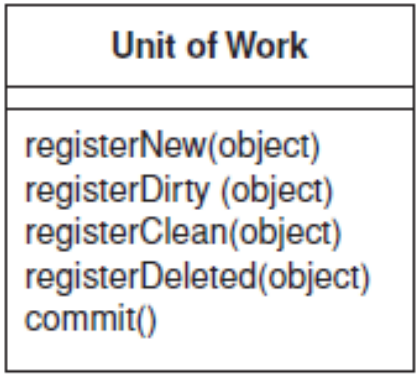
SOEN 387: Web-Based Enterprise Application Design

Chapter 11. Object-Relational Behavioral Patterns

Unit of Work

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.*



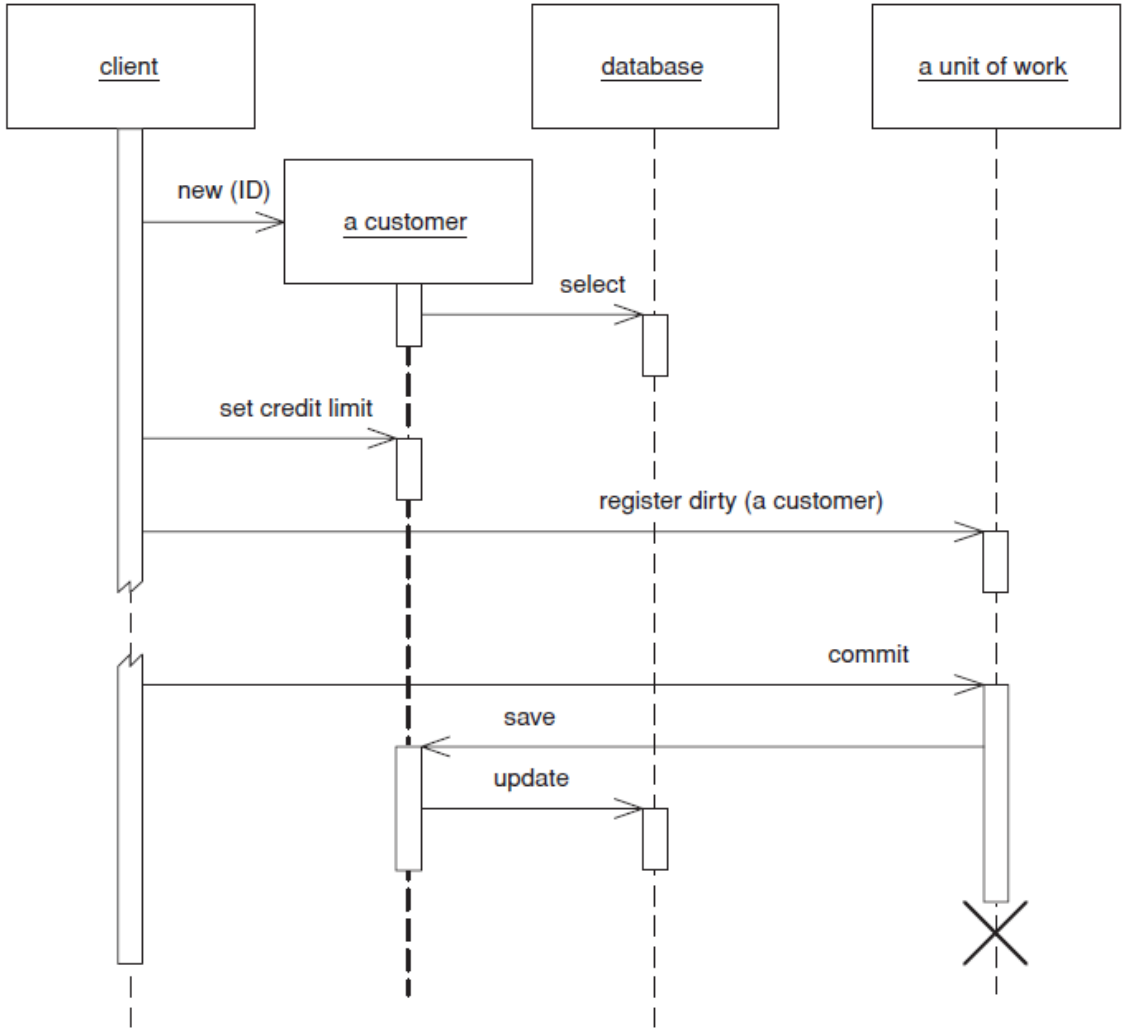
The obvious things that cause you to deal with the database are changes: New object created and existing ones updated or deleted.

