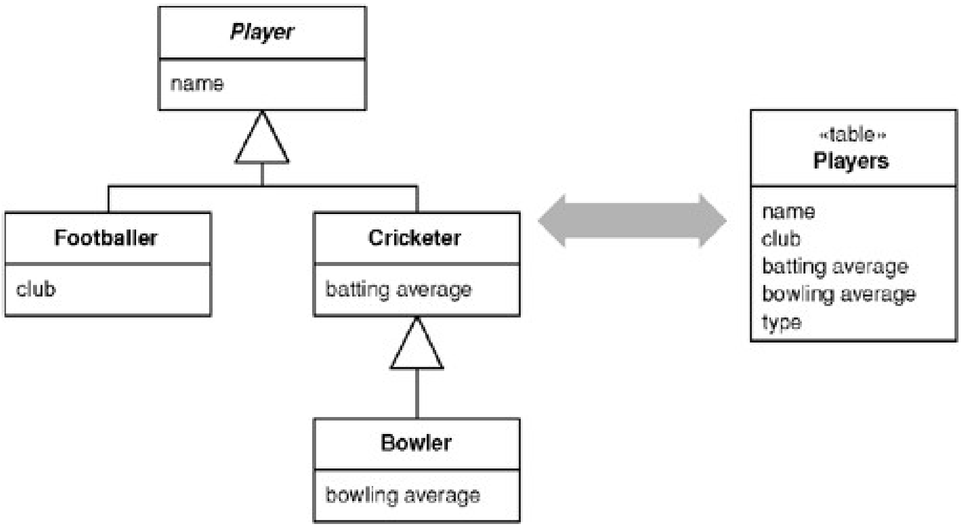
Single Table Inheritance

Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.



Relational databases don't support inheritance, so when mapping from objects to databases we have to consider how to represent our nice inheritance structures in relational tables. When mapping to a relational database, we try to minimize the joins that can quickly mount up when processing an inheritance structure in multiple tables. Single Table Inheritance maps all fields of all classes of an inheritance structure into a single table.

How It Works

In this inheritance mapping scheme we have one table that contains all the data for all the classes in the inheritance hierarchy. Each class stores the data that's relevant to it in one table row. Any columns in the database that aren't relevant are left empty. The basic mapping behavior follows the general scheme of Inheritance Mappers (302).

When loading an object into memory you need to know which class to instantiate. For this you have a field in the table that indicates which class should be used. This can be the name of the class or a code field. A code field needs to be interpreted by some code to map it to the relevant class. This code needs to be extended when a class is added to the hierarchy. If you embed the class name in the table you can just use it directly to instantiate an instance. The class name, however, will take up more space and may be less easy to process by those using the database table structure directly. As well it may more closely couple the class structure to the database schema.

In loading data you read the code first to figure out which subclass to instantiate. On saving the data the code needs be written out by the superclass in the hierarchy.

When to Use It

Single Table Inheritance is one of the options for mapping the fields in an inheritance hierarchy to a relational database. The alternatives are Class Table Inheritance (285) and Concrete Table Inheritance (293).

These are the strengths of Single Table Inheritance:

- There's only a single table to worry about on the database.
- There are no joins in retrieving data.
- Any refactoring that pushes fields up or down the hierarchy doesn't require you to change the database.

The weaknesses of Single Table Inheritance are

- Fields are sometimes relevant and sometimes not, which can be confusing to people using the tables directly.
- Columns used only by some subclasses lead to wasted space in the database. How much this is actually a problem depends on the specific data characteristics and how well the database compresses empty columns. Oracle, for example, is very efficient in trimming wasted space, particularly if you keep your optional columns to the right side of the database table. Each database has its own tricks for this.
- The single table may end up being too large, with many indexes and frequent locking, which may hurt performance. You can avoid this by having separate index tables that either list keys of rows that have a certain property or that copy a subset of fields relevant to an index.
- You only have a single namespace for fields, so you have to be sure that you don't use the same name for different fields. Compound names with the name of the class as a prefix or suffix help here.

