Software Engineering

The S-Team

Project Phase 1- Merged Document

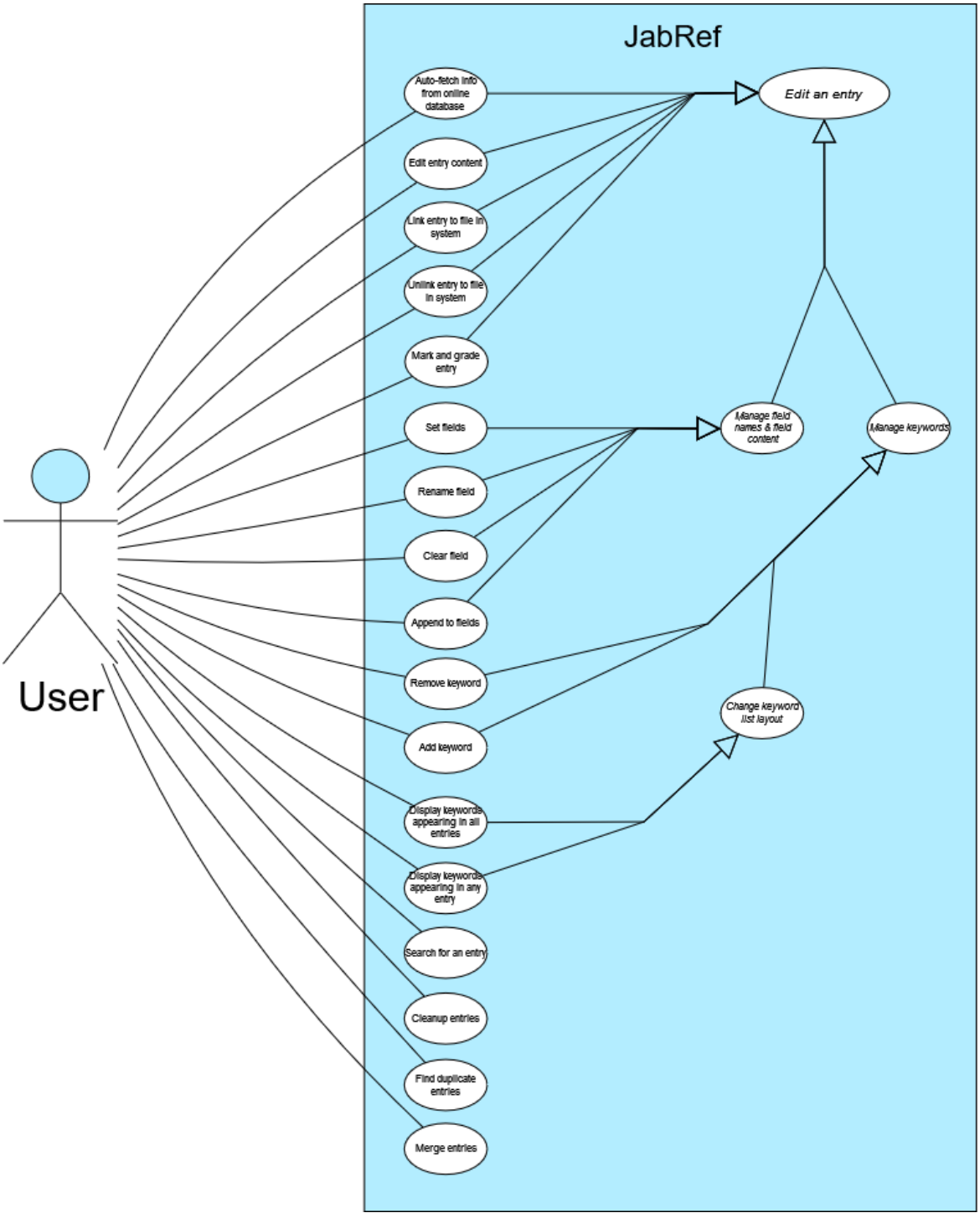
Team members:

1. *Miguel Real (55677)*
2. *Gonçalo Virgínia (56773)*
3. *João Vieira (56971)*
4. *Guilherme Pereira (57066)*
5. *Gabriela Costa (58625)*

Use Case Diagrams & Descriptions

Miguel Real (55677)

Use Case Diagram



Use Case Descriptions

Name: Auto-fetch info from online database
Id: UC03.0.1
Description: Automatically fetches information on entry from online databases.
Actors:
    Main - User
    Secondary – None

Name: Edit entry content
Id: UC03.0.2
Description: Edits content of an entry.
Actors:
    Main - User
    Secondary - None

Name: Link entry to file in system
Id: UC03.0.3
Description: Links entry to a file on the user's system.
Actors:
    Main - User
    Secondary – None

Name: Unlink entry to file in system
Id: UC03.0.4
Description: Unlinks entry to a file on the user's system.
Actors:
    Main - User
    Secondary - None

Name: Mark and grade entry
Id: UC03.0.5
Description: Allows the user to mark and rate the entry according to several criteria.
Actors:
    Main - User
    Secondary - None

Name: Set fields
Id: UC03.0.6.1
Description: Allows the user to set the content of a field.
Actors:
    Main - User
    Secondary – None

Name: Rename field

Id: UC03.0.6.2
Description: Renames a field.
Actors:
Main - User
Secondary – None

Name: Clear field
Id: UC03.0.6.3
Description: Removes a field from the entries.
Actors:
Main - User
Secondary - None

Name: Append to fields
Id: UC03.0.6.4
Description: Allows a user to add to the content of a field.
Actors:
Main - User
Secondary - None

Name: Remove keyword
Id: UC03.0.7.1
Description: Removes a keyword.
Actors:
Main - User
Secondary - None

Name: Add keyword
Id: UC03.0.7.2
Description: Adds a keyword.
Actors:
Main - User
Secondary - None

Name: Display keywords appearing in all entries
Id: UC03.0.7.3.1
Description: Allows the user to make it so only keywords that appear in all entries appear in the keyword list.
Actors:
Main - User
Secondary - None

Name: Display keywords appearing in any entry
Id: UC03.0.7.3.2
Description: Allows the user to make it so all keywords appear in the keyword list.

Actors:
Main - User
Secondary - None

Name: Search for an entry
Id: UC03.1
Description: Searches library for an entry.
Actors:
Main - User
Secondary - None

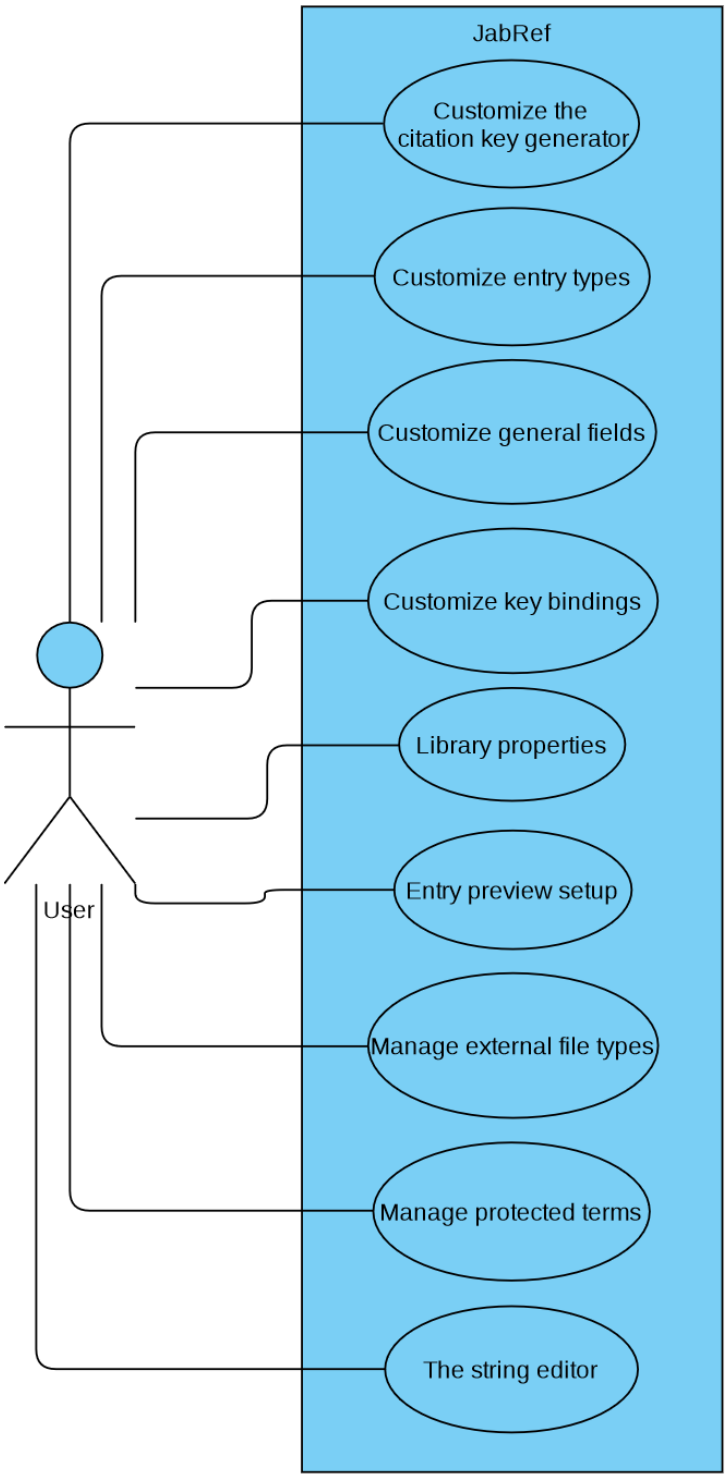
Name: Cleanup entries
Id: UC03.2
Description: Cleans up the entries in the library according to parameters the user can set.
Actors:
Main - User
Secondary - None

Name: Find duplicate entries
Id: UC03.3
Description: Finds duplicate entries in the library.
Actors:
Main - User
Secondary - None

Name: Merge entries
Id: UC03.4
Description: Merges two entries into one, how differences in each are shown can be set by the user.
Actors:
Main - User
Secondary – None

Gonçalo Virgínia (56773)

Use Case Diagram



Use Case Descriptions

Name: Customize the citation key generator
Id: UC01.0
Description: The pattern used in the auto generation of citation labels can be set for each of the standard entry types.
Actors:
Main - User
Secondary - None

Name: Customize entry types
Id: UC01.1
Description: When customizing an entry type, you both define how its entry editor should look, and what it takes for JabRef to consider an entry complete. You can both make changes to the existing entry types, and define new ones.
Actors:
Main - User
Secondary - None

Name: Customize general fields
Id: UC01.2
Description: You can add an arbitrary number of tabs to the entry editor. These will be present for all entry types.
Actors:
Main - User
Secondary - None

Name: Customize key bindings
Id: UC01.3
Description: Customize key bindings.
Actors:
Main - User
Secondary - None

Name: Library properties
Id: UC01.4
Description: Each library can have specific properties that can be modified.
Actors:
Main - User
Secondary - None

Name: Entry preview setup
Id: UC01.5
Description: You can change the layout, style, and display the entry preview as a separate tab.

Actors:
Main - User
Secondary - None

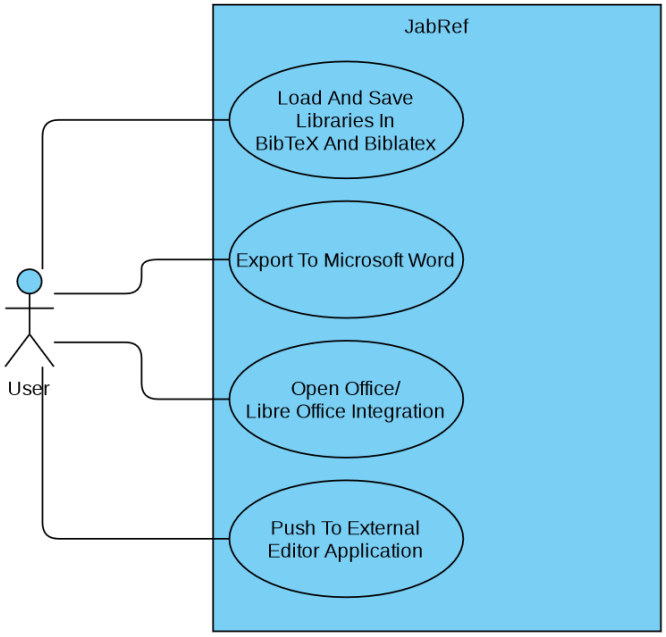
Name: Manage external file types
Id: UC01.6
Description: A file type is specified by its name, a graphical icon, a file extension and an application to view the files.
Actors:
Main - User
Secondary - None

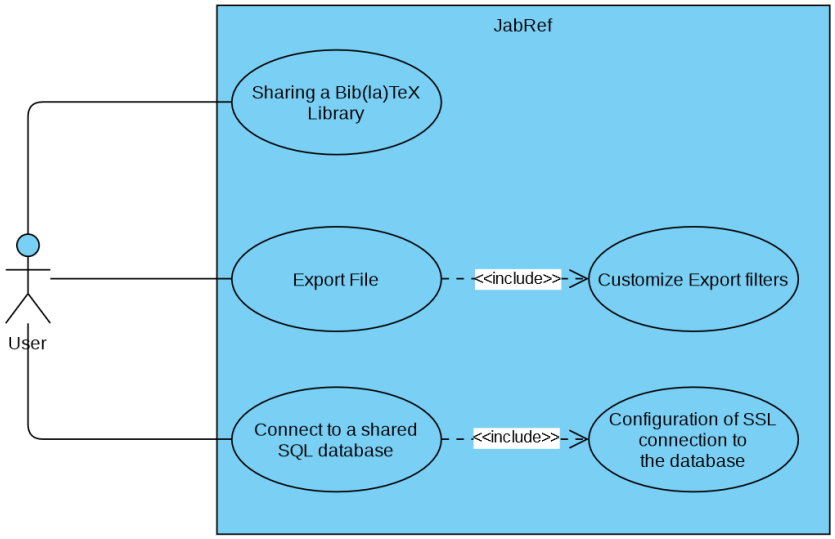
Name: Manage protected terms
Id: UC01.6
Description: Manage protected terms.
Actors:
Main - User
Secondary - None

Name: The string editor
Id: UC01.7
Description: Strings can be edited by Library → Edit string constants or pressing a button in the toolbar.
Actors:
Main - User
Secondary – None

João Vieira (56971)

Use Case Diagram





Use Case Descriptions

Name: LoadAndSaveLibrariesInBibTeXAndBiblatex
Id: UC04.1.0
Description: User load and saves his libraries directly in the BibTeX/biblatex .bib format
Actors:
Main - User
Secondary - None

Name: ExportToMicrosoftWord
Id: UC04.1.1
Description: User can export his desired entries to Office 2007 XML format
Actors:
Main - User
Secondary – None

Name: OpenOffice/LibreOfficeIntegration
Id: UC04.1.2
Description: User can insert citations format a Bibliography in an OpenOffice or LibreOffice Writer document from JabRef.
Actors:
Main - User
Secondary - None

Name: PushToExternalEditorApplication
Id: UC04.1.3
Description: User can push desired entreis to a external editor application.
Actors:
Main - User
Secondary - None

Name: SharingABib(la)TeX
Id: UC04.2.0
Description: User can share his Bib(la)TeX library
Actors:
Main - User
Secondary - None

Name: ExportFile
Id: UC04.2.1
Description: User can export his entries into a file.
Actors:
Main - User
Secondary - None

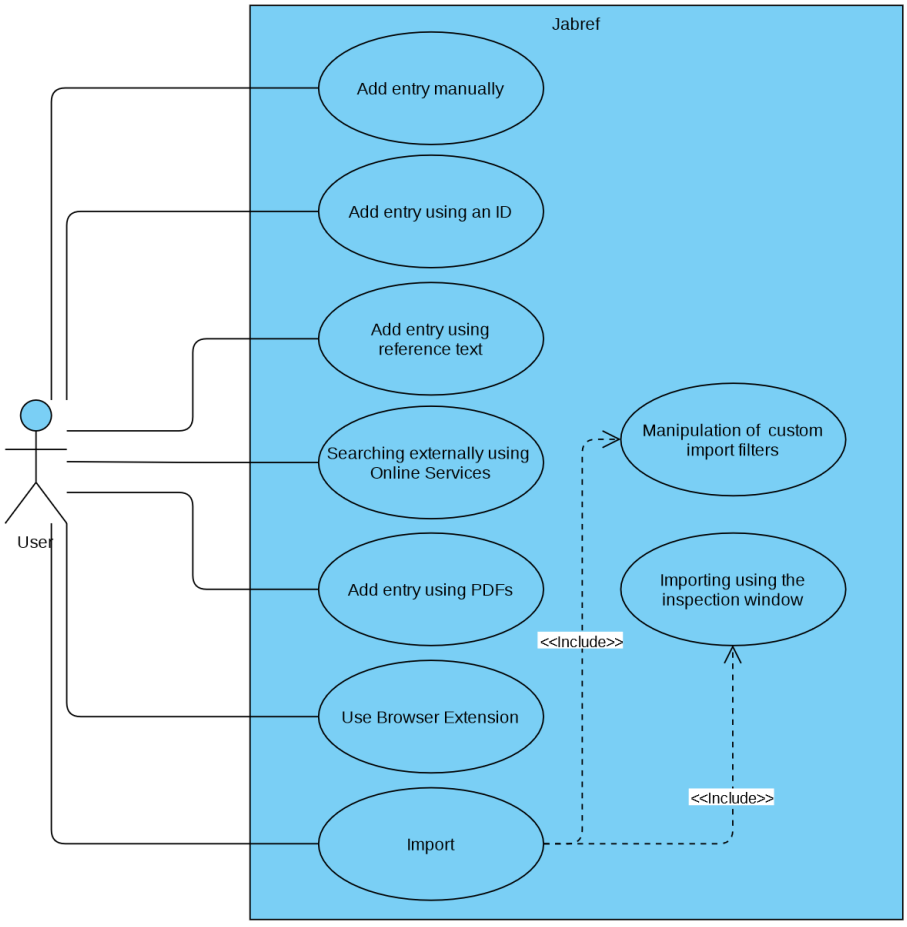
Name: CustomizeExportFilters
Id: UC04.2.1.1
Description: User can define and use his own export filters.
Actors:
Main - User
Secondary – None

