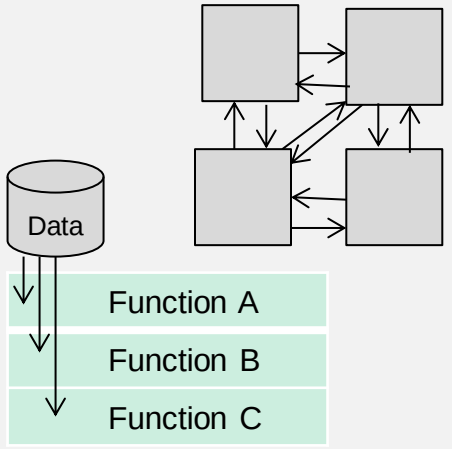
Design Issues

Lecture 12

System Design Concepts

Subsystems

- Coupling: dependency between two subsystems
- Cohesion: dependencies within a subsystem
- Desire LOW coupling and HIGH cohesion

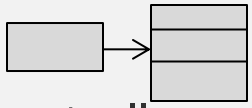


Refinement

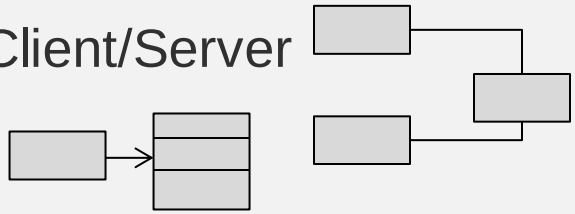
- Layering
- Partitions

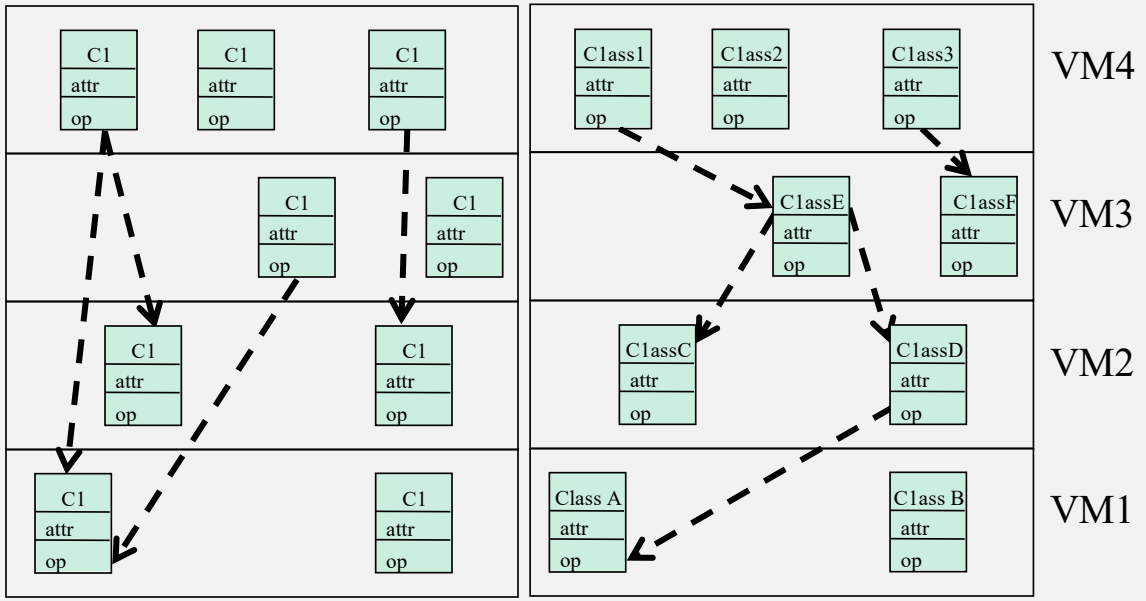
Software Architecture Patterns

- Repository



- Model/View/Controller
- Client/Server





Data Design

Some objects in the system model need to be persistent:

- Values for their attributes have a lifetime longer than a single execution

A persistent object can be realized with one of the following mechanisms:

- Filesystem:
  - If the data are used by multiple readers but a single writer
- Database:
  - If the data are used by concurrent writers and readers.

