Merged Doc

December

2021

Débora Dias - 57537
Gonçalo Prata - 52912
João Silva - 58316
Mariana Maximiano - 57921
Miguel Pauleta - 41999

Índice

Débora Dias

Code Metrics

Complexity metrics

Method	CogC	ev(G)	iv(G)	v(G)
org.jabref.cli.ArgumentProcessor.fetch(String)	9	4	6	6
org.jabref.cli.ArgumentProcessor.importAndOpenFiles()	28	1	16	16
org.jabref.cli.ArgumentProcessor.processArguments()	39	5	33	33

Caption:
CogC – Cognitive complexity
ev(G) – Essential cyclomatic complexity
iv(G) – Design complexity
v(G) – Cyclomatic complexity

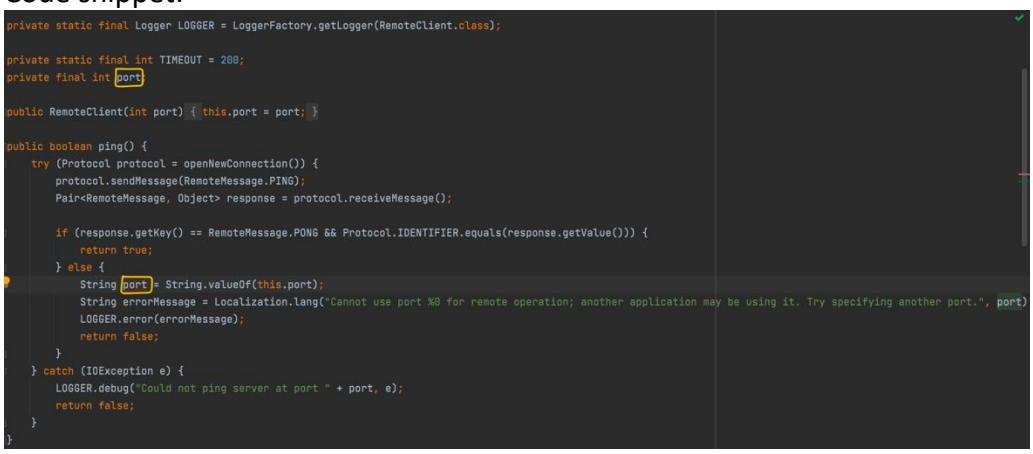
Analysis of the collected metrics:

As we may observe, the first method has the lowest values, having a Cognitive complexity of 9, an Essential cyclomatic complexity of 4, a Design complexity of 6 and a Cyclomatic complexity of 6.
Cognitive Complexity is a measure of how difficult a unit of code is to intuitively understand. In the last two methods, the cognitive complexity is higher which means that the lines of code are more difficult to read. This could be a trouble spot because the code needs to be easy to read not just for who made the code itself but for other people to read it.
The second method has the lowest value for Essential cyclomatic complexity (1), Essential complexity is the measure of the degree to which a module contains unstructured constructs.
The numbers of Design complexity, a measure of the module's decision structure as it relates to calls to other modules, and Cyclomatic complexity, measures the number of linearly independent paths through a given program, are the same in each method.

Regarding the identified code smells, none of them reflect on this metrics.

Code Smells

Code smell 1- Local variables should not shadow class fields
Code snippet:

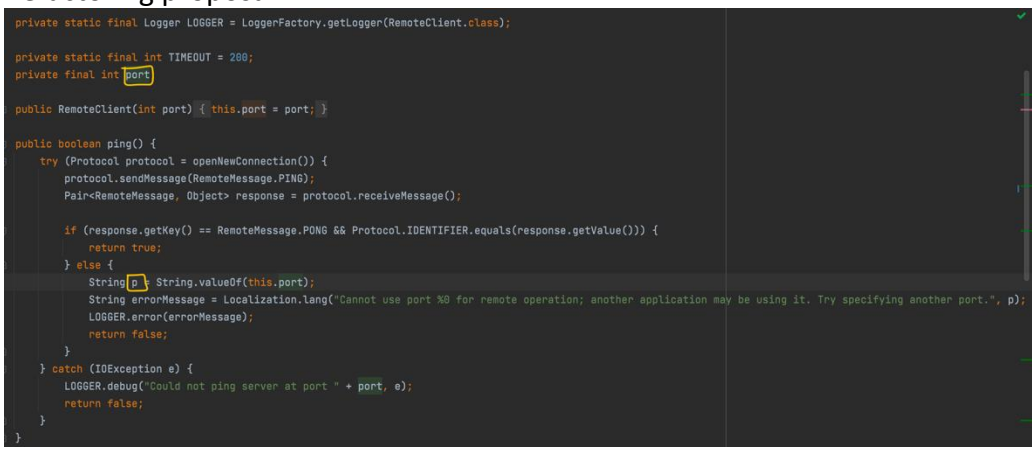


```
private static final Logger LOGGER = LoggerFactory.getLogger(RemoteClient.class);

private static final int TIMEOUT = 200;
private final int port;

public RemoteClient(int port) { this.port = port; }

public boolean ping() {
    try (Protocol protocol = openNewConnection()) {
        protocol.sendMessage(RemoteMessage.PING);
        Pair<RemoteMessage, Object> response = protocol.receiveMessage();

        if (response.getKey() == RemoteMessage.PONG && Protocol.IDENTIFIER.equals(response.getValue())) {
            return true;
        } else {
            String port = String.valueOf(this.port);
            String errorMessage = Localization.lang("Cannot use port %0 for remote operation; another application may be using it. Try specifying another port.", port);
            LOGGER.error(errorMessage);
            return false;
        }
    } catch (IOException e) {
        LOGGER.debug("Could not ping server at port " + port, e);
        return false;
    }
}
```

Location of the code:
src/main/java/jabref/logic/remote/client/RemoteClient.java

In this specific piece of code there is overriding or shadowing of the variable port declared in an outer scope. This can make the code difficult to read and can have a huge impact on the maintainability of the code or it could even lead to bugs since maintainers might be confused and use the wrong variable.
This code smell can easily be fixed by renaming the local variable so that there is no overriding or shadowing of the variable declared before.

Refactoring proposal:



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

private static final int TIMEOUT = 200;
private final int port;

public RemoteClient(int port) { this.port = port; }

public boolean ping() {
    try (Protocol protocol = openNewConnection()) {
        protocol.sendMessage(RemoteMessage.PING);
        Pair<RemoteMessage, Object> response = protocol.receiveMessage();

        if (response.getKey() == RemoteMessage.PONG && Protocol.IDENTIFIER.equals(response.getValue())) {
            return true;
        } else {
            String p = String.valueOf(this.port);
            String errorMessage = Localization.lang("Cannot use port %0 for remote operation; another application may be using it. Try specifying another port.", p);
            LOGGER.error(errorMessage);
            return false;
        }
    } catch (IOException e) {
        LOGGER.debug("Could not ping server at port " + port, e);
        return false;
    }
}
```

Code smell 2- Pattern Matching for "instanceof" operator should be used instead of simple "instanceof" + cast

Code snippet:

```
private void handleMessage(Protocol protocol, RemoteMessage type, Object argument) throws IOException {
    switch (type) {
        case PING:
            protocol.sendMessage(RemoteMessage.PONG, Protocol.IDENTIFIER);
            break;
        case SEND_COMMAND_LINE_ARGUMENTS:
            if (argument instanceof String[]) {
                messageHandler.handleCommandLineArguments((String[]) argument, preferencesService);
                protocol.sendMessage(RemoteMessage.OK);
            } else {
                throw new IOException("Argument for 'SEND_COMMAND_LINE_ARGUMENTS' is not of type String[]. Got " + argument);
            }
            break;
        default:
            throw new IOException("Unhandled message to server " + type);
    }
}
```

Location of the code:
src/main/java/jabref/logic/remote/server/RemoteListenerServer.java

A java feature "Pattern matching for instanceof" is present in this specific piece of code. This feature replaces the previous technique that consisted in 3 operations: check the variable type, cast it, and assign the casted value to the new variable.
This rule raises an issue when an instanceof check followed by a cast and an assignment could be replaced by pattern matching.
This code smell can be fixed by using the declared variable instead of an instanceof check followed by a cast and an assignment.

Refactoring proposal:

```
private void handleMessage(Protocol protocol, RemoteMessage type, Object argument) throws IOException {
    switch (type) {
        case PING:
            protocol.sendMessage(RemoteMessage.PONG, Protocol.IDENTIFIER);
            break;
        case SEND_COMMAND_LINE_ARGUMENTS:
            if (argument instanceof String[] s) {
                messageHandler.handleCommandLineArguments(s, preferencesService);
                protocol.sendMessage(RemoteMessage.OK);
            } else {
                throw new IOException("Argument for 'SEND_COMMAND_LINE_ARGUMENTS' is not of type String[]. Got " + argument);
            }
            break;
        default:
            throw new IOException("Unhandled message to server " + type);
    }
}
```

Code smell 3- Unused "private" fields should be removed
Code snippet:

```
 */
public class Protocol implements AutoCloseable {

    public static final String IDENTIFIER = "jabref";

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

    private final Socket socket;
    private final ObjectOutputStream out;
    private final ObjectInputStream in;

    public Protocol(Socket socket) throws IOException {
        this.socket = socket;
        this.out = new ObjectOutputStream(socket.getOutputStream());
        this.in = new ObjectInputStream(socket.getInputStream());
    }
```

Location of the code:
src/main/java/jabref/logic/remote/shared/Protocol.java

In this specific piece of code there is what can be considered dead code. Removing dead code, in this case a private field that is declared but never used, will improve maintainability and readability since maintainers won't have to wonder what the variable is used for. This code smell can easily be fixed by removing the dead code.

Refactoring proposal:

```
public class Protocol implements AutoCloseable {

    public static final String IDENTIFIER = "jabref";

    private final Socket socket;
    private final ObjectOutputStream out;
    private final ObjectInputStream in;

    public Protocol(Socket socket) throws IOException {
        this.socket = socket;
        this.out = new ObjectOutputStream(socket.getOutputStream());
        this.in = new ObjectInputStream(socket.getInputStream());
    }
```

Design Patterns

Factory:

Code:

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

import ...

public class SpecialFieldMenuItemFactory {
    public static MenuItem getSpecialFieldSingleItem(SpecialField field,
                                                     ActionFactory factory,
                                                     JabRefFrame frame,
                                                     DialogService dialogService,
                                                     PreferencesService preferencesService,
                                                     UndoManager undoManager,
                                                     StateManager stateManager) {
        SpecialFieldValueViewModel specialField = new SpecialFieldValueViewModel(field.getValues().get(0));

        return factory.createMenuItem(specialField.getAction(),
                new SpecialFieldViewModel(field, preferencesService, undoManager)
                        .getSpecialFieldAction(field.getValues().get(0), frame, dialogService, stateManager));
    }

    public static Menu createSpecialFieldMenu(SpecialField field,
                                              ActionFactory factory,
                                              JabRefFrame frame,
                                              DialogService dialogService,
                                              PreferencesService preferencesService,
                                              UndoManager undoManager,
                                              StateManager stateManager) {

        return createSpecialFieldMenu(field, factory, preferencesService, undoManager, specialField ->
                new SpecialFieldViewModel(field, preferencesService, undoManager)
                        .getSpecialFieldAction(specialField.getValue(), frame, dialogService, stateManager));
    }

    public static Menu createSpecialFieldMenu(SpecialField field,
                                              ActionFactory factory,
                                              PreferencesService preferencesService,
                                              UndoManager undoManager,
                                              Function<SpecialFieldValueViewModel, Command> commandFactory) {
        SpecialFieldViewModel viewModel = new SpecialFieldViewModel(field, preferencesService, undoManager);
        Menu menu = factory.createMenu(viewModel.getAction());

        for (SpecialFieldValue Value : field.getValues()) {
            SpecialFieldValueViewModel valueViewModel = new SpecialFieldValueViewModel(Value);
            menu.getItems().add(factory.createMenuItem(valueViewModel.getAction(), commandFactory.apply(valueViewModel)));
        }
        return menu;
    }
}
```

Location:
src/main/java/org/jabref/gui/specialfields/SpecialFieldMenuItemFactory.java

Reasoning:
Due to having optional fields, this implementation of the abstract method implements different specifications of the same object.
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Builder:

Code:

