SOEN 387: Web-Based Enterprise Application Design

Chapter 16. Offline Concurrency Patterns

Optimistic Offline Lock

Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.

Often a business transaction executes across a series of system transactions.
We can not depend on our database manager alone to ensure that the business transaction will leave the record data in a consistent state.
Data integrity is at risk once two sessions begin to work on the same records and lost updates are quite possible.
Also, with one session editing data that another is reading an inconsistent read becomes likely.

Optimistic Offline Lock solves this problem by validating that the changes about to be committed by one session don't conflict with the changes of another session.

A successful pre-commit validation is, in a sense, obtaining a lock indicating it is okay to go ahead with the changes to the record data.

Whereas Pessimistic Offline Lock assumes that the chance of session conflict is high and therefore limits the system's concurrency.

Optimistic Offline Lock assumes that the chance of conflict is low. The expectation that session conflict isn't likely allows multiple users to work with the same data at the same time.

An *Optimistic Offline Lock* is obtained by validating that, in the time since a session loaded a record, another session has not altered it.

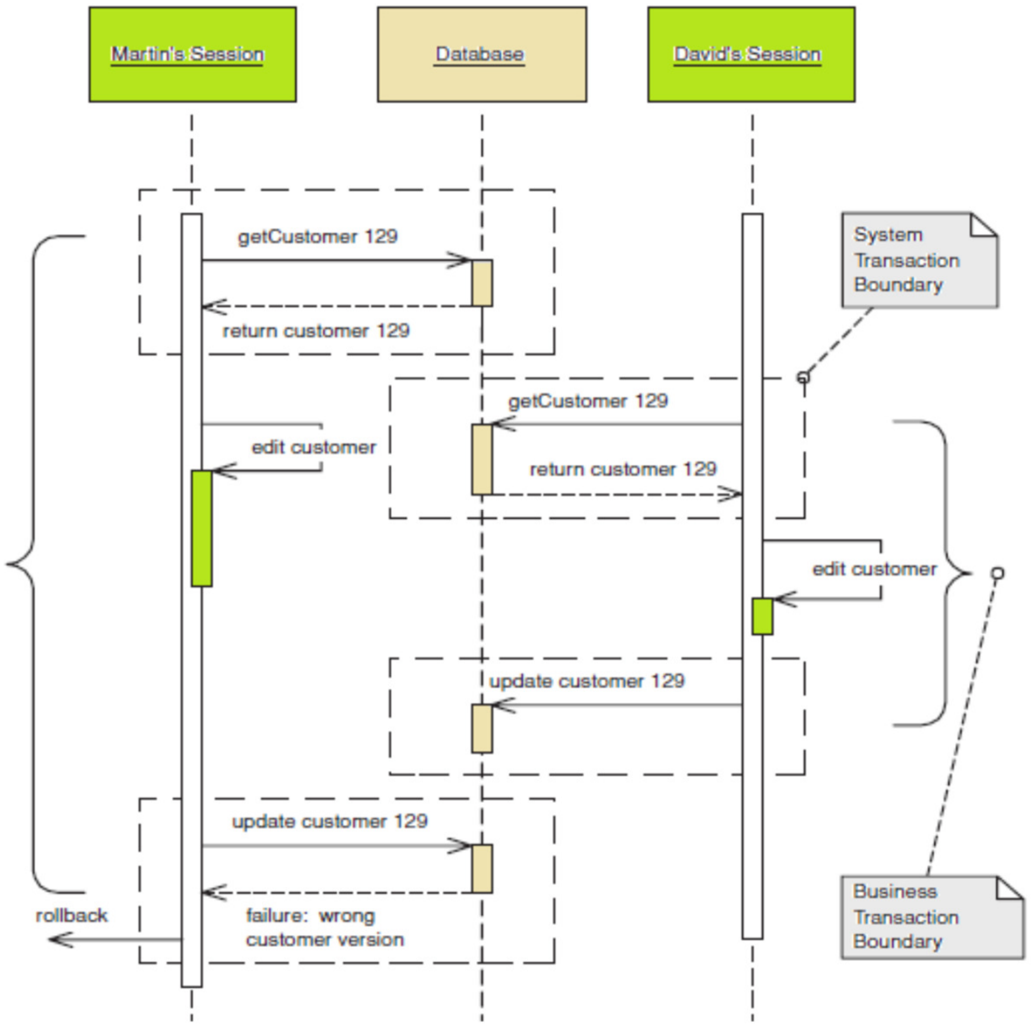
It can be acquired at any time but is valid only during the system transaction in which it is obtained.
Thus, in order that a business transaction not corrupt record data it must acquire an *Optimistic Offline Lock* for each member of its change set during the system transaction in which it applies changes to the database.

