Software Engineering 2: Code Inspection
version 1

Chitti Eleonora, De Nicolao Pietro, Delbono Alex
Politecnico di Milano

January 5, 2016

Contents

Chapter 1

Description of the code

1.1 Assigned classes

The classes assigned to our group are the following:

- CoordinatorLog
- CoodinatorLogSection
- RecoveryManager

All these classes are located in the `com.sun.jts.CosTransactions` package of Java GlassFish Server[1].

1.2 Functional role of classes

1.2.1 Java Transaction Service

All the classes to review are part of the Java Transaction Service (JTS) implementation by Sun.

The complete specification of Java Transaction Service is available on Oracle's website [3] and defines JTS as follows:

> This is the Java Transaction Service (JTS) Specification. JTS specifies the implementation of a transaction manager which supports the JTA specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification at the low-level.

[1] `https://glassfish.java.net/`

> JTS uses the CORBA OTS interfaces for interoperability and portability (that is, **CosTransactions** and CosTSPortability). These interfaces define a standard mechanism for any implementation that utilizes IIOP (Internet InterORB Protocol) to generate and propagate transaction context between JTS Transaction Managers. Note, this also permits the use of other API over the IIOP transport mechanism to be used; for example, RMI over IIOP is allowed.

The Wikipedia page about the Java Transaction Service[2] also gives us some useful information:

> The **Java Transaction Service (JTS)** is a specification for building a transaction manager that maps onto the Object Management Group (OMG) Object Transaction Service (OTS) used in the Common Object Request Broker Architecture (CORBA) architecture. It uses General Inter-ORB Protocol (IIOP) to propagate the transactions between multiple JTS transaction managers.

Summing up, we are analyzing Sun's implementation of the JTS specification, which is the Java mapping of the CORBA Object Transaction Service Specification [4].

The JTS specification also provides a component-like diagram (Figure 1.1) describing the structure of a possible implementation of a Transaction Manager.

1.2.2 CosTransactions package

The CosTransactions package is the Java implementation of the CosTransactions component, which is defined in the Transaction Service Specification of the Object Management Group [4].

The particular subset of classes assigned to us implement part of the *Reliability control system* [5, p. 311] of the transaction manager. It ensures consistency and durability of the data in case of failure.

1.2.3 CoordinatorLog class

The CoordinatorLog class is contained in the CosTransactions package and is package-private. From the JavaDoc for the class, L.79:

[2] `https://en.wikipedia.org/wiki/Java_transaction_service`

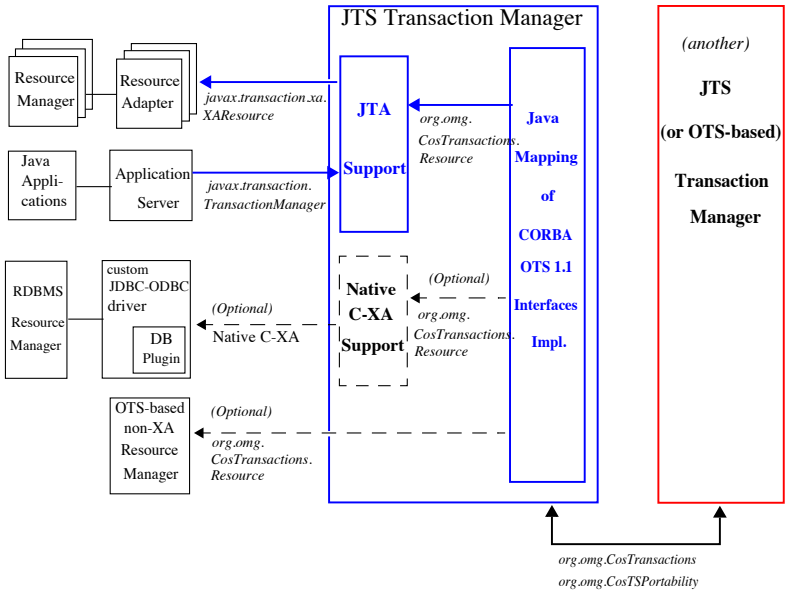

Figure 1.1: This figure is taken from the JTS specification and shows the implementation of a JTS Transaction Manager.

