**Dependency Metrics**

Before to analysis of the data collected of the dependency metric results, it is importante to understand the following keywords:

**Cyclic:** This metric is used to measure the number of dependency graphs that packages as nodes reach the same abstraction by following one or more paths forming a cycle.

**Dcy**: This metric is used to measure the number of abstract classes the current abstraction depends on.

**Dcy\*:** This metric is used to measure the number of indirect dependency between class abstractions.

**Dpt:** this metric is used to measure the number of abstract classes that depends on the current class.

**Dpt\*:** This metric is used to measure the number of abstract classes that depends indirectly on the current class.

**PDcy:** This metric is used to measure the number of packages that the current abstraction depends on.

**PDpt:** This metric is used to measure the number of packages that the current abstraction is used.

**Troubleshooting possible cases**

Most of the packages has a high Cyclic and Dcy\* values (in this case, mosto f the classes has 784 value on Cyclic metric and 1337 value on Dcy\* metric), meaning that most of the classes on this packages depends on each other and one change on one class, has the potencial to affect the other dependable classes, causing cascade of changes.

Some classes has high values on Dpt which means that this classes are used alot in many implementations of other classes.

# Martin Package Metrics Report

I begin this report by first preseting the definitions of each parameter of the metrics to better explain what each thing represent in the project’s context then later show the possible troublesome spots in the project’s code due to the metrics results.

The definitions were gathered from

# Definitions of the Parameters

**Efferent Coupling (Ce) -** This metric is used to measure interrelationships between classes. As defined, it is a number of classes in a given package, which depends on the classes in other packages. It enables us to measure the vulnerability of the package to changes in packages on which it depends.

The high value of the metric Ce> 20 indicates instability of a package, change in any of the numerous external classes can cause the need for changes to the package. Preferred values for the metric Ce are in the range of 0 to 20, higher values cause problems with care and development of code.

**Afferent Coupling (Ca) -** This metric is an addition to metric Ce and is used to measure another type of dependencies between packages, i.e. incoming dependencies. It enables us to measure the sensitivity of remaining packages to changes in the analysed package.

High values of metric Ca usually suggest high component stability. This is due to the fact that the class depends on many other classes. Therefore, it can’t be modified significantly because, in this case, the probability of spreading such changes increases.

Preferred values for the metric Ca are in the range of 0 to 500.

**Abstractness (A) -** This metric is used to measure the degree of abstraction of the package and is somewhat similar to the instability (later presented). Regarding the definition, abstractness is the number of abstract classes in the package to the number of all classes.

**Instability(I) -** This metric is used to measure the relative susceptibility of class to changes. According to the definition instability is the ration of outgoing dependencies to all package dependencies and it accepts value from 0 to 1.

According to Robert Martin, the optimal case would be that instability of the class is compensated by its abstractness, which satisfies the equation I + A = 1. Classes that were well designed should group themselves around this graph end points along the main sequence.

**Normalized distance from Main Sequence (D)** **-** This metric is used to measure the distance between stability and abstractness and is calculated using the formula  
D = | A + I -1 |, where A is Abstractness and I is Instability.  
This value should be as low as possible as it indicates it’s close to the main sequence. We should always avoid the two extreme situations which are:  
 A = 0 and I = 0, meaning that the package is extremely stable and concrete, meaning that it can’t be extended.

A = 1 and I = 1, most impossible as a completely abstract package must have some sort of connection to the outside, so that the instance that uses said functionalities of the package could be created.

**Troubleshooting possible cases**

The package “org.jabref.architecture” present values of 1 to both A and I parameters, meaning that it’s a completely abstract package with no connections to the outside.

The packages “org.jabref.logic.logging”, “org.jabref.logic.l10n” and many more present in the excel sheet present values of 0 to both A and I parameters, meaning that each package cannot be extended as it is very stable and concrete

Many of the packages present in the project dont satisfy the equation presented by Robert Martin (I + A = 1) for being an optimal package, for example the package “org.jabref.logic.preview”, with values of 1 for A and 0.02 for I, representing that the package may present unnecessary dependancy coupling.

