Pedro Lourenço 57577

Code Smell – 1

Data Class

This code smell occurs when the class is too small. These classes only have getter and setter methods and no real funcionalities.

Code Snippet:

```
public class Customer {

    private String firstName;
    private String lastName;
    private String title;
    private String house;
    private String street;
    private String city;
    private String postcode;
    private String country;


    public String getHouse() {
        return house;
    }

    public void setHouse(String house) {
        this.house = house;
    }

    public String getStreet() {
        return street;
    }
```

Code Location:

The classe is in the folloying path:

"src-gen\main\java\org\jabref\logic\importer\fileformat\medline\NameOfSubstance.java"

A new class is created with two instance variables:

```java
public class NameOfSubstance {

    @XmlValue
    protected String content;
    @XmlAttribute(name = "UI", required = true)
    @XmlSchemaType(name = "anySimpleType")
    protected String ui;
```

The class only have four methods (getters and setters):

```java
public String getContent() { return content; }
public void setContent(String value) { this.content = value; }
public String getUI() { return ui; }
public void setUI(String value) { this.ui = value; }
```

Justification:

As shown in the images, the class NameOfSubstance have nothing more than two instance variables and respective getters and setters.

Refactoring proposal:

This code smell may indicate that this is not a necessary class. We have two solutions: try to complete the class with real functionalaties (if there's any funcionality to add) or only manipulate the data in the classes that use it.

Code Smell – 2

Shotgun Surgery

This code smell occurs when a change needs to be made and then a lot of other classes need to be touched.

Code Location:

The classe is in the folloying path:

"src/main/java/org/jabref/model/entry/field.StandartField"

Code Snippet:

As we saw in the dependency metrics (Dpt metric) the class StandardField has a lot of dependents (322).

```
public enum StandardField implements Field {

    ABSTRACT( name: "abstract"),
    ADDENDUM( name: "addendum"),
    ADDRESS( name: "address"),
    AFTERWORD( name: "afterword", FieldProperty.PERSON_NAMES),
    ANNOTE( name: "annote"),
    ANNOTATION( name: "annotation"),
    ANNOTATOR( name: "annotator", FieldProperty.PERSON_NAMES),
    ARCHIVEPREFIX( name: "archiveprefix"),
    ASSIGNEE( name: "assignee", FieldProperty.PERSON_NAMES),
```

An example of a dependency in class FieldsFactory:

```
public static List<Field> getIdentifierFieldNames() {
    return Arrays.asList(StandardField.DOI, StandardField.EPRINT, StandardField.PMID);
}
```

Justification:

As explained above, the StardardField have a lof of dependents and if we to change this class we might have to change some of the 322 classes in question.

Refactoring proposal:

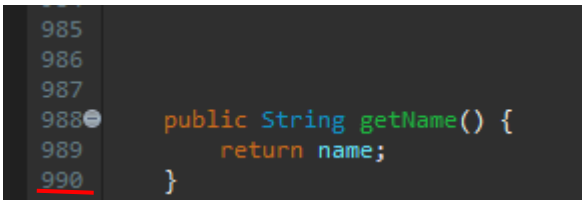
This code smell inidicates that we should reduce the amount of methods in different classes, but we should be careful not to move too many methods to just a few methods. We should do that beacause the class StandardFields have 322 dependents.

Code Smell – 3

Large Class

This code smell occurs when the class is bigger then it is supposed to be.

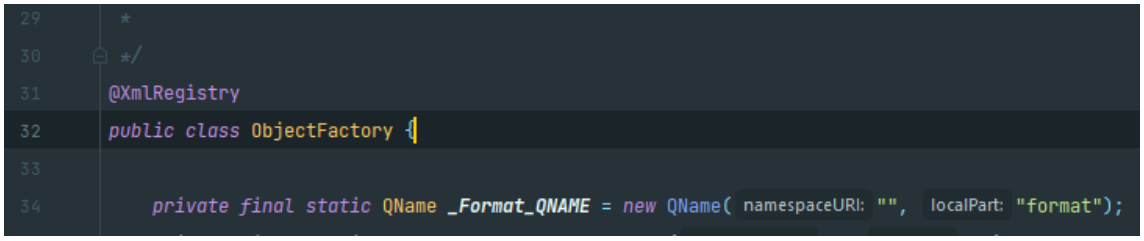
Code Snippet:



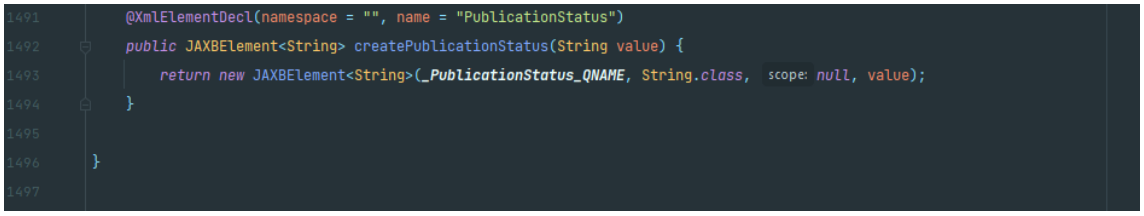
Code Location:

The classe is in the following path:

"src-gen\main\java\org\jabref\logic\importer\fileformat\medline\ObjectFactory"



•••



Justification:

As shown in the images, the class ObjectFactory is much bigger than the normal (1497).

Refactoring proposal:

The class ObjectFactory is used for the Factory Pattern (used to hide the creation of objects). However, it should be subdivided in smaller factories so we don't have class with 1497 lines.

Dependency Metrics

The dependency metrics are subdivided in Class Metrics, Interface Metrics and Package Metrics. Each one is composed by seven metrics (Package Metrics only have four metrics), which are *number of cyclic dependencies* (Cyclic), *number of dependencies* (Dcy), *number of transitive dependencies* (Dcy*), *number of dependents* (Dpt), *number of transitive dependents* (Dpt*), *number of package dependencies* (PDcy) and *number of dependent packages* (PDpt). The four Metrics that are a part of Package Metrics are Cyclic, PDcy, PDpt and PDpt* (number of transitively dependent packages).

The chosen ones to be analised are Cyclic, Dcy and Dpt (in Class Metrics).

Cyclic Metric

The number of cyclic dependencies calculates the number of classes which each class directly or indirectly depends on, and which in turn directly or indirectly depend on it. This means that there is a cycle between two or more classes.

Class	Value	Source path
ArgumentProcessor	784	src/main/java/org/jabref/cli/ArgumentProcessor
AuxCommandLine	784	src/main/java/org/jabref/cli/AuxCommandLine
JabRefCLI	784	src/main/java/org/jabref/cli/JabRefCLI
ClipBoardManager	784	src/main/java/org/jabref/gui/ClipBoardManager
DefaultInjector	784	src/main/java/org/jabref/gui/DefaultInjector

There are a lot of classes with high cyclic dependencies, these are just some exemples. This makes the system less maintable beacause understanding and testing the code is harder.

This might indicate the code smell Inappropriate Intimacy (when classes depend too much on each other).

Dcy Metric

Calculates the number of classes which each class directly depends on. A higher value means there are a lot of classes that can be changed and affect the class we are talking about.

Class	Value	Source path
JabRefFrame	115	src/main/java/org/jabref/gui/JabRefFrame
JabRefPreferences	101	src/main/java/org/jabref/preferences/JabRefPreferences
LayoutEntry	86	src/main/java/org/jabref/logic/layout/LayoutEntry
MedlineImporter	66	src/main/java/org/jabref/logic/importer/fileformat/MedlineImporter
ArgumentProcessor	59	src/main/java/org/jabref/cli/ArgumentProcessor
LibraryTab	55	src/main/java/org/jabref/gui/LibraryTab

There are six classes with fifty or more dependencies. There are 115 that can affect JabRefFrame class when they are changed.

This might indicate the code smell Shotgun Surgery (when a change is done, and a lot of classes need to be touched). JabRef class is affected 115 other classes.

Dpt Metric

The number of dependents calculates the number of classes which directly depend on each class. A high value of dependents might indicate that there are a lot of classes needing changes.

Class	Value	Source path
BibEntry	591	src/main/java/org/jabref/model/entry/BibEntry
Localization	392	src/main/java/org/jabref/logic/l19n/Localization
StandardField	322	src/main/java/org/jabref/model/entry/field.StandartField
BibDatabaseContext	256	src/main/java/org/jabref/model/database/BibDatabaseContext
BibDatabase	227	src/main/java/org/jabref/model/database/BibDatabase

There are five classes with 200 or more dependents, which is too much. For exemple, BibEntry have 591 classes depending on it.

This might indicate the code smell Shotgun Surgery (when a change is done, and a lot of classes need to be touched). BibEntry class affect 591 other classes.

Design Pattern – 1

Template

Firstly, defines an algorithm generaly in a superclass and let some steps to be defined in the subclasses. This tecnique is used when we have two or more classes with similar funcionalities. This is a good soluction beacause is better than make changes in two diferente places.

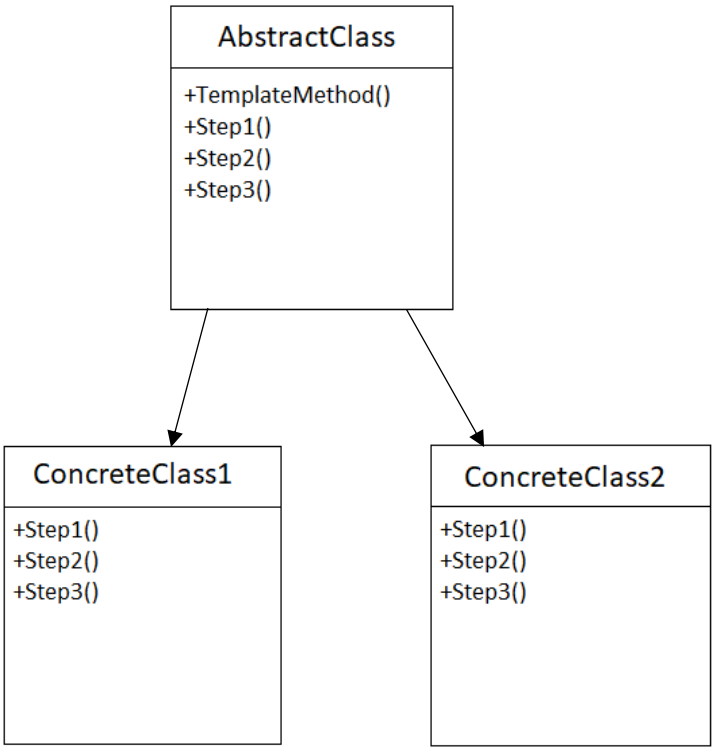
Code Snippet:

```
public abstract class PastaDish {
        final void makeRecipe() {
                boilWater();
                addPasta();
                cookPasta();
                drainAndPlate();
                addSauce();
                addProtein();
                addGarnish();
        }

        abstract void addPasta();
        abstract void addSauce();
        abstract void addProtein();
        abstract void addGarnish();

        private void boilWater() {
                System.out.println("Boiling water");
        }
        ...
}
```

```
public class SpaghettiMeatballs extends PastaDish {
        public void addPasta() {
                System.out.println("Add spaghetti");
        }
        public void addProtein() {
                System.out.println("Add meatballs");
        }
        public void addSauce() {
                System.out.println("Add tomato sauce");
        }
        public void addGarnish() {
                System.out.println("Add Parmesan cheese");
        }
}
```

```
public class PenneAlfredo extends PastaDish {
        public void addPasta() {
                System.out.println("Add penne");
        }
        public void addProtein() {
                System.out.println("Add chicken");
        }
        public void addSauce() {
                System.out.println("Add Alfredo sauce");
        }
        public void addGarnish() {
                System.out.println("Add parsley");
        }
}
```

Structure:



Code Location:

The classes are in the folloying path:

"src\main\java\org\jabref\gui\entryeditor\FieldsEditorTab.java"

A new abstract class (superClass) is created:

```
abstract class FieldsEditorTab extends EntryEditorTab {
```

A new abstract method is created:

```
protected abstract Set<Field> determineFieldsToShow(BibEntry entry);
```

"src\main\java\org\jabref\gui\entryeditor\OptionalFieldsTabBase"

Method determideFieldsToShow(BibEntry entry):

```
protected Set<Field> determineFieldsToShow(BibEntry entry) {
    Optional<BibEntryType> entryType = entryTypesManager.enrich(entry.getType(), databaseContext.getMode());
    if (entryType.isPresent()) {
        Set<Field> allKnownFields = entryType.get().getAllFields();
        Set<Field> otherFields = entry.getFields().stream().filter(field → !allKnownFields.contains(field)).collect(Collectors.toCollection(LinkedHashSet::new));

        otherFields.removeAll(entryType.get().getDeprecatedFields());
        otherFields.removeAll(entryType.get().getOptionalFields().stream().map(BibField::getField).collect(Collectors.toSet()));
        otherFields.remove(InternalField.KEY_FIELD);
        otherFields.removeAll(customTabFieldNames);
        return otherFields;
    } else {
        // Entry type unknown → treat all fields as required
        return Collections.emptySet();
    }
}
```

"src\main\java\org\jabref\gui\entryeditor\OthersFieldsTab"

Method determideFieldsToShow(BibEntry entry):

```
protected Set<Field> determineFieldsToShow(BibEntry entry) {
    Optional<BibEntryType> entryType = entryTypesManager.enrich(entry.getType(), databaseContext.getMode());
    if (entryType.isPresent()) {
        if (isPrimaryOptionalFields) {
            return entryType.get().getPrimaryOptionalFields();
        } else {
            return entryType.get().getSecondaryOptionalNotDeprecatedFields();
        }
    } else {
        // Entry type unknown → treat all fields as required
        return Collections.emptySet();
    }
}
```

Justification:

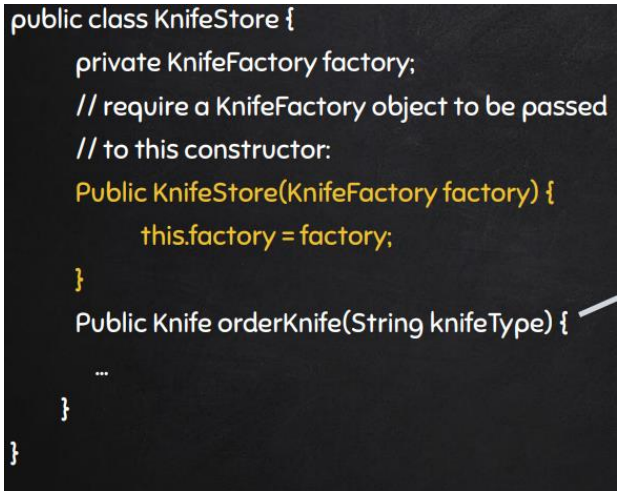
As shown in the superclass(FieldsEditorTab), it has to be abstract and have at least one abstract method, with no implementation. This implementation must be done in the subclasses (OthersFieldsTab and OptionFieldsTabBase) and it should be different from one class to another.

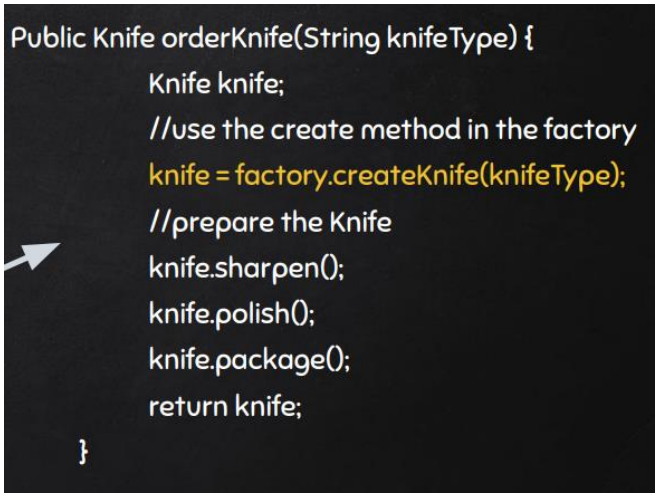
Design Pattern – 2

Factory

The purpose is to hide the creation of instances. With this pattern, the code does not depend on concrete classes. A specific class can now be replaced with no impact in the client code.

