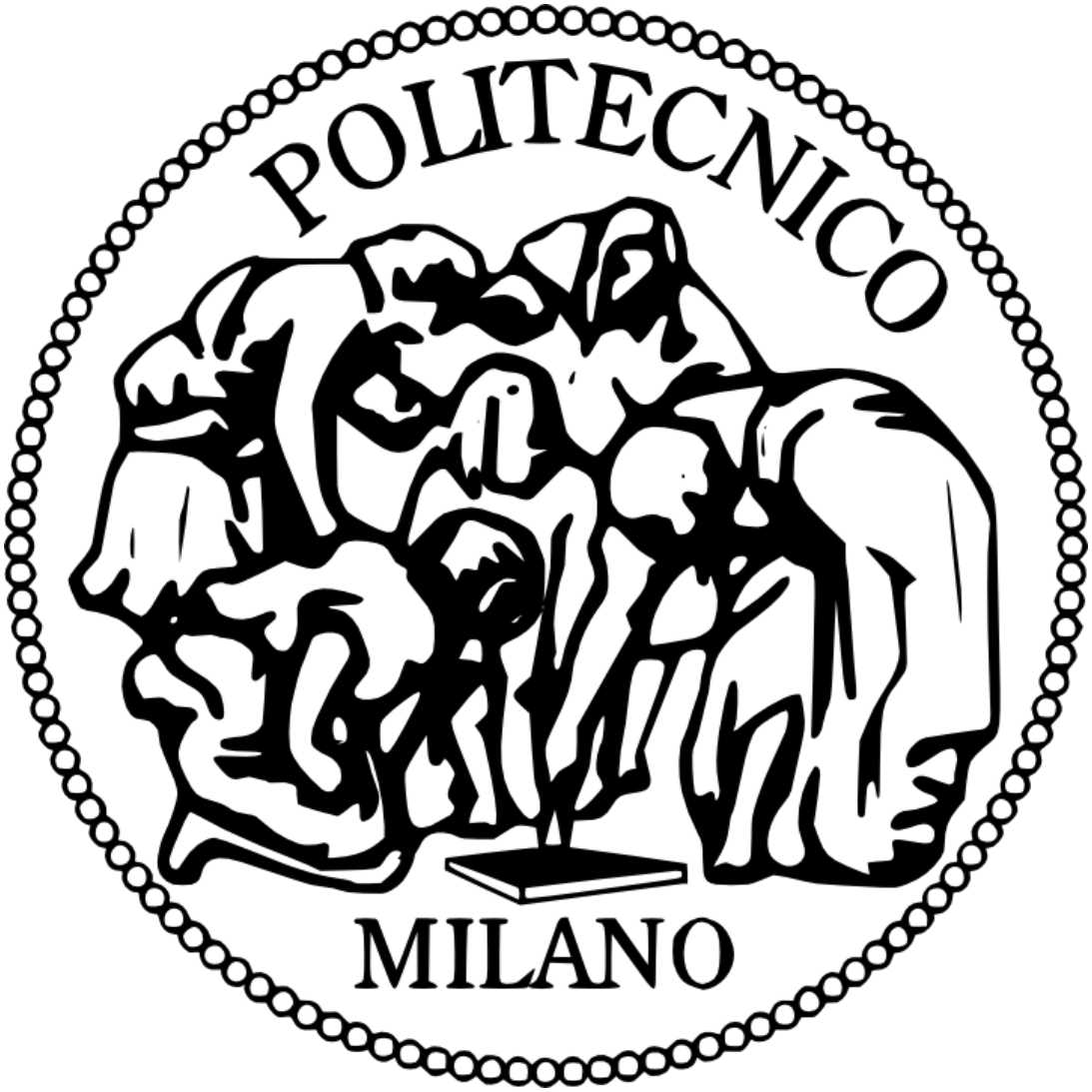

Code Inspection Document:

Glassfish 4.1.1

Authors:

Bucci Giovanni

De Togni Riccardo

1. Introduction

2. Inspected Classes

In Java, the idea of a namespace is embodied in Java packages. All code belongs to a package, although that package need not be explicitly named.[1] The only inspected class is Archivist, a class contained in the com.sun.enterprise.deployment.archivist package.

