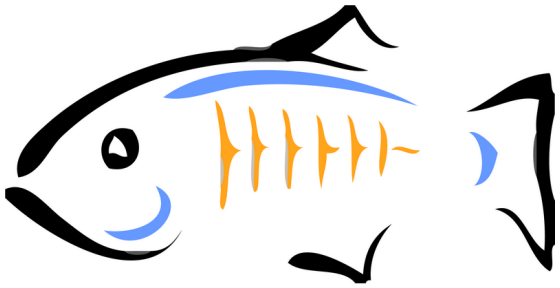

Software Engineering 2 Project, AA 2015-2016

Assignment 3: Code Inspection

Fabio Catania, Matteo Di Napoli, Nicola Di Nardo

January 4, 2016

Contents

1 Assigned Classes

1.1 SystemEnvironment

Location: appserver/registration/registration-impl/src/main/java/com/sun/ enterprise/registration/impl/environment/SystemEnvironment.java

Methods:

- *getSystemEnvironment( )*
- *getCommandOutput(String . . . command)*
- *getFileContent(String filename)*

1.2 LinuxSystemEnvironment

Location: appserver/registration/registration-impl/src/main/java/com/sun/ enterprise/registration/impl/environment/LinuxSystemEnvironment.java

Methods:

- *getLinuxCpuManufacturer( )*

1.3 WindowsSystemEnvironment

Location: appserver/registration/registration-impl/src/main/java/com/sun/ enterprise/registration/impl/environment/WindowsSystemEnvironment.java

Methods:

- *WindowsSystemEnvironment( )*
- *getWmicResult(String alias , String verb , String property)*
- *getFullWmicResult(String alias , String verb , String property)*

2 Functional Role

2.1 SystemEnvironment

SystemEnvironment class collects the environment data with the best effort from the underlying platform. It has a package-private constructor and provides externally a static factory-pattern method to dynamically initialize instances of its different subclasses depending on the type of Operating System.

It finally provides a set of getters and setters for the various SO properties inherited and used by its SO-specific subclasses and two methods to get command output and file content given specific paths as parameter.

2.2 LinuxSystemEnvironment

The class LinuxSystemEnvironment extends the superclass SystemEnvironment that belongs to the same package. It contains three static class variables representing, as an integer, different components of the system environment.

Besides, there is another instance variable called dmiInfo of type String in order to store information gathered through the method getLinuxDMIInfo. The class presents a single constructor and a group of 13 methods responsible of the interaction with the operating system in order to gather various information about it.

LinuxSystemEnvironment, in fact, aims to collect the environment data from the underlying Linux platform. To obtain that kind of information it is necessary a collaboration with the operating system and the file system. In order to allow applications to access OS and platform specific information, Linux environment has provided several files (example PSN) where these information can be found.

Conversely, when files are not useful the system environment provides the DMI framework. The Desktop Management Interface (DMI) generates a standard framework for managing and tracking information in a desktop, notebook and a server computer, by abstracting these components from the software that manages them.

In this code has been used the command dmidecode in order to obtain information. In particular, there is a method called getLinuxDMIInfo in charge of information gathering.

2.3 WindowsSystemEnvironment

The class *WindowsSystemEnvironment* extends the superclass *SystemEnvironment* that belongs to the same package. It contains the class variable *logger* of type *RegistrationLogger* that is used to signal that a problem has occurred. The class presents a single constructor and a group of six methods responsible of the interaction with the operating system.

WindowsSystemEnvironment aims to collect the environment data from the underlying Windows platform. To obtain that kind of information it is necessary a collaboration with the operating system.

In order to allow applications to access OS and platform specific information, Microsoft has provided a SLQ-Like API called WMI (Windows Management Instrumentation). WMI is provided as a default configuration within XP-Home and later operating systems. Windows Management Instrumentation Command-line (WMIC) extends WMI and uses its power to enable systems management from the command line. Both can be executed from many programming languages (including C/C++, .NET, Visual Basic, COM, C#, Java).

In this code has been used WMIC. In particular, in the constructor of the class are set these attributes: systemModel, systemManufacturer, cpuManufacturer, serialNumber, physmem, sockets, cores, virtcpus, cpuname and clockrate.

