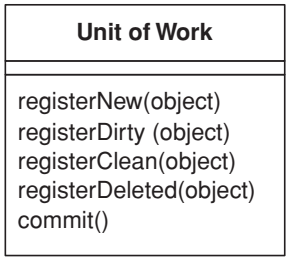
Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.

You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.

A *Unit of Work* keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

How It Works

The obvious things that cause you to deal with the database are changes: new object created and existing ones updated or deleted. *Unit of Work* is an object that keeps track of these things. As soon as you start doing something that may affect a database, you create a *Unit of Work* to keep track of the changes. Every time you create, change, or delete an object you tell the *Unit of Work*. You can also let it know about objects you've read so that it can check for inconsistent reads by verifying that none of the objects changed on the database during the business transaction.

The key thing about *Unit of Work* is that, when it comes time to commit, the *Unit of Work* decides what to do. It opens a transaction, does any concurrency checking (using *Pessimistic Offline Lock (426)* or *Optimistic Offline Lock (416)*), and writes changes out to the database. Application programmers never explicitly call methods for database updates. This way they don't have to keep track of what's changed or worry about how referential integrity affects the order in which they need to do things.

Of course for this to work the *Unit of Work* needs to know what objects it should keep track of. You can do this either by the caller doing it or by getting the object to tell the *Unit of Work*.

With **caller registration** (Figure 11.1), the user of an object has to remember to register the object with the *Unit of Work* for changes. Any objects that aren't registered won't be written out on commit. Although this allows forgetfulness

