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 1.0

Table of contents

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*

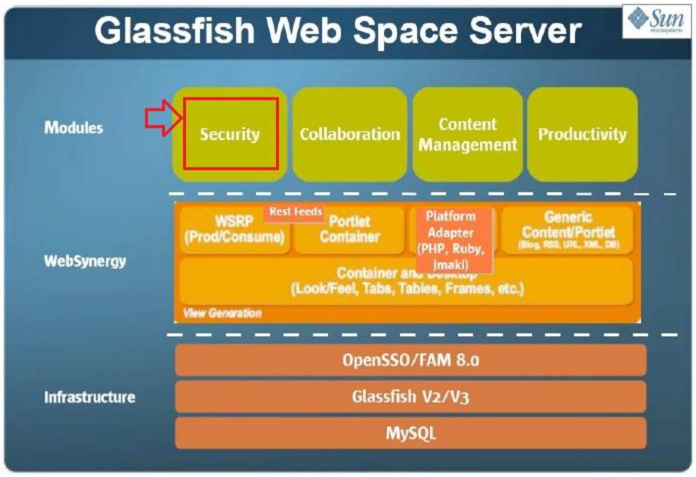
The class is subject to the terms of the *GPL Version 2 Licence*.

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.

The class is a member of the 'Security' Module of the Glassfish WebSpaceServer.



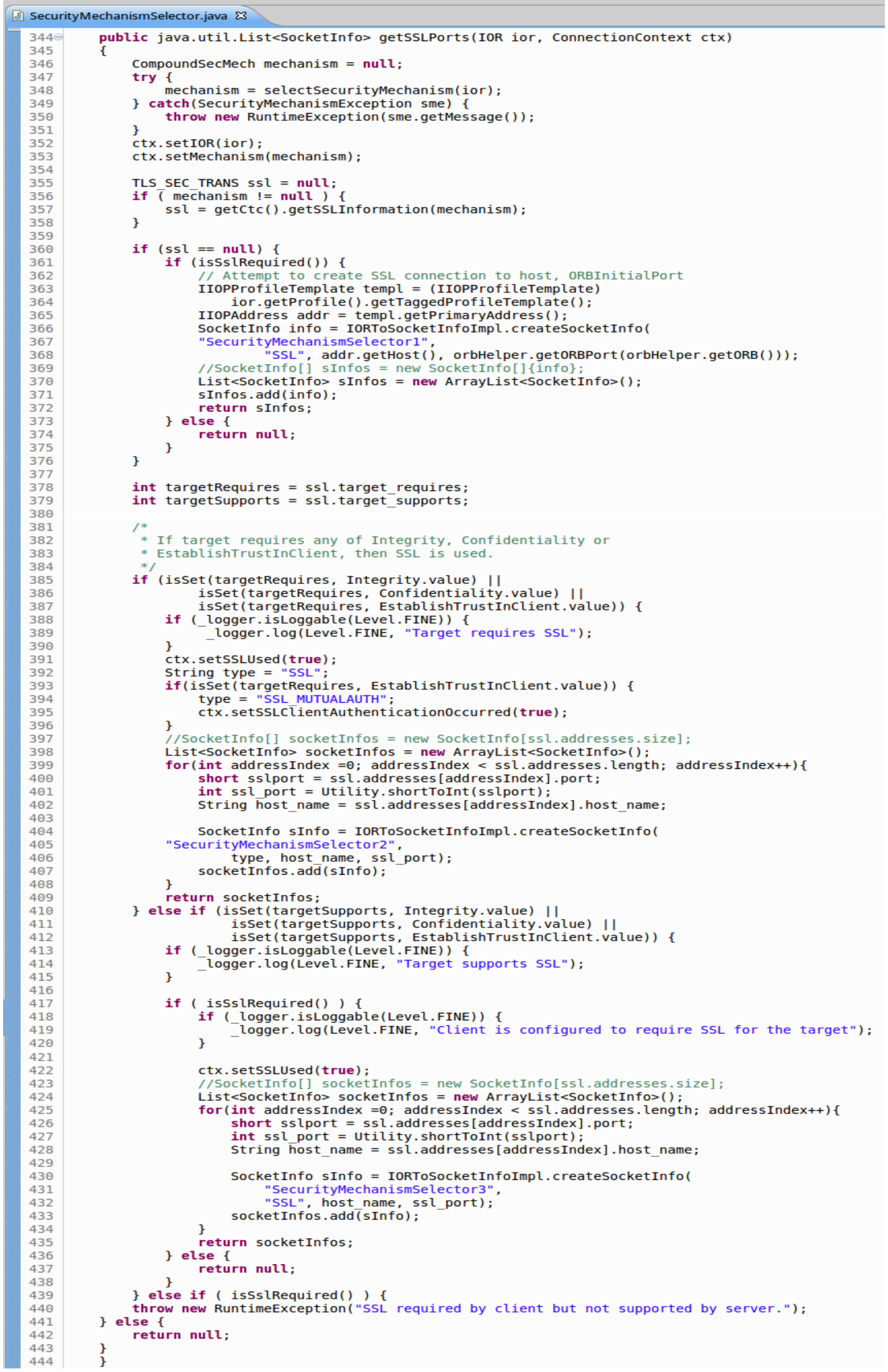
Some example of operation concerning security, that the class can do with the help of methods, are listed below (this information can be found in the Javadoc):

