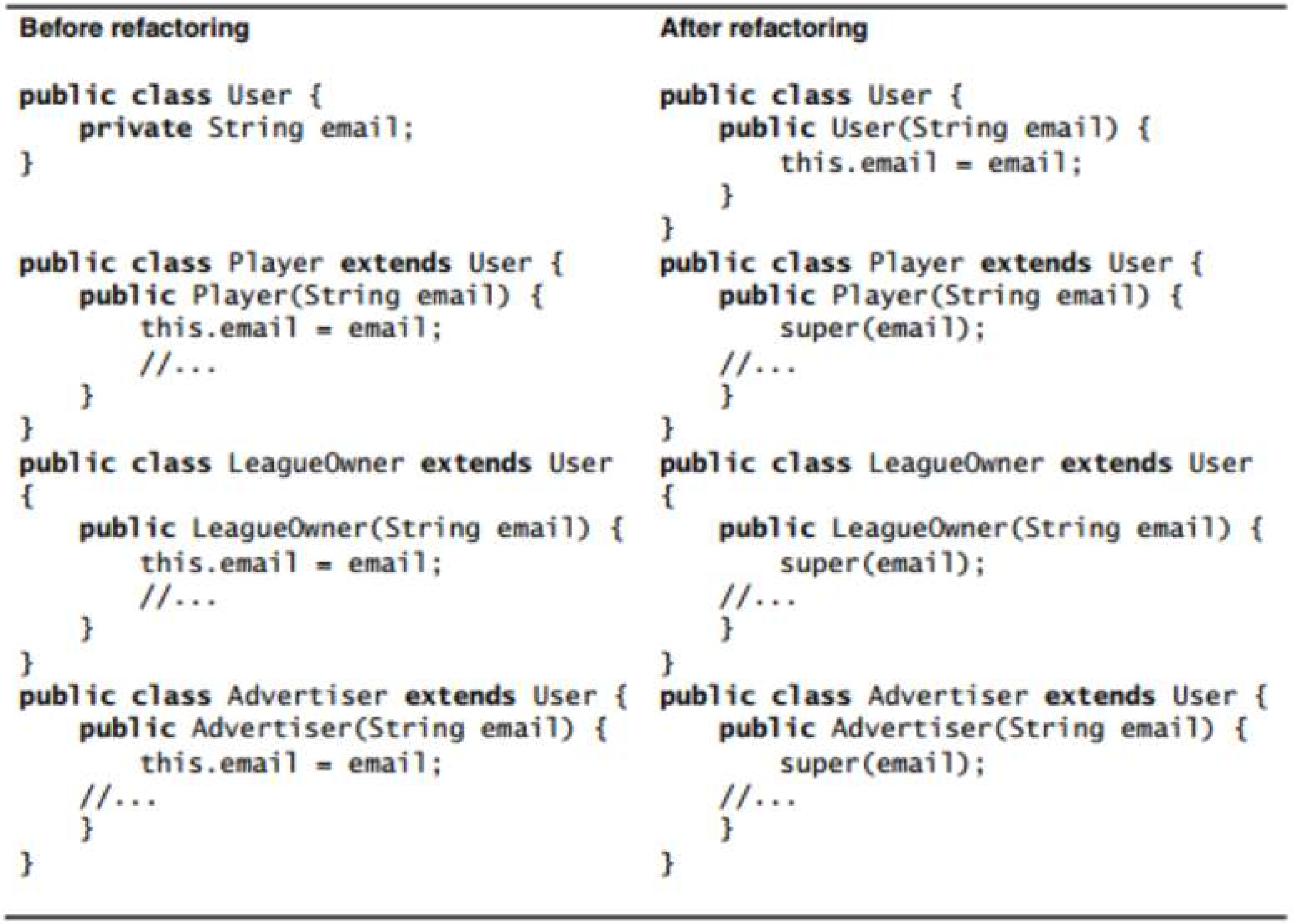
```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

After refactoring

```
public class User {
    protected String email;
}
public class Player extends User {
    //...
}
public class LeagueOwner extends User
{
    //...
}
public class Advertiser extends User {
    //...
}
```

Figure 10-3 Applying the *Pull Up Field* refactoring.

Refactoring(#2)



Before refactoring

```
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;
    //...
    }
}
```

After refactoring

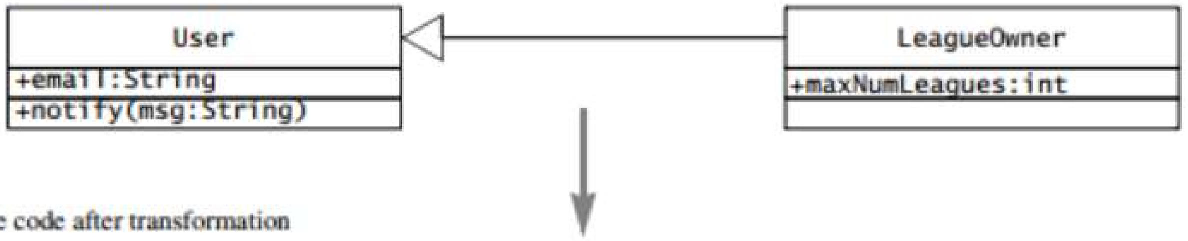
```
public class User {
    public User(String email) {
        this.email = email;
    }
}
public class Player extends User {
    public Player(String email) {
        super(email);
    //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        super(email);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

Figure 10-4 Applying the *Pull Up Constructor Body* refactoring.

10.3.3 Forward Engineering

- Forward engineering
  - applied to a set of model elements and results in a set of corresponding source code statements, such as a class declaration, a Java expression, or a database schema.
- The purpose of forward engineering
  - to maintain a strong correspondence between the object design model and the code, and to reduce the number of errors introduced during implementation, thereby decreasing implementation effort.
- Figure 10-5 Realization of the User and LeagueOwner classes

