POLITECNICO DI MILANO
Computer Science and Engineering
Software Engineering 2 Project

Code Inspection

Authors: Greta Ghiotti
Raffaele Malvermi
Mirco Mutti

Version 1.1

Date 6/01/16

Reference Professor: Mirandola Raffaela

Index

Version 1.1:
Fixing of the image caption at page 4.

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

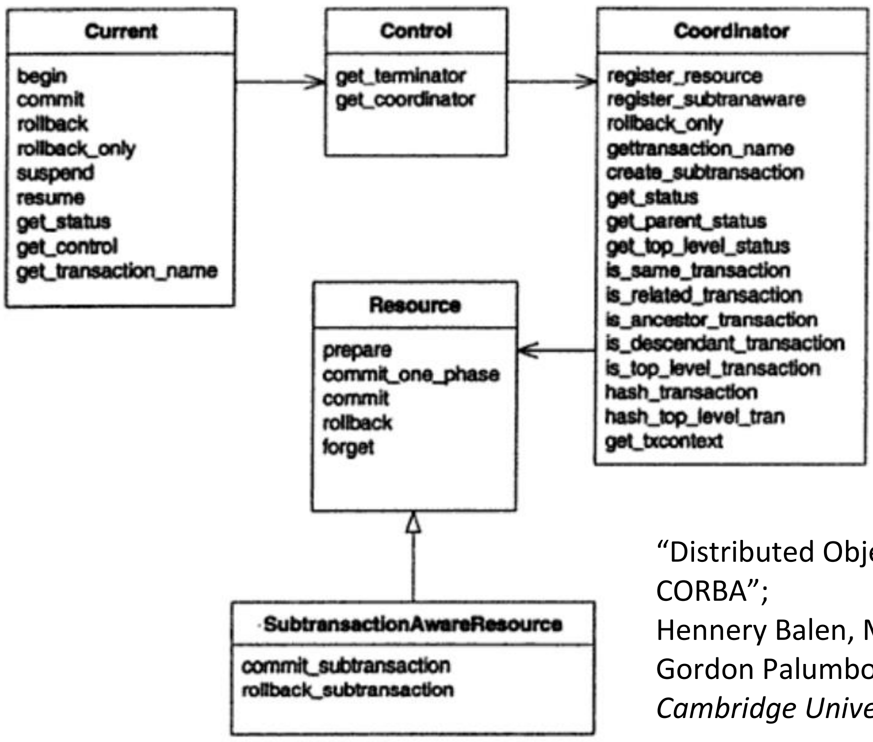
Functional role of assigned set of classes

The System we have to inspect adoptees 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.



"Distributed Object Architectures with CORBA";
Hennery Balen, Mark Elenko, Jan Jones, Gordon Palumbo;
Cambridge University Press.

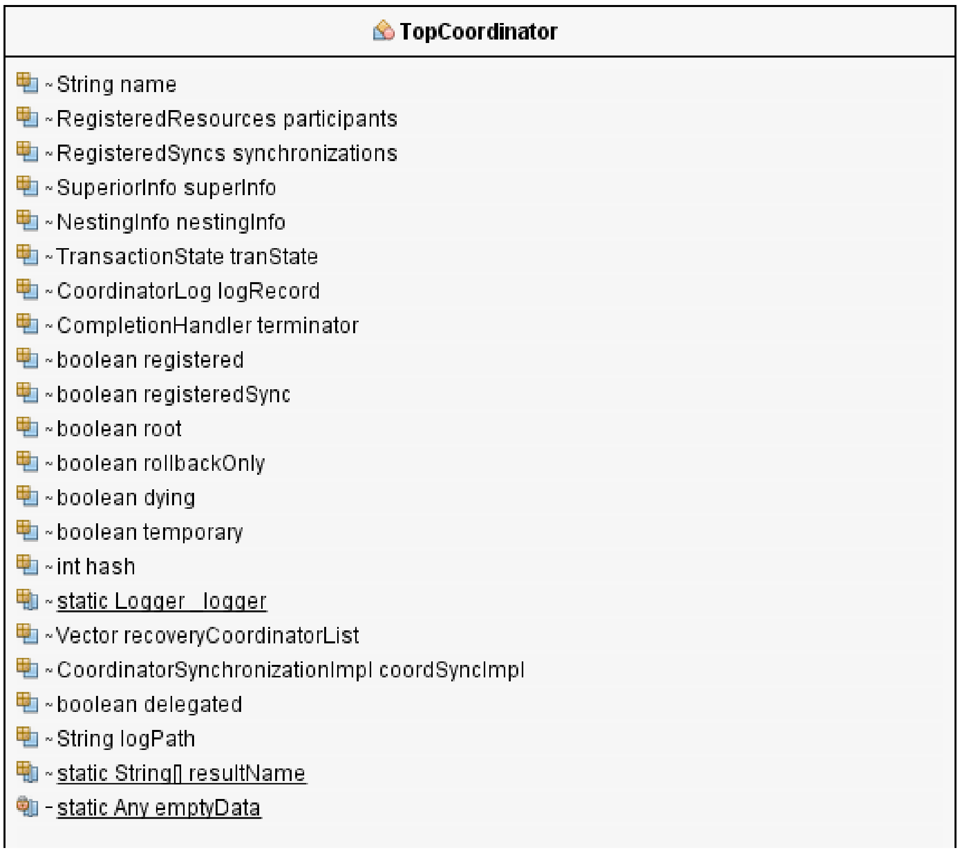
To manage the tree structure each TopCoordinator owns the references of the SuperiorInfo and NestingInfo objects: the first one includes the information of the parent Coordinator, the second includes the reference of each sub-node in the tree and also the entire sequence of ancestors. But, in certain 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:

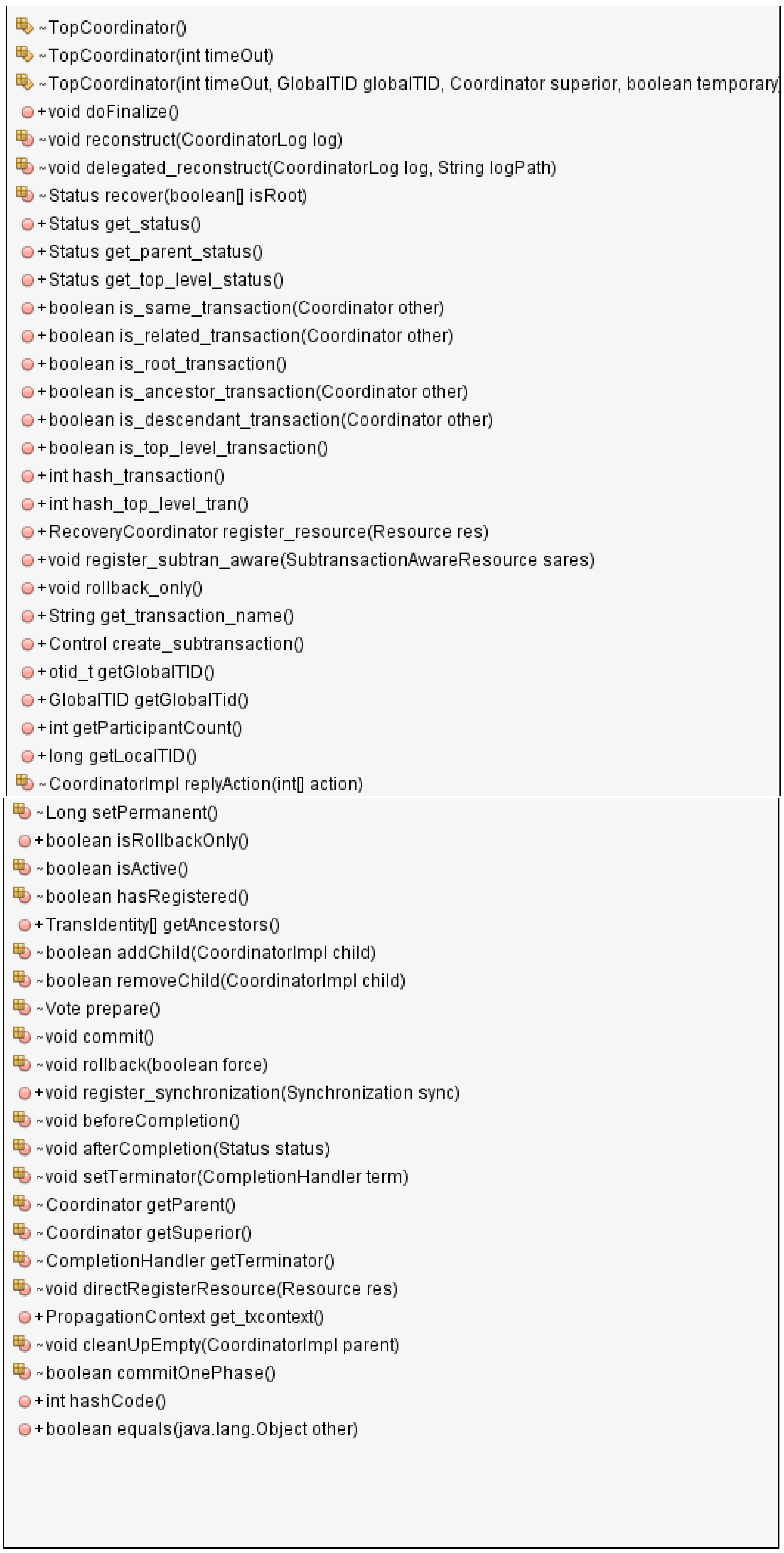
- Two Phase Commit Protocol operations (commit, rollback, …)
- Multi-thread synchronization managing operations (register_synchronization as instance)
- Failure recovery operations

