Code Quality Report for Double Black Diamond ERP
SOEN 390 - Team #12

Tyler Znoj 4005987
Killian Kelly 40014508
Abdul Sirawan 40074202
Adrien Kamran 40095393
Qandeel Arshad 40041524
Cedrik Dubois 40064475
Anthony Chraim 40091014
Matthew Giancola 40019131

Table of Contents

1 - Methodology

1.1 - What did we analyze?

Our code base is built on a number of frameworks and languages, primarily Python through Django. However, as this is a web app, front end languages and frameworks are also used, such as HTML/JS/CSS through Bootstrap and SQL through Ajax/jQuery.

As the code volume for these last elements is relatively low (limited Ajax/jQuery calls) and already standardized (Bootstrap Studio exports clean code), we decided to focus our analysis on our Python code.

1.2 - How did we analyze?

We chose to analyze our Python code through a static analysis tool by the name of PyLint. Once integrated as part of the requirements of our Docker container, we were able to run PyLint through the container's command line as follows:

pylint --output-format=json accounting ERP dashboard inventory manufacturing notifications sales vendors > code-quality-report.json

PyLint ran through each module of our Django project (accounting, dashboard, etc.) and found convention errors, warnings, and functional errors according to the following structure:

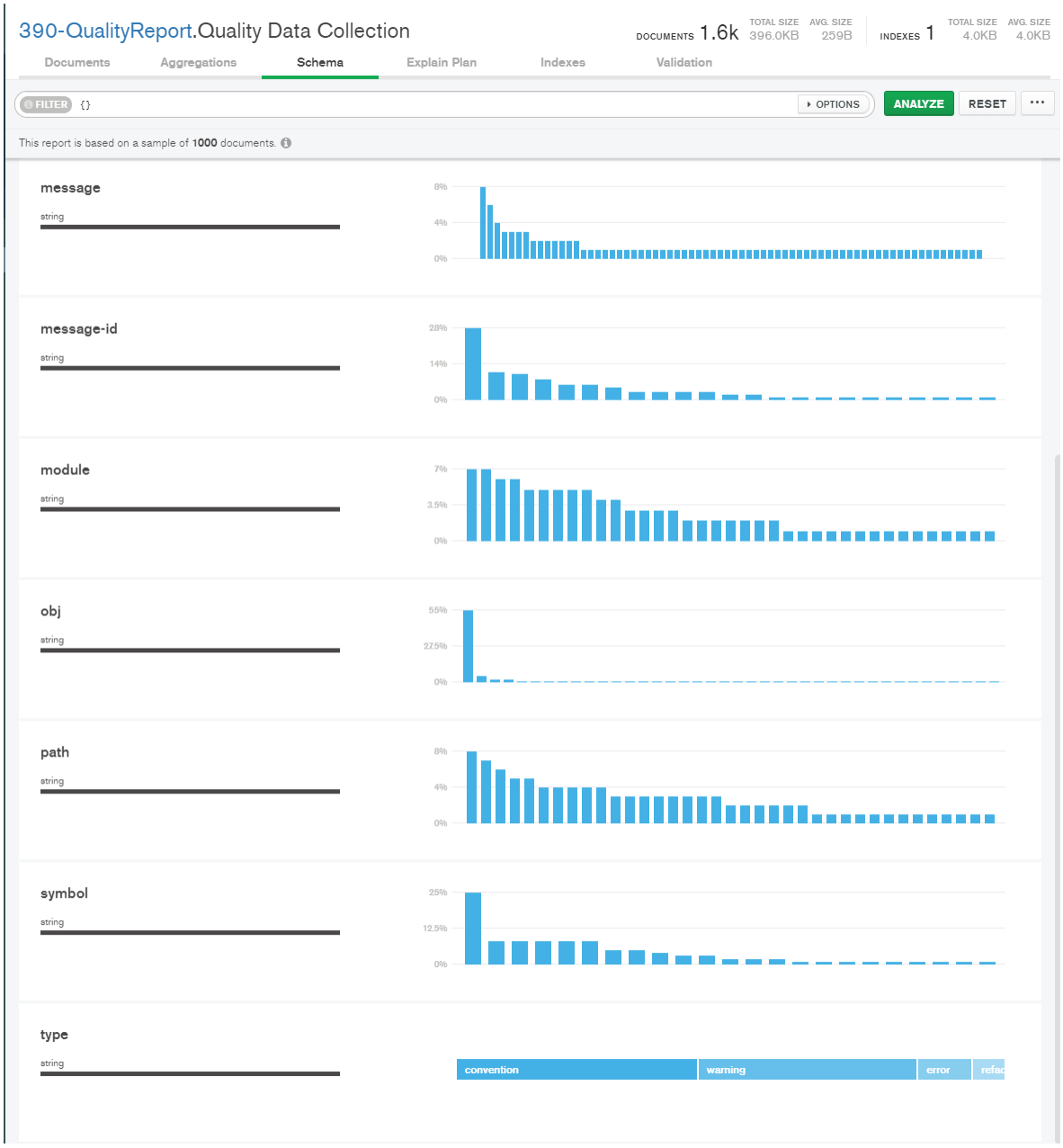

```
_id: ObjectId("606e53821c80c03c40221a4b")
type: "convention"
module: "accounting.admin"
obj: ""
line: 1
column: 0
path: "accounting/admin.py"
symbol: "missing-module-docstring"
message: "Missing module docstring"
message-id: "C0114"
```


