POLITECNICO
MILANO 1863

Code Inspection

Andrea Gussoni Federico Amedeo Izzo Niccolò Izzo

January 5, 2016

Contents

Chapter 1

Classes assigned to the group

The classes assigned to us are two, for a total of 5 methods:

- ClientAuthConfigHelper.java
  Namespace: `appserver/security/jaspic-provider-framework/src/main/java/com/sun/jaspic/config/helper/ClientAuthConfigHelper.java`
  For this class the method assigned is:
  - createAuthContext [Start Line 84]
- AuthConfigProviderHelper.java
  Namespace: `appserver/security/jaspic-provider-framework/src/main/java/com/sun/jaspic/config/helper/AuthConfigProviderHelper.java`
  For this class the methods assigned are:
  - getModuleTypes [Start Line 92]
  - selfRegister [Start Line 126]
  - contextsAreEqual [Start Line 213]
  - oldRefresh [Start Line 233]

Chapter 2

Functional role of assigned set of classes

The functional role of the given classes is the following:

2.1 ClientAuthConfigHelper Class

This is a helper class of ClientAuthConfig, used to get the necessary Context class to enable the use of AuthModules security subsystem. In particular the ClientAuthConfig class represents client's interface to the AuthConfig subsystem that manages the configuration of AuthModules. AuthModules are components used to secure requests such as HTTP, EJB, SOAP. Callers do not operate directly on AuthModules, instead they use ClientAuthContext or ServerAuthContext to manage the invocation of modules. An AuthConfig implementation chooses the modules to be invokes based on the id and intercept chosen, and encapsulates those modules in a ClientAuthContext or ServerAuthContext, returning that instance.
The main methods implemented in the ClientAuthConfigHelper class are the following:

- createAuthContext: which creates a new ClientAuthContext loading the necessary modules.
- validateResponse: which invokes the configured modules upon a received response.
- secureRequest: invokes the configured modules to secure (e.g. attach username,password and/or encrypt) the initial request

- getAuthContext: is the method invoked on the ClientAuthConfigHelper to get the ClientAuthContext as explained above.

References: Knowledge was taken from

- AuthConfig javadoc page: `http://glassfish.pompel.me/com/sun/enterprise/security/jauth/AuthConfig.html`
- ClientAuthContext javadoc page: `http://glassfish.pompel.me/com/sun/enterprise/security/jauth/ClientAuthContext.html`

2.2 AuthConfigProviderHelper Class

This class is used to obtain authentication context configuration objects, like ClientAuthConfig or ServerAuthConfig, which functionality and use is explained in the ClientAuthConfigHelper description before.
The main modules provided are:

- getModuleTypes: Returns an object containing the module types, whether it is ServerAuthModule or ClientAuthModule.
- selfRegister: method called by AuthConfigFactory to register the AuthConfigProvider itself.
- getClientCallbackHandler and getServerCallbackHandler: used to provide a CallbackHandler.
- getClientAuthConfig: returns an instance of the class ClientAuthConfigHelper, described above.
- getServerAuthConfig: same as before but returns ServerAuthConfigHelper.
- contextsAreEqual: checks if two RegistrationContext are equal.

References: Knowledge was taken from

- AuthConfigProvider javadoc page: `https://java.net/downloads/jaspic-spec/releases/1.1/MR/javadoc/javax/security/auth/message/config/AuthConfigProvider.html`
- AuthConfigFactory javadoc page: `https://java.net/downloads/jaspic-spec/releases/1.1/MR/javadoc/javax/security/auth/message/config/AuthConfigFactory.html`

Chapter 3

List of issues found by applying the checklist

3.1 ClientAuthConfigHelper Class

The Javadoc of this class contains only the methods signatures, without any comment about the class functionality. Therefore we had to infer the functionality of the class from the description of an interface implemented here. This interface, named **ClientAuthContext()** is implemented via an anonymous class and sent to the caller of the method **createAuthContext()**. For better clarity it is suggested to describe the role of this class also in its own Javadoc [Java Source Files, 23].

