Code inspection

Mirko Basilico, Federico Dazzan
12/06/2016
version 1.0

June 12, 2016

Contents

1 Assigned class

We had to analyze the class Namespace, that has the following pattern:
netty-socketio/src/main/java/com/corundumstudio/socketio/namespace/Namespace.java

2 Functional role of the class

Being this class completely undocumented and without any comment inside it was quite hard to get an understanding of what this class really does. To accomplish that we had to do a reverse engineering of all the imports and methods; one thing that helped was realizing the class is from a software called socket.io, actually this is an opensource java implementation of the original socket.io software written in javascript. On the socket.io website we were able to learn that this is a software in which other applications are able to build on, basically a library for realtime web applications. It enables realtime, bi-directional communication between web clients and servers. One of the function this software offers is "Namespaces", Socket.IO allows you to "namespace" your sockets, which essentially means assigning different endpoints or paths.

The class we are analyzing is the main class implementing this function in the code, one of the most important function of namespaces is providing rooms, this means that within each namespace it's possible to define arbitrary channels that sockets can join and leave. The class uses ConcurrentMaps from the default java library to keep track of the room association to client and viceversa, most of the methods in our class use this maps to perform the intended function. The other important part is "listeners", these components are used to perform relevant actions on clients behaviors like connecting or disconnecting.

3 List of issues found by applying the checklist

This chapter illustrates the issues found in the assigned code, following the checklist provided by the professor.

Here we report only the points of the checklist for which we have found some issues in the code

3.1 Point 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.

Problems found

- (Line 264) The name of the variable "clients" is inappropriate. In fact, when the method "join" of line 263 is called, giving it the map clientRooms as first parameter, with key and value of the corresponding type (as happens at line 282),the variable "clients" will refer to a set that will contain the names of all the rooms in which there is the client associated to the given key map. The name "clients" instead suggests that this variable refers to a set of socketIOClient. A possible name that may be more appropriate for this variable could be "mapValues", since it is a name which is fine for any kind of maps given as parameter to the join method.
- (line 267) As mentioned in the previous point for the variable "clients" holds also for the variable "oldClients" of the same method. A possible name for this variables could be "oldMapValues".

```
262
263     private <K, V> void join(ConcurrentMap<K, Set<V>> map, K key, V value) {
264         Set<V> clients = map.get(key);
265         if (clients == null) {
266             clients = Collections.newSetFromMap(PlatformDependent.<V, Boolean>newConcurrentHashMap());
267             Set<V> oldClients = map.putIfAbsent(key, clients);
268             if (oldClients != null) {
269                 clients = oldClients;
270             }
271         }
272         clients.add(value);
273         // object may be changed due to other concurrent call
274         if (clients != map.get(key)) {
275             // re-join if queue has been replaced
276             join(map, key, value);
277         }
278     }
279
280     public void join(String room, UUID sessionId) {
281         join(roomClients, room, sessionId);
282         join(clientRooms, sessionId, room);
283     }
```

- (line 307) The name of the method "getRooms" is inappropriate, since it suggests that the method returns all the rooms of the namespace, while it returns only the rooms associated with the SocketIOClient given as parameter. A possible meaningful name for this method could be getClientRooms.

```
307     public Set<String> getRooms(SocketIOClient client) {
308         Set<String> res = clientRooms.get(client.getSessionId());
309         if (res == null) {
310             return Collections.emptySet();
311         }
312         return Collections.unmodifiableSet(res);
313     }
```

3.2 Point 5

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

Problems found

The following methods have name that are not verbs:

- (Line 126)

```
126     public void onEvent(NamespaceClient client, String eventName, List<Object> args, AckRequest ackRequest) {
```

- (Line 172)

```
172     public void onDisconnect(SocketIOClient client) {
```

- (Line 192)

```
192     public void onConnect(SocketIOClient client) {
```

3.3 Point 11

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

Problems found

The following if statements with only one statement to execute are not surrounded by curly braces:

- (line 225)

```
225         if (this == obj)
226             return true;
```

- (line 227)

```
227         if (obj == null)
228             return false;
```

- (line 229)

```
229         if (getClass() != obj.getClass())
230             return false;
```

- (line 233)

```
233             if (other.name != null)
234                 return false;
```

- (line 235)

```
235         } else if (!name.equals(other.name))
236             return false;
```

3.4 Point 13

Where practical, line length does not exceed 80 characters.

