Section 5.2
Mapping to Collections

1. Overview
2. Mapping associations
3. Optimizing associations

5.2.1 Overview

- How do we map associations to collections?
  - associations in UML
    - they are represented as links between objects
    - they can be unidirectional or bidirectional
  - associations in a programming language
    - they are represented as references to other objects
      - the exact kind of reference is not important
      - it could be a pointer, a C++ reference, etc.
    - by nature, associations in programming language are *unidirectional*

5.2.2 Mapping Associations

- Mapping associations to programming constructs
  - associations are implemented as:
    - single references
      - one object stores a handle to another object
    - collections
      - one object stores references to several objects of the same class
  - references are always unidirectional between two objects
  - bidirectional associations require more work

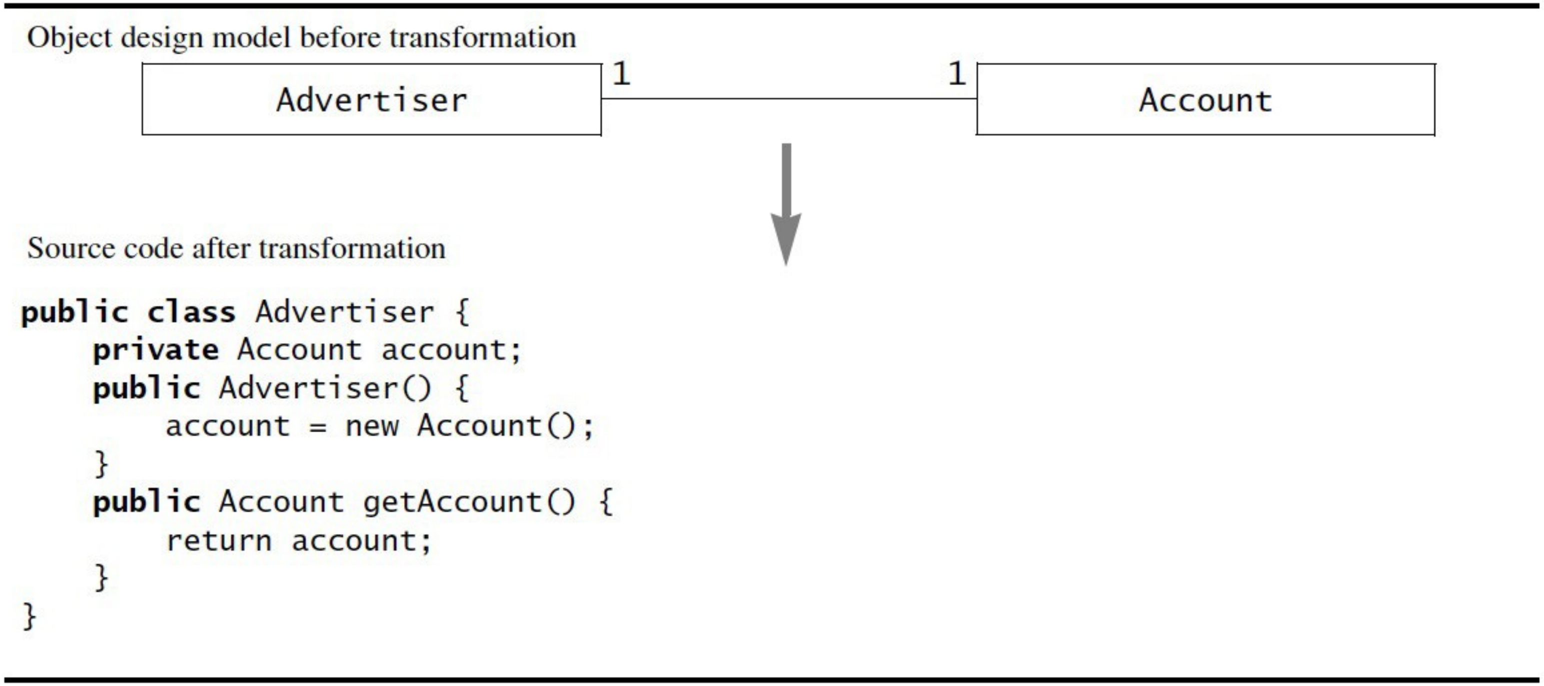
©2018 Christine Laurendeau, based on Bruegge and Dutoit, Object-Oriented Software Engineering, 3rd ed., Pearson, 2010

Mapping Associations (cont.)