Data Management Questions

- How often is the database accessed?
  - What is the expected request (query) rate? The worst case?
  - What is the size of typical and worst case requests?
- Do the data need to be archived?
- Should the data be distributed?
  - Does the system design try to hide the location of the databases (location transparency)?
- Is there a need for a single interface to access the data?
- What is the query format?
- Should the data format be extensible?

Storage Efficiency

Minimize null values and redundancy

Reduce update anomalies

Normalization process optimizes the data storage designed for storage efficiency

Normalization Process

1. Remove repeating group (x normal form)
2. Remove any partial dependency (x normal form)
3. Remove any transitive dependencies (x normal form)

4. Mapping a class to a table
5. Mapping association

Normalization Process

SalesPerson number	SalesPerson Name	Sales Area	Customer Number	Customer Name	Wharehouse Number	Warehouse Location	Sales Amount
3462	Babu	South	1	DataBank	4	Sutrapur	13540
			2	CodeRank	3	Palashi	10600
			3	BallRoll	3	Gopibag	9700
3593	Sufian	North	4	AdaBig	2	Uttara	11560
			5	BiteBig	2	Uttara	2560
			6	DrinkBig	1	Banani	8000
etc							

First Normal Form (1NF)

Remove repeating groups.

The primary key with repeating group attributes are moved into a new table.

When a relation contains no repeating groups, it is in first normal form.

SalesPerson number	SalesPerson Name	Sales Area	Customer Number	Customer Name	Wharehouse Number	Warehouse Location	Sales Amount

SalesPerson number	SalesPerson Name	Sales Area
3462	Abu	South
3593	Sufian	North
etc		

SalesPerson number	Customer Number	Customer Name	Wharehouse Number	Warehouse Location	Sales Amount
3462	1	DataBank	4	Sutrapur	13540
3462	2	CodeRank	3	Palashi	10600
3462	3	BallRoll	3	Palashi	9700
3593	4	AdaBig	2	Uttara	11560
3593	5	BiteBig	2	Uttara	2560
3593	6	DrinkBig	1	Banani	8000
etc					

The original unnormalized relation SALES-REPORT is separated into two relations, SALESPERSON (3NF) and SALESPERSON-CUSTOMER (1NF).

Second Normal Form (2NF)

Remove any partially dependent attributes and place them in another relation.

A partial dependency is when the data are dependent on a part of a primary key.

A relation is created for the data that are only dependent on part of the key and another for data that are dependent on both parts.

SalesPerson number	Customer Number	Customer Name	Whareho use Number	Warehou se Location	Sales Amount

Second Normal Form

The relation SALESPERSON-CUSTOMER is separa into a relation called CUSTOMER-WAREHOUSE (2NF) a a relation called SALE (1NF).

SalesPerson number	Customer Number	Sales Amount
3462	1	13540
3462	2	10600
3462	3	9700
3593	4	11560
3593	5	2560
3593	6	8000
etc		

Customer Number	Customer Name	Whareho use Number	Warehou se Location
1	DataBank	4	Sutrapur
2	CodeRank	3	Chankar pul
3	BallRoll	3	Palashi
4	AdaBig	2	Uttara
5	BiteBig	2	Basundh ara
6	DrinkBig	1	Banani
etc			

Third Normal Form (3NF)

Must be in 2NF

Remove any transitive dependencies.

A transitive dependency is when nonkey attributes are dependent not only on the primary key, but also on a nonkey attribute.

Third Normal Form

Customer Number	Customer Name	Wharehouse Number	Warehouse Location

Customer Number	Customer Name	Wharehouse Number
1	DataBank	4
2	CodeRank	3
3	BallRoll	3
4	AdaBig	2
5	BiteBig	2
6	DrinkBig	1
etc		

Wharehouse Number	Warehouse Location
4	Sutrapur
3	Palashi
2	Uttara
1	Banani
etc	

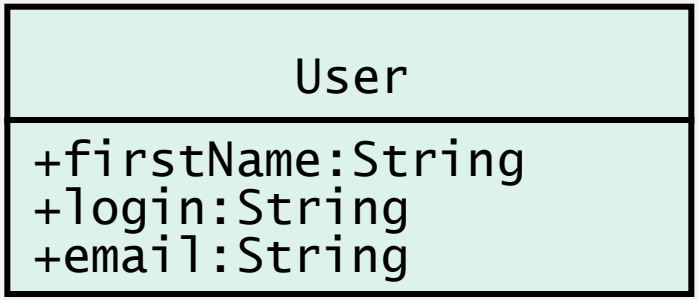
The relation CUSTOMER-WAREHOUSE is separated into two relations called CUSTOMER (1NF) and WAREHOUSE (3NF).

