Politecnico di Milano
A.A. 2015-2016
Software Engineering 2

Code Inspection

Giuseppe Canonaco, Andrea Di Giosaffatte

5 January 2016 – Version 1.1

Table of Contents

1. Assigned Class

1.1. Namespace Pattern and Class Name

Location of the assigned class:

appserver/transaction/jta/src/main/java/com/sun/enterprise/transaction/

Name of the assigned class:

JavaEETransactionManagerSimplified

2. Functional Role Of The Class

2.1. Overview

The JavaEETransactionManagerSimplified provides the services and management functions required to support transaction demarcation and synchronization. It coordinates the completion (commit or rollback) of global transactions across multiple resource managers.
A transaction demarcation defines a section of sequential actions that represent a transactional behavior. To control transaction boundary demarcations and perform transaction operations are used methods like *begin()*, *commit()*, *rollback()*, etc.
Through the *registerSynchronization(Synchronization sync)*[783-797] method of the JavaEETransactionManagerSimplified class, as previously stated, is provided a synchronization mechanism. This method firstly obtain the transaction associated to the transaction manager.

```
789 Transaction tran = getTransaction();
```

Then over that transaction calls registerSynchronization, which, as we can infer from the Transaction interface javadoc, is used to register a Synchronization callback object on the transaction.

```
791 tran.registerSynchronization(sync);
```

The, just discussed, Synchronization object enables a mechanism that allows the interested party(mainly the application server) to be notified before and after the transaction completes.

Note: A resource manager provides the application access to resources. An example of such a resource manager is a relational database server. In glassfish the resource manager is represented by the classes which are implementations of the TransactionalResource interface, indeed, the interface javadoc states: “TransactionalResource interface to be implemented by the resource handlers”. For instance, when the method *enlistXAResource(Transaction tran, TransactionalResource h)* is called, an XAResource is enlisted into the transaction.

2.1. Method 1: begin(int timeout)

Public void begin (int timeout)

This method is used to start a new transaction. In order to accomplish this task a bunch of operations are executed. First of all, it is checked if a transaction already exists, because nested transaction are not supported.

```
812 if ( transactions.get() != null )
        throw new
NotSupportedException(sm.getString("enterprise_distributedtx.notsupported_nested_transaction"));
```

Then a transaction handler is set, the delegate, which will support either only local transactions with a single non-XA resource or distributed transactions with XA resources (this can be inferred having a look at the classes implementing the JavaEETransactionManagerDelegate interface).

```
815 setDelegate();
```

Afterwards, it is immediately checked whether the delegate is associated with any transaction, if so, it has to be thrown a NotSupportedException due to what was already stated about the nested transactions.

```
819 if ( getStatus() != Status.STATUS_NO_TRANSACTION )
820         throw new
NotSupportedException(sm.getString("enterprise_distributedtx.notsupported_nested_transaction"));
```

Now, if the monitoring over the transaction is enabled:

- a read lock over the delegate is acquired [*824 getDelegate().getReadLock().lock();*];
- through initJavaEETransaction(timeout) a new transaction is initialized;
- some operations to get the transaction monitored are performed;
- finally the read lock previously acquired is released [*834 getDelegate().getReadLock().unlock();*];

If the monitoring is not enabled then the transaction is just initialized through the *initJavaEETransaction* method (this method, used also in the monitoring enabled case, creates the transaction and sets it as the current one into the transaction manager, and, of course, bind the transaction manager to the transaction just created).
The transaction monitoring is performed by adding the newly created transaction to the active transaction collection, then an event to signal the transaction activation is fired (have a look at TransactionServiceProbeProvider) and eventually the class name of the component which has performed the current invocation is stored within the transaction(through the invocation manager is obtained the container of the needed information and through that container, which is the component invocation object, the information is retrieved and the inserted into the transaction).

```
827 activeTransactions.add(tx);
828 monitor.transactionActivatedEvent();
829 ComponentInvocation inv = invMgr.getCurrentInvocation();
829 if (inv != null && inv.getInstance() != null) {
830     tx.setComponentName(inv.getInstance().getClass().getName());
}
```

Note: Invocation manager provides interface to keep track of component context on a per-thread basis (this is from InvocationManager interface javadoc). Therefore it allows to keep track of all the information involved in a component method call. The getCurrentInvocation method returns the current component invocation object associated with the current thread , the component invocation object returned contains and is used to obtain information about the component which has been just called. ComponentInvocation is a class used to represent the context of a call over a component on a per-thread basis.

