A.Y. 2015/2016

POLITECNICO DI MILANO

COMPUTER SCIENCE AND ENGINEERING

SOFTWARE ENGINEERING 2

MyTaxiService

Inspection Document

Authors

SARA PISANI – 854223

LEONARDO TURCHI – 853738

05/01/2016

MyTaxiService-Inspection.pdf · V 0.5

Table of contents

Table of contents 3

1. Introduction 8

1.1 Code inspection checklist 8

1.1.1 Naming Conventions 8

1.1.2 Indention 9

1.1.3 Braces 9

1.1.4 File organization 10

1.1.5 Wrapping lines 10

1.1.6 Comments 10

1.1.7 Java source files 10

1.1.8 Package and import statements 11

1.1.9 Class and interface declarations 11

1.1.10 Initialization and declarations 11

1.1.11 Method calls 12

1.1.12 Arrays 12

1.1.13 Object comparison 12

1.1.14 Output format 12

1.1.15 Computation, comparisons and assignments 13

1.1.16 Exceptions 13

1.1.17 Flow of control 13

1.1.18 Files 14

2. Classes that were assigned to the group 15

3. Functional role of assigned set of classes 17

4. List of issues found by applying the checklist 18

4.1 Analysis of the method getSSLPorts 18

4.1.1 Expected return 20

4.1.2 Naming conventions 20

4.1.3 Indention 21

4.1.4 Braces 21

4.1.5 File organization 22

4.1.6 Wrapping lines 22

4.1.7 Comments 22

4.1.8 Java source files 23

4.1.9 Package and import statements 23

4.1.10 Class and interface declarations 23

4.1.11 Initialization and declarations 24

4.1.12 Method calls 24

4.1.13 Arrays 24

4.1.14 Object comparison 24

4.1.15 Output format 24

4.1.16 Computation, comparisons and assignments 25

4.1.17 Exceptions 25

4.1.18 Flow of control 25

4.1.19 Files 25

4.2 Analysis of the method selectSecurityContext 26
4.2.1 Naming conventions 26
4.2.2 Indention 27
4.2.3 Braces 27
4.2.4 File organization 28
4.2.5 Wrapping lines 28
4.2.6 Comments 28
4.2.7 Java source file 28
4.2.8 Package and import statement 28
4.2.9 Class and interface declaration 28
4.2.10 Initialization and declaration 28
4.2.11 Method calls 29
4.2.12 Arrays 32
4.2.13 Object comparison 32
4.2.14 Output Format 32
4.2.15 Computation, Comparisons and assignments 32
4.2.16 Exceptions 32
4.2.17 Flow of control 33
4.2.18 Files 33
4.3 Analysis of the method sendUsernameAndPassword 34
4.3.1 Naming conventions 34
4.3.2 Indention 35
4.3.3 Braces 35

4.3.4 File organization 35
4.3.5 Wrapping lines 35
4.3.6 Comments 35
4.3.7 Java source file 35
4.3.8 Package and import statement 35
4.3.9 Class and interface declaration 35
4.3.10 Initialization and declaration 35
4.3.11 Method calls 35
4.3.12 Arrays 35
4.3.13 Object comparision 36
4.3.14 Output format 36
4.3.15 Computation, Comparisons and assignments 36
4.3.16 Exceptions 36
4.3.17 Flow of control 36
4.3.18 Files 36
5. Any other problem you have highlighted 36
6. Additional Material 36
7. Hours of works 36

AGGIORNARE SOMMARIO

1. Introduction

Code inspection is the systematic examination (often known as peer review) of computer source code. It is intended to find
mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills.

Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.
In this document will be applied Code Inspection techniques (supported by the review checklist) for the purpose of evaluating the general quality of selected code extracts from a release of the Glassfish 4.1application server.

1.1 Code inspection checklist

1.1.1 Naming Conventions

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
2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster, class ImageSprite
4. Interface names should be capitalized like classes
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(), computeTemperature()
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.

