NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Deliverable Phase 1

Software Engineering

Clara Sousa 58403 Paula Inês Lopes 58655 Pedro Reis 58751
Ricardo Pereira 57912 Rita Silva 57960

December 6, 2021

Contents

1 Document Description and Overall Review

It is the aim of this document to merge all the information produced by the Scrum Team throughout the 5 sprints during which the first phase of the project took place. Hence, it presents the final versions of use case diagrams, use case descriptions, design patterns and whatnot, taking into consideration their first presentation, correspondent review by co-workers and final overall review.

The management of the sprints were made early on, and nothing was delivered in the first two since the team was developing the project with a focus on user stories, instead of tasks. Such mistake occured after reading Agile literature and discussing it with employees of companies who use Agile on a daily-basis; after having spoken and debated with the professors, the team understood the real goals of the project and re-started it.

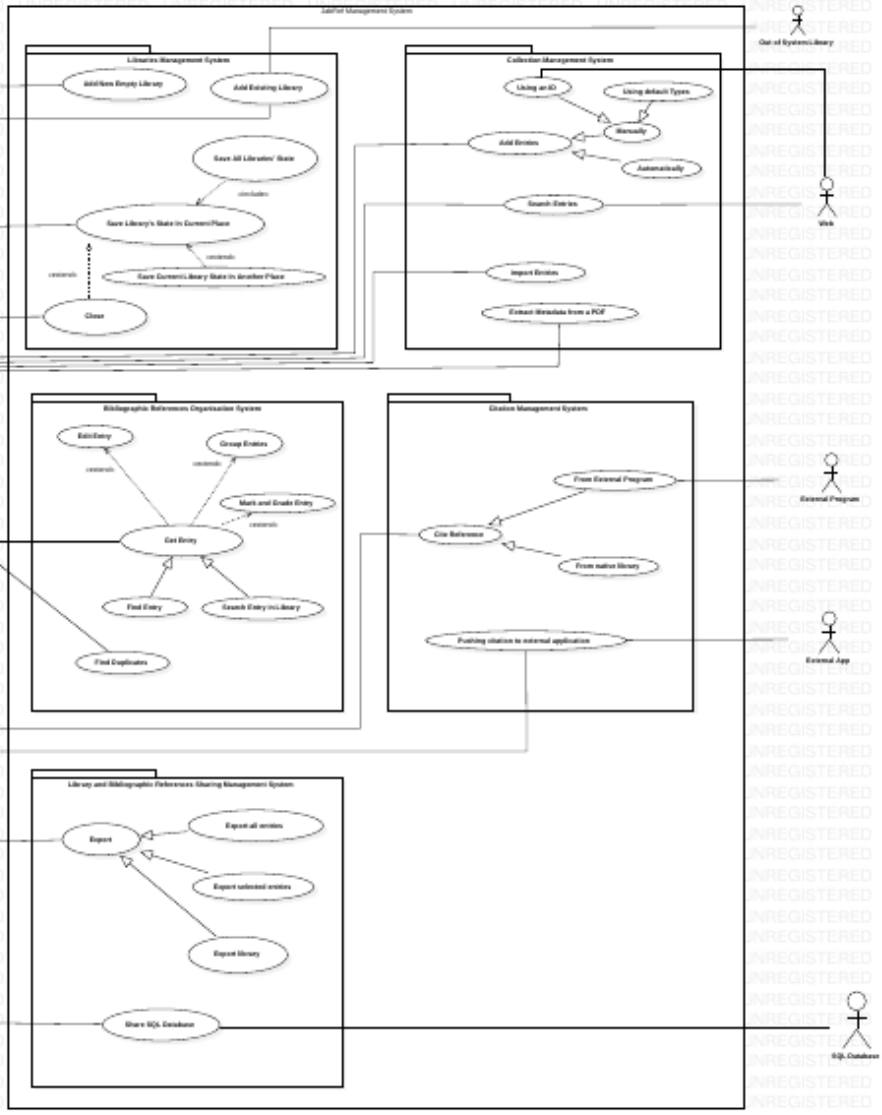
Throughout the project, the sprint plans were met (with the exception of sprint 4, where some tasks had to be continued in the following sprint) and daily meetings allowed the members of the team to constantly review, re-do and re-analyse the different tasks, making it easy for everyone to present their questions and understand the corresponding solutions.

The first uploaded document was the Functional Requirements Analysis. This document is a simple version of the common Agile Requirements Analysis, and it was created so as to help the group not only to better comprehend the structure of JabRef and its functionalities, but also to improve the team's efficiency since a more realistic plan was drawn.

2 Use Case Diagrams, Descriptions and Identification of Actors

During the creating of use case descriptions, the IDs given to the use cases matched the IDs of the corresponding tasks. Yet, the S Team decided to simplify the identification of the use cases, which is why this section presents a re-structure on the use case IDs.

The JabRef Funcionalities Use Case Diagram is presented in the following picture, but a larger, more readable version of it is presented at the end of this document, in attachment.


JabRef Management System
Out of System Library
Libraries Management System
Add New Empty Library
Add Existing Library
Save All Libraries' State
Save Library's State in Current Place
Save Current Library State in Another Place
Close
Collection Management System
Using an ID
Using default Types
Manually
Add Entries
Automatically
Search Entries
Import Entries
Extract Metadata from a PDF
Web
Bibliographic References Organization System
Edit Entry
Group Entries
Mark and Grade Entry
Get Entry
Find Entry
Search Entry in Library
Find Duplicates
Author
Citation Management System
From External Program
Cite Reference
From native library
Pushing citation to external application
External Program
External App
Library and Bibliographic References Sharing Management System
Export
Export all entries
Export selected entries
Export library
Share SQL Database
SQL Database

2.1 Managing Libraries

2.1.1 Identification of Actors

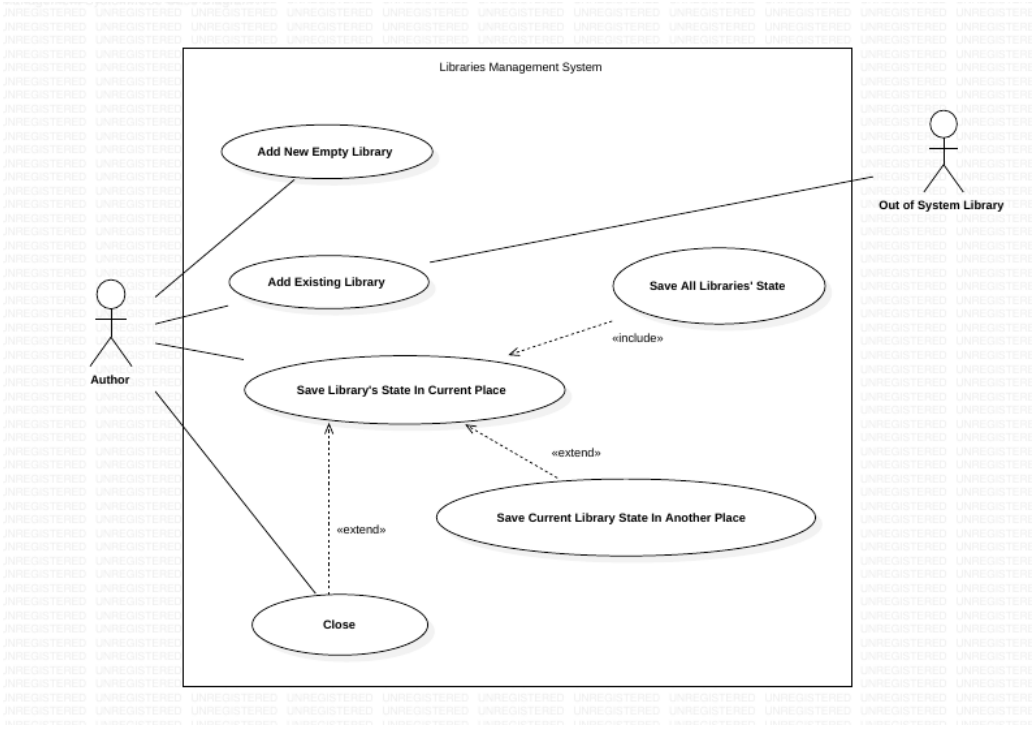
The identified actors in this use case were:

- Author: a user of JabRef. Characteristics: human, active.
- Out of system library: a library that already exists outside of the author's JabRef application. Characteristics: non-human, passive.

2.1.2 Use Case Description

- **Use Case Name**: Add a Library to the Program
- **ID**: A
- **Description**: An author adds a library to their JabRef application
- **Main Actor**: Author
- **Secondary Actor**: Out of system library

2.1.3 Use Case Diagram



2.2 Managing the Collection of Bibliographic References

2.2.1 Identification of Actors

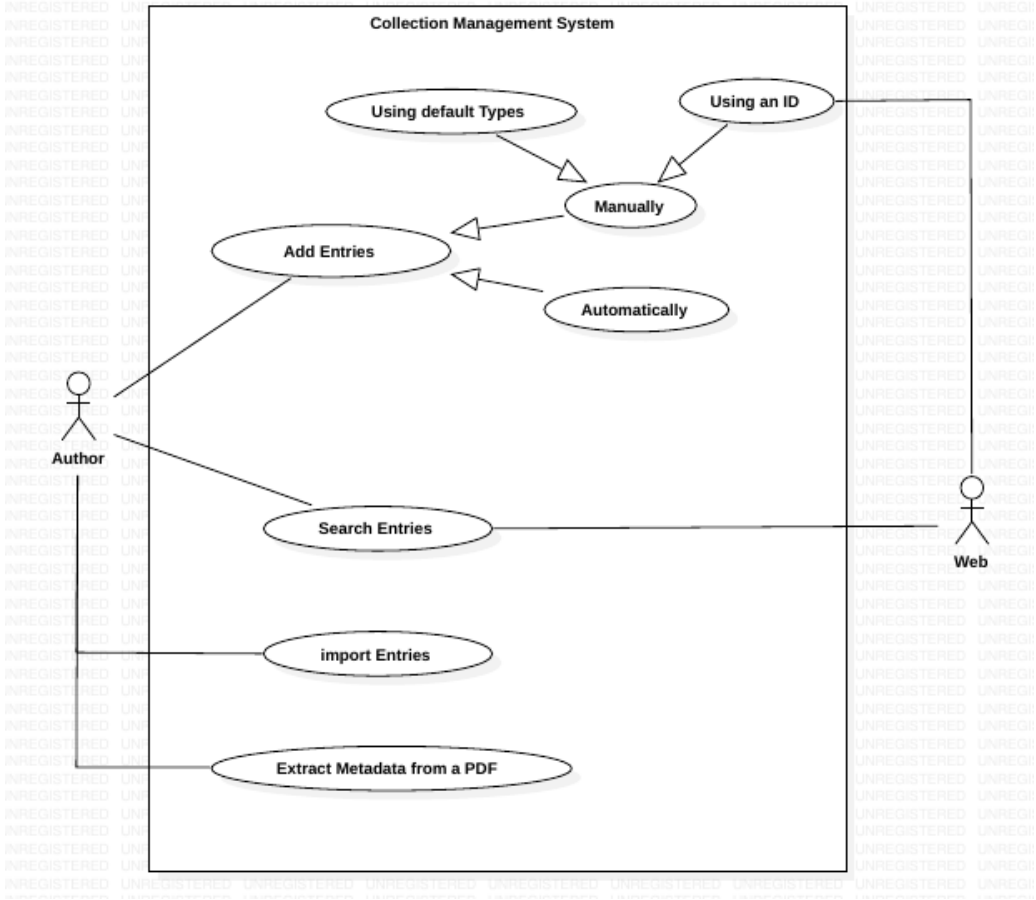
The identified actors in this use case were:

- Author: a user of JabRef. Characteristics: human, active
- Web: online scientific catalogues like CrossRef, Google Scholar or IEEEXplore. It can also represent. online databases. Characteristics: non-human, passive.

