

POLITECNICO DI MILANO
Computer Science and Engineering
Software Engineering 2 Project



Code Inspection

Authors: Greta Ghiotti
Raffaele Malvermi
Mirco Mutti

Version 1.0

Date 5/01/16

Reference Professor: Mirandola Raffaella

Index

Classes assigned	3
Functional role of assigned set of classes	4
List of issues	9
Class inspection	9
Method Commit()	11
Method rollback(boolean force)	15
Method register_synchronization(Synchronization sync)	20
Issues Statistics	24

Classes assigned

The class that was assigned to us is the TopCoordinator class.

Methods:

- Name: commit()
Start Line: 2041
Location:
appserver/transaction/jts/src/main/java/com/sun/jts/CosTransactions/TopCoordinator.java
- Name: rollback(boolean force)
Start Line: 2207
Location:
appserver/transaction/jts/src/main/java/com/sun/jts/CosTransactions/TopCoordinator.java
- Name: register_synchronization(Synchronization sync)
Start Line: 2401
Location:
appserver/transaction/jts/src/main/java/com/sun/jts/CosTransactions/TopCoordinator.java

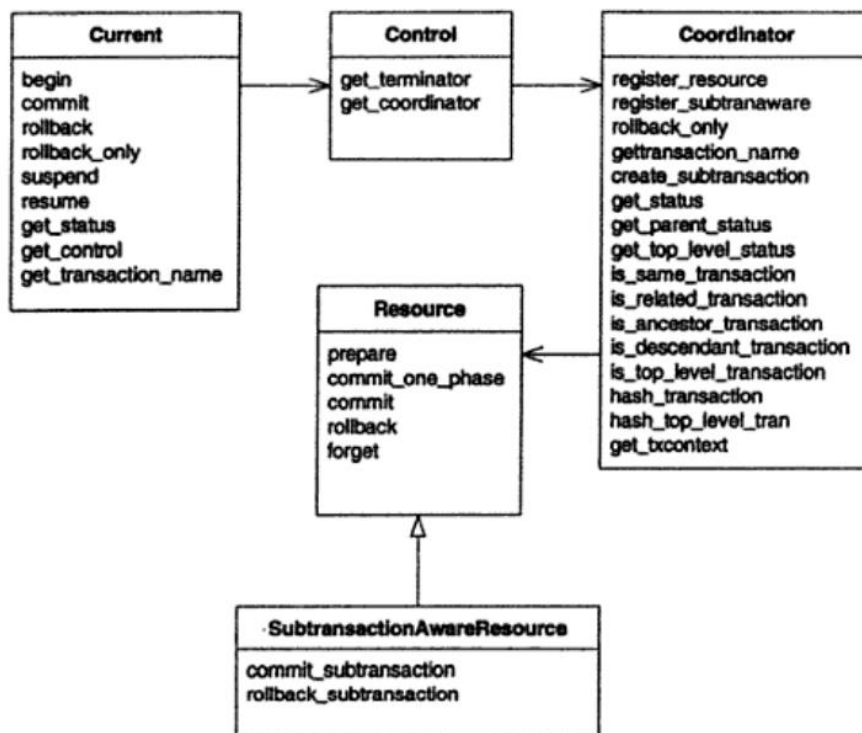
Functional role of assigned set of classes

The System we have to inspect adopts the standard two phase commit protocol which is implemented in the CosTransaction package. This package allows the system to manage distributed transactions using CORBA architecture, including the synchronization of all the threads associated with a certain transaction.

This package also includes the class TopCoordinator: in the following paragraphs we report a description of its functional role together with a brief description on how we have inferred our conclusions.

The TopCoordinator is bijective associated to a unique transaction, more specifically the Javadoc reports that the class is the *"implementation of the standard Coordinator interface that is used for top-level transactions"*. This object manages the relationship between the relative transaction and its resources; in particular, as reported in the class Javadoc overview, *"it allows Resources to be registered for participation in a top-level transaction"*.

In the System implementation each transaction could have a set of sub-transactions and, in according to this, top-level transactions are organized hierarchically in a tree data structure. Then, for each transaction tree we have a corresponding Coordinator tree.



To manage the tree structure each TopCoordinator owns the references of the SuperiorInfo and NestingInfo objects: the first one includes the information of the parent Coordinator, the second includes the reference of each sub-node in the tree and also the entire sequence of ancestors. But, in certain methods implementation, the access to the tree structure it's even simpler: each TopCoordinator cares only about its set of participant resources that includes also sub-transactions (as we can notice in the class diagram above); it's the class CoordinatorResourceImpl that make it possible: it allows transaction to be seen as resources by the TopCoordinator.
