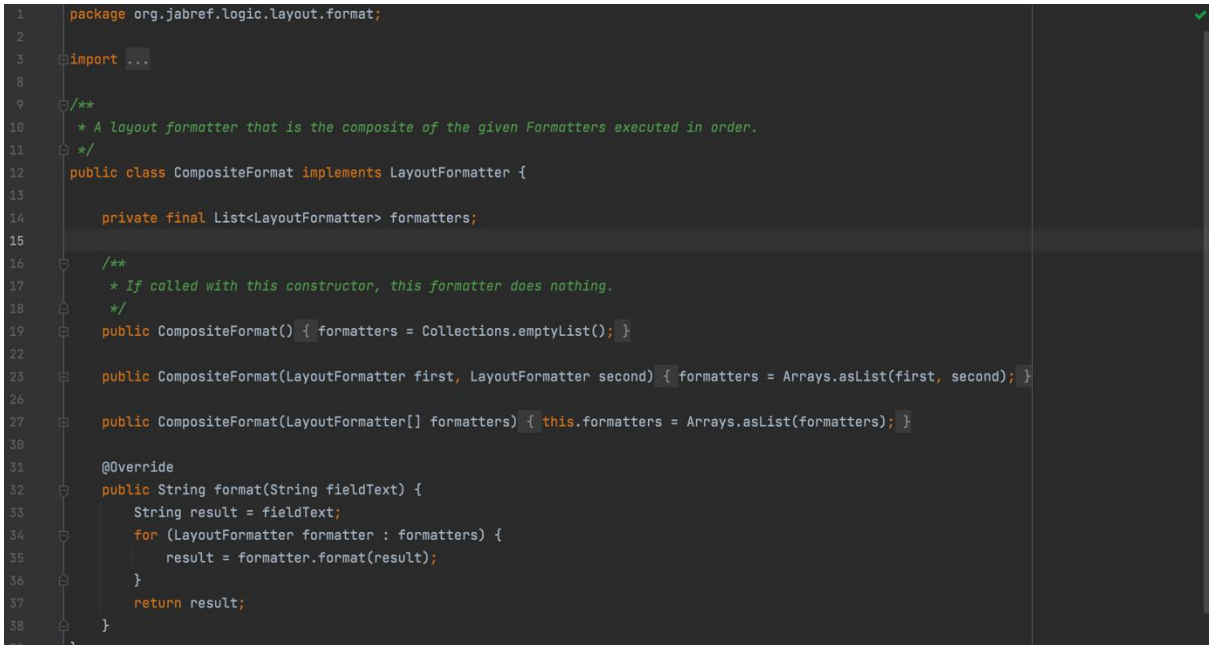
```
    public static class Builder {

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

        public Builder(PreviewPreferences previewPreferences) {
            this.previewCycle = previewPreferences.getPreviewCycle();
            this.previewCyclePosition = previewPreferences.getPreviewCyclePosition();
            this.previewPanelDividerPosition = previewPreferences.getPreviewPanelDividerPosition();
            this.previewStyle = previewPreferences.getPreviewStyle();
            this.previewStyleDefault = previewPreferences.getDefaultPreviewStyle();
            this.showPreviewAsExtraTab = previewPreferences.showPreviewAsExtraTab();
        }

        public Builder withShowAsExtraTab(boolean showAsExtraTab) {
            this.showPreviewAsExtraTab = showAsExtraTab;
            return this;
        }

        public Builder withPreviewCycle(List<PreviewLayout> previewCycle) {
            this.previewCycle = previewCycle;
            return withPreviewCyclePosition(previewCyclePosition);
        }

        public Builder withPreviewCyclePosition(int position) {
            if (previewCycle.isEmpty()) {
                previewCyclePosition = 0;
            } else {
                previewCyclePosition = position;
                while (previewCyclePosition < 0) {
                    previewCyclePosition += previewCycle.size();
                }
                previewCyclePosition %= previewCycle.size();
            }
            return this;
        }

        public Builder withPreviewPanelDividerPosition(Number previewPanelDividerPosition) {
            this.previewPanelDividerPosition = previewPanelDividerPosition;
            return this;
        }

        public Builder withPreviewStyle(String previewStyle) {
            this.previewStyle = previewStyle;
            return this;
        }

        public PreviewPreferences build() {
            return new PreviewPreferences(previewCycle, previewCyclePosition, previewPanelDividerPosition, previewStyle, previewStyleDefault,
        }
    }
}
```

Location:
src/main/java/org/jabref/preferences/PreviewPreferences.java

Reasoning:
In this piece of code there are multiple construction methods under same method. This resembles the builder design pattern.
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Composite:

Code:

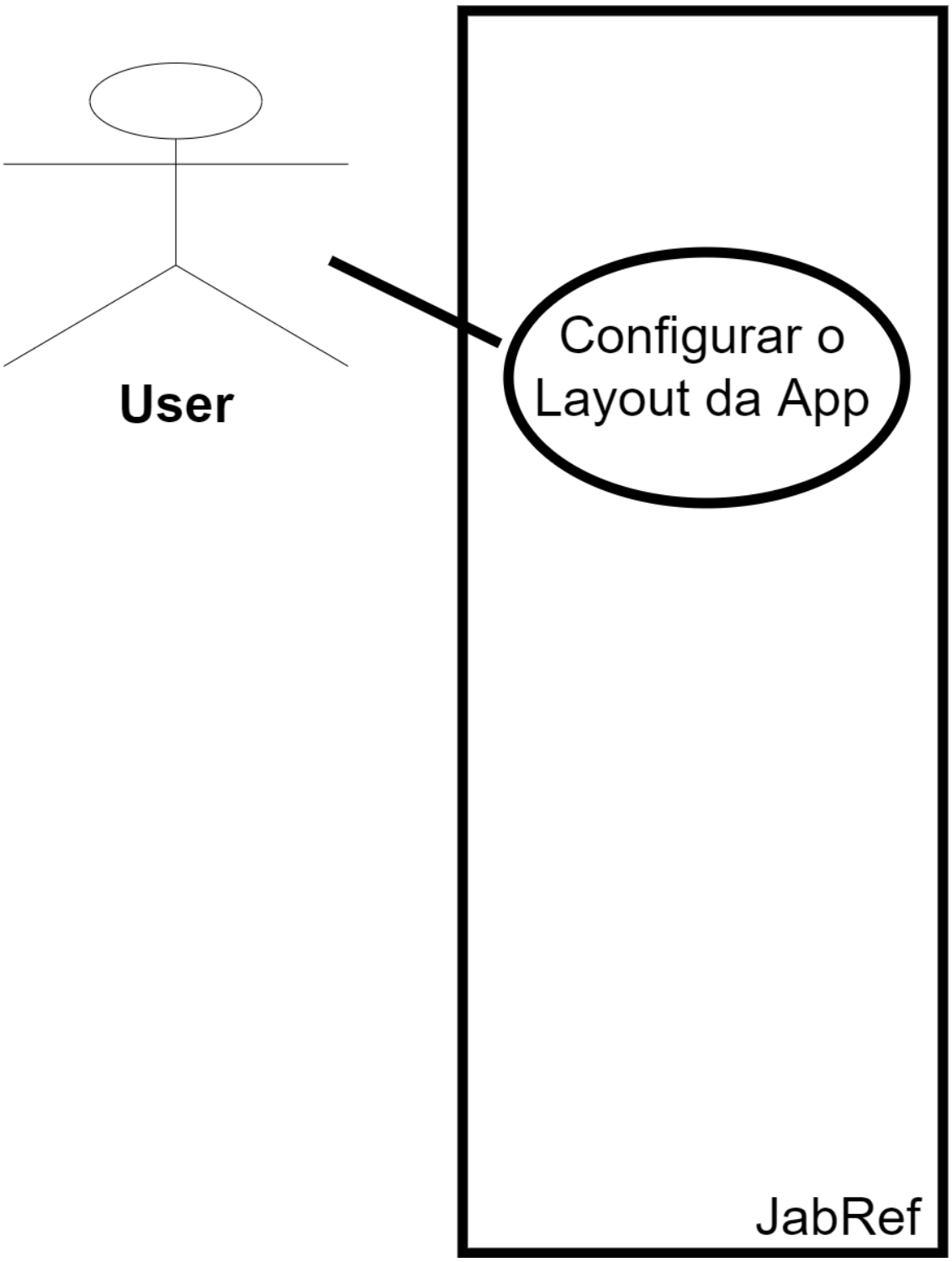


```java
package org.jabref.logic.layout.format;

import ...

/**
 * A layout formatter that is the composite of the given Formatters executed in order.
 */
public class CompositeFormat implements LayoutFormatter {

    private final List<LayoutFormatter> formatters;

    /**
     * If called with this constructor, this formatter does nothing.
     */
    public CompositeFormat() { formatters = Collections.emptyList(); }

    public CompositeFormat(LayoutFormatter first, LayoutFormatter second) { formatters = Arrays.asList(first, second); }

    public CompositeFormat(LayoutFormatter[] formatters) { this.formatters = Arrays.asList(formatters); }

    @Override
    public String format(String fieldText) {
        String result = fieldText;
        for (LayoutFormatter formatter : formatters) {
            result = formatter.format(result);
        }
        return result;
    }
```

Location:
src/main/java/org/jabref/logic/layout/format/CompositeFormat.java

Reasoning:
The pattern present in this piece of code enables multiple ways to create a same type of object
Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Use Case Diagrams



Use Case Diagrams:

Use Case “Configurar o layout da app”:

Ator Principal: Utilizador

Ator(es) Secundário(s): nenhum

Descrição: o utilizador pode configurar o layout/aparência da aplicação tais como a "font". Existem diferentes estilos e o utilizador pode criar o seu próprio estilo

Reviews

Code Smells:

Gonçalo – Code smell 1: Not sure the refactoring proposal is the most appropriate way to resolve this code smell but also not sure about how to fix it!

João – Code smell 2: I believe this refactoring proposal to be too abstract. The idea is there but the proposal code needs some work. Maybe try something more specific.

Mariana – Code smell 3: Nothing to note. Seems simple enough.

Design Patterns:

Gonçalo – Design Pattern 1:
I believe this this design pattern to be well identified but poorly justified.
This pattern allows the class to be responsible for keeping track of its sole instance by ensuring that no other instance is created by intercepting requests to create new objects and by providing the sole way to access the instance. Further, this prevents lazy creation which means the object isn't created until needed.
Maybe elaborate the reasoning a little bit more.

João – Design Pattern 2:
I view this analysis to be a little too straight forward. I believe you can elaborate the reasoning making it more substantial and explanatory. I can see how you preferred a more 'straight-to-the-point' approach on this well identified pattern, but perhaps you can enrich your doc. --

Mariana – Design Pattern 3:
Note: I believe the title to your 2nd design pattern was forgotten.

My review would be to elaborate your reasoning since I view it to be insufficient. I understand the 'straight-to-the-point' approach on this well identified pattern, but I see this as an opportunity to enrich your doc and make it less minimal.

Use case:

Miguel – After reviewing this use case and its diagram, I believe there is nothing to add to the existing document.

Gonçalo Prata

Code Metrics

Lines of Code metrics

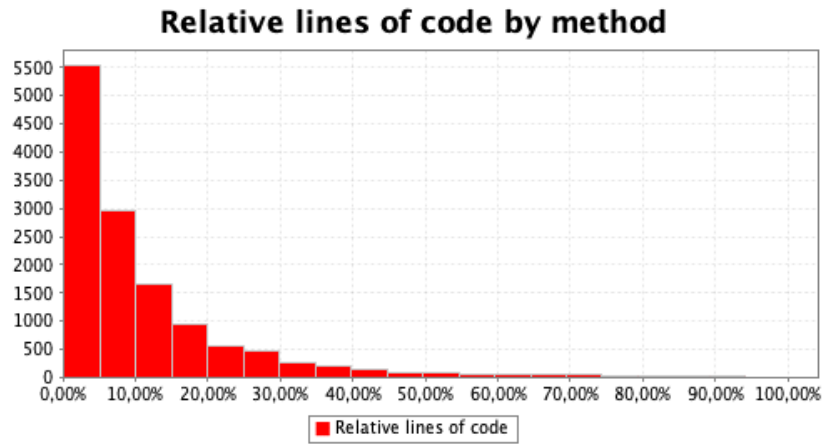
Method	CLOC	JLOC	LOC	NCLOC	RLOC
null.format(String)	0	0	4	4	66,67%
null.getStyleableProperty(TitledPane)	0	0	8	8	53,33%
org.jabref.architecture.MainArchitectureTests.doNotUseJavaAWT(JavaClasses)	0	0	6	6	5,41%

Caption:
CLOC – Comment lines of code
JLOC – Javadoc lines of code
LOC – Lines of code
NCLOC – Non-comment lines of code
RLOC – Relative lines of code

Analysis of the collected metrics:

LOC is the count of the number of lines of text in a file or directory. The number of lines indicates the size of a given file and gives some indication of the work involved. Considering the data in this chart, we can observe that there are no commented lines of code in the presented methods. There are also no Javadoc lines of code in neither of them. The first method presents a higher percentage of relative lines of code meanwhile the third one has a very low percentage. Relative LOC is very used for automated testing. This technique improves branch and statement coverage and fault detection.