7 of the jabref’s packages’ Ca values surpass the recommended maximum limit which is 500 (Them being “org.jabref.logic.cleanup”, “org.jabref.model.strings”, “org.jabref.model.openoffice.style”, “org.jabref.model.metadata”, “org.jabref.logic.layout”, “org.jabref.model.groups” and “org.jabref.gui.icon”), possible pointing out that the package is very dependant on many other classes so it can’t be modifed too significantly or it would cause the changes to spread towards other classes, more specific, a shotgun surgery code smell.

There’s way too many packages in the project that surpass the recommended maximum value of Ce (default value being 20), causing them to be instable as a change in other external classes that make use of the package may cause changes to said package, again, possible shotgun surgery, innapropriate intimacy and feature envy code smells may be present in these packages.

Many of the packages dont present a very low value of D, meaning that they are very far from the main sequence of code due to the lack of proportionality between the value A and I.

**MOOD Metrics Report**

The MOOD (Metrics for Object-Oriented Design) metrics are a set of metrics which includes:

* Atribute Hiding Factor (AHF): The obscurity (contrary of visibility) of the attributes of a class. Public attributes are visible to all classes inside their class’s package and protected attributes are visible to all subclasses.
* Attribute Inheritance Factor (AIF): How many of a class’s fields can be inherited.
* Coupling Factor (CF): Dependencies per class (includes super classes).
* Method Hiding Factor (MHF): The obscurity of the methods of a class. For *public* methods it’s assumed they are visible to all classes inside their class’s package and for *protected* it’s assumed they are visible to all subclasses.
* Methond Inheritance Factor (MIF): How many of the non-overriden methods of a class are available from an inherited class.
* Polymorphism Factor (PF): How much polymorphism is used. This is calculated by dividing the number of overriding methods by the number of potential overrides in the subclasses.

**Analysis**

With this information we can take closer look into the collected metrics and take some conclusions.

For the values in the *architecture, CLI, migrations, preferences and stylestester* packages it meets our expectations.

*CLI* is mostly a proxy package to other packages so the coupling factor will naturally be high, the same for *migrations, preferences* and *styletester*.

The packages we should be worried about are the *gui, logic* and *model*, as these are the areas of most intese traffic in our application and as such we want to make sure that there isn’t evidence of any code that could introduce bugs in the future. These packages are also the ones that reflect the most on the overall metrics for the project.

*Trouble Spots*

Unwanted Visibility: Usually for AHF we wan’t values close to 100%, but all around the project we can see that AHF is really low which could indicate a huge problem with visibility. With high visibility comes the possibility for tampering with instance variables which could alter the flow of the program in unexpected ways. If this does reveal to be a problem the first package we should look into is ***model***, as it handles critical data and has one of the lowest AHF.

According to this metric set, there doesn’t seem to be any other trouble spots on the project, although, as mentioned previously, there are some packages with high coupling factor and it could be worth looking into a different way of implementing the functionalities of these packages without such high coupling.

**Complexity Metrics**

MetricsReloaded Plugin

Complexity metrics are measured based on cyclomatic complexity. The complexity of a module is the number of independent cycles in the flow graph (all the paths that can be traversed during a program execution). This metric correlates complexity with maintenance effort, meaning the more complex a module is the harder it is to maintain.

|  |  |  |
| --- | --- | --- |
| package | Average Cyclomatic Complexity | Total Cyclomatic Complexity |
| |  |  |  | | --- | --- | --- | | org.jabref.gui |  |  | | 2,01 | 595 |
| |  |  |  | | --- | --- | --- | | org.jabref.logic |  |  | | 1,18 | 13 |
| |  |  |  | | --- | --- | --- | | org.jabref.migrations |  |  | | 2,56 | 100 |
| |  |  |  | | --- | --- | --- | | org.jabref.model |  |  | | 1,96 | 106 |
| |  |  |  | | --- | --- | --- | | org.jabref.preferences |  |  | | 1,44 | 511 |

According to this table, on average every package has only 2 independent paths except for preferences and logic that have 1 path.

**Cognitive Complexity**

This complexity tells us how hard it is for a person to understand a method.

