International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:6, No:5, 2012

Concurrent Access to Complex Entities

Cosmin Rablou

***Abstract*—**In this paper we present a way of controlling the concurrent access to data in a distributed application using the Pessimistic Offline Lock design pattern. In our case, the application processes a complex entity, which contains in a hierarchical structure different other entities (objects). It will be shown how the complex entity and the contained entities must be locked in order to control the concurrent access to data.

***Keywords*—**Object-oriented programming, Pessimistic Lock, Design pattern, Concurrent access to data, Processing complex entities

I. Introduction

EVERY distributed business application must deal with the issue of data being accessed and updated by different users at the same time. If there is no control of the concurrency implemented, this can lead to data inconsistencies. [1]

In order to avoid this, a business application must implement a sort of concurrency control. If one user wants to update a record from the database, then it must be prevented that other users change the same record at the same time. This situation is known under the name of synchronizing (or locking) the access of users to the same data.

II. Locking Strategies

There are two different strategies of implementing a concurrency control to the database:

- Optimistic Lock [2] – can be implemented when there is a low chance that different users will access and then change the same entity at the same time. However, when a simultaneous access occurs, the last user that updates the data must choose an action (rollback or overwrite/merge the data).
- Pessimistic Lock [3] – the first user that accesses the entity locks it, so that the other users can't change it. When the user updates the data, the lock is released so that the other users can access it.

The disadvantage of the Pessimistic Lock is the fact that a user cannot change an entity if another user has already locked the same entity. But this is something that a user can easily understand and accept.

Cosmin Rablou has graduated the Faculty of Cybernetics, Statistics and Economic Informatics, Bucharest in 2001. He joined the same year the team at Derdack GmbH, Germany, where his main focus was on the telecommunications and mobile solutions development. In 2007 he joined OctaVIA AG, where he mainly develops SAP applications for telecommunications.

Cosmin Rablou is currently writing his Ph.D. dissertation on design patterns.

However, the disadvantage of the Optimistic Lock is that the changes that the user has done to the entity must be rolled back, if another user changes the data in between. This leads often to frustration, as in this case the changes are lost. The user must start processing the entity from the beginning.

From my experience, when it comes to business data, the better (and the user-friendlier) choice is the Pessimistic Lock.

III. Complex Entity

A business complex entity is an object that contains data from more than one table. The complex entity has a hierarchical structure, as it contains different objects or structures or a collection of objects of the same type. Usually, the entity and the included objects are in a composition relation, which means the included objects are managed solely through the complex entity. When the object that represents the complex entity is destroyed, the contained objects are destroyed, as well. [4]

The business partner in a FICA SAP module is an example of an entity that contains both simple entities and collections of entities. The address, the control data and the status are simple entities. There is a one-to-one relation between the business partner and a simple entity.

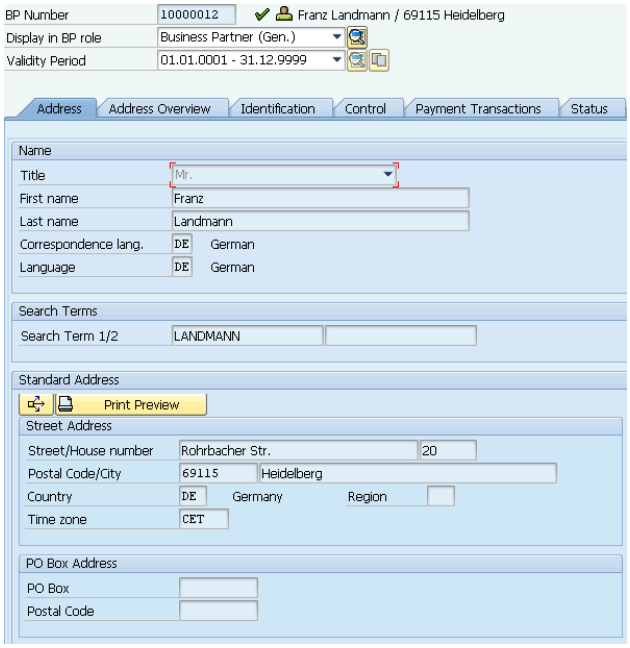

Fig. 1 SAP business partner containing different objects

The bank details and the payment cards represent collections of entities that are included in the business partner. The relationship between the complex entity and the contained entity is in this case one-to-many.