Analyzing this data, we can identify code smells, for example, the fact that there are no comments in the lines of code.



Code Smells

Code smell 1- Records should be used instead of ordinary classes when representing immutable data structure
Code snippet:

```java
package org.jabref.logic.shared.event;

import org.jabref.model.database.BibDatabaseContext;

/**
 * A new {@link ConnectionLostEvent} is fired, when the connection to the shared database gets lost.
 */
public class ConnectionLostEvent {

    private final BibDatabaseContext bibDatabaseContext;

    /**
     * @param bibDatabaseContext Affected {@link BibDatabaseContext}
     */
    public ConnectionLostEvent(BibDatabaseContext bibDatabaseContext) {
        this.bibDatabaseContext = bibDatabaseContext;
    }

    public BibDatabaseContext getBibDatabaseContext() {
        return this.bibDatabaseContext;
    }
}
```

Location of the code:
src/main/java/jabref/logic/remote/shared/event/ConnectionLostEvent.java

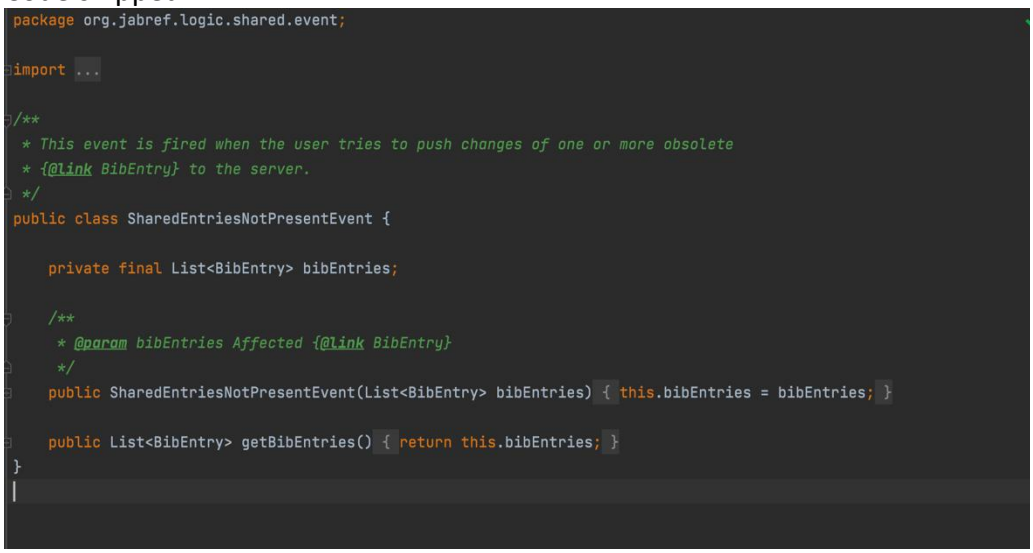
In this specific piece of code there is the opportunity to introduce records which represent immutable read-only data structure and should be used instead of creating immutable classes. This code smell can easily be fixed by refactoring the class declaration to “record ConnectionLostEvent(BibDatabaseContext bibDatabaseContext)”.

Refactoring proposal:

```java
package org.jabref.logic.shared.event;

import org.jabref.model.database.BibDatabaseContext;

/**
 * A new {@link ConnectionLostEvent} is fired, when the connection to the shared database gets lost.
 */
record ConnectionLostEvent(BibDatabaseContext bibDatabaseContext) { }
```

Code smell 2- Records should be used instead of ordinary classes when representing immutable data structure

Code snippet:

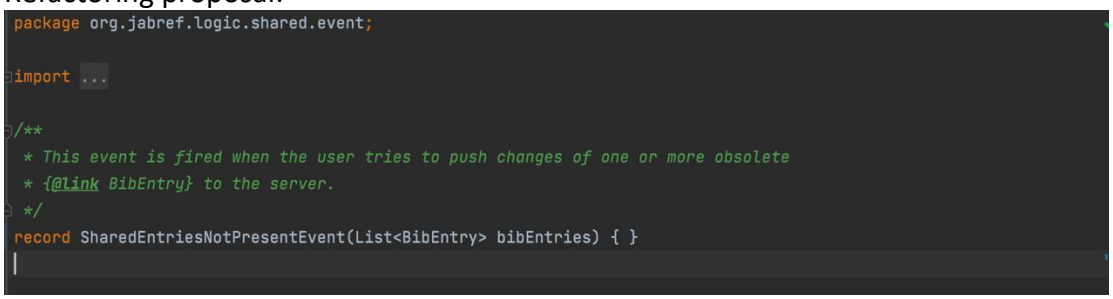


```java
package org.jabref.logic.shared.event;

import ...

/**
 * This event is fired when the user tries to push changes of one or more obsolete
 * {@link BibEntry} to the server.
 */
public class SharedEntriesNotPresentEvent {

    private final List<BibEntry> bibEntries;

    /**
     * @param bibEntries Affected {@link BibEntry}
     */
    public SharedEntriesNotPresentEvent(List<BibEntry> bibEntries) { this.bibEntries = bibEntries; }

    public List<BibEntry> getBibEntries() { return this.bibEntries; }
}
```

Location of the code:
src/main/java/jabref/logic/remote/shared/event/SharedEntriesNotPresentEvent.java

Again, this piece of code shows the missed opportunity to introduce records which represent immutable read-only data structure and should be used instead of creating immutable classes. This can be fixed by refactoring the class declaration to use “record SharedEntriesNotPresentEvent(List<BibEntry> bibEntries)”.

Refactoring proposal:



```java
package org.jabref.logic.shared.event;

import ...

/**
 * This event is fired when the user tries to push changes of one or more obsolete
 * {@link BibEntry} to the server.
 */
record SharedEntriesNotPresentEvent(List<BibEntry> bibEntries) { }
```

Code smell 3- Records should be used instead of ordinary classes when representing immutable data structure

Code snippet:

```java
package org.jabref.logic.shared.event;

import ...

/**
 * A new {@link UpdateRefusedEvent} is fired, when the user tries to push changes of an obsolete {@link BibEntry} to the server.
 */
public class UpdateRefusedEvent {

    private final BibDatabaseContext bibDatabaseContext;
    private final BibEntry localBibEntry;
    private final BibEntry sharedBibEntry;

    /**
     * @param bibDatabaseContext Affected {@link BibDatabaseContext}
     * @param localBibEntry      Affected {@link BibEntry}
     */
    public UpdateRefusedEvent(BibDatabaseContext bibDatabaseContext, BibEntry localBibEntry, BibEntry sharedBibEntry) {
        this.bibDatabaseContext = bibDatabaseContext;
        this.localBibEntry = localBibEntry;
        this.sharedBibEntry = sharedBibEntry;
    }

    public BibDatabaseContext getBibDatabaseContext() { return this.bibDatabaseContext; }

    public BibEntry getLocalBibEntry() { return localBibEntry; }

    public BibEntry getSharedBibEntry() { return sharedBibEntry; }
}
```

Location of the code:
src/main/java/jabref/logic/shared/event/UpdateRefusedEvent.java

The problem that was described before can also be found in this chunk of code. Once again, the fix is to use “record UpdateRefusedEvent(BibDatabaseContext bibDatabaseContext, BibEntry localBibEntry, BibEntry sharedBibEntry)”, instead of the used class declaration.

Refactoring proposal:

```java
package org.jabref.logic.shared.event;

import ...

/**
 * A new {@link UpdateRefusedEvent} is fired, when the user tries to push changes of an obsolete {@link BibEntry} to the server.
 */
record UpdateRefusedEvent(BibDatabaseContext bibDatabaseContext, BibEntry localBibEntry, BibEntry sharedBibEntry){


}
```

Design Patterns

Singleton:

Code:

```
// 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;
```

Location:
src/main/java/org/jabref/gui/externalfiletype/ExternalFileTypes.java

Reasoning:

```
// The only instance of this class:
```

Factories:

Code:

```
/**
 * Constructs a {@link TableCell} based on an optional value of the cell and a bunch of
specified converter methods.
 *
 * @param <S> view model of table row
 * @param <T> cell value
 */
public class OptionalValueTableCellFactory<S, T> extends ValueTableCellFactory<S,
Optional<T>> {

    private BiFunction<S, T, Node> toGraphicIfPresent;
    private Node defaultGraphic;

    public OptionalValueTableCellFactory<S, T> withGraphicIfPresent(BiFunction<S, T,
Node> toGraphicIfPresent) {
        this.toGraphicIfPresent = toGraphicIfPresent;
        setToGraphic();
        return this;
    }

    public OptionalValueTableCellFactory<S, T> withDefaultGraphic(Node defaultGraphic)
{
        this.defaultGraphic = defaultGraphic;
        setToGraphic();
        return this;
    }

    private void setToGraphic() {
        withGraphic((rowItem, item) -> {
            if (item.isPresent() && toGraphicIfPresent != null) {
                return toGraphicIfPresent.apply(rowItem, item.get());
            } else {
                return defaultGraphic;
            }
        });
    }
}
```

Location:
src/main/java/org/jabref/gui/util/OptionalValueTableCellFactory.java

Reasoning:
Due to having optional fields, this implementation of the abstract method actually implements different specifications of the same object

Builder:

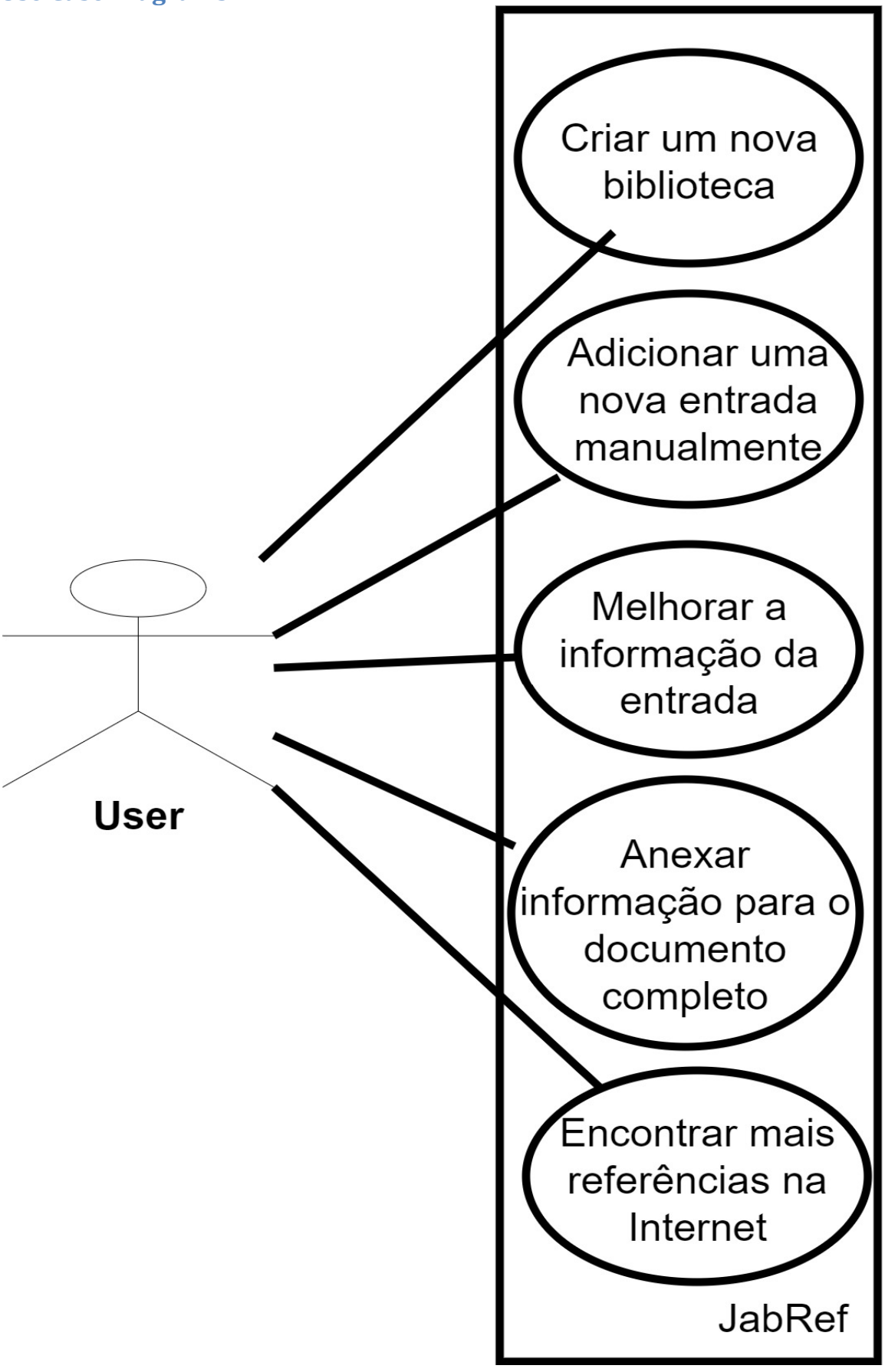
Code:

```
public static class Builder {

        private Path initialDirectory;

        public DirectoryDialogConfiguration build() {
            return new DirectoryDialogConfiguration(initialDirectory);
        }

        public Builder withInitialDirectory(Path directory) {

            directory = directory.toAbsolutePath();
            // Dir must be a folder, not a file
            if (!Files.isDirectory(directory)) {
                directory = directory.getParent();
            }
            // The lines above work also if the dir does not exist at all!
            // NULL is accepted by the filechooser as no inital path

            if (!Files.exists(directory)) {

                directory = null;
            }
            initialDirectory = directory;
            return this;
        }

        public Builder withInitialDirectory(String directory) {
            withInitialDirectory(Path.of(directory));
            return this;
        }
    }
}
```

Location:
src/main/java/org/jabref/gui/util/DirectoryDialogConfiguration.java

Reasoning:
Multiple construction methods under same method

Use Case Diagrams



Use Case “Enviar e receber dados para uma base de dados SQL”:

Ator Principal: User

Ator(es) Secundário(s): não existem

Descrição: Este diagrama demonstra os passos inicias que um utilizador tipicamente irá seguir após instalar a aplicação.