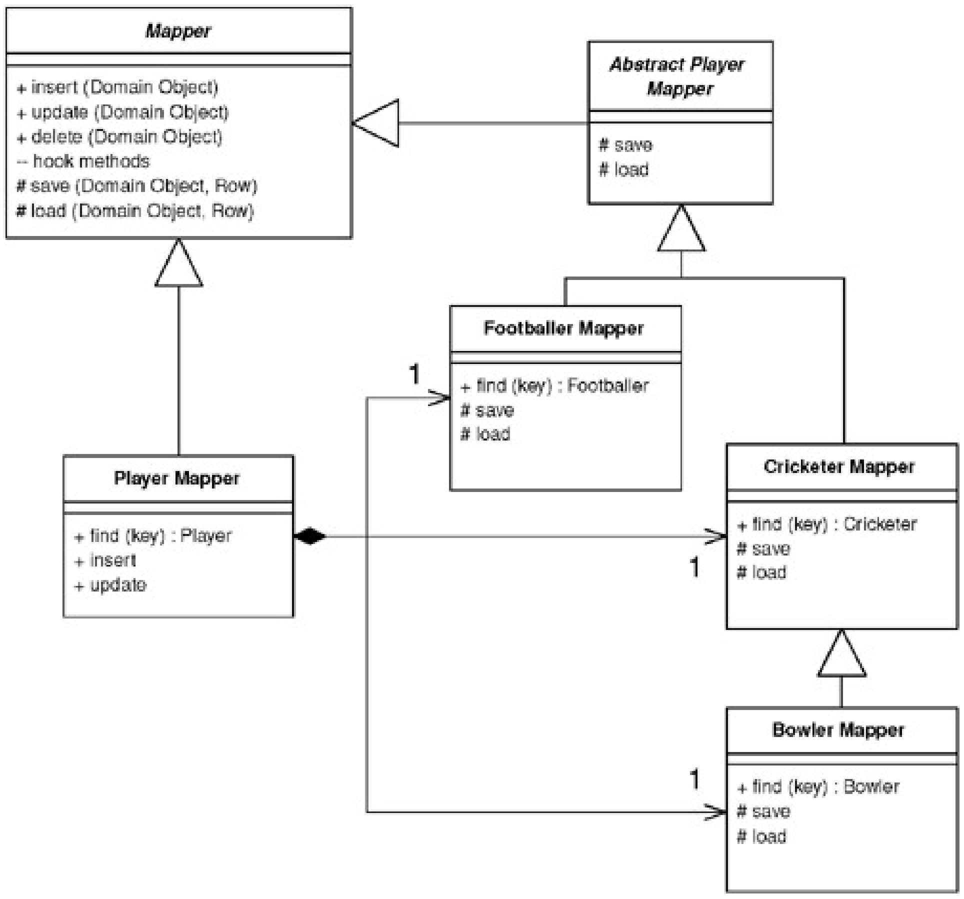
Rremember that you don't need to use one form of inheritance mapping for your whole hierarchy. It's perfectly fine to map half a dozen similar classes in a single table, as long as you use Concrete Table Inheritance (293) for any classes that have a lot of specific data.

Example: A Single Table for Players (C#)

Like the other inheritance examples, I've based this one on Inheritance Mappers (302), using the classes in Figure 12.8. Each mapper needs to be linked to a data table in an ADO.NET data set. This link can be made generically in the mapper superclass. 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;}
```

Figure 12.8. The generic class diagram of Inheritance Mappers (302).



Since there is only one table, this can be defined by the abstract player mapper.

```
class AbstractPlayerMapper...

      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...

      abstract public String TypeCode {get;}

class CricketerMapper...

      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...

      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...

      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();

class CricketerMapper...

      protected override DomainObject CreateDomainObject() {
         return new Cricketer();
      }
```

I load the data into the new object with a series of load methods, one on each class in the hierarchy.

```
class CricketerMapper...

      protected override void Load(DomainObject obj, DataRow row) {
         base.Load(obj,row);
         Cricketer cricketer = (Cricketer) obj;
         cricketer.battingAverage = (double)row["battingAverage"];
      }

class AbstractPlayerMapper...

      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...

      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");
            }
         }
      }
```

Updating an Object

The basic operation for updating is the same for all objects, so I can define the operation on the mapper superclass.

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
```

The save method is similar to the load method each class defines it to save the data it contains.

```
class CricketerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         base.Save(obj, row);
         Cricketer cricketer = (Cricketer) obj;
         row["battingAverage"] = cricketer.battingAverage;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         Player player = (Player) obj;
         row["name"] = player.name;
         row["type"] = TypeCode;
      }
```

The player mapper forwards to the appropriate concrete mapper.

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer)
            return fmapper;
         if (obj is Bowler)
            return bmapper;
         if (obj is Cricketer)
            return cmapper;
         throw new Exception("No mapper available");
      }
```

Inserting an Object

Insertions are similar to updates; the only real difference is that a new row needs to be made in the table before saving.

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save (arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }

class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

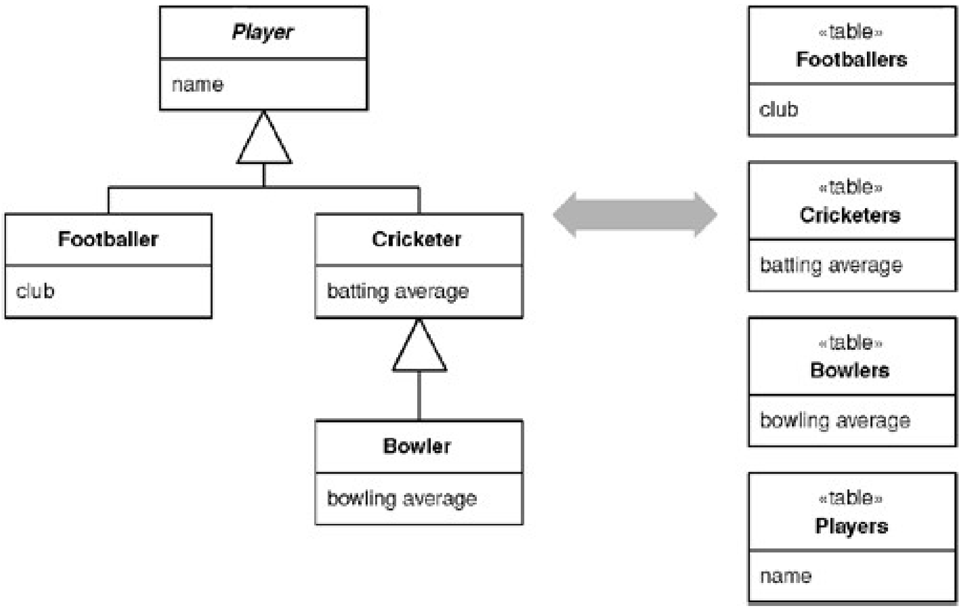
Deleting an Object

Deletes are pretty simple. They're defined at the abstract mapper level or in the player wrapper.

