Politecnico di Milano
A.A. 2015-2016

Software Engineering 2

Code Inspection

Matteo Bulloni (852676), Marco Cannici (852527)

5 January 2016

Contents

1 Assigned class **3**

2 Functional Role **3**
2.1 JTS Transaction Service 3
2.2 CurrentTransaction class 3
2.3 endAborted method 3
2.4 sendingReply method 4

3 Code Inspection **5**
3.1 Class Analysis 5
3.2 Method analysis: "endAborted" 9
3.3 Method analysis: "sendingReply" 12

4 Appendix **18**
4.1 Reference documents 18
4.2 Hours of work 18

1 Assigned class

The class assigned to us is called "**CurrentTransaction**" (namespace: `com.sun.jts.CosTransactions.CurrentTransaction`) which is located in the following path relative to the root of GlassFish project: `appserver/transaction/jts/src/main/java/com/sun/jts/CosTransactions/CurrentTransaction.java`

The methods of the "**CurrentTransaction**" class assigned to us are the following:

- Name: ***endAborted****(boolean [ ] aborted , boolean endAssociation)*
  Start Line: 374
- Name: ***sendingReply****(int id , PropagationContextHolder holder)*
  Start Line: 1035

2 Functional Role

2.1 JTS Transaction Service

2.2 CurrentTransaction class

The "CurrentTransaction" class is a static class that does not implement any interface and is used to keep track of the associations between transactions and threads.

The following is the JavaDoc of the class:

```
/**This class manages association of transactions with threads in a process,
 * and associated state/operations.
 *
 * @version 0.01
 *
 * @author Simon Holdsworth, IBM Corporation
 *
 * @see
*/
```

For each thread the class keeps track of the transactions with which it is associated to, the list of suspended transactions (which are transactions that have been suspended because a new request has been received while they were running) and the list of RegisteredStatics objects that will be informed of any changes in the associations of the thread with the transactions. The class exposes methods to modify the current association of the thread and the list of suspended transactions and to retrieve the list of transactions associated to the current thread. It also exposes methods to notify the Control object that a reply or a request has been (or is about to be) either received or sended.

2.3 endAborted method

This is a private method of the class "CurrentTransaction", it is called by *release()*, *sendingReply()* and *sendingRequest()* methods to ensure that the Control object associated with the current thread does not represent a transaction that has already been aborted, eventually terminating the current association and replacing it with an active one.

The following are the JavaDoc and the declaration of the method:

```
    /**Ensures that an association with an aborted transaction is dealt with
    ↪ cleanly.
     *
     *
     * TN - do not dissociate thread even if it's aborted!!
     *
     * If the current Control object represents a transaction that has been
     * aborted, this method replaces the association by one with the first
     * ancestor that has not been aborted, if any, or no association, and the
     * method returns true as the output parameter. Otherwise the method
     ↪ returns
     * false as the output parameter.
     * <p>
     * If there is a current Control object in either case it is returned,
     * otherwise null is returned.
     *
     * @param aborted  A 1-element array which will hold the aborted
     ↪ indicator.
     *
     * @return  The current Control object.
     *
     * @see
     */
    private static ControlImpl
        endAborted( boolean[/*1*/] aborted, boolean endAssociation) {
```

The method checks if the transaction associated with the current thread has already been aborted (communicating it to the caller through the output parameter "**aborted**") by checking his status. In that case, and if the method has been called with "**endAssociation**" argument set to true the method replaces the association to the current thread with the first ancestor that has not been aborted by calling *popAborted()* 's Control method, resuming it. The method also deals with informing all the registered StaticResource objects that a new thread association has been established.

2.4 sendingReply method

This is a public method of the "CurrentTransaction" class and it is called to inform the Coordinator of the current transaction that an imminent reply is about to be performed and so the association between the transaction and the current thread should be ended.