- Implementing different kinds of associations
  - unidirectional one-to-one
  - bidirectional one-to-one
  - one-to-many
  - many-to-many
  - qualified associations
  - association classes

Unidirectional One-to-One Associations

- Mapped as a reference within source object to destination



Object design model before transformation

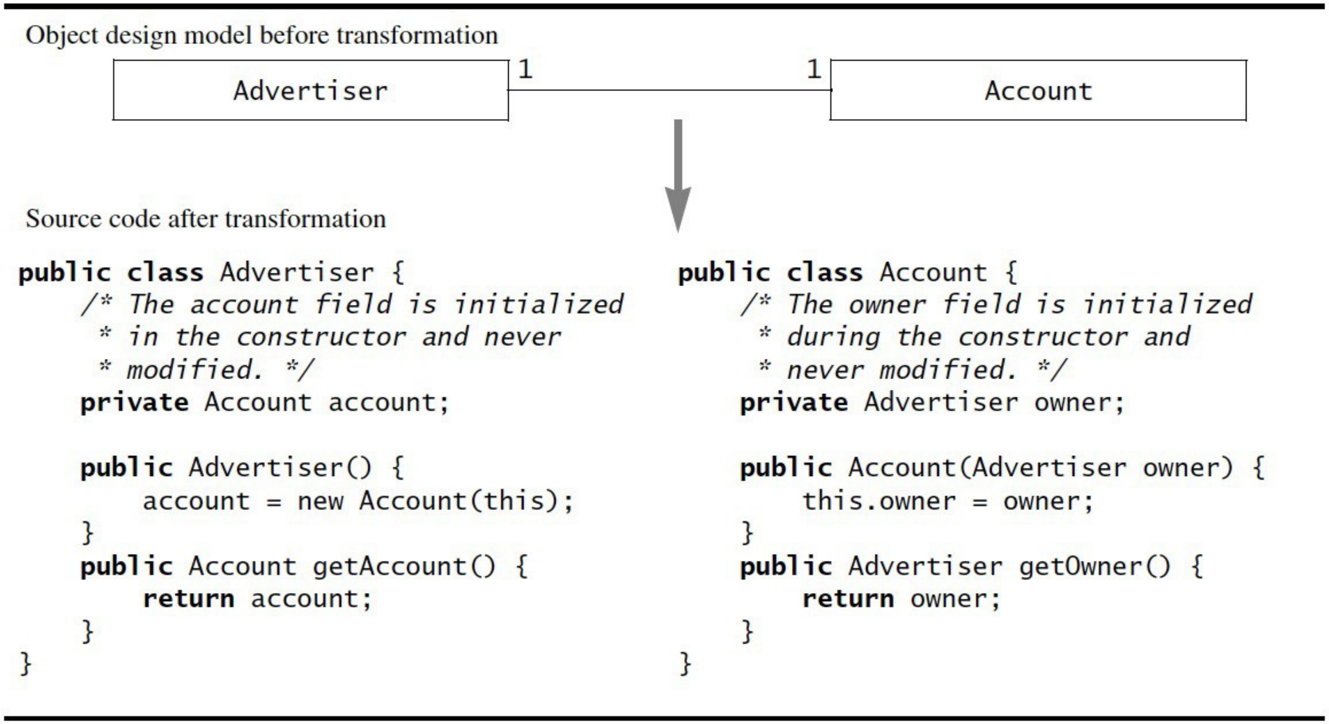
Source code after transformation

```
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).

Bidirectional One-to-One Associations

- Mapped as:
  - a reference within the source object to the destination object
  - a reference within the destination object to the source object
- Consistency must be ensured



Object design model before transformation

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;
    }
}
```

