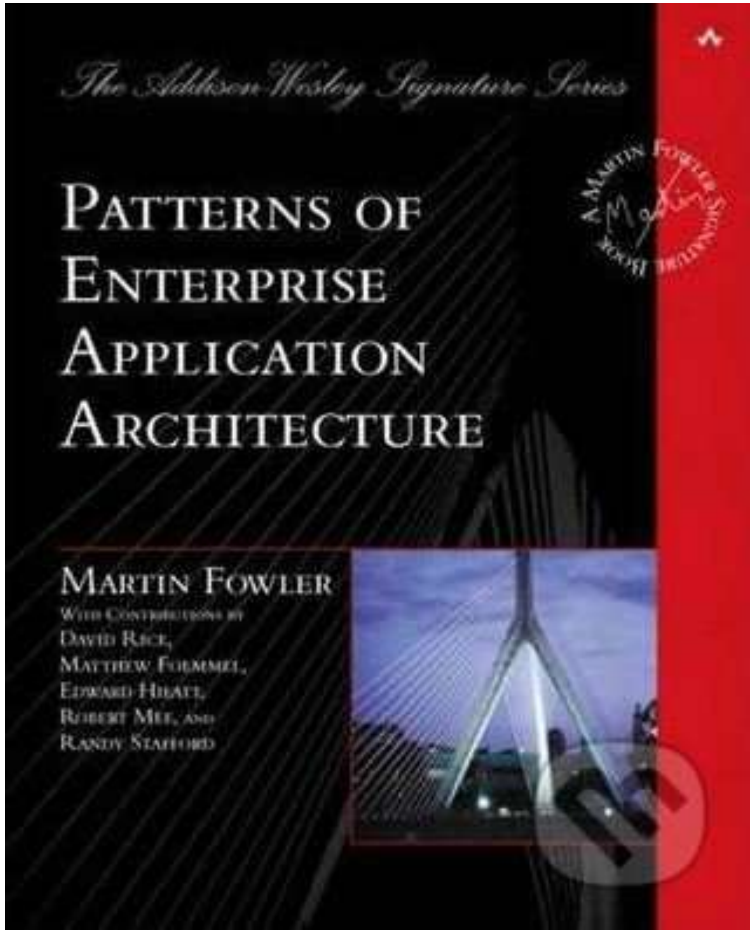
INHERITANCE

What this presentation …won’t be about

- Inheritance as one of the pillars of OOP
- Implementation of inheritance

What this presentation …will be about