Name: ConnectToASharedSQLDatabase
Id: UC04.2.2
Description: User can connect to a remote database
Actors:
Main - User
Secondary - None

Name: ConfigurationOfSSLConnectionToTheDatabase
Id: UC04.2.2.1
Description: User can configurate his connectio to the database
Actors:
Main - User
Secondary – None

Guilherme Pereira (57066)

Use Case Diagram



Use Case Descriptions

Name: AddManualEntry
Id: UC05.0
Description: The user adds a new entry.
Actors:
Main - User
Secondary - None

Name: AddIDEntry
Id: UC05.1
Description: The user adds a new entry with a given ID.
Actors:
Main - User
Secondary - None

Name: AddRefTextEntry
Id: UC05.2
Description: The user adds a new entry by providing a reference text.
Actors:
Main - User
Secondary - None

Name: AddOnlineEntry
Id: UC05.3
Description: The user adds a new entry from within the system databases.
Actors:
Main - User
Secondary - None

Name: AddPDFEntry
Id: UC05.4
Description: The user adds a new entry with a given PDF file.
Actors:
Main - User
Secondary – None

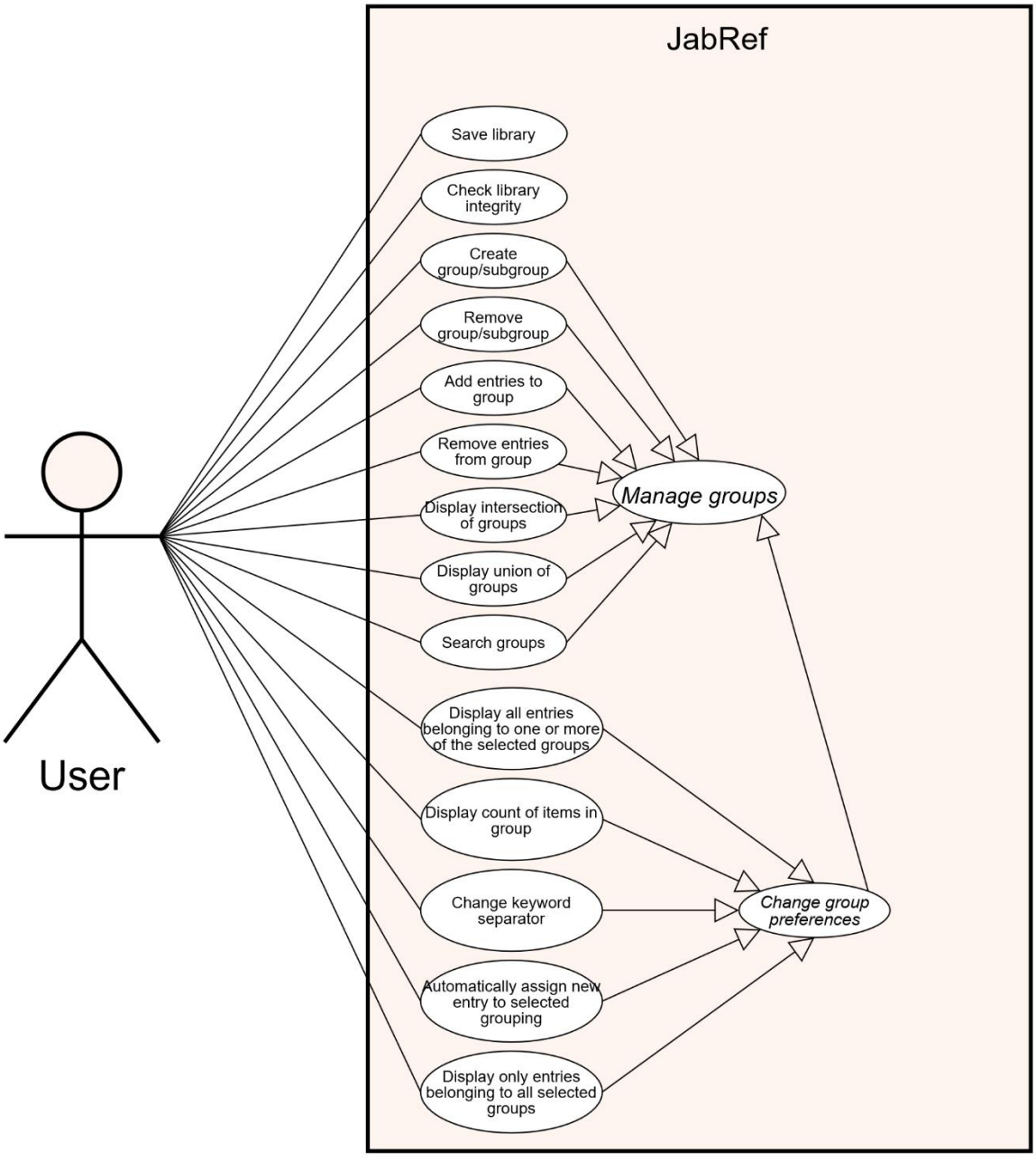
Name: AddBrowserEntry
Id: UC05.5
Description: The user adds a new entry with Jabref's browser extension.
Actors:
Main - User
Secondary - None

Name: AddInspWindEntry
Id: UC05.6.1
Description: The user selects which entries from an assortment of them and chooses which ones to add in an inspection window.
Actors:
Main - User
Secondary - None

Name: ManImportFilters
Id: UC05.6.2
Description: The user defines and uses it's own import filters.
Actors:
Main - User
Secondary – None

Gabriela Costa (58625)

Use Case Diagram



Use Case Descriptions

Name: Save library
Id: UC02.0
Description: Saves the current library.
Actors:
    Main - User
    Secondary - None

Name: Check library integrity
Id: UC02.1
Description: Checks the integrity of a library.
Actors:
    Main - User
    Secondary - None

Name: Create group/subgroup
Id: UC02.2.1
Description: Creates a group or subgroup.
Actors:
    Main - User
    Secondary - None

Name: Remove group/subgroup
Id: UC02.2.2
Description: Removes a group/subgroup.
Actors:
    Main - User
    Secondary – None

Name: Add entries to group
Id: UC02.2.3
Description: Adds an entry to the selected group.
Actors:
    Main - User
    Secondary - None

Name: Remove entries from group
Id: UC02.2.4
Description: Removes an entry from group.
Actors:
    Main - User
    Secondary – None

Name: Display intersection of groups
Id: UC02.2.5
Description: Sets the displayed entries of a group to be the intersection between itself and its subgroups.
Actors:
Main - User
Secondary – None

Name: Display union of groups
Id: UC02.2.6
Description: Sets the displayed entries of a group to be the union of it's own and its subgroup's.
Actors:
Main - User
Secondary - None

Name: Search groups
Id: UC02.2.7
Description: Searches for an entry in the selected fields.
Actors:
Main - User
Secondary - None

Name: Display all entries belonging to one or more of the selected groups
Id: UC02.2.8.1
Description: Sets the entries to be displayed when selecting multiple groups, in this case all from every group.
Actors:
Main - User
Secondary - None

Name: Display count of items in group
Id: UC02.2.8.2
Description: Displays how many items are in a group.
Actors:
Main - User
Secondary - None

Name: Change keyword separator
Id: UC02.2.8.3
Description: Changes the character used to separate keywords.
Actors:
Main - User
Secondary – None

Name: Automatically assign new entry to selected grouping
Id: UC02.2.8.4
Description: Allows the user to decide if new entries should be assigned automatically to the selected group.
Actors:
Main - User
Secondary - None

Name: Display only entries belonging to all selected groups
Id: UC02.2.8.5
Description: Sets the entries to be displayed when selecting multiple groups, in this case only the ones common to every group.
Actors:
Main - User
Secondary – None

Metrics Sets

Miguel Real (55677)

Martin Packaging Metrics

Preface

The aim of this report is to document, evaluate and explain metric data retrieved from the JabRef project, more specifically Martin Packaging Metrics, extracted using the Metrics Reloaded 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].

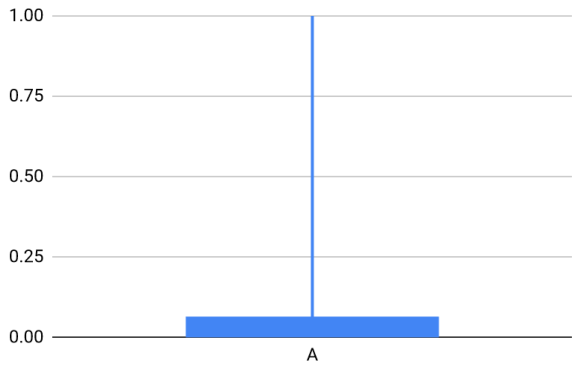
Analysis of JabRef Martin Packaging Metrics

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).

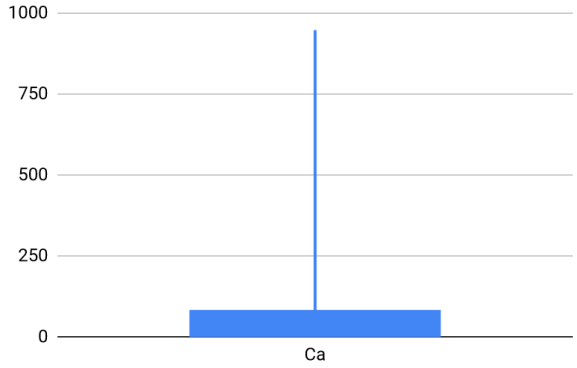
Given such low average abstraction, one could argue this could prove a problem in terms of dependency and stability. This, however, cannot be completely determined without looking at other metrics and comparing them to this one.



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

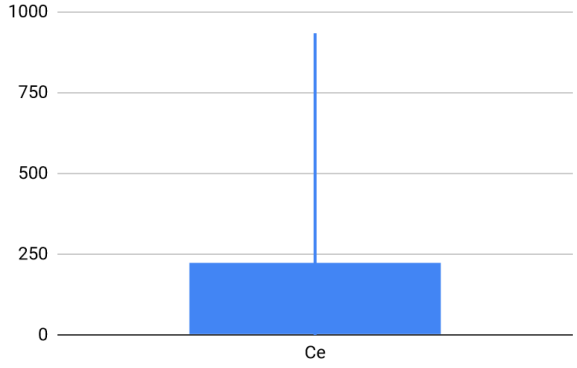
Similarly to the previous metric, a clear trend is shown when it comes to Afferent Couplings. The large majority of packages has very low *Ca* values meaning for most packages very few if any exterior classes depend on the classes inside them (most packages have fewer than 100, often drastically less 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.



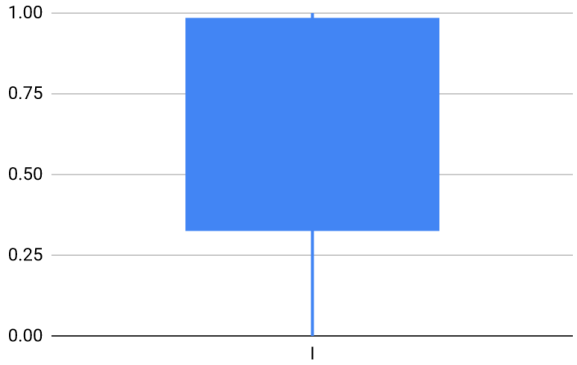
There are, however, some select few cases at the extreme with very high *Ca* values (one with 584 and another with 947, to name a few of the worst cases). This does prove to be a problem, as a lot of classes depend on this package which can easily lead to Shotgun Surgery code smells, as any needed change 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.

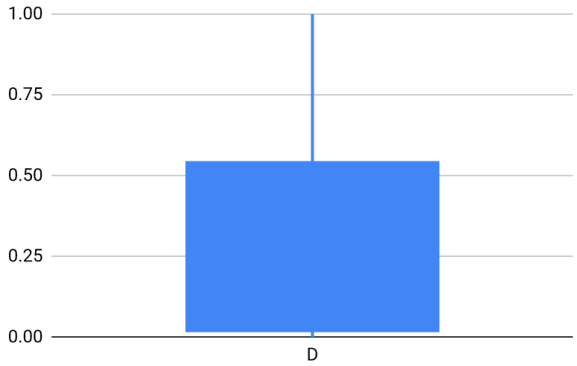


Instability - I



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.

Gonçalo Virgínia (56773)

Chidamber and Kemerer Metrics

Briefing

Chidamber and Kemerer metrics pertain to information on object coupling, inheritance depth, cohesion between methods (or lack thereof), number of subclasses.. amongst other metrics.

The specific metrics discussed in this report (extracted using the MetricsReloaded plugin for IntelliJ IDEA) and their corresponding abbreviations, are as follows:

CBO - Coupling Between Objects

DIT - Depth of Inheritance Tree

LCOM - Lack of Cohesion of Methods

NOC - Number of Children

RFC - Response for Class

WMC - Weighted Method Complexity

CBO - Coupling Between Objects

CBO = number of classes to which a class is coupled

Classes are considered coupled when methods declared in one class use methods or instance variables defined by another class.

Multiple accesses of one class to another class are counted as one access. Only method calls and variable references are counted - the use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are not counted.

If a method call is polymorphic (either because of Overrides or Overloads), all the classes to which the call can go are included in the coupled count.

In general, high CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the chance that changes in one class will imply changes in other coupled classes, and therefore natural refactoring that comes with growth becomes more difficult.

CBO > 14 is considered too high, as concluded in: Can Metrics Help Bridging the Gap Between the Improvement of OO Design Quality and Its Automation?

DIT - Depth of Inheritance Tree

DIT = maximum inheritance path from the class to the root class

The deeper a class is in an inheritance hierarchy, the more methods and variables it is likely to inherit, making it more complex.

Although deep trees promote reuse as a result of method inheritance, a high DIT has been found to increase faults. However, it's not necessarily the classes deepest in the class hierarchy that have the most faults, usually the most fault-prone classes are the ones in the middle of the tree, since usually the root and deepest classes are consulted often, and, due to familiarity, have low fault-proneness compared to classes in the middle.

The Visual Studio .NET documentation recommends a DIT <= 5, other sources allow up to 8.

LCOM - Lack of Cohesion of Methods

Take each pair of methods in the class. If they access disjoint sets of instance variables, increase P by one. If they share at least one variable access, increase Q by one.

LCOM = P − Q, if P > Q

LCOM = 0 otherwise

LCOM = 0 indicates a cohesive class.

LCOM > 0 indicates that the class can be split into two or more classes, since its variables belong in disjoint sets.

A high LCOM value indicates disparateness in the functionality provided by the class. This metric can be used to identify classes that are attempting to achieve many different objectives, and consequently are likely to behave in less predictable ways than classes that have lower LCOM values.

NOC - Number of Children

NOC = number of immediate subclasses of a class

NOC measures the breadth of a class hierarchy, whereas DIT measures the depth.

Depth is generally better than breadth, since it promotes reuse of methods through inheritance. Inheritance levels can be added to increase the depth and reduce the breadth.

A high NOC may indicate:

- High reuse of base class. Inheritance is a form of reuse.
- Base class may require more testing.

- Improper abstraction of the parent class.
- Misuse of sub-classing. In such a case, it may be necessary to group related classes and introduce another level of inheritance.

A class with a high NOC and a high WMC indicates complexity at the top of the class hierarchy. The class is potentially influencing a large number of descendant classes. This can be a sign of poor design.

Not all classes should have the same number of subclasses. Classes higher up in the hierarchy should have more subclasses than those lower down.

RFC - Response for Class

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set.

RFC = M + R (First-step measure)

RFC' = M + R' (Full measure)

M = number of methods in the class

R = number of remote methods directly called by methods of the class

R' = number of remote methods called, recursively through the entire call tree

A given method is counted only once in R (and R') even if it is executed by several methods M.

Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.

A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand.

The use of RFC' should be preferred over RFC. RFC was originally defined as a first-level metric because it was not practical to consider the full call tree in manual calculation. With an automated code analysis tool, getting RFC' values is no longer problematic. As RFC' considers the entire call tree and not just one first level of it, it provides a more thorough measurement of the executed code.

WMC - Weighted Method Complexity

WMC is simply the method count for a class.

WMC = number of methods defined in class

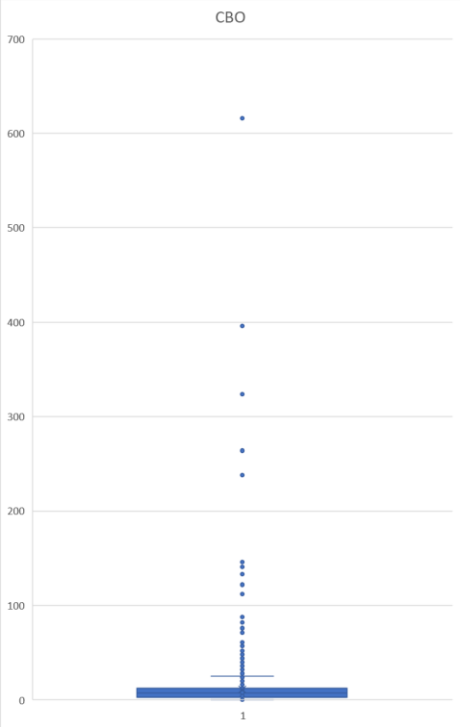
A high WMC has been found to lead to more faults. Classes with many methods are likely to be more application specific, limiting the possibility of reuse. A large number of methods also means a greater potential impact on subclasses, since they inherit (all or some of) the methods of the base class.

A high WMC value may indicate a class that should be restructured into smaller classes.

A good WMC is not clear-cut. One way is to limit the number of methods in a class to a couple of dozen. Another way is to have a limit of 10% of classes that can have more than 24 methods. This allows large classes but most classes should be small.

Chidamber and Kemerer Metrics in JabRef

CBO - Coupling Between Objects

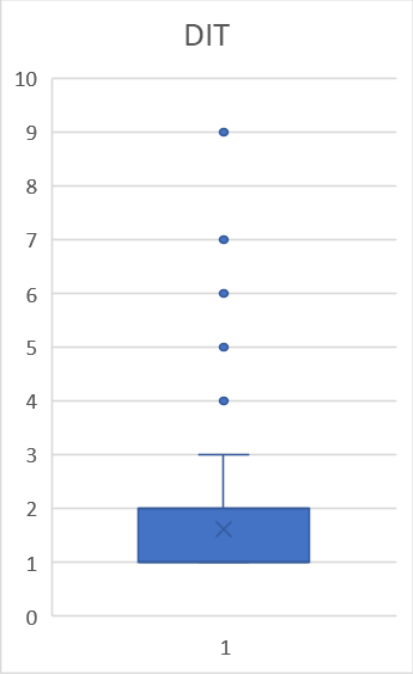


As can be observed in the collected metrics sheet, although the average CBO (11.11) is below the recommended 14, a total of 354 classes have a CBO above that.