The Javadoc reports:

"In addition the TopCoordinator recovery interface can be used if the connection to a 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

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



```
~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.

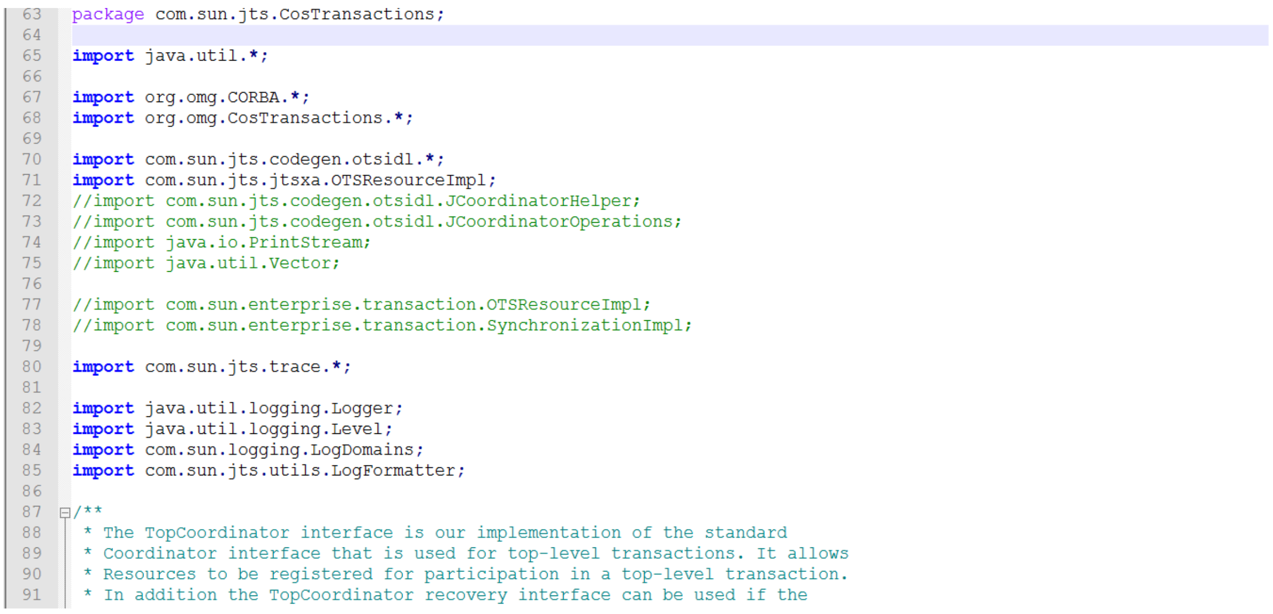
We also have found some documentation about the concepts underlying this implementation:

- 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

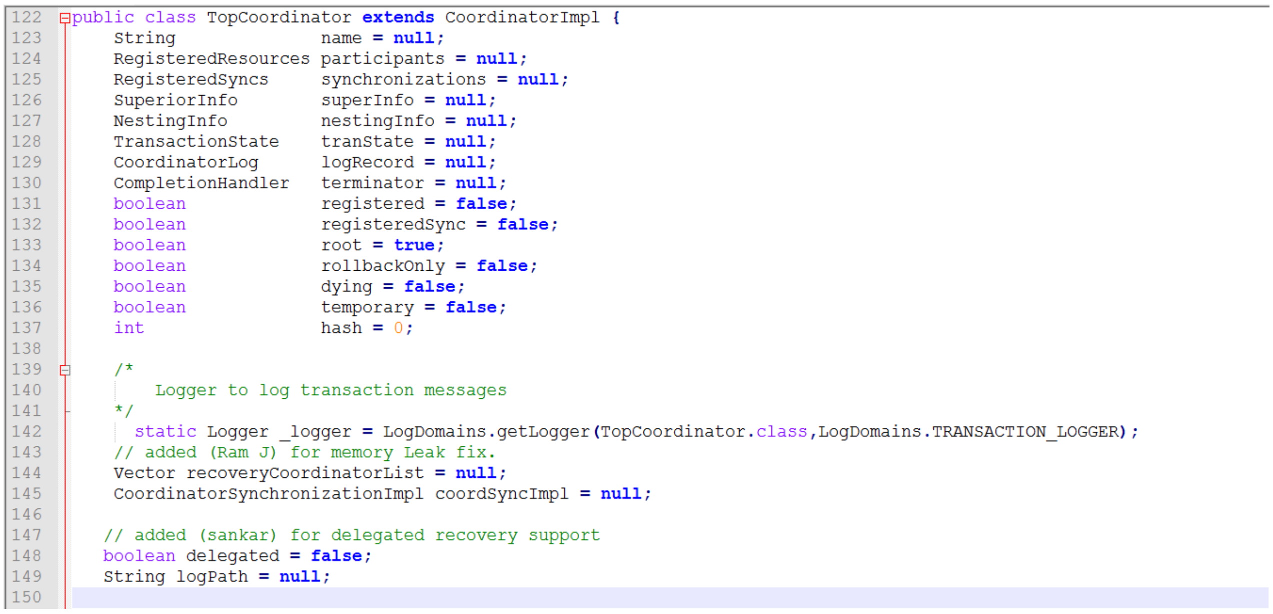
Class inspection

<u>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.</u>



```
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



```
public class TopCoordinator extends CoordinatorImpl {
    String              name = null;
    RegisteredResources participants = null;
    RegisteredSyncs     synchronizations = null;
    SuperiorInfo        superInfo = null;
    NestingInfo         nestingInfo = null;
    TransactionState    tranState = null;
    CoordinatorLog      logRecord = null;
    CompletionHandler   terminator = null;
    boolean             registered = false;
    boolean             registeredSync = false;
    boolean             root = true;
    boolean             rollbackOnly = false;
    boolean             dying = false;
    boolean             temporary = false;
    int                 hash = 0;

    /*
        Logger to log transaction messages
    */
      static Logger _logger = LogDomains.getLogger(TopCoordinator.class,LogDomains.TRANSACTION_LOGGER);
    // added (Ram J) for memory Leak fix.
    Vector recoveryCoordinatorList = null;
    CoordinatorSynchronizationImpl coordSyncImpl = null;

   // added (sankar) for delegated recovery support
   boolean delegated = false;
   String logPath = null;

