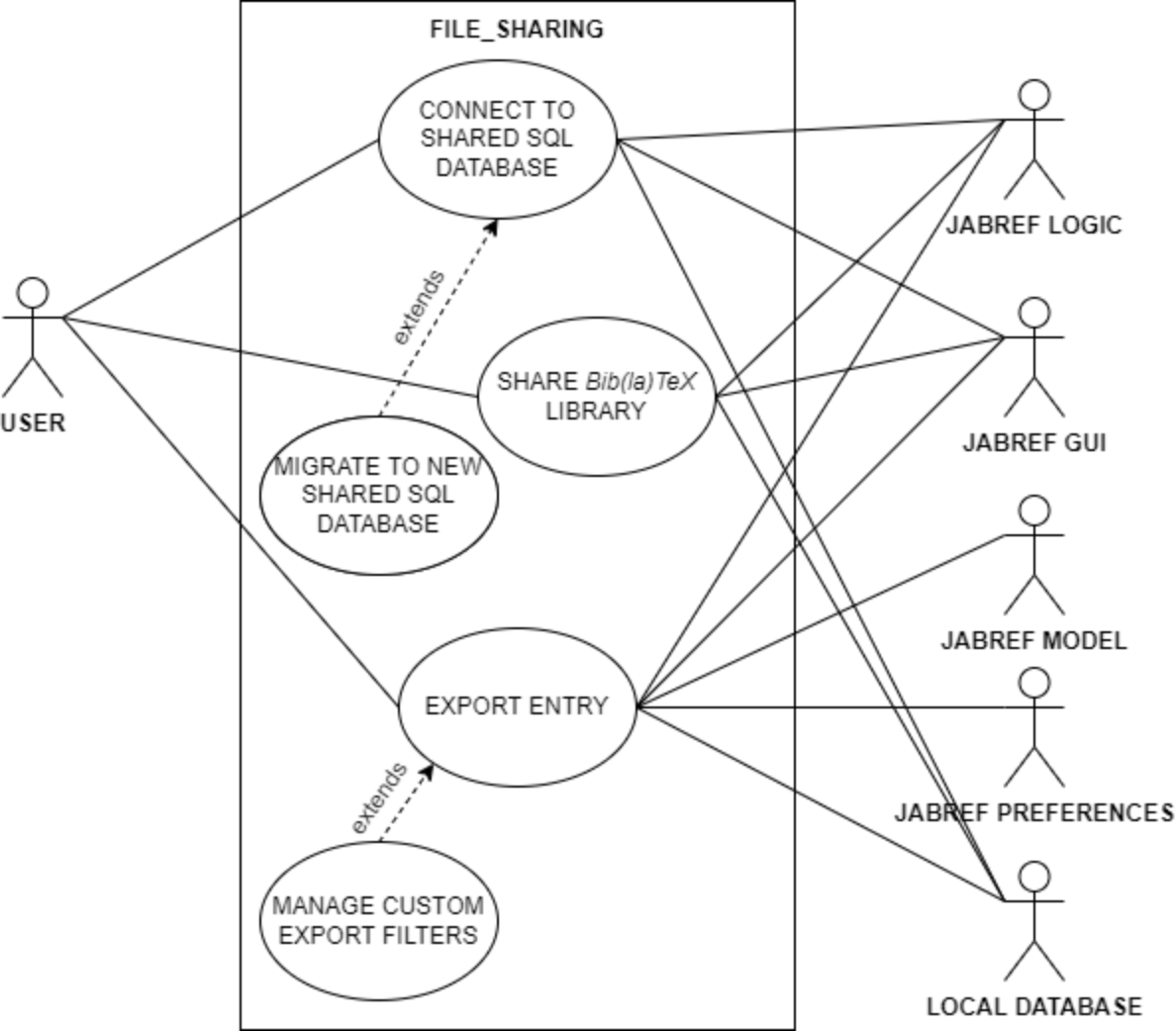
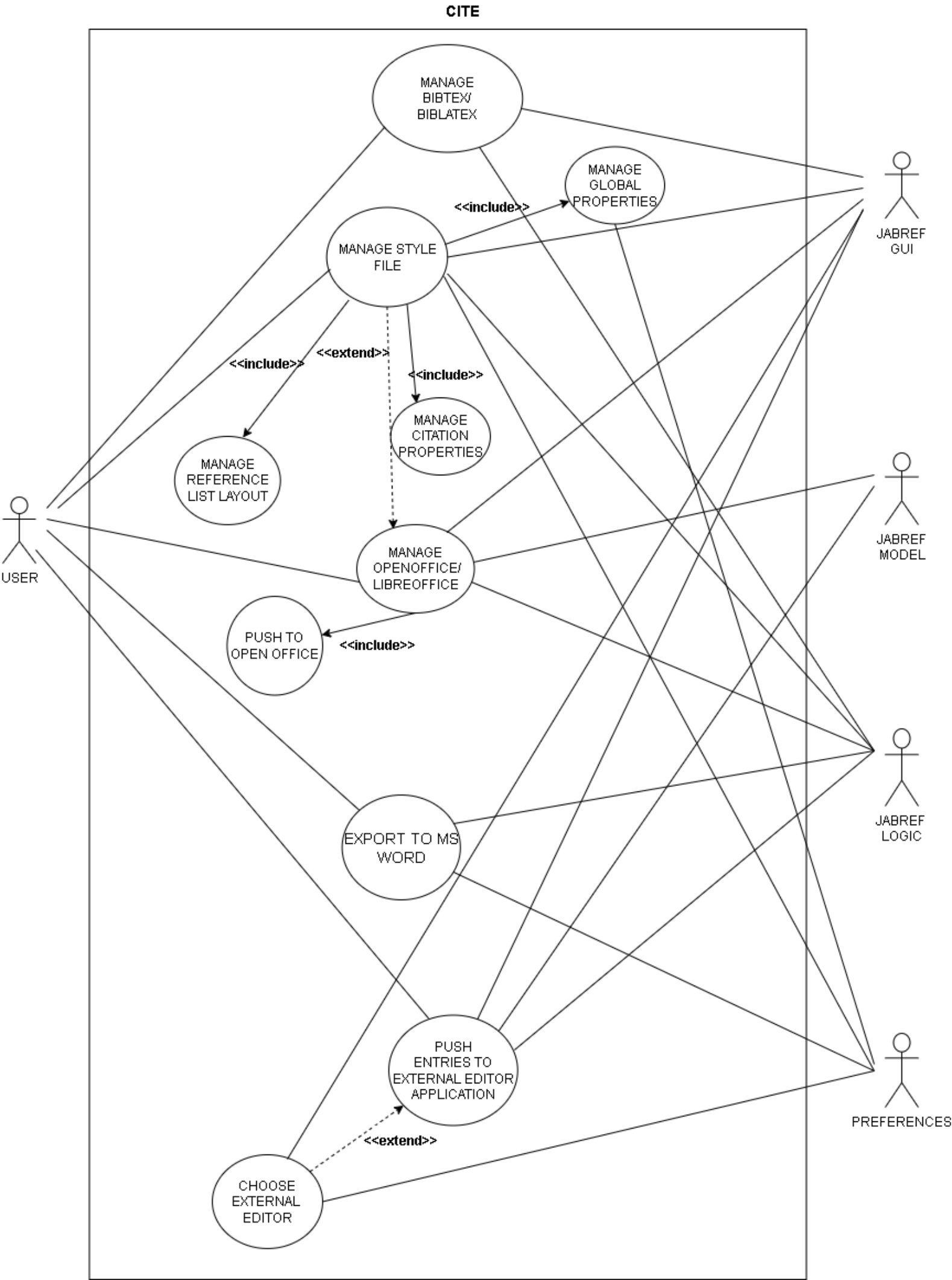
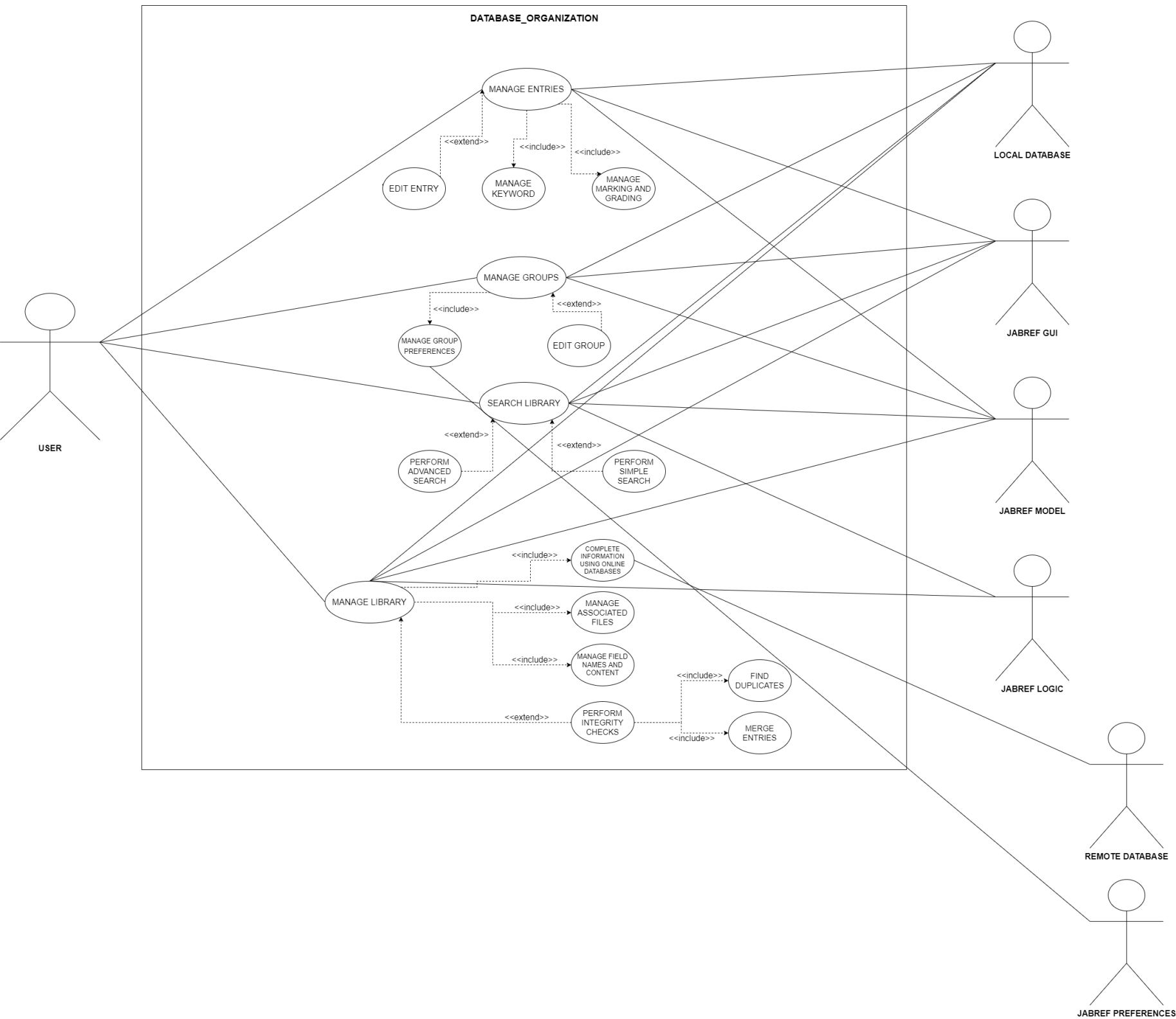

FILE_SHARING
CONNECT TO SHARED SQL DATABASE
SHARE Bib(la)TeX LIBRARY
MIGRATE TO NEW SHARED SQL DATABASE
EXPORT ENTRY
MANAGE CUSTOM EXPORT FILTERS
extends
extends
USER
JABREF LOGIC
JABREF GUI
JABREF MODEL
JABREF PREFERENCES
LOCAL DATABASE


ENTRY COLLECTION
ADD ENTRY MANUALLY
<<extend>>
ADD ENTRY USING AN IDENTIFIER
ADD ENTRY USING REFERENCE TEXT
SEARCHING ONLINE USING REMOTE DATABASES
ADD ENTRY USING PDF'S
ADD REFERENCES USING A BROWSER EXTENSION
IMPORT
<<extend>>
MANAGE CUSTOM IMPORT FILTERS
<<include>>
IMPORT INSPECTION WINDOW
<<include>>
<<include>>
DELETE THE ENTRY
ADD THE ENTRY TO A GROUP
USER
JABREF MODEL
JABREF LOGIC
REMOTE DATABASE
JABREF UI (CLI + GUI)


CITE
MANAGE BIBTEX/ BIBLATEX
MANAGE GLOBAL PROPERTIES
<<include>>
MANAGE STYLE FILE
<<include>>
<<extend>>
<<include>>
MANAGE CITATION PROPERTIES
MANAGE REFERENCE LIST LAYOUT
MANAGE OPENOFFICE/ LIBREOFFICE
PUSH TO OPEN OFFICE
<<include>>
EXPORT TO MS WORD
PUSH ENTRIES TO EXTERNAL EDITOR APPLICATION
<<extend>>
CHOOSE EXTERNAL EDITOR
USER
JABREF GUI
JABREF MODEL
JABREF LOGIC
PREFERENCES


DATABASE_ORGANIZATION
MANAGE ENTRIES
EDIT ENTRY
MANAGE KEYWORD
MANAGE MARKING AND GRADING
<<extend>>
<<include>>
<<include>>
MANAGE GROUPS
MANAGE GROUP PREFERENCES
EDIT GROUP
<<include>>
<<extend>>
SEARCH LIBRARY
PERFORM ADVANCED SEARCH
PERFORM SIMPLE SEARCH
<<extend>>
<<extend>>
MANAGE LIBRARY
COMPLETE INFORMATION USING ONLINE DATABASES
MANAGE ASSOCIATED FILES
MANAGE FIELD NAMES AND CONTENT
PERFORM INTEGRITY CHECKS
FIND DUPLICATES
MERGE ENTRIES
<<include>>
<<include>>
<<include>>
<<extend>>
<<include>>
<<include>>
USER
LOCAL DATABASE
JABREF GUI
JABREF MODEL
JABREF LOGIC
REMOTE DATABASE
JABREF PREFERENCES


Configuration
Customize the citation key generator
Edit special field markers
Change default key pattern
Change citation key pattern for one library
Remove unwanted characters
Replace regular expressions
Customize entry types
Edit entry type
Add entry type
Remove entry type
Modify library properties
Library Mode
Save actions
Library Encoding
Save sort order
Library protection
Override default file directories
Customize general fields
Customize key bindings
Setup Entry preview
Display customized layout
Display customized style
Modify customized preview style
Manage external file types
Manage protected terms
Edit string constants
«includes»
«extends»
User
Model
GUI
Logic
Remote Database
Local Database

Code Smells

Below I describe the three “code smells” I found, as proposed.

The first “code smell”: Shotgun surgery

```
public ComplexSearchQueryBuilder defaultFieldPhrase(String
defaultFieldPhrase) {
    if (Objects.requireNonNull(defaultFieldPhrase).isBlank()) {
        throw new IllegalArgumentException("Parameter must not be
blank");
    }
    // Strip all quotes before wrapping
    this.defaultFieldPhrases.add(String.format("\"%s\"",
defaultFieldPhrase.replace("\"", "")));
    return this;
}

/**
 * Adds author and wraps it in quotes
 */
public ComplexSearchQueryBuilder author(String author) {
    if (Objects.requireNonNull(author).isBlank()) {
        throw new IllegalArgumentException("Parameter must not be
blank");
    }
    // Strip all quotes before wrapping
    this.authors.add(String.format("\"%s\"", author.replace("\"",
"")));
    return this;
}

/**
 * Adds title phrase and wraps it in quotes
 */
public ComplexSearchQueryBuilder titlePhrase(String titlePhrase) {
    if (Objects.requireNonNull(titlePhrase).isBlank()) {
        throw new IllegalArgumentException("Parameter must not be
blank");
    }
    // Strip all quotes before wrapping
    this.titlePhrases.add(String.format("\"%s\"",
titlePhrase.replace("\"", "")));
    return this;
}

/**
 * Adds abstract phrase and wraps it in quotes
 */
public ComplexSearchQueryBuilder abstractPhrase(String abstractPhrase)
{
    if (Objects.requireNonNull(abstractPhrase).isBlank()) {
        throw new IllegalArgumentException("Parameter must not be
```

```
blank");
    }
    // Strip all quotes before wrapping
    this.titlePhrases.add(String.format("\"%s\"",
abstractPhrase.replace("\"", "")));
    return this;
}
```

This long parameter list can be found in **jabref > logic >importer > fetcher > transformers > ComplexSearchQuery.**

In this case, there are blocks of code very similar, present in many places of the code. A new method, containing the recurrent code that deals with the small changes between blocks to prevent such repetition, is recommended. Another observation would be creating a new constant containing the phrase "Parameter must not be blank".

```
if (Objects.requireNonNull(abstractPhrase).isBlank()) {
    throw new IllegalArgumentException("Parameter must not be blank");
}
```

The second “code smell”: long parameter list

```
private ComplexSearchQuery(List<String> defaultField, List<String>
authors, List<String> titlePhrases, List<String> abstractPhrases,
Integer fromYear, Integer toYear, Integer singleYear, String journal,
String doi) {
    this.defaultField = defaultField;
    this.authors = authors;
    this.titlePhrases = titlePhrases;
    this.abstractPhrases = abstractPhrases;
    this.fromYear = fromYear;
    // Some APIs do not support, or not fully support, year based
search. In these cases, the non applicable parameters are ignored.
    this.toYear = toYear;
    this.journal = journal;
    this.singleYear = singleYear;
    this.doi = doi;
}
```

This long parameter list can be found in **jabref > logic >importer > fetcher > transformers > ComplexSearchQuery.**

Here, the method has nine parameters which I believe is too much. The following parameters seem to be related and could be stored in a specific object - Integer fromYear, Integer toYear, and Integer singleYear.

The third “code smell”: long method

```
@Override
public Optional<BibEntry> performSearchById(String identifier) throws
FetcherException {
    Optional<DOI> doi = DOI.parse(identifier);

    try {
        if (doi.isPresent()) {
            Optional<BibEntry> fetchedEntry;

            // mEDRA does not return a parsable bibtex string
            if (getAgency(doi.get()).isPresent() &&
"medra".equalsIgnoreCase(getAgency(doi.get()).get())) {
                return new Medra().performSearchById(identifier);
            }
            URL doiURL = new URL(doi.get().getURIAsASCIIString());

            // BibTeX data
            URLDownload download = getUrlDownload(doiURL);
            download.addHeader("Accept",
MediaTypes.APPLICATION_BIBTEX);
            String bibtexString;
            try {
                bibtexString = download.asString();
            } catch (IOException e) {
                // an IOException will be thrown if download is unable
to download from the doiURL
                throw new FetcherException(Localization.lang("No DOI
data exists"), e);
            }

            // BibTeX entry
            fetchedEntry = BibtexParser.singleFromString(bibtexString,
preferences, new DummyFileUpdateMonitor());
            fetchedEntry.ifPresent(this::doPostCleanup);

            // Check if the entry is an APS journal and add the
article id as the page count if page field is missing
            if (fetchedEntry.isPresent() &&
```

```
fetchedEntry.get().hasField(StandardField.DOI)) {
                BibEntry entry = fetchedEntry.get();
                if (isAPSJournal(entry,
entry.getField(StandardField.DOI).get()) &&
!entry.hasField(StandardField.PAGES)) {
                    setPageCountToArticleId(entry,
entry.getField(StandardField.DOI).get());
                }
            }

            return fetchedEntry;
        } else {
            throw new FetcherException(Localization.lang("Invalid DOI:
'%0'.", identifier));
        }
    } catch (IOException e) {
        throw new FetcherException(Localization.lang("Connection
error"), e);
    } catch (ParseException e) {
        throw new FetcherException("Could not parse BibTeX entry", e);
    } catch (JSONException e) {
        throw new FetcherException("Could not retrieve Registration
Agency", e);
    }
}
```

This long method can be found in **jabref > logic >importer > fetcher > transformers > DoiFetcher.**

The method above is hard to read. It could be divided into more methods to reduce its complexity and allow a clearer way to analyze it. One suggestion would be extracting the following block and inserting it into a new method to improve the general method's readability:

```
try {
    bibtexString = download.asString();
} catch (IOException e) {
    // an IOException will be thrown if download is unable to download
from the doiURL
    throw new FetcherException(Localization.lang("No DOI data
exists"), e);
}
```

Francisco Abreu Freire, nº 58667

Code Smells

First *code smell*: Long Parameter List

```
private void writeMetadatatoPdf(List<ParserResult> loaded, String
filesAndCitekeys, Charset encoding, XmpPreferences xmpPreferences,
FilePreferences filePreferences, BibDatabaseMode databaseMode,
BibEntryTypesManager entryTypesManager, FieldWriterPreferences
fieldWriterPreferences, boolean writeXMP, boolean embeddBibfile) {
    if (loaded.isEmpty()) {
        LOGGER.error("The write xmp option depends on a valid import
option.");
        return;
    }
    ParserResult pr = loaded.get(loaded.size() - 1);
    BibDatabaseContext databaseContext = pr.getDatabaseContext();
    BibDatabase dataBase = pr.getDatabase();

    XmpPdfExporter xmpPdfExporter = new
XmpPdfExporter(xmpPreferences);
    EmbeddedBibFilePdfExporter embeddedBibFilePdfExporter = new
EmbeddedBibFilePdfExporter(databaseMode, entryTypesManager,
fieldWriterPreferences);

    if ("all".equals(filesAndCitekeys)) {
        for (BibEntry entry : dataBase.getEntries()) {
            writeMetadatatoPDFsOfEntry(databaseContext,
entry.getCitationKey().orElse("<no cite key defined>"), entry,
encoding, filePreferences, xmpPdfExporter, embeddedBibFilePdfExporter,
writeXMP, embeddBibfile);
        }
        return;
    }

    Vector<String> citeKeys = new Vector<>();
    Vector<String> pdfs = new Vector<>();
    for (String fileOrCiteKey : filesAndCitekeys.split(",")) {
        if (fileOrCiteKey.toLowerCase(Locale.ROOT).endsWith(".pdf")) {
            pdfs.add(fileOrCiteKey);
        } else {
            citeKeys.add(fileOrCiteKey);
        }
    }

    writeMetadatatoPdfByCitekey(databaseContext, dataBase, citeKeys,
encoding, filePreferences, xmpPdfExporter, embeddedBibFilePdfExporter,
writeXMP, embeddBibfile);
    writeMetadatatoPdfByFileNames(databaseContext, dataBase, pdfs,
encoding, filePreferences, xmpPdfExporter, embeddedBibFilePdfExporter,
writeXMP, embeddBibfile);

}
```

This long parameter list code smell can be found in **jabref > cli > ArgumentProcessor.**

This method has ten parameters, which is too much. Some of them could be transformed into parameter objects (for example, the various preferences parameters) in order to reduce the parameter count.

Second *code smell*: Reminder Comments

```
// TODO: Move to OS.java
public static NativeDesktop getNativeDesktop() {
    if (OS.WINDOWS) {
        return new Windows();
    } else if (OS.OS_X) {
        return new OSX();
    } else if (OS.LINUX) {
        return new Linux();
    }
    return new DefaultDesktop();
}
```

This reminder comment code smell can be found in **jabref > gui > desktop > JabRefDesktop.**

These comments take on a reminder nature, indicating something that needs to be done, which designates bad code.