2.1 Functional Role

Archivist, as the javadoc says "[…] contains all common behaviour for Archivists. Archivists Classes are responsible for reading and writing correct J2EE Archives". Archives in Java Enterprise Edition are container for modules or applications, which are composed of one or more module. When a Developer has to deploy a module on a web server for instance, it has to package it as a Web Archive (.war files). Archivist is an Abstract class since it is declared "abstract" and contains many abstract methods. The main known implementations are AppClientArchivist, ApplicationArchivist and WebArchivist.

The function of each analysed method will be presented in the specific section.

2.2 Found Issues and improvable features

2.2.1 Class Declaration

Archivist already observe all the declaration rules. Packages are the first statements, then there are imports and after that start the class code, which is the only public class of the file.

2.2.2 Attributes

Some issues in the attributes declaration have been found in this class. In order to bring all the attributes in a state that respects the standard they have been just re-ordered since the naming rules were already respected. The schedule followed is presented below:

[1] https://en.wikipedia.org/wiki/Namespace#In_programming_languages

- Static Class Variables
- Instance Variables

Both ordered by visibility:

- Public
- Protected
- Package Level
- Private

Here it is presented a before/after situation, it allows to understand at a glance all the modifications:

Attributes: Before and After

```
    protected static final Logger logger =
            DOLUtils.getDefaultLogger();

    public static final String MANIFEST_VERSION_VALUE = "1.0";

    // the path for the underlying archive file
    protected String path;

    // should we read or save the runtime info.
    protected boolean handleRuntimeInfo = true;

    // default should be false in production
    protected boolean annotationProcessingRequested = false;

    // attributes of this archive
    protected Manifest manifest;

    // standard DD file associated with this archivist
    protected DeploymentDescriptorFile<T> standardDD;

    // configuration DD files associated with this archivist
    protected List<ConfigurationDeploymentDescriptorFile> confDDFiles;

    // the sorted configuration DD files with precedence from
    // high to low
    private List<ConfigurationDeploymentDescriptorFile> sortedConfDDFiles;

    // configuration DD file that will be used
    private ConfigurationDeploymentDescriptorFile confDD;

    // resources...
    private static final LocalStringManagerImpl localStrings =
            new LocalStringManagerImpl(Archivist.class);

    // class loader to use when validating the DOL

    protected ClassLoader classLoader = null;

    // boolean for XML validation
    private boolean isValidatingXML = true;

    // boolean for runtime XML validation
    private boolean isValidatingRuntimeXML = true;

    // xml validation error level reporting/recovering
    private String validationLevel = "parsing";

    // runtime xml validation error level reporting/recovering
    private String runtimeValidationLevel = "parsing";

    // error handler for annotation processing
    private ErrorHandler annotationErrorHandler = null;

    private static final String WSDL = ".wsdl";
    private static final String XML = ".xml";
    private static final String XSD = ".xsd";


    protected static final String APPLICATION_EXTENSION = ".ear";
    protected static final String APPCLIENT_EXTENSION = ".jar";
    protected static final String WEB_EXTENSION = ".war";
    protected static final String WEB_FRAGMENT_EXTENSION = ".jar";
    protected static final String EJB_EXTENSION = ".jar";
    protected static final String CONNECTOR_EXTENSION = ".rar";
    //Used to detect the uploaded files which always end in ".tmp"
    protected static final String UPLOAD_EXTENSION = ".tmp";

    private static final String PROCESS_ANNOTATION_FOR_OLD_DD =
            "process.annotation.for.old.dd";

    private static final boolean processAnnotationForOldDD =
            Boolean.getBoolean(PROCESS_ANNOTATION_FOR_OLD_DD);

    protected T descriptor;

    @Inject
    protected ServiceLocator habitat;

    @Inject
    protected ServiceLocator locator;

    @Inject
    SJSASFactory annotationFactory;

    @Inject
    ArchiveFactory archiveFactory;