As we can see in the diagram below, this class implements a set of methods belonging to three different functionalities:

- Two Phase Commit Protocol operations (commit. rollback. ...)
- Multi-thread synchronization managing oper: "Distributed Object Architectures with CORBA";
- Failure recovery operations

Hennery Balen, Mark Elenko, Jan Jones,
Gordon Palumbo;
Cambridge University Press.

The Javadoc reports:

*"In addition the TopCoordinator recovery interface c
superior Coordinator is lost after a transaction is prepared. As an instance of this class may
be accessed from multiple threads within a process, serialisation for thread-safety is
necessary in the implementation. The information managed should be reconstructible in
the case of a failure."*

 TopCoordinator
<div>  ~String name </div> <div>  ~RegisteredResources participants </div> <div>  ~RegisteredSyncs synchronizations </div> <div>  ~SuperiorInfo superInfo </div> <div>  ~NestingInfo nestingInfo </div> <div>  ~TransactionState tranState </div> <div>  ~CoordinatorLog logRecord </div> <div>  ~CompletionHandler terminator </div> <div>  ~boolean registered </div> <div>  ~boolean registeredSync </div> <div>  ~boolean root </div> <div>  ~boolean rollbackOnly </div> <div>  ~boolean dying </div> <div>  ~boolean temporary </div> <div>  ~int hash </div> <div>  ~static <u>Logger logger</u> </div> <div>  ~Vector recoveryCoordinatorList </div> <div>  ~CoordinatorSynchronizationImpl coordSyncImpl </div> <div>  ~boolean delegated </div> <div>  ~String logPath </div> <div>  ~static <u>String[] resultName</u> </div> <div>  ~static <u>Any emptyData</u> </div>

```

~TopCoordinator()
~TopCoordinator(int timeOut)
~TopCoordinator(int timeOut, GlobalTID globalTID, Coordinator superior, boolean temporary)
+void doFinalize()
~void reconstruct(CoordinatorLog log)
~void delegated_reconstruct(CoordinatorLog log, String logPath)
~Status recover(boolean[] isRoot)
+Status get_status()
+Status get_parent_status()
+Status get_top_level_status()
+boolean is_same_transaction(Coordinator other)
+boolean is_related_transaction(Coordinator other)
+boolean is_root_transaction()
+boolean is_ancestor_transaction(Coordinator other)
+boolean is_descendant_transaction(Coordinator other)
+boolean is_top_level_transaction()
+int hash_transaction()
+int hash_top_level_tran()
+RecoveryCoordinator register_resource(Resource res)
+void register_subtran_aware(SubtransactionAwareResource sares)
+void rollback_only()
+String get_transaction_name()
+Control create_subtransaction()
+otid_t getGlobalTID()
+GlobalTID getGlobalTid()
+int getParticipantCount()
+long getLocalTID()
~CoordinatorImpl replyAction(int[] action)
~Long setPermanent()
+boolean isRollbackOnly()
~boolean isActive()
~boolean hasRegistered()
+TransIdentity[] getAncestors()
~boolean addChild(CoordinatorImpl child)
~boolean removeChild(CoordinatorImpl child)
~Vote prepare()
~void commit()
~void rollback(boolean force)
+void register_synchronization(Synchronization sync)
~void beforeCompletion()
~void afterCompletion(Status status)
~void setTerminator(CompletionHandler term)
~Coordinator getParent()
~Coordinator getSuperior()
~CompletionHandler getTerminator()
~void directRegisterResource(Resource res)
+PropagationContext get_txcontext()
~void cleanUpEmpty(CoordinatorImpl parent)
~boolean commitOnePhase()
+int hashCode()
+boolean equals(java.lang.Object other)

```

This brief description about the functional role of the inspected class is based on reading the source code of the whole TopCoordinator class, all the classes involved in the inspected methods and the relative Javadocs.

We also have found some documentation about the concepts underlying this implementation:

- <http://www.erlang.org/doc/apps/cosTransactions>
- <https://docs.oracle.com/database>
- <https://courses.cs.washington.edu/courses/csep545/01wi/lectures/class7.pdf>
- The set of slide "Advanced Databases. Distributed Commit and Recovery Protocols" of the Database2 course by Mario Braga
- "Distributed Object Architectures with CORBA"
Hennery Balen, Mark Elenko, Jan Jones, Gordon Palumbo; *Cambridge University Press*