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***.

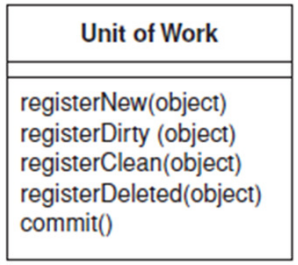
The key thing about ***Unit of Work*** *is that, when it comes time to commit, the Unit of Work decides what to do. It opens a transaction, does any concurrency checking (using* ***Pessimistic Offline Lock*** *or* ***Optimistic Offline Lock*** *), and writes changes out to the database. Application programmers never explicitly call methods for database updates.*

*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** , 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.*



Having the caller register a changed object.

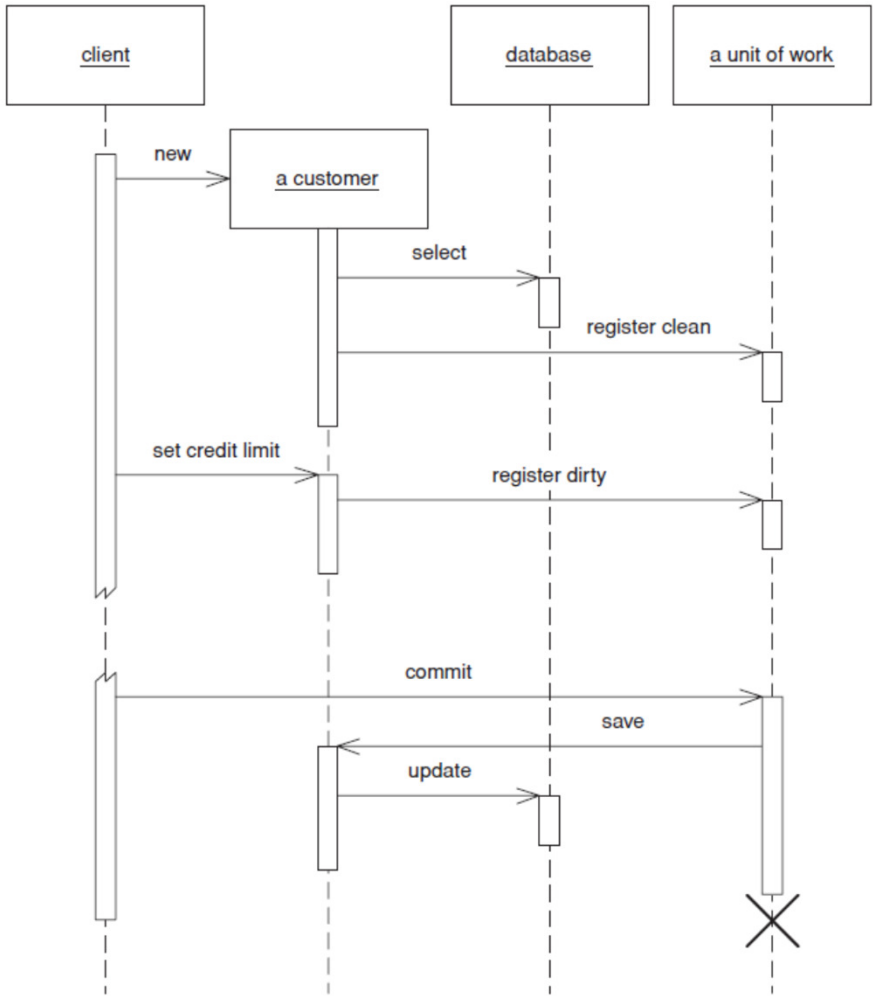


*With **object registration** , the responsibility is removed from the caller.*

*The way to implement object registration 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.



Getting the receiver object to register itself.