7. Constants are declared using all uppercase with words separated by an underscore. Examples: MIN_WIDTH, MAX_HEIGHT

1.1.2 Indention

8. Three or four spaces are used for indentation and done so consistently
9. No tabs are used to indent

1.1.3 Braces

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

    Example: avoid this:

```
if
(
condition
)
doThis();
```

    Instead do this:

```
if
(
condition
)
{
doThis();
}
```

1.1.4 File organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods)
13. Where practical, line length does not exceed 80 characters
14. When line length must exceed 80 characters, it does NOT exceed 120 characters

1.1.5 Wrapping lines

15. Line break occurs after a comma or an operator
16. Higher level breaks are used
17. A new statement is aligned with the beginning of the expression at the same level as the previous line

1.1.6 Comments

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

1.1.7 Java source files

20. Each java source file contains a single public class or interface
21. The public class is the first class or interface in the file
22. Check that the external program interfaces are implemented consistently with what is described in the Javadoc
23. Check that the Javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you)

1.1.8 Package and import statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

1.1.9 Class and interface declarations

25. The class or interface declarations shall be in the following order:
    A. class/interface documentation comment
    B. class or interface statement
    C. class/interface implementation comment, if necessary
    D. class (static) variables
        a. first public class variables
        b. next protected class variables
        c. next package level (no access modifier)
        d. last private class variables
    E. instance variables
        a. first public instance variables
        b. next protected instance variables
        c. next package level (no access modifier)
        d. last private instance variables
    F. constructors
    G. methods
26. Methods are grouped by functionality rather than by scope or accessibility.
27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate

1.1.10 Initialization and declarations

28. Check that variables and class members are of the correct type.
    Check that they have the right visibility (public/private/protected)
29. Check that variables are declared in the proper scope

30. Check that constructors are called when a new object is desired
31. Check that all object references are initialized before use
32. Variables are initialized where they are declared, unless dependent upon a computation
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

1.1.11 Method calls

34. Check that parameters are presented in the correct order
35. Check that the correct method is being called, or should it be a different method with a similar name
36. Check that method returned values are used properly

1.1.12 Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)
38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds
39. Check that constructors are called when a new array item is desired

1.1.13 Object comparison

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

1.1.14 Output format

41. Check that displayed output is free of spelling and grammatical errors

42. Check that error messages are comprehensive and provide guidance as to how to correct the problem
43. Check that the output is formatted correctly in terms of line stepping and spacing

1.1.15 Computation, comparisons and assignments

44. Check that the implementation avoids "brutish programming" (see http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html)
45. Check order of computation/evaluation, operator precedence and parenthesizing
46. Check the liberal use of parenthesis is used to avoid operator precedence problems
47. Check that all denominators of a division are prevented from being zero
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding
49. Check that the comparison and Boolean operators are correct
50. Check throw-catch expressions, and check that the error condition is actually legitimate
51. Check that the code is free of any implicit type conversions

1.1.16 Exceptions

52. Check that the relevant exceptions are caught
53. Check that the appropriate action are taken for each catch block

1.1.17 Flow of control

54. In a switch statement, check that all cases are addressed by break or return
55. Check that all switch statements have a default branch
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions

1.1.18 Files

57. Check that all files are properly declared and opened
58. Check that all files are closed properly, even in the case of an error
59. Check that EOF conditions are detected and handled correctly
60. Check that all file exceptions are caught and dealt with accordingly

2. Classes that were assigned to the group

The class assigned to the group is SecurityMechanismSelector

This class extends the generic Object class, and is a *Singleton*.
It belongs to the package *com.sun.enterprise.iiop.security*.

It uses many *import* to be able to call and use objects from the classes:

- *Java util*
- *Java security*
- *Sun corba*
- *Sun enterprise*
- *Org glassfish*
- *Javax*

This class implements the interface called

PostConstruct (from package *org.glassfish.hk2.api*).

Author: *Jerome Dochez*

Description: *classes implementing this interface register an interest in being notified when the instance has been created and the component is about to be place into commission.*

The class's constructor is responsible to read the client and server preferences from the config files.

The author of the class is *Nithya Subramanian.*

Refer to glassfish project homepage for a contact

- *glassfish.dev.java.net*

The class is subject to the terms of the *GPL Version 2 Licence*.

FATTO DA
LEO

3. Functional role of assigned set of classes

The class `SecurityMechanismSelector` is responsible for making various decisions for selecting security information to be sent in the IIOP message based on target configuration and client policies.

NOTE

This class can be called concurrently by multiple client threads. However, none of its methods need to be synchronized because the methods either do not modify state or are idempotent.

