**Martin Packaging Metrics**

Miguel Real 55677

**Preface**

The aim of this report is to document, evaluate and explain metric data retrieved from the JabRef project, more specifically [Martin Packaging Metrics](http://web.archive.org/web/20140701184524/http://www.objectmentor.com/resources/articles/oodmetrc.pdf), extracted using the [Metrics Reloaded](https://github.com/BasLeijdekkers/MetricsReloaded) plugin for IntelliJ IDEA. We will analyze the source code in terms of *Abstractness*, *Afferent* and *Efferent Coupling*, *Distance from Main Sequence*, and *Instability*.

**Metrics**

**Dependency & Stability**

Object Oriented designs are geared towards being robust, maintainable, and reusable, however, if used carelessly, they can prove to be the exact opposite. Designs that don’t fulfill this goal are very typically highly interdependent, leading to designs that Robert Martin, in his ‘94 paper “*OO Design Quality Metrics*”, calls “rigid”, “fragile” and “difficult to reuse”. But what do these things mean? And how can we gauge the dependencies of our design? Ultimately, good designs that fulfill the “robust, maintainable and reusable” mantra have dependencies on “stable” classes. We must now define these terms.

**Rigidity** can be defined as the feasibility of making a change in the code taking into account having to deal with all the cascading changes (and the associated rise in cost) that such a change might provoke. A rigid design is one that is unlikely to be changed due to any modification to any part of the design creating a long chain of necessary changes in order to keep the code working, so much so that the cost associated with all that adaptation is much higher than the benefit the initial change would bring.

**Fragility**, on the other hand, describes how much of a program’s code will become broken if a change in it is made. A fragile design will break in many places if a change is made, including in conceptually unrelated parts of the code. This leads to a cycle of problem fixing where one fix breaks some other part of the code that, when fixed, breaks yet another part of the code and so on. This drastically harms the credibility of the development team.

Finally, **difficulty of reuse**, as the name might imply, describes how hard it is to reuse a part of a design due to them being highly dependent on other parts of the design. This makes the cost of separating that part of the code higher than just redeveloping the design outright.

The **Stability** of a class can be measured in terms of **Responsibility** and **Independence**. The former describes how heavily a class is depended upon, a Responsible class has a lot of other classes depending on it. The latter dictates how heavily a class depends on others, an Independent class doesn’t depend on anything else.

Responsible classes are quite stable, as any change on them will necessitate changes in all their dependents, so they are unlikely to be changed. In order to have the most stable classes, however, we need to make them both Responsible and Independent, as not only are they unlikely to be changed due to how that will affect others, but they themselves will most likely not require change due to not depending on any others.

**Class Categories**

It is rare for classes to be individually reused, usually a class is a component part of a set of classes from which it can’t easily be separated. Any reuse of that class will require reuse of the set. These classes are highly cohesive and are called **Class Categories**. Categories can be very well delineated in Java by using packages, where a package equates to a category.

There are 3 rules that class categories obey to (in order of importance, meaning less important rules can be sacrificed for more important ones):

1. Classes in a category are highly sensitive to changes in each other. If one is changed, the others will have to be as well. If one is open to expansion, they all are.
2. Classes in a category must be reused together. Being interdependent, they cannot be separated.
3. Classes in a category are related in function/purpose.

Dependencies within a class are expected and nigh on unavoidable. This means that for the purposes of optimizing design, we will focus on managing dependencies *between* categories, and the concepts previously discussed (stability, independence and responsibility) can and will be applied to categories.

**Stability & Abstraction**

Where does abstraction factor into this argument? In truth, abstraction provides a way for systems to maintain stability while allowing for expansion, as without it a maximally stable system is unchangeable. This has heavy roots in the Open/Closed principle.

Therefore, we should strive to include high abstraction just as we strive to include maximum stability. From this we can surmise that if stable categories must also be highly abstract, unstable categories will be highly concrete.