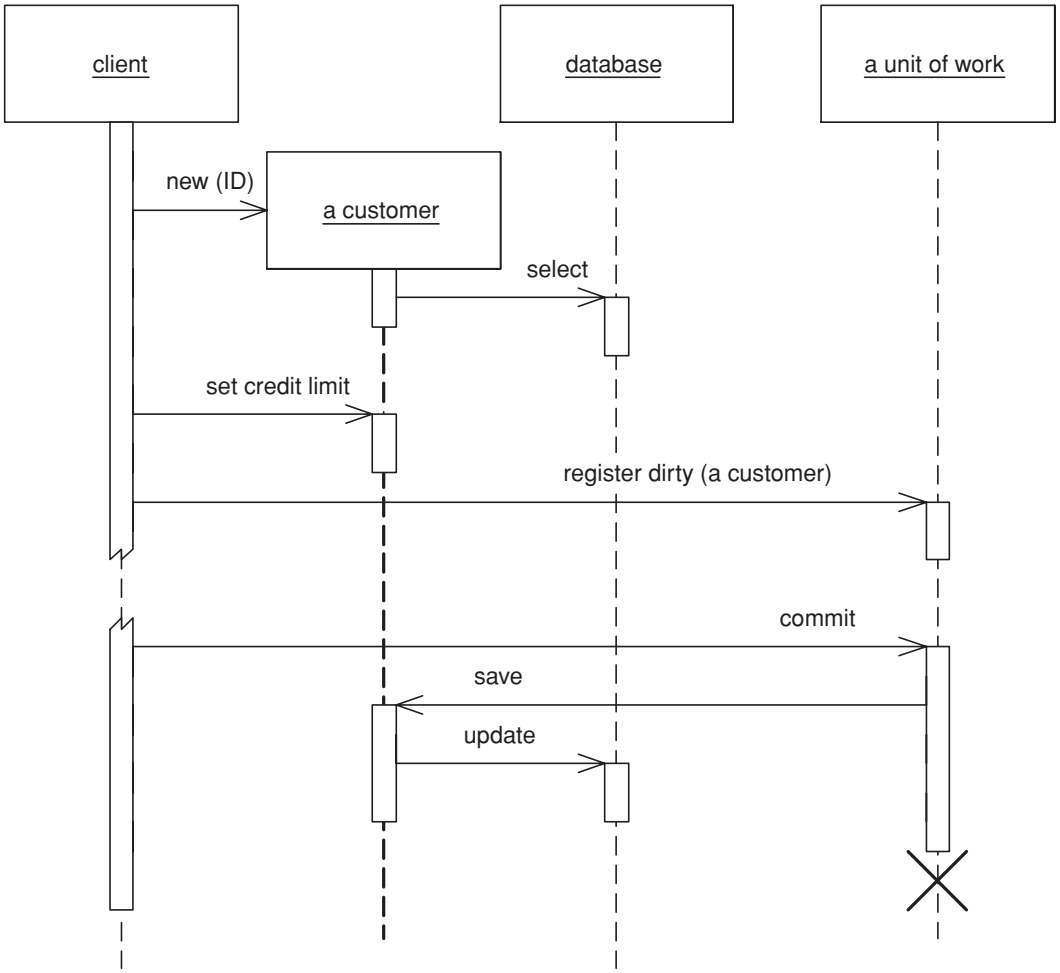


Figure 11.1 *Having the caller register a changed object.*

to cause trouble, it does give flexibility in allowing people to make in-memory changes that they don't want written out. Still, I would argue that it's going to cause far more confusion than would be worthwhile. It's better to make an explicit copy for that purpose.

With **object registration** (Figure 11.2), the onus is removed from the caller. The usual trick here is to place registration methods in object methods. Loading an object from the database registers the object as clean; the setting methods register the object as dirty. For this scheme to work the *Unit of Work* needs either to be passed to the object or to be in a well-known place. Passing the

