MOOD Metrics

The MOOD metrics set works on a system or subsystem (“Collection of classes organized in some way to offer a given functionality as a whole.”, as described in the paper *The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics*) level to provide an analysis of the most basic mechanisms found in object-oriented programming. This set is comprised of 6 metrics: AHF (Attribute Hiding Factor), AIF (Attribute Inheritance Factor), CF (Coupling Factor), MHF (Method Hiding Factor), MIF (Method Inheritance Factor) and PF (Polymorphism Factor). Each of these will be discussed and analyzed in the context of this project as a whole, and in the JabRef.main module. The analysis relating to the full project will provide a complete overview of our code, while the analysis of the main module – which contains the JabRef subsystem – should provide a more interesting overview for the purposes of this course’s project.

It is relevant to mention that each metric is presented in the form of a percentage, with the value representing the number of times a certain mechanism is used divided by the number of times it could have possibly been used in the code.

AHF – Attribute Hiding Factor

The AHF provides a percentage of variable encapsulation in a system. The percentage is given by the sum of (1-V(a)) for each class in the system, divided by the sum of every field in every class of the system.

V(a) represents the visibility of a given attribute a, i.e., the number of classes a is visible in, divided by the number of classes in the system (the origin class is excluded from this formula).

This can be represented by the formula below:

The percentages found for the Attribute Hiding Factor go as follows:

|  |  |
| --- | --- |
| Project | AHF |
| JabRef | 78.33% |

|  |  |
| --- | --- |
| Module | AHF |
| main | 75.7% |

These values indicate a variable encapsulation factor of over 75%.

AIF – Attribute Inheritance Factor

As the name suggests, the AIF metric presents a percentage of attribute inheritance in the system. This value is calculated (for each class in the system) as the number of inherited (and not overridden) attributes in a given class, divided by the sum of all available attributes in said class.

The formula for this can be written as:

The percentages found for the Attribute Inheritance Factor go as follows:

|  |  |
| --- | --- |
| Project | AIF |
| JabRef | 23.20% |

|  |  |
| --- | --- |
| Module | AIF |
| main | 25.70% |

These values indicate that, across every class analyzed, the percentage of inherited variables on average rounds the 25% of all available variables.

CF – Coupling Factor

In this metric, coupling is defined as *non-inheritance references* to other classes. These references can be attribute types, method argument types, return values or calls to methods belonging to the other class.

With that in mind, the coupling factor percentage is given by the sum for every class in the system of the total references to other classes (total sum of couplings in the system) divided by the highest possible number of couplings in the system.

A formula for this can be written as:

The percentages found for the Coupling Factor go as follows:

|  |  |
| --- | --- |
| Project | CF |
| JabRef | 0.69% |

|  |  |
| --- | --- |
| Module | CF |
| main | 1.01% |

The values found suggest that both the project as a whole, as well as the main module we’re analyzing have very low coupling ratios. This is a desirable quality as high coupling is usually associated with code smells and poor programming.

MHF – Method Hiding Factor

Similarly to the AHF metric, the MHF metric provides a percentage of method encapsulation in the system. This gives us an idea of how many classes a given method might be visible from, on average.

Much like in the AHF metric, this percentage is given by the sum of (1-V(m)) for each class in the system, divided by the sum of every method in every class of the system.

V(m) represents the visibility of a given method m, i.e., the number of classes m is available in, divided by the number of classes in the system (the origin class of the method is excluded from this formula).

The formula for this metric can be written as the following:

The percentages found for the Method Hiding Factor go as follows:

|  |  |
| --- | --- |
| Project | MHF |
| JabRef | 36.93% |

|  |  |
| --- | --- |
| Module | MHF |
| main | 27.28% |

These values indicate a method encapsulation factor of around 30% in our considered systems.

MIF – Method Inheritance Factor

Once again, similarly to the AIF metric, the MIF metric gives us the percentage of method inheritance in the system.

This value is calculated (for each class in the system) as the number of inherited (and not overridden) methods in a given class, divided by the sum of all available methods in said class.