3 List of Issues

3.1 SystemEnvironment

3.1.1 Naming Conventions

1-2) *All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests. - If one-character variables are used, they are used only for temporary throwaway variables, such as those used in for loops.*

Some variable names are monosyllabic and not defining their usage in *getCommandOutput()* and *getFileContent()* methods.

Lines 342-349:

```
Process p = null;
ProcessStreamDrainer psd = null;
try {
    ProcessBuilder pb = new ProcessBuilder(command);
    p = pb.start();
    psd = ProcessStreamDrainer.save("RegEnvCommandProcess", p)
        ;
    return psd.getOutString();
}
```

Lines 371-377:

```
File f = new File(filename);
if (!f.exists()) {
    return "";
}
StringBuilder sb = new StringBuilder();
BufferedReader br = null;
```

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.*

Not all SystemEnvironment class attributes have the words following the first in the name of the variable capitalized: see *hostname, physmem, virtcpus, cpuname, clockrate.*

Lines 60-76:

```
public class SystemEnvironment {
    private String hostname;
    private String hostId;
    private String osName;
    private String osVersion;
```

```
private String osArchitecture;
private String systemModel;
private String systemManufacturer;
private String cpuManufacturer;
private String serialNumber;
private String physmem;
private String sockets;
private String cores;
private String virtcpus;
private String cpuname;
private String clockrate;
private static SystemEnvironment sysEnv = null;
```

3.1.2 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)*

General comments should be written before the package statement.

Lines: 42-50:

```
package com.sun.enterprise.registration.impl.environment;

// The Service Tags team maintains the latest version of the
    implementation
// for system environment data collection.  JDK will include a
     copy of
// the most recent released version for a JDK release.  We
    rename
// the package to com.sun.servicetag so that the Sun
    Connection
// product always uses the latest version from the com.sun.scn
    .servicetags
// package. JDK and users of the com.sun.servicetag API
// (e.g. NetBeans and SunStudio) will use the version in JDK.
```

3.1.3 Comments

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

All the non-public methods (*getCommandOutput()* and *getFileContent()*) are not documented neither provided with significant in-code comments. Their usage is quite straight-forward though.

3.1.4 Class and Interface Declarations

25) *Order of class or interface declarations*

Static variables should precede instance variables, while the static SysEnv instance is the last of the attributes (D, E points of checklist). Factory

static method *getSystemEnvironment()* precede the package-private class constructor *SystemEnvironment()* (F, G points of checklist).

27) *Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.*

getCommandOutput() method declares and uses a ProcessStreamDrainer instance that is an object belonging to another Glassfish package. This strengthens object coupling and weaken the object cohesion.

3.1.5 Object Comparison

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

sysEnv variable is compared to null with == and not with equals , but this is actually correct in case of to-null comparisons (NullPointerException expected otherwise).

Line 80:

```
if (sysEnv == null) {...
```

Line 353:

```
if (p != null) {...
```

Line 381:

```
while ((line = br.readLine()) != null) {...
```

Line 395:

```
if (br != null) {...
```

3.1.6 Output Format

42) *Check that error messages are comprehensive and provide guidance as to how to correct the problem*

Different exceptions are ignored without leaving error messages and don't provide guidance to error explanation. See point 53.

3.1.7 Exceptions

53) *Check that the appropriate action are taken for each catch block*

Relevant exceptions are caught but not dealt with in various points of getCommandOutput() and getFileContent() methods. They should instead at least be accompanied by an error message, preferably added to a logger.

Lines 344-366:

```
try {
    ProcessBuilder pb = new ProcessBuilder(command);
    p = pb.start();
    psd = ProcessStreamDrainer.save("RegEnvCommandProcess", p)
        ;
    return psd.getOutString();
} catch (Exception e) {
    // ignore exception
    return "";
} finally {
    if (p != null) {
        try {
            p.getErrorStream().close();
        } catch (IOException e) {
            // ignore
        }
        try {
            p.getInputStream().close();
        } catch (IOException e) {
            // ignore
        }
        p = null;
        }
    }
```

Lines 376-402:

```
StringBuilder sb = new StringBuilder();
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader(f));
    String line = null;
    while ((line = br.readLine()) != null) {
        line = line.trim();
        if (line.length() > 0) {
            if (sb.length() > 0) {
                sb.append("\n");
            }
            sb.append(line);
        }
    }
    return sb.toString();
} catch (Exception e) {
    // ignore exception
    return "";
```

```
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            // ignore
        }
    }
}
```

3.1.8 Files

58) *Check that all files are closed properly, even in the case of an error*

File f opened at line 371 of *getFileContent()* method is not explicitly closed, but his stream is automatically closed by closing the BufferedReader instance that manages the file input.