2.2.2 Use Case Description

- **Use Case Name**: Add an Entry to the Program
- **ID**: B
- **Description**: An author collects entries automatically or manually to their JabRef application.
- **Main Actor**: Author
- **Secondary Actor**: Web

2.2.3 Use Case Diagram



2.3 Organising Bibliographic References

2.3.1 Identification of Actors

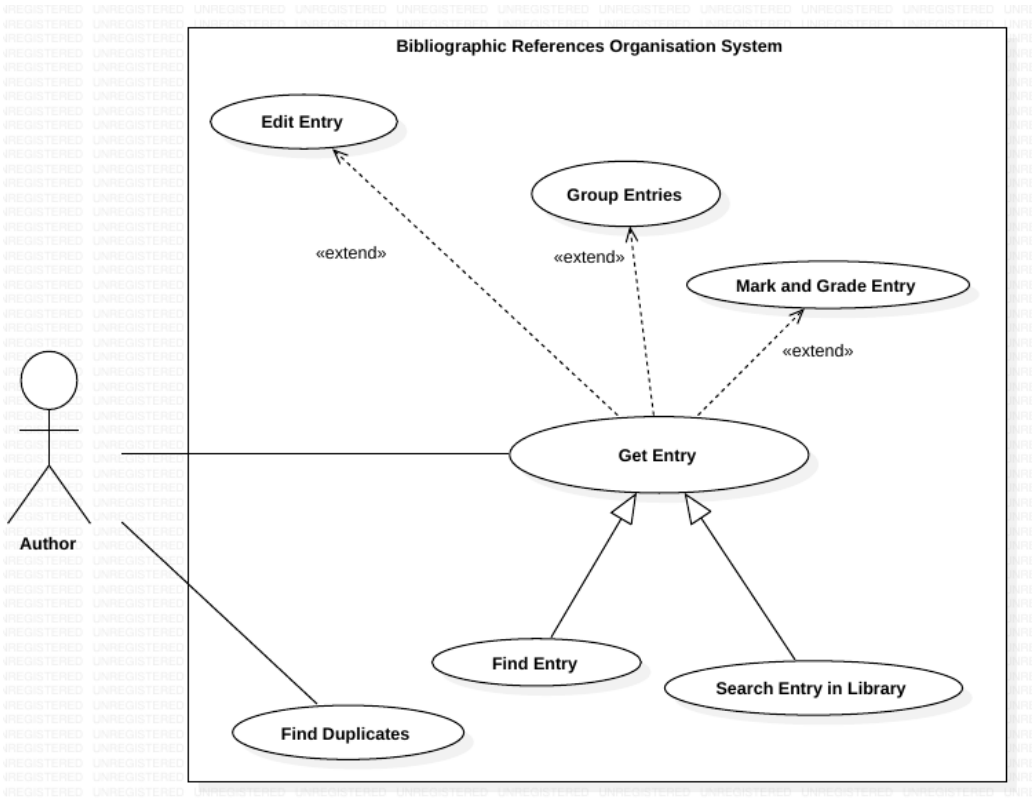
The identified actors in this use case were:

- Author: a user of JabRef. Characteristics: human, active.

2.3.2 Use Case Description

- **Use Case Name**: Organise an entry in the program
- **ID**: C
- **Description**: Organise a bibliographic reference (or several) in the author's JabRef Application
- **Main Actor**: Author
- **Secondary Actor**: None

2.3.3 Use Case Diagram



2.4 Citing Bibliographic References

2.4.1 Identification of Actors

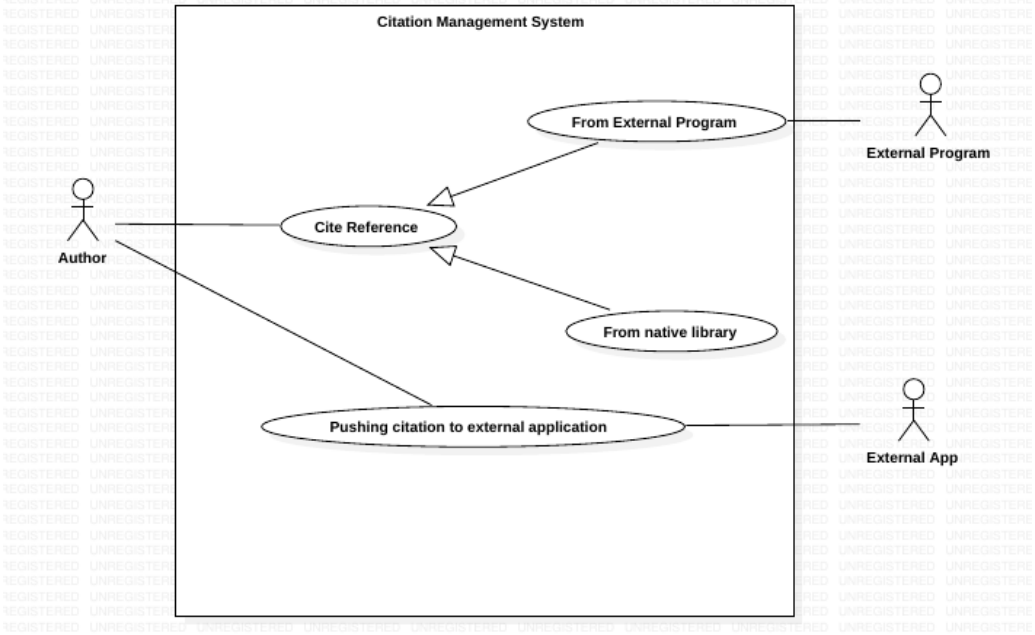
The identified actors in this use case were:

- Author: a user of JabRef. Characteristics: human, active.
- External Program: a program such as Microsoft, Open, Bib(La)Tex, etc. Characteristics: non-human, passive.
- External App: an application such as Emacs, Kile, LyX, Texmaker, TeXstudio, Vim and WinEdt. Characteristics: non-human, passive.

2.4.2 Use Case Description

- **Use Case Name**: Citing an entry in the program
- **ID**: D
- **Description**: An author cites a bibliographic reference to an external program from their JabRef application
- **Main Actor**: Author
- **Secondary Actors**: External Program, External App

2.4.3 Use Case Diagram



2.5 Sharing Bibliographic References

2.5.1 Identification of Actors

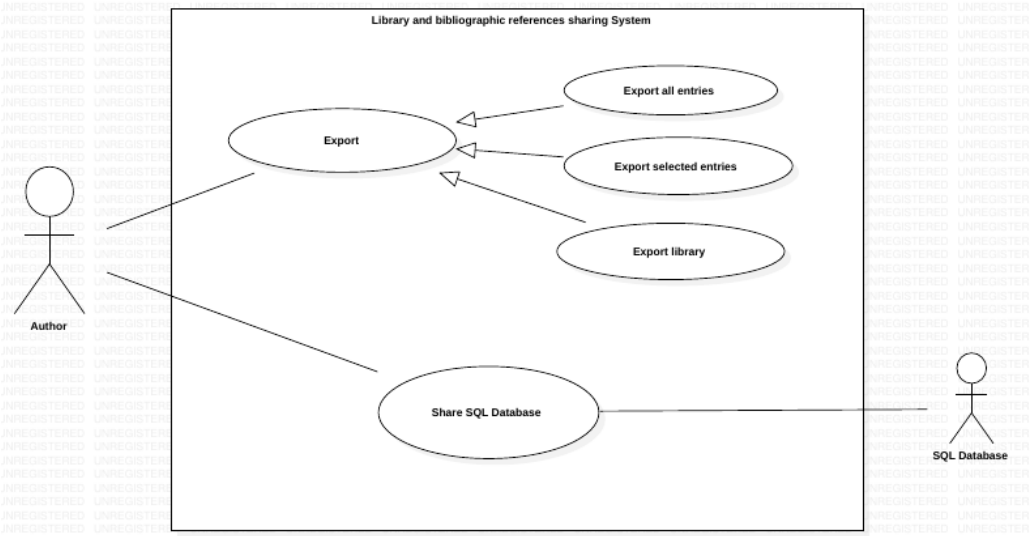
The identified actors in this use case were:

- Author: a user of JabRef. Characteristics: human, active.
- SQL Database: a SQL database that is connected to the author's JabRef application. Characteristics: non-human, passive.

2.5.2 Use Case Description

- **Use Case Name**: Share a library or bibliographic references
- **ID**: E
- **Description**:
- **Main Actor**: Author
- **Secondary Actor**: SQL Database

2.5.3 Use Case Diagram



3 Design Patterns

In Software Engineering, the term Design Pattern refers to typical solutions to commonly occurring problems in software design. There are three different types of Design Patterns: Creational, Structural and Behaviour; examples of all these kinds of patterns have been found in the JabRef codesbase.

3.1 Creational Design Patterns

Creational Design Patterns are design patterns that deal with object creation mechanisms, such as creation or cloning of new objects. The types of Creational Patterns identified in this codebase were Singleton and Factory.

3.1.1 Factory

1. SpecialFieldMenuItemFactory

   - **Code Snippet**:

```
48      public static Menu createSpecialFieldMenu(SpecialField field,
49                                                ActionFactory factory,
50                                                PreferencesService preferencesService,
51                                                UndoManager undoManager,
52                                                Function<SpecialFieldValueViewModel, Command> commandFactory) {
53          SpecialFieldViewModel viewModel = new SpecialFieldViewModel(field, preferencesService, undoManager);
54          Menu menu = factory.createMenu(viewModel.getAction());
55
56          for (SpecialFieldValue Value : field.getValues()) {
57              SpecialFieldValueViewModel valueViewModel = new SpecialFieldValueViewModel(Value);
58              menu.getItems().add(factory.createMenuItem(valueViewModel.getAction(), commandFactory.apply(valueViewModel)));
59          }
60          return menu;
61      }
```

   - **Location**: src/main/java/org/jabref/gui/specialfields/SpecialFieldMenuItemFactory.java
   - **Explanation**: The purpose of a factory object is to work as a "virtual" constructor and have methods that create objects, instead of a typical constrcutor call; this is what the methods in this *SpecialFieldMenuItemFactory* class do, by creating different objects (e.g.: creating a menu from the created object *SpecialFieldViewModel*) that relate to their correspondent methods (respectively, the method *createSpecialFieldMenu*).

2. ExporterFactory

- **Code Snippet**:

```java
public class ExporterFactory {

    /**
     * Global variable that is used for counting output entries when exporting:
     *
     * @deprecated find a better way to do this
     */
    @Deprecated public static int entryNumber;

    private final List<Exporter> exporters;

    private ExporterFactory(List<Exporter> exporters) { this.exporters = Objects.requireNonNull(exporters); }

    public static ExporterFactory create(List<TemplateExporter> customFormats,
                                         LayoutFormatterPreferences layoutPreferences,
                                         SavePreferences savePreferences,
                                         XmpPreferences xmpPreferences,
                                         BibDatabaseMode bibDatabaseMode,
                                         BibEntryTypesManager entryTypesManager) {
```

- **Location**: src/main/java/org/jabref/logic/exporter/ExporterFactory.java
- **Explanation**: The goal of a Factory design pattern is to work as a "virtual" constructor and define a method that should be used for creating objects, instead of direct construtor calls. The purpose is to hide the creation of instances of a given type so the client does not depend on concrete classes and provide hooks for subclasses. Through this code snippet, the *ExporterFactory* acts as a creator and if there are multiple clients that want to instantiate the same set of classes, this cuts out redundant code.

3. ActionFactory

   - **Code Snippet**:

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

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

    return checkMenuItem;
}
```

   - **Location**: src/main/java/org/jabref/gui/actions/ActionFactory.java
   - **Explanation**: This *ActionFactory* class is a factory method pattern because it is responsible for the creation of several instances of objects like *MenuItem*, *CheckMenuItem*, *Menu*, *Button*, etc. This way other classes(client) can use ActionFactory to create actions(objects) like the mentioned before and this process will be hidden behind the ActionFactory so the client code does not depend on concrete classes.

4. EntryTypeFactory

   - **Code Snippet**:

```java
private static boolean isBiblatex(EntryType type) {
    return BiblatexEntryTypeDefinitions.ALL.stream().anyMatch(bibEntryType -> bibEntryType.getType().equals(type));
}

public static EntryType parse(String typeName) {

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

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

   - **Location**: src/main/java/org/jabref/model/entry/types/EntryTypeFactory.java
   - **Explanation**: This *EntryTypeFactory* class is used to make different types of EntryTypes; in other words, it is used as a factory of entry types. In fact, *EntryTypeFactory* helps hiding the creation of objects from the client and this way the client code does not depend on concrete classes.

3.1.2 Singleton

1. JabRefPreferences
    - **Code Snippet**:

```
443     // The constructor is made private to enforce this as a singleton class:
444     private JabRefPreferences() {
445         try {
446             if (new File("jabref.xml").exists()) {
447                 importPreferences(Path.of("jabref.xml"));
448             }
449         } catch (JabRefException e) {
450             LOGGER.warn("Could not import preferences from jabref.xml: " + e.getMessage(), e);
451         }
452
```

    - **Location**: src/main/java/org/jabref/preferences/JabRefPreferences.java
    - **Explanation**: Since a variable is used to ensure that there is only one instance of a *JabRefPreferences*, we can see this is a Singleton pattern. We can also see, as presented in the code snippet, that the Singleton class is enforced by the use of a private constructor.

2. ExternalFileTypes

- **Code Snippet**:

```java
// Do not make this class final, as it otherwise can't be mocked for tests
public class ExternalFileTypes {

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

    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;
    }
```

- **Location**: /src/main/java/org/jabref/gui/externalfiletype/ExternalFileTypes.java
- **Explanation**: This class, ExternalFileTypes, uses a variable to use the only instance that is possible to create, and returns that instance with the public method *getInstance()*.

3.2 Structural Design Patterns

The design patterns that describe how objects are connected to each other are called Structural Design Patterns. The identified patterns of this kind were the Adpater, Composite, Proxy and Decorator.