Object design model before transformation



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) {
        // ....
    }
    /* 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 */
}
```

Figure 10-5 Realization of the `User` and `LeagueOwner` classes (UML class diagram and Java excerpts). In this transformation, the public visibility of `email` and `maxNumLeagues` denotes that the methods for getting and setting their values are public. The actual fields representing these attributes are private.

10.3.4 Reverse Engineering

- Reverse engineering
  - applied to a set of source code elements and results in a set of model elements.
- The purpose of this type of transformation
  - to recreate the model for an existing system, either because the model was lost or never created, or because it became out of sync with the source code

10.3.5 Transformation Principles

- To avoid introducing new errors, all transformations should follow these principles:
  - *1) Each transformation must address a single criteria.*
    - A transformation should improve the system with respect to only one design goal.
    - Ex) One transformation can aim to improve response time.
    - Ex) Another transformation can aim to improve coherence.
  - 2) Each transformation must be local.
    - A transformation should change only a few methods or a few classes at once.
    - If a transformation changes an interface (e.g., adding a parameter to a method), then the client classes should be changed one at the time

- 3) Each transformation must be applied in isolation to other changes.
  - If you are improving the performance of a method, you should not add new functionality.
  - If you are adding new functionality, you should not optimize existing code.

- 4) Each transformation must be followed by a validation step(검증단계)
  - After completing a transformation and before initiating the next one, validate the changes
    - It is always easier to find and repair a bug shortly after it was introduced than later.

10.4 Mapping Activities

- Optimizing the Object Design Model (Section 10.4.1)
- Mapping Associations to Collections (Section 10.4.2)
- Mapping Contracts to Exceptions (Section 10.4.3)
- Mapping Object Models to a Persistent Storage Schema (Section 10.4.4).

10.4.1 Optimizing the Object Design Model

- four simple but common optimizations:
  - 1) adding associations to optimize access paths,
  - 2) collapsing objects into attributes,
  - 3) delaying expensive computations,
  - 4) And caching the results of expensive computations.

- 1) Optimizing access paths
  - Common sources of inefficiency
    - A) the repeated traversal of multiple associations,
    - B) the traversal of associations with “many” multiplicity,
    - C) and the misplacement of attribute

- A) the repeated traversal of multiple associations,
  - Frequent operations should not require many traversals, but should have a direct connection between the querying object and the queried object.
- B) the traversal of associations with “many” multiplicity,