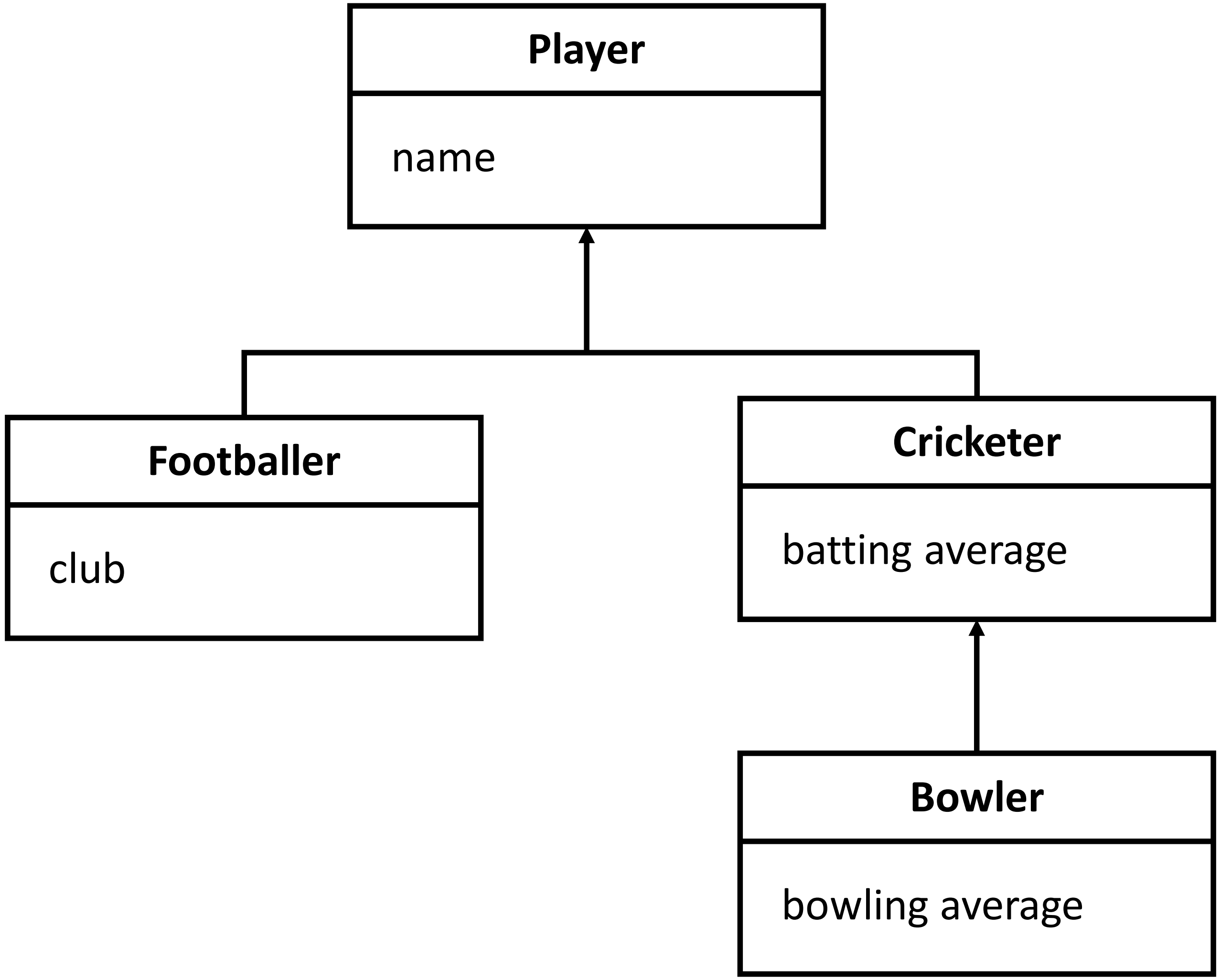
- Inheritance as a part of “Enterprise application”
- Inheritance in relation with relational database
- Martin Fowler: **Patterns of Enterprise Application Architecture (2002/2003)**

