Design pattern

1. **Prototype** (package org.jabref.gui.importer.actions.AppendDatabaseAction)

Prototype is used for creating a new object which is the same or similar to the original object. In our project, prototype is used when creating data entry objects. Below is a piece of code from the mergeFromBibtex method, which can merge the data entry read from the Bibtex file.

```
for (BibEntry originalEntry : fromDatabase.getEntries()) {
   BibEntry entry = (BibEntry) originalEntry.clone();
   UpdateField.setAutomaticFields(entry, overwriteOwner,
overwriteTimeStamp,
           Globals.prefs.getUpdateFieldPreferences());
   entriesToAppend.add(entry);
}
```

Code Snippet (1): Prototype using clone method

As the Code Snippet (1) shows, each data entry is encapsulated in a BibEntry object using the clone method. Since every entry has the same fields, prototype can easily create new objects efficiently.

```
 @Override
    public Object clone() {
        BibEntry clone = new BibEntry(IdGenerator.next(), type.getValue());
        clone.fields = FXCollections.observableMap(new
ConcurrentHashMap<>(fields));
        return clone;
    }
```

Code Snippet (2): Definition of clone method

Developers overwrite the clone method. This method returns a newly created object, which is different from the original one. This is how the prototype pattern is ensured.

2. **FactoryMethod** (package org.jabref.logic.util.FileType)

In general, factory pattern defines a factory interface and lets subclasses decide which class to create depending on different requirements. In the project, FileType interface is used for underlying file types that are extendable.

```
public interface FileType {

    default List<String> getExtensionsWithDot() {
        return getExtensions().stream()
                              .map(extension -> "*." + extension)
                              .collect(Collectors.toList());
```

```
    }

    List<String> getExtensions();
}
```

Code Snippet (3): FileType interface

Enum StandardFileType(package org.jabref.logic.util) implements FileType interface, it realizes the getExtensions method to return different file types. In the enum StandardFileType, different types are defined, as shown in Code Snippet (4).

```
    BIBTEXML("bibx", "xml"),
    ENDNOTE("ref", "enw"),
    ISI("isi", "txt"),
    MEDLINE("nbib", "xml"),
    MEDLINE_PLAIN("nbib", "txt"),
    PUBMED("fcgi"),
    SILVER_PLATTER("dat", "txt"),
```

Code Snippet (4): File types (in StandardFileType)

Based on the input, a private List object "*extensions"* will be initialized in the construction method, as shown in Code Snippet (5).

```
private final List<String> extensions;

    StandardFileType(String... extensions) {
        this.extensions = Arrays.asList(extensions);
    }
```

Code Snippet (5): Construction method for StandardFileType

In conclusion, abstract factory provide a basic interface for actual classes. Specific return values will be defined in the actual classes to realize different goals based on the different requirements specified.

3. **Builder** (package org.jabref.model.entry.BibEntryTypeBuilder)

In the builder pattern, different components are separated.

```
    private Set<BibField> fields = new HashSet<>();
    private Set<OrFields> requiredFields = new HashSet<>();
```

Code Snippet (6): Attributes in BibEntryTypeBuilder

BibEntryTypeBuilder has two attributes, *fields* and *requireFields*. They are built in different methods with different parameters. Notice Code Snippet (7).

```
public BibEntryTypeBuilder withDetailFields(Collection<Field> newFields) {
```

```
        this.fields = Streams.concat(fields.stream(),
newFields.stream().map(field -> new BibField(field, FieldPriority.DETAIL)))
                             .collect(Collectors.toSet());
        return this;
    }
public BibEntryTypeBuilder withRequiredFields(Set<OrFields> requiredFields)
{
        this.requiredFields = requiredFields;
        return this;
    }
```

Code Snippet (7): Different constructors in BibEntryTypeBuilder

As the name suggests, BibEntryTypeBuilder is used for creating a BibEntryType object. There is a build method included in BibEntryTypeBuilder.

```
public BibEntryType build() {
        // Treat required fields as important ones
        Stream<BibField> requiredAsImportant = requiredFields.stream()

.flatMap(TreeSet::stream)
                                                          .map(field ->
new BibField(field, FieldPriority.IMPORTANT));
        Set<BibField> allFields = Stream.concat(fields.stream(),
requiredAsImportant).collect(Collectors.toSet());
        return new BibEntryType(type, allFields, requiredFields);
    }
```

Code Snippet (8): Build method

BibEntryTypeBuilder is used in BibEntryTypeManager class which controls the building process at a higher level.