  - For associations with “many” multiplicity, you should try to decrease the search time by reducing the “many” to “one.”
- C) the misplacement of attribute
  - reconsider folding these attributes into the calling class
  - After folding several attributes, some classes may not be needed anymore and can simply removed from the model

- 2) Collapsing objects(객체의축소): Turning objects into attributes
  - Some of its classes may have few attributes or behaviors left. Such classes, when associated only with one other class, can be collapsed into an attribute, thus reducing the overall complexity of the model.

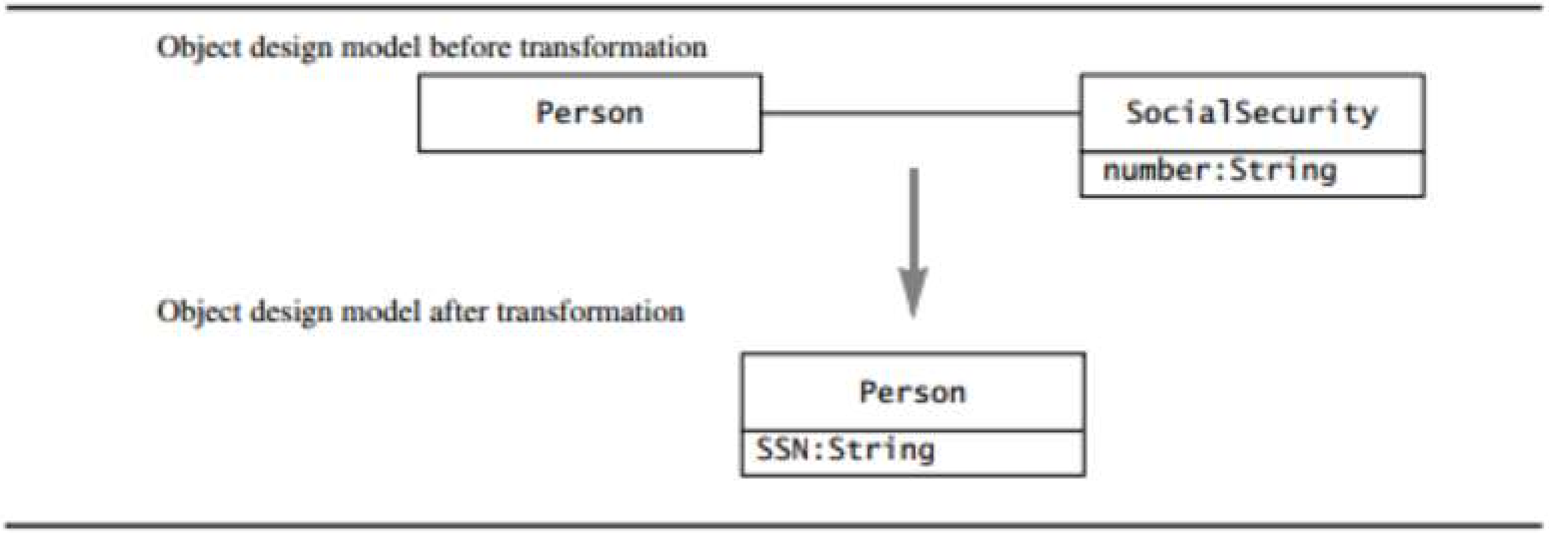

Object design model before transformation
Person
SocialSecurity
number:String
Object design model after transformation
Person
SSN:String

Figure 10-6 Collapsing an object without interesting behavior into an attribute (UML class diagram).

- 3) Delaying expensive computations
  - the image data need not be loaded until the image is displayed
  - We can realize such an optimization using a **Proxy design pattern**

- 프록시 패턴(Proxy Pattern)이란?
  - 프록시는 대리인이라는 뜻으로, 무엇인가를 대신 처리하는 의미이다. 일종의 비서라고 생각하면 된다.사장실에 바로 들어가서 사장님을 바로 만나는게 아닌 비서를 통해 사장님을 만나는 개념이라고 생각할 수 있겠다.

    어떤 객체를 사용하고자 할 때, 객체를 직접 참조하는 것이 아니라 해당 객체를 대행(대리, proxy)하는 객체를 통해 대상 객체에 접근하는 방식을 사용하면 해당 객체가 메모리에 존재하지 않아도 기본적인 정보를 참조하거나 설정할 수 있고,
    실제 객체의 기능이 반드시 필요한 시점까지 객체의 생성을 미룰 수 있다.

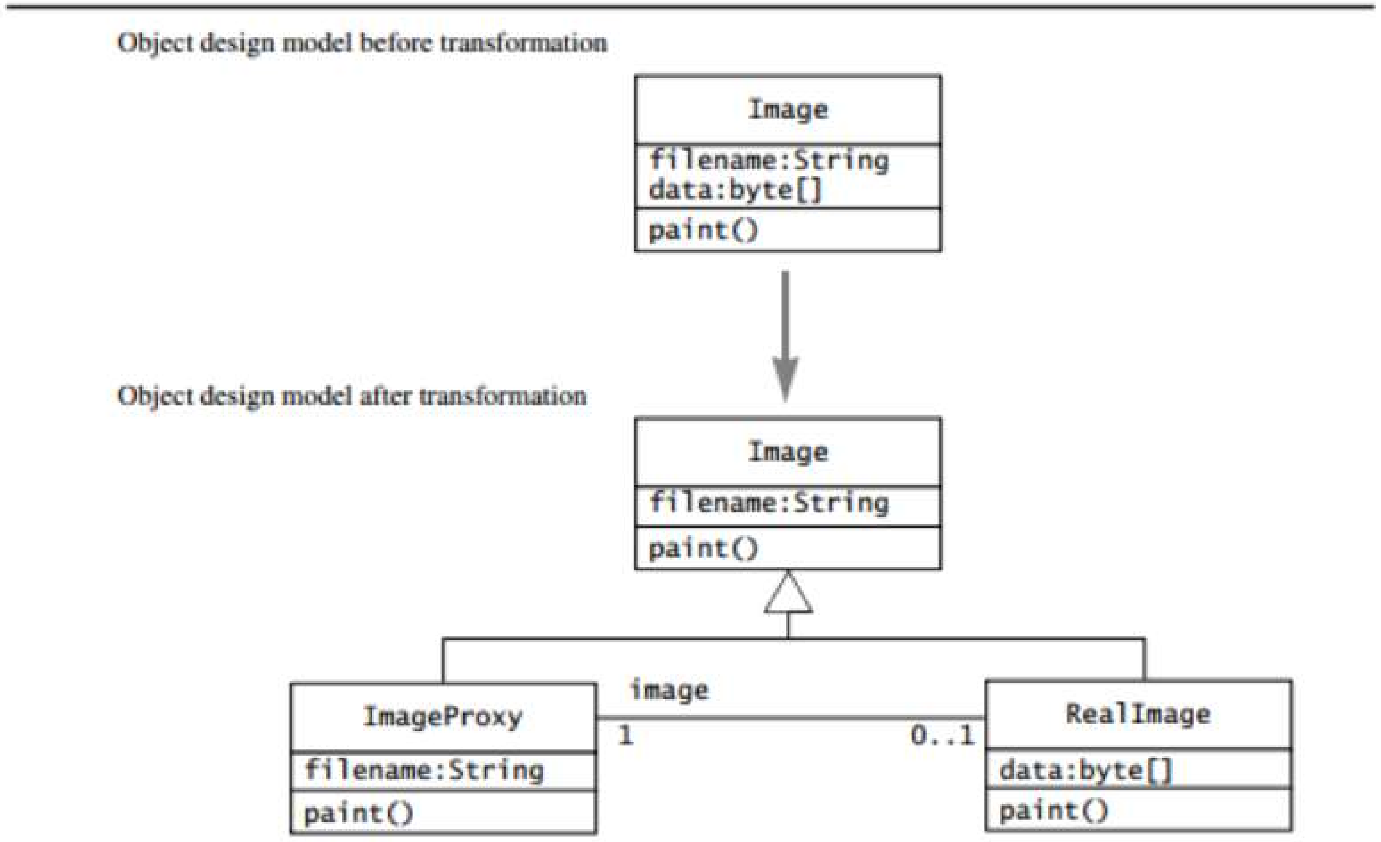


Figure 10-7 Delaying expensive computations to transform the object design model using a Proxy design pattern (UML class diagram).

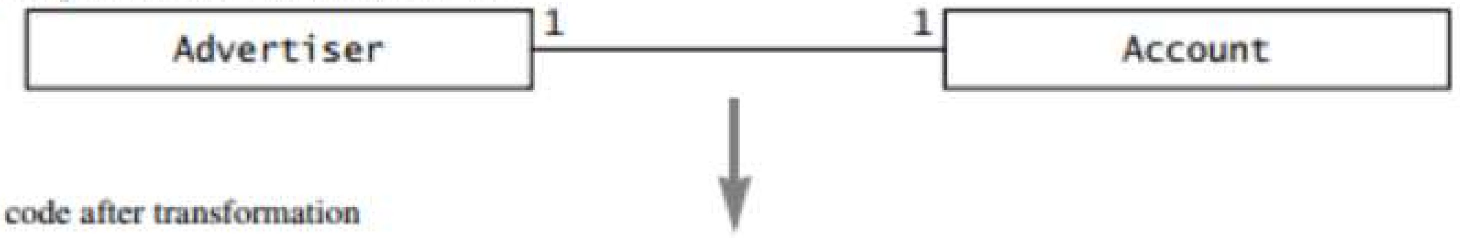
- 4) Caching the result of expensive computations
  - Some methods are called many times, but their results are based on values that do not change or change only infrequently.
  - the result of the computation should be cached as a private attribute

10.4.2 Mapping Associations to Collections

- Associations
  - UML concepts that denote collections of bidirectional links between two or more objects.
  - Object-oriented programming languages, however, do not provide the concept of association.
  - Instead, they provide references, in which one object stores a handle to another object, and collections,

• 1) **Unidirectional one-to-one associations.**

Object design model before transformation

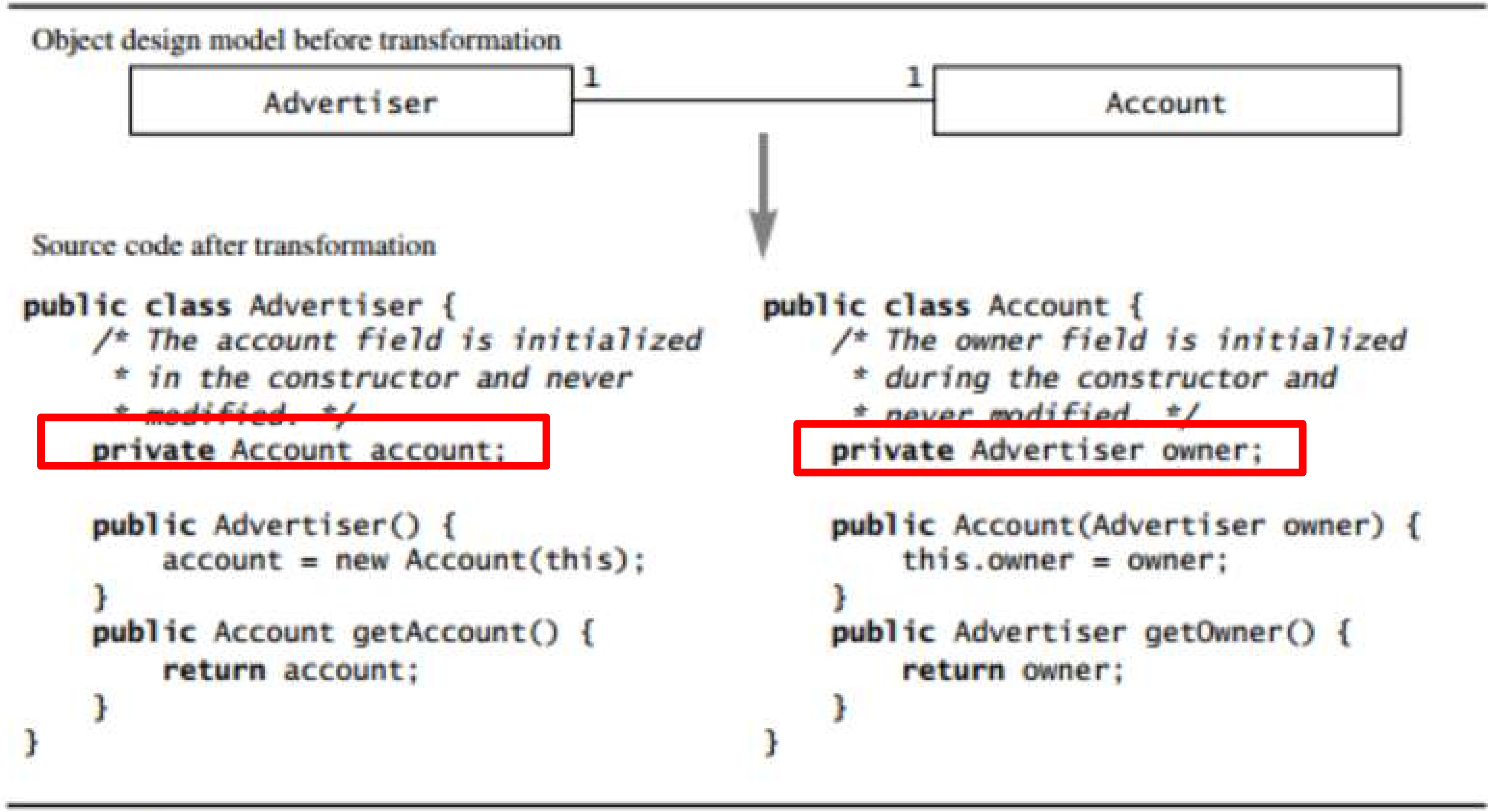


Source code after transformation

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

Figure 10-8 Realization of a unidirectional, one-to-one association (UML class diagram and Java).

- 2) **Bidirectional one-to-one associations.**



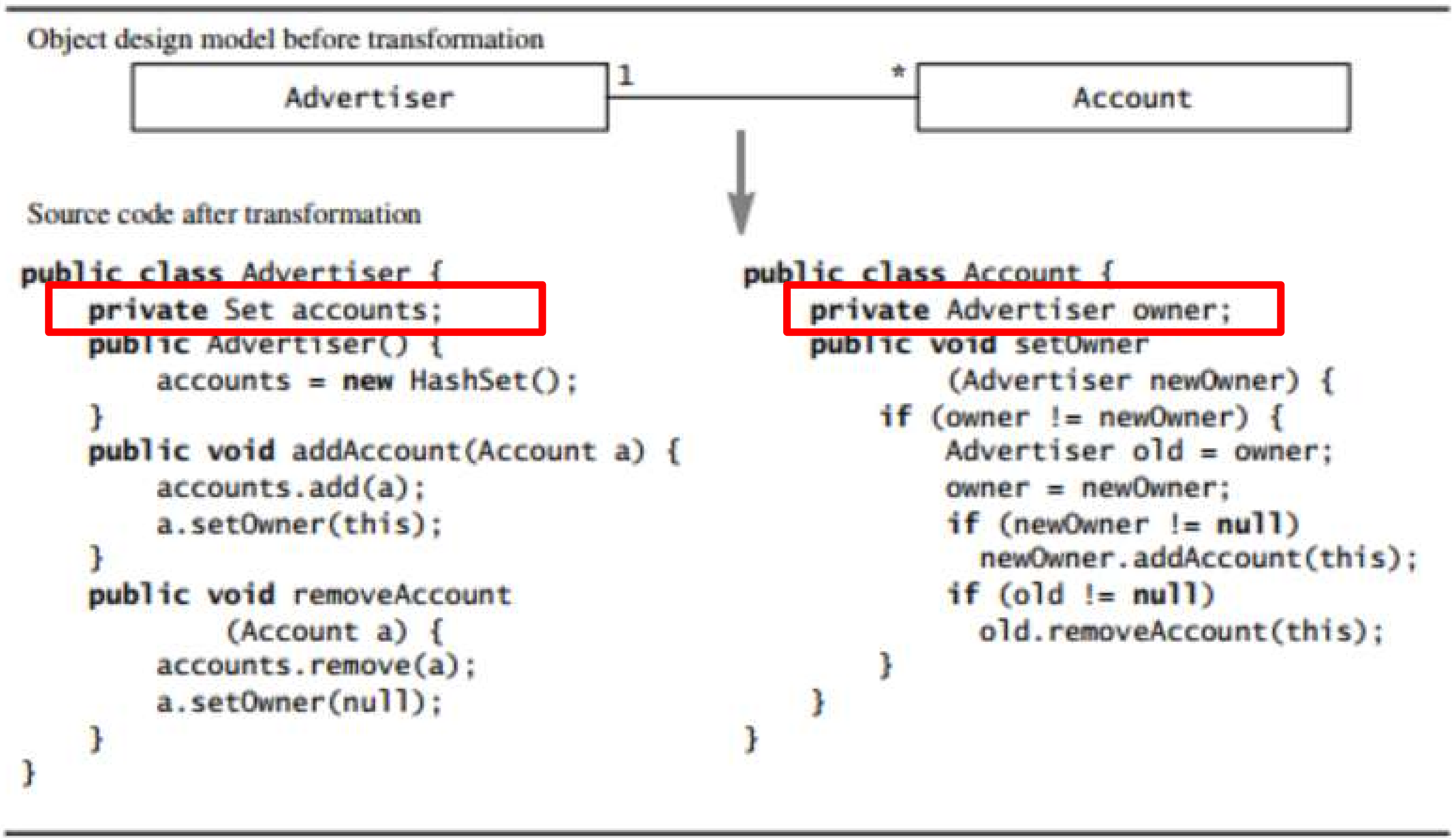
Object design model before transformation
Advertiser
1
1
Account
Source code after transformation

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

Figure 10-9 Realization of a bidirectional one-to-one association (UML class diagram and Java excerpts).

- 3 **One-to-many associations.**


Object design model before transformation

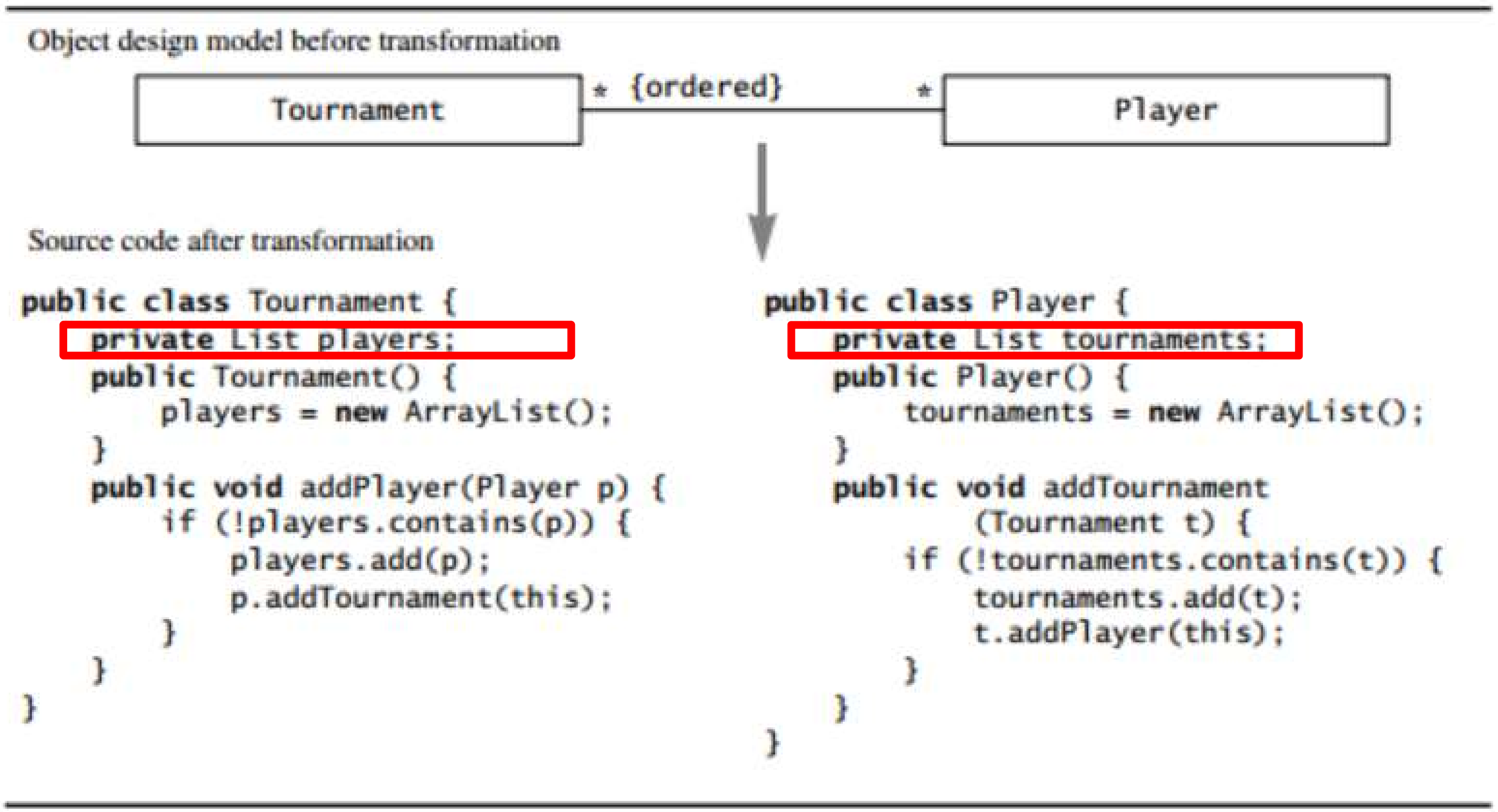
Source code after transformation

