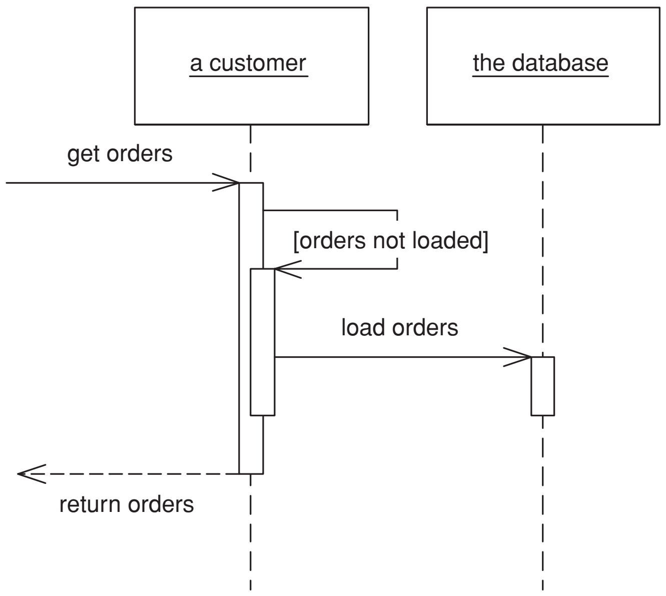
Lazy Load

An object that doesn't contain all of the data you need but knows how to get it.



For loading data from a database into memory it's handy to design things so that as you load an object of interest you also load the objects that are related to it. This makes loading easier on the developer using the object, who otherwise has to load all the objects he needs explicitly.

However, if you take this to its logical conclusion, you reach the point where loading one object can have the effect of loading a huge number of related objects—something that hurts performance when only a few of the objects are actually needed.

A *Lazy Load* interrupts this loading process for the moment, leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used. As many people know, if you're lazy about doing things you'll win when it turns out you don't need to do them at all.

How It Works

There are four main ways you can implement *Lazy Load*: lazy initialization, virtual proxy, value holder, and ghost.

Lazy initialization [Beck Patterns] is the simplest approach. The basic idea is that every access to the field checks first to see if it's null. If so, it calculates the value of the field before returning the field. To make this work you have to ensure that the field is self-encapsulated, meaning that all access to the field, even from within the class, is done through a getting method.

Using a null to signal a field that hasn't been loaded yet works well, unless null is a legal value field value. In this case you need something else to signal that the field hasn't been loaded, or you need to use a *Special Case (496)* for the null value.

Using lazy initialization is simple, but it does tend to force a dependency between the object and the database. For that reason it works best for *Active Record (160)*, *Table Data Gateway (144)*, and *Row Data Gateway (152)*. If you're using *Data Mapper (165),* you'll need an additional layer of indirection, which you can obtain by using a **virtual proxy** [Gang of Four]. A virtual proxy is an object that looks like the object that should be in the field, but doesn't actually contain anything. Only when one of its methods is called does it load the correct object from the database.

The good thing about a virtual proxy is that it looks exactly like the object that's supposed to be there. The bad thing is that it isn't that object, so you can easily run into a nasty identity problem. Furthermore you can have more than one virtual proxy for the same real object. All of these proxies will have different object identities, yet they represent the same conceptual object. At the very least you have to override the equality method and remember to use it instead of an identity method. Without that, and discipline, you'll run into some very hard-to-track bugs.

In some environments another problem is that you end up having to create lots of virtual proxies, one for each class you're proxying. You can usually avoid this in dynamically typed languages, but in statically typed languages things often get messy. Even when the platform provides handy facilities, such as Java's proxies, other inconveniences can come up.

These problem don't hit you if you only use virtual proxies for collections classes, such as lists. Since collections are *Value Objects (486)*, their identity doesn't matter. Additionally you only have a few collection classes to write virtual collections for.