The following are the JavaDoc and the declaration of the method:

```
    /**Informs the object's Coordinator that a reply is being sent to the
    ↪ client.
     *
     * @param id      The request identifier.
     * @param holder  The context to be returned on the reply.
     *
     * @exception INVALID_TRANSACTION  The current transaction has
     ↪ outstanding work
     *   on this reply, and has been marked rollback-only, or the reply is
     ↪ returning
     *   when a different transaction is active from the one active when the
     ↪ request
     *   was imported.
     * @exception TRANSACTION_ROLLEDBACK  The current transaction has already
     ↪  been
     *   rolled back.
     *
     * @see
     */
    static void sendingReply( int id,
                                       PropagationContextHolder holder )
        throws INVALID_TRANSACTION, TRANSACTION_ROLLEDBACK {
```

The method is responsible to check that the current transaction is actually still active and there are no pending computation that must be terminated. To accomplish the first task the "**endAborted()**" method is called to check if the transaction has already been aborted, and if so a *TRANSACTION_ROLLBACK* exception is raised communicating that the transaction is already completed (*CompletionStatus.COMPLETED_YES*). For what concern the second task, the method checks the Coordinator by calling his "*replyAction*" method which returns an identifier of his current state:

- If there are still subtransactions that have not been completed yet (the value *CoordinatorImpl.activeChildren* has been returned) an *INVALID_TRANSACTION* exception is raised communicating the error code "*MinorCode.UnfinishedSubtransactions*"
- If the transaction is still associated to a thread different from the current one or there are outgoing requests of the Coordinator that have not been completed yet an *INVALID_TRANSACTION* exception is raised communicating the error code "*MinorCode.DeferredActivities*"

Finally, the method deals with terminating the association with the transaction and the current thread keeping consistent the list of transactions associated with the current thread, and resuming the last transaction, associated with the current thread, that had been suspended by calling "*endCurrent()*" method.

3 Code Inspection

3.1 Class Analysis

1. **All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.**

   - method names:
     - line 1302: "endAll()" method is not implemented
     - line 1358: "shutdown()" method is not implemented
     - line 1371: "dump()" method is not implemented
   - method variables:

  – line 750: the argument “id” of the method “sendingRequest” is never been used inside the function so it could be removed

```
750       static void sendingRequest( int id,
751                                                       PropagationContextHolder
                                                          ↪  holder )
752            throws TRANSACTION_ROLLEDBACK, TRANSACTION_REQUIRED {
```

6. **Class variables, also called attributes, are mixed case, but might begin with an underscore (‘_’) followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: _windowHeight, timeSeriesData.**

   - line 111: the variable “m_tid” doesn’t respect the naming convention because the underscore could only appear at the beginning of the name

```
111       private static ThreadLocal m_tid=new ThreadLocal();
```

23. **Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).**

    For the following public and friendly methods and class variables is not provided a javadoc documentation:

    - line 346: “isTxAssociated()” public method

```
346       public static boolean isTxAssociated() {
```

    - line 102 “statsOn()” friendly method

```
102       static boolean statsOn=false;
```

    - line 119 “_logger”: firendly class variable

```
119     static Logger _logger = LogDomains.getLogger(CurrentTransaction.
        ↪ class, LogDomains.TRANSACTION_LOGGER);
```

    The following are methods for which it is reported a javadoc documentation but the meaning of some arguments or thrown exception is not clarified:

- line 374: the meaning of the argument “endAssociation” is not provided in the documentation

```
353     /**Ensures that an association with an aborted transaction is dealt with cleanly.
354      *
355      *
356      * TN - do not dissociate thread even if it's aborted!!
357      *
358      * If the current Control object represents a transaction that  has been
359      * aborted, this method replaces the association by one with the first
360      * ancestor that has not been aborted, if any, or no association, and the
361      * method returns true as the output parameter. Otherwise the method returns
362      * false as the output parameter.
363      * <p>
364      * If there is a current Control object in either case it is returned,
365      * otherwise null is returned.
366      *
367      * @param aborted  A 1-element array which will hold the aborted indicator.
368      *
369      * @return  The current Control object.
370      *
371      * @see
372      */
373     private static ControlImpl
374         endAborted( boolean[/*1*/] aborted, boolean endAssociation ) {
```

- line 493: It is not specified when the method “getCurrent” could raise the exception TRANSACTION_ROLLEDBACK

```
480     /**Returns the current Control object.
481      * <p>
482      * That is, the Control object that corresponds to the thread
483      * under which the operation was invoked. If there is no such association the
484      * null value is returned.
485      *
486      * @param
487      *
488      * @return  The current Control object.
489      *
490      *
491      * @see
492      */
493     public static ControlImpl getCurrent()
494         throws TRANSACTION_ROLLEDBACK {
```

- line 1199: the meaning of the argument “timeout” is not provided in the documentation

```
1192    /**
```

```
     * Recreates a transaction based on the information contained
     ↪ in the
     * transaction id (tid) and associates the current thread of
     ↪ control with
     * the recreated transaction.
     *
     * @param tid  the transaction id.
     */
    public static void recreate(GlobalTID tid, int timeout) {
```