Code Snippet:





Code Location:

The classes are in the folloying path:

"src-gen\main\java\org\jabref\logic\importer\fileformat\medline\ObjectFactory"

A lot of *QName*'s are created, with diferent *localPart*'s:

```
private final static QName _Format_QNAME = new QName( namespaceURI: "",  localPart: "format");
private final static QName _B_QNAME = new QName( namespaceURI: "",  localPart: "b");
private final static QName _I_QNAME = new QName( namespaceURI: "",  localPart: "i");
private final static QName _Sup_QNAME = new QName( namespaceURI: "",  localPart: "sup");
private final static QName _Sub_QNAME = new QName( namespaceURI: "",  localPart: "sub");
private final static QName _U_QNAME = new QName( namespaceURI: "",  localPart: "u");
private final static QName _Year_QNAME = new QName( namespaceURI: "",  localPart: "Year");
```

Different *QName*'s are used to create diferent *JAXBElement<Text>*'s:

```
public JAXBElement<Text> createFormat(Text value) {
    return new JAXBElement<Text>(_Format_QNAME, Text.class,  scope: null, value);
}
```

```
public JAXBElement<Text> createI(Text value) { return new JAXBElement<Text>(_I_QNAME, Text.class,  scope: null, value); }
```

```
public JAXBElement<Text> createB(Text value) { return new JAXBElement<Text>(_B_QNAME, Text.class,  scope: null, value); }
```

Justification:

As we can see in the images, the class *ObjectFactory* is used to hide the creation of *JAXElement<Text>* classes. In the first image, we create all the *QName's* we need. Then, we use it to create all kinds of *JAXElement< Text>.*

Design Pattern – 3

Strategy

In Strategy pattern, objects are created and they might have multiple strategies and algoritms. The strategy object changes the executing algorithm of the context object, depending on which algorithm is chosen.

Code Snippet:

Interface:

```
public interface ICalculateInterface {

      int Calculate(int value1, int value2);
```

Algorithm Minus:

```
public class Minus {

     public int Calculate(int value1, int value2) {
            return value1 - value2;
         }
}
```

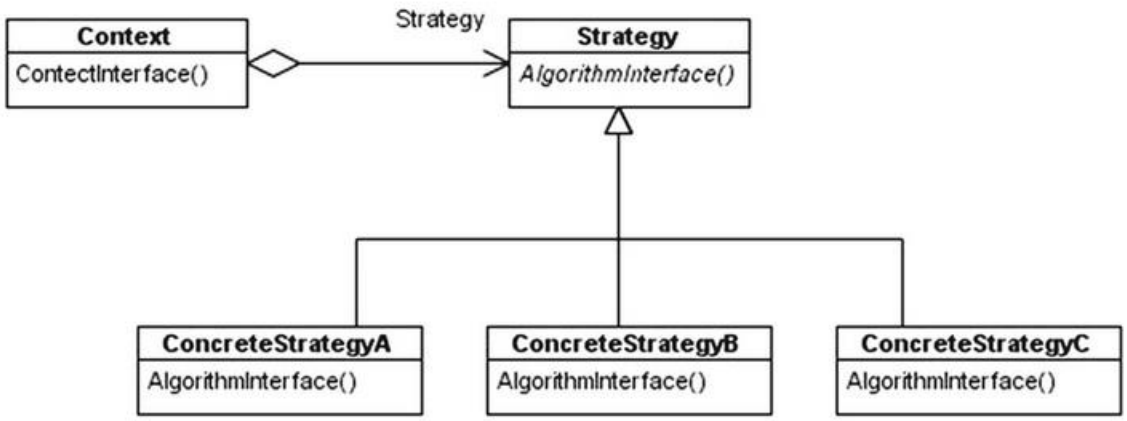
Algorithm Plus:

```
public class Plus {

     public int Calculate(int value1, int value2) {
        return value1 + value2;
     }
}
```

Client Class:

```
public class CalculateClient {

    private ICalculateInterface calculateInterface;

    public CalculateClient(ICalculateInterface strategy) {
        calculateInterface = strategy;
    }

    public int Calculate(int value1, int value2) {
        return calculateInterface.Calculate(value1, value2);
    }
}
```

Structure:



Code Location:

The classes are in the folloying path:

"src-gen\main\java\org\jabref\gui\autocompleter"

The interface is created with the method analyze():

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

Two different classes with the same interface implemente the method analize() in different ways (with different algorithms):

First algorithm:

```
public class AppendWordsStrategy implements AutoCompletionStrategy {

    protected String getDelimiter() { return " "; }

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

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

Second algorithm:

```
public class ReplaceStrategy implements AutoCompletionStrategy {

    @Override
    public AutoCompletionInput analyze(String input) { return new AutoCompletionInput( prefix: "", input); }
}
```

Client Class that chose any algorithm to use beacause of the interface:

```java
public class AutoCompletionTextInputBinding<T> extends AutoCompletionBinding<T> {

    /**
     * String converter to be used to convert suggestions to strings.
     */
    private StringConverter<T> converter;
    private AutoCompletionStrategy inputAnalyzer;
```

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

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

    getCompletionTarget().textProperty().addListener(textChangeListener);
    getCompletionTarget().focusedProperty().addListener(focusChangedListener);
}
```

Justification:

As we can see in the images, there is a variable (inputAnalyzer) that use the interface type (AutoCompletionStrategy). It means that this variable can save any algorithm, depending on which one it recives.

Use case diagram by Pedro Lourenço

User case: Write Directory

ID: 1

Description: The User writes the directory

Main actor: User

Secondary actors: None

Use case: Choose type file

ID: 2

Description: The user choose the file of the type to search

Main actor: User

Secondary actors: None

Use case: Any file

ID: 3

Description: One of the types of files the user can choose to search

Main actor: User

Secondary actors: None

Use case: PDF file

ID: 4

Description: One of the types (PDF) of files the user can choose to search

Main actor: User

Secondary actors: None

Use case: Bibtex library file

ID: 5

Description: One of the types (Bibtex library) of files the user can choose to search

Main actor: User

Secondary actors: None

Use case: Chose when was the last edit

ID: 6

Description: The user choose when was the last edited

Main actor: User

Secondary actors: None

Use case: All time

ID: 7

Description: One of the time limits (All time) the user can choose for the last edit

Main actor: User

Secondary actors: None

Use case: Last Year

ID: 8

Description: One of the time limits (Last Year) the user can choose for the last edit

Main actor: User

Secondary actors: None

Use case: Last Month

ID: 9

Description: One of the time limits (Last Month) the user can choose for the last edit

Main actor: User

Secondary actors: None

Use case: Last Week

ID: 10

Description: One of the time limits (Last Week) the user can choose for the last edit

Main actor: User

Secondary actors: None

Use case: Last Day

ID: 11

Description: One of the time limits (Last Day) the user can choose for the last edit

Main actor: User

Secondary actors: None

Use case: Sort by

ID: 12

Description: The user choose how the search is sorted

Main actor: User

Secondary actors: None

Use case: Newest first

ID: 13

Description: sort the search starting for the newest file

Main actor: User

Secondary actors: None

Use case: Oldest first

ID: 14

Description: sort the search starting for the oldest file

Main actor: User

Secondary actors: None

Use case: Default

ID: 15

Description: sort the search by the normal order

Main actor: User

Secondary actors: None

Use case: Search

ID: 16

Description: The user initiates the search

Main actor: User

Secondary actors: System

Use case: Show search results

ID: 17

Description: The system show the results after the user initiates the search

Main actor: System

Secondary actors: User

Use case: Export

ID: 18

Description: The system export the data after the user initiates the search

Main actor: System

Secondary actors: User

Use case: Import

ID: 19

Description: The system import the data after the user initiates the search

Main actor: System

Secondary actors: User

Bernardo Reis 57802

Code Smells Element 1 – Long Parameter List

This code smells occurs when there is a function with a long list of parameters.

Code Snippet

```
public class OptionalFieldsTab extends OptionalFieldsTabBase {
    public OptionalFieldsTab(BibDatabaseContext databaseContext,
                             SuggestionProviders suggestionProviders,
                             UndoManager undoManager,
                             DialogService dialogService,
                             PreferencesService preferences,
                             StateManager stateManager,
                             BibEntryTypesManager entryTypesManager,
                             ExternalFileTypes externalFileTypes,
                             TaskExecutor taskExecutor,
                             JournalAbbreviationRepository journalAbbreviationRepository) {
        super(
                Localization.lang( key: "Optional fields"),
                isPrimaryOptionalFields: true,
                databaseContext,
                suggestionProviders,
                undoManager,
                dialogService,
                preferences,
                stateManager,
                entryTypesManager,
                externalFileTypes,
                taskExecutor,
                journalAbbreviationRepository
        );
    }
}
```

Code Location

The path where this code smells is found (class path):

"src\main\java\org\jabref\logic\crawler\Crawler.java"

```
/**
 * Creates a crawler for retrieving studies from E-Libraries
 *
 * @param studyRepositoryRoot The path to the study repository
 */
public Crawler(Path studyRepositoryRoot, SlrGitHandler gitHandler, GeneralPreferences generalPreferences,
               ImportFormatPreferences importFormatPreferences, SavePreferences savePreferences,
               BibEntryTypesManager bibEntryTypesManager, FileUpdateMonitor fileUpdateMonitor) throws IllegalArgumentException, IOException, ParseException {
    studyRepository = new StudyRepository(studyRepositoryRoot, gitHandler, generalPreferences, importFormatPreferences, fileUpdateMonitor, savePreferences, bibEntryTypesManager);
    StudyDatabaseToFetcherConverter studyDatabaseToFetcherConverter = new StudyDatabaseToFetcherConverter(studyRepository.getActiveLibraryEntries(), importFormatPreferences);
    this.studyFetcher = new StudyFetcher(studyDatabaseToFetcherConverter.getActiveFetchers(), studyRepository.getSearchQueryStrings());
}
```

Justification

As it shows above, the constructor Crawler has a long list of parameters, exactly 10 parameters, sometimes having this long parameter list can be difficult to read or use the method.

Refactoring proposal

There are a few solutions to this problem. The first solution is to create sub methods to divide the parameters between them. The other solution is to put together some of the parameters with similarities into a new object.

Code Smells Element 2 – Long Method List

This code smells occurs when exist a large and complex method.

Code Snippet

```
74 @    private BibEntry parseEntry(Element e) {
75          String author = null;
76          String editor = null;
77          String title = null;
78          String publisher = null;
79          String year = null;
80          String address = null;
81          String series = null;
```

•••

```
418         }
419         if (volume != null) {
420             result.setField(StandardField.VOLUME, volume);
421         }
422         if (journal != null) {
423             result.setField(StandardField.JOURNAL, journal);
424         }
425         if (ppn != null) {
426             result.setField(new UnknownField( name: "ppn_GVK"), ppn);
427         }
428         if (url != null) {
429             result.setField(StandardField.URL, url);
430         }
431         if (note != null) {
432             result.setField(StandardField.NOTE, note);
433         }
434
435         if ("article".equals(entryType) && (journal != null)) {
436             result.setField(StandardField.JOURNAL, journal);
437         } else if ("incollection".equals(entryType) && (booktitle != null)) {
438             result.setField(StandardField.BOOKTITLE, booktitle);
439         }
440
441         return result;
442     }
```

Method parseEntry() has 361 lines

Code Location

The path where this code smells is found (class path):

"src\main\java\org\jabref\gui\openoffice\OOBibBase.java"

```
427  @      private List<String> refreshCiteMarkersInternal(List<BibDatabase> databases, OOBibStyle style)
428                 throws WrappedTargetException, IllegalArgumentException, NoSuchElementException,
429                 UndefinedCharacterFormatException, UnknownPropertyException, PropertyVetoException,
430                 CreationException, BibEntryNotFoundException {
431
432             List<String> cited = findCitedKeys();
433             Map<String, BibDatabase> linkSourceBase = new HashMap<>();
434             Map<BibEntry, BibDatabase> entries = findCitedEntries(databases, cited, linkSourceBase);
435
436             XNameAccess xReferenceMarks = getReferenceMarks();
437
438             List<String> names;
439             if (style.isSortByPosition()) {
440                 // We need to sort the reference marks according to their order of appearance:
441                 names = sortedReferenceMarks;
442             } else if (style.isNumberEntries()) {
443                 // We need to sort the reference marks according to the sorting of the bibliographic
444                 // entries:
445                 SortedMap<BibEntry, BibDatabase> newMap = new TreeMap<>(entryComparator);
446                 for (Map.Entry<BibEntry, BibDatabase> bibtexEntryBibtexDatabaseEntry : entries.entrySet()) {
447                     newMap.put(bibtexEntryBibtexDatabaseEntry.getKey(), bibtexEntryBibtexDatabaseEntry.getValue());
448                 }
449                 entries = newMap;
450                 // Rebuild the list of cited keys according to the sort order:
451                 cited.clear();
452                 for (BibEntry entry : entries.keySet()) {
```

•••

```
709                 if (hadBibSection && (getBookmarkRange(OOBibBase.BIB_SECTION_NAME) == null)) {
710                     // We have overwritten the marker for the start of the reference list.
711                     // We need to add it again.
712                     cursor.collapseToEnd();
713                     OOUtil.insertParagraphBreak(text, cursor);
714                     insertBookMark(OOBibBase.BIB_SECTION_NAME, cursor);
715                 }
716             }
717
718             List<String> unresolvedKeys = new ArrayList<>();
719             for (BibEntry entry : entries.keySet()) {
720                 if (entry instanceof UndefinedBibtexEntry) {
721                     String key = ((UndefinedBibtexEntry) entry).getKey();
722                     if (!unresolvedKeys.contains(key)) {
723                         unresolvedKeys.add(key);
724                     }
725                 }
726             }
727             return unresolvedKeys;
728         }
```

Justification

As it shows above is a very long method with 301 lines of code and a lot of complexity, probably this method can be reduced and optimized so it can be easier to read and understand.

Refactoring proposal

There are a few solutions to this problem. Probably the method is more complex and long than it needs to be so it can be reduced and optimized. Other solution is to divide the method into sub methods so to be easier to understand.

Code Smells Element 3 – Switch Statements

This code smells occurs when there are switch statements scattered throughout the code, some of which can be changed using polymorphism.

Code Snippet

```
public class Animal {
    private final static int DOG = 0;
    private final static int CAT = 1;

    private int type;

…

    public void say() {
        switch (type) {
        case DOG:
            System.out.println("Woof!");
            break;
        case CAT:
            System.out.println("Meow!");
            break;
        }
    }
}
```

```
public interface Animal {
    public void say();
}

public class Dog implements Animal {
    …
    public void say() {
        System.out.println("Woof!");
    }
}

public class Cat implements Animal {
    …
    public void say() {
        System.out.println("Meow!");
    }
}
```

Code Location

The path where this code smells is found (class path):

"src\main\java\org\jabref\gui\util\TooltipTextUtil.java"

```
public static Text createText(String textString, TextType textType) {
    Text text = new Text(textString);
    switch (textType) {
        case BOLD:
            text.getStyleClass().setAll("tooltip-text-bold");
            break;
        case ITALIC:
            text.getStyleClass().setAll("tooltip-text-italic");
            break;
        case MONOSPACED:
            text.getStyleClass().setAll("tooltip-text-monospaced");
            break;
        default:
            break;
    }
    return text;
}
```

Justification

As it shows above the switch statement can be removed because it is used to differentiate text types so instead the use of polymorphism is preferred.

Refactoring proposal

There is one solution to this problem. As said earlier, the use of switch statements to differentiate types can be replaced using polymorphism for example the types bold, italic and monospaced will be subclasses which will extend the super class text.

Design Pattern 1 - Template Pattern

This pattern is based on the existence of a superclass and several subclasses extended from that superclass. This pattern is used when there are several similar objects do the same things but in a different way, e.g. abstract class with one or more abstract methods that are implemented in the respective subclasses each in a different way.

This pattern saves a few lines of code and provides an understandable code without repeated methods.

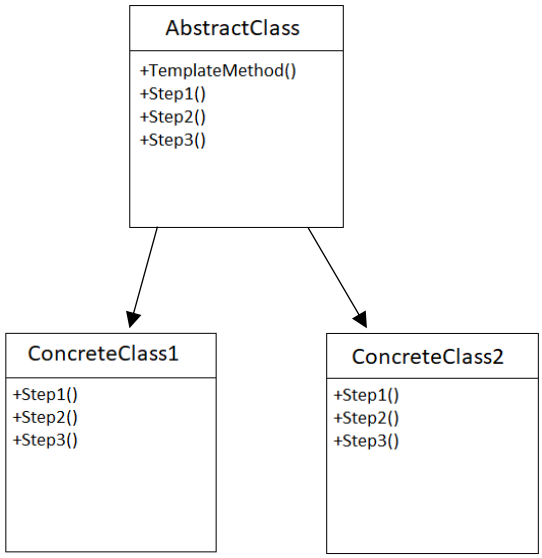
Code Snippet

```
public abstract class PastaDish {
        final void makeRecipe() {
                boilWater();
                addPasta();
                cookPasta();
                drainAndPlate();
                addSauce();
                addProtein();
                addGarnish();
}
abstract void addPasta();
abstract void addSauce();
abstract void addProtein();
abstract void addGarnish();
private void boilWater() {
        System.out.println("Boiling water")
}
```

```
public class SpaghettiMeatballs extends PastaDish {
        public void addPasta() {
                System.out.println("Add spaghetti");
        }
        public void addProtein() {
                System.out.println("Add meatballs");
        }
        public void addSauce() {
                System.out.println("Add tomato sauce");
        }
        public void addGarnish() {
                System.out.println("Add Parmesan cheese");
        }
}
```

```
  public class PenneAlfredo extends PastaDish {
          public void addPasta() {
                  System.out.println("Add penne");
          }
          public void addProtein() {
                  System.out.println("Add chicken");
          }
          public void addSauce() {
                  System.out.println("Add Alfredo sauce");
          }
          public void addGarnish() {
                  System.out.println("Add parsley");
          }
  }
```

Structure



Code Location

The classes are in a package in the following path:

"src/main/java/ org/jabref/logic/citationKeyPattern"

The abstract superclass is call AbstractCitationKeyPattern and have one abstract method call getLastLevelCitationKeyPattern(EntryType key). This method is implemented in several subclasses named: DatabaseCitationKeyPattern and GlobalCitationKeyPattern.