Problems found

The following lines exceed 80 characters, and they can be easily broken, making the code more readable:

- (Line 96) In this case the line is already broken, but it still exceed 80 characters. To avoid this, the line can be broken after the first parameter, instead of after the second one.

```
96      public void addMultiTypeEventListener(String eventName, MultiTypeEventListener listener,
97              Class<?>... eventClass) {
```

- (Line 112)

```
112     public <T> void addEventListener(String eventName, Class<T> eventClass, DataListener<T> listener) {
```

- (Line 126)

```
126     public void onEvent(NamespaceClient client, String eventName, List<Object> args, AckRequest ackRequest) {
```

- (Line 176)

```
176         storeFactory.pubSubStore().publish(PubSubType.LEAVE, new JoinLeaveMessage(client.getSessionId(), getName(), getName()));
```

- (Line 194)

```
194         storeFactory.pubSubStore().publish(PubSubType.JOIN, new JoinLeaveMessage(client.getSessionId(), getName(), getName()));
```

- (Line 252)

```
252         storeFactory.pubSubStore().publish(PubSubType.JOIN, new JoinLeaveMessage(sessionId, room, getName()));
```

- (Line 287)

```
287         storeFactory.pubSubStore().publish(PubSubType.LEAVE, new JoinLeaveMessage(sessionId, room, getName()));
```

3.5 Point 18

Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

Problems found

The class is very poorly commented. In fact the comments at the beginning of the class don't adequately explain what the class do, and there is no comments that explains what the methods do.

3.6 Point 23

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

Problems found

There is no Javadoc associated to the class.

3.7 Point 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.

Problems found

The following variables are not declared and initialized at the beginning of the block to which they belong:

- (Line 101) oldEntry

```
99          if (entry == null) {
100             entry = new EventEntry();
101             EventEntry<?> oldEntry = eventListeners.putIfAbsent(eventName, entry);
```

- (Line 116) oldEntry

```
114         if (entry == null) {
115             entry = new EventEntry<T>();
116             EventEntry<?> oldEntry = eventListeners.putIfAbsent(eventName, entry);
```

- (Line 267) oldClients

```
265         if (clients == null) {
266             clients = Collections.newSetFromMap(PlatformDependent.<V, Boolean>newConcurrentHashMap());
267             Set<V> oldClients = map.putIfAbsent(key, clients);
```

- (Line 326) result

```
319     public Iterable<SocketIOClient> getRoomClients(String room) {
320         Set<UUID> sessionIds = roomClients.get(room);
321 
322         if (sessionIds == null) {
323             return Collections.emptyList();
324         }
325 
326         List<SocketIOClient> result = new ArrayList<SocketIOClient>();
```

3.8 Point 40

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

Problems found

In the following case the objects are compared with "==" instead of with "equals", or with "!=" instead of with "!equals":

- (Line 140)

```
140             if (ackMode == AckMode.AUTO_SUCCESS_ONLY) {
```

- (Line 149)

```
149         if (ackMode == AckMode.AUTO || ackMode == AckMode.AUTO_SUCCESS_ONLY) {
```

- (Line 229)

```
229         if (getClass() != obj.getClass())
```

- (Line 274)

```
274         if (clients != map.get(key)) {
```

3.9 Other problems found that are not in the checklist

The execution flow will never enter the "if" at lines 102 and 117 because "oldEntry" will never be different from null. "entry" gets its value at line 98 and to enter the if at line 99 there must not be an association to "eventname" in "eventListeners", but this also implies that "eventListeners.putIfAbsent(eventName, entry)" will always return null.

As mentioned holds also for the "if" at lines 268.

```
95      @Override
96      public void addMultiTypeEventListener(String eventName, MultiTypeEventListener listener,
97              Class<?>... eventClass) {
98          EventEntry entry = eventListeners.get(eventName);
99          if (entry == null) {
100             entry = new EventEntry();
101             EventEntry<?> oldEntry = eventListeners.putIfAbsent(eventName, entry);
102             if (oldEntry != null) {
103                 entry = oldEntry;
104             }
105         }
```

```
110     @Override
111     @SuppressWarnings({"unchecked", "rawtypes"})
112     public <T> void addEventListener(String eventName, Class<T> eventClass, DataListener<T> listener) {
113         EventEntry entry = eventListeners.get(eventName);
114         if (entry == null) {
115             entry = new EventEntry<T>();
116             EventEntry<?> oldEntry = eventListeners.putIfAbsent(eventName, entry);
117             if (oldEntry != null) {
118                 entry = oldEntry;
119             }
120         }
```

```
263     private <K, V> void join(ConcurrentMap<K, Set<V>> map, K key, V value) {
264         Set<V> clients = map.get(key);
265         if (clients == null) {
266             clients = Collections.newSetFromMap(PlatformDependent.<V, Boolean>newConcurrentHashMap());
267             Set<V> oldClients = map.putIfAbsent(key, clients);
268             if (oldClients != null) {
269                 clients = oldClients;
270             }
271         }
```

4 Appendix

4.1 Document used

- Assignment 3 - Code Inspection.pdf (assignment instruction and code inspection checklist)

4.2 Hours of work

- Mirko Basilico: ~20 hours;
- Federico Dazzan: ~20 hours.