

Code Smells Identified

Shotgun Surgery

In this package, the class `/jabref/src/main/java/org/jabref/migrations/PreferencesMigrations.java` has multiple strings with the same value. If there is a change of one string, it is inconvenient to search all the strings with the same value and change.

Possible Solution:

If possible, create constants, enumerators to store all the strings with the same value and/or functions to adapt the current solution.

Examples: Image 1 and 2.

Feature Envy

In the previous class has functions that focus on manipulating data of other classe rather than itself.

Possible Solution:

Reduce the amount of method calls by moving the current methods implementations pr creating new methods inside each class.

Examples:

```
static void restoreVariablesForBackwardCompatibility(JabRefPreferences preferences) {
    List<String> oldColumnNames = preferences.getStringList(JabRefPreferences.COLUMN_NAMES);
    List<String> fieldColumnNames = oldColumnNames.stream()
        .filter(columnName -> columnName.startsWith("field:") || columnName.startsWith("special:"))
        .map(columnName -> {
            if (columnName.startsWith("field:")) {
                return columnName.substring(6);
            } else { // special
                return columnName.substring(8);
            }
        }).collect(Collectors.toList());

    if (!fieldColumnNames.isEmpty()) {
        preferences.putStringList(3 "columnNames", fieldColumnNames);
    }
}
```

```
static void changeColumnVariableNamesFor51(JabRefPreferences preferences) {
    // The variable names have to be hardcoded, because they have changed between 5.0 and 5.1
    List<String> oldColumnNames = preferences.getStringList( key: 1 "columnNames");
    List<String> columnNames = preferences.getStringList(JabRefPreferences.COLUMN_NAMES);
    if (!oldColumnNames.isEmpty() && columnNames.isEmpty()) {
        preferences.putStringList(JabRefPreferences.COLUMN_NAMES, preferences.getStringList( key: 2 "columnNames"));
        preferences.putStringList(JabRefPreferences.COLUMN_WIDTHS, preferences.getStringList( key: "columnWidths"));
        preferences.putStringList(JabRefPreferences.COLUMN_SORT_TYPES, preferences.getStringList( key: "columnSortTypes"));
        preferences.putStringList(JabRefPreferences.COLUMN_SORT_ORDER, preferences.getStringList( key: "columnSortOrder"));
    }
}
```

Innapropriate Intimacy

In the class

/jabref/src/main/java/org/jabref/migrations/MergeReviewIntoCommentMigration.java has public methods that calls and has access to private parts, such as variables and functions, of other classes.

Possible Solution:

Create new private functions that gets all the variables needed to the current method.

Example:

```
public static boolean needsMigration(ParserResult parserResult) {  
    return parserResult.getDatabase().getEntries().stream()  
        .anyMatch(bibEntry -> bibEntry.getField(StandardField.REVIEW).isPresent());  
}
```

Cognitive Complexity

```
/**  
 * Determine whether there starts a special char at pos (e.g., oe, AE). Return it as string.  
 * If nothing found, return Optional.empty()  
 */  
* <p>  
* Also used by BibtexPurify  
*  
* @param c the current "String"  
* @param pos the position  
* @return the special LaTeX character or null  
*/  
public static Optional<String> findSpecialChar(char[] c, int pos) {  
    if ((pos + 1) < c.length) {  
        if ((c[pos] == 'o') && (c[pos + 1] == 'e')) {  
            return Optional.of("oe");  
        }  
        if ((c[pos] == 'O') && (c[pos + 1] == 'E')) {  
            return Optional.of("OE");  
        }  
        if ((c[pos] == 'a') && (c[pos + 1] == 'e')) {  
            return Optional.of("ae");  
        }  
        if ((c[pos] == 'A') && (c[pos + 1] == 'E')) {  
            return Optional.of("AE");  
        }  
        if ((c[pos] == 's') && (c[pos + 1] == 's')) {  
            return Optional.of("ss");  
        }  
        if ((c[pos] == 'A') && (c[pos + 1] == 'A')) {  
            return Optional.of("AA");  
        }  
        if ((c[pos] == 'a') && (c[pos + 1] == 'a')) {  
            return Optional.of("aa");  
        }  
    }  
    if ("ijoOIL".indexOf(c[pos]) >= 0) {  
        return Optional.of(String.valueOf(c[pos]));  
    }  
}
```

```

    }
    return Optional.empty();
}

```

Located in jabref/src/main/java/org/jabref/logic/bst/BibtexCaseChanger.java

Problem: The cognitive complexity (Measurement of how hard to understand the control flow of a method is) present in this method is too high (value of 23 passing the default value of 15)

Refactor proposal:

Change type of c from “char[]” to “String[]”, which would represent all two character segments made from the “current string”, and use the String method “contains()” or “equalsIgnoreCase()” to check if each two character segment follows one of the mentioned patterns. If so, return that segment, otherwise return null if we reached the end of our string.

Excess Comments

```

// Comments from bibtex.web:

// sp_ptr := str_start[pop_lit1];
int i = 0;

// sp_end := str_start[pop_lit1+1];
int n = s.length();

// sp_brace_level := 0;
int braceLevel = 0;

// while (sp_ptr < sp_end) do begin
while (i < n) {
    // incr(sp_ptr);
    i++;
    // if (str_pool[sp_ptr-1] = left_brace) then
    // begin
    if (c[i - 1] == '{') {
        // incr(sp_brace_level);
        braceLevel++;
        // if ((sp_brace_level = 1) and (sp_ptr < sp_end)) then
        if ((braceLevel == 1) && (i < n)) {
            // if (str_pool[sp_ptr] = backslash) then
            // begin
            if (c[i] == '\\') {
                // incr(sp_ptr); {skip over the |backslash|}
                i++; // skip over backslash
                // while ((sp_ptr < sp_end) and (sp_brace_level
                // > 0)) do begin
                while ((i < n) && (braceLevel > 0)) {
                    // if (str_pool[sp_ptr] = right_brace) then
                    if (c[i] == '}') {
                        // decr(sp_brace_level)
                        braceLevel--;
                    } else if (c[i] == '{') {
                        // incr(sp_brace_level);
                        braceLevel++;
                    }
                    // incr(sp_ptr);
                    i++;
                    // end;
                }
                // incr(num_text_chars);
                result++;
                // end;
            }
            // end
        }
    }

    // else if (str_pool[sp_ptr-1] = right_brace) then
    // begin
} else if (c[i - 1] == '}') {
    // if (sp_brace_level > 0) then

```

```

    if (braceLevel > 0) {
        // decr(sp_brace_level);
        braceLevel--;
        // end
    }
} else { // else
    // incr(num_text_chars);
    result++;
}
}
stack.push(result);

```

Located in jabref/src/main/java/org/jabref/logic/bst/VM.java

Problem: Multiple sections of code commented out, affecting the code's readability and understanding

Refactor proposal: Remove them from the source code and replace with comments that explain what logic is happening in this section. An overall comment explaining the first ever if and else should be enough to understand the code.

