Lines of code metrics

The collected metrics were the number of lines of code in our project, along with information derived from those lines.

There were 7 different types of metrics collected:

* Interface metrics:
  + Lines of code (LOC), which include comments, documentation and code.
  + Comment lines of code (CLOC), which only include comments.
  + Javadoc lines of code (JLOC), which only include javadoc lines.
  + Non-comment lines of code (NCLOC), which only include non-commented lines of code.
* Package metrics:
  + CLOC
  + CLOC(rec)
  + JLOC
  + JLOC(rec)
  + LOC
  + LOC(rec)
  + LOCt
  + LOCt(rec)
  + NCLOC
  + NCLOCp
  + NCLOCp(rec)  
    NCLOCt
  + NCLOCt(rec)

In the above metrics, (rec) indicates that the counting was done recursively, t indicates that the lines of code counted were test code and p indicates that the lines counted were product code.

* Module metrics, which calculate the same data as above, for each module, with the added distinction of counting the lines of code of each programming language separately.
* File type metrics, which calculate the LOC and NCLOC for each file type in the project.
* Project metrics, which calculate the same data as the module metrics, but for the whole project instead.
* Class metrics, which calculate the CLOC, JLOC and LOC for each class.
* Method metrics, which calculate the CLOC, JLOC, LOC. NCLOC and RLOC metrics for each method. The RLOC metric is the relation between the lines of code of the method and the total lines of code in the class where the method is located.

Potential trouble spots

In the collected metrics, we can observe some potential trouble spots, namely in the method and class metrics. For example, by analyzing the lines of code in our classes, we can see that the InGameController holds the most lines of code, possibly indicating some issues like, for example, code that isn’t distributed correctly and that should be in another class. Another way we can find potential trouble spots is by analyzing methods. If we check the methods that have the highest amount of comment and documentation lines, we’ll usually arrive at methods that are too complex and confusing, requiring the use of many comments in order to understand what is happening. This can be observed in the assignWorkers method in the ColonyPlan class and in the travelToTarget method in the Mission class. Furthermore, we can also take a look at the relative lines of code in order to find potential trouble methods and classes. For example, the method generateAttackResult occupies the entirety of the HitpointsCombatModel class, and has a considerable size. This could indicate that this class is not entirely necessary, or that the method can be subdivided into smaller, less complex methods in order to increase readability.

Relation to code smells

The code smells identified were located with the help of these metrics. The CombatResult class is a data class that was found using the lines of code metrics, by searching for classes with very little lines of code. The csCombat was found by searching through the methods with the most lines of code, and the assignWorkers method was found by searching through the methods with the most comment lines.

Tiago Sousa, 63324