Using the builder pattern, the building processes of *fields* and *requiredFields* are separated, then the builder class assembles every part together. The advantage is that the code can be extended more often as each component has its individual building process. BibEntryTypeManager only controls the high-level build method and does not know the specifications of the building process of different parts, which in turn lowers coupling.

4. **Adapter** (package org.jabref.model.strings.LatexToUnicodeAdapter)

There is a class called LatexToUnicodeAdapter which only has one method called format.

```
public class LatexToUnicodeAdapter {
    public static String format(String inField) {
        Objects.requireNonNull(inField);
```

```
        String toFormat =
underscoreMatcher.matcher(inField).replaceAll(replacementChar);
        toFormat = Normalizer.normalize(LaTeX2Unicode.convert(toFormat),
Normalizer.Form.NFC);
        return
underscorePlaceholderMatcher.matcher(toFormat).replaceAll("_");
    }
}
```

Code Snippet (9): LatexToUnicodeAdapter

As observed from Code Snippet (10), this method (LatexToUnicodeAdapter.format) is in fact called in LatexToUnicodeFormatter.format() method. These have the same method name and it is clearly observed that format() in LatexToUnicodeAdapter is an improvement or adaption of the format() in LatexToUnicodeFormatter.

```
public String format(String inField) {
        return LatexToUnicodeAdapter.format(inField);
    }
```

Code Snippet (10): LatexToUnicodeFormatter.format()

Adapter is usually used when there are new requirements and the current method does not support these features. Here, a newly-defined method in the LatexToUnicodeAdapter is called in the original class, which is LatexToUnicodeFormatter, to meet new requirements.

5. **Iterator (**package org.jabref.logic.l10n.Language)

Enum in Java is a good implementation of the Iterator pattern. In our project, there is an enum named Language which is used to set up the language of the system. Each object in the enum is a Language object with two attributes: displayName and id.

```
public enum Language {

    BAHASA_INDONESIA("Bahasa Indonesia", "in"),
    BRAZILIAN_PORTUGUESE("Brazilian Portuguese", "pt_BR"),
    DANISH("Dansk", "da"),
    GERMAN("Deutsch", "de"),
    ENGLISH("English", "en"),
    SPANISH("Español", "es"),
    FRENCH("Français", "fr"),
    ITALIAN("Italiano", "it"),
    JAPANESE("Japanese", "ja"),
    DUTCH("Nederlands", "nl"),
    NORWEGIAN("Norsk", "no"),
```

```
    PERSIAN("Persian (فارسی)", "fa"),
    PORTUGUESE("Português","pt"),
    RUSSIAN("Russian", "ru"),
    SIMPLIFIED_CHINESE("Simplified Chinese", "zh"),
    SVENSKA("Svenska", "sv"),
    TURKISH("Turkish", "tr"),
    VIETNAMESE("Vietnamese", "vi"),
    GREEK("ε λ λ η ν ι κ ά", "el"),
    TAGALOG("Tagalog/Filipino", "tl"),
    POLISH("Polish", "pl");

    private final String displayName;
    private final String id;

    Language(String displayName, String id) {
        this.displayName = displayName;
        this.id = id;
    }
```

Code Snippet (11): Enum Language

With the help of Iterator, the internal functions of iteration will not be exposed to external classes. Instead, external classes will call methods defined in the Iterator class. Besides, enum could be used in different objects that need enumeration.

Issue under Consideration

[Title made of two sentences: Capitalization is OK, but warning is displayed #5832]

As it is reported in this open issue, a warning sign (small yellow triangle) will show up underneath the text field while entering the title in the entry editor, when the title is made up of two sentences, for example, “A title made of two sentences. This is the second sentence”. The capitalization of the first letter in any sentence right after the first sentence triggers such a warning, which is an error with the software because a title which is made of two sentences or two capitalized phrases is very common in practice.

We reproduced this bug locally, by using titles such as “Cognitive radio. An integrated agent architecture for software defined radio” and “Anxiety disorders and phobias: A cognitive perspective”. The warning sign appeared immediately after we entered the second capital letter “A” in both cases.