2.2. Method 2: commit( )

Public void commit ( )

This method is used to accomplish the commit procedure over a transaction. There are two kind of transaction: local and distributed. If the transaction manager is associated with a local transaction, then this method commits the transaction on its own by directly calling the commit method over the transaction itself.

```
854 tx.commit();
```

In the case of local transaction, if the monitoring is enabled, before committing the transaction a read lock is acquired over the delegate.

```
850 if(monitoringEnabled){
851             getDelegate().getReadLock().lock();
852             acquiredlock = true;
853 }
```

Otherwise, if the transaction associated to the transaction manager is not local, then the delegate is asked to commit the distributed transaction, and subsequently, through the onTxCompletion method of the JavaEETransactionImpl class, are closed all the opened resources associated to the distributed transaction being committed (see SimpleResource interface javadoc and the implementation of onTxCompletion in JavaEETransactionImpl class).

```
859 getDelegate().commitDistributedTransaction();
862 ((JavaEETransactionImpl)tx).onTxCompletion(true);
```

At last, the transaction and the delegate are eliminated, and if the read lock was acquired is, of course, released.

```
868     setCurrentTransaction(null);
869     delegates.set(null);
870     if(acquiredlock){
871       getDelegate().getReadLock().unlock();
872     }
```

2.3. Method 3: rollback( )

Public void rollback ( )

This method accomplishes the rollback procedure over a transaction. As already stated there are two kind of transaction: local and distributed. If the transaction associated to the transaction manager is local, then is rolled back calling the rollback method over the transaction itself.

```
887 tx.rollback();
```

Moreover, if the monitoring over the transaction is enabled, then, before rolling back the local transaction, is acquired a read lock over the delegate.

```
883         if(monitoringEnabled){
884            getDelegate().getReadLock().lock();
885            acquiredlock = true;
886         }
```

Otherwise, if the transaction is distributed, the roll back procedure is asked to be performed by the delegate, and subsequently all the resources associated to the just rolled back transaction are closed using the method onTxCompletion of the JavaEETransactionImpl class.

```
892 getDelegate().rollbackDistributedTransaction();
895 ((JavaEETransactionImpl)tx).onTxCompletion(false);
```

At last, the transaction and the delegate are removed, and if the read lock was acquired, then, of course, it is released.

```
901      setCurrentTransaction(null);
902      delegates.set(null);
903      if(acquiredlock){
904         getDelegate().getReadLock().unlock();
905      }
```

2.4. Method 4: resume(Transaction tobj)

Public void resume (Transaction tobj)

This method is used to resume the transaction context association of the calling thread with a previously suspended transaction, which is represented by the supplied Transaction object. First of all, it is checked if the current thread has already an associated transaction and if this is the case an IllegalStateException is thrown.

```
988            JavaEETransaction tx = transactions.get();
989            if ( tx != null )
990                  throw new IllegalStateException(
991                        sm.getString("enterprise_distributedtx.transaction_exist_on_currentThread"));
```

Then the Transaction object is checked: if it exists, then it is checked if the object has a valid status. If the status is not valid (rolledback, commited, no_transaction or unknown) an InvalidTransactionException is thrown. This kind of exception is thrown also if the Transaction object doesn't exist.

```
993             if ( tobj != null ) {
994                     int status = tobj.getStatus();
995                     if (status == Status.STATUS_ROLLEDBACK ||
996                             status == Status.STATUS_COMMITTED ||
997                             status == Status.STATUS_NO_TRANSACTION ||
998                             status == Status.STATUS_UNKNOWN) {
999                         throw new InvalidTransactionException(sm.getString(
1000                            "enterprise_distributedtx.resume_invalid_transaction", tobj));
1001                    }
1002            } else {
1003                    throw new InvalidTransactionException(sm.getString(
1004                            "enterprise_distributedtx.resume_invalid_transaction", "null"));
1005            }
```

At last, it is checked if the Transaction object is an instance of the JavaEETransactionImpl class. This class provides optimized local transaction support when a transaction uses zero/one non-XA resource, as its javadoc states. If the object is an instance of JavaEETransactionImpl, it is checked if it is not local: if so, the resume procedure is performed by the delegate. After this check, the Transaction object, opportunely casted, is set as the current transaction.

```
1007            if ( tobj instanceof JavaEETransactionImpl ) {
1008                    JavaEETransactionImpl javaEETx = (JavaEETransactionImpl)tobj;
1009                    if ( !javaEETx.isLocalTx() )
1010                            getDelegate().resume(javaEETx.getJTSTx());
1012                    setCurrentTransaction(javaEETx);
1013            }
```

If the Transaction object is not an instance of the JavaEETransactionImpl class, then the resume procedure is performed by the delegate.

```
1014            else {
1015                    getDelegate().resume(tobj);
1016            }
```