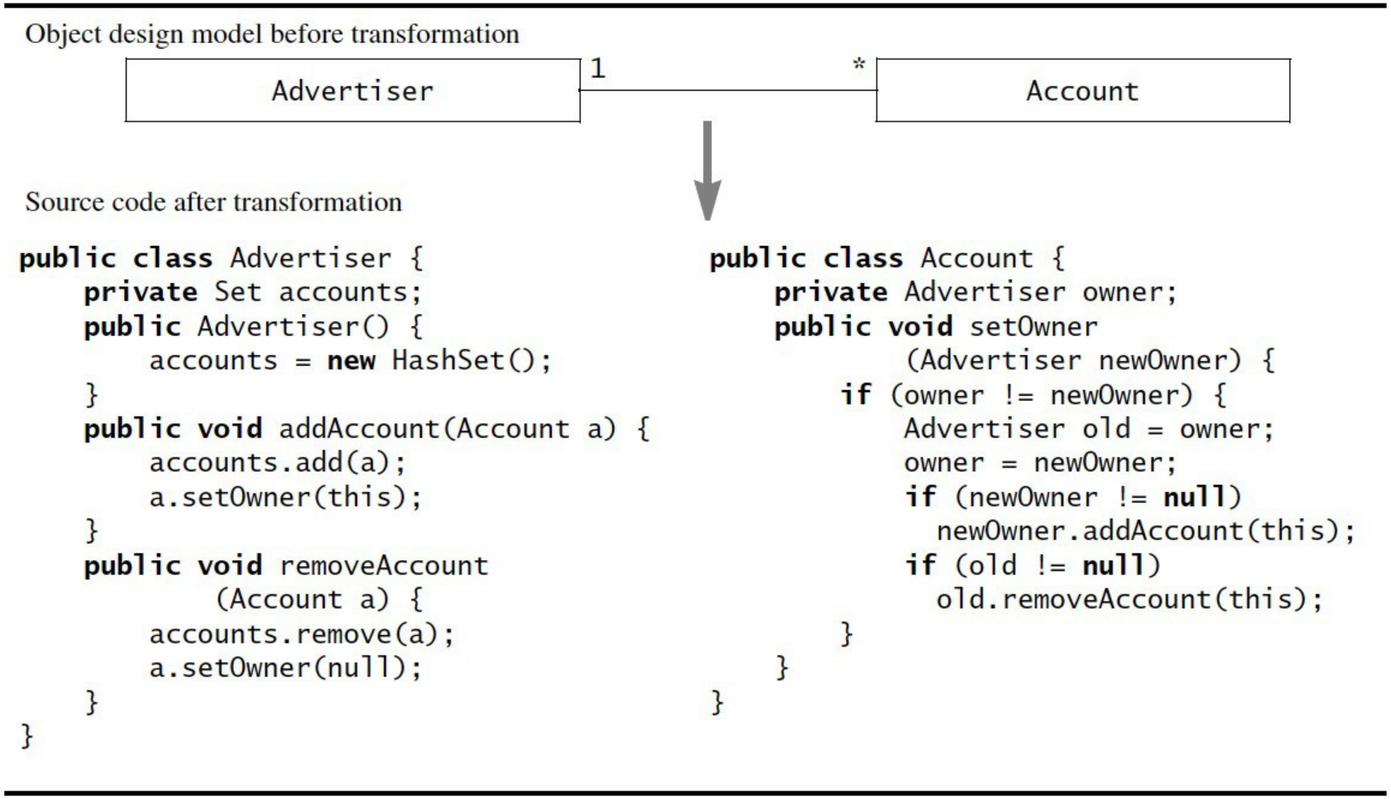
```
public class Account {
    /* The owner field is initialized
     * during the constructor and
     * never modified. */
    private Advertiser owner;

    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).

One-to-Many Associations

- Within source object, collection of references to destination