The most common implementation is to associate a version number with each record in your system.

When a record is loaded that number is maintained by the session along with all other session state.

Getting the *Optimistic Offline Lock* is a matter of comparing the version stored in your session data to the current version in the record data.

Once the verification succeeds, all changes, including an increment of the version, can be committed. The version increment is what prevents inconsistent record data, as a session with an old version can not acquire the lock.



System Transaction: from the application to the database
Business Transaction: from the user to an application

With an RDBMS data store the verification is a matter of adding the version number to the criteria of any SQL statements used to update or delete a record.

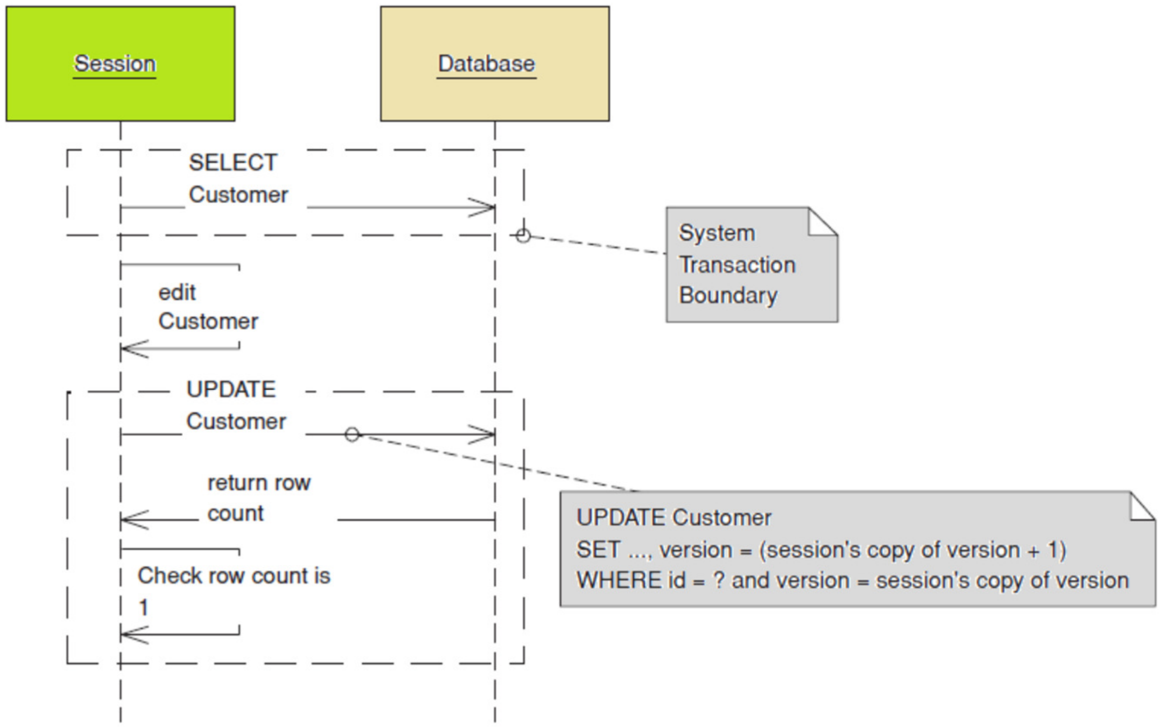
A single SQL statement can both acquire the lock and update the record data.