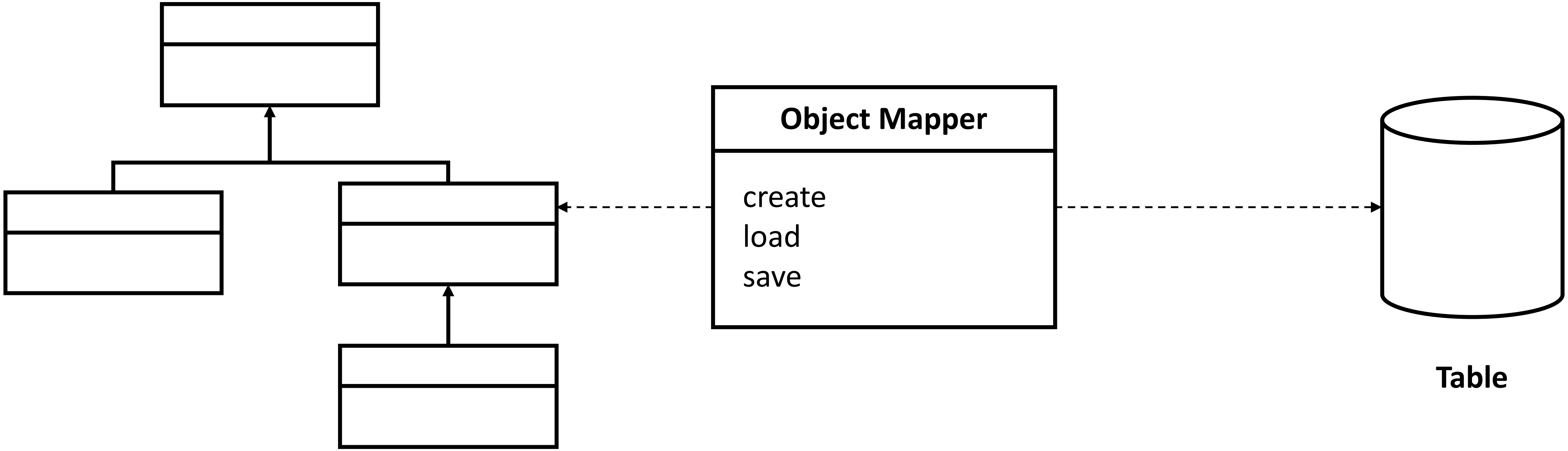

The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
A Martin Fowler Signature Book
Martin Fowler
With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford

Enterprise applications

“Enterprise software, also known as enterprise application software (EAS), is computer software used to satisfy the needs of an organization rather than individual users.!” ***Wikipedia***

- Persistent data
- A lot of data
- Concurrent data access
- Complex business (il)logic
- Integration with other enterprise systems