```
class Mapper...

      public virtual void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id);
         row.Delete();
      }

class PlayerMapper...

      public override void Delete (DomainObject obj) {
        MapperFor(obj).Delete(obj);
    }
```

Class Table Inheritance

Represents an inheritance hierarchy of classes with one table for each class.



A very visible aspect of the object-relational mismatch is the fact that relational databases don't support inheritance. You want database structures that map clearly to the objects and allow links anywhere in the inheritance structure. Class Table Inheritance supports this by using one database table per class in the inheritance structure.

How It Works

The straightforward thing about Class Table Inheritance is that it has one table per class in the domain model. The fields in the domain class map directly to fields in the corresponding tables. As with the other inheritance mappings the fundamental approach of Inheritance Mappers (302) applies.

One issue is how to link the corresponding rows of the database tables. A possible solution is to use a common primary key value so that, say, the row of key 101 in the footballers table and the row of key 101 in the players table correspond to the same domain object. Since the superclass table has a row for each row in the other tables, the primary keys are going to be unique across the tables if you use this scheme. An alternative is to let each table have its own primary keys and use foreign keys into the superclass table to tie the rows together.

The biggest implementation issue with Class Table Inheritance is how to bring the data back from multiple tables in an efficient manner. Obviously, making a call for each table isn't good since you have multiple calls to the database. You can avoid this by doing a join across the various component tables; however, joins for more than three or four tables tend to be slow because of the way databases do their optimizations.

On top of this is the problem that in any given query you often don't know exactly which tables to join. If you're looking for a footballer, you know to use the footballer table, but if you're looking for a group of players, which tables do you use? To join effectively when some tables have no data, you'll need to do an outer join, which is nonstandard and often slow. The alternative is to read the root table first and then use a code to figure out what tables to read next, but this involves multiple queries.

When to Use It

Class Table Inheritance, Single Table Inheritance (278) and Concrete Table Inheritance (293) are the three alternatives to consider for inheritance mapping.

The strengths of Class Table Inheritance are

- All columns are relevant for every row so tables are easier to understand and don't waste space.
- The relationship between the domain model and the database is very straightforward.

The weaknesses of Class Table Inheritance are

- You need to touch multiple tables to load an object, which means a join or multiple queries and sewing in memory.
- Any refactoring of fields up or down the hierarchy causes database changes.
- The supertype tables may become a bottleneck because they have to be accessed frequently.
- The high normalization may make it hard to understand for ad hoc queries.

You don't have to choose just one inheritance mapping pattern for one class hierarchy. You can use Class Table Inheritance for the classes at the top of the hierarchy and a bunch of Concrete Table Inheritance (293) for those lower down.

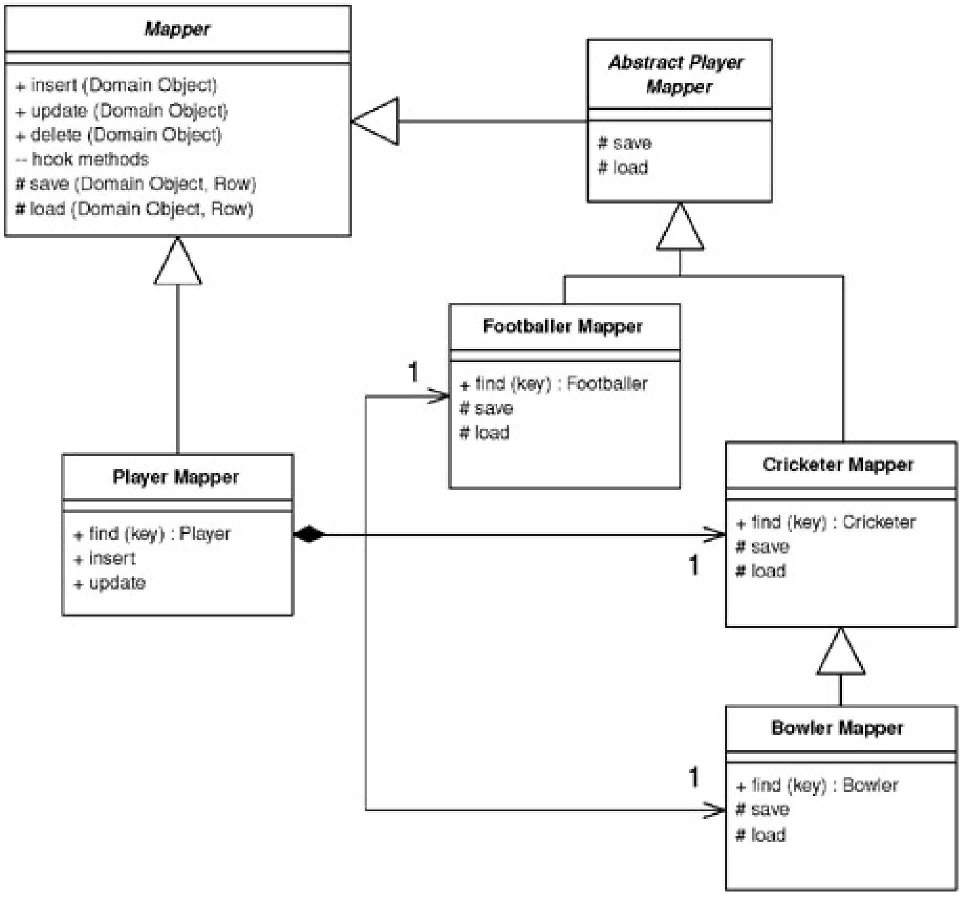
Further Reading

A number of IBM texts refer to this pattern as Root-Leaf Mapping [Brown et al.].

Example: Players and Their Kin (C#)

Here's an implementation for the sketch. Again I'll follow the familiar (if perhaps a little tedious) theme of players and the like, using Inheritance Mappers (302) (Figure 12.9).

Figure 12.9. The generic class diagram of Inheritance Mappers (302).



Each class needs to define the table that holds its data and a type code for it.