Third *code smell*: Long Method

```
private Node createToolbar() {
    final ActionFactory factory = new
ActionFactory(Globals.getKeyPrefs());

    final Region leftSpacer = new Region();
    final Region rightSpacer = new Region();

    final PushToApplicationAction pushToApplicationAction =
getPushToApplicationsManager().getPushToApplicationAction();
    final Button pushToApplicationButton =
factory.createIconButton(pushToApplicationAction.getActionInformation(
), pushToApplicationAction);

pushToApplicationsManager.registerReconfigurable(pushToApplicationButt
on);

    // Setup Toolbar

    ToolBar toolBar = new ToolBar(
```

```
            new HBox(

factory.createIconButton(StandardActions.NEW_LIBRARY, new
NewDatabaseAction(this, prefs)),

factory.createIconButton(StandardActions.OPEN_LIBRARY, new
OpenDatabaseAction(this, prefs, dialogService, stateManager)),

factory.createIconButton(StandardActions.SAVE_LIBRARY, new
SaveAction(SaveAction.SaveMethod.SAVE, this, prefs, stateManager))),

            leftSpacer,

            globalSearchBar,

            rightSpacer,

            new HBox(

factory.createIconButton(StandardActions.NEW_ARTICLE, new
NewEntryAction(this, StandardEntryType.Article, dialogService, prefs,
stateManager)),

factory.createIconButton(StandardActions.NEW_ENTRY, new
NewEntryAction(this, dialogService, prefs, stateManager)),
                    createNewEntryFromIdButton(),

factory.createIconButton(StandardActions.NEW_ENTRY_FROM_PLAIN_TEXT,
new ExtractBibtexAction(dialogService, prefs, stateManager)),

factory.createIconButton(StandardActions.DELETE_ENTRY, new
EditAction(StandardActions.DELETE_ENTRY, this, stateManager))
            ),

            new Separator(Orientation.VERTICAL),

            new HBox(
                    factory.createIconButton(StandardActions.UNDO, new
UndoRedoAction(StandardActions.UNDO, this, dialogService,
stateManager)),
                    factory.createIconButton(StandardActions.REDO, new
UndoRedoAction(StandardActions.REDO, this, dialogService,
stateManager)),
                    factory.createIconButton(StandardActions.CUT, new
EditAction(StandardActions.CUT, this, stateManager)),
                    factory.createIconButton(StandardActions.COPY, new
EditAction(StandardActions.COPY, this, stateManager)),
                    factory.createIconButton(StandardActions.PASTE,
new EditAction(StandardActions.PASTE, this, stateManager))
            ),

            new Separator(Orientation.VERTICAL),

            new HBox(
                    pushToApplicationButton,

factory.createIconButton(StandardActions.GENERATE_CITE_KEYS, new
GenerateCitationKeyAction(this, dialogService, stateManager,
taskExecutor, prefs)),
```

```
factory.createIconButton(StandardActions.CLEANUP_ENTRIES, new
CleanupAction(this, prefs, dialogService, stateManager))
            ),

            new Separator(Orientation.VERTICAL),

            new HBox(

factory.createIconButton(StandardActions.OPEN_GITHUB, new
OpenBrowserAction("https://github.com/JabRef/jabref")),

factory.createIconButton(StandardActions.OPEN_FACEBOOK, new
OpenBrowserAction("https://www.facebook.com/JabRef/")),

factory.createIconButton(StandardActions.OPEN_TWITTER, new
OpenBrowserAction("https://twitter.com/jabref_org"))
            ),

            new Separator(Orientation.VERTICAL),

            new HBox(
                    createTaskIndicator()
            )
    );

    leftSpacer.setPrefWidth(50);
    leftSpacer.setMinWidth(Region.USE_PREF_SIZE);
    leftSpacer.setMaxWidth(Region.USE_PREF_SIZE);
    HBox.setHgrow(globalSearchBar, Priority.ALWAYS);
    HBox.setHgrow(rightSpacer, Priority.SOMETIMES);

    toolBar.getStyleClass().add("mainToolbar");

    return toolBar;
}
```

This long method code smell can be found in **jabref > gui > JabRefFrame.**

Even though this method is setting up a user interface, it is still hard to read. A solution to make the method more readable and easier to understand would be to divide it into smaller methods.

Francisco Pires, nº 58208

Code Smells

First code smell: long method, 'masked' by too many comments

```
private static List<BibEntryDiff> compareEntries(List<BibEntry>
originalEntries, List<BibEntry> newEntries) {
    List<BibEntryDiff> differences = new ArrayList<>();

    // Create pointers that are incremented as the entries of each base are
used in
    // successive order from the beginning. Entries "further down" in the
new database
    // can also be matched.
    int positionNew = 0;

    // Create a HashSet where we can put references to entries in the new
    // database that we have matched. This is to avoid matching them twice.
    Set<Integer> used = new HashSet<>(newEntries.size());
    Set<BibEntry> notMatched = new HashSet<>(originalEntries.size());

    // Loop through the entries of the original database, looking for exact
matches in the new one.
    // We must finish scanning for exact matches before looking for near
matches, to avoid an exact
    // match being "stolen" from another entry.
    mainLoop:
    for (BibEntry originalEntry : originalEntries) {
        // First check if the similarly placed entry in the other base
matches exactly.
        if (!used.contains(positionNew) && (positionNew <
newEntries.size())) {
            double score =
DuplicateCheck.compareEntriesStrictly(originalEntry,
newEntries.get(positionNew));
            if (score > 1) {
                used.add(positionNew);
                positionNew++;
                continue;
            }
        }
        // No? Then check if another entry matches exactly.
        for (int i = positionNew + 1; i < newEntries.size(); i++) {
            if (!used.contains(i)) {
                double score =
DuplicateCheck.compareEntriesStrictly(originalEntry, newEntries.get(i));
                if (score > 1) {
                    used.add(i);
                    continue mainLoop;
                }
            }
        }

        // No? Add this entry to the list of non-matched entries.
        notMatched.add(originalEntry);
    }

    // Now we've found all exact matches, look through the remaining
entries, looking for close matches.
    for (Iterator<BibEntry> iteratorNotMatched = notMatched.iterator();
```

```
iteratorNotMatched.hasNext(); ) {
        BibEntry originalEntry = iteratorNotMatched.next();

        // These two variables will keep track of which entry most closely
matches the one we're looking at.
        double bestMatch = 0;
        int bestMatchIndex = -1;
        if (positionNew < (newEntries.size() - 1)) {
            for (int i = positionNew; i < newEntries.size(); i++) {
                if (!used.contains(i)) {
                    double score =
DuplicateCheck.compareEntriesStrictly(originalEntry, newEntries.get(i));
                    if (score > bestMatch) {
                        bestMatch = score;
                        bestMatchIndex = i;
                    }
                }
            }
        }

        if (bestMatch > MATCH_THRESHOLD) {
            used.add(bestMatchIndex);
            iteratorNotMatched.remove();

            differences.add(new BibEntryDiff(originalEntry,
newEntries.get(bestMatchIndex)));
        } else {
            differences.add(new BibEntryDiff(originalEntry, null));
        }
    }

    // Finally, look if there are still untouched entries in the new
database. These may have been added.
    for (int i = 0; i < newEntries.size(); i++) {
        if (!used.contains(i)) {
            differences.add(new BibEntryDiff(null, newEntries.get(i)));
        }
    }

    return differences;
}
```

it can be found in: **jabref > logic > bibtex > comparator > BibDatabaseDiff.java**, lines 43-122

This method is extensive and should be divided in smaller parts for easier comprehension instead of explained through a large quantity of comments.

Second code smell: Comments take a 'reminder' nature,

```
public int compare(BibEntry e1, BibEntry e2) {
    // default equals
    // TODO: with the new default equals this does not only return 0 for
identical objects,
    //       but for all objects that have the same id and same fields
    if (Objects.equals(e1, e2)) {
        return 0;
```

```
    }

    Object f1 = e1.getField(sortField).orElse(null);
    Object f2 = e2.getField(sortField).orElse(null);

    if (binary) {
        // We just separate on set and unset fields:
        if (f1 == null) {
            return f2 == null ? (next == null ? idCompare(e1, e2) :
next.compare(e1, e2)) : 1;
        } else {
            return f2 == null ? -1 : (next == null ? idCompare(e1, e2) :
next.compare(e1, e2));
        }
    }

    // If the field is author or editor, we rearrange names so they are
    // sorted according to last name.
    if (sortField.getProperties().contains(FieldProperty.PERSON_NAMES)) {
        if (f1 != null) {
            f1 = AuthorList.fixAuthorForAlphabetization((String)
f1).toLowerCase(Locale.ROOT);
        }
        if (f2 != null) {
            f2 = AuthorList.fixAuthorForAlphabetization((String)
f2).toLowerCase(Locale.ROOT);
        }
    } else if (sortField.equals(InternalField.TYPE_HEADER)) {
        // Sort by type.
        f1 = e1.getType();
        f2 = e2.getType();
    } else if (sortField.equals(InternalField.KEY_FIELD)) {
        f1 = e1.getCitationKey().orElse(null);
        f2 = e2.getCitationKey().orElse(null);
    } else if (sortField.isNumeric()) {
        try {
            Integer i1 = Integer.parseInt((String) f1);
            Integer i2 = Integer.parseInt((String) f2);
            // Ok, parsing was successful. Update f1 and f2:
            f1 = i1;
            f2 = i2;
        } catch (NumberFormatException ex) {
            // Parsing failed. Give up treating these as numbers.
            // TODO: should we check which of them failed, and sort based
on that?
        }
    }

    if (f2 == null) {
        if (f1 == null) {
            return next == null ? idCompare(e1, e2) : next.compare(e1, e2);
        } else {
            return -1;
        }
    }

    if (f1 == null) { // f2 != null here automatically
        return 1;
    }

    int result;
```

```
    if ((f1 instanceof Integer) && (f2 instanceof Integer)) {
        result = ((Integer) f1).compareTo((Integer) f2);
    } else if (f2 instanceof Integer) {
        Integer f1AsInteger = Integer.valueOf(f1.toString());
        result = f1AsInteger.compareTo((Integer) f2);
    } else if (f1 instanceof Integer) {
        Integer f2AsInteger = Integer.valueOf(f2.toString());
        result = ((Integer) f1).compareTo(f2AsInteger);
    } else {
        String ours = ((String) f1).toLowerCase(Locale.ROOT);
        String theirs = ((String) f2).toLowerCase(Locale.ROOT);
        int comp = ours.compareTo(theirs);
        result = comp;
    }
    if (result != 0) {
        return descending ? -result : result; // Primary sort.
    }
    if (next == null) {
        return idCompare(e1, e2); // If still equal, we use the unique IDs.
    } else {
        return next.compare(e1, e2); // Secondary sort if existent.
    }
}
```

it can be found in: **jabref > logic > bibtex > comparator > EntryComparator.java**, lines 51-52 and 93-94

These comments refer to things that are yet to be done. Having this method call other methods which took care of these parts would allow for easier understanding and implementation of such parts.

Third code smell: Feature envy,

```
/**
 * @implNote Should be kept in sync with {@link MetaData#equals(Object)}
 */
public List<String> getDifferences(PreferencesService preferences) {
    List<String> changes = new ArrayList<>();

    if (originalMetaData.isProtected() != newMetaData.isProtected()) {
        changes.add(Localization.lang("Library protection"));
    }
    if (!Objects.equals(originalMetaData.getGroups(),
newMetaData.getGroups())) {
        changes.add(Localization.lang("Modified groups tree"));
    }
    if (!Objects.equals(originalMetaData.getEncoding(),
newMetaData.getEncoding())) {
        changes.add(Localization.lang("Library encoding"));
    }
    if (!Objects.equals(originalMetaData.getSaveOrderConfig(),
newMetaData.getSaveOrderConfig())) {
        changes.add(Localization.lang("Save sort order"));
    }
```

```
    if
(!Objects.equals(originalMetaData.getCiteKeyPattern(preferences.getGlobalCi
tationKeyPattern()),
newMetaData.getCiteKeyPattern(preferences.getGlobalCitationKeyPattern())))
{
        changes.add(Localization.lang("Key patterns"));
    }
    if (!Objects.equals(originalMetaData.getUserFileDirectories(),
newMetaData.getUserFileDirectories())) {
        changes.add(Localization.lang("User-specific file directory"));
    }
    if (!Objects.equals(originalMetaData.getLatexFileDirectories(),
newMetaData.getLatexFileDirectories())) {
        changes.add(Localization.lang("LaTeX file directory"));
    }
    if (!Objects.equals(originalMetaData.getDefaultCiteKeyPattern(),
newMetaData.getDefaultCiteKeyPattern())) {
        changes.add(Localization.lang("Default pattern"));
    }
    if (!Objects.equals(originalMetaData.getSaveActions(),
newMetaData.getSaveActions())) {
        changes.add(Localization.lang("Save actions"));
    }
    if (originalMetaData.getMode() != newMetaData.getMode()) {
        changes.add(Localization.lang("Library mode"));
    }
    if (!Objects.equals(originalMetaData.getDefaultFileDirectory(),
newMetaData.getDefaultFileDirectory())) {
        changes.add(Localization.lang("General file directory"));
    }
    if (!Objects.equals(originalMetaData.getContentSelectors(),
newMetaData.getContentSelectors())) {
        changes.add(Localization.lang("Content selectors"));
    }
    return changes;
}
```

it can be found in: **jabref > logic > bibtex > comparator > MetaDataDiff.java**, lines 32-75

This method having to be kept 'in sync' manually with **jabref > model > metadata > MetaData.java** class shows an inappropriate use of classes, having MetaDataDiff more interested in MetaData than it is on itself. This class is aimed at identifying and recording the changes made to MetaData and should therefore be part of, an extension or extending the class MetaData.