With domain classes you can get around these problems by using a **value holder**. This concept, which I first came across in Smalltalk, is an object that wraps some other object. To get the underlying object you ask the value holder for its value, but only on the first access does it pull the data from the database. The disadvantages of the value holder are that the class needs to know that it's present and that you lose the explicitness of strong typing. You can avoid identity problems by ensuring that the value holder is never passed out beyond its owning class.

A **ghost** is the real object in a partial state. When you load the object from the database it contains just its ID. Whenever you try to access a field it loads its full state. Think of a ghost as an object, where every field is lazy-initialized in one fell swoop, or as a virtual proxy, where the object is its own virtual proxy. Of course, there's no need to load all the data in one go; you may group it in groups that are commonly used together. If you use a ghost, you can put it immediately in its *Identity Map (195)*. This way you maintain identity and avoid all problems due to cyclic references when reading in data.

A virtual proxy/ghost doesn't need to be completely devoid of data. If you have some data that's quick to get hold of and commonly used, it may make sense to load it when you load the proxy or ghost. (This is sometimes referred to as a "light object.")

Inheritance often poses a problem with *Lazy Load*. If you're going to use ghosts, you'll need to know what type of ghost to create, which you often can't tell without loading the thing properly. Virtual proxies can suffer from the same problem in statically typed languages.

Another danger with *Lazy Load* is that it can easily cause more database accesses than you need. A good example of this **ripple loading** is if you fill a collection with *Lazy Loads* and then look at them one at a time. This will cause you to go to the database once for each object instead of reading them all in at once. I've seen ripple loading cripple the performance of an application. One way to avoid it is never to have a collection of *Lazy Loads* but, rather make the collection itself a *Lazy Load* and, when you load it, load all the contents. The limitation of this tactic is when the collection is very large, such as all the IP addresses in the world. These aren't usually linked through associations in the object model, so that doesn't happen very often, but when it does you'll need a Value List Handler [Alur et al.].

Lazy Load is a good candidate for aspect-oriented programming. You can put *Lazy Load* behavior into a separate aspect, which allows you to change the lazy load strategy separately as well as freeing the domain developers from having to deal with lazy loading issues. I've also seen a project post-process Java bytecode to implement *Lazy Load* in a transparent way.

Often you'll run into situations where different use cases work best with different varieties of laziness. Some need one subset of the object graph; others need another subset. For maximum efficiency you want to load the right subgraph for the right use case.

The way to deal with this is to have separate database interaction objects for the different use cases. Thus, if you use *Data Mapper (165)*, you may have two order mapper objects: one that loads the line items immediately and one that loads them lazily. The application code chooses the appropriate mapper depending on the use case. A variation on this is to have the same basic loader object but defer to a strategy object to decide the loading pattern. This is a bit more sophisticated, but it can be a better way to factor behavior.

In theory you might want a range of different degrees of laziness, but in practice you really need only two: a complete load and enough of a load for identification purposes in a list. Adding more usually adds more complexity than is worthwhile.

When to Use It

Deciding when to use *Lazy Load* is all about deciding how much you want to pull back from the database as you load an object, and how many database calls that will require. It's usually pointless to use *Lazy Load* on a field that's stored in the same row as the rest of the object, because most of the time it doesn't cost any more to bring back extra data in a call, even if the data field is quite large—such as a *Serialized LOB (272)*. That means it's usually only worth considering *Lazy Load* if the field requires an extra database call to access.

In performance terms it's about deciding when you want to take the hit of bringing back the data. Often it's a good idea to bring everything you'll need in one call so you have it in place, particularly if it corresponds to a single interaction with a UI. The best time to use *Lazy Load* is when it involves an extra call and the data you're calling isn't used when the main object is used.

Adding *Lazy Load* does add a little complexity to the program, so my preference is not to use it unless I actively think I'll need it.