DA FINIRE

(SARA)

(su documento assignm. dice anche di mettere grafici..(forse del funzionamento della classe o dei metodi))

4. List of issues found by applying the checklist

4.1 Analysis of the method getSSLPorts()

SecurityMechanismSelector.java

```
344    public java.util.List<SocketInfo> getSSLPorts(IOR ior, ConnectionContext ctx)
345    {
346        CompoundSecMech mechanism = null;
347        try {
348            mechanism = selectSecurityMechanism(ior);
349        } catch(SecurityMechanismException sme) {
350            throw new RuntimeException(sme.getMessage());
351        }
352        ctx.setIOR(ior);
353        ctx.setMechanism(mechanism);
354
355        TLS_SEC_TRANS ssl = null;
356        if ( mechanism != null ) {
357            ssl = getCtc().getSSLInformation(mechanism);
358        }
359
360        if (ssl == null) {
361            if (isSslRequired()) {
362                // Attempt to create SSL connection to host, ORBInitialPort
363                IIOPProfileTemplate templ = (IIOPProfileTemplate)
364                    ior.getProfile().getTaggedProfileTemplate();
365                IIOPAddress addr = templ.getPrimaryAddress();
366                SocketInfo info = IORToSocketInfoImpl.createSocketInfo(
367                "SecurityMechanismSelector1",
368                        "SSL", addr.getHost(), orbHelper.getORBPort(orbHelper.getORB()));
369                //SocketInfo[] sInfos = new SocketInfo[]{info};
370                List<SocketInfo> sInfos = new ArrayList<SocketInfo>();
371                sInfos.add(info);
372                return sInfos;
373            } else {
374                return null;
375            }
376        }
377
378        int targetRequires = ssl.target_requires;
379        int targetSupports = ssl.target_supports;
380
381        /*
382         * If target requires any of Integrity, Confidentiality or
383         * EstablishTrustInClient, then SSL is used.
384         */
385        if (isSet(targetRequires, Integrity.value) ||
386                isSet(targetRequires, Confidentiality.value) ||
387                isSet(targetRequires, EstablishTrustInClient.value)) {
388            if (_logger.isLoggable(Level.FINE)) {
389                _logger.log(Level.FINE, "Target requires SSL");
390            }
391            ctx.setSSLUsed(true);
392            String type = "SSL";
393            if(isSet(targetRequires, EstablishTrustInClient.value)) {
394                type = "SSL_MUTUALAUTH";
395                ctx.setSSLClientAuthenticationOccurred(true);
396            }
397            //SocketInfo[] socketInfos = new SocketInfo[ssl.addresses.size];
398            List<SocketInfo> socketInfos = new ArrayList<SocketInfo>();
399            for(int addressIndex =0; addressIndex < ssl.addresses.length; addressIndex++){
400                short sslport = ssl.addresses[addressIndex].port;
401                int ssl_port = Utility.shortToInt(sslport);
402                String host_name = ssl.addresses[addressIndex].host_name;
403
404                SocketInfo sInfo = IORToSocketInfoImpl.createSocketInfo(
405            "SecurityMechanismSelector2",
406                        type, host_name, ssl_port);
407                socketInfos.add(sInfo);
408            }
409            return socketInfos;
410        } else if (isSet(targetSupports, Integrity.value) ||
411                    isSet(targetSupports, Confidentiality.value) ||
412                    isSet(targetSupports, EstablishTrustInClient.value)) {
413            if (_logger.isLoggable(Level.FINE)) {
414                _logger.log(Level.FINE, "Target supports SSL");
415            }
416
417            if ( isSslRequired() ) {
418                if (_logger.isLoggable(Level.FINE)) {
419                    _logger.log(Level.FINE, "Client is configured to require SSL for the target");
420                }
421
422                ctx.setSSLUsed(true);
423                //SocketInfo[] socketInfos = new SocketInfo[ssl.addresses.size];
424                List<SocketInfo> socketInfos = new ArrayList<SocketInfo>();
425                for(int addressIndex =0; addressIndex < ssl.addresses.length; addressIndex++){
426                    short sslport = ssl.addresses[addressIndex].port;
427                    int ssl_port = Utility.shortToInt(sslport);
428                    String host_name = ssl.addresses[addressIndex].host_name;
429
430                    SocketInfo sInfo = IORToSocketInfoImpl.createSocketInfo(
431                        "SecurityMechanismSelector3",
432                        "SSL", host_name, ssl_port);
433                    socketInfos.add(sInfo);
434                }
435                return socketInfos;
436            } else {
437                return null;
438            }
439        } else if ( isSslRequired() ) {
440            throw new RuntimeException("SSL required by client but not supported by server.");
441        } else {
442            return null;
443        }
444        }
445
446
```

Expected return

This method will return a list of SocketInfo objects.

