Assigment 1

Allan Sousa | 201800149 Breno Pimentel | 201800170

1 Project Description

The project implements a graphical interface to manage and visualize the **todo.txt** file format. The code is divided in two main paths, one for the GUI related code: `chschmid/jdotxt`, and the other for the main program logic needed, such as task filters and parsers: `todotxt/todotxttouch`. It also provides a test suite which validates the most important functionalities.

2 Static Testing

Static testing is a method which checks a program for potential defects or standard deviation without actually executing it. It can be performed both manually or by using automated tools. Nowadays, developers take it for granted (for the most part) because compilers and IDEs already performs static checking of our code (e.g. IDEs and compilers will detect wrong return types without actually executing the code). This, however, does not decrease the importance of static checking, and it does not mean that compilers and IDEs catch everything. OOP design flaws and coding-standard deviations are examples of what kind of stuff that static checkers can detect. It is also important because it can catch a lot of bugs before execution, thus decreasing overall debugging time, while also facilitating work between multiple developers by enforcing standards.

3 Static Testing Tools

In this project, we've selected two static tools to implement:

Checkstyle

Checkstyle is a best practices and coding standard enforcing tool. While it can check if the code follows a previously defined set of rules regarding naming conventions and formatting, it can also check for OOP design flaws (at least the flaws from a more theoretical point of view). For example, it can check for Extension classes and for Interfaces that do not describe a type (do not contain any methods, just constants). In the context of our assigned project (jdotxt), we set up the rules as follows:

- Firstly, we used Google's default ruleset as a reference, since a considerable amount of the indentation was already following some of the ruleset's defaults.
- Then, we proceeded to do minor adjustments that we thought would make sense:
    - We deactivated the `VariableDeclarationUsageDistance` rule, since declaring all variables in the beginning of the scope is a matter of style, and should not present a problem if used by all contributors.

SpotBugs

Spotbugs is a static analysis tool used to look for bugs in Java applications. It is able to check for more than 400 bug patterns divided in different priorities and categories, in order to perform these checks, it uses multiple detectors which are specialized in finding a type of bug. In its configuration, for instance, we can opt for the level of confidence (priority) in a bug that should be reported by the tool or the type of bug to look for. The setup of this tool is as follows:

- We used all the detectors that are on by default;
- Both High and Medium priority levels were set;
- We removed the check of bugs related to the STYLE category, as these are already checked more efficiently by the *Checkstyle* tool also used.

```
<Match>
    <Not>
        <Confidence value="3"/>
    </Not>
    <Not>
        <Bug category="STYLE" />
    </Not>
</Match>
```

4 Reports

Checkstyle

Running the tool (with the previously defined setup) yields 14117 warnings across 68 files and 0 errors. Most of which are related to naming/comments convention and whitespaces being used inconsistently. There are, however, 4 warnings related to possible coding malpractices.

SpotBugs

The tool reported a total of 74 bugs, most of these were in the category of malicious code, such as mutable attributes returned in getter methods, thus exposing these. A minor part was related to bad practices and performance issues.

5 Bugs Sample

Checkstyle

The 5 "bugs" that we selected to fix are :

- **MissingSwitchDefault**: The default switch clause is missing.
- **MultipleVariableDeclarations**: There are multiple variables declarations on the same line.
- **OneStatementPerLine**: There are multiple statements on the same line.
- **FileTabCharacter**: Tabs are used for indentation project-wise. This is not appropriated for multiple developers, since it brings formatting issues.
- **UpperEll**: There are variables of the `long` type being defined with a lower case 'l'. It is a minor "offense", but the Java specification defines it as the go to, since a lower-case 'L' looks like a '1'.

SpotBugs

The issues selected to be addressed were:

- **MS_SHOULD_BE_FINAL**: A variable isn't final, but should be.
- **EI_EXPOSE_REP**: The return of a private and mutable attribute of an object can expose it.
- **DM_DEFAULT_ENCODING**: This issue is raised when a byte to String conversion is made without specifying an encoding format.
- **NP_NULL_ON_SOME_PATH**: Possible null pointer dereference.
- **RV_RETURN_VALUE_IGNORED_BAD_PRACTICE**: The return value of a method is ignored.