- May be unidirectional or bidirectional



Object design model before transformation

Advertiser
1
*
Account

Source code after transformation

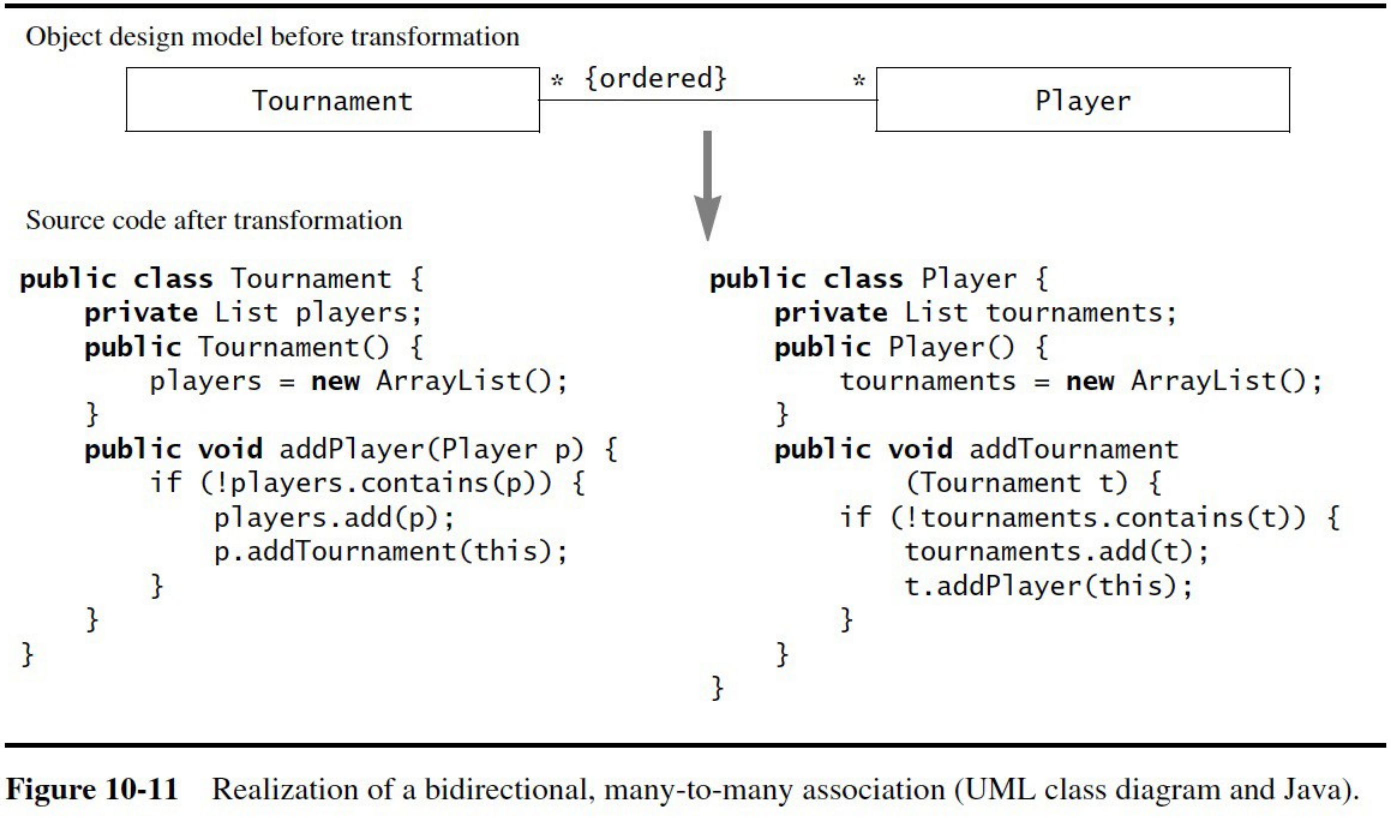
```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount
            (Account a) {
        accounts.remove(a);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