4.1.1 Naming conventions

The name of the method is appropriate, because suggests that the action will be a 'selection' and the caller is going to get a list of 'ports'.
Moreover, the conformation of name is adequate because is a verbs with the first letter of each addition word capitalized; and finally the word SSL that is an acronym for Secure Socket Layer is all capitalized as it should be.

LINE 355 - The variable "ssl" has not an adequate name: it's an acronym, but is used the generic SSL acronym. Because the SSL is recurrent in the code (and it could be used in so different ways) in every case is required a specific meaning for the variable's use.

LINE 392 - The variable "type" has not an adequate name: it is not acceptable because the meaning of 'type' is too generic (eg. type of what?) and the variable is not used in different points in the code.

LINE 401 / 427 - The variable "ssl_port" has not an adequate name: the developer has been forced to insert an underscore to differentiate this variable from the previous 'sslport' one, but, in any case, it should be done in other ways like 'ssl_port_int' for an integer and 'ssl_port_short' for a small int.

LINE 402 / 428 - The variable "host_name": if the use of this var is even outside of the if construct, it should be renamed as hostname, but in this case is used only for the

next method call, so it can be considered as a temporary variable and in this case it could be right.

4.1.2 Indention

LINE 356/439 - Inside the 'if' condition, there are 2 spaces at beginning and at ending. There are 2 line of thought for doing that, but according with the rest of the code, it seems a simple error.

LINE 386/387 - These two lines are the following of the Boolean clause of the previous if, but they are too indented.

LINE 389/394/395 - These two lines are too indented.

LINE 405 – the line "SecurityMecha…" is a carriage return for the previous code, but is not indented well at all.

LINE 440 to 443 – This line should indent more, besides, a tab has been used instead of some spaces.

4.1.3 Braces

The convention used for the parenthesis is called "Kernighan And Ritchie": the curly brace is opened just after the declared operation, on the same line. The closure will be done wrapping the line.

There is no violation of parenthesis convention inside the whole method, with the exception of the method declaration, in the:

LINE 344 - The parenthesis should be on the previous line according to the

"Kernighan And Ritchie" convention

4.1.4 File organization

LINE 419 – the total length of the line is 85, but in this case is acceptable because in other way the string content will be truncated.

LINE 385/410 – the total length is <80, and it is the only way to write this clause, because all on the same line it will be over 120 chars.

LINE 363/364 – these 2 lines must be concatenated: there is a cast of a returned variable, and it is to make the code more legible, even if there is an exceed of the 80 chars (in total the resulting line will be <120 chars)

LINE 425 – the total length of this line exceeds the 80 chars, but in this case is necessary because the 'for' clause. And in any case in under the 120 chars limit.

LINE 387/409/413/435/438 – After these lines, is not mandatory but it could be acceptable (in this case it will turn the code more legible), to insert a new line to split the outer 'if' and its inner complex code.

4.1.5 Wrapping lines

No violation detected.

Indention mistakes are listed in the 'Indention' section.

4.1.6 Comments

LINE 369/397/423 – there is a 'commented out' code, in this case is impossible to know if is better the commented code or the one in use.

So it will be better if the commented code is argued with few words, or in the best case, deleted.

LINE 355 – a comment is required, it will make the following code more easy to understand.

LINE 344 – the Javadoc is missing, the method is not described.
There isn't a description for the returned value and a description of the thrown exceptions.
To understand the whole method, a user has to go to deep in the code, so a little description of the method behaviour or for some of its sub routine is required.