**Dependency Metrics & Abstractness**

With the previous concepts in mind, Martin identified 4 metrics to measure responsibility, independence and stability of categories (packages) as well as how abstract a category is:

* Afferent Couplings (Ca): # of classes outside a category that depend on classes within the category.
* Efferent Couplings (Ce): # of classes inside a category that depend on classes outside the category.
* Instability (I): Ce / (Ca + Ce) - Rates the stability/instability of a category (in a range of 0 to 1) where the lower the value the higher the stability.
* Abstractness (A): # of abstract classes in a category / # of classes in the category - Rates how abstract a category is in a range of 0 to 1 where 0 is concrete and 1 is completely abstract.

**The Main Sequence**

Taken from the astronomical concept of main sequence stars, the Main Sequence is a line in the graph that correlates Abstraction with Instability where A = I, meaning that categories that sit in it have balanced abstraction and instability. While it would be ideal for categories to sit in the extremes of the main sequence, in reality it is more realistic to strive for categories that sit on or as close as possible to the main sequence.

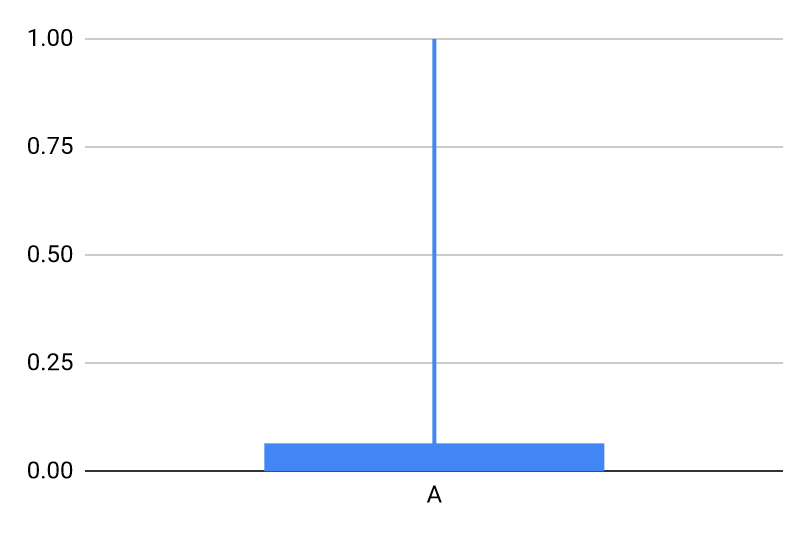
**Distance from Main Sequence**

We can now introduce our final metric, Distance from Main Sequence (D), that indicates how close/far a category is from the main sequence.

* Distance from Main Sequence (D/Dn): abs(A + I - 1) - Perpendicular distance of a category from the main sequence in the abstraction/instability graph, normalized for the range [0 , 1].

**Analysis of JabRef Martin Packaging Metrics**

*And their correlation with Code Smells*

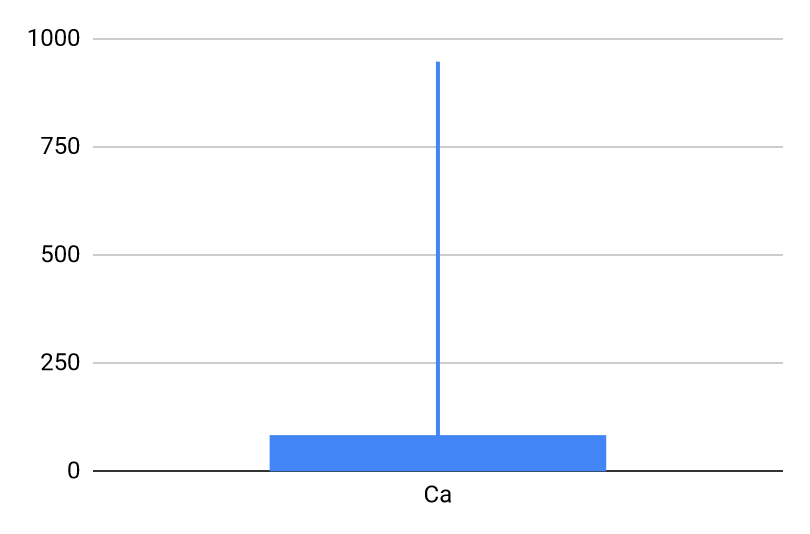


**Abstractness - A**

We see a very pronounced trend in the JabRef project in terms of the *A* metric. Most packages are almost entirely concrete, with only a select few including any abstraction and even fewer having high abstraction (only 3 packages are completely abstract).

