CPSC 410 - Assignment 4

Exercise 1

1. It is static program analysis and value-agnostic. The value of string does not influence the checker to distinguish between `===` and `.equals()`. The smallest program unit is `Operator/Keyword`.
2. It is dynamic program analysis. The smallest program unit is `Class/File`.
3. It is static program analysis and value-agnostic. The return value does not influence the checker to count the return statements. The smallest program unit is `Method/Function`.
4. It is raw-data analysis. The smallest program unit is `Project/Executable`.
5. It is static program analysis and value-sensitive. The checker needs to compare the old and the new values of variables. The smallest program unit here is `Class/File`.
6. It is raw-data analysis and the smallest program unit is `Class/File`.

Exercise 2

Naming conventions

- Category: raw-data analysis
- Usage: The module warns the programmers about bad naming in their code which may cause confusion or potential bugs.

Bitwise Operation issues

- Category: static program analysis.
- Usage: The module inspects the bit-wise operation in the code and report potential code errors or unnecessary operations.

Dependency issues

- Category: meta-property analysis.
- Usage: The module warn the programmers about the potential dependency issues such as unused dependencies, cyclic dependencies or too many transitive dependencies.

Run Main with coverage

- Category: dynamic program analysis
- Description: The module execute the code and reports the code coverage to the programmers (how many lines are covered in this executions).

Exercise 3

- Briefly summarize your results up to this point (i.e. which project you picked, how many issues the analyses found and if there are any noteworthy observations).

> I picked fastjson, a Java project that can be used to convert Java Objects into their JSON representations. The inspection found 300713 warnings, 74 weak warnings and 837 errors. Most of these are related to code style issues.

- Pick 3 results found by the analyses you find interesting, and inspect them more closely. For each of them, briefly describe what the "issue" found by the analysis is.

> Issue 1: Feature Envy. Description: Feature envy is defined as occurring when a method calls methods on another class three or more times. Calls to library classes, parent classes, contained or containing classes are not counted for purposes of this inspection. Feature envy is often an indication that functionality is located in the wrong class. 1028 warnings are found in this project. I found them interesting because sometimes it is unavoidable to call methods in other class, but such practice is considered as a warning by the analyzer.

> Issue 2: Magic Number. Description: magic numbers are numeric literals used without being named by a constant declaration. Magic numbers can result in code whose intention is unclear, and may result in errors if a magic number is changed in one code location but not another. I found it interesting as my project is relatively large (it took me 40 mins to run the inspection). However, there are 4340 warnings of magic number. If any bugs cause because of it, these bugs would be very difficult to detect.

> Issue 3: Duplicated code fragment. Description: duplicated blocks of code from the selected scope: the same file, same module, dependent modules, or the entire project. I found it interesting as avoiding duplicated code is a common sense for a programmer. The warning appears 228 times for a trending project is funny.

- For each of the previously described results, explain whether you think that fixing it would make your (or the author's) code better, and if you think working on this issue would be "time well spent" (i.e. if you think the effort of fixing the code would pay off in the long run).

> Issue 1: In most cases, fixing the feature envy would reduce the coupling between classes and locate the methods in the appropriate classes. If such fixing would not take a lot of time and effort, I would suggest doing so as it may benefit the programmers in the long run.

> Issue 2: I highly recommend fixing magic numbers in the project. Magic number could cause bugs easily in the code. For example, if we use a same number multiple times in the code and we would change the value of them, we need to manually check every occurrence of the number. If we define the number as a constant, we could just change the value of the constant. It will cause less bugs in the long run.

> Issue 3: I highly recommend fixing this program. Duplicated fragment indicates highly coupling between classes. It will easily cause bugs in the project. Fixing such problems may require some works but it will reduce the coupling and improve the abstraction of the code in the long run.

- Based on your findings, briefly summarize (in 3-4 sentences) what advice would you give a friend or colleague that is interested in using static analysis? Can you recommend using the approach you took for this exercise? If not, can you think of a way to use static analyses more effectively?

> I think static analysis could provide some insights about the projects that we are working on. For examples, the issues such as feature envy and magic number detected by static analysis as I mentioned above can reduce the possibility of bugs. However, in real coding practice, the programmers should not only rely on the analysis. Instead, they should write tests to test their code. Also, I found running the inspection GUI in Intellij for large project is time consuming and demanding for computer hardwares (my project took me 40 mins to run the inspection, and it warned multiple times for low memory).