3.2.1 Adapter

1. CSLAdapter

   - **Code Snippet**:

```
68      private void initialize(String newStyle, CitationStyleOutputFormat newFormat) throws IOException {
69          if ((cslInstance == null) || !Objects.equals(newStyle, style)) {
70              // lang and forceLang are set to the default values of other CSL constructors
71              cslInstance = new CSL(dataProvider, new JabRefLocaleProvider(),
72                      new DefaultAbbreviationProvider(), newStyle, "en-US");
73              style = newStyle;
74          }
75
76          if (!Objects.equals(newFormat, format)) {
77              cslInstance.setOutputFormat(newFormat.getFormat());
78              format = newFormat;
79          }
80      }
```

   - **Location**: src/main/java/org/jabref/logic/citationstyle/CSLAdapter.java
   - **Explanation**: The Adpator is a design pattern where there is the conversion of an interface (of a given class) into another interface the client is expecting to see. The *CSLAdapter* class, is used as an Adapter to CSL (CitationStylePreviewLayout) classes. It recreates a CSL after a certain operation; in this case, after changing the style. The Adapter allows classes which would usually be incompatible, to work together.

3.2.2 Composite

1. CompositeIdFetcher

   - **Code Snippet**:

```java
public Optional<BibEntry> performSearchById(String identifier) throws FetcherException {
    Optional<DOI> doi = DOI.parse(identifier);
    if (doi.isPresent()) {
        return new DoiFetcher(importFormatPreferences).performSearchById(doi.get().getNormalized());
    }
    Optional<ArXivIdentifier> arXivIdentifier = ArXivIdentifier.parse(identifier);
    if (arXivIdentifier.isPresent()) {
        return new ArXiv(importFormatPreferences).performSearchById(arXivIdentifier.get().getNormalized());
    }
    Optional<ISBN> isbn = ISBN.parse(identifier);
    if (isbn.isPresent()) {
        return new IsbnFetcher(importFormatPreferences).performSearchById(isbn.get().getNormalized());
    }
    Optional<IacrEprint> iacrEprint = IacrEprint.parse(identifier);
    if (iacrEprint.isPresent()) {
        return new IacrEprintFetcher(importFormatPreferences).performSearchById(iacrEprint.get().getNormalized());
    }

    return Optional.empty();
}
```

   - **Location**: src/main/java/org/jabref/logic/importer/CompositeIdFetcher.java
   - **Explanation**: In this class, new leaves are being created in different methods, regarding the first composite object. In this code snippet, for example, new objects are created in each "return new" line, according to the characteristics of the *Optional* $<DOI>$ object. The Composite pattern allows clients to treat both individual objects and compositions of objects uniformly.

2. CompositeFormat

- **Code Snippet**:

```
19      public CompositeFormat() { formatters = Collections.emptyList(); }
22
23      public CompositeFormat(LayoutFormatter first, LayoutFormatter second) { formatters = Arrays.asList(first, second); }
26
27      public CompositeFormat(LayoutFormatter[] formatters) { this.formatters = Arrays.asList(formatters); }
30
31      @Override
32      public String format(String fieldText) {
33          String result = fieldText;
34          for (LayoutFormatter formatter : formatters) {
35              result = formatter.format(result);
36          }
37          return result;
38      }
```

- **Location**: src/main/java/org/jabref/logic/layout/format/CompositeFormat.java
- **Explanation**: The goal of a Composite Design Pattern goal is to compose objects into tree structures to represent part-whole hierarchies. This pattern lets clients treat individual objects and compositions of objects uniformly. Through this code snippet, the component, *LayoutFormatter*, declares the interface for the objects in the composition and implements their default behaviour (their format). The composite, *CompositeFormat*, defines the behavior for the components children and stores their child componentes.

3. CompositeSearchBasedFetcher

- **Code Snippet**:

```java
public Optional<BibEntry> performSearchById(String identifier) throws FetcherException {
    Optional<DOI> doi = DOI.parse(identifier);
    if (doi.isPresent()) {
        return new DoiFetcher(importFormatPreferences).performSearchById(doi.get().getNormalized());
    }
    Optional<ArXivIdentifier> arXivIdentifier = ArXivIdentifier.parse(identifier);
    if (arXivIdentifier.isPresent()) {
        return new ArXiv(importFormatPreferences).performSearchById(arXivIdentifier.get().getNormalized());
    }
    Optional<ISBN> isbn = ISBN.parse(identifier);
    if (isbn.isPresent()) {
        return new IsbnFetcher(importFormatPreferences).performSearchById(isbn.get().getNormalized());
    }
    Optional<IacrEprint> iacrEprint = IacrEprint.parse(identifier);
    if (iacrEprint.isPresent()) {
        return new IacrEprintFetcher(importFormatPreferences).performSearchById(iacrEprint.get().getNormalized());
    }

    return Optional.empty();
}
```

- **Location**: src/main/java/org/jabref/logic/importer/fetcher/CompositeSearchBasedFetcher.java
- **Explanation**: This *CompositeSearchBasedFetcher* class is a Composite Design Pattern because it allows to compose objects into tree structures. The interface *SearchBasedFetcher* implements the default behavior provided by the *performSearch()* method; then, the composite class defines the behavior for the children component and stores them in a set of *SearchBasedFetchers* called fetchers. This allows the client to treat individual objects and compositions uniformly, keeping it simple and easier to add new kinds of components. This particular composite class does not have a remove method to allow for the removal of children.

3.2.3 Decorator

1. IconValidationDecorator

   - **Code Snippet**:

```
@Override
public Node createDecorationNode(ValidationMessage message) {
    Node graphic = Severity.ERROR == message.getSeverity() ? createErrorNode() : createWarningNode();
    graphic.getStyleClass().add(Severity.ERROR == message.getSeverity() ? "error-icon" : "warning-icon");
    Label label = new Label();
    label.setGraphic(graphic);
    label.setTooltip(createTooltip(message));
    label.setAlignment(position);
    return label;
}

@Override
protected Tooltip createTooltip(ValidationMessage message) {
    Tooltip tooltip = new Tooltip(message.getText());
    tooltip.getStyleClass().add(Severity.ERROR == message.getSeverity() ? "tooltip-error" : "tooltip-warning");
    return tooltip;
}
```

   - **Location**: src/main/java/org/jabref/gui/util/IconValidationDecorator.java
   - **Explanation**: This IconValidationDecorator class attach additional responsibilities to an icon. In this Decorator Design Pattern, the *BaseDecorator* (or the decorator abstract class) is in reality the *GraphicValidationDecoration* class, which *IconValidationDecorator* class extends, and this *IconValidationDecorator* is only one of its concrete decorator classes that will each provide an increment of behavior.

3.2.4 Proxy

1. ProxyAuthenticator
    - **Code Snippet**:

```java
public class ProxyAuthenticator extends Authenticator {

    @Override
    protected PasswordAuthentication getPasswordAuthentication() {
        if (getRequestorType() == RequestorType.PROXY) {
            String prot = getRequestingProtocol().toLowerCase(Locale.ROOT);
            String host = System.getProperty(prot + ".proxyHost", "");
            String port = System.getProperty(prot + ".proxyPort", "80");
            String user = System.getProperty(prot + ".proxyUser", "");
            String password = System.getProperty(prot + ".proxyPassword", "");
            if (getRequestingHost().equalsIgnoreCase(host) && (Integer.parseInt(port) == getRequestingPort())) {
                return new PasswordAuthentication(user, password.toCharArray());
            }
        }
        return null;
    }
}
```

    - **Location**: src/main/java/org/jabref/logic/net/ProxyAuthenticator.java
    - **Explanation**: This class returns an object "PasswordAuthentication" if there is a match between the "System" object and a requestor; that is, the ProxyAuthenticator class works as an intermediate, which meets the definition of a proxy object in a Proxy design pattern.

3.3 Behavioural Design Patterns

Behavioural Design Patterns identify common communication patterns among objects. By doing so, these patterns increase flexibility in carrying out communication. The Behavioural Design Patterns found in the provided codebase were the Template, Command and State.

3.3.1 Command

1. AuxCommandLine
    - **Code Snippet**:

```java
public class AuxCommandLine {
    private final String auxFile;
    private final BibDatabase database;

    public AuxCommandLine(String auxFile, BibDatabase database) {
        this.auxFile = StringUtil.getCorrectFileName(auxFile, "aux");
        this.database = database;
    }

    public BibDatabase perform() {
        BibDatabase subDatabase = null;

        if (!auxFile.isEmpty() && (database != null)) {
            AuxParser auxParser = new DefaultAuxParser(database);
            AuxParserResult result = auxParser.parse(Path.of(auxFile));
            subDatabase = result.getGeneratedBibDatabase();
            // print statistics
            System.out.println(new AuxParserResultViewModel(result).getInformation(true));
        }
        return subDatabase;
    }
}
```

    - **Location**: src/main/java/org/jabref/cli/AuxCommandLine.java
    - **Explanation**: This AuxCommandLine class is a Command pattern because it features a constructor that creates a command and an execute function called *perform()*. This allows a sender to create a command using this class to encapsulate an order, and the command can be manipulated as an object and placed into a queue so that different commands can be scheduled to be completed at different times because the command is only sent when it is executed. This command pattern is incomplete and could be improved with the addition of a unexecute/undo function in order to be able to undo or redo a command before sending it.

3.3.2 State

1. StateManager

   - **Code Snippet**:

```
46      private final CustomLocalDragboard localDragboard = new CustomLocalDragboard();
47      private final ObservableList<BibDatabaseContext> openDatabases = FXCollections.observableArrayList();
48      private final OptionalObjectProperty<BibDatabaseContext> activeDatabase = OptionalObjectProperty.empty();
49      private final ReadOnlyListWrapper<GroupTreeNode> activeGroups = new ReadOnlyListWrapper<>(FXCollections.observableAr
50      private final ObservableList<BibEntry> selectedEntries = FXCollections.observableArrayList();
51      private final ObservableMap<BibDatabaseContext, ObservableList<GroupTreeNode>> selectedGroups = FXCollections.observ
52      private final OptionalObjectProperty<SearchQuery> activeSearchQuery = OptionalObjectProperty.empty();
53      private final ObservableMap<BibDatabaseContext, IntegerProperty> searchResultMap = FXCollections.observableHashMap()
54      private final OptionalObjectProperty<Node> focusOwner = OptionalObjectProperty.empty();
55      private final ObservableList<Task<?>> backgroundTasks = FXCollections.observableArrayList(task -> new Observable[] {
56      private final EasyBinding<Boolean> anyTaskRunning = EasyBind.reduce(backgroundTasks, tasks -> tasks.anyMatch(Task::i
57      private final EasyBinding<Double> tasksProgress = EasyBind.reduce(backgroundTasks, tasks -> tasks.filter(Task::isRun
58      private final ObservableMap<String, DialogWindowState> dialogWindowStates = FXCollections.observableHashMap();
59
60      public StateManager() { activeGroups.bind(Bindings.valueAt(selectedGroups, activeDatabase.orElse(null))); }
63
64      public CustomLocalDragboard getLocalDragboard() { return localDragboard; }
67
68      public ObservableList<BibDatabaseContext> getOpenDatabases() { return openDatabases; }
71
72      public OptionalObjectProperty<BibDatabaseContext> activeDatabaseProperty() { return activeDatabase; }
```

   - **Location**: src/main/java/org/jabref/gui/StateManager.java
   - **Explanation**: This *StateManager* class allows an object to alter its behavior when its internal state changes, making it seem as if the object changed classes. This meets the definition of a State Design Pattern: an objects' behavior depends on its state and changes at run-time depending on that same state.

3.3.3 Template

1. TemplateExporter

   - **Code Snippet**:

```java
public void export(final BibDatabaseContext databaseContext, final Path file,
                   final Charset encoding, List<BibEntry> entries) throws Exception {
    Objects.requireNonNull(databaseContext);
    Objects.requireNonNull(entries);

    Charset encodingToUse = StandardCharsets.UTF_8;
    if (encoding != null) {
        encodingToUse = encoding;
    } else if (this.encodingOverwritten != null) {
        encodingToUse = this.encodingOverwritten;
    }

    if (entries.isEmpty()) { // Do not export if no entries to export -- avoids exports with only template text
        return;
    }

    try (AtomicFileWriter ps = new AtomicFileWriter(file, encodingToUse)) {
        Layout beginLayout = null;

        // Check if this export filter has bundled name formatters:
        // Add these to the preferences, so all layouts have access to the custom name formatters:
        readFormatterFile();

        List<String> missingFormatters = new ArrayList<>(1);
```

   - **Location**: src/main/java/org/jabref/logic/exporter/TemplateExporter.java
   - **Explanation**: The goal of a Template Design Pattern is to define the skeleton of an algorithm in an operation, deferring some steps to subclasses, where its subclasses can override the method implementation as per need. Through this code snippet, the abstract class *Exporter*, defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm. The concrete class *TemplateExporter* implements the primitive operations to carry out specific steps of the algorithm.