25. **The class or interface declarations shall be in the following order:**

    The following class variables should be declared before the private ones because they are friendly (as described in the point **28.** these variables could be declared private because they are used only inside this class)

    - line 102: “statsOn” friendly variable

```
101     //store the suspended and associated transactions support only
        ↪  if stats are required
102     static boolean statsOn=false;
```

    - line 119: “ _logger” friendly variable

```
116   /*
117     Logger to log transaction messages
118   */
119   static Logger _logger = LogDomains.getLogger(CurrentTransaction.
      ↪ class, LogDomains.TRANSACTION_LOGGER);
```

26. **Methods are grouped by functionality rather than by scope or accessibility.**

    - riga 493 getCurrent(): forse meglio se messa all’inizio tra getter/setter
    - riga 521 getCurrentCoordinator(): forse meglio se messa all’inizio tra getter/setter

27. **Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.**

    - TODO ....
    - SonarQube:
      – duplicated line OK
      – long methods OK ($<$ 100 lines)
      – class complexty OK ($<$ 200 ciclomatic complexity)
      – breaking encapsulation TODO ....

– coupling / cohesion OK (< 20 class dependancies)

28. **Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).**

    The following class variables could be declared private because they are only used inside the current class:

    - line 102: “statsOn” friendly variable

```
102      static boolean statsOn=false;
```

    - line 119: “_logger” friendly variable

```
119    static Logger _logger = LogDomains.getLogger(CurrentTransaction.class, LogDomains.TRANSACTION_LOGGER);
```

3.2 Method analysis: “endAborted”

```
353     /**Ensures that an association with an aborted transaction is dealt with cleanly.
354      *
355      *
356      * TN - do not dissociate thread even if it's aborted!!
357      *
358      * If the current Control object represents a transaction that has been
359      * aborted, this method replaces the association by one with the first
360      * ancestor that has not been aborted, if any, or no association, and the
361      * method returns true as the output parameter. Otherwise the method returns
362      * false as the output parameter.
363      * <p>
364      * If there is a current Control object in either case it is returned,
365      * otherwise null is returned.
366      *
367      * @param aborted  A 1-element array which will hold the aborted indicator.
368      *
369      * @return  The current Control object.
370      *
371      * @see
372      */
373     private static ControlImpl
374         endAborted( boolean[/*1*/] aborted, boolean endAssociation) {
375
376         // Get the current thread identifier, and the corresponding Control object
377         // if there is one.
378
379         boolean completed = true;
380         aborted[0] = false;
381
382         ControlImpl result = (ControlImpl)m_tid.get();
383
```

```
    // If there is a current Control object, and it represents a
    ↪ transaction that
    // has been aborted, then we need to end its association with the
    ↪ current
    // thread of control.

    if( result != null )
        try {
            completed = (result.getTranState() != Status.StatusActive);
        } catch( Throwable exc ) {
_logger.log(Level.FINE,"", exc);
        }

    if( result != null && completed ) {
        if (endAssociation) {
      synchronized(CurrentTransaction.class){
  if(statsOn){
          Thread thread = Thread.currentThread();
                threadContexts.remove(thread);
  }
  m_tid.set(null);

              // XA support: If there was a current IControl, inform all
              ↪ registered
              // StaticResource objects of the end of the thread
              ↪ association.
              // Allow any exception to percolate to the caller.

              if( statics != null )
                    statics.distributeEnd(result,false);

              // Discard all stacked controls that represent aborted or
              ↪ unrecognised
              // transactions.

              result = result.popAborted();

              // If there is a valid ancestor, make it the current one.

              if( result != null ) {
    m_tid.set(result);
    if(statsOn){
            Thread thread = Thread.currentThread();
                      threadContexts.put(thread,result);
                      suspended.removeElement(result);
    }
              }

              // XA support: If there is a stacked context, inform all
              ↪ registered
              // StaticResource objects of the new thread association.
              // Allow any exception to percolate to the caller.

              if( statics != null )
                    statics.distributeStart(result,false);
}
        }
        aborted[0] = true;
    }

if(_logger.isLoggable(Level.FINEST))
{
  Thread thread = Thread.currentThread();
  _logger.logp(Level.FINEST,"CurrentTransaction","endAborted()",
      "threadContexts.get(thread)␣returned␣" +
      result + "␣for␣current␣thread␣" + thread);
}