List of issues

Class inspection

We are going to show only the parts of the class TopCoordinator that showcased errors.
Then, we are going to analyse the methods that we have been assigned.

```
63 package com.sun.jts.CosTransactions;
64
65 import java.util.*;
66
67 import org.omg.CORBA.*;
68 import org.omg.CosTransactions.*;
69
70 import com.sun.jts.codegen.otsidl.*;
71 import com.sun.jts.jtsxa.OTSResourceImpl;
72 //import com.sun.jts.codegen.otsidl.JCoordinatorHelper;
73 //import com.sun.jts.codegen.otsidl.JCoordinatorOperations;
74 //import java.io.PrintStream;
75 //import java.util.Vector;
76
77 //import com.sun.enterprise.transaction.OTSResourceImpl;
78 //import com.sun.enterprise.transaction.SynchronizationImpl;
79
80 import com.sun.jts.trace.*;
81
82 import java.util.logging.Logger;
83 import java.util.logging.Level;
84 import com.sun.logging.LogDomains;
85 import com.sun.jts.utils.LogFormatter;
86
87 /**
88  * The TopCoordinator interface is our implementation of the standard
89  * Coordinator interface that is used for top-level transactions. It allows
90  * Resources to be registered for participation in a top-level transaction.
91  * In addition the TopCoordinator recovery interface can be used if the
```

line 72-78: several import statements commented out without reporting the reason

```
122 public class TopCoordinator extends CoordinatorImpl {
123     String name = null;
124     RegisteredResources participants = null;
125     RegisteredSyncs synchronizations = null;
126     SuperiorInfo superInfo = null;
127     NestingInfo nestingInfo = null;
128     TransactionState tranState = null;
129     CoordinatorLog logRecord = null;
130     CompletionHandler terminator = null;
131     boolean registered = false;
132     boolean registeredSync = false;
133     boolean root = true;
134     boolean rollbackOnly = false;
135     boolean dying = false;
136     boolean temporary = false;
137     int hash = 0;
138
139     /*
140      * Logger to log transaction messages
141      */
142     static Logger _logger = LogDomains.getLogger(TopCoordinator.class, LogDomains.TRANSACTION_LOGGER);
143     // added (Ram J) for memory Leak fix.
144     Vector recoveryCoordinatorList = null;
145     CoordinatorSynchronizationImpl coordSyncImpl = null;
146
147     // added (sankar) for delegated recovery support
148     boolean delegated = false;
149     String logPath = null;
150 }
```

Javadoc updated only until version 0.02

line 122: TopCoordinator extends CoordinatorImpl instead of JCoordinatorPOA (as written in Javadocs)

line 135: the attribute dying is never used in this class

lines 900/918: both get_parent_status() and get_top_level_status() methods return the same result, and this is visible in the javadoc, too

line 2158: the definition of the destroy() method misses in the class (on the other hand, it might be a private method and its presence is not needed in the javadoc)

```
899  */
900  public Status get_parent_status() {
901      Status result = get_status();
902      return result;
903  }
904
905  /**
906   * Gets the local state of the transaction.
907   * For a top-level transaction this operation is equivalent
908   * to the get_status method.
909   * This operation references no instance variables and so can be
910   * implemented locally in a proxy class.
911   *
912   * @param
913   *
914   * @return The status of the transaction.
915   *
916   * @see
917   */
918  public Status get_top_level_status() {
919
920      Status result = get_status();
921      return result;
922  }
```

Class Declaration

line 142: _logger might be declared at the beginning of the Class definition

lines 900/918: both get_parent_status() and get_top_level_status() methods return the same result, and this is visible in the javadoc, too

addChild, commit, rollback are big methods

Method Commit()

```
2021  /**
2022  * Directs the TopCoordinator to commit the transaction.
2023  * The TopCoordinator directs all registered Resources to commit. If any
2024  * Resources raise Heuristic exceptions, the information is recorded,
2025  * and the Resources are directed to forget the transaction before the
2026  * Coordinator returns a heuristic exception to its caller.
2027  *
2028  * @param
2029  *
2030  * @return
2031  *
2032  * @exception HeuristicMixed A Resource has taken an heuristic decision
2033  * which has resulted in part of the transaction being rolled back.
2034  * @exception HeuristicHazard Indicates that heuristic decisions may have
2035  * been taken which have resulted in part of the transaction
2036  * being rolled back.
2037  * @exception NotPrepared The transaction has not been prepared.
2038  *
2039  * @see
2040  */
2041  void commit() throws HeuristicMixed, HeuristicHazard, NotPrepared {
2042
2043      // Until we actually distribute prepare flows, synchronize the method.
2044
2045
2046      synchronized(this) {
2047          if(_logger.isLoggable(Level.FINE))
2048          {
2049              _logger.log(Level.FINE,"TopCoordinator","commit()",
2050                  "Within TopCoordinator.commit()"+"GTID is :"+
2051                  superInfo.globalTID.toString());
2052          }
2053
2054          // If the TopCoordinator voted readonly,
2055          // produce a warning and return.
2056
2057          if (tranState.state == TransactionState.STATE_PREPARED_READONLY) {
2058              return;
2059          }
2060
2061          // GDH
2062          // If the TopCoordinator has already completed due to recovery
2063          // resync thread, return. (Note there is no
2064          // need to deal with state ROLLED_BACK here as nothing should have
2065          // caused us to enter that state and subsequently receive a commit.
2066          // However the opposite cannot be said to be true as presumed abort
2067          // can cause a rollback to occur when
2068          // replay_completion is called on a transaction that
2069          // has gone away already.
2070
2071          if (tranState.state == TransactionState.STATE_COMMITTED) {
2072              return;
2073          }
2074
2075          // If the TopCoordinator is in the wrong state, return immediately.
2076
2077          if (!tranState.setState(TransactionState.STATE_COMMITTING)) {
2078              _logger.log(Level.SEVERE,"jts.transaction_wrong_state","commit");
2079              String msg = LogFormatter.getLocalizedMessage(_logger,
2080                  "jts.transaction_wrong_state",
2081                  new java.lang.Object[] { "commit"});
2082              throw new org.omg.CORBA.INTERNAL(msg);
2083              //NotPrepared exc = new NotPrepared();
2084              //Commented out as code is never executed
2085              //throw exc;
2086          }
2087
2088          // Release the lock before proceeding with commit.
2089
2090      }
2091
2092      // Commit all participants. If a fatal error occurs during
2093      // this method, then the process must be ended with a fatal error.
2094
2095      Throwable heuristicExc = null;
```

```

2096     Throwable internalExc = null;
2097     if (participants != null) {
2098         try {
2099             participants.distributeCommit();
2100         } catch (Throwable exc) {
2101             if (exc instanceof HeuristicMixed ||
2102                 exc instanceof HeuristicHazard) {
2103                 heuristicExc = exc;
2104             } else if (exc instanceof INTERNAL) {
2105
2106                 // ADDED(Ram J) percolate any system exception
2107                 // back to the caller.
2108                 internalExc = exc; // throw (INTERNAL) exc;
2109             } else {
2110                 _logger.log(Level.WARNING, "", exc);
2111             }
2112         }
2113     }
2114
2115     // The remainder of the method needs to be synchronized.
2116
2117     synchronized(this) {
2118
2119         // Record that objects have been told to commit.
2120
2121         // Set the state
2122
2123         if (!tranState.setState(TransactionState.STATE_COMMITTED)) {
2124             _logger.log(Level.SEVERE, "jts.transaction_wrong_state", "commit");
2125             String msg = LogFormatter.getLocalizedMessage(_logger,
2126                 "jts.transaction_wrong_state",
2127                 new java.lang.Object[] { "commit"});
2128             throw new org.omg.CORBA.INTERNAL(msg);
2129         }
2130
2131         // Clean up the TopCoordinator after a commit. In the case where
2132         // the TopCoordinator is a root, the CoordinatorTerm object must be
2133         // informed that the transaction has completed so that if another
2134         // caller has committed the transaction the object normally
2135         // responsible for terminating the transaction can take the
2136         // appropriate action. NOTE: This may DESTROY the TopCoordinator
2137         // object so NO INSTANCE VARIABLES should be referenced after the
2138         // call. In the case where the TopCoordinator is a subordinate, the
2139         // CoordinatorResource object must be informed that the transaction
2140         // has been completed so that it can handle any subsequent requests
2141         // for the transaction.
2142
2143         if (terminator != null) {
2144             terminator.setCompleted(false, (heuristicExc != null || internalExc != null));
2145         }
2146
2147         /* commented out (Ram J) for memory leak fix.
2148         // If there are no registered Synchronization objects,
2149         // there is nothing left to do, so get the RecoveryManager
2150         // to forget about us, then self-destruct.
2151
2152         if (!root && (synchronizations == null ||
2153             !synchronizations.involved())
2154             ) {
2155             RecoveryManager.removeCoordinator(superInfo.globalTID,
2156                 superInfo.localTID,
2157                 false);
2158             destroy();
2159         }
2160         */
2161
2162         // added (Ram J) for memory leak fix
2163         // if subordinate, send out afterCompletion. This will
2164         // destroy the CoordinatorSynchronization and coordinator.
2165         if (!root) {
2166             afterCompletion(Status.StatusCommitted);
2167         }
2168
2169         /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
2170         /* NO INSTANCE VARIABLES MAY BE ACCESSED FROM THIS POINT ON.          */
2171         /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/