Example: Lazy Initialization (Java)

The essence of lazy initialization is code like this:

```
class Supplier...

   public List getProducts() {
      if (products == null) products = Product.findForSupplier(getID());
      return products;
   }
```

In this way the first access of the products field causes the data to be loaded from the database.

Example: Virtual Proxy (Java)

The key to the virtual proxy is providing a class that looks like the actual class you normally use but that actually holds a simple wrapper around the real class. Thus, a list of products for a supplier would be held with a regular list field.

```
class SupplierVL...

   private List products;
```

The most complicated thing about producing a list proxy like this is setting it up so that you can provide an underlying list that's created only when it's accessed. To do this we have to pass the code that's needed to create the list into the virtual list when it's instantiated. The best way to do this in Java is to define an interface for the loading behavior.

```
public interface VirtualListLoader {
   List load();
}
```

Then we can instantiate the virtual list with a loader that calls the appropriate mapper method.

```
class SupplierMapper...

   public static class ProductLoader implements VirtualListLoader {
      private Long id;
      public ProductLoader(Long id) {
         this.id = id;
      }
      public List load() {
         return ProductMapper.create().findForSupplier(id);
      }
   }
```

During the load method we assign the product loader to the list field.

```
class SupplierMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
      String nameArg = rs.getString(2);
      SupplierVL result = new SupplierVL(id, nameArg);
      result.setProducts(new VirtualList(new ProductLoader(id)));
      return result;
   }
```

The virtual list's source list is self-encapsulated and evaluates the loader on first reference.

```
class VirtualList...

   private List source;
   private VirtualListLoader loader;
   public VirtualList(VirtualListLoader loader) {
      this.loader = loader;
   }
   private List getSource() {
      if (source == null) source = loader.load();
      return source;
   }
```

The regular list methods to delegate are then implemented to the source list.

```
class VirtualList...

   public int size() {
      return getSource().size();
   }
   public boolean isEmpty() {
      return getSource().isEmpty();
   }
   // ... and so on for rest of list methods
```

This way the domain class knows nothing about how the mapper class does the *Lazy Load*. Indeed, the domain class isn't even aware that there is a *Lazy Load*.

Example: Using a Value Holder (Java)

A value holder can be used as a generic *Lazy Load*. In this case the domain type is aware that something is afoot, since the product field is typed as a value holder. This fact can be hidden from clients of the supplier by the getting method.

```
class SupplierVH...

   private ValueHolder products;
   public List getProducts() {
      return (List) products.getValue();
   }
```

The value holder itself does the *Lazy Load* behavior. It needs to be passed the necessary code to load its value when it's accessed. We can do this by defining a loader interface.

```
class ValueHolder...

   private Object value;
   private ValueLoader loader;
   public ValueHolder(ValueLoader loader) {
      this.loader = loader;
   }
   public Object getValue() {
      if (value == null) value = loader.load();
      return value;
   }
public interface ValueLoader {
   Object load();
}
```

A mapper can set up the value holder by creating an implementation of the loader and putting it into the supplier object.

```
class SupplierMapper...

   protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
      String nameArg = rs.getString(2);
      SupplierVH result = new SupplierVH(id, nameArg);
      result.setProducts(new ValueHolder(new ProductLoader(id)));
      return result;
   }
   public static class ProductLoader implements ValueLoader {
      private Long id;
      public ProductLoader(Long id) {
         this.id = id;
      }
      public Object load() {
         return ProductMapper.create().findForSupplier(id);
      }
   }
```

Example: Using Ghosts (C#)

Much of the logic for making objects ghosts can be built into *Layer Supertypes (475)*. As a consequence, if you use ghosts you tend to see them used everywhere. I'll begin our exploration of ghosts by looking at the domain object *Layer Supertype (475)*. Each domain object knows if it's a ghost or not.

```
class Domain Object...

      LoadStatus Status;
      public DomainObject (long key) {
         this.Key = key;
      }
      public Boolean IsGhost {
         get {return Status == LoadStatus.GHOST;}
      }
      public Boolean IsLoaded {
         get {return Status == LoadStatus.LOADED;}
      }
      public void MarkLoading() {
         Debug.Assert(IsGhost);
         Status = LoadStatus.LOADING;
      }
      public void MarkLoaded() {
         Debug.Assert(Status == LoadStatus.LOADING);
         Status = LoadStatus.LOADED;
      }
   enum LoadStatus {GHOST, LOADING, LOADED};
```

Domain objects can be in three states: ghost, loading, and loaded. I like to wrap status information with read-only properties and explicit status change methods.