```

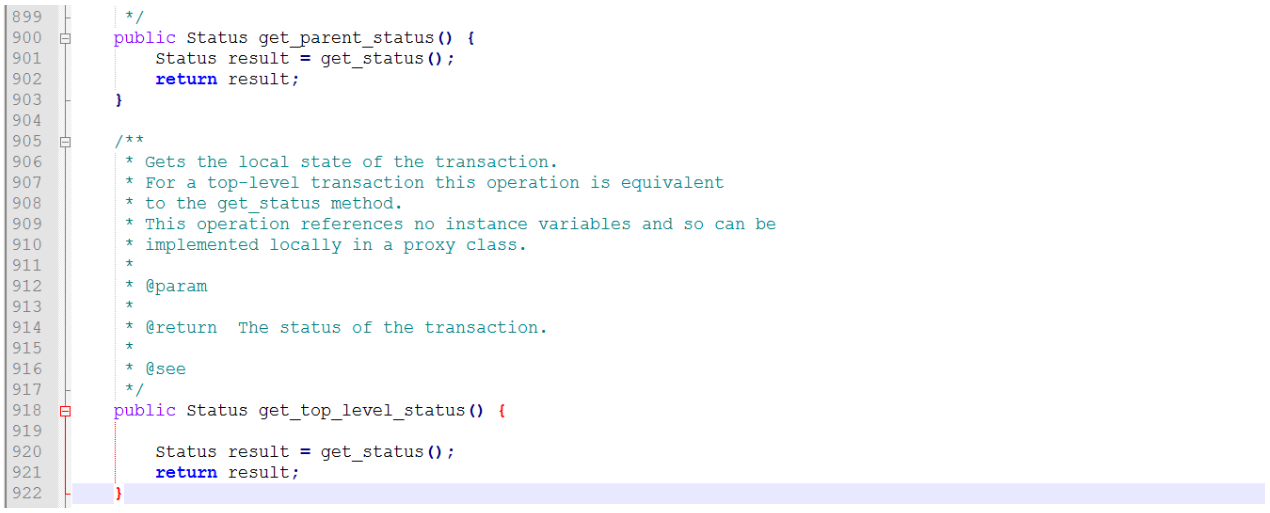
Javadoc updated only until version 0.02

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      }
```

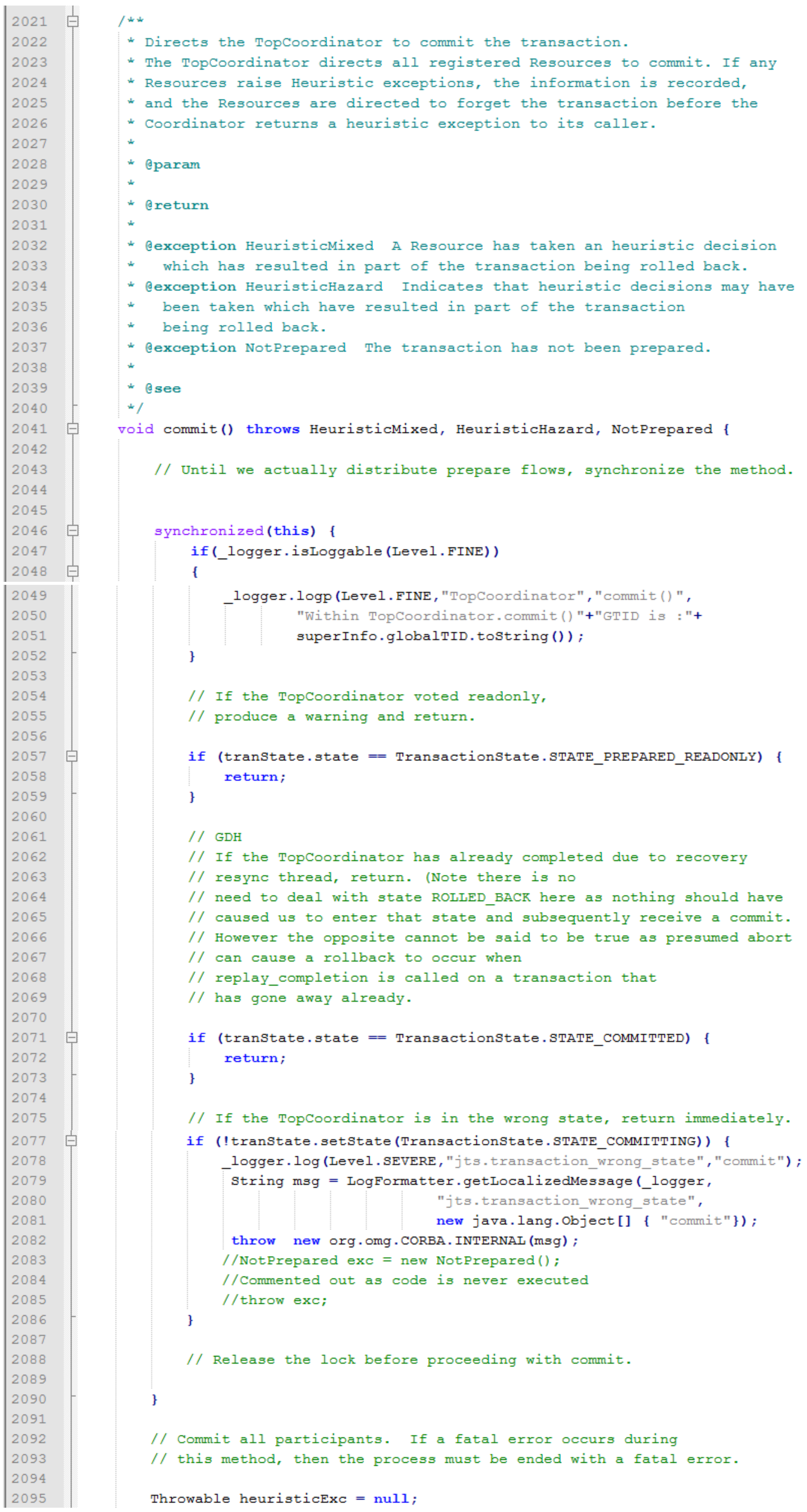
Class Declaration

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

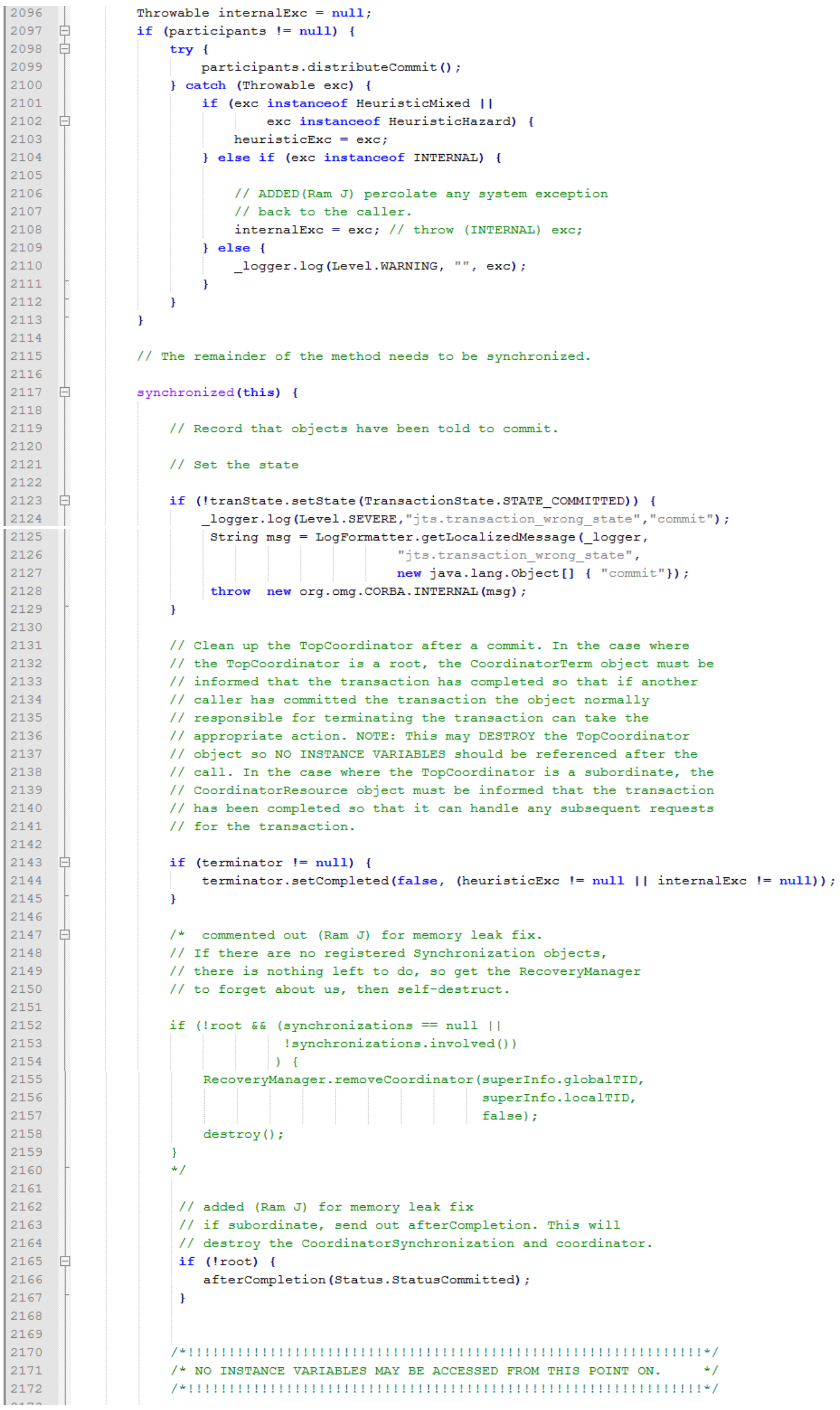
lines 900/918: both get_parent_status() and get_top_level_status() methods return the same result, and this is visible in the javadoc, too

addChild, commit, rollback are big methods

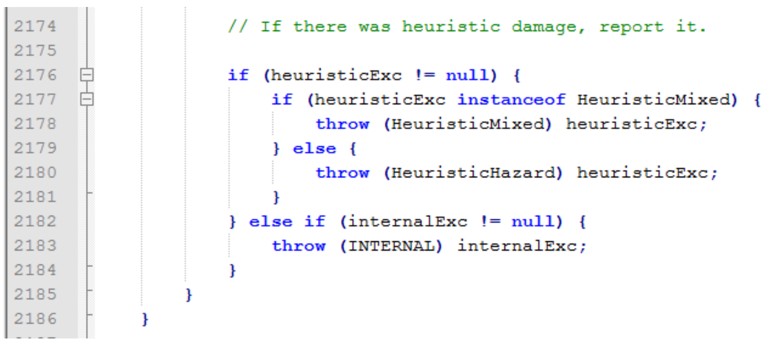
Method Commit()