```
class AbstractPlayerMapper...

      abstract public String TypeCode {get;}
      protected static String TABLENAME = "Players";

class FootballerMapper...

      public override String TypeCode {
         get {return "F";}
      }
      protected new static String TABLENAME = "Footballers";
```

Unlike the other inheritance examples, this one doesn't have a overridden table name because we have to have the table name for this class even when the instance is an instance of the subclass.

Loading an Object

If you've been reading the other mappings, you know the first step is the find method on the concrete mappers.

```
class FootballerMapper...

      public Footballer Find(long id) {
         return (Footballer) AbstractFind (id, TABLENAME);
      }
```

The abstract find method looks for a row matching the key and, if successful, creates a domain object and calls the load method on it.

```
class Mapper...

      public DomainObject AbstractFind(long id, String tablename) {
         DataRow row = FindRow (id, tableFor(tablename));
         if (row == null) return null;
         else {
            DomainObject result = CreateDomainObject();
            result.Id = id;
            Load(result);
            return result;
         }
      }
      protected DataTable tableFor(String name) {
         return Gateway.Data.Tables[name];
      }
      protected DataRow FindRow(long id, DataTable table) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      protected DataRow FindRow (long id, String tablename) {
         return FindRow(id, tableFor(tablename));
      }
      protected abstract DomainObject CreateDomainObject();

class FootballerMapper...

      protected override DomainObject CreateDomainObject(){
         return new Footballer();
      }
```

There's one load method for each class which loads the data defined by that class.

```
class FootballerMapper...

      protected override void Load(DomainObject obj) {
         base.Load(obj);
         DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
         Footballer footballer = (Footballer) obj;
         footballer.club = (String)row["club"];
      }

class AbstractPlayerMapper...

      protected override void Load(DomainObject obj) {
         DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
         Player player = (Player) obj;
         player.name = (String)row["name"];
      }
```

As with the other sample code, but more noticeably in this case, I'm relying on the fact that the ADO.NET data set has brought the data from the database and cached it into memory. This allows me to make several accesses to the table-based data structure without a high performance cost. If you're going directly to the database, you'll need to reduce that load. For this example you might do this by creating a join across all the tables and manipulating it.

The player mapper determines which kind of player it has to find and then delegates the correct concrete mapper.

```
class PlayerMapper...

      public Player Find (long key) {
         DataRow row = FindRow(key, tableFor(TABLENAME));
         if (row == null) return null;
         else {
            String typecode = (String) row["type"];
            if (typecode == bmapper.TypeCode)
               return bmapper.Find(key);
            if (typecode == cmapper.TypeCode)
               return cmapper.Find(key);
            if (typecode == fmapper.TypeCode)
               return fmapper.Find(key);
            throw new Exception("unknown type");
         }
      }
      protected static String TABLENAME = "Players";
```

Updating an Object

The update method appears on the mapper superclass

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg);
      }
```

It's implemented through a series of save methods, one for each class in the hierarchy.

```
class FootballerMapper...

      protected override void Save(DomainObject obj) {
         base.Save(obj);
         DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
         Footballer footballer = (Footballer) obj;
         row["club"] = footballer.club;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj) {
         DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
         Player player = (Player) obj;
         row["name"] = player.name;
         row["type"] = TypeCode;
      }
```

The player mapper's update method overrides the general method to forward to the correct concrete mapper.

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer)
            return fmapper;
         if (obj is Bowler)
            return bmapper;
         if (obj is Cricketer)
            return cmapper;
         throw new Exception("No mapper available");
      }
```

Inserting an Object

The method for inserting an object is declared on the mapper superclass. It has two stages: creating new database rows and then using the save methods to update these blank rows with the necessary data.

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg);
      }
```

Each class inserts a row into its table.

```
class FootballerMapper...

      protected override void AddRow (DomainObject obj) {
         base.AddRow(obj);
         InsertRow (obj, tableFor(TABLENAME));
      }

class AbstractPlayerMapper...

      protected override void AddRow (DomainObject obj) {
         InsertRow (obj, tableFor(TABLENAME));
      }

class Mapper...

      abstract protected void AddRow (DomainObject obj);
      protected virtual void InsertRow (DomainObject arg, DataTable table) {
         DataRow row = table.NewRow();
         row["id"] = arg.Id;
         table.Rows.Add(row);
      }
```