The final step is for the business transaction to inspect the row count returned by the SQL execution. A row count of 1 indicates success; 0 indicates that the record has been changed or deleted.

With a row count of 0 the business transaction must rollback the system transaction to prevent any changes from entering the record data. At this point the business transaction must either abort or attempt to resolve the conflict and retry.

When informing a user of a failed update due to a concurrency violation a proper application will tell when the record was altered and by whom.

It is a bad idea to use the modification timestamp rather than a version count for your optimistic checks because system clocks are simply too unreliable, especially if you're coordinating across multiple servers.


Session
Database
SELECT Customer
System Transaction Boundary
edit Customer
UPDATE Customer
return row count
Check row count is 1
UPDATE Customer
SET ..., version = (session's copy of version + 1)
WHERE id = ? and version = session's copy of version

UPDATE optimistic check.

When to Use It

Optimistic concurrency management is appropriate when the chance of conflict between any two business transactions is low.

If conflicts are likely it is not user friendly to announce one only when the user has finished his work and is ready to commit. Eventually the user will assume the failure of business transactions and stop using the system.

Pessimistic Offline Lock is more appropriate when the chance of conflict is high or the expense of a conflict is unacceptable.

Optimistic locking is much easier to implement and not prone to the same defects and runtime errors as a *Pessimistic Offline Lock.*

The pessimistic version works well as a complement to its optimistic counterpart, so rather than asking when to use an optimistic approach to conflict avoidance, ask when the optimistic approach alone is not good enough.

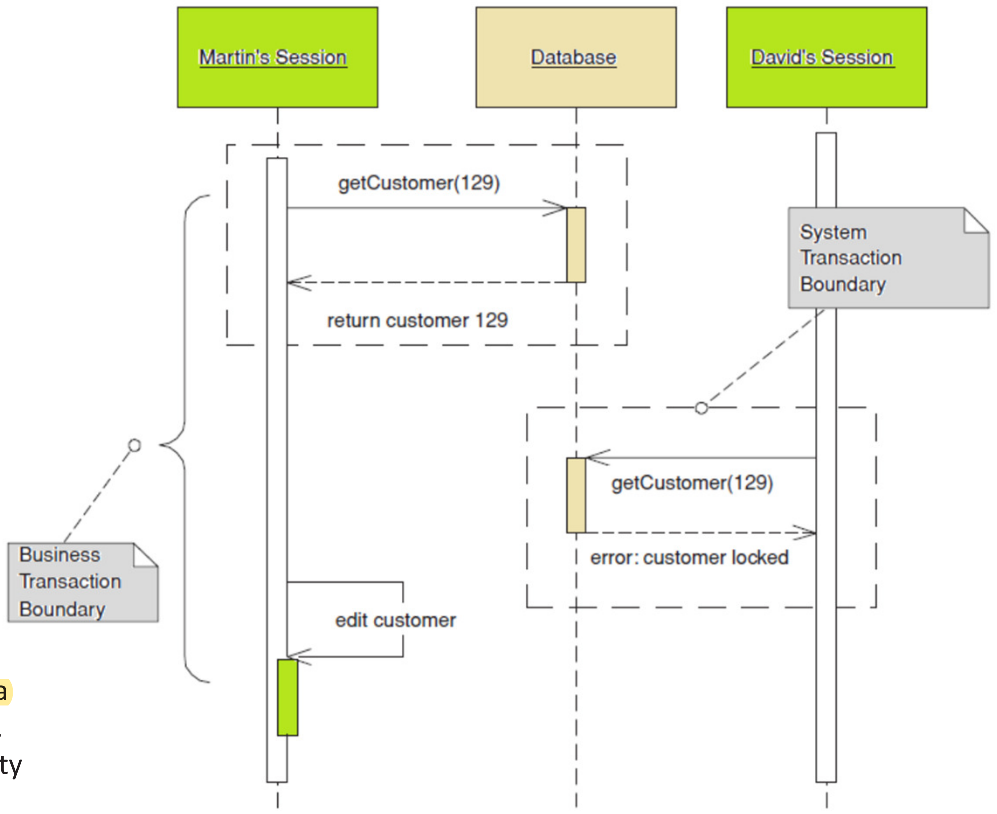
The correct approach to concurrency management will maximize concurrent access to data while minimizing conflicts.