1.3 - How did we parse the results?

As the results were exported to a json file, it was possible to import the data to a MongoDB collection, then perform an analysis using MongoDB Compass.

2 - Results

2.1 - Overview



PyLint's output was a file with 1.6 thousand json documents, as noted above. From a sample of the first 1000 documents, 49% related to PEP8 convention errors, 41% related to general Python warnings, 5% related to Python errors, and 5% related to non-essential refactoring suggestions.

2.2 - Analysis by type

In order to better understand which errors are the most frequent, we filtered the results by type and collected those which represented 5% or more of the errors within each type.

2.2.1 - Convention

Convention errors relate exclusively to the PEP8 style guide. If a line of code does not respect the style guide, the error's type will be listed under "Convention", and by its nature, does not affect the performance of the system in any way.

Here are some insights into these errors:

- Line exceeds the standardized size (25%)
- Function is missing a docstring (19%)
- Object/function name does not respect naming standard (17%)
- Class is missing a docstring (13%)
- Trailing whitespace (12%)
- Module is missing a docstring (8%)

2.2.2 - Warning

Warnings are raised whenever a non-fatal error is present. Enough warnings could cause performance issues in the long run, as they often relate to unused imports and useless/unreachable code.

Here are some insights into these errors:

- Unused wildcard import (59%)
- Unused import (14%)
- Wildcard import (9%)
- Deprecated method (7%)

2.2.3 - Error

Errors are legitimate and possibly fatal faults in the code. In our case, over 94% of the errors caught by PyLint were false positives triggered by Django (specifically, "[x] has no [y] member").

2.2.4 - Refactor

Refactoring suggestions are hints given by PyLint on changes that could be made to improve the code quality that do not originate from an error.

Here are some insights into these suggestions:

- Unnecessary "else" after "return" (19%)
- Too few public methods (19%)
- Either all return statements in a function should return an expression, or none at all (9%)

3 - Conclusions

Given the results of our analysis, it would seem that the majority of issues in our code stem from PEP8 style violations and unused imports. Both of these types of errors can be fixed with the use of built-in reformatting features found in certain IDEs, such as Pycharm.

Otherwise, it seems like the quality of our code is generally quite high, given the high test coverage and lack of bugs. As is the case with any good deployment, it would be worth returning to PyLint with every sprint to perform a new analysis and refactor where necessary.