6 Fixes

Checkstyle

The 5 "bugs" that we selected to fix were fixed as follows :

- **MissingSwitchDefault**: The default switch clauses were added.
  - Before:

```
switch (mode) {
    case PROJECT: return Jdotxt.taskBag.getProjects(false);
    case CONTEXT: return Jdotxt.taskBag.getContexts(false);
}
```

  - After:

```
switch (mode) {
    case PROJECT: return Jdotxt.taskBag.getProjects(false);
    case CONTEXT: return Jdotxt.taskBag.getContexts(false);
    default: break;
}
```

- **MultipleVariableDeclarations**: Variable declarations were divided into multiple lines.
  - Before:

```
private boolean reloadDialogVisible, unresolvedFileModification;
```

  - After:

```
private boolean reloadDialogVisible;
private boolean unresolvedFileModification;
```

- **OneStatementPerLine**: Multiple statements in a line were divided into multiple lines.
  - Before:

```
toolbar.getButtonSave().addActionListener(new ActionListener() {public void actionPerformed(ActionEvent arg0) {saveTasks(
```

  - After:

```
toolbar.getButtonSave().addActionListener(
                new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                saveTasks(false);
                        }
                }
                );
```

- **FileTabCharacter**: We replaced all tabs used for indentation by 4 spaces.
- **UpperEll**: Lower-case 'L's were substituted by upper-case.
    - Before:

```
long date_ms = 0l;
```

    - After:

```
long date_ms = 0L;
```

SpotBugs

- **MS_SHOULD_BE_FINAL**: The fix for this issue was to add the `final` keyword in the declaration.
    - Before:

```
public static Color COLOR_GRAY_PANEL = new Color(221, 221, 221);
```

    - After:

```
public final static Color COLOR_GRAY_PANEL = new Color(221, 221, 221);
```

- **EI_EXPOSE_REP**: The fix for this issue was to return a copy of the mutable object instead of itself. In some cases, such as the Java Swing components, we had to create a method to perform a deep copy of it.
    - Before:

```
public Date getDueDate() {
    return dueDate;
}
...
public JButton getSaveButton() {
    return saveButton;
}
```

    - After:

```
public Date getDueDate() {
    return new Date(dueDate.getTime());
}
...
public JButton getSaveButton() {
    return (JButton) Util.cloneObject(saveButton);
}
```

- **DM_DEFAULT_ENCODING**: The enconding format was specified when converting bytes to a string.
    - Before:

```
  int c;
  byte[] buffer = new byte[8192];
  StringBuilder sb = new StringBuilder();
  while ((c = is.read(buffer)) != -1) {
        sb.append(new String(buffer, 0, c));
  }
```

- After:

```
byte[] buffer = new byte[8192];
StringBuilder sb = new StringBuilder();
while (is.read(buffer) != -1) {
	sb.append(
		StandardCharsets.UTF_8.decode(ByteBuffer.wrap(buffer))
	);
}
```

- **NP_NULL_ON_SOME_PATH**: In this case, the object `dir` could have a null value which could throw a `NullPointerException` in the second *if* statement.
  - Before:

```
if (dir!=null && !dir.exists()) {
	...
}
if (!dir.exists()) {
	if (!dir.mkdirs()) {
		...
	}
}
```

  - After:

```
if(dir == null) return;
if (!dir.exists()) {
	...
}
if (!dir.exists()) {
	if (!dir.mkdirs()) {
		...
	}
}
```

- **RV_RETURN_VALUE_IGNORED_BAD_PRACTICE**: The return value of `createNewFile` was checked and raises an adequate exception.
  - Before:

```
try {
	if (!TODO_TXT_FILE.exists()) {
		Util.createParentDirectory(TODO_TXT_FILE);
		TODO_TXT_FILE.createNewFile();
	}
}
catch (IOException e) {
	throw new TodoException("Error initializing LocalFile", e);
}
```

  - After:

```
try {
	if (!TODO_TXT_FILE.exists()) {
		Util.createParentDirectory(TODO_TXT_FILE);
		if(!TODO_TXT_FILE.createNewFile()) throw new FileAlreadyExistsException("File already exists");
	}
} catch (FileAlreadyExistsException ignored) {}
catch (IOException e) {
	throw new TodoException("Error initializing LocalFile", e);
}
```

7 Test Reports After Fixes

Checkstyle

After correcting the warnings mentioned above, the new report yields 1570 warnings across 67 files and 0 errors. It is worth mentioning that, again, most of the errors are indentation errors that could be (and will be) fixed with regex, but we decided to focus on malpractice warnings to document in this file, as these warnings are more related to code clarity and readability than style. [e.g. multiple statements per line are very confusing in large codebases]

SpotBugs

After addressing the category of bugs mentioned, the *Spotbugs* report presented 34 bugs, all High priority ones were identified and fixed. The remaining were all of Medium priority.