    protected List<ExtensionsArchivist> extensionsArchivists;
```

```
    public static final String MANIFEST_VERSION_VALUE = "1.0";

    protected static final Logger logger =
            DOLUtils.getDefaultLogger();

    protected static final String APPLICATION_EXTENSION = ".ear";
    protected static final String APPCLIENT_EXTENSION = ".jar";
    protected static final String WEB_EXTENSION = ".war";
    protected static final String WEB_FRAGMENT_EXTENSION = ".jar";
    protected static final String EJB_EXTENSION = ".jar";
    protected static final String CONNECTOR_EXTENSION = ".rar";
    //Used to detect the uploaded files which always end in ".tmp"
    protected static final String UPLOAD_EXTENSION = ".tmp";

    private static final String PROCESS_ANNOTATION_FOR_OLD_DD =
            "process.annotation.for.old.dd";

    private static final boolean processAnnotationForOldDD =
            Boolean.getBoolean(PROCESS_ANNOTATION_FOR_OLD_DD);

    // resources...
    private static final LocalStringManagerImpl localStrings =
            new LocalStringManagerImpl(Archivist.class);

    private static final String WSDL = ".wsdl";
    private static final String XML = ".xml";
    private static final String XSD = ".xsd";

    // the path for the underlying archive file
    protected String path;

    // should we read or save the runtime info.
    protected boolean handleRuntimeInfo = true;

    // default should be false in production
    protected boolean annotationProcessingRequested = false;

    // attributes of this archive
    protected Manifest manifest;

    // standard DD file associated with this archivist
    protected DeploymentDescriptorFile<T> standardDD;

    // configuration DD files associated with this archivist
    protected List<ConfigurationDeploymentDescriptorFile> confDDFiles;

    protected T descriptor;

    @Inject
    protected ServiceLocator habitat;

    @Inject
    protected ServiceLocator locator;

    // class loader to use when validating the DOL
    protected ClassLoader classLoader = null;

    protected List<ExtensionsArchivist> extensionsArchivists;

    @Inject
    SJSASFactory annotationFactory;

    @Inject
    ArchiveFactory archiveFactory;

    // the sorted configuration DD files with precedence from
    // high to low
    private List<ConfigurationDeploymentDescriptorFile> sortedConfDDFiles;

    // configuration DD file that will be used
    private ConfigurationDeploymentDescriptorFile confDD;

    // boolean for XML validation
    private boolean isValidatingXML = true;

    // boolean for runtime XML validation
    private boolean isValidatingRuntimeXML = true;

    // xml validation error level reporting/recovering
    private String validationLevel = "parsing";

    // runtime xml validation error level reporting/recovering
    private String runtimeValidationLevel = "parsing";

    protected List<ExtensionsArchivist> extensionsArchivists;
```

2.3 Method Analysis

2.3.1 Method: processAnnotations

Annotation processing is an operation for scanning and processing annotations at compile time. This method processes the annotations contained in a Bundle Descriptor. The processing depends on the type of descriptor passed as parameter.

The method is found to be well-parenthesized (Kernigan and Ritchie style) and correctly indented. All other standard rules are observed except the line length of 120, which is violated in line 606. This issue has been corrected with the insertion of a line break and a "+" operator placed just before it.
The comparisons are always made with "==" but it do not violate the rule 40 because are all comparison to null pointer and that is the only way to make it.
It is worth to highlight that since Archivist is abstract cannot be instanced and this method cannot have an output.
Personally, we would have created a local variable that contains the to-be-returned object (initialized to null) in order to eliminate all the return statement that breaks the execution before the end. In this way the "one return only" will be respected.

Now we will proceed with the checklist:

Naming Conventions:

1. OK
2. No one-character variables
3. //[2]
4. //
5. OK
6. //
7. // NO CONSTANTS

Indentation:

8. Four Spaces to indent are used all over the class
9. OK

[2] From now on, "//" will mean that the correspondent number does not refer to the method and for that it will not be analysed.

Braces:

10. Kernigan and Ritchie style used
11. OK

File Organisation:

12. OK
13. @line 606-607 length is over 80 chars
14. @line 606-607 length is over 120 chars → solution already presented above

Wrapping Lines:

15. OK
16. OK
17. OK

Comments:

18. Comments are quite well explaining what the method does, maybe some more introduction needed
19. No commented code

Java Source Files:

20. //
21. //
22. //
23. Javadoc contains the correct method declaration

Package and Import Statement:

24. //

Class/Interface Declaration:

25. //
26. //
27. //

Initialization and Declarations:

28. OK
29. OK

30. OK
31. OK
32. OK
33. OK

Method Calls:

34. OK
35. OK
36. Return value are used in a legal way. Above it is presented a personal optimization

Arrays:

37. No arrays used
38. //
39. //

Object Comparison:

40. (Explained Above)

Output Format:

41. No Text Output: It returns a ProcessingResult Object
42. OK
43. //

Computation, Comparisons and Assignments:

44. No brutish programming
45. OK
46. OK
47. No divisions
48. No mathematical operations
49. OK
50. OK
51. OK

Exceptions:

52. All exception are managed by the throw clause: AnnotationProcessException is directly built and thrown by the method
53. OK

Flow of Control:

54. No switches
55. No switches
56. No Loops

Files:

57. OK [EX: Line 600]
58. The only file opened by the method is closed by the throw call (line 607)
59. //
60. File Exceptions are generally managed by IOException → It is suggested to handle them in the subclasses which extend Archivist

2.3.2 Method: readStandardDeploymentDescriptor

This method is responsible to get a Standard Deployment Descriptor and instance it. In other words this method receive an Archive and returns an initialized descriptor instance. In case something goes wrong and it is impossible to get an Input Stream from the archive the method returns the Default Bundle Descriptor. That is to prevent application crash for instance during a server restart when there are no physical descriptors for a while.

Now we will proceed with the checklist:

Naming Conventions:

1. OK
2. No one-character variables
3. //
4. //
5. OK
6. //
7. // NO CONSTANTS

Indentation:

8. Four Spaces to indent are used all over the class

9. OK

Braces:

10. Kernigan and Ritchie style used
11. OK

File Organisation:

12. OK
13. OK
14. OK

Wrapping Lines:

15. OK
16. OK
17. OK

Comments:

18. Comments are quite well explaining what the method does, maybe some more introduction needed
19. No commented code

Java Source Files:

20. //
21. //
22. //
23. Javadoc contains the correct method declaration

Package and Import Statement:

24. //

Class/Interface Declaration:

25. //
26. //
27. //

Initialization and Declarations:

28. OK

29. OK
30. OK
31. OK
32. OK
33. OK

Method Calls:

34. OK
35. OK
36. OK

Arrays:

37. No arrays used
38. //
39. //

Object Comparison:

40. Comparison made with “==” because they refers to null pointers and not to objects

Output Format:

41. No text output: It returns a Bundle Descriptor
42. OK
43. //

Computation, Comparisons and Assignments:

44. No brutish programming
45. OK
46. OK
47. No divisions
48. No mathematical operations
49. OK
50. OK
51. OK

Exceptions:

52. All exception are managed by the throw clause
53. OK

Flow of Control:

54. No switches
55. No switches
56. No Loops

Files:

57. OK [EX: Line 600]
58. Issue: @Line 660 a file is opened but never closed → The closing of it is not the responsibility of this method because the method returns just a BundleDescriptor linked to that file
59. //
60. File Exceptions are generally managed by IOException → It is suggested to handle them in the subclasses which extend Archivist

2.3.3 Method: write

The task of this method is to save an archive in a specific output file. Archives are compressed file containing java classes, and it is possible that fragments of data are modified during the process; it is necessary to give a way to save the changes made.