3.1.1 createAuthContext method

3.1.1.1 One character variable [Naming Convention, 2]

In the method createAuthContext() the one character variable **m** has not a throwaway purpose. In fact the variable **m** is not used as a counter of a loop but contains a full data structure. This is a violation of rule [Naming Convention, 2].

```
1  ClientAuthModule[] m;
   try {
3      m = acHelper.getModules(new ClientAuthModule[0], authContextID);
   } catch (AuthException ae) {
5      logIfLevel(Level.SEVERE, ae,
               "ClientAuthContext: ", authContextID,
7              "of AppContext: ", getAppContext(),
               "unable to load client auth modules");
9      throw ae;
   }
11
   MessagePolicy requestPolicy =
```

```
13          mpDelegate.getRequestPolicy(authContextID, properties);
    MessagePolicy responsePolicy =
15          mpDelegate.getResponsePolicy(authContextID, properties);

17  boolean noModules = true;
    for (int i = 0; i < m.length; i++) {
19      if (m[i] != null) {
            if (isLoggable(Level.FINE)) {
21              logIfLevel(Level.FINE, null,
                        "ClientAuthContext: ", authContextID,
23                      "of AppContext: ", getAppContext(),
                        "initializing module");
25          }
            noModules = false;
27          checkMessageTypes(m[i].getSupportedMessageTypes());
            m[i].initialize(requestPolicy, responsePolicy,
29                  cbh, acHelper.getInitProperties(i, properties));
        }
31  }
    if (noModules) {
33      logIfLevel(Level.WARNING, null,
                "CLientAuthContext: ", authContextID,
35              "of AppContext: ", getAppContext(),
                "contains no Auth Modules");
37  }
    return m;
```

code/ClientAuthConfigHelper/fragment_1.java

3.1.1.2 Wrapping lines when breaking line [Wrapping Lines, 17]

The method declaration of **ClientAuthConfigHelper** violates the rule [Wrapping Lines, 17] that states that when breaking lines the new line statement should be aligned as the same level of the expression on the previous line. Probably this decision was taken in order to achieve a more compact signature representation, since this methods accepts a large number of parameters and we still have to fulfill the 80 characters limit.

```
    protected ClientAuthConfigHelper(String loggerName, EpochCarrier
        providerEpoch,
2           AuthContextHelper acHelper, MessagePolicyDelegate mpDelegate,
            String layer, String appContext, CallbackHandler cbh)
4           throws AuthException {
      super(loggerName, providerEpoch, mpDelegate, layer, appContext, cbh);
6     this.acHelper = acHelper;
    }
```

code/ClientAuthConfigHelper/fragment_2.java

3.1.1.3 Variable Initialization [Initialization and Declarations, 32]

The variable **contextMap** is not initialized at the moment of the declaration but afterwards. This violates the rule [Initialization and Declarations,

32]. Indeed it is a threat to leave this variable uninitialized, someone unaware of this implementation detail could just forget to call the initialization method and generate a NullPointerException. Maybe this method was created to provide a quick way of reset the variable's content, if this is the case the method can be kept but the variable must be however initialized in his declaration sentence.

```
final static AuthStatus[] vR_SuccessValue = {AuthStatus.SUCCESS};
final static AuthStatus[] sR_SuccessValue = {AuthStatus.SEND_SUCCESS};
HashMap<String, HashMap<Integer, ClientAuthContext>> contextMap;
AuthContextHelper acHelper;

protected ClientAuthConfigHelper(String loggerName, EpochCarrier
    providerEpoch,
        AuthContextHelper acHelper, MessagePolicyDelegate mpDelegate,
        String layer, String appContext, CallbackHandler cbh)
        throws AuthException {
    super(loggerName, providerEpoch, mpDelegate, layer, appContext, cbh);
    this.acHelper = acHelper;
}

protected void initializeContextMap() {
    contextMap = new HashMap<String, HashMap<Integer, ClientAuthContext>>();
}
```