```java
/**
 * Directs the TopCoordinator to commit the transaction.
 * The TopCoordinator directs all registered Resources to commit. If any
 * Resources raise Heuristic exceptions, the information is recorded,
 * and the Resources are directed to forget the transaction before the
 * Coordinator returns a heuristic exception to its caller.
 *
 * @param
 *
 * @return
 *
 * @exception HeuristicMixed  A Resource has taken an heuristic decision
 *   which has resulted in part of the transaction being rolled back.
 * @exception HeuristicHazard  Indicates that heuristic decisions may have
 *   been taken which have resulted in part of the transaction
 *   being rolled back.
 * @exception NotPrepared  The transaction has not been prepared.
 *
 * @see
 */
void commit() throws HeuristicMixed, HeuristicHazard, NotPrepared {

    // Until we actually distribute prepare flows, synchronize the method.


    synchronized(this) {
        if(_logger.isLoggable(Level.FINE))
        {
            _logger.logp(Level.FINE,"TopCoordinator","commit()",
                    "Within TopCoordinator.commit()"+"GTID is :"+
                    superInfo.globalTID.toString());
        }

        // If the TopCoordinator voted readonly,
        // produce a warning and return.

        if (tranState.state == TransactionState.STATE_PREPARED_READONLY) {
            return;
        }

        // GDH
        // If the TopCoordinator has already completed due to recovery
        // resync thread, return. (Note there is no
        // need to deal with state ROLLED_BACK here as nothing should have
        // caused us to enter that state and subsequently receive a commit.
        // However the opposite cannot be said to be true as presumed abort
        // can cause a rollback to occur when
        // replay_completion is called on a transaction that
        // has gone away already.

        if (tranState.state == TransactionState.STATE_COMMITTED) {
            return;
        }

        // If the TopCoordinator is in the wrong state, return immediately.
        if (!tranState.setState(TransactionState.STATE_COMMITTING)) {
            _logger.log(Level.SEVERE,"jts.transaction_wrong_state","commit");
             String msg = LogFormatter.getLocalizedMessage(_logger,
                                       "jts.transaction_wrong_state",
                                       new java.lang.Object[] { "commit"});
             throw  new org.omg.CORBA.INTERNAL(msg);
            //NotPrepared exc = new NotPrepared();
            //Commented out as code is never executed
            //throw exc;
        }

        // Release the lock before proceeding with commit.

    }

    // Commit all participants.  If a fatal error occurs during
    // this method, then the process must be ended with a fatal error.

    Throwable heuristicExc = null;
```



```java
Throwable internalExc = null;
if (participants != null) {
    try {
        participants.distributeCommit();
    } catch (Throwable exc) {
        if (exc instanceof HeuristicMixed ||
                exc instanceof HeuristicHazard) {
            heuristicExc = exc;
        } else if (exc instanceof INTERNAL) {

            // ADDED(Ram J) percolate any system exception
            // back to the caller.
            internalExc = exc; // throw (INTERNAL) exc;
        } else {
            _logger.log(Level.WARNING, "", exc);
        }
    }
}

// The remainder of the method needs to be synchronized.

synchronized(this) {

    // Record that objects have been told to commit.

    // Set the state

    if (!tranState.setState(TransactionState.STATE_COMMITTED)) {
        _logger.log(Level.SEVERE,"jts.transaction_wrong_state","commit");
         String msg = LogFormatter.getLocalizedMessage(_logger,
                                "jts.transaction_wrong_state",
                                new java.lang.Object[] { "commit"});
         throw  new org.omg.CORBA.INTERNAL(msg);
    }

    // Clean up the TopCoordinator after a commit. In the case where
    // the TopCoordinator is a root, the CoordinatorTerm object must be
    // informed that the transaction has completed so that if another
    // caller has committed the transaction the object normally
    // responsible for terminating the transaction can take the
    // appropriate action. NOTE: This may DESTROY the TopCoordinator
    // object so NO INSTANCE VARIABLES should be referenced after the
    // call. In the case where the TopCoordinator is a subordinate, the
    // CoordinatorResource object must be informed that the transaction
    // has been completed so that it can handle any subsequent requests
    // for the transaction.

    if (terminator != null) {
        terminator.setCompleted(false, (heuristicExc != null || internalExc != null));
    }

    /*  commented out (Ram J) for memory leak fix.
    // If there are no registered Synchronization objects,
    // there is nothing left to do, so get the RecoveryManager
    // to forget about us, then self-destruct.