> The CoordinatorLog interface provides operations to record transaction-specific information that needs to be persistently stored at a particular point in time, and subsequently restored.

The CoordinatorLog class keeps the **transaction log** and writes it to persistent storage when needed. The log is a fundamental component in transaction managers and database systems. The log and the recovery manager together ensure consistency (i.e. the data must not stay in an inconsistent state) and durability (i.e. the committed transactions' data must be preserved) especially in case of failures. The log allows to determine which transactions ended with a commit, which ones aborted and which ones are still running. Details about log structure and reliability management are well explained in database theory books (see for example [5]).

CoordinatorLog implements the *LogUpcallTarget* interface that resides in the same package and is also package-private. Its JavaDoc reads:

> The LogUpcallTarget interface provides an operation that the log will call in the event it goes short-on-storage. This class must be sub-classed in order to implement the method that will handle the situation.

The LogUpcallTarget provides the signature for the `upcall()` method.

1.2.4 CoordinatorLogSection class

The CoordinatorLogSection class is contained in the CosTransactions package, is package-private and does not implement any interface. This class stores the data about the section of a CoordinatorLog. The CoordinatorLog keeps track of the sections through an Hashtable (`sectionMapping` attribute). From the JavaDoc for the class:

> The CoordinatorLogSection class stores information relevant to a section.

1.2.5 RecoveryManager class

The RecoveryManager class is contained in the CosTransactions package, it is public and does not implement any interface. From the JavaDoc for the class, L.85:

> This class manages information required for recovery, and also general state regarding transactions in a process.

This class may be the implementation of the *RecoveryCoordinator* interface defined in the OMG OTS specification [4, p. 47].

The RecoveryManager class is the component responsible to ensure consistency and durability in case of a fault, just as in relational databases. When the system restarts after a failure, the RecoveryManager redoes committed transactions and undoes uncommitted (or aborted) ones, like in the *warm restart* protocol in database systems [5, p. 319].

Chapter 2

Results of inspection

This chapter contains the results of the code inspection that we did on the assigned classes and methods. All the points of the checklist [2] were checked, and we also found other bad practices not listed in the checklist.

2.1 Notation

- The items of the code inspection checklist [2] will be referred as follows: **C1**, **C2**, ...
- A specific line of code will be referred as follows: L.1234.
- An interval of lines of code will be referred as follows: L.1234-1289.

2.2 Issues

2.2.1 CoordinatorLog class

1. **C7** There is a constant in the class which does not follow the naming convention at L.110.
2. **C12** Blank lines at the beginning and at the end of method bodies are not consistent throughout the class. Missing blank line after ending of method block after L.1606.
3. **C23** The CoordinatorLog constructor at L.206 is missing the JavaDoc. All the duplicated methods with the `logPath` parameter (see **C27**) are missing the JavaDoc.
4. **C25** There are several problems in the class declaration.

- Commented out code at L.112.
- Static and non-static attributes are interleaved in the CoordinatorLog declaration (L.107-150).
- The visibility order of attributes is not respected (for instance L.143-144).
- A constructor is declared in the middle of other methods (L.324).
- There is a static block used for initialization at L.1104.
- Static and non-static methods are mixed.