code/ClientAuthConfigHelper/fragment_3.java

3.1.1.4 Variable Declaration [Initialization and Declarations, 33]

The variables **requestPolicy** and **responsePolicy** are not declared at the beginning of a block, as stated in the rule [Initialization and Declarations, 33]. These variables are not in a for loop, therefore they must be declared at the beginning of theirs current block, in order to obtain a better readability of the whole code.

```
ClientAuthModule[] module = init();

ClientAuthModule[] init() throws AuthException {

    ClientAuthModule[] m;
    try {
        m = acHelper.getModules(new ClientAuthModule[0], authContextID);
    } catch (AuthException ae) {
        logIfLevel(Level.SEVERE, ae,
                "ClientAuthContext: ", authContextID,
                "of AppContext: ", getAppContext(),
                "unable to load client auth modules");
        throw ae;
    }

    MessagePolicy requestPolicy =
            mpDelegate.getRequestPolicy(authContextID, properties);
    MessagePolicy responsePolicy =
            mpDelegate.getResponsePolicy(authContextID, properties);
```

code/ClientAuthConfigHelper/fragment_4.java

3.1.1.5 Typo in logging [Output Format, 42]

There is a typo in the log message at line 130, **CLientAuthContext** should be **ClientAuthContext**.

```
if (noModules) {
    logIfLevel(Level.WARNING, null,
            "CLientAuthContext: ", authContextID,
            "of AppContext: ", getAppContext(),
            "contains no Auth Modules");
}
```

code/ClientAuthConfigHelper/fragment_5.java

3.1.1.6 Meaningful name [Naming Convention, 1]

The name **init()** for this method is not meaningful enough, it should at least be replaced with **initializeModules()**, or another self-explanatory name. Also considering that this method has an important role in the this class and handles the complete loading and initialization of all the authentication modules. This is a violation of rule [Naming Convention, 1].

```
ClientAuthModule[] init() throws AuthException {

    ClientAuthModule[] m;
    try {
        m = acHelper.getModules(new ClientAuthModule[0], authContextID);
    } catch (AuthException ae) {
        logIfLevel(Level.SEVERE, ae,
                "ClientAuthContext: ", authContextID,
                "of AppContext: ", getAppContext(),
                "unable to load client auth modules");
        throw ae;
    }

    MessagePolicy requestPolicy =
            mpDelegate.getRequestPolicy(authContextID, properties);
    MessagePolicy responsePolicy =
            mpDelegate.getResponsePolicy(authContextID, properties);

    boolean noModules = true;
    for (int i = 0; i < m.length; i++) {
        if (m[i] != null) {
            if (isLoggable(Level.FINE)) {
                logIfLevel(Level.FINE, null,
                        "ClientAuthContext: ", authContextID,
                        "of AppContext: ", getAppContext(),
                        "initializing module");
            }
            noModules = false;
            checkMessageTypes(m[i].getSupportedMessageTypes());
```

```
30              m[i].initialize(requestPolicy, responsePolicy,
                        cbh, acHelper.getInitProperties(i, properties));
32          }
        }
34      if (noModules) {
            logIfLevel(Level.WARNING, null,
36                  "CLientAuthContext: ", authContextID,
                    "of AppContext: ", getAppContext(),
38                  "contains no Auth Modules");
        }
40      return m;
    }
```

code/ClientAuthConfigHelper/fragment_6.java

3.1.1.7 Lack of Javadoc documentation [Java Source Files, 22]

The Javadoc provided to us lacks informations about the interface **ClientAuthModule** implemented in this class, this means that the checks about the correctness of the parameters passed to the methods that implement the interface cannot be verified. The Javadoc contains an homonymous interface with different method signatures. The correct interface Javadoc is however present in the JavaEE 7 documentation, and according to that documentation, the parameters' order is correct.

```
1 import javax.security.auth.message.config.ClientAuthConfig;
  import javax.security.auth.message.config.ClientAuthContext;
3 import javax.security.auth.message.module.ClientAuthModule;
  //the object is initialized
5 ClientAuthModule[] module = init();
  //and then some methods that implement the interface are used.
7 //code here
  status[i] = module[i].validateResponse(arg0, arg1, arg2);
9 //code here
  status[i] = module[i].secureRequest(arg0, arg1);
11 //code here
  module[i].cleanSubject(arg0, arg1);
```

code/ClientAuthConfigHelper/fragment_7.java

3.1.1.8 Duplicated Code [Class and Interface Declarations, 27]

As we can see in the attached piece of code the methods **validateResponse()** and **secureRequest()** contain a lot of duplicated code that should be re-factored in order to avoid problems for future code revisions. We can see indeed that the only changing part between the two methods are the messages of the logs.

```
public AuthStatus validateResponse(MessageInfo arg0, Subject arg1, Subject
    arg2) throws AuthException {
2     AuthStatus[] status = new AuthStatus[module.length];
      for (int i = 0; i < module.length; i++) {
```

```
        if (module[i] == null) {
            continue;
        }
        if (isLoggable(Level.FINE)) {
            logIfLevel(Level.FINE, null,
                    "ClientAuthContext: ", authContextID,
                    "of AppContext: ", getAppContext(),
                    "calling vaidateResponse on module");
        }
        status[i] = module[i].validateResponse(arg0, arg1, arg2);
        if (acHelper.exitContext(vR_SuccessValue, i, status[i])) {
            return acHelper.getReturnStatus(vR_SuccessValue,
                    AuthStatus.SEND_FAILURE, status, i);
        }
    }
    return acHelper.getReturnStatus(vR_SuccessValue,
            AuthStatus.SEND_FAILURE, status, status.length - 1);
}

public AuthStatus secureRequest(MessageInfo arg0, Subject arg1) throws
    AuthException {
    AuthStatus[] status = new AuthStatus[module.length];
    for (int i = 0; i < module.length; i++) {
        if (module[i] == null) {
            continue;
        }
        if (isLoggable(Level.FINE)) {
            logIfLevel(Level.FINE, null,
                    "ClientAuthContext: ", authContextID,
                    "of AppContext: ", getAppContext(),
                    "calling secureResponse on module");
        }
        status[i] = module[i].secureRequest(arg0, arg1);
        if (acHelper.exitContext(sR_SuccessValue, i, status[i])) {
            return acHelper.getReturnStatus(sR_SuccessValue,
                    AuthStatus.SEND_FAILURE, status, i);
        }
    }
    return acHelper.getReturnStatus(sR_SuccessValue,
            AuthStatus.SEND_FAILURE, status, status.length - 1);
}
```

code/ClientAuthConfigHelper/fragment_8.java

3.1.1.9 Duplicated Code [Class and Interface Declarations, 27]

The method **cleanSubject()** seems to have a lot of shared code with the previous two methods except for the log message (at least for the first part). It is highly suggested to re-factor the code in order to reuse a common structure in all the cases and changing only the desired part (e.g. using a private method).

```
public void cleanSubject(MessageInfo arg0, Subject arg1) throws
    AuthException {
    for (int i = 0; i < module.length; i++) {
        if (module[i] == null) {
            continue;
```

```
        }
        if (isLoggable(Level.FINE)) {
            logIfLevel(Level.FINE, null,
                    "ClientAuthContext: ", authContextID,
                    "of AppContext: ", getAppContext(),
                    "calling cleanSubject on module");
        }
        module[i].cleanSubject(arg0, arg1);
    }
}
```

code/ClientAuthConfigHelper/fragment_9.java

3.2 AuthConfigProviderHelper Class

3.2.1 Class Wide Issues or Methods not assigned.

Frequently, in the class, the convention regarding the maximum length of a line is not respected, and often we have lines longer than 100 characters, although the limit of 120 characters is never exceeded. This behavior may be voluntary in order to avoid too many line-breaks, if it is not the case, a re-indentation must be considered.