192 classes have a CBOin the 20-100 range, and 14 have a CBO > 100, the highest being the BibEntry class with 616.

The amount of classes with absurdly high values for coupling are a clear indication of the Shotgun Surgery, Inappropriate Intimacy and, Feature Envy code smells, especially the first one, since changes/refactoring in one class may (in most of the aforementioned cases, *will*) require changes to be made to many different related classes.

DIT - Depth of Inheritance Tree

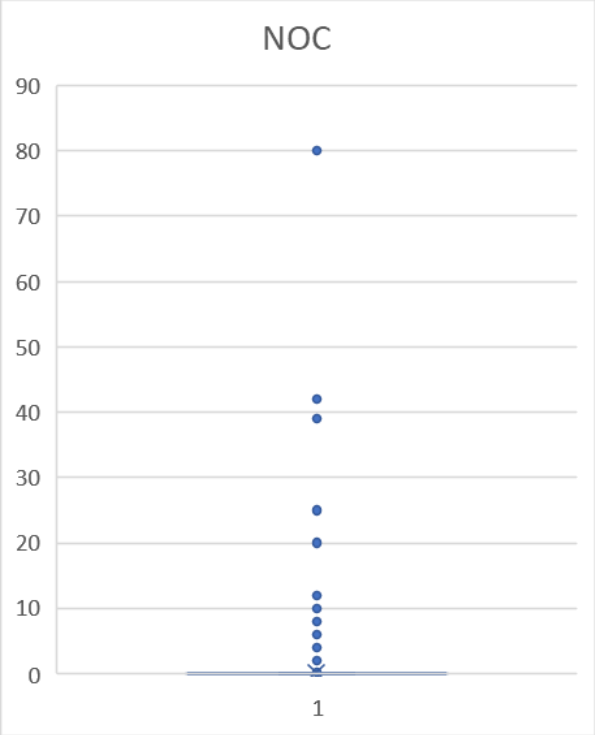


Inheritance tree depth in JabRef doesn't seem to be as big of an issue as CBO, seeing as the average DIT per class is only 1.62 and the highest values for this metric are 7 and 9 for 28 classes. Although, this might be an indication of some needed refactoring to the middle classes of a number of inheritance trees, in order to distribute responsibilities and deliver better abstraction overall.

LCOM - Lack of Cohesion of Methods

Although LCOM (also known as LCOM1) has some critiques for being too simplistic, and therefore spawned more intricate metrics (LCOM2, LCOM3..), there is a clear volume of classes (25) with an LCOM value above 10 (67 being the highest). This is an indication of the Large Class and Long Parameter List code smells, which indicate that a number of these classes should have their responsibilities divided into smaller classes. These were indeed some of the documented code smells in our report.

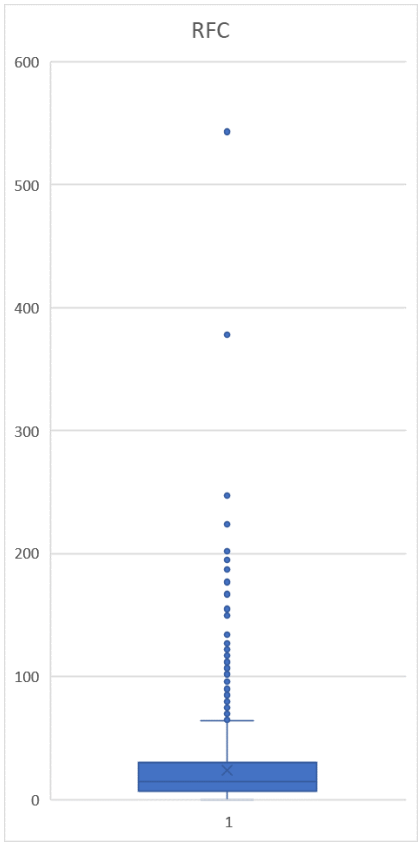
NOC - Number of Children



The average value for each classes' NOC is only 0.23, with a small number of classes having an extremely high value. Although not very problematic, this might indicate that another level of abstraction/inheritance should be added in some of these trees, increasing the average DIT (which is desirable up to a certain point) but greatly reducing the NOC, promoting reuse.

An outlier class, which happens to be the one with the highest NOC (80), is the SimpleCommand abstract class, since it is the base for an implementation of the Command Pattern (as documented in the Design Patterns section), and therefore is not necessarily an indication of poor design.

RFC - Response for Class

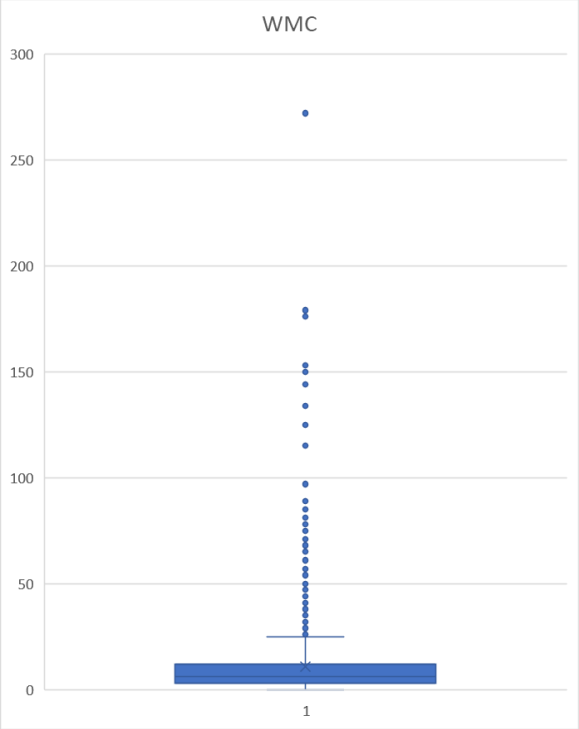


The average RFC per class is 23.86, with a large number of classes (47) holding values above 100 (543 being the highest).

Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes, which may indicate code smells similar to the ones related with CBO.

Additionally, classes with a high RFC hold a higher level of complexity and are therefore harder to understand and make changes from the get go.

WMC - Weighted Method Complexity



There are 173 total classes with a WMC above 25 (272 being the highest).

Although some classes in the system require a higher method count, the number of classes with an absurdly high WMC is a clear indication that their responsibilities should be subdivided into smaller classes (Large Class code smell, similar to the conclusion of the LCOM metric).

João Vieira (56971)

Dependency Metrics

Introduction

Utilizing the metrics reloaded plugin in IntelliJ IDEA, we can obtain metrics on the number of many different types of dependencies.

All this metrics gives us information that allows us to identify potential trouble spots in the codebase. However, this is not the case for this project, as the values obtained in the metrics are within the plugin's thresholds, therefore the values of this metrics are not enough to identify potential code smells and other problems, as we will see further in this report.

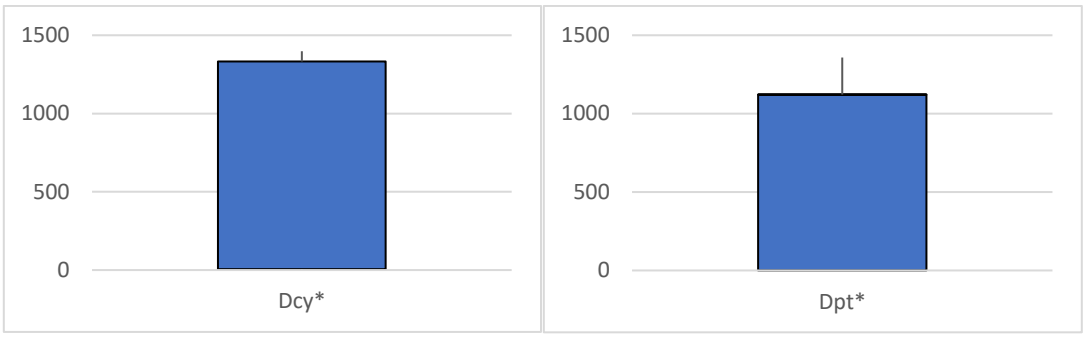
To simplify the reading process, the following abbreviations to the various dependencies were used:

- Cyclic – Number of cyclic dependencies
- Dcy – Number of dependencies
- PDpt – Number of dependent packages
- Dpt – Number of dependents
- PDcy – Number of package dependencies
- Dcy*– Number of transitive dependencies
- Dpt*– Number of transitive dependents

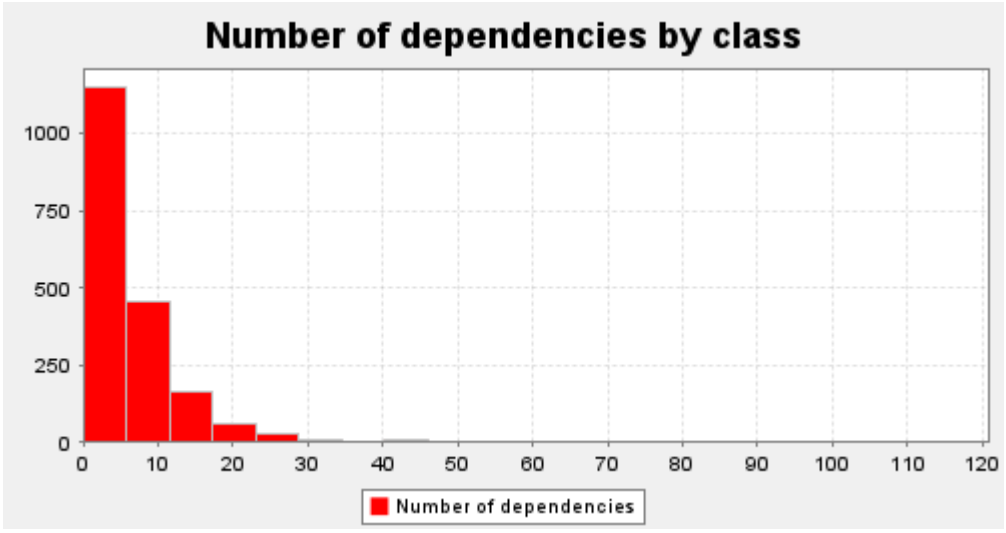
Class Metrics

	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
Min	0	0	0	0	0	0	0
Max	784	115	1398	591	1358	52	103
Avg	310,8336	6,049285	765,8389	5,225225	751,0705	3,705882	2,503445

As I said in the beginning none of the values exceed the thresholds set by the plugin, but those thresholds aren't necessarily relevant to our project, so we utilized boxplots and histograms to get a more accurate representation of our project. Of the seven boxplots, the following stood out.

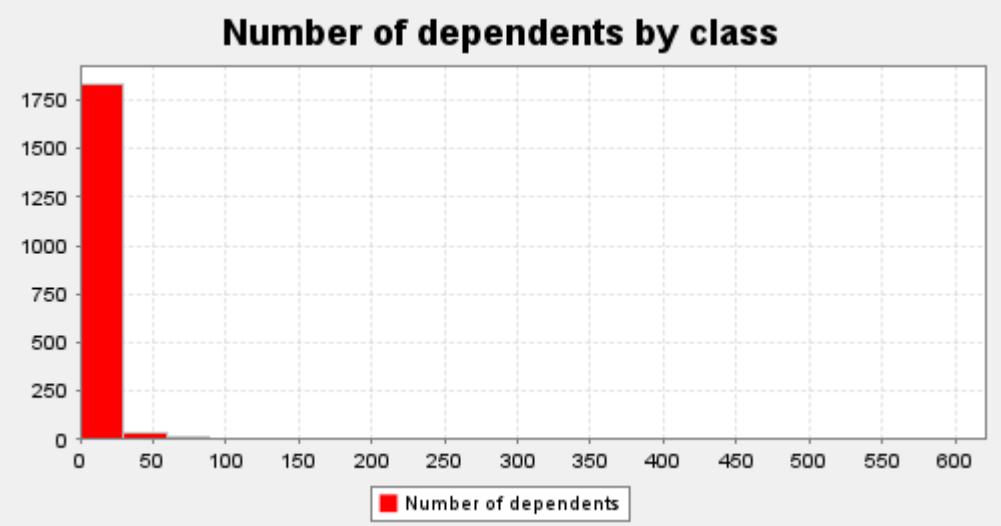


These two boxplot graphs tell us that although most classes have their number of transitive dependencies (left) and transitive dependents (right) within the thresholds of the plugin and near the average value, there is a small number of classes that stand out and have some extreme values. Transitive dependencies aren't usually problem as most of them can be solved by the compiler. That being said, many transitive dependencies could lead to the creation of cyclic dependencies and potentially cause many unwanted negative effects.



There are a few classes that have some extreme number of dependencies, being one them the org.jabref.gui.JabRefFrame class that has the highest amount of dependencies with 113. Looking at the code we can see some methods that show signs of the code smell Inappropriate Intimacy:

```java
private Button createNewEntryFromIdButton() {
    Button newEntryFromIdButton = new Button();

    newEntryFromIdButton.setGraphic(IconTheme.JabRefIcons.IMPORT.getGraphicNode());
    newEntryFromIdButton.getStyleClass().setAll("icon-button");
    newEntryFromIdButton.setFocusTraversable(false);
    newEntryFromIdButton.disableProperty().bind(ActionHelper.needsDatabase(stateManager).not());
    newEntryFromIdButton.setOnMouseClicked(event -> {
        GenerateEntryFromIdDialog entryFromId = new GenerateEntryFromIdDialog(getCurrentLibraryTab(), dialogService, prefs, taskExecutor, stateManager);

        if (entryFromIdPopOver == null) {
            entryFromIdPopOver = new PopOver(entryFromId.getDialogPane());
            entryFromIdPopOver.setTitle(Localization.lang( key: "Import by ID"));
            entryFromIdPopOver.setArrowLocation(PopOver.ArrowLocation.TOP_CENTER);
            entryFromIdPopOver.setContentNode(entryFromId.getDialogPane());
            entryFromIdPopOver.show(newEntryFromIdButton);
            entryFromId.setEntryFromIdPopOver(entryFromIdPopOver);
        } else if (entryFromIdPopOver.isShowing()) {
            entryFromIdPopOver.hide();
        } else {
            entryFromIdPopOver.setContentNode(entryFromId.getDialogPane());
            entryFromIdPopOver.show(newEntryFromIdButton);
            entryFromId.setEntryFromIdPopOver(entryFromIdPopOver);
        }
    });
    newEntryFromIdButton.setTooltip(new Tooltip(Localization.lang( key: "Import by ID")));

    return newEntryFromIdButton;
}
```



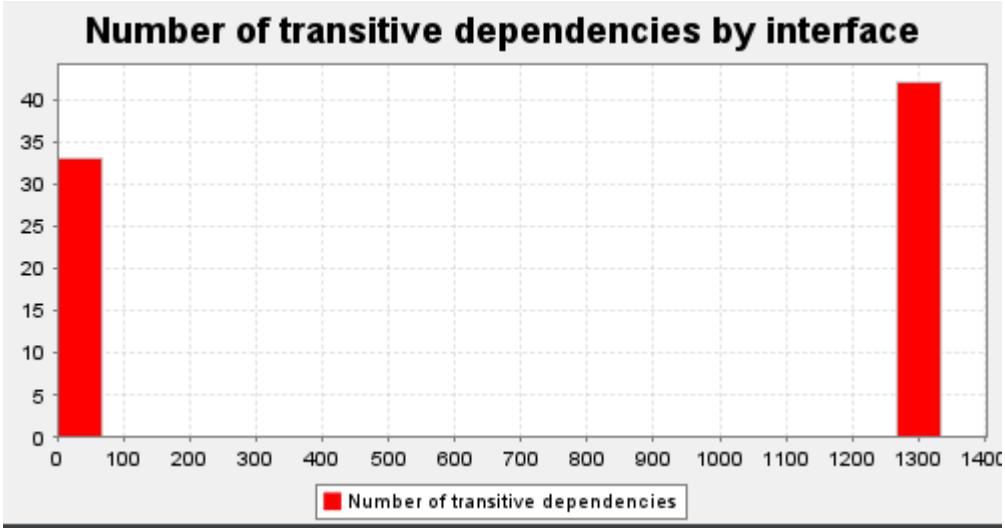
In the case of the number of dependents we are able to observe that there are some classes with extreme values.

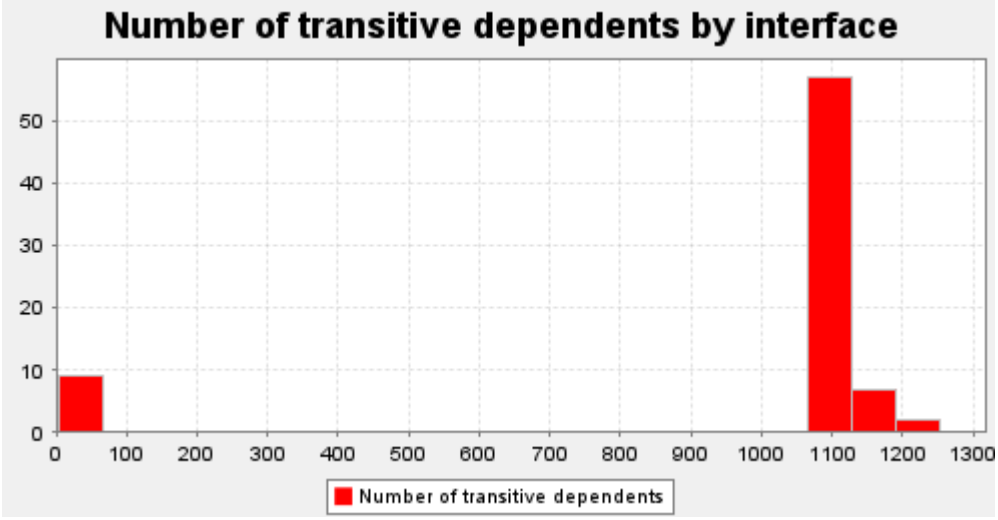
The org.jabref.model.entry.BibEntry with 591 number of dependents, almost one hundred times bigger than the average, is a very crucial class, that must be altered very carefully, as any alteration could cause a lot of classes from the project to break.