- ✓ To Read the client and server **preferences** from the *config* files.
- ✓ To determine if **SSL** should be used to connect to the target based on client and target policies.
- ✓ To **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.
- ✓ To 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.
- ✓ To create the security context to be used by the CSIV2 layer to marshal in the service context of the IIOP message from a web component or EJB invoking another EJB.

- ✓ To get the security context to send username and password in the service context.
- ✓ To get the security context to **propagate** principal/distinguished name in the service context.
- ✓ To return whether the server is **trusted** or not based on configuration information.
- ✓ To **check** if a given domain is trusted.
- ✓ To get the username and password either from the JAAS subject or from thread local storage.
- ✓ To **lookup** the real name that is required by the server and set it up in the *PasswordCredential* class.
- ✓ To get the principal/distinguished name from thread local storage.
- ✓ Select the security mechanism from the list of compound security mechanisms.
- ✓ To evaluate a client's conformance to the **security policies** configured on the target.
- ✓ To interpret client credentials after **validation**.

4. List of issues found by applying the checklist

4.1 Analysis of the method getSSLPorts()



```
public java.util.List<SocketInfo> getSSLPorts(IOR ior, ConnectionContext ctx)
{
    CompoundSecMech mechanism = null;
    try {
        mechanism = selectSecurityMechanism(ior);
    } catch(SecurityMechanismException sme) {
        throw new RuntimeException(sme.getMessage());
    }
    ctx.setIOR(ior);
    ctx.setMechanism(mechanism);

    TLS_SEC_TRANS ssl = null;
    if ( mechanism != null ) {
        ssl = getCtc().getSSLInformation(mechanism);
    }

    if (ssl == null) {
        if (isSslRequired()) {
            // Attempt to create SSL connection to host, ORBInitialPort
            IIOPProfileTemplate templ = (IIOPProfileTemplate)
                ior.getProfile().getTaggedProfileTemplate();
            IIOPAddress addr = templ.getPrimaryAddress();
            SocketInfo info = IORToSocketInfoImpl.createSocketInfo(
            "SecurityMechanismSelector1",
                    "SSL", addr.getHost(), orbHelper.getORBPort(orbHelper.getORB()));
            //SocketInfo[] sInfos = new SocketInfo[]{info};
            List<SocketInfo> sInfos = new ArrayList<SocketInfo>();
            sInfos.add(info);
            return sInfos;
        } else {
            return null;
        }
    }

    int targetRequires = ssl.target_requires;
    int targetSupports = ssl.target_supports;

    /*
     * If target requires any of Integrity, Confidentiality or
     * EstablishTrustInClient, then SSL is used.
     */
    if (isSet(targetRequires, Integrity.value) ||
            isSet(targetRequires, Confidentiality.value) ||
            isSet(targetRequires, EstablishTrustInClient.value)) {
        if (_logger.isLoggable(Level.FINE)) {
             _logger.log(Level.FINE, "Target requires SSL");
        }
        ctx.setSSLUsed(true);
        String type = "SSL";
        if(isSet(targetRequires, EstablishTrustInClient.value)) {
            type = "SSL_MUTUALAUTH";
            ctx.setSSLClientAuthenticationOccurred(true);
        }
        //SocketInfo[] socketInfos = new SocketInfo[ssl.addresses.size];
        List<SocketInfo> socketInfos = new ArrayList<SocketInfo>();
        for(int addressIndex =0; addressIndex < ssl.addresses.length; addressIndex++){
            short sslport = ssl.addresses[addressIndex].port;
            int ssl_port = Utility.shortToInt(sslport);
            String host_name = ssl.addresses[addressIndex].host_name;

            SocketInfo sInfo = IORToSocketInfoImpl.createSocketInfo(
        "SecurityMechanismSelector2",
                type, host_name, ssl_port);
            socketInfos.add(sInfo);
        }
        return socketInfos;
    } else if (isSet(targetSupports, Integrity.value) ||
                isSet(targetSupports, Confidentiality.value) ||
                isSet(targetSupports, EstablishTrustInClient.value)) {
        if (_logger.isLoggable(Level.FINE)) {
            _logger.log(Level.FINE, "Target supports SSL");
        }

        if ( isSslRequired() ) {
            if (_logger.isLoggable(Level.FINE)) {
                _logger.log(Level.FINE, "Client is configured to require SSL for the target");
            }

            ctx.setSSLUsed(true);
            //SocketInfo[] socketInfos = new SocketInfo[ssl.addresses.size];
            List<SocketInfo> socketInfos = new ArrayList<SocketInfo>();
            for(int addressIndex =0; addressIndex < ssl.addresses.length; addressIndex++){
                short sslport = ssl.addresses[addressIndex].port;
                int ssl_port = Utility.shortToInt(sslport);
                String host_name = ssl.addresses[addressIndex].host_name;

                SocketInfo sInfo = IORToSocketInfoImpl.createSocketInfo(
                    "SecurityMechanismSelector3",
                    "SSL", host_name, ssl_port);
                socketInfos.add(sInfo);
            }
            return socketInfos;
        } else {
            return null;
        }
    } else if ( isSslRequired() ) {
    throw new RuntimeException("SSL required by client but not supported by server.");
} else {
    return null;
}
}
```