Long Parameter List

```

public CitationKeyPatternPreferences(boolean shouldAvoidOverwriteCiteKey,
                                     boolean shouldWarnBeforeOverwriteCiteKey,
                                     boolean shouldGenerateCiteKeysBeforeSaving,
                                     KeySuffix keySuffix,
                                     String keyPatternRegex,
                                     String keyPatternReplacement,
                                     String unwantedCharacters,
                                     GlobalCitationKeyPattern keyPattern,
                                     Character keywordDelimiter) {

    this.shouldAvoidOverwriteCiteKey = shouldAvoidOverwriteCiteKey;
    this.shouldWarnBeforeOverwriteCiteKey = shouldWarnBeforeOverwriteCiteKey;
    this.shouldGenerateCiteKeysBeforeSaving = shouldGenerateCiteKeysBeforeSaving;
    this.keySuffix = keySuffix;
    this.keyPatternRegex = keyPatternRegex;
    this.keyPatternReplacement = keyPatternReplacement;
    this.unwantedCharacters = unwantedCharacters;
    this.keyPattern = keyPattern;
    this.keywordDelimiter = keywordDelimiter;
}

```

Located in

jabref/src/main/java/org/jabref/logic/citationkeypattern/CitationKeyPatternPreferences.java

Problem: Constructor with too many parameters (value of 9 parameters surpassing the default 7 allowed)

Refactor Proposal: Make a data class to either include all the booleans or all the Strings (or maybe both) to reduce the amount of global variables and parameters required for this method.

3 parameters for keySuffix, GlobalCitationKeyPattern and Character;

2 (5 parameters total with two data classes containing 3 variables each) if using a data class for all booleans and another for the Strings;

4 (7 parameters total with a data class containing 3 variables) if using only for booleans or strings;

1 (4 parameters total with a data class containing 6 variables) if using the data class for all booleans and strings.)

Long Class

[_java/org/jabref/gui/preview/CopyCitationAction.java: line 94-174](#)

Problem: This class is doing more than it should. There should be an uniform way of formatting citations to the various formats.

Refactoring Proposal: Extract the citation process methods on this class and place it in a suiting class the *citationstyle* package.

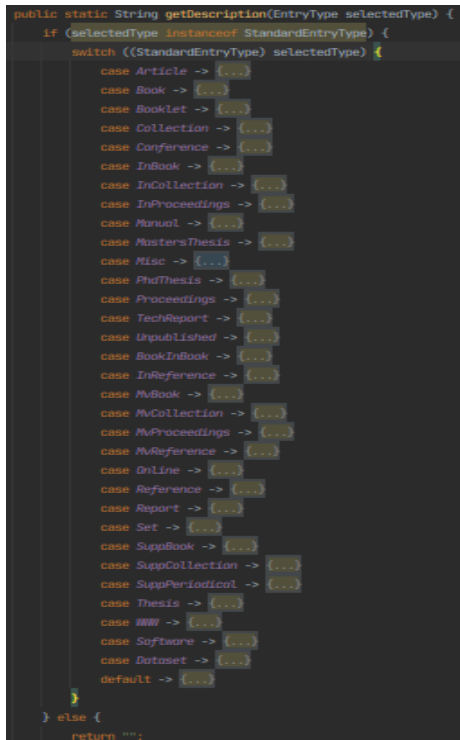
```
/**
 * Inserts each citation into a HTML body and copies it to the clipboard
 */
protected static ClipboardContent processHtml(List<String> citations) {
    String result = "<!DOCTYPE html>" + OS.NEWLINE +
        "<html>" + OS.NEWLINE +
        "    <head>" + OS.NEWLINE +
        "        <meta charset=\"utf-8\">" + OS.NEWLINE +
        "    </head>" + OS.NEWLINE +
        "    <body>" + OS.NEWLINE + OS.NEWLINE;

    result += String.join(CitationStyleOutputFormat.HTML.getLineSeparator(), citations);
    result += OS.NEWLINE +
        "    </body>" + OS.NEWLINE +
        "</html>" + OS.NEWLINE;

    ClipboardContent content = new ClipboardContent();
    content.putString(result);
    content.putHtml(result);
    return content;
}
```

Switch Statement

java/org/jabref/gui/EntryTypeView.java: line 210-316

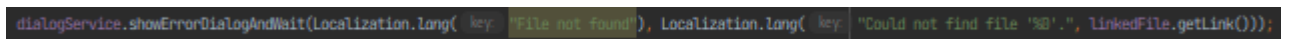
A screenshot of a code editor showing a Java method `getDescription` that takes an `EntryType` parameter. The method first checks if the parameter is an instance of `StandardEntryType`. If so, it enters a `switch` statement that iterates through 31 different `StandardEntryType` enum values, each followed by a `return` statement and a comment indicating the description is not implemented. The values include `Article`, `Book`, `Booklet`, `Collection`, `Conference`, `InBook`, `InCollection`, `InProceedings`, `Manual`, `MastersThesis`, `Misc`, `PhdThesis`, `Proceedings`, `TechReport`, `Unpublished`, `BookInBook`, `InReference`, `MyBook`, `MyCollection`, `MyProceedings`, `MyReference`, `Online`, `Reference`, `Report`, `Set`, `SuppBook`, `SuppCollection`, `SuppPeriodical`, `Thesis`, `WWW`, `Software`, and `Dataset`. A `default` case is also present. If the parameter is not an instance of `StandardEntryType`, the method returns an empty string.

Problem: This switch statement iterates through all the values in the Enum. If we wanted to add a new type of entry, or remove one, we would need to refactor 2 classes.

Refactoring Proposal: Extract this method and add *description* as an instance variable in the `StandardEntryType` Enum.

Lack of constants

java/org/jabref/gui/fieldeditors/LinkedFileViewModel.java: lines 208, 226, 260, 312

A screenshot of a Java code snippet showing two calls to `dialogService.showErrorDialogAndWait`. Both calls use `Localization.lang` with a `key` parameter. The first key is `"File not found"` and the second is `"Could not find file '%0'."`, where `%0` is a placeholder for a file link obtained from `LinkedFile.getLink()`.

Problem: The String “File not found” and “Could not find file ‘%0’.” can be found repeated throughout this class. This increases chances of creating inconsistencies.