Utilizing the “Search in Path” function with keywords such as “capital” and “warning”, we located the essence of the warning that gets triggered: `TitleChecker.java`. The method `checkValue()` takes in a title string and runs a basic algorithm shown in Code Snippet (12).

```
/**
```

```
* Algorithm:
* - remove trailing whitespaces
* - ignore first letter as this can always be written in caps
* - remove everything that is in brackets
* - check if at least one capital letter is in the title
*/
```

Code Snippet (12): TitleChecker.java [Line 23 - 29]

The problem with this code is that the algorithm would only run once for checking whether the title is valid. For any title made of multiple sentences or multiple phrases (with first word capitalized in each phrase), the 4th check “check if at least one capital letter is in the title” will return false and in turn gives us a warning that “capital letters are not masked using curly brackets{}.” It should be mentioned that curly brackets are used in JabRef for case-sensitive input such as author names and organizations, but capitalization in sentences and phrases have not been addressed properly.

Use “Anxiety disorders and phobias: A cognitive perspective” as example input, and run the basic algorithm checkValue(), we get:

```
* - remove trailing whitespaces
New string =  “Anxietydisordersandphobias:Acognitiveperspective”
* - ignore first letter as this can always be written in caps
New string = “anxietydisordersandphobias:Acognitiveperspective”
* - remove everything that is in brackets
New string = “anxietydisordersandphobias:Acognitiveperspective”
* - check if at least one capital letter is in the title
The new string “anxietydisordersandphobias:Acognitiveperspective” has at least one
capital letter, return False. (Warning sign is triggered!)
*/
```

Code Snippet (13): Algorithm Analysis

Possible Solution:
Pull Request for fixing this issue has been submitted. **[Status: Open]**

```
/**
* Modified algorithm:
* - remove everything that is in brackets
* - split the title into subTitles based on the delimiters
* - for each subTitle:
* -    remove trailing whitespaces
* -    ignore first letter as this can always be written in caps
* -    check if at least one capital letter is in the title
*/
```

Code Snippet (14): Modified Algorithm

Actual implementation

```
public class TitleChecker implements ValueChecker {

   private static final Pattern INSIDE_CURLY_BRAKETS =
Pattern.compile("\\{[^}\\{]*\\}");
   private static final Predicate<String> HAS_CAPITAL_LETTERS =
Pattern.compile("[\\p{Lu}\\p{Lt}]").asPredicate();

   private final BibDatabaseContext databaseContext;

   public TitleChecker(BibDatabaseContext databaseContext) {
       this.databaseContext = databaseContext;
   }

   /**
    * Algorithm:
    * - remove everything that is in brackets
    * - split the title into subTitles based on the delimiters
    * - for each subTitle:
    * -    remove trailing whitespaces
    * -    ignore first letter as this can always be written in caps
    * -    check if at least one capital letter is in the title
    */
   @Override
   public Optional<String> checkValue(String value) {
       if (StringUtil.isBlank(value)) {
           return Optional.empty();
       }

       if (databaseContext.isBiblatexMode()) {
           return Optional.empty();
       }

       String valueOnlySpacesWithinCurlyBraces = value;
       while (true) {
           Matcher matcher =
INSIDE_CURLY_BRAKETS.matcher(valueOnlySpacesWithinCurlyBraces);
           if (!matcher.find()) {
               break;
```

```
            }
            valueOnlySpacesWithinCurlyBraces = matcher.replaceAll("");
        }

        // Delimiters are . ! ? ; :
        String[] delimitedStr =
valueOnlySpacesWithinCurlyBraces.split("(\\.|\\!|\\?|\\;|\\:)");
        for (String subStr : delimitedStr) {
            subStr = subStr.trim();
            if (subStr.length() > 0) {
                subStr = subStr.substring(1);
            }
            boolean hasCapitalLettersThatBibtexWillConvertToSmallerOnes =
HAS_CAPITAL_LETTERS.test(subStr);
            if (hasCapitalLettersThatBibtexWillConvertToSmallerOnes) {
                return Optional.of(Localization.lang("capital letters are
not masked using curly brackets {}"));
            }
        }
        return Optional.empty();
    }
}
```

Code Snippet (15): Implementation submitted to fix issue