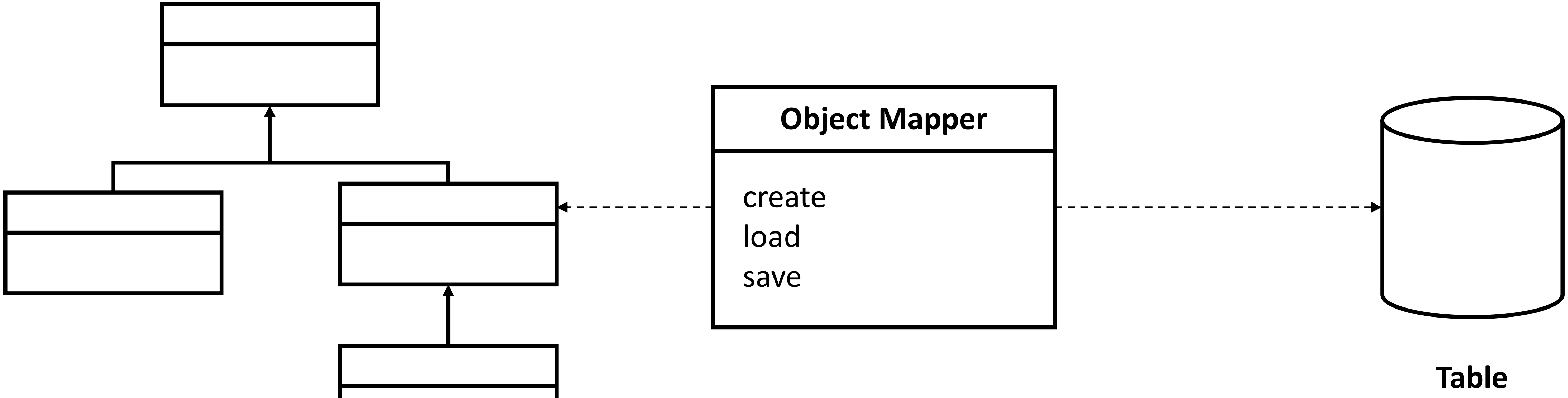

Player
name
Footballer
club
Cricketer
batting average
Bowler
bowling average



Object Mapper

create
load
save

Table



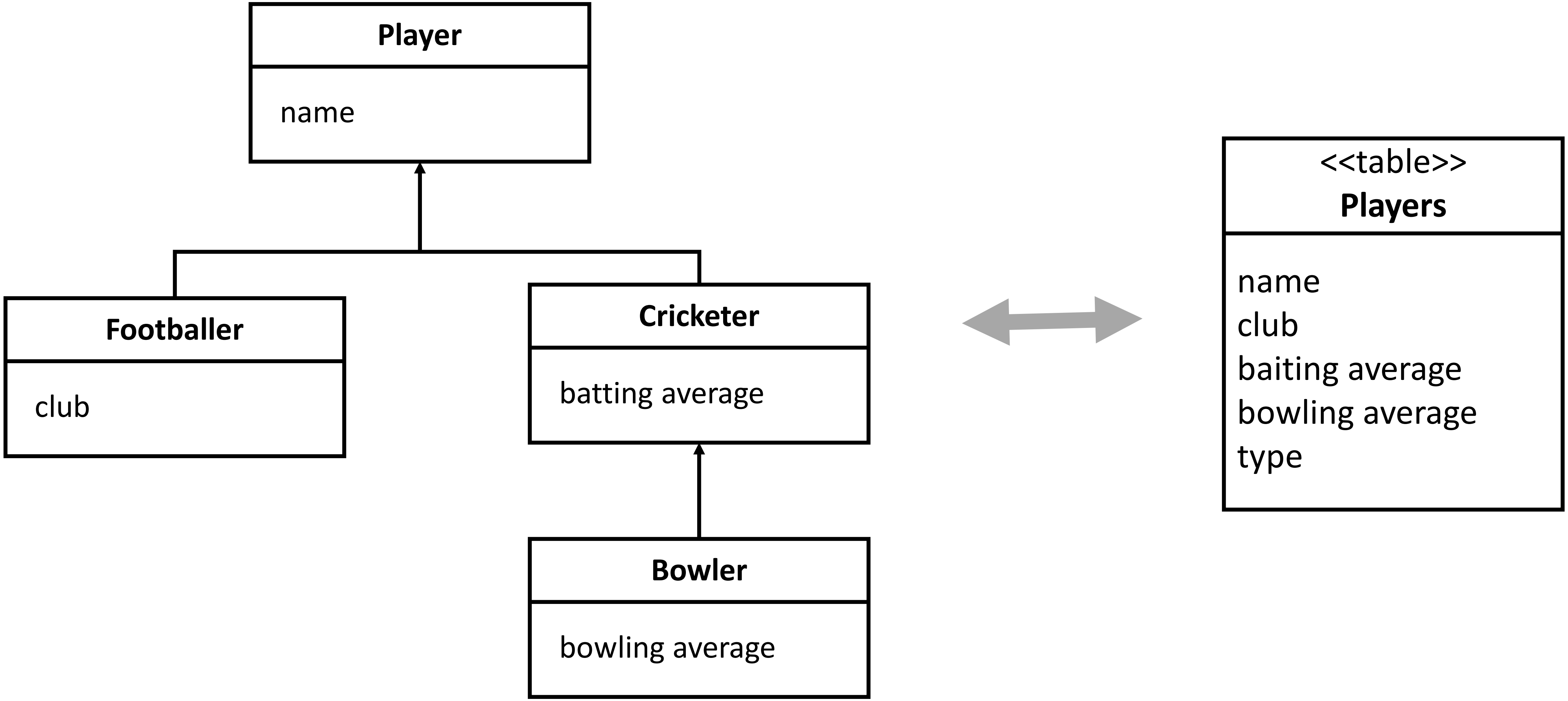
Relational databases don’t support inheritance

Object-Relational structural patterns

- Single table inheritance
- Class table inheritance
- Concrete table inheritance