Refactoring Proposal: Creating class constants for these Strings.

Long Parameter List

java/org/jabref/preferences/GuiPreferences.java: line 28-35

```
public GuiPreferences(double positionX,  
                      double positionY,  
                      double sizeX,  
                      double sizeY,  
                      boolean windowMaximised,  
                      boolean shouldOpenLastEdited,  
                      List<String> lastFilesOpened,  
                      Path lastFocusedFile, double sidePaneWidth) {
```

Problem: The constructor of this class has too many parameters (9 parameters, which is greater than the 7 authorized).

Refactoring Proposal: Create a new class which deals with the *positionX*, *positionY*, *sizeX* and *sizeY* and this code smell should be solved.

Inappropriate Naming

java/org/jabref/preferences/JabRefPreferences.java: line 146 and 413

```
public static final String LANGUAGE = "language";  
private Language language;
```

Problem: Names should not differ only by capitalization.

Refactoring Proposal: change *private Language language* to *private Language currLanguage*.

Cognitive Complexity

java/org/jabref/preferences/JabRefPreferences.java: lines 747-789

```
private static Optional<String> getNextUnit(Reader data) throws IOException {
    // character last read
    // -1 if end of stream
    // initialization necessary, because of Java compiler
    int c = -1;

    // last character was escape symbol
    boolean escape = false;

    // true if a STRINGLIST_DELIMITER is found
    boolean done = false;

    StringBuilder res = new StringBuilder();
    while (!done && ((c = data.read()) != -1)) {
        if (c == '\\') {
            if (escape) {
                escape = false;
                res.append('\\');
            } else {
                escape = true;
            }
        } else {
            if (c == STRINGLIST_DELIMITER) {
                if (escape) {
                    res.append(STRINGLIST_DELIMITER);
                } else {
                    done = true;
                }
            } else {
                res.append((char) c);
            }
            escape = false;
        }
    }
    if (res.length() > 0) {
        return Optional.of(res.toString());
    } else if (c == -1) {
        // end of stream
        return Optional.empty();
    } else {
        return Optional.of("");
    }
}
```

Problem: Method is too complex thus hard to maintain.

Refactoring Proposal: More comments should be added to better explain this method or it should be broken into smaller methods.

Long Method

```
public static String addDotIfAbbreviation(String name) {  
    1 if ((name == null) 2 || name.isEmpty()) {  
        return name;  
    }  
    // If only one character (uppercase letter), add a dot and return immediately:  
    3 if ((name.length() == 1) 4 && Character.isLetter(name.charAt(0)) &&  
        Character.isUpperCase(name.charAt(0))) {  
        return name + ".";  
    }  
  
    StringBuilder sb = new StringBuilder();  
    char lastChar = name.charAt(0);  
    5 for (int i = 0; i < name.length(); i++) {  
        6 if (i > 0) {  
            lastChar = name.charAt(i - 1);  
        }  
        char currentChar = name.charAt(i);  
        sb.append(currentChar);  
  
        7 if (currentChar == '.') {  
            // A.A. -> A. A.  
            8 if (((i + 1) < name.length()) 9 && Character.isUpperCase(name.charAt(i + 1))) {  
                sb.append(' ');  
            }  
        }  
    }  
}
```

Located at `src/java/org/jabref/model/author.java` line 39 the method `addDotIfAbbreviation`

Problem: it's 82 lines long and has a cognitive complexity of 41 (way above the default value of 15)

Refactor Proposal:

Split into several methods to reduce complexity and length. The inner code on the outer for (lines 52 to 117) could be moved to a new method that takes the input chars (`currentChar` and `lastChar`) and the String Builder. The same could then be done for the inner for loop (lines 99 to 116)

Long Class

Located at `src/java/org/jabref/model/entry/BibEntry.java`

Problem: Class spans over too many lines, over 1000, and provides too many methods

Refactor proposal:

Create a new class `BibEntryFields` that stores the fields for the `BibEntry` and provides all related functionality, allowing to remove responsibility and methods from `BibEntry.java`

```
947
948  public void setDate(Date date) {...}
953
954  public Optional<Month> getMonth() { return getFieldOrAlias(StandardField.MONTH).flatMap(Month::parse); }
957
958  public OptionalBinding<String> getFieldBinding(Field field) {...}
964
965  public OptionalBinding<String> getCiteKeyBinding() { return getFieldBinding(InternalField.KEY_FIELD); }
968
969  public Optional<FieldChange> addFile(LinkedFile file) {...}
974
975  public Optional<FieldChange> addFile(int index, LinkedFile file) {...}
980
981  public ObservableMap<Field, String> getFieldsObservable() { return fields; }
984
985  /** Returns a list of observables that represent the data of the entry. */
988  public Observable[] getObservables() { return new Observable[] {fields, type}; }
991
992  public void addLinkedFile(BibEntry entry, LinkedFile linkedFile, LinkedFile newLinkedFile, List<LinkedF
1010
1011  }
1012
```

Duplicated Code

```
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if ((o == null) || (getClass() != o.getClass())) {
        return false;
    }

    MatcherSet that = (MatcherSet) o;

    return Objects.equals(matchers, that.matchers);
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if ((o == null) || (getClass() != o.getClass())) {
        return false;
    }
    BibEntry entry = (BibEntry) o;
    return Objects.equals(type.getValue(), entry.type.getValue())
        && Objects.equals(fields, entry.fields)
        && Objects.equals(commentsBeforeEntry, entry.commentsBeforeEntry);
}
```

Location: src/main/java/org/jabref/model/search/matchers/MatcherSet.java,
src/java/org/jabref/model/entry/BibEntry.java and several other places

Problem: This segment of the equals method has similar duplicates many times across the code. This could be an issue if another generic case needs to be considered which would alter the result or allow a faster matching.

Refactor Proposal:

Extract the common part of the method to a static method and pass the non common part of the verification as a lambda

Design Patterns Identified

These are some of the design patterns that can be identified throughout the project.