CODE SMELLS

The 3 code smells I identified are the following:

Long Method

```java
public static void openExternalViewer(BibDatabaseContext databaseContext,
                                      PreferencesService preferencesService,
                                      String initialLink,
                                      Field initialField)
        throws IOException {
    String link = initialLink;
    Field field = initialField;
    if (StandardField.PS.equals(field) || StandardField.PDF.equals(field)) {
        // Find the default directory for this field type:
        List<Path> directories = databaseContext.getFileDirectories(preferencesService.getFilePreferences());

        Optional<Path> file = FileHelper.find(link, directories);

        // Check that the file exists:
        if (file.isEmpty() || !Files.exists(file.get())) {
            throw new IOException("File not found (" + field + "): '" + link + "'.");
        }
        link = file.get().toAbsolutePath().toString();

        // Use the correct viewer even if pdf and ps are mixed up:
        String[] split = file.get().getFileName().toString().split( regex: "\\.");
        if (split.length >= 2) {
            if ("pdf".equalsIgnoreCase(split[split.length - 1])) {
                field = StandardField.PDF;
            } else if ("ps".equalsIgnoreCase(split[split.length - 1])
                    || ((split.length >= 3) && "ps".equalsIgnoreCase(split[split.length - 2]))) {
                field = StandardField.PS;
            }
        }
    } else if (StandardField.DOI.equals(field)) {
        openDoi(link);
```

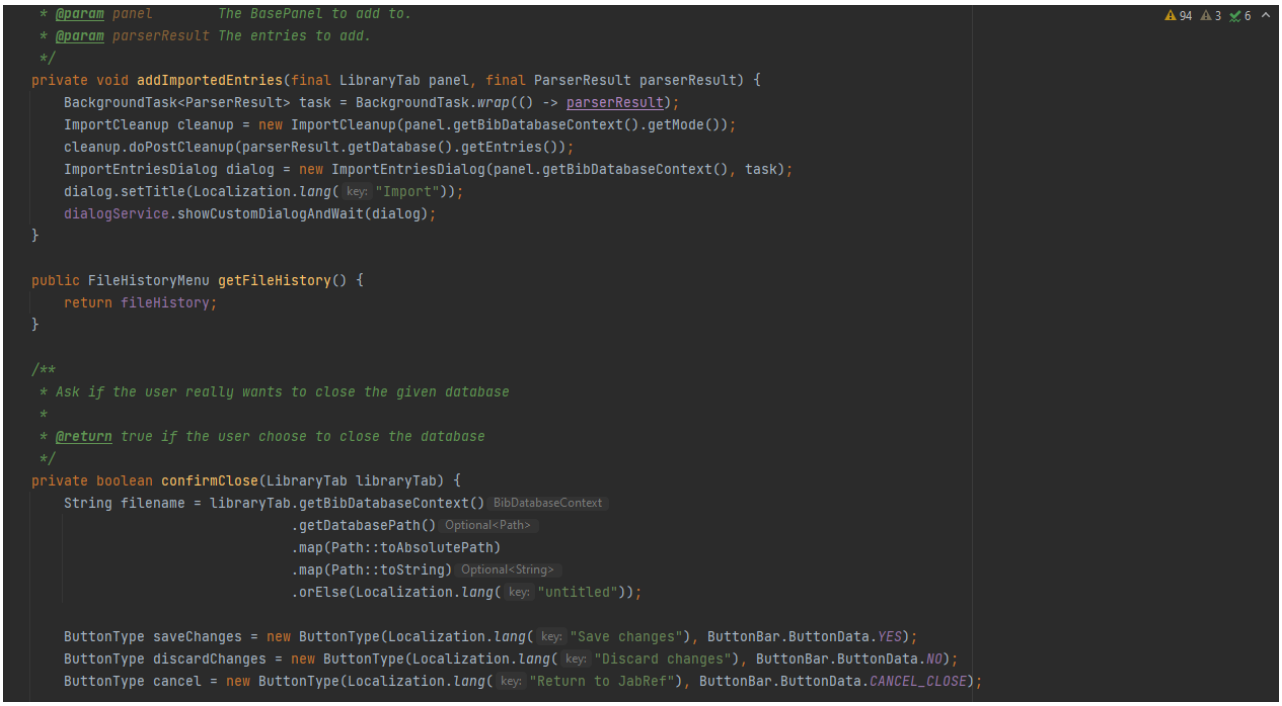
```java
            openDoi(link);
            return;
        } else if (StandardField.EPRINT.equals(field)) {
            link = ArXivIdentifier.parse(link) Optional<ArXivIdentifier>
                                  .map(ArXivIdentifier::getExternalURI) Optional<Optional<URI>>
                                  .filter(Optional::isPresent)
                                  .map(Optional::get) Optional<URI>
                                  .map(URI::toASCIIString) Optional<String>
                                  .orElse(link);
            // should be opened in browser
            field = StandardField.URL;
        }

        if (StandardField.URL.equals(field)) {
            openBrowser(link);
        } else if (StandardField.PS.equals(field)) {
            try {
                NATIVE_DESKTOP.openFile(link, StandardField.PS.getName());
            } catch (IOException e) {
                LOGGER.error("An error occurred on the command: " + link, e);
            }
        } else if (StandardField.PDF.equals(field)) {
            try {
                NATIVE_DESKTOP.openFile(link, StandardField.PDF.getName());
            } catch (IOException e) {
                LOGGER.error("An error occurred on the command: " + link, e);
            }
        } else {
            LOGGER.info("Message: currently only PDF, PS and HTML files can be opened by double clicking");
        }
    }
```

This code smell can be found in the following path: jabref -> src -> main -> java -> org.jabref -> gui -> desktop -> JabRefDesktop.java (lines 54 to 114)

The first code smell identified is of type "Long Method". As the name suggests, this code smell is related to methods that are too long and too complex, like we see in this case.

Refactoring suggestion: Given the code snippet, one way to refactor this method is by separating it into smaller methods, each one with a certain functionality. For example, we can create a separate method to find the default directory for some field type or using the correct viewer (based on the comments inside of this method).

Large Class



```
 * @param panel        The BasePanel to add to.
 * @param parserResult The entries to add.
 */
private void addImportedEntries(final LibraryTab panel, final ParserResult parserResult) {
    BackgroundTask<ParserResult> task = BackgroundTask.wrap(() -> parserResult);
    ImportCleanup cleanup = new ImportCleanup(panel.getBibDatabaseContext().getMode());
    cleanup.doPostCleanup(parserResult.getDatabase().getEntries());
    ImportEntriesDialog dialog = new ImportEntriesDialog(panel.getBibDatabaseContext(), task);
    dialog.setTitle(Localization.lang( key: "Import"));
    dialogService.showCustomDialogAndWait(dialog);
}

public FileHistoryMenu getFileHistory() {
    return fileHistory;
}

/**
 * Ask if the user really wants to close the given database
 *
 * @return true if the user choose to close the database
 */
private boolean confirmClose(LibraryTab libraryTab) {
    String filename = libraryTab.getBibDatabaseContext() BibDatabaseContext
                                .getDatabasePath() Optional<Path>
                                .map(Path::toAbsolutePath)
                                .map(Path::toString) Optional<String>
                                .orElse(Localization.lang( key: "untitled"));

    ButtonType saveChanges = new ButtonType(Localization.lang( key: "Save changes"), ButtonBar.ButtonData.YES);
    ButtonType discardChanges = new ButtonType(Localization.lang( key: "Discard changes"), ButtonBar.ButtonData.NO);
    ButtonType cancel = new ButtonType(Localization.lang( key: "Return to JabRef"), ButtonBar.ButtonData.CANCEL_CLOSE);
```

This code smell can be found in the following path: jabref -> src -> main -> java -> org.jabref -> gui -> JabRefFrame.java

This case is an example of a “Large Class”, which is also a code smell. Through this snippet alone, we can see three completely unrelated methods already, which can give us a glimpse of how diversified this class’ functionalities are. Knowing that JabRefFrame.java has about 1300 lines of code, we can conclude that this class alone has a great responsibility to the entire system, and that’s a big problem, since it can serve as a black hole, meaning that this amount of responsibilities can attract more and more responsibilities in the future.

Refactoring suggestion: Although it can become very time-consuming to fix large classes due to the numerous links they have with other classes, one way to solve this problem is by separating them into different classes, with different functionalities. For example, using the presented code snippet, if there is more methods related to imports just like

addImportedEntries(), we could create a class only dedicated to import functionalities. We could use a similar strategy for other functionalities.

Long Parameter List

```java
/**
 * @param cursor Where to insert.
 * @param pageInfo A single pageInfo for a list of entries. This is what we get from the GUI.
 */
public static void insertCitationGroup(XTextDocument doc,
                                       OOFrontend frontend,
                                       XTextCursor cursor,
                                       List<BibEntry> entries,
                                       BibDatabase database,
                                       OOBibStyle style,
                                       CitationType citationType,
                                       String pageInfo)
    throws
    NoDocumentException,
    NotRemoveableException,
    WrappedTargetException,
    PropertyVetoException,
    CreationException,
    IllegalTypeException {

    List<String> citationKeys = OOListUtil.map(entries, EditInsert::insertEntryGetCitationKey);

    final int totalEntries = entries.size();
    List<Optional<OOText>> pageInfos = OODataModel.fakePageInfos(pageInfo, totalEntries);

    List<CitationMarkerEntry> citations = new ArrayList<>(totalEntries);
    for (int i = 0; i < totalEntries; i++) {
        Citation cit = new Citation(citationKeys.get(i));
        cit.lookupInDatabases(Collections.singletonList(database));
        cit.setPageInfo(pageInfos.get(i));
        citations.add(cit);
    }
```

This code smell can be found in the following path: jabref -> src -> main -> java -> org.jabref -> logic -> openoffice -> action -> EditInsert.java (lines 59 to 66)

In this code snippet, we can see that the insertCitationGroup() method has 8 parameters, which is considered an excessive amount of parameters for just one method. Therefore we can conclude that this illustrates the "Long parameter list" code smell.

Refactoring suggestion: One way to reduce the amount of parameters is by creating parameter objects. For example, we could associate "style", "citationType" and "frontend" into just one object and passing it as a parameter.

Code Smells

Bellow are the 3 code smells I identified as was delegated to during the team meetings.

The first Code Smell Identified was of type: “Duplicate Code”.

```
299                 if ((i + 1) < layoutEntries.size()) {
300                     if (layoutEntries.get(i + 1).doLayout(bibtex, database).trim().isEmpty()) {
301                         i++;
302                         previousSkipped = true;
303                         continue;
304                     }
```

This code smell is located in the following path: jabref -> src -> main -> java -> org. jabref -> logic -> layout -> format -> LayoutEntry.java (lines 299 through 304).

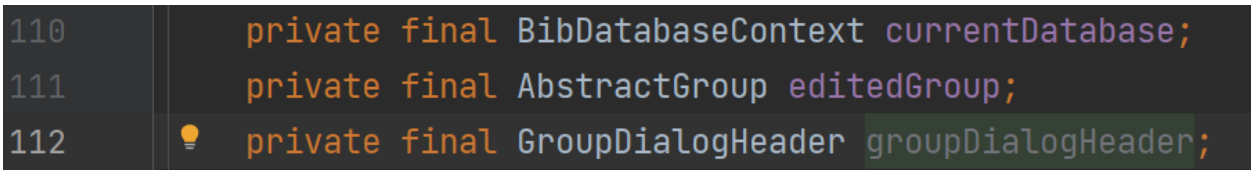
In this case the collapsible ***if*** statements in line 299 and 300 could be merged to increase readability.