4 Code Smells

A Code Smell is a surface indication that usually corresponds to a deeper problem in the system. In this part of the document all the identified code smells in the JabRef codebase are presented. Some of the code smells were added during Sprint Restrospectives, in open discussions.

4.1 Commented Code

JournalEditorViewModel

- **Code Snippet**:

```
28          if (journalAbbreviationRepository.isKnownName(name)) {
29              Optional<String> nextAbbreviation = journalAbbreviationRepository.getNextAbbreviation(name);
30
31              if (nextAbbreviation.isPresent()) {
32                  text.set(nextAbbreviation.get());
33                  // TODO: Add undo
34                  // panel.getUndoManager().addEdit(new UndoableFieldChange(entry, editor.getName(), text, nextAbbreviation));
35              }
36          }
```

- **Location**: /src/main/java/org/jabref/gui/fieldeditors/JournalEditorViewModel.java
- **Explanation**: This class presents a line of code that is commented, taking a "reminder" nature. Commented Code can show that a programmer is insecure about their work and it can lead to a continuous agglomeration of commented code lines, making the program illegible.
- **Refactoring Proposal**: The best way to fix this code smell is to do it (like in "TODO"): really work on the commented code and make it function properly.

4.2 Data Class

1. Action

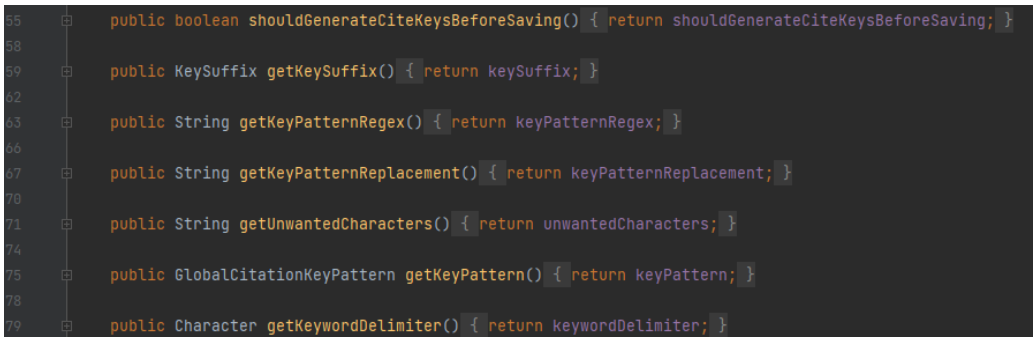
   - **Code Snippet**:

```
8   public interface Action {
9       default Optional<JabRefIcon> getIcon() { return Optional.empty(); }
12
13      default Optional<KeyBinding> getKeyBinding() { return Optional.empty(); }
16
17      String getText();
18
19      default String getDescription() { return ""; }
22  }
```

   - **Location**: src/main/java/org/jabref/gui/actions/Action.java
   - **Explanation**: *Action* is a very small classe that has no real functionality, being able to present only getter methods such as *getIcon()* and *getKeyBinding()*. This matches the definition of the code smell called Data Class.
   - **Refactoring Proposal**: Instead of having this class, perhaps the methods in it should be placed in another, better suited class.

2. CitationKeyPatternPreferences

   - **Code Snippet**:



```
65      public boolean shouldGenerateCiteKeysBeforeSaving() { return shouldGenerateCiteKeysBeforeSaving; }
68
69      public KeySuffix getKeySuffix() { return keySuffix; }
62
63      public String getKeyPatternRegex() { return keyPatternRegex; }
66
67      public String getKeyPatternReplacement() { return keyPatternReplacement; }
70
71      public String getUnwantedCharacters() { return unwantedCharacters; }
74
75      public GlobalCitationKeyPattern getKeyPattern() { return keyPattern; }
78
79      public Character getKeywordDelimiter() { return keywordDelimiter; }
```

   - **Location**: src/main/java/org/jabref/logic/citationkeypattern/CitationKeyPatternPreferences.java
   - **Explanation**: In this particular code snippet we can see the Data Class code smell; it can be seen since the class presents only a constructor and getter methods, and no real functionality - it only stores and returns data.
   - **Refactoring Proposal**: The code could be fixed by moving some methods from the client code to this class (if those methods would be better suited to this class), or perhaps other features could be added to this class so it could have a real functionality. One other, more "modern" way of solving this code smell would be to transform this class into a Record class (Record is a new Java type used to save and organise data).

3. GuiPreferences

- **Code Snippet**:

```
55      public void setPositionX(double positionX) { this.positionX.set(positionX); }
58
59      public double getPositionY() { return positionY.get(); }
62
63      public DoubleProperty positionYProperty() { return positionY; }
66
67      public void setPositionY(double positionY) { this.positionY.set(positionY); }
70
71      public double getSizeX() { return sizeX.get(); }
74
75      public DoubleProperty sizeXProperty() { return sizeX; }
78
79      public void setSizeX(double sizeX) { this.sizeX.set(sizeX); }
```

- **Location**: /src/main/java/org/jabref/preferences/GuiPreferences.java
- **Explanation**: It's a data class because its only methods are getters and setters, and it has no real functionality.
- **Refactoring Proposal**: An easy way to fix this code smell would be to check if these getters/setters methods could be done by another classes, and move the methods to said class, being able to delete the class *GuiPreferences*.

4.3 Data Clump

FieldFormatterCleanupsPanelViewModel

- **Code Snippet**:

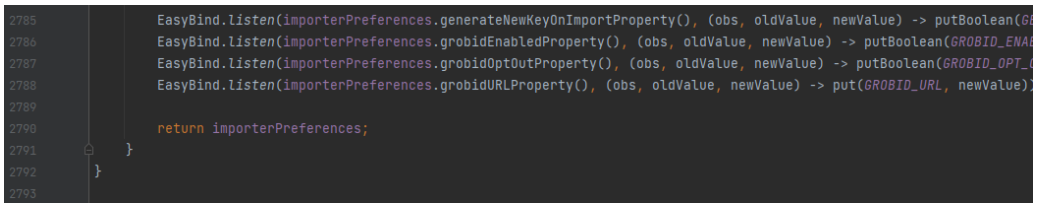
```
11      private final boolean shouldAvoidOverwriteCiteKey;
12      private boolean shouldWarnBeforeOverwriteCiteKey;
13      private final boolean shouldGenerateCiteKeysBeforeSaving;
14      private final KeySuffix keySuffix;
15      private final String keyPatternRegex;
16      private final String keyPatternReplacement;
17      private final String unwantedCharacters;
18      private final GlobalCitationKeyPattern keyPattern;
19      private final Character keywordDelimiter;
```

- **Location**: src/main/java/org/jabref/logic/citationkeypattern/CitationKeyPatternPreferences.java
- **Explanation**: Right at the beginning of this class we can see that it contains identical groups of variables - clumps.
- **Refactoring Proposal**: A refactoring proposal for this code smell is to simply group variables from the same clump and place them in a single class, making sure data classes are not being created. This improves the coupling in program, by lowering it.

4.4 Divergent Class

JabRefPreferences

- **Code Snippet**:



- **Location**: src/main/java/org/jabref/preferences/JabRefPreferences.java
- **Explanation**: This class matches the definition of a Divergent Class because it became so big (it presents 2291 lines of code) that its essential purpose got lost and the class ended up with a poor separation of concerns. This code smell was found as the most concerning trouble spot in one of the sets of code metrics. Divergent classes can be extremely problematic and jeopardize code maintenance, and they relate to Large Class code smells, given their size.
- **Refactoring Proposal**: In order to develop good quality software, each class should be created with one single purpose so as to reduce the variety of necessary changes. Hence, this class should be split into several classes that present more specific purposes.

4.5 Duplicate Code

1. ExporterFactory

   - **Code Snippet**:

```
44  exporters.add(new TemplateExporter("HTML", "html", "html", null, StandardFileType.HTML, layoutPreferences, savePreferences));
45  exporters.add(new TemplateExporter(Localization.lang("Simple HTML"), "simplehtml", "simplehtml", null, StandardFileType.HTML, layoutPreferences, save
46  exporters.add(new TemplateExporter("DocBook 5.1", "docbook5", "docbook5", null, StandardFileType.XML, layoutPreferences, savePreferences));
47  exporters.add(new TemplateExporter("DocBook 4", "docbook4", "docbook4", null, StandardFileType.XML, layoutPreferences, savePreferences));
48  exporters.add(new TemplateExporter("DIN 1505", "din1505", "din1505winword", "din1505", StandardFileType.RTF, layoutPreferences, savePreferences));
49  exporters.add(new TemplateExporter("BibO RDF", "bibordf", "bibordf", null, StandardFileType.RDF, layoutPreferences, savePreferences));
50  exporters.add(new TemplateExporter(Localization.lang("HTML table"), "tablerefs", "tablerefs", "tablerefs", StandardFileType.HTML, layoutPreferences,
51  exporters.add(new TemplateExporter(Localization.lang("HTML list"), "listrefs", "listrefs", "listrefs", StandardFileType.HTML, layoutPreferences, save
52  exporters.add(new TemplateExporter(Localization.lang("HTML table (with Abstract & BibTeX)"), "tablerefsabsbib", "tablerefsabsbib", "tablerefsabsbib",
53  exporters.add(new TemplateExporter("Harvard RTF", "harvard", "harvard", "harvard", StandardFileType.RTF, layoutPreferences, savePreferences));
54  exporters.add(new TemplateExporter("ISO 690 RTF", "iso690rtf", "iso690RTF", "iso690rtf", StandardFileType.RTF, layoutPreferences, savePreferences));
55  exporters.add(new TemplateExporter("ISO 690", "iso690txt", "iso690", "iso690txt", StandardFileType.TXT, layoutPreferences, savePreferences));
56  exporters.add(new TemplateExporter("Endnote", "endnote", "EndNote", "endnote", StandardFileType.TXT, layoutPreferences, savePreferences));
```

   - **Location**: /src/main/java/org/jabref/logic/exporter/ExporterFactory.java
   - **Explanation**: Duplicate code can be seen in this code snippet: the same parameters are repeated in the same place, when only one of its presences is truly necessary. For example, on line 53, the string "harvard" is used several times when it would only be necessary to write it once.
   - **Refactoring Proposal**: The simple and efficient way to solve this code smell is to eliminate the parameters that are being repeated in the function and replace those with a single variable (with an array, for example).

2. Version

- **Code Snippet**:

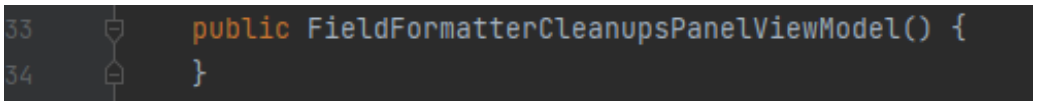
```
115     public boolean isNewerThan(Version otherVersion) {
116         Objects.requireNonNull(otherVersion);
117         if (Objects.equals( a: this, otherVersion)) {
118             return false;
119         } else if (this.getFullVersion().equals(BuildInfo.UNKNOWN_VERSION)) {
120             return false;
121         } else if (otherVersion.getFullVersion().equals(BuildInfo.UNKNOWN_VERSION)) {
122             return false;
123         }
124
125         // compare the majors
126         if (this.getMajor() > otherVersion.getMajor()) {
127             return true;
128         } else if (this.getMajor() == otherVersion.getMajor()) {
129             // if the majors are equal compare the minors
130             if (this.getMinor() > otherVersion.getMinor()) {
```

- **Location**: src/main/java/org/jabref/logic/util/Version.java
- **Explanation**: This snippet presents duplicate code because the first 3 if statements return false.
- **Refactoring Proposal**: Instead of having multiple if statements, it could be possible to simply group up all the conditions into one if statement.

4.6 Empty Method

FieldFormatterCleanupsPanelViewModel

- **Code Snippet**:



- **Location**: src/main/java/org/jabref/gui/commonfxcontrols/FieldFormatterCleanupsPanelViewModel.java
- **Explanation**: An Empty Method occurs when a class presents one or more methods that do not execute anything. Methods should never be empty. However, there are a few cases where a method might not have a body. For example, if is not yet or never will be supported, or if the method is intentionally blank override (probably because of another code smell called failed request). Exceptions for this code smell are empty methods in abstract classes.
- **Refactoring Proposal**: A refactoring proposal for this code smell is as simple as adding a nested comment explaining why the method needs to be empty or throwing an exception.