The player mapper delegates to the appropriate concrete mapper.

```
class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

Deleting an Object

To delete an object, each class deletes a row from the corresponding table in the database.

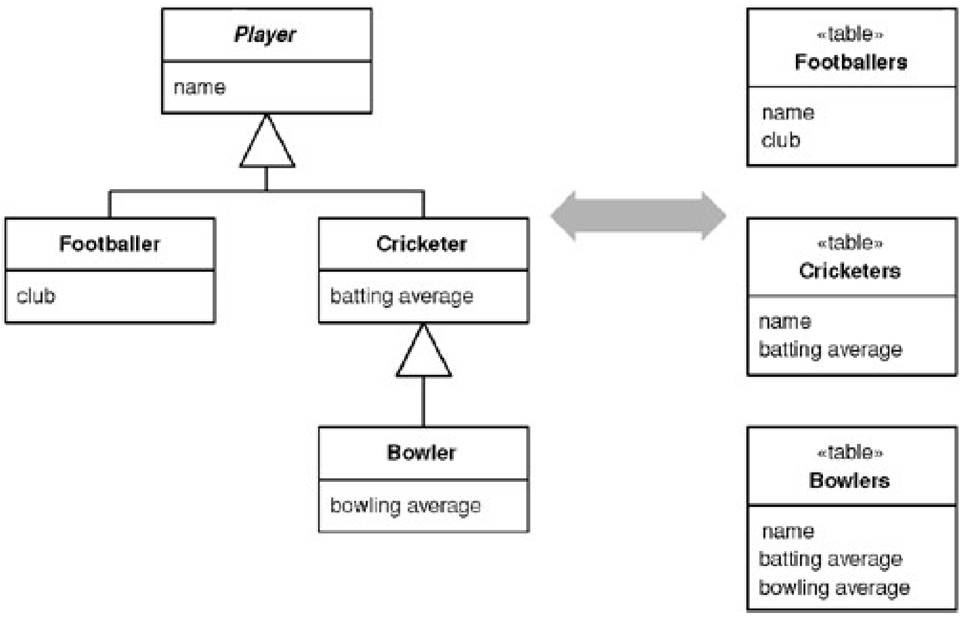
```
class FootballerMapper...

      public override void Delete(DomainObject obj) {
         base.Delete(obj);
         DataRow row = FindRow(obj.Id, TABLENAME);
         row.Delete();
      }
class AbstractPlayerMapper...

      public override void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id, tableFor(TABLENAME));
         row.Delete();
      }

class Mapper...

      public abstract void Delete(DomainObject obj);
```

The player mapper again wimps out of all the hard work and just delegates to the concrete mapper.

```
class PlayerMapper...

      override public void Delete(DomainObject obj) {
         MapperFor(obj).Delete(obj);
      }
```

Concrete Table Inheritance

Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.



As any object purist will tell you, relational databases don't support inheritance a fact that complicates object-relational mapping. Thinking of tables from an object instance point of view, a sensible route is to take each object in memory and map it to a single database row. This implies Concrete Table Inheritance, where there's a table for each concrete class in the inheritance hierarchy.

I'll confess to having had some difficulty naming this pattern. Most people think of it as leaf oriented since you usually have one table per leaf class in a hierarchy. Following that logic, I could call this pattern leaf table inheritance, and the term "leaf" is often used for this pattern. Strictly, however, a concrete class that isn't a leaf usually gets a table as well, so I decided to go with the more correct, if less intuitive term.

How It Works

Concrete Table Inheritance uses one database table for each concrete class in the hierarchy. Each table contains columns for the concrete class and all its ancestors, so any fields in a superclass are duplicated across the tables of the subclasses. As with all of these inheritance schemes the basic behavior uses Inheritance Mappers (302).

You need to pay attention to the keys with this pattern. Punningly, the key thing is to ensure that keys are unique not just to a table but to all the tables from a hierarchy. A classic example of where you need this is if you have a collection of players and you're using Identity Field (216) with table-wide keys. If keys can be duplicated between the tables that map the concrete classes, you'll get multiple rows for a particular key value. Thus, you thus need a key allocation system that keeps track of key usage across tables; also, you can't rely on the database's primary key uniqueness mechanism.

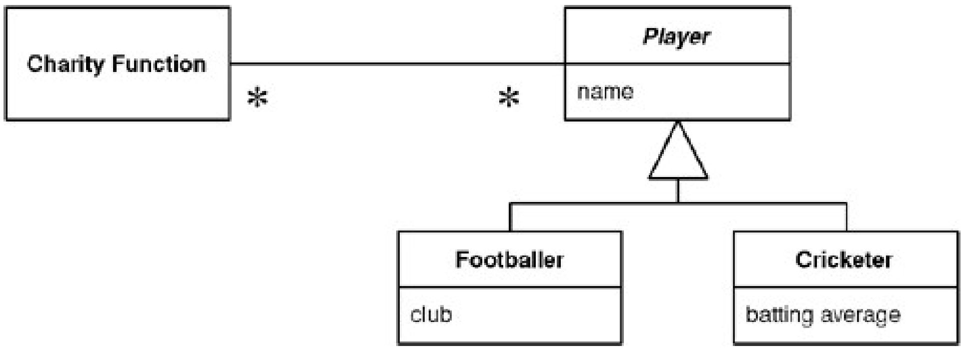
This becomes particularly awkward if you're hooking up to databases used by other systems. In many of these cases you can't guarantee key uniqueness across tables. In this situation you either avoid using superclass fields or use a compound key that involves a table identifier.

You can get around some of this by not having fields that are typed to the superclass, but obviously that compromises the object model. As alternative is to have accessors for the supertype in the interface but to use several private fields for each concrete type in the implementation. The interface then combines values from the private fields. If the public interface is a single value, it picks whichever of the private values aren't null. If the public interface is a collection value, it replies with the union of values from the implementation fields.

For compound keys you can use a special key object as your ID field for Identity Field (216). This key uses both the primary key of the table and the table name to determine uniqueness.

Related to this are problems with referential integrity in the database. Consider an object model like Figure 12.10. To implement referential integrity you need a link table that contains foreign key columns for the charity function and for the player. The problem is that there's no table for the player, so you can't put together a referential integrity constraint for the foreign key field that takes either footballers or cricketers. Your choice is to ignore referential integrity or use multiple link tables, one for each of the actual tables in the database. On top of this you have problems if you can't guarantee key uniqueness.

Figure 12.10. A model that causes referential integrity problems for Concrete Table Inheritance.



If you're searching for players with a select statement, you need to look at all tables to see which ones contain the appropriate value. This means using multiple queries or using an outer join, both of which are bad for performance. You don't suffer the performance hit when you know the class you need, but you do have to use the concrete class to improve performance.

This pattern is often referred to as along the lines of leaf table inheritance. Some people prefer a variation where you have one table per leaf class instead of one table per concrete class. If you don't have any concrete superclasses in the hierarchy, this ends up as the same thing. Even if you do have concrete superclasses the difference is pretty minor.

When to Use It

When figuring out how to map inheritance, Concrete Table Inheritance, Class Table Inheritance (285), and Single Table Inheritance (278) are the alternatives.

The strengths of Concrete Table Inheritance are:

- Each table is self-contained and has no irrelevant fields. As a result it makes good sense when used by other applications that aren't using the objects.
- There are no joins to do when reading the data from the concrete mappers.
- Each table is accessed only when that class is accessed, which can spread the access load.

The weaknesses of Concrete Table Inheritance are:

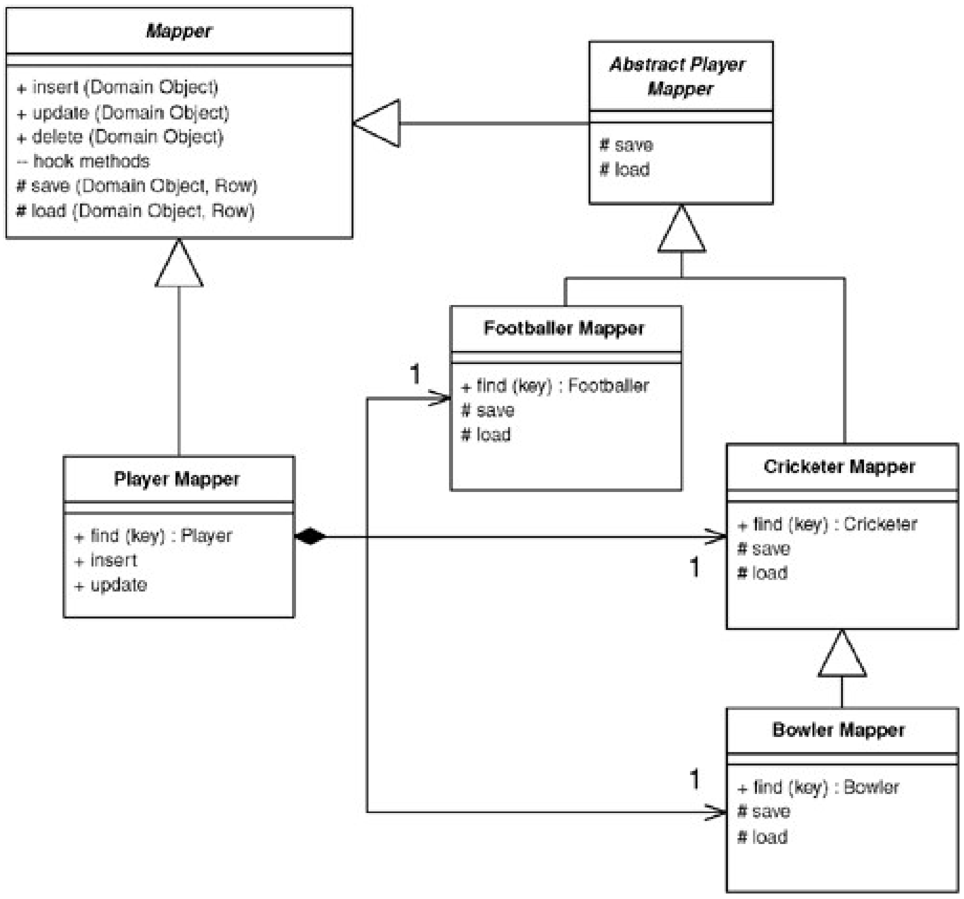
- Primary keys can be difficult to handle.
- You can't enforce database relationships to abstract classes.
- If the fields on the domain classes are pushed up or down the hierarchy, you have to alter the table definitions. You don't have to do as much alteration as with Class Table Inheritance (285), but you can't ignore this as you can with Single Table Inheritance (278).
- If a superclass field changes, you need to change each table that has this field because the superclass fields are duplicated across the tables.
- A find on the superclass forces you to check all the tables, which leads to multiple database accesses (or a weird join).

Remember that the trio of inheritance patterns can coexist in a single hierarchy. So you might use Concrete Table Inheritance for one or two subclasses and Single Table Inheritance (278) for the rest.

Example: Concrete Players (C#)

Here I'll show you an implementation for the sketch. As with all inheritance examples in this chapter, I'm using the basic design of classes from Inheritance Mappers (302), shown in Figure 12.11.

Figure 12.11. The generic class diagram of Inheritance Mappers (302).



Each mapper is linked to the database table that's the source of the data. In ADO.NET a data set holds the data table.