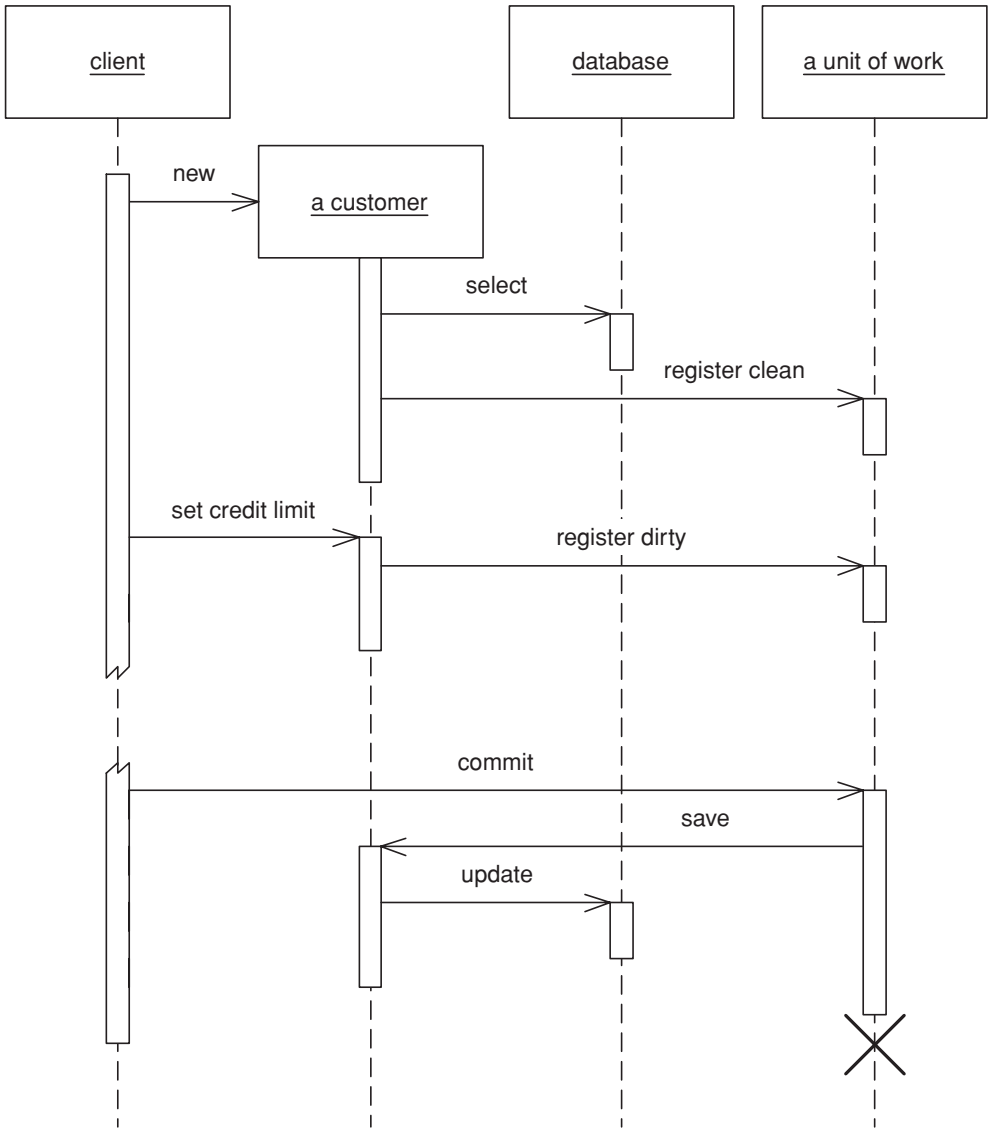


Figure 11.2 *Getting the receiver object to register itself.*

Unit of Work around is tedious but usually no problem to have it present in some kind of session object.

Even object registration leaves something to remember; that is, the developer of the object has to remember to add a registration call in the right places. The consistency becomes habitual, but is still an awkward bug when missed.

This is a natural place for code generation to generate appropriate calls, but that only works when you can clearly separate generated and nongenerated code. This problem turns out to be particularly suited to aspect-oriented programming. I've also come across post-processing of the object files to pull this off. In this example a post-processor examined all the Java .class files, looked for the appropriate methods and inserted registration calls into the byte code. Such finicking around feels dirty, but it separates the database code from the regular code. Aspect-oriented programming will do this more cleanly with source code, and as its tools become more commonplace I expect to see this strategy being used.

Another technique I've seen is **unit of work controller** (Figure 11.3), which the TOPLink product uses. Here the *Unit of Work* handles all reads from the database and registers clean objects whenever they're read. Rather than marking objects as dirty the *Unit of Work* takes a copy at read time and then compares the object at commit time. Although this adds overhead to the commit process, it allows a selective update of only those fields that were actually changed; it also avoids registration calls in the domain objects. A hybrid approach is to take copies only of changed objects. This requires registration, but it supports selective update and greatly reduces the overhead of the copy if there are many more reads than updates.

Object creation is often a special time to consider caller registration. It's not uncommon for people to create objects that are only supposed to be transient. A good example of this is in testing domain objects, where the tests run much faster without database writes. Caller registration can make this apparent. However, there are other solutions, such as providing a transient constructor that doesn't register with the *Unit of Work* or, better still, providing a *Special Case (496)Unit of Work* that does nothing with a commit.

Another area where a *Unit of Work* can be helpful is in update order when a database uses referential integrity. Most of the time you can avoid this issue by ensuring that the database only checks referential integrity when the transaction commits rather than with each SQL call. Most databases allow this, and if available there's no good reason not to do it. If not, the *Unit of Work* is the natural place to sort out the update order. In smaller systems this can be done with explicit code that contains details about which tables to write first based on the foreign key dependencies. In a larger application it's better to use metadata to