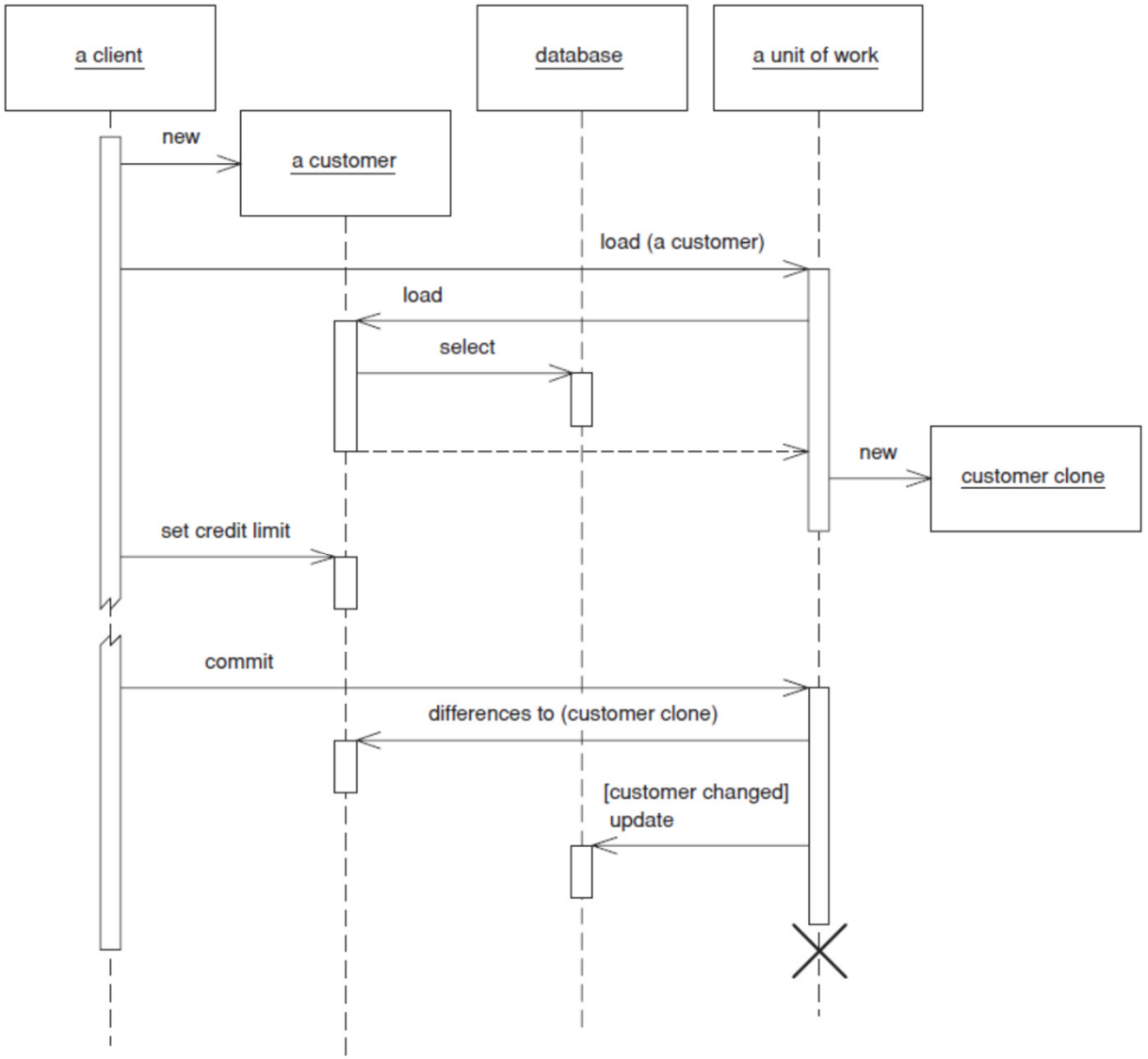
Unit of Work

registerNew(object)
registerDirty (object)
registerClean(object)
registerDeleted(object)
commit()

unit of work controller *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.