Pessimistic Offline Lock

Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.

The first approach to try is *Optimistic Offline Lock* . However, that pattern has its problems. If several people access the same data within a business transaction, one of them will commit easily but the others will conflict and fail.

Since the conflict is only detected at the end of the business transaction, the victims will do all the transaction work only to find at the last minute that the whole thing will fail and their time will have been wasted.



Pessimistic Offline Lock prevents conflicts by avoiding them altogether. It forces a business transaction to acquire a lock on a piece of data before it starts to use it, so that, most of the time, once you begin a business transaction you can be pretty sure you'll complete it without being bounced by concurrency control.

If you are using *Pessimistic Offline Lock* as a complement to *Optimistic Offline Lock* you need to determine which record types to lock.

As for lock types, the first option is an **exclusive write lock**, which require only that a business transaction acquire a lock in order to edit session data.

This avoids conflict by not allowing two business transactions to make changes to the same record simultaneously. What this locking scheme ignores is the reading of data, so if it is not critical that a view session have the most recent data this strategy will suffice.

If it becomes critical that a business transaction must always have the most recent data, regardless of its intention to edit, use the **exclusive read lock**.

This requires that a business transaction acquire a lock simply to load the record. Clearly such a strategy has the potential to severely restrict a system's concurrency.

A third strategy combines the two lock types to provide the restrictive locking of the exclusive read lock as well as the increased concurrency of the exclusive write lock, called the **read/write lock**.

The relationship of the read and write locks is the key to getting the best of both worlds:

- Read and write locks are mutually exclusive. A record can not be write locked if any other business transaction owns a read lock on it; it can not be read-locked if any other business transaction owns a write lock on it.

- Concurrent read locks are acceptable. The existence of a single read lock prevents any business transaction from editing the record, so there's no harm in allowing any number of sessions as readers once one has been allowed to read.

In choosing the correct lock type think about maximizing system concurrency, meeting business needs, and minimizing code complexity.

When to Use It

Pessimistic Offline Lock is appropriate when the chance of conflict between concurrent sessions is high. A user should never have to throw away work.

Locking is also appropriate when the cost of a conflict is too high regardless of its likelihood.

Locking every entity in a system will almost surely create tremendous data contention problems, so remember that *Pessimistic Offline Lock* is very complementary to *Optimistic Offline Lock* and only use *Pessimistic Offline Lock* where it is truly required.

If you have to use *Pessimistic Offline Lock,* you should also consider a long transaction. Long transactions* are never a good thing, but in some situations they may be no more damaging than *Pessimistic Offline Lock* and much easier to program. Do some load testing before you choose.

*A transaction that spans multiple requests is generally known as a **long transaction**.

Coarse-Grained Lock

Locks a set of related objects with a single lock.

Objects can often be edited as a group. Perhaps you have a customer and its set of addresses. If so, when using the application it makes sense to lock all of these items if you want to lock any one of them.