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

Many-to-Many Associations

- Mapped as:
  - within each source object, collection of references to destination
  - within each destination object, collection of references to source



Object design model before transformation

Source code after transformation

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

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

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

5.2.3 Optimizing Associations

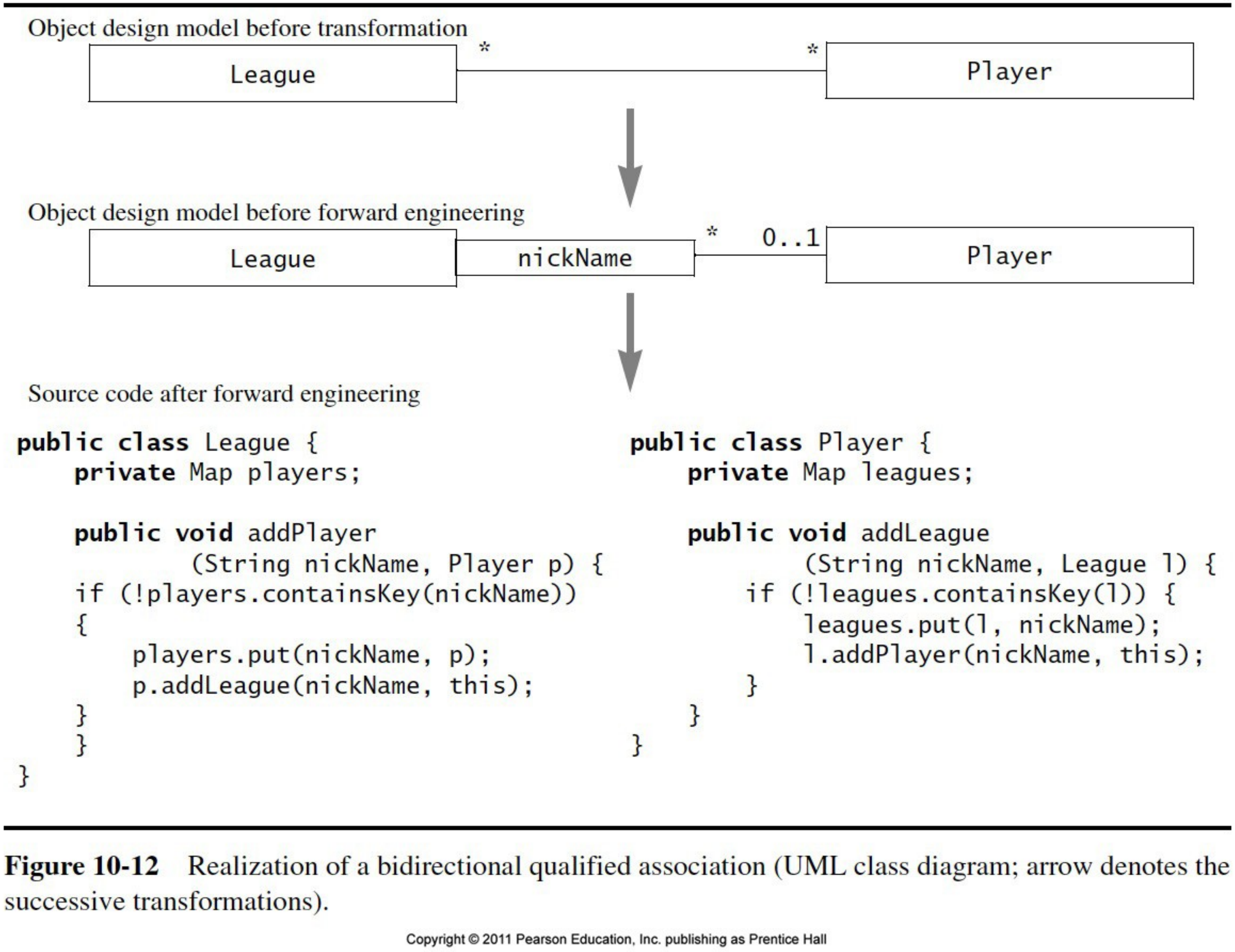
- Associations with a “many” side can be problematic
  - they can be slow to access
  - it can be difficult to maintain consistency

- Solutions
  - qualified associations
  - association classes

Qualified Associations

- Why use qualified associations?
  - they are used to reduce the multiplicity on the "many" side of an association
  - they can be used with one-to-many or many-to-many associations
  - they are mapped as:
    - an additional *qualifier* attribute on the destination object
      - it must have a unique value
    - a keyed collection (e.g. `Map`) on the source object, where:
      - the key is the destination object qualifier
      - the value is the destination object

Qualified Associations (cont.)



Object design model before transformation

Object design model before forward engineering

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).