Factory

1. src/main/java/org/iabref/model/entry/field/FieldFactory.java

This class hides the logic of creating a Field or set of Fields by providing a single access point for their creation.

```
public static Field parseField(String fieldName) {  
    return OptionalUtil.<Field>orElse(OptionalUtil.<~>orElse(OptionalUtil.<~>orElse(  
        InternalField.fromName(fieldName),  
        StandardField.fromName(fieldName),  
        SpecialField.fromName(fieldName),  
        IEEEField.fromName(fieldName))  
        .orElse(new UnknownField(fieldName));  
}
```

Figure 1: Creating a single field from its name

```
public static Set<Field> parseFieldList(String fieldNames) {  
    return Arrays.stream(fieldNames.split(FieldFactory.DELIMITER))  
        .filter(StringUtil::isNotBlank)  
        .map(FieldFactory::parseField)  
        .collect(Collectors.toCollection(LinkedHashSet::new));  
}
```

Figure 2: Creating multiple fields from their name

2. src/main/java/org/iabref/model/entry/types/EntryTypeFactory.java

This class hides the logic of creating an EntryType by allowing the user to create one from a single String.

```

public static EntryType parse(String typeName) {

    List<EntryType> types = new ArrayList<>(Arrays.<~>asList(StandardEntryType.values()));
    types.addAll(Arrays.<~>asList(IEEETranEntryType.values()));
    types.addAll(Arrays.<~>asList(SystematicLiteratureReviewStudyEntryType.values()));

    return types.stream().filter(type -> type.getName().equals(typeName.toLowerCase(Locale.ENGLISH)))
        .findFirst().orElse(new UnknownEntryType(typeName));
}

```

Figure 3: Create a single EntryType from its type name

3. [java/org/jabref/gui/actions/ActionFactory.java](#)

This class allows the user to create various JavaFX components associated with actions without having to worry about the actual process of creation or association.

```

public MenuItem configureMenuItem(Action action, Command command, MenuItem menuItem) {...}

public MenuItem createMenuItem(Action action, Command command) {...}

public CheckMenuItem createCheckMenuItem(Action action, Command command, boolean selected) {...}

public Menu createMenu(Action action) {...}

public Menu createSubMenu(Action action, MenuItem... children) {...}

public Button createIconButton(Action action, Command command) {...}

public ButtonBase configureIconButton(Action action, Command command, ButtonBase button) {...}

```

Figure 4: Methods for creating different JavaFX components associated with actions

Strategy

1. [src/main/java/org/jabref/gui/autocompleter](#)

The *AutoCompletionStrategy* provides a template for solving an auto completion problem.

In this package we can also find problem solvers, like *AppendWordsStrategy*, *AppendPersonNamesStrategy* and *ReplaceStrategy*.

```
public interface AutoCompletionStrategy {
    AutoCompletionInput analyze(String input);
}
```

Figure 5: Template for a problem solver algorithm

```
public class AppendWordsStrategy implements AutoCompletionStrategy {

    protected String getDelimiter() { return " "; }

    @Override
    public AutoCompletionInput analyze(String input) {
        return determinePrefixAndReturnRemainder(input, getDelimiter());
    }

    private AutoCompletionInput determinePrefixAndReturnRemainder(String input, String delimiter) {
        int index = input.toLowerCase(Locale.ROOT).lastIndexOf(delimiter);
        if (index >= 0) {
            String prefix = input.substring(0, index + delimiter.length());
            String rest = input.substring(index + delimiter.length());
            return new AutoCompletionInput(prefix, rest);
        } else {
            return new AutoCompletionInput("", input);
        }
    }
}
```

Figure 6: AppendWordsStrategy, an algorithm to solve the problem

Proxy

1. [java/org/jabref/logic/bst/PurifyFunction.java](http://java.org/jabref/logic/bst/PurifyFunction.java)

This class only does error checking and the actual work is done by *BibtexPurify*. This is a **protection proxy**.

```

public void execute(BstEntry context) {
    Stack<Object> stack = vm.getStack();

    if (stack.isEmpty()) {
        throw new VMException("Not enough operands on stack for operation purify$");
    }
    Object o1 = stack.pop();

    if (!(o1 instanceof String)) {
        vm.warn( string: "A string is needed for purify$");
        stack.push( item: "");
        return;
    }

    stack.push(BibtexPurify.purify((String) o1, vm));
}

```

Figure 7: Protecting BibtexPurify from receiving invalid arguments

2. [java/org/jabref/logic/citationstyle/CitationStyleGenerator.java](http://java.org/jabref/logic/citationstyle/CitationStyleGenerator.java)

This class provides a protection proxy to a CLS_ADAPTER by protecting the user from uncaught exceptions.

```

public static List<String> generateCitations(List<BibEntry> bibEntries, String style, CitationStyleOutputFormat outputFormat) {
    try {
        return CSL_ADAPTER.makeBibliography(bibEntries, style, outputFormat);
    } catch (IllegalArgumentException ignored) {
        LOGGER.error("Could not generate BibEntry citation. The CSL engine could not create a preview for your item.", ignored);
        return Collections.singletonList(Localization.lang( key: "Cannot generate preview based on selected citation style."));
    } catch (IOException | ArrayIndexOutOfBoundsException e) {
        LOGGER.error("Could not generate BibEntry citation", e);
        return Collections.singletonList(Localization.lang( key: "Cannot generate preview based on selected citation style."));
    } catch (TokenMgrException e) {
        LOGGER.error("Bad character inside BibEntry", e);
        // sadly one cannot easily retrieve the bad char from the TokenMgrError
        return Collections.singletonList(Localization.lang( key: "Cannot generate preview based on selected citation style." ) +
            outputFormat.getLineSeparator() +
            Localization.lang( key: "Bad character inside entry" ) +
            outputFormat.getLineSeparator() +
            e.getLocalizedMessage());
    }
}

```

Figure 8: Protecting the user from un-handled exceptions raised by the CLS

Singleton

1. [java/org/jabref/logic/logging/LogMessages.java](#)

```
public class LogMessages {

    private static LogMessages instance = new LogMessages();

    private final ObservableList<LogEvent>
        messages = FXCollections.observableArrayList();

    private LogMessages() {}

    public static LogMessages getInstance() { return instance; }

    public ObservableList<LogEvent> getMessages() { return FXCollections.unmodifiableObservableList(messages); }

    public void add(LogEvent event) { messages.add(event); }

    public void clear() { messages.clear(); }

}
```

Figure 9: *LogMessages.java*

This class provides a single access point, through the static method *getInstance()*, for the instance where error messages are stored.