    if (!root && (synchronizations == null ||
                  !synchronizations.involved())
                 ) {
        RecoveryManager.removeCoordinator(superInfo.globalTID,
                                          superInfo.localTID,
                                          false);
        destroy();
    }
    */

     // added (Ram J) for memory leak fix
     // if subordinate, send out afterCompletion. This will
     // destroy the CoordinatorSynchronization and coordinator.
     if (!root) {
        afterCompletion(Status.StatusCommitted);
     }


    /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
    /* NO INSTANCE VARIABLES MAY BE ACCESSED FROM THIS POINT ON.      */
    /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
```



```
            // If there was heuristic damage, report it.

            if (heuristicExc != null) {
                if (heuristicExc instanceof HeuristicMixed) {
                    throw (HeuristicMixed) heuristicExc;
                } else {
                    throw (HeuristicHazard) heuristicExc;
                }
            } else if (internalExc != null) {
                throw (INTERNAL) internalExc;
            }
        }
    }
```

Naming conventions

line 2041: HeuristicMixed, HeuristicHazard and NotPrepared should finish with "Exception" (HeuristciMixedException, 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

Indention

line 2046-2052: indention made using tabs
line 2079-2084: indention 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: indention 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

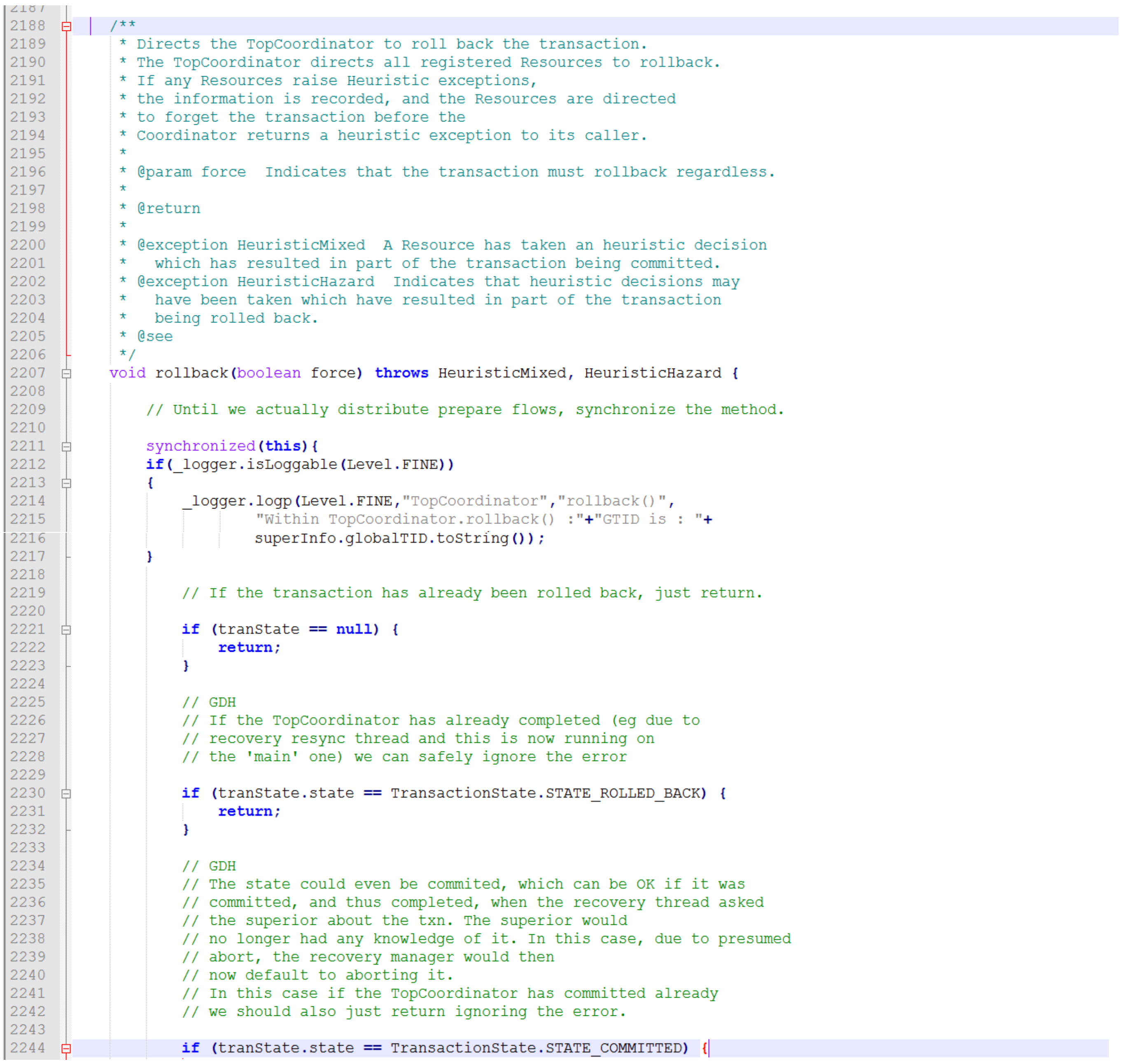
Computation, Comparisons and Assignments

line 2100-2110: brutish use of instanceof comparison; several catch branches are better
line 2166: Status.StatusCommited is not an attribute of the class Status
(STATUS_COMMITED 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

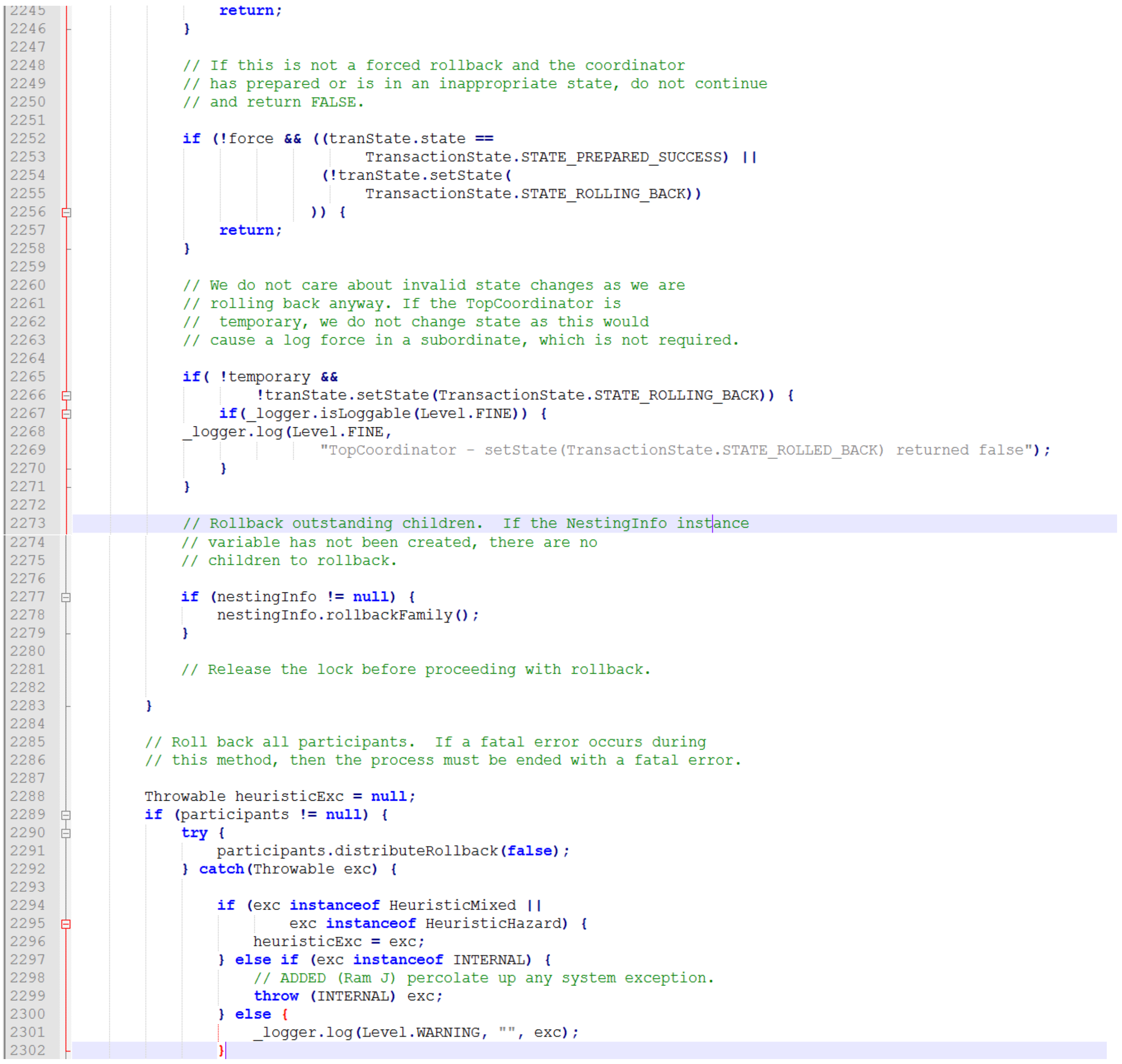
Exceptions

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

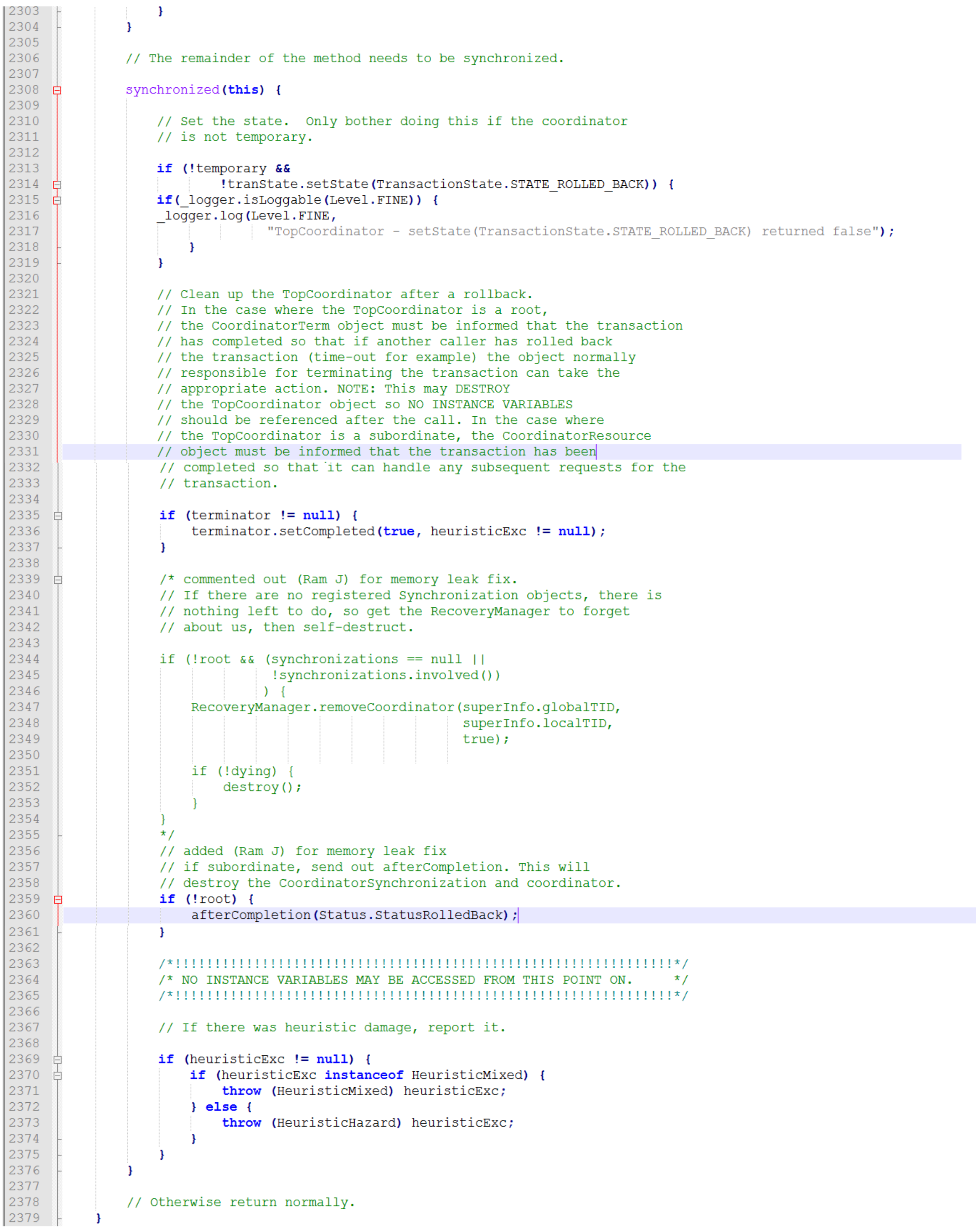
Method rollback(boolean force)



```java
/**
 * Directs the TopCoordinator to roll back the transaction.
 * The TopCoordinator directs all registered Resources to rollback.
 * If any Resources raise Heuristic exceptions,
 * the information is recorded, and the Resources are directed
 * to forget the transaction before the
 * Coordinator returns a heuristic exception to its caller.
 *
 * @param force  Indicates that the transaction must rollback regardless.
 *
 * @return
 *
 * @exception HeuristicMixed  A Resource has taken an heuristic decision
 *   which has resulted in part of the transaction being committed.
 * @exception HeuristicHazard  Indicates that heuristic decisions may
 *   have been taken which have resulted in part of the transaction
 *   being rolled back.
 * @see
 */
void rollback(boolean force) throws HeuristicMixed, HeuristicHazard {

    // Until we actually distribute prepare flows, synchronize the method.

    synchronized(this){
    if(_logger.isLoggable(Level.FINE))
    {
        _logger.logp(Level.FINE,"TopCoordinator","rollback()",
                "Within TopCoordinator.rollback() :"+"GTID is : "+
                superInfo.globalTID.toString());
    }

        // If the transaction has already been rolled back, just return.

        if (tranState == null) {
            return;
        }

        // GDH
        // If the TopCoordinator has already completed (eg due to
        // recovery resync thread and this is now running on
        // the 'main' one) we can safely ignore the error

        if (tranState.state == TransactionState.STATE_ROLLED_BACK) {
            return;
        }

        // GDH
        // The state could even be commited, which can be OK if it was
        // committed, and thus completed, when the recovery thread asked
        // the superior about the txn. The superior would
        // no longer had any knowledge of it. In this case, due to presumed
        // abort, the recovery manager would then
        // now default to aborting it.
        // In this case if the TopCoordinator has committed already
        // we should also just return ignoring the error.

        if (tranState.state == TransactionState.STATE_COMMITTED) {
```



```
        return;
    }

    // If this is not a forced rollback and the coordinator
    // has prepared or is in an inappropriate state, do not continue
    // and return FALSE.

    if (!force && ((tranState.state ==
                        TransactionState.STATE_PREPARED_SUCCESS) ||
                   (!tranState.setState(
                        TransactionState.STATE_ROLLING_BACK))
                  )) {
        return;
    }

    // We do not care about invalid state changes as we are
    // rolling back anyway. If the TopCoordinator is
    //  temporary, we do not change state as this would
    // cause a log force in a subordinate, which is not required.

    if( !temporary &&
            !tranState.setState(TransactionState.STATE_ROLLING_BACK)) {
        if(_logger.isLoggable(Level.FINE)) {
    _logger.log(Level.FINE,
                   "TopCoordinator - setState(TransactionState.STATE_ROLLED_BACK) returned false");
        }
    }

    // Rollback outstanding children.  If the NestingInfo instance
    // variable has not been created, there are no
    // children to rollback.

    if (nestingInfo != null) {
        nestingInfo.rollbackFamily();
    }

    // Release the lock before proceeding with rollback.