```

```
446         return result;
447     }
```

8. **Three or four spaces are used for indentation and done so consistently**
   Blocks of four spaces are used for indentation along the method (even if multiple times in the form of tab characters instead of spaces (see point **9.** below)), but many times the indentation rules are not applied correctly:

   - Line **392** not correctly indented
   - Content of if() block fromline **396** to **434** not correctly indented
   - Content of synchronized() block from line **397** to **433** and its closing bracket at line **433** not correctly indented
   - Content of if() block from line **398** to **401** not correctly indented
   - Content of if() block at line **408** not correctly indented
   - Content of if() block from line **418** to **425** not correctly indented
   - Content of if() block from line **420** to **424** not correctly indented
   - Content of if() block at line **431** not correctly indented Lines **442**, **443** not correctly indented

9. **No tabs are used to indent**

   Starting from line **398** until line **444**, lines **435**-6-7 excluded, each line that is not a blank line is indented using tabs instead of spaces.

10. **Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block)**

    The author has used the "Kernighan and Ritchie" bracing style along all the method, except for the if() block from line **438** to line **444**, where he used the "Allman" style. This lack of consistency should be avoided.

11. **All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces**

    - if() block from line **388** to **393** not surrounded by curly braces
    - if() block from line **408** to **409** not surrounded by curly braces
    - if() block from line **431** to **432** not surrounded by curly braces

13. **Where practical, line length does not exceed 80 characters**
    All the lines of code of the method do not exceed 80 characters; however, some lines of either Javadoc or comments do:

- Javadoc: line **353**
- Comments: lines **376**, **384**, **385**, **404**, **405**, **411**, **427**

The peak is at line **411**, which is 90 characters long.

17. **A new statement is aligned with the beginning of the expression at the same level as the previous line**

    As already mentioned in point **8.** together with other indentation errors, there are some lines of subsequent instructions that should be aligned since they are at the same level, but are not:

    - Lines **399** and **400**
    - Lines **421**, **422**, **423**
    - Lines **441**, **442**, **443**: lines **442** and **443** should be aligned with the open bracket at line **441**

3.3 Method analysis: "sendingReply"

```java
/**Informs the object's Coordinator that a reply is being sent to the client.
 *
 * @param id      The request identifier.
 * @param holder  The context to be returned on the reply.
 *
 * @exception INVALID_TRANSACTION  The current transaction has outstanding work
 *   on this reply, and has been marked rollback-only, or the reply is returning
 *   when a different transaction is active from the one active when the request
 *   was imported.
 * @exception TRANSACTION_ROLLEDBACK  The current transaction has already been
 *   rolled back.
 *
 * @see
 */
static void sendingReply( int id,
                                       PropagationContextHolder holder )
    throws INVALID_TRANSACTION, TRANSACTION_ROLLEDBACK {

    // Zero out context information.
    // Ensure that the cached reference to the ORB is set up, and that the Any
    // value in the context is initialised.
    //$ The following is necessary for the context to be marshallable. It is a
    //$ waste of time when there is no transaction, in which case we should be
    //$ throwing the TRANSACTION_REQUIRED exception (?).

    if( emptyContext.implementation_specific_data == null ) {
        ORB orb = Configuration.getORB();
        emptyContext.implementation_specific_data = orb.create_any();
        emptyContext.implementation_specific_data.insert_boolean(false);
    }