In AbstractCitationKeyPattern superclass:

```
public abstract List<String> getLastLevelCitationKeyPattern(EntryType key);
```

In DatabaseCitationKeyPattern subclass:

```
public class DatabaseCitationKeyPattern extends AbstractCitationKeyPattern {

    private final GlobalCitationKeyPattern globalCitationKeyPattern;

    public DatabaseCitationKeyPattern(GlobalCitationKeyPattern globalCitationKeyPattern) {
        this.globalCitationKeyPattern = globalCitationKeyPattern;
    }

    @Override
    public List<String> getLastLevelCitationKeyPattern(EntryType entryType) {
        return globalCitationKeyPattern.getValue(entryType);
    }
}
```

In GlobalCitationKeyPattern subclass:

```
public class GlobalCitationKeyPattern extends AbstractCitationKeyPattern {

    public GlobalCitationKeyPattern(List<String> bibtexKeyPattern) { defaultPattern = bibtexKeyPattern; }

    public static GlobalCitationKeyPattern fromPattern(String pattern) {
        return new GlobalCitationKeyPattern(split(pattern));
    }

    @Override
    public List<String> getLastLevelCitationKeyPattern(EntryType entryType) { return defaultPattern; }
}
```

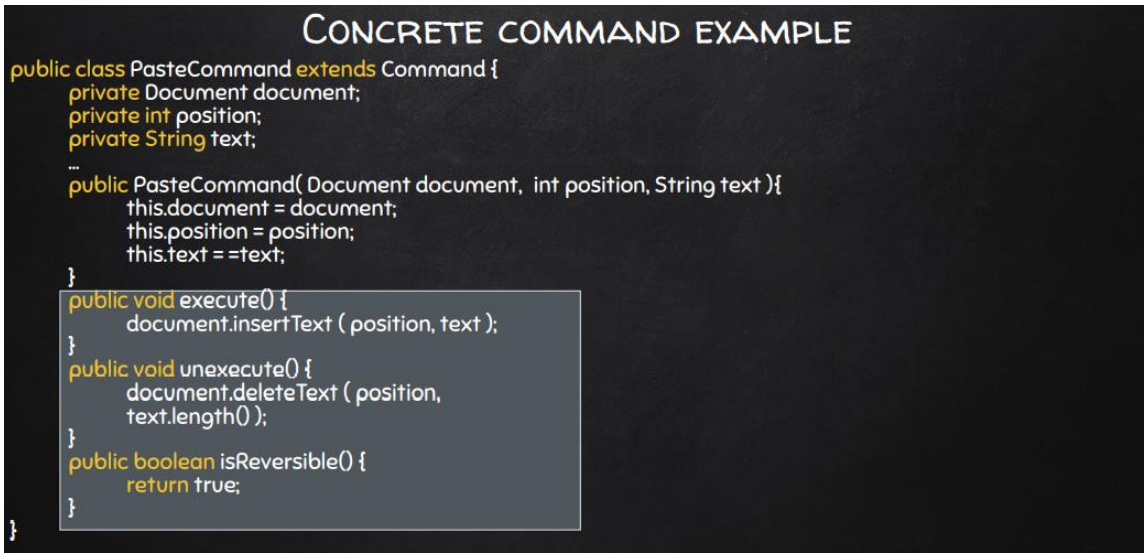
Justification

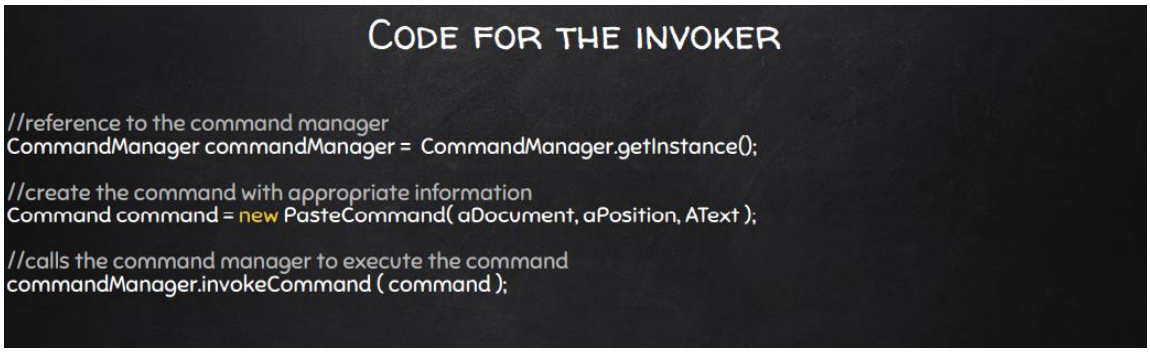
As shown above, the template pattern is being used because the project has an abstract class called AbstractCitationKeyPattern that has an abstract method called getLastLevelCitationKeyPattern(EntryType key). The abstract method is implemented in the various subclasses: DatabaseCitationKeyPattern and GlobalCitationKeyPattern.

Design Pattern 2 – Command Pattern

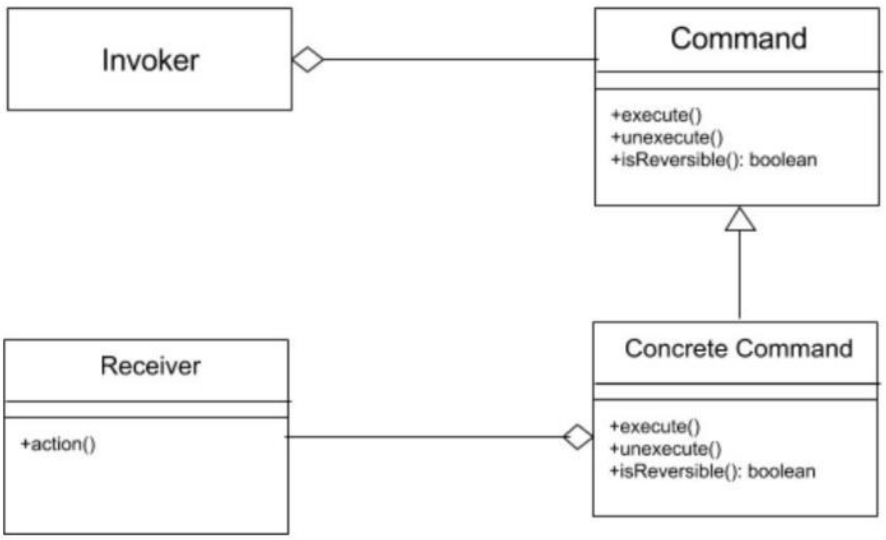
This pattern consists of using a class as a command, this can be beneficial for executing and initializing a set of actions. With a command class you can manipulate these commands from the invoker. Commands can be saved , reversed, or executed in an order.

Code Snippet





Structure



Code Location

The pattern is found in various destinations. The command interface is found in the following path:

"de/saxsys/mvvmfx/utils/commands/Command.java"

```
public interface Command {

    This method will be called when the command is invoked. This has to get called from FX Thread
    void execute();


    Determines whether the command can be executed in it's current state.
    Returns: true if the Command can executed, otherwise false.
    boolean isExecutable();

    See Also: isExecutable()
    ReadOnlyBooleanProperty executableProperty();

    Determines whether the command can not execute in it's current state.
    Returns: true if the Command can not execute, otherwise false.
    boolean isNotExecutable();

    See Also: isNotExecutable()
    ReadOnlyBooleanProperty notExecutableProperty();


    Signals whether the command is currently executing. This can be useful especially for commands that are
    executed asynchronously.
    Returns: true if the Command is running, otherwise false.
    boolean isRunning();
```

After some extends and implements, the concrete command class called NewEntryAction that has the following path:

"src\main\java\org\jabref\gui\importer\NewEntryAction.java"

```
public abstract class SimpleCommand extends CommandBase {
```

```
public abstract class CommandBase implements Command {
```

```
public class NewEntryAction extends SimpleCommand {

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

    private final JabRefFrame jabRefFrame;
    /**
     * The type of the entry to create.
     */
    private Optional<EntryType> type;
    private final DialogService dialogService;
    private final PreferencesService preferences;

    public NewEntryAction(JabRefFrame jabRefFrame, DialogService dialogService, PreferencesService preferences, StateManager stateManager) {
        this.jabRefFrame = jabRefFrame;
        this.dialogService = dialogService;
        this.preferences = preferences;

        this.type = Optional.empty();

        this.executable.bind(needsDatabase(stateManager));
    }

    public NewEntryAction(JabRefFrame jabRefFrame, EntryType type, DialogService dialogService, PreferencesService preferences, StateManager stateManager) {
        this(jabRefFrame, dialogService, preferences, stateManager);
        this.type = Optional.of(type);
    }

    @Override
    public void execute() {
        if (jabRefFrame.getBasePanelCount() <= 0) {
            LOGGER.error("Action 'New entry' must be disabled when no database is open.");
            return;
        }

        if (type.isPresent()) {
            jabRefFrame.getCurrentLibraryTab().insertEntry(new BibEntry(type.get()));
        } else {
            EntryTypeView typeChoiceDialog = new EntryTypeView(jabRefFrame.getCurrentLibraryTab(), dialogService, preferences);
            EntryType selectedType = dialogService.showCustomDialogAndWait(typeChoiceDialog).orElse( other: null);
            if (selectedType == null) {
                return;
            }
```

The final class is the invoker class, that it is called JabRefFrame. JabRefFrame has the following path:

“src/main/java/org/jabref/gui/JabRefFrame.java”

```
public class JabRefFrame extends BorderPane {

    public static final String FRAME_TITLE = "JabRef";

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

    private final SplitPane splitPane = new SplitPane();
    private final PreferencesService prefs = Globals.prefs;
    private final GlobalSearchBar globalSearchBar;
```

```
private void initKeyBindings() {
    addEventFilter(KeyEvent.KEY_PRESSED, event -> {
        Optional<KeyBinding> keyBinding = Globals.getKeyPrefs().mapToKeyBinding(event);
        if (keyBinding.isPresent()) {
            switch (keyBinding.get()) {
                case FOCUS_ENTRY_TABLE:
                    getCurrentLibraryTab().getMainTable().requestFocus();
                    event.consume();
                    break;
                case NEXT_LIBRARY:
                    tabbedPane.getSelectionModel().selectNext();
                    event.consume();
                    break;
                case PREVIOUS_LIBRARY:
                    tabbedPane.getSelectionModel().selectPrevious();
                    event.consume();
                    break;
                case SEARCH:
                    getGlobalSearchBar().focus();
                    break;
                case NEW_ARTICLE:
                    new NewEntryAction( jabRefFrame: this, StandardEntryType.Article, dialogService, prefs, stateManager).execute();
                    break;
                case NEW_BOOK:
                    new NewEntryAction( jabRefFrame: this, StandardEntryType.Book, dialogService, prefs, stateManager).execute();
                    break;
                case NEW_INBOOK:
                    new NewEntryAction( jabRefFrame: this, StandardEntryType.InBook, dialogService, prefs, stateManager).execute();
                    break;
                case NEW_MASTERSTHESIS:
                    new NewEntryAction( jabRefFrame: this, StandardEntryType.MastersThesis, dialogService, prefs, stateManager).execute();
                    break;
```

Justification

As shown above, the command pattern is being used because the project has a concrete command class called NewEntryAction that ends up implementing the command interface. The interface and the class NewEntryAction have all the method that a command pattern has, and the invoker(JabRefFrame) manipulates the command by creating a new one and executing it.

Design Pattern 3 – Adapter Pattern

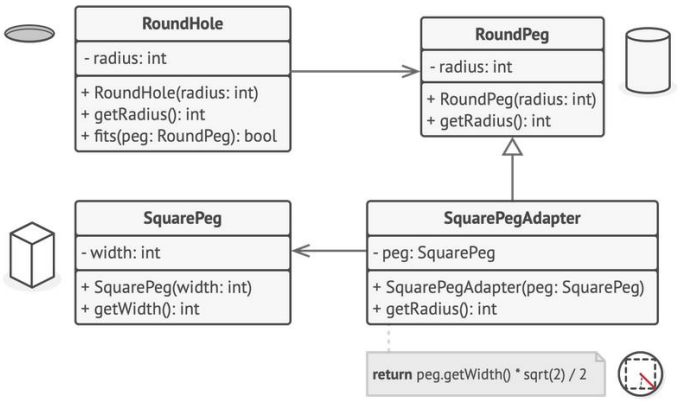
This pattern consists of having an adapter that facilitates communication between two existing systems by providing a compatible interface.

Code Snippet

The Adapter pretends to be a round peg, with a radius equal to a half of the square's diameter (in other words, the radius of the smallest circle that can accommodate the square peg).

```
// Say you have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Return the radius of the hole.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Return the radius of the peg.


// But there's an incompatible class: SquarePeg.
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Return the square peg width.


// An adapter class lets you fit square pegs into round holes.
// It extends the RoundPeg class to let the adapter objects act
// as round pegs.
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2


// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

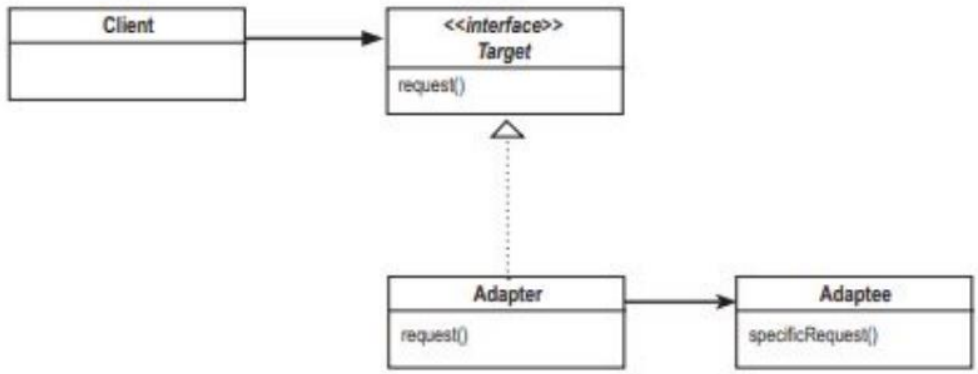
This example of the **Adapter** pattern is based on the classic conflict between square pegs and round holes.



Adapting square pegs to round holes.

The Adapter pretends to be a round peg, with a radius equal to a half of the square's diameter (in other words, the radius of the smallest circle that can accommodate the square peg).

Structure



Code Location

This adapter pattern is present in the project. The adapter class called LatexToUnicodeAdapter is in the following path:

"src\main\java\org\jabref\model\strings\LatexToUnicodeAdapter.java"

```java
public class LatexToUnicodeAdapter {

    private static final Pattern UNDERSCORE_MATCHER = Pattern.compile("_(?!\\{)");

    private static final String REPLACEMENT_CHAR = "\uFFFD";

    private static final Pattern UNDERSCORE_PLACEHOLDER_MATCHER = Pattern.compile(REPLACEMENT_CHAR);

    /**
     * Attempts to resolve all LaTeX in the String.
     *
     * @param inField a String containing LaTeX
     * @return a String with LaTeX resolved into Unicode, or the original String if the LaTeX could not be parsed
     */
    public static String format(String inField) {
        Objects.requireNonNull(inField);
        return parse(inField).orElse(Normalizer.normalize(inField, Normalizer.Form.NFC));
    }

    /**
     * Attempts to resolve all LaTeX in the String.
     *
     * @param inField a String containing LaTeX
     * @return an {@code Optional<String>} with LaTeX resolved into Unicode or {@code empty} on failure.
     */
    public static Optional<String> parse(String inField) {
        Objects.requireNonNull(inField);
        String toFormat = UNDERSCORE_MATCHER.matcher(inField).replaceAll(REPLACEMENT_CHAR);
        var parsingResult : Parsed<String, Object, String> = LaTeX2Unicode.parse(toFormat);
        if (parsingResult instanceof Parsed.Success) {
            String text = parsingResult.get().value();
            toFormat = Normalizer.normalize(text, Normalizer.Form.NFC);
            return Optional.of(UNDERSCORE_PLACEHOLDER_MATCHER.matcher(toFormat).replaceAll( replacement: "_"));
        }
        return Optional.empty();
    }
}
```

There are various classes that uses the adapter *LatexToUnicodeAdapter*, one example is the method *createAuthorList(String unparsedAuthors)* in the *BracketedPattern* class. The path of the *BracketedPattern* class is:

"src/main/java/org/jabref/logic/citationkeypattern/BracketedPattern.java"

