POLITECNICO MILANO 1863

CODE INSPECTION DOCUMENT

Paolo Antonini 858242
Andrea Corneo 849793

GlassFish 4.1

version 1.1 – 5th February 2016

Contents

1 Introduction

1.1 Purpose and scope

Code inspection is the systematic examination of computer source code, in order to improve the overall quality of software. We are to apply code inspection techniques to evaluate the general quality of selected code extracts from a release of the GlassFish 4.1 application server.

We are going to analyse a portion of `SecurityMechanismSelector` class, from `com.sun.enterprise.iiop.security` package.

1.2 References

We are reviewing GlassFish source code, version 4.1.1, revision 64219[1]. The code under analysis is typeset right in the document.

As a reference, we quote the code inspection checklist in appendix A.

[1] This is the link to checkout the whole code: `https://svn.java.net/svn/glassfish~svn/tags/4.1.1@64219`

1.3 Overview of the document

This document develops as follows. In chapter 2 we provide some general information about the class we are assigned and its functional role, to better understand the context we are moving in. Chapter 3 represents the core of the document, because the thorough analysis of the methods is detailed there.

Appendix A contains the whole code inspection checklist, as a support.

2
Class presentation and functional analysis

`SecurityMechanismSelector` class is part of the security module on the server side of GlassFish. In particular, the class belongs to `com.sun.enterprise.iiop.security` package, which provides security infrastructure and technology integration to Enterprise JavaBeans[1].

[1] We recall that an enterprise bean is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfils the purpose of the application.

We understand that the objective of the class is to select the appropriate security information to be sent in the IIOP message to the client. IIOP stands for Internet Inter-ORB Protocol, which is what makes it possible for distributed programs written in different programming languages to communicate over the Internet.

Our understanding of the document is confirmed by the Javadoc description of the class, quoted below:

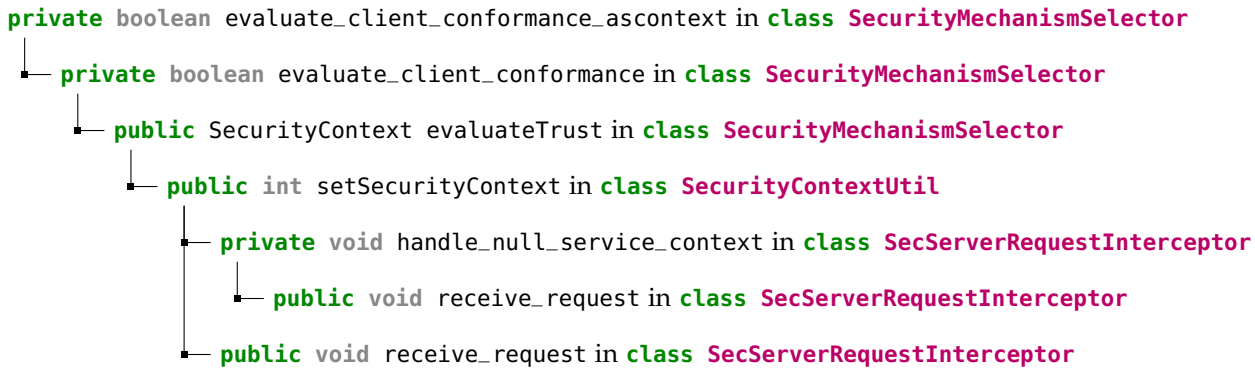
```
108 /**
109  * This class is responsible for making various decisions for selecting
110  * security information to be sent in the IIOP message based on target
111  * configuration and client policies.
112  * Note: This class can be called concurrently by multiple client threads.
113  * However, none of its methods need to be synchronized because the methods
114  * either do not modify state or are idempotent.
115  *
116  * @author Nithya Subramanian
117  
118  */
119 
120 @Service
121 @Singleton
122 public final class SecurityMechanismSelector implements PostConstruct {
```

In particular, we are assigned the following methods from the class to analyse:

1. `evaluate_client_conformance_ascontext`;
2. `evaluate_client_conformance_sascontext`;
3. `evaluate_client_conformance`.

We are going to describe them in detail later (chapter 3), but for now it is useful to present the call hierarchy of the first one, generated through NetBeans IDE. The call hierarchy tree for the

second method is exactly the same as the one presented, save for the root, obviously; as of the third method, it is the only caller of the first two). So, here follows the call hierarchy tree for `evaluate_client_conformance_ascontext`:



By reading the documentation of the methods in the tree we understand that the class under analysis offers some methods to select the appropriate security context for the server, based on target configuration and client policies, and returns them to `setSecurityContext`, which authenticates the client. The result of the authentication process is passed to the server-side request interceptor, and in particular to `receive_request` method. A request interceptor is, roughly, a software element designed to transfer context information between clients and servers.