```
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 (old != null)
              old.removeAccount(this);
        }
    }
}
```

Ber **Figure 10-10** Realization of a bidirectional, one-to-many association (UML class diagram and Java).

- 4) **Many-to-many associations.**

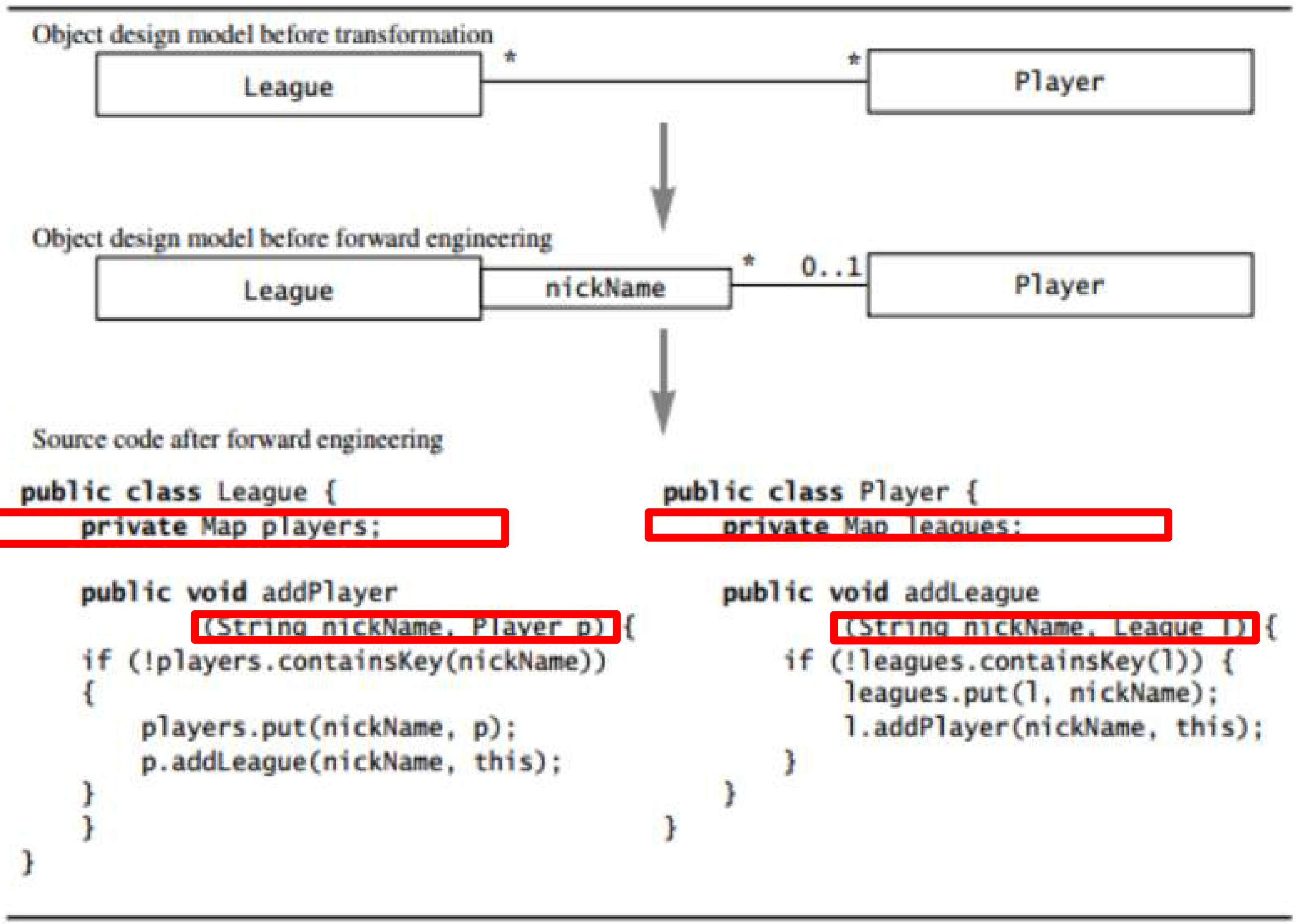

Object design model before transformation
Tournament
* {ordered}
*
Player
Source code after transformation

```
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);
        }
    }
}
```

Figure 10-11 Realization of a bidirectional, many-to-many association (UML class diagram and Java).

- 5) **Qualified associations.**



Object design model before transformation
League
*
*
Player
Object design model before forward engineering
League
nickName
*
0..1
Player
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);
        }
    }
}
```

Figure 10-12 Realization of a bidirectional qualified association (UML class diagram; arrow denotes the successive transformations).

- **6) Associations classes.**

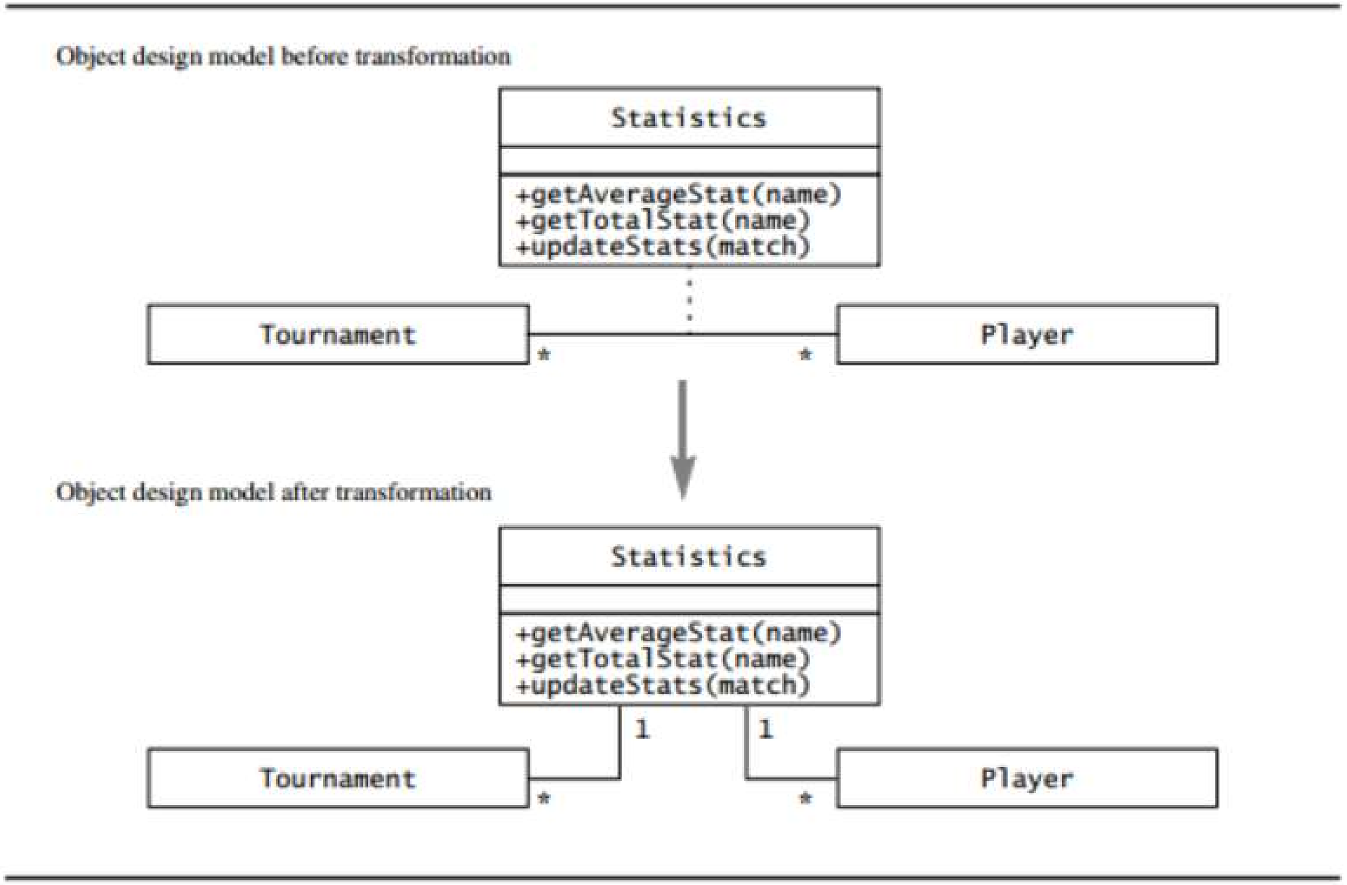
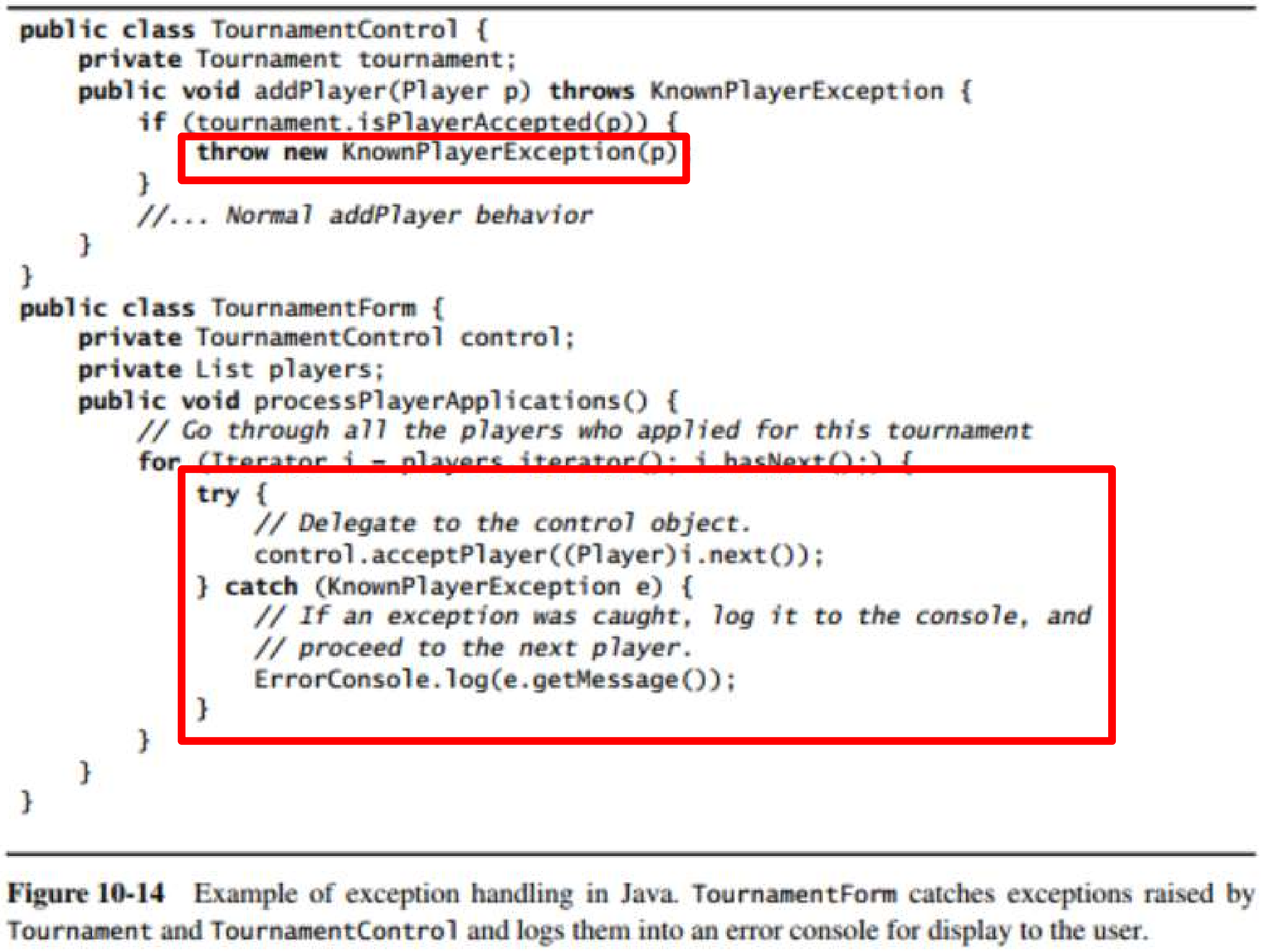


Figure 10-13 Transformation of an association class into an object and two binary associations (UML class diagram).

10.4.3 Mapping Contracts to Exceptions

- many object-oriented languages, including Java, do not provide built-in support for contracts.
- However, we can use their exception mechanisms as building blocks for signaling and handling contract violations.
- an exception with the **throw** keyword followed by an exception object

- **Figure 10-14** Example of exception handling in Java. TournamentForm catches exceptions raised by Tournament and TournamentControl and logs them into an error console for display to the user.