```
private static AuthorList createAuthorList(String unparsedAuthors) {
    return AuthorList.parse(unparsedAuthors).getAuthors().stream()
                     .map((author) -> {
                         // If the author is an institution, use an institution key instead of the full name
                         String lastName = author.getLast()
                                                 .map(lastPart -> isInstitution(author) ?
                                                         generateInstitutionKey(lastPart) :
                                                         LatexToUnicodeAdapter.format(lastPart))
                                                 .orElse( other: null);
                         return new Author(
                                 author.getFirst().map(LatexToUnicodeAdapter::format).orElse( other: null),
                                 author.getFirstAbbr().map(LatexToUnicodeAdapter::format).orElse( other: null),
                                 author.getVon().map(LatexToUnicodeAdapter::format).orElse( other: null),
                                 lastName,
                                 author.getJr().map(LatexToUnicodeAdapter::format).orElse( other: null));
                     })
                     .collect(AuthorList.collect());
}
```

Justification

As shown above, the adapter pattern is being used because the project has an adapter class called *LatexToUnicodeAdapter* that is used by various other classes, for example the *BracketedPattern* class. This adapter facilitates the use of a conversion from Latex to Unicode that other classes use to format the strings.

Metrics Set 1 – Complexity Metrics

The complexity metrics measure the complexity of executable code within procedures. High complexity is more difficult to understand, and future code changes will take longer, so code development will be more complicated. Usually, high complexity is related to the high number of lines of code, but this is not always the case. There may be function with a few lines of code but with a lot of complexity.

The complexity metrics are sub divided in 5 different metrics each with different targets.

The first one, method metrics is composed by 4 metrics:

- CogC - Cognitive complexity.
- ev(G) - Essential cyclomatic complexity.
- iv(G) – Design complexity.
- v(G) – Cyclomatic complexity.

The second, class metrics is composed by 3 metrics:

- OCavg – Average operation complexity.
- OCmax – Maximum operation complexity.
- WMC - Weighted method complexity.

The last three, package metric, module metrics and project metrics are composed by 2 metrics:

- v(G)avg – Average cyclomatic complexity.
- v(G)tot – Total cyclomatic complexity.

For this analysis the chosen metrics are CogC, v(G) and iv(G), all from method metrics.

CogC metric - cognitive complexity metric

The cognitive complexity is used to know how difficult the code is to intuitively understand. Unlike Cyclomatic Complexity, which is used to know how difficult your code will be to test. For example, the cognitive complexity increases when there are breaks in the flow of the code.

Parts of code with higher cognitive complexity will be harder to read and understand.

Table 1 - Methods with the highest cognitive complexity:

Method	Value CogC	Source path
refreshCiteMarkersInternal (List<BibDatabase> databases, OOBibStyle style)	189	src\main\java\org\jabref\gui\openoffice\OOBibBase.java
importDatabase (BufferedReader reader)	188	src\main\java\org\jabref\logic\importer\fileformat\RisImporter.java
formatName (Author author, String format, Warn warn)	154	src\main\java\org\jabref\logic\bst\BibtexNameFormatter.java
parseEntry (Element e)	135	src\main\java\org\jabref\logic\importer\fileformat\GvkParser.java
importDatabase (BufferedReader reader)	131	src\main\java\org\jabref\logic\importer\fileformat\BiblioscapeImporter.java
getEntryFromPDFContent (String firstpageContents, String lineSeparator)	128	src\main\java\org\jabref\logic\importer\fileformat\PdfContentImporter.java

In this table, cognitive complexity is ordered from largest to smallest so it can be observed that method *refreshCiteMarkersInternal(…)* is the one with the highest cognitive complexity, it means the method is difficult to read and understand and to modify in the future, probably with a high cognitive complexity there might be a code smell, for example a long method, or a large class because usually complexity is related to number of lines of code (not always).

v(G) metric - cyclomatic complexity metric

The cyclomatic complexity also known as McCabe's complexity, it is a count of the linearly independent paths through the code. In other words, simply the number of decisions that the code needs to make. Decisions are caused by conditional statements(if, for, while).

Table 2 - Methods with the highest cyclomatic complexity:

Method	Value v(G)	Source path
transformSpecialCharacter (long c)	148	src\main\java\org\jabref\logic\layout\format\RTFChars.java
importDatabase (BufferedReader reader)	110	src\main\java\org\jabref\logic\importer\fileformat\RisImporter.java
getDescription (Field field)	96	src\main\java\org\jabref\gui\fieldeditors\FieldNameLabel.java
getSourceField (Field targetField, EntryType targetEntry, EntryType sourceEntry)	80	src\main\java\org\jabref\model\entry\BibEntry.java
parseEntry (Element e)	79	src\main\java\org\jabref\logic\importer\fileformat\GvkParser.java
importDatabase (BufferedReader reader)	76	src\main\java\org\jabref\logic\importer\fileformat\BiblioscapeImporter.java

In this table, cyclomatic complexity is ordered from largest to smallest so it can be observed that method *transformSpecialCharacter(…)* is the one with the highest cyclomatic complexity, it means that the method has a lot of conditions, cycles or functions calls so the method has many linearly independent paths, in this case 148, so there might be a code smell, for example a long method because how bigger the cyclomatic complexity is the more functions calls and conditions (if, for, while) it has.

iv(G) metric – design complexity metric

The design complexity is referent to how interlinked a methods control flow is with calls to other methods. Design complexity values vary between 1 to v(G) (cyclomatic complexity). Design complexity also represents the minimal number of tests necessary to exercise the integration of the method with the methods it calls.

Table 3 - Methods with the highest design complexity:

Method	Value iv(G)	Source path
importDatabase (BufferedReader reader)	102	src\main\java\org\jabref\logic\importer\fileformat\RisImporter.java
getDescription (Field field)	96	src\main\java\org\jabref\gui\fieldeditors\FieldNameLabel.java
parseEntry (Element e)	69	src\main\java\org\jabref\logic\importer\fileformat\GvkParser.java
importDatabase (BufferedReader reader)	67	src\main\java\org\jabref\logic\importer\fileformat\BiblioscapeImporter.java
getFieldValue (BibEntry entry, String pattern, Character keywordDelimiter, BibDatabase database)	55	src\main\java\org\jabref\logic\citationkeypattern\BracketedPattern.java
refreshCiteMarkersInternal(List<BibDatabase> databases, OOBibStyle style)	53	src\main\java\org\jabref\gui\openoffice\OOBibBase.java

In this table, design complexity is ordered from largest to smallest so it can be observed that method *importDatabase(..)* is the one with the highest design complexity. As said above, the design complexity is part of v(G) cyclomatic complexity, but focused more on the calls to other methods, so the highest design complexity, the method has more interconnections to other methods, and a higher number of methods calls.

Summary

All these metrics discussed above contribute to finding code smells about complexity, as said above the high complexity is related to the high number of lines of code normally, so it is related to the code smell long method. In this project some of my co-workers found this code smell, for example Martim Gouveia 57482.

Use case descriptions - Library Properties

Diagram General, Override default file directories and Library protection

Use case: Library encoding

ID: 1

Description : The user chooses a library encoding

Main actors: User

Secondary actors: N/A

Use case: Library encoding - UTF-8

ID: 2

Description : An example of what the user can choose for library encoding

Main actors: User

Secondary actors: N/A

Use case: Library encoding - Big5

ID: 3

Description : An example of what the user can choose for library encoding

Main actors: User

Secondary actors: N/A

Use case: Library encoding - GBK

ID: 4

Description : An example of what the user can choose for library encoding

Main actors: User

Secondary actors: N/A

Use case: Library mode

ID: 5

Description : The user chooses a library mode

Main actors: User

Secondary actors: N/A

Use case: Library mode - BibTex

ID: 6

Description : The user chooses BibTex for library mode

Main actors: User

Secondary actors: N/A

Use case: Library mode - biblatex

ID: 7

Description : The user chooses biblatex for library mode

Main actors: User

Secondary actors: N/A

Use case: General File Directory - Browse

ID: 8

Description : The user browses a new file directory for General File Directory

Main actors: User

Secondary actors: N/A

Use case: General File Directory - Textbox

ID: 9

Description : The user writes a new file directory for General File Directory

Main actors: User

Secondary actors: N/A

Use case: User-specific file directory - Browse

ID: 10

Description : The user browses a new file directory for User-specific file directory

Main actors: User

Secondary actors: N/A

Use case: User-specific file directory - Write

ID: 11

Description : The user writes a new file directory for User-specific file directory

Main actors: User

Secondary actors: N/A

Use case: Latex file directory - Browse

ID: 12

Description : The user browses a new file directory for Latex file directory

Main actors: User

Secondary actors: N/A

Use case: Latex file directory - Write

ID: 13

Description : The user writes a new file directory for Latex file directory

Main actors: User

Secondary actors: N/A

Use case: Refuse to save library before external changes have been reviewed

ID: 14

Description : The user decides to check box (activate or deactivate) the option.

Main actors: User

Secondary actors: N/A

Diagram Save sort order

Use case: order - Keep original order

ID: 15

Description : The user decides if he wants to keep the original order

Main actors: User

Secondary actors: N/A

Use case: order - Use current table sort order

ID: 16

Description : The user decides if he wants to use current table sort order

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order

ID: 17

Description : The user decides if he wants to use specified order

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - move up

ID: 18

Description : The user moves up the selected type in the order

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - move down

ID: 19

Description : The user moves down the selected type in the order

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - add type

ID: 20

Description : The user adds a new type

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - remove type

ID: 21

Description : The user removes a type

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - Descending

ID: 22

Description : The user chooses the general sort order of the line

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - Choose type

ID: 23

Description : The user chooses a new type to add

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - Choose type - author

ID: 24

Description : An example of a type to add

Main actors: User

Secondary actors: N/A

Use case: order - Use specified order - Choose type - Title

ID: 25

Description : An example of a type to add

Main actors: User

Secondary actors: N/A

Diagram Save actions

Use case: Enable field formatters

ID: 26

Description : The user chooses if he wants to enable or disable the field formatters

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Reset to recommended

ID: 27

Description : The user resets field formatters to recommended

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Remove all

ID: 28

Description : The user removes all the field formatters

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Add

ID: 29

Description : The user adds new field formatter

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Remove

ID: 30

Description : The user removes the selected field formatter

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Chose Formatter name

ID: 31

Description : The user chooses a new formatter name

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Chose Formatter name - Clear

ID: 32

Description : An example of a formatter name

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Chose Formatter name - Capitalize

ID: 33

Description : An example of a formatter name

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Chose Field name

ID: 34

Description : The user chooses a new field name

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Chose Field name - Abstract

ID: 35

Description : An example of a field name

Main actors: User

Secondary actors: N/A

Use case: Enable field formatters - Chose Field name - Annote

ID: 36

Description : An example of a field name

Main actors: User

Secondary actors: N/A

Pedro Perdigão - 58165

ES - Sprint 2 – Code Smells 1

Primitive Obsession

This code smell occurs when we rely too much on the use of built-in types (or primitives like int, long, float or strings). Although these primitives will be needed in the code, they should exist at the lowest levels of the code. To fix this may be possible to logically group some of them into their own class. Even better, move the behaviour associated with this data into the class too.

- **Code Snippet (example)**

Instead of creating a primitive for each method (first image) we can create some classes, in this case Adress and SocialSecurity (second image), and create there the related primitive methods, in order to create only two primitives (of type Adress and SocialSecurity) instead of seven like (String Adress, City, State, etc).

- **Code Location**

Path:

"src\main\java\org\jabref\gui\bibtexextractor\BibtexExtractor.java"

```
private final List<String> urls = new ArrayList<>();
private final List<String> authors = new ArrayList<>();
private String year = "";
private String pages = "";
private String title = "";
private boolean isArticle = true;
private String journalOrPublisher = "";
```

Justification

Like we can see is present a primitive obsession because there are too many primitives (String, boolean) that can be in one class, creating then only one variable in this one, which type is the class we created.

Refactoring Proposal

In this case we should create a class that stores the primitives and create methods to get those primitives and set them. This way we trade a lot of primitives to an object.

Example:

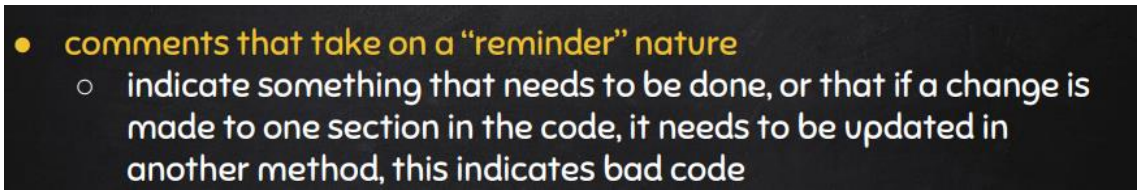
```
private Article article;
private String journalOrPublisher = "";

public class Article{
    private final List<String> urls;
    private final List<String> authors;
    private String year;
    private String pages;
    private String title;

    public Article(String year, String pages, String title){
        this.year = year;
        this.pages = pages;
        this.title = title;
        urls = new ArrayList<>();
        authors = new ArrayList<>();
    }

    public String getYear(){
        return year;
    }

    public void setYear(String year){
        this.year = year;
    }

    ...
}
```

ES - Sprint 2 – Code Smells 2

Comments

Some comments can be interpreted by “reminders”, indicates that something needs to be done, or that if a change is made in one section of the code, another section must be changed or updated, indicating bad code.

- **Code Snippet**



- **Code Location**

  Path:
  "src\main\java\org\jabref\logic\importer\fileformat\BiblioscapeImporter.java"

  Line: 183

```
    } else if ("C4".equals(entry.getKey())) {
        comments.add("Custom4: "
                + entry.getValue());
    } else if ("C5".equals(entry.getKey())) {
        comments.add("Custom5: "
                + entry.getValue());
    } else if ("C6".equals(entry.getKey())) {
        comments.add("Custom6: "
                + entry.getValue());
    } else if ("DE".equals(entry.getKey())) {
        hm.put(StandardField.ANNOTE, entry
                .getValue().toString());
    } else if ("CA".equals(entry.getKey())) {
        comments.add("Categories: "
                + entry.getValue());
    } else if ("TH".equals(entry.getKey())) {
        comments.add("Short Title: "
                + entry.getValue());
    } else if ("SE".equals(entry.getKey())) {
        hm.put(StandardField.CHAPTER, entry
                .getValue().toString());
        // else if (entry.getKey().equals("AC"))
        //   hm.put("",entry.getValue().toString());
        // else if (entry.getKey().equals("LP"))
        //   hm.put("",entry.getValue().toString());
    }
}
```

Justification

Like we can see there are 4 lines of comments which indicates a potential "reminder" for something that need to be implemented or updated.

Refactoring Proposal

In this case we should delete this comments, because they are just “reminders” for a potential implementation or update, or we make sure that we implement the rest of the method without need to update or make some alteration to another class.

ES - Sprint 2 – Code Smells 3

Duplicated Code

This code smell occurs when blocks of code exist in the design that are similar but have slight differences appearing in multiple places in the code. If is needed to implement an update is harder to do it, so we should implement only in one location, decreasing the chance of a block not being updated.

- **Code Snippet**

Occurs when blocks of code exist in the design that are similar, but have slight differences appearing in multiple places in the software.

Problem:

- if something needs to change, then the code needs to be updated in multiple places.
- This applies to adding functionalities, updating an algorithm, or fixing a bug.

Instead, if the code only needed to be updated in one location, it is easier to implement the change. It also reduces the chance that a block of code was missed in an update or change.

D.R.Y. principle, or "Don't Repeat Yourself"

Programs should be written so that they can perform the same tasks but with less code

```
function calculateTax(subtotal, country, state, taxrates) {
  let taxrate;
  if(country === 'US') {
    taxrate = taxrates[state];
  } else {
    taxrate = taxrates[country];
  }
  return subtotal + subtotal * taxrate;
}

function findTimeZone(country, state, zones) {
  let timezone;
  if(country === 'US') {
    timezone = zones[state];
  } else {
    timezone = zones[country];
  }
  return timezone;
}
```

- **Code Location**

  Path:
  "src\main\java\org\jabref\logic\importer\fileformat\BiblioscapeImporter.java"