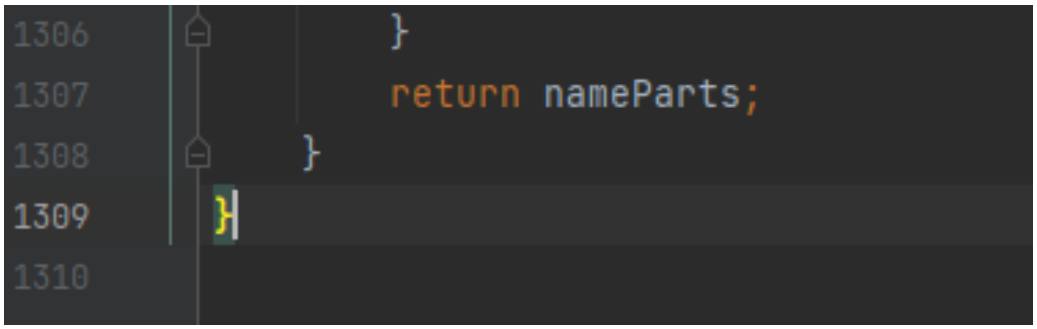
4.7 Large Class

BracketedPattern

- **Code Snippet**:

```
/**
 * The BracketedExpressionExpander provides methods to expand bracketed expressions, such as
 * [year]_[author]_[firstpage], using information from a provided BibEntry. The above-mentioned expression would yield
 * 2017_Kitsune_123 when expanded using the BibTeX entry "@Article{ authors = {O. Kitsune}, year = {2017},
 * pages={123-ô}}".
 */
public class BracketedPattern {
    private static final Logger LOGGER = LoggerFactory.getLogger(BracketedPattern.class);

```

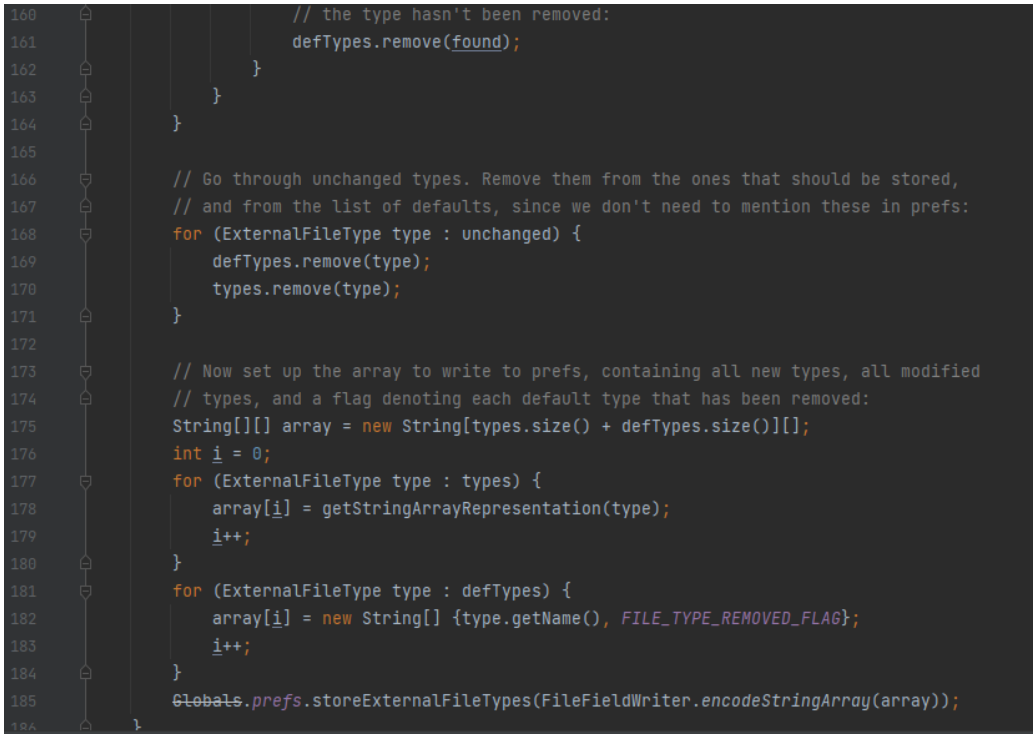


- **Location**: src/main/java/org/jabref/logic/citationkeypattern/BracketedPattern
- **Explanation**: This class is too large (it presents 1309 lines; it is a "blackhole class". A large class such as this shows that the programmer has not found suitable classes for the methods whithin it, so they decide to leave everything "class-less" in it. This code smell oftens leads to an increase in the number of comments, contributing to the development of another code smell - Too Many Comments.
- **Refactoring Proposal**: Every class should have a specific purpose so that cohesion among classes can be high. Therefore, one possible fix to this code smell could be to group the different methods and functionalities of the class into other classes, whenever they have a common ground.

4.8 Long Method

1. ExternalFileTypes
    - **Code Snippet**:

```
135 @   public void setExternalFileTypes(List<ExternalFileType> types) {
136
137         // First find a list of the default types:
138         List<ExternalFileType> defTypes = new ArrayList<>(getDefaultExternalFileTypes());
139         // Make a list of types that are unchanged:
140         List<ExternalFileType> unchanged = new ArrayList<>();
141
142         externalFileTypes.clear();
143         for (ExternalFileType type : types) {
144             externalFileTypes.add(type);
145
146             // See if we can find a type with matching name in the default type list:
147             ExternalFileType found = null;
148             for (ExternalFileType defType : defTypes) {
149                 if (defType.getName().equals(type.getName())) {
150                     found = defType;
151                     break;
152                 }
153             }
154             if (found != null) {
155                 // Found it! Check if it is an exact match, or if it has been customized:
156                 if (found.equals(type)) {
157                     unchanged.add(type);
158                 } else {
```



```
                    // the type hasn't been removed:
                    defTypes.remove(found);
                }
            }
        }

        // Go through unchanged types. Remove them from the ones that should be stored,
        // and from the list of defaults, since we don't need to mention these in prefs:
        for (ExternalFileType type : unchanged) {
            defTypes.remove(type);
            types.remove(type);
        }

        // Now set up the array to write to prefs, containing all new types, all modified
        // types, and a flag denoting each default type that has been removed:
        String[][] array = new String[types.size() + defTypes.size()][];
        int i = 0;
        for (ExternalFileType type : types) {
            array[i] = getStringArrayRepresentation(type);
            i++;
        }
        for (ExternalFileType type : defTypes) {
            array[i] = new String[] {type.getName(), FILE_TYPE_REMOVED_FLAG};
            i++;
        }
        Globals.prefs.storeExternalFileTypes(FileFieldWriter.encodeStringArray(array));
    }
```

- **Location**: src/main/java/org/jabref/gui/externalfiletype/ExternalFileTypes.java
- **Explanation**: The method *setExternalFileTypes(List <ExternalFileType >types)* is too long and too complex; it presents a very complicated disposition of if statements. This probably means that the method is doing more things than it should be doing, or that it is doing them in a more complex way than necessary.
- **Refactoring Proposal**: A solution for this code smell could be to re-arrange the if statements by creating more specific conditions. Another option could be to decompose this method into smaller ones.

2. VM

- **Code Snippet**:

```
119     private VM(CommonTree tree) {
120         this.tree = tree;
121
122         this.buildInFunctions = new HashMap<>( initialCapacity: 37);
123
124         /*
125          * Pops the top two (integer) literals, compares them, and pushes
126          * the integer 1 if the second is greater than the first, 0
127          * otherwise.
128          */
129         buildInFunctions.put(">", context -> {
130             if (stack.size() < 2) {
131                 throw new VMException("Not enough operands on stack for operation >");
132             }
133             Object o2 = stack.pop();
134             Object o1 = stack.pop();
135
136             if (!((o1 instanceof Integer) && (o2 instanceof Integer))) {
137                 throw new VMException("Can only compare two integers with >");
138             }
139
140             stack.push(((Integer) o1).compareTo((Integer) o2) > 0 ? VM.TRUE : VM.FALSE);
141         });
```

```
605         /*
606          * Pops the top two (function) literals, and keeps executing the
607          * second as long as the (integer) literal left on the stack by
608          * executing the first is greater than 0.
609          */
610         buildInFunctions.put("while$", this::whileFunction);
611
612         buildInFunctions.put("width$", new WidthFunction( vm: this));
613
614         /*
615          * Pops the top (string) literal and writes it on the output buffer
616          * (which will result in stuff being written onto the bbl file when
617          * the buffer fills up).
618          */
619         buildInFunctions.put("write$", context -> {
620             String s = (String) stack.pop();
621             VM.this.bbl.append(s);
622         });
623     }
```

- **Location**: src/main/java/org/jabref/logic/bst/VM.java
- **Explanation**: *VM(CommonTree)* is way too long, matching the definition of Long Method code smell. In fact, this method is so long (it goes from line 119 to 623) that only a portion of it is presented in the code snippet. A long method indicates that this method is more complex that it should be: it can have more things happening inside him than he should have. For the sake of legibility, it is important to fix long methods. In this case, the method consists on a collection of functions of different goals.
- **Refactoring Proposal**: The easiest fix for this code smell would be divide this method into other methods, each responsible for a specific functionality.

4.9 Long Parameter List

1. ArgumentProcessor

   - **Code Snippet**:

```
208
209     private void writeMetadatatoPDFsOfEntry(BibDatabaseContext databaseContext, String citeKey, BibEntry entry, Charset encoding, FilePreferences filePrefe
210         try {
```

```
208
209     nces filePreferences, XmpPdfExporter xmpPdfExporter, EmbeddedBibFilePdfExporter embeddedBibFilePdfExporter, boolean writeXMP, boolean embeddBibfile) {
210
```

   - **Location**: src/main/java/org/jabref/cli/ArgumentProcessor.java
   - **Explanation**: A code smell of the type Long Parameter List is present when a certain method presents too many parameters (usually, more than 4); in this class, the method *writeMetadatatoPDFsOfEntry* has 9 parameters.
   - **Refactoring Proposal**: In order to solve this code smell, one could simply group the parameters into different types / classes, introducing new parameter objects. E.g.: have a class *Entry* that would contain the parameters *databaseContext*, *citeKey*, *entry* and *encoding*; have a class *Exporter* that would contain *filePreferences*, *xmpPdfExporter*, *embeddedBibFilePdfExporter*.

2. ArgumentProcessor

   - **Code Snippet**:

```
342
343     private void writeMetadatatoPdfByFileNames(BibDatabaseContext databaseContext, BibDatabase dataBase, Vector<String> fileNames, Charset encoding, Fil
344         for (String fileName : fileNames) {
```

```
342
343 @   s filePreferences, XmpPdfExporter xmpPdfExporter, EmbeddedBibFilePdfExporter embeddedBibFilePdfExporter, boolean writeXMP, boolean embeddBibfile) {
344
```

   - **Location**: src/main/java/org/jabref/cli/ArgumentProcessor.java
   - **Explanation**: A Long Parameter List code smell indicates that a new structute should be created, or it shows that the method is doing things it should not be doing (such as creating other code smells like Inappropriate Intimacy). As seen in the code snippet, the method *writeMetadatatoPdfByFileNames* has too many arguments.
   - **Refactoring Proposal**: A refactoring proposal for this code smell would be to limit the number of parameters to a maximum of 4 (even though the threshold established by JabRef was 7). Let's say we created two classes: one that stored all the information about the database and it's context - *BibData* class - and a second class that stores the files preferences as well as its PDF exporter types - *File* class. This way, it would be possible to access the information of each argument inside its class. For example, *BibData* class should have a method that returns the *databaseContext* and another one that returns the *dataBase*. Hence, after refactoring, the code snippet would look like: *writeMetadatatoPdfByFileNames(BibData data, Vector <String >fileNames, Charset encoding, File f, boolean writeXMP, boolean embeddBibfile)...*

4.10 Message Chains

1. BibEntryWriter

   - **Code Snippet**:

```
List<Field> requiredFields = type.get() BibEntryType
                                    .getRequiredFields() Set<OrFields>
                                    .stream() Stream<OrFields>
                                    .flatMap(Collection::stream) Stream<Field>
                                    .sorted(Comparator.comparing(Field::getName))
                                    .collect(Collectors.toList());

for (Field field : requiredFields) {
    writeField(entry, out, field, indentation);
}

// Then optional fields
List<Field> optionalFields = type.get() BibEntryType
                                    .getOptionalFields() Set<BibField>
                                    .stream() Stream<BibField>
                                    .map(BibField::getField) Stream<Field>
                                    .sorted(Comparator.comparing(Field::getName))
                                    .collect(Collectors.toList());
```

   - **Location**: src/main/java/org/jabref/logic/bibtex/BibEntryWriter.java
   - **Explanation**: The Message Chain code smell can be seen here when one must call several objects in order to get, for example, the list of *requiredFields*. This means that the client is dependent on navigation along the class structure and any changes in this relationships would require to modify the client.
   - **Refactoring Proposal**: This particular code smell can be fixed by delegating the task of returning the *requiredFields* and the *otionalFields* to the *BibEntryType* class.

2. CitationKeyGenerator

   - **Code Snippet**:

```
@   public static String removeUnwantedCharacters(String key, String unwantedCharacters) {
        String newKey = key.chars() IntStream
                            .filter(c -> unwantedCharacters.indexOf(c) == -1)
                            .filter(c -> !DISALLOWED_CHARACTERS.contains((char) c))
                            .collect(StringBuilder::new,
                                    StringBuilder::appendCodePoint, StringBuilder::append) StringBuilder
                            .toString();
```

   - **Location**: src/main/java/org/jabref/logic/citationkeypattern/CitationKeyGenerator.java

- **Explanation**: This class presents a code smell of Message Chains because the object that the client called is calling another object, which calls a new object, creating a large sequential call of different objects: a chain. In this code snipped, we can see that several objects are sequentially called in order to create the *newKey*.
- **Refactoring Proposal**: A solution to this code smell would be to delegate the task of returning the *newKey* to this class.