O utilizador irá criar uma biblioteca onde terá as diferentes entradas. O utilizador irá depois adicionar uma entrada manualmente (com mais prática e tempo o utilizador pode começar a usar ferramentas externas para adicionar entradas). O utilizador após adicionar uma entrada pode melhorar a informação da mesma, editando certos campos ou até anexar um documento à entrada para completar a informação da mesma. Para completar os passos iniciais, o utilizador pode procurar mais referências na Internet.

Reviews

Use Case by Debora

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, concluo que não há alterações que sejam necessárias de ser feitas

Code smells

1 by João

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, concluo que não há alterações que sejam necessárias de ser feitas.

2 by Mariana

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, concluo que não há alterações que sejam necessárias de ser feitas.

3 by Miguel

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, verifiquei que a secção "reasoning" não está preenchida

Design patterns

1 by João

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, concluo que não há alterações que sejam necessárias de ser feitas.

2 by Mariana

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, verifiquei que a secção "reasoning" não está preenchida.

3 by Miguel

Gonçalo Prata (52912) - 3/12 - Após verificar os resultados produzidos, concluo que não há alterações necessárias.

João Silva

Code Metrics

Chidamber Kemerer metrics

Class	CBO	DIT	LCOM	NOC	RFC	WMC
org.jabref.JabRefPreferencesTest	2	1	2	0	8	2
org.jabref.TestIconsProperties	0	1	1	0	23	3
org.jabref.architecture.MainArchitectureTests	4	1	9	0	36	12

Caption:
CBO – Coupling between objects
DIT – Depth of inheritance tree
LCOM – Lack of cohesion of methods
NOC – Number of children
RFC – Response for class
WMC – Weighted method complexity

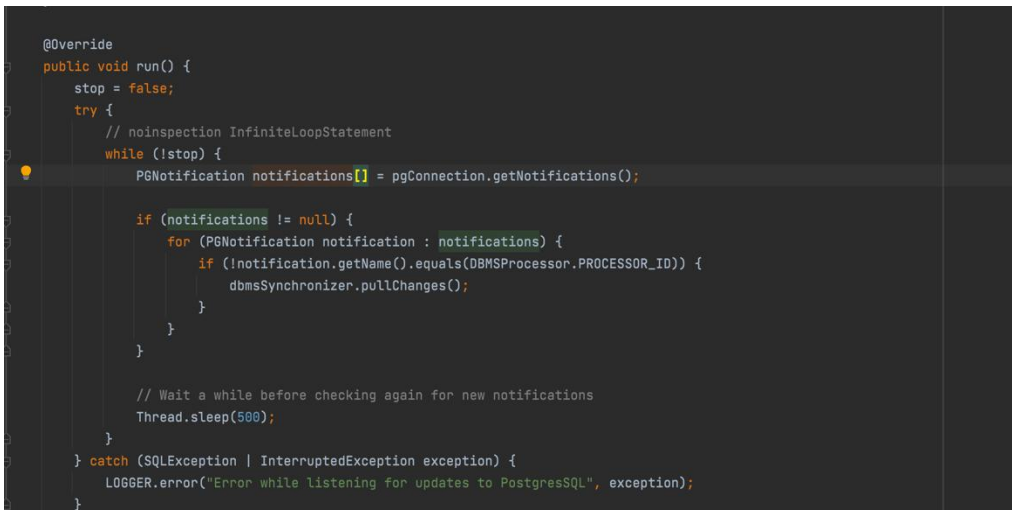
Short analysis:

CBO is the number of classes to which a class is coupled. We consider that two classes are coupled when methods of one of the classes uses methods of the other one. Excessive coupling is not good coding. Looking into the values of the first column, it is possible to observe that they're not too high which means that there's not excessive coupling in these classes.
DIT is the maximum inheritance path from the class to the root class. Higher the values of DIT, higher the possibility to find faults in the code. As we can observe the Depth of inheritance tree is 1 in all the presented classes which is not a high value. All the classes have no children.
WMC is the number of methods in a class. Naturally, more methods the class has, more possibility to lead to faults in the code. In the presented classes only the last one has an higher number of methods which may lead to more errors or bad coding. The third class is also the one with more LCO methods, what can be a trouble spot.

Regarding code smells, it is possible to identify Future Envy and Inappropriate Intimacy based on the CBO. This code smells happen when a method is more interested in some class than the one it is in or when two classes depend too much on each other.

Code Smells

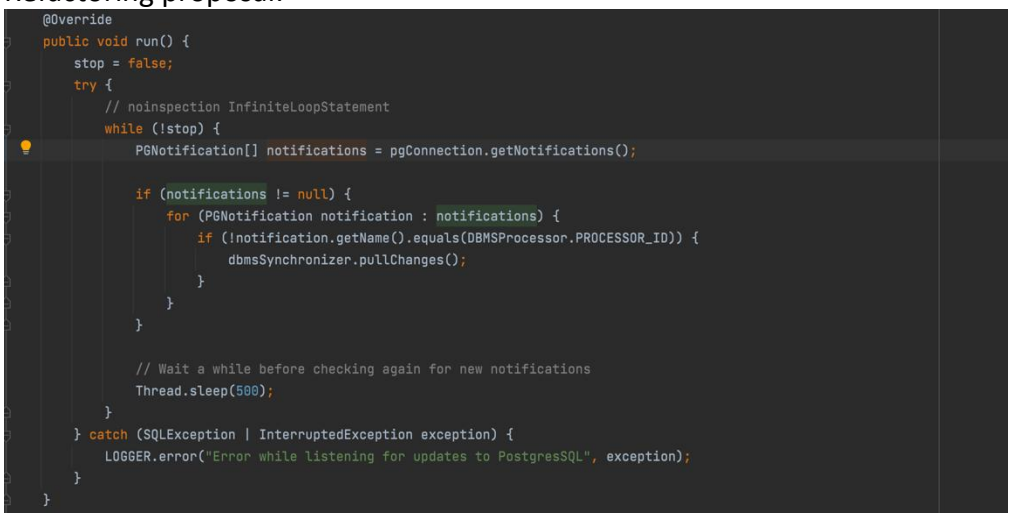
Code smell 1: Array designators "[]" should be on the type, not the variable
Code:



```
@Override
public void run() {
    stop = false;
    try {
        // noinspection InfiniteLoopStatement
        while (!stop) {
            PGNotification notifications[] = pgConnection.getNotifications();

            if (notifications != null) {
                for (PGNotification notification : notifications) {
                    if (!notification.getName().equals(DBMSProcessor.PROCESSOR_ID)) {
                        dbmsSynchronizer.pullChanges();
                    }
                }
            }

            // Wait a while before checking again for new notifications
            Thread.sleep(500);
        }
    } catch (SQLException | InterruptedException exception) {
        LOGGER.error("Error while listening for updates to PostgresSQL", exception);
    }
```

Location:
src/main/java/jabref/logic/remote/shared/listener/PostgresSQLNotificationListener.java

In this specific snippet, the array designator is located on the variable. For better code readability the array designators should always be located on the type. If not, maintainers will have to look at both the type and the variable name to decide if it is an array.
This code smell can easily be fixed by moving the array designator to the type.
Refactoring proposal:



```
@Override
public void run() {
    stop = false;
    try {
        // noinspection InfiniteLoopStatement
        while (!stop) {
            PGNotification[] notifications = pgConnection.getNotifications();

            if (notifications != null) {
                for (PGNotification notification : notifications) {
                    if (!notification.getName().equals(DBMSProcessor.PROCESSOR_ID)) {
                        dbmsSynchronizer.pullChanges();
                    }
                }
            }

            // Wait a while before checking again for new notifications
            Thread.sleep(500);
        }
    } catch (SQLException | InterruptedException exception) {
        LOGGER.error("Error while listening for updates to PostgresSQL", exception);
    }
}
```

Code smell 2: Asserts should not be used to check the parameters of a public method
Code:

```
public void putAllDBMSConnectionProperties(DatabaseConnectionProperties properties) {
    assert (properties.isValid());

    setType(properties.getType().toString());
    setHost(properties.getHost());
    setPort(String.valueOf(properties.getPort()));
    setName(properties.getDatabase());
    setUser(properties.getUser());
    setUseSSL(properties.isUseSSL());
    setKeystoreFile(properties.getKeyStore());
    setServerTimezone(properties.getServerTimezone());

    try {
        setPassword(new Password(properties.getPassword().toCharArray(), properties.getUser()).encrypt());
    } catch (GeneralSecurityException | UnsupportedEncodingException e) {
        LOGGER.error("Could not store the password due to encryption problems.", e);
    }
}
}
```

Location:
src/main/java/jabref/logic/remote/shared/prefs/SharedDatabasePreferences.java

In this specific snippet, there is an inappropriate use of an assertion since it was used for parameter validation. This can´t happen because assertions can be disabled at runtime therefore a bad operational setting would eliminate the intended checks. Also, it would be thrown an AsserionError which is very different from an Exception.
This code smell can easily be fixed by using some kind of exception rather than an assertion.

Refactoring proposal:

```
public void putAllDBMSConnectionProperties(DatabaseConnectionProperties properties) {
    if (!properties.isValid()) {/** throw some kind of exception */};

    setType(properties.getType().toString());
    setHost(properties.getHost());
    setPort(String.valueOf(properties.getPort()));
    setName(properties.getDatabase());
    setUser(properties.getUser());
    setUseSSL(properties.isUseSSL());
    setKeystoreFile(properties.getKeyStore());
    setServerTimezone(properties.getServerTimezone());

    try {
        setPassword(new Password(properties.getPassword().toCharArray(), properties.getUser()).encrypt());
    } catch (GeneralSecurityException | UnsupportedEncodingException e) {
        LOGGER.error("Could not store the password due to encryption problems.", e);
    }
}
```

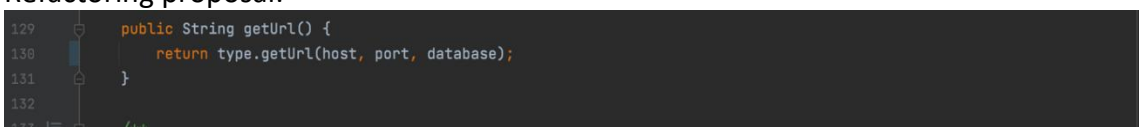
Code smell 3: Local variables should not be declared and then immediately returned or thrown

Code:

Location:
src/main/java/jabref/logic/remote/shared/security/DBMSConnectionProperties.java

In this specific piece of code there is a common bad practice. Declaring a variable only to immediately return or throw it is useless.
This code smell can easily be fixed by returning the value of the local variable instead.