Refactoring suggestion:

```
if (fieldText == null) {
    if ( ((i + 1) < layoutEntries.size()) && (layoutEntries.get(i + 1).doLayout(bibtex, database).trim().isEmpty()) ){
            i++;
            previousSkipped = true;
            continue;
    }
```

Code Smells

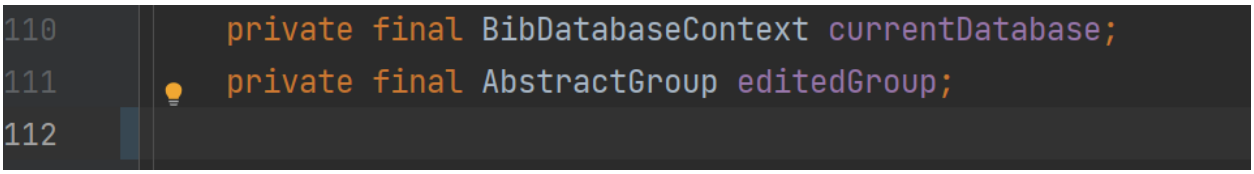
The second Code Smell Identified was of type: “Dead Code”.



This code smell is located in the following path: jabref -> src -> main -> java -> org. jabref -> gui -> groups -> GroupDialogViewModel.java (lines 110 through 112).

If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for.

Refactoring suggestion:



Code Smells

The third Code Smell Identified was of type: “Dispensable Comments”.

```
/**
 * This is an immutable class representing information of either <CODE>author</CODE> or <CODE>editor</CODE> field in
 * <p>
 * Constructor performs parsing of raw field text and stores preformatted data. Various accessor methods return autho
 * <p>
 * Parsing algorithm is designed to satisfy two requirements: (a) when author's name is typed correctly, the result s
 * <ol>
 * <li> 'author field' is a sequence of tokens;
 * <ul>
 * <li> tokens are separated by sequences of whitespaces (<CODE>Character.isWhitespace(c)==true</CODE>),
 * commas (,), dashes (-), and tildas (~);
 * <li> every comma separates tokens, while sequences of other separators are
 * equivalent to a single separator; for example: "a - b" consists of 2 tokens
 * ("a" and "b"), while "a,-,b" consists of 3 tokens ("a", "", and "b")
 * <li> anything enclosed in braces belongs to a single token; for example:
 * "abc x{a,b,-~ c}x" consists of 2 tokens, while "abc xa,b,-~ cx" consists of 4
 * tokens ("abc", "xa","b", and "cx");
 * <li> a token followed immediately by a dash is "dash-terminated" token, and
 * all other tokens are "space-terminated" tokens; for example: in "a-b- c - d"
 * tokens "a" and "b" are dash-terminated and "c" and "d" are space-terminated;
 * <li> for the purposes of splitting of 'author name' into parts and
 * construction of abbreviation of first name, one needs definitions of first
 * letter of a token, case of a token, and abbreviation of a token:
 * <ul>
 * <li> 'first letter' of a token is the first letter character (<CODE>Character.isLetter(c)==true</CODE>)
 * that does not belong to a sequence of letters that immediately follows "\"
 * character, with one exception: if "\" is followed by "aa", "AA", "ae", "AE",
 * "l", "L", "o", "O", "oe", "OE", "i", or "j" followed by non-letter, the
 * 'first letter' of a token is a letter that follows "\"; for example: in
```

This code smell is located in the following path: jabref -> src -> main -> java -> org. jabref -> model -> entry -> types -> AuthorList.java (lines 13 through 114).

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand, in this case there’s a huge block of comments that could be completely removed.

Refactoring suggestion: Delete the comment block.

Design Patterns

First *design pattern*: Adapter Pattern

```
/**
 * Adapter class for the latex2unicode lib. This is an alternative to
our LatexToUnicode class
 */
public class LatexToUnicodeAdapter {

        ...
```

This adapter design pattern can be found in **jabref > model > strings > LatexToUnicodeAdapter.**

This class acts as an adapter class, which converts LaTeX into Unicode characters, in order for those strings to be compatible with other classes.

Second *design pattern*: Facade Pattern

```
/**
 * Facade to unify the access to the citation style engine. Use these
methods if you need rendered BibTeX item(s) in a
 * given journal style. This class uses {@link CSLAdapter} to create
output.
 */
public class CitationStyleGenerator {

        ...
```

This facade design pattern can be found in **jabref > logic > citationstyle > CitationStyleGenerator.**

This class acts as a wrapper class that unifies the whole “citation style engine”-related classes.

Third *design pattern*: <u>Factory Pattern</u>

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

        ...
```

This factory design pattern can be found in **jabref > gui > actions > ActionFactory.**

This class's purpose is to hide the creation of instances of a given type (in this case, menus, items, icons, etc) behind an interface.

Francisco Pires, nº 58208

Design Patterns

The first design pattern: Singleton Class

```
private BibtexCaseChanger() {
}
```

Located in: **jabref > logic > bst > BibtextCaseChanger.java**

A private constructor, as the above, ensures there is only one instance of this class to be produced and exist throughout an application's lifetime and guarantees that a single global point of access exists to it.

The second design pattern: Template Method

```
/**
 * Base class for export formats based on templates.
 */
public class TemplateExporter extends Exporter {
```

Located in: **jabref > logic > exporter > TemplateExporter.java**

This class creates a template implementation for all export formats, leaving the implementation to its subclasses.

The third design pattern: Factory Pattern

```
public class ExporterFactory {
```

Located in: **jabref > logic > exporter > ExporterFactory.java**

The class implements a factory creation method for the many exports, hiding their implementation behind an interface.

Gonçalo Vicêncio, nº 57944

Design Patterns

The first Design Pattern Identified is: Caching Pattern.

```
20  public class CitationStyleCache {
21
22      private static final int CACHE_SIZE = 1024;
23
24      private PreviewLayout citationStyle;
25      private final LoadingCache<BibEntry, String> citationStyleCache;
26
27  @   public CitationStyleCache(BibDatabaseContext database) {
```

It can be found in the path: jabref -> src -> main -> java -> org. jabref -> logic -> citationstyle -> CitationStyleCache.java

The caching pattern avoids expensive re-acquisition of resources by not releasing them immediately after use. The resources retain their identity, are kept in some fast-access storage, and are re-used to avoid having to acquire them again.

In this case the class: **CitationStyleCache.java** is going to work on caching citations for quicker access, while **CitationStyleGenerator.java** will generate them slower.

Design Patterns

The second Design Pattern Identified is: Registry Pattern.

```
116  public class AuthorList {
117
118      private static final WeakHashMap<String, AuthorList> AUTHOR_CACHE = new WeakHashMap<>();
119      private final List<Author> authors;
120      private AuthorList latexFreeAuthors;
121
122      /**
123       * Creates a new list of authors.
```

It can be found in the path: jabref -> src -> main -> java -> org. jabref -> model -> entry -> types -> AuthorList.java

The Registry Pattern denotes the storage of the objects of a single class and provides a global point of access to them. Similar to Multiton pattern, only difference is that in a registry there is no restriction on the number of objects.

In this case the class: **AuthorList.java** serves as a registry of objects of type **Author.java.**

Design Patterns

The third Design Pattern Identified is: Singleton Pattern.

```
12  public class AppearancePreferences {
13      private final BooleanProperty shouldOverrideDefaultFontSize;
14      private final IntegerProperty mainFontSize;
15      private final ObjectProperty<Theme> theme;
16
```

```
428     private PreviewPreferences previewPreferences;
429     private SidePanePreferences sidePanePreferences;
430     private AppearancePreferences appearancePreferences;
431     private ImporterPreferences importerPreferences;
```

Instancing of the Singleton.

It can be found in the path: jabref -> src -> main -> java -> org. jabref -> preferences -> AppearencePreferences.java

This design pattern ensures a class only has one instance and provides a global point of access to it.

Particularly, the **AppearancePreferences.java** class is only instantiated once, in the **JabRefPreferences.java** class.

Design Patterns

Below I describe the three design patterns I found, as proposed.

The first “design pattern”: Singleton Class

```
private JabRefPreferences() {
    try {
        if (new File("jabref.xml").exists()) {
            importPreferences(Path.of("jabref.xml"));
        }
    } catch (JabRefException e) {
        LOGGER.warn("Could not import preferences from jabref.xml: " +
e.getMessage(), e);
    }

    // load user preferences
    prefs = PREFS_NODE;


        ...
```

This block of code is located in **jabref > preferences > JabRefPreferences.java**

A private constructor, such as the above, ensures there is only one instance of this class to be produced and exist throughout an application's lifetime and guarantee that exists a global point of access to it.

The second “design pattern”: Adapter Class

```
/**
 * Provides an adapter class to CSL. It holds a CSL instance under the
*hood that is only recreated when
 * the style changes.
*/

public class CSLAdapter {

        ...
```

This block of code is located in **jabref > logic > citationstyle > CSLAdapter.**

As described in the comment this class acts as an adapter class, converting incompatible objects to make them compatible with other classes. In this exemple, converting the bibEntry from our own representation to a representation that the CLS library uses to create the preview string, for example.

The third “design pattern”: Template Method

```
DBMSProcessor.java

/**
 * Creates and sets up the needed tables and columns according to the
database type.
 *
 * @throws SQLException
 */
protected abstract void setUp() throws SQLException;

/**
 * Escapes parts of SQL expressions such as a table name or a field
name to match the conventions of the database
 * system using the current dbmsType.
 * <p>
 * This method is package private, because of DBMSProcessorTest
 *
 * @param expression Table or field name
 * @return Correctly escaped expression
 */
abstract String escape(String expression);


--------------------------------------------------------------------------
MySQLProcessor.java

@Override
public void setUp() throws SQLException {
    connection.createStatement().executeUpdate(
            "CREATE TABLE IF NOT EXISTS `ENTRY` (" +
                    "`SHARED_ID` INT(11) NOT NULL PRIMARY KEY
AUTO_INCREMENT, " +
                    "`TYPE` VARCHAR(255) NOT NULL, " +
                    "`VERSION` INT(11) DEFAULT 1)");

    connection.createStatement().executeUpdate(
            "CREATE TABLE IF NOT EXISTS `FIELD` (" +
                    "`ENTRY_SHARED_ID` INT(11) NOT NULL, " +
                    "`NAME` VARCHAR(255) NOT NULL, " +
                    "`VALUE` TEXT DEFAULT NULL, " +
                    "FOREIGN KEY (`ENTRY_SHARED_ID`) REFERENCES
`ENTRY`(`SHARED_ID`) ON DELETE CASCADE)");

    connection.createStatement().executeUpdate(
            "CREATE TABLE IF NOT EXISTS `METADATA` (" +
                    "`KEY` varchar(255) NOT NULL," +
                    "`VALUE` text NOT NULL)");
}

@Override
String escape(String expression) {
    return "`" + expression + "`";
}

--------------------------------------------------------------------------
```

```
class DBMSProcessorTest {

    private DBMSConnection dbmsConnection;
    private DBMSProcessor dbmsProcessor;
    private DBMSType dbmsType;