}

// Roll back all participants.  If a fatal error occurs during
// this method, then the process must be ended with a fatal error.

Throwable heuristicExc = null;
if (participants != null) {
    try {
        participants.distributeRollback(false);
    } catch(Throwable exc) {

        if (exc instanceof HeuristicMixed ||
                exc instanceof HeuristicHazard) {
            heuristicExc = exc;
        } else if (exc instanceof INTERNAL) {
            // ADDED (Ram J) percolate up any system exception.
            throw (INTERNAL) exc;
        } else {
            _logger.log(Level.WARNING, "", exc);
        }
```



```
            }
        }

        // The remainder of the method needs to be synchronized.

        synchronized(this) {

            // Set the state.  Only bother doing this if the coordinator
            // is not temporary.

            if (!temporary &&
                    !tranState.setState(TransactionState.STATE_ROLLED_BACK)) {
            if(_logger.isLoggable(Level.FINE)) {
            _logger.log(Level.FINE,
                          "TopCoordinator - setState(TransactionState.STATE_ROLLED_BACK) returned false");
                }
            }

            // Clean up the TopCoordinator after a rollback.
            // In the case where the TopCoordinator is a root,
            // the CoordinatorTerm object must be informed that the transaction
            // has completed so that if another caller has rolled back
            // the transaction (time-out for example) the object normally
            // responsible for terminating the transaction can take the
            // appropriate action. NOTE: This may DESTROY
            // the TopCoordinator object so NO INSTANCE VARIABLES
            // should be referenced after the call. In the case where
            // the TopCoordinator is a subordinate, the CoordinatorResource
            // object must be informed that the transaction has been
            // completed so that it can handle any subsequent requests for the
            // transaction.

            if (terminator != null) {
                terminator.setCompleted(true, heuristicExc != null);
            }

            /* commented out (Ram J) for memory leak fix.
            // If there are no registered Synchronization objects, there is
            // nothing left to do, so get the RecoveryManager to forget
            // about us, then self-destruct.

            if (!root && (synchronizations == null ||
                          !synchronizations.involved())
                         ) {
                RecoveryManager.removeCoordinator(superInfo.globalTID,
                                                  superInfo.localTID,
                                                  true);

                if (!dying) {
                    destroy();
                }
            }
            */
            // added (Ram J) for memory leak fix
            // if subordinate, send out afterCompletion. This will
            // destroy the CoordinatorSynchronization and coordinator.
            if (!root) {
                afterCompletion(Status.StatusRolledBack);
            }

            /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
            /* NO INSTANCE VARIABLES MAY BE ACCESSED FROM THIS POINT ON.     */
            /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/

            // If there was heuristic damage, report it.

            if (heuristicExc != null) {
                if (heuristicExc instanceof HeuristicMixed) {
                    throw (HeuristicMixed) heuristicExc;
                } else {
                    throw (HeuristicHazard) heuristicExc;
                }
            }
        }

        // Otherwise return normally.
    }
```

Naming conventions

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

Indention

line 2212: missing indention before the if-clause
line 2212 - 2217: indention made using tabs
line 2268: wrong indention
line 2315: wrong indention
line 2316: wrong indention

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

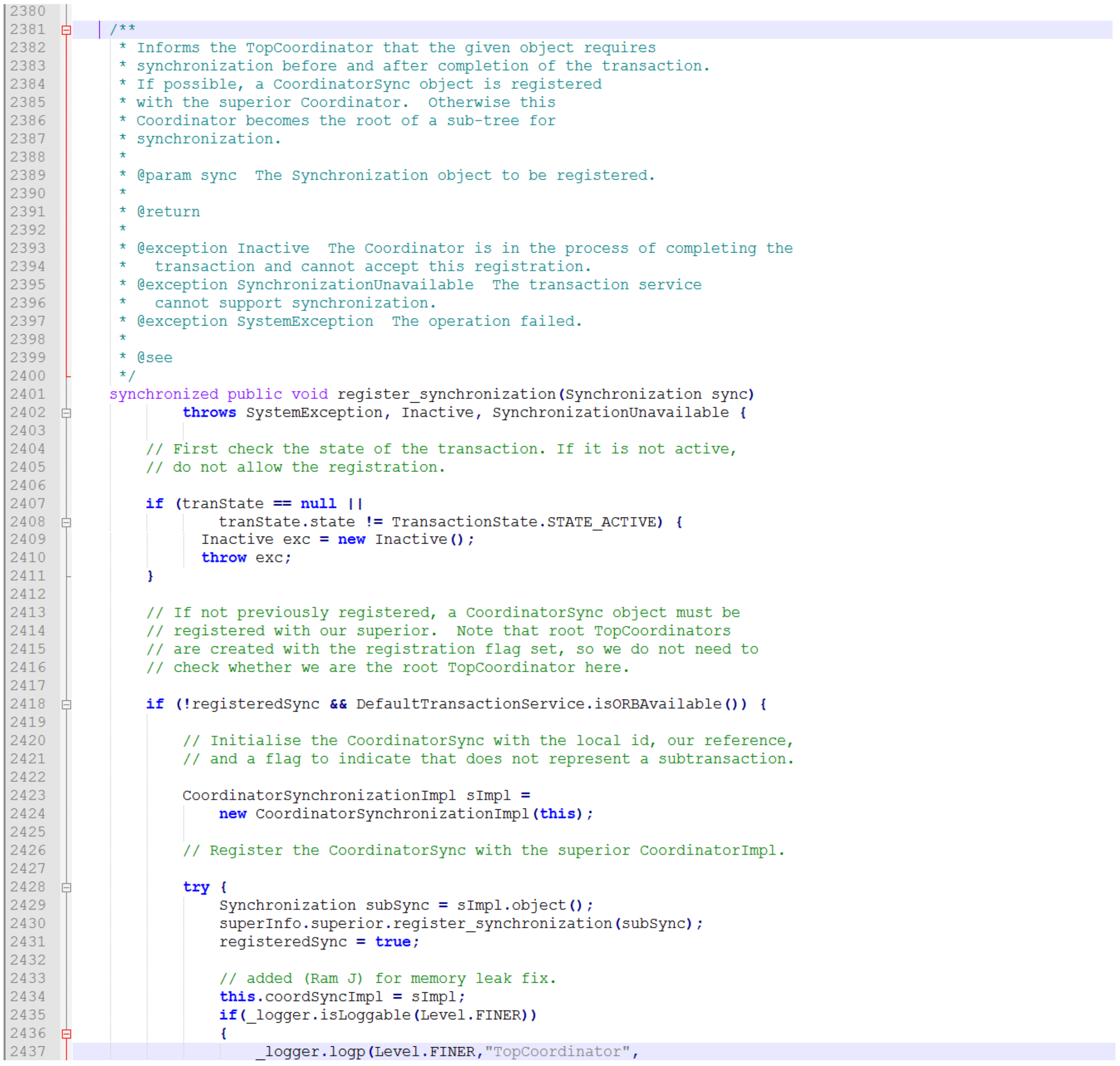
Computation, Comparisons and Assignments

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

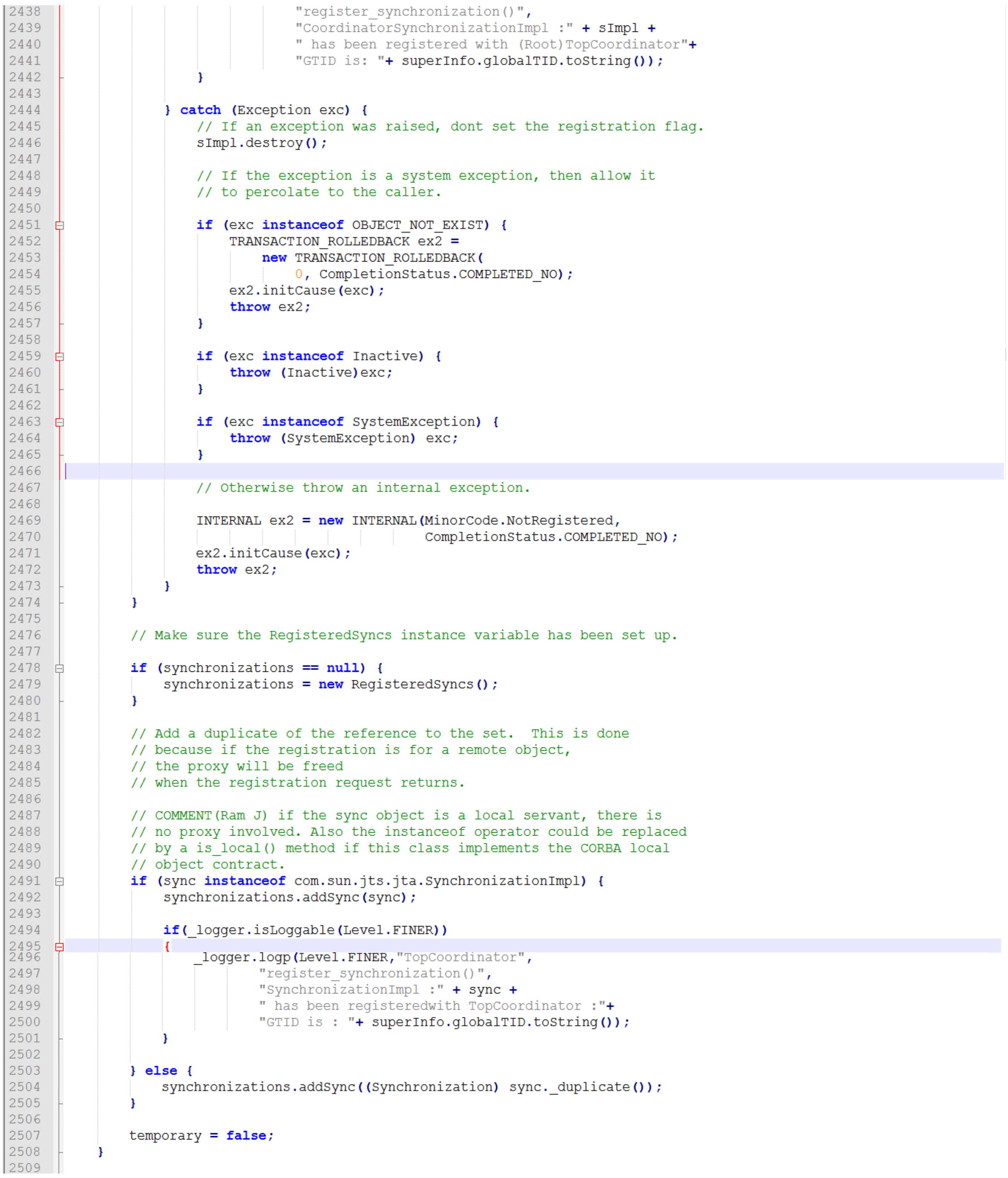
Exceptions

line 2207: INTERNAL exception not reported in the method signature

Method register_synchronization(Synchronization sync)