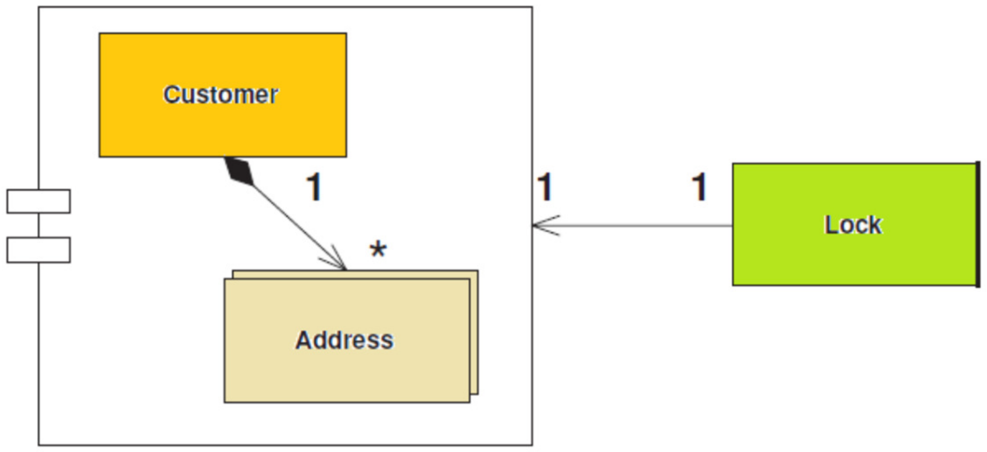
Having a separate lock for individual objects presents a number of challenges.

First, anyone manipulating them has to write code that can find them all in order to lock them. This is easy enough for a customer and its addresses, but it gets tricky as you get more locking groups. And what if the groups get complicated?

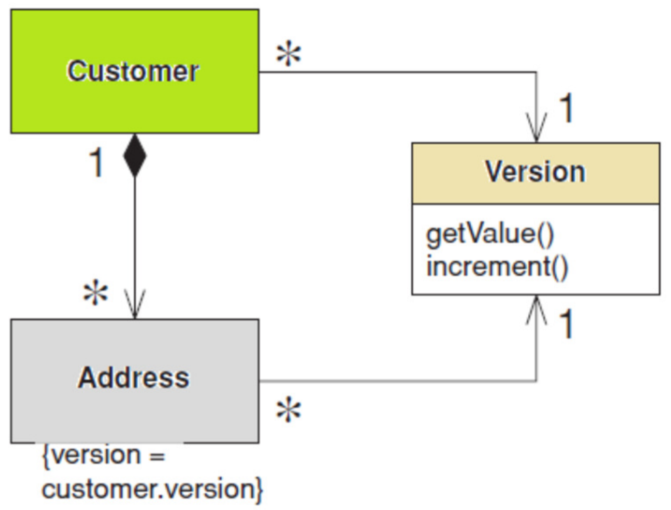
If your locking strategy requires that an object be loaded in order to be locked, such as with *Optimistic Offline Lock* , locking a large group affects performance.

And with *Pessimistic Offline Lock* a large lock set is a management headache and increases lock table contention.

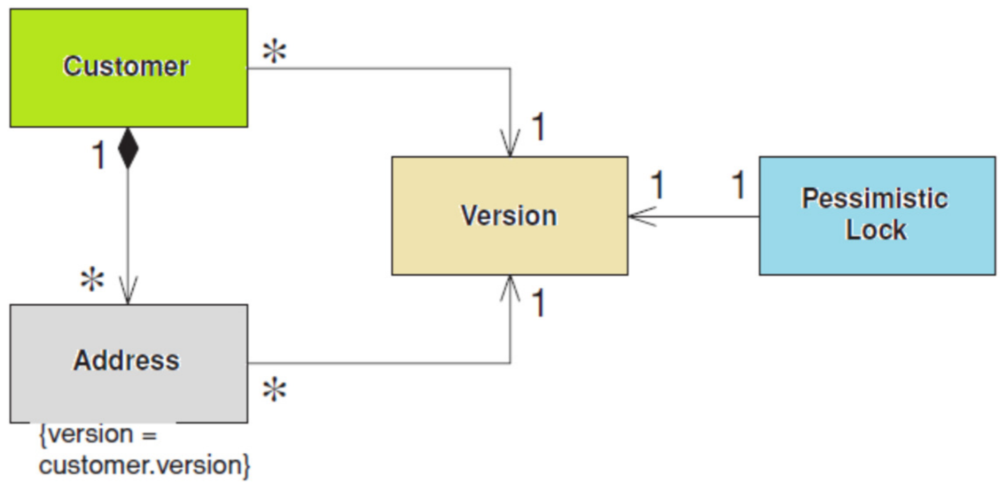
A *Coarse-Grained Lock* is a single lock that covers many objects. It not only simplifies the locking action itself but also frees you from having to load all the members of a group in order to lock them.