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

The file in object contains a single public class, and it is the first class in the file.

The Javadoc is incomplete or missing, the class is not described properly.
For example, there is no reference to the return behaviour of many methods.
Many public methods (not only assigned ones) don't have any documentation, as well as the classes and interfaces.

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.

LINE 344 – There is a call for a list declaration, but it is redundant because the java.util.list class were already imported at the beginning of the file.

4.1.9 Class and interface declarations

The class has not duplicate methods, and all the global variables 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 is done in the correct way. But for other cases (in which there is no 'null' comparison) 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.

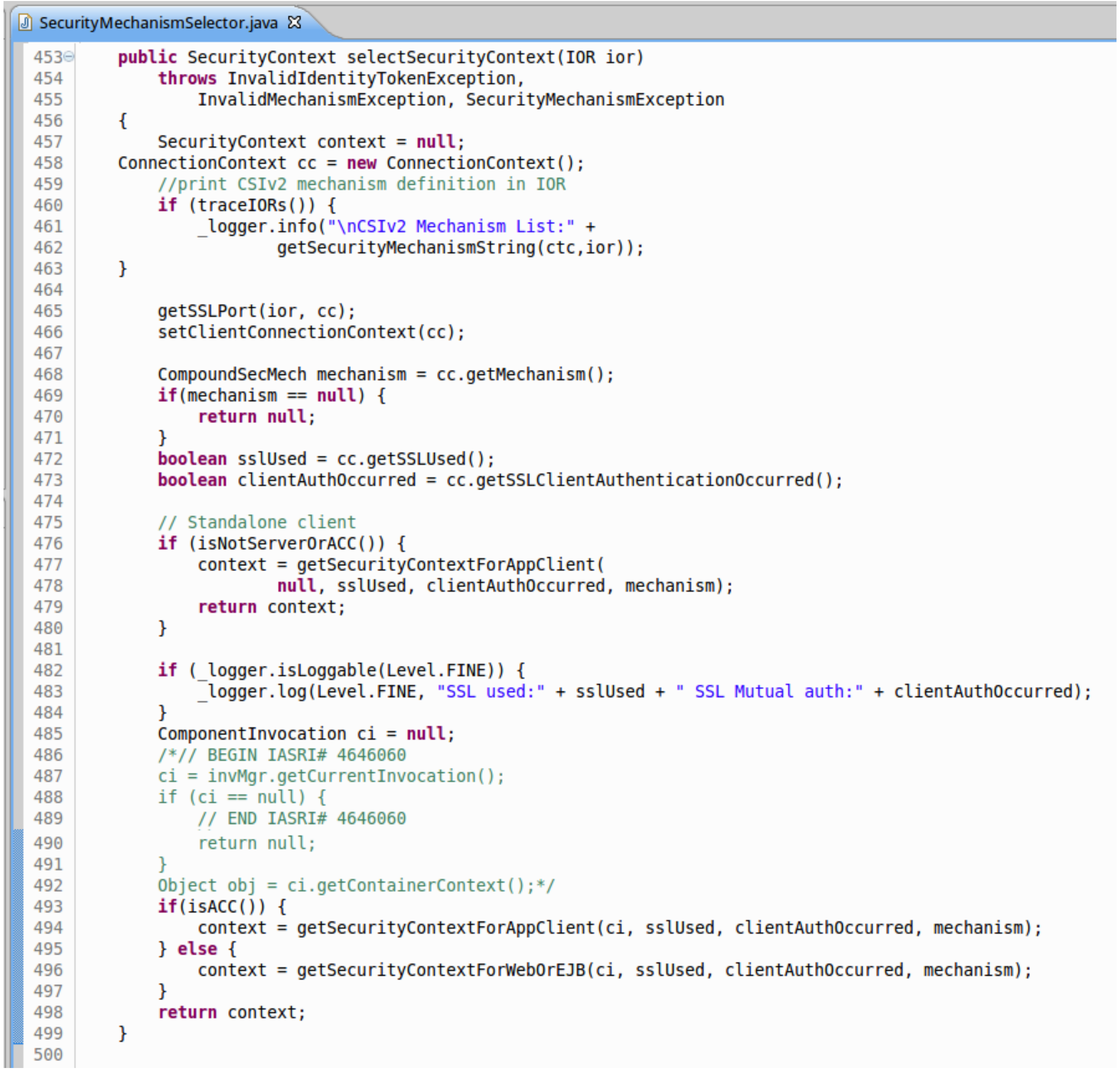
4.1.17 Flow of control

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()



```
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 return a public SecurityContext object.

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

