NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Department of Computer Science

NOVA School of Science and Technology

Metrics:

Lines of Code

Guilherme Fernandes

60173

P6

December 2022

1. Lines of Code Metric:

Lines of code (or LOC) is a metric that analyses the number and type of the lines each section of a project has. At first it seems to be a simple metric (and it really is) but can give out valuable information to improve our code.

One thing to take into consideration is that more lines doesn't mean better code, it can mean the opposite. A good developer might need 10 lines to write the same script that a bad developer needed 50 for.

1.1. **LOC** (Lines of code):

Description: Number of lines of code.

1.2. **CLOC** (Comment lines of code):

Description: Number of lines of comments.

1.3. **JLOC** (Javadoc lines of code):

Description: Number of lines of documentation.

1.4. **NCLOC** (Non-comment lines of code):

Description: Number of lines which aren't comments.

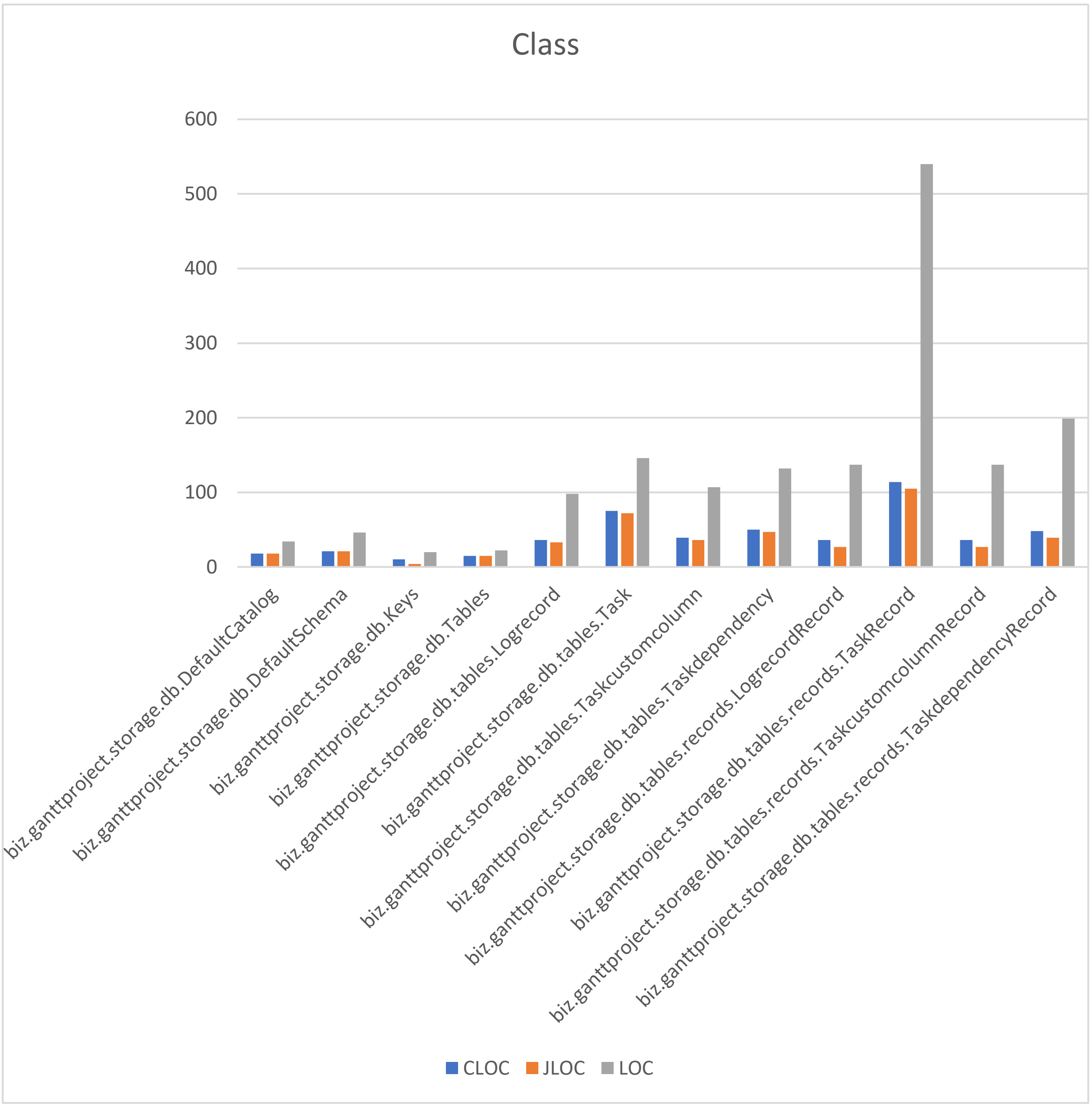
1.5. **RLOC** (Relative lines of code):

Description: Percentage of lines of code.

2. Potential trouble spots in the code:

2.1. Data analysis:

Description: After retrieving the metrics from the IDE and exporting them to an excel sheet, I converted the data into bar charts to easily pinpoint problems associated with extreme values. I tried to insert a table using the metrics for all the methods but found that the one for the classes was easier to build/analyse.



After a quick glance at the chart, we can see that there is a value that's skyrocketing when compared to the others – LOC of the class "TaskRecord". This means that this class has almost **4 times** the number of lines of code (540) than the average (134).