4.11 Shotgun Surgery

1. JabRefCLId

   - **Code Snippet**:

```
82      public boolean isFileExport() { return cl.hasOption("output"); }
85
86      public String getFileExport() { return cl.getOptionValue("output"); }
89
90      public boolean isBibtexImport() { return cl.hasOption("importBibtex"); }
93
94      public String getBibtexImport() { return cl.getOptionValue("importBibtex"); }
97
98      public boolean isFileImport() { return cl.hasOption("import"); }
101
102     public String getFileImport() { return cl.getOptionValue("import"); }
```

   - **Location**: src/main/java/org/jabref/cli/JabRefCLI.java
   - **Explanation**:The Shotgun Surgery design pattern occurs when, by making a change in one place, a numerous amount of classes all over the design need to be changed in order to make that one change happen. In this case, if, for example, the string *importBibtex* was to be changed, one would have to manually change all the places where it exists, throughout the class.
   - **Refactoring Proposal**: One easy way to avoid this uncessary re-writing that is so prone to inadvertently producing issues is to create constants that are then used in the methods of the class; this way, if one wishes to change the value of a string, all that must be done is to change the value of that constant.

2. JabrefCLI

   - **Code Snippet**:

```
199     options.addOption(Option
200             .builder( option: "ib")
201             .longOpt("importBibtex")
202             .desc(String.format("%s: '%s'", Localization.lang( key: "Import BibTeX"), "-ib @article{entry}"))
203             .hasArg()
204             .argName("BIBTEXT_STRING")
205             .build());
206
207     options.addOption(Option
208             .builder( option: "o")
209             .longOpt("output")
210             .desc(String.format("%s: '%s'", Localization.lang( key: "Export an input to a file"), "-i db.bib -o db.htm,html"))
211             .hasArg()
212             .argName("FILE[,FORMAT]")
213             .build());
```

   - **Location**: src/main/java/org/fabref/cli/JabrefCLI.java

- **Explanation**: This is an example of a Shotgun Surgery code smell, because in order to change all appearances of "%s%s", one is required to manually change each and every one of the,
- **Refactoring Proposal**: One way to fix this particular code smell is to create a constant to which the "%s%s" will be assigned. Then, whenever we need to change the value of "%s%s", we can do so by changing the constant. Another way to deal with this would be to create a class that, for example, stores these objects and has other methods that return them as well. Then, the method *removeUnwantedCharacters* would call an instance of this class.

3. ExporterFactory

- **Code Snippet**:

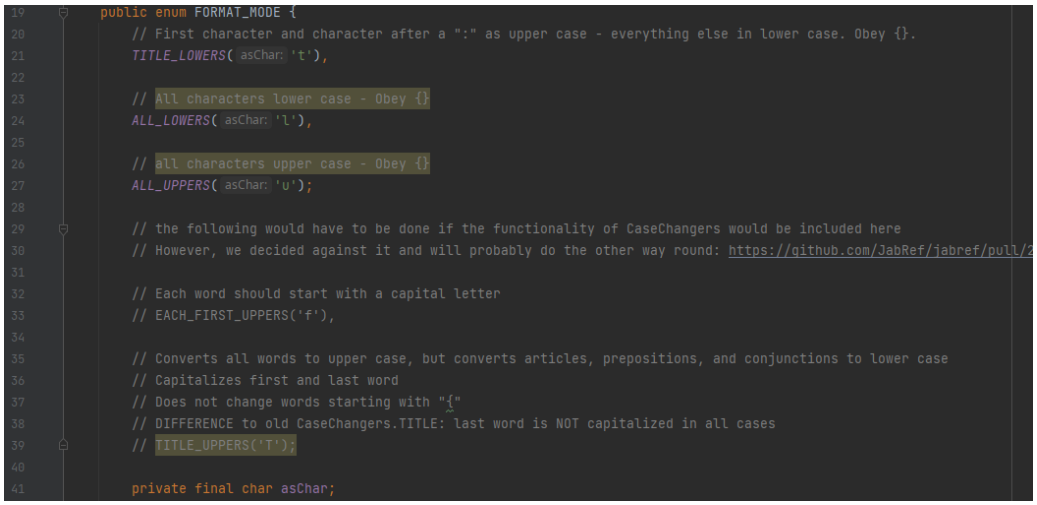
```
exporters.add(new TemplateExporter("HTML", "html", "html", null, StandardFileType.HTML, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter(Localization.lang("Simple HTML"), "simplehtml", "simplehtml", null, StandardFileType.HTML, layoutPreferences, save
exporters.add(new TemplateExporter("DocBook 5.1", "docbook5", "docbook5", null, StandardFileType.XML, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter("DocBook 4", "docbook4", "docbook4", null, StandardFileType.XML, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter("DIN 1505", "din1505", "din1505winword", "din1505", StandardFileType.RTF, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter("BibO RDF", "bibordf", "bibordf", null, StandardFileType.RDF, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter(Localization.lang("HTML table"), "tablerefs", "tablerefs", "tablerefs", StandardFileType.HTML, layoutPreferences,
exporters.add(new TemplateExporter(Localization.lang("HTML list"), "listrefs", "listrefs", "listrefs", StandardFileType.HTML, layoutPreferences, save
exporters.add(new TemplateExporter(Localization.lang("HTML table (with Abstract & BibTeX)"), "tablerefsabsbib", "tablerefsabsbib", "tablerefsabsbib",
exporters.add(new TemplateExporter("Harvard RTF", "harvard", "harvard", "harvard", StandardFileType.RTF, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter("ISO 690 RTF", "iso690rtf", "iso690RTF", "iso690rtf", StandardFileType.RTF, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter("ISO 690", "iso690txt", "iso690", "iso690txt", StandardFileType.TXT, layoutPreferences, savePreferences));
exporters.add(new TemplateExporter("Endnote", "endnote", "EndNote", "endnote", StandardFileType.TXT, layoutPreferences, savePreferences));
```

- **Location**: /src/main/java/org/jabref/logic/exporter/ExporterFactory.java
- **Explanation**: We can see how this class presents a Shotgun Surgery code smell: if the value of the string "harvard" has to be changed, 3 changes will have to be performed.
- **Refactoring Proposal**: This code smell could be fixed by using a constant that would represent strings that are repeated; that is, having a constant "SCHOOL" to which "harvard" would be assigned.

4.12 Too many comments

BibtexCaseChanger

- **Code Snippet**:



```
public enum FORMAT_MODE {
    // First character and character after a ":" as upper case - everything else in lower case. Obey {}.
    TITLE_LOWERS( asChar: 't'),

    // All characters lower case - Obey {}
    ALL_LOWERS( asChar: 'l'),

    // all characters upper case - Obey {}
    ALL_UPPERS( asChar: 'u');

    // the following would have to be done if the functionality of CaseChangers would be included here
    // However, we decided against it and will probably do the other way round: https://github.com/JabRef/jabref/pull/2

    // Each word should start with a capital letter
    // EACH_FIRST_UPPERS('f'),

    // Converts all words to upper case, but converts articles, prepositions, and conjunctions to lower case
    // Capitalizes first and last word
    // Does not change words starting with "{"
    // DIFFERENCE to old CaseChangers.TITLE: last word is NOT capitalized in all cases
    // TITLE_UPPERS('T');

    private final char asChar;
```

- **Location**: src/main/java/org/jabref/logic/bst/BibtexCaseChanger.java
- **Explanation**: The comments in this class take on a “reminder” nature, which means that if something needs to be changed, one would need to update the code - in this particular case, if the functionalities of CaseChangers. This could mean that the code of this class is way more complicated than it needs to be.
- **Refactoring Proposal**: A possible solution to fix this code smell would be to re-structure it for the sake of simplicity.

5 Metrics

Software Metrics constitute measurements of software characteristics such as code legibility, cohesian, coupling and dependencies between classes, uncertainty in achieving goals, etc.

Code Metrics are, for this reason, extremely important to analyse when evaluation the quality of software programs: low complexity, low coupling, high cohesion and maximum understandability are requirements for good coding practises.

In this section, we present the analysis of code metrics using the plugin "Metrics Reloaded" in IntelliJ. The chosen metrics sets were Complexity Metrics, Dependency Metrics, Lines of Code Metrics, MOOD Metrics and Martin Packaging Metrics.

Complexity Metrics

6 DEZEMBRO

Clara Sousa 58403

Complexity metrics are used to predict critical information about reliability and maintainability of software systems.

Parameters

1. Methods

When evaluating each method from a program, the structure of the assessment is:

$$\text{Location of method} + \text{CogC} + \text{ev(G)} + \text{iv(G)} + \text{v(G)}$$

Where:

- **CogC** stands for Cognitive Complexity, which measures of how difficult a unit of code is to intuitively understand. Good values of Cognitive Complexity are those smaller or equal to 15;
- **ev(G)** refers to Essential Cyclomatic Complexity, which tells how much complexity is left once we have removed a well-structured complexity. Good values of ev(G) are those smaller or equal to 3;
- **iv(G)** refers to Design Complexity: the measure of uncertainty in achieving the specified Functional Requirements. Good values of Design Complexity are those smaller or equal to 8;
- **v(G)** represents the Cyclomatic Complexity, which measures the number of possible independent paths through a method or function. Values of v(G) between 1 and 4 reveal a low complexity; values from 5 to 7 show medium complexity; from 8 to 10 the complexity is refered to as high, and finally equal or higher than 11 illustrates a very high complexity.

2. Classes

When evaluating classes from a program, the structure of the assessment is:

Location of class + OCavg + OCmax + WMC

Where:

- **OCavg** represents the average cyclomatic complexity of the non-abstract methods in each class
- **OCmax** refers to the maximum cyclomatic complexity of the non-abstract methods in each class
- **WMC** calculates the total cyclomatic complexity of the methods in each class

3. Packages/Modules

When evaluating packages or modules from a program, the structure of the assessment is:

Location of package/module + v(G)avg + v(G)tot

Where:

- **v(G)avg** represents the average cyclomatic complexity in each package/module
- **v(G)tot** refers to the total cyclomatic complexity in each package/module

Analysis

Cognitive Complexity

The vast majority of methods and classes present a Cognitive Complexity below 15, which shows that, overall, the code is not very difficult to understand. In fact, when it comes to Cognitive Complexity, methods present an mode value of 0 and an average value of 1.70; as to classes, these present values of 1 and 1.90, respectively.

Trouble spots are all those that present a Cognitive Complexity score higher than 15; in specific, the method *refreshCiteMarkersInternal(List<BibDatabase>,OOBibStyle)* (in gui.openoffice.OOBibBase) and the class FieldNameLabel (in gui.fieldeditors) constitute the most problematic cases of legibility, with values of 189 and 49, respectively.

Design Complexity

Regarding the Design Complexity, most methods present rather positive-looking values: the vast majority has a score of 1 and the overall average of methods corresponds to 1.88. These are extremely low values. Yet, there are still many methods with scores between 8 and 20, which is more worrisome.

All the trouble spots correspond to the methods with Design Complexity values above 8, and the most concerning on is *importDatabase(BufferedReader)* (in logic.importer.fileformat.RisImporter), with a score of 102.

Cyclomatic Complexities

Most methods present acceptable, although not brilliant values when it comes to Essential Cyclomatic Complexity and Cyclomatic Complexity, with average scores of 1.32 and 2.15, respectively. As to what regards the first complexity, the most common score is 1, altough there are still too many methods scoring above 3, and the average score is 1.32; as to the last, the most common score is also 1 and the average is 2.15, which reveals very low levels of complexity, even though there are many other methods scoring 8 or more, which illustrates higher values of complexity

Regarding Essential Cyclomatic Complexity, trouble spots are all the methods with scores higher than 3; as to Cyclomatic Complexity, trouble spots can be found in methods scoring higher than 4, even though the truly concerning cases are those higher than 8. Two of the most troubling spots are *getDescription(Field)* (in gui.fieldeditors.FieldNameLabel) and *importDatabase(BufferedReader)* (in logic.importer.fileformat.RisImporter), scoring 96 in Essential Cyclomatic Complexity and 102 in Cyclomatic Complexity, respectively.

When looking at the big picture provided by the analysis, one could expect to see some cases of code smells like the Long Method and the Large Classe – and the team did indeed identify Long Methods (in classes *ExternalFileTypes.java* and *VM)* and a Large Classes (in *BracketedPattern)* . In fact, the JabRef codebase does present some severe issues when it comes to complexity, but most methods and classes are actually within the acceptable, conventioned range values.

Dependency Metrics

6 DEZEMBRO

Paula Lopes 58655

Introduction

Like we describe code as high-level/low-level, dependencies consist on low/high levels (classes,modules, packages..) depending on each other.