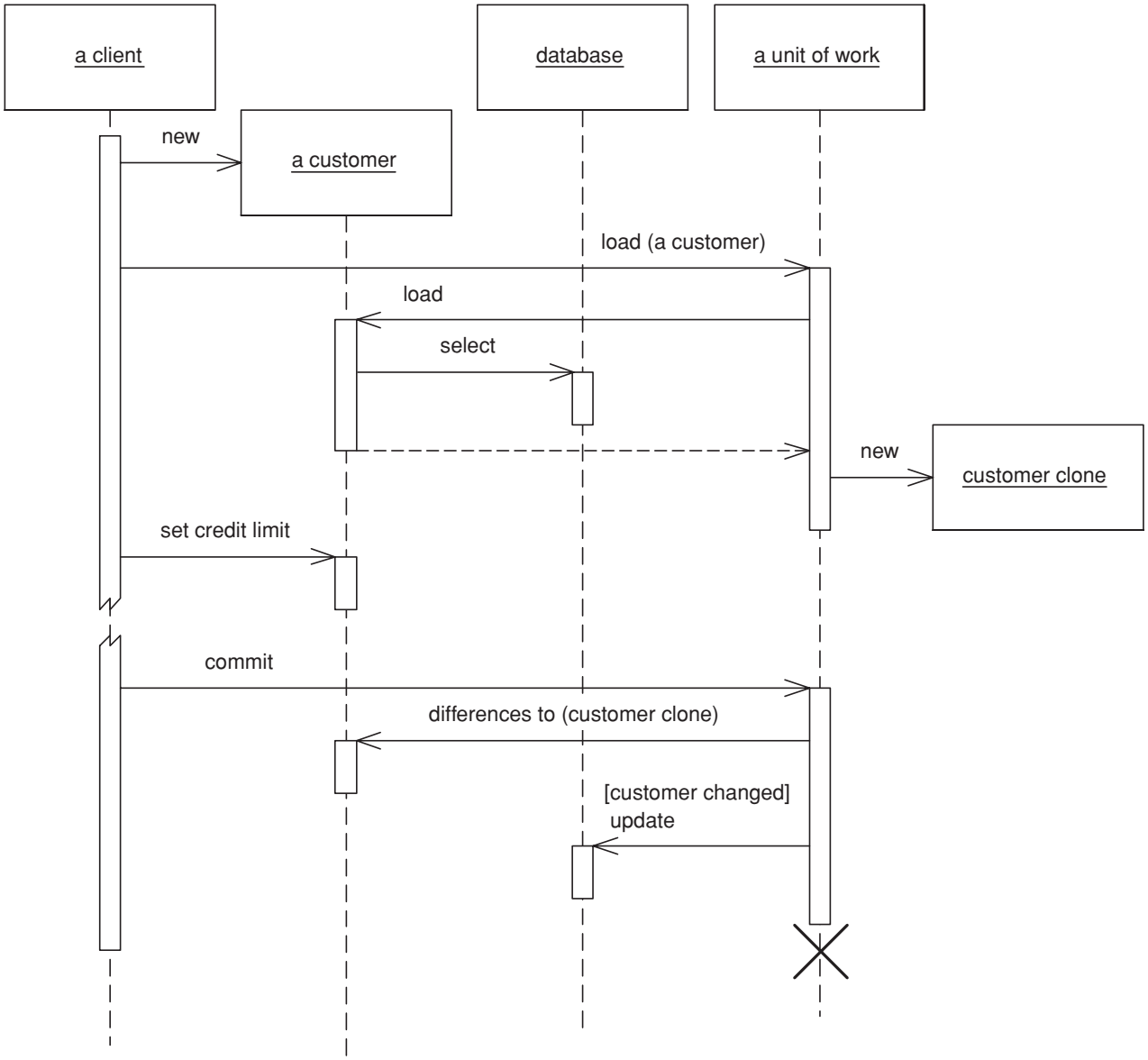


Figure 11.3 *Using the* Unit of Work *as the controller for database access.*

figure out which order to write to the database. How you do that is beyond the scope of this book, and it's a common reason to use a commercial tool. If you have to do it yourself, I'm told the key to the puzzle is a topological sort.

You can use a similar technique to minimize deadlocks. If every transaction uses the same sequence of tables to edit, you greatly reduce the risk of deadlocks. The *Unit of Work* is an ideal place to hold a fixed sequence of table writes so that you always touch the tables in the same order.

Objects need to be able to find their current *Unit of Work*. A good way to do this is with a thread-scoped *Registry (480)*. Another way is to pass the *Unit of*

Work to objects that need it, either in method calls or when you create an object. In either case make sure that more than one thread can't get access to a *Unit of Work*—there lies the way to madness.

Unit of Work makes an obvious point of handling batch updates. The idea behind a **batch update** is to send multiple SQL commands as a single unit so that they can be processed in a single remote call. This is particularly important when many updates, inserts, and deletes are sent in rapid succession. Different environments provide different levels of support for batch updates. JDBC has a facility that allows you to batch individual statements. If you don't have this feature, you can mimic it by building up a string that has multiple SQL statements and then submitting as one statement. [Nilsson] describes an example of this for Microsoft platforms. However, if you do this check to see if it interferes with statement precompilation.

Unit of Work works with any transactional resource, not just databases, so you can also use it to coordinate with message queues and transaction monitors.