    @BeforeEach
    public void setup() throws Exception {
        this.dbmsType = TestManager.getDBMSTypeTestParameter();
        this.dbmsConnection =
TestConnector.getTestDBMSConnection(dbmsType);
        this.dbmsProcessor =
DBMSProcessor.getProcessorInstance(TestConnector.getTestDBMSConnection
(dbmsType));
```

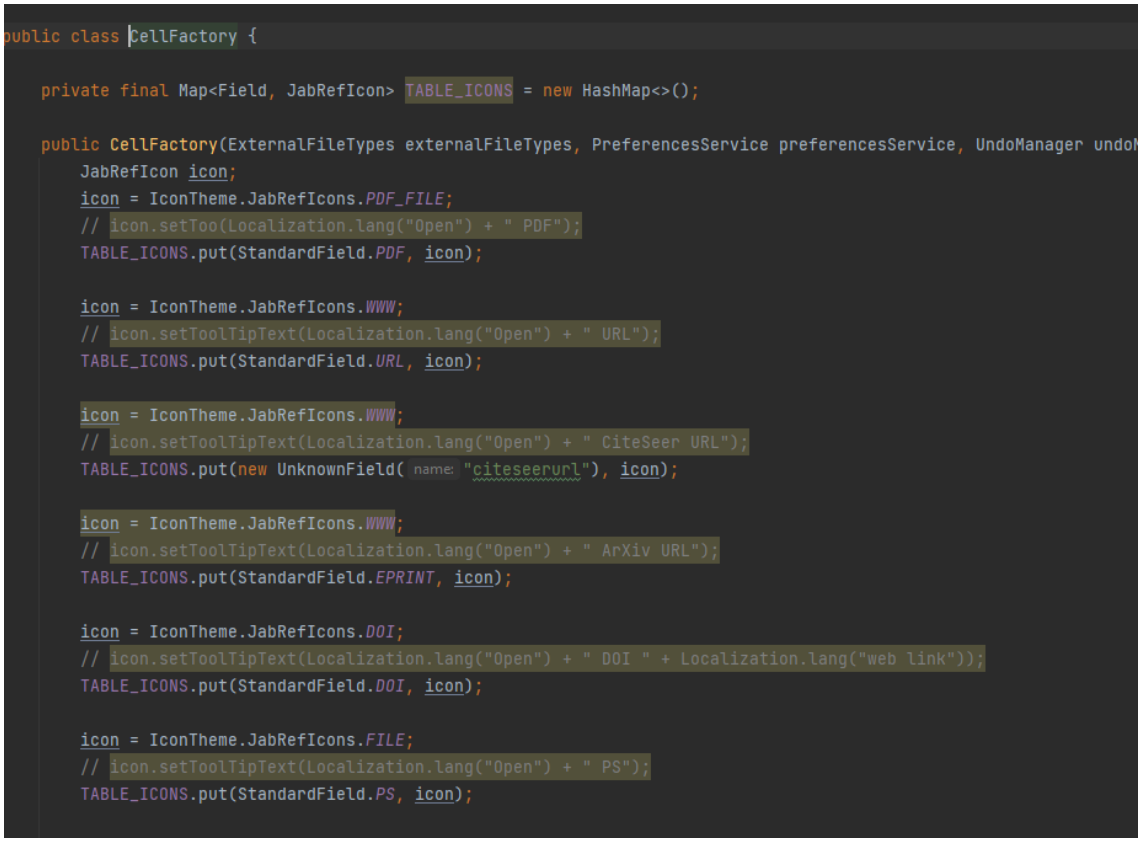
This block of code is located in **jabref > logic > shared > DBMSProcessorTest.java.**

The abstract class DBMSProcessor defines the skeleton of the methods and describes them in a general way, leaving the implementation and details to its subclasses. These methods are abstract in the DBMSProcessor class and consequently implemented in the following classes: OracleProcessor, MySQLProcessor, PostgreSQLProcessor, to be used in the class DBMSProcessorTest.

DESIGN PATTERNS

The 3 Design Patterns I identified were the following:

FACTORY METHOD PATTERN

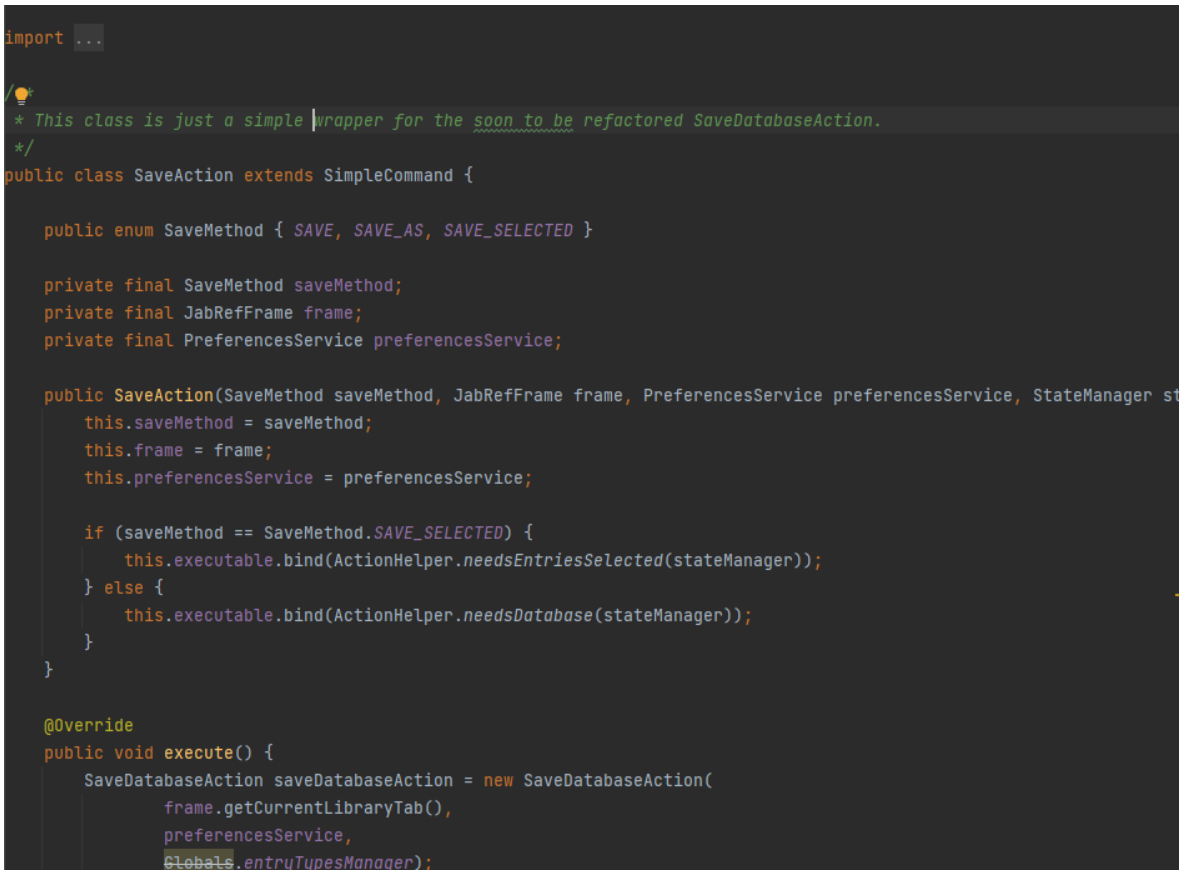


```
public class CellFactory {

    private final Map<Field, JabRefIcon> TABLE_ICONS = new HashMap<>();

    public CellFactory(ExternalFileTypes externalFileTypes, PreferencesService preferencesService, UndoManager undoM
        JabRefIcon icon;
        icon = IconTheme.JabRefIcons.PDF_FILE;
        // icon.setToo(Localization.lang("Open") + " PDF");
        TABLE_ICONS.put(StandardField.PDF, icon);

        icon = IconTheme.JabRefIcons.WWW;
        // icon.setToolTipText(Localization.lang("Open") + " URL");
        TABLE_ICONS.put(StandardField.URL, icon);

        icon = IconTheme.JabRefIcons.WWW;
        // icon.setToolTipText(Localization.lang("Open") + " CiteSeer URL");
        TABLE_ICONS.put(new UnknownField( name: "citeseerurl"), icon);

        icon = IconTheme.JabRefIcons.WWW;
        // icon.setToolTipText(Localization.lang("Open") + " ArXiv URL");
        TABLE_ICONS.put(StandardField.EPRINT, icon);

        icon = IconTheme.JabRefIcons.DOI;
        // icon.setToolTipText(Localization.lang("Open") + " DOI " + Localization.lang("web link"));
        TABLE_ICONS.put(StandardField.DOI, icon);

        icon = IconTheme.JabRefIcons.FILE;
        // icon.setToolTipText(Localization.lang("Open") + " PS");
        TABLE_ICONS.put(StandardField.PS, icon);
```

This "Factory Method" pattern can be found in jabref -> src -> main ->java -> org.jabref -> gui -> maintainable -> CellFactory.java

We can say that CellFactory.java uses a "Factory Method" pattern since it was designed for hiding the whole process of creating certain types of instances, such as icons, external file types or special fields, for example.

FAÇADE PATTERN



```java
import ...

/*
 * This class is just a simple wrapper for the soon to be refactored SaveDatabaseAction.
 */
public class SaveAction extends SimpleCommand {

    public enum SaveMethod { SAVE, SAVE_AS, SAVE_SELECTED }

    private final SaveMethod saveMethod;
    private final JabRefFrame frame;
    private final PreferencesService preferencesService;

    public SaveAction(SaveMethod saveMethod, JabRefFrame frame, PreferencesService preferencesService, StateManager st
        this.saveMethod = saveMethod;
        this.frame = frame;
        this.preferencesService = preferencesService;

        if (saveMethod == SaveMethod.SAVE_SELECTED) {
            this.executable.bind(ActionHelper.needsEntriesSelected(stateManager));
        } else {
            this.executable.bind(ActionHelper.needsDatabase(stateManager));
        }
    }

    @Override
    public void execute() {
        SaveDatabaseAction saveDatabaseAction = new SaveDatabaseAction(
                frame.getCurrentLibraryTab(),
                preferencesService,
                Globals.entryTypesManager);
```

This "Façade" pattern can be found in jabref -> src -> main -> java -> org.jabref -> gui -> exporter -> SaveAction.java

We can say that this class uses a "Façade" pattern since its whole purpose is to hide the complexity of a class (in this case, SaveDatabaseAction.java) and its communication with other classes (in this case, JabRefFrame.java or SaveMethod.java, for example) by providing a much simpler interface.

SINGLETON PATTERN

```java
/**
 * Creates a new auto-completion binding between the given textInputControl
 * and the given suggestion provider.
 */
private AutoCompletionTextInputBinding(final TextInputControl textInputControl,
                                       Callback<ISuggestionRequest, Collection<T>> suggestionProvider) {

    this(textInputControl,
            suggestionProvider,
            AutoCompletionTextInputBinding.defaultStringConverter(),
            new ReplaceStrategy());
}

private AutoCompletionTextInputBinding(final TextInputControl textInputControl,
                                       final Callback<ISuggestionRequest, Collection<T>> suggestionProvider,
                                       final StringConverter<T> converter) {
    this(textInputControl, suggestionProvider, converter, new ReplaceStrategy());
}

private AutoCompletionTextInputBinding(final TextInputControl textInputControl,
                                       final Callback<ISuggestionRequest, Collection<T>> suggestionProvider,
                                       final StringConverter<T> converter,
                                       final AutoCompletionStrategy inputAnalyzer) {