```

```
        // COMMENT(Ram J) There is no need to send an empty context, if a tx
        // is not available. The PI based OTS hooks will not send a tx
        ↪ context
        // in the reply.
        /*
        holder.value = emptyContext;
        */

        // Ensure that the current Control object is valid.  Return
        ↪ immediately if not.

        boolean[] outBoolean = new boolean[1];
        ControlImpl current = endAborted(outBoolean, true);  // end
        ↪ association
        if( outBoolean[0] ) {
            importedTransactions.remove(Thread.currentThread());
            TRANSACTION_ROLLEDBACK exc = new TRANSACTION_ROLLEDBACK(0,
            ↪ CompletionStatus.COMPLETED_YES);
            throw exc;
        }

        // Get the global identifier of the transaction that was imported
        ↪ into this
        // thread.  If there is none, that is an error.

        Thread thread = Thread.currentThread();
        GlobalTID importedTID = (GlobalTID)importedTransactions.remove(thread
        ↪ );

        // If there is no import information, and no current transaction,
        ↪ then return
        // the empty context.

        if( importedTID == null && current == null ) {
            return;
        }

        // Check that the current transaction matches the one that was
        ↪ imported.

        StatusHolder outStatus = new StatusHolder();
        try {
            if( importedTID == null ||
                current == null ||
                !importedTID.isSameTID(current.getGlobalTID(outStatus)) ||
                outStatus.value != Status.StatusActive ) {
                INVALID_TRANSACTION exc = new INVALID_TRANSACTION(MinorCode.
                ↪ WrongContextOnReply,CompletionStatus.COMPLETED_YES);
                throw exc;
            }
        } catch( SystemException ex ) {
            _logger.log(Level.FINE,"", ex);
            INVALID_TRANSACTION exc = new INVALID_TRANSACTION(MinorCode.
            ↪ WrongContextOnReply,CompletionStatus.COMPLETED_YES);
            throw exc;
        }

        //$Get the Coordinator reference.

        CoordinatorImpl coord = null;
        Coordinator coordRef = null;
        try {
            if (Configuration.isLocalFactory()) {
                coord = (CoordinatorImpl) current.get_localCoordinator();
            } else {
                coordRef = current.get_coordinator();
                coord = CoordinatorImpl.servant(coordRef);
            }

            //    _logger.log(Level.FINE,"Servant = "+coord);

            // Check the Coordinator before sending the reply.
            // We must do this before ending the thread association to allow
            ↪ the
            // Coordinator to take advantage of registration on reply if
            ↪ available.
            // Note that if the Coordinator returns forgetMe, the global
            ↪ identifier
            // will have been destroyed at this point.

            CoordinatorImpl forgetParent = null;
            int[] outInt = new int[1];
            //StatusHolder outStatus = new StatusHolder();
            try {
                forgetParent = coord.replyAction(outInt);
            } catch( Throwable exc ) {
                _logger.log(Level.FINE,"", exc);
            }

            int replyAction = outInt[0];
            if( replyAction == CoordinatorImpl.activeChildren ) {
                try {
                    coord.rollback_only();
                } catch( Throwable ex ) {
                    _logger.log(Level.FINE,"", ex);
                }

                INVALID_TRANSACTION exc = new INVALID_TRANSACTION(MinorCode.
                ↪ UnfinishedSubtransactions,
                                                               CompletionStatus
                                                               ↪ .
                                                               ↪ COMPLETED_YES
                                                               ↪ );
                throw exc;
            }

            // End the current thread association.

            endCurrent(false);

            // If the transaction needs to be cleaned up, do so now.
            // We ignore any exception the end_current may have raised in
            ↪ this case.
            // The Control object is destroyed before the Coordinator so that
            ↪  it is not
            // in the suspended set when the Coordinator is rolled back.

            if( replyAction == CoordinatorImpl.forgetMe ) {
                current.destroy();
                coord.cleanUpEmpty(forgetParent);
            }

            // Otherwise, we have to check this reply.

            else {
                if( current.isAssociated() ||
                        current.isOutgoing() ) {
                    try {
                        coord.rollback_only();
                    } catch( Throwable exc ) {
                        _logger.log(Level.FINE,"", exc);
                    }

                    INVALID_TRANSACTION exc = new INVALID_TRANSACTION(
                    ↪ MinorCode.DeferredActivities,
1167                                                                      CompletionStatus
                                                                          ↪ .
                                                                          ↪ COMPLETED_YES
                                                                          ↪ );
1168                        throw exc;
1169                    }
1170
1171                    current.destroy();
1172                }
1173
1174            } catch( INVALID_TRANSACTION exc ) {
1175                throw exc;
1176            } catch( Unavailable exc ) {
1177                _logger.log(Level.FINE,"", exc);
1178                // Ignore
1179            } catch( SystemException exc ) {
1180                _logger.log(Level.FINE,"", exc);
1181                // Ignore
1182            }
1183
1184            // Create a context with the necessary information.
1185            // All we propagate back is the transaction id and implementation
            ↪ specific data.
1186
1187            holder.value = new PropagationContext(0,new TransIdentity(null,null,
            ↪ importedTID.realTID),
1188                                                  new TransIdentity[0],
                                                      ↪ emptyContext.
                                                      ↪ implementation_specific_data
                                                      ↪ );
1189
1190        }
```