```
class Mapper...

      public Gateway Gateway;
      private IDictionary identityMap = new Hashtable();
      public Mapper (Gateway gateway) {
         this.Gateway = gateway;
      }
      private DataTable table {
         get {return Gateway.Data.Tables[TableName];}
      }
      abstract public String TableName {get;}
```

The gateway class holds the data set within its data property. The data can be loaded up by supplying suitable queries.

```
class Gateway...

      public DataSet Data = new DataSet();
```

Each concrete mapper needs to define the name of the table that holds its data.

```
class CricketerMapper...

      public override String TableName {
         get {return "Cricketers";}
      }
```

The player mapper has fields for each concrete mapper.

```
class PlayerMapper...

      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 that returns an object given a key value.

```
class CricketerMapper...

      public Cricketer Find(long id) {
         return (Cricketer) AbstractFind(id);
      }
```

The abstract behavior on the superclass finds the right database row for the ID, creates a new domain object of the correct type, and uses the load method to load it up (I'll describe the load in a moment).

```
class Mapper...

      public DomainObject AbstractFind(long id) {
         DataRow row = FindRow(id);
         if (row == null) return null;
         else {
            DomainObject result = CreateDomainObject();
            Load(result, row);
            return result;
         }
      }
      private DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         if (results.Length == 0) return null;
         else return results[0];
      }
      protected abstract DomainObject CreateDomainObject();

class CricketerMapper...

      protected override DomainObject CreateDomainObject(){
         return new Cricketer();
      }
```

The actual loading of data from the database is done by the load method, or rather by several load methods: one each for the mapper class and for all its superclasses.

```
class CricketerMapper...
```

```
      protected override void Load(DomainObject obj, DataRow row) {
         base.Load(obj,row);
         Cricketer cricketer = (Cricketer) obj;
         cricketer.battingAverage = (double)row["battingAverage"];
      }
class AbstractPlayerMapper...

      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"];
      }
```

This is the logic for finding an object using a mapper for a concrete class. You can also use a mapper for the superclass: the player mapper, which it needs to find an object from whatever table it's living in. Since all the data is already in memory in the data set, I can do this like so:

```
class PlayerMapper...

      public Player Find (long key) {
         Player result;
         result = fmapper.Find(key);
         if (result != null) return result;
         result = bmapper.Find(key);
         if (result != null) return result;
         result = cmapper.Find(key);
         if (result != null) return result;
         return null;
      }
```

Remember, this is reasonable only because the data is already in memory. If you need to go to the database three times (or more for more subclasses) this will be slow. It may help to do a join across all the concrete tables, which will allow you to access the data in one database call. However, large joins are often slow in their own right, so you'll need to do some benchmarks with your own application to find out what works and what doesn't. Also, this will be an outer join, and as well as slowing the syntax it's nonportable and often cryptic.

Updating an Object

The update method can be defined on the mapper superclass.

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
```

Similar to loading, we use a sequence of save methods for each mapper class.

```
class CricketerMapper...
```

```
      protected override void Save(DomainObject obj, DataRow row) {
         base.Save(obj, row);
         Cricketer cricketer = (Cricketer) obj;
         row["battingAverage"] = cricketer.battingAverage;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj, DataRow row) {
         Player player = (Player) obj;
         row["name"] = player.name;
      }
```

The player mapper needs to find the correct concrete mapper to use and then delegate the update call.

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer)
            return fmapper;
         if (obj is Bowler)
            return bmapper;
         if (obj is Cricketer)
            return cmapper;
         throw new Exception("No mapper available");
      }
```

Inserting an Object

Insertion is a variation on updating. The extra behavior is creating the new row, which can be done on the superclass.

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save (arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }
```

Again, the player class delegates to the appropriate mapper.

```
class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

Deleting an Object

Deletion is very straightforward. As before, we have a method defined on the superclass:

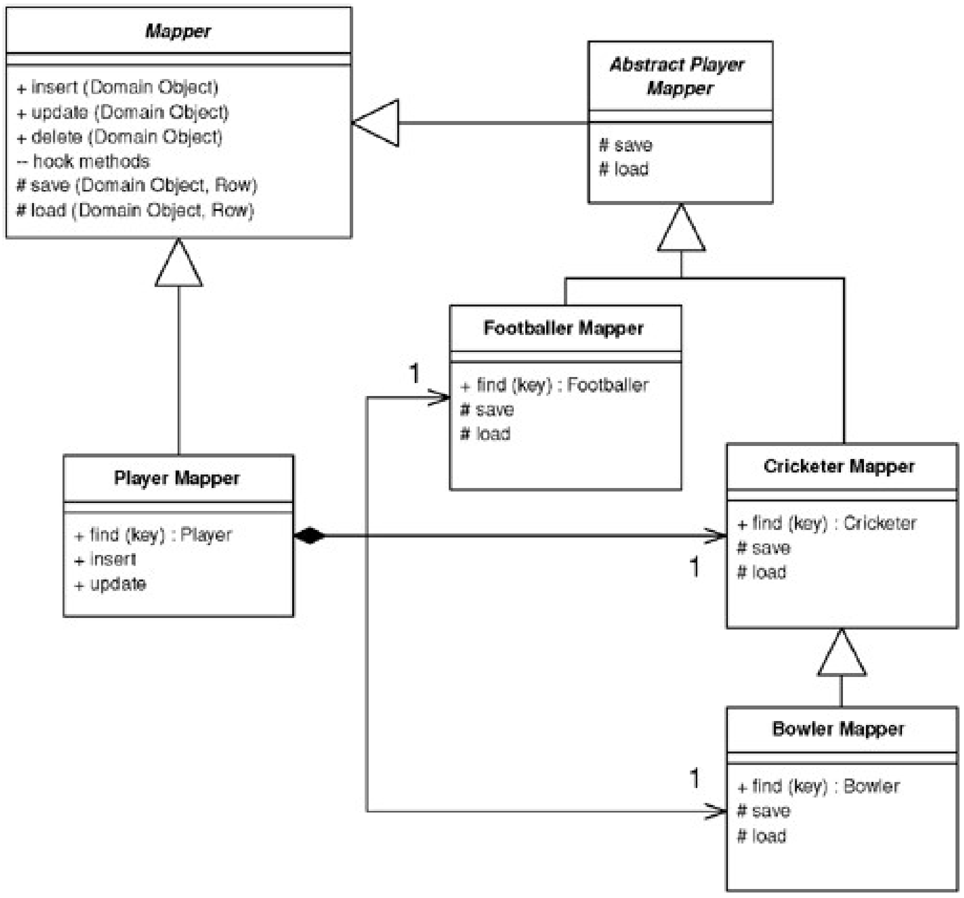
```
class Mapper...

      public virtual void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id);
         row.Delete();
      }
```

and a delegating method on the player mapper.

```
class PlayerMapper...

      public override void Delete (DomainObject obj) {
         MapperFor(obj).Delete(obj);
      }
```

Inheritance Mappers

A structure to organize database mappers that handle inheritance hierarchies.



When you map from an object-oriented inheritance hierarchy in memory to a relational database you have to minimize the amount of code needed to save and load the data to the database. You also want to provide both abstract and concrete mapping behavior that allows you to save or load a superclass or a subclass.

Although the details of this behavior vary with your inheritance mapping scheme (Single Table Inheritance (278), Class Table Inheritance (285), and Concrete Table Inheritance (293)) the general structure works the same for all of them.

How It Works

You can organize the mappers with a hierarchy so that each domain class has a mapper that saves and loads the data for that domain class. This way you have one point where you can change the mapping. This approach works well for concrete mappers that know how to map the concrete objects in the hierarchy. There are times, however, where you also need mappers for the abstract classes. These can be implemented with mappers that are actually outside of the basic hierarchy but delegate to the appropriate concrete mappers.

To best explain how this works, I'll start with the concrete mappers. In the sketch the concrete mappers are the mappers for footballer, cricketer, and bowler. Their basic behavior includes the find, insert, update, and delete operations.

The find methods are declared on the concrete subclasses because they will return a concrete class. Thus, the find method on BowlerMapper should return a bowler, not an abstract class. Common OO languages can't let you change the declared return type of a method, so it's not possible to inherit the find operation and still declare a specific return type. You can, of course, return an abstract type, but that forces the user of the class to downcast which is best to avoid. (A language with dynamic typing doesn't have this problem.)

The basic behavior of the find method is to find the appropriate row in the database, instantiate an object of the correct type (a decision that's made by the subclass), and then load the object with data from the database. The load method is implemented by each mapper in the hierarchy which loads the behavior for its corresponding domain object. This means that the bowler mapper's load method loads the data specific to the bowler class and calls the superclass method to load the data specific to the cricketer, which calls its superclass method, and so on.

The insert and update methods operate in a similar way using a save method. Here you can define the interface on the superclass indeed, on a Layer Supertype (475). The insert method creates a new row and then saves the data from the domain object using the save hook methods. The update method just saves the data, also using the save hook methods. These methods operate similarly to the load hook methods, with each class storing its specific data and calling the superclass save method.

This scheme makes it easy to write the appropriate mappers to save the information needed for a particular part of the hierarchy. The next step is to support loading and saving an abstract class in this example, a player. While a first thought is to put appropriate methods on the superclass mapper, that actually gets awkward. While concrete mapper classes can just use the abstract mapper's insert and update methods, the player mapper's insert and update need to override these to call a concrete mapper instead. The result is one of those combinations of generalization and composition that twist your brain cells into a knot.

I prefer to separate the mappers into two classes. The abstract player mapper is responsible for loading and saving the specific player data to the database. This is an abstract class whose behavior is just used only by the concrete mapper objects. A separate player mapper class is used for the interface for operations at the player level. The player mapper provides a find method and overrides the insert and update methods. For all of these its responsibility is to figure out which concrete mapper should handle the task and delegate to it.

Although a broad scheme like this makes sense for each type of inheritance mapping, the details do vary. Therefore, it's not possible to show a code example for this case. You can find good examples in each of the inheritance mapping pattern sections: Single Table Inheritance (278), Class Table Inheritance (285), and Concrete Table Inheritance (293).

When to Use It

This general scheme makes sense for any inheritance-based database mapping. The alternatives involve such things as duplicating superclass mapping code among the concrete mappers and folding the player's interface into the abstract player mapper class. The former is a heinous crime, and the latter is possible but leads to a player mapper class that's messy and confusing. On the whole, then, its hard to think of a good alternative to this pattern.