```java
// depending on bibtexType, decide where to place the titleRT and
// titleTI
if (bibtexType.equals(StandardEntryType.Article)) {
    if (titleST != null) {
        hm.put(StandardField.JOURNAL, titleST);
    }
    if (titleTI != null) {
        hm.put(StandardField.TITLE, titleTI);
    }
} else if (bibtexType.equals(StandardEntryType.InBook)) {
    if (titleST != null) {
        hm.put(StandardField.BOOKTITLE, titleST);
    }
    if (titleTI != null) {
        hm.put(StandardField.TITLE, titleTI);
    }
} else {
    if (titleST != null) {
        hm.put(StandardField.BOOKTITLE, titleST);
    }
    if (titleTI != null) {
        hm.put(StandardField.TITLE, titleTI);
    }
}
```

Justification

If we notice the following part (“if(titleST != null){ hm.put(StandardField.TITLE, titleST);}”) repeats six times during the method changing only the parameter StandardField.TITLE and titleST.

Refactoring Proposal

In this case we could create a private method that receive StandardField.TITLE and titleST (and the other types) as a parameter and execute what he should.

The method should be:

```
if (bibtexType.equals(StandardEntryType.Article)) {
    putInMap(hm, StandardField.JOURNAL, StandardField.TITLE, titleST, titleTI);
} else if (bibtexType.equals(StandardEntryType.InBook)) {
    putInMap(hm, StandardField.BOOKTITLE, StandardField.TITLE, titleST, titleTI);
} else {
    putInMap(hm, StandardField.BOOKTITLE, StandardField.TITLE, titleST, titleTI);
}
```

```
private void putInMap(Map<Field, String> hm, Field field,  Field field2, String title, String title2){
    if (title != null) {
        hm.put(field, title);
    }
    if (title2 != null) {
        hm.put(field2, title2);
    }
}
```

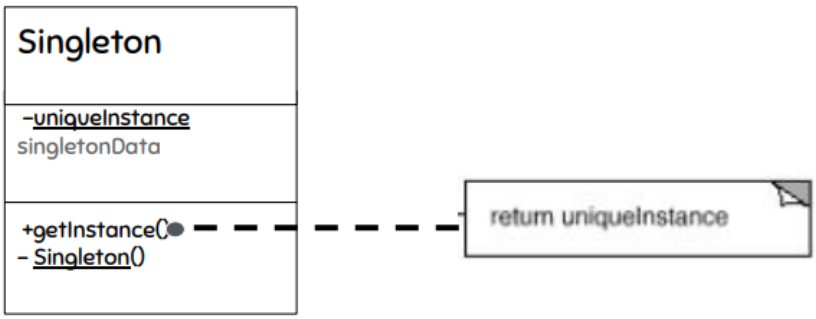
ES - Sprint 1 – Design Pattern (GOF) 1

Singleton

Describes a way to create an object. A class has only one instance (or limited few) and provide a global point to access it.

This pattern permits a “Lazy creation”, that means that the object is not created until is truly needed. As the object is not created until the getInstance() method is called, our program is more efficient. Especially, if the object is large than this pattern might be helpful.

- **Structure**



- **Singleton java example (Code Snippet)**

```
public class ExampleSingleton { // lazy construction

        // the class variable is null if no instance is instantiated
        private static ExampleSingleton uniqueInstance = null;

        private ExampleSingleton() {
                ...
        }
        // lazy construction of the instance
        public static ExampleSingleton getInstance() {
                if (uniqueInstance == null) {
                        uniqueInstance = new ExampleSingleton();
                }
                return uniqueInstance;
        }
...
}
```

Code location:

The class is in the following path:

"src\main\java\org\jabref\gui\extrenalfiletype\ExternalFileTypes.java"

- **Instance**

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

- **Constructor**

```
private ExternalFileTypes() { updateExternalFileTypes(); }
```

- **getInstance() method**

```
public static ExternalFileTypes getInstance() {
    if (ExternalFileTypes.singleton == null) {
        ExternalFileTypes.singleton = new ExternalFileTypes();
    }
    return ExternalFileTypes.singleton;
}
```

Justification:

As shown above the class (ExternalFileTypes) has only one instance ("*singleton*") and must be static.

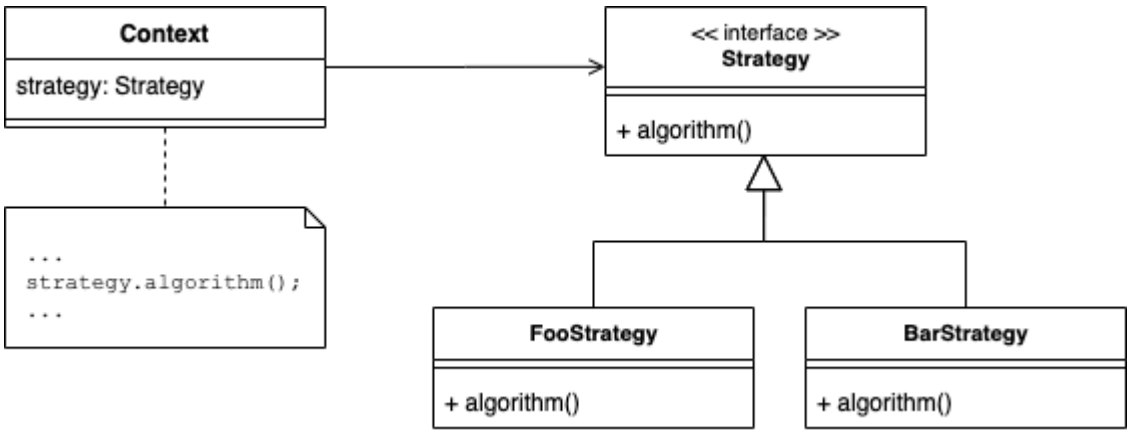
The constructor of class is private and the method getInstance() that instantiates the class "if" is not already instantiated. This provides global access to a class that is restricted to one instance.

ES - Sprint 2 – Design Pattern (GOF) 2

Strategy

This pattern enables to select an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

- **Structure**



- **Strategy java example (Code Snippet)**
    1. Algorithms Interface

```java
/* Encapsulated family of Algorithms
 * Interface and its implementations
 */
public interface IBrakeBehavior {
    public void brake();
}
```

    2. Algorithms classes

```java
public class BrakeWithABS implements IBrakeBehavior {
    public void brake() {
        System.out.println("Brake with ABS applied");
    }
}

public class Brake implements IBrakeBehavior {
    public void brake() {
        System.out.println("Simple Brake applied");
    }
}
```

    3. Class that changes algorithm

```java
/* Client that can use the algorithms above interchangeably */
public abstract class Car {
    private IBrakeBehavior brakeBehavior;

    public Car(IBrakeBehavior brakeBehavior) {
      this.brakeBehavior = brakeBehavior;
    }

    public void applyBrake() {
        brakeBehavior.brake();
    }

    public void setBrakeBehavior(IBrakeBehavior brakeType) {
        this.brakeBehavior = brakeType;
    }
}
```

Code location:

The class is in the following path:

"src\main\java\org\jabref\model\groups\WordKeywordGroup.java"

- **Interface that all Algorithms implement**

```
interface SearchStrategy {
    boolean contains(BibEntry entry);
}
```

- **Algorithm One**

```
class StringSearchStrategy implements SearchStrategy {
    Set<String> searchWords;

    StringSearchStrategy() {
        searchWords = new HashSet<>(StringUtil.getStringAsWords(searchExpression));
    }

    @Override
    public boolean contains(BibEntry entry) {
        Set<String> content = entry.getFieldAsWords(searchField);
        if (caseSensitive) {
            return content.containsAll(searchWords);
        } else {
            return containsCaseInsensitive(content, searchWords);
        }
    }
}
```

- **Algorithm Two**

```
class TypeSearchStrategy implements SearchStrategy {

    Set<EntryType> searchWords;

    TypeSearchStrategy() {
        searchWords = KeywordList.parse(searchExpression, keywordSeparator) KeywordList
                                 .stream() Stream<Keyword>
                                 .map(word -> EntryTypeFactory.parse(word.get())) Stream<EntryType>
                                 .collect(Collectors.toSet());
    }

    @Override
    public boolean contains(BibEntry entry) {
        return searchWords.stream()
                          .anyMatch(word -> entry.getType().equals(word));
    }
}
```

- **Algorithm Three**

```
class KeywordListSearchStrategy implements SearchStrategy {

    private final KeywordList searchWords;

    KeywordListSearchStrategy() { searchWords = KeywordList.parse(searchExpression, keywordSeparator); }

    @Override
    public boolean contains(BibEntry entry) {
        KeywordList fieldValue = entry.getFieldAsKeywords(searchField, keywordSeparator);
        return ListUtil.allMatch(searchWords, fieldValue::contains);
    }
}
```

- **Class that chooses what algorithm should use**

```
/**
 * Matches entries if a given field contains a specified word.
 */
public class WordKeywordGroup extends KeywordGroup implements GroupEntryChanger {

    protected final Character keywordSeparator;
    private final SearchStrategy searchStrategy;
    private final boolean onlySplitWordsAtSeparator;

    public WordKeywordGroup(String name, GroupHierarchyType context, Field searchField,
                            String searchExpression, boolean caseSensitive, Character keywordSeparator,
                            boolean onlySplitWordsAtSeparator) {
        super(name, context, searchField, searchExpression, caseSensitive);

        this.keywordSeparator = keywordSeparator;
        this.onlySplitWordsAtSeparator = onlySplitWordsAtSeparator;

        if (onlySplitWordsAtSeparator) {
            if (InternalField.TYPE_HEADER.equals(searchField)) {
                searchStrategy = new TypeSearchStrategy();
            } else {
                searchStrategy = new KeywordListSearchStrategy();
            }
        } else {
            searchStrategy = new StringSearchStrategy();
        }
    }
```

(The variable searchStrategy is of type SearchStrategy, that means it's the variable that saves the algorithm that should be used. At the end of the constructor_____, inside the if statement, the program choose what algorithm should use)

Justification:

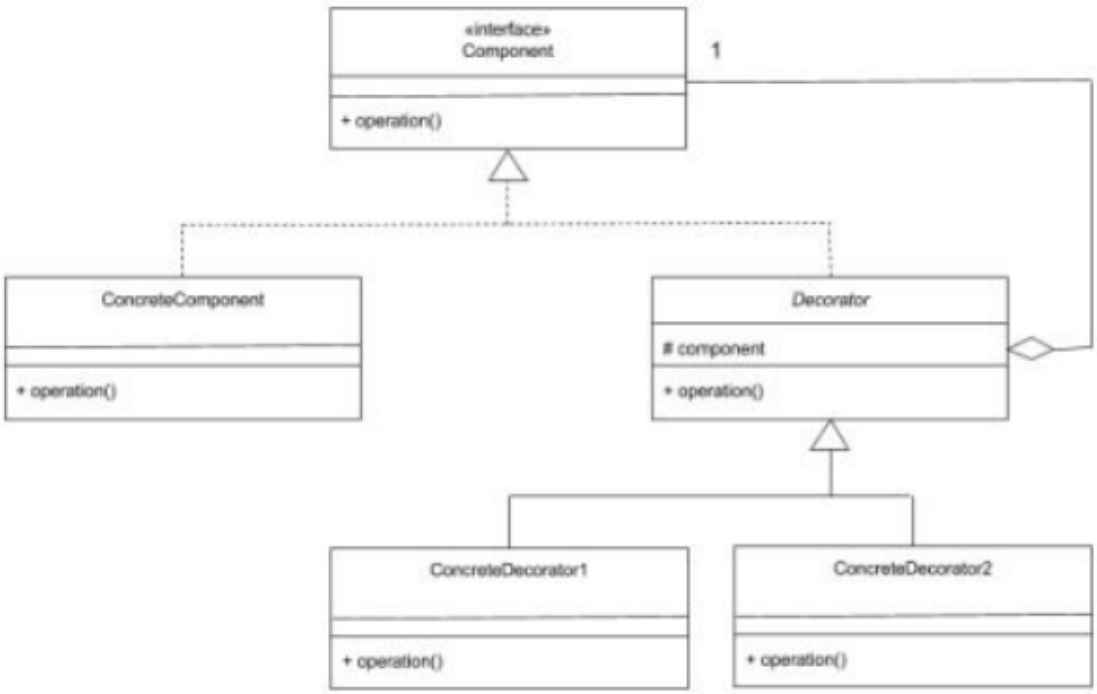
Like we saw above exist a variable, which type is the interface that is implemented by all the algorithms. This variable is used to control what algorithm should be used by the program.

ES - Sprint 2 – Design Pattern (GOF) 3

Decorator

Allows additional behaviours or responsibilities to be dynamically attached to an object, using aggregation/composition to combine behaviours at run time. This design pattern makes use of interfaces and inheritance, so that the classes conform to a common type whose instances can be stacked in a compatible way. This builds a coherent combination of behaviour overall.

- **Structure**



- **Decorator java example (Code Snippet)**
    1. Component Interface

```java
public interface Arma{

  public void montar();

}
```

    2. Concrete Component

```java
public class ArmaBase implements Arma{

  @Override
  public void montar(){
    System.out.println("Essa é uma arma base");
  }

}
```

    3. Decorator

```java
public class ArmaDecorator implements Arma{

  public Arma arma;

  public ArmaDecorator(Arma arma){
    this.arma = arma;
  }

  @Override
  public void montar(){
    this.arma.montar();
  }

}
```

4. Concrete Decorator 1

```java
public class Mira extends ArmaDecorator{

  public Mira(Arma arma){
    super(arma);
  }

  @Override
  public void montar(){
    super.montar();
    System.out.println("Adicionando mira a arma");
  }

}
```

5. Concrete Decorator 2

```java
public class Silenciador extends ArmaDecorator{

  public Silenciador(Arma arma){
    super(arma);
  }

  @Override
  public void montar(){
    super.montar();
    System.out.println("Adicionando silenciador a arma");
  }

}
```

Code location:

- **Component Interface**

  "src\main\java\org\jabref\model\search\SearchMatcher.java"

```java
@FunctionalInterface
public interface SearchMatcher {
    boolean isMatch(BibEntry entry);
}
```

- **Concrete Decorator**

  "src\main\java\org\jabref\model\search\matchers\NotMatcher.java"

```java
public class NotMatcher implements SearchMatcher {

    private final SearchMatcher otherMatcher;

    public NotMatcher(SearchMatcher otherMatcher) { this.otherMatcher = Objects.requireNonNull(otherMatcher); }

    @Override
    public boolean isMatch(BibEntry entry) { return !otherMatcher.isMatch(entry); }
}
```

- **Decorator**
  "src\main\java\org\jabref\model\search\matchers\MatcherSet.java"

```java
public abstract class MatcherSet implements SearchMatcher {

    protected final List<SearchMatcher> matchers = new ArrayList<>();

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

        MatcherSet that = (MatcherSet) o;

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

    @Override
    public int hashCode() { return Objects.hash(matchers); }

    public void addRule(SearchMatcher newRule) { matchers.add(Objects.requireNonNull(newRule)); }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("MatcherSet{");
        sb.append("matchers=").append(matchers);
        sb.append('}');
        return sb.toString();
    }
}
```

(matchers is the protected variable that stores a collection of type SearchMatcher (Component Interface) and the method isMatch (BibEntry entry) is implemented on the concrete decorator classes)

- **Concrete Decorator 1**

  "src\main\java\org\jabref\model\search\matchers\AndMatcher.java"

```java
public class AndMatcher extends MatcherSet {

    @Override
    public boolean isMatch(BibEntry entry) {
        return matchers.stream()
                       .allMatch(rule -> rule.isMatch(entry));
    }
}
```

- **Concrete Decorator 2**

"src\main\java\org\jabref\model\search\matchers\OrMatcher.java"