LINE 459 and 486/492 – 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 458 and 485 – Since the variables' name has not a clear meaning, could be better have a comment to make the following code more easy to understand.

4.2.7 Java source file

The file in object contains a single public class, and it is the first class in the 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

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

4.2.10 Initialization and declaration

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

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 return 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

LINE 469 – The only case in which the operator "==" is used is in this line, but with a "null" parameter is acceptable.

4.2.14 Output Format

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

4.2.15 Computation, Comparisons and assignments

There are not operations between integer variables, cast or implicit conversions.

4.2.16 Exceptions

There are no try-catch constructs and the method could throws not handled exceptions, as we can see at the beginning of the method (‘throws’ clause).

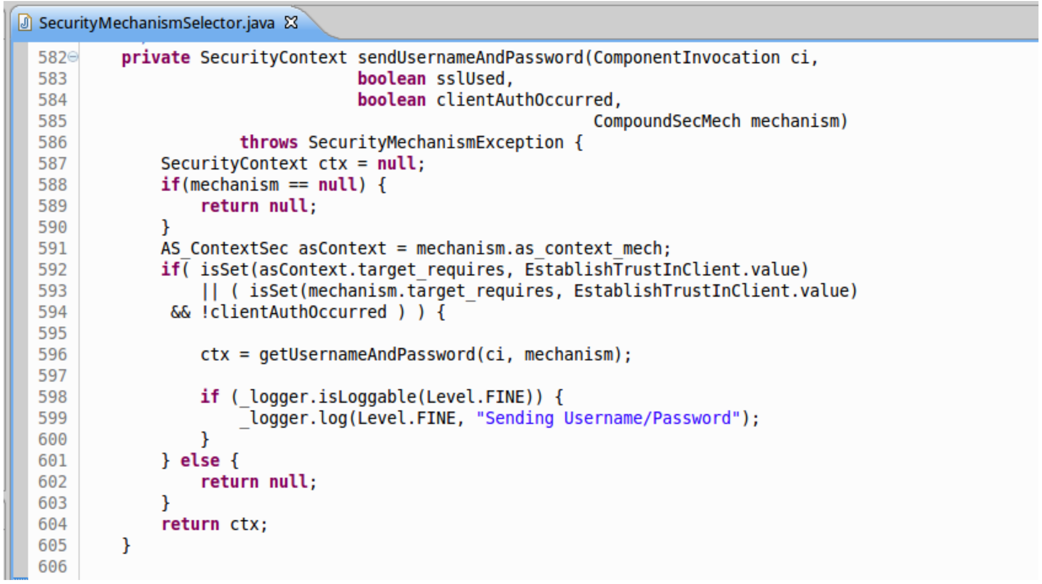
4.2.17 Flow of control

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

4.2.18 Files

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

4.3 Analysis of the method sendUsernameAndPassword()