Refactoring proposal:



Design Patterns

Singleton (One example):

Code:

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

Location:
src/main/java/org/jabref/preferences/JabRefPreferences.java

Reason:

```
// The only instance of this class:
```

Builder (Two examples):

Code:

```
public static class Builder {

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

   public Builder(PreviewPreferences previewPreferences) {
      this.previewCycle = previewPreferences.getPreviewCycle();
      this.previewCyclePosition = previewPreferences.getPreviewCyclePosition();
      this.previewPanelDividerPosition =
previewPreferences.getPreviewPanelDividerPosition();
      this.previewStyle = previewPreferences.getPreviewStyle();
      this.previewStyleDefault = previewPreferences.getDefaultPreviewStyle();
      this.showPreviewAsExtraTab = previewPreferences.showPreviewAsExtraTab();
   }

   public Builder withShowAsExtraTab(boolean showAsExtraTab) {
      this.showPreviewAsExtraTab = showAsExtraTab;
      return this;
   }

   public Builder withPreviewCycle(List<PreviewLayout> previewCycle) {
      this.previewCycle = previewCycle;
      return withPreviewCyclePosition(previewCyclePosition);
   }

   public Builder withPreviewCyclePosition(int position) {
      if (previewCycle.isEmpty()) {
         previewCyclePosition = 0;
      } else {
         previewCyclePosition = position;
         while (previewCyclePosition < 0) {
            previewCyclePosition += previewCycle.size();
         }
         previewCyclePosition %= previewCycle.size();
      }
      return this;
   }

   public Builder withPreviewPanelDividerPosition(Number previewPanelDividerPosition) {
      this.previewPanelDividerPosition = previewPanelDividerPosition;
```

```
        return this;
    }

    public Builder withPreviewStyle(String previewStyle) {
        this.previewStyle = previewStyle;
        return this;
    }

    public PreviewPreferences build() {
        return new PreviewPreferences(previewCycle, previewCyclePosition,
previewPanelDividerPosition, previewStyle, previewStyleDefault, showPreviewAsExtraTab);
    }
}
```

Location:
src/main/java/org/jabref/preferences/PreviewPreferences.java

Reason:
Multiple construction methods under same method.

Code:

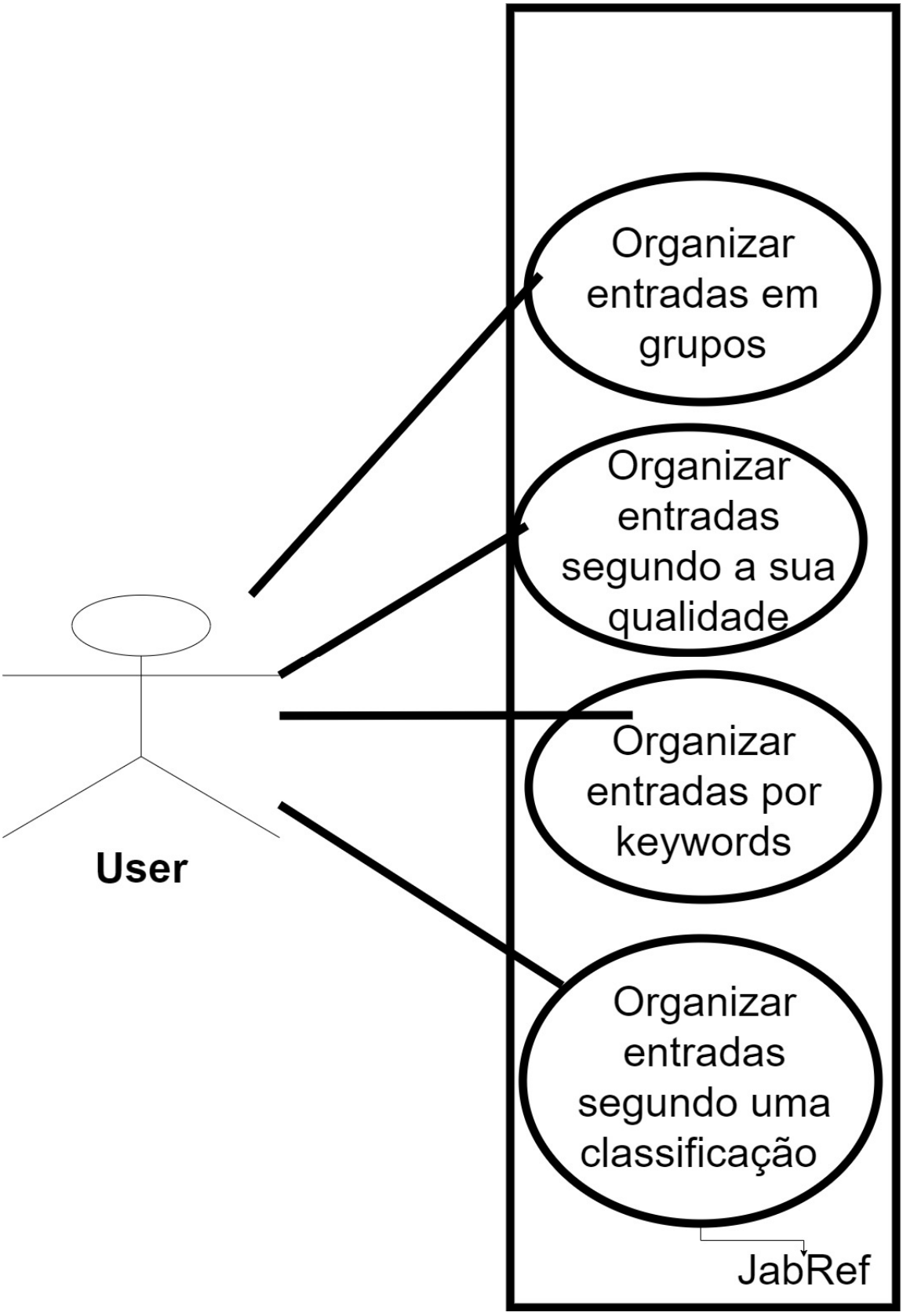
```
private ComplexSearchQueryBuilder() {
}

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

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

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

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

```
    return this;
}

public ComplexSearchQueryBuilder fromYearAndToYear(Integer fromYear, Integer toYear) {
    if (Objects.nonNull(singleYear)) {
        throw new IllegalArgumentException("You can not use single year and year range
search.");
    }
    this.fromYear = Objects.requireNonNull(fromYear);
    this.toYear = Objects.requireNonNull(toYear);
    return this;
}

public ComplexSearchQueryBuilder singleYear(Integer singleYear) {
    if (Objects.nonNull(fromYear) || Objects.nonNull(toYear)) {
        throw new IllegalArgumentException("You can not use single year and year range
search.");
    }
    this.singleYear = Objects.requireNonNull(singleYear);
    return this;
}

public ComplexSearchQueryBuilder journal(String journal) {
    if (Objects.requireNonNull(journal).isBlank()) {
        throw new IllegalArgumentException("Parameter must not be blank");
    }
    this.journal = String.format("\"%s\"", journal.replace("\"", ""));
    return this;
}

public ComplexSearchQueryBuilder DOI(String doi) {
    if (Objects.requireNonNull(doi).isBlank()) {
        throw new IllegalArgumentException("Parameter must not be blank");
    }
    this.doi = doi.replace("\"", "");
    return this;
}

public ComplexSearchQueryBuilder terms(Collection<Term> terms) {
    terms.forEach(term -> {
        String termText = term.text();
        switch (term.field().toLowerCase()) {
            case "author" -> this.author(termText);
            case "title" -> this.titlePhrase(termText);
            case "abstract" -> this.abstractPhrase(termText);
            case "journal" -> this.journal(termText);
            case "doi" -> this.DOI(termText);
```

```
            case "year" -> this.singleYear(Integer.valueOf(termText));
            case "year-range" -> this.parseYearRange(termText);
            case "default" -> this.defaultFieldPhrase(termText);
        }
    });
    return this;
}

/**
 * Instantiates the AdvancesSearchConfig from the provided Builder parameters
 * If all text fields are empty an empty optional is returned
 *
 * @return ComplexSearchQuery instance with the fields set to the values defined in the
building instance.
 * @throws IllegalStateException An IllegalStateException is thrown in case all text search
fields are empty.
 *                          See:
https://softwareengineering.stackexchange.com/questions/241309/builder-pattern-when-
to-fail/241320#241320
 */
public ComplexSearchQuery build() throws IllegalStateException {
    if (textSearchFieldsAndYearFieldsAreEmpty()) {
        throw new IllegalStateException("At least one text field has to be set");
    }
    return new ComplexSearchQuery(defaultFieldPhrases, authors, titlePhrases,
abstractPhrases, fromYear, toYear, singleYear, journal, doi);
}
```

Location:
src/main/java/org/jabref/logic/importer/fetcher/ComplexSearchQuery.java

Reason:
Multiple construction methods under same method.


Organizar entradas em grupos
Organizar entradas segundo a sua qualidade
Organizar entradas por keywords
Organizar entradas segundo uma classificação
User
JabRef

“Organize Entries”

Main Actor: User

Short description: A user can organize the different entries he has added into different categories. A user can organize their entries into different groups, for that it is necessary to create a group by pressing the '+' key (default) and typing its name. A user can organize their entries by their quality, there are 3 different qualities: green, orange and red. A user can organize their entries using 'keywords', which are added to entries in a specific field within them. A user can organize their entries through a rating that has a range from 0 to 5 stars.

Reviews

Use case (Gonçalo Prata):

Review: After reading what was written by my colleague, I conclude I have very little to comment on. I can only say that I agree with this idea, thinking that it could prove itself to be extremely useful.

Design Patterns:

Mariana Maximiano - Review: Despite understanding the reasoning behind this design pattern, meaning, the obvious Factory Pattern used in this construction, I believe the explanation was shallow. I had difficulty justifying some patterns too, so I understand these difficulties. To conclude, Despite being, as I previously said, shallow, it is well identified.

Miguel Pauleta - Review: Despite understanding the reasoning behind this design pattern, meaning, the obvious Factory Pattern used in this construction, I believe the explanation was shallow. I had difficulty justifying some patterns too, so I understand these difficulties. To conclude, Despite being, as I previously said, shallow, it is well identified.

Débora Dias - Review: It's explained in a concise way, which I consider an advantage. This identification is accompanied by a quick explanation of what is the Composite Pattern, making this review a lot easier. Maybe it could be a little more elaborated, but for the current purpose of the this first phase, it's just enough.

Code Smells:

Mariana Maximiano - Review: Nothing to say. It's well identified, and the refactoring proposal is equally well.

Miguel Pauleta – Review: Although I understand what the problem might be here, I don't think that a String being repeated 4 times is an actual code smell. Regardless, the refactoring proposal it's alright.

Débora Dias - Review: Everything's fine. Nothing to note.

Mariana Maximiano

Code Metrics

MOOD metrics

Project	AHF	AIF	CF	MHF	MIF	PF
project	78,33%	23,20%	0,68%	36,93%	18,28%	49,59%

Caption:
AHF – Attribute hiding factor
AIF – Attribute inheritance factor
CF – Coupling factor
MHF – Method hiding factor
MIF – Method inheritance factor
PF – Polymorphism factor

Analysis of the collected metrics:

MOOD metrics are designed to provide a summary of the overall quality of a project. In an ideal world all the attributes would be hidden and AHF = 100% would be the perfect percentage. Observing the AHF percentage (78,33%) we may conclude that it is within acceptable values.
Regarding the MIF and the AIF:
MIF = inherited methods / total methods available in classes
AIF = inherited attributes / total attributes available in classes
A class that inherits lots of methods (attributes) from its ancestor classes contributes to a high MIF (AIF). A child class that redefines its ancestors' methods (attributes) and adds new ones contributes to a lower MIF (AIF). An independent class that does not inherit and has no children contributes to a lower MIF (AIF). The AIF and MIF values shouldn't be too high or too low. The acceptable MIF range is 20% to 80% and the acceptable AIF range is 0% to 48%, according to research. AIF is between acceptable values. MIF is a bit lower than it should be. This may be a trouble spot in the code.
PF measures the degree of method overriding in the class inheritance tree. The Polymorphism factor has an average percentage.
Coupling Factor measures the actual couplings among classes in relation to the maximum number of possible couplings. Analyzing the chart, it is possible to assume that almost no classes are coupled in the project.
The number of visible methods is a measure of the class functionality. A low MHF indicates insufficiently abstracted implementation and a high MHF indicates very little functionality. The project has MHF = 36,93%, which isn't too high but also not too low, so we may conclude that it is an acceptable percentage.

As for the code smells related to this metrics, we can identify the refused request regarding inheritance of classes and methods.

Code Smells

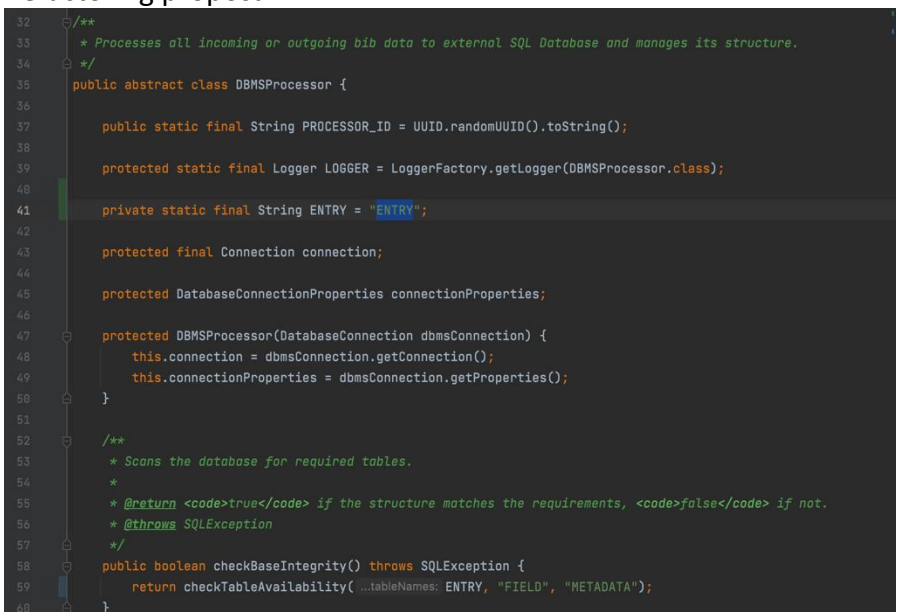
Code smell 1- String literals should not be duplicated
Code snippet:

```
*/
public boolean checkBaseIntegrity() throws SQLException {
    return checkTableAvailability( ...tableNames: 1 "ENTRY", "FIELD", "METADATA");
}
```

Location of the code:
src/main/java/jabref/logic/remote/shared/security/DBMSProcessor.java

In this specific code the string “ENTRY” is repeated 11 times. The same goes for many other strings in the code. This shows lack of attention to repeated code and therefore the code can get long and repetitive.
This code smell can easily be fixed by creating constants for all the words repeated various times. In this case, the word “ENTRY”. The same fix should be applied to all the others.

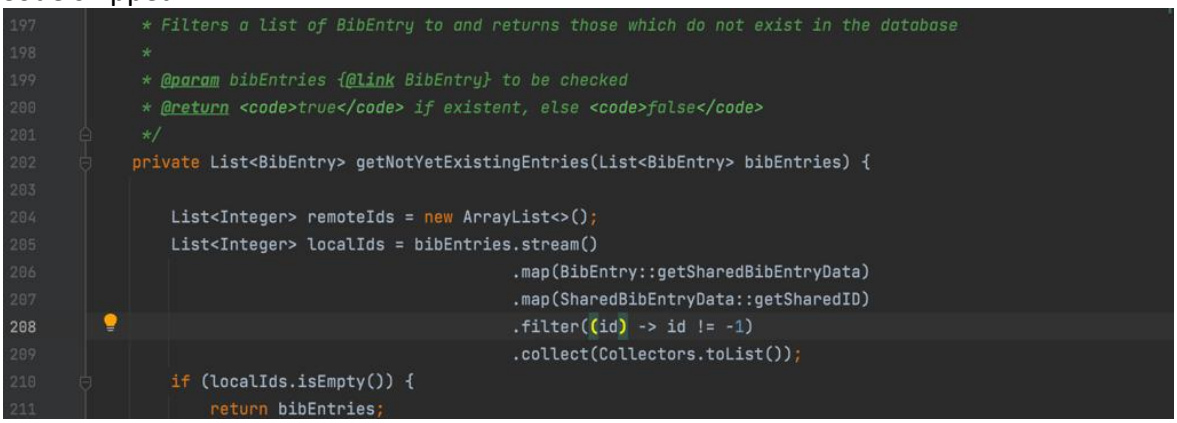
Refactoring proposal:



```
/**
 * Processes all incoming or outgoing bib data to external SQL Database and manages its structure.
 */
public abstract class DBMSProcessor {

    public static final String PROCESSOR_ID = UUID.randomUUID().toString();

    protected static final Logger LOGGER = LoggerFactory.getLogger(DBMSProcessor.class);

    private static final String ENTRY = "ENTRY";

    protected final Connection connection;

    protected DatabaseConnectionProperties connectionProperties;

    protected DBMSProcessor(DatabaseConnection dbmsConnection) {
        this.connection = dbmsConnection.getConnection();
        this.connectionProperties = dbmsConnection.getProperties();
    }

    /**
     * Scans the database for required tables.
     *
     * @return <code>true</code> if the structure matches the requirements, <code>false</code> if not.
     * @throws SQLException
     */
    public boolean checkBaseIntegrity() throws SQLException {
        return checkTableAvailability( ...tableNames: ENTRY, "FIELD", "METADATA");
    }
```

Code smell 2- Parentheses should be removed from a single lambda input parameter when its type is inferred

Code snippet:

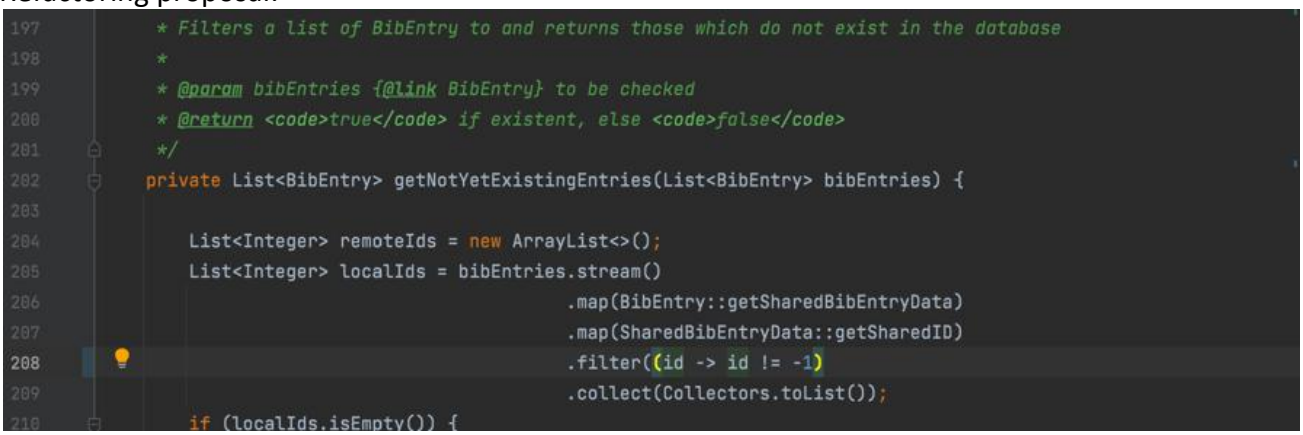


```
197     * Filters a list of BibEntry to and returns those which do not exist in the database
198     *
199     * @param bibEntries {@link BibEntry} to be checked
200     * @return <code>true</code> if existent, else <code>false</code>
201     */
202    private List<BibEntry> getNotYetExistingEntries(List<BibEntry> bibEntries) {
203
204        List<Integer> remoteIds = new ArrayList<>();
205        List<Integer> localIds = bibEntries.stream()
206                                           .map(BibEntry::getSharedBibEntryData)
207                                           .map(SharedBibEntryData::getSharedID)
208                                           .filter((id) -> id != -1)
209                                           .collect(Collectors.toList());
210        if (localIds.isEmpty()) {
211            return bibEntries;
```

Location of the code: src/main/java/jabref/logic/remote/shared/security/ DBMSProcessor.java

In this specific piece of code there are useless parentheses. It is preferred, if they are not necessary, that parentheses are not used. This code smell can easily be fixed by removing them.

Refactoring proposal:



```
197     * Filters a list of BibEntry to and returns those which do not exist in the database
198     *
199     * @param bibEntries {@link BibEntry} to be checked
200     * @return <code>true</code> if existent, else <code>false</code>
201     */
202    private List<BibEntry> getNotYetExistingEntries(List<BibEntry> bibEntries) {
203
204        List<Integer> remoteIds = new ArrayList<>();
205        List<Integer> localIds = bibEntries.stream()
206                                           .map(BibEntry::getSharedBibEntryData)
207                                           .map(SharedBibEntryData::getSharedID)
208                                           .filter(id -> id != -1)
209                                           .collect(Collectors.toList());
210        if (localIds.isEmpty()) {
```

Code smell 3- Parentheses should be removed from a single lambda input parameter when its type is inferred
Code snippet:

```
202     private List<BibEntry> getNotYetExistingEntries(List<BibEntry> bibEntries) {
203
204         List<Integer> remoteIds = new ArrayList<>();
205         List<Integer> localIds = bibEntries.stream()
206                                            .map(BibEntry::getSharedBibEntryData)
207                                            .map(SharedBibEntryData::getSharedID)
208                                            .filter((id) -> id != -1)
209                                            .collect(Collectors.toList());
210         if (localIds.isEmpty()) {
211             return bibEntries;
212         }
213         try {
214             StringBuilder selectQuery = new StringBuilder()
215                     .append("SELECT * FROM ")
216                     .append(escape( expression: "ENTRY"));
217
218             try (ResultSet resultSet = connection.createStatement().executeQuery(selectQuery.toStrin
219                 while (resultSet.next()) {
220                     int id = resultSet.getInt("SHARED_ID");
221                     remoteIds.add(id);
222                 }
223             }
224         } catch (SQLException e) {
225             LOGGER.error("SQL Error: ", e);
226         }
227         return bibEntries.stream().filter((entry) ->
228                 !remoteIds.contains(entry.getSharedBibEntryData().getSharedID()))
229                         .collect(Collectors.toList());
230     }
```

Location of the code:
src/main/java/jabref/logic/remote/shared/DBMSProcessor.java

Again, in this piece of code happens the same as described before. The parentheses should only be used when necessary.
This code smell can easily be fixed by removing them.

Refactoring proposal:

```
202    private List<BibEntry> getNotYetExistingEntries(List<BibEntry> bibEntries) {
203
204        List<Integer> remoteIds = new ArrayList<>();
205        List<Integer> localIds = bibEntries.stream()
206                                           .map(BibEntry::getSharedBibEntryData)
207                                           .map(SharedBibEntryData::getSharedID)
208                                           .filter((id) -> id != -1)
209                                           .collect(Collectors.toList());
210        if (localIds.isEmpty()) {
211            return bibEntries;
212        }
213        try {
214            StringBuilder selectQuery = new StringBuilder()
215                    .append("SELECT * FROM ")
216                    .append(escape( expression: "ENTRY"));
217
218            try (ResultSet resultSet = connection.createStatement().executeQuery(selectQuery.toStrin
219                while (resultSet.next()) {
220                    int id = resultSet.getInt("SHARED_ID");
221                    remoteIds.add(id);
222                }
223            }
224        } catch (SQLException e) {
225            LOGGER.error("SQL Error: ", e);
226        }
227        return bibEntries.stream().filter(entry ->
228                !remoteIds.contains(entry.getSharedBibEntryData().getSharedID()))
229                        .collect(Collectors.toList());
230    }
```

Design Patterns

Factories:

Code:

```
public class ViewModelTextFieldTableCellVisualizationFactory<S, T> implements
Callback<TableColumn<S, T>, TableCell<S, T>> {

    private static final PseudoClass INVALID_PSEUDO_CLASS =
PseudoClass.getPseudoClass("invalid");

    private Function<S, ValidationStatus> validationStatusProperty;
    private StringConverter<T> stringConverter;

    public ViewModelTextFieldTableCellVisualizationFactory<S, T>
withValidation(Function<S, ValidationStatus> validationStatusProperty) {
        this.validationStatusProperty = validationStatusProperty;
        return this;
    }

    public void install(TableColumn<S, T> column, StringConverter<T> stringConverter)
{
        column.setCellFactory(this);
        this.stringConverter = stringConverter;
    }
```

Location:
src/main/java/org/jabref/gui/util/ViewModelTextFieldTableCellVisualizationFactory.java

Reasoning:
Paired with src/main/java/org/jabref/gui/util/ValueTableCellFactory.java

Code:

```
/**
 * Constructs a {@link TableCell} based on the value of the cell and a bunch of specified
converter methods.
 *
 * @param <S> view model of table row
 * @param <T> cell value
 */
public class ValueTableCellFactory<S, T> implements Callback<TableColumn<S, T>,
TableCell<S, T>> {

    private Function<T, String> toText;
    private BiFunction<S, T, Node> toGraphic;
    private BiFunction<S, T, EventHandler<? super MouseEvent>>
toOnMouseClickedEvent;
    private Function<T, BooleanExpression> toDisableExpression;
    private Function<T, BooleanExpression> toVisibleExpression;
    private BiFunction<S, T, String> toTooltip;
    private Function<T, ContextMenu> contextMenuFactory;
    private BiFunction<S, T, ContextMenu> menuFactory;

    public ValueTableCellFactory<S, T> withText(Function<T, String> toText) {
        this.toText = toText;
        return this;
    }

    public ValueTableCellFactory<S, T> withGraphic(Function<T, Node> toGraphic) {
        this.toGraphic = (rowItem, value) -> toGraphic.apply(value);
        return this;
    }

    public ValueTableCellFactory<S, T> withGraphic(BiFunction<S, T, Node> toGraphic)
{
        this.toGraphic = toGraphic;
        return this;
    }

    public ValueTableCellFactory<S, T> withTooltip(BiFunction<S, T, String> toTooltip) {
        this.toTooltip = toTooltip;
        return this;
    }

    public ValueTableCellFactory<S, T> withTooltip(Function<T, String> toTooltip) {
        this.toTooltip = (rowItem, value) -> toTooltip.apply(value);
        return this;
    }

    public ValueTableCellFactory<S, T> withOnMouseClickedEvent(BiFunction<S, T,
EventHandler<? super MouseEvent>> toOnMouseClickedEvent) {
        this.toOnMouseClickedEvent = toOnMouseClickedEvent;
        return this;
    }

    public ValueTableCellFactory<S, T> withOnMouseClickedEvent(Function<T,
EventHandler<? super MouseEvent>> toOnMouseClickedEvent) {
        this.toOnMouseClickedEvent = (rowItem, value) ->
```

```
toOnMouseClickedEvent.apply(value);
        return this;
    }

    public ValueTableCellFactory<S, T> withDisableExpression(Function<T,
BooleanExpression> toDisableBinding) {
        this.toDisableExpression = toDisableBinding;
        return this;
    }

    public ValueTableCellFactory<S, T> withVisibleExpression(Function<T,
BooleanExpression> toVisibleBinding) {
        this.toVisibleExpression = toVisibleBinding;
        return this;
    }

    public ValueTableCellFactory<S, T> withContextMenu(Function<T, ContextMenu>
contextMenuFactory) {
        this.contextMenuFactory = contextMenuFactory;
        return this;
    }

    public ValueTableCellFactory<S, T> withMenu(BiFunction<S, T, ContextMenu>
menuFactory) {
        this.menuFactory = menuFactory;
        return this;
    }
```

Location:
src/main/java/org/jabref/gui/util/ValueTableCellFactory.java

Reasoning:
Paired with
src/main/java/org/jabref/gui/util/ViewModelTextFieldTableCellVisualizationFactory.java

Builder:

Code:

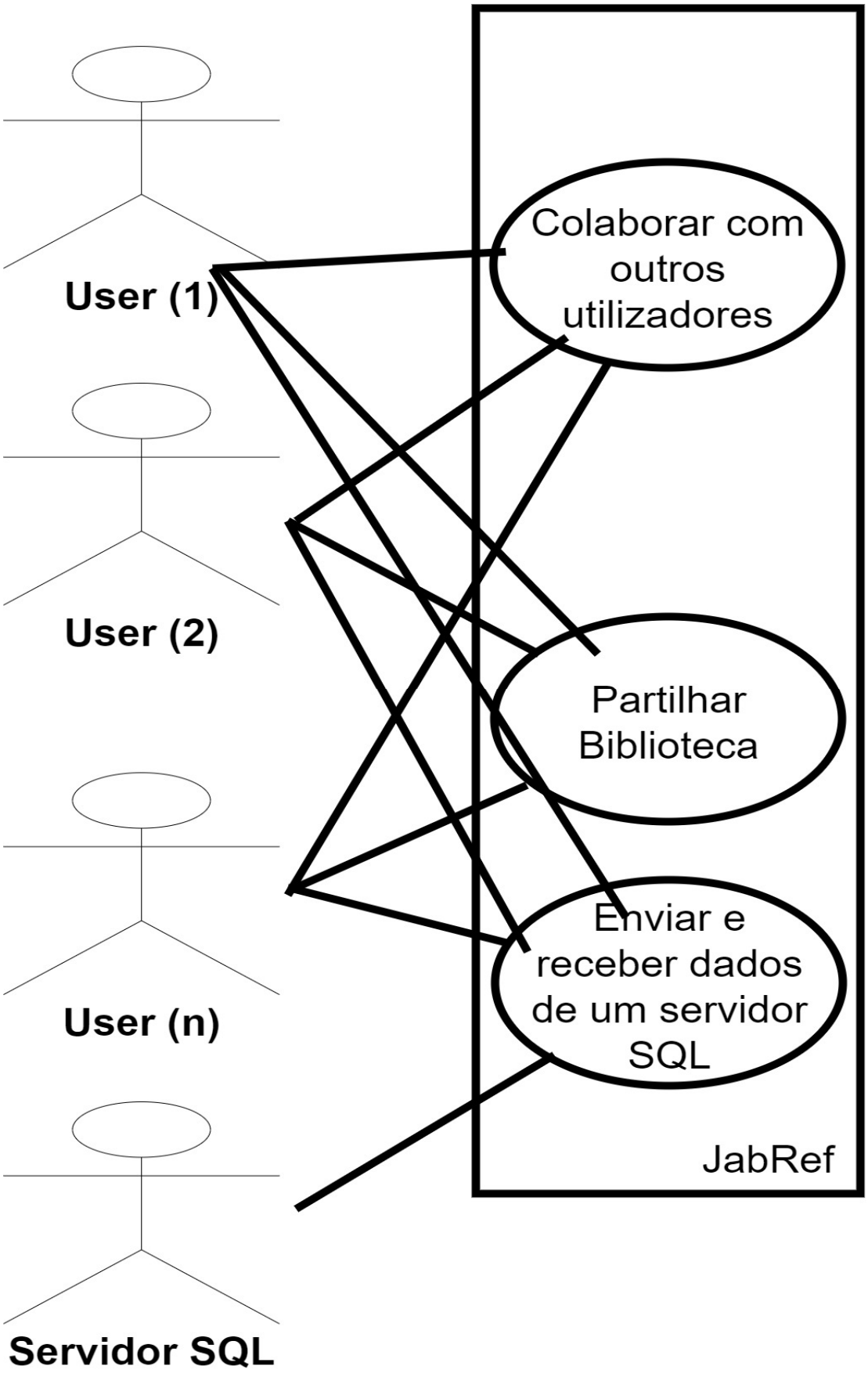
```
public class BibEntryTypeBuilder {

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

    public BibEntryTypeBuilder withType(EntryType type) {
        this.type = type;
        return this;
    }

    public BibEntryTypeBuilder withImportantFields(Set<BibField> newFields) {
        return
withImportantFields(newFields.stream().map(BibField::getField).collect(Collectors.toColl
ection(LinkedHashSet::new)));
    }

    public BibEntryTypeBuilder withImportantFields(Collection<Field> newFields) {
        this.fields = Streams.concat(fields.stream(), newFields.stream().map(field -> new
BibField(field, FieldPriority.IMPORTANT)))
                        .collect(Collectors.toCollection(LinkedHashSet::new));
        return this;
    }

    public BibEntryTypeBuilder withImportantFields(Field... newFields) {
        return withImportantFields(Arrays.asList(newFields));
    }

    public BibEntryTypeBuilder withDetailFields(Collection<Field> newFields) {
        this.fields = Streams.concat(fields.stream(), newFields.stream().map(field -> new
BibField(field, FieldPriority.DETAIL)))
                        .collect(Collectors.toCollection(LinkedHashSet::new));
        return this;
    }

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

    public BibEntryTypeBuilder withRequiredFields(Set<OrFields> requiredFields) {
        this.requiredFields = requiredFields;
        return this;
    }

    public BibEntryTypeBuilder withRequiredFields(Field... requiredFields) {
        this.requiredFields =
Arrays.stream(requiredFields).map(OrFields::new).collect(Collectors.toCollection(Linked
HashSet::new));
        return this;
    }

    public BibEntryTypeBuilder withRequiredFields(OrFields first, Field... requiredFields) {
        this.requiredFields = Stream.concat(Stream.of(first),
Arrays.stream(requiredFields).map(OrFields::new)).collect(Collectors.toCollection(Linked
HashSet::new));
```

```
        return this;
    }

    public BibEntryTypeBuilder withRequiredFields(List<OrFields> first, Field...
requiredFields) {
        this.requiredFields = Stream.concat(first.stream(),
Arrays.stream(requiredFields).map(OrFields::new)).collect(Collectors.toCollection(Linked
HashSet::new));
        return this;
    }

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

Location:
src/main/java/org/jabref/model/entry/BibEntryTypeBuilder.java

Reasoning:
Multiple construction methods under same method

Use Case Diagrams



Ator Principal: Utilizadores

Ator(es) Secundário(s): servidor SQL

Descrição: Um utilizador consegue ter uma biblioteca onde colabora com outros localizadores através de um servidor SQL ou através da partilha de uma biblioteca. Um utilizador pode partilhar uma biblioteca se a exportar para um dos seguintes formatos: txt, rtf, rdf, xml, html, htm, csv, ou ris. O servidor SQL pode ser acedido por diferentes utilizadores e também pode ser usado por um só utilizador, neste caso, o servidor pode ser usado como um software de controlo de versão (como git).

Reviews

Design Pattern Miguel (1) :

The design pattern is well identified. As the code shows the subclasses are coupled and can be seen as twins. In my opinion, this choice of pattern could've been better justified so it would be easier to comprehend for whoever analysis it.

Design pattern Débora (2):

It has a clear description, and it is very direct. The description could be a little more detailed.

Design Pattern Gonçalo (3):

Could have a bit more detail in the description but it is well identified and it goes straight to the point.

Code Smell Miguel (1):

The code smell is well identified. The code smell has an well put explanation and I think it's very clear for whoever reads it.

Code Smell Débora (2):

This one is well justified and has a good refactoring proposal. Nothing further to note.

Code Smell Gonçalo (3):

It seems a direct identification of the code smell and it is well explained. The refactoring proposal seems simple and also straight to the point.

Use Case Diagram

The use case is clearly identified and described.

Miguel Pauleta

Code Metrics

Dependency metrics

Class	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
org.jabref.JabRefPreferencesTest	0	2	1 334	0	0	1	0
org.jabref.TestIconsProperties	0	0	0	0	0	0	0
org.jabref.architecture.MainArchitectureTests	0	3	3	1	1	1	1

Caption:
Cyclic – number of cyclic dependencies
Dcy – number of dependencies
Dcy* – number of transitive dependencies
Dpt – number of dependents
Dpt* – number of transitive dependents
pDcy – number of package dependencies
pDpt – number of dependent packages

Analysis of the collected metrics:

In this data file, we have the example of three different classes with very different values regarding the code's dependencies. As we can see, none of them has cyclic dependencies. The first class, although it doesn't have a lot of dependencies (2), it has 1334 transitive dependencies. A transitive dependency is any dependency that is induced by the components that the program references directly. It has no dependents or transitive dependents. This class has one package dependency.
The second class doesn't have any dependencies or dependents.
The third one has 3 dependencies and 3 transitive dependencies, has 1 dependent and 1 transitive dependent. This class also has 1 package dependency and 1 dependent package. The amount of transitive dependencies in this class could be a trouble spot in the code because a class shouldn't depend so much on others.

The Dependency metrics is associated with inappropriate intimacy, which is a code smell that occurs when two classes depend too much on one another.

Code smells

Code smell 1- Generic exceptions should never be thrown
Code snippet:

```
/**
 * @return Java Class path for establishing JDBC connection.
 */
public String getDriverClassPath() throws Error {
   return this.driverPath;
}
```

Location of the code:
src/main/java/jabref/logic/remote/shared/security/DBMSType.java

In this specific piece of code, a generic exception is thrown. Using such generic exceptions as Error, RuntimeException, Throwable, and Exception prevents calling methods from handling true, system-generated exceptions differently than application-generated errors.

This code smell can easily be fixed by throwing a more specific exception.

Refactoring proposal:

```
/**
 * @return Java Class path for establishing JDBC connection.
 */
public String getDriverClassPath() throws SpecificException {
   return this.driverPath; //this method throws SpecificException

}
```

Code smell 2- String literals should not be duplicated
Code snippet:

```
try (Statement statement = oracleConnection.createStatement()) {
          ((OracleStatement)
statement).setDatabaseChangeRegistration(databaseChangeRegistration);
          StringBuilder selectQuery = new StringBuilder()
                  .append("SELECT 1 FROM ")
                  .append(escape("ENTRY"))
                  .append(", ")
                  .append(escape("METADATA"));
          // this execution registers all tables mentioned in selectQuery
          statement.executeQuery(selectQuery.toString());
      }
   } catch (SQLException e) {
      LOGGER.error("SQL Error: ", e);
   }
}
```

Location of the code:
src/main/java/jabref/logic/remote/shared/security/OracleProcessor.java

In this specific code the literal “SQL Error: “is repeated 4 times. It shouldn’t happen since it makes the code repetitive therefore difficult to read.
This code smell can easily be fixed by creating a constant to replace all these literals.