Interface Metrics

	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
Min	0	0	0	1	5	0	1
Max	784	56	1335	219	1254	30	67
Avg	386,7733	2,586667	747,3867	21,54667	993,52	1,813333	6,866667

When observing this histograms, theres some values that should concern us.

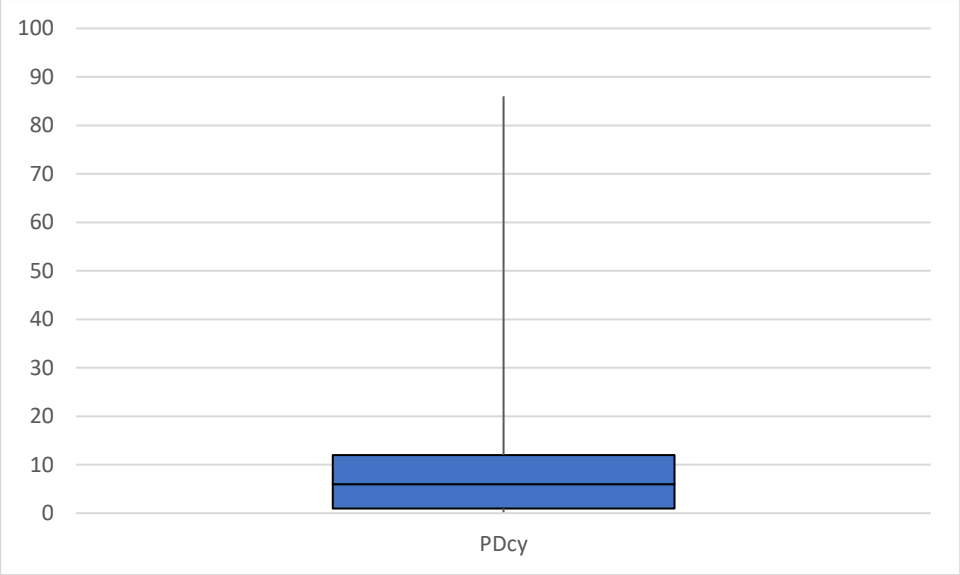


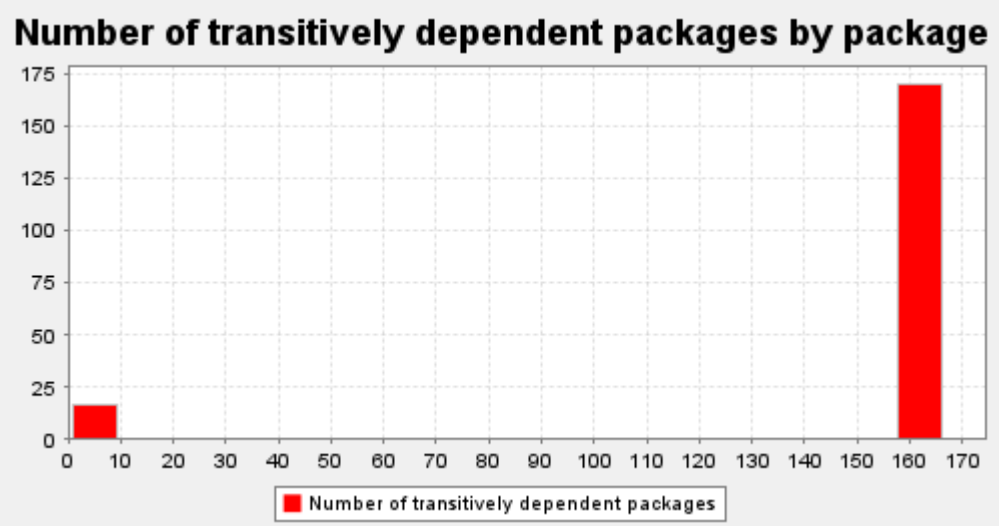


Most of our interfaces have their number of transitive dependents with values between 1121 and 1254. Which could lead to the creation of more cyclic dependencies which can cause the involving classes and interfaces to be dependent on one another, making it very hard to develop and refactor the code in the future, as one change in one class or interface would affect many and possibly lead to many unwanted negative effects.

Package Metrics

	Cyclic	PDcy	PDpt	PDpt*
Min	0	0	0	1
Max	153	86	102	166
Avg	106,1351	8,234234	8,234234	147,5348





The following two graphs stand out because, transitive dependencies seem to be a recurring issue in the project. Like I said before, transitive dependencies can lead to creation of cyclic dependencies, making it harder to develop and refactor the code in the future.

Conclusion

In conclusion, our project is acceptable and reasonable state when it comes to dependencies. However future development of code must take into consideration these metrics, as all of them showed a very high number of cyclic and transitive dependencies, which, has I said before, can be problematic since they can lead to more cyclic dependencies that can result in unwanted negative effects when trying to refactor the code.

Guilherme Pereira (57066)

Complexity Metrics

Introduction

In this report there will be an analysis of the complexity metrics regarding the JabRef project. There will be an overall exposure of the said metrics on various levels (project, modules, packages, classes & methods), but the focus will be on the classes and methods levels (mainly the latter) where a discussion of the results will take place and a correlation of those results with code smells present in the code will be made.

Across the report various abbreviations will appear:
- v(G)avg – average cyclomatic complexity
- v(G)tot – total cyclomatic complexity
- OCavg – average operation complexity
- OCmax – maximum operation complexity
- WMC – weighted method complexity
- CogC – cognitive complexity
- ev(G) – essential cyclomatic complexity
- iv(G) – design complexity
- v(G) – cyclomatic complexity

All the metrics data was extracted using the IntelliJ IDEs plugin MetricsReloaded.

Complexity

Complexity is normally measured with cyclomatic complexity, which was developed to determine the stability and level of confidence in a program. In more detail, it measures the number of linearly independent paths through a program module.
A lower cyclomatic complexity in a program means it is easier to test.

Project

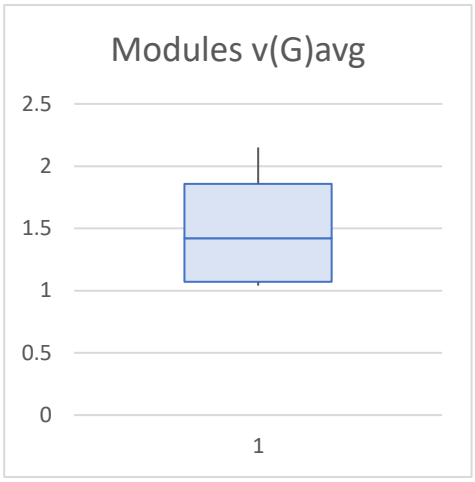
project	v(G)avg	v(G)tot
project	1,79	23243

The project as whole doesn't exceed the metric thresholds defined by the plugin, presenting an average cyclomatic complexity of 1,79.

Modules

module	v(G)avg	v(G)tot
Total		**23243**
Average	**1,79**	5810,75

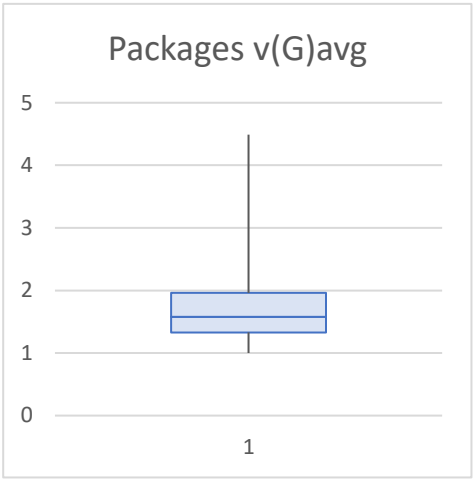
The modules also don't exceed the metric thresholds, presenting the same average cyclomatic complexity as the project. Even so, we can start to identify some modules that exceed that average, as can be seen with the boxplot on the right.



Packages

package	v(G)avg	v(G)tot
Total		**23243**
Average	**1,79**	127,01

The packages, like the project and the modules, don't exceed the thresholds defined by the plugin, maintaining an average cyclomatic complexity of 1,79. But when comparing their boxplot to the modules', there appears to be some modules with a significantly higher average cyclomatic complexity than the average, reaching a maximum of 4,49 among them.

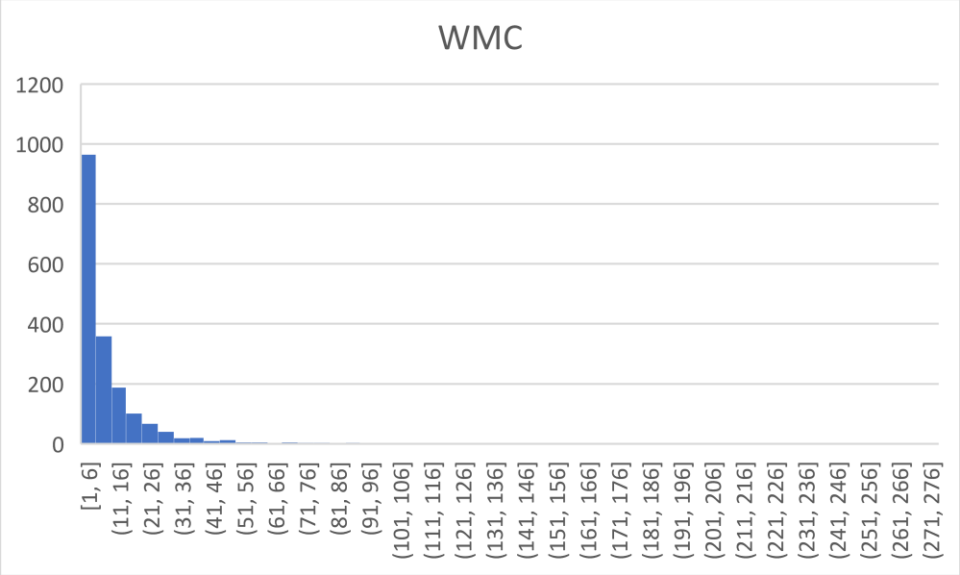


Classes

class	OCavg	OCmax	WMC
Total			**20543**
Average	1,59	3,09	10,89

Note: when dealing with the classes, the plugin defines two clear thresholds for OCavg and WMC: 3 and 30, respectively.

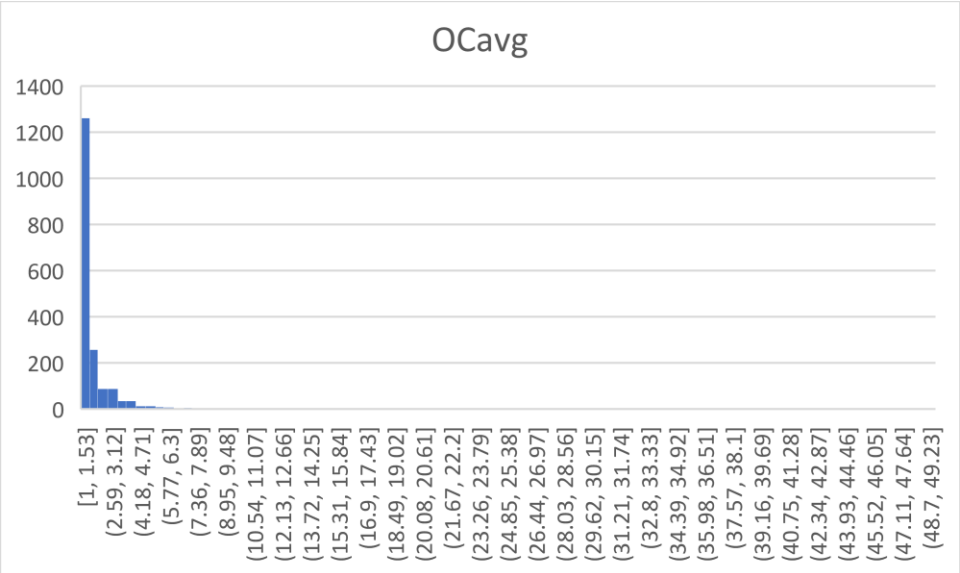
Let us look at the distribution of the weight of all the different classes:



We can see that, when compared to the threshold, most of the classes are within the boundary stipulated by the plugin, but there are still some with extreme values (outliers). Those outliers represent classes that have a high sum of complexities of their methods, so overall the class itself has a high complexity and may be difficult to get into when compared to those with a lower WMC value.

That doesn't mean that evaluating the WMC of a class is the best way to approach its complexity since that same complexity can be very well distributed by a lot of different operations that make the code itself more digestible and easier to manipulate if needed. Of course, this last statement doesn't apply to the classes where one method by itself has a score higher than 30.

Therefore, let us analyze the distribution of the average operation complexity of all the different classes:



It is evident that most of the average operation complexity values are below the average, but the outliers are still present here with values that go up to 49,0. The classes with a high average operation complexity are the problematic ones: there is most likely a bad distribution of the complexity into different methods, increasing the chance of a code smell being present then.

The three classes where the OCavg is the highest are:
- org.jabref.gui.fieldeditors.FieldNameLabel (49,00)
- org.jabref.logic.layout.format.RTFChars (25,00)
- org.jabref.logic.layout.format.GetOpenOfficeType (15,00)

Methods

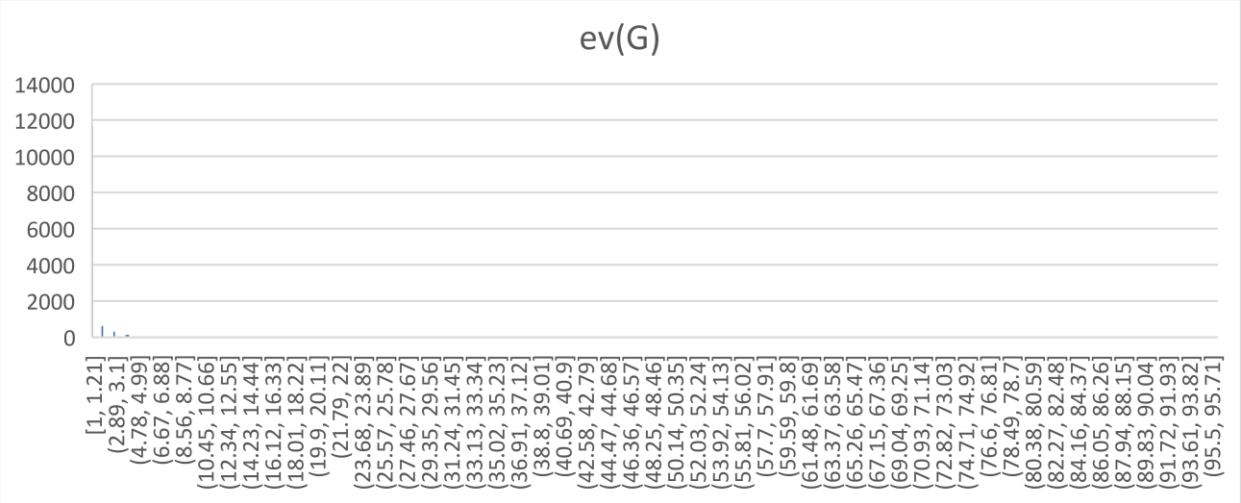
methods	CogC	ev(G)	iv(G)	v(G)
Total	**15130**	**15697**	**20735**	**23164**
Average	1,17	1,22	1,61	1,79

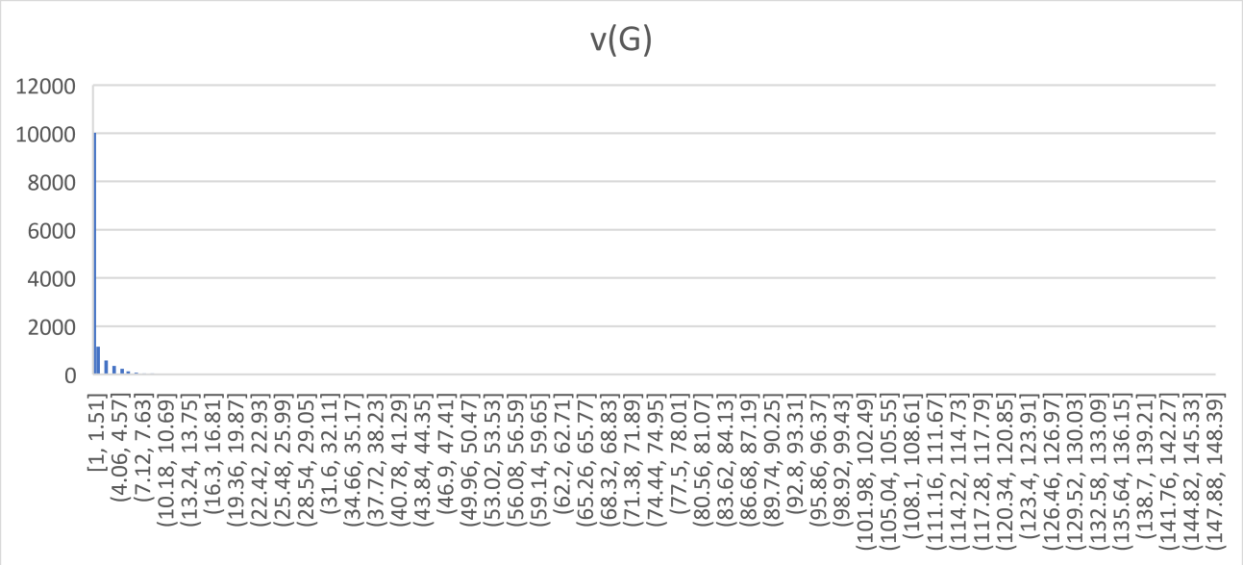
Note: when dealing with the methods, the plugin defines clear thresholds for all its subjects of evaluation: 15, 3, 8, 30, in order. The focus will be in the ev(G) (3) and v(G) (30).

ev(G) is the essential cyclomatic complexity that indicates how much complexity is left after removing well-structured complexity (for example a for loop where a condition is stated at the start of the loop). It mostly tells us how easy a method is to understand.
v(G) is the normal cyclomatic complexity stated at the beginning of the report.

Knowing this, we can say that if a method as a high v(G) but low ev(G), then it is difficult to test as a whole, but can be broken up into smaller operations to make the testing easier and, therefore, easier to refactor in case of the presence of a code smell.

Kicking off by examining the essential and non essential cyclomatic complexities ditributions of all of the methods:





We can see that in both cases, most of the methods have complexities below their respective thresholds and in range of their average. But, as expected, we can identify the existence of outliers on both histograms, leading to belive that some methods have very high complexity (way above the thresholds already mentioned), with 96 for ev(G) and 148 for v(G).