Line 371:

```
File f = new File(filename);
```

Line 379:

```
br = new BufferedReader(new FileReader(f));
```

Line 394-402:

```
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            // ignore
        }
    }
}
```

59) *Check that EOF conditions are detected and handled correctly*

EOF conditions are handled by BufferedReader *readline()* method, that is the method used to read from the input file. This method returns null when EOF is hit, so to handle EOF conditions is sufficient to handle the null case as has been done in the while loop.

Line 381:

```
while ((line = br.readLine()) != null) {...
```

60) *Check that all file exceptions are caught and dealt with accordingly*

To deal with input file getFileContent() method uses java abstract File object representation, that is independent from the actual SO and its personal representation of the string path. To assure the possibility of opening and using the file File object exists() method is used. If it returns false, then getFileContent() method returns without doing nothing.

All the exceptions that can rise at the moment of the file reading are instead not dealt with: they are all included in a generic catch (*Exception e*) block and just ignored.

Lines 371-374:

```
File f = new File(filename);
if (!f.exists()) {
    return "";
}
```

Lines 378-379, 391-394 (FileReader exceptions managing):

```
try {
    br = new BufferedReader(new FileReader(f));
    ...
} catch (Exception e) {
    // ignore exception
    return "";
}
```

3.2 LinuxSystemEnvironment

3.2.1 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*

   Local variable names not defining their usage.

   Lines: 93-96

```
String tmp = getLinuxPSNInfo(CPU);
if (tmp.length() > 0) {
    return tmp;
}
```

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*

   The attribute dmiInfo might begin with an underscore in order to distinguish it from local variables.

   Line: 72

```
private String dmiInfo = null;
```

3.2.2 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)*

   General comments should be written before the package statement.

   Lines: 41-49

```
package com.sun.enterprise.registration.impl.environment;

// The Service Tags team maintains the latest version of the
    implementation
// for system environment data collection.  JDK will include a
     copy of
// the most recent released version for a JDK release.  We
    rename
// the package to com.sun.servicetag so that the Sun
    Connection
// product always uses the latest version from the com.sun.scn
    .servicetags
// package. JDK and users of the com.sun.servicetag API
// (e.g. NetBeans and SunStudio) will use the version in JDK.
```

13) There is a comment that exceeds 80 characters but in certain cases it is allowed for practical use.

Line: 90

```
* @return The cpu manufacturer (an empty string if not found
    or an error occurred)
```

14) *When line length must exceed 80 characters, it does NOT exceed 120 characters*

The comment above does not exceed 120 characters.

3.2.3 Comments

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

At the beginning there is no general comment to explain what the class represents and how it works. A short description should be needed. Besides, comments on blocks of code are missing. It is not clear what the piece of code does.

Lines: 92-110 (the entire method)

```
private String getLinuxCpuManufacturer() {
    String tmp = getLinuxPSNInfo(CPU);
    if (tmp.length() > 0) {
        return tmp;
    }

    String contents = getFileContent("/proc/cpuinfo");
    for (String line : contents.split("\n")) {
        if (line.contains("vendor_id")) {
            String[] ss = line.split(":", 2);
            if (ss.length > 1) {
                return ss[1].trim();
            }
        }
    }

    // returns an empty string if it can't be found or an
        error happened
    return getLinuxDMIInfo("dmi_type_4", "manufacturer");
}
```

3.2.4 Java Source Files

23) *Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you)*

There is no javadoc for the considered class, nothing to check.

3.2.5 Class and Interface Declarations

25) *Order of class or interface declarations*

The order of components of the class doesn't respect the classical order, since constructor precedes instance and class variables.