The first step in implementing *Coarse-Grained Lock* is to create a single point of contention for locking a group of objects. This makes only one lock necessary for locking the entire set.



Sharing a version.



Locking a shared version.

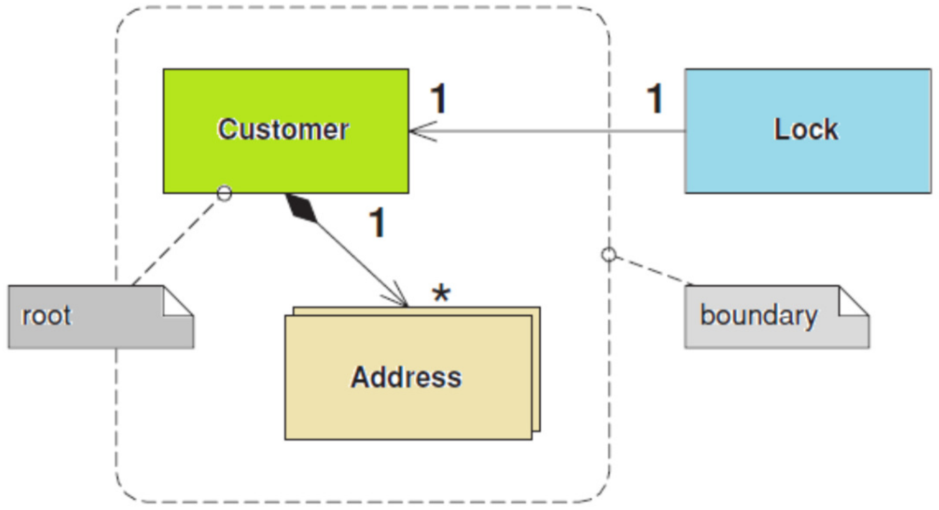
With *Optimistic Offline Lock*, having each item in a group share a version (see Figure above) creates the single point of contention, which means sharing the *same* version.

Incrementing this version will lock the entire group with a **shared lock**.

A shared *Pessimistic Offline Lock* requires that each member of the group share some sort of lockable token, on which it must then be acquired.
As *Pessimistic Offline Lock* is often used as a complement to *Optimistic Offline Lock*, a shared version object makes an excellent candidate for the lockable token role (see Figure above).

Root lock is an alternative to a shared lock. Each aggregate (cluster of associated objects) has a root that provides the only access point to members of the set and a boundary that defines what's included in the set.

The aggregate's characteristics call for a Coarse-Grained Lock, since working with any of its members requires locking all of them. Locking an aggregate yields an alternative to a shared lock (see Figure below). By definition locking the root locks all members of the aggregate. The root lock gives us a single point of contention



Locking the root.

The most obvious reason to use a *Coarse-Grained Lock* is to satisfy business requirements. This is the case when locking an aggregate.
Consider a lease object that owns a collection of assets. It probably doesn't make business sense for one user to edit the lease and another user to simultaneously edit an asset.
Locking either the asset or the lease ought to result in the lease and all of its assets being locked.

Implicit Lock

Allows framework or layer supertype code to acquire offline locks.