The most intrusive element of ghosts is that every accessor needs to be modified so that it will trigger a load if the object actually is a ghost.

```
class Employee...

      public String Name {
         get {
            Load();
            return _name;
         }
         set {
            Load();
            _name = value;
         }
      }
      String _name;

class Domain Object...

      protected void Load() {
         if (IsGhost)
            DataSource.Load(this);
      }
```

Such a need, which is annoying to remember, is an ideal target for aspect-oriented programming for post-processing the bytecode.

In order for the loading to work, the domain object needs to call the correct mapper. However, my visibility rules dictate that the domain code may not see the mapper code. To avoid the dependency, I need to use an interesting combination of *Registry (480)* and *Separated Interface (476)* (Figure 11.4). I define a *Registry (480)* for the domain for data source operations.

```
class DataSource...

      public static void Load (DomainObject obj) {
         instance.Load(obj);
      }
```

The instance of the data source is defined using an interface.

```
class DataSource...

   public interface IDataSource {
      void Load (DomainObject obj);
   }
```

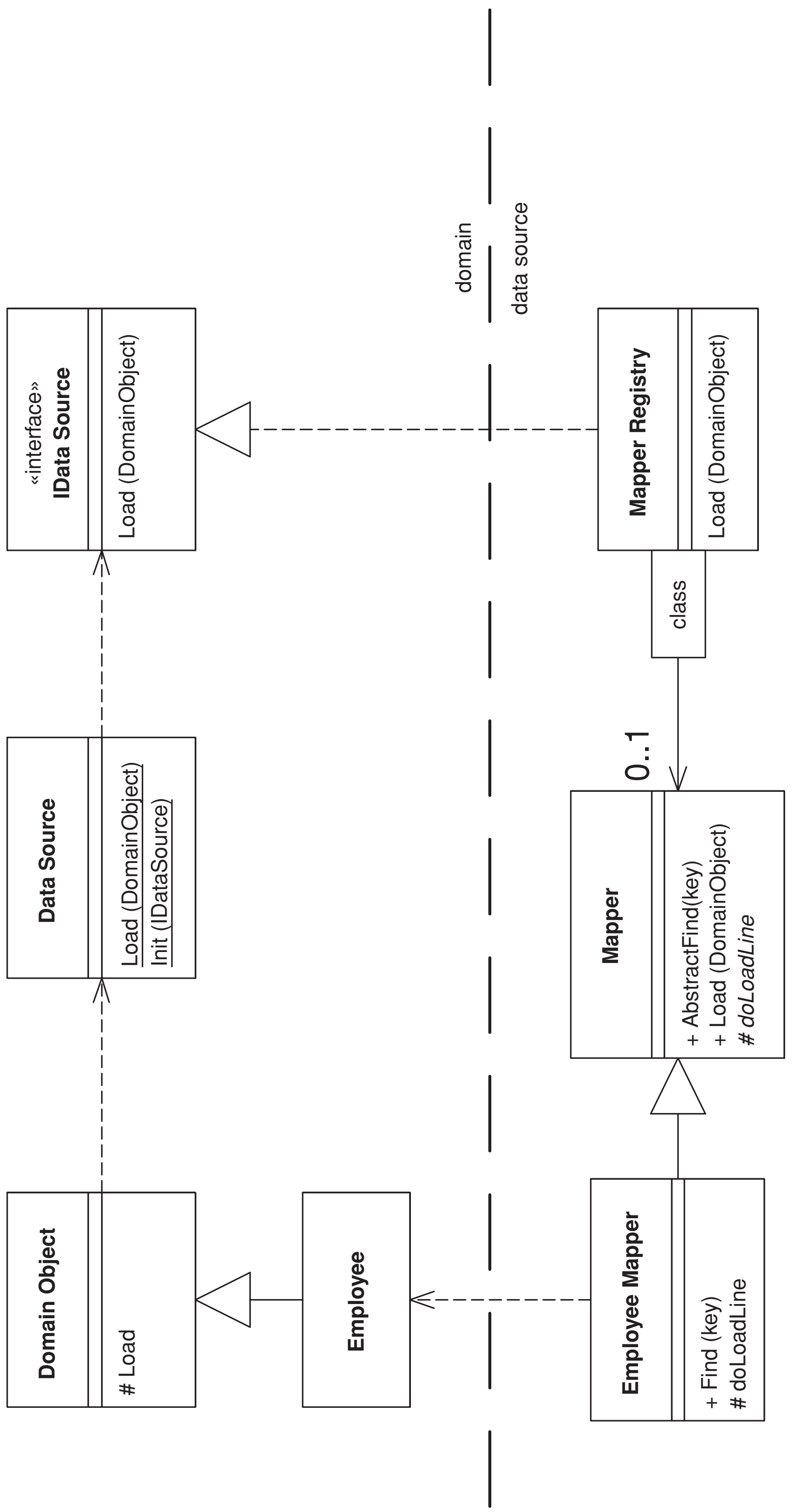


Figure 11.4 *Classes involved in loading a ghost.*

A registry of mappers, defined in the data source layer, implements the data source interface. In this case I've put the mappers in a dictionary indexed by domain type. The load method finds the correct mapper and tells it to load the appropriate domain object.

```
class MapperRegistry : IDataSource...

      public void Load (DomainObject obj) {
         Mapper(obj.GetType()).Load (obj);
      }
      public static Mapper Mapper(Type type) {
         return (Mapper) instance.mappers[type];
      }
      IDictionary mappers = new Hashtable();
```

The preceding code shows how the domain objects interact with the data source. The data source logic uses *Data Mappers (165)*. The update logic on the mappers is the same as in the case with no ghosts—the interesting behavior for this example lies in the finding and loading behavior.