    super(textInputControl, suggestionProvider, converter);
    this.converter = converter;
    this.inputAnalyzer = inputAnalyzer;
```

This "Singleton" pattern can be found in jabref -> src -> main -> java -> org.jabref->gui->autocompleter-> AutoCompletionTextInputBinding.java

By analysing this snippet of code, we can see that this class uses a "Singleton" pattern. We can conclude this from the use of the "private" keyword on these 3 constructors. Its purpose is to create 3 different and unique objects which all together will contribute to the auto-completion binding mechanism.

Pedro Simões, Nº 58674

MOOD METRICS

REPORT

This report is about Mood Metrics. The Mood metrics are used to guide and assess object-oriented design quality and potential productivity gains. The given plugin presented me with many columns.

These were AHF (attribute hiding factor), AIF (attribute inheritance factor), CF (coupling factor), MHF (method hiding factor), MIF (method inheritance factor), PF (polymorphism factor).

The AHF corresponds to the percentage of invisibility of attributes in the project. It is obtained by dividing the number of visible attributes in a diagram by the number of all the attributes.

The AIF represents the percentage of effective Inheritance of attributes. It is obtained by dividing the number of all inherited attributes of all classes by the sum of all attributes available.

The CF represents the percentage of couplings amongst classes, not calculable to inheritance. Concerning the maximum probable number of couplings in the class diagram. It is obtained by dividing the number of associations between all classes by the number of classes squared minus the number of classes.

PF represents the actual number of possible different polymorphic circumstances concerning the maximum number of likely distinct polymorphic circumstances. The PF is computed by dividing the total number of overridden methods in all classes by the result of multiplying the number of new methods times the number of descendants for all classes, respectively.

The MHF depicts the percentage of invisibility of methods in a class. It is obtained by dividing the number of all visible methods in all classes by the number of all methods in the classes.

The MIF expresses the same principle as AIF, but it is the percentage of effective Inheritance of methods. It is calculated by dividing the number of all inherited methods in all classes by the sum of all methods available of all classes.

IDENTIFICATION OF POSSIBLE TROUBLE SPOTS IN THE CODEBASE

Analyzing the values of the columns, I can conclude a few things.

The ideal value of the AHF should be 100%. Our value of 78.33%, which indicates that most of the attributes are hidden e only accessible to the correspondent class methods.Ideal

An exemplary value for the AIF should be around 0% and 48%, so I would say our value is acceptable. The ideal should be zero because all methods should be private.

In regards to CF, its value should not exceed 12%. Our value is 0,68%, which means the coupling between classes is great.

The model range of the MHF is 8% to 25%, meaning that the design includes a high proportion of specialized methods that are not available for reuse.

The perfect value of MIF would be around 20% and 80%. Our value is below this range. This can be caused by a child that redefines the methods of its parents or by a class that has no children.

In regards to PF, the value should not be too high or too low. A value of 50% is excellent.

RELATION WITH CODE SMELLS

Using these metrics I could not find a relation with the identified code smells.

Chidamber-Kemerer Metrics

REPORT

In this report, I will be explaining the CK. CK is used for measuring the design of object-oriented programs. I will focus on the maintainability, understandability, and modifiability of classes. Using the MetricsReloaded plugin, we are presented with many parameters used to study the classes, such as CBO, DIT, LCOM, NOC, RFC, and WMC.

CBO, which stands for coupling between objects, is a count of the number of classes that are coupled to a particular class. In other words, where the methods of one class call the methods or access the variables of the other. In our project, some classes have numbers as high as 135 and as low as 0. A value of 0 indicates that a class has no relationship to any other class. A small value is good and indicates that a class is loosely coupled. A high number may indicate that a class is tightly coupled, this is not good and would complicate modifications. In our project, we have an average of 11.15, and even being a large project, this number seems to be not ideal.

DIT, which stands for depth of inheritance tree, is the number of ancestor classes that can affect a class. It is the maximum length from the node to the root. In our project, we have numbers as high as 7.0 and as low as 0. The higher the number, the lower is the maintainability and readability. In our project, we have an average of 1.62, which seems to be a good value.

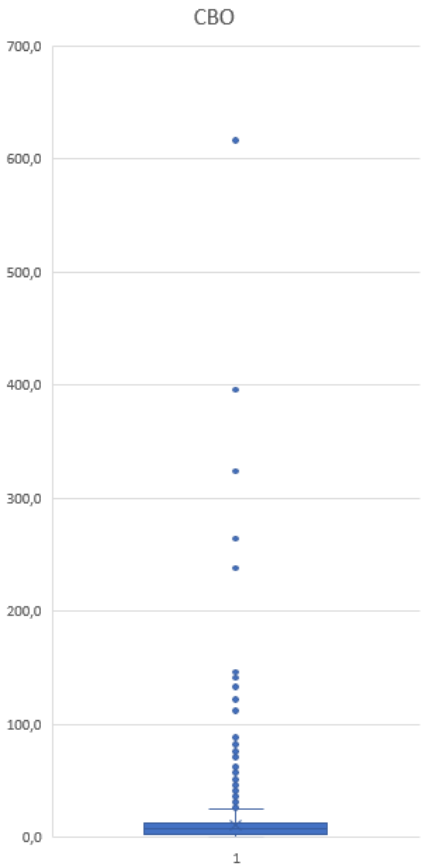
LCOM, which stands for lack of cohesion methods, is used to measure the cohesion of each class. In this project, the numbers get as high as 12 and as low as 0. A high LCOM value could indicate that the design of the class is poor. A suggestion would be to divide the class. The average in this project is 2.23, which seems to be a good value.

NOC, which stands for the number of children, is used to measure the number of classes that derive directly from the current class. In this project, the numbers get as high as 80 and as low as 0. The higher the number of children, the greater the likelihood of improper abstraction of the parent and may indicate a misuse of subclassing. The average is 0.23, which seems to be a good value, where the average range is close to 1.

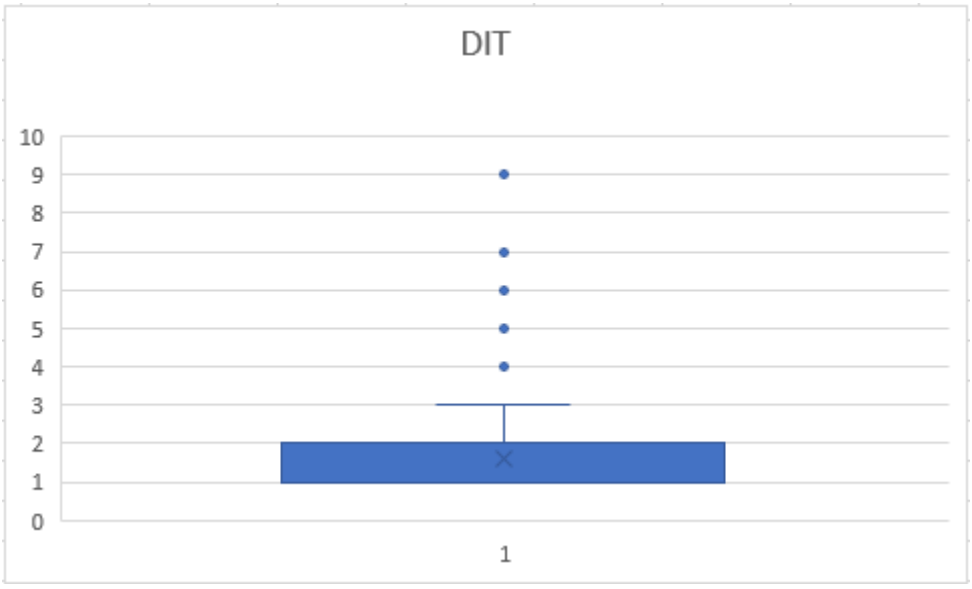
RFC, which stands for the response of class, is the total number of methods that can potentially be executed in response to a message arriving at an object of a class. It is a measure of the potential interaction of a given class with other classes, it allows us to judge the dynamics of the behavior of the corresponding object in the system. This metric characterizes the dynamic component of the external links of classes. The higher the value of RFC, the longer takes to debug and test, and the complexity of the class increases because it is. The average in this project is 23.86 which seems to be in the mean range.

WMC, which stands for weighted method complexity, corresponds to the sum of the complexity of all methods in a class. Higher values for the WMC, mean large complexity. In this project, the total complexity is equal to 20543.0 and the average corresponds to 10.89, which seems to be good values.

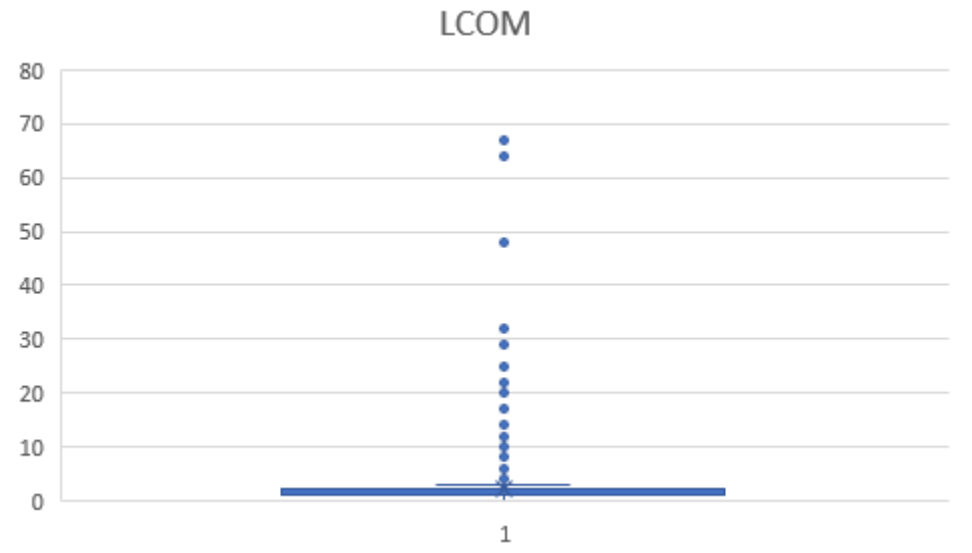
IDENTIFICATION OF POSSIBLE TROUBLE SPOTS IN THE CODEBASE



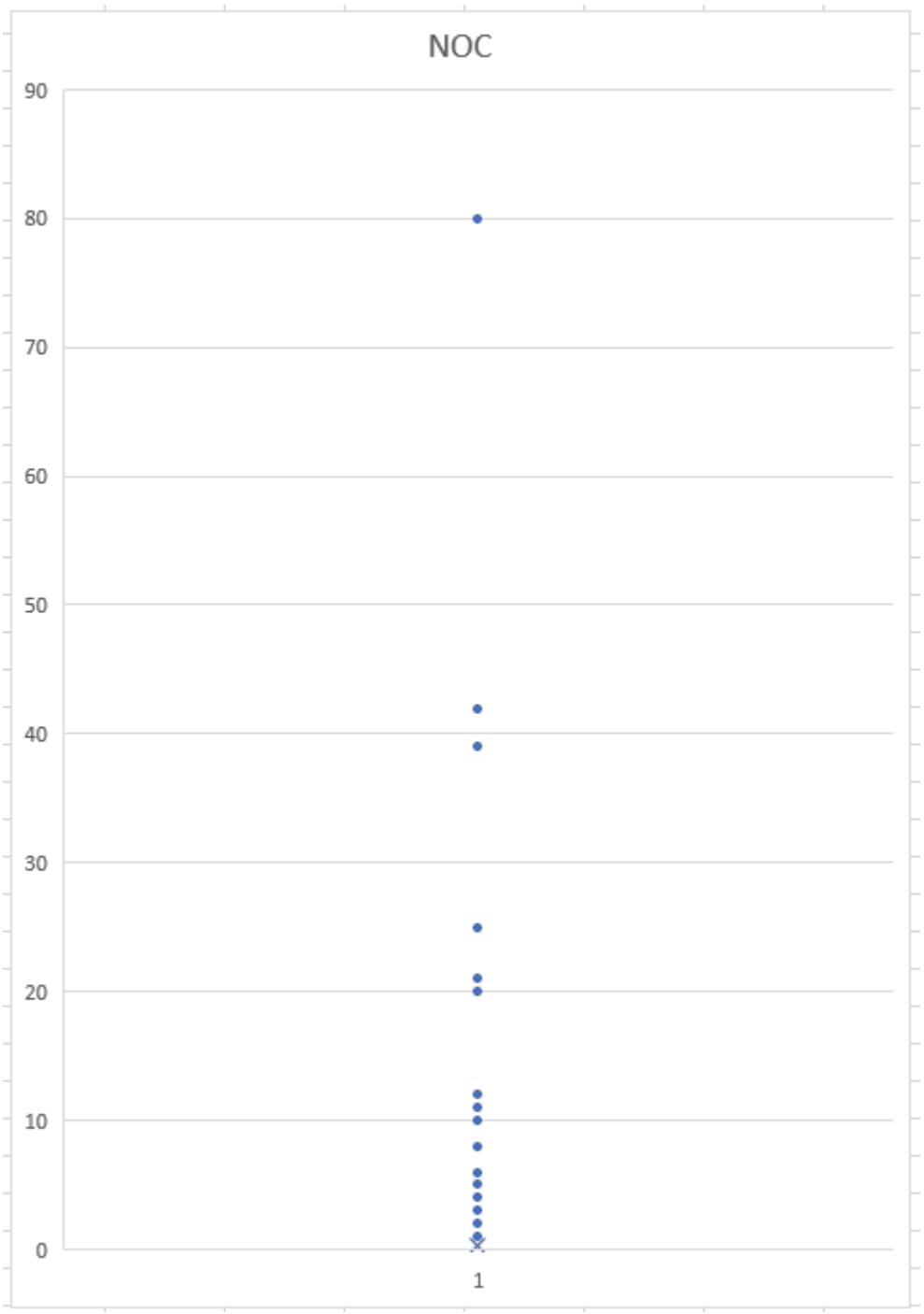
Using the CBO, I could detect two particular situations in the codebase where the value gained is much higher than the obtained average. While the mean is around 11.15, these values were 616 and 396, they correspond to the following classes: BibEntry.java and Localization.java. These situations are just an example. Many other classes have high values as well. This indicates that these classes are very tightly coupled. Not ideal.



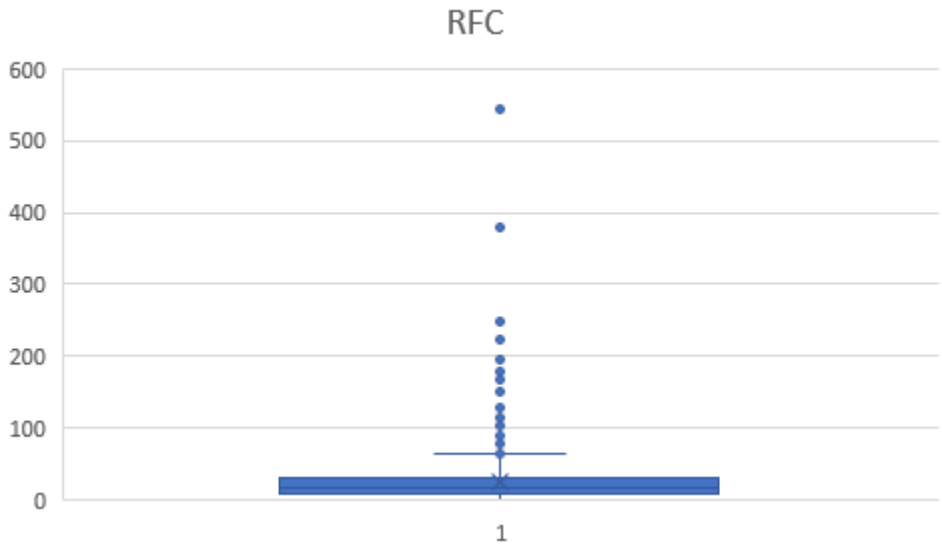
Using DIT, I could conclude that some classes exceeded the average by a lot. Two of those classes (CitationKeyPatterns.java and IconCell.java) had enormous values for the number of ancestors class, meaning that these classes will be arduous to analyze and read. While the average is 1.62, the value for both of these classes is 9.0.



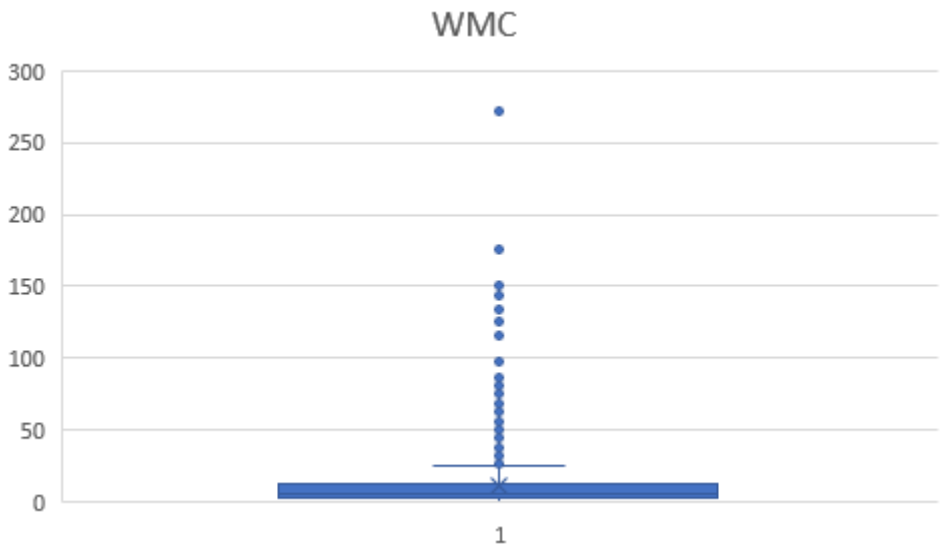
In the LCOM column, the same happens. There are particular situations where there are values considerably large. Two of those situations involve the following classes: AuthorListTest.java and TreeNodeTest.java. While the average is 2.23, the values for these classes are 67 and 64. Although these are Test classes, such values indicate there is space for improvement.



Using NOC, there is one situation in particular that differentiates from the rest. That would be the following class: SimpleCommand.java. The value for this class is 80, while the average is 0.23. An extremely important class in the project as a whole, but with an enormous number of children.



Once again, using RFC, there are two classes whose values are way over the average. These classes are JabRefPreferences.java and JabRefFrame.java. Their values are respectively 543 and 378, while the mean is 23.86. This may indicate that these classes are highly complex.



Finally, using the WMC, there is again a particular situation where the class's value exceeds the average considerably. This class is JabRefPreferences.java. Its value corresponds to 272, while the average is 10.89. Meaning that the sum of the complexity of all methods in this class is immense.In conclusion, this is a very complex class as a whole.

RELATION WITH CODE SMELLS

The classes that stood out negatively were the following: BibEntry.java, Localization.java, CitationKeyPatterns.java, IconCell.java, AuthorListTest.java, TreeNodeTest.java, SimpleCommand.java, JabRefPreferences.java and JabRefFrame.java.

Of the above mentioned, the only class our group found code smells in is JabRefFrame.java.

The two code smells found by the group are Long Method and Large Class.