The formula for this can be written as:

The percentages found for the Method Inheritance Factor go as follows:

|  |  |
| --- | --- |
| Project | MIF |
| JabRef | 18.3% |

|  |  |
| --- | --- |
| Module | MIF |
| main | 23.14% |

From these values we can say that, across the entire system analyzed, we will find an average of 20% of inherited methods from all of the methods available in a given class.

PF – Polymorphism Factor

The Polymorphism Factor metric studies the ratio of existing polymorphic features to potential polymorphic features across all the classes in a system. A polymorphic feature is defined here as a method in a given class which can possibly be overridden by descendants of that class.

Before moving on into the formula, it’s important to establish the following convention. A class can be composed of 2 different kinds of methods: methods overridden from ascendant classes, and new methods firstly defined in that class. The first kind will be represented in this metric’s formula as *Mo* , while the second will be represented as *Mn*.

Now we can translate this metric into the formula presented below:

The numerator of this fraction will account for the total sum of overridden methods in our system, while the denominator will account for all the possible overridden methods for each class in the system. This last value would represent a scenario in which every single new method defined in a given class C is overridden in its descendants.

The percentages found for the Polymorphism Factor go as follows:

|  |  |
| --- | --- |
| Project | PF |
| JabRef | 49.81% |

|  |  |
| --- | --- |
| Module | PF |
| main | 49.47% |

From this we can gather that around half of all newly defined methods in an average class will be overridden in descendant classes.

Trouble Spots & Identified Code Smells

It’s important to mention first that the MOOD metrics present only a dimensionless project overview of the usage of core concepts of object-oriented programming. As such, the act of finding “trouble spots” throughout the code is not facilitated by this tool, as the smallest unit we can analyze consists of an entire module.

With this in mind, it is however possible to analyze and discuss the values found for each of the metrics, and what they might mean in the context of our codebase. In this approach I will compare the values collected from our project with possible extreme cases.

AHF – Attribute Hiding Factor

Related to the encapsulation principle in object-oriented programming.

Lowest extreme: with a 0% AHF we would be looking at a codebase where each attribute in each class would be visible in every other class in the system. This is the least desirable scenario we could find in a program, as it enables situations such as attributes being modified unexpectedly by any part of our system, or any object having full access to the internal state of another.

Highest extreme: with a 100% AHF we would find ourselves with a program where any given field can only be directly accessed by the class owning it. This would be a near ideal scenario, where no fields are ever exposed directly, and there is a specific way that determines how they are accessed.

Our values:

|  |  |
| --- | --- |
| Project | AHF |
| JabRef | 78.33% |

|  |  |
| --- | --- |
| Module | AHF |
| main | 75.7% |

As seen above, the values for this project are closer to the highest extreme, which is beneficial in terms of code quality for the reasons mentioned previously.

AIF – Attribute Inheritance Factor

Related to the inheritance principle in object-oriented programming.

Lowest extreme: with a 0% AIF none of the classes in the system would have any inherited attributes in them. From this we can conclude that this system would be lacking in inheritance features, which are crucial in an object-oriented paradigm. Of course, each system will be organized differently, however a large project based on OOP with no inheritance at all can hardly be a well-structured one.

Highest extreme: a 100% AIF would indicate that each attribute in each class was inherited from another class. Such a thing is not possible, however we can analyze a hypothetical case in which most of the attributes across our codebase are inherited. This would simply indicate a highly hierarchical structure to our system, which is not necessarily positive nor negative: different systems will have different inheritance needs.

Our values:

|  |  |
| --- | --- |
| Project | AIF |
| JabRef | 23.20% |

|  |  |
| --- | --- |
| Module | AIF |
| main | 25.70% |

Taking into account these values only, it is plausible to say that the JabRef project sits in the middle of the two situations described above. The inheritance principle *is* used, as evidenced by the fact that around 25% of all variables in a given class will be inherited, but the system isn’t structured in one large inheritance tree.

CF – Coupling Factor

Related to the association principle in object-oriented programming.

Lowest extreme: on 0% CF we would have a system where no class interacts with another, which is not feasible. However, in a hypothetical system where there is very little interaction between classes, we would likely find very high cohesion, i.e., a proper separation of responsibilities, and classes that do not need to be concerned with other classes’ internal state. Loose coupling is commonly considered an essential design principle.