Concrete mapper classes have their own find methods that use an abstract method and downcast the result.

```
class EmployeeMapper...

      public Employee Find (long key) {
         return (Employee) AbstractFind(key);
      }

class Mapper...

      public DomainObject AbstractFind (long key) {
         DomainObject result;
         result = (DomainObject) loadedMap[key];
         if (result == null) {
            result = CreateGhost(key);
            loadedMap.Add(key, result);
         }
         return result;
      }
      IDictionary loadedMap = new Hashtable();
      public abstract DomainObject CreateGhost(long key);

class EmployeeMapper...

      public override DomainObject CreateGhost(long key) {
         return new Employee(key);
      }
```

As you can see, the find method returns an object in its ghost state. The actual data does not come from the database until the load is triggered by accessing a property on the domain object.

```
class Mapper...

      public void Load (DomainObject obj) {
         if (! obj.IsGhost) return;
         IDbCommand comm = new OleDbCommand(findStatement(), DB.connection);
         comm.Parameters.Add(new OleDbParameter("key",obj.Key));
         IDataReader reader = comm.ExecuteReader();
         reader.Read();
         LoadLine (reader, obj);
         reader.Close();
      }
      protected abstract String findStatement();
      public void LoadLine (IDataReader reader, DomainObject obj) {
         if (obj.IsGhost) {
            obj.MarkLoading();
            doLoadLine (reader, obj);
            obj.MarkLoaded();
         }
      }
      protected abstract void doLoadLine (IDataReader reader, DomainObject obj);
```

As is common with these examples, the *Layer Supertype (475)* handles all of the abstract behavior and then calls an abstract method for a particular subclass to play its part. For this example I've used a data reader, a cursor-based approach that's the more common for the various platforms at the moment. I'll leave it to you to extend this to a data set, which would actually be more suitable for most cases in .NET.