3
Analysis of the methods

In this chapter we are going to check the compliance to the checklist (see appendix A for reference) of the three methods we are assigned.

Each section of the chapter goes through a specific method, whose code we show entirely, for the sake of completeness. In order to improve readability, the code of the method has been split into fragments, which are followed by a report of the issues we spotted.

Tabs are shown, highlighted by a ⇥ symbol.

3.1 `evaluate_client_conformance_ascontext`

```
1204    /* Evaluates a client's conformance to a security policies
1205     * at the client authentication layer.
1206     *
1207     * returns true if conformant ; else returns false
1208     */
```

3.1.1 Issues

```
1209    private boolean evaluate_client_conformance_ascontext(
1210                        SecurityContext ctx,
1211                        EjbIORConfigurationDescriptor iordesc,
1212                        String realmName)
1213    {
1214
1215        boolean client_authenticated = false;
```

5 the method name `evaluate_client_conformance_ascontext` (line 1209) does not comply with the rule, since it contains underscore characters; given that other methods in the class follow the same pattern (lowercase words, separated by underscores), maybe this is done intentionally to improve readability.

6 the variable name `client_authenticated` (line 1215) does not comply with the rule, as it contains an underscore.

10 the only inconsistency in the bracing style is in line 1213, since elsewhere in the method K&R[1] style is consistently adopted (the opening brace should be placed at the end of the method declaration).

[1] K&R is the well-known shorthand for "Kernighan and Ritchie".

```
1217        // get requirements and supports at the client authentication layer
1218        AS_ContextSec ascontext = null;
1219        try {
1220            ascontext = this.getCtc().createASContextSec(iordesc, realmName);
1221        } catch (Exception e) {
1222            _logger.log(Level.SEVERE, "iiop.createcontextsec_exception",e);
1223
1224            return false;
1225        }
```

52 the `try-catch` group in lines 1219 to 1225 is roughly managed: catching a generic `Exception` does not allow a detailed log of the error.

```
1228        /**************************************************************************
1229         * Conformance Matrix:
1230         *
1231         * |------------|---------------------|---------------------|------------|
1232         * | ClientAuth | targetrequires.ETIC | targetSupports.ETIC | Conformant |
1233         * |------------|---------------------|---------------------|------------|
1234         * |     Yes    |          0          |      1              |    Yes     |
1235         * |     Yes    |          0          |      0              |    No      |
1236         * |     Yes    |          1          |      X              |    Yes     |
1237         * |     No     |          0          |      X              |    Yes     |
1238         * |     No     |          1          |      X              |    No      |
1239         * |------------|---------------------|---------------------|------------|
1240         *
1241         * Abbreviations: ETIC - EstablishTrusInClient
1242         *
1243         **************************************************************************/
1244
1245        if  ( (ctx != null) && (ctx.authcls != null) &&  (ctx.subject != null))
1246            client_authenticated = true;
1247        else
1248            client_authenticated = false;
```

11 single line statements in the `if-else` block (lines 1245 to 1248) should be surrounded by curly braces.

44 to avoid an example of "brutish programming", lines 1245 to 1248, together with line 1215, can be collapsed to the following statement:

```
boolean client_authenticated = (ctx != null) && (ctx.authcls != null) && (ctx.subject != null);
```

However, this solution is not trouble-free (e.g., it exceeds the 80 character limit stated in rule 13).

```
1250        if (client_authenticated) {
1251            if ( ! ( isSet(ascontext.target_requires, EstablishTrustInClient.value)
1252                     || isSet(ascontext.target_supports, EstablishTrustInClient.value))){
1253            return false; // non conforming client
1254            }
1255            // match the target_name from client with the target_name in policy
1256
1257            byte [] client_tgtname = getTargetName(ctx.subject);
1258
```

```
1259            if (ascontext.target_name.length != client_tgtname.length){
1260                return false; // mechanism did not match.
1261            }
1262            for (int i=0; i < ascontext.target_name.length ; i ++)
1263                if (ascontext.target_name[i] != client_tgtname[i]){
1264                    return false; // mechanism did not match
1265                }
1266        } else {
1267            if ( isSet(ascontext.target_requires, EstablishTrustInClient.value)){
1268                return false;  // no mechanism match.
1269            }
1270        }
1271        return true;
1272    }
```

6 the `client_tgtname` variable (line 1257) does not comply with standard naming rules, because it contains an underscore.