```java
public class OrMatcher extends MatcherSet {

    @Override
    public boolean isMatch(BibEntry entry) { return ListUtil.anyMatch(matchers, rule -> rule.isMatch(entry)); }
}
```

Justification:

Like we saw above exist a protected variable, which type is the interface. The class Decorator is abstract, and all the classes that extends this one (Concrete Decorators) implement the method isMatcher (BibEntry entry).

ES – Sprint 2 - Lines of Code Metrics

This metrics is used to measure the size of a program by counting the number of lines in the text of the program's source code, subdivided in Method metrics, Class metrics, Interface metrics, Package metrics, Module metrics, File type metrics and Project metrics. Each one is composed by the following metrics: 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), CLOC (rec) – comment lines of code (recursive), JLOC (rec) – Javadoc lines of code (recursive), LOC(rec) – lines of code (recursive), LOC_p (lines of product code, LOC_p(rec) – lines of product code (recursive), LOC_t (lines of test code), LOC_t(rec) – lines of test code (recursive), $NCLOC_p$ (non-comment lines of code(product)), $NCLOC_p$(rec) – non-comment lines of code (product, recursive), $NCLOC_t$ (non-comment lines of code (test)), L(Groovy) – lines of Groovy, L(HTML) – lines of HTML, L(J) – lines of Java, L(XML) – lines of XML, and the last one L(KT) – lines of Kotlin.

In this case we are only consider CLOC, JLOC and LOC in Class metrics.

- CLOC – comment lines of code

  Consists in the number of lines of comments present in the class.

Class	CLOC	Path
HTMLUnicodeConversionMaps	824	src\main\java\org\jabref\logic\util\strings\HTMLUnicodeConversionMaps.java
BracketedPattern	438	src\main\java\org\jabref\logic\citationkeypattern\BracketedPattern.java
…	…	…
VersionPreferences	0	src\main\java\org\jabref\preferences\VersionPreferences.java
StyleTesterView	0	src\main\java\org\jabref\styletester\StyleTesterView.java

This metric can indicate a code smell, showing that the class might be reviewed. The code smell could be, the class not having any comments, meaning that could be hard for someone else (or even the developers) to understand the code is doing or should be doing, or the class have too many comments, what could indicate that's something wrong with the code or a bad design is being covered up. An example of this code smell was found and explain by the team member Martim Gouveia – 57482. When review the class, can be found comments that look like "reminder", or comments that indicate that something needs to be done or that if a change is made to that section, another need to be updated, generating another code smell, that was found and explain by the team member Pedro Perdigão – 58165.

- JLOC – Javadoc lines of code
  Consists in the number of lines of Javacode present in the class.

Class	JLOC	Path
BracketedPattern	376	src\main\java\org\jabref\logic\citationkeypattern\BracketedPattern.java
TreeNode	300	src\main\java\org\jabref\model\TreeNode.java
…	…	…
VersionPreferences	0	src\main\java\org\jabref\preferences\VersionPreferences.java
StyleTesterView	0	src\main\java\org\jabref\styletester\StyleTesterView.java

The explanation in the last section applies to this, as well as the code smells explained and found.

- LOC – lines of code
  Consists in the number of lines of code present in the class.

Class	LOC	Path
JabRefPreferences	2291	src\main\java\org\jabref\preferences\JabRefPreferences.java
BibtexParserTest	1422	src\main\java\org\jabref\logic\importer\fileformat\BibtexParserTest.java
…	…	…
JabrefIconProvider	6	src\main\java\org\jabref\gui\icon\JabrefIconProvider.java
LocalizationLocator	6	src\main\java\org\jabref\logic\l10n\LocalizationLocator.java

This metric can indicate a code smell, showing that the class might be reviewed. If this metric shows a hight number, like 2291 as we see on the previous table, maybe indicates a code smell for large classes, meaning that class has more responsibility that she needed. We can combat this code smell by creating new classes for pass some methods to those classes or just pass a few methods for other classes. This code smell was identified and explained by the team member Pedro Lourenço – 57577.
In another hand, if the metric shows a low number, like 6 as we see on the previous table, maybe indicates a code smell for a data class, meaning that class is too small, that means that only contain only data and no real functionality, only getter and setter methods. This code smell was identified and explained by the team member Martim Gouveia – 57482.

Es – Sprint 2 – Use case diagram description

Use case: Open editor when a new entry is created

ID: 0

Description: Button that when activated open editor when a new entry is created

Main actors: User

Secondary actors: N/A

Use case: Show BibTeX source by default

ID: 1

Description: Button that when activated show BibTeX source by default

Main actors: User

Secondary actors: N/A

Use case: Show 'Related Articles' tab

ID: 2

Description: Button that when activated show 'Related Articles' tab

Main actors: User

Secondary actors: N/A

Use case: Show 'Related Articles' tab - Accept recommendations from Mr.DLib

ID: 3

Description: Button that depends on the use case (ID 2) when activated accept recommendations from Mr.DLib

Main actors: User

Secondary actors: N/A

Use case: Show 'LaTeX Citations' tab

ID: 4

Description: Button that when activated show 'LaTeX Citations' tab

Main actors: User

Secondary actors: N/A

Use case: Show validation messages

ID: 5

Description: Button that when activated show validation messages

Main actors: User

Secondary actors: N/A

Use case: Allows integers in 'edition' field in BibTeX mode

ID: 6

Description: Button that when activated allows integers in 'edition' field in BibTeX mode

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion

ID: 7

Description: Button that when activated enables use autocompletion

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - Affected fields

ID: 8

Description: Textbox that depends on the use case (ID 7) that when activated the user can write the fields to be affected

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - Name format

ID: 9

Description: depends on the use case (ID 7) that when activated the user can choose one of the name formats

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - Name format - Autocomplete names in 'Firstname Lastname' format only

ID: 10

Description: The user chooses autocomplete names in 'Firstname Lastname' format only

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - Name format - Autocomplete names in 'Lastname, Firstname' format only

ID: 11

Description: The user chooses autocomplete names in 'Lastname, Firstname' format only

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - Name format - Autocomplete names in both formats

ID: 12

Description: The user chooses autocomplete names in both formats

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - First names

ID: 13

Description: depends on the use case (ID 7) that when activated the user can choose one of the first names

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - First names - Use abbreviated firstname whenever possible

ID: 14

Description: The user chooses to use abbreviated firstname whenever possible

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - First names - Use full firstname whenever possible

ID: 15

Description: The user chooses to use full firstname whenever possible

Main actors: User

Secondary actors: N/A

Use case: Use autocompletion - First names - use abbreviatedand full firstname

ID: 16

Description: The user chooses to use abbreviatedand full firstname

Main actors: User

Secondary actors: N/A

Pedro Caldeirão - 57945

Chidamber and Kemerer metrics

The chidamber-kemerer metrics suit are composed by six metrics, these being: coupling between objects (CBO), depth of inheritance tree (DIT), lack of cohesion of methods (LCOM), number of children (NOC), response for class (RFC) and weighted method complexity (WCM).

For this analysis the metrics chosen were the CBO, DIT and WCM metrics.

CBO metric

The CBO metric, measures the amount of accesses that one class has with the other classes, counting multiple accesses as one. A higher CBO means there is an excessive coupling between object classes, therefore the class is more dependent leading to a way more harder maintenence of said class.

Classes with the highest CBO

Class	CBO	Source path
BibEntry	616	src/main/java/org/jabref/model/entry/BibEntry
Localization	396	src/main/java/org/jabref/logic/l10n/Localization
StandardField	324	src/main/java/org/jabref/model/entry/field/StandardField
BibDatabaseContext	264	src/main/java/org/jabref/model/database/BibDatabaseContext
BibDatabase	238	src/main/java/org/jabref/model/database/BibDatabase
JabRefFrame	146	src/main/java/org/jabref/gui/JabRefFrame

As it shows, in the table, the BibEntry class has highest CBO in the whole project, which can only mean there is an exceeding coupling between classes, with the same happening to the other six classes.

Relative to code smells, some of the potential code smells that could be found by analysis of this metric would be innapropritate intimacy or feature envy, in the project none of these code smell were identified.

DIT metric

The DIT metric measures the maximum inheritance path from the class to the root class. As the inheritance tree deepens, more methods and variables are going to be inherited making the resulting class really complex, although it makes the class more reusable due to method inheritance. It can be concluded that a high DIT number means a higher chance of faults occurring in the classes of the hierarchy being more evident in the middle classes because these are the ones that receive less checks.

Class	DIT	Source path
HighlightTableRow	9	src/main/java/org/jabref/gui/commonfxcontrols/CitationKeyPatternPanel/HighlightTableRow
IkonliCell	9	src/main/java/org/jabref/gui/groups/GroupDialogView/IkonliCell
RadioButtonCell	9	src/main/java/org/jabref/gui/util/RadioButtonCell
DataBaseChangePane	7	src/main/java/org/jabref/gui/collab/DataBaseChangePane
EditorTextArea	7	src/main/java/org/jabref/gui/fieldeditors/EditorTextArea
EditorTextField	7	src/main/java/org/jabref/gui/fieldeditors/EditorTextField

By analysis of the table, the top three classes all share the same DIT number, 9, this indicates that the classes in the hierarchy of these are the most fault-prone, being recommended a review.

Relative to code smell identification, some of the potential code smells that could be found by analisys of this metric would be the refused request and speculative generality code smell. In the project none of them were identified.

WCM metric

The WCM metric measures/counts the number of methods in a class. A class with high WMC number tends to be more application specific limiting the possibility of reuse. This metric can also helps us predict the amount of time that the class is going to cost to mantain and develop.

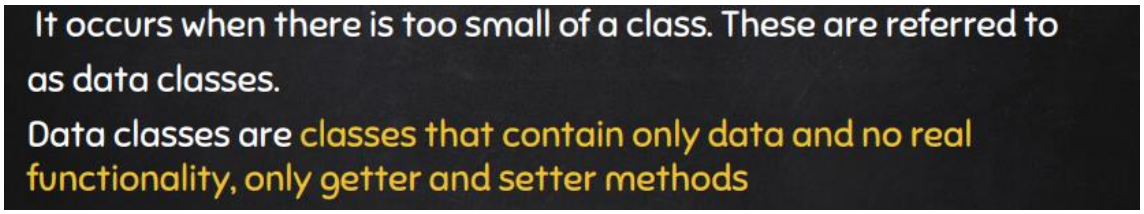
Class	WMC	Source path
JabRefPreferences	272	src/main/java/org/jabref/preferences/JabRefPreferences
OOBibStyle	179	src/main/java/org/jabref/logic/openoffice/style/OOBibStyle
OOBibBase	176	src/main/java/org/jabref/gui/openoffice/OOBibBase
BracketedPattern	176	src/main/java/org/jabref/logic/citationkeyPattern/BracketedPattern
MedlineImporter	153	src/main/java/org/jabref/logic/importer/fileformat/MedlineImporter
BibtexParser	151	src/main/java/org/jabref/logic/importer/fileformat/BibtexParser

Analysing the table, the JabRefPreferences class is the one with biggest number of methods in the whole project, so it would be better to separate several of this methods to smaller classes, with the same being applied to the others 5 classes.

Relative to code smells, some of the potential code smells that would be identified by analysis of this metric the long method code smell, large class. In the project, both of them were identified.

Data Class

CodeSnippet (Description)



Location in the code

Class MainTableFormatPreferences

Src/main/java/org/jabref/gui/mantainable/ MainTableNameFormatPreferences

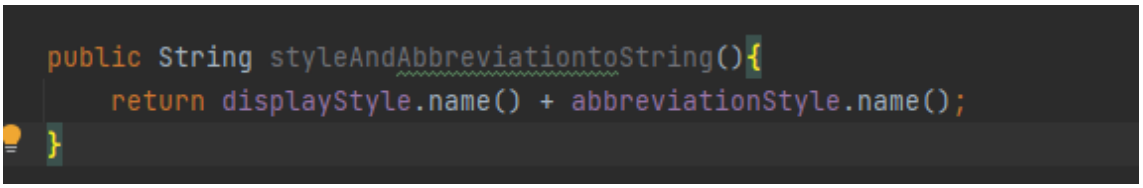
```java
package org.jabref.gui.maintable;

public class MainTableNameFormatPreferences {

    public enum DisplayStyle {
        NATBIB, AS_IS, FIRSTNAME_LASTNAME, LASTNAME_FIRSTNAME
    }

    public enum AbbreviationStyle {
        NONE, LASTNAME_ONLY, FULL
    }

    private final DisplayStyle displayStyle;
    private final AbbreviationStyle abbreviationStyle;

    public MainTableNameFormatPreferences(DisplayStyle displayStyle,
                                          AbbreviationStyle abbreviationStyle) {

        this.displayStyle = displayStyle;
        this.abbreviationStyle = abbreviationStyle;
    }

    public DisplayStyle getDisplayStyle() { return displayStyle; }

    public AbbreviationStyle getAbbreviationStyle() { return abbreviationStyle; }
}
```

Justification

This class can be considered a data class, because the only methods that it has, are getter's, thus not having any real functionality.

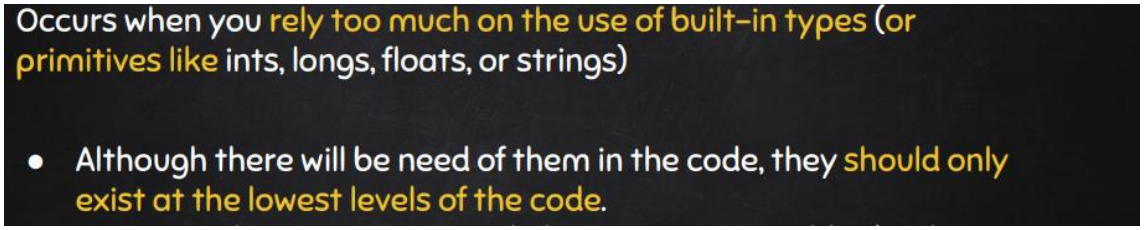
Refactoring proposal

The refactoring proposal for this code smell would to implement additional methods to give some functionality for the class itself, just like a toString method that in this case creates a string of the names of the display style and abreviation style of the user.



Primitive obsession

Code snnipet (example)



Location in the code

AuthorListParser Class

Src/main/java/org/jabref/logic/importer/AuthorListParser

```
    private String original;
    /**
     * index of the start in original, for example to p
     */
    private int tokenStart;
    /**
     * index of the end in original, for example to poi
     */
    private int tokenEnd;
    /**
     * end of token abbreviation (always: tokenStart <
     * Token.WORD
     */
    private int tokenAbbrEnd;
    /**
     * either space of dash
     */
    private char tokenTerm;
    /**
     * true if upper-case token, false if lower-case
     */
    private boolean tokenCase;
```

Justification

In this case, it is present the primitive obsession code smell because there are too many global variables (not low level of code) using the primitive data-types such as String, int and boolean.

Refactoring proposal

A refactoring proposal for this code smell would be to create an object that includes the majority of the variables, in this case anything related to a token, so the variables tokenStart, tokenEnd, tokenAbbrEnd, tokenTerm and tokenCase would belong to this class “TokenClass”, for exemple.

TokenClass

```
public class TokenClass {

    private int tokenStart;

    private int tokenEnd;

    private int tokenAbbrEnd;

    private char tokenTerm;

    private boolean tokenCase;
    public TokenClass(int tokenStart, int tokenEnd, int tokenAbbrEnd, char tokenTerm, boolean tokenCase){
            this.tokenAbbrEnd = tokenAbbrEnd;
            this.tokenCase = tokenCase;
            this.tokenEnd = tokenEnd;
            this.tokenStart = tokenStart;
            this.tokenTerm = tokenTerm;
    }

    public int getTokenStart() {
        return tokenStart;
    }

    public int getTokenEnd() {
        return tokenEnd;
    }
```

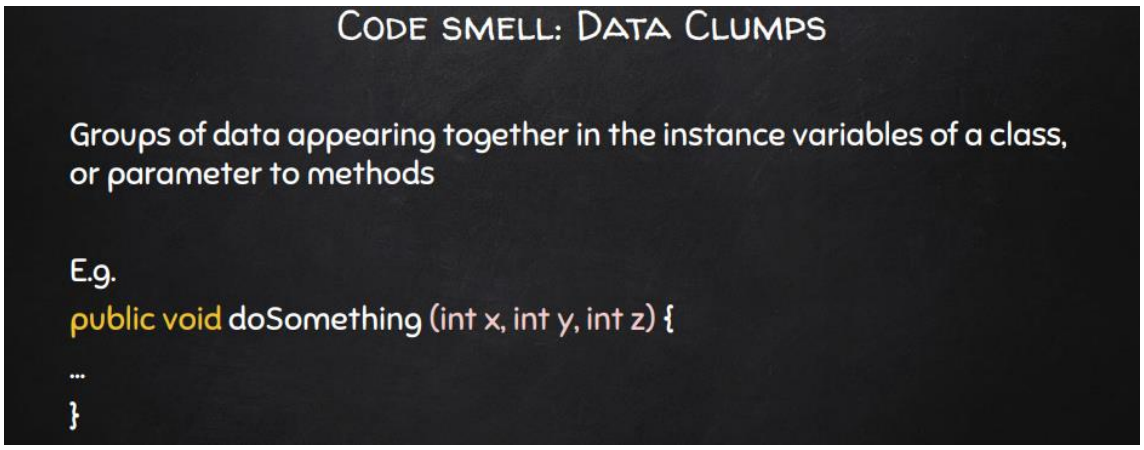
```java
public int getTokenAbbrEnd() {
    return tokenAbbrEnd;
}

public char getTokenTerm() {
    return tokenTerm;
}

public boolean isTokenCase() {
    return tokenCase;
}


public String tokenInfotoString() {
    return "TokenClass{" +
            "tokenStart=" + tokenStart +
            ", tokenEnd=" + tokenEnd +
            ", tokenAbbrEnd=" + tokenAbbrEnd +
            ", tokenTerm=" + tokenTerm +
            ", tokenCase=" + tokenCase +
            '}';
}
```

TokenClass being called in the AuthorListParser class