2.5. Method 5: getActiveTransactions( )

Public ArrayList getActiveTransactions( )

This method returns an ArrayList containing objects through which the details of the currently active transactions can be obtained. It is called when transaction monitoring is enabled. First of all, the ArrayList (tranBeans) that will contain all the objects related to the active transactions is created and initialized. The HashTable (txnTable), previously defined at the beginning of the class, is initialized too. Then a clone of the active transactions, contained in a List (activeTransactions) defined at the beginning of the class, is obtained.

```
1128            ArrayList tranBeans = new ArrayList();
1129            txnTable = new Hashtable();
1130            Object[] activeCopy = activeTransactions.toArray();
```

The whole clone is scanned and for each transaction, through the delegate, a TransactionAdminBean object is created. This object, through getter methods, simplifies the access to the relevant monitoring information of each transaction like its id, its status, its elapsed time and, of course, the transaction itself.

```
1131            for(int i=0;i<activeCopy.length;i++){
1132                    try{
1133                            Transaction tran = (Transaction)activeCopy[i];
1134                            TransactionAdminBean tBean =
getDelegate().getTransactionAdminBean(tran);
```

If the TransactionAdminBean object doesn't exist (it shouldn't happen) a warning appears in the logger. Otherwise, the transaction and its id are put in the HashTable. Then the TransactionAdminBean object is added to the ArrayList. If the effective level of the logger would accept a "fine" message, then the information about the adding of the transaction is logged.

```
1135            if (tBean == null) {
1137                    _logger.warning("enterprise_distributedtx.txbean_null" + tran);
1138            } else {
1139              if (_logger.isLoggable(Level.FINE))
1140                    _logger.log(Level.FINE, "TM: Adding txnId " + tBean.getId() + " to txnTable");
1142              txnTable.put(tBean.getId(), tran);
1143              tranBeans.add(tBean);
1144            }
```

During all the operations regarding the construction of the ArrayList, all the exceptions are logged with a "severe" level. At last, the ArrayList containing TransactionAdminBean objects, one for each active transaction, is returned to the caller.

```
1145            }catch(Exception ex){
1146                    _logger.log(Level.SEVERE,
1147                    "transaction.monitor.error_while_getting_monitor_attr", ex);
1148            }
1149        }
1150       return tranBeans;
1151   }
```

3. Checklist

3.1. Issues Found by Applying the Checklist

This section provides a list of the issues found by applying the checklist. For each not-fulfilled-point in the checklist, the relative lines of code are listed.

- Point 8: on line 823 there is an exceeding space, indeed, instead of four there are five indentation spaces;
- Point 11: on line 812 there is an if statement which executes only one instruction and this instruction is not wrapped by curly braces; the same for line 819, 989, 1009 and 1139;
- Point 12: after line 291 there must be a blank line or a comment and the same after line 157;
- Point 13: line 813 exceed 80 characters but not 120;

- Point 15: on line 984 there is a line break that doesn't occurs after a comma or an operator; the same for line 990, 999 and 1003;
- Point 17: line 823 is not aligned with the beginning of the expression at the same level as the previous block, the same for line 833, 836, 860, 867, 893, 900, 1002, 1138 and 1145;

- Point 18: line 824 the comment doesn't explain why that lock is needed, the same for line 834, 851, 871, 884 and 904; the lines 822, 839, 874 don't explain what IASRI is;

- Point 25D: the static variables are not the before instance variables see lines 112 and 134;

- Point 25E: instance variable are not in the right order. For instance there are first protected, then private, and then again protected instance variables. See lines 103, 105 and 107;

- Point 27: there is code duplication. Instead of the lines 868 and 869 there would have been the clearThreadTx() method invocation. The same can be said of lines 901 and 902;

- Point 33: on line 829 there is a declaration which is not at the beginning of the relative block;

- Point 52-53: within the begin method there is a try construct without a catch and the methods within the try block don't raise exceptions; for the commit and rollback methods, again, we have try blocks without catches but this time exceptions can be raised (for instance by commit, rollback, commitDistributedTransaction,rollbackDistributedTransaction). The fact that exceptions can be raised and there are try finally construct is, maybe, due to the fact that there are resources that need to be closed and there is no need to handle the exceptions that might be thrown.

3.2. Reference Documents

- Assignment 3 (Code Inspection) which can be retrieved on the beep page of the course.

3.3. Used Tools

The tools used to create the Code Inspection document are:

- Microsoft Office Word 2011: to redact and to format this document.

For redacting and writing this document we have spent **20 hours** per person.