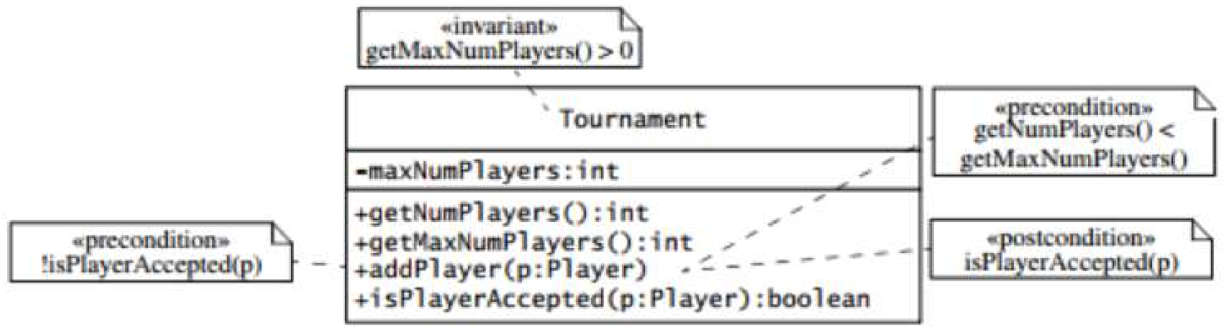


```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
            throw new KnownPlayerException(p)
        }
        //... Normal addPlayer behavior
    }
}
public class TournamentForm {
    private TournamentControl control;
    private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and
                // proceed to the next player.
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```

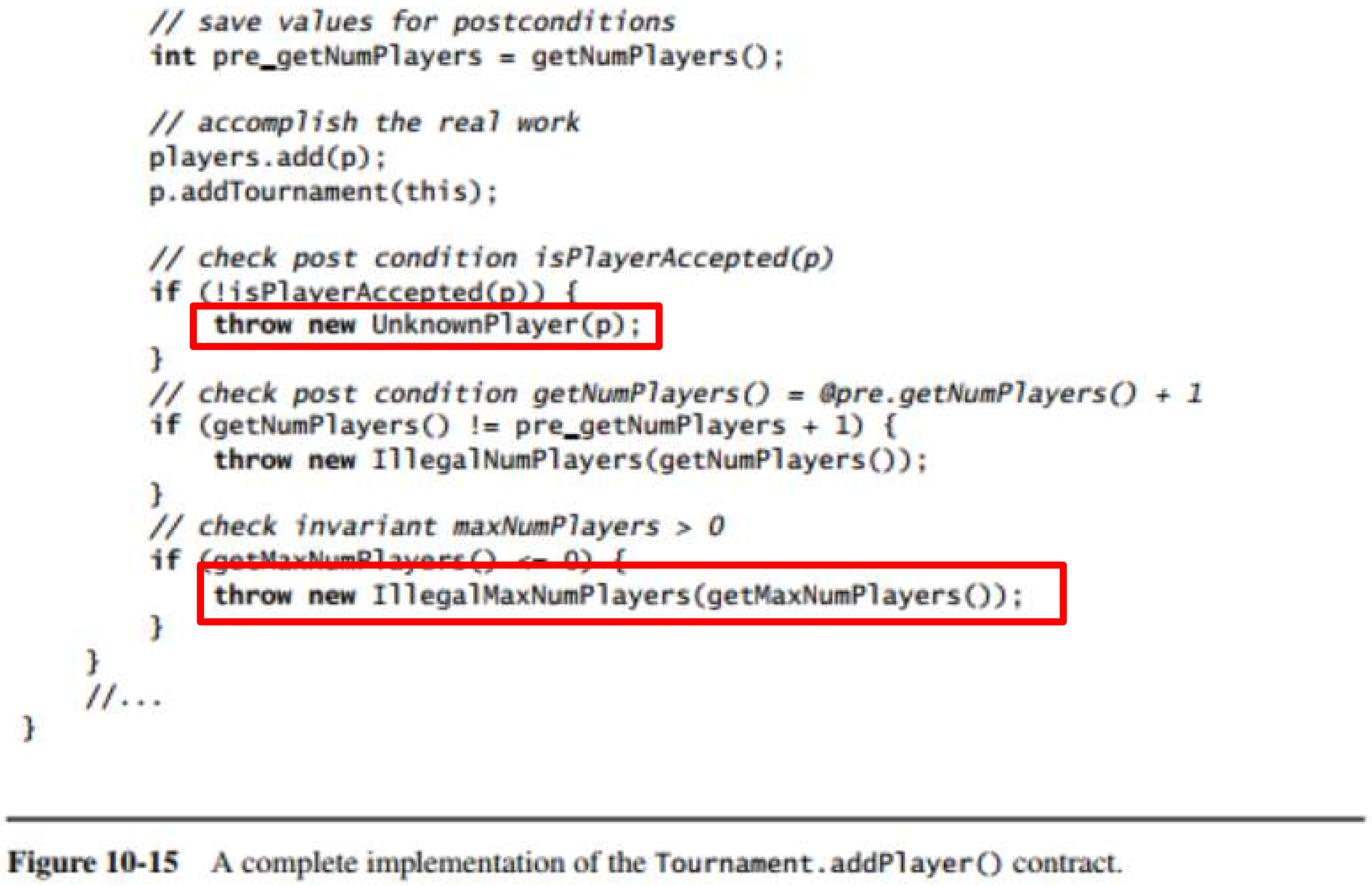
Figure 10-14 Example of exception handling in Java. `TournamentForm` catches exceptions raised by `Tournament` and `TournamentControl` and logs them into an error console for display to the user.

- **Figure 10-15** A complete implementation of the Tournament.addPlayer() contract.

- If we mapped every contract following the above steps, we would ensure that all preconditions, postconditions, and invariants are checked for every method invocation, and that violations are detected within one method invocation.



```
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();
```



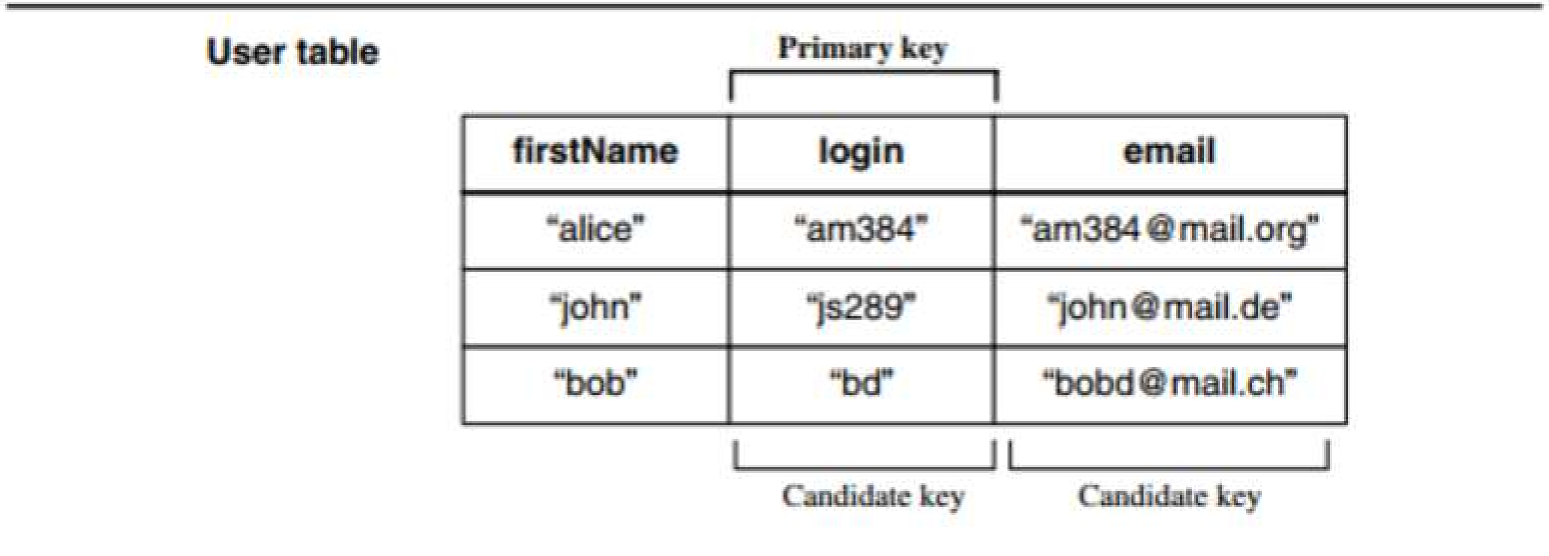
```
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();