Association Classes

- Why association classes?
  - used to hold attributes and operations specific to an association
  - they are implemented as separate object with binary associations
  - each binary association is mapped to a set of reference attributes

Association Classes (cont.)

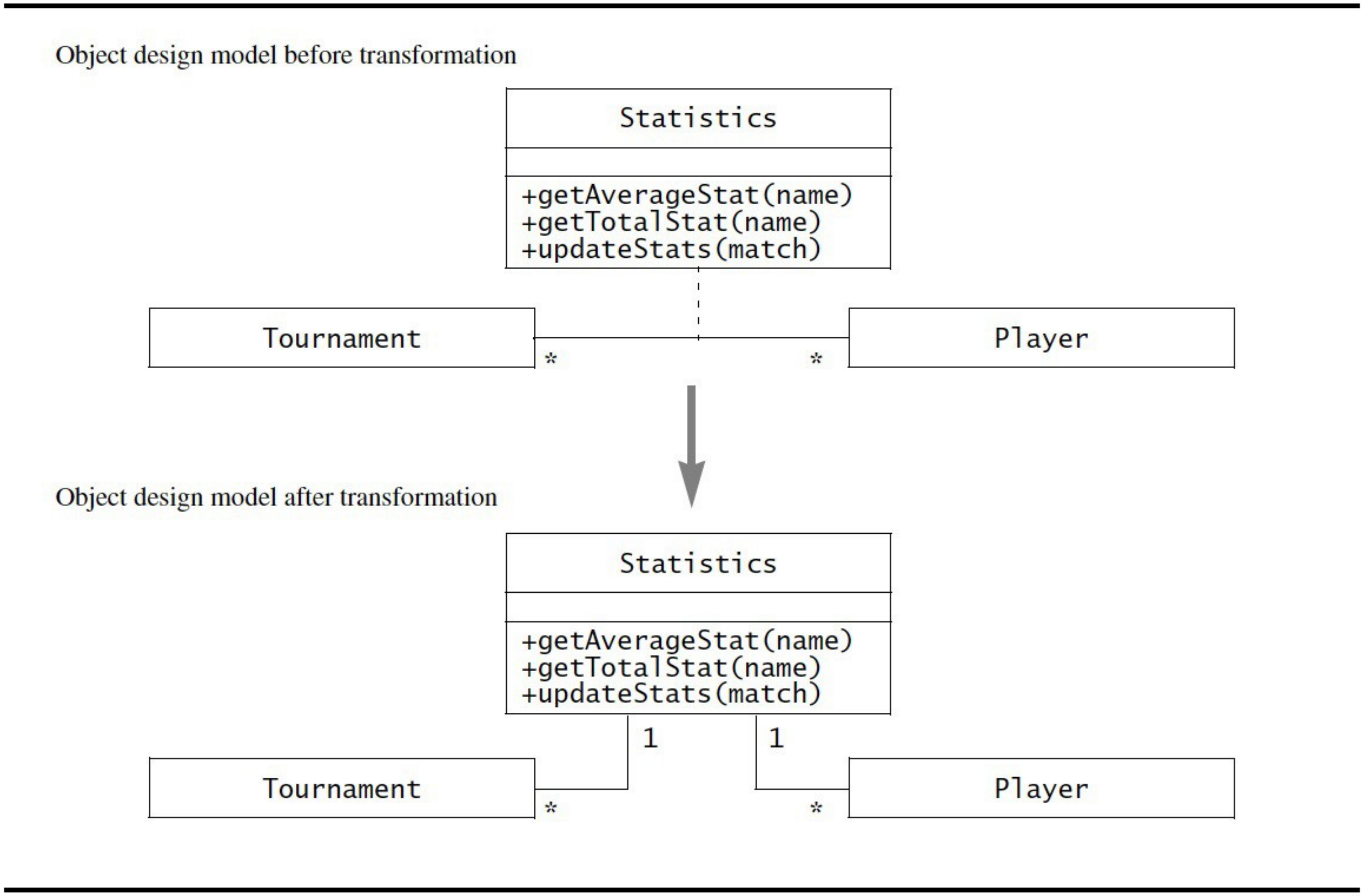


Figure 10-13 Transformation of an association class into an object and two binary associations (UML class diagram).