Lines: 58-77

```
LinuxSystemEnvironment() {
    setHostId(getLinuxHostId());

     setSystemModel(getLinuxModel());
     setSystemManufacturer(getLinuxSystemManufacturer());
     setCpuManufacturer(getLinuxCpuManufacturer());
     setSerialNumber(getLinuxSN());
     setPhysMem(getLinuxPhysMem());
     setSockets(getLinuxSockets());
     setCores(getLinuxCores());
     setVirtCpus(getLinuxVirtCpus());
     setCpuName(getLinuxCpuName());
     setClockRate(getLinuxClockRate());
}

private String dmiInfo = null;

private static final int SN  = 1;
private static final int SYS = 2;
private static final int CPU = 3;
private static final int MODEL = 4;
```

3.2.6 Output Format

42) *Check that error messages are comprehensive and provide guidance as to how to correct the problem*

If an error during the search of information through the method getLinuxDMIInfo occurs, the considered method returns an empty string as in the case there is no manufacturer specified (and no error occurs).

Line: 109

```
// returns an empty string if it can't be found or an error
    happened
return getLinuxDMIInfo("dmi type 4", "manufacturer");
```

3.3 WindowsSystemEnvironment

3.3.1 Naming Conventions

1-2) *All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests. - If one-character variables are used, they are used only for temporary throwaway variables, such as those used in for loops.*

Variable names monosyllabic and not defining their usage

Lines: 80-83

```
if (procId != null) {
    String[] s = procId.split(",");
    cpuMfr = s[s.length - 1].trim();
}
```

Lines: 95-103

```
try {
    // look in the current working directory
    File f = new File("TempWmicBatchFile.bat");
    if (f.exists()) {
        boolean b = f.delete();
        if (!b)
            logger.finest("Could not delete" + f.
                getAbsolutePath());
    }
}
```

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

No constant is declared in this class.

To be precise, at lines 180 and 232 the method *write()* of the class BufferedWriter is used with the constant value *13* as parameter. But it is easy to notice that it is an arbitrary number, in fact its value is not important to determinate the behaviour of the program.

Lines 180-181:

```
bw = new BufferedWriter(new OutputStreamWriter(p.
    getOutputStream()));
bw.write(13);
```

Lines 231-233:

```
bw = new BufferedWriter(
    new OutputStreamWriter(p.getOutputStream()));
bw.write(13);
```

3.3.2 Braces

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

There is an if-statement that have only one instruction to execute not surrounded by curly braces.

Lines 100-101:

```
if (!b)
    logger.finest("Could not delete" + f.getAbsolutePath());
```

3.3.3 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).*

General comments should be written before the package statement.

Lines 42-50:

```
package com.sun.enterprise.registration.impl.environment;

// The Service Tags team maintains the latest version of the
    implementation
// for system environment data collection.  JDK will include a
     copy of
// the most recent released version for a JDK release.
    We rename
// the package to com.sun.servicetag so that the Sun
    Connection
// product always uses the latest version from the com.sun.scn
    .servicetags
// package. JDK and users of the com.sun.servicetag API
// (e.g. NetBeans and SunStudio) will use the version in JDK.
```

13) *Where practical, line length does not exceed 80 characters*

The conventional line length limit of 80 characters is exceeded at lines 72 (87 characters), 176 (96 characters), 180 (82 characters), 185 (84 characters), 222 (83 characters), 227 (96 characters), 238 (84 characters) and 245 (102 characters).

Line 72:

```
setSystemManufacturer(getWmicResult("computersystem", "get", "
    manufacturer"));
```

Line 176:

```
ProcessBuilder pb = new ProcessBuilder("cmd", "/C", "WMIC",
    alias, verb, property);
```

Line 180:

```
bw = new BufferedWriter(new OutputStreamWriter(p.
    getOutputStream()));
```

Line 185:

```
in = new BufferedReader(new InputStreamReader(p.getInputStream
    ()));
```

Line 222:

```
private String getFullWmicResult(String alias, String verb,
    String property) {
```

Line 227:

```
ProcessBuilder pb = new ProcessBuilder("cmd", "/C", "WMIC",
    alias, verb, property);
```

Line 238:

```
in = new BufferedReader(new InputStreamReader(p.getInputStream
    ()));
```

Line 245:

```
if (line.toLowerCase(Locale.US).indexOf(property.toLowerCase(
    Locale.US)) != -1) {
```

14) *When line length must exceed 80 characters, it does NOT exceed 120 characters*

As related, when line length exceed 80 characters, it does NOT exceed 120 characters. Consequently, it is not necessary to break that lines because they are readable enough.

3.3.4 Java Source Files

23) *Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you)*

There is no javadoc for the considered class, nothing to check.