        // 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());
        }
    }
    //...
}
```

Figure 10-15 A complete implementation of the `Tournament.addPlayer()` contract.

10.4.4 Mapping Object Models to a Persistent Storage Schema

- the steps involved in mapping an object model to a relational database using Java and database schemas
- A **schema** is a description of the data, that is, a meta-model for data

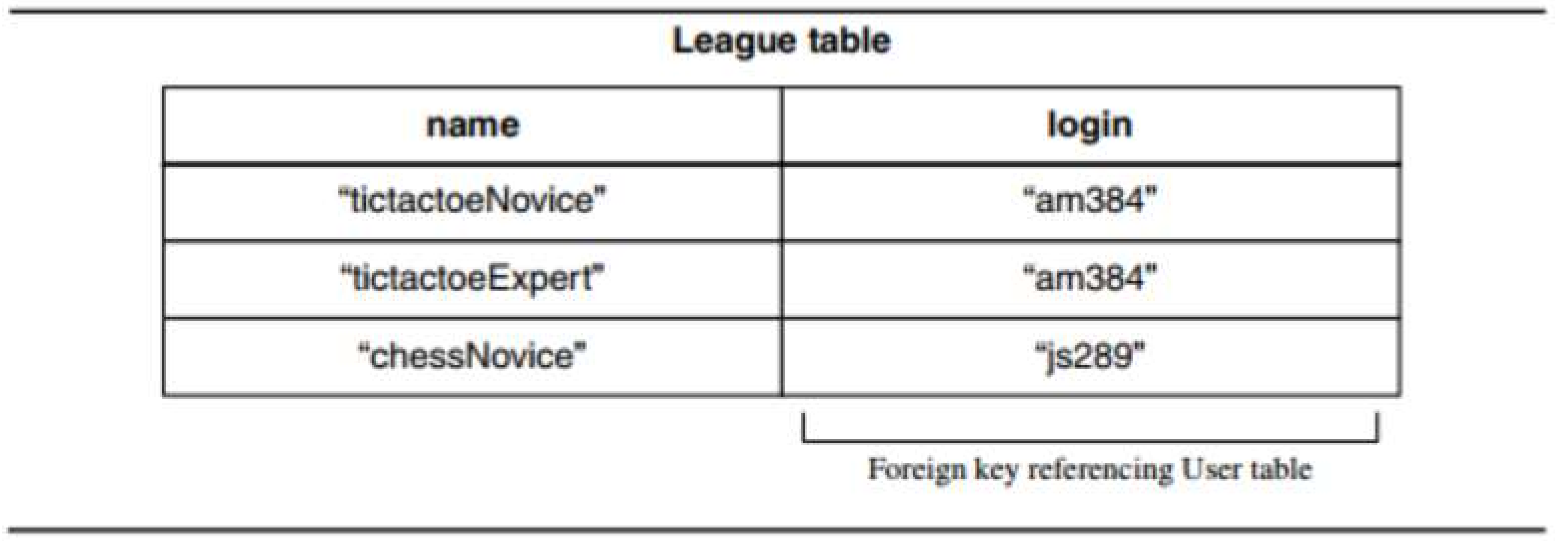
- **Figure 10-16** An example of a relational table, with three attributes and three data records.
  - PRIMARY KEY
  - CANDIDATE KEY


User table
Primary key
Candidate key
Candidate key

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

Figure 10-16 An example of a relational table, with three attributes and three data records.

- **Figure 10-17** An example of a foreign key. The owner attribute in the League table refers to the primary key of the User table in Figure 10-16.



League table

name	login
"tictactoeNovice"	"am384"
"tictactoeExpert"	"am384"
"chessNovice"	"js289"

Foreign key referencing User table

Figure 10-17 An example of a foreign key. The `owner` attribute in the `League` table refers to the primary key of the `User` table in Figure 10-16.

- ***Mapping classes and attributes***
  - When mapping the persistent objects to relational schemata, we focus first on the classes and their attributes.
  - We map each class to a table with the same name. For each attribute, we add a column in the table with the name of the attribute in the class.

- **Figure 10-18** Forward engineering of the User class to a database table.
  - The type of the id column is a long integer that we increment every time we create a new object.

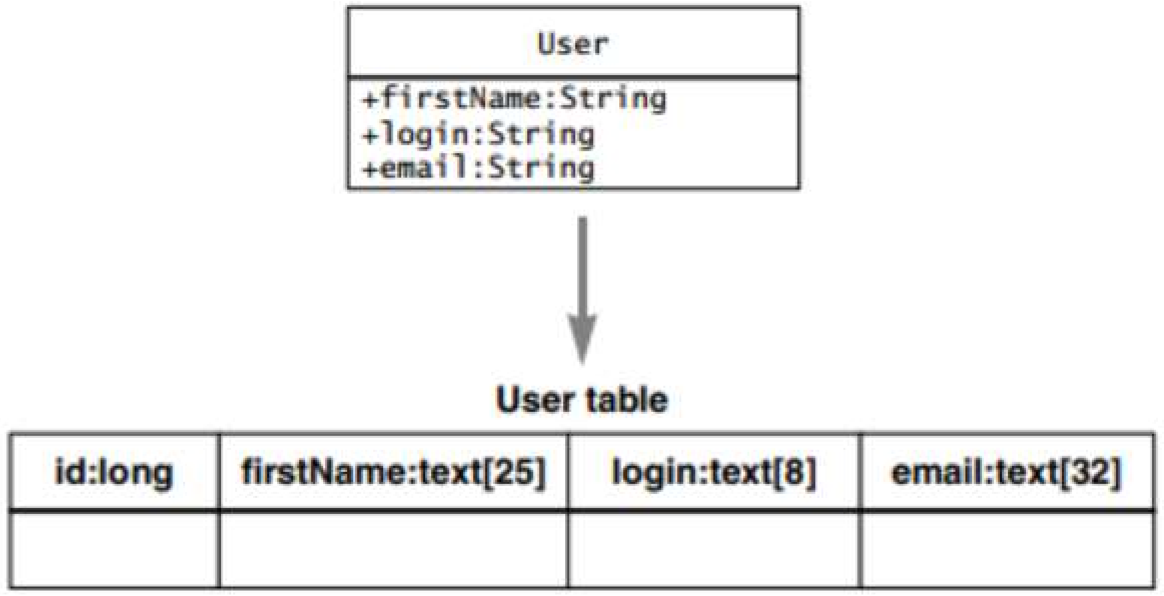


Figure 10-18 Forward engineering of the `User` class to a database table.

- ***Mapping associations***