```

```

2174 // If there was heuristic damage, report it.
2175
2176 if (heuristicExc != null) {
2177     if (heuristicExc instanceof HeuristicMixed) {
2178         throw (HeuristicMixed) heuristicExc;
2179     } else {
2180         throw (HeuristicHazard) heuristicExc;
2181     }
2182 } else if (internalExc != null) {
2183     throw (INTERNAL) internalExc;
2184 }
2185 }
2186

```

Naming conventions

- line 2041: HeuristicMixed, HeuristicHazard and NotPrepared should finish with "Exception" (HeuristicMixedException, as an example)
- line 2082: INTERNAL should finish with the word 'Exception' (InternalException)
- line 2082: INTERNAL is defined using the package path, but the package has already been imported
- line 2128: INTERNAL is defined using the package path, but the package has already been imported
- line 2166: afterCompletion(...) method is named without using a verb

Indentation

- line 2046-2052: indentation made using tabs
- line 2079-2084: indentation made using tabs
- line 2079/2082: wrong insertion of ' ' symbols to indent code lines
- line 2125: wrong insertion of ' ' symbols to indent code lines
- line 2125-2128: indentation made using tabs
- line 2162-2167: wrong indentation (too much ' ' symbols)

Braces

- line 2047-2052: Allman style used once in the method

File Organization

- line 2045: blank line is not necessary
- line 2083: a blank line should be between code and comment lines
- line 2086: a blank line should be between code and comment lines

line 2108: missing blank lines before and after the code line "internalExc = exc;"
line 2154: wrong newline

Comments

line 2043: comment line is not clear (when the synchronization has to begin and finish?)
line 2055: comment line states that the following branch should produce a warning message, but the code below doesn't do this
line 2106: comment lines are referred to an action done in 2183 (throw INTERNAL)
line 2108: code line commented out without rationale
line 2119: comment line not followed by relative code

Initializations and Declaration

line 2125: variable msg should be declared at the beginning of the if-clause block

Computation, Comparisons and Assignments

line 2100-2110: brutish use of instanceof comparison; several catch branches are better
line 2166: Status.StatusCommitted is not an attribute of the class Status
(STATUS_COMMITTED is)
line 2095-2096: brutish programming, it would be better to declare two distinct variables for the heuristic exceptions because he knows exactly the type

Exceptions

line 2099: NotPrepared exceptions are not expected even if distributeCommit() method throws it
line 2104: expected INTERNAL exception, but distributeCommit() method doesn't throw it