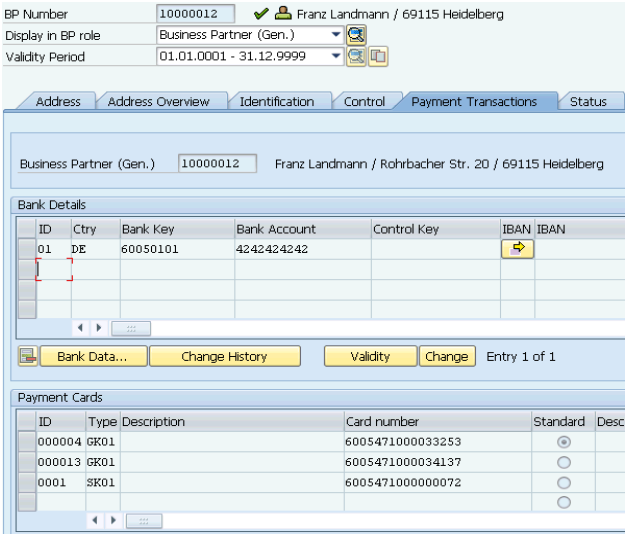

Fig. 2 SAP business partner containing collections of objects

It is important to understand the relation between entities and the database tables.

A simple entity contains at least a record from a table in the database. It is possible that the entity contains also other records, which are bound to the main record by the means of a foreign key.

A collection of entities is an array of entities and therefore is represented by a record set. Each record can have other referencing records, which are connected to the main record through a foreign key.

A complex entity groups simple entities and/or collection of entities and therefore represents a complex structure in the database.

IV. Problem

If more than one user tries to process the same entity at the same time, this can lead to data inconsistency.

Imagine the following scenario:

- An user reads the information about a business entity from the database (as a record from a table)
- A second user requests the access to the same entity
- The first user changes something in the entity and updates the record in the database accordingly
- The second user makes another change and updates the record later than first user.

The changes made by the first user are now lost, as the data saved by the second user did not contain the changes made by the first user.

This situation is known under the name “lost update” and this is only one example of data inconsistency that might occur when different users process data simultaneously.

However, when processing a complex entity, this problem reaches a new level of difficulty.

This is due to the fact that there are different types of entities (objects) that are contained in the main entity. Different users can request access to the same entity. The users can even request access to entities on different levels, by using different applications.

So the problem is to prevent the change of a complex entity when another user is changing at least one of the included entities. Furthermore, when a user changes the complex entity, no other user is allowed to change the included entities.

It must be considered also the fact that the requests to change the complex entity and the included entity might come from different applications.

V. Solution

The data inconsistencies occur when several users are processing the same data at the same time. In order to avoid such situations, the first user that accesses the data must also lock it.

In this case, the first user is the only one who can process the entity and later save the changes in the database. While the first user locks the entity, no other user is allowed to process it. Another user can only process the entity when the first user has finished updating it.

However, as the complex entity contains a hierarchical data structure, it is not enough to lock only a record. When a user is accessing a complex entity, it is necessary to lock both the data directly included in the complex entity and the data belonging to the simple entities contained in the complex entity.

This means that it is necessary to implement Lock and Unlock methods in all entities. Therefore, it is useful to define an entity interface that includes the methods Lock und Unlock. Even better, the interface can also include the Save method. The Save method updates the data in the database and eventually initiates the process of unlocking the entity.

It is important that all applications that process the data use the same type of locking mechanism for the same entity (no matter if complex or simple). If not so, the locking mechanism would only guarantee a proper processing within the application boundaries. Cross-application processing of the same entity would still lead to data inconsistency.

This means the developers of a new application must always consider the locking strategy and mechanism already implemented by existing applications.

VI. Structure

The structure of the presented solution is depicted in the next class diagram.

The following components are included in the class diagram:

- The model – as defined in the MVC-pattern [5], the model contains the business data and rules. When processing a complex entity, the model can be

reduced to a Singleton [6], as the business data and rules are mainly grouped in the complex entity.

- The interface for the entities – defines methods that have to be implemented by the entities, like Lock, Unlock and Save. These methods are implemented both in the complex entity and in the simple entities. The methods for setting and getting the attributes of the entities cannot be included in this interface, as their signature differs from entity to entity.
- The complex entity – contains not only the business data belonging to the complex entity, but also the simple entities and collections of simple entities.
- The simple entity – groups the business data that belong to the simple entity.