We can identify the top three methods with the highest complexities of each histogram, making sure to present both values for the methods of both histograms:

ev(G) histogram	ev(G)	v(G)
org.jabref.gui.fieldeditors.FieldNameLabel.getDescription(Field)	96	96
org.jabref.logic.citationkeypattern.BracketedPattern.getFieldValue(BibEntry,String,Character,BibDatabase)	49	56
org.jabref.logic.layout.format.RTFChars.transformSpecialCharacter(long)	46	148

v(G) histogram	ev(G)	v(G)
org.jabref.logic.layout.format.RTFChars.transformSpecialCharacter(long)	46	148
org.jabref.logic.importer.fileformat.RisImporter.importDatabase(BufferedReader)	4	110
org.jabref.gui.fieldeditors.FieldNameLabel.getDescription(Field)	96	96

In both histograms, there are two methods present in both above tables, making it evident that they are the methods more difficult to test and understand. Both also belong in the classes where OCavg was the highest.

```java
private String transformSpecialCharacter(long c) {
    if (((192 <= c) && (c <= 197)) || (c == 256) || (c == 258) || (c == 260)) {
        return "A";
    }
    if (((224 <= c) && (c <= 229)) || (c == 257) || (c == 259) || (c == 261)) {
        return "a";
    }
```

```java
public String getDescription(Field field) {
    if (field.isStandardField()) {
        StandardField standardField = (StandardField) field;
        switch (standardField) {
            case ABSTRACT:
                return Localization.lang( key: "This field is intended for recording abstracts, to be printed by a special bibliography style.");
            case ADDENDUM:
                return Localization.lang( key: "Miscellaneous bibliographic data usually printed at the end of the entry.");
```

By looking at those methods closely we can see that the complexity is due to the existence of multiple ifs, a switch statement or a combination of both. Although the first method is understandabily complex due to the high number of special characters, there is probably no other way to simplify it due to its nature, but the second is probably an example of a Switch Statement code smell due to the existence of three different if statements, each with a different switch statement.

Switch Statement code smells should be a common ocurrence when there is a high complexity method since the easiest way to increase the complexity of a method is by adding ifs and/or switch statements. The correlation above is a good example of that.

Conclusion

In conclusion, complexity metrics are a very useful way to evaluate code and to identify problem spots where the occurrence of code smells might be happening. When it comes to JabRef and its complexity, most of the project has a complexity within the thresholds specified by the plugin, but there are still some outlier classes and methods that are outside those values, being good candidates for trouble spots where code smells are probably happening.

Gabriela Costa (58625)

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:

$$\frac{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} \sum_{j=1}^{Number\ of\ attributes\ in\ a\ given\ class\ Ci} (1 - V(attribute\ j\ in\ a\ class\ Ci))}{Total\ Attributes\ in\ the\ System}$$

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:

$$\frac{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} Inherited\ attributes\ in\ a\ class\ Ci}{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} Available\ attributes\ in\ a\ class\ Ci}$$

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:

$$\frac{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system}[\sum_{j=1}^{Number\ of\ classes\ in\ the\ system} Couplings\ between\ a\ class\ Cj\ and\ Ci]}{(Number\ of\ classes\ in\ the\ system)^2 - Number\ of\ classes\ in\ the\ system}$$

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:

$$\frac{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system}\sum_{j=1}^{Number\ of\ attributes\ in\ a\ given\ class\ Ci}(1 - V(method\ j\ in\ a\ class\ Ci))}{Total\ Methods\ in\ the\ System}$$

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:

$$\frac{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} Inherited\ methods\ in\ a\ class\ Ci}{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} Available\ methods\ in\ a\ class\ Ci}$$

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 M_o , while the second will be represented as M_n.

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

$$\frac{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} M_o in\ a\ given\ class\ Ci}{\sum_{i=1}^{Number\ of\ classes\ in\ the\ system} [M_n\ in\ a\ given\ class\ Ci * Number\ of\ descendants\ of\ Ci]}$$

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.

Design Patterns

Miguel Real (55677)

Note: For brevity, it's to be assumed all packages/files mentioned henceforth are located in "src/main/java/org/jabref/".

1. Template Method Pattern (Behavioral)
   Package of origin: "logic/importer/fetcher/transformers/"

   Brief explanation of the found pattern: The abstract AbstractQueryTransformer class includes a transform() method with several implementations (each of which can be seen as independent instances of this pattern). Each of these utilize several abstract methods (*steps*) declared in the AbstractQueryTransformer class. It then falls upon the concrete classes in this package to implement each step of the algorithm in their own way in order to produce a functional algorithm when the transform() method is called.

   This is a textbook example of the Template Method pattern, which is based on a method (the *template*) in an abstract class that defines an algorithm by calling other methods (the *steps*). These methods (usually abstract, though not always) will then be implemented by subclasses in order to tailor the algorithm to their needs.

```java
private Optional<String> transform(FieldQueryNode query) {
    String term = query.getTextAsString();
    switch (query.getFieldAsString()) {
        case "author" -> {
            return Optional.of(handleAuthor(term));
        }
        case "title" -> {
            return Optional.of(handleTitle(term));
        }
        case "journal" -> {
            return Optional.of(handleJournal(term));
        }
        case "year" -> {
            String s = handleYear(term);
            return s.isEmpty() ? Optional.empty() : Optional.of(s);
        }
        case "year-range" -> {
            String s = handleYearRange(term);
            return s.isEmpty() ? Optional.empty() : Optional.of(s);
        }
        case "doi" -> {
            String s = handleDoi(term);
            return s.isEmpty() ? Optional.empty() : Optional.of(s);
        }
        case NO_EXPLICIT_FIELD -> {
            return handleUnFieldedTerm(term);
        }
        default -> {
            // Just add unknown fields as default
            return handleOtherField(query.getFieldAsString(), term);
        }
    }
}
```

One of the implementations of the transform() method. Several of the methods called (handleAuthor(), handleTitle(), handleJournal(), etc.) are defined as abstract in this class and are then implemented in subclasses.

Here we can see a concrete implementation of a step, handleAuthor(), from ScholarQueryTransformer.java:

```java
@Override
protected String handleAuthor(String author) { return createKeyValuePair( fieldAsString: "author", author); }
```

2. Builder Pattern (Creational)
   Package of origin: "model/entry/"

Brief explanation of the found pattern: We can observe a Builder pattern in the interactions between the BibEntryType (*Product*), BibEntryTypeBuilder (*Builder*), and BibEntryTypesManager (*Director*) classes. In the parse() method of the latter class a builder object is created and given constraints on the product it will build. A product is then returned by calling the builder class's build() method, which returns a BibEntryType with the characteristics set down by the director class previously.

Although this behaviour is consistent with the Builder pattern, it is not a textbook implementation of it, as that would involve a slightly different approach, with a global Builder

variable in the Director, a build/make method in the Director which given the desired product characteristics would call the necessary component-building methods of the builder class (using the aforementioned builder variable), and a builder class whose only purpose in the pattern is to hold the operations responsible for the creation of a product's component parts.

```
29 @     public static Optional<BibEntryType> parse(String comment) {
```

```
46              BibEntryTypeBuilder entryTypeBuilder = new BibEntryTypeBuilder()
47                      .withType(type)
48                      .withImportantFields(FieldFactory.parseFieldList(optFields))
49                      .withRequiredFields(FieldFactory.parseOrFieldsList(reqFields));
50              return Optional.of(entryTypeBuilder.build());
```

parse() method of the BibEntryTypesManager class, abridged to only include code relevant to the pattern in discussion (lines 46-50). A builder class is created and given directions on the product it will build.

```
75      public BibEntryType build() {
76          // Treat required fields as important ones
77          Stream<BibField> requiredAsImportant = requiredFields.stream()
78                                                               .flatMap(Set::stream)
79                                                               .map(field -> new BibField(field, FieldPriority.IMPORTANT));
80          Set<BibField> allFields = Stream.concat(fields.stream(), requiredAsImportant).collect(Collectors.toCollection(LinkedHashSet::new));
81          return new BibEntryType(type, allFields, requiredFields);
82      }
83  }
```

Excerpt from the BibEntryTypeBuilder class, the methods seen are a sample of the various existing methods to constrain the characteristics of the product that will be created.

```
35 @    public BibEntryTypeBuilder withImportantFields(Collection<Field> newFields) {
36          this.fields = Streams.concat(fields.stream(), newFields.stream().map(field -> new BibField(field, FieldPriority.IMPORTANT)))
37                               .collect(Collectors.toCollection(LinkedHashSet::new));
38          return this;
39      }
40
41      public BibEntryTypeBuilder withImportantFields(Field... newFields) {
42          return withImportantFields(Arrays.asList(newFields));
43      }
44
45 @    public BibEntryTypeBuilder withDetailFields(Collection<Field> newFields) {
46          this.fields = Streams.concat(fields.stream(), newFields.stream().map(field -> new BibField(field, FieldPriority.DETAIL)))
47                               .collect(Collectors.toCollection(LinkedHashSet::new));
48          return this;
49      }
50
51      public BibEntryTypeBuilder withDetailFields(Field... fields) { return withDetailFields(Arrays.asList(fields)); }
```

build() method from BibEntryTypeBuilder. This method (usually part of the Director class) will create a Product based on the characteristics dictated by the Director.

3. Prototype Pattern (Creational)
File of origin: "logic/integrity/IntegrityMessage.java"

Brief explanation of the found pattern: IntegrityMessage is a small class focused around 3 private variables that store its characteristics (entry, field, and message). The constructor is used as a setter, receiving 3 parameters that correspond to the 3 variables. It includes some relatively unremarkable methods, like getter methods for the 3 variables and toString(), equals() and hashCode() methods which do what one would expect. More interestingly though, it also includes a clone() method, which is the one relevant for this design pattern, its purpose is to create another instance of the IntegrityMessage class with the exact same fields as the current one (a clone).

This is a textbook implementation of the Prototype pattern, which is defined by an object having a clone() method whose function is to create another instance of the same class with identical field values, essentially a clone (as suggested by the method's namesake).

```
8   public final class IntegrityMessage implements Cloneable {
9       private final BibEntry entry;
10      private final Field field;
11      private final String message;
12
13      public IntegrityMessage(String message, BibEntry entry, Field field) {
14          this.message = message;
15          this.entry = entry;
16          this.field = field;
17      }
```

The 3 main variables of the IntegrityMessage class along with its constructor (which works as a setter for the aforementioned variables).

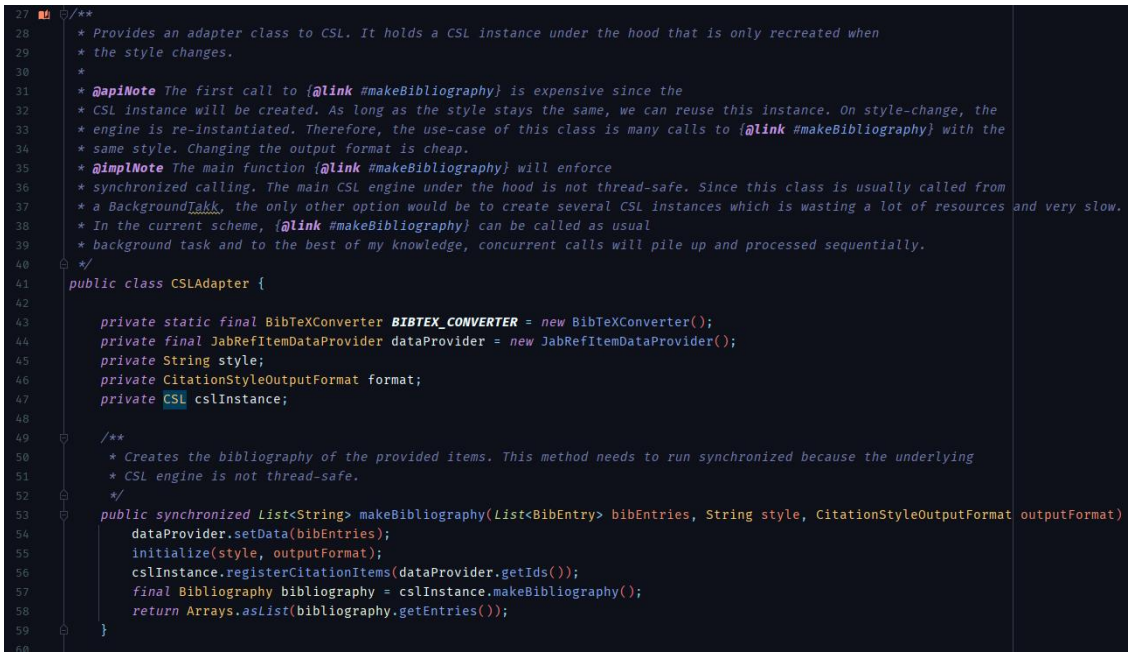
```
36      @Override
37  @   public Object clone() {
38          return new IntegrityMessage(message, entry, field);
39      }
```

clone() method of the IntegrityMessage class, its purpose is to create another instance of IntegrityMessage with the exact same fields as the current one (a clone).

Gonçalo Virgínia (56773)

1. Adapter Pattern (Structural)

src/main/java/org/jabref/logic/citationstyle/CSLAdapter.java



```java
/**
 * Provides an adapter class to CSL. It holds a CSL instance under the hood that is only recreated when
 * the style changes.
 *
 * @apiNote The first call to {@link #makeBibliography} is expensive since the
 * CSL instance will be created. As long as the style stays the same, we can reuse this instance. On style-change, the
 * engine is re-instantiated. Therefore, the use-case of this class is many calls to {@link #makeBibliography} with the
 * same style. Changing the output format is cheap.
 * @implNote The main function {@link #makeBibliography} will enforce
 * synchronized calling. The main CSL engine under the hood is not thread-safe. Since this class is usually called from
 * a BackgroundTakk, the only other option would be to create several CSL instances which is wasting a lot of resources and very slow.
 * In the current scheme, {@link #makeBibliography} can be called as usual
 * background task and to the best of my knowledge, concurrent calls will pile up and processed sequentially.
 */
public class CSLAdapter {

    private static final BibTeXConverter BIBTEX_CONVERTER = new BibTeXConverter();
    private final JabRefItemDataProvider dataProvider = new JabRefItemDataProvider();
    private String style;
    private CitationStyleOutputFormat format;
    private CSL cslInstance;

    /**
     * Creates the bibliography of the provided items. This method needs to run synchronized because the underlying
     * CSL engine is not thread-safe.
     */
    public synchronized List<String> makeBibliography(List<BibEntry> bibEntries, String style, CitationStyleOutputFormat outputFormat)
        dataProvider.setData(bibEntries);
        initialize(style, outputFormat);
        cslInstance.registerCitationItems(dataProvider.getIds());
        final Bibliography bibliography = cslInstance.makeBibliography();
        return Arrays.asList(bibliography.getEntries());
    }
```

```java
    /**
     * Initialized the static CSL instance if needed.
     *
     * @param newStyle  journal style of the output
     * @param newFormat usually HTML or RTF.
     * @throws IOException An error occurred in the underlying JavaScript framework
     */
    private void initialize(String newStyle, CitationStyleOutputFormat newFormat) throws IOException {
        if ((cslInstance == null) || !Objects.equals(newStyle, style)) {
            // lang and forceLang are set to the default values of other CSL constructors
            cslInstance = new CSL(dataProvider, new JabRefLocaleProvider(),
                    new DefaultAbbreviationProvider(), newStyle,  lang: "en-US");
            style = newStyle;
        }

        if (!Objects.equals(newFormat, format)) {
            cslInstance.setOutputFormat(newFormat.getFormat());
            format = newFormat;
        }
    }
```

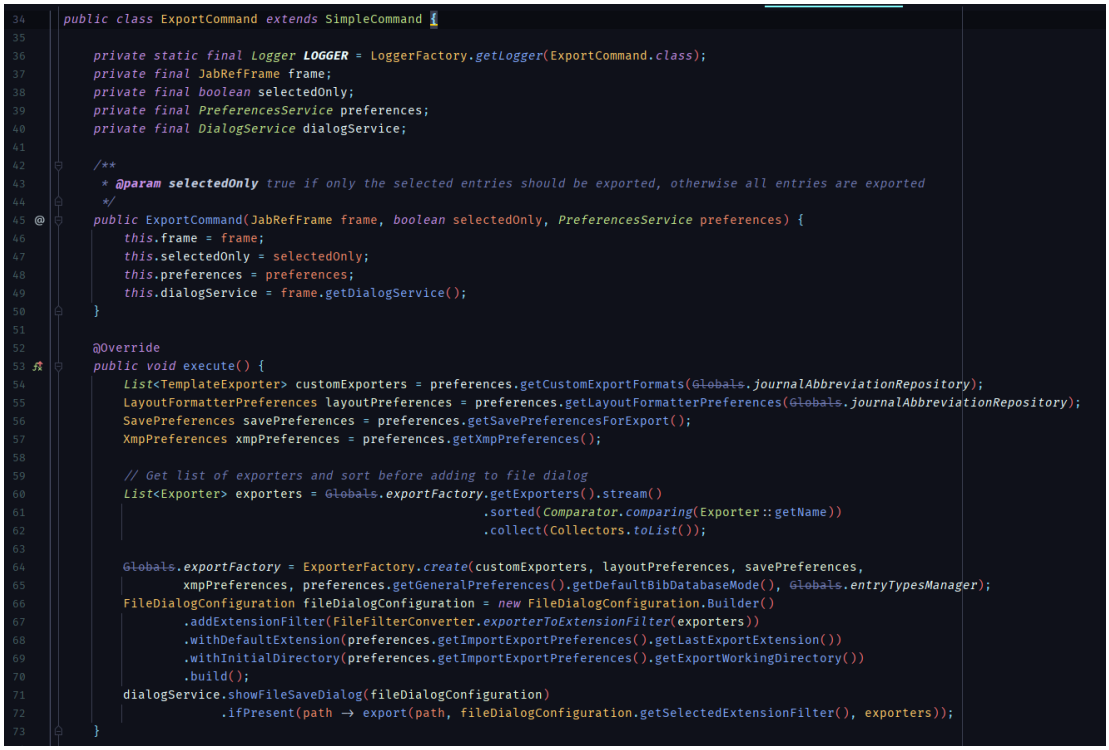
Description: The CSLAdapter class, as its name suggests, is a fairly standard implementation of the identified design pattern, as its only responsibility is to provide a conversion of the contents obtained from the instantiated CSL (Citation Style Language) object into, as it stands, a List of Bibliography Entries.