Method rollback(boolean force)

```
2187
2188 /**
2189  * Directs the TopCoordinator to roll back the transaction.
2190  * The TopCoordinator directs all registered Resources to rollback.
2191  * If any Resources raise Heuristic exceptions,
2192  * the information is recorded, and the Resources are directed
2193  * to forget the transaction before the
2194  * Coordinator returns a heuristic exception to its caller.
2195  *
2196  * @param force Indicates that the transaction must rollback regardless.
2197  *
2198  * @return
2199  *
2200  * @exception HeuristicMixed A Resource has taken an heuristic decision
2201  *       which has resulted in part of the transaction being committed.
2202  * @exception HeuristicHazard Indicates that heuristic decisions may
2203  *       have been taken which have resulted in part of the transaction
2204  *       being rolled back.
2205  * @see
2206  */
2207 void rollback(boolean force) throws HeuristicMixed, HeuristicHazard {
2208
2209     // Until we actually distribute prepare flows, synchronize the method.
2210
2211     synchronized(this){
2212         if(_logger.isLoggable(Level.FINE))
2213         {
2214             _logger.logp(Level.FINE,"TopCoordinator","rollback()",
2215                 "Within TopCoordinator.rollback() :"+"GTID is : "+
2216                 superInfo.globalTID.toString());
2217         }
2218
2219         // If the transaction has already been rolled back, just return.
2220
2221         if (tranState == null) {
2222             return;
2223         }
2224
2225         // GDH
2226         // If the TopCoordinator has already completed (eg due to
2227         // recovery resync thread and this is now running on
2228         // the 'main' one) we can safely ignore the error
2229
2230         if (tranState.state == TransactionState.STATE_ROLLED_BACK) {
2231             return;
2232         }
2233
2234         // GDH
2235         // The state could even be committed, which can be OK if it was
2236         // committed, and thus completed, when the recovery thread asked
2237         // the superior about the txn. The superior would
2238         // no longer had any knowledge of it. In this case, due to presumed
2239         // abort, the recovery manager would then
2240         // now default to aborting it.
2241         // In this case if the TopCoordinator has committed already
2242         // we should also just return ignoring the error.
2243
2244         if (tranState.state == TransactionState.STATE_COMMITTED) {
```

```

2245         return;
2246     }
2247
2248     // If this is not a forced rollback and the coordinator
2249     // has prepared or is in an inappropriate state, do not continue
2250     // and return FALSE.
2251
2252     if (!force && ((tranState.state ==
2253         TransactionState.STATE_PREPARED_SUCCESS) ||
2254         (!tranState.setState(
2255             TransactionState.STATE_ROLLING_BACK))
2256         )) {
2257         return;
2258     }
2259
2260     // We do not care about invalid state changes as we are
2261     // rolling back anyway. If the TopCoordinator is
2262     // temporary, we do not change state as this would
2263     // cause a log force in a subordinate, which is not required.
2264
2265     if( !temporary &&
2266         !tranState.setState(TransactionState.STATE_ROLLING_BACK)) {
2267         if( _logger.isLoggable(Level.FINE)) {
2268             _logger.log(Level.FINE,
2269                 "TopCoordinator - setState(TransactionState.STATE_ROLLED_BACK) returned false");
2270         }
2271     }
2272
2273     // Rollback outstanding children. If the NestingInfo instance
2274     // variable has not been created, there are no
2275     // children to rollback.
2276
2277     if (nestingInfo != null) {
2278         nestingInfo.rollbackFamily();
2279     }
2280
2281     // Release the lock before proceeding with rollback.
2282
2283 }
2284
2285 // Roll back all participants. If a fatal error occurs during
2286 // this method, then the process must be ended with a fatal error.
2287
2288 Throwable heuristicExc = null;
2289 if (participants != null) {
2290     try {
2291         participants.distributeRollback(false);
2292     } catch(Throwable exc) {
2293
2294         if (exc instanceof HeuristicMixed ||
2295             exc instanceof HeuristicHazard) {
2296             heuristicExc = exc;
2297         } else if (exc instanceof INTERNAL) {
2298             // ADDED (Ram J) percolate up any system exception.
2299             throw (INTERNAL) exc;
2300         } else {
2301             _logger.log(Level.WARNING, "", exc);
2302         }
2303     }
2304 }

```



```

2303     }
2304 }
2305
2306 // The remainder of the method needs to be synchronized.
2307
2308 synchronized(this) {
2309
2310     // Set the state. Only bother doing this if the coordinator
2311     // is not temporary.
2312
2313     if (!temporary &&
2314         !tranState.setState(TransactionState.STATE_ROLLED_BACK)) {
2315         if(_logger.isLoggable(Level.FINE)) {
2316             _logger.log(Level.FINE,
2317                 "TopCoordinator - setState(TransactionState.STATE_ROLLED_BACK) returned false");
2318         }
2319     }
2320
2321     // Clean up the TopCoordinator after a rollback.
2322     // In the case where the TopCoordinator is a root,
2323     // the CoordinatorTerm object must be informed that the transaction
2324     // has completed so that if another caller has rolled back
2325     // the transaction (time-out for example) the object normally
2326     // responsible for terminating the transaction can take the
2327     // appropriate action. NOTE: This may DESTROY
2328     // the TopCoordinator object so NO INSTANCE VARIABLES
2329     // should be referenced after the call. In the case where
2330     // the TopCoordinator is a subordinate, the CoordinatorResource
2331     // object must be informed that the transaction has been
2332     // completed so that it can handle any subsequent requests for the
2333     // transaction.
2334
2335     if (terminator != null) {
2336         terminator.setCompleted(true, heuristicExc != null);
2337     }
2338
2339     /* commented out (Ram J) for memory leak fix.
2340     // If there are no registered Synchronization objects, there is
2341     // nothing left to do, so get the RecoveryManager to forget
2342     // about us, then self-destruct.
2343
2344     if (!root && (synchronizations == null ||
2345         !synchronizations.involved())
2346         ) {
2347         RecoveryManager.removeCoordinator(superInfo.globalTID,
2348             superInfo.localTID,
2349             true);
2350
2351         if (!dying) {
2352             destroy();
2353         }
2354     }
2355     */
2356     // added (Ram J) for memory leak fix
2357     // if subordinate, send out afterCompletion. This will
2358     // destroy the CoordinatorSynchronization and coordinator.
2359     if (!root) {
2360         afterCompletion(Status.StatusRolledBack);
2361     }
2362
2363     /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
2364     /* NO INSTANCE VARIABLES MAY BE ACCESSED FROM THIS POINT ON.      */
2365     /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
2366
2367     // If there was heuristic damage, report it.
2368
2369     if (heuristicExc != null) {
2370         if (heuristicExc instanceof HeuristicMixed) {
2371             throw (HeuristicMixed) heuristicExc;
2372         } else {
2373             throw (HeuristicHazard) heuristicExc;
2374         }
2375     }
2376 }
2377
2378 // Otherwise return normally.
2379 }