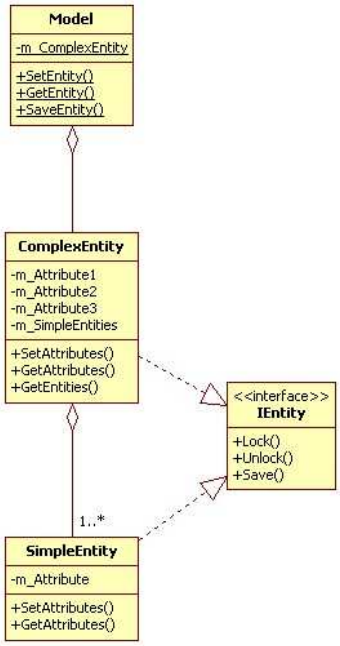

Fig. 3 Class diagram for locking a complex entity

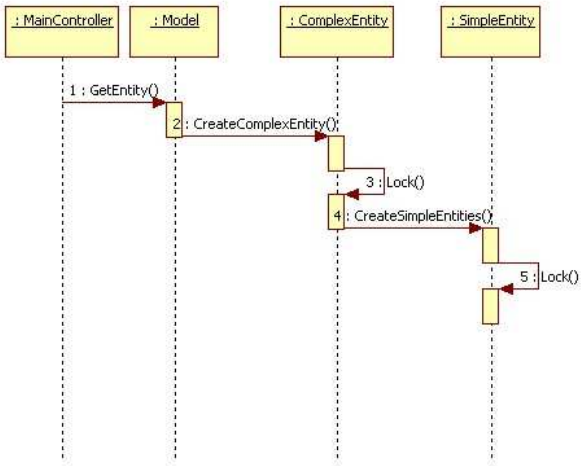

Fig. 4 Sequence diagram for locking the complex entity

When a complex entity is instantiated, it is immediately locked. The complex instance triggers then the instantiation of the simple entities, which then will also be locked.

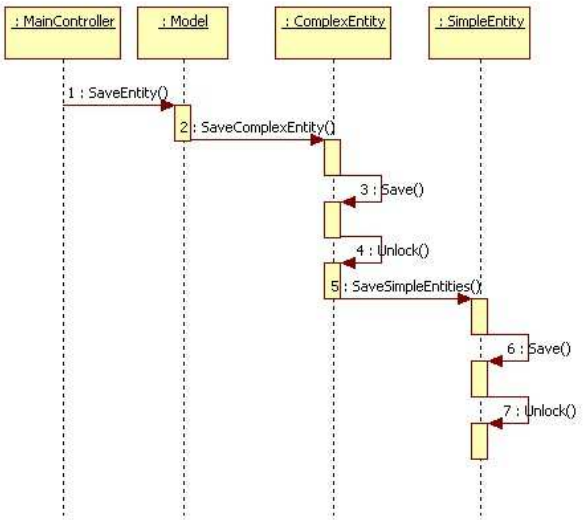

Fig. 5 Sequence diagram for saving and unlocking the complex entity

When the user saves the entity, the data is written to the database and the complex entity is unlocked. Furthermore, the complex entity initiates the saving of the simple entities. The Save method in the simple entities unlocks these, too.

VII. Consequences

The main advantage of using the Pessimistic Lock when processing a complex entity is that it ensures a cross-application concurrency control and thus eliminates the data inconsistencies.

The main downside of the solution is the fact that the data is locked for an undefined time interval.

When a complex entity is processed, the locking affects not only the complex entity, but also the included entities. Thus, no other user can change the locked entities, not even by using another application. This is not a disadvantage, as long as another user actually processes the entity.

However, it is possible that the connection between the server and the user that processes the entity is lost. The entity would remain locked and there would be no way to unlock it.

The solution to this problem is to use the destructor of the entity to unlock it. Even if the connection is interrupted, at some point the session of the user on the server will time out. When the session times out, the server removes all the objects related to the session from the memory and the destructor of the entity is called, thus unlocking the object.

In order to increase the availability of the complex entities, it is possible to define in the application two different ways of acquiring a complex entity:

- A display mode, where the user can see the complex entity, but cannot process it. In this mode the entity is not locked.

- A change mode, where the user locks the complex entity for processing.

The user loads in the beginning the entity in the display mode. He/she must actively switch to the change mode in order to process the entity. This reduces the interval where the entity is locked to the minimum, therefore increasing the availability of the entities.

Another problem that might occur when locking entities is the deadlock. Imagine the scenario where our application locks the complex entity and want to acquire a lock on an included simple entity. In the meantime, an existing application locks first the simple entity and then tries to lock the complex entity. This would lead to a deadlock, as both applications would wait for the other entity, which is already locked.

This kind of situation can be avoided if every application implements the same order of acquiring the lock. In our example, it would be necessary that both applications lock first the complex entity and then the simple entity. Luckily enough, this is also the logical way of acquiring the locks and therefore such collisions are quite rare.