- Inheritance mappers

Single table inheritance

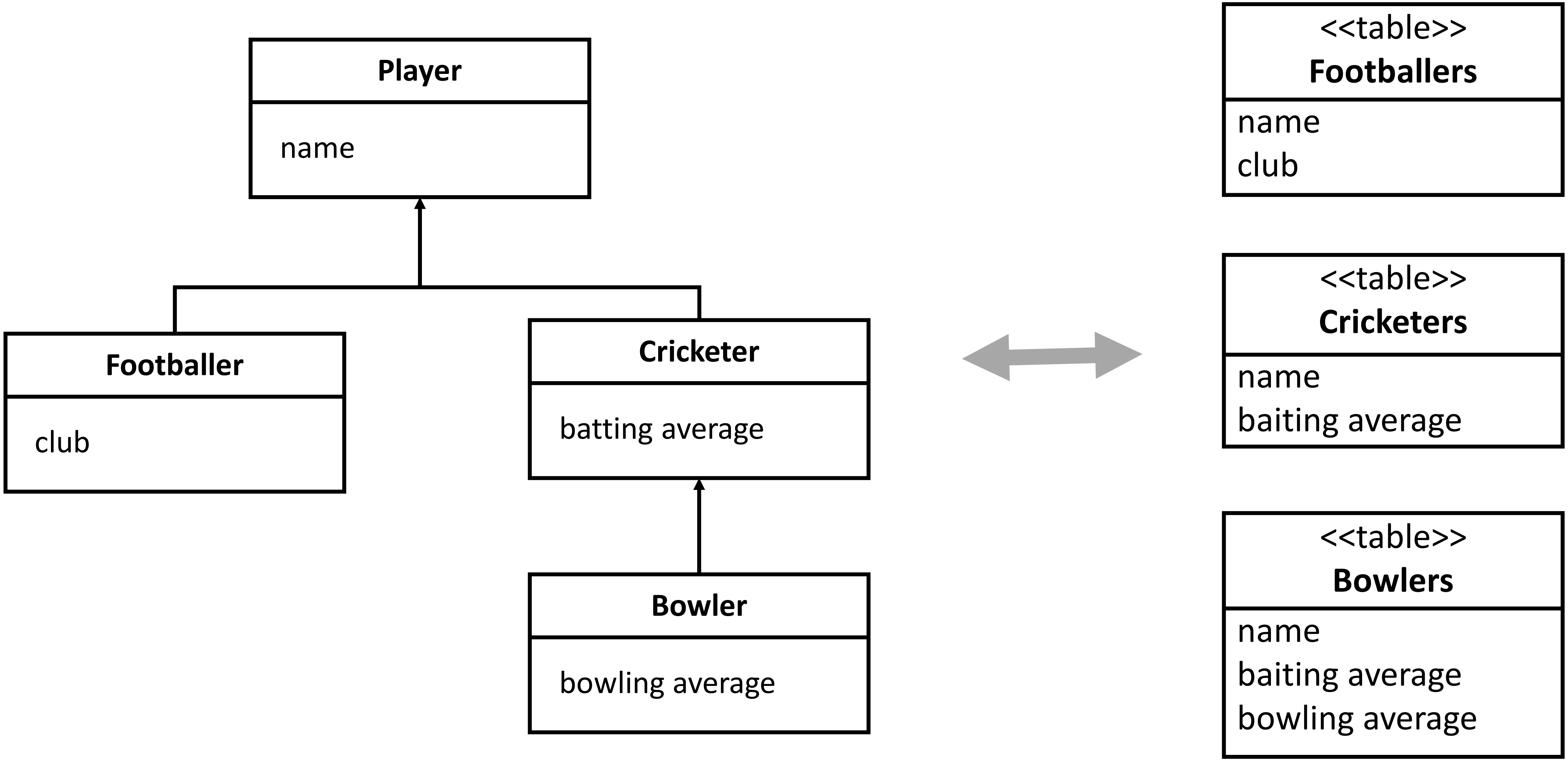


Single table inheritance

- Strengths of STI:
    - Single table in database
    - No joins in retrieving data
    - Moving fields up/down the hierarchy **does not** require database changes

- Weaknesses of STI:
    - Fields might not be relevant for everybody
    - Fields used only by some subclasses lead to wasted space
    - Single tables may end up being too large – may hurt performance
    - Single namespace for fields