The main noteworthy difference between this class and the typical adapter implementation, is that the present class doesn't incorporate a constructor which receives a CSL instance, instead, the makeBibliography method (which is the only public method in the class, seen in the first code snippet) uses a private initialize method with 2 arguments received from the aforementioned public method (as seen in the second code snippet).

Additionally, although not particularly connected to the design pattern itself, the class also implements its own private JabRefItemDataProvider class, which implements the ItemDataProvider interface required as an argument for the instantiation of the adapted CSL object.

2. Command Pattern (Behavioral)

src/main/java/org/jabref/gui/exporter/ExportCommand.java



```java
public class ExportCommand extends SimpleCommand {

    private static final Logger LOGGER = LoggerFactory.getLogger(ExportCommand.class);
    private final JabRefFrame frame;
    private final boolean selectedOnly;
    private final PreferencesService preferences;
    private final DialogService dialogService;

    /**
     * @param selectedOnly true if only the selected entries should be exported, otherwise all entries are exported
     */
    public ExportCommand(JabRefFrame frame, boolean selectedOnly, PreferencesService preferences) {
        this.frame = frame;
        this.selectedOnly = selectedOnly;
        this.preferences = preferences;
        this.dialogService = frame.getDialogService();
    }

    @Override
    public void execute() {
        List<TemplateExporter> customExporters = preferences.getCustomExportFormats(Globals.journalAbbreviationRepository);
        LayoutFormatterPreferences layoutPreferences = preferences.getLayoutFormatterPreferences(Globals.journalAbbreviationRepository);
        SavePreferences savePreferences = preferences.getSavePreferencesForExport();
        XmpPreferences xmpPreferences = preferences.getXmpPreferences();

        // Get list of exporters and sort before adding to file dialog
        List<Exporter> exporters = Globals.exportFactory.getExporters().stream()
                                                        .sorted(Comparator.comparing(Exporter::getName))
                                                        .collect(Collectors.toList());

        Globals.exportFactory = ExporterFactory.create(customExporters, layoutPreferences, savePreferences,
                xmpPreferences, preferences.getGeneralPreferences().getDefaultBibDatabaseMode(), Globals.entryTypesManager);
        FileDialogConfiguration fileDialogConfiguration = new FileDialogConfiguration.Builder()
                .addExtensionFilter(FileFilterConverter.exporterToExtensionFilter(exporters))
                .withDefaultExtension(preferences.getImportExportPreferences().getLastExportExtension())
                .withInitialDirectory(preferences.getImportExportPreferences().getExportWorkingDirectory())
                .build();
        dialogService.showFileSaveDialog(fileDialogConfiguration)
                     .ifPresent(path -> export(path, fileDialogConfiguration.getSelectedExtensionFilter(), exporters));
    }
```

src/main/java/org/jabref/gui/importer/ImportCommand.java

```java
/**
 * Perform import operation
 */
public class ImportCommand extends SimpleCommand {

    private final JabRefFrame frame;
    private final boolean openInNew;
    private final DialogService dialogService;
    private final PreferencesService preferences;

    /**
     * @param openInNew Indicate whether the entries should import into a new database or into the currently open one.
     */
    public ImportCommand(JabRefFrame frame, boolean openInNew, PreferencesService preferences, StateManager stateManager) {
        this.frame = frame;
        this.openInNew = openInNew;
        this.preferences = preferences;

        if (!openInNew) {
            this.executable.bind(needsDatabase(stateManager));
        }

        this.dialogService = frame.getDialogService();
    }

    @Override
    public void execute() {
        SortedSet<Importer> importers = Globals.IMPORT_FORMAT_READER.getImportFormats();

        FileDialogConfiguration fileDialogConfiguration = new FileDialogConfiguration.Builder()
                .addExtensionFilter(FileFilterConverter.ANY_FILE)
                .addExtensionFilter(FileFilterConverter.forAllImporters(importers))
                .addExtensionFilter(FileFilterConverter.importerToExtensionFilter(importers))
                .withInitialDirectory(preferences.getImportExportPreferences().getImportWorkingDirectory())
                .build();
        dialogService.showFileOpenDialog(fileDialogConfiguration)
                     .ifPresent(path -> doImport(path, importers, fileDialogConfiguration.getSelectedExtensionFilter()));
    }
```

Description: The 2 classes specified above (ExportCommand and ImportCommand) are virtually perfect examples of the Command Pattern, since they are both extensions of the same SimpleCommand abstract class, although, this abstract class itself is an extension of the CommandBase abstract class that implements the base Command interface, the last 2 aforementioned being from a completely separate external library.

They both implement a standard execute method, which is called after the desired commands' instantiation, which will contain all the information about the request in a stand-alone object.

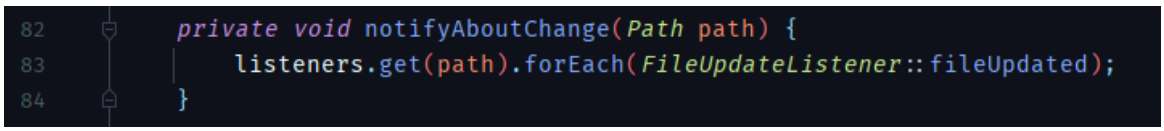
This pattern is especially useful for GUI elements (which happens to be the exact use here), since multiple ways to access similar actions don't have to be tied down with their own implementation of the same operation. This enables extensibility by reusing or creating a new command and facilitates the implementation of other related functionalities, such as a command queue and undoing operations with the information stored inside each command.

3. Observer Pattern (Behavioral)

src/main/java/org/jabref/gui/util/DefaultFileUpdateMonitor.java

```java
/**
 * This class monitors a set of files for changes. Upon detecting a change it notifies the registered {@link
 * FileUpdateListener}s.
 * <p>
 * Implementation based on https://stackoverflow.com/questions/16251273/can-i-watch-for-single-file-change-with-watchservice-not-the-whole-directory
 */
public class DefaultFileUpdateMonitor implements Runnable, FileUpdateMonitor {

    private static final Logger LOGGER = LoggerFactory.getLogger(DefaultFileUpdateMonitor.class);

    private final Multimap<Path, FileUpdateListener> listeners = ArrayListMultimap.create( expectedKeys: 20,  expectedValuesPerKey: 4);
    private volatile WatchService watcher;
    private final AtomicBoolean notShutdown = new AtomicBoolean( initialValue: true);
    private Optional<JabRefException> filesystemMonitorFailure;
```

```java
    @Override
    public void run() {
        try (WatchService watcher = FileSystems.getDefault().newWatchService()) {
            this.watcher = watcher;
            filesystemMonitorFailure = Optional.empty();

            while (notShutdown.get()) {
                WatchKey key;
                try {
                    key = watcher.take();
                } catch (InterruptedException | ClosedWatchServiceException e) {
                    return;
                }

                for (WatchEvent<?> event : key.pollEvents()) {
                    WatchEvent.Kind<?> kind = event.kind();

                    if (kind == StandardWatchEventKinds.OVERFLOW) {
                        Thread.yield();
                        continue;
                    } else if (kind == StandardWatchEventKinds.ENTRY_CREATE || kind == StandardWatchEventKinds.ENTRY_MODIFY) {
                        // We only handle "ENTRY_CREATE" and "ENTRY_MODIFY" here, so the context is always a Path
                        /unchecked/
                        WatchEvent<Path> ev = (WatchEvent<Path>) event;
                        Path path = ((Path) key.watchable()).resolve(ev.context());
                        notifyAboutChange(path);
                    }
                    key.reset();
                }
                Thread.yield();
            }
        } catch (IOException e) {
            filesystemMonitorFailure = Optional.of(new WatchServiceUnavailableException(e.getMessage(),
                                                                                        e.getLocalizedMessage(), e.getCause()));
            LOGGER.warn(filesystemMonitorFailure.get().getLocalizedMessage(), e);
        }
    }
```



```java
    private void notifyAboutChange(Path path) {
        listeners.get(path).forEach(FileUpdateListener::fileUpdated);
    }
```

Description: Standard implementation of the observer pattern. In this case, the DefaultFileUpdateMonitor class contains a map of FileUpdateListener(s), whose keys are their corresponding Path, which is used as an argument for the notifyAboutChange method (visible in the third code snippet and used in the run method in the second snippet), thus, only a handful of listeners with the specified Path will get updated.

João Vieira (56971)

Design Pattern #10- Facade Pattern

Found:

@ SE2122_55677_56773__56971_57066_58655/src/main/java
/org/jabref/cli/ArgumentProcessor.java

The class ArgumentProcessor is first instantiated in JabRefMain and I believe this class follows the Facade Design Pattern because it provides a convenient access for the client to interact with JabRef functionalities.

```
try {
    // Process arguments
    ArgumentProcessor argumentProcessor = new ArgumentProcessor(arguments, ArgumentProcessor.Mode.INITIAL_START, preferences);
    // Check for running JabRef
    if (!handleMultipleAppInstances(arguments, preferences) || argumentProcessor.shouldShutDown()) {
```

The ArgumentProcessor class is responsible for interacting with many different packages and classes (i.e.: gui, logic, model, preferences, PreferenceService.java, JabRefCLI.java, etc.). With its methods this class interacts with all the necessary objects to fulfill the client's request, without forcing the client to interact with all the different objects. As we can see in the code below:

```
public ArgumentProcessor(String[] args, Mode startupMode, PreferencesService preferencesService) throws org.apache.commons.cli.ParseException {
    cli = new JabRefCLI(args);
    this.startupMode = startupMode;
    this.preferencesService = preferencesService;
    parserResults = processArguments();
}
```

```
private static Optional<ParserResult> importBibtexToOpenBase(String argument, ImportFormatPreferences importFormatPreferences) {
    BibtexParser parser = new BibtexParser(importFormatPreferences, new DummyFileUpdateMonitor());
    try {
        List<BibEntry> entries = parser.parseEntries(argument);
        ParserResult result = new ParserResult(entries);
        result.setToOpenTab();
        return Optional.of(result);
    } catch (ParseException e) {
        System.err.println(Localization.lang( key: "Error occurred when parsing entry") + ": " + e.getLocalizedMessage());
        return Optional.empty();
    }
```

```
private void saveDatabase(BibDatabase newBase, String subName) {
    try {
        System.out.println(Localization.lang( key: "Saving") + ": " + subName);
        GeneralPreferences generalPreferences = preferencesService.getGeneralPreferences();
        SavePreferences savePreferences = preferencesService.getSavePreferences();
        AtomicFileWriter fileWriter = new AtomicFileWriter(Path.of(subName), generalPreferences.getDefaultEncoding());
        BibWriter bibWriter = new BibWriter(fileWriter, OS.NEWLINE);
        BibDatabaseWriter databaseWriter = new BibtexDatabaseWriter(bibWriter, generalPreferences, savePreferences, Globals.entryTypesManager);
        databaseWriter.saveDatabase(new BibDatabaseContext(newBase));

        // Show just a warning message if encoding did not work for all characters:
        if (fileWriter.hasEncodingProblems()) {
            System.err.println(Localization.lang( key: "Warning") + ": "
                    + Localization.lang(
                    key: "The chosen encoding '%0' could not encode the following characters:",
                    generalPreferences.getDefaultEncoding().displayName())
                    + " " + fileWriter.getEncodingProblems());
        }
    } catch (IOException ex) {
        System.err.println(Localization.lang( key: "Could not save file.") + "\n" + ex.getLocalizedMessage());
    }
}
```

Design Pattern #11- Builder Pattern

Found:
@SE2122_55677_56773__56971_57066_58655/src/main/java
/org/jabref/gui/actions/ActionFactory.java

```java
package org.jabref.gui.actions;

import ...

/**
 * Helper class to create and style controls according to an {@link Action}.
 */
public class ActionFactory {

    private static final Logger LOGGER = LoggerFactory.getLogger(ActionFactory.class);

    private final KeyBindingRepository keyBindingRepository;

    public ActionFactory(KeyBindingRepository keyBindingRepository) {
        this.keyBindingRepository = Objects.requireNonNull(keyBindingRepository);
    }

    /**
     * For some reason the graphic is not set correctly by the {@link ActionUtils} class, so we have to fix this by hand
     */
    private static void setGraphic(MenuItem node, Action action) {
        node.graphicProperty().unbind();
        action.getIcon().ifPresent(icon -> node.setGraphic(icon.getGraphicNode()));
    }
```

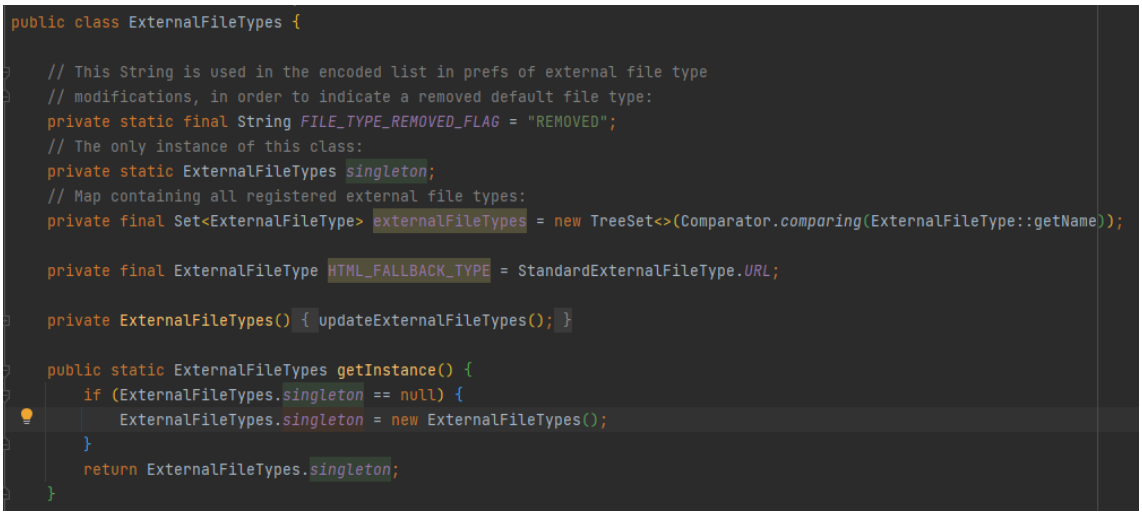
```java
public MenuItem configureMenuItem(Action action, Command command, MenuItem menuItem) {
    ActionUtils.configureMenuItem(new JabRefAction(action, command, keyBindingRepository, Sources.FromMenu), menuItem);
    setGraphic(menuItem, action);

    // Show tooltips
    if (command instanceof SimpleCommand) {
        EasyBind.subscribe(
                ((SimpleCommand) command).statusMessageProperty(),
                message -> {
                    Label label = getAssociatedNode(menuItem);
                    if (label != null) {
                        label.setMouseTransparent(false);
                        if (StringUtil.isBlank(message)) {
                            label.setTooltip(null);
                        } else {
                            label.setTooltip(new Tooltip(message));
                        }
                    }
                }
        );
    }

    return menuItem;
}

public MenuItem createMenuItem(Action action, Command command) {
    MenuItem menuItem = new MenuItem();
    configureMenuItem(action, command, menuItem);
    return menuItem;
}

public CheckMenuItem createCheckMenuItem(Action action, Command command, boolean selected) {
    CheckMenuItem checkMenuItem = ActionUtils.createCheckMenuItem(new JabRefAction(action, command, keyBindingRepository, Sources.FromMenu));
    checkMenuItem.setSelected(selected);
    setGraphic(checkMenuItem, action);

    return checkMenuItem;
}
```

I believed this to be an implementation of The Build Pattern, as this class allows us to create of objects that implement the Action interface step by step, and customize it to suit our needs using the same construction code.

Design Pattern #12- Singleton Pattern

Found:

@@SE2122_55677_56773__56971_57066_58655/src/main/java
/org/jabref/gui/externalfiletype/ExternalFileTypes.java



```java
public class ExternalFileTypes {

    // This String is used in the encoded list in prefs of external file type
    // modifications, in order to indicate a removed default file type:
    private static final String FILE_TYPE_REMOVED_FLAG = "REMOVED";
    // The only instance of this class:
    private static ExternalFileTypes singleton;
    // Map containing all registered external file types:
    private final Set<ExternalFileType> externalFileTypes = new TreeSet<>(Comparator.comparing(ExternalFileType::getName));

    private final ExternalFileType HTML_FALLBACK_TYPE = StandardExternalFileType.URL;

    private ExternalFileTypes() { updateExternalFileTypes(); }

    public static ExternalFileTypes getInstance() {
        if (ExternalFileTypes.singleton == null) {
            ExternalFileTypes.singleton = new ExternalFileTypes();
        }
        return ExternalFileTypes.singleton;
    }
```

This is clear example o the Singleton design pattern, as it ensures that this class will only have one instance, while allowing us a public access point to this instance.

Guilherme Pereira (57066)

1- Observer pattern

found @ src/main/java/org/jabref/gui/LibraryTab/IndexUpdateListener.java

```
private final IndexingTaskManager indexingTaskManager = new IndexingTaskManager(Globals.TASK_EXECUTOR);
```

```
private class IndexUpdateListener {

    public IndexUpdateListener() {
        try {
            indexingTaskManager.addToIndex(PdfIndexer.of(bibDatabaseContext, preferencesService.getFilePreferences()), bibDatabaseContext);
        } catch (IOException e) {
            LOGGER.error("Cannot access lucene index", e);
        }
    }

    @Subscribe
    public void listen(EntriesAddedEvent addedEntryEvent) {
        try {
            PdfIndexer pdfIndexer = PdfIndexer.of(bibDatabaseContext, preferencesService.getFilePreferences());
            for (BibEntry addedEntry : addedEntryEvent.getBibEntries()) {
                indexingTaskManager.addToIndex(pdfIndexer, addedEntry, bibDatabaseContext);
            }
        } catch (IOException e) {
            LOGGER.error("Cannot access lucene index", e);
        }
    }
```

I identified this as an implementation of the observer design pattern because a subject (public class LibraryTab that holds the private class IndexUpdateListener) keeps an observer (indexingTaskManager) and, when an event happens, the subject is notified of it and calls the correct method from the observer so he can be updated accordingly, as show in the snippet above.