4.1.7 Java source files

61. Each java source file contains a single public class or interface
62. The public class is the first class or interface in the file
63. Check that the external program interfaces are implemented consistently with what is described in the Javadoc
64. Check that the Javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you)

4.1.8 Package and import statements

All the imports are written at the beginning of the class.
So inside the method in object, there is no relevant stuff to be discussed.

4.1.9 Class and interface declarations

The class has not duplicate methods, and all the global vars are declared in the correct way.
Inside the method in object, there is no relevant stuff to be discussed.

4.1.10 Initialization and declarations

Inside the method in object, there is no relevant stuff to be discussed.

4.1.11 Method calls

LINE 352/353 – there are two method calls, and everyone is coded in a good way. In the first case there is a use of the parameter 'mechanism' that is a variable just set in the previous if clause. In the second case, is just passed to another class the input parameter of the method under study.

LINE 363 – that is an example of good programming, because after the return of the value from the 'getProfile()' class, there is a cast to check if effectively the returned value is of the correct type.

LINE 428 – in this case there is an assignation of a value (string). But in case of a mistaken type in the 'ssl' class there will be an error, because there isn't a try-catch clause nor a 'cast' of the returned value.

4.1.12 Arrays

There are no arrays in the method, and all the lists are well programmed.

4.1.13 Object comparison

LINE 360 – the comparison of the two objects should be done with the use of '.equals()'

4.1.14 Output format

The only output of this method is for the logger.

In any case there are no spelling errors, and they are all well written.

4.1.15 Computation, comparisons and assignments

The code is free of "brutish programming", and the only parts when it could be optimized are in LINE 385 and LINE 410, where in any case it seems to be the only way to do this check.

There are no division by zero (and in general no operation with integers).
There are no Boolean comparison operations.

4.1.16 Exceptions

LINE 404 and 430 – in case of an error in the method's execution, the entire thread will abort, so in these case it is better a try-catch surround.

LINE 349 – the try-catch construct is consistent and the thrown exception is well catch.

LINE 440 – there is a 'throw new' call, but in the LINE 344 there isn't a 'throws' statement.

4.1.17 Flow of control

All the loops are correctly formed, with initialization, increment, termination.
There are no switch-case statements, and all the if-else are correctly written.

4.1.18 Files

There is no file execution in this method.
So inside the method in object, there is no relevant stuff to be discussed.

4.2 Analysis of the method selectSecurityContext()

SecurityMechanismSelector.java

```
453    public SecurityContext selectSecurityContext(IOR ior)
454        throws InvalidIdentityTokenException,
455            InvalidMechanismException, SecurityMechanismException
456    {
457        SecurityContext context = null;
458    ConnectionContext cc = new ConnectionContext();
459        //print CSIv2 mechanism definition in IOR
460        if (traceIORs()) {
461            _logger.info("\nCSIv2 Mechanism List:" +
462                    getSecurityMechanismString(ctc,ior));
463    }
464
465        getSSLPort(ior, cc);
466        setClientConnectionContext(cc);
467
468        CompoundSecMech mechanism = cc.getMechanism();
469        if(mechanism == null) {
470            return null;
471        }
472        boolean sslUsed = cc.getSSLUsed();
473        boolean clientAuthOccurred = cc.getSSLClientAuthenticationOccurred();
474
475        // Standalone client
476        if (isNotServerOrACC()) {
477            context = getSecurityContextForAppClient(
478                    null, sslUsed, clientAuthOccurred, mechanism);
479            return context;
480        }
481
482        if (_logger.isLoggable(Level.FINE)) {
483            _logger.log(Level.FINE, "SSL used:" + sslUsed + " SSL Mutual auth:" + clientAuthOccurred);
484        }
485        ComponentInvocation ci = null;
486        /*// BEGIN IASRI# 4646060
487        ci = invMgr.getCurrentInvocation();
488        if (ci == null) {
489            // END IASRI# 4646060
490            return null;
491        }
492        Object obj = ci.getContainerContext();*/
493        if(isACC()) {
494            context = getSecurityContextForAppClient(ci, sslUsed, clientAuthOccurred, mechanism);
495        } else {
496            context = getSecurityContextForWebOrEJB(ci, sslUsed, clientAuthOccurred, mechanism);
497        }
498        return context;
499    }
500
```

Expected return

This method will

4.2.1 Naming conventions

The method's name has an appropriate meaning, because suggests that the action will be a "selection" and in fact, according to the Javadoc for the selected version of Glassfish "Select the security context to be used by the CSIV2 layer based on whether

the current component is an application client or a web/EJB component".
Besides, the conformation of name is adequate because is a verbs with the first letter of each addition word capitalized.