Concrete table inheritance



Player

name

Footballer

club

Cricketer

batting average

Bowler

bowling average

<<table>>
Footballers

name
club

<<table>>
Cricketers

name
baiting average

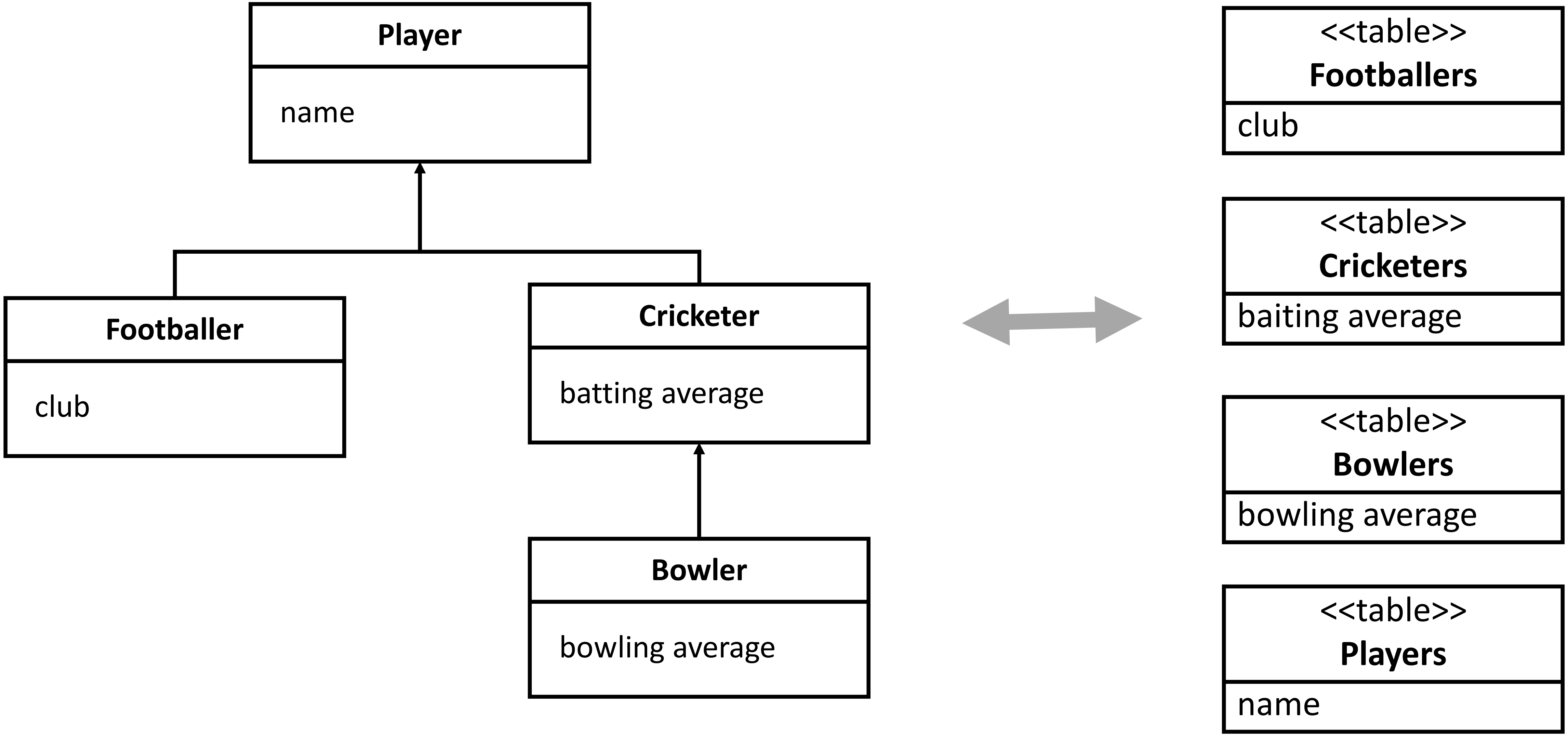
<<table>>
Bowlers

name
baiting average
bowling average

Concrete table inheritance

- Strengths of CTI:
  - No irrelevant fields
  - No joins when reading the data from concrete mappers
  - Each table is accessed only when concrete class is accessed
- Weakness of CTI:
  - Primary keys can be difficult to handle
  - Moving fields up/down the hierarchy **does** require database changes
  - Superclass fields are duplicated across the tables – changes in superclass mean changes in each table
  - A find on superclass forces you to check all the tables