```
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 return a private SecurityContext object.

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.

LINE 587 - The variable "cxt" 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.

4.3.2 Indention

LINE 583 and 584 - A tab has been used instead of some spaces.

LINE 585 – 24 spaces must be eliminated in order to align the code.
LINE 586 – 12 spaces must be added in order to align the code.

LINE 594 - This line should indent more, besides, a tab has been used instead of some spaces.

LINE 603 – The parenthesis should indent more to be aligned with the “else”.

4.3.3 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.

4.3.4 File organization

LINE 590 and 591 - A blank line should be left between the two part of code.

LINE 595 - The blank line is not required.

LINE 597 - The blank line is not required.

LINE 603 and 604 - A blank line should be left between the two part of code.

4.3.5 Wrapping lines

LINE 592 and 593 - The “||” (OR operator) should stay on the first line, because a line break is required after an operator. This modify is acceptable, because the maximum value of 80 characters per line is respected.

LINE 593 and 594 - The “&&” (AND operator) should stay on the first line, because a line break is required after an operator. This modify is acceptable, because the maximum value of 80 characters per line is respected.

4.3.6 Comments

LINE 587 – Since the variables' name has not a clear meaning, could be better have a comment to make the following code more easy to understand.

4.3.7 Java source file

The file in object contains a single public class, and it is the first class in the file.

4.3.8 Package and import statement

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

4.3.9 Class and interface declaration

The class has not duplicate methods, and all the global variables are declared in the correct way.

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

4.3.10 Initialization and declaration

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

4.3.11 Method calls

LINE 591 - “mechanism.as_context_mech” must be of type AS_ContextSec to satisfy the line of code AS_ContextSec asContext = mechanism.as_context_mech.
This condition is respected and it is verifiable analyzing the class CompoundSecMech (com.sun.corba.ee.org.omg.CSIIOP.CompoundSecMech)

LINE 592 - isSet(asContext.target_requires, EstablishTrustInClient.value)
This method must return a boolean and must have two parameters of type int as input.
The input parameters “asContext.target_requires” is of type public short and EstablishTrustInClient.value is of type static final short.
In this specific case it does not cause problems: Is guaranteed that the ranges of “short int are at least -32767 .. +32767, and the range of short int is a subset of the range of int.

LINE 592 - isSet(mechanism.target_requires, EstablishTrustInClient.value)
This method must return a boolean and must have two parameters of type int as input.
The input parameters “mechanism.target_requires” is of type short and “EstablishTrustInClient.value” is of type static final short.
As said before, not even in this case the substitution of int with short cause problems.

LINE 596 - getUsernameAndPassword(ci, mechanism);
This method must return a SecurityContext and must have, in order, two parameters of type ComponentInvocation and CompoundSecMech.
The variables “ci” and “mechanism” are the input parameter of the method sendUsernameAndPassword, defined as ComponentInvocation and CompoundSecMech “mechanism” CompoundSecMech.
The return value is a SecurityContext, so satisfies the two lines of code SecurityContext ctx = null; and
ctx = getUsernameAndPassword(ci, mechanism);

LINE 598 - _logger.isLoggable(Level.FINE))

The method isLoggable returns a boolean, so allow to execute the code after the "if" (if the value is true) or after the "else" (if the value is false).

LINE 599 - _logger.log(Level.FINE, "Sending Username/Password")
This is a void method that must have, in order, two parameters of type Level and String. The input parameter "Level.FINE" is of type Level: this condition is verifiable analyzing the class Level (Resource – java.util.logging.Level – Eclipse Platform). The other parameter ""Sending Username/Password"" is a String.

4.3.12 Arrays

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

4.3.13 Object comparison

LINE 588 – The only case in which the operator "==" is used is in this line, but with a "null" parameter is acceptable.