  - One-to-one and one-to-many associations are implemented as a so-called **buried association**
  - Many-to-many associations are implemented as a separate table.

- **Figure 10-19** Mapping of the LeagueOwner/League association as a buried association.
  - one-to-many associations
  - map this association by adding a owner column to the League table referring to the primary key of the LeagueOwner table

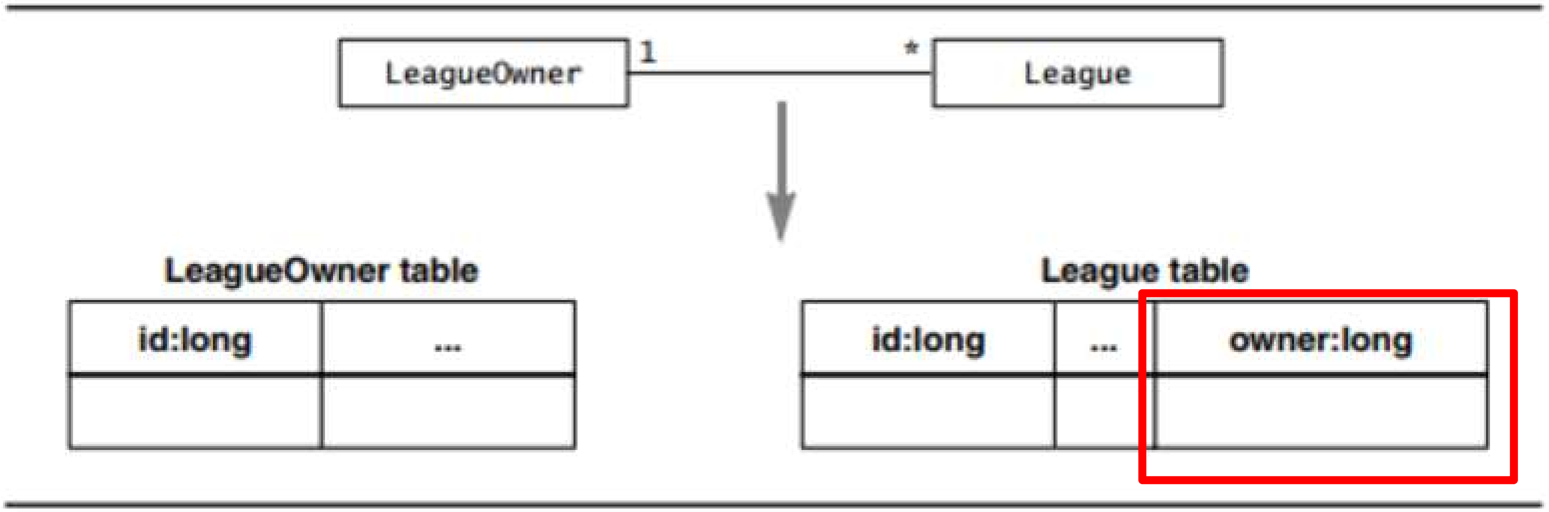
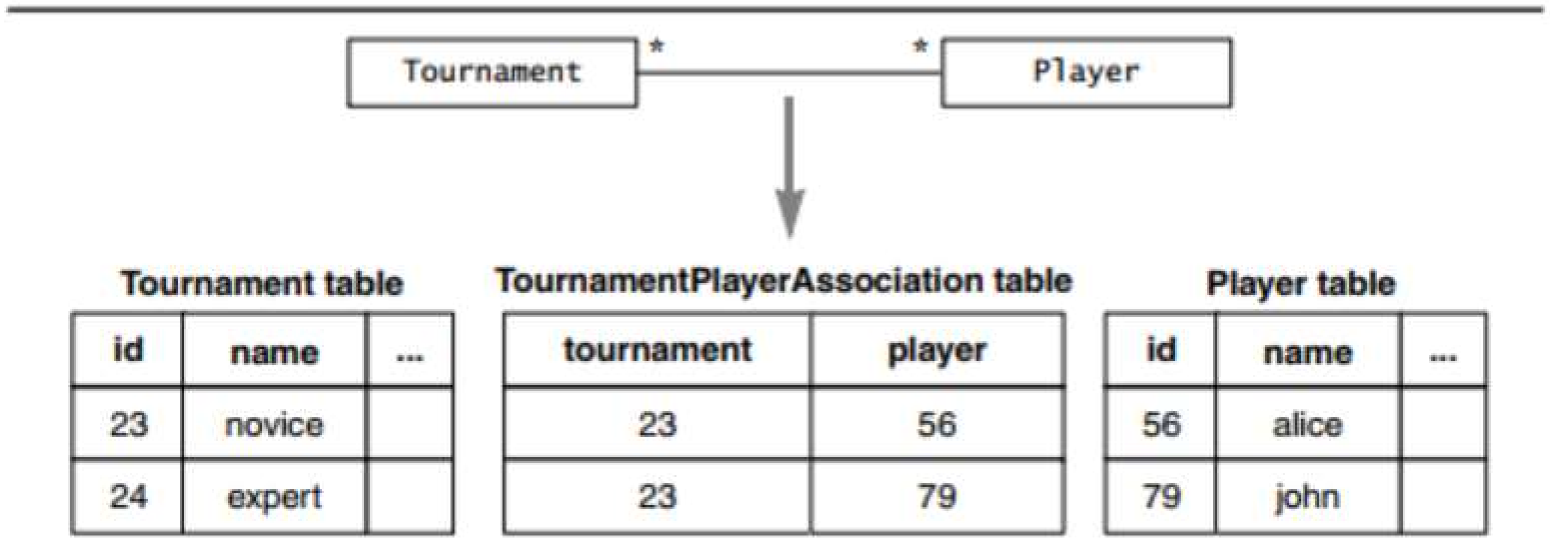


Figure 10-19 Mapping of the LeagueOwner/League association as a buried association.

- **Figure 10-20** Mapping of the Tournament/Player association as a separate table.
  - Many-to-many associations



Tournament table

id	name	...
23	novice	
24	expert	

TournamentPlayerAssociation table

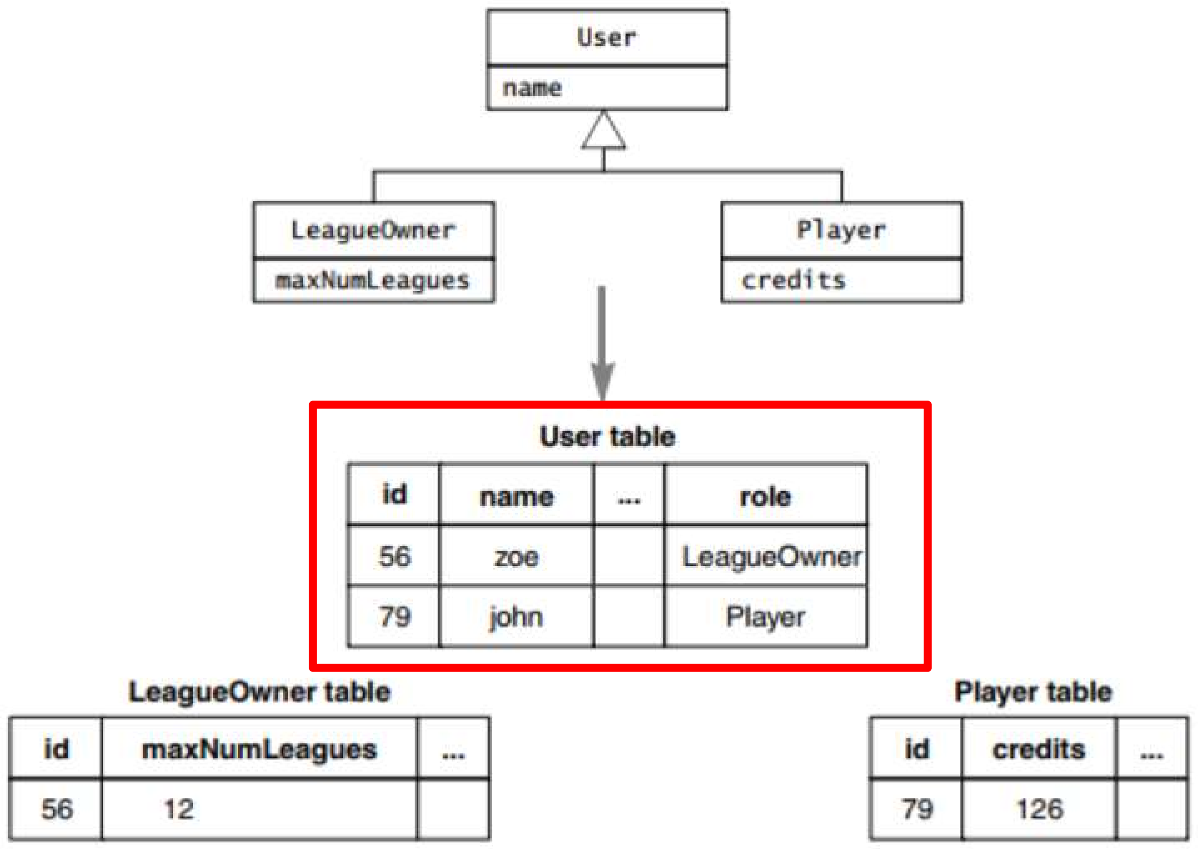
tournament	player
23	56
23	79

Player table

id	name	...
56	alice	
79	john	

Figure 10-20 Mapping of the Tournament/Player association as a separate table.

- ***Mapping inheritance relationships***
  - In the first option, called *vertical mapping*, similar to a one-to-one association, each class is represented by a table and uses a foreign key to link the subclass tables to the superclass table.
  - In the second option, called *horizontal mapping*, the attributes of the superclass are pushed down into the subclasses, essentially duplicating columns in the tables corresponding to subclasses



User table

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

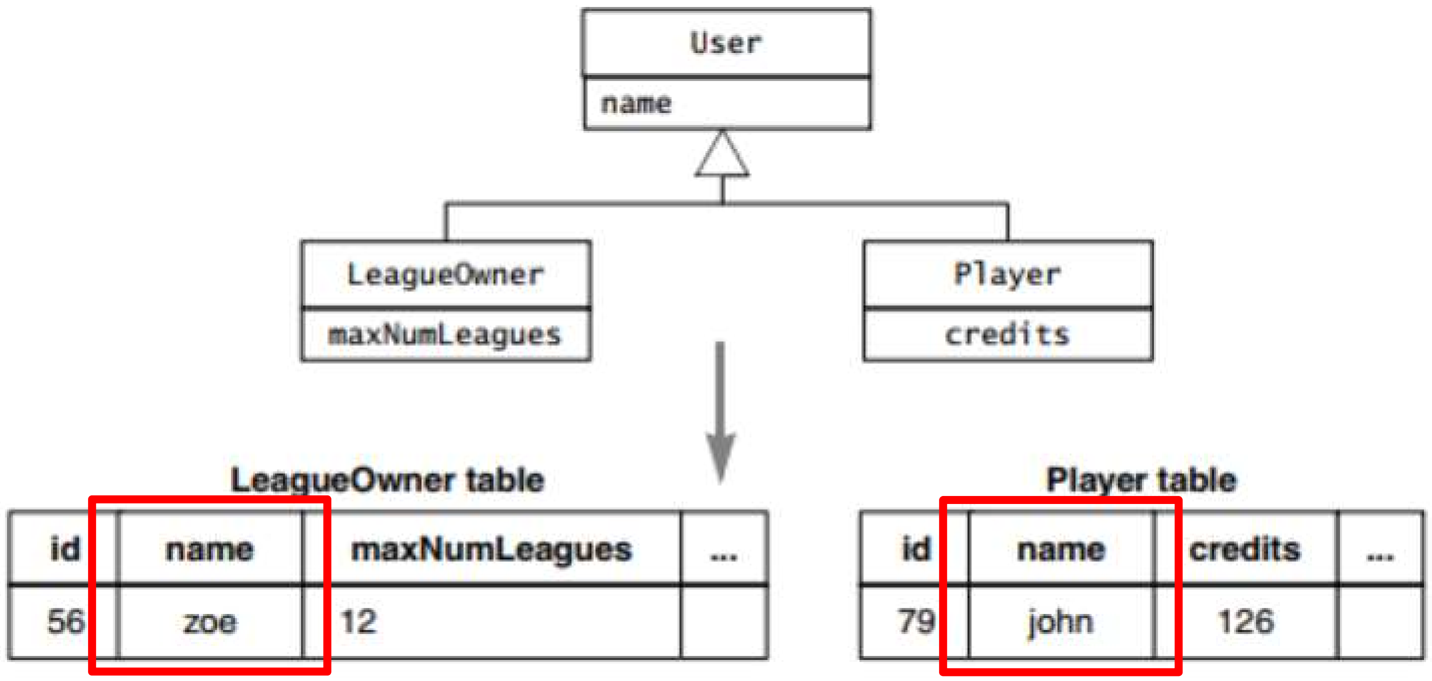
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Figure 10-21 Realizing the `User` inheritance hierarchy with a separate table.



LeagueOwner table

id	name	maxNumLeagues	...
56	zoe	12	

Player table

id	name	credits	...
79	john	126	

Figure 10-22 Realizing the User inheritance hierarchy by duplicating columns.

끝