5. **C27** The class is full of duplicated code, which is very bad for maintainability. There are couples of methods that differ only for the presence of the `logPath` parameter:

   (a) `reUse()` at L.264, L.288

   (b) `setLocalTID()` at L.865, L.879

   (c) `openLog()` at L.1127, L.1188

   (d) `getLogged()` at L.1248, L.1317

   (e) `addLog()` at L.1388, L.1399

   (f) `removeLog()` at L.1418, L.1465

   (g) `keypoint()` at L.1525, L.1607

   (h) `finalizeAll()` at L.1720, L.1759

   (i) `startKeypoint()` at L.1807, L.1874

   This is also highlighted in the comment at L.153:

   > All the methods which take "String logPath" as parameter are same as the ones with out that parameter. These methods are added for delegated recovery support

   These couples of methods respectively differ for a single line; where the `logPath` argument is not present, then we have:

   ```
   CoordinatorLogStateHolder logStateHolder = defaultLogStateHolder;
   ```

   When the `logPath` argument is present, instead we have:

   ```
   CoordinatorLogStateHolder logStateHolder = getStateHolder(logPath);
   ```

A feasible solution to remove this ugly duplication would be to edit the methods without the additional argument, making them call the corresponding methods with `logPath` equal to null. There, a simple conditional could handle both the cases.

6. **C28** Two attributes are of raw type `Hashtable`. Generics could be used to ensure type safety and to clarify the code.

7. **C27**, **C28** Some attributes (L.139-144) are package-private. If these attributes are accessed outside the CoordinatorLog class, that should be done via getters and setters. Anyways, the attributes of classes should be private unless there is a good reason to do otherwise, because non-private attributes can break encapsulation.

8. In the CoordinatorLog class, the name `logPath` is used both for a private attribute for the class (L.134) and as an argument in many methods which are clearly not setters (see above). This is a bad practice and generates confusion.

9. The `CoordinatorLog.java` file contains 4 classes (CoordinatorLog, CoordinatorLogStateHolder, CoordinatorLogSection, SectionPool): they should be split in separate files.

finalizeAll()

Two similar functions are assigned:

1. `synchronized static void finalizeAll()`

2. `synchronized static void finalizeAll(String logPath)`

As stated in subsection 2.2.1, these two methods are almost identical except for the first line of code, so they will be analyzed together.

1. **C8** Indentation with 4 spaces is used everywhere except for L.1721, L.1760.

2. **C11** No braces for single-statement blocks at L.1736, L.1742, L.1747, L.1775, L.1781, L.1786.

3. **C17** Indentation is wrong (L.1721, L.1760).

4. **C36** `LogFile.close()` boolean return value is discarded (L.1742, L.1781).

startKeypoint()

Two similar functions are assigned:

1. `synchronized static boolean startKeypoint( LogLSN keypointStartLSN )`
2. `synchronized static boolean startKeypoint( LogLSN keypointStartLSN, String logPath )`

As stated in subsection 2.2.1, these two methods are almost identical except for the first line of code, so they will be analyzed together.

1. **C11** No braces for single-statement blocks at L.1830, L.1897.
2. **C44** Brutish programming at L.1858-1865, L.1925-1932: a sequence of characters is assigned using an array of bytes. A better solution would use the native Java method `String.getBytes()`.
3. **C27** Duplicated code into the two `startKeyPoint` methods.

dump()

The main problem of this method is that it does nothing. Indeed:

1. the method has no return value;
2. the method only sets some local variables;
3. all the called methods have no side-effects;
4. the method is not required by any interface implemented by the CoordinatorLog class.

Actions are replaced with comments which shortly explain what should be done. This method should be either removed or modified.

1. **C13** Line length exceeds 80 characters at L.1968 with 86 characters and at L.1981 with 94 characters.
2. **C19** Code commented at L.1968 without date and with trivial motivation.
3. **C23** The Java code is incomplete and the specification is generic, although the function does nothing.

2.2.2 CoordinatorLogSection class

1. The clause `extends java.lang.Object` (L.2078) in the class declaration is superfluous and should be eliminated.

2. **C28** Four attributes are of raw type `Vector`. Java Generics could be used to ensure type safety and to clarify the code.

3. **C27**, **C28** All the attributes (L.2079-2085-) are package-private. Their visibility should be changed to private and getters and setters should be added where appropriate in order to ensure encapsulation and reduce coupling.

doFinalize()

1. **C11** Braces are not used around the following one-statement blocks: L.2108, L.2111, L.2114, L.2117.