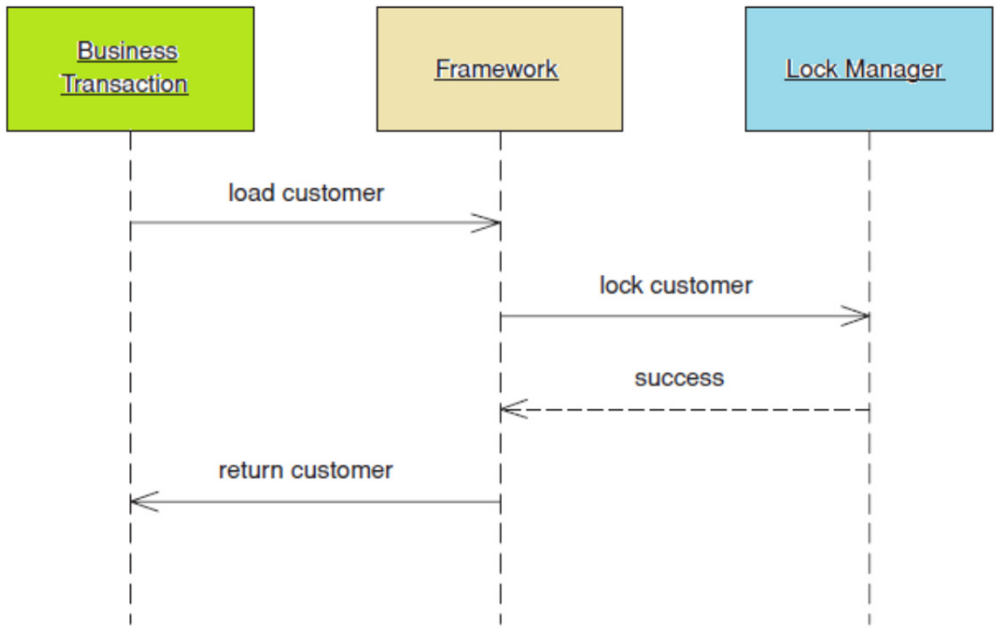
The key to any locking scheme is that there are no gaps in its use.

Forgetting to write a single line of code that acquires a lock can render an entire offline locking scheme useless.

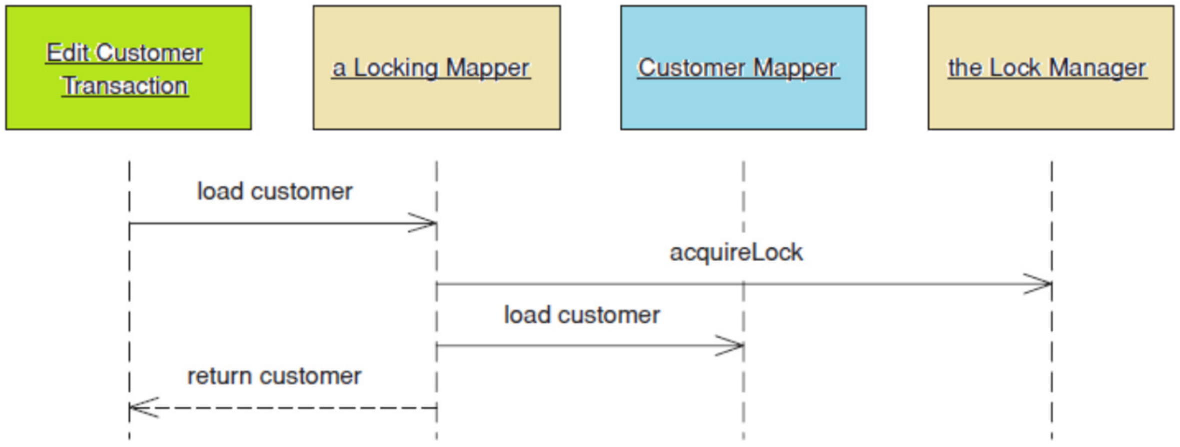
Failing to retrieve a read lock where other transactions use write locks means you might not get up-to-date session data; failing to use a version count properly can result in unknowingly writing over someone's changes.

Generally, if an item might be locked *anywhere* it must be locked *everywhere*. Ignoring its application's locking strategy allows a business transaction to create inconsistent data.

Not releasing locks won't corrupt your record data, but it will eventually bring productivity to a halt.



Most enterprise applications make use of some combination of framework, *Layer Supertypes* , and code generation provides us with ample opportunity to facilitate *Implicit Lock*.



Locking mapper.

Implementing *Implicit Lock* is a matter of factoring your code such that any locking mechanics that *absolutely cannot* be skipped can be carried out by your application framework.