Given such low average abstraction, one could argue this could prove a problem in terms of dependency and stability. This, however, cannot be completely determined without looking at other metrics and comparing them to this one.

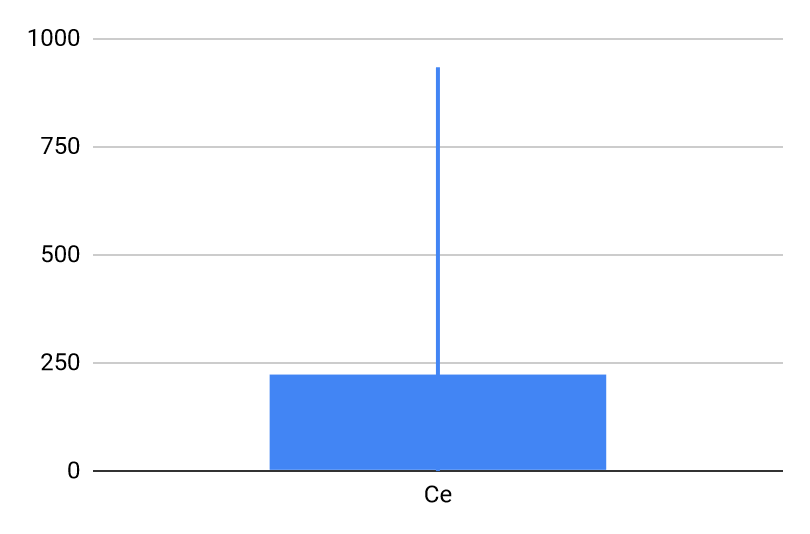
By examining the extremes we see that, at the maximum of *A*=1, we have packages that are composed entirely of interfaces, which will then be implemented elsewhere in the code. This method of interface encapsulation leads to higher interdependency between packages (leads to an increase in *Ca* and *Ce* as would be expected) and higher coupling. On the other hand, at the *A*=0 minimum we see completely concrete classes, which can (and in fact did) lead to Code Smells such as Long Method and Duplicate Code (to name a few that were actually documented by the team, multiple times) that can stem from insufficient use of abstraction.



**Afferent Couplings - Ca**

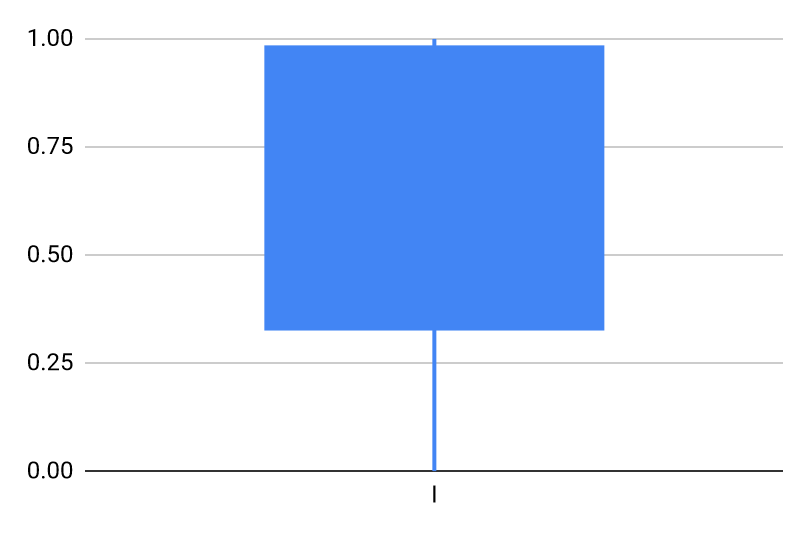
Similarly to the previous metric, a clear trend is shown when it comes to Afferent Couplings. The large majority of packages has very low *Ca* values meaning for most packages very few if any exterior classes depend on the classes inside them (most packages have fewer than 100, often drastically less but rarely 0). This is a positive metric for the most part, as we generally want to reduce coupling when possible, and there appears to have been an effort to uphold that standard, for the most part.

There are, however, some select few cases at the extreme with very high *Ca* values (one with 584 and another with 947, to name a few of the worst cases). This does prove to be a problem, as a lot of classes depend on this package which can easily lead to Shotgun Surgery code smells, as any needed change within this package will most likely require a lot of other classes to be changed as well. This is compounded by the fact that, as we just saw, Abstractness is generally very low meaning the very large majority of methods in classes are concrete, and therefore will require concrete changes to their code should any change occur in classes they depend on.

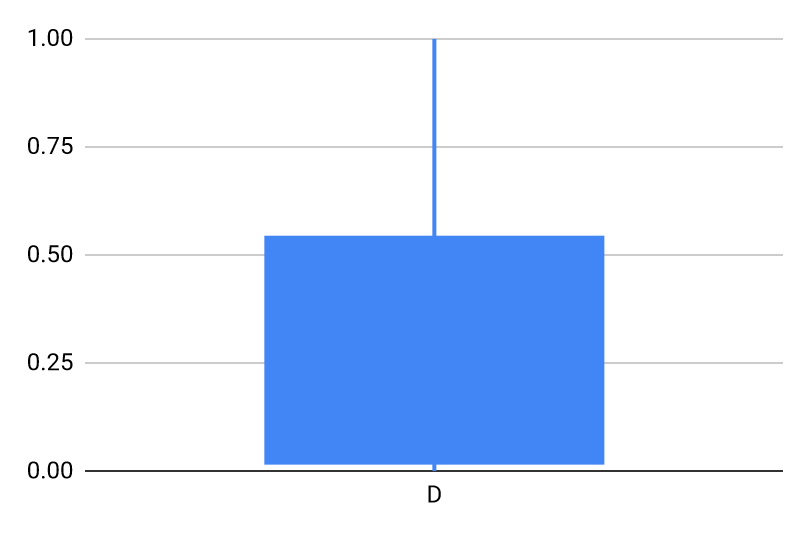


**Efferent Couplings - Ce**

Efferent Couplings are a lot higher on average than Afferent Couplings, meaning that most classes depend on several others outside their own packages. This can reveal “trouble spots” of classes with very strong coupling to many others. This high outside dependency makes it very difficult to reuse code from those classes, as it will break at the points where it requires outside assistance.

**Instability - I**

When it comes to Instability, we see that packages tend to be more on the unstable side, oftentimes having an *I* value of more than 0.5. This demonstrates a prevalence of Irresponsible and Dependent classes, as already discussed in Afferent/Efferent Couplings. Abstraction is also low as we’ve already seen, leading to very interdependent and concrete code that will be inflexible to change. Problems persist even in the opposite extreme, as the very few stable packages have little to no abstraction, meaning they will be completely inflexible to any sort of change, leading to a lot of rigidity.



**Distance from Main Sequence - D**

We see the effects of all the problems already discussed when analysing Main Sequence Distance, most packages are not on the main sequence and it is not uncommon for them to be quite far, over 0.5 in several instances. This confirms what has been alluded to thus far, that the JabRef codebase has quite severe problems with rigidity, fragility, and difficulty of reuse.

**In conclusion**, while there seems to have been an attempt at proper utilization of OO design, including some quite good examples of it, the JabRef code suffers from a lot of problems that I mainly attribute to how long it has been actively developed (2003-present, over 18 years) and its very large codebase, coupled with very little sense of direction and a lack of organization and leadership.