Highest extreme: in sequence to what was said about the previous extreme, it seems clear that with 100% CF we would go against the fundamental design principle that aims to prevent tight coupling between classes. As such, this is the extreme we will want to stay away from when structuring our project.

Our values:

|  |  |
| --- | --- |
| Project | CF |
| JabRef | 0.69% |

|  |  |
| --- | --- |
| Module | CF |
| main | 1.01% |

According to what was said above, the JabRef project displays an appropriate percentage for the coupling factor metric - being near to 0% - meaning that the project as a whole benefits from a loose coupling.

MHF – Method Hiding Factor

Related to the encapsulation principle in object-oriented programming.

Lowest extreme: with a 0% MHF we would be looking at a system in which every method from every class is available to all the other classes in the project. Unlike the AHF metric, where this would not be a desirable situation, the MHF metric indirectly measures the “usefulness” of the average class for the system: the lower its value, the more methods an average class will offer. In more realistic terms, a 0% MHF is most likely not practical, as many times classes will require private methods to aid their logic in some form.

Highest extreme: following the logic from above, 100% MHF is not the extreme we’re looking to lean towards in our code. Having close to 100% method hiding factor would mean that most methods serve no purpose outside the class they’re in. This may indicate that the very little purpose that each class offers to our system is overly complicated, requiring many private methods to aid that logic and the code should thus be refactored to better distribute responsibilities.

Our values:

|  |  |
| --- | --- |
| Project | MHF |
| JabRef | 36.93% |

|  |  |
| --- | --- |
| Module | MHF |
| main | 27.28% |

In the JabRef project we can find a relatively low MHF, leaning towards the lowest and most productive extreme. This suggests an overall relevance of the average class in the context of this system, with only up to 36.93% of its methods being hidden from outer classes.

MIF – Method Inheritance Factor

Related to the inheritance principle in object-oriented programming.

Lower extreme: the same explanation that was given for the AIF metric can be applied here. With a 0% MIF none of the classes in the system would have any inherited methods in them, indicating the use of no inheritance features, which are crucial in an object-oriented paradigm.

Higher extreme: similarly, a value near to 100% as a MIF would indicate that every method in every class was inherited from another class. This would indicate a system structured in a highly hierarchical way, which, as stated before, is not necessarily positive nor negative, it simply represents a particular structuring choice.

Our values:

|  |  |
| --- | --- |
| Project | MIF |
| JabRef | 18.3% |

|  |  |
| --- | --- |
| Module | MIF |
| main | 23.14% |

According to these values, one can argue that although the inheritance principle *is* definitely used, the system isn’t structured in one large inheritance tree, with only around 20% of the methods in an average class being inherited from other classes.

PF – Polymorphism Factor

Related to the polymorphism principle in object-oriented programming.

Lower extreme: with 0% PF, the system in analysis would be devoid of polymorphic features. Polymorphism is one of the core features of object-oriented programming, being tightly related to inheritance. Polymorphism, as it is analyzed here, only accounts for overridden methods – which can only happen with the help of inheritance. Having a low percentage of this factor would not only indicate a lack in this crucial feature, but also in inheritance features. It has been discussed above why this would be a problem with the design of the code.

Higher extreme: with a value near to 100% for the PF, every method found in the code would be overridden by descendant classes. This reveals a nuclear inheritance issue with the code, where every method inherited by every class is useless to it.

Our values:

|  |  |
| --- | --- |
| Project | PF |
| JabRef | 49.81% |

|  |  |
| --- | --- |
| Module | PF |
| main | 49.47% |

The values observed in the JabRef project for this metric sit neatly in the middle of the 2 previously mentioned scenarios. However, since in this case a tendency to either of the extremes would present an issue, it can be argued that, for our project, these are perfectly acceptable values.

Conclusions

As has been found while discussing the relevant metrics, the JabRef project adequately uses all the core concepts of object-oriented programming, with percentages sitting in appropriate values for every studied metric.

Knowing this, it is expected that no correlation between these percentages and the code smells identified by the S Team can be made, and that no particular trouble spots were found.

References

Abreu, F. B., R. Esteves, and M. Goulão, "The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics", TOOLS'96 (Technology of Object Oriented Languages and Systems), Santa Barbara, CA, EUA, 1996.

Gabriela Costa 58625