The package with the most cognitive complex methods is java/org/jabref/logic such as the one located in jabref/src/main/java/org/jabref/logic/bst/BibtexCaseChanger.java which the team has already pointed out in the code smells.

**Essential Cyclomatic Complexity**

This complexity shows how much complexity is left once we have removed the well structured complexity (i.e. a for loop which we know when it is going to finish). Methods with lower Essential complexity are easier to break into smaller methods, on the other hand methods with higher Essential complexity are more difficult to understand, maintain and test.

Gui and Logic are the packages with higher values of Essential Complexity. The method *org.jabref.gui.fieldeditors.FieldNameLabel.getDescription(Field)* has the highest Essential complexity because it has a big case with many returns in it.

**Design Complexity**

The Design Complexity is related to how interlinked a methods control flow is with calls to other methods.

The packages Gui and Logic have the most methods with the highest Design Complexity meaning it is harder to understand at once their interconnections. *org.jabref.logic.importer.fileformat.RisImporter.importDatabase(BufferedReader)* is the method with the worst design because it is long and calls many other methods, having many interconnections.

**Cyclomatic Complexity**

This complexity calculates how many independent paths there are in a method, thus how many tests are necessary.  
Once again, Gui and Logic are the packages with the highest complexity. *org.jabref.logic.layout.format.RTFChars.transformSpecialCharacter(long)* is the method with the highest complexity due to the amount of ifs there are that increment the number of independent paths

In conclusion, the packages Gui and Logic are the ones with methods that have the highest complexities, therefore the ones we should focus on improvements.

**Chidamber Metrics**

For this analysis the metrics for System B on the NASA Study1 will be used as reference values.

**CBO – Coupling Between Object Classes**

org.jabref.model.entry.BibEntry has the highest value of 616 for this metric which highly exceeds the reference maximum. The reason behind this is that BibEntry is a super class with over 1000 lines being the central object of the application. This class probably holds too much responsibility and should be split into smaller classes which handle certain aspects of a BibEntry. The overall average for this metric of 13.33 is still way above the reference value of 1.25, which indicates excess of class responsibilities is most likely common across the project.

**DIT – Depth of the Inheritance Tree**

org.jabref.gui.util.RadioButtonCell, org.jabref.gui.groups.GroupDialogView.IkonliCell and org.jabref.gui.commonfxcontrols.CitationKeyPatternPanel.HighlightTableRow all have the same highest DIT of 9, a little bit above our reference maximum of 4. However, these classes inherit classes on external libraries, therefore having little significance for the project’s quality.

**LCOM – Lack of Cohesion of Methods**

org.jabref.model.strings.StringUtil has the highest value of LCOM however being an util class with nothing but static methods that provide extended functionality over the Java String class this is to be expected and not necessarily an issue. However org.jabref.gui.util.NoSelectionModel has the second highest LCOM value of 15. This happens because the entire class is composed of empty methods, therefore there is no correlation between any methods. While this is not the type of issue this metric is meant to locate, a class which does not implement the methods of their super class signals that there may be a bad design, in this situation perhaps Disabling selection should be handled with the use of an optional variable rather than an empty class.

**NOC – Number of Children**

org.jabref.gui.actions.SimpleCommand has the highest NOC of 80, which is also way above our reference maximum of 21. This is because this class provides the basic functionality for a command and therefore has methods which gets reused often. However it may be possible that some subclasses could be grouped under other subclasses to reuse even more code. This would reduce the NOC for SimpleCommand.

**RFC – Response for a class**

org.jabref.preferences.JabRefPreferences has the highest RFC value of 543. This is however bellow the reference maximum of 827. The high value is due to the fact JabRefPreferences handles all the preferences classes, therefore interacting with those classes often.

**WMC – Weighted Methods Per Class**

org.jabref.preferences.JabRefPreferences also holds the highest WMC of 272, although this is also lower than the reference maximum of 381. As explained for the RFC, JabRefPreferences handles logic for all the preferences in the application, which results in a large number of methods.

**References:**

1Laing, V. and C. Coleman. “Principal Components of Orthogonal Object-Oriented Metrics.” (2001).