VIII. Implementation

The following code, that shows how to implement the concurrency control for a complex entity is part of an ASP.NET application written in C#. The application uses SQLServer as a database.

The definition of the entity interface contains at least the methods for locking, unlocking and saving the entity.

This interface is implemented in the complex entity and in the contained entities.

```
interface IEntity
{
   bool Lock();
   void Unlock();
   bool Save();
}
```

The constructor of the complex entity has as a parameter the key that uniquely identifies the data included in the object (the primary key). This key can have a null value, when the complex entity does not exist yet in the database, as it is just being created by the application.

The constructor loads the data belonging directly to the complex entity and initializes also the included entities.

```
   public Invoice(int nID)
{
   m_nID = nID;
   m_InvoicePos = new ArrayList();
   LoadInvoicePositions();
}
```

In this case the application does not lock the entities from the beginning. This happens only later, when the user is switching to the change mode.

The complex entity must then lock its own data and then loop over the included entities in order to initiate the locking process for these, too.

The data is locked using special forms of the SQL Select command in a transaction.

Sadly enough, the SQL standard does not offer a general form of the Select command for locking records. However, each database system offers its own command for locking.

For example, Oracle uses the Select command with the clause “For Update”. SQLServer uses the Select command with the clause “With (Updlock, Rowlock)” for the same purpose.

```
public bool Lock()
{
   if (m_nID == 0)
      return false;

   string strSQL = "SELECT * FROM Invoices WITH
(UPDLOCK, ROWLOCK) WHERE ID = @ID";

   m_Connection = new
SqlConnection(m_SqlConnectionString);

   DataSet lDS = new DataSet();

   try
   {
      m_Connection.Open();
      m_Transaction =
m_Connection.BeginTransaction(IsolationLevel.Serializable);
      SqlCommand lCommand = new SqlCommand(strSQL,
m_Connection, m_Transaction);
      lCommand.Parameters.Add("@ID", SqlDbType.Int);
      lCommand.Parameters["@ID"].Value = m_nID;
      lCommand.CommandTimeout = 1;
      m_Adapter = new SqlDataAdapter(lCommand);
      m_Adapter.Fill(lDS);
   }
   catch (Exception err)
   {
      return false;
   }

   for (int i = 0; i <= m_InvoicePos.Count - 1; i++)
   {
      IEntity lInvoicePos = (IEntity)m_InvoicePos[i];
      if (lInvoicePos.Lock() == false)
         return false;
   }
   return true;
}
```

The Save method updates the data in the database, unlocks the data belonging to the entity and finally initiates the Save procedure for the included entities. The Save method of the included entity saves and unlocks the respective entity.

```
public bool Save()
{
   if (m_nID == 0)
   {
```

```
      if (Insert() == false)
         return false;
   }
   else
   {
      if (Modify() == false)
         return false;
   }
   Unlock();
   for (int i = 0; i <= m_InvoicePos.Count - 1; i++)
   {
      IEntity lInvoicePos = (IEntity)m_InvoicePos[i];
      if (lInvoicePos.Save() == false)
         return false;
   }
   return true;
}
```

The destructor of the object must call the method Unlock, to make sure that the entity is unlocked when the object is destroyed. This way, even if the user forgets to properly close the application, the lock will be released when the session on the server times out.

```
~Invoice()
{
   Unlock();
}
```

The Unlock method of the complex entity unlocks its own records and initiates the unlock process of the simple entities.

```
public void Unlock()
{
   if (m_Connection != null)
   {
      // Close connection to unlock the record
      if (m_Connection != null)
      {
         m_Connection.Close();
         m_Connection = null;
      }
      m_Adapter = null;
      m_Transaction = null;
   }
   for (int i = 0; i <= m_InvoicePos.Count - 1; i++)
   {
      IEntity lInvoicePos = (IEntity)m_InvoicePos[i];
      lInvoicePos.Unlock();
   }
}
```

REFERENCES

[1] Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional, 2002, pp. 64-65
[2] Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional, 2002, pp. 416-425
[3] Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional, 2002, pp. 426-437
[4] Cosmin Rablou, "Processing complex entities in MVC applications", *World Academy of Science, Engineering and Technology*, Issue 62, February 2012, Florence, Italy, pp. 2549.
[5] Glenn E. Krasner, Stephen T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80", Journal of Object-Oriented Programming, August/September 1988, pp. 26–49.
[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design patterns: elements of reusable object-oriented software", Addison Wesley, 1994, pp. 127–134.