```java
private TokenClass token;
```

Data clumps (too many parameters in a function).

Code snippet



Location in the code

Method writeMetadataPdfByFileNames in the class ArgumentProcessor

Src/main/java/org/jabref/cli/ArgumentProcessor

```
private void writeMetadatatoPdfByFileNames(BibDatabaseContext databaseContext, BibDatabase dataBase, Vector<String> fileNames, Charset encoding,
```

```
FilePreferences filePreferences, XmpPdfExporter xmpPdfExporter, EmbeddedBibFilePdfExporter embeddedBibFilePdfExporter,
```

```
boolean writeXMP, boolean embeddBibfile)
```

Justification:

In this case, the data clumps code smell is identafiable beacuse the method writeMetadataPdfByFileNames has a total of 9 parameters.

Refactoring proposal:

A refactoring proposal for this code smell would be to create objects that provide the same functionality as some of the parameters, encapsulating the objects in the original parameters.

So, the new method would have the signature:

```
private void writeMetadatatoPdfByFileNames(BibDatabase bibDatabase,Vector<String> fileNames,
                                           Charset encoding,FilePreferences filePreferences,Exporter exporter,boolean writeXMP,boolean embeddBibfile)
```

With this change the number of parameters reduces from 9 to 7 which it isn't ideal, but given the responsabilities assigned to the method it is inevitable to not have a lot of parameters.

Padrões de design identificados (GOF)

Factory method

Code Snippet

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Factory object

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");
```

Factory instantiated

Code location:

LoggerFactory class, method getLogger

Slf4-api-2.0.0-alpha5.jar/org/slf4j

```
public static Logger getLogger(Class<?> clazz) {
    Logger logger = getLogger(clazz.getName());
    if (DETECT_LOGGER_NAME_MISMATCH) {
        Class<?> autoComputedCallingClass = Util.getCallingClass();
        if (autoComputedCallingClass != null && nonMatchingClasses(clazz, autoComputedCallingClass)) {
            Util.report(String.format("Detected logger name mismatch. Given name: \"%s\"; computed name: \"%s\".", logger.getName(),
                    autoComputedCallingClass.getName()));
            Util.report( msg: "See " + LOGGER_NAME_MISMATCH_URL + " for an explanation");
        }
    }
    return logger;
}
```

DiffHighlighting class

Src/main/java/org/jabref/gui/mergeentries/DiffHighlighting

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

RebuildFulltextSearchIndexAction class

Src/main/java/org/jabref/gui/search/RebuildFulltextSearchIndexAction

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

Justification

In this case the factory method is being instantiated beacuse two diferent loggers are being created by the same method (getLogger(Class<?> clazz)) that return the corresponding logger, given the name of the class, just like in the first code snnipet (but this one is with shapes).

Use case: General

ID: 1

Description : The user generates a new key for imported entries

Main actors: User

Secondary actors: N/A

Use case: Custom DOI URL

ID: 2

Description : The user uses a custom DOI base URL for article access

Main actors: User

Secondary actors: DOI base

Use case: Export sort order

ID: 3

Description: The user chooses the order for the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Keep original

ID: 4

Description: The user chooses mantains the original order for the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Current table

ID: 5

Description: The user chooses mantains the order of the current table for the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified

ID: 6

Description: The user chooses a specific order for the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified - MoveUp

ID: 7

Description: The user moves the selected line up the order of the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified - MoveDown

ID: 8

Description: The user moves the selected line down the order of the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified - Remove

ID: 9

Description: The user removes the line from the export order

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified - Descending

ID: 10

Description: The user chooses the general sort order of the line

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified - Add line

ID: 11

Description: The user adds a new line for the order of the export

Main actors: User

Secondary actors: N/A

Use case: Export sort order - Specified - OrderList

ID: 12

Description: The user chooses a type from a list of values for the selected line

Main actors: User

Secondary actors: N/A

Use case: Remote service

ID: 13

Description: The user sends pdf files and raw citation strings to an online service to determine Metadata

Main actors: User

Secondary actors: Grobid

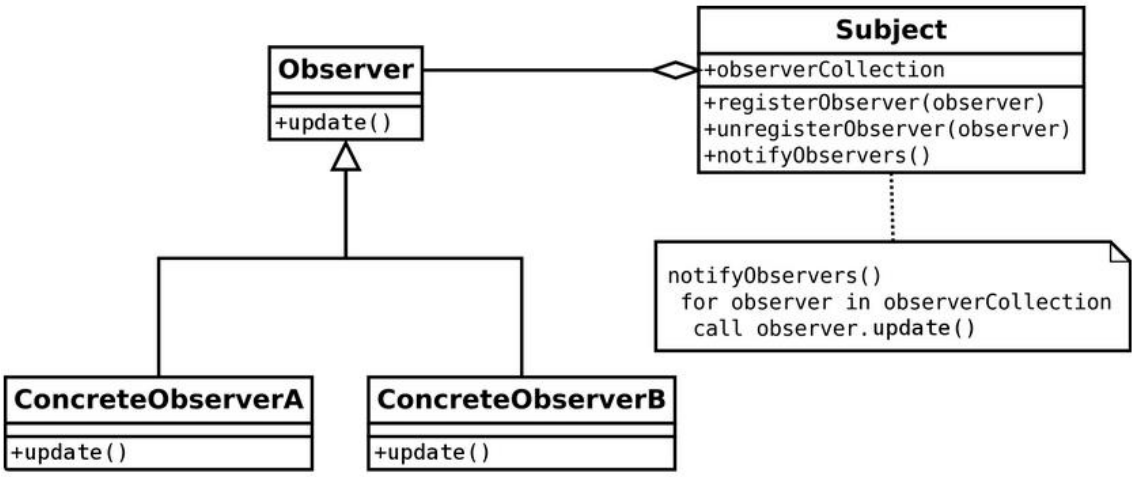
Observer pattern

Code Snippet

```java
import java.util.ArrayList;
import java.util.List;

public class MessagePublisher implements Subject {

    private List<Observer> observers = new ArrayList<>();

    @Override
    public void attach(Observer o) {
        observers.add(o);
    }

    @Override
    public void detach(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyUpdate(Message m) {
        for(Observer o: observers) {
            o.update(m);
        }
    }
}
```

This pattern is caracterized by mantaining a list of observers and notify them, thus update them of any change of state.

Strutcture



Code location

Registers the new listener

Src/main/java/org/jabref/gui/LibraryTab/

```java
this.getDatabase().registerListener(new IndexUpdateListener());
```

Methods that update/notify the listeners by adding or removing an entry

Src/main/java/org/jabref/gui/LibraryTab/IndexUpdateListener

```java
@Subscribe
public void listen(EntriesAddedEvent addedEntryEvent) {
    try {
        PdfIndexer pdfIndexer = PdfIndexer.of(bibDatabaseContext, preferencesService.getFilePreferences());
        for (BibEntry addedEntry : addedEntryEvent.getBibEntries()) {
            indexingTaskManager.addToIndex(pdfIndexer, addedEntry, bibDatabaseContext);
        }
    } catch (IOException e) {
        LOGGER.error("Cannot access lucene index", e);
    }
}

@Subscribe
public void listen(EntriesRemovedEvent removedEntriesEvent) {
    try {
        PdfIndexer pdfIndexer = PdfIndexer.of(bibDatabaseContext, preferencesService.getFilePreferences());
        for (BibEntry removedEntry : removedEntriesEvent.getBibEntries()) {
            indexingTaskManager.removeFromIndex(pdfIndexer, removedEntry);
        }
    } catch (IOException e) {
        LOGGER.error("Cannot access lucene index", e);
    }
}
```

Justification

As shown above, the observer pattern is being used because everytime an entry is either added or removed, all the listeners will update.

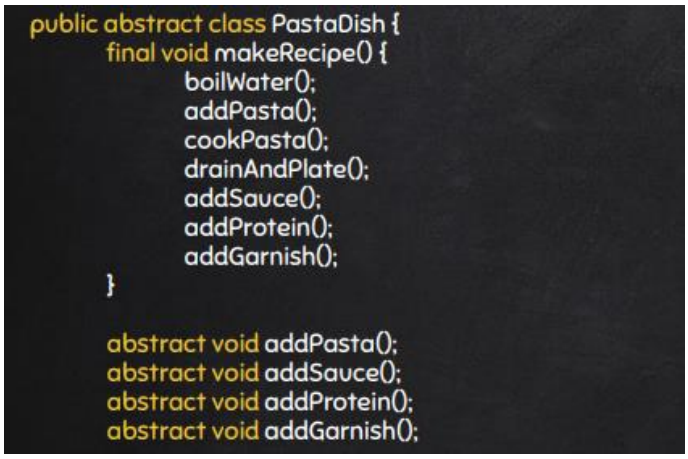
In this case, the observe ris the indexUpdateListenerClass and the subject, the LibraryTab class.

Although it is not present an unregister listener method, it still can be considered an observer pattern.

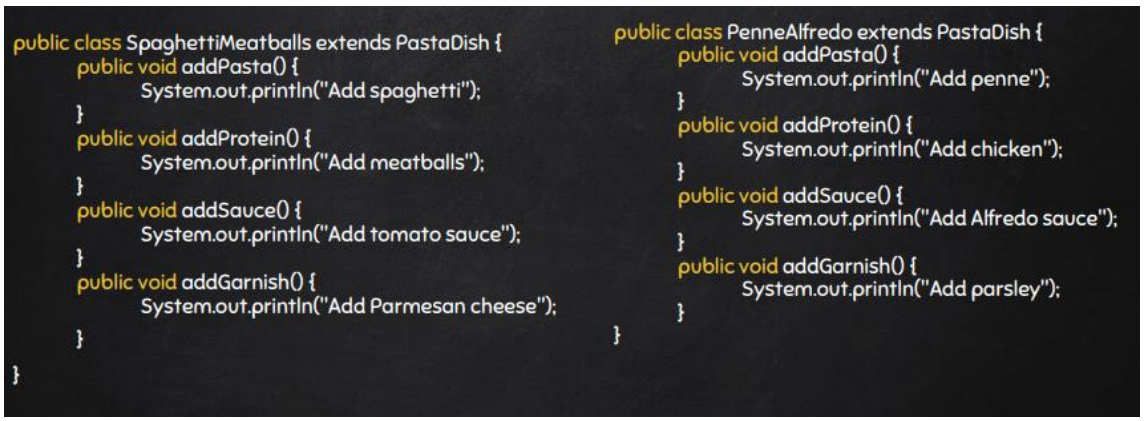
Template Method pattern

The template method pattern defines the general algorithm steps, postponing the implementation of some steps to the subclasses that extend it. It's best used when we can generalize two subclasses to a new superclasses.

Code Snippet



```
public abstract class PastaDish {
        final void makeRecipe() {
                boilWater();
                addPasta();
                cookPasta();
                drainAndPlate();
                addSauce();
                addProtein();
                addGarnish();
        }

        abstract void addPasta();
        abstract void addSauce();
        abstract void addProtein();
        abstract void addGarnish();
```



```
public class SpaghettiMeatballs extends PastaDish {
        public void addPasta() {
                System.out.println("Add spaghetti");
        }
        public void addProtein() {
                System.out.println("Add meatballs");
        }
        public void addSauce() {
                System.out.println("Add tomato sauce");
        }
        public void addGarnish() {
                System.out.println("Add Parmesan cheese");
        }
}
```

```
public class PenneAlfredo extends PastaDish {
        public void addPasta() {
                System.out.println("Add penne");
        }
        public void addProtein() {
                System.out.println("Add chicken");
        }
        public void addSauce() {
                System.out.println("Add Alfredo sauce");
        }
        public void addGarnish() {
                System.out.println("Add parsley");
        }
}
```

Code location

SidePaneComponent class (Abstract)

Src/main/java/org/jabref/gui/sidepane/SidePaneComponent

```
public abstract class SidePaneComponent
```

Abstract method in the SidePaneComponent class getType()

```java
/**
 * @return the type of this component
 */
public abstract SidePaneType getType();
```

WebSearchPane class in method getType()

Src/main/java/org/jabref/gui/importer/fetcher/WebSearchPane

```java
@Override
public SidePaneType getType() { return SidePaneType.WEB_SEARCH; }
```

GroupSidePane class in method getType()

Src/main/java/org/jabref/gui/groups/GroupSidePane

```java
@Override
public SidePaneType getType() { return SidePaneType.GROUPS; }
```

Justification

In this case the method pattern is being instanciated beacause the WebSearchPane and the GroupSidePane clases extend the same abstract method, getType() of the class SidePaneComponent, also abstract.

Identified code smells – Martim Gouveia 57482

Comments

```
/*
 * text.length$ Pops the top (string) literal, and pushes the number
 * of text characters it contains, where an accented character (more
 * precisely, a "special character", defined in Section 4) counts as
 * a single text character, even if it's missing its matching right
 * brace, and where braces don't count as text characters.
 *
 * From BibTeXing: For the purposes of counting letters in labels,
 * BibTEX considers everything contained inside the braces as a
 * single letter.
 */
buildInFunctions.put("text.length$", context -> textLengthFunction());

/*
 * Pops the top two literals (the integer literal len and a string
 * literal, in that order). It pushes the substring of the (at most) len
 * consecutive text characters starting from the beginning of the
 * string. This function is similar to substring$, but this one
 * considers a \special character", even if it's missing its matching
 * right brace, to be a single text character (rather than however many
 * ASCII characters it actually comprises), and this function doesn't
 * consider braces to be text characters; furthermore, this function
 * appends any needed matching right braces.
 */
buildInFunctions.put("text.prefix$", new TextPrefixFunction( vm: this));
```

Lines 547-571

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

Lines 651-680

Location of code smells: multiple methods of the VM class in

- src/main/java/org/jabref/logic/bst/VM.java

I considered these comments a code smell because they have unnecessarily long explanations of the functions of the code and cover every possible exception which it is not necessary.

I would propose removing most of the comments that cover very specific exceptions and would try to reduce the comments that explain the functions of the code. It should also be possible to rename some of the variables and methods to names that are self-explanatory and don’t require as many comments.

Identified code smells – Martim Gouveia 57482

Long Method

```
private List<String> refreshCiteMarkersInternal(List<BibDatabase> databases, OOBibStyle style)
        throws WrappedTargetException, IllegalArgumentException, NoSuchElementException,
        UndefinedCharacterFormatException, UnknownPropertyException, PropertyVetoException,
        CreationException, BibEntryNotFoundException {

    List<String> cited = findCitedKeys();
    Map<String, BibDatabase> linkSourceBase = new HashMap<>();
    Map<BibEntry, BibDatabase> entries = findCitedEntries(databases, cited, linkSourceBase);

    XNameAccess xReferenceMarks = getReferenceMarks();

    List<String> names;
    if (style.isSortByPosition()) {
        // We need to sort the reference marks according to their order of appearance:
        names = sortedReferenceMarks;
    } else if (style.isNumberEntries()) {
        // We need to sort the reference marks according to the sorting of the bibliographic
        // entries:
        SortedMap<BibEntry, BibDatabase> newMap = new TreeMap<>(entryComparator);
        for (Map.Entry<BibEntry, BibDatabase> bibtexEntryBibtexDatabaseEntry : entries.entrySet()) {
            newMap.put(bibtexEntryBibtexDatabaseEntry.getKey(), bibtexEntryBibtexDatabaseEntry.getValue());
        }
        entries = newMap;
        // Rebuild the list of cited keys according to the sort order:
        cited.clear();
        for (BibEntry entry : entries.keySet()) {
            cited.add(entry.getCitationKey().orElse( other: null));
        }
```

Lines 427-728

This method is located in:

- src/main/java/org/jabref/gui/openoffice/OOBibBase.java

I considered this method a code smell because of its size, the method is over 300 lines long and that is too long for any method, such a high number of lines can cause problems in readability and in extending the functions of the method.