.NET Implementation

In .NET the *Unit of Work* is done by the disconnected data set. This makes it a slightly different pattern from the classical variety. Most *Units of Work* I've come across register and track changes to objects. .NET reads data from the database into a data set, which is a series of objects arranged like database tables, rows, and columns. The data set is essentially an in-memory mirror image of the result of one or more SQL queries. Each data row has the concept of a version (current, original, proposed) and a state (unchanged, added, deleted, modified), which, together with the fact that the data set mimics the database structure, makes for straightforward writing of changes to the database.

When to Use It

The fundamental problem that *Unit of Work* deals with is keeping track of the various objects you've manipulated so that you know which ones you need to consider to synchronize your in-memory data with the database. If you're able to do all your work within a system transaction, the only objects you need to worry about are those you alter. Although *Unit of Work* is generally the best way of doing this, there are alternatives.

Perhaps the simplest alternative is to explicitly save any object whenever you alter it. The problem here is that you may get many more database calls than you want since, if you alter one object at three different points in your work, you get three calls rather than one call in its final state.

To avoid multiple database calls, you can leave all your updates to the end. To do this you need to keep track of all the objects that have changed. You can

use variables in your code for this, but they soon become unmanageable once you have more than a few. Variables often work fine with a *Transaction Script (110)*, but they can be very difficult with a *Domain Model (116)*.

Rather than keep objects in variables you can give each object a dirty flag that you set when the object changes. Then you need to find all the dirty objects at the end of your transaction and write them out. The value of this technique hinges on how easy it is to find the dirty objects. If all of them are in a single hierarchy, then you can traverse the hierarchy and write out any that have been changed. However, a more general object network, such as a *Domain Model (116)*, is harder to traverse.

The great strength of *Unit of Work* is that it keeps all this information in one place. Once you have it working for you, you don't have to remember to do much in order to keep track of your changes. Also, *Unit of Work* is a firm platform for more complicated situations, such as handling business transactions that span several system transactions using *Optimistic Offline Lock (416)* and *Pessimistic Offline Lock (426)*.

Example: *Unit of Work* with Object Registration (Java)

by David Rice

Here's a *Unit of Work* that can track all changes for a given business transaction and then commit them to the database when instructed to do so. Our domain layer has a *Layer Supertype (475)*, `DomainObject`, with which the *Unit of Work* will interact. To store the change set we use three lists: new, dirty, and removed domain objects.

```
class UnitOfWork...

   private List newObjects = new ArrayList();
   private List dirtyObjects = new ArrayList();
   private List removedObjects = new ArrayList();
```

The registration methods maintain the state of these lists. They must perform basic assertions such as checking that an ID isn't null or that a dirty object isn't being registered as new.

```
class UnitOfWork...

   public void registerNew(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
      Assert.isTrue("object not removed", !removedObjects.contains(obj));
      Assert.isTrue("object not already registered new", !newObjects.contains(obj));
      newObjects.add(obj);
   }
```

```
public void registerDirty(DomainObject obj) {
   Assert.notNull("id not null", obj.getId());
   Assert.isTrue("object not removed", !removedObjects.contains(obj));
   if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
      dirtyObjects.add(obj);
   }
}
public void registerRemoved(DomainObject obj) {
   Assert.notNull("id not null", obj.getId());
   if (newObjects.remove(obj)) return;
   dirtyObjects.remove(obj);
   if (!removedObjects.contains(obj)) {
      removedObjects.add(obj);
   }
}
public void registerClean(DomainObject obj) {
   Assert.notNull("id not null", obj.getId());
}
```


Notice that `registerClean()` doesn't do anything here. A common practice is to place an *Identity Map (195)* within a *Unit of Work*. An *Identity Map (195)* is necessary almost any time you store domain object state in memory because multiple copies of the same object would result in undefined behavior. Were an *Identity Map (195)* in place, `registerClean()` would put the registered object in it. Likewise `registerNew()` would put a new object in the map and `registerRemoved()` would remove a deleted object from the map. Without the *Identity Map (195)* you have the option of not including `registerClean()` in your *Unit of Work*. I've seen implementations of this method that remove changed objects from the dirty list, but partially rolling back changes is always tricky. Be careful when reversing any state in the change set.