2. [java/org/jabref/preferences/JabRefPreferences.java](#)

This class provides a global access point to a single instance, the constructor is private to ensure there is only one instance is created.

```
private static JabRefPreferences singleton;
```

Figure 10: *The only instance of JabRefPreferences*

```
private JabRefPreferences() {...}

/** @return Instance of JaRefPreferences ...*/
@Deprecated
public static JabRefPreferences getInstance() {
    if (JabRefPreferences.singleton == null) {
        JabRefPreferences.singleton = new JabRefPreferences();
    }
    return JabRefPreferences.singleton;
}
```

Figure 11: *Ensuring there only exists one instance*

3. [java/org/jabref/gui/externalfiletype/ExternalFileTypes.java](#)

This class provides access to the available external file types.

```
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;
}
```

Figure 12: *ExternalFileTypes.java*

Observer

1. [java/org/jabref/model/model/database](#)

The BibDatabase allows listeners to subscribe and unsubscribe from the eventBus, which acts like the Subject. In the same package KeyChangeListener represents the listener which gets notified of the events from the BibDatabase's event bus and

```
Registers an listener object (subscriber) to the internal event bus. The following events are posted: -
EntriesAddedEvent - EntryChangedEvent - EntriesRemovedEvent
Params: listener – listener (subscriber) to add

public void registerListener(Object listener) {
    this.eventBus.register(listener);
}
```

Figure 13: *BibDataBase.java* registering listeners in the event bus handles them accordingly.

```

public class KeyChangeListener {

    private final BibDatabase database;

    public KeyChangeListener(BibDatabase database) { this.database = database; }

    @Subscribe
    public void listen(FieldChangedEvent event) {...}

    @Subscribe
    public void listen(EntriesRemovedEvent event) {...}
}

```

Figure 15: *KeyChangeListener.java* listening for the events in the *BibDatabase*

```

Posts an event to all registered subscribers. This method will return successfully after the event has been
posted to all subscribers, and regardless of any exceptions thrown by subscribers.

If no subscribers have been subscribed for event's class, and event is not already a DeadEvent, it will be
wrapped in a DeadEvent and reposted.

Params: event – event to post.

public void post(Object event) {
    Iterator<Subscriber> eventSubscribers = subscribers.getSubscribers(event);
    if (eventSubscribers.hasNext()) {
        dispatcher.dispatch(event, eventSubscribers);
    } else if (!(event instanceof DeadEvent)) {
        // the event had no subscribers and was not itself a DeadEvent
        post(new DeadEvent( source: this, event));
    }
}
}

```

Figure 14: *EventBus.java* forwarding the events to all the event subscribers

2. src/main/java/org/jabref/model/util

The *FileUpdateListener* and *FileUpdateMonitor* match the observer and subject, respectively.

We can add and remove new observers by calling *addListenerForFile(...)* and *removeListener(...)* in the subject. The observer gets updated through the *fileUpdated()* method.

```

public interface FileUpdateMonitor {

    /**
     * Add a new file to monitor.
     *
     * @param file The file to monitor.
     * @throws IOException if the file does not exist.
     */
    void addListenerForFile(Path file, FileUpdateListener listener) throws IOException;

    /**
     * Removes a listener from the monitor.
     *
     * @param path The path to remove.
     */
    void removeListener(Path path, FileUpdateListener listener);
}

```

Figure 16: FileUpdateMonitor.java registering listeners

```

public interface FileUpdateListener {

    /**
     * The file has been updated. A new call will not result until the file has been modified again.
     */
    void fileUpdated();
}

```

Figure 17: FileUpdateListener

Builder

1. [java/org/jabref/model/entry/BibEntryTypeBuilder.java](#)

This class lets the user create a `BibEntryType` step-by-step, with all the required details.

```
public class BibEntryTypeBuilder {

    private EntryType type = StandardEntryType.Misc;
    private Set<BibField> fields = new LinkedHashSet<>();
    private Set<OrFields> requiredFields = new LinkedHashSet<>();

    public BibEntryTypeBuilder withType(EntryType type) {...}

    public BibEntryTypeBuilder withImportantFields(Set<BibField> newFields) {...}

    public BibEntryTypeBuilder withImportantFields(Collection<Field> newFields) {...}

    public BibEntryTypeBuilder withImportantFields(Field... newFields) {...}

    public BibEntryTypeBuilder withDetailFields(Collection<Field> newFields) {...}

    public BibEntryTypeBuilder withDetailFields(Field... fields) { return withDetailFields(Arrays.asList(fields)); }

    public BibEntryTypeBuilder withRequiredFields(Set<OrFields> requiredFields) {...}

    public BibEntryTypeBuilder withRequiredFields(Field... requiredFields) {...}

    public BibEntryTypeBuilder withRequiredFields(OrFields first, Field... requiredFields) {...}

    public BibEntryTypeBuilder withRequiredFields(List<OrFields> first, Field... requiredFields) {...}

    public BibEntryType build() {...}
}
```

Figure 18: *BibEntryTypeBuilder.java*

2. [java/org/jabref/preferences/PreviewPreferences.java](#)

This class allows the user to build a *PreviewPreferences* class step-by-step.

```

public static class Builder {

    private boolean showPreviewAsExtraTab;
    private List<PreviewLayout> previewCycle;
    private int previewCyclePosition;
    private Number previewPanelDividerPosition;
    private String previewStyle;
    private final String previewStyleDefault;

    public Builder(PreviewPreferences previewPreferences) {...}

    public Builder withShowAsExtraTab(boolean showAsExtraTab) {...}

    public Builder withPreviewCycle(List<PreviewLayout> previewCycle) {...}

    public Builder withPreviewCyclePosition(int position) {...}

    public Builder withPreviewPanelDividerPosition(Number previewPanelDividerPosition) {...}

    public Builder withPreviewStyle(String previewStyle) {...}

    public PreviewPreferences build() {
        return new PreviewPreferences(previewCycle, previewCyclePosition, previewPanelDividerPosi
    }
}

```

Figure 19: PreviewPreferences.java

Facade

1. [java/org/jabref/preferences/JabRefPreferences.java](#)

Every preference is stored inside *JabRefPreferences* which provides convenient access to a particular sub-preference. It knows where to direct the client's request and how to operate.

```
@Override
public FileLinkPreferences getFileLinkPreferences() {...}

@Override
public void storeFileDirForDatabase(List<Path> dirs) { this.fileDirForDatabase = dirs; }

@Override
public LayoutFormatterPreferences getLayoutFormatterPreferences(JournalAbbreviationRepository repository) {...}

@Override
public OpenOfficePreferences getOpenOfficePreferences() {...}

@Override
public void setOpenOfficePreferences(OpenOfficePreferences openOfficePreferences) {...}

@Override
public VersionPreferences getVersionPreferences() {...}

@Override
public void storeVersionPreferences(VersionPreferences versionPreferences) {...}

@Override
public JournalAbbreviationPreferences getJournalAbbreviationPreferences() {...}

@Override
public CleanupPreferences getCleanupPreferences(JournalAbbreviationRepository abbreviationRepository) {...}
```

Figure 20: Methods for accessing different preferences

Command

1. [java/org/jabref/gui/actions/SimpleCommand.java](#)

This class is an abstraction of a command that is executed by the actions of GUI elements, such as buttons. These actions provide normalized access to the Jabref low level functionalities.

```
public class AboutAction extends SimpleCommand {

    private final AboutDialogView aboutDialogView;

    public AboutAction() { this.aboutDialogView = new AboutDialogView(); }

    @Override
    public void execute() {
        DialogService dialogService = Injector.instantiateModelOrService(DialogService.class);
        dialogService.showCustomDialog(aboutDialogView);
    }

    public AboutDialogView getAboutDialogView() { return aboutDialogView; }
}
```

Figure 21: An action

Dependency Metrics

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

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

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

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

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

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

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

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

Troubleshooting possible cases

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

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

Martin Package Metrics Report

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

The definitions were gathered from

Definitions of the Parameters

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

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

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

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

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

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

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

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

Normalized distance from Main Sequence (D) - This metric is used to measure the distance between stability and abstractness and is calculated using the formula

$D = |A + I - 1|$, where A is Abstractness and I is Instability.