2- Factory pattern

found @ src/main/java/org/jabref/model/entry/field/FieldFactory.java

```
17    public class FieldFactory {
```

```
public static Set<Field> getStandardFieldsWithCitationKey() {
    EnumSet<StandardField> allFields = EnumSet.allOf(StandardField.class);

    LinkedHashSet<Field> standardFieldsWithBibtexKey = new LinkedHashSet<>( initialCapacity: allFields.size() + 1);
    standardFieldsWithBibtexKey.add(InternalField.KEY_FIELD);
    standardFieldsWithBibtexKey.addAll(allFields);

    return standardFieldsWithBibtexKey;
}

public static Set<Field> getBookNameFields() {
    return getFieldsFiltered(field -> field.getProperties().contains(FieldProperty.BOOK_NAME));
}

public static Set<Field> getPersonNameFields() {
    return getFieldsFiltered(field -> field.getProperties().contains(FieldProperty.PERSON_NAMES));
}
```

I identified this as an impure implementation of the factory pattern (since it deviates from the textbook definition). A Product (interface Field, present in the same package of the Creator) has different

implementations, as in different field properties, which will work as concrete products. A Creator (class FieldFactory) has various methods that return those different types of fields (mostly as sets of those), therefore returning existing objects with a specific property. It deviates from the textbook definition because the Creator doesn't have methods that return new objects, but sets of existing ones with a specific property, and the Product doesn't have different implementations per se, but has a specific property attached, simulating polymorphism.

3- Singleton pattern

found @ src/main/java/org/jabref/preferences/JabRefPreferences.java

```
// The only instance of this class:
private static JabRefPreferences singleton;
```

```
// The constructor is made private to enforce this as a singleton class:
private JabRefPreferences() {
```

```
public static JabRefPreferences getInstance() {
    if (JabRefPreferences.singleton == null) {
        JabRefPreferences.singleton = new JabRefPreferences();
    }
    return JabRefPreferences.singleton;
}
```

I identified this as a textbook implementation of the singleton pattern. It has the two steps needed for it: has a private constructor JabRafPreferences (so other objects can't use the `new` operator with the class) and there's a static method that calls the private constructor to assign the desired object to a static variable, so all calls to the method return the cached object.

Gabriela Costa (58625)

1- Façade Pattern – Structural Design Pattern

Found at at /src/main/java/org/jabref/model/metadata/MetaData.java

The MetaData class holds in itself objects from multiple other classes. Many of those objects qualify as metadata and together they form a highly complex subsystem with many functionalities. A few examples of metadata objects can be seen below:

```
55      private final EventBus eventBus = new EventBus();
62      private SaveOrderConfig saveOrderConfig;
64      private FieldFormatterCleanups saveActions;
68      private final ContentSelectors contentSelectors = new ContentSelectors();
```

Access to certain functionalities of these objects, as well as access to the objects themselves is then granted by several methods in the MetaData class. An example of this can be seen in the following methods:

-Access to objects:

```
79      public Optional<SaveOrderConfig> getSaveOrderConfig() {
80          return Optional.ofNullable(saveOrderConfig);
81      }
180     public ContentSelectors getContentSelectors() {
181         return contentSelectors;
182     }
```

-Access to certain functionalities of the objects:

```
184     public List<ContentSelector> getContentSelectorList() {
185         return contentSelectors.getContentSelectors();
186     }
301     public void registerListener(Object listener) {
302         this.eventBus.register(listener);
303     }
```

The MetaData class works as a façade class which wraps metadata representing objects, providing a simple communication link between the rest of the system and the wrapped classes, as well as simplifying the interaction with a subsystem that is comprised of several moving parts. The façade class is then used in multiple places throughout the code, replacing what would have to be a systematic instantiation of all of the “encapsulated” classes.

2- Template Method – Behavioral Design Pattern

Found at /src/main/java/org/jabref/model/groups/AbstractGroup.java

The template method design pattern allows a superclass to define the structure of a method while delegating the specifics of the implementation to subclasses. This results in a public method in the superclass that, although it might include additional logic and even other non-abstract methods, will include at least one abstract method.

The example found in the AbstractGroup class is a rather simplified implementation of this. The isMatch() method uses the abstract method contains() - which is implemented independently by each AbstractGroup subclass - and works as a template method.

Other implementations of this design pattern can have more complex template methods, with multiple abstract methods within them as well as some logic.

3- Prototype – Creational Design Pattern

Found at /src/main/java/org/jabref/model/entry/BibEntry.java

The prototype design pattern allows the cloning of specific objects while still making the resulting object independent of its origin. This is implemented through a cloning method inside the class to be cloned, which returns a new object based on its own fields. Classes that allow cloning of their objects should share a common interface for this purpose.

The BibEntry class implements the *Cloneable* interface and overrides the clone() method, thus being a prototype class.
As seen below, the BibEntry class returns an object that is similar to the original one in every field of the class, with the exception of the id, which is generated when the new clone object is created.

```java
    /**
     * Returns a clone of this entry. Useful for copying.
     * This will set a new ID for the cloned entry to be able to distinguish both copies.
     */
    @Override
    public Object clone() {
        BibEntry clone = new BibEntry(type.getValue());
        clone.fields = FXCollections.observableMap(new ConcurrentHashMap<>(fields));
        clone.commentsBeforeEntry = commentsBeforeEntry;
        clone.parsedSerialization = parsedSerialization;
        clone.changed = changed;
        return clone;
    }
```

Code Smells

Miguel Real (55677)

Note: Before reviewing the code smells, note (to avoid repeating package directory) that they were located in: "src/main/java/org/jabref/logic/", as this was the package I chose to scan for code smells.

1. Duplicate Code

   Found in "bst/BstPreviewLayout.java"

```
24      private final String name;
79      @Override
80      public String getDisplayName() { return name; }
83
84      @Override
85      public String getName() {
86          return name;
87      }
```

   Suggested Fix: Here we have two methods that do the exact same thing, and are therefore redundant. My suggestion is to simply remove one of them, more specifically getDisplayName(), which was added after getName() already existed and exists for the same purpose, to return the private global variable name.

2. Inappropriate Intimacy & Duplicate Code

Found in "importer/fetcher/ComplexSearchQuery.java"

```
40  @   public static ComplexSearchQuery fromTerms(List<Term> terms) {
41          ComplexSearchQueryBuilder builder = ComplexSearchQuery.builder();
42          terms.forEach(term -> {
43              String termText = term.text();
44              switch (term.field().toLowerCase()) {
45                  case "author" -> builder.author(termText);
46                  case "title" -> builder.titlePhrase(termText);
47                  case "abstract" -> builder.abstractPhrase(termText);
48                  case "journal" -> builder.journal(termText);
49                  case "year" -> builder.singleYear(Integer.valueOf(termText));
50                  case "year-range" -> builder.parseYearRange(termText);
51                  case "doi" -> builder.DOI(termText);
52                  case "default" -> builder.defaultFieldPhrase(termText);
53                  // add unknown field as default field
54                  default -> builder.defaultFieldPhrase(termText);
55              }
56          });
57          return builder.build();
58      }
253 @       public ComplexSearchQueryBuilder terms(Collection<Term> terms) {
254             terms.forEach(term -> {
255                 String termText = term.text();
256                 switch (term.field().toLowerCase()) {
257                     case "author" -> this.author(termText);
258                     case "title" -> this.titlePhrase(termText);
259                     case "abstract" -> this.abstractPhrase(termText);
260                     case "journal" -> this.journal(termText);
261                     case "doi" -> this.DOI(termText);
262                     case "year" -> this.singleYear(Integer.valueOf(termText));
263                     case "year-range" -> this.parseYearRange(termText);
264                     case "default" -> this.defaultFieldPhrase(termText);
265                 }
266             });
267             return this;
268         }
```

Suggested Fix: The ComplexSearchQuery class has a fromTerms() method which simply creates a builder class and proceeds to utilize its methods exclusively to re-do the work that the terms() method (from ComplexSearchQueryBuilder, the aforementioned builder class) already does.

This displays not only Duplicate Code but also Inappropriate Intimacy, since the fromTerms() method of the ComplexSearchQuery class uses methods from the ComplexSearchQueryBuilder class almost exclusively. Our suggested fix involves deleting lines 42-56 of the method fromTerms(), and replacing its return statement with builder.terms(terms).build().

In this way we avoid the repeated code in fromTerms() and terms() and we solve the inappropriate intimacy displayed by fromTerms() regarding ComplexSearchQueryBuilder.

3. Dead Code

Found in "formatter/bibtexfields/CleanupUrlFormatter.java"

```
29      @Override
30      public String format(String value) {
31          String decodedLink = value;
32          String toDecode = value;
33
34          Matcher matcher = PATTERN_URL.matcher(value);
35          if (matcher.find()) {
36              toDecode = matcher.group(1);
37          }
38          return URLDecoder.decode(toDecode, StandardCharsets.UTF_8);
39      }
```

Found in "util/CoarseChangeFilter.java"

```
17  public class CoarseChangeFilter {
18
19      private final BibDatabaseContext context;
20      private final EventBus eventBus = new EventBus();
21
22      private Optional<Field> lastFieldChanged;
23      private Optional<BibEntry> lastEntryChanged;
24      private int totalDelta;
```

Found in "importer/fileformat/EndnoteImporter.java"

```
32  public class EndnoteImporter extends Importer {
33
28      private final Collection<String> searchKeys = Arrays.asList
29              ("all", "tit", "per", "thm", "slw", "txt", "num", "kon", "ppn", "bkl", "erj");
36      private static final Pattern A_PATTERN = Pattern.compile("%A .*");
37      private static final Pattern E_PATTERN = Pattern.compile("%E .*");
38
39      private final ImportFormatPreferences preferences;
```

Found in "importer/fetcher/GvkFetcher.java"

Suggested Fix: The first class has a method with an unused local variable decodedLink. The other classes have unused private global variables totalDelta, preferences and searchKeys. This may have been due to them being planned to have a use during development but the features were completed without needing them and they were either forgotten or left in the code in anticipation of future use. My fix is obvious and simple, these variables should simply be deleted, as they are unused and do not affect their classes or the program/application as a whole.

Gonçalo Virgínia (56773)

1. Long Parameter List

The following 3 examples of this code smell are all found in the preferences package located in "src/main/java/org/jabref/preferences".

GuiPreferences.java

```
28      public GuiPreferences(double positionX,
29                            double positionY,
30                            double sizeX,
31                            double sizeY,
32                            boolean windowMaximised,
33                            boolean shouldOpenLastEdited,
34                            List<String> lastFilesOpened,
35                            Path lastFocusedFile, double sidePaneWidth) {
36          this.positionX = new SimpleDoubleProperty(positionX);
37          this.positionY = new SimpleDoubleProperty(positionY);
38          this.sizeX = new SimpleDoubleProperty(sizeX);
39          this.sizeY = new SimpleDoubleProperty(sizeY);
40          this.windowMaximised = new SimpleBooleanProperty(windowMaximised);
41          this.shouldOpenLastEdited = new SimpleBooleanProperty(shouldOpenLastEdited);
42          this.lastFilesOpened = FXCollections.observableArrayList(lastFilesOpened);
43          this.lastFocusedFile = new SimpleObjectProperty<>(lastFocusedFile);
44          this.sidePaneWidth = new SimpleDoubleProperty(sidePaneWidth);
45      }
```

ImportExportPreferences.java

```
22      public ImportExportPreferences(String nonWrappableFields,
23                                     boolean resolveStringsForStandardBibtexFields,
24                                     boolean resolveStringsForAllStrings,
25                                     String nonResolvableFields,
26                                     boolean alwaysReformatOnSave,
27                                     Path importWorkingDirectory,
28                                     String lastExportExtension,
29                                     Path exportWorkingDirectory) {
30          this.nonWrappableFields = new SimpleStringProperty(nonWrappableFields);
31          this.resolveStringsForStandardBibtexFields = new SimpleBooleanProperty(resolveStringsForStandardBibtexFields);
32          this.resolveStringsForAllStrings = new SimpleBooleanProperty(resolveStringsForAllStrings);
33          this.nonResolvableFields = new SimpleStringProperty(nonResolvableFields);
34          this.alwaysReformatOnSave = new SimpleBooleanProperty(alwaysReformatOnSave);
35          this.importWorkingDirectory = new SimpleObjectProperty<>(importWorkingDirectory);
36          this.lastExportExtension = new SimpleStringProperty(lastExportExtension);
37          this.exportWorkingDirectory = new SimpleObjectProperty<>(exportWorkingDirectory);
38      }
```

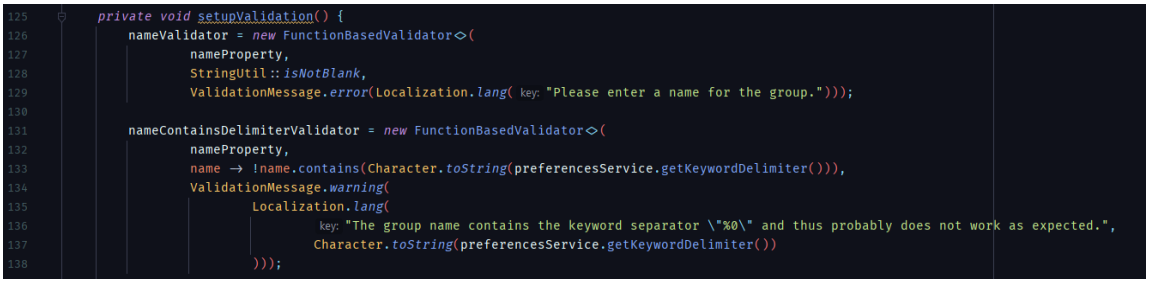
ExternalApplicationsPreferences.java

```
14      public ExternalApplicationsPreferences(String eMailSubject,
15                                             boolean shouldAutoOpenEmailAttachmentsFolder,
16                                             String pushToApplicationName,
17                                             String citeCommand,
18                                             boolean useCustomTerminal,
19                                             String customTerminalCommand,
20                                             boolean useCustomFileBrowser,
21                                             String customFileBrowserCommand) {
22
23          this.eMailSubject = eMailSubject;
24          this.shouldAutoOpenEmailAttachmentsFolder = shouldAutoOpenEmailAttachmentsFolder;
25          this.pushToApplicationName = pushToApplicationName;
26          this.citeCommand = citeCommand;
27          this.useCustomTerminal = useCustomTerminal;
28          this.customTerminalCommand = customTerminalCommand;
29          this.useCustomFileBrowser = useCustomFileBrowser;
30          this.customFileBrowserCommand = customFileBrowserCommand;
31      }
```

Fix: Either bundling some arguments together inside an immutable class (for example the position and size parameters in the GuiPreferences constructor), passing the caller class as a parameter, or, passing as parameters the instances that contain the variables listed as parameters in these constructors.

2. Long Method

src/main/java/org/jabref/gui/groups/GroupDialogViewModel.java



```
125     private void setupValidation() {
126         nameValidator = new FunctionBasedValidator<>(
127                 nameProperty,
128                 StringUtil::isNotBlank,
129                 ValidationMessage.error(Localization.lang( key: "Please enter a name for the group.")));
130
131         nameContainsDelimiterValidator = new FunctionBasedValidator<>(
132                 nameProperty,
133                 name -> !name.contains(Character.toString(preferencesService.getKeywordDelimiter())),
134                 ValidationMessage.warning(
135                         Localization.lang(
136                                 key: "The group name contains the keyword separator \"%0\" and thus probably does not work as expected.",
137                                 Character.toString(preferencesService.getKeywordDelimiter())
138                         )));
```

...

```
252
253         typeKeywordsProperty.addListener((obs, _oldValue, isSelected) -> {
254             if (isSelected) {
255                 validator.addValidators(keywordFieldEmptyValidator, keywordRegexValidator, keywordSearchTermEmptyValidator);
256             } else {
257                 validator.removeValidators(keywordFieldEmptyValidator, keywordRegexValidator, keywordSearchTermEmptyValidator);
258             }
259         });
260
261         typeTexProperty.addListener((obs, oldValue, isSelected) -> {
262             if (isSelected) {
263                 validator.addValidators(texGroupFilePathValidator);
264             } else {
265                 validator.removeValidators(texGroupFilePathValidator);
266             }
267         });
268     }
```

Fix: With a total of 143 lines, this method could be greatly improved by, for example, separating the logic for the assignment of each Validator instance variable into different methods, which would then be called by the main setupValidation() method.