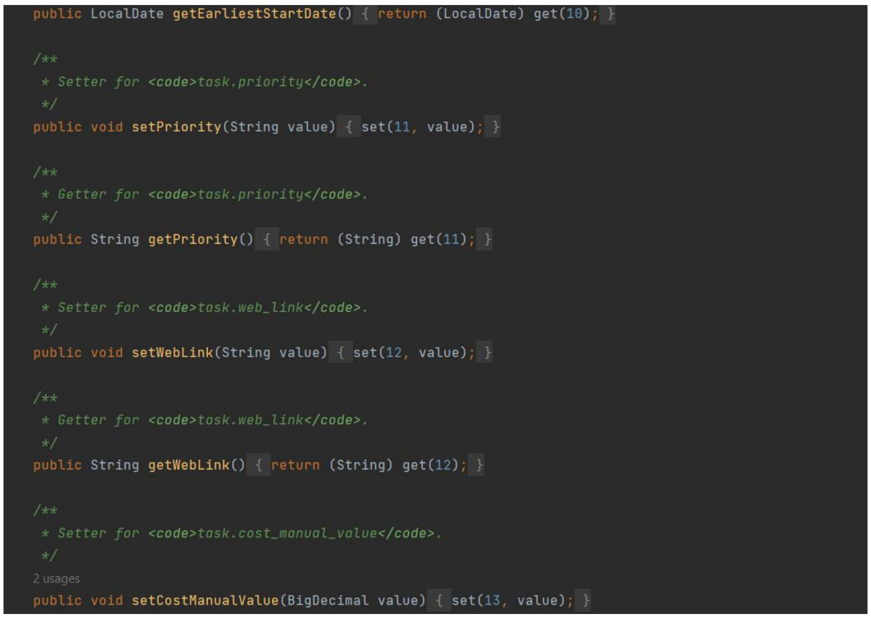
Location of the file: biz/ganttproject/storage/db/tables/records/TaskRecord.java

2.2. Code analysis:

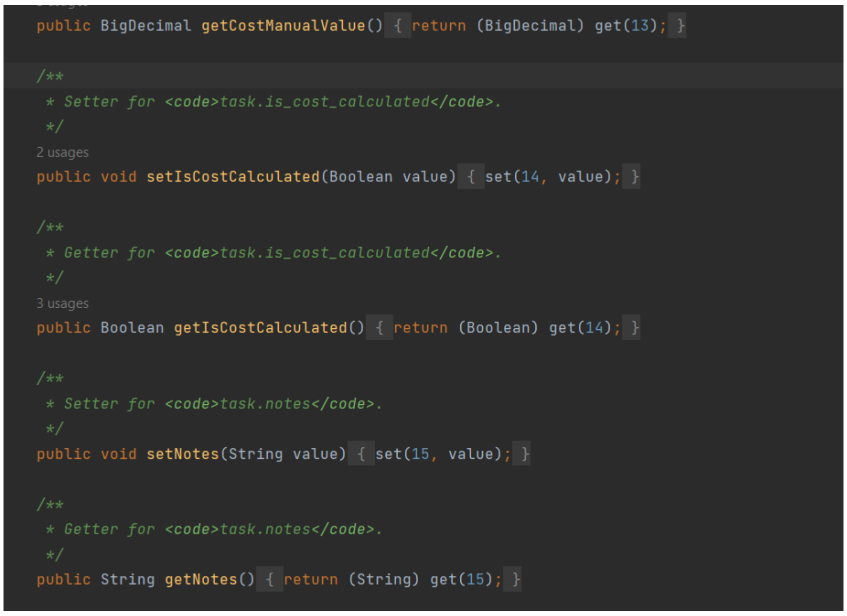
When looking into the problematic class, I realized that several code smells were present, here are some of them:

- **Large class** – This one was the easier to spot and probably has the easiest fix. I'd recommend splitting this class into several others to delegate tasks between them and to avoid making one class do all the work.

- **Too many getters and setters** – In this class there are around 15 getters and 15 setters that represent a bad practice of coding. As said by **Maximiliano Contieri** in his blog "*Getters and Setters are a poorly established practice. Instead of focusing on object behaviour (essential), we are desperate to know object guts (accidental) and violate their implementation.* "- https://blog.devgenius.io/code-smell-68-getters-68571a0f7fa8
  There are several fixes to this problem, one being to shift all this getters to a single method that would return the contents of the original object, avoiding this repetition (Boilerplate code).



```
public LocalDate getEarliestStartDate() { return (LocalDate) get(10); }

/**
 * Setter for <code>task.priority</code>.
 */
public void setPriority(String value) { set(11, value); }

/**
 * Getter for <code>task.priority</code>.
 */
public String getPriority() { return (String) get(11); }

/**
 * Setter for <code>task.web_link</code>.
 */
public void setWebLink(String value) { set(12, value); }

/**
 * Getter for <code>task.web_link</code>.
 */
public String getWebLink() { return (String) get(12); }

/**
 * Setter for <code>task.cost_manual_value</code>.
 */
2 usages
public void setCostManualValue(BigDecimal value) { set(13, value); }
```



```
public BigDecimal getCostManualValue() { return (BigDecimal) get(13); }

/**
 * Setter for <code>task.is_cost_calculated</code>.
 */
2 usages
public void setIsCostCalculated(Boolean value) { set(14, value); }

/**
 * Getter for <code>task.is_cost_calculated</code>.
 */
3 usages
public Boolean getIsCostCalculated() { return (Boolean) get(14); }

/**
 * Setter for <code>task.notes</code>.
 */
public void setNotes(String value) { set(15, value); }

/**
 * Getter for <code>task.notes</code>.
 */
public String getNotes() { return (String) get(15); }
```