This value should be as low as possible as it indicates it's close to the main sequence.

We should always avoid the two extreme situations which are:

A = 0 and I = 0, meaning that the package is extremely stable and concrete, meaning that it can't be extended.

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

Troubleshooting possible cases

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

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

Many of the packages present in the project don't satisfy the equation presented by Robert Martin ($I + A = 1$) for being an optimal package, for example the package "org.jabref.logic.preview", with values of 1 for A and 0.02 for I, representing that the package may present unnecessary dependency coupling.

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

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

Many of the packages don't present a very low value of D, meaning that they are very far from the main sequence of code due to the lack of proportionality between the value A and I.

MOOD Metrics Report

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

Attribute Hiding Factor (AHF): The obscurity (contrary of visibility) of the attributes of a class. Public attributes are visible to all classes inside their class's package and protected attributes are visible to all subclasses.

Attribute Inheritance Factor (AIF): How many of a class's fields can be inherited.

Coupling Factor (CF): Dependencies per class (includes super classes).

Method Hiding Factor (MHF): The obscurity of the methods of a class. For *public* methods it's assumed they are visible to all classes inside their class's package and for *protected* it's assumed they are visible to all subclasses.

Method Inheritance Factor (MIF): How many of the non-overridden methods of a class are available from an inherited class.

Polymorphism Factor (PF): How much polymorphism is used. This is calculated by dividing the number of overriding methods by the number of potential overrides in the subclasses.

Analysis

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

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

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

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

Trouble Spots

Unwanted Visibility: Usually for AHF we want values close to 100%, but all around the project we can see that AHF is really low which could indicate a huge problem with visibility. With high visibility comes the possibility for tampering with instance variables which could alter the flow

of the program in unexpected ways. If this does reveal to be a problem the first package we should look into is *model*, as it handles critical data and has one of the lowest AHF.

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

Complexity Metrics

MetricsReloaded Plugin

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

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

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

Cognitive Complexity

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

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

Essential Cyclomatic Complexity

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

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

Design Complexity

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

The packages Gui and Logic have the most methods with the highest Design Complexity meaning it is harder to understand at once their interconnections.

org.jabref.logic.importer.fileformat.RisImporter.importDatabase(BufferedReader) is the method with the worst design because it is long and calls many other methods, having many interconnections.

Cyclomatic Complexity

This complexity calculates how many independent paths there are in a method, thus how many tests are necessary.

Once again, Gui and Logic are the packages with the highest complexity.

org.jabref.logic.layout.format.RTFChars.transformSpecialCharacter(long) is the method with the highest complexity due to the amount of ifs there are that increment the number of independent paths

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

Chidamber Metrics

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

CBO – Coupling Between Object Classes

org.jabref.model.entry.BibEntry has the highest value of 616 for this metric which highly exceeds the reference maximum. The reason behind this is that *BibEntry* is a super class with over 1000 lines being the central object of the application. This class probably holds too much responsibility and should be split into smaller classes which handle certain aspects of a

BibEntry. The overall average for this metric of 13.33 is still way above the reference value of 1.25, which indicates excess of class responsibilities is most likely common across the project.

DIT – Depth of the Inheritance Tree

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

LCOM – Lack of Cohesion of Methods

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

NOC – Number of Children

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

RFC – Response for a class

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

WMC – Weighted Methods Per Class

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

References:

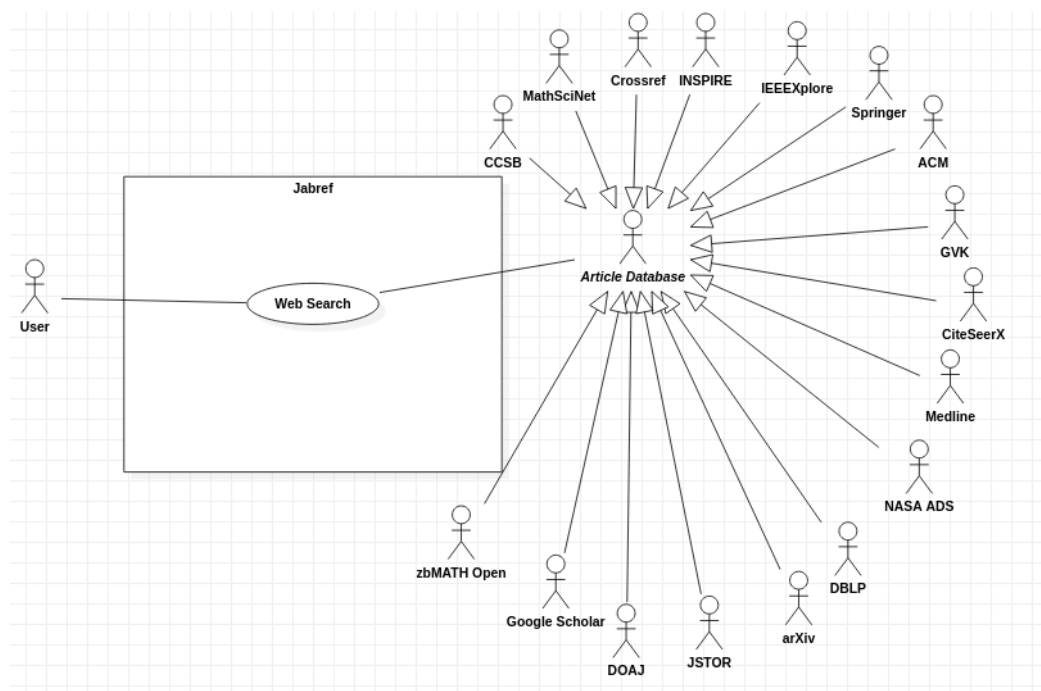
¹Laing, V. and C. Coleman. "Principal Components of Orthogonal Object-Oriented Metrics." (2001).