```

Naming conventions

line 2360: afterCompletion(...) method is named without using a verb

Indentation

line 2212: missing indentation before the if-clause

line 2212 - 2217: indentation made using tabs

line 2268: wrong indentation

line 2315: wrong indentation

line 2316: wrong indentation

Braces

2213-2217: Allman style used once in this method

File Organization

line 2293: useless blank line

line 2298: missing blank line between code and additional comment

line 2356: missing blank line before the additional comment

line 2359: missing blank line between code and comments

line 2379: blank line missing before the final bracket

Comments

line 2209: comment line is not clear (when the synchronization has to begin and finish?)

line 2331: comment lines verbose and unclear

line 2344-2354: comment lines verbose and unclear about the commented out code managing

Output Format

line 2269: wrong state is reported, it should be Transaction.STATE_ROLLING_BACK

Computation, Comparisons and Assignments

line 2297-2299: distributeRollback(...) method never throws Internal exceptions
line 2294-2303: brutish use of instanceof comparison; several catch branches are better
line 2360: Status class never imported
line 2360: if Status class is provided by javax.transaction, then it should be
Status.STATUS_ROLLEDBACK and not Status.StatusRolledBack

Exceptions

line 2207: INTERNAL exception not reported in the method signature

Method

register_synchronization(Synchronization sync)

```
2380
2381 /**
2382  * Informs the TopCoordinator that the given object requires
2383  * synchronization before and after completion of the transaction.
2384  * If possible, a CoordinatorSync object is registered
2385  * with the superior Coordinator. Otherwise this
2386  * Coordinator becomes the root of a sub-tree for
2387  * synchronization.
2388  *
2389  * @param sync The Synchronization object to be registered.
2390  *
2391  * @return
2392  *
2393  * @exception Inactive The Coordinator is in the process of completing the
2394  * transaction and cannot accept this registration.
2395  * @exception SynchronizationUnavailable The transaction service
2396  * cannot support synchronization.
2397  * @exception SystemException The operation failed.
2398  *
2399  * @see
2400  */
2401 synchronized public void register_synchronization(Synchronization sync)
2402     throws SystemException, Inactive, SynchronizationUnavailable {
2403
2404     // First check the state of the transaction. If it is not active,
2405     // do not allow the registration.
2406
2407     if (tranState == null ||
2408         tranState.state != TransactionState.STATE_ACTIVE) {
2409         Inactive exc = new Inactive();
2410         throw exc;
2411     }
2412
2413     // If not previously registered, a CoordinatorSync object must be
2414     // registered with our superior. Note that root TopCoordinators
2415     // are created with the registration flag set, so we do not need to
2416     // check whether we are the root TopCoordinator here.
2417
2418     if (!registeredSync && DefaultTransactionService.isORBAvailable()) {
2419
2420         // Initialise the CoordinatorSync with the local id, our reference,
2421         // and a flag to indicate that does not represent a subtransaction.
2422
2423         CoordinatorSynchronizationImpl sImpl =
2424             new CoordinatorSynchronizationImpl(this);
2425
2426         // Register the CoordinatorSync with the superior CoordinatorImpl.
2427
2428         try {
2429             Synchronization subSync = sImpl.object();
2430             superInfo.superior.register_synchronization(subSync);
2431             registeredSync = true;
2432
2433             // added (Ram J) for memory leak fix.
2434             this.coordSyncImpl = sImpl;
2435             if (_logger.isLoggable(Level.FINER))
2436             {
2437                 _logger.logp(Level.FINER, "TopCoordinator",
```

```

2438         "register_synchronization()",
2439         "CoordinatorSynchronizationImpl : " + sImpl +
2440         " has been registered with (Root)TopCoordinator"+
2441         "GTID is: " + superInfo.globalTID.toString());
2442     }
2443
2444     } catch (Exception exc) {
2445         // If an exception was raised, dont set the registration flag.
2446         sImpl.destroy();
2447
2448         // If the exception is a system exception, then allow it
2449         // to percolate to the caller.
2450
2451         if (exc instanceof OBJECT_NOT_EXIST) {
2452             TRANSACTION_ROLLEDBACK ex2 =
2453                 new TRANSACTION_ROLLEDBACK(
2454                     0, CompletionStatus.COMPLETED_NO);
2455             ex2.initCause(exc);
2456             throw ex2;
2457         }
2458
2459         if (exc instanceof Inactive) {
2460             throw (Inactive)exc;
2461         }
2462
2463         if (exc instanceof SystemException) {
2464             throw (SystemException) exc;
2465         }
2466
2467         // Otherwise throw an internal exception.
2468
2469         INTERNAL ex2 = new INTERNAL(MinorCode.NotRegistered,
2470                                     CompletionStatus.COMPLETED_NO);
2471         ex2.initCause(exc);
2472         throw ex2;
2473     }
2474 }
2475
2476 // Make sure the RegisteredSyncs instance variable has been set up.
2477
2478 if (synchronizations == null) {
2479     synchronizations = new RegisteredSyncs();
2480 }
2481
2482 // Add a duplicate of the reference to the set. This is done
2483 // because if the registration is for a remote object,
2484 // the proxy will be freed
2485 // when the registration request returns.
2486
2487 // COMMENT(Ram J) if the sync object is a local servant, there is
2488 // no proxy involved. Also the instanceof operator could be replaced
2489 // by a is_local() method if this class implements the CORBA local
2490 // object contract.
2491 if (sync instanceof com.sun.jts.jta.SynchronizationImpl) {
2492     synchronizations.addSync(sync);
2493
2494     if (_logger.isLoggable(Level.FINER))
2495     {
2496         _logger.logp(Level.FINER, "TopCoordinator",
2497             "register_synchronization()",
2498             "SynchronizationImpl : " + sync +
2499             " has been registered with TopCoordinator : "+
2500             "GTID is : " + superInfo.globalTID.toString());
2501     }
2502 } else {
2503     synchronizations.addSync((Synchronization) sync._duplicate());
2504 }
2505
2506 temporary = false;
2507 }
2508
2509