3.2.1.1 Variables Order Declaration [Class and Interface Declarations, 25]

At the beginning of the class the friendly variables are declared after the private ones, on the contrary of what is said in the rule [Class and Interface Declarations, 25].

```
public static final String LAYER_NAME_KEY = "message.layer";
public static final String ALL_LAYERS = "*";
public static final String LOGGER_NAME_KEY = "logger.name";
public static final String AUTH_MODULE_KEY = "auth.module.type";
public static final String SERVER_AUTH_MODULE = "server.auth.module";
public static final String CLIENT_AUTH_MODULE = "client.auth.module";
private ReentrantReadWriteLock instanceReadWriteLock = new
    ReentrantReadWriteLock();
private Lock rLock = instanceReadWriteLock.readLock();
private Lock wLock = instanceReadWriteLock.writeLock();
HashSet<String> selfRegistered;
EpochCarrier epochCarrier;
```

code/AuthConfigProviderHelper/fragment_3.java

3.2.1.2 Variable Initialization [Initialization and Declarations, 32]

The variables **selfRegistered** and **epochCarrier** are not initialized at the moment of the declaration, and though they are not dependent on the result

of a computation this violates the indications of the rule [Initialization and Declarations, 32].

```
HashSet<String> selfRegistered;
EpochCarrier epochCarrier;

protected AuthConfigProviderHelper() {
    selfRegistered = new HashSet<String>();
    epochCarrier = new EpochCarrier();
}
```

code/AuthConfigProviderHelper/fragment_4.java

3.2.1.3 Variable Declaration [Initialization and Declarations, 33]

The variable **contexts** is not declared at the beginning of a block, as stated in the rule [Initialization and Declarations, 33]. This variable is not used as a counter in a loop, therefore it must be declared at the beginning of the current block, in order to obtain a better readability of the whole code.

```
if (noModules) {
    logIfLevel(Level.WARNING, null,
            "CLientAuthContext: ", authContextID,
            "of AppContext: ", getAppContext(),
            "contains no Auth Modules");
}
```

code/ClientAuthConfigHelper/fragment_5.java

3.2.2 getModuleTypes method

3.2.2.1 Lack of comments [Comments, 18]

The method does not contain any comment that can help understanding what type of operation it is doing. Consider to insert at least a brief description at the beginning of the block of code. This is a violation of rule [Comments, 18].

3.2.2.2 Variable Initialization [Initialization and Declarations, 32]

The variable **rvalue** is initialized when declared, but as it seems dependent on a computation it is re-initialized below on the basis of the result of a computation. In the middle of the method the variable is never used. Consider to assign the effective value to the object at the end of the computation. This is a violation of rule [Initialization and Declarations, 32].

```
protected Class[] getModuleTypes() {
  Class[] rvalue = new Class[]{
    javax.security.auth.message.module.ServerAuthModule.class,
```

```
4       javax.security.auth.message.module.ClientAuthModule.class
    };
6   Map<String, ?> properties = getProperties();
    if (properties.containsKey(AUTH_MODULE_KEY)) {
8     String keyValue = (String) properties.get(AUTH_MODULE_KEY);
      if (SERVER_AUTH_MODULE.equals(keyValue)) {
10      rvalue = new Class[]{
          javax.security.auth.message.module.ServerAuthModule.class
12      };
      } else if (CLIENT_AUTH_MODULE.equals(keyValue)) {
14      rvalue = new Class[]{
          javax.security.auth.message.module.ClientAuthModule.class
16      };
      }
18  }
    return rvalue;
20 }
```

code/AuthConfigProviderHelper/fragment_2.java

3.2.3 selfRegister method

3.2.3.1 One character variable [Naming Convention, 2]

The usage of the one character variable is border-line, it isn't a variable used in a loop, and although it may be considered a case of throwaway variable, it is highly recommended to give it a more meaningful name. This suggestion was made according to the rule [Naming Convention, 2].

```
for (String i : regID) {
2   if (selfRegistered.contains(i)) {
      RegistrationContext c = getFactory().getRegistrationContext(i);
4     if (c != null && !c.isPersistent()) {
        toBeUnregistered.add(i);
6     }
    }
8 }
```

code/AuthConfigProviderHelper/fragment_1.java

3.2.4 contextsAreEqual method

3.2.4.1 Lack of comments [Comments, 18]

The method does not contain any comment that can help understanding what type of operation it is doing. Consider to insert at least a brief description at the beginning of the block of code. This is stated by the rule [Comments, 18].