2. **C17** Inconsistent alignment before the equality sign at L.2120.

3. **C18** Missing comments in method body.

4. **C27** All the `if` statements are identical to the ones in the method `synchronized void reUse()` at L.2145.

reUse()

1. **C8** Wrong indentation at L.2163. Inconsistent alignment at L.2159.

2. **C9** Tabs are used for indentation at L.2163.

3. **C11** Braces are not used around the following one-statement blocks: L.2147, L.2150, L.2153, L.2156.

4. **C17** Inconsistent alignment before the equality sign at L.2159.

5. **C18** Missing comments in method body.

2.2.3 RecoveryManager class

1. **C1** Methods `getUniqueRMSet()` (L.792) and `getInDoubtXids()` (L.864) start with "get", but they are not getters of any attribute of the class: their name is misleading and should be changed.

2. **C17** Wrong alignment at L.171.

3. **C18** JavaDoc is missing for the parameters of methods:
   (a) `getUniqueRMSet(Enumeration xaResourceList)` (L.789-791)
   (b) `getInDoubtXids(XAResource xaResource)` (L.860-861)
4. **C19** There is a lot of commented code for example in the body of the `shutdown` method at L.746. There are also some methods entirely commented out:
   - L.1305-1349
   - L.1759-1816
5. **C28** Four attributes (L.145-157) are of raw type `Hashtable`. Generics could be used to ensure type safety and to clarify the code.
6. **C28** One attribute (L.171) is package-private. Attributes should mostly be private and accessed via getters and setters.
7. There is a static attribute `lockObject` (L.167) used as a lock object for the entire class. This is not really necessary since a lock can be acquired on the Class object.[1]
8. There is a *TODO* within a commented code block at L.930 which should be checked.
9. All the attributes and methods in this class are `static`. The use of the singleton design pattern should be considered instead.

initialise()

1. **C9** Tabs are used for indentation at L.195-199, L.210.
2. **C13** If tabs are considered as one character every line stays within 80 characters, but if a tab is considered as 4 characters (like 4 spaces), L.210 is too long.
3. **C10** Braces are used with the "Kernighan and Ritchie" style, but at L.196 "Allman style" is followed.
4. **C19** Commented out code is not motivated and does not contain a date (L.200).

[1]The Java™ Tutorials: Intrinsic Locks and Synchronization (`https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html`)

5. **C23** The comments do not explain the code well: for example at L.191 it is not clear what the comment refers to and why there is commented out code at line L.200.

6. **C53** At L.222 the `catch` block simply prints the stack trace: probably this is not the most appropriate behaviour.

Appendix A

Appendix

A.1 Software and tools used

- LaTeX for typesetting this document.
- GitHub[1] for version control and distributed work.

A.2 Hours of work

The statistics about commits and code contribution are available on GitHub [2]. Please keep in mind that many commits are actually group work (when this is the case, it is stated in the commit message).

- Eleonora Chitti: 7 hours
- Alex Delbono: 6 hours
- Pietro De Nicolao: 10 hours

[1] `https://github.com`
[2] `https://github.com/pietrodn/se2-mytaxiservice`

Bibliography

[1] Software Engineering 2 Project, AA 2015/2016, *Project goal, schedule and rules*

[2] Software Engineering 2 Project, AA 2015/2016, *Assignment 3: Code Inspection*

[3] Oracle, *Java^{TM} Transaction Service Specification*
Version: 1.0, Release: December 8, 1999
`http://download.oracle.com/otn-pub/jcp/7309-jts-1.0-spec-oth-JSpec/jts1_0-spec.pdf`

[4] Object Management Group, *Transaction Service Specification*
September 2003, Version 1.4
`http://doc.omg.org/formal/2003-09-02.pdf`

[5] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone, *Database Systems concepts, languages & architectures*, McGraw-Hill, 1999