Mapping Object Models

UML object models can be mapped to relational databases

- The mapping:
  - Each class is mapped to its own table
  - Each class attribute is mapped to a column in the table
  - An instance of a class represents a row in the table
- Methods are not mapped

Mapping a Class to a 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”	“rusa@mail.bd”
“bob”	“bd”	“bobd@mail.nc”

Candidate key Candidate key

League table

name	login
“tictactoeNovice”	“am384”
“tictactoeExpert”	“bd”
“chessNovice”	“js289”

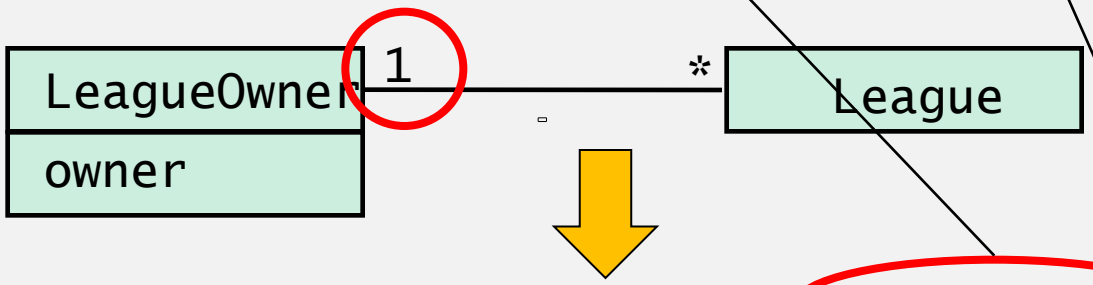
Foreign key referencing User table

Buried Association

Associations with multiplicity “one” can be implemented using a foreign key

For one-to-many associations we add 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.



LeagueOwner table

id:long	owner:long	...

League table

id:long	...	owner:long

Another Example for Buried Association

Transaction Table

transactionID	portfolioID

Foreign Key

Portfolio Table

portfolioID	...

Mapping Many-To-Many Associations

In this case we need a separate table for the association

Separate table for the association "Serves"

Primary Key

City Table

cityName
Houston
Albany
Munich
Hamburg

Airport Table

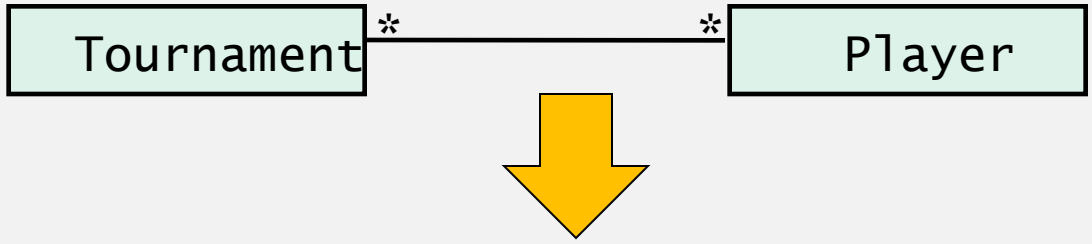
airportCode	airportName
IAH	Intercontinental
HOU	Hobby
ALB	Albany County
MUC	Munich Airport
HAM	Hamburg Airport