```java
/**
 * Informs the TopCoordinator that the given object requires
 * synchronization before and after completion of the transaction.
 * If possible, a CoordinatorSync object is registered
 * with the superior Coordinator.  Otherwise this
 * Coordinator becomes the root of a sub-tree for
 * synchronization.
 *
 * @param sync  The Synchronization object to be registered.
 *
 * @return
 *
 * @exception Inactive  The Coordinator is in the process of completing the
 *   transaction and cannot accept this registration.
 * @exception SynchronizationUnavailable  The transaction service
 *   cannot support synchronization.
 * @exception SystemException  The operation failed.
 *
 * @see
 */
synchronized public void register_synchronization(Synchronization sync)
        throws SystemException, Inactive, SynchronizationUnavailable {

    // First check the state of the transaction. If it is not active,
    // do not allow the registration.

    if (tranState == null ||
            tranState.state != TransactionState.STATE_ACTIVE) {
          Inactive exc = new Inactive();
          throw exc;
    }

    // If not previously registered, a CoordinatorSync object must be
    // registered with our superior.  Note that root TopCoordinators
    // are created with the registration flag set, so we do not need to
    // check whether we are the root TopCoordinator here.

    if (!registeredSync && DefaultTransactionService.isORBAvailable()) {

        // Initialise the CoordinatorSync with the local id, our reference,
        // and a flag to indicate that does not represent a subtransaction.

        CoordinatorSynchronizationImpl sImpl =
            new CoordinatorSynchronizationImpl(this);

        // Register the CoordinatorSync with the superior CoordinatorImpl.

        try {
            Synchronization subSync = sImpl.object();
            superInfo.superior.register_synchronization(subSync);
            registeredSync = true;

            // added (Ram J) for memory leak fix.
            this.coordSyncImpl = sImpl;
            if(_logger.isLoggable(Level.FINER))
            {
                _logger.logp(Level.FINER,"TopCoordinator",
```



```java
                        "register_synchronization()",
                        "CoordinatorSynchronizationImpl :" + sImpl +
                        " has been registered with (Root)TopCoordinator"+
                        "GTID is: "+ superInfo.globalTID.toString());
            }

        } catch (Exception exc) {
            // If an exception was raised, dont set the registration flag.
            sImpl.destroy();

            // If the exception is a system exception, then allow it
            // to percolate to the caller.

            if (exc instanceof OBJECT_NOT_EXIST) {
                TRANSACTION_ROLLEDBACK ex2 =
                    new TRANSACTION_ROLLEDBACK(
                        0, CompletionStatus.COMPLETED_NO);
                ex2.initCause(exc);
                throw ex2;
            }

            if (exc instanceof Inactive) {
                throw (Inactive)exc;
            }

            if (exc instanceof SystemException) {
                throw (SystemException) exc;
            }

            // Otherwise throw an internal exception.

            INTERNAL ex2 = new INTERNAL(MinorCode.NotRegistered,
                                            CompletionStatus.COMPLETED_NO);
            ex2.initCause(exc);
            throw ex2;
        }
    }

    // Make sure the RegisteredSyncs instance variable has been set up.

    if (synchronizations == null) {
        synchronizations = new RegisteredSyncs();
    }

    // Add a duplicate of the reference to the set.  This is done
    // because if the registration is for a remote object,
    // the proxy will be freed
    // when the registration request returns.

    // COMMENT(Ram J) if the sync object is a local servant, there is
    // no proxy involved. Also the instanceof operator could be replaced
    // by a is_local() method if this class implements the CORBA local
    // object contract.
    if (sync instanceof com.sun.jts.jta.SynchronizationImpl) {
        synchronizations.addSync(sync);

        if(_logger.isLoggable(Level.FINER))
        {
            _logger.logp(Level.FINER,"TopCoordinator",
                    "register_synchronization()",
                    "SynchronizationImpl :" + sync +
                    " has been registeredwith TopCoordinator :"+
                    "GTID is : "+ superInfo.globalTID.toString());
        }

    } else {
        synchronizations.addSync((Synchronization) sync._duplicate());
    }

    temporary = false;
}
```

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

Indention

line 2409: wrong indention
line 2410: wrong indention
line 2435: indention using tabs
line 2437-2441: indention using tabs
line 2496-2500: indention 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	#	%
Indention	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