Class table inheritance



Player

name

Footballer

club

Cricketer

batting average

Bowler

bowling average

<<table>>
Footballers

club

<<table>>
Cricketers

baiting average

<<table>>
Bowlers

bowling average

<<table>>
Players

name

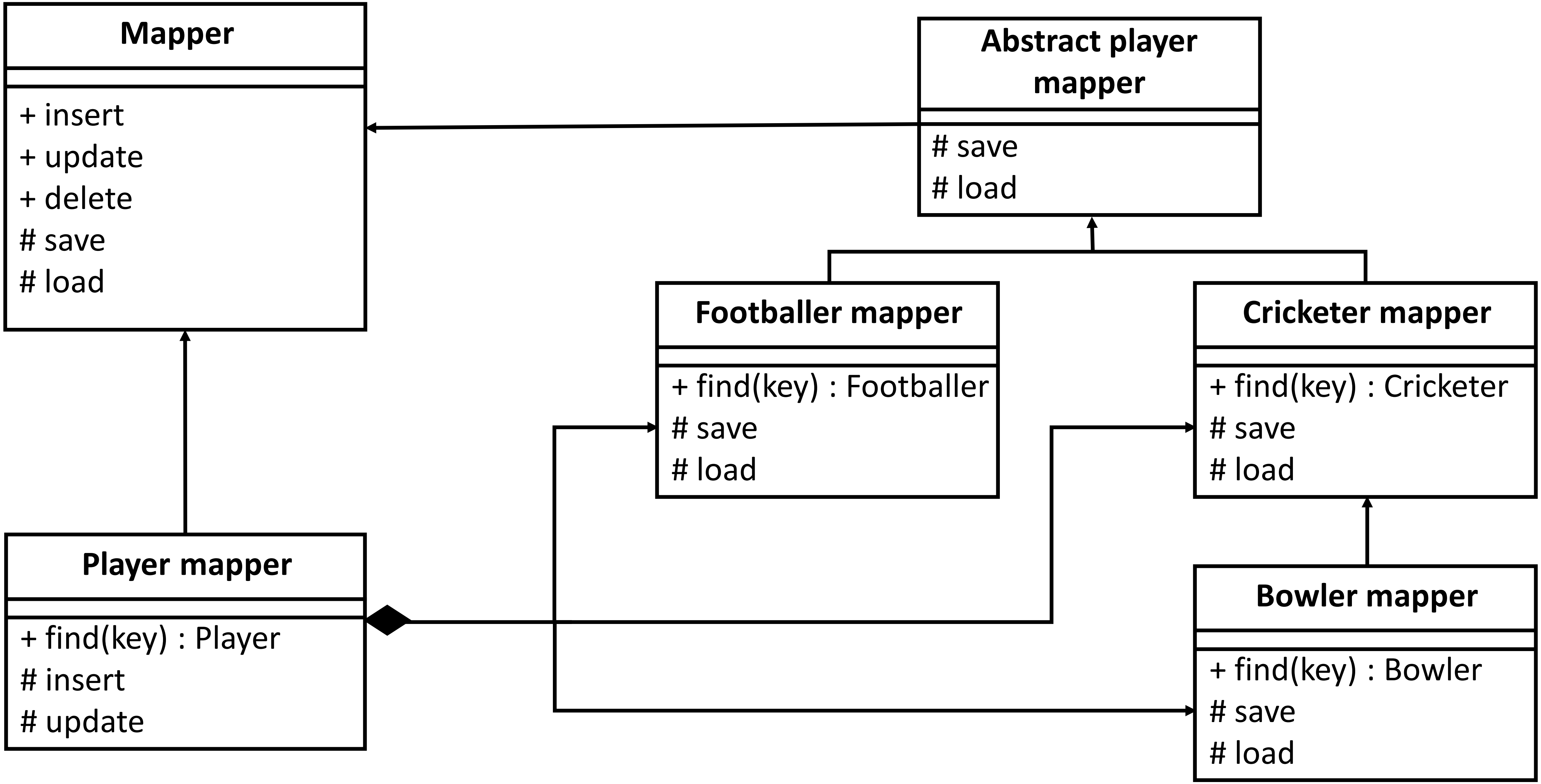
Class table inheritance

- Strengths of CTI:
  - All columns are relevant for every row – no wasted space
  - The relationship **database** x **domain model** is straightforward
- Weaknesses of CTI:
  - Need of accessing multiple tables in order to load object – joins or multiple queries
  - Moving fields up/down the hierarchy **does** require database changes
  - The supertype tables may become bottleneck because they have to be accessed frequently