11 the **`if`** statement in the **`for`** group (lines 1262 to 1265) should be surrounded with braces.

13 a limited number of lines in this fragment exceeds significantly the 80 character per line limit (most notably, line 1252); however, when it comes to nested **`if`** clauses and to method calls inside the **`if`** conditions, it may be difficult to comply with this limit.

15 the condition of the **`if`** statement in lines 1251 to 1252 breaks before `||` operator, instead of after the operator itself, which is preferable.

3.1.2 *Suggestions*

To improve readability, we suggest to include the **`for`** block in lines 1262 to 1265 within an **`else`** clause (obviously, thanks to the use of **`return`** statement in line 1264, the behaviour of the method does not change).

3.2 `evaluate_client_conformance_sascontext`

```
1274    /* Evaluates a client's conformance to a security policy
1275     * at the sas context layer. The security policy
1276     * is derived from the EjbIORConfigurationDescriptor.
1277     *
1278     * returns true if conformant ; else returns false
1279     */
```

3.2.1 *Issues*

```
1280    private boolean evaluate_client_conformance_sascontext(
1281                        SecurityContext ctx,
1282                        EjbIORConfigurationDescriptor iordesc)
1283    {
1284
1285       boolean caller_propagated = false;
```

5 the method name `evaluate_client_conformance_sascontext` (line 1280) does not comply with the rule, due to the underscore characters inside.

6 the variable name `caller_propagated` (line 1285) does not comply with the rule, because it contains an underscore as separator.

10 the only inconsistency in the bracing style is in line 1283, since elsewhere in the method K&R style is consistently adopted.

```
1287          // get requirements and supports at the sas context layer
1288        SAS_ContextSec sascontext = null;
1289        try {
1290            sascontext = this.getCtc().createSASContextSec(iordesc);
1291        } catch (Exception e) {
1292            _logger.log(Level.SEVERE,"iiop.createcontextsec_exception",e);
1293            return false;
1294        }
```

52 the `try-catch` group in lines 1289 to 1294 is roughly managed: catching a generic `Exception` does not allow a detailed log of the error.

```
1297        if  ( (ctx != null) && (ctx.identcls != null) &&  (ctx.subject != null))
1298            caller_propagated = true;
1299        else
1300            caller_propagated = false;
```

11 single line statements in the `if-else` block (lines 1297 to 1300) should be surrounded by curly braces.

44 to avoid an example of "brutish programming", lines 1297 to 1300, together with line 1285, can be collapsed to the following statement:

```
boolean caller_propagated = (ctx != null) && (ctx.identcls != null) && (ctx.subject != null);
```

However this line is too long (it exceeds the 80 character limit stated in rule 13).

```
1302        if (caller_propagated) {
1303            if ( ! isSet(sascontext.target_supports, IdentityAssertion.value))
1304                return false; // target does not support IdentityAssertion
1305
1306            /* There is no need further checking here since SecServerRequestInterceptor
1307             * code filters out the following:
1308             * a. IdentityAssertions of types other than those required by level 0
1309             *    (for e.g. IdentityExtension)
1310             * b. unsupported identity types.
1311             *
1312             * The checks are done in SecServerRequestInterceptor rather than here
1313             * to minimize code changes.
1314             */
1315            return true;
1316        }
1317        return true; //  either caller was not propagated or mechanism matched.
1318    }
```

11 single line statement in the `if` block (lines 1303 to 1304) should be surrounded by curly braces.

3.3 evaluate_client_conformance

```
1322     /**
1323      * Evaluates a client's conformance to the security policies configured
1324      * on the target.
1325      * Returns true if conformant to the security policies
1326      * otherwise return false.
1327      *
1328      * Conformance checking is done as follows:
1329      * First, the object_id is mapped to the set of EjbIORConfigurationDescriptor.
1330      * Each EjbIORConfigurationDescriptor corresponds to a single CompoundSecMechanism
1331      * of the CSIv2 spec. A client is considered to be conformant if a
1332 CompoundSecMechanism
1333      * consistent with the client's actions is found i.e. transport_mech,
1334 as_context_mech
1335      * and sas_context_mech must all be consistent.
1336      *
1337      */
```

3.3.1 *Issues*

```
1338     private boolean evaluate_client_conformance(SecurityContext   ctx,
1339                                                 byte[]            object_id,
1340                                                 boolean           ssl_used,
1341                                                 X509Certificate[] certchain)
1342     {
```

5 the method name `evaluate_client_conformance` (line 1338) does not comply with the rule, owing to the underscore characters within.