4.3.14 Output format

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

4.3.15 Computation, Comparisons and assignments

LINE 592 and 593 – during these calls of method isSet, an implicit convertion is done: input parameters are "short int" but the method requires two "int".

4.3.16 Exceptions

There are not try-catch constructs and the method could throw not handled exceptions.

4.3.17 Flow of control

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

4.3.18 Files

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

5. Additional Material

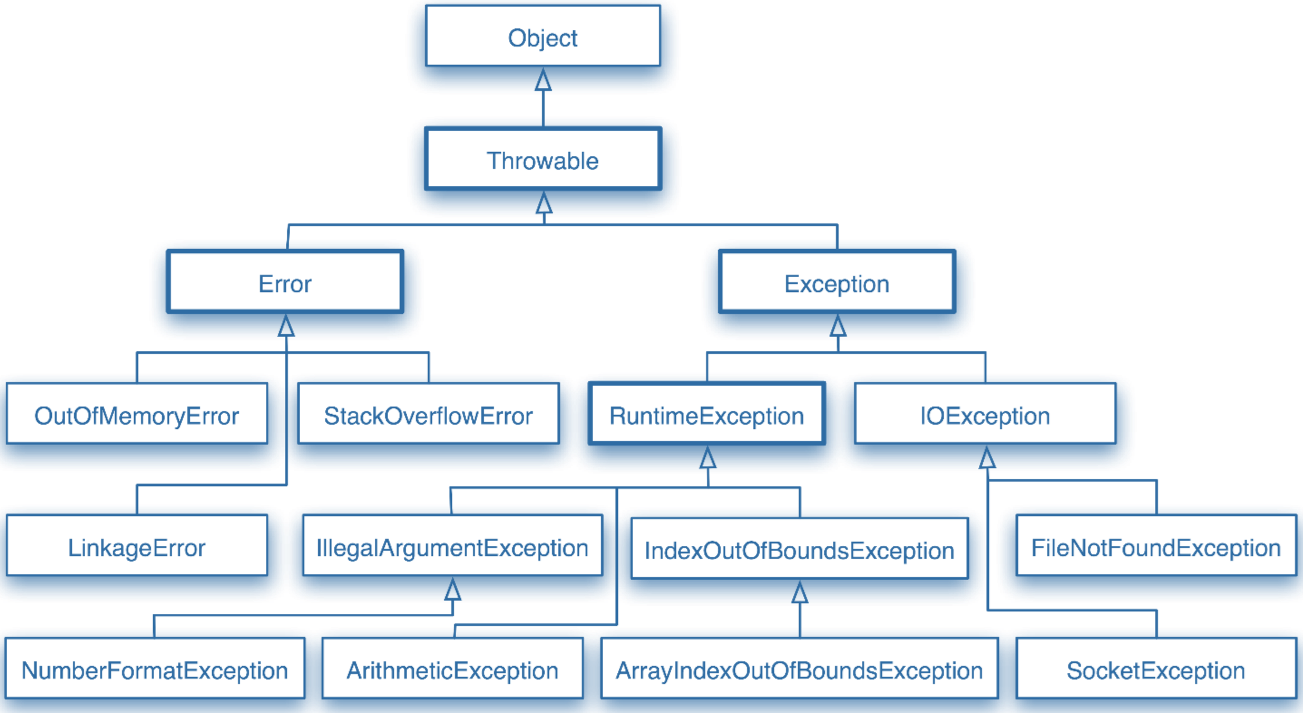
- URL to find the classes/methods to inspect
  http://assignment.pompel.me/

- URL to find the Javadoc for the selected version of Glassfish
  http://glassfish.pompel.me

- URL to download the virtual machine
  http://www.dropbox.com/s/3nwjyd6v0b8360j/box.ova?dl=0

- URL to find a small guide to avoid brutish programming
  http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html

- The exception class hierarchy (for Java), to keep track of all exception scenarios:



6. Software and tools

- Microsoft Word: to redact this document.
- GitHub: to share the material of this project.
- VirtualBox: to open virtual machine.
- Eclipse Luna: to open Glassfish’s code.
- Java/Glassfish Online Documentation.

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)
+8h (03/01)
+6h (04/01)

TOTAL ~ 40 hours

- Leonardo Turchi: ~ 20 hours
- Sara Pisani: ~ 20 hours