```

Naming conventions

line 2401: Inactive and SynchronizationUnavailable should be written with the word Exception at the end of the named
line 2429: object() method isn't named using a verb
line 2451-2456: TRANSACTION_ROLLEDBACK and OBJECT_NOT_EXIST should be written following exception naming rules
line 2455: initCause() method not named using a verb
line 2469: NotRegistered should be capitalized because it's a constant
line 2469: INTERNAL should be written following exception naming rules
line 2478: synchronizations doesn't clarify what objects the attribute is referencing to
line 2491: SynchronizationImpl is declared using the import path
line 2504: _duplicate() shouldn't begin with '_' symbol

Indentation

line 2409: wrong indentation
line 2410: wrong indentation
line 2435: indentation using tabs
line 2437-2441: indentation using tabs
line 2496-2500: indentation using tabs

Braces

line 2436: Allman style used once instead of K&R
line 2436: Allman style used once instead of K&R

File Organization

line 2433: blank line needed after the comment
line 2445: blank line needed after and before the comment
line 2490: blank line needed after the comment
line 2493: useless blank line
line 2502: useless blank line

Comments

line 2418: comment doesn't clarify why we need to check the ORB
line 2420-2421: comment about an initialization with a flag, but CoordinatorSynchronizationImpl(this) doesn't treat it

line 2445: comment doesn't seem relative to the code line below
line 2482-2490: comments doesn't corresponds to the branch below but to the else one
line 2504-2507: lack of comments explaining what the method is doing in those cases
(setting temporary to false)

Output Format

line 2499: ' ' needed between the words "registered" and "with"

Computation, Comparisons and Assignments

line 2444-2473: brutish programming, catch branches should replace the use of th
instanceof comparisons
line 2472: INTERNAL exception thrown with its superclass visibility (implicit casting)

Exceptions

line 2402: it's not clear when SynchronizationUnavailable exception is thrown

Issues Statistics

We report in the following tables some statistics on the issues we have found considering the checklist on the assigned methods.

	Source code lines	#issues found	#issues per line
Total	486	79	0.16
Commit	166	28	0.16
Rollback	192	21	0.10
Register_synchronization	128	30	0.23

Issues per class	#	%
Indentation	16	20.2
Naming Conventions	15	19
File Organization	15	19
Comments	13	16.4
Computation, Comparisons and Assignments	9	11.3
Braces	4	5
Exceptions	4	5
Output Format	2	2.5
Initialization & Declarations	2	2.5

Number of hours

We have spent 28 hours per person to redact the CodeInspection document.

Greta Ghiotti: 28 hours

Raffaele Malvermi: 28 hours

Mirco Mutti: 28 hours