6 the names of the two parameters `object_id` (line 1339) and `ssl_used` (line 1340) contain underscores, so they do not comply with the rule.

10 the only inconsistency in the bracing style is in line 1342, since elsewhere in the method K&R style is consistently adopted.

```
1343         // Obtain the IOR configuration descriptors for the Ejb using
1344         // the object_id within the SecurityContext field.
1345 
1346         // if object_id is null then nothing to evaluate. This is a sanity
1347         // check - for the object_id should never be null.
1348 
1349         if (object_id == null)
1350             return true;
1351 
1352         if (protocolMgr == null)
1353             protocolMgr = orbHelper.getProtocolManager();
1354 
1355         // Check to make sure protocolMgr is not null.
1356         // This could happen during server initialization or if this call
1357         // is on a callback object in the client VM.
1358         if (protocolMgr == null)
1359             return true;
1360 
```

```
1361        EjbDescriptor ejbDesc = protocolMgr.getEjbDescriptor(object_id);
1362
1363        Set iorDescSet = null;
1364        if (ejbDesc != null) {
1365    ⤑       iorDescSet = ejbDesc.getIORConfigurationDescriptors();
1366    ⤑   }
1367    ⤑   else {
1368    ⤑       // Probably a non-EJB CORBA object.
1369    ⤑       // Create a temporary EjbIORConfigurationDescriptor.
1370    ⤑       iorDescSet = getCorbaIORDescSet();
1371    ⤑   }
1372
1373    ⤑   if(_logger.isLoggable(Level.FINE)) {
1374    ⤑       _logger.log(Level.FINE,
1375    ⤑   ⤑   ⤑   "SecurityMechanismSelector.evaluate_client_conformance: iorDescSet: " + iorDescSet);
1376    ⤑   }
```

9 in lines 1365 to 1371 and lines 1373 to 1376 tabs are used to indent, which is to be avoided. Moreover, we suggest the use of an auto-formatting tool to fix the wild indentation in this fragment of code.

11 the **if** groups in lines 1349 to 1350, lines 1352 to 1353, and lines 1358 to 1359 should surround their single line statements with braces.

13 line 1375 does not comply with the 80 characters limit; however, it would be impossible to do so, unless the string is divided; this line is still acceptable, though, since it is less than 120 characters long (rule 14).

44 the initialisation of `iorDescSet` to **null** (line 1363) is useless, since the immediately following **if-else** group changes its value for sure.

```
1378        /* if there are no IORConfigurationDescriptors configured, then
1379         * no security policy is configured. So consider the client
1380         * to be conformant.
1381         */
1382        if (iorDescSet.isEmpty())
1383            return true;
1384
1385        // go through each EjbIORConfigurationDescriptor trying to find
1386        // a find a CompoundSecMechanism that matches client's actions.
1387        boolean checkSkipped = false;
1388        for (Iterator itr = iorDescSet.iterator(); itr.hasNext();) {
1389            EjbIORConfigurationDescriptor iorDesc =
1390                (EjbIORConfigurationDescriptor) itr.next();
1391            if(skip_client_conformance(iorDesc)){
1392    ⤑       ⤑   if(_logger.isLoggable(Level.FINE)) {
1393    ⤑       ⤑       _logger.log(Level.FINE,
1394    ⤑       ⤑   ⤑       "SecurityMechanismSelector.evaluate_client_conformance: skip_client_conformance");
1395    ⤑       ⤑   }
1396                checkSkipped = true;
1397                continue;
1398            }
```

```
1399            if (! evaluate_client_conformance_ssl(iorDesc, ssl_used, certchain)){
1400    ⤑   ⤑if(_logger.isLoggable(Level.FINE)) {
1401    ⤑   ⤑   _logger.log(Level.FINE,
1402    ⤑   ⤑   ⤑   ⤑"SecurityMechanismSelector.evaluate_client_conformance: evaluate_client_conformance_ssl");
1403    ⤑   ⤑}
1404                checkSkipped = false;
1405                continue;
1406            }
1407            String realmName = "default";
1408            if(ejbDesc != null && ejbDesc.getApplication() != null) {
1409                realmName = ejbDesc.getApplication().getRealm();
1410            }
1411            if(realmName == null) {
1412                realmName = iorDesc.getRealmName();
1413            }
1414            if (realmName == null) {
1415                realmName = "default";
1416            }
1417            if ( ! evaluate_client_conformance_ascontext(ctx, iorDesc ,realmName)){
1418    ⤑   ⤑if(_logger.isLoggable(Level.FINE)) {
1419    ⤑   ⤑   _logger.log(Level.FINE,
1420    ⤑   ⤑   ⤑   ⤑"SecurityMechanismSelector.evaluate_client_conformance: evaluate_client_conformance_ascontext");
1421    ⤑   ⤑}
1422                checkSkipped = false;
1423                continue;
1424            }
1425            if  ( ! evaluate_client_conformance_sascontext(ctx, iorDesc)){
1426    ⤑   ⤑if(_logger.isLoggable(Level.FINE)) {
1427    ⤑   ⤑   _logger.log(Level.FINE,
1428    ⤑   ⤑   ⤑   ⤑"SecurityMechanismSelector.evaluate_client_conformance: evaluate_client_conformance_sascontext");
1429    ⤑   ⤑}
1430               checkSkipped = false;
1431               continue;
1432            }
1433            return true;  // security policy matched.
1434        }
1435        if(checkSkipped)
1436            return true;
1437        return false; // No matching security policy found
1438    }
```