This class has the following values from the metrics.

CLASS	CBO	DIT	LCOM	NOC	RFC	WMC
JabRefFrame	146	6	2	0	378	115

The Large Class code smell is proved by the values of the CBO and RFC.

The CBO indicates that many classes are coupled to JabRefFrame. As said by my colleague, this class is very diverse in terms of functionalities.

This class, having so many classes coupled proves that this class has a great responsibility to the entire system. The RFC value shows that this class has numerous methods that can potentially be executed in response to a message arriving at an object of this class. Once again demonstrates that this class has an enormous amount of responsibilities.

The Long Method is proved by the value of the WMC.

This large number indicates that the methods of this class are very complex.

This proves that some changes are welcomed, as proposed by my colleague.

Francisco Freire 58667

Martin Packaging Metrics

Collected Metrics Explanation:

Abstractness (A): $\frac{\text{number of abstract classes and interfaces in a package}}{\text{total number of classes and interfaces in a package}}$

This value ranges between 0 and 1. The closer it is to 1, the more abstract the package is.

Afferent Coupling (Ca):

The number of classes in other packages that are dependent on classes from this package. The higher the number, the higher the responsibility this package has.

Efferent Coupling (Ce):

The number of classes in this package that are dependent on classes from other packages. The higher the number, the more dependent on other packages this package is.

Distance from the main sequence (D): $|\text{abstractness} + \text{instability} - 1|$

This value ranges between 0 and 1. A package is optimal (on the main sequence) if it's balanced between abstractness and instability. The closer it is to 1, the further from the main sequence the package is.

Instability (I): $\frac{\text{efferent coupling}}{\text{efferent coupling} + \text{afferent coupling}}$

This value ranges between 0 and 1. The closer it is to 1, the more unstable the package is (regarding its adaptability to change; for example, losing access to other packages).

JabRef's packages' ***abstractness***:

Seeing as JabRef's average abstractness is 0.06, on a range from 0.00 to 1.00, we can conclude JabRef used a very small percentage of abstract classes and interfaces throughout the whole project (6%).

JabRef's packages' ***afferent and efferent couplings and instability***:

JabRef's afferent and efferent couplings total number is the same for both, 58621. Therefore, we can conclude JabRef's instability on each package is, on average, 0.5, making JabRef as a whole as stable as it is unstable. This number could be improved.

JabRef's packages' ***distance from the main sequence***:

Analyzing JabRef's distance from the main sequence, we can understand its packages are, in general, somewhat balanced between abstractness and instability, since, on average, optimality is 0.3 (the best possible value being 0.0).

These metrics do not seem to be related to any code smell we identified.

Francisco Pires, nº 58208

Class Lines Of Code Metrics

Collected Metrics Explanation:

Comment Lines Of Code (CLOC): Lines of code that are exclusively comments.
Javadoc Lines Of Code (JLOC): Lines of code that are comments pertaining to Javadoc information.
Lines Of Code (LOC): "LOC is literally the count of the number of lines of text in a file or directory."[1]

Lines Of Code is a type of metric which suffers from "fundamental" problems, due to the amount of information it dismisses, such as effort per line or usefulness of said lines.

Checking for Method LOC metrics could tell us about which methods incur into Long Method code smells, but I decided to focus only on the Class LOC Metrics due to it being, in my opinion, the most useful for easily identifying problems.

Comment Lines Of Code:

Comparing the number of comment lines to the average comment lines could point out which classes incur in code smells for commenting unnecessarily.

It is to note that disparities in numbers of comment lines may not always reveal comment amount code smells, such can be inferred by comparing CLOC to LOC

Javadoc Lines Of Code:

As with comment lines, use of Javadoc LOCs can identify code smells related to comments. Taking both into account at once can easily direct to classes with such code smells.

Lines Of Code:

By subtracting the number of CLOCs and JLOCs to the total number of LOCs we can get the classes' effective lines and compare them to the average to understand which classes infer in codes smells like the Large Class code smell.

Data Analysis:

All classes	CLOC (Comment Lines of code)	JLOC (Javadoc lines of code)	LOC (Lines of code)
Total	20139.0	12852.0	142843.0
Average	10.666843220338983	6.807203389830509	75.65836864406779

1888 Classes in total.

This table shows the CLOCs, JLOCs and LOCs in total and in average for the collection of all classes. Taking a moment to analyse the data, we can deduct that:

Percentage of CLOC that are JDOC = (12852.0 * 100) / 20139.0 = approx. 63,816%

Number of effective lines = 142843.0 – 20139.0 = 122 704 lines

Percentage of effective lines = (122 704 * 100) / 142843.0 = approx. 85,901%

Percentage of ineffective lines = 100% - 85,901% = 14,098%

In group, we have concluded that such an amount is decent, the comments are very useful, and this number of comments is likely to be helpful.

However, we can identify classes where these amounts can exceed the average largely, being more of a hindrance rather than helpful. Therefore, to understand where the code smells are, we should look at the classes where comments and lines exceed the average by a large margin.

Gonçalo Vicêncio, nº 57944

References: [1] - https://confluence.atlassian.com/fisheye/about-the-lines-of-code-metric-960155778.html; 6/12/2021 at 17:18

COMPLEXITY METRICS

There are many different ways of measuring the complexity of a project. We can analyse the complexity through more specific ways like per method or class, or more general ways like per package, module or the project itself.

Method metrics

Cognitive complexity (CogC) – It can be described as a measure of how difficult a portion of code is to read and intuitively understand.

Cyclomatic Complexity (v(G)) – It's **a count of the linearly independent paths through source code. In other words, it can be described as the number of decisions a given block of code needs to make. It can also serve as a way to measure how difficult a certain portion of code is to be tested.**

Essential Cyclomatic Complexity (ev(G)) – It tells how much complexity is left once we have removed the well-structured complexity.

Class Metrics

Maximum Operation Complexity (OCmax) – Shows the maximum complexity present in a certain class.

Average Operation Complexity (OCavg) – Determines the average complexity of operations in a certain class.

Weighted Method Complexity (WMC) – Measures the complexity of a certain class. Its value can be determined by the cyclomatic complexities of its methods.

Package, Module and Project Metrics

As the name suggests, these metrics are related to packages, modules or the project itself respectively. All of these 3 types of metrics use the same type of measurements:

Average Cyclomatic Complexity (v(G)avg) – It describes the average complexity of a certain file by analysing the complexity of its functions. It can be determined by the sum of the cyclomatic complexities of all function definitions divided by the number of function definitions in the file.

Total Cyclomatic Complexity (v(G)tot) – It describes the total cyclomatic complexity of a certain package, module or project.

JabRef's overall analysis

When evaluating the Cyclomatic Complexity of a certain project, it's common to use the following range values:

v(G)avg below 4 – Good complexity

v(G)avg below 5 and 7 – Medium complexity

v(G)avg below 8 and 10 – High complexity

v(G)avg above 10 – Extreme complexity

By analysing the overall complexity of the JabRef project, we can see that the average Cyclomatic Complexity is about 1,79, which means that JabRef has a very good complexity overall.

RELATION BETWEEN COMPLEXITY AND CODE SMELLS

Although JabRef has a pretty good overall complexity throughout the whole project, there are some exceptions. For example, some methods and classes have extremely high complexity values. Here are some cases:

In method metrics:

formatName(Author,String,Warn) : present in jabref -> src -> main -> java -> org.jabref -> logic -> bst -> BibTexNameFormatter.java, which has a cognitive complexity (CogC) of 154.0, when the usual maximum value for cognitive complexity should be about 15.

In class metrics:

RTFChars.java : present in jabref -> src -> main -> java -> org.jabref -> logic -> bst, which has an average operation complexity (OCavg) of 25.0, maximum operation complexity (OCmax) of 46 and Weighted method complexity (WMC) of 75, which are all very high values of complexity.

Possible code smell relation:

After analysing some of these examples from both methods and classes, the main conclusion is that most of these are all related to a common code smell, in this case, the “Long method” code smell. This can be explained by the fact that usually, the longer a method is, the higher is the probability that this method is responsible for a great variety of functionalities, therefore increasing its level of complexity.