3. Comments/Dead Code

src/main/java/org/jabref/gui/groups/GroupTreeViewModel.java

```
                        group,
                        keepPreviousAssignments,
                        removePreviousAssignments,
                        database.getEntries());
                // stateManager.getEntriesInCurrentDatabase());

                // TODO: Add undo
                // Store undo information.
                // AbstractUndoableEdit undoAddPreviousEntries = null;
                // UndoableModifyGroup undo = new UndoableModifyGroup(GroupSelector.this, groupsRoot, node, newGroup);
                // if (undoAddPreviousEntries == null) {
                //    panel.getUndoManager().addEdit(undo);
                // } else {
                //    NamedCompound nc = new NamedCompound("Modify Group");
                //    nc.addEdit(undo);
                //    nc.addEdit(undoAddPreviousEntries);
                //    nc.end();/
                //      panel.getUndoManager().addEdit(nc);
                // }
                // if (!addChange.isEmpty()) {
                //    undoAddPreviousEntries = UndoableChangeEntriesOfGroup.getUndoableEdit(null, addChange);
                // }

                dialogService.notify(Localization.lang( key: "Modified group \"%0\".", group.getName()));
                writeGroupChangesToMetaData();

                // This is ugly but we have no proper update mechanism in place to propagate the changes, so redraw everything
                refresh();
            });
        });
    }
```

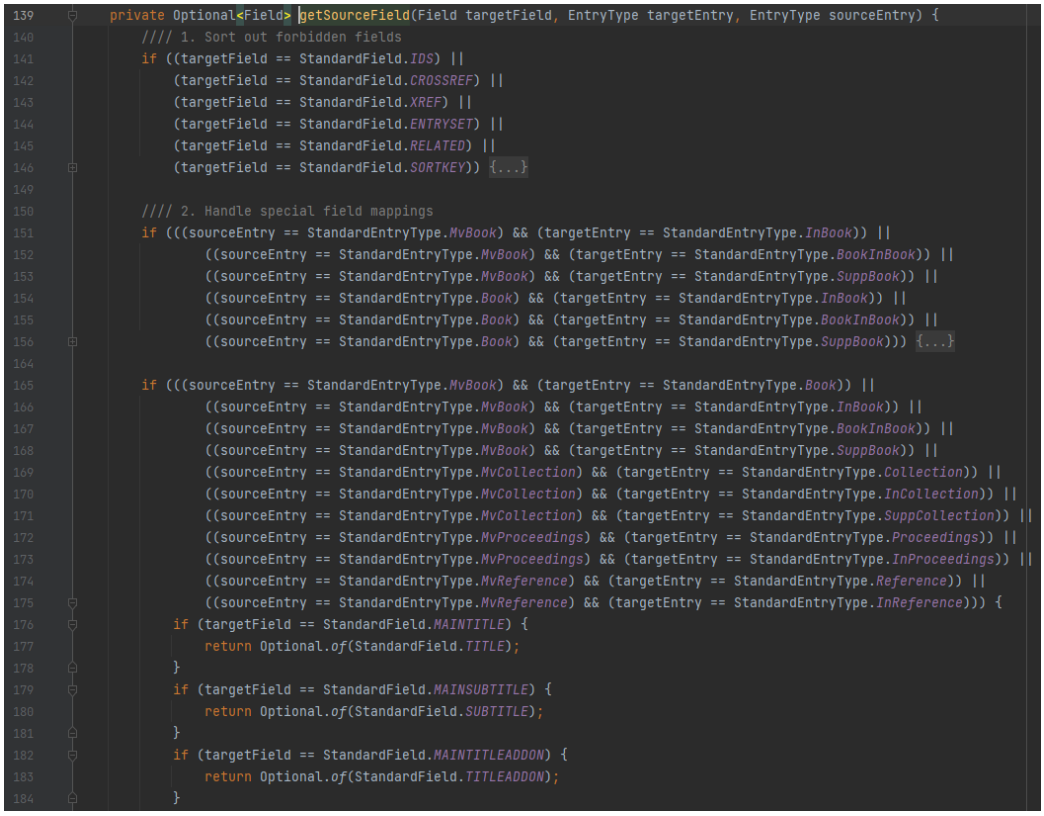
```
    public void addSelectedEntries(GroupNodeViewModel group) {
        // TODO: Warn
        // if (!WarnAssignmentSideEffects.warnAssignmentSideEffects(node.getNode().getGroup(), panel.frame())) {
        //    return; // user aborted operation

        group.getGroupNode().addEntriesToGroup(stateManager.getSelectedEntries());

        // TODO: Add undo
        // NamedCompound undoAll = new NamedCompound(Localization.lang("change assignment of entries"));
        // if (!undoAdd.isEmpty()) { undo.addEdit(UndoableChangeEntriesOfGroup.getUndoableEdit(node, undoAdd)); }
        // panel.getUndoManager().addEdit(undoAll);

        // TODO Display massages
        // if (undo == null) {
        //    frame.output(Localization.lang("The group \"%0\" already contains the selection.",
        //            node.getGroup().getName()));
        //    return;
        // }
        // panel.getUndoManager().addEdit(undo);
        // final String groupName = node.getGroup().getName();
        // if (assignedEntries == 1) {
        //    frame.output(Localization.lang("Assigned 1 entry to group \"%0\".", groupName));
        // } else {
        //    frame.output(Localization.lang("Assigned %0 entries to group \"%1\".", String.valueOf(assignedEntries),
        //            groupName));
        // }
    }
```

Fix: Deletion. If the code had/has to be altered, the previous/current implementation should be discarded and reimplemented in a new commit, there is no use for commenting out previous/temporary code that doesn't work, as well as comments that indicate problems with the current implementation in the code itself, as seen in line 238.

João Vieira (56971)

Code Smell #10- Switch Statements

Found: @SE2122_55677_56773__56971_57066_58655/src/main/java/org/jabref/model/entry/BibEntry.java



```
139    private Optional<Field> getSourceField(Field targetField, EntryType targetEntry, EntryType sourceEntry) {
140        //// 1. Sort out forbidden fields
141        if ((targetField == StandardField.IDS) ||
142            (targetField == StandardField.CROSSREF) ||
143            (targetField == StandardField.XREF) ||
144            (targetField == StandardField.ENTRYSET) ||
145            (targetField == StandardField.RELATED) ||
146            (targetField == StandardField.SORTKEY)) {...}
149
150        //// 2. Handle special field mappings
151        if (((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.InBook)) ||
152                ((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.BookInBook)) ||
153                ((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.SuppBook)) ||
154                ((sourceEntry == StandardEntryType.Book) && (targetEntry == StandardEntryType.InBook)) ||
155                ((sourceEntry == StandardEntryType.Book) && (targetEntry == StandardEntryType.BookInBook)) ||
156                ((sourceEntry == StandardEntryType.Book) && (targetEntry == StandardEntryType.SuppBook))) {...}
164
165        if (((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.Book)) ||
166                ((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.InBook)) ||
167                ((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.BookInBook)) ||
168                ((sourceEntry == StandardEntryType.MvBook) && (targetEntry == StandardEntryType.SuppBook)) ||
169                ((sourceEntry == StandardEntryType.MvCollection) && (targetEntry == StandardEntryType.Collection)) ||
170                ((sourceEntry == StandardEntryType.MvCollection) && (targetEntry == StandardEntryType.InCollection)) ||
171                ((sourceEntry == StandardEntryType.MvCollection) && (targetEntry == StandardEntryType.SuppCollection)) ||
172                ((sourceEntry == StandardEntryType.MvProceedings) && (targetEntry == StandardEntryType.Proceedings)) ||
173                ((sourceEntry == StandardEntryType.MvProceedings) && (targetEntry == StandardEntryType.InProceedings)) ||
174                ((sourceEntry == StandardEntryType.MvReference) && (targetEntry == StandardEntryType.Reference)) ||
175                ((sourceEntry == StandardEntryType.MvReference) && (targetEntry == StandardEntryType.InReference))) {
176            if (targetField == StandardField.MAINTITLE) {
177                return Optional.of(StandardField.TITLE);
178            }
179            if (targetField == StandardField.MAINSUBTITLE) {
180                return Optional.of(StandardField.SUBTITLE);
181            }
182            if (targetField == StandardField.MAINTITLEADDON) {
183                return Optional.of(StandardField.TITLEADDON);
184            }
```

```
            // those fields are no more available for the same-name inheritance strategy
            if ((targetField == StandardField.TITLE) ||
                (targetField == StandardField.SUBTITLE) ||
                (targetField == StandardField.TITLEADDON)) {
                return Optional.empty();
            }

            // for these fields, inheritance is not allowed for the specified entry types
            if (targetField == StandardField.SHORTTITLE) {
                return Optional.empty();
            }
        }

        if (((sourceEntry == StandardEntryType.Book) && (targetEntry == StandardEntryType.InBook)) ||
            ((sourceEntry == StandardEntryType.Book) && (targetEntry == StandardEntryType.BookInBook)) ||
            ((sourceEntry == StandardEntryType.Book) && (targetEntry == StandardEntryType.SuppBook)) ||
            ((sourceEntry == StandardEntryType.Collection) && (targetEntry == StandardEntryType.InCollection)) ||
            ((sourceEntry == StandardEntryType.Collection) && (targetEntry == StandardEntryType.SuppCollection)) ||
            ((sourceEntry == StandardEntryType.Reference) && (targetEntry == StandardEntryType.InReference)) ||
            ((sourceEntry == StandardEntryType.Proceedings) && (targetEntry == StandardEntryType.InProceedings))) {
            if (targetField == StandardField.BOOKTITLE) {
                return Optional.of(StandardField.TITLE);
            }
            if (targetField == StandardField.BOOKSUBTITLE) {
                return Optional.of(StandardField.SUBTITLE);
            }
            if (targetField == StandardField.BOOKTITLEADDON) {
                return Optional.of(StandardField.TITLEADDON);
            }

            // those fields are no more available for the same-name inheritance strategy
            if ((targetField == StandardField.TITLE) ||
                (targetField == StandardField.SUBTITLE) ||
                (targetField == StandardField.TITLEADDON)) {
                return Optional.empty();
            }
```

```

            // for these fields, inheritance is not allowed for the specified entry types
            if ((targetField == StandardField.SHORTTITLE)) {
                return Optional.empty();
            }
        }

        if (((sourceEntry == IEEETranEntryType.Periodical) && (targetEntry == StandardEntryType.Article)) ||
            ((sourceEntry == IEEETranEntryType.Periodical) && (targetEntry == StandardEntryType.SuppPeriodical))) {
            if (targetField == StandardField.JOURNALTITLE) {
                return Optional.of(StandardField.TITLE);
            }
            if (targetField == StandardField.JOURNALSUBTITLE) {
                return Optional.of(StandardField.SUBTITLE);
            }

            // those fields are no more available for the same-name inheritance strategy
            if ((targetField == StandardField.TITLE) ||
                (targetField == StandardField.SUBTITLE)) {
                return Optional.empty();
            }

            // for these fields, inheritance is not allowed for the specified entry types
            if ((targetField == StandardField.SHORTTITLE)) {
                return Optional.empty();
            }
        }

        //// 3. Fallback to inherit the field with the same name.
        return Optional.ofNullable(targetField);
    }

```

This method has too many ifs, not only some of this ifs also have a very complex set of conditions that must be verified, there is also a lot of duplicated ifs and some these are all the possible combinations between the involving enumerator types. This makes code feel and look very bloated, hard to read and follow and it and adds a lot of complexity to the operation.

Proposed Solution:

To fix this first I would evaluate if every single if is necessary, because some of this ifs return the exact same value. There is also a lot of duplicated ifs, so those should probably be moved to its own method, same could probably be applied to the ifs that have all the possible combinations between enumerator types. A case could also be made to replace these conditions with polymorphism, but I'm not sure on how viable this solution would be.

Code Smell #11- Duplicated Code

Found:

@SE2122_55677_56773__56971_57066_58655/src/main/java/org/jabref/model/entry/identified/DOI.java

```
public String getDOI() { return doi; }
```

```
@Override
public String getNormalized() {
    return doi;
}
```

These two methods are practically identical and return the same information

Proposed Solution:

To fix this I believe the simplest solution is to remove one of these methods.

Code Smell #12- String literals should not be duplicated

Found:
@SE2122_55677_56773__56971_57066_58655/src/main/java/org/jabref/model/openoffice/ootext/OOTextIntoOO.java

...

```
/**
 *  "ParaStyleName" is an OpenOffice Property name.
 */
private static final String PARA_STYLE_NAME = "ParaStyleName";

/*
 * Character property names used in multiple locations below.
 */
private static final String CHAR_ESCAPEMENT_HEIGHT = "CharEscapementHeight";
private static final String CHAR_ESCAPEMENT = "CharEscapement";
private static final String CHAR_STYLE_NAME = "CharStyleName";
private static final String CHAR_UNDERLINE = "CharUnderline";
private static final String CHAR_STRIKEOUT = "CharStrikeout";
```

...

```
final Set<String> knownToFail = Set.of("ListAutoFormat",
                                       "ListId",
                                       "NumberingIsNumber",
                                       "NumberingLevel",
                                       "NumberingRules",
                                       "NumberingStartValue",
                                       "ParaChapterNumberingLevel",
                                       "ParaIsNumberingRestart",
                                       "ParaStyleName");
```

...

```
...
static final Set<String> CONTROLLED_PROPERTIES = Set.of(

    /* Used for SuperScript, SubScript.
     *
     * These three are interdependent: changing one may change others.
     */
    "CharEscapement", "CharEscapementHeight", "CharAutoEscapement",

    /* used for Bold */
    "CharWeight",

    /* Used for Italic */
    "CharPosture",

    /* Used for strikeout. These two are interdependent. */
    "CharStrikeout", "CharCrossedOut",

    /* Used for underline. These three are interdependent, but apparently
     * we can leave out the last two.
     */
    "CharUnderline", // "CharUnderlineColor", "CharUnderlineHasColor",

    /* Used for lang="zxx", to silence spellchecker. */
    "CharLocale",

    /* Used for CitationCharacterFormat.  */
    "CharStyleName",

    /* Used for <smallcaps> and <span style="font-variant: small-caps"> */
    "CharCaseMap");
...
```

As we can see, in the areas highlighted in yellow, we have a lot of occurrences where we use string literals instead of the already appropriated defined constants, duplicating this string literals.

This makes the process of refactoring error-prone, since we must be sure to update all occurrences. Making constants a much more appealing solution since they can be referenced in many places and only need to be updated in a single place, when we refactor something.

Proposed Solution:

I recommend we change all the occurrences highlighted in yellow to the appropriate constants, like so:

```
...

/**
 *  "ParaStyleName" is an OpenOffice Property name.
 */
private static final String PARA_STYLE_NAME = "ParaStyleName";

/*
 * Character property names used in multiple locations below.
 */
private static final String CHAR_ESCAPEMENT_HEIGHT = "CharEscapementHeight";
private static final String CHAR_ESCAPEMENT = "CharEscapement";
private static final String CHAR_STYLE_NAME = "CharStyleName";
private static final String CHAR_UNDERLINE = "CharUnderline";
private static final String CHAR_STRIKEOUT = "CharStrikeout";

...

final Set<String> knownToFail = Set.of("ListAutoFormat",
                                       "ListId",
                                       "NumberingIsNumber",
                                       "NumberingLevel",
                                       "NumberingRules",
                                       "NumberingStartValue",
                                       "ParaChapterNumberingLevel",
                                       "ParaIsNumberingRestart",
                                       PARA_STYLE_NAME);

...
```

```
...
static final Set<String> CONTROLLED_PROPERTIES = Set.of(

    /* Used for SuperScript, SubScript.
     *
     * These three are interdependent: changing one may change others.
     */
    CHAR_ESCAPEMENT, CHAR_ESCAPEMENT_HEIGHT, "CharAutoEscapement",

    /* used for Bold */
    "CharWeight",

    /* Used for Italic */
    "CharPosture",

    /* Used for strikeout. These two are interdependent. */
    CHAR_STRIKEOUT, "CharCrossedOut",

    /* Used for underline. These three are interdependent, but apparently
     * we can leave out the last two.
     */
    CHAR_UNDERLINE, // "CharUnderlineColor", "CharUnderlineHasColor",

    /* Used for lang="zxx", to silence spellchecker. */
    "CharLocale",

    /* Used for CitationCharacterFormat.  */
    CHAR_STYLE_NAME,

    /* Used for <smallcaps> and <span style="font-variant: small-caps"> */
    "CharCaseMap");
...
```

Guilherme Pereira (57066)

1- Long method

found @ src/main/java/org/jabref/logic/importer/fileformat/IsiImporter.java

```
142         @Override
143         public ParserResult importDatabase(BufferedReader reader) throws IOException {
```

```
327             return new ParserResult(bibEntries);
328         }
```

I identified this as a code smell because the method itself is 185 lines long and has several comments explaining what is happening at some parts of the method, but none at others making it difficult and taxing to understand.

A possible fix could be passing some of the logic into different methods with names that make it easy to infer what they are doing without the need to go look at them, also acting as a quick way to explain what is happening without comments (which can turn into a whole different type of code smell).

2- Dead code/speculative generality

found @ src/main/java/org/jabref/model/entry/field/InternalField.java

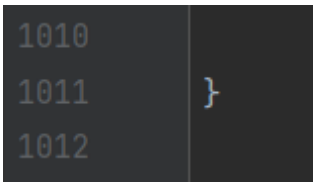
```
34      InternalField(String name, FieldProperty first, FieldProperty... rest) {
35          this.name = name;
36          this.properties = EnumSet.of(first, rest);
37      }
```

I identified this as a code smell because this constructor is never used so it is dead code, but it could also be a code smell of type speculative generality since it could be code prepared to be used in the future.

A possible fix could be the deletion of that snippet of code.

3- Large class

found @ src/main/java/org/jabref/model/entry/BibEntry.java



(Chauhan, 2019) uses the threshold 750 lines to determine when a class is considered a large class. Using that knowledge, I identified this as a code smell since the class has 1012 lines.

A possible fix could be the division of the class into two minor classes to divide their work better or create a subclass that handles behavior that can be implemented in a different way or that is used in rare cases, for example, by creating two other classes: one that handles all Keyword related behaviors and the other all Field related behaviors.

Bibliography

Chauhan, S. S. (2019). *Code Smells Quantification: A Case Study On Large Open Source Research Codebase.*

Gabriela Costa (58625)

1- Duplicate Code

Found at src/main/java/org/jabref/model/entry/BibEntry.java

```
82      private ObservableMap<Field, String> fields = FXCollections.observableMap(new ConcurrentHashMap<>());
792     public Map<Field, String> getFieldMap() { return fields; }
981     public ObservableMap<Field, String> getFieldsObservable() {
982         return fields;
983     }
```

I consider this a "duplicate code" code smell since both methods, getFieldMap() and getFieldsObservable() end up returning exactly the same thing. To fix this, we could simply eliminate either method.

2- Long Method

Found at src/main/java/org/jabref/model/openoffice/ootext/OOTextIntoOO.java

```
148     public static void write(XTextDocument doc, XTextCursor position, OOText ootext)
149         throws
150         WrappedTargetException,
151         CreationException {...}
342
```

This seems to be a "long method" code smell for several reasons. Firstly, the method is almost 200 lines long. Secondly, the method has several moving parts inside, which can easily be compartimentalized into separate methods. And finally, one can find several comments within the method attempting to explain what each part of it does, suggesting that the method is too complicated.
My proposal to mitigate this consists in taking the following switch statement, and placing it into a separate method:

```
190             // Handle tags:
191             switch (tagName) {...}
322
```

However, the switch statement itself presents a problem, since it is 130 lines long. This leads us to our next code smell.

3- Switch Statements

The switch statement presented above has over 20 cases. While some cases are simple and have clear similarities between them, others are very long.

To fix this, I would first suggest creating a method to handle the recurring portion of code shown below:

```
193                formatStack.pushLayer(setCharWeight(FontWeight.BOLD));
194                expectEnd.push( item: "/" + tagName);
```

This method would receive the formatStack, the expectEnd, the argument for the pushLayer() method, as well as the tagName. It would solve the code repetition seen throughout most of the switch cases. Next, the cases “p”, “oo:referenceToPageNumberOfReferenceMark” and “span” should be handled in separate methods. And finally, the last cases corresponding to closing tags can be merged into one case only, separated by commas, and the code inside the case should be moved into a separate method.