9 in lines 1392 to 1395, lines 1400 to 1403, lines 1418 to 1421, and lines 1426 to 1429 tabs are used to indent, which is to be avoided. Moreover, indentation is wildly done in this fragment of code: we suggest the use of an auto-formatting tool to fix this issue.

11 the `if` groups in lines 1382 to 1383 and lines 1435 to 1436 should surround their single line statements with braces.

13 line 1394, line 1402 , line 1420, and line 1428 do not comply with the 80 characters limit; these lines are still acceptable, though, in compliance with rule 14.

32 since the initialisation of `realmName` string is potentially useless, we suggest a refactoring of the block lines 1407 to 1416 to improve readability and reduce complexity:

```
String realmName;

if(ejbDesc != null && ejbDesc.getApplication() != null) {
    realmName = ejbDesc.getApplication().getRealm();

    if(realmName == null) {
        realmName = iorDesc.getRealmName();

        if (realmName == null) {
            realmName = "default";
        }
    }
} else {
    realmName = "default";
}
```

3.3.2 *Suggestions*

We suggest to nest the **if** block in lines 1358 to 1359 inside the previous one (lines 1352 to 1353), in order to avoid a double check in case `protocolMgr` is not **null**.

Moreover, the following fragment is repeated several times in this method (lines 1373 to 1376, lines 1392 to 1395, lines 1400 to 1403, lines 1418 to 1421, and lines 1426 to 1429), with slight differences each time. In particular, the only varying part is the portion of string `"XYZ"`:

```
if(_logger.isLoggable(Level.FINE)) {
    _logger.log(Level.FINE,
             "SecurityMechanismSelector.evaluate_client_conformance: " + "XYZ");
}
```

We suggest to substitute all occurrences of this fragment with a call to a private method, to which `"XYZ"` is passed as a parameter, in order to save lines of code, reduce complexity and improve readability.

3.4 *Final considerations*

In general the portion of class we were assigned suffers from minor stylistic issues. Most of them will be easily corrected automatically by any of the major IDEs (among them, NetBeans and Eclipse). There are also minor redundancy issues, which can be spotted and corrected with little effort.

However, please note that security mechanisms are a complex matter, and the functional analysis and bug spotting are far beyond the objectives of this document. Deep and thorough testing is needed to guarantee the quality of code.

A
Code inspection checklist

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 capitalised.
   Examples: `class Raster`; `class ImageSprite`.
4. Interface names should be capitalised like classes.
5. Method names should be verbs, with the first letter of each addition word capitalised.
   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 capitalised.
   Examples: `_windowHeight`, `timeSeriesData`.
7. Constants are declared using all uppercase with words separated by an underscore.
   Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

Indention

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

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();
}
```

File organisation

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.

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.

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.

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).

Package and import statements

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

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;
        i. first public class variables;
        ii. next protected class variables;
        iii. next package level (no access modifier);
        iv. last private class variables.
    (e) instance variables;
        i. first public instance variables;
        ii. next protected instance variables;
        iii. next package level (no access modifier);
        iv. 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.

Initialisation 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 initialised before use.
32. Variables are initialised 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.

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.

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.

Object comparison

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

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.

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 parenthesising.
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.

Exceptions

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

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 initialisation, increment and termination expressions.

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.

Appendix

Hours of work

The writing of this document took the following amount of time:

Paolo Antonini 12 hours.

Andrea Corneo 10 hours.

Version control

- **1.0**, 5th January 2016: first release;
- **1.1**, 5th February 2016: final release, with fixes to page headers.