Single table vs. Class table vs. Concrete table

- Trade-off between performance, duplicate data, readability,…
- Trio of patterns can coexist in a single hierarchy:
  - Concrete table inheritance + Single table inheritance
  - Class table inheritance + Concrete table inheritance
  - Possibly more…

Generic inheritance mapper



```
// The gateway's data property is a data set that can be loaded by a query.
class Mapper {
   ...
   protected DataTable table {
   get {return Gateway.Data.Tables[TableName];}
   }
   protected Gateway Gateway;
   abstract protected String TableName {get;}
}
// Since there is only one table, this can be defined by the abstract player
mapper.
class AbstractPlayerMapper : Mapper {
   ...
   protected override String TableName {
   get {return "Players";}
   }
}
```

```
// Each class needs a type code to help the mapper code figure out what kind of player it's
dealing with. The type code is defined on the superclass and implemented in the subclasses.
class AbstractPlayerMapper : Mapper {
   ...
   abstract public String TypeCode {get;}
}
class CricketerMapper : AbstractPlayerMapper {
   ...
   public const String TYPE_CODE = "C";
   public override String TypeCode {
   get {return TYPE_CODE;}
   }
}
```

```
// The player mapper has fields for each of the three concrete mapper classes.
class PlayerMapper : Mapper {
   ...
   private BowlerMapper bmapper;
   private CricketerMapper cmapper;
   private FootballerMapper fmapper;
   public PlayerMapper (Gateway gateway) : base (gateway) {
      bmapper = new BowlerMapper(Gateway);
      cmapper = new CricketerMapper(Gateway);
      fmapper = new FootballerMapper(Gateway);
   }
}
```

```
// Loading an Object from the Database
// Each concrete mapper class has a find method to get an object from the data.
class CricketerMapper : AbstractPlayerMapper {
   ...
   public Cricketer Find(long id) {
      return (Cricketer) AbstractFind(id);
   }
}
```

```
// This calls generic behavior to find an object.
class Mapper {
...
   protected DomainObject AbstractFind(long id) {
      DataRow row = FindRow(id);
      return (row == null) ? null : Find(row);
   }
   protected DataRow FindRow(long id) {
      String filter = String.Format("id = {0}", id);
      DataRow[] results = table.Select(filter);
      return (results.Length == 0) ? null : results[0];
   }
   public DomainObject Find (DataRow row) {
      DomainObject result = CreateDomainObject();
      Load(result, row);
    return result;
   }
   abstract protected DomainObject CreateDomainObject();
}
```

```
// I load the data into the new object with a series of load
methods, one on each class in the hierarchy.
class CricketerMapper : AbstractPlayerMapper {
   ...
   protected override void Load(DomainObject obj, DataRow row) {
      base.Load(obj,row);
      Cricketer cricketer = (Cricketer) obj;
      cricketer.battingAverage = (double)row["battingAverage"];
   }
}
class AbstractPlayerMapper : Mapper {
   ...
   protected override void Load(DomainObject obj, DataRow row) {
      base.Load(obj, row);
      Player player = (Player) obj;
      player.name = (String)row["name"];
   }
}
class Mapper {
   ...
   protected virtual void Load(DomainObject obj, DataRow row) {
      obj.Id = (int) row ["id"];
   }
}
```

```
// I can also load a player through the player mapper. It needs to
read the data and use the type code to determine which concrete
mapper to use.
class PlayerMapper : Mapper {
   ...
   public Player Find (long key) {
   DataRow row = FindRow(key);
   if (row == null) return null;
   else {
   String typecode = (String) row["type"];
   switch (typecode){
      case BowlerMapper.TYPE_CODE:
         return (Player) bmapper.Find(row);
      case CricketerMapper.TYPE_CODE:
         return (Player) cmapper.Find(row);
      case FootballerMapper.TYPE_CODE:
         return (Player) fmapper.Find(row);
      default:
         throw new Exception("unknown type");
   }
  }
}
```