Use Cases

These are the use cases we created for some generic uses of the JabRef tool.

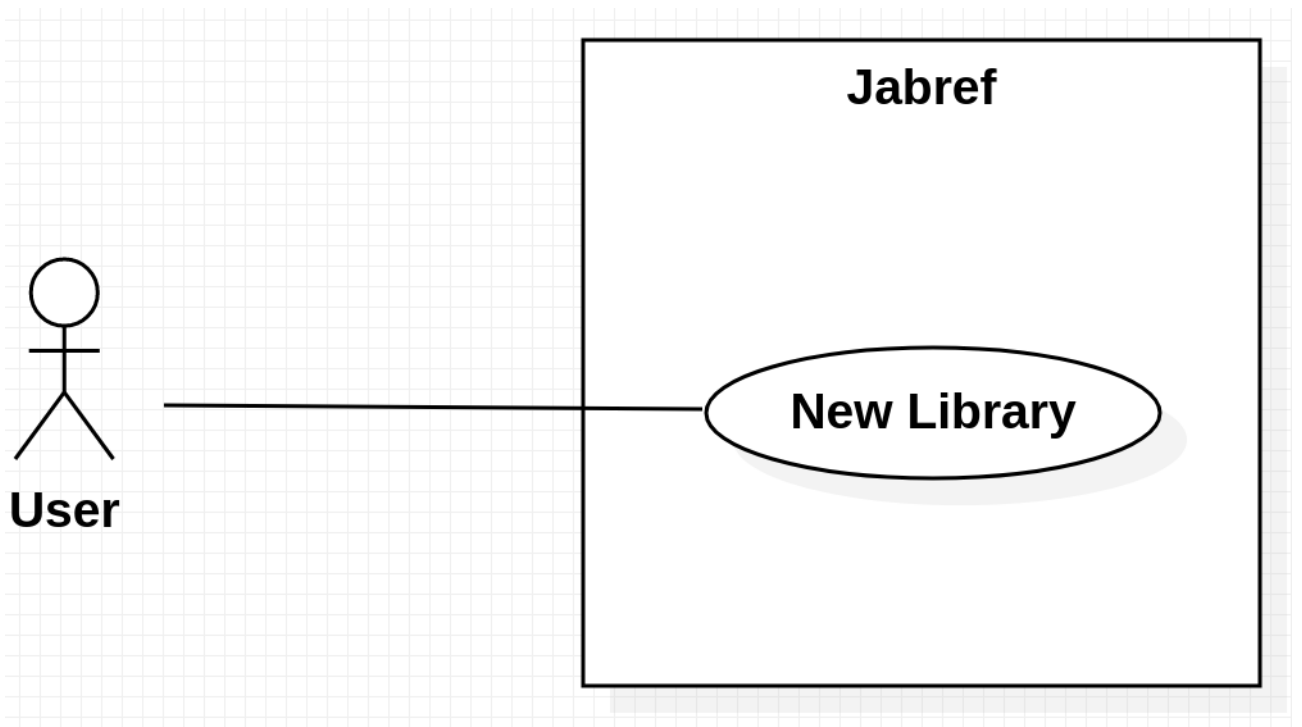
Search Article

Name	Search articles
ID	SEARCH_ARTICLES
Description	The user wants to search the web for articles on a topic.
Primary Actors	User
Secondary Actors	CCSB, MathSciNet, Crossref, Inspire, IEEEExplore, Springer, ACM, GVK, CiteSeerX, Medline, NASA ADS, DBLP, arXiv, JSTOR, DOAJ, Google Scholar, zbMATH Open
Pre-conditions	<ul style="list-style-type: none">- The application must be running- A library must be selected- There must be an internet connection
Main Flow	<ol style="list-style-type: none">1. The use case begins when the user executes the <i>Web Search</i> action.2. The use case ends when the article database replies with matching articles.
Post-conditions	The list of articles is presented to the user.



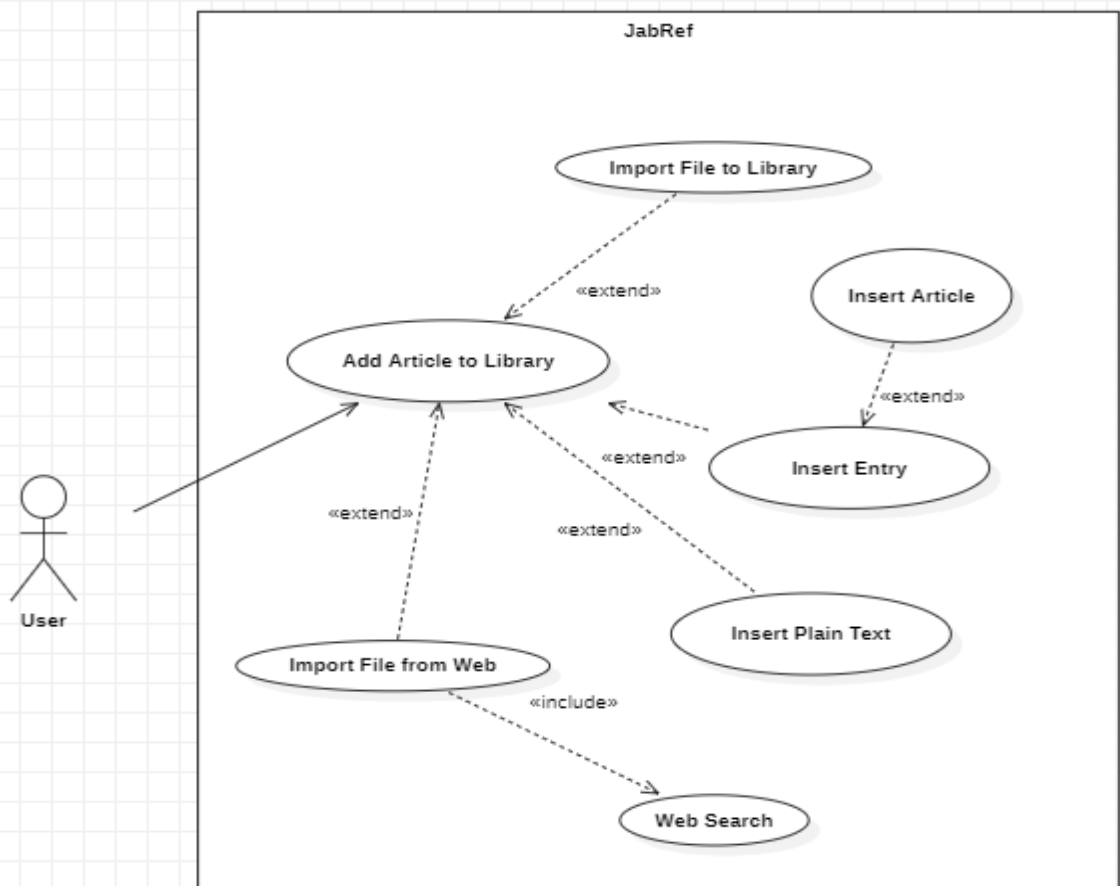
Create a Library

Name	Create a new library
ID	CREATE_LIB
Description	The user wants to create a new library to store entries.
Actors	User
Pre-conditions	There are no pre-conditions associated with this use case, besides the application to be running
Main Flow	1. The use case begins when the user executes the <i>new library</i> action. 2. The use case ends when a new database is created
Post-conditions	The library is added to the jabref frame and the user can add entries to it.