The method accepts a readable archive and a file path as input, but does not have a return value.

It checks if the path inserted represents an existing file, in order to open it if present, or to create it;

Then copies the contents of the given archive in a new archive ad-hoc instanced and saved in the target file.

Finally it makes sure that every stream and every file is closed in order to avoid critical situations.

The method is well-parenthesized (Kernigan and Ritchie style) and correctly indented.

All other standard rules are observed (correct naming of elements, indention, etc.) except

for line length that in some cases exceeds the 80 characters and in those cases that can't be modified but the readability is not influenced and they don't exceed the 120 characters limit so it is fine to leave them as they are.

The comparisons are always made with “==” or “!=” but since they are comparisons to null pointers it do not violate the rule about using *equals* method because this is the only way to make them.

Exceptions are correctly detected but not managed by the method since it is abstract so it just throws the found exceptions. These exceptions will be catched by other components dedicated to solving the specific kind of error; this is done in order to reduce coupling and permitting a generalization of the methods.

Now we will proceed with the checklist:

Naming Conventions:

1. OK
1. //
2. //
3. //
4. OK
5. OK
6. OK

Indentation:

7. OK
8. OK

Braces:

9. OK
10. OK

File Organisation:

11. OK
12. Some lines exceed 80 characters
13. OK

Wrapping Lines:

14. OK

15. OK

16. OK

Comments:

17. OK

18. OK

Java Source Files:

19. //

20. //

21. OK

22. OK

Package and Import Statement:

23. //

Class/Interface Declaration:

24. OK

25. OK

26. OK

Initialization and Declarations:

27. OK

28. OK

29. OK

30. OK

31. OK

32. OK

Method Calls:

33. OK

34. OK

35. OK

Arrays:

36. //

37. //

38. //

Object Comparison:

39. OK (see explanation)

Output Format:

40. //

41. //

42. //

Computation, Comparisons and Assignments:

43. OK

44. OK

45. OK

46. //

47. //

48. OK

49. OK

50. OK

Exceptions:

51. OK

52. OK

Flow of Control:

53. //

54. //

55. //

Files:

56. OK

57. OK

58. OK

59. OK

2.3.4 Method: writeContents

This is the method in charge to copy content from a source archive to a destination archive, both given as input values, together with the vector containing indications about the files to skip.

The method copies the Jar elements first, then writes the deployment descriptors and finally the manifest file.

The method is well-parenthesized (Kernigan and Ritchie style) and correctly indented.

All other standard rules are observed (correct naming of elements, indention, etc.) except for line length that in some cases exceeds the 80 characters and in those cases that can't be modified but the readability is not influenced and they don't exceed the 120 characters limit so it is fine to leave them as they are.

The comparisons are always made with “==” or “!=” but since they are comparisons to null pointers it do not violate the rule about using *equals* method because this is the only way to make them.

Now we will proceed with the checklist:

Naming Conventions:

1. OK
2. //
3. //
4. //
5. OK
6. OK

7. OK

Indentation:

8. OK

9. OK

Braces:

10. OK

11. OK

File Organisation:

12. OK

13. Some lines exceed 80 characters

14. OK

Wrapping Lines:

15. OK

16. OK

17. OK

Comments:

18. OK

19. OK

Java Source Files:

20. //

21. //

22. OK

23. OK

Package and Import Statement:

24. //

Class/Interface Declaration:

25. OK

26. OK

27. OK

Initialization and Declarations:

28. OK

29. OK

30. OK

31. OK

32. OK

33. OK

Method Calls:

34. OK

35. OK

36. OK

Arrays:

37. OK

38. OK

39. OK

Object Comparison:

40. OK (see explanation)

Output Format:

41. //

42. //

43. //

Computation, Comparisons and Assignments:

44. OK

45. OK

46. OK

47. //

48. //

49. OK

50. OK

51. OK

Exceptions:

52. OK

53. OK

Flow of Control:

54. //

55. //

56. //

Files:

57. OK

58. OK

59. OK

60. OK