Generally, these occur within inheritance hierarchy (a subtype depends on its' supertype). However, when a supertype depends on its' dependents this creates a cyclic dependency. Having a dependency from a bottom layer to a top layer will make the system less maintainable and error-prone. Therefore, each layer should depend on layers below them and never above. Otherwise, testing the system will be more difficult, since the layers are tightly coupled and mutually dependent of each other, they cannot be tested separately.

Analysis

1. Class Dependencies (CD)

   When evaluating each class from a program, the structure of the assessment is:

   Location of Class + Cyclic + Dcy + Dcy* + Dpt + Dpt* + PDcy + PDpt

   Where:

   - **Cyclic** (*Number of cyclic class dependencies*) measures, for each class X, the number of classes X depends on and depend on X
   - **Dcy** (*Number of dependencies*) measures, for each class X, the number of classes X depends on
   - **Dcy*** (*Number of transitive dependencies*) measures, for each class X, the number of classes X depends on, directly or indirectly
   - **Dpt** (*Number of dependants*) measures, for each class X, the number of classes that depend on X
   - **Dpt*** (*Number of transitive dependants*) measures, for each class X, the number of classes that depend on X, directly or indirectly
   - **PDcy** (*Number of package dependencies*) measures, for each class X, the number of packages X depends on
   - **PDpt** (*Number of package dependants*) measures, for each class X, the number of packages that depend on X

2. Interface Dependencies (ID)

When evaluating each Interface from a program, the structure of the assessment is:

Location of Interface + Cyclic + Dcy + Dcy* + Dpt + Dpt* + PDcy + PDpt

Where:

- **Cyclic** (*Number of cyclic interface dependencies*) measures, for each Interface X, the number of interfaces X depends on and depend on X
- **Dcy** (*Number of dependencies*) measures, for each Interface X, the number of interfaces X depends on
- **Dcy*** (*Number of transitive dependencies*) measures, for each Interface X, the number of interfaces X depends on, directly or indirectly
- **Dpt** (*Number of dependants*) measures, for each Interface X, the number of interfaces that depend on X
- **Dpt*** (*Number of transitive dependants*) measures, for each Interface X, the number of interfaces that depend on X, directly or indirectly
- **PDcy** (*Number of package dependencies*) measures, for each Interface X, the number of packages X depends on
- **PDpt** (*Number of package dependants*) measures, for each Interface X, the number of packages that depend on X

3. Package Dependencies (PD)

When evaluating each Package from a program, the structure of the assessment is:

Location of Package + Cyclic + PDcy + PDpt + PDpt*

Where:

- **Cyclic** (*Number of cyclic package dependencies*) measures, for each Package X, the number of packages X depends on and depend on X
- **PDcy** (*Number of package dependencies*) measures, for each Package X, the number of packages X depends on
- **PDpt** (*Number of package dependants*) measures, for each Package X, the number of packages that depend on X
- **PDpt*** (*Number of transitive package dependants)* measures, for each Package X, the number of packages that depend on X, directly and indirectly

Conclusion

From the graphics delivered regarding dependency metrics, we can conclude that any class, interface, package with a number of cylic dependencies over the average (respectively 420, 397, 112) is considered an outlier.
In jabref the vast majority of classes present trouble spots. There are around 783 classes and interfaces along with 153 packages that surpass the average. Hence, we can conclude that there are many classes, interfaces and packages that ought to be improved to decrease the existing dependencies.

In JabRef Classes, these dependencies occur mainly because the number of classes most classes depend on directly or indirectly (Dcy*) overpass the average of 769, with a mode of 1333. As a result, the number of classes that depend on a specific class(Dpt*), generally also surpass the average of 1015, with a mode of 1121. A similar conclusion can be made for the Interfaces of JabRef.

As for the Packages the program has, the dependencies shown occur because the number of packages that depend on most packages directly or indirectly (PDpt*) exceed the average of 152, with a mode of 162.

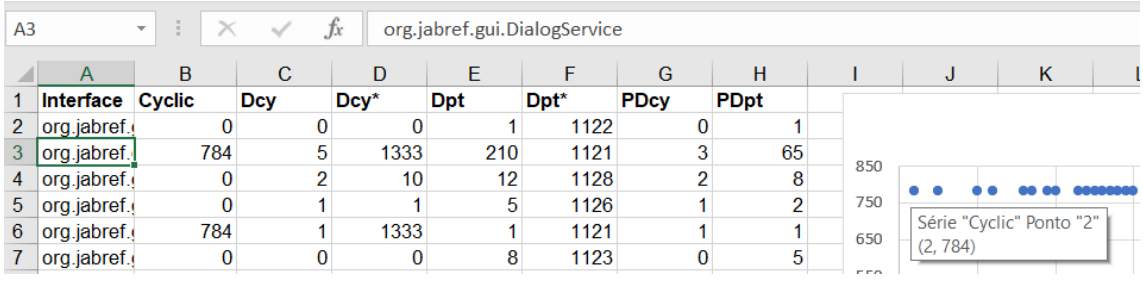
This is not surprising for a project of this dimension, since the use of structural patterns commonly introduce more complexity (Adaptor, Proxy, Composite, Decorator). However, this choice also brings code smells such as Inappropriate Intimacy to play.

Trouble spots

To find a trouble spot, we can simply click on a dot in the boxplot graphics with the title ‘Cyclic’ on a critical zone (where axis of y is greater than the average value) and aftherwards we are shown the corresponding class/interface/package in the form of (row, value). On the row after the presented row, we can find the location of the trouble.

Example:

Lets say we wanted to find a class considered as a trouble spot. If we click on the first dot above the average on the cyclic graph, we are met with (2,784), which means on the row 3 the class org.jabref.gui.DialogService with a value of 784 cyclic dependencies is a trouble spot.



	A	B	C	D	E	F	G	H
1	Interface	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
2	org.jabref.(	0	0	0	1	1122	0	1
3	org.jabref.(	784	5	1333	210	1121	3	65
4	org.jabref.(	0	2	10	12	1128	2	8
5	org.jabref.(	0	1	1	5	1126	1	2
6	org.jabref.(	784	1	1333	1	1121	1	1
7	org.jabref.(	0	0	0	8	1123	0	5

Code smells

Eventhough we didn’t officially identify the code smell of Inappropriate Intimacy, some of the code smells indentified often would lead to this particular code smell. Such as the Long Parameter List in class ArgumentProcessor from package jabref.cli.

Lines Of Code Metrics

6 DEZEMBRO

Pedro Reis 58751

Lines of code Metrics

Introduction

Lines of Code metrics shows how many lines of code there is in your project, interface, module, package, class, method or by file type. This metric simply counts all lines of code or only commented lines of code, Javadoc lines of code, non-comment lines of code, etc. However, this metrics do not take into account the intelligence content and the layout of content and because of that, the LOC metrics are better to monitor the size of the code units and should be avoided when it comes to software estimation because there are better options for it. For example, productivity cannot be measured by LOC because 100 lines of code of one complex algorithm may take the same time of work to do as a system of 10000 lines. Also, this metrics isn't linear because when using software industry productivity averages a 10000 LOC system would require 13.5 staff months. If effort increased linearly, a 100000 LOC system would require 135 staff months. But it actually requires 170 staff months and the error size of 35 staff months is really significant because you had a budget for only 135 staff months, and you will still need more 35 staff months to finish the project.

Parameters

1. Method – When evaluating each method from a program, the structure of the assessment is: Location of method + CLOC + JLOC + LOC + NCLOC + RLOC
   Where:
   - CLOC – Counts the number of Commented lines of code in the method.
   - JLOC – Counts the Javadoc lines of code in the method.
   - LOC – Counts the Lines of code in the method.
   - NCLOC – Counts the Non-comment lines of code in the method.
   - RLOC – Calculates the percentage of lines of code of the method relative to the lines of code of the class where it belongs to.

2. Class – When evaluating each class from a program, the structure of the assessment is: Location of class + CLOC + JLOC + LOC
   Where:
   - CLOC – Counts the number of Commented lines of code in the class.
   - JLOC – Counts the Javadoc lines of code in the class.
   - LOC – Counts the Lines of code in the class.

3. Interface – When evaluating each interface from a program, the structure of the assessment is: Location of interface + CLOC + JLOC + LOC + NCLOC
   Where:
   - CLOC – Counts the number of Commented lines of code in the interface.
   - JLOC – Counts the Javadoc lines of code in the interface.
   - LOC – Counts the Lines of code in the interface.
   - NCLOC – Counts the Non-comment lines of code in the interface.

4. Package – When evaluating each package from a program, the structure of the assessment is: Location of package + CLOC + CLOC (rec) + JLOC + JLOC (rec) + LOC, LOC (rec) + LOCp + LOCp (rec) + LOCt + LOCt (rec) + NCLOC + NCLOCp + NCLOCp (rec) + NCLOCt
   Where:
   - CLOC – Counts the number of Commented lines of code in the package.
   - CLOC (rec) - Counts the number of Commented lines of code recursively in the package.
   - JLOC – Counts the Javadoc lines of code in the package.
   - JLOC (rec) - Counts the number of Javadoc lines of code recursively in the package.
   - LOC – Counts the Lines of code in the package.
   - LOC (rec) - Counts the number of lines of code recursively in the package.
   - LOCp – Counts the number of lines of product code in the package.
   - LOCp (rec) - Counts the number of lines of product code recursively in the package.
   - LOCt – Counts the number of lines of test code in the package.
   - LOCt (rec) – Counts the number of lines of test code recursively in the package.
   - NCLOC – Counts the Non-comment lines of code in the package.
   - NCLOCp – Counts the Non-comment lines of product code in the package.
   - NCLOCp (rec) – Counts the Non-comment lines of product code recursively in the package.
   - NCLOCt – Counts the Non-comment lines of test code in the package.

5. Module – When evaluating each module from a program, the structure of the assessment is: Location of module + JLOC + L (CSS) + L (Groovy) + L (HTML) + L(J) + L(JS) + L(JSP) + L(KT) + L(XML) + LOC + LOCp + LOCt + NCLOC + NCLOCp + NCLOCt
   Where:
   - JLOC – Counts the Javadoc lines of code in the module.
   - L(CSS) – Counts the Lines of CSS in the module.
   - L(Groovy) - Counts the number of lines of Groovy in the module.
   - L(HTML) – Counts the number of lines of HTML in the module.
   - L(J) – Counts the number of lines of Java in the module.
   - L(JS) – Counts the number of lines of JavaScript in the module.
   - L(JSP) – Counts the number of lines of JSP in the module.
   - L(KT) – Counts the number of lines of Kotlin in the module.
   - L(XML) – Counts the number of lines of XML in the module.

- LOC – Counts the number of lines of code in the module.
- LOCp – Counts the number of lines of product code in the module.
- LOCt – Counts the number of lines of test code in the module.
- NCLOC – Counts the Non-comment lines of code in the module.
- NCLOCp – Counts the Non-comment lines of product code in the package.
- NCLOCt – Counts the Non-comment lines of test code in the package.

6. File type – When evaluating each file type of a program, the structure of the assessment is: Location of package + LOC + NCLOC
   Where:
   - LOC – Counts the Lines of code of the file type.
   - NCLOC – Counts the Non-comment lines of code of the file type.

7. Project – When evaluating the project of a program, the structure of the assessment is: project + CLOC + JLOC + L (CSS) + L (Groovy) + L (HTML) + L(J) + L(JS) + L(JSP) + L(KT) + L(XML) + LOC + LOCp + LOCt + NCLOC + NCLOCp + NCLOCt
   Where:
   - CLOC – Counts the number of Commented lines of code in the project.
   - JLOC – Counts the Javadoc lines of code in the project.
   - L(CSS) – Counts the Lines of CSS in the project.
   - L(Groovy) - Counts the number of lines of Groovy in the project.
   - L(HTML) – Counts the number of lines of HTML in the project.
   - L(J) – Counts the number of lines of Java in the project.
   - L(JS) – Counts the number of lines of JavaScript in the project.
   - L(JSP) – Counts the number of lines of JSP in the project.
   - L(KT) – Counts the number of lines of Kotlin in the project.
   - L(XML) – Counts the number of lines of XML in the project.
   - LOC – Counts the number of lines of code in the project.
   - LOCp – Counts the number of lines of product code in the project.
   - LOCt – Counts the number of lines of test code in the project.
   - NCLOC – Counts the Non-comment lines of code in the project.
   - NCLOCp – Counts the Non-comment lines of product code in the project.
   - NCLOCt – Counts the Non-comment lines of test code in the project.

Analysis

Method

The LOC metrics presents an average of lines of code for the methods of 10,43 which means the method size of the program perfectly fits between the recommended range that should be 4 to 40 lines. The commented lines average is 1,3 which means 12,5% of the methods are commented lines and this ratio could be improved to 30% for example, in order to have better explained methods. The relative lines of code average is 5% which

means on average each method corresponds to 5% of the lines of code of a class. This ratio seems good but there is not any information online to confirm this.