LINE 458 - The variable "cc" has not an adequate name: it's an acronym that doesn't explain its sense and this is not acceptable because the variable is not used in a temporary way, but in different point of the code.

LINE 485 - The variable "ci" has not an adequate name: it's an acronym that doesn't explain its sense and this is not acceptable because the variable is not used in a temporary way, but in different point of the code.

4.2.2 Indention

LINE 458 - This line should indent more, besides, a tab has been used instead of some spaces.

LINE 463 - The parenthesis of "if" should indent more, besides, a tab has been used instead of some spaces.

4.2.3 Braces

The convention used for the parenthesis is called "Kernighan And Ritchie": the curly brace is opened just after the declared operation, on the same line. The closure will be done wrapping the line.

LINE 456 - The parenthesis should be on the previous line according to the "Kernighan And Ritchie" convention

4.2.4 File organization

LINE 461 and 462 - The concatenation could stay on the same line, to not loose the logical sense of the operation. This modify is acceptable, because the result will be 77 characters on a single line (with a maximum value of 80 characters per line tolerated)

LINE 471 and 472 - A blank line should be left between the two part of code.

LINE 484 and 485 - A blank line should be left between the two part of code.

4.2.5 Wrapping lines

LINE 455 - There is a comma between the two exception, so there should be a wrapping line.

4.2.6 Comments

commenti scarni: non sempre c'è la javadoc – da fare

4.2.7 Java source file

4.2.8 Package and import statement

There is no relevant stuff to be discussed in this section.

4.2.9 Class and interface declaration

Da fare

4.2.10 Initialization and declaration

Da fare

4.2.11 Method calls

In this method other methods are called and is important to control that these calls have been done in the correct way.

LINE 462 - getSecurityMechanismString(ctc,ior)

This method must return a String and must have, in order, two parameters of type CSIV2TaggedComponentInfo and IOR.

The return is done calling the method getSecurityMechanismString(tCI, mechList, typeId), that returns a String.

The input parameters "ctc" is of type CSIV2TaggedComponentInfo and "ior" is of type IOR.

LINE 465 - getSSLPort(ior, cc);

This method has been analysed in the previous chapter.

LINE 466 - setClientConnectionContext(cc)

This is a void method, besides it isn't a return statement.

The input parameter "cc" is of type ConnectionContext.

LINE 468 - cc.getMechanism() deve restituire una CompoundSecMech

This method must return a variable of type CompoundSecMech, in fact, according to the javadoc

```
/**
* Return the selected compound security mechanism.
*/
```

returns "mechanism" that is a private variable of type CompoundSecMech.

This check has been done analyzing the method in the class ConnectionContext.

LINE 472 - cc.getSSLUsed()

This method must return a variable of type Boolean, in fact, according to the javadoc

```
/**
* Return true if SSL was used to invoke the EJB.
*/
```

returns "ssl" that is a private variable of type Boolean.

This check has been done analyzing the method in the class ConnectionContext.

LINE 472 - cc.getSSLClientAuthenticationOccurred()

This method must return a variable of type Boolean, in fact, according to the javadoc

```
/**
* Return true if SSL client authentication has happened, false otherwise.
*/
```

returns "sslClientAuth" that is a private variable of type Boolean.

This check has been done analyzing the method in the class ConnectionContext.

LINE 477 - getSecurityContextForAppClient(null, sslUsed, clientAuthOccurred, mechanism)

This method must return a SecurityContext and must have, in order, four parameters of type ComponentInvocation, boolean, boolean and CompoundSecMech.

The return is done calling the method SendUsernameAndPassword that returns a variable of type SecurityContext.

The parameter "ci", defined as null, is not of the required type, but in this case it doesn't cause problems:

the parameter is passed to the method "sendUsernameAndPassword", that returns the variable "cxt" given by the the method getUsernameAndPassword. This last method has "ci" as input parameter, but in the code this variable is not used, in fact the lines of code that concern "ci" have been commented. The other parameters are of the correct type, in fact "sslUsed" is of type boolean, "clientAuthOccurred" is of type Boolean and "mechanism" is of type CompoundSecMech.