Serves Table

cityName	airportCode
Houston	IAH
Houston	HOU
Albany	ALB
Munich	MUC
Hamburg	HAM

Another Many-to-Many Association Mapping

We need 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	

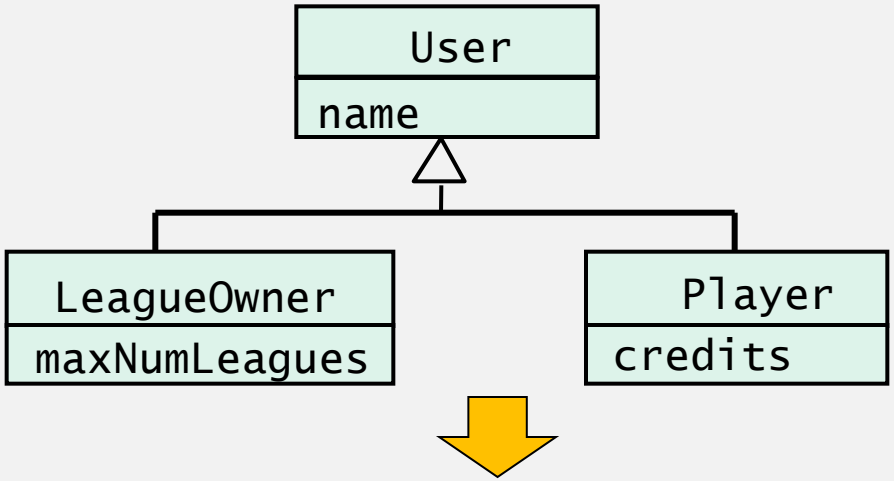
Realizing Inheritance

Relational databases do not support inheritance

Two possibilities to map an inheritance association 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 Vertical mapping)



User table

id	name	...	
56	zoe		LeagueOwner
79	john		Player

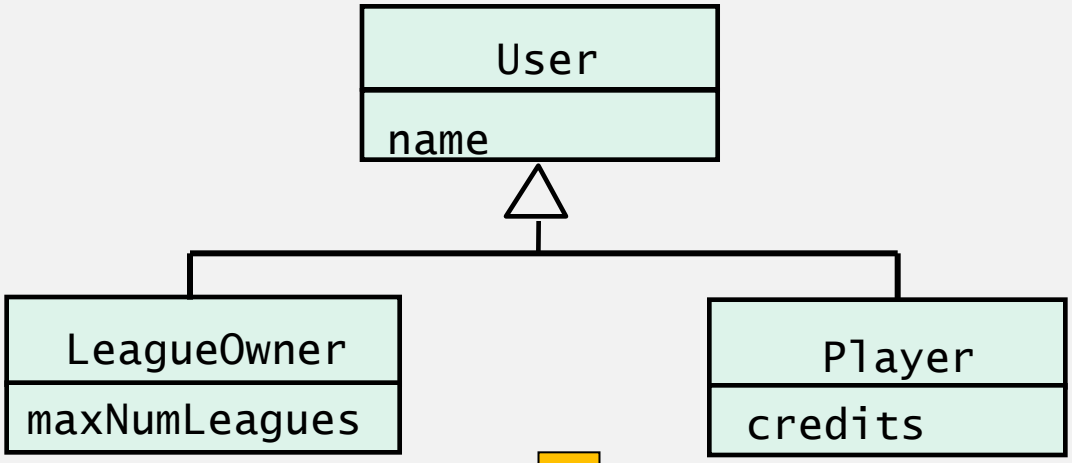
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Realizing inheritance by duplicating columns (Horizontal Mapping)