Refactoring proposal:

```
private static final SQL="SQL Error:";
```

```
}
   } catch (SQLException e) {
      LOGGER.error(SQL, e);
   }
}
```

Code smell 3-

Code snippet:

```
@Override
public void startNotificationListener(DBMSSynchronizer dbmsSynchronizer) {
    // Disable cleanup output of ThreadedHousekeeper
    // Logger.getLogger(ThreadedHousekeeper.class.getName()).setLevel(Level.SEVERE);
    try {
        connection.createStatement().execute("LISTEN jabrefLiveUpdate");
        // Do not use `new PostgresSQLNotificationListener(...)` as the object has to exist
continuously!
        // Otherwise the listener is going to be deleted by GC.
        PGConnection pgConnection = connection.unwrap(PGConnection.class);
        listener = new PostgresSQLNotificationListener(dbmsSynchronizer, pgConnection);
        JabRefExecutorService.INSTANCE.execute(listener);
    } catch (SQLException e) {
        LOGGER.error("SQL Error: ", e);
    }
}
```

Location of the code:
src/main/java/jabref/logic/remote/shared/security/PostgreSQLProcessor.java

Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required.

Refactoring proposal:
The proposal would be to just delete the commented code.

Design Pattern

Factories:

Code:

```
**
 * Constructs a {@link TreeTableCell} based on the view model of the row and a bunch
of specified converter methods.
 *
 * @param <S> view model
 */
public class ViewModelTreeTableCellFactory<S> implements
Callback<TreeTableColumn<S, S>, TreeTableCell<S, S>> {

    private Callback<S, String> toText;
    private Callback<S, Node> toGraphic;
    private Callback<S, EventHandler<? super MouseEvent>> toOnMouseClickedEvent;
    private Callback<S, String> toTooltip;

    public ViewModelTreeTableCellFactory<S> withText(Callback<S, String> toText) {
        this.toText = toText;
        return this;
    }

    public ViewModelTreeTableCellFactory<S> withGraphic(Callback<S, Node> toGraphic)
{
        this.toGraphic = toGraphic;
        return this;
    }

    public ViewModelTreeTableCellFactory<S> withIcon(Callback<S, JabRefIcon> toIcon)
{
        this.toGraphic = viewModel -> toIcon.call(viewModel).getGraphicNode();
        return this;
    }

    public ViewModelTreeTableCellFactory<S> withTooltip(Callback<S, String> toTooltip)
{
        this.toTooltip = toTooltip;
        return this;
    }

    public ViewModelTreeTableCellFactory<S> withOnMouseClickedEvent(
            Callback<S, EventHandler<? super MouseEvent>> toOnMouseClickedEvent) {
        this.toOnMouseClickedEvent = toOnMouseClickedEvent;
        return this;
    }
```

Location:
src/main/java/org/jabref/gui/util/ViewModelTreeTableCellFactory.java

Reasoning:
Twins with src/main/java/org/jabref/gui/util/ViewModelTreeCellFactory.java

Code:

```
/**
 * Constructs a {@link TreeTableCell} based on the view model of the row and a bunch
of specified converter methods.
 *
 * @param <S> view model
 * @param <T> cell value
 */
public class ViewModelTreeCellFactory<T> implements Callback<TreeView<T>,
TreeCell<T>> {

    private Callback<T, String> toText;
    private Callback<T, Node> toGraphic;
    private Callback<T, EventHandler<? super MouseEvent>> toOnMouseClickedEvent;
    private Callback<T, String> toTooltip;

    public ViewModelTreeCellFactory<T> withText(Callback<T, String> toText) {
        this.toText = toText;
        return this;
    }

    public ViewModelTreeCellFactory<T> withGraphic(Callback<T, Node> toGraphic) {
        this.toGraphic = toGraphic;
        return this;
    }

    public ViewModelTreeCellFactory<T> withIcon(Callback<T, JabRefIcon> toIcon) {
        this.toGraphic = viewModel -> toIcon.call(viewModel).getGraphicNode();
        return this;
    }

    public ViewModelTreeCellFactory<T> withTooltip(Callback<T, String> toTooltip) {
        this.toTooltip = toTooltip;
        return this;
    }

    public ViewModelTreeCellFactory<T> withOnMouseClickedEvent(Callback<T,
EventHandler<? super MouseEvent>> toOnMouseClickedEvent) {
        this.toOnMouseClickedEvent = toOnMouseClickedEvent;
        return this;
    }

public void install(TreeView<T> treeView) {
    treeView.setCellFactory(this);
}
```

Location:
src/main/java/org/jabref/gui/util/ViewModelTreeCellFactory.java

Reasoning:
Twins with src/main/java/org/jabref/gui/util/ViewModelTreeTableCellFactory.java

Builder:

Code:

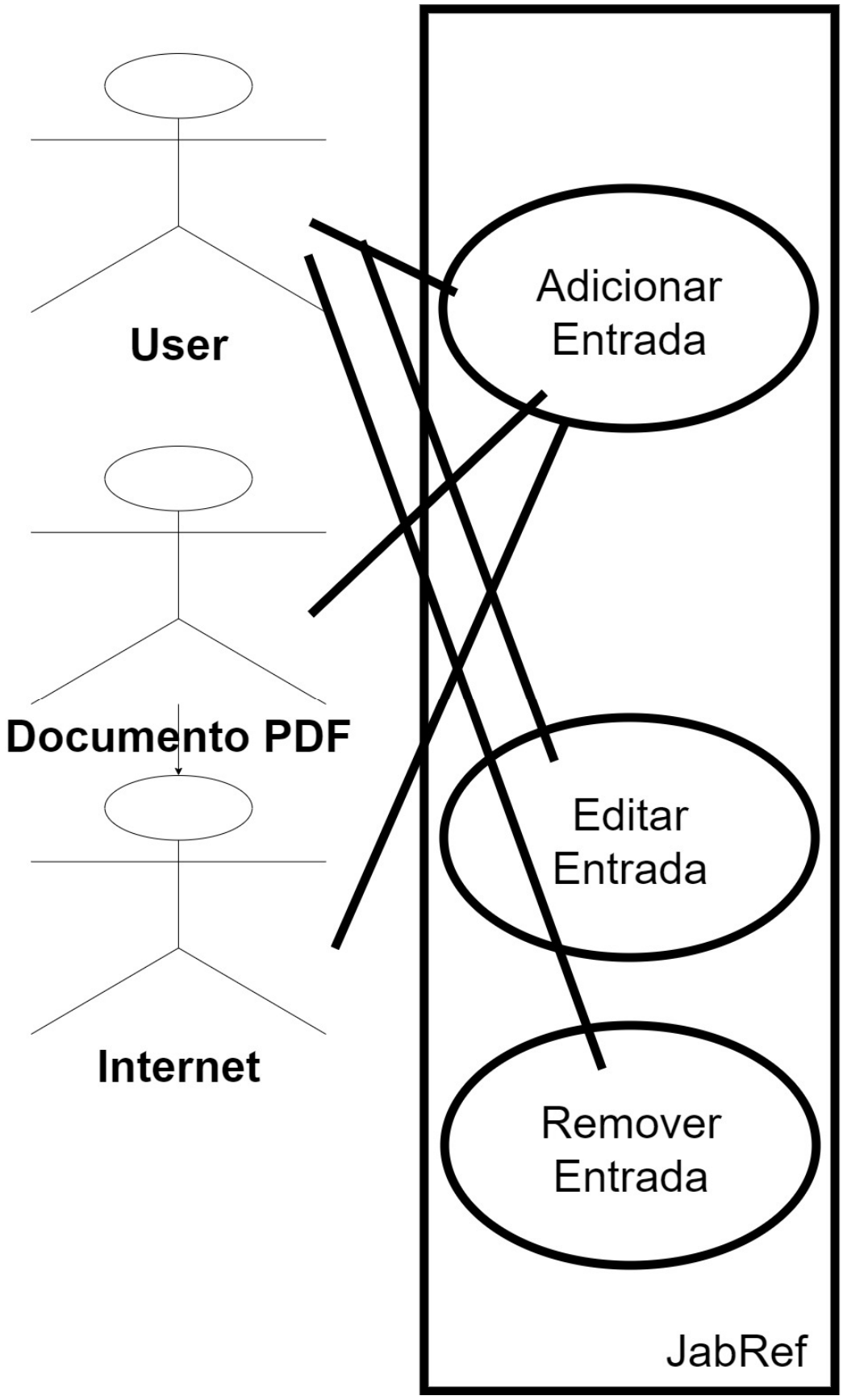
```
public static class Builder {

        private final List<FileChooser.ExtensionFilter> extensionFilters = new
ArrayList<>();
        private Path initialDirectory;
        private FileChooser.ExtensionFilter defaultExtension;
        private String initialFileName;

        public FileDialogConfiguration build() {
            return new FileDialogConfiguration(initialDirectory, extensionFilters,
defaultExtension, initialFileName);
        }

        public Builder withInitialDirectory(Path directory) {
            if (directory == null) { // It could be that somehow the path is null, for example
if it got deleted in the meantime
                initialDirectory = null;
            } else { // Dir must be a folder, not a file
                if (!Files.isDirectory(directory)) {
                    directory = directory.getParent();
                }
                // The lines above work also if the dir does not exist at all!
                // NULL is accepted by the filechooser as no inital path
                // Explicit null check, if somehow the parent is null, as Files.exists throws an
NPE otherwise
                if ((directory != null) && !Files.exists(directory)) {
                    directory = null;
                }
                initialDirectory = directory;
            }
            return this;
        }

        public Builder withInitialDirectory(String directory) {
            if (directory != null) {
                withInitialDirectory(Path.of(directory));
            } else {
                initialDirectory = null;
            }
            return this;
        }

        public Builder withInitialFileName(String initialFileName) {
            this.initialFileName = initialFileName;
            return this;
        }

        public Builder withDefaultExtension(FileChooser.ExtensionFilter extensionFilter) {
            defaultExtension = extensionFilter;
            return this;
```

```
        }

        public Builder withDefaultExtension(FileType fileType) {
            defaultExtension = FileFilterConverter.toExtensionFilter(fileType);
            return this;
        }

        public Builder withDefaultExtension(String description, FileType fileType) {
            defaultExtension = FileFilterConverter.toExtensionFilter(description, fileType);
            return this;
        }

        public Builder withDefaultExtension(String fileTypeDescription) {
            extensionFilters.stream()
                            .filter(type ->
type.getDescription().equalsIgnoreCase(fileTypeDescription))
                            .findFirst()
                            .ifPresent(extensionFilter -> defaultExtension = extensionFilter);

            return this;
        }

        public Builder addExtensionFilter(FileChooser.ExtensionFilter filter) {
            extensionFilters.add(filter);
            return this;
        }

        public Builder addExtensionFilter(List<FileChooser.ExtensionFilter> filters) {
            extensionFilters.addAll(filters);
            return this;
        }

        public Builder addExtensionFilter(FileType... fileTypes) {
            Stream.of(fileTypes)
                  .map(FileFilterConverter::toExtensionFilter)
                  .forEachOrdered(this::addExtensionFilter);
            return this;
        }

        public Builder addExtensionFilter(String description, FileType fileType) {
            extensionFilters.add(FileFilterConverter.toExtensionFilter(description, fileType));
            return this;
        }
    }
}
```

Location:
src/main/java/org/jabref/gui/util/FileDialogConfiguration.java

Reasoning:
Multiple construction methods under same method

Use Case Diagram:



Use Case “Operações relacionadas com entradas”:

Ator Principal: Utilizador
Ator(es) Secundário(s): Documento PDF ; Internet
Descrição: o utilizador pode adicionar, editar e remover uma entrada. Uma entrada pode ser adicionada manualmente, através de um ID (tal como o ISBN), um texto como referência ou através de um PDF. Editar uma entrada consiste em modificar o conteúdo de uma entrada. Remover uma entrada retira-a da biblioteca.

Reviews

Use case diagram by Mariana

simple, clear and accurate. approved.

Code smells

1 by Débora

looks good. approved.

2 by Gonçalo
looks good. approved.

3 by João
looks good. approved.

Design patterns

1 by Débora
description could be more detailed. approved.

2 by Gonçalo
description could be more detailed. approved.

3 by João
description could be more detailed. approved.