For this employee object, I'll show three kinds of property: a name that's a simple value, a department that's a reference to another object, and a list of timesheet records that shows the case of a collection. All are loaded together in the subclass's implementation of the hook method.

```
class EmployeeMapper...

      protected override void doLoadLine (IDataReader reader, DomainObject obj) {
         Employee employee = (Employee) obj;
         employee.Name = (String) reader["name"];
         DepartmentMapper depMapper =
            (DepartmentMapper) MapperRegistry.Mapper(typeof(Department));
         employee.Department = depMapper.Find((int) reader["departmentID"]);
         loadTimeRecords(employee);
      }
```

The name's value is loaded simply by reading the appropriate column from the data reader's current cursor. The department is read by using the find method on the department's mapper object. This will end up setting the property to a ghost of the department; the department's data will only be read when the department object itself is accessed.

The collection is the most complicated case. To avoid ripple loading, it's important to load all the time records in a single query. For this we need a special list implementation that acts as a ghost list. This list is just a thin wrapper around a real list object, to which all the real behavior is just delegated. The only thing the ghost does is ensure that any accesses to the real list triggers a load.

```
class DomainList...

      IList data {
         get {
            Load();
            return _data;
         }
         set {_data = value;}
      }
      IList _data = new ArrayList();
      public int Count {
         get {return data.Count;}
      }
```

The domain list class is used by domain objects and is part of the domain layer. The actual loading needs access to SQL commands, so I use a delegate to define a loading function than can be supplied by the mapping layer.

```
class DomainList...

      public void Load () {
         if (IsGhost) {
            MarkLoading();
            RunLoader(this);
            MarkLoaded();
         }
      }
      public delegate void Loader(DomainList list);
      public Loader RunLoader;
```

Think of a delegate as a special variety of *Separated Interface (476)* for a single function. Indeed, declaring an interface with a single function in it is a reasonable alternative way of doing this.