3.3.5 Object Comparison

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

While == (and its counterpart !=) tests for reference equality, *.equals()* tests for logical equality. In the piece of code in case strings are compared with == instead of *.equals()* , but since this is happening only in case of to-null comparisons, its actually correct (*.equals()* would have caused NullPointerException).

Line 187 and 240:

```
while ((line = in.readLine()) != null) {...
```

3.3.6 Output Format

43) *Check that the output is formatted correctly in terms of line stepping and spacing*

A space is missed between Could not delete and the path of the file.

Line 101:

```
logger.finest("Could not delete" + f.getAbsolutePath());
```

3.3.7 Computation, Comparisons and Assignments

44) *Check that the implementation avoids brutish programming*

In order to avoid the so called brutish programming, numbers in the code must be substituted by constants and named constants are preferred to numeric literals. As just said at point 7, the number *13* used as parameter at lines 181 and 233 should be substituted by a named constant.

The other 17 tasks about brute force solutions are not necessary because the piece of code does not present critical situations.

There is only an unusual practice at the beginning of the constructor ignoring the first call result of the method getWmicResult but the reason is well explained by the comments above it.

Lines: 66-69

```
// run a call to make sure things are initialized
// ignore the first call result as the system may
// give inconsistent data on the first invocation ever
getWmicResult("computersystem", "get", "model");
```

50) *Check throw-catch expressions, and check that the error condition is actually legitimate*

In this class, in particular in the methods *getWmicResult()* and *getFullWmicResult()*, are used objects of type ProcessBuilder, BufferedReader and BufferedWriter. It is right to think that they could generate IOException and that is the reason of the try-catch expressions at the following lines.

Lines 175-200:

```
try {
    ProcessBuilder pb = new ProcessBuilder("cmd", "/C", "WMIC"
        , alias, verb, property);
    Process p = pb.start();
    // need this for executing windows commands (at least
    // needed for executing wmic command)
    bw = new BufferedWriter(new OutputStreamWriter(p.
        getOutputStream()));
    bw.write(13);

    p.waitFor();
    if (p.exitValue() == 0) {
        in = new BufferedReader(new InputStreamReader(p.
            getInputStream()));
        String line = null;
        while ((line = in.readLine()) != null) {
            line = line.trim();
            if (line.length() == 0) {
                continue;
            }
            res = line;
        }
        // return the *last* line read
        return res;
    }

} catch (Exception e) {
    // ignore the exception
}
```

Lines 226-254:

```
try {
    ProcessBuilder pb = new ProcessBuilder("cmd", "/C", "WMIC"
        , alias, verb, property);
    Process p = pb.start();
    // need this for executing windows commands (at least
    // needed for executing wmic command)
    bw = new BufferedWriter(
    new OutputStreamWriter(p.getOutputStream()));
    bw.write(13);
    bw.flush();

    p.waitFor();
    if (p.exitValue() == 0) {
```

```
            in = new BufferedReader(new InputStreamReader(p.
                getInputStream()));
            String line = null;
            while ((line = in.readLine()) != null) {
                line = line.trim();
                if (line.length() == 0) {
                    continue;
                }
                if (line.toLowerCase(Locale.US).indexOf(property.
                    toLowerCase(Locale.US)) != -1) {
                    continue;
                }
                res.append(line).append("\n");
            }
        }

    } catch (Exception e) {
    // ignore the exception
    }
```

3.3.8 Exceptions

53) *Check that the appropriate action are taken for each catch block*

Anytime an exception is caught it is ignored and it is bad. It is recommended at least to communicate that the error has occurred.

In addition to the above blocks,

Lines: 103-106 (here it's actually been logged a message but then no action is performed)

```
catch (Exception e) {
    logger.finest(e.getMessage());
    // ignore the exception
}
```

Lines 202-206:

```
try {
    in.close();
} catch (IOException e) {
    // ignore
}
```

Lines 209-212:

```
try {
    bw.flush();
} catch (IOException e) {     // ignore...
}
```

Lines 213-216:

```
try {
    bw.close();
} catch (IOException e) {    // ignore...
}
```

Lines 256-260:

```
try {
    in.close();
} catch (IOException e) {
    // ignore
}
```

Lines 264-268:

```
try {
    bw.close();
} catch (IOException e) {
    // ignore
}
```