Despite having a good average, there are some trouble spots that we can identify and there's one particular method which stands out from the rest for having 429 lines of code which is org.jabref.logic.bst.VM.VM(CommonTree) and it has been identified as a long method code smell. Regarding the RLOC metric, it is possible to identify some methods that occupy over 90% of a class, for example org.jabref.model.strings.UnicodeToReadableCharMap.UnicodeToReadableCharMap() which has a 99% RLOC but this one is not a code smell because it is a map constructor. There are others methods like org.jabref.logic.layout.format.RemoveLatexCommandsFormatter.format(String) that problematic because they are a long method code smell and they should be split.

Class

The LOC metrics presents an average of lines of code for the classes of 75,84 which means the class size of the program perfectly fits between the recommended range that should be 4 to 400 lines. The commented lines average is 13,21 which means 17,5% of the classes are commented lines and this ratio could be improved to fit inside the perfect range which is 30 to 75% of commented lines per class because such a lower percentage of comments per class could mean the class is very trivial or poorly explained.

Despite having a good average, there are some trouble spots that we can identify and there's one particular class which stands out from the rest for having 2291 lines of code which is org.jabref.preferences.JabRefPreferences and it has been identified as a divergent class code smell. There are other large classes for example the org.jabref.logic.citationkeypattern.BracketedPattern that we have also identified as a large class code smell and org.jabref.logic.util.strings.HTMLUnicodeConversionMaps that with 824 commented code lines it is more like a document.

Interface

The LOC metrics presents an average of lines of code for the interfaces of 22.03, I didn't find any information regarding the ideal range of an interface, but the size of the interfaces seems good. The commented lines average is 12,11 which means 55% of the interface are commented lines and this ratio seems to be perfect for an interface since it means everything is well explained.

For the trouble spots, there is one interface org.jabref.gui.DialogService which may have too many comments since it is composed by 83% of commented lines of code but since this is an interface it may not be worrisome but could mean that this interface is way too complex. There is also an interface org.jabref.preferences.PreferencesService with 164 lines of code and only 49 commented lines of code and this interface is then implemented by a class called JabPreferences which is a divergent class code smell and it is possible to predict this type of problems due to the size of the interface.

Package

The LOC metrics presents an average of lines of code for the packages of 821,74 and even though the metrics reloaded does not show the averages for the commented code lines counted recursively for each package, I did the average manually and ended up with an average of 166,3 and this means that we have around 20% of commented lines in all the packages.

Module

Not much to analyse here, only that the biggest modules are JabRef and JabRef.buildSrc.main and that the JabRef.buildSrc.main is where most of java code lines are.

FileType

This file type LOC metrics is a good way to see how many lines of code of each file type is the program composed by. Most of the program is made by Files supported via TextMate bundles lines of code, with a significant amount of JAVA lines of code too and some other minor filetypes like XML and others.

Project

The LOC metrics of the whole project allow us to see the dimension of this project, and with 1113118 lines of code and 171743 lines of product code I can conclude this is a big size project. Most of the lines of product code are made in JAVA and we also have a significant amount of XML lines. There is 21373 comment lines, and this corresponds to 12,44% of commented lines for all the lines of product code.

Conclusion

In conclusion, the most important lines of code metrics to look at are the method, class and interface, and thanks to this metrics we have been able to identify some classes and methods that were too big and needed to be split but also classes and methods that lacked comments or had simply too many comments. Despite this, the overall project is great and fits perfectly inside the recommended range for the classes and methods. LOC metrics are a simple metric and should be used primarily to monitor the size of the code units and to identify the code smells that are related to the size of the code units.

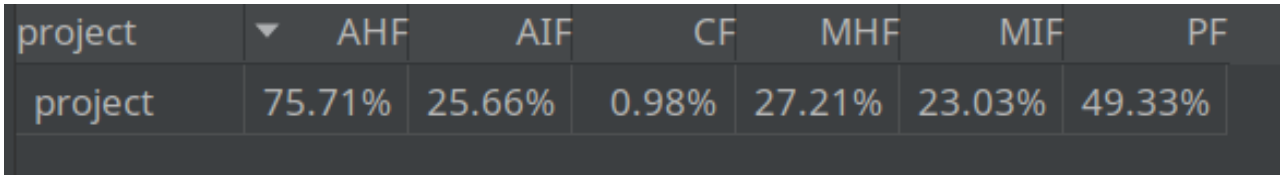
MOOD Metrics

6 DEZEMBRO

Ricardo Pereira 57912

MOOD Metrics

This metrics are used to measure the quality of some characteristics of OO(object oriented) programs, and this are: AHF, AIF, CF, MHF, MIF and PF.



project	AHF	AIF	CF	MHF	MIF	PF
project	75.71%	25.66%	0.98%	27.21%	23.03%	49.33%

- **AHF(Atribute Hiding Factor)**: All attributes should be hidden, only being accessed by the corresponding class methods. Sum of all hidden attributes (in all classes) divided by the sum of visible attributes and hidden attributes(in all classes). Should be 100%. In these case it is 75.71%.

  Importance: Low(1)
- **AIF(Attribute Inheritance Factor)**: Should be used but not too extensively. Sum of the inherited attributes divided by the sum of defined attributes and inherited attributes. In this case it was 25.66%.

  Importance: Low(1)
- **CF(Coupling Factor)**: Measure of coupling between classes. It is desirable that the classes communicate with as few other classes as possible, so the lower the better. In this project it was 0.98%.

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$

  TC = Total number of Classes.
  is_client(Cc, Cs) = if((Cc => Cs) and (Cc != Cs)) return 1 else return 0

  Importance: High(3)
- **MHF(Method Hiding Factor)**: All methods should be hidden, only being accessed by the corresponding classes. Sum of all hidden methods (in all classes) divided by the sum of visible methods and hidden methods(in all classes)(should be 100%). In these case it is 27.21%.

  Importance: Medium(2)
- **MIF(Method Inheritance Factor)**: Should be used but not too extensively. Sum of the inherited methods divided by the sum of defined methods and inherited methods. In this case it was 23.03%.

  Importance: Medium(2)

- **PF(Polymorphism Factor)**: Represents the number of possible different polymorphic situations. Should be used but not too extensively. In this project it was 49.33%.

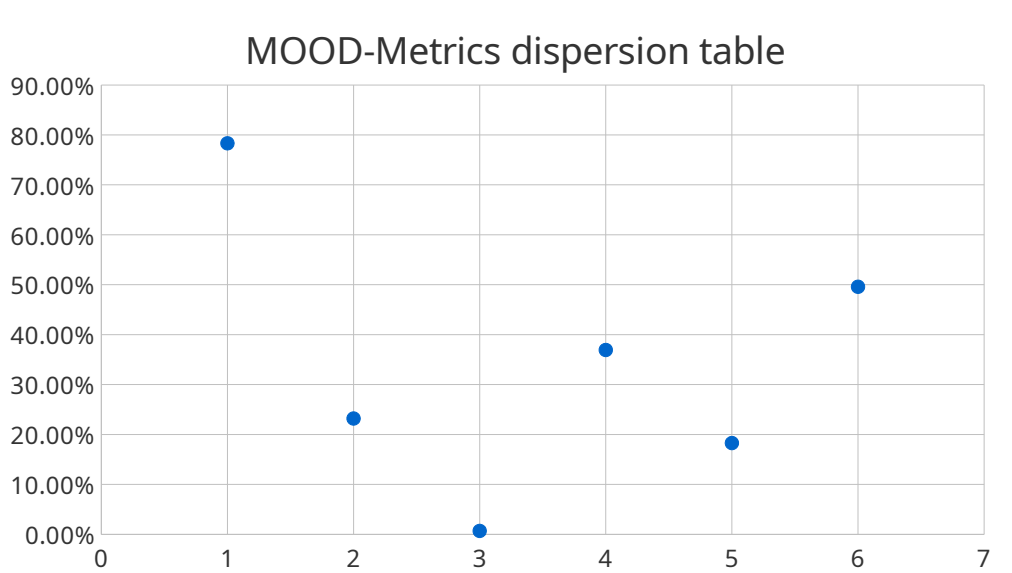
$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

Mo = methods that override other methods
Mn = new methods
DC = classes that derivate from another class
TC = total number of classes

Importance: Medium(2)

Factor Evaluation:

Factor	MIN	MAX	MIN Tolerance	MAX Tolerance
AHF	75.2%	100%	67.7%	100%
AIF	52.7%	66.3%	37.4%	75.7%
CF	0%	11.2%	0%	24.3%
MHF	12.7%	21.8%	9.5%	36.9%
MIF	66.4%	78.5%	60.9%	84.4%
PF	2.7%	9.6%	1.7%	15.1%



- **AHF:** 75.71% is a good value.
- **AIF:** 25.66% is a really bad value, even with the minimum tolerance the value should be higher, so the project should have a higher attribute inheritance factor, but not too high.
- **CF:** 0.98% is perfect, the lower the better.
- **MHF:** This value should be lower, it's a little bit too high. To fix we should hide more methods, to do this we could make some of them private.
- **MIF:** 23.03% is a really bad value, even with the minimum tolerance the value should be way higher, so the project should have a higher method inheritance factor.
- **PF:** 49.33% is a really bad factor, even with the maximum tolerance the value should be lower. To fix this we could create less superclasses and subclasses.

Conclusion:

We can conclude with the evaluation that we should really fix the AIF, MIF and PF factors, the project has too much polymorphism but the inheritance(methods and attributes) is too low, this could be due the **Large Class** code smells that we found where the programmers mostly used a large class instead of using subclasses, making the AIF/MIF factors too small. This could be due other code smells that we didn't identify, the **Speculative Generality** code smell, where the programmers added more than the necessary subclasses thinking on the future and ended not using them as much as they tought they would, or the **Switch Statements** code smell, where they could have used the subclasses more instead of the switch statements. The MHF factor it's a little too high but not too overwhelming for now. The AHF, CF have both good values.

Martin Packaging Metrics

6 DEZEMBRO

Rita Silva 57960

Martin packaging metrics

This metric is used to measure interdependence between classes.

It is characterized by:

- **Afferent couplings(Ca)**

  Measures the total number of external classes coupled to classes of a package because of incoming coupling.

  All classes are counted only once.

  If the package does not contain any class or external classes do not use any of the classes of the package, then the value of Ca is zero.

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

  In the graph it's seen that the number of incoming dependencies isn't unexpected, since the hight number suggests high stability.

- **Efferent couplings (Ce)**

  Number of classes within the package that depend on external classes.

  That is Afferent Couplings are Arriving references, Efferent Coupling are Exiting or Escaping references.

  References to test classes and library classes are not included.

  If the classes in the same package do not use any external class linkage, then the value of Ce is zero.

  The high value of the metric indicates instability of a package, that meaning that any change change in one of the numerous external classes can cause the need for changes to the package.

  Ce should be between 0 and 20.

  With the graph we can see that there are a considerable number of classes that do depend on external ones to their package in comparation with the values of the graph related to afferent coupling.

- **Abstractness(A)**

  Ration between number of abstract classes (interfaces included) in the package being analysed and the total number of classes in that package.

  This ranges from 0 to 1. 0 indicates a completely concrete package while 1 na abstract one.

  In the graph calculated we can see that there are more packages that are concrete than there are abstract ones.

- **Instability (I)**

  Ration between Ce and Ce + Ca.

  This indicates the flexibility of the package in relation to change.

  Ranges from 0 to 1, where 0 indicates a completely stable package while 1 a unstable one.

  From the graph we can conclude that there are a large number of unstable packages, going over 60 even. This means that all those classes have a reason to change, they are very unstable and that can be caused by the dependencies that the classes have between one another, in other words, if one class changes those that depend on it also need to change, therefore the packages are unstable.

- **Distance from main sequence(D)**

  Calculated by the formula $|A + I - 1|$.

  This indicates the balance between A and I.

  This metric has a range of [0,1], where the closer D is to 0, the better.

  A=0 and I=0 is a highly stable and concrete category, as such it is not desirable because it is rigid. It cannot be extended because it is not abstract. And it is very difficult to change because of its stability. A=1 and I=1 is also undesirable (perhaps impossible) because it is maximally abstract and yet has no dependents. It, too, is rigid because the abstractions are impossible to extend.

  In this case we can see that althought we see a lot of packages having D closer to zero (or even zero), there are still many closer to 1 (or even reaching one). This means that all of those need to be reexamined and reconstructed.

Conclusion

package	A	Ca	Ce	D	I
Average	0,06	312,90	317,68	0,32	0,51

Overall this is a “good” pattern for packages, it would be better if we solved the packages with the D very close to 1 (from 0.6 to 1) althought in average its value is closer to 0, we should also minimize the value of Ce, it’s average is 317,68 (way above 20), this can be achieved if we dealt with the two shotgun surgery code smell we found (because this kind of code smell prevents change and therefore has a higher instability), and the smells known as message chains too (this code smell contrinutes to excess coupling between classes).

JabRef Funcionalities Use Case Diagram

6 DEZEMBRO

Scrum Team

UNREGISTERED