`commit()` will locate the *Data Mapper (165)* for each object and invoke the appropriate mapping method. `updateDirty()` and `deleteRemoved()` aren't shown, but they would behave like `insertNew()`, which is as expected.

```
class UnitOfWork...

   public void commit() {
      insertNew();
      updateDirty();
      deleteRemoved();
   }
   private void insertNew() {
      for (Iterator objects = newObjects.iterator(); objects.hasNext();) {
         DomainObject obj = (DomainObject) objects.next();
         MapperRegistry.getMapper(obj.getClass()).insert(obj);
      }
   }
```

Not included in this *Unit of Work* is the tracking of any objects we've read and want to check for inconsistent read errors upon commit. This is addressed in *Optimistic Offline Lock (416)*.

Next we need to facilitate object registration. First each domain object needs to find the *Unit of Work* serving the current business transaction. Since that *Unit of Work* will be needed by the entire domain model, passing it around as a parameter is probably unreasonable. As each business transaction executes within a single thread we can associate the *Unit of Work* with the currently executing thread using the `java.lang.ThreadLocal` class. Keeping things simple, we'll add this functionality by using static methods on our *Unit of Work* class. If we already have some sort of session object associated with the business transaction execution thread we should place the current *Unit of Work* on that session object rather than add the management overhead of another thread mapping. Besides, the *Unit of Work* logically belongs to the session.

```
class UnitOfWork...

   private static ThreadLocal current = new ThreadLocal();
   public static void newCurrent() {
      setCurrent(new UnitOfWork());
   }
   public static void setCurrent(UnitOfWork uow) {
      current.set(uow);
   }
   public static UnitOfWork getCurrent() {
      return (UnitOfWork) current.get();
   }
```

Now we can now give our abstract domain object the marking methods to register itself with the current *Unit of Work*.

```
class DomainObject...

   protected void markNew() {
      UnitOfWork.getCurrent().registerNew(this);
   }
   protected void markClean() {
      UnitOfWork.getCurrent().registerClean(this);
   }
   protected void markDirty() {
      UnitOfWork.getCurrent().registerDirty(this);
   }
   protected void markRemoved() {
      UnitOfWork.getCurrent().registerRemoved(this);
   }
```

Concrete domain objects need to remember to mark themselves new and dirty where appropriate.

```
class Album...

   public static Album create(String name) {
      Album obj = new Album(IdGenerator.nextId(), name);
      obj.markNew();
      return obj;
   }
   public void setTitle(String title) {
      this.title = title;
      markDirty();
   }
```


Not shown is that the registration of removed objects can be handled by a `remove()` method on the abstract domain object. Also, and if you've implemented `registerClean()` your *Data Mappers (165)* will need to register any newly loaded object as clean.

The final piece is to register and commit the *Unit of Work* where appropriate. This can be done either explicitly or implicitly. Here's what explicit *Unit of Work* management looks like:

```
class EditAlbumScript...

   public static void updateTitle(Long albumId, String title) {
      UnitOfWork.newCurrent();
      Mapper mapper = MapperRegistry.getMapper(Album.class);
      Album album = (Album) mapper.find(albumId);
      album.setTitle(title);
      UnitOfWork.getCurrent().commit();
   }
```

Beyond the simplest of applications, implicit *Unit of Work* management is more appropriate as it avoids repetitive, tedious coding. Here's a servlet *Layer Supertype (475)* that registers and commits the *Unit of Work* for its concrete subtypes. Subtypes will implement `handleGet()` rather than override `doGet()`. Any code executing within `handleGet()` will have a *Unit of Work* with which to work.

```
class UnitOfWorkServlet...

   final protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      try {
         UnitOfWork.newCurrent();
         handleGet(request, response);
         UnitOfWork.getCurrent().commit();
```

```
        } finally {
            UnitOfWork.setCurrent(null);
        }
    }
    abstract void handleGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException;
```

The above servlet example is obviously a bit simplistic, in that it skips system transaction control. If you were using *Front Controller (344)*, you would be more likely to wrap *Unit of Work* management around your commands rather than `doGet()`. Similar wrapping can be done with just about any execution context.