5. **Method names should be verbs, with the first letter of each addition word capitalized.**

   - line **1048**: the called method "create_any()" should be renamed in "createAny()"
   - line **1049**: the called method "insert_boolean()" should be renamed in "insertBoolean()"
   - line **1105**: the called method "get_localCoordinator()" should be renamed in "getLocalCoordinator()"
   - line **1107**: the called method "get_coordinator()" should be renamed in "getCoordinator()"
   - line **1108**: the called method "servant()" should be renamed in "getServant()"
   - line **1131** and **1161**: the called method "rollback_only()" should be renamed

8. **Three or four spaces are used for indentation and done so consistently**

   - line **1167** not correctly indented (2 more spaces)
   - line **1188** not correctly indented (2 more spaces)

13. **Where practical, line length does not exceed 80 characters.**

    - Some lines of the javadoc documentation of this method exceed 80 characters
    - Some lines of the following comment blocks exceed 80 characters length:
        - block from line **1040** to **1044**
        - comment at line **1059**
        - comment at line **1069**
        - comment at line **1075**
        - block from line **1114** to **1116**
        - block from line **1146** to **1147**
        - comment at line **1185**

14. **When line length must exceed 80 characters, it does NOT exceed 120 characters.**

    - line **1090**: line length is 129 characters
    - line **1095**: line length is 125 characters

17. **A new statement is aligned with the beginning of the expression at the same level as the previous line.**

    - line **1036**: the argument “holder” should be indented at the same level of the argument “id”
    - line **1137**: the argument “CompletionStatus.COMPLETED_YES” should be indented at the same level of the previous argument
    - line **1159**: “current.isOutgoing()” should be indented at a lower level
    - line **1167**: the argument “CompletionStatus.COMPLETED_YES” should be indented at the same level of the previous argument

19. **Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.**

    - line **1056**: It is not specified a date after which the commented code can be deleted
    - line **1111**: It is not specified neither the reason nor the date
    - line **1121**: It is not specified neither the reason nor the date

29. **Check that variables are declared in the proper scope**

    - line **1102**: “coordRef” declaration should be moved inside the else block at line 1106 because the variable is used only there.

33. **Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.**

    - line **1061**, **1062**, **1072**, **1073**, **1084**, **1101**, **1119**, **1120**: declarations should be moved at the beginning of the function (line **1038**)
    - line **1136**: the declaration of the exception should be moved at the beginning of the if() block at line **1130**. The exception could be also immediately thrown instead of assigning it to a temporary variable.
    - line **1166**: the declaration of the exception should be moved at the beginning of the if() block at line **1160**. The exception could be also immediately thrown instead of assigning it to a temporary variable.

40. **Check that all objects (including Strings) are compared with "equals" and not with "=="**

    - line **1089**: "!=" is used instead of !equals()

42. **Check that error messages are comprehensive and provide guidance as to how to correct the problem**
    At lines **1094**, **1125**, **1133**, **1163**, **1177**and **1180** it is not provided an explanation for the logged exception

44. **Check that the implementation avoids "brutish programming"**

    - line **1065**: The constructor of TRANSACTION_ROLLEDBACK should be called using the constant "Undefined" declared in MinorCode class instead of "0"

51. **Check that the code is free of any implicit type conversions**

    - line **1062**: The called function "endAborted()" uses a one-element array to pass the boolean argument by reference. It should be better to use the object type Boolean in order to avoid indexes from going out-of-bounds
    - line **1123**: The called function "replyAction()" uses a one-element array to pass the integer argument by reference. It should be better to use the object type Integer in order to avoid indexes from going out-of-bounds

52. **Check that the relevant exceptions are caught**

    - line **1124**: it should be catched a "SystemException" instead of "Throwable"
    - line **1131** and **1161**: it should be catched an "Inactive" exception instead of "Throwable"

4 Appendix

4.1 Reference documents

- Transaction Service Specification
  Version: 1.4
  Author: OMB - Object Management Group
  Link: `http://www.omg.org/spec/TRANS/1.4/`
- Java™ Transaction Service (JTS) Specification
  Version: 1.0
  Author: Sun Microsystems Inc
  Link: `http://download.oracle.com/otndocs/jcp/7309-jts-1.0-spec-oth-JSpec/`

4.2 Hours of work

Here is how long it took to redact this document:

- Matteo Bulloni: ~ #### hours
- Marco Cannici: ~ #### hours