In this context the method is correctly used and it is clear looking at the Javadoc:

```
/**
* Create the security context to be used by the CSIV2 layer
* to marshal in the service context of the IIOP message from an appclient
* or standalone client.
* @return the security context.
*/
```

LINE 494 - getSecurityContextForAppClient(ci, sslUsed, clientAuthOccurred, mechanism)

In this invocation, the parameter "ci" is of the required type (ComponentInvocation), so in this case it shouldn't cause problems.

LINE 496 - getSecurityContextForWebOrEJB(ci, sslUsed, clientAuthOccurred, mechanism)

This method must return a SecurityContext and must have, in order, four parameters of type ComponentInvocation, boolean, boolean and CompoundSecMech.

The return is "cxt", a variable defined calling the method propagateIdentity that returns a variable of type SecurityContext.

The input parameters "ci" is of ComponentInvocation, "sslUsed" is of type boolean, "clientAuthOccurred" is of type boolean and "mechanism" is of type

CompoundSecMech.

In this context the method is correctly used and it is clear looking at the Javadoc:

```
/**
* Create the security context to be used by the CSIV2 layer
* to marshal in the service context of the IIOP message from an web
* component or EJB invoking another EJB.
* @return the security context.
*/
```

4.2.12 Arrays

There is no relevant stuff to be discussed in this section.

4.2.13 Object comparison

4.2.14 Output Format

There is no relevant stuff to be discussed in this section.

4.2.15 Computation, Comparisons and assignments

4.2.16 Exceptions

There is no relevant stuff to be discussed in this section.

4.2.17 Flow of control

There is no relevant stuff to be discussed in this section.

4.2.18 Files

There is no relevant stuff to be discussed in this section.

4.3 Analysis of the method sendUsernameAndPassword()

SecurityMechanismSelector.java

```
582    private SecurityContext sendUsernameAndPassword(ComponentInvocation ci,
583                            boolean sslUsed,
584                            boolean clientAuthOccurred,
585                                                    CompoundSecMech mechanism)
586                throws SecurityMechanismException {
587        SecurityContext ctx = null;
588        if(mechanism == null) {
589            return null;
590        }
591        AS_ContextSec asContext = mechanism.as_context_mech;
592        if( isSet(asContext.target_requires, EstablishTrustInClient.value)
593            || ( isSet(mechanism.target_requires, EstablishTrustInClient.value)
594         && !clientAuthOccurred ) ) {
595
596            ctx = getUsernameAndPassword(ci, mechanism);
597
598            if (_logger.isLoggable(Level.FINE)) {
599                _logger.log(Level.FINE, "Sending Username/Password");
600            }
601        } else {
602            return null;
603        }
604        return ctx;
605    }
606
```

Expected return

This method will

4.3.1 Naming conventions

The method's name has an appropriate meaning, because suggests that the action will be a " sending of data" and in fact, according to the Javadoc for the selected version of Glassfish

“Get the security context to send username and password in the service context.

@param whether username/password will be sent over plain IIOP or over IIOP/SSL.

@return the security context.

@exception SecurityMechanismException if there was an error”.

Besides, the conformation of name is adequate because is a verbs with the first letter

of each addition word capitalized.

4.3.2 Indention

4.3.3 Braces

4.3.4 File organization

4.3.5 Wrapping lines

4.3.6 Comments

4.3.7 Java source file

4.3.8 Package and import statement

4.3.9 Class and interface declaration

4.3.10 Initialization and declaration

4.3.11 Method calls

4.3.12 Arrays

4.3.13 Object comparision

4.3.14 Output format

4.3.15 Computation, Comparisons and assignments

4.3.16 Exceptions

4.3.17 Flow of control

4.3.18 Files

5. Any other problem you have highlighted

SARA

LEO

6. Additional Material

7. Hours of works

Here is the time spent for redact this document:

[sum of hours spent by team's members]

+3h (22/12)

+8h (27/12)

+8h (28/12)

+2h (30/12)

+5h (02/01)

+h (03/01)

+h (/01)

+h (/01)

TOTAL ~ 112 hours

- Leonardo Turchi: ~ 56hours
- Sara Pisani: ~ 56hours