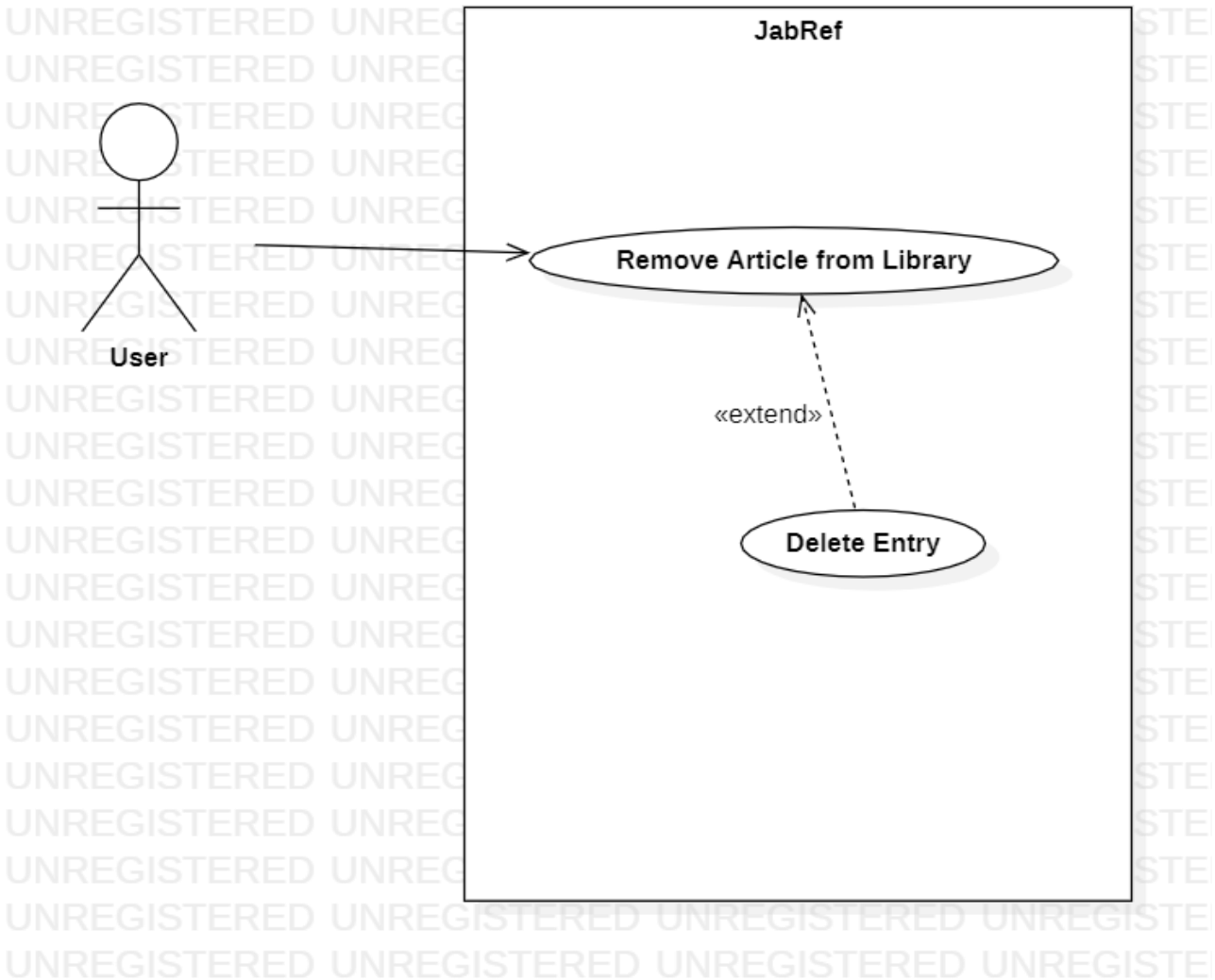
Insert an Article into a Library

Name	Insert an article into a library
ID	INSERT_ARTICLE
Description	The user wants to insert an article into a library.
Actors	User
Pre-conditions	The user must have selected a library.
Main Flow	<ol style="list-style-type: none"> 1. The use case begins when the user executes the <i>new entry</i> action. 2a. If the user wants to manually add an article, it goes to <i>insert entry</i> action. 2b. Use case ends when the user reaches the <i>insert article</i> action. 3. If the user wants to import a local file representing an article, it goes to <i>import file to Library</i> action and ends. 4. If the user wants to insert an article using text only, it goes to <i>insert plain text</i> action and ends. 5. If the user wants to import a file from the internet, it goes to <i>import file from Web</i> action then finishes when it reaches the <i>web search</i> action.
Post-conditions	The article has been created and inserted into the selected library.



Remove an Article from a Library

Name	Remove an article from a library
ID	REMOVE_ARTICLE
Description	The user wants to remove an article from a library
Actors	User
Pre-conditions	User must have selected an article in a library.
Main Flow	1. The use case begins when the user chooses an article from a library. 2. The user goes to <i>Delete Entry</i> action. 3. The use case ends when the user confirms the deletion of the selected entry or aborts the <i>delete</i> action and then finishes.
Post-conditions	The selected article has been removed from the library.



Export a Library to Tex

Name:	Export the library to Tex.
Id:	EXPORT_LIB_TO_TEX_0
Description:	The current user would want to export the current saved application file to LaTeX file format.
Actors:	User
Pre-conditions:	User must have a file with atleast 1 entry
Main flow:	<ol style="list-style-type: none">1. The use case begins when the user creates a new application file or imports a existing one.2. The user save the current file.3. The system exports the application file located in a chosen path by the user.4. The use case ends when the file is written.
Alternative flow:	None
Post-conditions:	A file with the result of the export was created in the path chosen by the user.