LeagueOwner table

id	name	maxNumLeagues	...
56	zoe	12	

Player table

id	name	credits	...
79	john	126	

Comparison: Separate Tables vs Duplicated Columns

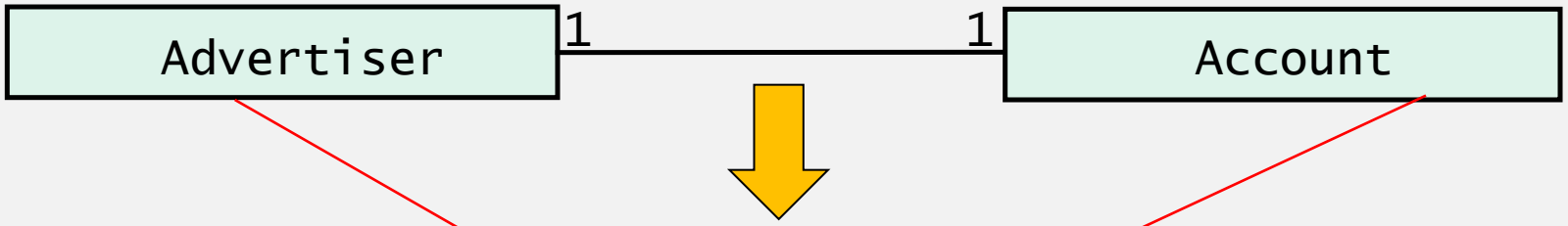
- The trade-off is between modifiability and response time
  - How likely is a change of the superclass?
  - What are the performance requirements for queries?
- Separate table mapping (Vertical mapping)
  - ☺ We can add attributes to the superclass easily by adding a column to the superclass table
  - ☹ Searching for the attributes of an object requires a join operation.
- Duplicated columns (Horizontal Mapping)
  - ☹ Modifying the database schema is more complex and error-prone
  - ☺ Individual objects are not fragmented across a number of tables, resulting in faster queries

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:

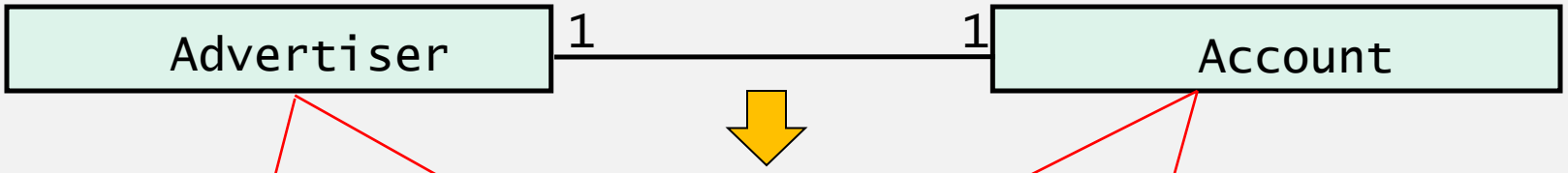


Source code after transformation:

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

Bidirectional one-to-one association

Object design model before transformation:



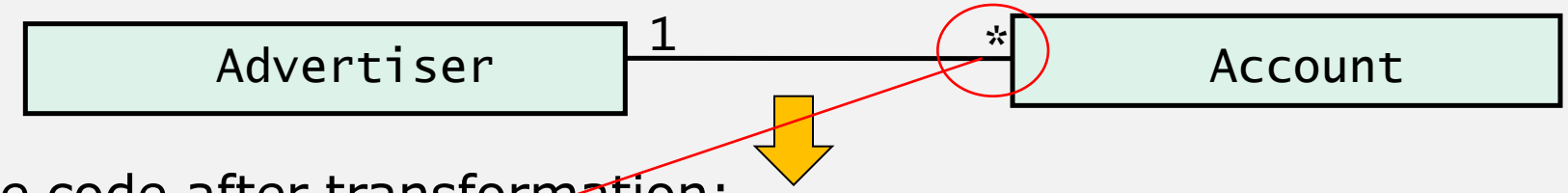
Source code after transformation:

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

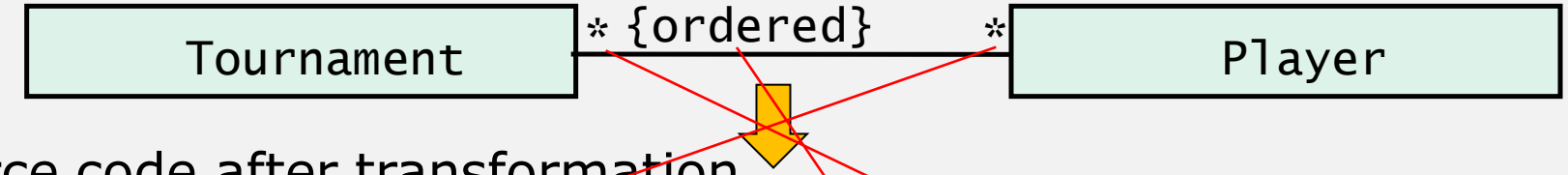
```
public class Account {
   private Advertiser owner;
   public void setOwner(Advertiser
   newOwner) {
     if (owner != newOwner) {
          Advertiser old = owner;
          owner = newOwner;
          if (newOwner != null)

     newOwner.addAccount(this);
          if (oldOwner != null)

     old.removeAccount(this);
     }
   }
}
```