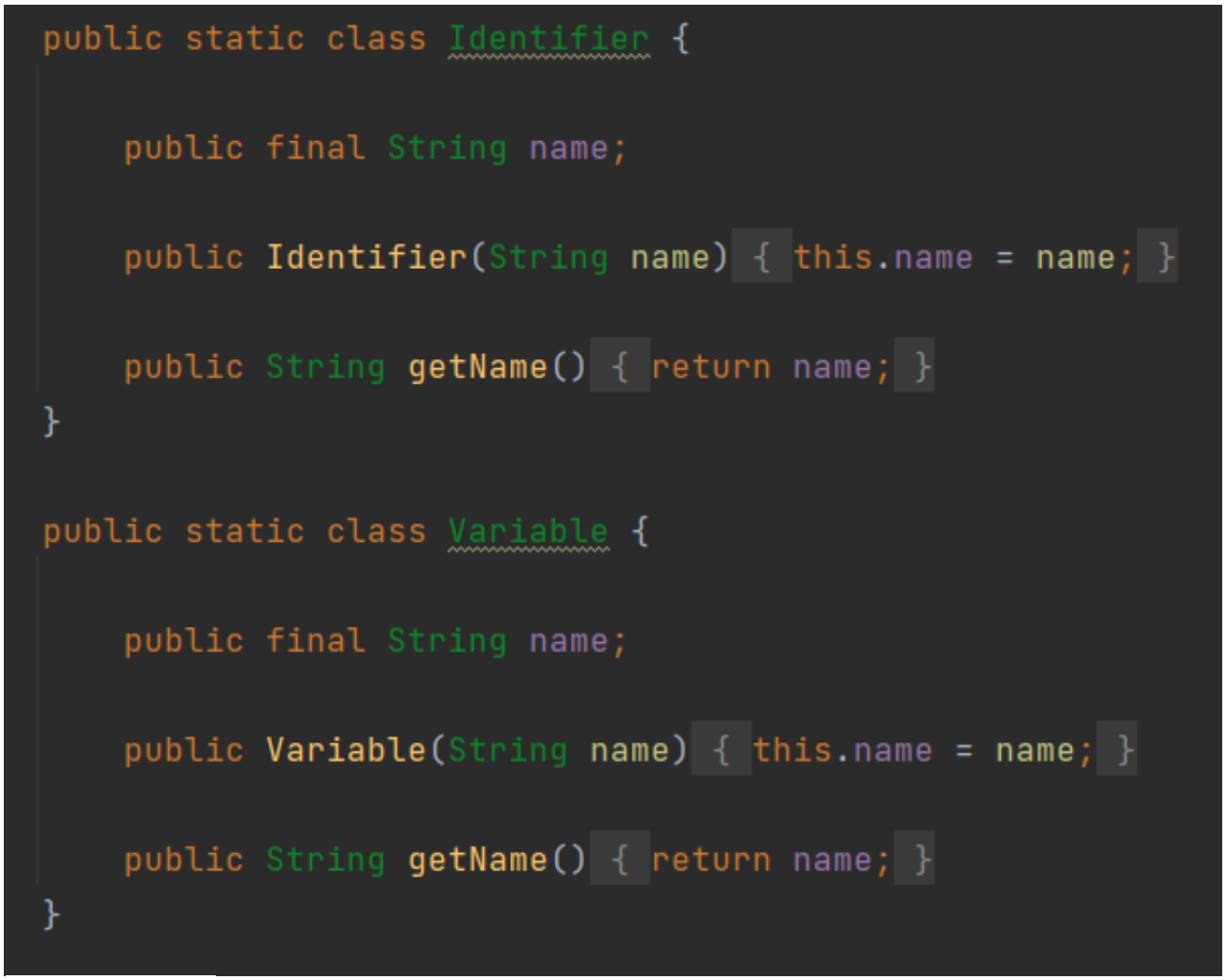
I propose that this method should be taken and split off into separate methods. This makes the code easier to understand and improve, it also prevents duplicated code from existing and there's no longer any need for most of the comments in the method.

```
// Remove all reference marks that don't look like JabRef citations:
List<String> tmp = new ArrayList<>();
for (String name : names) {
    if (CITE_PATTERN.matcher(name).find()) {
        tmp.add(name);
    }
}
names = tmp;
```

For example the code portion above could be separated into it's own method.

Identified code smells – Martim Gouveia 57482

Data Class



```
public static class Identifier {

    public final String name;

    public Identifier(String name) { this.name = name; }

    public String getName() { return name; }
}

public static class Variable {

    public final String name;

    public Variable(String name) { this.name = name; }

    public String getName() { return name; }
}
```

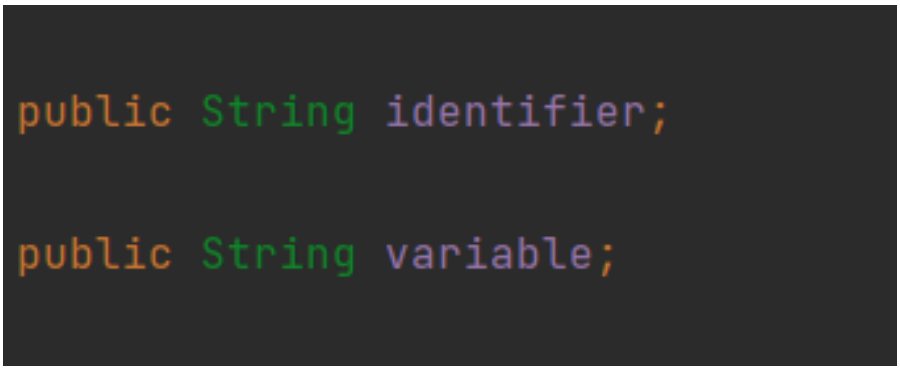
Lines: 75-99

Location of code smells:

- src/main/java/org/jabref/logic/bst/VM.java

I considered these classes a code smell because they only contain a getter for a single String that is stored in it, they are simply containing a variable that is used in the VM class without having any functionality on their own.

I would propose either replacing these classes with simple String variables or adding more functionality so they can simplify the VM class by doing some of its operations.



Example of possible refactoring.

Identified design patterns – Martim Gouveia 57482

Builder

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.

```
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 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);
}
```

Design pattern location:

- src/main/java/org/jabref/model/entry/BibEntryTypeBuilder.java

This is a builder design pattern because it lets us construct complex objects step by step, allowing us to control the type and representation of the object without having a constructor that is too big and has too many arguments.

Identified Design Patterns – Martim Gouveia 57482

Template

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but let's subclasses override specific steps of the algorithm without changing its structure.

```java
public abstract class Formatter {
 public abstract String getName();

 /**
  * Returns a unique key for the formatter that can be used for its identification
  *
  * @return the key of the formatter, always not null
  */
 public abstract String getKey();
```

```java
public class NormalizeDateFormatter extends Formatter {
    @Override
    public String getName() {
        return Localization.lang( key: "Normalize date");
    }
```

```java
public class NormalizeMonthFormatter extends Formatter {

    @Override
    public String getName() {
        return Localization.lang( key: "Normalize month");
    }
```

```java
public class NormalizeNewlinesFormatter extends Formatter {

    @Override
    public String getName() {
        return Localization.lang( key: "Normalize newline characters");
    }
```

Design Pattern location:

- src/main/java/org/jabref/logic/cleanup/Formatter.java

subclasses:

- src/main/java/org/jabref/logic/formatter/bibtexfields/NormalizeDateFormatter.java
- src/main/java/org/jabref/logic/formatter/bibtexfields/NormalizeMonthFormatter.java
- src/main/java/org/jabref/logic/cleanup/NormalizeNewlinesFormatter.java

I considered this a template method design pattern because the abstract class defines the skeleton of a method (getName in this case) but lets each one of its subclasses override the method in its own way avoiding repeated code.

In this case the Formatter class defines the getName() method letting each subclass implement it in the way they want.

Identified Design Patterns – Martim Gouveia 57482

Factory Method

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

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

    return checkMenuItem;
}

public Menu createMenu(Action action) {
    Menu menu = ActionUtils.createMenu(new JabRefAction(action, keyBindingRepository));

    // For some reason the graphic is not set correctly, so let's fix this
    setGraphic(menu, action);
    return menu;
}
```

Design pattern location:

- src/main/java/org/jabref/gui/actions/ActionFactory.java

```java
file.getItems().addAll(
        factory.createMenuItem(StandardActions.NEW_LIBRARY, new NewDatabaseAction( jabRefFrame: this, prefs)),
        factory.createMenuItem(StandardActions.OPEN_LIBRARY, getOpenDatabaseAction()),
        fileHistory,
        factory.createMenuItem(StandardActions.SAVE_LIBRARY, new SaveAction(SaveAction.SaveMethod.SAVE,  frame: this, prefs, stateManager)),
        factory.createMenuItem(StandardActions.SAVE_LIBRARY_AS, new SaveAction(SaveAction.SaveMethod.SAVE_AS,  frame: this, prefs, stateManager)),
        factory.createMenuItem(StandardActions.SAVE_ALL, new SaveAllAction( frame: this, prefs)),
```

Here the factory object is being used in another class to create different menuItem objects.

This is a factory method design pattern because this single class can be used by other classes to instantiate objects of different classes (menuItem, checkMenuItem, menu…) like a factory.

Using this design pattern avoids repeated code and simplifies the creation process of the objects because the factory class can create all of the objects that are needed instead of the classes using them.

Martin packaging metrics

The Martin packaging metrics or software packaging metrics were mentioned by Robert Cecil Martin in “Agile software development: principles, patterns, and practices” and can be subdivided into 5 different metrics: Afferent couplings (Ca), Efferent couplings (Ce), Abstractness (A), Instability (I), and Distance (D).

Afferent couplings

The afferent couplings metric measures the number of classes in other packages that depend upon classes within the package, this metric is an indicator of the package’s responsibility (Afferent couplings signal inward).

Ca	*Package*
13 210	org.jabref.model.entry.field
10 388	org.jabref.model.entry
4 109	org.jabref.logic.l10n

These are the packages that are most responsible, this means that a lot of classes from other packages are dependent on classes in this package.

Efferent couplings

The afferent couplings metric measures the number of classes in other packages that the classes in a package depend upon, this metric is an indicator of the package’s dependence on externalities (Efferent couplings signal outward).

Ce	*Package*
3 133	org.jabref.model.entry.types
2 707	org.jabref.logic.importer.fileformat
1 901	org.jabref.gui

These are the packages that are most dependent on externalities, this means that they depend on a lot of classes that are from other packages.

Abstractness

The abstractness metric measures the ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package. The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.

A	*Package*
1,00	org.jabref.architecture
1,00	org.jabref.logic.preview
1,00	org.jabref.model.openoffice.backend
…	…
0,00	org.jabref.gui.bibtexextractor
0,00	org.jabref.gui.auximport
0,00	org.jabref.cli

These are the packages with the most and least Abstractness. The packages with A=1 are 100% abstract (only abstract classes and interfaces) and the packages with A=0 are 100% concrete (no abstract classes or interfaces).

Instability

The Instability metric measures the ratio of efferent coupling to total coupling (I = Ce / (Ce + Ca)), this is an indicator of the package's resilience to change. This metric ranges from 0 to 1, with I=0 indicating a completely stable package and I=1 indicating a completely unstable package.

I	*Package*
1,00	org.jabref.architecture
1,00	org.jabref.gui.logging
1,00	org.jabref.logic.openoffice.action
…	…
0,00	org.jabref.gui.util.uithreadaware
0,00	org.jabref.logic.help
0,00	org.jabref.gui.util.uithreadaware

These are the packages that are most and least resilient to change, this means that in the most instable classes most of the couplings are Efferent and in the least instable classes most of the couplings are Afferent.

Distance

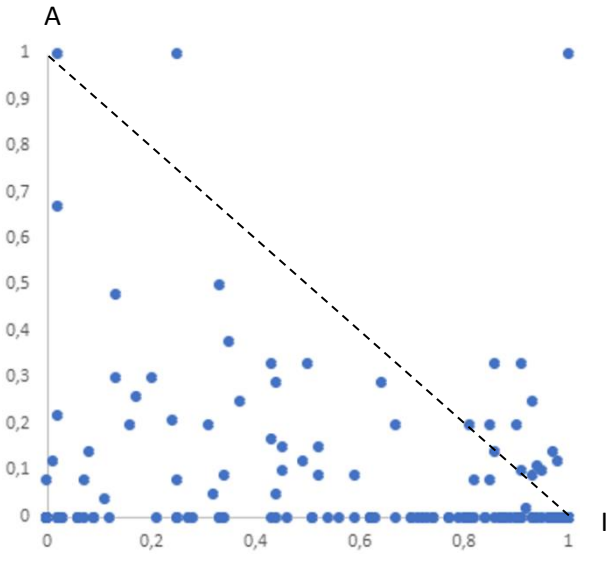
The distance metric measures the perpendicular distance of a package from the idealized line A + I = 1. D is calculated as D = |A + I - 1|, this metric is an indicator of the package's balance between abstractness and stability. Ideal packages are either completely abstract and stable (I=0, A=1) or completely concrete and unstable (I=1, A=0). The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.

D	*A*	*I*	*Package*
1,00	0,00	0,00	org.jabref.gui.util.uithreadaware
1,00	0,00	0,00	org.jabref.logic.help
1,00	0,00	0,00	org.jabref.logic.journals

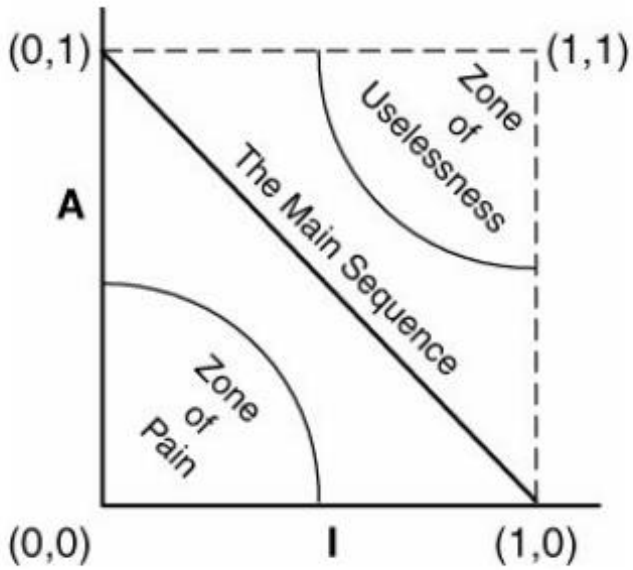
These are the most distant packages from the main sequence, this means that they either have mainly abstract classes and Efferent couplings (dependent on externalities) or mainly non abstract classes and Afferent couplings (other packages are dependent on this one).

Packages with A=0 and I=0 are highly stable and concrete, such a component is not desirable because it is rigid. It cannot be extended because it is not abstract and it is very difficult to change because of its stability, this can cause a Shotgun Surgery code smell.

Packages with A=1 and I=1 are undesirable because they are maximally abstract and yet have no dependents, such components are useless and can cause a Lazy Class or Speculative Generality code smell.



Distance of the components of the project



Bibliografia: Robert C. Martin (2002). Agile software development: principles, patterns, and practices

Use Case Descriptions – Martim Gouveia 57482

Use case: Manage Libraries

ID: 0

Main Actor: User

Secondary Actor: N/A

Description: The user can do many actions to manage the libraries.

Use case: New Library

ID: 1

Main Actor: User

Secondary Actor: N/A

Description: Creates a new library.

Use case: Open Library

ID: 2

Main Actor: User

Secondary Actor: N/A

Description: Opens an existing library.

Use case: Save Library

ID: 3

Main Actor: User

Secondary Actor: N/A

Description: Saves an existing library.

Use case: Close Library

ID: 4

Main Actor: User

Secondary Actor: N/A

Description: Closes the current library.

Use case: Manage Entries

ID: 5

Main Actor: User

Secondary Actor: N/A

Description: The user can do many actions to manage the entries.

Use case: New Entry

ID: 6

Main Actor: User

Secondary Actor: N/A

Description: Creates a new entry.

Use case: New Article

ID: 7

Main Actor: User

Secondary Actor: N/A

Description: Creates a new article.

Use case: Delete Entry

ID: 8

Main Actor: User

Secondary Actor: N/A

Description: Deletes an entry.

Use case: Edit Entry

ID: 9

Main Actor: User

Secondary Actor: N/A

Description: The user can do many actions to edit the entries.

Use case: Read Status Action

ID: 10

Main Actor: User

Secondary Actor: N/A

Description: The user can do many actions to change the status of an entry.

Use case: Clear

ID: 11

Main Actor: User

Secondary Actor: N/A

Description: Clears the status.

Use case: Set Read

ID: 12

Main Actor: User

Secondary Actor: N/A

Description: Sets the status to read.

Use case: Set Skimmed

ID: 13

Main Actor: User

Secondary Actor: N/A

Description: Sets the status to skimmed.

Use case: Rank Action

ID: 14

Main Actor: User

Secondary Actor: N/A

Description: The user can do many actions to change the rank of an entry.

Use case: Clear

ID: 15

Main Actor: User

Secondary Actor: N/A

Description: Clears the rank.

Use case: Set 1

ID: 16

Main Actor: User

Secondary Actor: N/A

Description: Sets the rank to 1.

Use case: Set 2

ID: 17

Main Actor: User

Secondary Actor: N/A

Description: Sets the rank to 2.

Use case: Set 3

ID: 18

Main Actor: User

Secondary Actor: N/A

Description: Sets the rank to 3.

Use case: Set 4

ID: 19

Main Actor: User

Secondary Actor: N/A

Description: Sets the rank to 4.

Use case: Set 5

ID: 20

Main Actor: User

Secondary Actor: N/A

Description: Sets the rank to 5.