3.2.5 Brutish Programming [Computation, Comparisons and Assignments, 44]

This method is entirely composed by some comparisons between objects passed as parameters, and returns a boolean value. No comments are provided to know how this comparison in performed, and it seems a magic trick to perform a task. This violates the rule on avoiding Brutish Programming [Computation, Comparisons and Assignments, 44]

```
public boolean contextsAreEqual(RegistrationContext a, RegistrationContext b
    ) {
    if (a == null || b == null) {
        return false;
    } else if (a.isPersistent() != b.isPersistent()) {
        return false;
    } else if (!a.getAppContext().equals(b.getAppContext())) {
        return false;
    } else if (!a.getMessageLayer().equals(b.getMessageLayer())) {
        return false;
    } else if (!a.getDescription().equals(b.getDescription())) {
        return false;
    }
    return true;
}
```

code/AuthConfigProviderHelper/fragment_6.java

3.2.5.1 One character variable [Naming Convention, 2]

In this method the two objects passed as parameters are identified with a one character variable. This behavior is considered incorrect, due to a violation of the rule [Naming Convention, 2], and also because a meaningful name would help in understanding what this method is doing considering the lack of documentation.

3.2.6 oldRefresh method

3.2.6.1 Lack of comments [Comments, 18]

In the description of the method there is an ambiguous reference to a specification that is here implemented and about its properties. Consider to make this part more clear maybe with an explicit reference to the specification. This is a violation of rule [Comments, 18].

Chapter 4

Other problems highlighted

4.1 Deep Nesting of Conditional Instructions

During the *Ingegneria del Software - 085885* course, we used the tool **Sonar** to perform some static analysis and evaluation of the code we wrote. One of the warnings that the tool often gave to us regarded the deep nesting of conditional instructions (like **if**, **for**, **while**). The tool suggested to refactor the code in order to have a minor level of nesting of instructions. We noticed that often in the assigned class the conditional instructions are deeply nested, and this fact often compromises or makes very difficult to understand the meaning of the code. This fact also makes very hard to use some code-analysis techniques to check the correctness of every possible path of execution.

4.2 Lack of Documentation

As highlighted often in the other sections, the classes and in particular the methods assigned to us lack of any kind of documentation about what they do. We tried to do a Code Inspection on the basis of what we were able to infer by the code itself. Also we found on-line a Javadoc that seems to be more exhaustive than the one provided. In this case we warmly suggest to expand as much as possible the documentation of these classes, also because they are responsible for the encryption and authentication of the web sessions, and considering that nowadays bugs in the implementations of cryptographic functions seems to affect a lot of software, a extensive documentation may help to correct future bugs in a faster and more efficient way.

4.3 Indention

Often the limit of line-lenght of 80 characters is exceeded although the hard-limit of 120 characters is never exceeded. If this behavior is intentional (maybe to improve the readability of the source code) no further actions have to be taken. Otherwise it is highly suggested to re-factor the code in order to improve the readability.

Chapter 5

Checklist

In order to perform the code inspection assignement we used the following checklist provided us by the course teachers.

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`.

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 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.

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.

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.

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 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.

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 initialization, 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.

Chapter 6

Appendix

6.1 Used Software

For the redaction of this document we used various software, here a little list:

- LaTeX framework (MacTeX on OS X and TeX Live on GNU/Linux) to generate the document.
- Various editor to edit the source file:
  - Sublime Text 3 (Beta)
  - Atom.io
  - Vim
- Self-Hosted ShareLaTeX to collaboratively edit the document.
- Git to version the source, GitHub to host the repository.
- Teamspeak and Hangouts used to organize conference-calls in order to work together.

6.2 Hours of Work

We tried to distribute in an equal way the workload for each team-member. In particular we tried to arrange physical meetings or conference calls for the parts where important choices had to be made. The code analysis was made in a redundant way, so to find also the defects that just one of us could have missed. We estimated that each of us spent on average 12 hours on the whole process, from the code analysis to the drafting of this document. Therefore this task took a total of 36 hours of work.