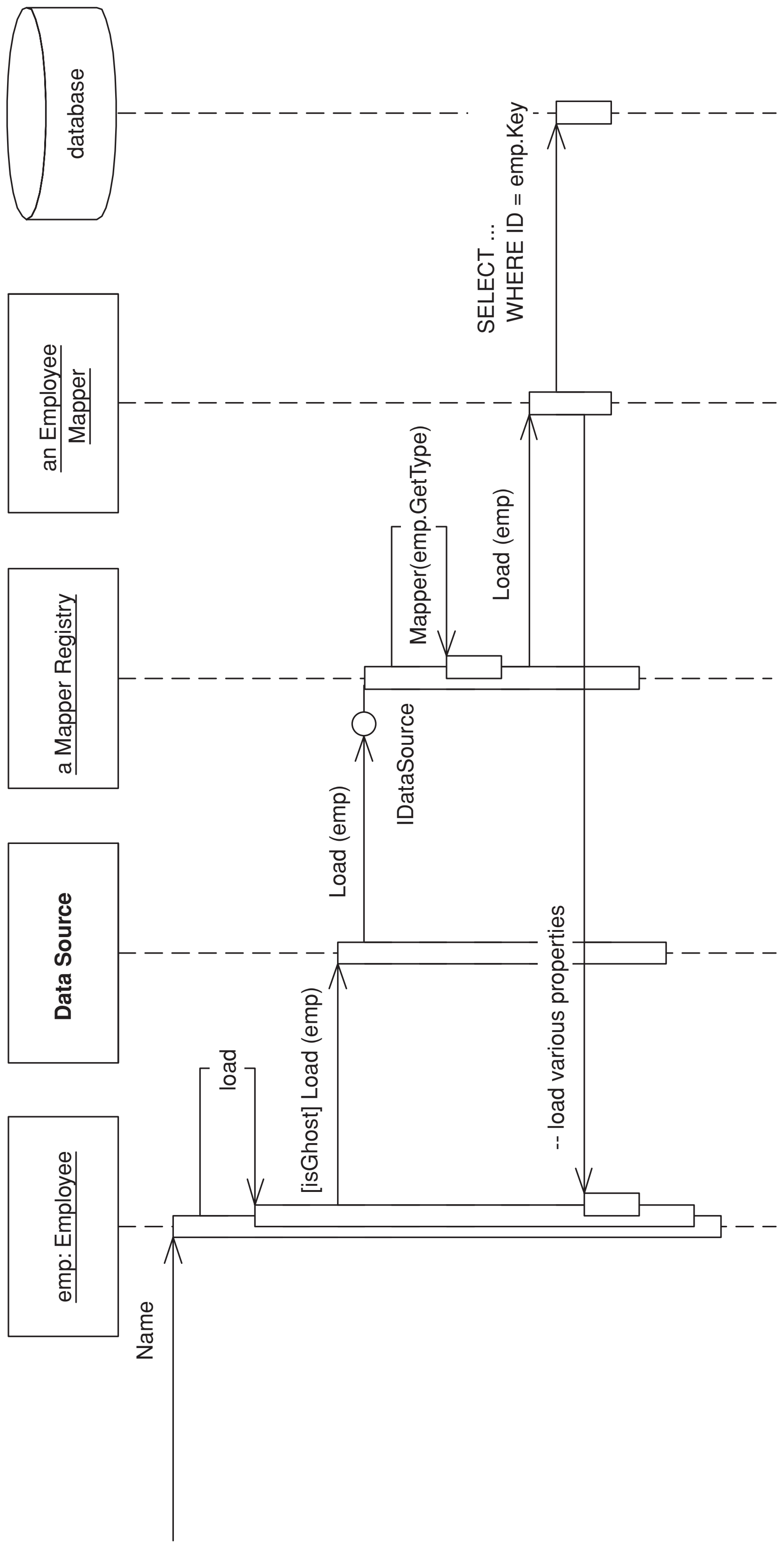


Figure 11.5 *The load sequence for a ghost.*

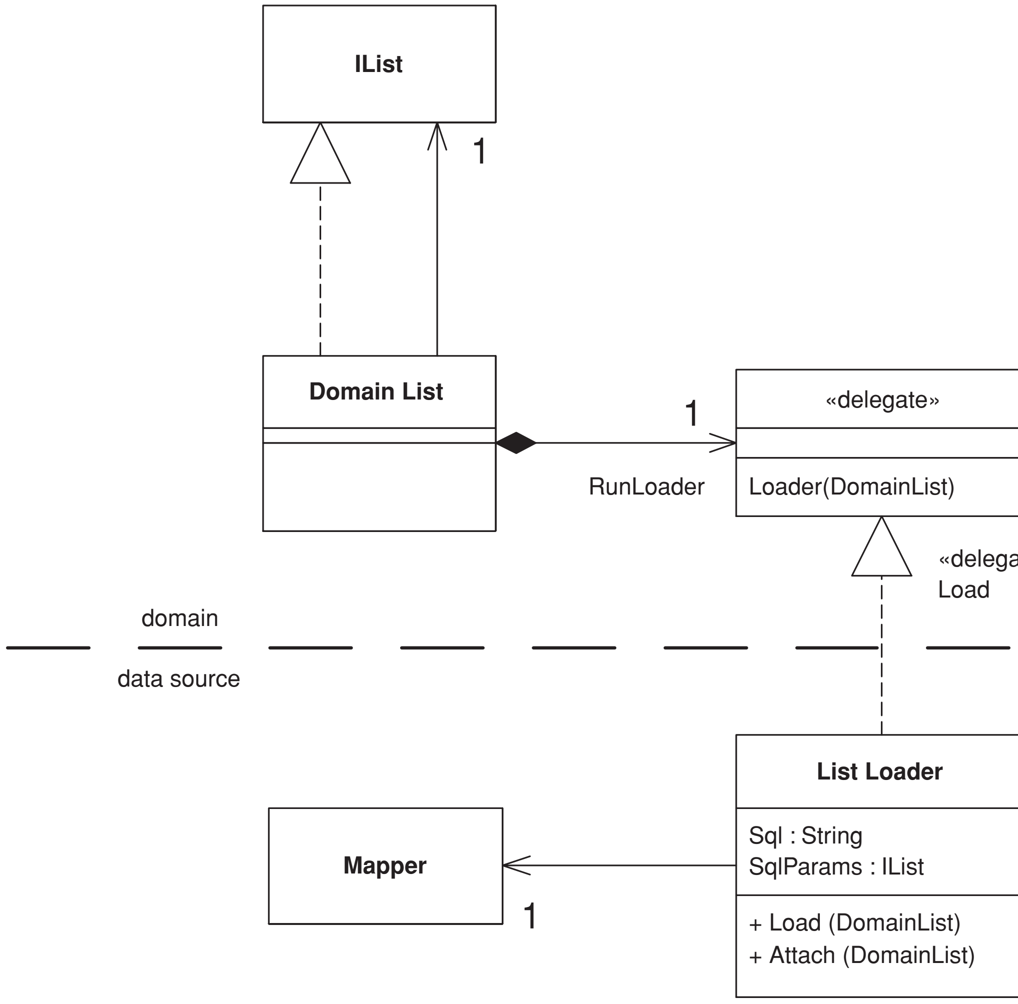


Figure 11.6 *Classes for a ghost list. As yet there's no accepted standard for showing delegates in UML models. This is my current approach.*

The loader itself has properties to specify the SQL for the load and mapper to use for mapping the time records. The employee's mapper sets up the loader when it loads the employee object.

```
class EmployeeMapper...

    void loadTimeRecords(Employee employee) {
        ListLoader loader = new ListLoader();
        loader.Sql = TimeRecordMapper.FIND_FOR_EMPLOYEE_SQL;
        loader.SqlParams.Add(employee.Key);
        loader.Mapper = MapperRegistry.Mapper(typeof(TimeRecord));
        loader.Attach((DomainList) employee.TimeRecords);
    }
```

```
class ListLoader...

      public String Sql;
      public IList SqlParams = new ArrayList();
      public Mapper Mapper;
```

Since the syntax for the delegate assignment is a bit complicated, I've given the loader an attach method.

```
class ListLoader...

      public void Attach (DomainList list) {
         list.RunLoader = new DomainList.Loader(Load);
      }
```

When the employee is loaded, the time records collection stays in a ghost state until one of the access methods fires to trigger the loader. At this point the loader executes the query to fill the list.

```
class ListLoader...

      public void Load (DomainList list) {
         list.IsLoaded = true;
         IDbCommand comm = new OleDbCommand(Sql, DB.connection);
         foreach (Object param in SqlParams)
            comm.Parameters.Add(new OleDbParameter(param.ToString(),param));
         IDataReader reader = comm.ExecuteReader();
         while (reader.Read()) {
            DomainObject obj = GhostForLine(reader);
            Mapper.LoadLine(reader, obj);
            list.Add (obj);
         }
         reader.Close();
      }
      private DomainObject GhostForLine(IDataReader reader) {
         return Mapper.AbstractFind((System.Int32)reader[Mapper.KeyColumnName]);
      }
```

Using ghost lists like this is important to reduce ripple loading. It doesn't completely eliminate it, as there are other cases where it appears. In this example, a more sophisticated mapping could load the department's data in a single query with the employee. However, always loading all the elements in a collection together helps eliminate the worst cases.