Bidirectional many-to-many association

Object design model before transformation



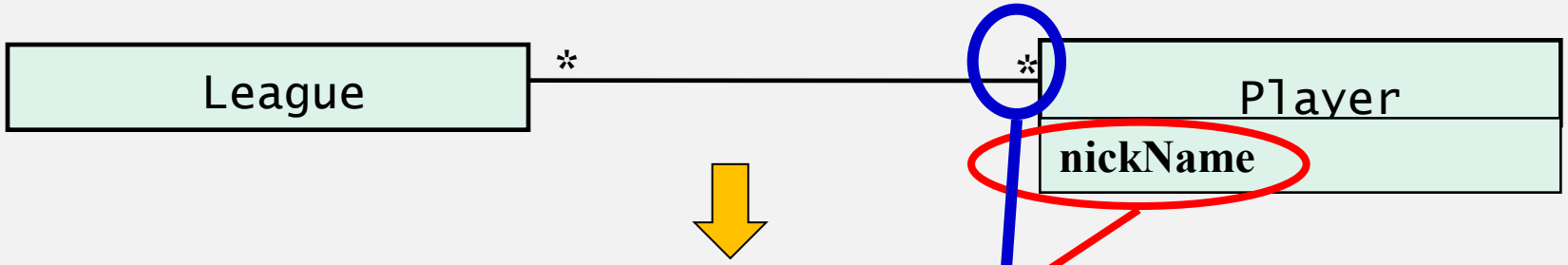
Source code after transformation

```
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);
      }
   }
}
```

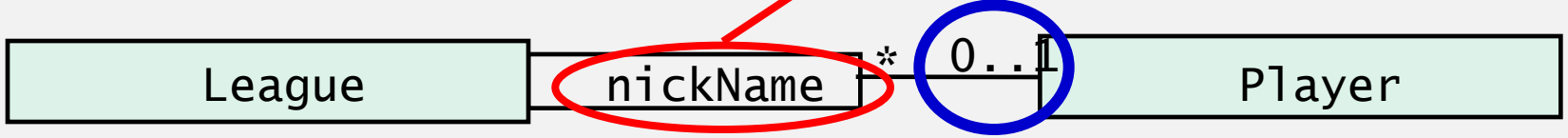
```
public class Player {
   private List tournaments;
   public Player() {
      tournaments = new
   ArrayList();
   }
   public void
   addTournament(Tournament t) {
      if
   (!tournaments.contains(t)) {
       tournaments.add(t);
        t.addPlayer(this);
      }
   }
}
```

Bidirectional qualified association

Object design model before model transformation

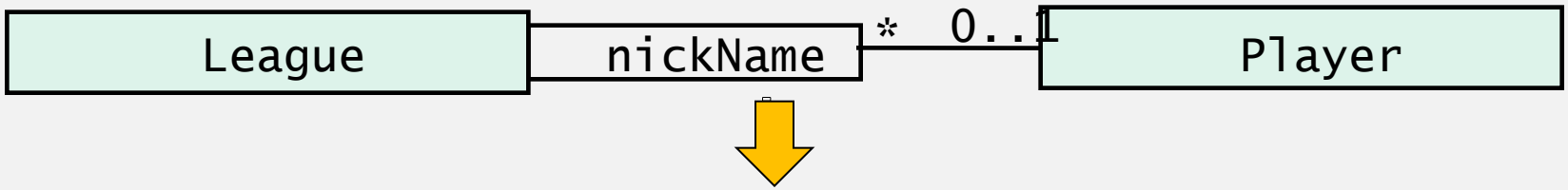


Object design model after model transformation



Bidirectional qualified association (2)

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);
      }
   }
}
```

Thank You