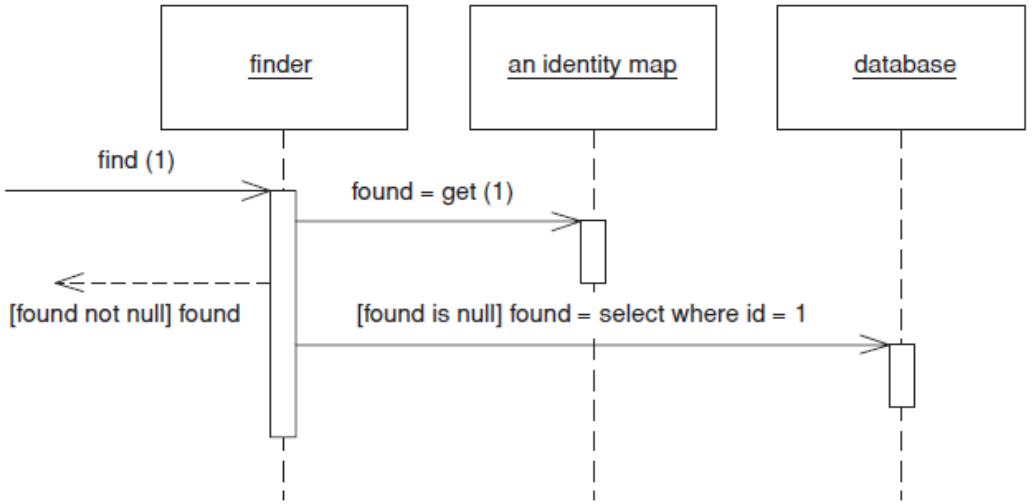
Although this adds overhead to the commit process, it allows a selective update of only those fields that were actually changed; it also avoids registration calls in the domain objects.



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

Identity Map

Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.



*An **Identity Map** keeps a record of all objects that have been read from the database. Whenever you want an object, you check the **Identity Map** first to see if you already have it.*

*The basic idea behind the **Identity Map** is to have a series of maps containing objects that have been pulled from the database.*

When you load an object from the database, you first check the map. If there's an object in it that corresponds to the one you're loading, you return it. If not, you go to the database, putting the objects into the map for future reference as you load them.

***Choice of Keys :** The first thing to consider is the key for the map. The obvious choice is the primary key of the corresponding database table.*

*In general you use an **Identity Map** to manage any object brought from a database and modified. The key reason is that you don't want a situation where two in-memory objects correspond to a single database record—you might modify the two records inconsistently and thus confuse the database mapping.*

*Another value in **Identity Map** is that it acts as a cache for database reads, which means that you can avoid going to the database each time you need some data.*

Example: Methods for an *Identity Map* (Java)

For each *Identity Map* we have a map field and accessors.

```
private Map people = new HashMap();

public static void addPerson(Person arg)
{
        soleInstance.people.put(arg.getID(), arg);
}
public static Person getPerson(Long key)
{
        return (Person) soleInstance.people.get(key);
}
public static Person getPerson(long key)
{
        return getPerson(new Long(key));
}
```

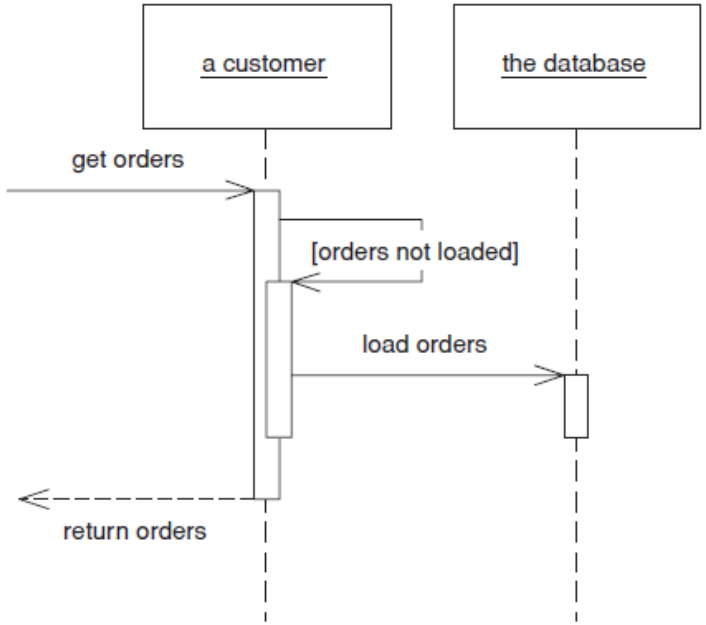
Lazy Load

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 needed 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.



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

Lazy initialization 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.

Example: Lazy Initialization (Java)

```
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.

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.

Value Holder is an object with a getValue method. Clients call getValue to get the real object, the first call triggers the load.

A **ghost** is the real object without any data. The first time you call a method the ghost loads the full data into its fields.

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.

One main issues 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.