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 |

**Legend:**

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 pathsthrough 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:

Text

Description automatically generated

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:

Text

Description automatically generated

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

Code snippet:

Text

Description automatically generated

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:

Text

Description automatically generated

**Code smell** 3- Unused "private" fields should be removed

Code snippet:

Text

Description automatically generated

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:

Graphical user interface, text

Description automatically generated

Design Patterns

# Factory:

Code:

Text

Description automatically generated

Text

Description automatically generated  
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:

Text

Description automatically generated

Text

Description automatically generated

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:Text

Description automatically generated

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

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% |

**Legend:**

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

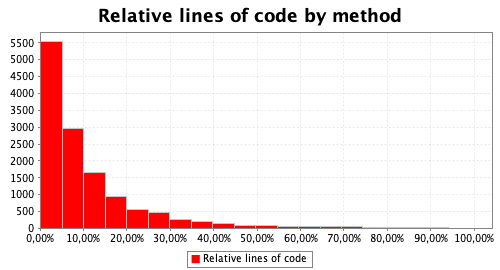
Analysis of the collected metrics:

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:

Text

Description automatically generated

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:

Graphical user interface, text

Description automatically generated

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

Code snippet:

Text

Description automatically generated

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:

Text

Description automatically generated

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

Code snippet:

Text

Description automatically generated

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:

Text

Description automatically generated

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

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 |

**Legend:**

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:

Text

Description automatically generated

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:

Text

Description automatically generated

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

Code:

Text

Description automatically generated

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:

Text

Description automatically generated

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.

Use Case Diagrams

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% |

**Legend:**

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 in 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.

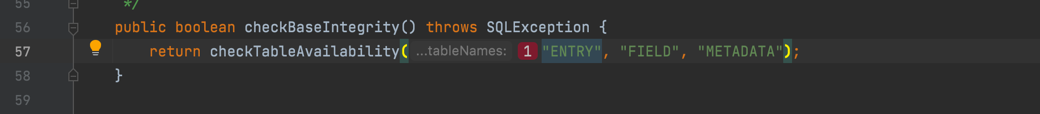
The number of visible methods is a measure of the class functionality. A low MHF indicates insufficiently abstracted implementation and a high MHF indicates very little functionality. The project has MHF = 36,93%, which isn’t too high but also not too low, so we may conclude that it is an acceptable 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:



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:

Text

Description automatically generated

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

Code snippet:

Text

Description automatically generated

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:

Text

Description automatically generated

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

Code snippet:

Text

Description automatically generated

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:

Text

Description automatically generated

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.*toCollection*(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*(LinkedHashSet::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*(LinkedHashSet::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*(LinkedHashSet::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

# 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 |

**Legend:**

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: