Contents

index **2**

code-howtos–IntelliJ **5**

code-howtos–bibtex **7**

code-howtos–code-quality **8**

code-howtos–custom-svg-icons **10**

code-howtos–error-handling **12**

code-howtos–eventbus **14**

code-howtos–faq **16**

code-howtos–fetchers **25**

code-howtos–http-server **29**

code-howtos–javafx **31**

code-howtos–jpackage **37**

code-howtos–localization **39**

code-howtos–logging **42**

code-howtos–openoffice–code-reorganization **44**

code-howtos–openoffice–ooresult-ooerror–ooresult-alternatives **50**

code-howtos–openoffice–ooresult-ooerror **61**

code-howtos–openoffice–order-of-appearance **66**

code-howtos–openoffice–overview **69**

code-howtos–openoffice–problems **76**

code-howtos–openoffice **79**

code-howtos–remote-storage-jabdrive **80**

code-howtos–remote-storage-sql **87**

code-howtos–remote-storage **89**

code-howtos–testing **90**

code-howtos–tools **95**

code-howtos–ui-recommendations **97**

code-howtos–xmp-parsing **99**

code-howtos **101**

contributing **106**

decisions–0000-use-markdown-architectural-decision-records **107**

decisions–0001-use-crowdin-for-translations **108**

decisions–0002-use-slf4j-for-logging **109**

decisions–0003-use-gradle-as-build-tool **111**

decisions–0004-use-mariadb-connector **113**

decisions–0005-fully-support-utf8-only-for-latex-files **115**

decisions–0006-only-translated-strings-in-language-file **117**

decisions–0007-human-readable-changelog **119**

decisions–0008-use-public-final-instead-of-getters **120**

decisions–0009-use-plain-junit5-for-testing **121**

decisions–0010-use-h2-as-internal-database **124**

decisions–0011-test-external-links-in-documentation **125**

decisions–0012-handle-different-bibEntry-formats-of-fetchers **127**

decisions–0013-add-native-support-biblatex-software **129**

decisions–0014-separate-URL-creation-to-enable-proper-logging **131**

decisions–0015-support-an-abstract-query-syntax-for-query-conversion **134**

decisions–0016-mutable-preferences-objects **137**

decisions–0017-allow-model-access-logic **138**

decisions–0018-use-regular-expression-to-split-multiple-sentence-titles **140**

decisions–0019-implement-special-fields-as-separate-fields **141**

decisions–0020-use-Jackson-to-parse-study-yml **143**

decisions–0021-keep-study-as-a-dto **145**

decisions–0022-remove-stop-words-during-query-transformation **146**

decisions–0023-localized-preferences **148**

decisions–0025-reviewdog-reviews **150**

decisions–0026-use-jna-to-determine-default-directory **151**

decisions–0027-synchronization **153**

decisions–0028-http-return-bibtex-string **155**

decisions–0029-cff-export-multiple-entries **157**

decisions–0030-use-apache-commons-io-for-directory-monitoring **159**

decisions–0031-use-current-tab-for-deciding-style-type-for-oo **161**

decisions–0032-store-chats-in-local-user-folder **162**

decisions–0033-store-chats-in-mvstore **164**

decisions–0034-use-citation-key-for-grouping-chat-messages **166**

decisions–0035-generate-embeddings-online **168**

decisions–0036-use-textarea-for-chat-content **170**

decisions–0037-rag-architecture-implementation **173**

decisions–0038-use-entryId-for-bibentries **177**

decisions–0039-use-apache-velocity-as-template-engine **178**

decisions–0040-display-front-cover-in-preview-tab **181**

decisions–0041-use-one-form-for-singular-and-plural **185**

decisions–0042-use-webview-for-summarization-content **188**

decisions–0043-show-merge-dialog-when-importing-a-single-pdf **190**

decisions–adr-template **192**

decisions **194**

getting-into-the-code–development-strategy **196**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–eclipse **198**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–intellij-11-code-into-ide **201**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–intellij-12-build **205**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–intellij-13-code-style **217**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–intellij-89-run-with-intellij **227**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–pre-01-github-account **230**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–pre-02-software **231**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–pre-03-code **232**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–trouble-shooting **234**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace–vscode **237**

getting-into-the-code–guidelines-for-setting-up-a-local-workspace **239**

getting-into-the-code–high-level-documentation **240**

getting-into-the-code **242**

requirements–ai **243**

requirements **244**

teaching **246**

Overview on Developing JabRef

This page presents all development information around JabRef. In case you are a end user, please head to the user documentation or to the general homepage of JabRef.

Starting point for new developers

On the page Setting up a local workspace, we wrote about the initial steps to get your IDE running. We strongly recommend to continue reading there. After you successfully cloned and build JabRef, you are invited to continue reading here.

How tos

- External: Sync your fork with the JabRef repository
- External (🇩🇪): Branches and pull requests: https://github.com/unibas-marcelluethi/software-engineering/blob/master/docs/week2/exercises/practical-exercises.md

Teaching Exercises

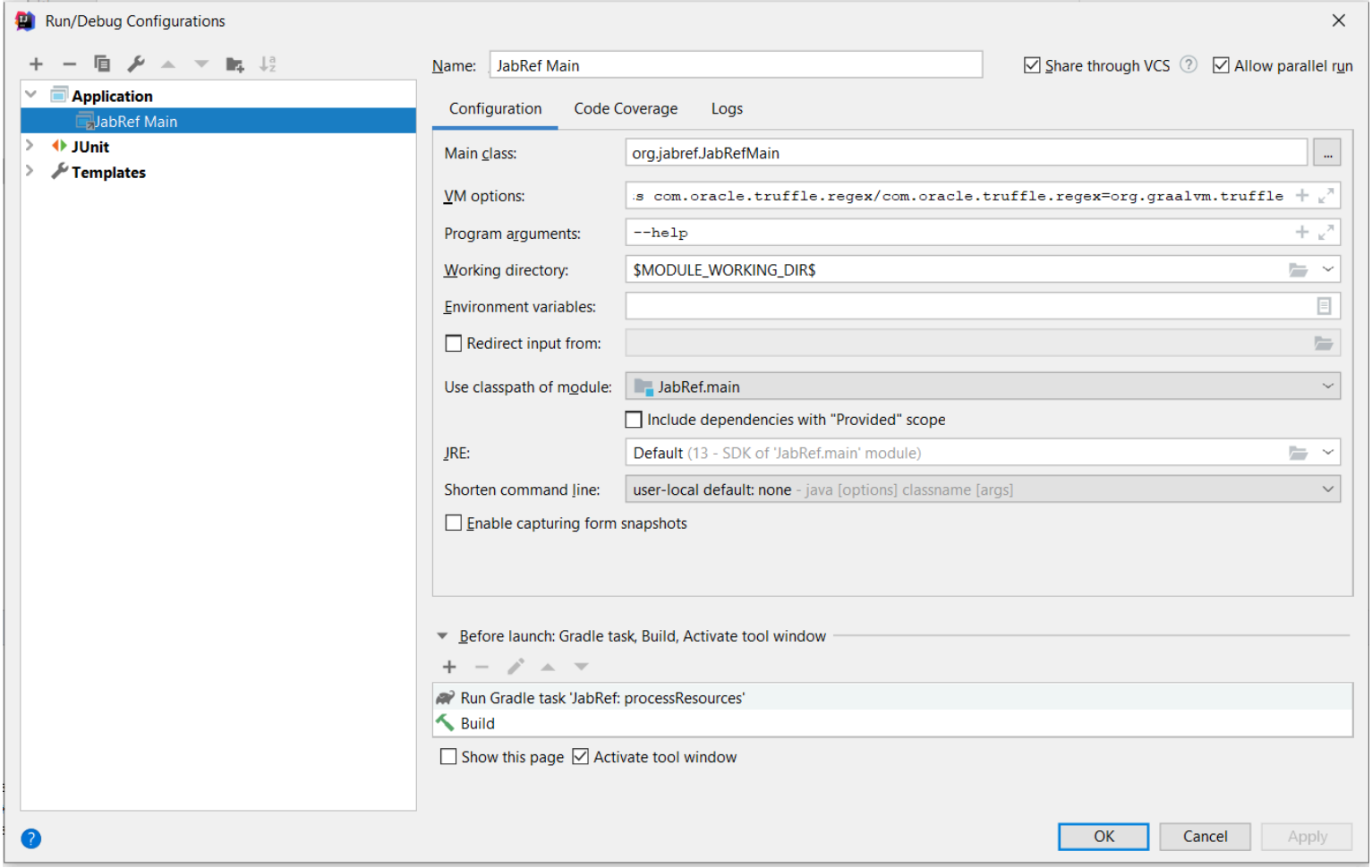
We are very happy that JabRef is part of Software Engineering trainings. Please head to Teaching for more information on using JabRef as a teaching object and on previous courses where JabRef was used.

Miscellaneous Hints

Command Line

The package `org.jabref.cli` is responsible for handling the command line options.

During development, one can configure IntelliJ to pass command line parameters:



Passing command line arguments using gradle is currently not possible as all arguments (such as `-Dfile.encoding=windows-1252`) are passed to the application.

Without jlink, it is not possible to generate a fat jar any more. During development, the capabilities of the IDE has to be used.

Groups

Diagram showing aspects of groups: Groups.uml.

Architectural Decision Records

Architectural decisions for JabRef are recorded.

For new ADRs, please use adr-template.md as basis. More information on MADR is available at https://adr.github.io/madr/. General information about architectural decision records is available at https://adr.github.io/.

FAQ

- Q: I get `java: package org.jabref.logic.journals does not exist`.

  A: You have to ignore `buildSrc/src/main` as source directory in IntelliJ as indicated in our setup guide.

  Also filed as IntelliJ issue IDEA-240250.

IntelliJ Hints

SUMMARY

Did you know that IntelliJ allows for reformatting selected code if you press `Ctrl + Alt + L`?

Key hints for IntelliJ

- `Shift+Shift` (AKA double-shift): Open the search dialog.
- `Ctrl+N`: Open the search dialog and select search for a class.
- `Ctrl+Shift+F`: Search everywhere in the code base.
- `Alt+F1` and then `Enter`: Locate the file in the search bar on the left side.
- `Ctrl+Shift+T`: Navigate from a class to the test class.

Show variable values in IntelliJ

1. Go to a test case (example: `org.jabref.model.entry.BibEntryTest#settingTypeToNullThrowsException`
2. Set the breakpoint to the first line
3. Execute the test
4. Go to the settings of the debugger and activate “Show Variable Values in Editor” and “Show Method Return Values”

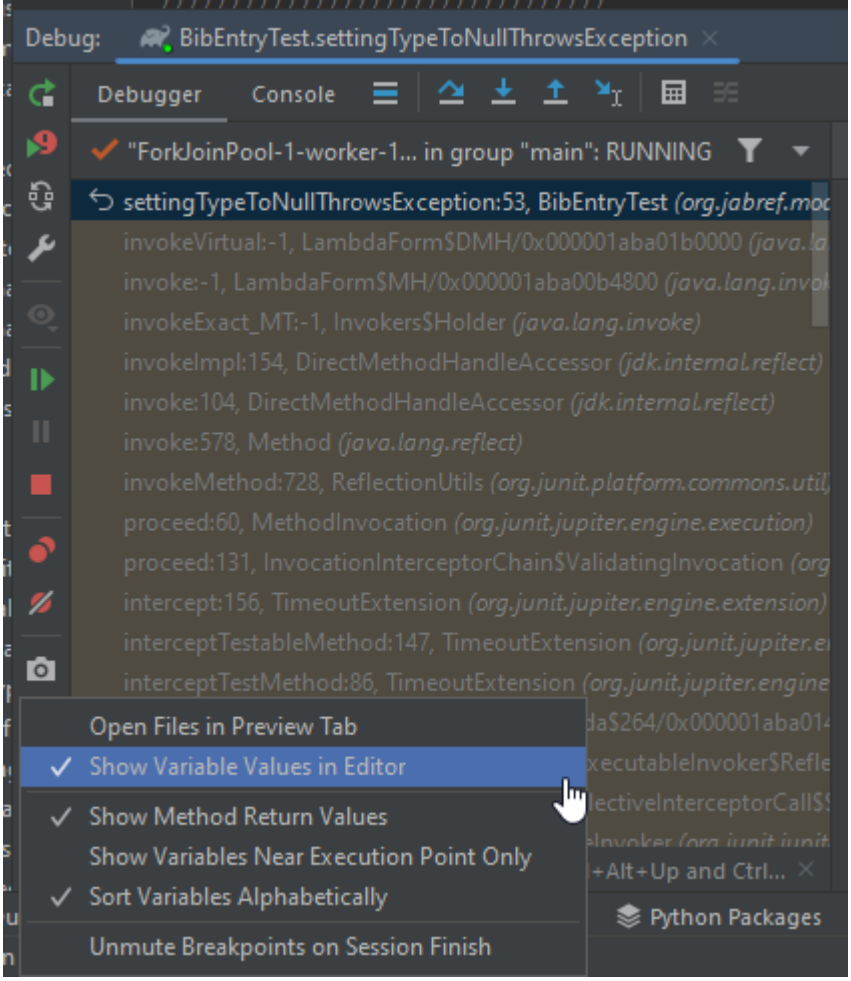


Figure: Debugger Configuration

JabRef's handling of BibTeX

The main class to handle a single BibTeX entry is `org.jabref.model.entry.BibEntry`. The content of a `.bib` file is handled in `org.jabref.model.database.BibDatabase`. Things not written in the `.bib` file, but required for handling are stored in `org.jabref.model.database.BibDatabaseContext`. For instance, this stores the mode of the library, which can be BibTeX or `biblatex`.

Standard BibTeX fields known to JabRef are modeled in `org.jabref.model.entry.field.StandardField`. A user-defined field not known to JabRef's code is modelled in `org.jabref.model.entry.field.UnknownField`. Typically, to get from a String to a `Field`, one needs to use `org.jabref.model.entry.field.FieldFactory#parseField(java.lang.String)`.

Cross-references

BibTeX allows for referencing other entries by the field `crossref` (`org.jabref.model.entry.field.StandardField#CROSSREF`). Note that BibTeX and `biblatex` handle this differently. The method `org.jabref.model.entry.BibEntry#getResolvedFieldOrAlias(org.jabref.model.entry.field.Field, org.jabref.model.database.BibDatabase)` handles this difference.

Code Quality

Code style checkers

JabRef has three code style checkers in place:

- Checkstyle for basic checks, such as wrong import order.
- Gradle Modernizer Plugin for Java library usage checks. It ensures that "modern" Java concepts are used (e.g., one should use `Deque` instead of `Stack`).
- OpenRewrite for advanced rules. OpenRewrite can also automatically fix issues. JabRef's CI toolchain does NOT automatically rewrite the source code, but checks whether OpenRewrite would rewrite something. As developer, one can execute `./gradlew rewriteRun` to fix the issues. Note that JabRef is available on the Moderne platform, too.

In case a check fails, the CI automatically adds a comment on the pull request.

Monitoring

We monitor the general source code quality at three places:

- codacy is a hosted service to monitor code quality. It thereby combines the results of available open source code quality checkers such as Checkstyle or PMD. The code quality analysis for JabRef is available at https://app.codacy.com/gh/JabRef/jabref/dashboard, especially the list of open issues. In case a rule feels wrong, it is most likely a PMD rule.
- codecov is a solution to check code coverage of test cases. The code coverage metrics for JabRef are available at https://codecov.io/github/JabRef/jabref.
- Teamscale is a popular German product analyzing code quality. The analysis results are available at https://demo.teamscale.com/findings.html#/jabref/?.

Up to date dependencies

We believe that updated dependencies are a sign of maintained code and thus an indicator of good quality. We use GitHub's dependabot to keep our versions up to date.

Moreover, we try to test JabRef with the latest Java Development Kit (JDK) builds. Our results can be seen at the Quality Outreach page.

Statistics

- 

- 

Background literature

We strongly recommend reading following two books on code quality:

- Java by Comparison is a book by three JabRef developers which focuses on code improvements close to single statements. It is fast to read and one gains much information from each recommendation discussed in the book.
- Effective Java is the standard book for advanced Java programming. Did you know that `enum` is the recommended way to enforce a singleton instance of a class? Did you know that one should refer to objects by their interfaces?

The principles we follow to ensure high code quality in JabRef is stated at our Development Strategy.

Custom SVG icons

JabRef uses Material Design Icons for most buttons on toolbars and panels. While most required icons are available, some specific ones cannot be found in the standard set, like Vim, Emacs, etc. Although custom icons might not fit the existing icons perfectly in style and level of detail, they will fit much better into JabRef than having a color pixel icon between all Material Design Icons.

This tutorial aims to describe the process of adding missing icons created in a vector drawing tool like Adobe Illustrator and packing them into a *true type font* (TTF) to fit seamlessly into the JabRef framework.

The process consists of 5 steps:

1. Download the template vector graphics from the Material Design Icons webpage. This gives you a set of existing underlying shapes that are typically used and the correct bounding boxes. You can design the missing icon based on this template and export it as an SVG file.
2. Pack the set of icons into a TTF with the help of the free IcoMoon tool.
3. Replace the existing `JabRefMaterialDesign.ttf` in the `src/main/resources/fonts` folder.
4. Adapt the class `org.jabref.gui.JabRefMaterialDesignIcon` to include all icons.
5. Adapt the class `org.jabref.gui.IconTheme` to make the new icons available in JabRef

Step 1. Designing the icon

Good icon design requires years of experience and cannot be covered here. Adapting color icons with a high degree of detail to look good in the flat, one-colored setting is an even harder task. Therefore, only 3 tips: 1. Look up some tutorials on icon design, 2. reuse the provided basic shapes in the template, and 3. export your icon in the SVG format.

Step 2. Packing the icons into a font

Use the IcoMoon tool for packing the icons.

1. Create a new set by importing the json file JabRefMaterialDesign.json.zip
2. Next to the icons, click on the hamburger menu, chose “Import to Set” to add a new icon (it will be added to the front) Rearrange them so that they have the same order as in `org.jabref.gui.JabRefMaterialDesignIcon`. This will avoid that you have to change the code

points for the existing glyphs. In the settings for your icon set, set the *Grid* to 24. This is important to get the correct spacing. The name of the font is `JabRefMaterialDesign`.

3. Next to the icons, click on the hamburger menu and click “Select all”.
4. Proceed with the font creating, set the font property name to `JabRefMaterialDesign` When your icon-set is ready, select all of them and download the font-package.

Step 3. Replace the existing `JabRefMaterialDesign.ttf`

Unpack the downloaded font-package and copy the `.ttf` file under `fonts` to `src/main/resources/fonts/JabRefMaterialDesign.ttf`.

Step 4. Adapt the class `org.jabref.gui.JabRefMaterialDesignIcon`

Inside the font-package will be a CSS file that specifies which icon (glyph) is at which code point. If you have ordered them correctly, you newly designed icon(s) will be at the end and you can simply append them to `org.jabref.gui.JabRefMaterialDesignIcon`:

```
    TEX_STUDIO("\ue900"),
    TEX_MAKER("\ue901"),
    EMACS("\ue902"),
    OPEN_OFFICE("\ue903"),
    VIM("\ue904"),
    LYX("\ue905"),
    WINEDT("\ue906"),
    ARXIV("\ue907");
```

Step 5. Adapt the class `org.jabref.gui.IconTheme`

If you added an icon that already existed (but not as flat Material Design Icon), then you need to change the appropriate line in `org.jabref.gui.IconTheme`, where the icon is assigned. If you created a new one, then you need to add a line. You can specify the icon like this:

```
APPLICATION_EMACS(JabRefMaterialDesignIcon.EMACS)
```

Error Handling in JabRef

Throwing and Catching Exceptions

Principles:

- All exceptions we throw should be or extend `JabRefException`; This is especially important if the message stored in the Exception should be shown to the user. `JabRefException` has already implemented the `getLocalizedMessage()` method which should be used for such cases (see details below!).
- Catch and wrap all API exceptions (such as `IOExceptions`) and rethrow them
  - Example:

```
try {
    // ...
} catch (IOException ioe) {
    throw new JabRefException("Something went wrong...",
        Localization.lang("Something went wrong...", ioe);
}
```

- Never, ever throw and catch `Exception` or `Throwable`
- Errors should only be logged when they are finally caught (i.e., logged only once). See **Logging** for details.
- If the Exception message is intended to be shown to the User in the UI (see below) provide also a localizedMessage (see `JabRefException`).

(Rationale and further reading: https://www.baeldung.com/java-exceptions)

Outputting Errors in the UI

Principle: Error messages shown to the User should not contain technical details (e.g., underlying exceptions, or even stack traces). Instead, the message should be concise, understandable for non-programmers and localized. The technical reasons (and stack traces) for a failure should only be logged.

To show error message two different ways are usually used in JabRef:

- showing an error dialog
- updating the status bar at the bottom of the main window

```
TODO: Usage of status bar and `DialogService`
```

Event Bus and Event System

What the EventSystem is used for

Many times there is a need to provide an object on many locations simultaneously. This design pattern is quite similar to Java's Observer, but it is much simpler and readable while having the same functional sense.

Main principle

`EventBus` represents a communication line between multiple components. Objects can be passed through the bus and reach the listening method of another object which is registered on that `EventBus` instance. Hence, the passed object is available as a parameter in the listening method.

Register to the `EventBus`

Any listening method has to be annotated with `@Subscribe` keyword and must have only one accepting parameter. Furthermore, the object which contains such listening method(s) has to be registered using the `register(Object)` method provided by `EventBus`. The listening methods can be overloaded by using different parameter types.

Posting an object

`post(object)` posts an object through the `EventBus` which has been used to register the listening/subscribing methods.

Short example

```
/* Listener.java */

import com.google.common.eventbus.Subscribe;

public class Listener {

   private int value = 0;

   @Subscribe
```

```
    public void listen(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }
}
```

```
/* Main.java */

import com.google.common.eventbus.EventBus;

public class Main {
    private static EventBus eventBus = new EventBus();

    public static void main(String[] args) {
        Main main = new Main();
        Listener listener = new Listener();
        eventBus.register(listener);
        eventBus.post(1); // 1 represents the passed event

        // Output should be 1
        System.out.println(listener.getValue());
    }
}
```

Event handling in JabRef

The `event` package contains some specific events which occur in JabRef.

For example: Every time an entry was added to the database a new `EntryAddedEvent` is sent through the `eventBus` which is located in `BibDatabase`.

If you want to catch the event you'll have to register your listener class with the `registerListener(Object listener)` method in `BibDatabase`. `EntryAddedEvent` provides also methods to get the inserted `BibEntry`.

Frequently Asked Questions (FAQ)

Following is a list of common errors encountered by developers which lead to failing tests, with their common solutions:

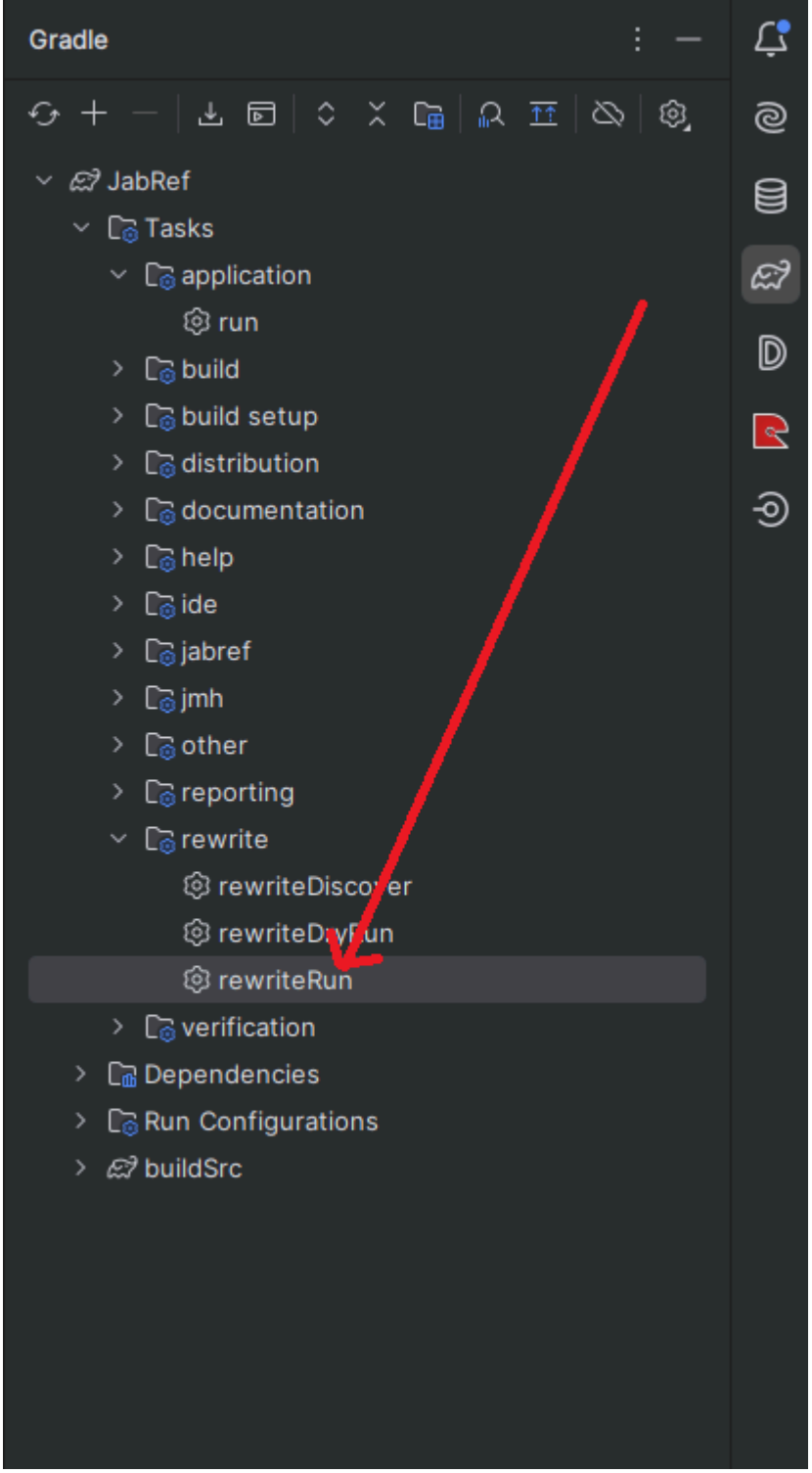
Failing tests

Failing **Checkstyle** tests

JabRef follows a pre-defined style of code for uniformity and maintainability that must be adhered to during development. To set up warnings and auto-fixes conforming to these style rules in your IDE, follow Step 3 of the process to set up a local workspace in the documentation. Ideally, follow all the set up rules in the documentation end-to-end to avoid typical set-up errors.
Note: The steps provided in the documentation are for IntelliJ, which is the preferred IDE for Java development. The `checkstyle.xml` is also available for VSCode, in the same directory as mentioned in the steps.

Failing **OpenRewrite** tests

Execute the Gradle task `rewriteRun` from the `rewrite` group of the Gradle Tool window in IntelliJ to apply the automated refactoring and pass the test:



Background: OpenRewrite is an automated refactoring ecosystem for source code.

`org.jabref.logic.l10n.LocalizationConsistencyTest findMissingLocalizationKeys` FAILED
You have probably used Strings that are visible on the UI (to the user) but not wrapped them using `Localization.lang(...)` and added them to the localization properties file.

Read more about the background and format of localization in JabRef here.

`org.jabref.logic.l10n.LocalizationConsistencyTest findObsoleteLocalizationKeys` FAILED
Navigate to the unused key-value pairs in the file and remove them. You can always click on the details of the failing test to pinpoint which keys are unused.

Background: There are localization keys in the localization properties file that are not used in the code, probably due to the removal of existing code. Read more about the background and

format of localization in JabRef here.

`org.jabref.logic.citationstyle.CitationStyle discoverCitationStyles` ERROR: Could not find any citation style. Tried with /ieee.csl.

Check the directory `src/main/resources/csl-styles`. If it is missing or empty, run `git submodule update`. Now, check inside if `ieee.csl` exists. If it does not, run `git reset --hard` **inside that directory**.

`java.lang.IllegalArgumentException`: Unable to load locale en-US ERROR: Could not generate BibEntry citation. The CSL engine could not create a preview for your item.

Check the directory `src/main/resources/csl-locales`. If it is missing or empty, run `git submodule update`. If still not fixed, run `git reset --hard` **inside that directory**.

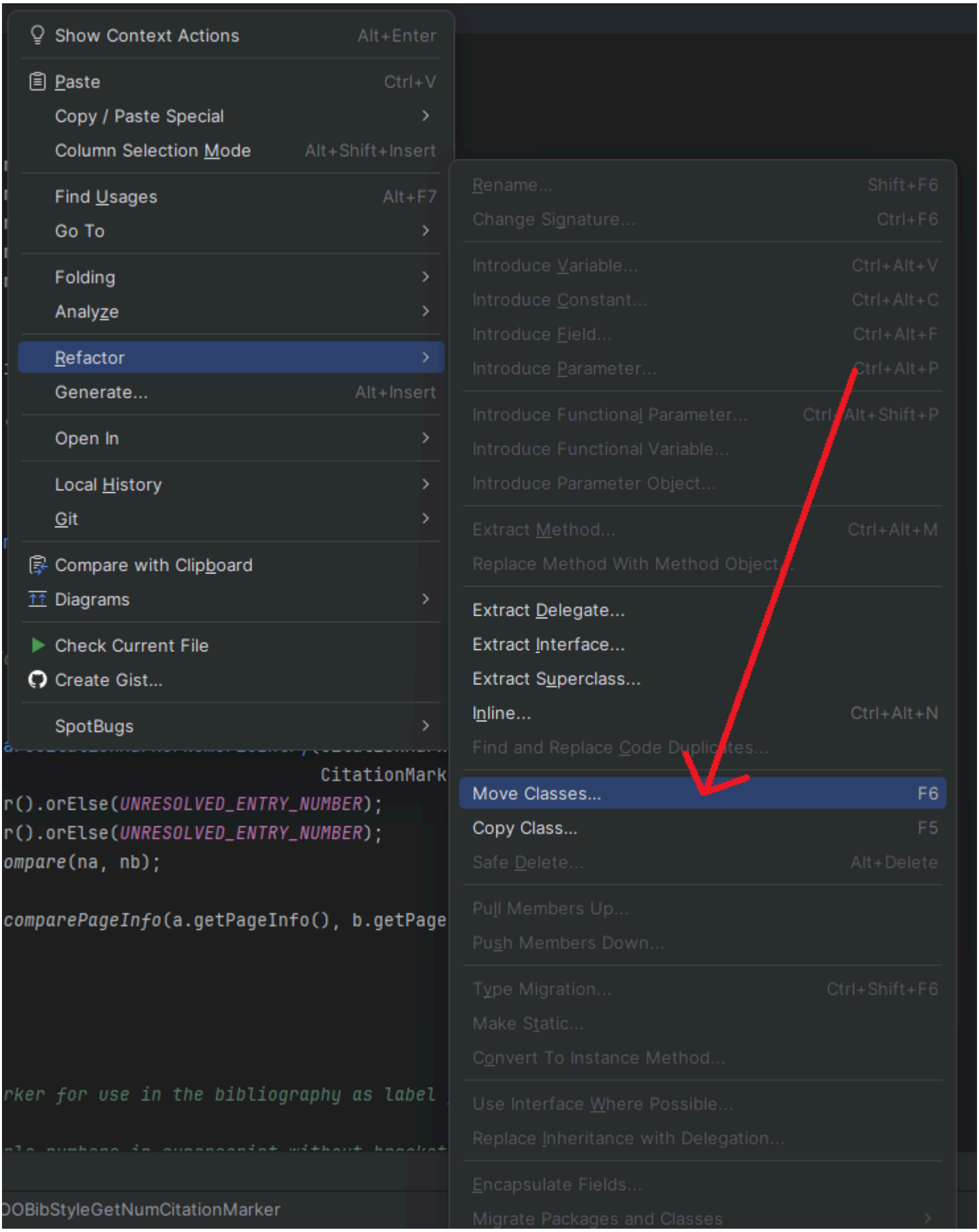
`org.jabref.architecture.MainArchitectureTest restrictStandardStreams` FAILED

Check if you've used `System.out.println(...)` (the standard output stream) to log anything into the console. This is an architectural violation, as you should use the Logger instead for logging. More details on how to log can be found here.

`org.jabref.architecture.MainArchitectureTest doNotUseLogicInModel` FAILED

One common case when this test fails is when you put any class purely containing business logic inside the `model` package (i.e., inside the directory `org/jabref/model/`). To fix this, shift the class to a sub-package within the `logic` package (i.e., the directory `org/jabref/logic/`). An efficient way to do this is to use Intellij's built-in refactoring capabilities - right-click on the file, go to "Refactor" and use "Move Class". The import statement for all the classes using this

class will be automatically adjusted according to the new location.



More information on the architecture can be found at ../getting-into-the-code/high-level-documentation.md.

`Check external href links in the documentation / check-links (push)` FAILED

This test is triggered when any kind of documentation is touched (be it the JabRef docs, or JavaDoc in code). If you changed something in the documentation, and particularly added/changed any links (to external files or websites), check if the links are correct and working. If you didn't change/add any link, or added correct links, the test is most probably failing due to any of the existing links being broken, and thus can be ignored (in the context of your contribution).

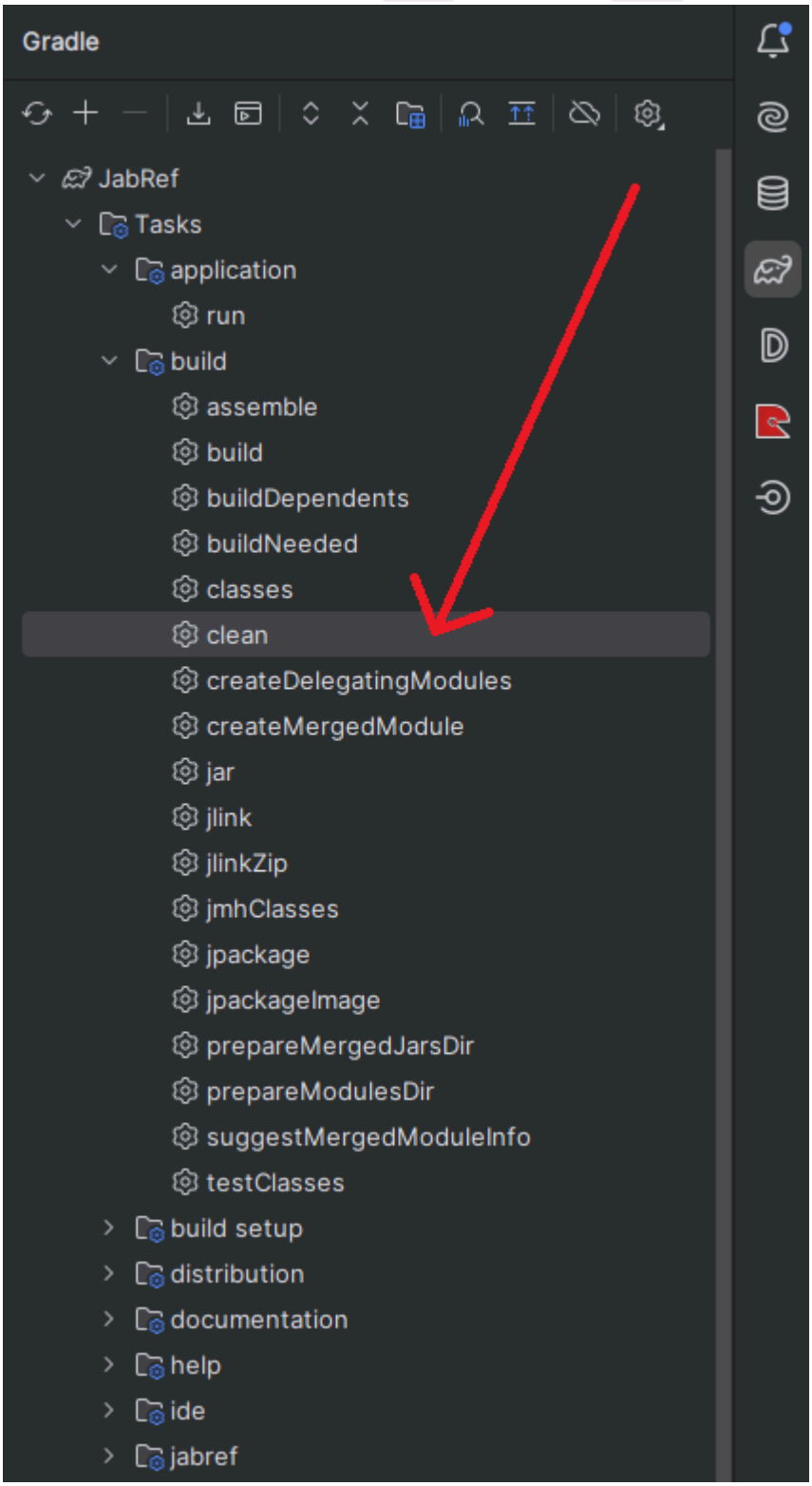
Failing **Fetcher** tests

Fetcher tests are run when any file in the `../fetcher` directory has been touched. If you have changed any fetcher logic, check if the changes are correct. You can look for more details on how to locally run fetcher tests here. Otherwise, since these tests depend on remote services, their failure can also be caused by the network or an external server, and thus can be ignored in the context of your contribution. For more information, you can look here.

Gradle outputs

```
ANTLR Tool version 4.12.0 used for code generation does not match the current runtime version 4.13.1
```

Execute the Gradle task `clean` from the `build` group of the Gradle Tool Window in IntelliJ:

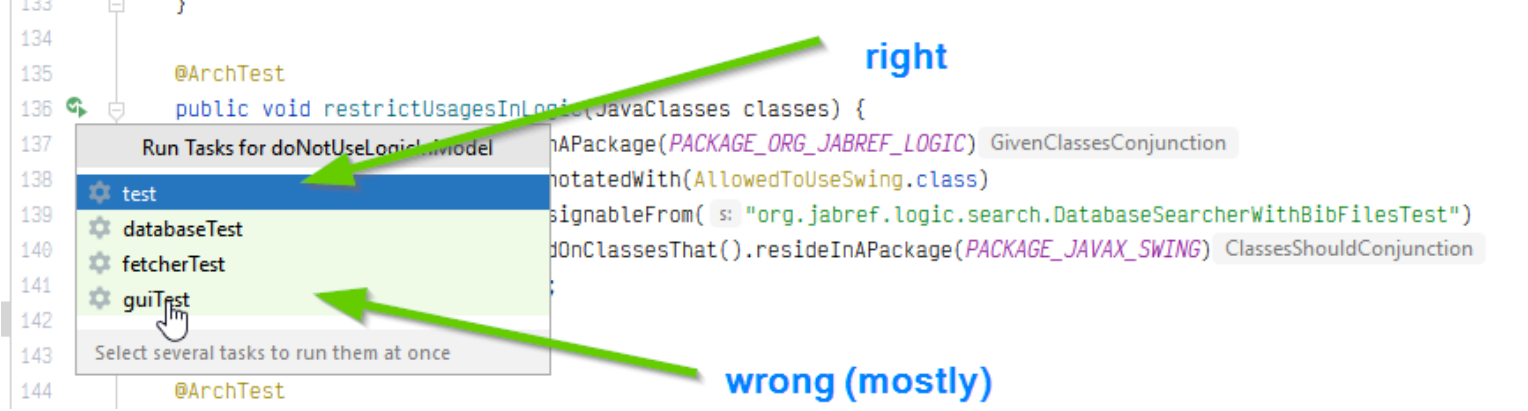


`BstVMVisitor.java:157: error: package BstParser does not exist`

Execute gradle task `clean` from the `build` group of the Gradle Tool Window in IntelliJ.

`No test candidates found`

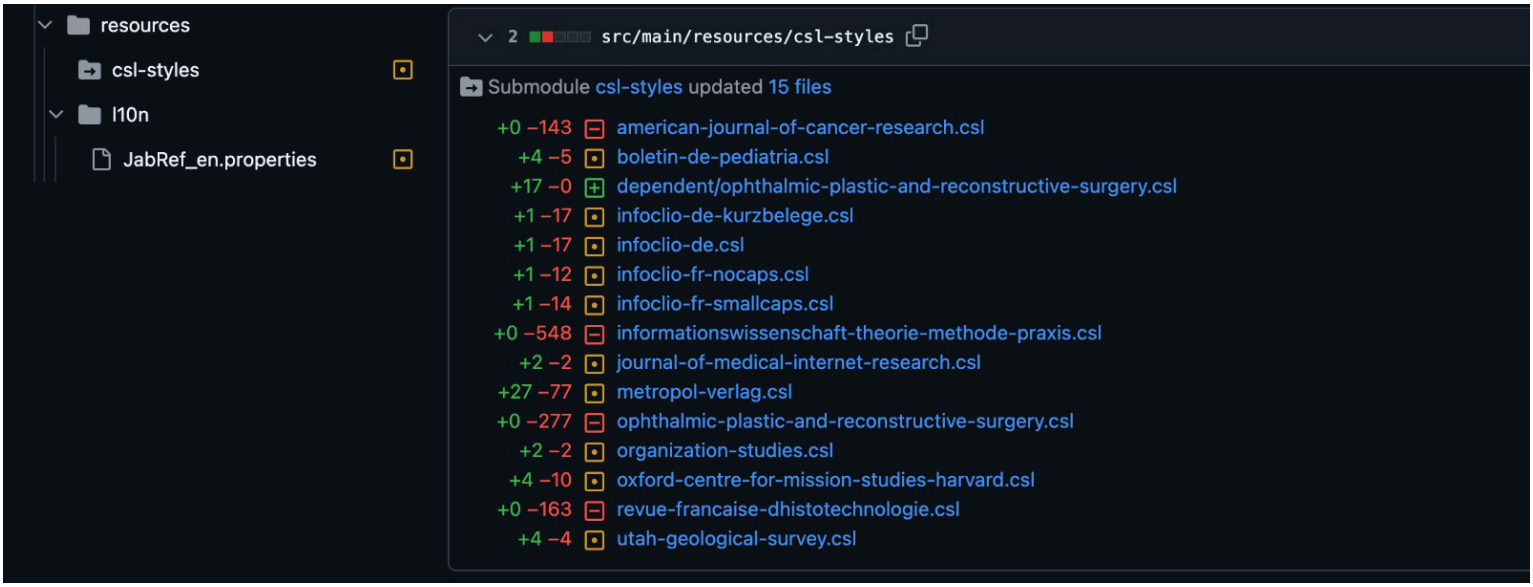
You probably chose the wrong gradle task:



Submodules

The problem

Sometimes, when contributing to JabRef, you may see `abbrv.jabref.org` or `csl-styles` or `csl-locales` among the changed files in your pull request. This means that you have accidentally committed your local submodules into the branch.



Context

JabRef needs external submodules (such as CSL style files) for some of its respective features. These are cloned once when you set up a local development environment, using `--recurse-submodules` (you may have noticed). These submodules, in the main branch, are automatically periodically updated but not fetched into local again when you pull, as they are set to be ignored in `.gitmodules` (this is to avoid merge conflicts). So when remote has updated submodules, and your local has the old ones, when you stage all files, these changes are noticed.

What's strange (mostly an IntelliJ bug): Regardless of CLI or GUI, These changes should ideally not be noticed on staging, as per the `.gitmodules` configuration. However, that is somehow overruled when using IntelliJ's CLI.

Fix

For `csl-styles`:

```
git merge origin/main
git checkout main -- src/main/resources/csl-styles
... git commit ...
git push
```

And similarly for `csl-locales` or `abbrv.jabref.org`.

ALTERNATIVE METHOD (IF THE ABOVE DOESN'T WORK)

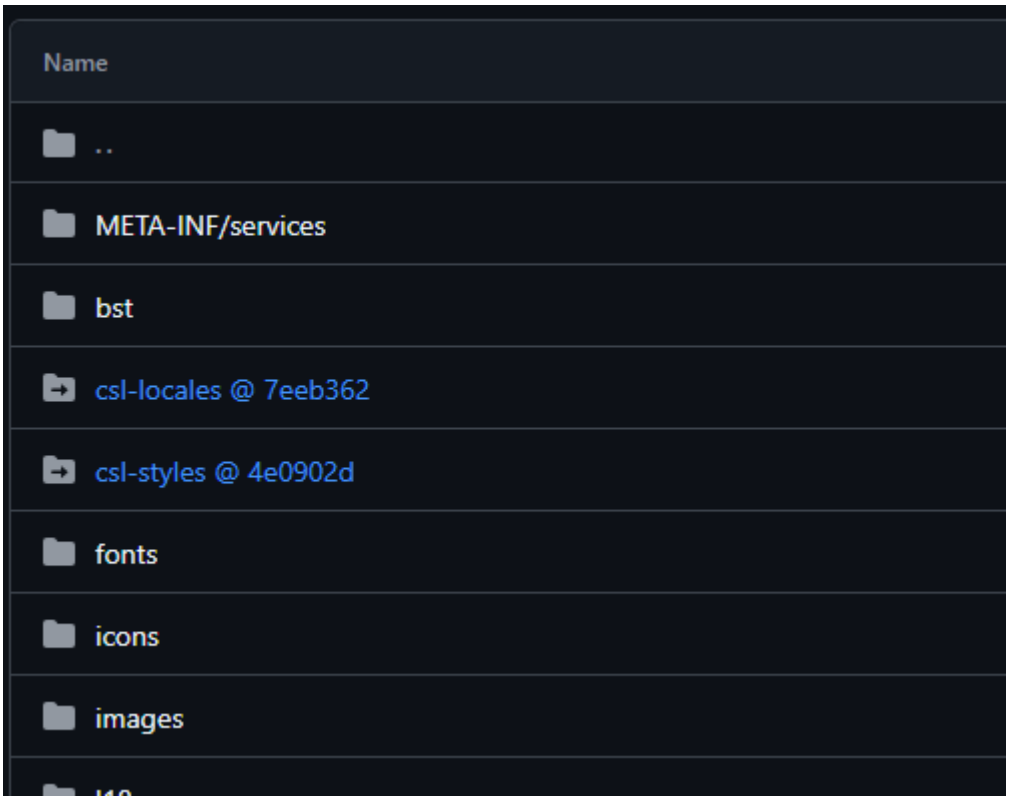
1. Edit `.gitmodules`: comment out `ignore = all` (for the respective submodules you are trying to reset)

   ```
   # ignore = all
   ```

2. `cd` into the changed submodules directory (lets say `csl-styles` was changed):

   ```
   cd src/main/resources/csl-styles
   ```

3. Find the latest submodule commit id from remote (github):

   Here, in the case of `csl-styles`, it is `4e0902d`.

4. Checkout the commit:

   ```
   git checkout 4e0902d
   ```

5. Now, IntelliJ's commit tab will notice that the submodules have been modified. This means we are on the right track.

6. Use IntelliJ’s git manager (commit tab) or `git gui` to commit submodule changes only. Repeat steps 2-5 for other submodules that are shown as modified in the PR. Then, push these changes.
7. Revert the changes in `.gitmodules` (that you made in step 1).

Prevention

To avoid this, avoid staging using `git add .` from CLI. Preferably use a GUI-based git manager, such as the one built in IntelliJ or open git gui from the command line. Even if you accidentally stage them, don’t commit all files, selectively commit the files you touched using the GUI based tool, and push.

Fetchers

Fetchers are the implementation of the search using online services. Some fetchers require API keys to get them working. To get the fetchers running in a JabRef development setup, the keys need to be placed in the respective environment variable. The following table lists the respective fetchers, where to get the key from and the environment variable where the key has to be placed.

Service	Key Source	Environment Variable	R
IEEEXplore	IEEE Xplore API portal	`IEEEAPIKey`	20
MathSciNet	(none)	(none)	De on cu ne
SAO/NASA Astrophysics Data System	ADS UI	`AstrophysicsDataSystemAPIKey`	50 ca
ScienceDirect		`ScienceDirectApiKey`	
SemanticScholar	https://www.semanticscholar.org/product/api#api-key-form	`SemanticScholarApiKey`	
Springer Nature	Springer Nature API Portal	`SpringerNatureAPIKey`	50 ca
Zentralblatt Math	(none)	(none)	De on cu ne
Biodiversity Heritage Library	Biodiversitylibrary	`BiodiversityHeritageApiKey`	-

“Depending on the current network” means that it depends on whether your request is routed through a network having paid access. For instance, some universities have subscriptions to MathSciNet.

On Windows, you have to log off and log on to let IntelliJ know about the environment variable change. Execute the gradle task `processResources` in the group “others” within IntelliJ to ensure

the values have been correctly written. Now, the fetcher tests should run without issues.

JabRef supports different kinds of fetchers:

- `EntryBasedFetcher`: Completes an existing bibliographic entry with information retrieved by the fetcher
- `FulltextFetcher`: Searches for a PDF for an exiting bibliography entry
- `SearchBasedFetcher`: Searches providers using a given query and returns a set of (new) bibliography entry. The user-facing side is implemented in the UI described at https://docs.jabref.org/collect/import-using-online-bibliographic-database.

There are more fetchers supported by JabRef. Investigate the package `org.jabref.logic.importer`. Another possibility is to investigate the inheritance relation of `WebFetcher` (Ctrl+H in IntelliJ).

Fulltext Fetchers

- all fulltext fetchers run in parallel
- the result with the highest priority wins
- `InterruptedException` `ExecutionException` `CancellationException` are ignored

Trust Levels

- `SOURCE` (highest): definitive URL for a particular paper
- `PUBLISHER`: any publisher library
- `PREPRINT`: any preprint library that might include non final publications of a paper
- `META_SEARCH`: meta search engines
- `UNKNOWN` (lowest): anything else not fitting the above categories

Current trust levels

All fetchers are contained in the package `org.jabref.logic.importer.fetcher`. Here we list the trust levels of some of them:

- DOI: SOURCE, as the DOI is always forwarded to the correct publisher page for the paper
- ScienceDirect: Publisher
- Springer: Publisher
- ACS: Publisher
- IEEE: Publisher
- Google Scholar: META_SEARCH, because it is a search engine
- Arxiv: PREPRINT, because preprints are published there
- OpenAccessDOI: META_SEARCH

Reasoning:

- A DOI uniquely identifies a paper. Per definition, a DOI leads to the right paper. Everything else is good guessing.
- We assume the DOI resolution surely points to the correct paper and that publisher fetches may have errors: For instance, a title of a paper may lead to different publications of it. One the conference version, the other the journal version. –> the PDF could be chosen randomly

Code was first introduced at PR#3882.

Background on embedding the keys in JabRef

The keys are placed into the `build.properties` file.

```
springerNatureAPIKey=${springerNatureAPIKey}
```

In `build.gradle`, these variables are filled:

```
"springerNatureAPIKey" : System.getenv('SpringerNatureAPIKey')
```

The `BuildInfo` class reads from that file and the key needs to be put into the map of default API keys in `JabRefCliPreferences::getDefaultFetcherKeys`.

```
keys.put(SpringerFetcher.FETCHER_NAME, buildInfo.springerNatureAPIKey);
```

The fetcher api key can then be obtained by calling the preferences.

```
importerPreferences.getApiKey(SpringerFetcher.FETCHER_NAME);
```

When executing `./gradlew run`, gradle executes `processResources` and populates `build/build.properties` accordingly. However, when working directly in the IDE, Eclipse keeps reading `build.properties` from `src/main/resources`. In IntelliJ, the task `JabRef Main` is executing `./gradlew processResources` before running JabRef from the IDE to ensure the `build.properties` is properly populated.

Committing and pushing changes to fetcher files

Fetcher tests are run when a PR contains changes touching any file in the `src/main/java/org/jabref/logic/importer/fetcher/` directory. Since these tests rely on remote services, some of them may fail due to the network or the external server.

To learn more about doing fetcher tests locally, see Fetchers in tests in Testing.

HTTP Server

JabRef has a built-in http server. For example, the resource for a library is implemented at `org.jabref.http.server.LibraryResource`.

Start http server

The class starting the server is `org.jabref.http.server.Server`.

Test files to server can be passed as arguments. If no files are passed, the last opened files are served. If that list is also empty, the file `src/main/resources/org/jabref/http/server/http-server-demo.bib` is served.

Starting with gradle

Does not work.

Current try:

```
./gradlew run -Pcomment=httpserver
```

However, there are with `ForkJoin` (discussion at https://discuss.gradle.org/t/is-it-ok-to-use-collection-parallelstream-or-other-potentially-multi-threaded-code-within-gradle-plugin-code/28003)

Gradle output:

```
> Task :run
2023-04-22 11:30:59 [main] org.jabref.http.server.Server.main()
DEBUG: Libraries served: [C:\git-repositories\jabref-all\jabref\src\main\resources\org\jabref\http\serve
2023-04-22 11:30:59 [main] org.jabref.http.server.Server.startServer()
DEBUG: Starting server...
<============-> 92% EXECUTING [2m 27s]
> :run
```

IntelliJ output, if `org.jabref.http.server.Server#main` is executed:

```
DEBUG: Starting server...
2023-04-22 11:44:59 [ForkJoinPool.commonPool-worker-1] org.glassfish.grizzly.http.server.NetworkListener
INFO: Started listener bound to [localhost:6051]
2023-04-22 11:44:59 [ForkJoinPool.commonPool-worker-1] org.glassfish.grizzly.http.server.HttpServer.star
```

```
INFO: [HttpServer] Started.
2023-04-22 11:44:59 [ForkJoinPool.commonPool-worker-1] org.jabref.http.server.Server.lambda$startServer$
DEBUG: Server started.
```

Developing with IntelliJ

IntelliJ Ultimate offers a Markdown-based http-client. One has to open the file `src/test/java/org/jabref/testutils/interactive/http/rest-api.http`. Then, there are play buttons appearing for interacting with the server.

Get SSL Working

When interacting with the Microsoft Word AddIn, a SSL-based connection is required. The Word-AddIn is currentely under development.

(Based on https://stackoverflow.com/a/57511038/873282)

Howto for Windows - other operating systems work similar:

1. As admin `choco install mkcert`
2. As admin: `mkcert -install`
3. `cd %APPDATA%\..\local\org.jabref\jabref\ssl`
4. `mkcert -pkcs12 jabref.desktop jabref localhost 127.0.0.1 ::1`
5. Rename the file to `server.p12`

Note: If you do not do this, you get following error message:

```
Could not find server key store C:\Users\USERNAME\AppData\Local\org.jabref\jabref\ssl\server.p12.
```

JavaFX

> JavaFX is an open source, next generation client application platform for desktop, mobile and embedded systems based on JavaSE. It is a collaborative effort by many individuals and companies with the goal of producing a modern, efficient, and fully featured toolkit for developing rich client applications.

JavaFX is used on JabRef for the user interface.

Resources

- JavaFX Documentation project: Collected information on JavaFX in a central place
- curated list of awesome JavaFX frameworks, libraries, books and etc...
- FXTutorials A wide range of practical tutorials focusing on Java, JavaFX and FXGL
- ControlsFX amazing collection of controls
- CSS Reference
- mvvm framework
- Validation framework
- additional bindings or EasyBind
- Undo manager
- Docking manager or DockFX
- Kubed: data visualization (inspired by d3)
- Foojay Java and JavaFX tutorials

Resources of historical interest

- FXExperience JavaFX Links of the week

Architecture: Model - View - (Controller) - ViewModel (MV(C)VM)

The goal of the MVVM architecture is to separate the state/behavior from the appearance of the ui. This is archived by dividing JabRef into different layers, each having a clear responsibility.

- The *Model* contains the business logic and data structures. These aspects are again encapsulated in the *logic* and *model* package, respectively.
- The *View* controls the appearance and structure of the UI. It is usually defined in a *FXML* file.
- *View model* converts the data from logic and model in a form that is easily usable in the gui. Thus it controls the state of the View. Moreover, the ViewModel contains all the logic needed to change the current state of the UI or perform an action. These actions are usually passed down to the *logic* package, after some data validation. The important aspect is that the ViewModel

contains all the ui-related logic but does *not* have direct access to the controls defined in the View. Hence, the ViewModel can easily be tested by unit tests.

- The *Controller* initializes the view model and binds it to the view. In an ideal world all the binding would already be done directly in the FXML. But JavaFX's binding expressions are not yet powerful enough to accomplish this. It is important to keep in mind that the Controller should be as minimalistic as possible. Especially one should resist the temptation to validate inputs in the controller. The ViewModel should handle data validation! It is often convenient to load the FXML file directly from the controller.

The only class which access model and logic classes is the ViewModel. Controller and View have only access the ViewModel and never the backend. The ViewModel does not know the Controller or View.

More details about the MVVM pattern can be found in an article by Microsoft and in an article focusing on the implementation with JavaFX.

Example

VIEWMODEL

- The ViewModel should derive from `AbstractViewModel`

```
public class MyDialogViewModel extends AbstractViewModel {
}
```

- Add a (readonly) property as a private field and generate the getters according to the JavaFX bean conventions:

```
private final ReadOnlyStringWrapper heading = new ReadOnlyStringWrapper();

public ReadOnlyStringProperty headingProperty() {
    return heading.getReadOnlyProperty();
}

public String getHeading() {
    return heading.get();
}
```

- Create constructor which initializes the fields to their default values. Write tests to ensure that everything works as expected!

```
public MyDialogViewModel(Dependency dependency) {
    this.dependency = Objects.requireNonNull(dependency);
    heading.set("Hello " + dependency.getUserName());
}
```

- Add methods which allow interaction. Again, don't forget to write tests!

```
public void shutdown() {
    heading.set("Goodbye!");
}
```

VIEW - CONTROLLER

- The “code-behind” part of the view, which binds the `View` to the `ViewModel`.
- The usual convention is that the controller ends on the suffix `*View`. Dialogs should derive from `BaseDialog`.

```
public class AboutDialogView extends BaseDialog<Void>
```

- You get access to nodes in the FXML file by declaring them with the `@FXML` annotation.

```
@FXML protected Button helloButton;
@FXML protected ImageView iconImage;
```

- Dependencies can easily be injected into the controller using the `@Inject` annotation.

```
@Inject private DialogService dialogService;
```

- It is convenient to load the FXML-view directly from the controller class.

  The FXML file is loaded using `ViewLoader` based on the name of the class passed to `view`. To make this convention-over-configuration approach work, both the FXML file and the View class should have the same name and should be located in the same package.

  Note that fields annotated with `@FXML` or `@Inject` only become accessible after `ViewLoader.load()` is called.

  a `View` class that loads the FXML file.

```
private Dependency dependency;

public AboutDialogView(Dependency dependency) {
        this.dependency = dependency;

        this.setTitle(Localization.lang("About JabRef"));

        ViewLoader.view(this)
                .load()
                .setAsDialogPane(this);
}
```

- Dialogs should use setResultConverter to convert the data entered in the dialog to the desired result. This conversion should be done by the view model and not the controller.

```
setResultConverter(button -> {
    if (button == ButtonType.OK) {
        return viewModel.getData();
    }
    return null;
});
```

- The initialize method may use data-binding to connect the ui-controls and the `ViewModel`. However, it is recommended to do as much binding as possible directly in the FXML-file.

```
@FXML
private void initialize() {
    viewModel = new AboutDialogViewModel(dialogService, dependency, ...);

    helloLabel.textProperty().bind(viewModel.helloMessageProperty());
}
```

- calling the view model:

```
@FXML
private void openJabrefWebsite() {
    viewModel.openJabrefWebsite();
}
```

VIEW - FXML

The view consists a FXML file `MyDialog.fxml` which defines the structure and the layout of the UI. Moreover, the FXML file may be accompanied by a style file that should have the same name as the FXML file but with a `css` ending, e.g., `MyDialog.css`. It is recommended to use a graphical design tools like SceneBuilder to edit the FXML file. The tool Scenic View is very helpful in debugging styling issues.

FXML

The following expressions can be used in FXML attributes, according to the official documentation

Type	Expression	Value point to	Remark
Location	`@image.png`	path relative to the current FXML file	
Resource	`%textToBeTranslated`	key in ResourceBundle	
Attribute variable	`$idOfControl` or `$variable`	named control or variable in controller (may be path in the namespace)	resolved only once at load time

Type	Expression	Value point to	Remark
Expression binding	`${expression}`	expression, for example `textField.text`	changes to source are propagated
Bidirectional expression binding	`#{expression}`	expression	changes are propagated in both directions (not yet implemented in JavaFX, see feature request)
Event handler	`#nameOfEventHandler`	name of the event handler method in the controller	
Constant	`<text><Strings fx:constant="MYSTRING"/></text>`	constant (here `MYSTRING` in the `Strings` class)	

JavaFX Radio Buttons Example

All radio buttons that should be grouped together need to have a ToggleGroup defined in the FXML code Example:

```
<VBox>
            <fx:define>
                <ToggleGroup fx:id="citeToggleGroup"/>
            </fx:define>
            <children>
                <RadioButton fx:id="inPar" minWidth="-Infinity" mnemonicParsing="false"
                             text="%Cite selected entries between parenthesis" toggleGroup="$citeToggleGroup"/>
                <RadioButton fx:id="inText" minWidth="-Infinity" mnemonicParsing="false"
                             text="%Cite selected entries with in-text citation" toggleGroup="$citeToggleGroup"/>
                <Label minWidth="-Infinity" text="%Extra information (e.g. page number)"/>
                <TextField fx:id="pageInfo"/>
            </children>
</VBox>
```

JavaFX Dialogs

All dialogs should be displayed to the user via `DialogService` interface methods. `DialogService` provides methods to display various dialogs (including custom ones) to the user. It also ensures the displayed dialog opens on the correct window via `initOwner()` (for cases where the user has multiple screens). The following code snippet demonstrates how a custom dialog is displayed to the user:

```
dialogService.showCustomDialog(new DocumentViewerView());
```

If an instance of `DialogService` is unavailable within current class/scope in which the dialog needs to be displayed, `DialogService` can be instantiated via the code snippet shown as follows:

```
DialogService dialogService = Injector.instantiateModelOrService(DialogService.class);
```

Properties and Bindings

JabRef makes heavy use of Properties and Bindings. These are wrappers around Observables. A good explanation on the concept can be found here: JavaFX Bindings and Properties

Features missing in JavaFX

- bidirectional binding in FXML, see official feature request

JPackage: Creating a binary and debug it

JabRef uses jpackage to build binary application bundles and installers for Windows, Linux, and macOS. For Gradle, we use the Badass JLink Plugin.

Build Windows binaries locally

Preparation: Install WiX Toolset

1. Open administrative shell
2. Use Chocolatey to install it: `choco install wixtoolset`

Create the application image:

```
./gradlew -PprojVersion="5.0.50013" -PprojVersionInfo="5.0-ci.13--2020-03-05--c8e5924" jpackageImage
```

Create the installer:

```
./gradlew -PprojVersion="5.0.50013" -PprojVersionInfo="5.0-ci.13--2020-03-05--c8e5924" jpackage
```

Debugging jpackage installations

Sometimes issues with modularity only arise in the installed version and do not occur if you run from source. Using remote debugging, it's still possible to hook your IDE into the running JabRef application to enable debugging.

Debugging on Windows

1. Open `build.gradle`, under jlink options remove `--strip-debug`
2. Build using `jpackageImage` (or let the CI build a new version)
3. Modify the `build\image\JabRef\runtime\bin\Jabref.bat` file, replace the last line with

   ```
   pushd %DIR% & %JAVA_EXEC% -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n -p "
   ```

4. Open your IDE and add a "Remote Debugging Configuration" for `localhost:8000`
5. Start JabRef by running the above bat file
6. Connect with your IDE using remote debugging

Localization

More information about this topic from the translator side is provided at Translating JabRef Interface.

All labeled UI elements, descriptions and messages shown to the user should be localized, i.e., should be displayed in the chosen language.

JabRef uses ResourceBundles (see Oracle Tutorial) to store `key=value` pairs for each String to be localized.

Localization in Java code

To show a localized String the following `org.jabref.logic.l10n.Localization` has to be used. The Class currently provides three methods to obtain translated strings:

```
    public static String lang(String key);

    public static String lang(String key, String... params);

    public static String menuTitle(String key, String... params);
```

The actual usage might look like:

```
    Localization.lang("Get me a translated String");
    Localization.lang("Using %0 or more %1 is also possible", "one", "parameter");
    Localization.menuTitle("Used for Menus only");
```

Localization in FXML

To write a localized string in FXML file, prepend it with `%`, like in this code:

```
<HBox alignment="CENTER_LEFT">
    <Label styleClass="space-after" text="%Want to help?" wrapText="true"/>
    <Hyperlink onAction="#openDonation" text="%Make a donation"/>
    <Label styleClass="space" text="%or" wrapText="true"/>
    <Hyperlink onAction="#openGithub" text="%get involved"/>
</HBox>
```

General hints

- Use the String you want to localize directly, do not use members or local variables: `Localization.lang("Translate me");` instead of `Localization.lang(someVariable)` (possibly in the form `someVariable = Localization.lang("Translate me")`
- Use `%x`-variables where appropriate: `Localization.lang("Exported %0 entry(s).", number)` instead of `Localization.lang("Exported ") + number + Localization.lang(" entry(s).");`
- Use a full stop/period (“.”) to end full sentences
- For pluralization, use a combined form. E.g., `Localization.lang("checked %0 entry(s)")`.

Checking for correctness

The tests in `org.jabref.logic.l10n.LocalizationConsistencyTest` check whether translation strings appear correctly in the resource bundles.

Adding a new key

1. Add new `Localization.lang("KEY")` to Java file. Run the `org.jabref.logic.LocalizationConsistencyTest`.
2. Tests fail. In the test output a snippet is generated which must be added to the English translation file.
3. Add snippet to English translation file located at `src/main/resources/l10n/JabRef_en.properties`
4. Please do not add translations for other languages directly in the properties. They will be overwritten by Crowdin

Adding a new Language

1. Add the new Language to the Language enum in https://github.com/JabRef/jabref/blob/master/src/main/java/org/jabref/logic/l10n/Language.java
2. Create an empty <locale code>.properties file
3. Configure the new language in Crowdin

If the language is a variant of a language `zh_CN` or `pt_BR` it is necessary to add a language mapping for Crowdin to the crowdin.yml file in the root. Of course the properties file also has to be named according to the language code and locale.

Background information

The localization is tested via the class LocalizationConsistencyTest.

Logging

JabRef uses the logging facade SLF4j. All log messages are passed internally to tinylog which handles any filtering, formatting and writing of log messages.

Obtaining a logger for a class:

```
private static final Logger LOGGER = LoggerFactory.getLogger(<ClassName>.class);
```

Please always use `LOGGER.debug` for debugging.

Example:

```
String example = "example";
LOGGER.debug("Some state {}", example);
```

Enable logging in `tinylog.properties`:

```
level@org.jabref.example.ExampleClass = debug
```

If the logging event is caused by an exception, please add the exception to the log message as:

```
  catch (SomeException e) {
     LOGGER.warn("Warning text.", e);
     ...
  }
```

When running tests, `tinylog-test.properties` is used. It is located under `src/test/resources`. As default, only `info` is logged. When developing, it makes sense to use `debug` as log level. One can change the log level per class using the pattern `level@class=debug` is set to `debug`. In the `.properties` file, this is done for `org.jabref.model.entry.BibEntry`.

Further reading

SLF4J also support parameterized logging, e.g. if you want to print out multiple arguments in a log statement use a pair of curly braces (`{}`). Head to https://www.slf4j.org/faq.html#logging_performance for examples.

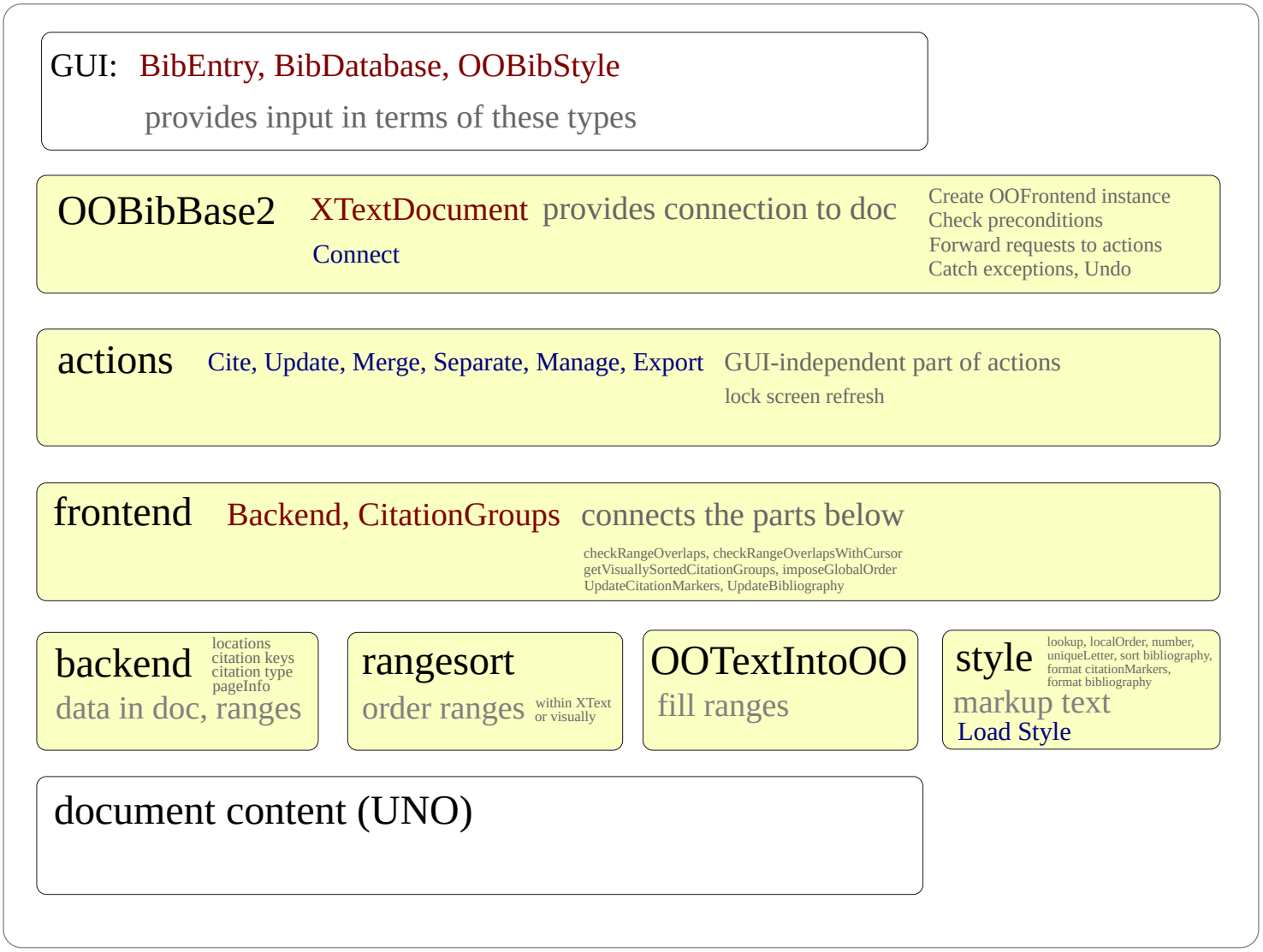
Code reorganization

Why

- Separate backend
- Separate GUI code (dialogs) and logic
- Data is now organized around `Citation`, `CitationGroup` instead of arrays for citation group fields, and arrays of arrays for citation fields.
  Also take `citationKey` as the central data unit, this is what we start with: unresolved `citationKeys` do not stop processing. Although we cannot sort them by author and year, we can still emit a marker that acts as a placeholder and shows the user the problematic key.

Result

Layers



By directories

- `model`
  - `util` : general utilities
    - (`OOPair`, `OOTuple3`) collect two or three objects without creating a new class
    - `OOResult` : while an Optional.empty can comunicate failure, it cannot provide details. `OOResult` allows an arbitrary error object to be provided in case of failure.
    - `OOVoidResult` : for functions returning no result on success, only diagnostics on failure.
    - `OOListUtil`: some utilities working on List
  - `uno` : helpers for various tasks via UNO.
    These are conceptually independent of JabRef code and logic.
  - `ootext` : to separate decisions on the format of references and citation marks from the actual insertion into the document, the earlier method `OOUtil.insertOOFormattedTextAtCurrentLocation` was extended to handle new tags that describe actions earlier done in code.
    - This became `OOTextIntoOO.write`
      - `(change)` Now all output to the document goes through this, not only those from Layout. This allows the citation markers and `jstyle:Title` to use these tags.
        - This allows some backward-compatible extensions to jstyle.
          `(change)` Added some extra keywords, in `{prefix}_MARKUP_BEFORE`, `{prefix}_MARKUP_AFTER` pairs to allow bracketing some parts of citation marks with text and/or open/close tag pairs.
    - `OOFormat` contains helpers to create the appropriate tags
    - `OOText` formalizes the distinction from `String`. I did not change `String` to `OOText` in old code, (in particular in OOStyle).
  - `rangesort` : ordering objects that have an `XTextRange`, optionally with an extra integer to break ties.
    - `RangeSort.partitionAndSortRanges` : since `XTextRangeCompare` can only compare `XTextRange` values in the same `XText`, we partition them accordingly and only sort within each partition.
    - `RangeSortable` (interface), `RangeSortEntry` (implements) :
      When we replace `XTextRange` of citation marks in footnotes with the range of the footnote mark, multiple citation marks may be mapped to the same location. To preserve the order between these, `RangeSortable` allows this order to be indicated by returning appropriate indices from `getIndexInPosition`
    - `RangeSortVisual` : sort in top-to-bottom left-to-right order.
      Needs a functional `XTextViewCursor`.
      Works on `RangeSortable` values.
    - `FunctionalTextViewCursor` : helper to get a functional `XTextViewCursor` (cannot always)

    - `RangeOverlapWithin` : check for overlaps within a set of `XTextRange` values. Probably O(n*log(n)). Used for all-to-all check of protected ranges.
    - `RangeOverlapBetween` : check for overlaps between two sets of `XTextRange` values. Assumes one set is small. O(n*k). Used for checking if the cursor is in a protected range.
  - `backend` : interfaces to be provided by backends.
    May change as new backends may need different APIs.
  - `style` : data structures and interfaces used while going from ordered list of citation groups to formatted citation markers and bibliography. Does not communicate with the document. Too long to fit here, starting a new section.

model/style

At the core,

- we have `Citation` values
  - represented in the document by their `citationKey`
  - each may have a `pageInfo`
- A citation group (`CitationGroup`) has
  - a list of citations (`citationsInStorageOrder`)
  - an identifier `CitationGroupId cgid`
    - this allows to refer to the group
    - also used to associate the group to its citation markers location (outside the style part, in `Backend52`)
  - `OODataModel dataModel` is here, in order to handle old (Jabref5.2) structure where pageInfo belonged to CitationGroup not Citation
  - `referenceMarkNameForLinking` is optional: can be used to crosslink to the citation marker from the bibliography.
- `CitationGroups` represents the collection of citation groups.
  Processing starts with creating a `CitationGroups` instance from the data stored in the document.
- `CitedKey` represents a cited source, with ordered back references (using `CitationPath`) to the corresponding citations.
- `CitedKeys` is just an order-preserving collection of `CitedKeys` that also supports lookup by `citationKey`. While producing citation markers, we also create a corresponding `CitedKeys` instance, and store it in `CitationGroups.bibliography`. This is already sorted, its entries have `uniqueLetter` or `number` assigned, but not converted to markup yet.

Common processing steps:

- We need `globalOrder` for the citation groups (provided externally)
  `CitationGroups.setGlobalOrder()`

- We need to look up each citationKey in the bibliography databases:
  - `CitationGroups.lookupCitations` collects the cited keys, looks up each, then distributes the results to the citations. Uses a temporary `CitedKeys` instance, based on unsorted citations and citation groups.
- `CitationGroups.imposeLocalOrder` fills `localOrder` in each `CitationGroup`
- Now we have order of appearance for the citations (`globalOrder` and `localOrder`).
  We can create a `CitedKeys` instance (`bibliography`) according to this order.
- For citations numbered in order of first appearance we number the sources and distribute the numbers to the corresponding citations.
- For citations numbered in order of bibliography, we sort the bibliography, number, distribute.
- For author-year citations we have to decide on the letters `uniqueLetter` used to distinguish sources. This needs order of first appearance of the sources and recognizing clashing citation markers. This is done in logic, in `OOProcessAuthorYearMarkers.createUniqueLetters()`
- We also mark first appearance of each source (`setIsFirstAppearanceOfSourceInCitations`)

The entry point for this processing is: `OOProcess.produceCitationMarkers`.

It fills

- each `CitationGroup.citationMarker`
- `CitationGroups.bibliography`
  - From bibliography `OOFormatBibliography.formatBibliography()` creates an `OOText` ready to be written to the document.

logic/style

- `StyleLoader` : not changed (knows about default styles) Used by GUI
- `OOPreFormatter` : LaTeX code to unicode and OOText tags. (not changed)
- `OOBibStyle` : is mostly concerned by loading/parsing jstyle files and presenting its pieces to the rest. Originally it also contains code to format numeric and author-year citation markers.
  - Details of their new implementations are in `OOBibStyleGetNumCitationMarker` and `OOBibStyleGetCitationMarker`
  - The new implementations
    - support pageInfo for each citation
    - support unresolved citations
    - instead of `List<Integer>` and (`List<BibEntry>` plus arrays and database) they expect more self-contained entries `List<CitationMarkerNumericEntry>`, `List<CitationMarkerEntry>`.
    - We have distinct methods for `getNormalizedCitationMarker(CitationMarkerNormEntry)` and `getNumCitationMarkerForBibliography(CitationMarkerNumericBibEntry)`.

    - The corresponding interfaces in model/style:
        - `CitationMarkerNumericEntry`
        - `CitationMarkerEntry`
        - `CitationMarkerNumericBibEntry`
        - `CitationMarkerNormEntry`
          describe their expected input entries.
- `OOProcess.produceCitationMarkers` is the main entry point for style application. Calls to specific implementations in `OOProcessCitationKeyMarkers`, `OOProcessNumericMarkers` and `OOProcessAuthorYearMarkers` according to jstyle flags.

logic/backend

Details of encoding and retrieving data stored in a document as well as the citation maker locations. Also contains dataModel-dependent code (which could probably be moved out once the datamodel is settled).

Creating and finding the bibliography (providing a cursor to write at) should be here too. These are currently in `UpdateBibliography`

logic/frontend

- `OOFrontend` : has a `Backend` and `CitationGroups`
    - Its constructor creates a backend, reads data from the document and creates a CitationGroups instance.
    - provides functionality that requires both access to the document and the CitationGroups instance
- `RangeForOverlapCheck` used in `OOFrontend`
- `UpdateBibliography` : Create, find and update the bibliography in the document using output from `produceCitationMarkers()`
- `UpdateCitationMarkers` create `CitationGroup`, update citation markers using output from `produceCitationMarkers()`

logic/action

GUI-independent part of implementations of GUI actions.

gui

- `OOError` : common error messages and dialog titles
    - adds `title` to `Jabrefexception`
    - converts from some common exception types using type-specific message
    - contains some dialog messages that do not correspond to exceptions

- `OOBibBase2` : most activity was moved out from here to parts discussed above.
  - connecting / selecting a document moved to `OOBibBaseConnect`
  - the rest connects higher parts of the GUI to actions in logic
    - does argument and precondition checking
    - catches all exceptions
    - shows error and warning dialogs
    - adds `enterUndoContext`, `leaveUndoContext` around action code

Alternatives to using OOResult and OOVoidResult in OOBibBase

(Talk about ADRs prompted me to think about alternatives to what I used.)

Situation:

- some tests return no data, only report problems
- we may need to get some resources that might not be available (for example: connection to a document, a functional textview cursor)
- some test depend on these resources

One strategy could be to use a single try-catch around the whole body, then showing a message based on the type of exceptions thrown.

[base case]

```
try {
   A a = f();
   B b = g(a);
   realAction(a,b);
} catch (FirstExceptionType ex) {
   showDialog( title, messageForFirstExceptionType(ex) );
} catch (SecondExceptionType ex) {
   showDialog( title, messageForSecondExceptionType(ex) );
} catch (Exception ex) {
   showDialog( title, messageForOtherExceptions(ex) );
}
```

This our base case.

It is not clear from the code, nor within the catch branches (unless we start looking into stack traces) which call (`f()`, `g(a)` or `realAction(a,b)`) resulted in the exception. This limits the specificity of the message and makes it hard to think about the “why” can we get this exception here?

Catch around each call?

A more detailed strategy would be to try-catch around each call.
In case we need a result from the call, this means either increasingly indented code (try-in-try).

```
try {
   A a = f();
   try {
      B b = g(a);
      try {
         realAction(ab);
      } catch (...){
         showDialog();
      }
   } catch (G ex) {
     showDialog(title, ex); // title describes which GUI action we are in
   }
} catch (F ex) {
   // could an F be thrown in g?
    showDialog( title, ex );
}
```

or (declare and fill later)

```
A a = null;
try {
   a = f();
} catch (F ex) {
    showDialog(title, ex);
    return;
}
B b = null;
try {
   b = g(a);
} catch (G ex) {
    showDialog(title, ex);
    return;
}
try {
    realAction(ab);
} catch (...){
    showDialog();
}
```

In either case, the code becomes littered with exception handling code.

Catch in wrappers?

We might push the try-catch into its own function.
If the wrapper is called multiple times, this may reduce duplication of the catch-and-assign-message part.

We can show an error dialog here: `title` carries some information from the caller, the exeption caught brings some from below.

We still need to notify the action handler (the caller) about failure. Since we have shown the dialog, we do not need to provide a message.

Notify caller with `Optional` result

With `Optional` we get something like this:

[DIALOG IN WRAP, RETURN OPTIONAL]

```
Optional<A> wrap_f(String title) {
    try {
      return Optional.of(f());
    } catch (F ex) {
      showDialog(title, ex);
      return Optional.empty();
    }
}

Optional<B> wrap_g(String title, A a) {
    try {
       return Optional.of(g(a));
    } catch (G ex) {
      showDialog(title, ex);
      return Optional.empty();
    }
}
```

and use it like this:

```
Optional<A> a = wrap_f(title);
if (a.isEmpty()) { return; }

Optional<B> b = wrap_g(title, a.get());
if (b.isEmpty()) { return; }
```

```
try {
   realAction(a.get(), b.get());
} catch (...) {
}
```

This looks fairly regular.

If `g` did not need `a`, we could simplify to

```
Optional<A> a = wrap_f(title);
Optional<B> b = wrap_g(title);
if (a.isEmpty() || b.isEmpty()) { return; }

try {
   realAction(a.get(), b.get());
} catch (...) {
}
```

Notify caller with `Result` result

With `Result` we get something like this:

[DIALOG IN WRAP, RETURN OORESULT]

```
OOResult<A, OOError> wrap_f() {
    try {
      return OOResult.ok(f());
    } catch (F ex) {
      return OOResult.error(OOError.from(ex));
    } catch (F2 ex) {
      String message = "...";
      return OOResult.error(new OOError(message, ex)); // [1]
    }
}
// [1] : this OOError constructor (explicit message but no title) is missing

Optional<B, OOError> wrap_g(A a) {
    try {
       return OOResult.ok(g(a));
    } catch (G ex) {
      return OOResult.error(OOError.from(ex));
    }
}
```

and use it like this:

```
OOResult<A, OOError> a = wrap_f();
if (testDialog(title, a)) { // [1]
    return;
}

// [1] needs boolean testDialog(String title, OOResultLike<OOError>... a);
//     where OOResultLike<T> is an interface with `OOVoidResult<T> asVoidResult()`
//     and is implemented by OOResult and OOVoidResult

OOResult<B, OOError> b = wrap_g(a.get());
if (testDialog(title, b)) { return; } // (checkstyle makes this 3 lines)

try {
   realAction(a.get(), b.get());
} catch (...) {
}
```

If `g` did not need `a`, we could simplify to

```
Optional<A> a = wrap_f();
Optional<B> b = wrap_g();
if (testDialog(title, a, b)) { // a single dialog can show both messages
  return;
}

try {
   realAction(a.get(), b.get());
} catch (...) {
}
```

Notify caller by throwing an exception

Or we can throw an exception to notify the caller.

To simplify code in the caller, I assume we are using an exception type not used elsewhere, but shared by all precondition checks.

[DIALOG IN WRAP, PRECONDITIONEXCEPTION]

```
A wrap_f(String title) throws PreconditionException {
    try {
       return f();
    } catch (F ex) {
      showDialog(title, ex)
```

```
      throw new PreconditionException();
    }
}

B wrap_g(String title, A a) throws PreconditionException {
    try {
       return g(a);
    } catch (G ex) {
      showDialog(title, ex);
      throw new PreconditionException();
    }
}
```

use

```
try {
   A a = wrap_f(title);
   B b = wrap_g(title, a);
   try {
     realAction(a, b);
   } catch (...) {
     showDialog(...)
   }
} catch( PreconditionException ) {
   // Only precondition checks get us here.
   return;
}
```

or (since PreconditionException is not thrown from realAction)

```
try {
   A a = wrap_f(title);
   B b = wrap_g(title, a);
   realAction(a, b);
} catch (...) {
   // Only realAction gets us here
   showDialog(...)
} catch( PreconditionException ) {
   // Only precondition checks get us here.
   return;
}
```

or (separate try-catch for preconditions and realAction)

```
A a = null;
B b = null;
try {
    a = wrap_f(title);
    b = wrap_g(title, a);
} catch( PreconditionException ) {
     return;
}
try {
     realAction(a, b);
} catch (...) {
}
```

or to reduce passing around the title part:

[PRECONDITIONEXCEPTION, DIALOG IN CATCH]

```
A wrap_f() throws PreconditionException {
    try {
       return f();
    } catch (F ex) {
      throw new PreconditionException(message, ex);
    }
}

B wrap_g(A a) throws PreconditionException {
    try {
       return g(a);
    } catch (G ex) {
      throw new PreconditionException(message, ex);
    }
}
```

use

```
try {
   A a = wrap_f();
   B b = wrap_g(a);
   try {
     realAction(a, b);
   } catch (...) {
      showDialog(...);
   }
```

```
} catch(PreconditionException ex) {
   showDialog(title, ex.message );
   return;
}
```

or

```
try {
   A a = wrap_f();
   B b = wrap_g(a);
   realAction(a, b);
} catch (...) {
   showDialog(...);
} catch(PreconditionException ex) {
   showDialog(title, ex.message );
   return;
}
```

Push associating the message further down

As the developers guide suggest, we could “Catch and wrap all API exceptions” and rethrow them as a `JabRefException` or some exception derived from it. In this case the try-catch part goes even further down, and in principle we could just

```
try {
   A a = f();
   B b = g(a);
   realAction(a, b);
} catch(JabRefException ex) {
   showDialog(title, ex.message );
   return;
}
```

Constraints:

- conversion to `JabRefException` cannot be done in `model` (since JabRefException is in `logic`)
- `JabRefException` expects a localized message. Or we need to remember which `JabRefException` instances are localized and which need to be caught for localizing the message.
- At the bottom we usually have very little information on higher level contexts: at a failure like `NoSuchProperty` we cannot tell which set of properties did we look in and why.
  For messages originating too deeply, we might want to override or extend the message anyway.

- for each exeption we might want to handle programmatically, we need a variant based on `JabRefException`

So we might end up:

```
try {
   A a = f();
   B b = g(a);
   realAction(a, b);
} catch(FDerivedFromJabRefException ex) {
   showDialog(title, messageForF );
} catch(GDerivedFromJabRefException ex) {
   showDialog(title, messageForG );
} catch(JabRefException ex) {
   showDialog(title, ex.message );
} catch(Exception ex) { // [1]
   showDialog(title, ex.message, ex );
   // [1] does "never catch Exception or Throwable" apply at this point?
   //     Probably should not: we are promising not to throw.
}
```

which looks very similar to the original version.

This again loses the information: can `GDerivedFromJabRefException` come from `realAction` or `f` or not? This is because we have pushed down the last catch/throw indefinitely (eliminating `wrap_f`) into a depth, where we cannot necessarily assign an appropriate message.

To a lesser extent this also happens in `wrap_f`: it only knows about the action that called it what we provide (`title` or nothing). It knows the precondition it checks: probably an optimal location to assign a message.

Summary: going from top to bottom, we move to increasingly more local context, our knowledge shifts towards the “in which part of the code did we have a problem” and away from the high level (“which action”).

One natural point to meet information from these to levels is the top level of action handlers. For precondition checking code a wrapper around code elsewhere may be considered. Using such wrappers may reduce duplication if called in multiple actions.

We still have to signal failure to the action handler: the options considered above were using an `Optional` and throwing an exception with the appropriate message.

The more promising variants were

- **[dialog in wrap, return Optional]**
  `Optional<A> wrap_f(String title)` (showDialog inside)

- pro: explicit return in caller
- con: explicit return in caller (boilerplate)
- con: passing in the title is repeated
  - would be ‘pro’ if we wanted title to vary within an action

- **[PreconditionException, dialog in catch]**
  `A wrap_f() throws PreconditionException`
  (with `showDialog` under `catch(PreconditionException ex)`)
  - con: hidden control flow
  - pro: no repeated `if(){return}` boilerplate
  - pro: title used only once

[using OOResult]

```
final String title = "Could not insert citation";

OOResult<XTextDocument, OOError> odoc = getXTextDocument();
if (testDialog(title,
               odoc,
               styleIsRequired(style),
               selectedBibEntryIsRequired(entries, OOError::noEntriesSelectedForCitation))) {
    return;
}
XTextDocument doc = odoc.get();

OOResult<OOFrontend, OOError> ofr = getFrontend(doc);
if (testDialog(title, ofr)) {
    return;
}
OOFrontend fr = ofr.get();

OOResult<XTextCursor, OOError> cursor = getUserCursorForTextInsertion(doc);
if (testDialog(title, cursor)) {
    return;
}
...
```

[using PreconditionException, dialog in catch]

```
final String title = "Could not insert citation";

try {
```

```
   XTextDocument doc = getXTextDocument();
   styleIsRequired(style);
   selectedBibEntryIsRequired(entries, OOError::noEntriesSelectedForCitation);
   OOFrontend fr = getFrontend(doc);
   XTextCursor cursor = getUserCursorForTextInsertion(doc);
   ...
} catch (PreconditionException ex) {
    showDialog(title, ex);
} catch (...) {
}
```

I would suggest using the latter,

- probably using `OOError` for `PreconditionException`
  - In this case `OOError` being in `gui` becomes an asset: we can be sure code in `logic` cannot throw it.
- We lose the capability to collect mmessages in a single dialog (we stop processing at the first problem).
- The division between precondition checking (only throws PreconditionException) and `realAction`becomes invisible in the action code.

About OOError, OOResult, and OOVoidResult

Context

Relieve GUI panel code

On the question of where should we catch exceptions in relation to GUI code it was suggested (Jonatan Asketorp here, “most of them (all?) should be handled latest in the ViewModel.”) that catching them early could help simplifying the higher levels.

Same messages in different contexts

Some types of exceptions are caught in *different GUI actions*, often resulting in basically the same error dialog, possibly only differing in the indicated context (which GUI action).

Problems found during *precondition checking* (for example: do we have a connection to a document) and error conditions (for example: lost connection to a document during an action) can overlap.

OOBibBase as a precondition and exception handling layer

Since most of the code originally in `OOBibBase` was moved to `logic` and almost all GUI actions go through `OOBibBase`, it seemed a good location to collect precondition checking and exception handling code.

Note: some of the precondition checking still needs to stay in `OpenOfficePanel`: for example to provide a list of selected `BibEntry` instances, it needs to go through some steps from `frame.getCurrentLibraryTab()` to `(!entries.isEmpty() && checkThatEntriesHaveKeys(entries))`

To avoid `OOBibBase` depending on the higher level `OpenOfficePanel` message texts needed in `OOBibBase` were moved from `OpenOfficePanel` to `OOError`. (Others stayed, but could be moved if that seems worthwile)

OOError

- `OOError` is a collection of data used in error dialogs.
  - It is a `JabRefException` with an added field: `localizedTitle`
  - It can store: a dialog title, a localized message (optionally a non-localized message as well) and a `Throwable`

- I used it in `OOBibBase` as a unified format for errors to be shown in an error dialog.
- Static constructors in `OOError` provide uniform translation from some exception types to `OOError` with the corresponding localized messages:
  `public static OOError from(SomeException ex)`
  There is also `public static OOError fromMisc(Exception ex)` for exception types not handled individually. (It has a different name, to avoid ambiguity)
- Another set of constructors provide messages for some preconditions.
  For example `public static OOError noDataBaseIsOpenForCiting()`

Some questions:

- Should we use static data instead of static methods for the precondition-related messages?
  - pro: why create a new instance for each error?
  - con: `OOError.setTitle()` currently just sets `this.localizedTitle` and returns `this`. For static instances this would modify a shared resource unless we create a new copy in `setTitle`. However `setTitle` can be called repeatedly on the same object: as we bubble up, we can be more specific about the context.
- Should we remove title from `OOError`?
  - pro: we almost always override its original value
  - con: may need to duplicate the title in different files (preconditions for an action in OpenOfficePanel and in OOBibBase)
- Should we include `OOError.showErrorDialog` ?
  - pro: since it was intended *for* error dialogs, it is nice to provide this.
  - con: the reference to `DialogService` forces it to `gui`, thus it cannot be used in `logic` or `model`
- Should we use `JabRefException` as base?
  - pro: `JabRefException` is mentioned as the standard form of errors in the developers guide. All Exceptions we throw should be or extend JabRefException
  - against: `JabRefException` is in `logic` cannot be used in model.
    (Could this be resolved by moving `JabRefException` to `model`?)

OOResult

During precondition checking

1. some tests return no data, only report problems
2. we may need to get some resources that might not be available (for example: connection to a document, a functional textview cursor)
3. some test depend on these resources

While concentrating on these and on “do not throw exceptions here” … using a Result type as a return value from precondition checking code seemed a good fit:

- Instead of throwing an exception, we can return some data describing the problem.
- Conceptually it is a data structure that either holds the result (of a computation) or and error value.
- It can be considered as an extended `Optional`, that can provide details on “why empty”?
- It can be considered as an alternative to throwing an exception: we return an `error` instead.
- Methods throwing checked exceptions cannot be used with for example `List.map`. Methods returning a Result could.
- `Result` shares the problem (with any other solutions) that in a function several types of errors may occur, but we can only return a single error type. Java solves this using checked exceptions being all descendants of Exception. (Also adds try/catch/catch to select cases based on the exceptions type, and some checking against forgotten cases of checked exception types)

In `OOBibBase` I used `OOError` as the unified error type: it can store error messages and wrap exceptions. It contains everything we need for an error dialog. On the other hand it does not support programmatic dissection.

Implementation

Unlike `Optional` and `List`, `Result` (in the sense used here) did not get into java standard libraries. There are some implementations of this idea for java on the net:

- bgerstle/result-java
- MrKloan/result-type
- david-bakin
- vavr-try

Generics allow an implementation built around

```
class OOResult<R, E> {
    private final Optional<R> result;
    private final Optional<E> error;
}
```

with an assumption that at any time exactly one of `result` and `error` is present.

> `class X<R,E> { boolean isOK; Object data; }` expresses this assumption more directly, (but omits the relation between the type parameters `<R,E>` and the type in `data`)

- Since `OOResult` encodes the state `isOK` in `result.isPresent()` (and equivalently in `errror.isEmpty()`), we cannot allow construction of instances where both values are `isEmpty`. In particular, `OOResult.ok(null)` and `OOResult.error(null)` are not allowed: it would make the state `isOK` ambiguous.
  It would also break the similarity to `Optional` to allow both `isEmpty` and `isOK` to be true.

- Not allowing null, has a consequence on `OOResult<Void,E>`
  According to baeldung.com/java-void-type, the only possible value for `Void` is `null` which we excluded.

  `OOResult<Void,E>.ok(null)` would look strange: in this case we need `ok()` without arguments.

To solve this problem, I introduced

```
class OOVoidResult<E> {
    private final Optional<E> error;
    ...
}
```

with methods on the error side similar to those in `OOError<R,E>`, and `OOVoidResult.ok()` to construct the success case with no data.

The relation between `Optional<E>` and `OOVoidResult<E>`

- Both `Optional` and `OOVoidResult` can store 0 or 1 values, in this respect they are equivalent
  - Actually, `OOVoidResult` is just a wrapper around an `Optional`
- In terms of communication to human readers when used, their connotation in respect to success and failure is the opposite:
  - `Optional.empty()` normally suggests failure, `OOVoidResult.ok()` mean success.
  - `Optional.of(something)` probably means success, `OOVoidResult.error(something)` indicates failure.
  - `OOVoidResult` is “the other half” (the failure branch) of `OOResult`
    - its content is accessed through `getError`, `mapError`, `ifError`, not `get`, `map`, `ifPresent`

`OOVoidResult` allows

- a clear distinction between success and failure when calls to “get” something that might not be available (`Optional`) and calls to precondition checking where we can only get reasons for failure (`OOVoidResult`) appear together.
  Using `Optional` for both is possible, but is more error-prone.
- it also allows using uniform verbs (`isError`, `getError`, `ifError`, return `OO{Void}Result.error`) for “we have a problem” when
  - checking preconditions (`OOVoidResult`) is mixed with
  - “I need an X” orelse “we have a problem” (`OOResult`)
- at a functions head:
  - `OOVoidResult<String> function()` says: no result, but may get an error message
  - `Optional<String> function()` says: a `String` result or nothing.

Summary: technically could use `Optional` for both situation, but it would be less precise, leaving more room for confusion and bugs. `OOVoidResult` forces use of `getError` instead of `get`, and `isError` or `isOk` instead of `isPresent` or `isEmpty`.

What does OOResult buy us?

The promise of `Result` is that we can avoid throwing exceptions and return errors instead. This allows the caller to handle these latter as data, for example may summarize / collect them for example into a single message dialog.

Handling the result needs some code in the caller. If we only needed checks that return only errors (not results), the code could look like this (with possibly more tests listed):

```
OOResult<XTextDocument, OOError> odoc = getXTextDocument();
if (testDialog(title,
               odoc,
               styleIsRequired(style),
               selectedBibEntryIsRequired(entries, OOError::noEntriesSelectedForCitation))) {
    return;
}
```

with a reasonably small footstep.

Dependencies of tests on earlier results complicates this: now we repeat the

```
if (testDialog(title,
               ...)) {
    return;
}
```

part several times.

Order of appearance of citation groups

The order of appearance of citations is decided on two levels:

1. their order within each citation group (`localOrder`), and
2. the order of the citation groups that appear as citation markers in the text (`globalOrder`).

This page is about the latter: how to decide the order of appearance (numbering sequence) of a set of citation markers?

Conceptually

In a continuous text it is easy: take the textual order of citation markers.

In the presence of figures, tables, footnotes/endnotes possibly far from the location they are referred to in the text or wrapped around with text it becomes less obvious what is the correct order.

Examples:

- References in footnotes: are they *after* the page content, or number them as if they appeared at the footnote mark? (JabRef does the latter)
- A figure with references in its caption. Text may flow on either or both sides.
  Where should we insert these in the sequence?
- In a two-column layout, a text frame or figure mostly, but not fully in the second column: shall we consider it part of the second column?

Technically

In LibreOffice, a document has a main text that supports the XText interface.
This allows several types of XTextContent to be inserted.

- Some of these allow text inside with further insertions.

Anchors

- Many, but not all XTextContent types support getting a “technical” insertion point or text range through getAnchor.
- In Libreoffice positioning both a frame and its anchor seems hard: moving the frame tends to also move the anchor.
- Consequence: producing an order of appearance for the citation groups based solely on `getAnchor` calls may be impossible.

- Allowing or requiring the user to insert “logical anchors” for frames and other “floating” parts might help to alleviate these problems.

Sorting within a `Text`

The text ranges occupied by the citation markers support the XTextRange interface.

- These provide access to the XText they are contained in.
- The Text service may support (optional) the XTextRangeCompare interface, that allows two XTextRange values to be compared if both belong to this `Text`

Visual ordering

- The cursor used by the user is available as an XTextViewCursor
- If we can get it and can set its position in the document to each XTextRange to be sorted, and ask its getPosition to provide coordinates “relative to the top left position of the first page of the document.”, then we can sort by these coordinates in top-to-bottom left-to-right order.
- Note: in some cases, for example when the cursor is in a comment (as in `Libreoffice: [menu:Insert]/[Comment]`), the XTextViewCursor is not available (I know of no way to get it).
- In some other cases, for example when an image is selected, the XTextViewCursor we normally receive is not ‘functional’: we cannot position it for getting coordinates for the citation marks. The FunctionalTextViewCursor class can solve this case by accessing and manipulating the cursor through XSelectionSupplier

Consequences of getting these visual coordinates and using them to order the citation markers

- allows uniform handling of the markers. Works in footnotes, tables, frames (apparently anywhere)
- requires moving the user visible cursor to each position and with screen refresh enabled. `(problem)` This results in some user-visible flashing and scrolling around in the document view.
- The expression “relative to the top left position of the first page of the document” is understood literally, “as on the screen”.
  `(problem)` Showing pages side by side or using a two-column layout will result in markers in the top half of the second column or page to be sorted before those on the bottom of the first column of the first page.

JabRef

Jabref uses the following steps for sorting citation markers (providing `globalOrder`):

1. the textranges of citation marks in footnotes are replaced by the textranges of the footnote marks.
2. get the positions (coordinates) of these marks

3. sort in top-to-bottom left-to-right order

`(problem)` In JabRef5.2 the positions of citation marks within the same footnote become indistinguishable, thus their order after sorting may differ from their order in the footnote text. This caused problems for

1. numbering order
   `(solved)` by keeping track of the order-in-footnote of citation markers during sorting using getIndexInPosition)
2. `click:Merge`: It examines *consecutive* pairs of citation groups if they can be merged. Wrong order may result in not discovering some mergeable pairs or attempting to merge in wrong order.
   `(solved)` by not using visual order, only XTextRangeCompare-based order within each XText here)

Overview

This is a partial overview of the OpenOffice/LibreOffice panel and the code behind.

- To access the panel: `JabRef:/[menu:View]/[OpenOffice/LibreOffice]`
- The user documentation is at https://docs.jabref.org/cite/openofficeintegration

I am going to refer to OpenOffice Writer and LibreOffice Writer as LibreOffice or LO: their UNO APIs are still mostly identical, but I only tested with LibreOffice and differences do exist.

Subject

- What is stored in a document, how.
- Generating citation markers and bibliography
  - (excluding the bibliography entries, which is delegated to the layout module)

The purpose of the panel

- Allow the user to insert **citations** in a LibreOffice writer document.
- Automatically format these according to some prescribed style as **citation markers**.
- Generate a **bibliography**, also formatted according to the style.
  - The bibliography consists of a title (e.g. “References”) and a sorted list of formatted bibliography entries, possibly prefixed with a marker (e.g. “[1]”)
- It also allows some related activities: connect to a document, select a style, group (“Merge”) the citations for nicer output, ungroup (“Separate”) them to move or delete them individually, edit (“Manage”) their page-info parts, and collect the database entries of cited sources to a new database.

Citation types

Citations (actually citation groups, see below) have three types depending on how the citation marker is intended to appear in the text:

- **Parenthesized**: “(Smith, 2000)”
- **In-text**: “Smith (2000)”
- **Invisible**: no visible citation mark.
  - An invisible citation mark lets the user to use any form for the citation by taking control (and responsibility) back from the style.

    - Like the other two citation types, they have a location in the document.
    - In the bibliography these behave as the other two citation types.
    - In LibreOffice (`LibreOffice:[Ctrl-F8]` or `LibreOffice:[menu:View]/[Field Shadings]`) shows reference marks with gray background. Invisible citation marks appear as a thin gray rectangle.
- These citation types correspond to `\citep{Smith2000}`, `\citet{Smith2000}` in natbib and `\nocite{Smith2000}`. I will use `\citep`, `\citet` and `\citen` in "LaTeX pseudocode" below.

PageInfo

The citations can be augmented with a string detailing which part of a document is cited, for example "page 11" or "chapter 2".

Sample citation markers (with LaTeX pseudocode):

- `\citep[page 11]{Smith2000}` "(Smith, 2000; page 11)"
- `\citet[page 11]{Smith2000}` "Smith (2000; page 11)"
- `\citen[page 11]{Smith2000}` ""
- This string is referred to as `pageInfo` in the code.
- In the GUI the labels "Cite special", "Extra information (e.g. page number)" are used.

Citation groups

Citations can be grouped.

A group of parenthesized citations share the parentheses around, like this:
"(Smith, 2000; Jones 2001)".

- Examples with pseudocode:
    - `\citep{Smith2000,Jones2001}` "(Smith, 2000; Jones 2001)"
    - `\citet{Smith2000,Jones2001}` "Smith (2000); Jones (2001)"
    - `\citen{Smith2000,Jones2001}` ""

From the user's point of view, citation groups can be created by

1. Selecting multiple entries in a bibliography database, then

    - `[click:Cite]` or
    - `[click:Cite in-text]` or
    - `[click:Cite special]` or
    - `[click:Insert empty citation]` in the panel.

    This method allows any of the citation types to be used.

2 `[click:Merge citations]` finds all sets of consecutive citations in the text and replaces each with a group.

- `(change)` The new code only merges consecutive parenthesized citations.
  - This is inconsistent with the solution used in `[click:Cite]`
  - My impression is that
    - groups of in-text or invisible citations are probably not useful
    - mixed groups are even less. However, with a numbered style there is no visual difference between parenthesized and in-text citations, the user may be left wondering why did merge not work.
      - One way out could be to merge as a “parenthesized” group. But then users switching between styles get a surprise, we have unexpectedly overridden their choice.
      - I would prefer a visible log-like warning that does not require a click to close and lets me see multiple warnings. Could the main window have such an area at the bottom?
- Starting with JabRef 5.3 there is also `[click:Separate citations]` that breaks all groups to single citations.
  - This allows
    - deleting individual citations
    - moving individual citations around (between citation groups)
    - (copy does not work)
    - (Moving a citation within a group has no effect on the final output due to sorting of citations within groups. See Sorting within a citation group)

In order to manage single citations and groups uniformly, we consider each citation in the document to belong to a citation group, even if it means a group containing a single citation.

Citation groups correspond to citation markers in the document. The latter is empty for invisible citation groups. When creating the citation markers, the citations in the group are processed together.

Citation styles

The details of how to format the bibliography and the citation markers are described in a text file.

- These normally use `.jstyle` extension, and I will refer to them as jstyle files.
- See the User documentation for details.
- I will refer to keywords in jstyle files as `jstyle:keyword` below.

Four major types citation of styles can be described by a jstyle.

- (1) `jstyle:BibTeXKeyCitations`
  - The citation markers show the citationKey.
  - It is not fully implemented
    - does not produce markers before the bibliography entries
    - does not show pageInfo
  - It is not advertised in the User documentation.
  - Its intended purpose may be
    - (likely) a proper style, with “[Smith2000]” style citation markers
    - (possibly) a style for “draft mode” that
      - can avoid lookup of citation markers in the database when only the citation markers are updated
      - can produce unique citation markers trivially (only needs local information)
      - makes the citation keys visible to the user
      - can work without knowing the order of appearance of citation groups
      - In case we expect to handle larger documents, a “draft mode” minimizing work during `[click:Cite]` may be useful.
- There are two types of numbered (`jstyle:IsNumberEntries`) citation styles:
  - (2) Citations numbered in order of first appearance (`jstyle:IsSortByPosition`)
  - (3) Citations numbered according to their order in the sorted bibliography
- (4) Author-year styles

Sorting

Sorting the bibliography

The bibliography is sorted in (author, year, title) order

- except for `jstyle:IsSortByPosition`, that uses the order of first appearance of the cited sources.

Ordering the citations

The order of appearance of citations (as considered during numbering and adding letters after the year to ensure that citation markers uniquely identify sources in the bibliography) is decided on two levels.

1. Their order within each citation group (`localOrder`), and
2. the order of the citation groups (citation markers) in the text (`globalOrder`).

SORTING WITHIN A CITATION GROUP (`localOrder`)

The order of citations within a citation group is controlled by `jstyle:MultiCiteChronological`.

- `true` asks for (year, author, title) ordering,

- `false` for (author, year, title).
- (There is no option for “in the order provided by the user”).

For author-year citation styles this ordering is used directly.

- The (author, year, title) order promotes discovering citations sharing authors and year and emitting them in a shorter form. For example as “(Smith 2000a,b)”.

For numbered styles, the citations within a group are sorted again during generation of the citation marker, now by the numbers themselves. The result of this sorting is not saved, only affects the citation marker.

- Series of consecutive number are replaced with ranges: for example “[1-5; 11]”

ORDER OF THE CITATION GROUPS (`globalOrder`)

The location of each citation group in the document is provided by the user. In a text with no insets, footnotes, figures etc. this directly provides the order. In the presence of these, it becomes more complicated, see Order of appearance of citation groups.

ORDER OF THE CITATIONS

- `globalOrder` and `localOrder` together provide the order of appearance of citations
- This also provides the order of first appearance of the cited sources.

  First appearance order of sources is used

  - in `jstyle:IsSortByPosition` numbered styles
  - in author-year styles: first appearance of “Smith200a” should precede that of “Smith200b”.
    To achieve this, the sources get the letters according the order of their first appearance.
    - This seems to contradict the statement “The bibliography is sorted in (author, year, title) order” above.
      It does not. As of JabRef 5.3 both are true.
      Consequence: in the references Smith2000b may precede Smith2000a. (reported)
  - Some author-year citation styles prescribe a higher threshold on the number of authors for switching to “FirstAuthor et al.” form (`jstyle:MaxAuthors`) at the first citation of a source (`jstyle:MaxAuthorsFirst`)

What is stored in a document (JabRef5.2)

- Each group of citations has a reference mark.

  (Reference marks are shown in LibreOffice in Navigator, under “References”.
  To show the Navigator: `LibreOffice:[menu:View]/[Navigator]` or `LibreOffice:[key:F5]`)

  Its purposes:

1. The text range of the reference mark tells where to write or update the citation mark.
2. The name of the reference mark
   - Lets us select only those reference marks that belong to us
   - Encodes the citation type
   - Contains the list of citation keys that belong to this group
   - It may contain an extra number, to make the name unique in the document
   - Format: `"JR_cite{number}_{type}_{citationKeys}"`, where
     - `{number}` is either empty or an unsigned integer (it can be zero) to make the name unique
     - `{type}` is 1, 2, or 3 for parenthesized, in-text and invisible
     - `{citationKeys}` contains the comma-separated list of citation keys
     - Examples:
       - `JR_cite_1_Smith2000` (empty number part, parenthesized, single citation)
       - `JR_cite0_2_Smith2000,Jones2001` (number part is 0, in-text, two citations)
       - `JR_cite1_3_Smith2000,Jones2001` (number part is 1, invisible, two citations)

- Each group of citations may have an associated pageInfo.
  - In LibreOffice, these can be found at `LibreOffice:/[menu:File]/[Properties]/[Custom Properties]`
  - The property names are identical to the name of the reference mark corresponding to the citation group.
  - JabRef 5.2 never cleans up these, they are left around.
    `(problem)` New citations may “pick up” these unexpectedly.
- The bibliography, if not found, is created at the end of the document.
  - The location and extent of the bibliography is the content of the Section named `"JR_bib"`. (In LibreOffice Sections are listed in the Navigator panel, under “Sections”)
  - JabRef 5.2 also creates a bookmark named `"JR_bib_end"`, but does not use it. During bibliography update it attempts to create it again without removing the old bookmark. The result is a new bookmark, with a number appended to its name (by LibreOffice, to ensure unique names of bookmarks).
    - Correction in new code: remove the old before creating the new.

How does it interact with the document?

- “stateless”
  JabRef is only loosely coupled to the document.
  Between two GUI actions it does not receive any information from LibreOffice.
  It cannot distinguish between the user changing a single character in the document or rewriting everything.
- Access data

- During a `[click:cite]` or `[click:Update]` we need the reference mark names.
  - Get all reference mark names
  - Filter (only ours)
  - Parse: gives citation type (for the group), citation keys
- Access/store pageInfo: based on reference mark name and property name being equal
- Creating a citation group: (`[click:cite]`)
  - Creates a reference mark at the cursor, with a name as described above.

- Update (refreshing citation markers and bibliography):
  - citation markers: the content of the reference mark
  - bibliography: the content of the Section (in LibreOffice sense) named `"JR_bib"`.

Problems

pageInfo should belong to citations, not citation groups

- Creating `[click:Separate]` revealed
  a `(problem)`: pageInfo strings are conceptually associated with citations, but the implementation associates them to citation groups.
  The number of available pageInfo slots changes during `[click:Merge]` and `[click:Separate]` while the number of citations remains fixed.
  - The proposed solution was to change the association.
    - Not only reference marks (citation groups) need unique identifiers, but also citations.
      Possible encoding for reference mark names:
      `JR_cite{type}_{number1}_{citationKey1},{number2}_{citationKey2}`
      where `{type}` encodes the citation type (for the group), `{citationKey1}` is made unique by choosing an appropriate number for `{number1}`
      This would allow `JR_cite_{number1}_{citationKey1}` to be used as a property name for storing the pageInfo.

      Changes required to
      - reference mark search, name generation and parsing
      - name generation and parsing for properties storing pageInfo values
      - in-memory representation
        - JabRef 5.2 does not collect pageInfo values, accesses only when needed.
          So it would be change to code accessing them.
        - The proposed representation does collect, to allow separation of getting from the document and processing
      - insertion of pageInfo into citation markers: JabRef 5.2 injects a single pageInfo before the closing parenthesis, now we need to handle several values
      - `[click:Manage citations]` should work on citations, not citation groups.

Backend

The choice of how do we represent the data and the citation marks in the document has consequences on usability.

Reference marks have some features that make it easy to mess up citations in a document

- They are **not visible** by default, the user is not aware of their boundaries (`LO:[key:Ctrl-F8]`, `LO:[View]/[Field shadings]` helps)
- They are **not atomic**:
  - the user can edit the content. This will be lost on `[click:Update]`
    If an `As character` or `To character` anchor is inserted, the corresponding frame or footnote is deleted.
  - by pressing Enter within, the user can break a reference mark into two parts.
    The second part is now outside the reference mark: `[click:Update]` will leave it as is, and replace the first part with the full text for the citation mark.
  - If the space separating to citation marks is deleted, the user cannot reliably type between the marks.
    The text typed usually becomes part of one of the marks. No visual clue as to which one.
    Note: `[click:Merge]` then `[click:Separate]` adds a single space between. The user can position the cursor before or after it. In either case the cursor is on a boundary: it is not clear if it is in or out of a reference mark.
    Special case: a reference mark at the start or end of a paragraph: the cursor is usually considered to be within at the corresponding edge.
- (good) They can be moved (Ctrl-X,Ctrl-V)
- They cannot be copied. (Ctrl-C, Ctrl-V) copies the text without the reference mark.
- Reference marks are lost if the document is saved as docx.
- I know of no way to insert text into an empty text range denoted by a reference mark
  - JabRef 5.3 recreates the reference mark (using insertReferenceMark) here
  - `(change)` I preferred to (try to) avoid this: NamedRangeReferenceMark.nrGetFillCursor returns a cursor between two invisible spaces, to provide the caller a location it can safely write some text. NamedRangeReferenceMark.nrCleanFillCursor removes these invisible spaces unless the content would become empty or a single character. By keeping the content at least two characters, we avoid the ambiguity at the edges: a cursor positioned between two characters inside is always within the reference mark. (At the edges it may or may not be inside.)
- `(change)` `[click:Cite]` at reference mark edges: safeInsertSpacesBetweenReferenceMarks ensures the we are not inside, by starting two new paragraphs, inserting two spaces between them, then removing the new paragraph marks.
- `(change)` guiActionInsertEntry checks if the cursor is in a citation mark or the bibliography.
- `(change)` `[click:Update]` does an exhaustive check for overlaps between protected ranges (citation marks and bibliography). This can become slow if there are many citations.

It would be nice if we could have a backend with better properties. We probably need multiple backends for different purposes. This would be made easier if the backend were separated from the rest of the code. This would be the purpose of logic/openoffice/backend.

Undo

- JabRef 5.3 does not collect the effects of GUI actions on the document into larger Undo actions.
  This makes the Undo functionality of LO impractical.
- `(change)` collect the effects of GUI actions into large chunks: now a GUI action can be undone with a single click.
  - except the effect on pageInfo: that is stored at the document level and is not restored by Undo.

Block screen refresh

- LibreOffice has support in XModel to “suspend some notifications to the controllers which are used for display updates.”
- `(change)` Now we are using this facility.

The LibreOffice Panel

TABLE OF CONTENTS

- Overview
- Order of appearance of citation groups
- Problems
- Code reorganization
- About OOError, OOResult, and OOVoidResult
- Alternatives to using OOResult and OOVoidResult in OOBibBase

Remote JabDrive storage

This describes the synchronization to JabDrive. JabRef Online also implements the JabDrive interface.

The setting is that clients synchronize their local view with a server. The server itself does not change data on itself. If it does, it needs to create a separate client connecting to the server. Thus, all changes are finally triggered by a client.

The following algorithm is highly inspired by the replication protocols of CouchDB and RxDB. For the explanation, we focus on the synchronization of entries. Nevertheless, the synchronization of other data (such as the groups tree) works similarly.

From a high-level perspective, the sync algorithm is very similar with git: both the server and the client have their own change histories, and the client has to first pull and merge changes from the server before pushing its new state to the server. The sync process is incremental and only examines entries updated since the last sync.

We call this the “pull-merge-push cycle”.

Data structures

We start by providing information on data structures. There are some explanations of data structures included if they are short. Longer explanations are put below at “The ‘pull-merge-push cycle’”.

Metadata for each item

In order to support synchronization, additional metadata is kept for each item:

- `ID`: An unique identifier for the entry (will be a UUID).
- `Revision`: The revision is a “generation Id” being increasing positive integer. This is based on Multiversion concurrency control (MVCC), where an increasing identifier (“time stamp”) is used.
- `hash`: This is the hash of the item (i.e., of all the data except for `Revision` and `hash`).
- (Client only) `dirty`: Marks whether the user changed the entry.

`ID` and `Revision` are handled in `org.jabref.model.entry.SharedBibEntryData`.

> **DIRTY FLAGS**
> Using dirty flags, the client keeps track of the changes that happened in the library since the last time the client was synchronized with the server. When the client loads a library

into memory, it computes the hash for each entry and compares it with the hash in the entry's metadata. In case of a difference between these hashes, the entry is marked dirty. Moreover, an entry's dirty flag is set whenever it is modified by the user in JabRef. The dirty flag is only cleared after a successful synchronization process.

There is no need to serialize the dirty flags on the client's side since they are recomputed upon loading.

Global time clock

The idea is that the server tracks a global (logical) monotone increasing "time clock" tracking the existing revisions. Each entry has its own revision, increased "locally". The "global revision id" keeps track of the global synchronization state. One can view it as aggregation on the synchronization state of all entries. Similar to the revision concept of Subversion.

Tombstones

Deleted items are persisted as tombstones, which contain the metadata `ID` and `Revision` only. Tombstones ensure that all synchronizing devices can identify that a previously existing entry has been deleted. On the client, a tombstone is created whenever an entry is deleted. Moreover, the client keeps a list of all entries in the library so that external deletions can be recognized when loading the library into memory. The local list of tombstones is cleared after it is sent to the server and the server acknowledged it. On the server, tombstones are kept for a certain time span (world time) that is strictly larger than the time devices are allowed to not sign-in before removed as registered devices.

Checkpoints

Checkpoints allow a sync task to be resumed from where it stopped, without having to start from the beginning.

The checkpoint locally stored by the client signals the logical time (generation Id) of the last server change that has been integrated into the local library. Checkpoints are used to paginate the server-side changes. In the implementation, the checkpoint is a tuple consisting of the server time of the latest change and the highest `ID` of the entry in the batch. However, it is better to not depend on these semantics.

The client has to store a checkpoint `LastSync` in its local database, and it is updated after every merge. The checkpoint is then used as the `Since` parameter in the next Pull phase.

The "pull-merge-push cycle"

Each sync cycle is divided into three phases:

1. `Pull phase`: The server sends its local changes to the client.
2. `Merge phase`: The client and server merge their local changes.
3. `Push phase`: The client sends its local changes to the server.

We assume that the server has some view on the library and the client has a view on the library.

STRAIGHT-FORWARD SYNCHRONIZATION

When the client connects to the server, one option for synchronization is to ask the server for all up-to-date entries and then using the `Revision` information to merge with the local data. However, this is highly inefficient as the whole database has to be sent over the wire. A small improvement is gained by first asking only for tuples of `ID` and `Revision`, and only pull the complete entry if the local data is outdated or in conflict. However, this still requires to send quite a bit of data. Instead, we will use the following refinement.

Pull Phase

The client pulls on first connect or when requested to pull. The client asks the server for a list of documents that changed since the last checkpoint. (Creating a checkpoint is explained further below.) The server responses with a batched list of these entries together with their `Revision` information. These entries could also be tombstones. Each batch includes also a checkpoint `To` that has the meaning “all changes to this point in time are included in the current batch”.

NOTE

Once the pull does not give any further changes, the client switches to an event-based strategy and observes new changes by subscribing to the event bus provided by the server. This is more an implementation detail than a conceptual difference.

Merge Phase

The pulled data from the server needs to be merged with the local view of the data. The data is merged on a per-entry basis. Based on the “generation ID” (`Revision`) of server and client, following cases can occur:

1. The server’s `Revision` is higher than the client’s `Revision`: Two cases need to be distinguished:
   a. The client’s entry is dirty. That means, the user has edited the entry in the meantime. Then the user is shown a message to resolve the conflict (see “Conflict Handling” below)
   b. The client’s entry is clean. That means, the user has not edited the entry in the meantime. In this case, the client’s entry is replaced by the server’s one (including the revision).
2. The server’s `Revision` is equal to the client’s `Revision`: Both entries are up-to-date and nothing has to be done. This case may happen if the library is synchronized by other means.
3. The server’s `Revision` is lower than the client’s `Revision`: This should never be the case, as revisions are only increased on the server. Show error message to user.

If the entry returned by the server is a tombstone, then:

- If the client's entry is also a tombstone, then we do not have to do anything.
- If the client's entry is dirty, then the user is shown a message to resolve the conflict (see "Conflict Handling") below.
- Otherwise, the client's entry is deleted. There is no need to keep track of this as a local tombstone.

CONFLICT HANDLING

If the user chooses to overwrite the local entry with the server entry, then the entry's `Revision` is updated as well, and it is no longer marked as dirty. Otherwise, its `Revision` is updated to the one provided by the server, but it is still marked as dirty. This will enable pushing of the entry to the server during the "Push Phase".

After the merging is done, the client sets its local checkpoint to the value of `To`.

Push Phase

The client sends the following information back to the server:

- The list of entries that are marked dirty (along with their `Revision` data).
- The list of entries that are new, i.e., that do not have an `ID` yet.
- The list of tombstones, i.e., entries that have been deleted.

The server accepts only changes if the provided `Revision` coincides with the `Revision` stored on the server. If this is not the case, then the entry has been modified on the server since the last pull operation, and then the user needs to go through a new pull-merge-push cycle.

During the push operation, the user is not allowed to locally edit these entries that are currently pushed. After the push operation, all entries accepted by the server are marked clean. Moreover, the server will generate a new revision number for each accepted entry, which will then be stored locally. Entries rejected (as conflicts) by the server stay dirty and their `Revision` remains unchanged.

Start the "pull-merge-push cycle" again

It is important to note that sync replicates the library only as it was at the point in time when the sync was started. So, any additions, modifications, or deletions on the server-side after the start of sync will not be replicated. For this reason, a new cycle is started.

Scenarios

Having discussed the general algorithm, we discuss scenarios which can happen during usage. In the following, `T` denotes the "global generation Id".

We focus on JabRef as client and a "user" using JabRef.

Sync stops after Pull

1. JabRef pulls changes since `T = 0`
2. JabRef starts with the merge and the user (in parallel) closes JabRef discarding any changes.
3. User opens JabRef again.
4. JabRef pulls changes again from `T = 0` (since the checkpoint is still `T = 0`) and JabRef has to redo the conflict resolution.

This is the best we can do, since the user decided to not save its previous work.

However, consider the same steps but now in step 2, the user decided to save their work. The locally stored checkpoint is still `T = 0`. Thus, the user has to redo the conflict resolution again. The difference is that the local version is the previously merge result now.

Future improvement: We could send checkpoints for every entry and after each conflict resolution set the local checkpoint to the checkpoint of the entry.

Sync stops after Merge

1. JabRef pulls changes since `T = 0`
2. JabRef finishes the merge (this sets the checkpoint `T = 1`).
3. User closes JabRef with discarding any changes (in particular, the checkpoint is not persisted as well).
4. User opens JabRef again.
5. JabRef pulls changes again from `T = 0` (since the checkpoint is still `T = 0`) and has to redo the conflict resolution.

This is the best we can do, since the user decided to not save their previous work.

If the user decides in step 3 to save their changes, then in step 5 JabRef would pull changes starting from `T = 1` and the user does not have to redo the conflict resolution.

Sync after successful sync of client changes

1. JabRef modifies local data: `{id: 1, value: 0, _rev=1, _dirty=false}` to `{id: 1, value: 1, _rev=1, _dirty=true}`. `id` is `ID` from above, `value` summarizes all fields of the entry, `_rev` is `Revision` from above, and `_dirty` the dirty flag.
2. JabRef pulls server changes. Suppose there are none.
3. Consequently, Merge is not necessary. JabRef sets checkpoint to `T = 1`.
4. JabRef pushes its changes to the server. Assume this corresponds to `T = 2` on the server. On the server, this updates `{id: 1, value: 0, _rev=1, updatedAt=1}` to `{id: 1, value: 1, _rev=2, updatedAt=2}` and on the client `{id: 1, value: 1, _rev=1, _dirty=true}` to `{id: 1, value: 1, _rev=2, _dirty=false}`.
5. Client pulls changes starting from `T = 1` (the last local checkpoint). Server responds with `{id: 1, value: 1, _rev=2}, checkpoint={2}`.

6. Client merges the ‘changes’, which in this case is trivial since the data on the server and client is the same.

This is not quite optimal since the last pull response contains the full data of the entry although this data is already at the client.

Possible future improvements:

- First pull only the `IDs` and `Revisions` of the server-side changes, and then filter out the ones we already have locally before querying the complete entry. Downside is that this solution always needs one more request (per change batch) and it is not clear if this outweighs the costs of sending the full entry.
- The server can remember where a change came from and then not send these changes back to that client. Especially if the server’s generation Id increased by one due to the update, this is straight-forward.

FAQs

Why do we need an identifier (`ID`)? Is the BibTeX key not enough?

The identifier needs to be unique at the very least across the library and should stay constant in time. Both features cannot be ensured for BibTeX keys. Note this is similar to the `shared_id` in the case of the SQL synchronization.

Why do we need revisions? Are `updatedAt` timeflags not enough?

The revision functions as “generation Id” known from Lamport clocks and common in synchronization. For instance, the Optimistic Offline Lock also uses these kinds of clocks.

A “generation Id” is essentially a clock local to the entry that ticks whenever the entry is synced with the server. As for us there is only one server, strictly speaking, it would suffice to use the global server time for this. Moreover, for the sync algorithm, the client would only need to store the revision/server time during the pull-merge-push cycle (to make sure that during this time the entry is not modified again on the server). Nevertheless, the generation Id is only a tiny data blob, and it gives a bit of additional security/consistency during the merge operation, so we keep it around all the time.

Why do we need an entry hash?

The hash is only used on the client to determine whether an entry has been changed outside of JabRef.

WHY DON’T WE NEED TO KEEP THE WHOLE REVISION HISTORY AS IT IS DONE IN COUCHDB?

The revision history is used by CouchDB to find a common ancestor of two given revisions. This is needed since CouchDB provides main-main sync. However, in our setting, we have a central server and thus the last synced revision is *the* common ancestor for both the new server and client revision.

Why is a dirty flag enough on the client? Why don't we need local revisions?

In CouchDB, every client has their own history of revisions. This is needed to have a deterministic conflict resolution that can run on both the server and client side independently. In this setting, it is important to determine which revision is older, which is then declared to be the winner. However, we do not need an automatic conflict resolution: Whenever there is a conflict, the user is asked to resolve it. For this it is not important to know how many times (and when) the user changed the entry locally. It suffices to know that it changed at some point from the last synced version.

Local revision histories could be helpful in scenarios such as the following:

1. Device A is offline, and the user changes an entry.
2. The user sends this changed entry to Device B (say, via git).
3. The user further modifies the entry on Device B.
4. The user syncs Device B with the server.
5. The user syncs Device A with the server.

Without local revisions, it is not possible for Device A to figure out that the entry from the server logically evolved from its own local version. Instead, it shows a conflict message since the entry changed locally (step 1) and there is a newer revision on the server (from step 4).

More Readings

- CouchDB style sync and conflict resolution on Postgres with Hasura: Explains how to implement a sync algorithm in the style of CouchDB on your own
- A Comparison of Offline Sync Protocols and Implementations
- Offline data synchronization: Discusses different strategies for offline data sync, and when to use which.
- Transaction Processing: Concepts and Techniques
- Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Remote SQL Storage

For user documentation, see https://docs.jabref.org/collaborative-work/sqldatabase.

Handling large shared databases

Synchronization times may get long when working with a large database containing several thousand entries. Therefore, synchronization only happens if several conditions are fulfilled:

- Edit to another field.
- Major changes have been made (pasting or deleting more than one character).

Class `org.jabref.logic.util.CoarseChangeFilter.java` checks both conditions.

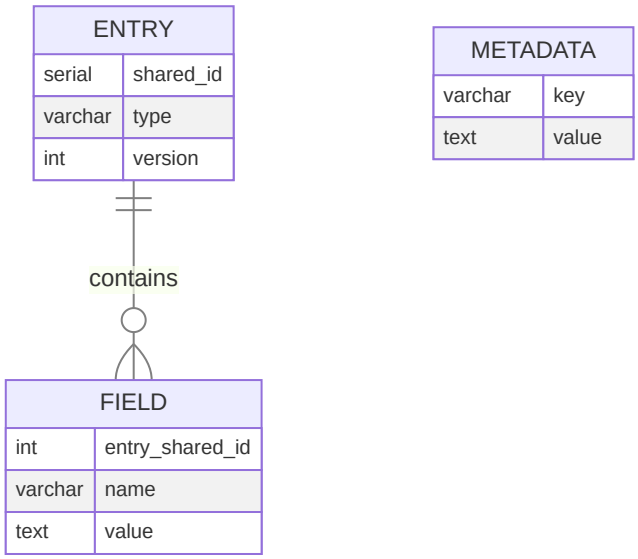
Remaining changes that have not been synchronized yet are saved at closing the database rendering additional closing time. Saving is realized in `org.jabref.logic.shared.DBMSSynchronizer.java`. Following methods account for synchronization modes:

- `pullChanges` synchronizes the database unconditionally.
- `pullLastEntryChanges` synchronizes only if there are remaining entry changes. It is invoked when closing the shared database (`closeSharedDatabase`).

Database structure

The following examples base on PostgreSQL. Other databases work similar.

The database structure is created at org.jabref.logic.shared.PostgreSQLProcessor#setUp.



The “secret sauce” is the `version` of an entry. This version is used as version in the sense of an Optimistic Offline Lock, which in turn is a well-established technique to prevent conflicts in concurrent business transactions. It assumes that the chance of conflict is low. Implementation details are found at https://www.baeldung.com/cs/offline-concurrency-control.

The `shared_id` and `version` are handled in `org.jabref.model.entry.SharedBibEntryData`.

Synchronization

PostgreSQL supports to register listeners on the database on changes. (MySQL does not). The listening is implemented at `org.jabref.logic.shared.listener.PostgresSQLNotificationListener`. It “just” fetches updates from the server when a change occurred there. Thus, the changes are not actively pushed from the server, but still need to be fetched by the client.

Remote Storage

JabRef supports kinds of remote storage:

- JabDrive
- SQL databases

The first one is the more modern approach allowing offline-work. The second approach makes use of the SQL features of databases and require direct online connections.

More details in JabDrive and SQL Storage respectively.

TABLE OF CONTENTS

- Remote JabDrive storage
- Remote SQL Storage

Testing JabRef

In JabRef, we mainly rely on basic JUnit unit tests to increase code coverage.

General hints on tests

Imagine you want to test the method `format(String value)` in the class `BracesFormatter` which removes double braces in a given string.

- *Placing:* all tests should be placed in a class named `classTest`, e.g. `BracesFormatterTest`.
- *Naming:* the name should be descriptive enough to describe the whole test. Use the format `methodUnderTest_ expectedBehavior_context` (without the dashes). So for example `formatRemovesDoubleBracesAtBeginning`. Try to avoid naming the tests with a `test` prefix since this information is already contained in the class name. Moreover, starting the name with `test` leads often to inferior test names (see also the Stackoverflow discussion about naming).
- *Test only one thing per test:* tests should be short and test only one small part of the method. So instead of

  ```
  void format() {
      assertEqual("test", format("test"));
      assertEqual("{test", format("{test"));
      assertEqual("test", format("test}}"));
  }
  ```

  we would have five tests containing a single `assert` statement and named accordingly (`formatDoesNotChangeStringWithoutBraces`, `formatDoesNotRemoveSingleBrace`, , etc.). See JUnit AntiPattern for background.
- Do *not just test happy paths*, but also wrong/weird input.
- It is recommended to write tests *before* you actually implement the functionality (test driven development).
- *Bug fixing:* write a test case covering the bug and then fix it, leaving the test as a security that the bug will never reappear.
- Do not catch exceptions in tests, instead use the `assertThrows(Exception.class, () -> doSomethingThrowsEx())` feature of junit-jupiter to the test method.

Coverage

IntelliJ has build in test coverage reports. Choose “Run with coverage”.

For a full coverage report as HTML, execute the gradle task `jacocoTestReport` (available in the “verification” folder in IntelliJ). Then, you will find <build/reports/jacoco/test/html/index.html> which shows the coverage of the tests.

Lists in tests

Instead of

```
assertTrue(actualList.isEmpty());
```

use

```
assertEquals(List.of(), actualList);
```

Similarly, to compare lists, instead of following code:

```
assertEquals(2, actualList.size());
assertEquals("a", actualList.get(0));
assertEquals("b", actualList.get(1));
```

use the following code:

```
assertEquals(List.of("a", "b"), actualList);
```

BibEntries in tests

- Use the `assertEquals` methods in `BibtexEntryAssert` to check that the correct BibEntry is returned.

Files and folders in tests

If you need a temporary file in tests, use the `@TempDir` annotation:

```
class TestClass{

  @Test
  void deletionWorks(@TempDir Path tempDir) {
  }
}
```

to the test class. A temporary file is now created by `Files.createFile(path)`. Using this pattern automatically ensures that the test folder is deleted after the tests are run. See https://www.geeksforgeeks.org/junit-5-tempdir/ for more details.

Loading Files from Resources

Sometimes it is necessary to load a specific resource or to access the resource directory

```
Path resourceDir = Paths.get(MSBibExportFormatTestFiles.class.getResource("MsBibExportFormatTest1.bib").
```

When the directory is needed, it is important to first point to an actual existing file. Otherwise the wrong directory will be returned.

Preferences in tests

If you modify preference, use following pattern to ensure that the stored preferences of a developer are not affected:

Or even better, try to mock the preferences and insert them via dependency injection.

```
@Test
public void getTypeReturnsBibLatexArticleInBibLatexMode() {
     // Mock preferences
     PreferencesService mockedPrefs = mock(PreferencesService.class);
     GeneralPreferences mockedGeneralPrefs = mock(GeneralPReferences.class);
     // Switch to BibLatex mode
     when(mockedPrefs.getGeneralPrefs()).thenReturn(mockedGeneralPrefs);
     when(mockedGeneralPrefs.getDefaultBibDatabaseMode())
        .thenReturn(BibDatabaseMode.BIBLATEX);

     // Now test
     EntryTypes biblatexentrytypes = new EntryTypes(mockedPrefs);
     assertEquals(BibLatexEntryTypes.ARTICLE, biblatexentrytypes.getType("article"));
}
```

To test that a preferences migration works successfully, use the mockito method `verify`. See `PreferencesMigrationsTest` for an example.

Database tests

PostgreSQL

To quickly host a local PostgreSQL database, execute following statement:

```
docker run -d -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -e POSTGRES_DB=postgres -p 5432:54
```

Set the environment variable `DBMS` to `postgres` (or leave it unset)

Then, all DBMS Tests (annotated with `@org.jabref.testutils.category.DatabaseTest`) run properly.

MySQL

A MySQL DBMS can be started using following command:

```
docker run -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=jabref -p 3800:3307 mysql:8.0 --port=3307
```

Set the environment variable `DBMS` to `mysql`.

Fetchers in tests

Fetcher tests can be run locally by executing the Gradle task `fetcherTest`. This can be done by running the following command in the command line:

```
./gradlew fetcherTest
```

Alternatively, if one is using IntelliJ, this can also be done by double-clicking the `fetcherTest` task under the `other` group in the Gradle Tool window (`JabRef > Tasks > other > fetcherTest`).

“No matching tests found”

In case the output is “No matching tests found”, the wrong test category is used.

Check “Run/Debug Configurations”

Example

```
:databaseTest --tests "org.jabref.logic.importer.fileformat.pdf.PdfMergeMetadataImporterTest.pdfMetadata
```

This tells Gradle that `PdfMergeMetadataImporterTest` should be executed as database test. However, it is marked as `@FetcherTest`. Thus, change `:databaseTest` to `:fetcherTest` to get the test running.

Advanced testing and further reading

On top of basic unit testing, there are more ways to test a software:

Type	Techniques	Tool (Java)	Kind of tests	Used In JabRef
Functional	Dynamics, black box, positive and negative	JUnit-QuickCheck	Random data generation	No, not intended, because other test kinds seem more helpful.
Functional	Dynamics, black box, positive and negative	GraphWalker	Model-based	No, because the BibDatabase doesn’t need to be tests

Type	Techniques	Tool (Java)	Kind of tests	Used In JabRef
Functional	Dynamics, black box, positive and negative	TestFX	GUI Tests	Yes
Functional	Dynamics, black box, negative	Lincheck	Testing concurrent algorithms	No
Functional	Dynamics, white box, negative	PIT	Mutation	No
Functional	Dynamics, white box, positive and negative	Mockito	Mocking	Yes
Non-functional	Dynamics, black box, positive and negative	JETM, Apache JMeter	Performance (performance testing vs load testing respectively)	No
Structural	Static, white box	CheckStyle	Constient formatting of the source code	Yes
Structural	Dynamics, white box	SpotBugs	Reocurreing bugs (based on experience of other projects)	No

Useful development tooling

This page lists some software we consider useful.

Browser plugins

- Refined GitHub - GitHub on steroids
- GitHub Issue Link Status - proper coloring of linked issues and PRs.
- Codecov Browser Extension - displaying code coverage directly when browsing GitHub
- Sourcegraph Browser Extension - Navigate through source on GitHub

git hints

Here, we collect some helpful git hints

- https://github.com/blog/2019-how-to-undo-almost-anything-with-git
- So you need to change your commit
- awesome hints and tools regarding git: https://github.com/dictcp/awesome-git

Rebase everything as one commit on main

- Precondition: `JabRef/jabref` is configured as upstream.
- Fetch recent commits and prune non-existing branches: `git fetch upstream --prune`
- Merge recent commits: `git merge upstream/main`
- If there are conflicts, resolve them
- Reset index to upstream/main: `git reset upstream/main`
- Review the changes and create a new commit using git gui: `git gui`
- Do a force push: `git push -f origin`

See also: https://help.github.com/articles/syncing-a-fork/

Tooling for Windows

(As Administrator - one time)

1. Install chocolatey
2. `choco install git.install -y --params "/GitAndUnixToolsOnPath /WindowsTerminal"`
3. `choco install notepadplusplus`
4. If you want to have your JDK also managed via chocolatey: `choco install temurin`

Then, each weak do `choco upgrade all` to ensure all tooling is kept updated.

General git tooling on Windows

- Use git for windows, no additional git tooling required
  - Git Credential Manager for Windows is included. Ensure that you include that in the installation. Aim: Store password for GitHub permanently for `https` repository locations
- Use notepad++ as editor for `git rebase -i`

Better console applications

CONEMU PLUS CLINK

- `choco install conemu clink`
- ConEmu -> Preview Version - Aim: Colorful console with tabs
  - At first start:
    - "Choose your startup task ...": `{Bash::Git bash}}
    - `OK`
    - Upper right corner: "Settings..." (third entrry Eintrag)
    - Startup/Tasks: Choose task no. 7 ("Bash::Git bash"). At "Task parameters" `/dir C:\git-repositories\jabref\jabref`
    - `Save Settings`
- clink - Aim: Unix keys (`Alt+B`, `Ctrl+S`, etc.) also available at the prompt of `cmd.exe`

OTHER BUNDLES

- Cmder - bundles ConEmu plus clink

Tools for working with XMP

- Validate XMP: https://www.pdflib.com/pdf-knowledge-base/xmp/free-xmp-validator

---

UI Design Recommendations

- Designing More Efficient Forms: Structure, Inputs, Labels and Actions
- Input form label alignment top or left?
  - For a usual form, place the label above the text field
  - If the user uses the form often to edit fields, then it might make sense to switch to left-aligned labels

Designing GUI Confirmation Dialogs

1. Avoid asking questions
2. Be as concise as possible
3. Identify the item at risk
4. Name your buttons for the actions

More information:

- StackOverflow: What are some alternatives to the phrase “Are you sure you want to XYZ” in confirmation dialogs?.
- JabRef issue discussing Yes/No/Cancel: koppor#149.

Name your buttons for the actions

`req~ui.dialogs.confirmation.naming~1`

Needs: impl

Form validation

- Only validate input after leaving the field (or after the user stopped typing for some time)
- The user shouldn’t be able to submit the form if there are errors
- However, disabling the submit button in case there are errors is also not optimal. Instead, clicking the submit button should highlight the errors.
- Empty required files should not be marked as invalid until the user a) tried to submit the form or b) focused the field, deleted its contents and then left the field (see Example).
- Ideally, the error message should be shown below the text field and not as a tooltip (so that users quickly understand what’s the problem). For example as in Boostrap.

XMP Parsing

Example XMP metadata from a PDF file (src/test/resources/org/jabref/logic/importer/fileformat/pdf/2024_SPLC_Becker.pdf):

```
<?xpacket begin="" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
      <dc:format>application/pdf</dc:format>
      <dc:identifier>doi:10.1145/3646548.3672587</dc:identifier>
    </rdf:Description>
    <rdf:Description rdf:about="" xmlns:prism="http://prismstandard.org/namespaces/basic/2.1/">
      <prism:doi>10.1145/3646548.3672587</prism:doi>
      <prism:url>https://doi.org/10.1145/3646548.3672587</prism:url>
    </rdf:Description>
    <rdf:Description rdf:about="" xmlns:crossmark="http://crossref.org/crossmark/1.0/">
      <crossmark:MajorVersionDate>2024-09-02</crossmark:MajorVersionDate>
      <crossmark:CrossmarkDomainExclusive>true</crossmark:CrossmarkDomainExclusive>
      <crossmark:CrossMarkDomains>
        <rdf:Seq>
          <rdf:li>dl.acm.org</rdf:li>
        </rdf:Seq>
      </crossmark:CrossMarkDomains>
      <crossmark:DOI>10.1145/3646548.3672587</crossmark:DOI>
    </rdf:Description>
    <rdf:Description rdf:about="" xmlns:pdfx="http://ns.adobe.com/pdfx/1.3/">
      <pdfx:CrossMarkDomains>
        <rdf:Seq>
          <rdf:li>dl.acm.org</rdf:li>
        </rdf:Seq>
      </pdfx:CrossMarkDomains>
      <pdfx:CrossmarkDomainExclusive>true</pdfx:CrossmarkDomainExclusive>
      <pdfx:doi>10.1145/3646548.3672587</pdfx:doi>
      <pdfx:CrossmarkMajorVersionDate>2024-09-02</pdfx:CrossmarkMajorVersionDate>
    </rdf:Description>
  </rdf:RDF>
```

```
</x:xmpmeta>
<?xpacket end="w"?>
```

`org.apache.xmpbox.xml.DomXmpParser` cannot ignore unknown namespaces. Therefore, we need to exact the known elements.

Code Howtos

This page provides some development support in the form of howtos. See also High Level Documentation.

Generic code how tos

We really recommend reading the book Java by Comparison.

Please read https://github.com/cxxr/better-java.

- try not to abbreviate names of variables, classes or methods
- use lowerCamelCase instead of snake_case
- name enums in singular, e.g. `Weekday` instead of `Weekdays` (except if they represent flags)

Dependency injection

JabRef uses a fork of the afterburner.fx framework by Adam Bien.

The main idea is to get instances by using `Injector.instantiateModelOrService(X.class)`, where `X` is the instance one needs. The method `instantiateModelOrService` checks if there is already an instance of the given class. If yes, it returns it. If not, it creates a new one. A singleton can be added by `com.airhacks.afterburner.injection.Injector#setModelOrService(X.class, y)`, where X is the class and y the instance you want to inject.

Cleanup and Formatters

We try to build a cleanup mechanism based on formatters. The idea is that we can register these actions in arbitrary places, e.g., onSave, onImport, onExport, cleanup, etc. and apply them to different fields. The formatters themselves are independent of any logic and therefore easy to test.

Example: NormalizePagesFormatter

Drag and Drop

Drag and Drop makes usage of the Dragboard. For JavaFX the following tutorial is helpful. Note that the data has to be serializable which is put on the dragboard. For drag and drop of Bib-entries between the maintable and the groups panel, a custom Dragboard is used, `CustomLocalDragboard` which is a generic alternative to the system one.

For accessing or putting data into the Clipboard use the `ClipboardManager`.

Get the JabRef frame panel

`JabRefFrame` and `BasePanel` are the two main classes. You should never directly call them, instead pass them as parameters to the class.

Get Absolute Filename or Path for file in File directory

JabRef stores files relative to one of multiple possible directories. The convert the relative path to an absolute one, there is the `find` method in `FileUtil`:

```
org.jabref.logic.util.io.FileUtil.find(org.jabref.model.database.BibDatabaseContext, java.lang.String, o
```

`String path` Can be the files name or a relative path to it. The Preferences should only be directly accessed in the GUI. For the usage in logic pass them as parameter

Get a relative filename (or path) for a file

JabRef offers multiple directories per library to store a file.. When adding a file to a library, the path should be stored relative to “the best matching” directory of these. This is implemented in `FileUtil`:

```
org.jabref.logic.util.io.FileUtil.relativize(java.nio.file.Path, org.jabref.model.database.BibDatabaseCo
```

Setting a Directory for a .bib File

- `@comment{jabref-meta: fileDirectory:<directory>`
- “fileDirectory” is determined by Globals.pref.get(“userFileDir”) (which defaults to “fileDirectory”
- There is also “fileDirectory-<username>”, which is determined by Globals.prefs.get(“userFileDirIndividual”)
- Used at DatabasePropertiesDialog

How to work with Preferences

`model` and `logic` must not know `JabRefPreferences`. See `ProxyPreferences` for encapsulated preferences and https://github.com/JabRef/jabref/pull/658 for a detailed discussion.

See https://github.com/JabRef/jabref/blob/master/src/main/java/org/jabref/logic/preferences/TimestampPreferences.java (via https://github.com/JabRef/jabref/pull/3092) for the current way how to deal with preferences.

Defaults should go into the model package. See Comments in this Commit

UI

Global variables should be avoided. Try to pass them as dependency.

“Special Fields”

keywords sync

`Database.addDatabaseChangeListener` does not work as the `DatabaseChangedEvent` does not provide the field information. Therefore, we have to use `BibtexEntry.addPropertyChangeListener(VetoableChangeListener listener)`.

Working with BibTeX data

Working with authors

You can normalize the authors using `org.jabref.model.entry.AuthorList.fixAuthor_firstNameFirst(String)`. Then the authors always look nice. The only alternative containing all data of the names is `org.jabref.model.entry.AuthorList.fixAuthor_lastNameFirst(String)`. The other `fix...` methods omit data (like the “von” parts or the junior information).

Benchmarks

- Benchmarks can be executed by running the `jmh` gradle task (this functionality uses the JMH Gradle plugin)
- Best practices:
  - Read test input from `@State` objects
  - Return result of calculations (either explicitly or via a `BlackHole` object)
- List of examples

Measure performance

Try out the YourKit Java Profiler.

equals

When creating an `equals` method follow:

1. Use the `==` operator to check if the argument is a reference to this object. If so, return `true`.
2. Use the `instanceof` operator to check if the argument has the correct type. If not, return `false`.
3. Cast the argument to the correct type.

4. For each “significant” field in the class, check if that field of the argument matches the corresponding field of this object. If all these tests succeed, return `true` otherwise, return `false`.
5. When you are finished writing your `equals` method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?

Also, note:

- Always override `hashCode` when you override equals (`hashCode` also has very strict rules) (Item 9 ofEffective Java)
- Don’t try to be too clever
- Don’t substitute another type for `Object` in the equals declaration

Files and Paths

Always try to use the methods from the nio-package. For interoperability, they provide methods to convert between file and path. https://docs.oracle.com/javase/tutorial/essential/io/path.html Mapping between old methods and new methods https://docs.oracle.com/javase/tutorial/essential/io/legacy.html#mapping

---

TABLE OF CONTENTS

- The LibreOffice Panel
- Code Quality
- Custom SVG icons
- Error Handling in JabRef
- Event Bus and Event System
- Fetchers
- Frequently Asked Questions (FAQ)
- HTTP Server
- IntelliJ Hints
- JPackage: Creating a binary and debug it
- JabRef’s handling of BibTeX
- JavaFX
- Localization
- Logging
- Remote Storage
- Testing JabRef
- UI Design Recommendations
- Useful development tooling
- XMP Parsing

Contributing

Please head to our contributing guide in the main repository.

Use Markdown Architectural Decision Records

Context and Problem Statement

We want to record architectural decisions made in this project independent whether decisions concern the architecture (“architectural decision record”), the code, or other fields. Which format and structure should these records follow?

Considered Options

- MADR 4.0.0 – The Markdown Architectural Decision Records
- Michael Nygard’s template – The first incarnation of the term “ADR”
- Sustainable Architectural Decisions – The Y-Statements
- Other templates listed at https://github.com/joelparkerhenderson/architecture_decision_record
- Formless – No conventions for file format and structure

Decision Outcome

Chosen option: “MADR 4.0.0”, because

- Implicit assumptions should be made explicit. Design documentation is important to enable people understanding the decisions later on. See also “A rational design process: How and why to fake it”.
- MADR allows for structured capturing of any decision.
- The MADR format is lean and fits our development style.
- The MADR structure is comprehensible and facilitates usage & maintenance.
- The MADR project is vivid.

Use Crowdin for translations

Context and Problem Statement

The JabRef UI is offered in multiple languages. It should be easy for translators to translate the strings.

Considered Options

- Use Crowdin
- Use popeye
- Use Lingohub
- Keep current GitHub flow. See the Step-by-step guide.

Decision Outcome

Chosen option: “Use Crowdin”, because Crowdin is easy to use, integrates in our GitHub workflow, and is free for OSS projects.

Use SLF4J together with log4j2 for logging

Context and Problem Statement

Up to version 4.1 JabRef uses apache-commons-logging 1.2 for logging errors and messages. However, this is not compatible with java 9 and is superseded by log4j.

Decision Drivers

- SLF4J provides a façade for several logging frameworks, including log4j and supports already java 9
- Log4j is already defined as dependency and SLF4J has already been required by a third party dependency

Considered Alternatives

- Log4j2
- SLF4J with Log4j2 binding
- SLF4J with Logback binding

Decision Outcome

Chosen option: “SLF4J with Log4j2 binding”, because comes out best (see below).

Pros and Cons of the Options

Log4j2

- Good, because dependency already exists
- Good, because Java 9 support since version 2.10
- Bad, because direct dependency

SLF4J with log4j2 binding

- Good, because it only requires minimal changes to our logging infrastructure
- Good, because Apache Log4j 2 is an upgrade to Log4j that provides significant improvements over its predecessor, Log4j 1.x, and provides many of the improvements available in Logback while fixing some inherent problems in Logback’s architecture.
- Good, because supports other loggers as well

- Good, because Java 9 support
- Good, because already defined
- Good, because migration tool available
- Good, because it is a façade for several loggers. Thus, the underlying implementation can easily be changed in the future.
- Bad, because logger statements require a slight different syntax

SLF4J with Logback binding

- Good, because migration tool available
- Good, because native implementation of SLF4J
- Bad, because Java 9 support only available in alpha
- Bad, because different syntax than log4j/commons logging

Use Gradle as build tool

Context and Problem Statement

Which build tool should be used?

Considered Options

- Maven
- Gradle
- Ant

Decision Outcome

Chosen option: “Gradle”, because it is lean and fits our development style.

Pros and Cons of the Options

Maven

- Good, because there is a plugin for almost everything
- Good, because it has good integration with third party tools
- Good, because it has robust performance
- Good, because it has a high popularity
- Good, if one favors declarative over imperative
- Bad, because getting a dependency list is not straight forward
- Bad, because it based on a fixed and linear model of phases
- Bad, because it is hard to customize
- Bad, because it needs plugins for everything
- Bad, because it is verbose leading to huge build files

Gradle

- Good, because its build scripts are short
- Good, because it follows the convention over configuration approach
- Good, because it offers a graph-based task dependencies
- Good, because it is easy to customize

- Good, because it offers custom dependency scopes
- Good, because it has good community support
- Good, because its performance can be 100 times more than maven’s performance.
- Bad, because not that many plugins are available/maintained yet
- Bad, because it lacks a wide variety of application server integrations
- Bad, because it has a medium popularity
- Bad, because it allows custom build scripts which need to be debugged

Ant

- Good, because it offers a lot of control over the build process
- Good, because it has an agile dependency manager
- Good, because it has a low learning curve
- Bad, because build scripts can quickly become huge
- Bad, because everything has to be written from scratch
- Bad, because no conventions are enforced which can make it hard to understand someone else’s build script
- Bad, because it has nearly no community support
- Bad, because it has a low popularity
- Bad, because it offers too much freedom

Links

- GADR: https://github.com/adr/gadr-java/blob/master/gadr-java–build-tool.md

Use MariaDB Connector

Context and Problem Statement

JabRef needs to connect to a MySQL database. See Shared SQL Database for more information.

Considered Options

- Use MariaDB Connector
- Use MySQL Connector

Other alternatives are listed at https://stackoverflow.com/a/31312280/873282.

Decision Outcome

Chosen option: “Use MariaDB Connector”, because comes out best (see below).

Pros and Cons of the Options

Use MariaDB Connector

The MariaDB Connector is a LGPL-licensed JDBC driver to connect to MySQL and MariaDB.

- Good, because can be used as drop-in replacement for MySQL connector

Use MySQL Connector

The MySQL Connector is distributed by Oracle and licensed under GPL-2. Source: https://github.com/mysql/mysql-connector-j/blob/release/9.x/LICENSE. Oracle added the Universal FOSS Exception, Version 1.0 to it, which seems to limit the effects of GPL. More information on the FOSS Exception are available at https://www.mysql.com/de/about/legal/licensing/foss-exception/.

- Good, because it stems from the same development team than MySQL
- Bad, because the “Universal FOSS Exception” makes licensing more complicated.

Fully Support UTF-8 Only For LaTeX Files

Context and Problem Statement

The feature search for citations displays the content of LaTeX files. The LaTeX files are text files and might be encoded arbitrarily.

Considered Options

- Support UTF-8 encoding only
- Support ASCII encoding only
- Support (nearly) all encodings

Decision Outcome

Chosen option: “Support UTF-8 encoding only”, because comes out best (see below).

Positive Consequences

- All content of LaTeX files are displayed in JabRef

Negative Consequences

- When a LaTeX files is encoded in another encoding, the user might see strange characters in JabRef

Pros and Cons of the Options

Support UTF-8 encoding only

- Good, because covers most tex file encodings
- Good, because easy to implement
- Bad, because does not support encodings used before around 2010

Support ASCII encoding only

- Good, because easy to implement
- Bad, because does not support any encoding at all

Support (nearly) all encodings

- Good, because easy to implement
- Bad, because it relies on Apache Tika's `CharsetDetector`, which resides in `tika-parsers`.

  This causes issues during compilation (see https://github.com/JabRef/jabref/pull/3421#issuecomment-524532832).

  Example: `error: module java.xml.bind reads package javax.activation from both java.activation and jakarta.activation`.

Only translated strings in language file

Context and Problem Statement

JabRef has translation files `JabRef_it.properties`, ... There are translated and untranslated strings. Which ones should be in the translation file?

Decision Drivers

- Translators should find new strings to translate easily
- New strings to translate should be written into `JabRef_en.properties` to enable translation by the translators
- Crowdin should be kept as translation platform, because 1) it is much easier for the translators than the GitHub workflow and 2) it is free for OSS projects.

Considered Options

- Only translated strings in language file
- Translated and untranslated strings in language file, have value the untranslated string to indicate untranslated
- Translated and untranslated strings in language file, have empty to indicate untranslated

Decision Outcome

Chosen option: “Only translated strings in language file”, because comes out best (see below).

Pros and Cons of the Options

Only translated strings in language file

- Good, because Crowdin supports it
- Bad, because translators need tooling to see untranslated strings
- Bad, because issues with FXML (https://github.com/JabRef/jabref/issues/3796)

Translated and untranslated strings in language file, have value the untranslated string to indicate untranslated

- Good, because no issues with FXML
- Good, because Crowdin supports it

- Bad, because untranslated strings cannot be identified easily in Latin languages

Translated and untranslated strings in language file, have empty to indicate untranslated

- Good, because untranslated strings can be identified easily
- Good, because works with FMXL (?)
- Bad, because Crowdin does not support it

Links

- Related to ADR-0001.

Provide a human-readable changelog

Context and Problem Statement

Changes of a release have to be communicated. How and which style to use?

Considered Options

- Keep-a-changelog format with freedom in the bullet points
- Keep-a-changelog format and fixed terms

Decision Outcome

Chosen option: “Keep-a-changelog format with freedom in the bullet points”, because

- Keep-a-changelog structures the changelog
- Each entry can be structured to be understandable
- Forcing to prefix each line with `We fixed`, `We changed`, … seems to be read strange.

  We nevertheless try to follow that style.

Further discussion can be found at #2277.

Use `public final` instead of getters to offer access to immutable variables

Context and Problem Statement

When making immutable data accessible in a java class, should it be using getters or by non-modifiable fields?

Considered Options

- Offer public static field
- Offer getters

Decision Outcome

Chosen option: “Offer public static field”, because getters used to be a convention which was even more manifested due to libraries depending on the existence on getters/setters. In the case of immutable variables, adding public getters is just useless since one is not hiding anything.

Positive Consequences

- Shorter code

Negative Consequences

- newcomers could get confused, because getters/setters are still taught

Use Plain JUnit5 for advanced test assertions

Context and Problem Statement

How to write readable test assertions? How to write readable test assertions for advanced tests?

Considered Options

- Plain JUnit5
- Hamcrest
- AssertJ

Decision Outcome

Chosen option: “Plain JUnit5”, because comes out best (see below).

Positive Consequences

- Tests are more readable
- More easy to write tests
- More readable assertions

Negative Consequences

- More complicated testing leads to more complicated assertions

Pros and Cons of the Options

Plain JUnit5

Homepage: https://junit.org/junit5/docs/current/user-guide/ JabRef testing guidelines: <../testing.md>

Example:

```
String actual = markdownFormatter.format(source);
assertTrue(actual.contains("Markup<br />"));
```

```
assertTrue(actual.contains("<li>list item one</li>"));
assertTrue(actual.contains("<li>list item 2</li>"));
assertTrue(actual.contains("> rest"));
assertFalse(actual.contains("\n"));
```

- Good, because Junit5 is “common Java knowledge”
- Bad, because complex assertions tend to get hard to read
- Bad, because no fluent API

Hamcrest

Homepage: https://github.com/hamcrest/JavaHamcrest

- Good, because offers advanced matchers (such as `contains`)
- Bad, because not full fluent API
- Bad, because entry barrier is increased

AssertJ

Homepage: https://joel-costigliola.github.io/assertj/

Example:

```
assertThat(markdownFormatter.format(source))
        .contains("Markup<br />")
        .contains("<li>list item one</li>")
        .contains("<li>list item 2</li>")
        .contains("> rest")
        .doesNotContain("\n");
```

- Good, because offers fluent assertions
- Good, because allows partial string testing to focus on important parts
- Good, because assertions are more readable
- Bad, because not commonly used
- Bad, because newcomers have to learn an additional language to express test cases
- Bad, because entry barrier is increased
- Bad, because expressions of test cases vary from unit test to unit test

Links

- German comparison between Hamcrest and AssertJ: https://www.sigs-datacom.de/uploads/tx_dmjournals/philipp_JS_06_15_gRfN.pdf

Use H2 as Internal SQL Database

Context and Problem Statement

We need to store data internally in a structured way to gain performance.

Decision Drivers

- Easy to integrate
- Easy to use
- Common technology

Considered Options

- H2 Database Engine
- SQLite

Decision Outcome

Chosen option: “H2 Database Engine”, because it was straight-forward to use.

Links

- Comparison at SQL Workbench

Test external links in documentation

Context and Problem Statement

The JabRef repository contains Markdown (`.md`) files documenting the JabRef code. The documentation contains links to external resources. For high-quality documentation, external links should be working.

Decision Drivers

- Checking external links should not cause issues in the normal workflow

Considered Options

- Check external links once a month
- Check external links in the “checkstyle” task
- Do not check external links

Decision Outcome

Chosen option: “Check external links once a month”, because it provides a basic quality baseline.

Positive Consequences

- Automatic notification of broken external links

Negative Consequences

- Some external sites need to be disabled. For instance, GitHub.com always returns “forbidden”.
- Contributors find it strange if external links are broken (example: user-documentation#526)

Pros and Cons of the Options

Check external links once a month

- Good, because does not interfere with the normal development workflow
- Bad, because an additional workflow is required

Check external links in the “checkstyle” task

- Good, because no separate workflow is required
- Bad, because checks fail independent of the PR (because external web sites can go down and go up independent of a PR)

Do not check external links

- Good, because no testing at all is required
- Bad, because external links break without any notice
- Bad, because external links have to be checked manually

Handle different bibentry formats of fetchers by adding a layer

Context and Problem Statement

All fetchers (except IDFetchers) in JabRef return BibEntries when fetching entries from their API. Some fetchers directly receive BibTeX entries from their API, the other fetchers receive their entries in some kind of exchange format such as JSON or XML and then parse this into BibEntries. Currently, all fetchers either return BibEntries in BibTeX or BibLaTeX format. This can lead to importing BibEntries of one format in a database of the other format. How can this inconsistency between fetchers, and their used formats be addressed?

Considered Options

- Pass fetchers the format, they have to create entries accordingly (in the correct format).
- Pass fetchers the format, they have to call a conversion method if necessary (in the correct format).
- Let the caller handle any format inconsistencies and the conversion.
- Introduce a new layer between fetchers and caller, such as a FetcherHandler, that manages the conversion

Decision Outcome

Chosen option: “Introduce a new layer between fetchers and caller, such as a FetcherHandler, that manages the conversion”, because it can compose all steps required during importing, not only format conversion of fetched entries. As described here (comment)

Pros and Cons of the Options

Introduce a new layer between fetchers and caller, such as a FetcherHandler, that manages the conversion

- Good, because fetchers do not have to think about conversion (Separation of concerns)
- Good, because no other code that currently relies on fetchers has to do the conversion
- Good, because this layer can be used for any kind of import to handle all conversion steps (not only format). As described here (comment)
- Good, because this layer can easily be extended if the import procedure changes

- Bad, because this requires a lot of code changes
- Bad, because this has to be tested extensively

Pass fetchers the format, they have to call a conversion method if necessary

- Good, because less code has to be written than with option “Pass fetchers the format, they have to create entries accordingly”
- Good, because code is already tested
- Good, because keeps all conversion code centralized (code reuse)
- Bad, because fetcher first creates the BibEntry in a possibly “wrong” format, this can easily lead to bugs due to e.g. code changes
- Bad, because adds dependency

Pass fetchers the format, they have to create entries accordingly

- Good, because fetchers already handle BibEntry creation (in their format of choice). This is part of his responsibility.
- Good, because fetchers only create BibEntries of the “correct” format. At no point there exists the chance of the wrong format being passed on due to e.g. code changes.
- Good, because the conversion does not have to take place
- Bad, because fetcher has to “know” all differences of the formats -> clutters the code.
- Bad, because this code has to be tested. Conversion already exists.

Let the caller handle any format inconsistencies and the conversion

- Good, because fetcher code does not have to change
- Good, because fetcher only has to fetch and does not need to know anything about the formats
- Bad, because programmers might assume that a certain format is used, e.g. the preferred format (which would not work as the databases that imports the entries does not have to conform to the preferred format)
- Bad, because at every place where fetchers are used, and the format matters, conversion has to be used, creating more dependencies

---

Add Native Support for BibLatex-Software

- Deciders: Oliver Kopp

Technical Story: 6574-Adding support for biblatex-software

Context and Problem Statement

Right now, JabRef does not have support for Biblatex-Software out of the box, users have to add custom entry types. With citing software becoming fairly common, native support is helpful.

Decision Drivers

- None of the existing flows should be impacted

Considered Options

- Add the new entry types to the existing biblatex types
- Add a divider with label Biblatex-Software under which the new entries are listed: Native support for Biblatex-Software
- Support via customized entry types: A user can load a customized bib file

Decision Outcome

Chosen option: “Add a new divider”, because comes out best (see below).

Positive Consequences

- Inbuilt coverage for a entry type that is getting more and more importance

Negative Consequences

- Adds a little bit more clutter to the Add Entry pane

Pros and Cons of the Options

Add the new entry types to the existing biblatex types

- Good, because there is no need for a new category in the add entry pane

Add a divider with label Biblatex-Software under which the new entries are listed: Native support for Biblatex-Software

- Good, since this gives the user a bit more clarity

Support via customized entry types: A user can load a customized bib file

- Good, because no code needs to be changed
- Bad, because documentation is needed
- Bad, because the users are not guided through the UI, but have to do other steps.

Separate URL creation to enable proper logging

Context and Problem Statement

Fetchers are failing. The reason why they are failing needs to be investigated.

- Claim 1: Knowing the URL which was used to query the fetcher eases debugging
- Claim 2: Somehow logging the URL eases debugging (instead of showing it in the debugger only)

How to properly log the URL used for fetching?

Decision Drivers

- Code should be easy to read
- Include URL in the exception instead of logging in case an exception is thrown already (see https://howtodoinjava.com/best-practices/java-exception-handling-best-practices/#6)

Considered Options

- Separate URL creation
- Create URL when logging the URL
- Include URL creation as statement before the stream creation in the try-with-resources block

Decision Outcome

Chosen option: “Separate URL creation”, because comes out best (see below).

Pros and Cons of the Options

Separate URL creation

```
URL urlForQuery;
try {
    urlForQuery = getURLForQuery(query);
} catch (URISyntaxException | MalformedURLException | FetcherException e) {
```

```
        throw new FetcherException(String.format("Search URI %s is malformed", query), e);
    }
    try (InputStream stream = getUrlDownload(complexQueryURL).asInputStream()) {
        ...
    } catch (IOException e) {
        throw new FetcherException("A network error occurred while fetching from " + urlForQuery.toStrin
    } catch (ParseException e) {
        throw new FetcherException("An internal parser error occurred while fetching from " + urlForQuer
    }
```

- Good, because exceptions thrown at method are directly caught
- Good, because exceptions in different statements belong to different catch blocks
- Good, because code to determine URL is written once
- OK, because “Java by Comparison” does not state anything about it
- Bad, because multiple try/catch statements are required
- Bad, because this style seems to be uncommon to Java coders

Create URL when logging the URL

The “logging” is done when throwing the exception.

Example code:

```
    try (InputStream stream = getUrlDownload(getURLForQuery(query)).asInputStream()) {
        ...
    } catch (URISyntaxException | MalformedURLException | FetcherException e) {
        throw new FetcherException(String.format("Search URI %s is malformed", query), e);
    } catch (IOException e) {
        try {
            throw new FetcherException("A network error occurred while fetching from " + getURLForQuery(
        } catch (URISyntaxException | MalformedURLException uriSyntaxException) {
            // does not happen
            throw new FetcherException("A network error occurred", e);
        }
    } catch (ParseException e) {
        try {
            throw new FetcherException("An internal parser error occurred while fetching from " + getURL
        } catch (URISyntaxException | MalformedURLException uriSyntaxException) {
            // does not happen
            throw new FetcherException("An internal parser error occurred", e);
        }
    }
```

- Good, because code inside the `try` statement stays the same

- OK, because “Java by Comparison” does not state anything about it
- Bad, because an additional try/catch-block is added to each catch statement
- Bad, because needs a `throw` statement in the `URISyntaxException` catch block (even though at this point the exception cannot be thrown), because Java otherwise misses a `return` statement.

Include URL creation as statement before the stream creation in the try-with-resources block

```
try (URL urlForQuery = getURLForQuery(query); InputStream stream = urlForQuery.asInputStream()) {
    ...
} catch (URISyntaxException | MalformedURLException | FetcherException e) {
    throw new FetcherException(String.format("Search URI %s is malformed", query), e);
} catch (IOException e) {
    throw new FetcherException("A network error occurred while fetching from " + urlForQuery.toStrin
} catch (ParseException e) {
    throw new FetcherException("An internal parser error occurred while fetching from " + urlForQuer
}
```

- Good, because the single try/catch-block can be kept
- Good, because logical flow is kept
- Bad, because does not compile (because URL is not an `AutoClosable`)

Query syntax design

Context and Problem Statement

All libraries use their own query syntax for advanced search options. To increase usability, users should be able to formulate their (abstract) search queries in a query syntax that can be mapped to the library specific search queries. To achieve this, the query has to be parsed into an AST.

Which query syntax should be used for the abstract queries? Which features should the syntax support?

Considered Options

- Use a simplified syntax that is derived of the lucene query syntax
- Formulate a own query syntax

Decision Outcome

Chosen option: “Use a syntax that is derived of the lucene query syntax”, because only option that is already known, and easy to implement. Furthermore parsers for lucene already exist and are tested. For simplicity, and lack of universal capabilities across fetchers, only basic query features and therefor syntax is supported:

- All terms in the query are whitespace separated and will be ANDed
- Default and certain fielded terms are supported
- Fielded Terms:
  - `author`
  - `title`
  - `journal`
  - `year` (for single year)
  - `year-range` (for range e.g. `year-range:2012-2015`)
- The `journal`, `year`, and `year-range` fields should only be populated once in each query
- The `year` and `year-range` fields are mutually exclusive
- Example:
  - `author:"Igor Steinmacher" author:"Christoph Treude" year:2017` will be converted to
  - `author:"Igor Steinmacher" AND author:"Christoph Treude" AND year:2017`

The supported syntax can be expressed in EBNF as follows:

Query := {Clause}
Clause:= [Field] Term
Field := author: | title: | journal: | year: | year-range: | default:
Term := Word | Phrase \

Word can be derived to any series of non-whitespace characters. Phrases are multiple words wrapped in quotes and may contain white-space characters within the quotes.
Note: Even though this EBNF syntactically allows the creation of queries with year and year-range fields, such a query does not make sense semantically and therefore will not be executed.

Positive Consequences

- Already tested
- Well known
- Easy to implement
- Can use an existing parser

Pros and Cons of the Options

Use a syntax that is derived of the lucene query syntax

- Good, because already exists
- Good, because already well known
- Good, because there already exists a parser for lucene syntax
- Good, because capabilities of query conversion can easily be extended using the flexible lucene framework

Formulate a own query syntax

- Good, because allows for flexibility
- Bad, because needs a new parser (has to be decided whether to use ANTLR, JavaCC, or LogicNG)
- Bad, because has to be tested
- Bad, because syntax is not well known
- Bad, because the design should be easily extensible, requires an appropriate design (high effort)

Mutable preferences objects

Context and Problem Statement

To create an immutable preferences object every time seems to be a waste of time and computer memory.

Considered Options

- Alter the existing object and return it (by a `with*` -method, similar to a builder, but changing the object at hand).
- Create a new object every time a preferences object should be altered.

Decision Outcome

Chosen option: “Alter the exiting object”, because the preferences objects are just wrappers around the basic preferences framework of JDK. They should be mutable on-the-fly similar to objects with a Builder inside and to be stored immediately again in the preferences.

Allow org.jabref.model to access org.jabref.logic

Context and Problem Statement

- How to create a maintainable architecture?
- How to split model, logic, and UI

Decision Drivers

- Newcomers should find the architecture “split” natural
- The architecture should be a help (and not a burden)

Considered Options

- `org.jabref.model` uses `org.jabref.model` (and external libraries) only
- `org.jabref.model` may use `org.jabref.logic` in defined cases
- `org.jabref.model` and `org.jabref.logic` may access each other freely

Decision Outcome

Chosen option: “`org.jabref.model` may use `org.jabref.logic` in defined cases”, because comes out best (see below).

Pros and Cons of the Options

`org.jabref.model` uses `org.jabref.model` (and external libraries) only

The model package does not access logic or other packages of JabRef. Access to classes of external libraries is allowed. The logic package may use the model package.

- Good, because clear separation of model and logic
- Bad, because this leads to an Anemic Domain Model

`org.jabref.model` may use `org.jabref.logic` in defined cases

- Good, because model and logic are still separated
- Neutral, because each exception has to be discussed and agreed

- Bad, because newcomers have to be informed that there are certain (agreed) exceptions for model to access logic

`org.jabref.model` and `org.jabref.logic` may access each other freely

- Bad, because may lead to spaghetti code
- Bad, because coupling between model and logic is increased
- Bad, because cohesion inside model is decreased

Use regular expression to split multiple-sentence titles

Context and Problem Statement

Some entry titles are composed of multiple sentences, for example: “Whose Music? A Sociology of Musical Language”, therefore, it is necessary to first split the title into sentences and process them individually to ensure proper formatting using ‘Sentence Case’ or ‘Title Case’

Considered Options

- Regular expression
- OpenNLP
- ICU4J

Decision Outcome

Chosen option: “Regular expression”, because we can use Java internal classes (Pattern, Matcher) instead of adding additional dependencies

Positive Consequences

- Less dependencies on third party libraries
- Smaller project size (ICU4J is very large)
- No need for model data (OpenNLP is a machine learning based toolkit and needs a trained model to work properly)

Negative Consequences

- Regular expressions can never cover every case, therefore, splitting may not be accurate for every title

Implement special fields as separate fields

Context and Problem Statement

How to implement special fields in BibTeX databases?

Considered Options

- Special fields as separate fields
- Special fields as keywords
- Special fields as values of a special field
- Special fields as sub-feature of groups

Decision Outcome

Chosen option: “Special fields as separate fields”, because comes out best (see below).

Pros and Cons of the Options

Special fields as separate fields

Example:

```
priority = {prio1},
printed = {true},
readstatus = {true},
```

- Good, because groups are another view to fields
- Good, because a special field leads to a special rendering
- Good, because groups pull information from the main table
- Good, because hard-coding presets is easier than generic configuration
- Good, because direct inclusion in main table
- Good, because groups are shown with color bars in the main table
- Good, because there are no “hidden groups” in JabRef
- Good, because can be easily removed (e.g., by a formatter)
- Good, because prepares future power of JabRef to make field properties configurable

- Bad, because bloats BibTeX file
- Bad, because requires more writing when editing BibTeX manually by hand

Special fields as keywords

Example:

```
keywords = {prio1, printed, read}
```

- Good, because does not bloat the BibTeX file. Typically, 50% of the lines are special fields
- Good, because the user can easily assign a special field. E.g, typing “, prio1” into keywords instead of “`\n priority = {prio1},`”
- Bad, because they need to be synchronized to fields (because otherwise, the maintable cannot render it)
- Bad, because keywords are related to the actual content
- Bad, because some users want to keep publisher keywords

Special fields as values of a special field

Example:

```
jabrefspecial = {prio1, printed, red}
```

- Good, because typing effort
- Bad, because handling in table gets complicated → one field is now multiple columns

Special fields as sub-feature of groups

- Good, because one concept rules them all
- Good, because groups already provide explicit handling of certain cases: groups based on keywords and groups based on author’s last names
- Bad, because main table implementation changes: Currently, each column in the main table represents a field. The main may mark entries belonging to certain groups, but does offer an explicit rendering of groups
- Bad, because groups are more a query on data of the entries instead of explicitly assigning entries to a group
- Bad, because explicit assignment and unassigment to a group is not supported by the main table

---

Use Jackson to parse study.yml

Context and Problem Statement

The study definition file is formulated as a YAML document. To access the definition within JabRef this document has to be parsed. What parser should be used to parse YAML files?

Considered Options

- Jackson
- SnakeYAML Engine
- yamlbeans
- eo-yaml
- Self-written parser

Decision Outcome

Chosen option: “Jackson”, because as it is a dedicated library for parsing YAML. `yamlbeans` also seem to be viable. They all offer similar functionality.

Pros and Cons of the Options

Jackson

- Good, because established YAML parser library
- Good, because supports YAML 1.2
- Good, because it can parse LocalDate

SnakeYAML Engine

- Good, because established YAML parser library
- Good, because supports YAML 1.2
- Bad, because cannot parse YAML into Java DTOs, only into basic Java structures, this then has to be assembled into DTOs

yamlbeans

- Good, because established YAML parser library
- Good, because nice getting started page

- Bad, because objects need to be annotated in the yaml file to be parsed into Java objects

eo-yaml

- Good, because established YAML parser library
- Good, because supports YAML 1.2
- Bad, because cannot parse YAML into Java DTOs

Own parser

- Good, because easily customizable
- Bad, because high effort
- Bad, because has to be tested extensively

Links

- Winery’s ADR-0009
- Winery’s ADR-0010

Keep study as a DTO

Context and Problem Statement

The study holds query and library entries that could be replaced respectively with complex query and fetcher instances. This poses the question: should the study remain a pure DTO object, or should it contain direct object instances?

Considered Options

- Keep study as DTO and use transformers
- Replace entries with instances

Decision Outcome

Chosen option: “Keep study as DTO and use transformers”, because comes out best (see below).

Pros and Cons of the Options

Keep study as DTO and use transformers

- Good, because no need for custom serialization
- Good, because deactivated fetchers can be documented (important for traceable Searching (SLRs))
- Bad, because Entries for databases and queries needed

Replace entries with instances

- Good, because no need for database and query entries
- Bad, because custom de-/serializers for fetchers and complex queries needed
- Bad, because harder to maintain than using “vanilla” Jackson de-/serialization
- …

Remove stop words during query transformation

Context and Problem Statement

When querying for a title of a paper, the title might contain stop words such as “a”, “for”, “and”. Some data providers return 0 results when querying for a stop word. When transforming a query to the Lucene syntax, the default Boolean operator `and` is used. When using IEEE, this often leads to zero search results.

Decision Drivers

- Consistent to the Google search engine
- Allow reproducible searches
- Avoid WTFs on the user’s side

Considered Options

- Remove stop words from the query
- Automatically enclose in quotes if no Boolean operator is contained

Decision Outcome

Chosen option: “Remove stop words from the query”, because comes out best.

Pros and Cons of the Options

Remove stop words from the query

- Good, because good search results if no Boolean operators are used
- Bad, because when using complex queries and stop words are used alone, they are silently removed

Automatically enclose in quotes if no Boolean operator is contained

- Good, because good search results if no Boolean operators are used
- Bad, because silently leads to different results
- Bad, because inconsistent to Google behavior

Localized Preferences

Note: This is not implemented yet

Context and Problem Statement

Currently, JabRef uses some localized preferences. Example: The email subject for sending references should be localized. A German user should use “Referenzen”, whereas the English default is “References”. In JabRef 5.x, it is implemented using `defaults.put(EMAIL_SUBJECT, Localization.lang("References"));` and `org.jabref.logic.preferences.JabRefCliPreferences#setLanguageDependentDefaultValues`.

We want to remove the localization-dependency from `JabRefPreferences`. The aim is to move the Localization to where the string is used.

The problems are:

- How to store default values?
- How to know if a user changed the string?
- What happens if the user changes the UI language? (If he configured a string, that should be kept. If the did not configure anything, the string should just change).

Considered Options

- Mark default value with `%`
- Localize defaults
- Store the preference only when it was changed by the user
- Store the unlocalized string

Decision Outcome

Chosen option: “Mark default value with `%`”, because it achieves goals without requiring too much refactoring and reuses a pattern already in use.

Pros and Cons of the Options

Mark default value with `%`

If user stores value, the value is stored as is. The default value is stored with `%` in front. A caller has to localize any stored value in the preferences if the string is ‘escaped’ by `%` as the

first character.

Code: If looked-up string starts with `%`, call `Localization.lang` of the substring (strating from 2nd character). Otherwise, use string as is.

- Good, because clear distinction between default value and user-supplied value.
- Good, because on update of JabRef's defaults, the string is not modified.
- Good, because already used in fxml files to indicate translateable strings.
- Good, because consistent to FXML.

Localize Defaults

Example: `defaults.put(EMAIL_SUBJECT, Localization.lang("References"));` in `JabRefGuiPreferences`.

- Good, because it is the current implementaton
- Bad, because it does not allow for language switching

Store the preference only when it was changed by the user

If the preference was changed by the user, it should be stored. If there is no setting by the user, leave it empty. When a consumer gets such an empty preference, it knows that it needs to read the default and localize it.

- Good, because easy to implement.
- Bad, because this won't work if users actually want something to be empty.

Store the unlocalized string

Consumers then check the string they got as a preference against the defaults. If it matches, localize it. Otherwise, use it.

Reviewdog findings are code reviews

Context and Problem Statement

JabRef offers guidelines to setup the local workspace. There is also a section on JabRef's code style. There are pull requests by newcomers, which do not follow that style guide.

How to quickly provide feedback to contributors that checkstyle was not matched?

Decision Drivers

- Be friendly to newcomers
- Provide fast feedback to contributors
- Lower the workload of maintainers
- Keep maintainers focused on the "real" challanges of the code changes

Considered Options

- Use Reviewdog's PullRequest review reporter
- Use Reviewdog's check reporter
- Use comment-failure-action

Decision Outcome

Chosen option: "Use Reviewdog's PullRequest review reporter", because resolves force to provide fast feedback. We do not want to use `comment-failure-action`, because it might produce too many comments. We accept that newcomers might be annoyed if quick automatic feedback by a bot is given: We value the time of our maintainers and want to keep them focused on the real challanges of the code changes.

Use Java Native Access to Determine Default Directory

Context and Problem Statement

JabRef needs to propose a file directory to a user for storing files. How to determine the “best” directory native for the OS the user runs.

Decision Drivers

- Low maintenance effort
- Follow JabRef’s architectural guidelines
- No additional dependencies

Considered Options

- Use Swing’s FileChooser to Determine Default Directory
- Use `user.home`
- AppDirs
- Java Native Access

Decision Outcome

Chosen option: “Java Native Access”, because comes out best (see below).

Pros and Cons of the Options

Use Swing’s FileChooser to Determine Default Directory

Swing’s FileChooser implemented a very decent directory determination algorithm. It thereby uses `sun.awt.shell.ShellFolder`.

- Good, because provides best results on most platforms.
- Good, because also supports localization of the folder name. E.g., `~/Dokumente` in Germany.
- Bad, because introduces a dependency on Swing and thereby contradicts the second decision driver.
- Bad, because GraalVM’s support Swing is experimental.

- Bad, because handles localization only on Windows.

Use `user.home`

There is `System.getProperty("user.home");`.

- Bad, because “The concept of a HOME directory seems to be a bit vague when it comes to Windows”. See https://stackoverflow.com/a/586917/873282 for details.
- Bad, because it does not include `Documents`: As of 2022, `System.getProperty("user.home")` returns `c:\Users\USERNAME` on Windows 10, whereas `FileSystemView` returns `C:\Users\USERNAME\Documents`, which is the “better” directory.

AppDirs

> AppDirs is a small java library which provides a path to the platform dependent special folder/directory.

- Good, because already used in JabRef.
- Bad, because does not use `Documents` on Windows, but rather `C:\Users\<Account>\AppData\<AppAuthor>\<AppName>` as basis.

Java Native Access

- Good, because no additional dependency required, as it is already loaded by AppDirs.
- Good, because it is well maintained and widely used.
- Good, because it provides direct access to `Documents` and other system variables.

Synchronization with remote databases

Context and Problem Statement

Synchronize the data in a library to a remote database, while handling conflicts and supporting offline-first paradigm.

Decision Drivers

- Updates from the remote should be pulled in
- No updates should get lost
- Easy to implement
- Easy to maintain

Considered Options

- “Optimistic offline lock” with hashes for local file support
- Algorithm based on “optimistic offline lock”
- Use CRDTs

Decision Outcome

Chosen option: “‘Optimistic offline lock’ with hashes for local file support”, because simplest option to resolves all forces.

Pros and Cons of the Options

“Optimistic offline lock” with hashes for local file support

The Optimistic Offline Lock is good for synchronizing clients with a server when there is no other modification of data on client side. However, users might modify the `.bib` file external of JabRef. They might also open an existing `.bib` file and synchronize that. Thus, there are additions needed to handle the local synchronization.

Moreover, the optimistic offline lock does not say how a set of data is synchronized.

Both shortcomings are resolved by our algotihm. This algorithm is described at Remote JabDrive storage.

Algorithm based on “optimistic offline lock”

Optimistic Offline Lock is a well-established technique to prevent conflicts in concurrent business transactions. It assumes that the chance of conflict is low. Implementation details are found at https://www.baeldung.com/cs/offline-concurrency-control.

This is implemented for the SQL database synchronization, which is described at Remote SQL Storage.

- Good, because this algorithm is already in place since 2016 for JabRef synchronizing with a PostgreSQL backend and a MySQL backend.
- Bad, because it assumes the client to be online 100% and does not have handlings of cases where the client disconnects and alters data in other ways.

Use CRDTs

See https://automerge.org/blog/automerge-2/ for details.

- Bad, because one needs to locally store a lot more metadata (e.g. for operational CRDTs you essentially need to have the full history of all edits). So you would need another file next to the bib file to store these.
- Bad, because CRDTs are mainly used when you need low latency and high frequency of edits (e.g. multi-user chat or text editing). Not really something we care about.

Return BibTeX string and CSL Item JSON in the API

Context and Problem Statement

In the context of an http server, when a http client `GETs` a JSON data structure containing BibTeX data, which format should that have?

Considered Options

- Offer both, BibTeX string and CSL JSON
- Return BibTeX as is as string
- Convert BibTeX to JSON

Decision Outcome

Chosen option: “Offer both, BibTeX string and CSL JSON”, because there are many browser libraries out there being able to parse BibTeX. Thus, we don’t need to convert it.

Pros and Cons of the Options

Offer both, BibTeX string and CSL JSON

- Good, because this follows “Backend for Frontend”
- Good, because Word Addin works seamless with the data provided (and does not need another dependency)
- Good, because other clients can work with BibTeX data
- Bad, because two serializations have to be kept

Return BibTeX as is as string

- Good, because we don’t need to think about any conversion
- Bad, because it is unclear how to ship BibTeX data where the entry is dependent on
- Bad, because client needs an additional parsing logic

Convert BibTeX to JSON

More thought has to be done when converting to JSON. There seems to be a JSON format from @citation-js/plugin-bibtex. We could do an additional self-made JSON format, but this increases

the number of available JSON serializations for BibTeX.

- Good, because it could flatten BibTeX data (example: `author = first # " and " # second`)
- Bad, because conversion is difficult in BibTeX special cases. For instance, if Strings are used (example: `author = first # " and " # second`) and one doesn't want to flatten ("normalize") this.

More Information

Existing JavaScript BibTeX libraries:

- bibtex-js
- bibtexParseJS
- @citation-js/plugin-bibtex

Exporting multiple entries to CFF

Context and Problem Statement

The need for an exporter to CFF format raised the following issue: How to export multiple entries at once? Citation-File-Format is intended to make software and datasets citable. It should contain one "main" entry of type `software` or `dataset`, a possible preferred citation and/or several references of any type.

Decision Drivers

- Make exported files compatible with official CFF tools
- Make exporting process logical for users

Considered Options

- When exporting:
  - Export non-`software` entries with dummy topmost `sofware` and entries as `preferred-citation`
  - Export non-`software` entries with dummy topmost `sofware` and entries as `references`
  - Forbid exporting multiple entries at once
  - Forbid exporting more than one software entry at once
  - Export entries in several files (i.e. one / file)
  - Export several `software` entries with one of them topmost and all others as `references`
- Export several `software` entries with a dummy topmost `software` element and all others as `references`
- When importing:
  - Only create one entry / file, enven if there is a `preferred-citation` or `references`
  - Add a JabRef `cites` relation from `software` entry to its `preferred-citation`
  - Add a JabRef `cites` relation from `preferred-citation` entry to the main `software` entry
  - Separate `software` entries from their `preferred-citation` or `references`

Decision Outcome

The decision outcome is the following.

- When exporting, JabRef will have a different behavior depending on entries type.

    - If multiple non-`software` entries are selected, then exporter uses the `references` field with a dummy topmost `software` element.
    - If several entries including a `software` or `dataset` one are selected, then exporter uses this one as topmost element and the others as `references`, adding a potential `preferred-citation` for the potential `cites` element of the topmost `software` entry.
    - If several entries including several `software` ones are selected, then exporter uses a dummy topmost element, and selected entries are exported as `references`. The `cites` or `related` fields won't be exported in this case.
    - JabRef will not handle `cites` or `related` fields for non-`software` elements.
- When importing, JabRef will create several entries: one main entry for the `software` and other entries for the potential `preferred-citation` and `references` fields. JabRef will link main entry to the preferred citation using a `cites` from the main entry, and wil link main entry to the references using a `related` from the main entry.

Positive Consequences

- Exported results comply with CFF format
- The export process is "logic" : an user who exports multiple files to CFF might find it clear that they are all marked as `references`
- Importing a CFF file and then exporting the "main" (software) created entry is consistent and will produce the same result

Negative Consequences

- Importing a CFF file and then exporting one of the `preferred-citation` or the `references` created entries won't result in the same file (i.e exported file will contain a dummy topmost `software` instead of the actual `software` that was imported)
- `cites` and `related` fields of non-`software` entries are not supported

Use Apache Commons IO for directory monitoring

Context and Problem Statement

In JabRef, there is a need to add a directory monitor that will listen for changes in a specified directory.

Currently, the monitor is used to automatically update the LaTeX Citations when a LaTeX file in the LaTeX directory is created, removed, or modified (#10585). Additionally, this monitor will be used to create a dynamic group that mirrors the file system structure (#10930).

Considered Options

- Use java.nio.file.WatchService
- Use io.methvin.watcher.DirectoryWatcher
- Use org.apache.commons.io.monitor

Decision Outcome

Chosen option: “Use org.apache.commons.io.monitor”, because comes out best (see below).

Pros and Cons of the Options

java.nio.file.WatchService

- Good, because it is a standard Java API for watching directories.
- Good, because it does not need polling, it is event-based for most operating systems.
- Bad, because:
  1. Does not detect files coming together with a new folder (JDK issue: JDK-8162948).
  2. Deleting a subdirectory does not detect deleted files in that directory.
  3. Access denied when trying to delete the recursively watched directory on Windows (JDK issue: JDK-6972833).
  4. Implemented on macOS by the generic `PollingWatchService`. (JDK issue: JDK-8293067)

io.methvin.watcher.DirectoryWatcher

- Good, because it implemented on top of the `java.nio.file.WatchService`, which is a standard Java API for watching directories.
- Good, because it resolves some of the issues of the `java.nio.file.WatchService`.
  - Uses `ExtendedWatchEventModifier.FILE_TREE` on Windows, which resolves issues (1, 3) of the `java.nio.file.WatchService`.
  - On macOS have native implementation based on the Carbon File System Events API, this resolves issue (4) of the `java.nio.file.WatchService`.
- Bad, because issue (2) of the `java.nio.file.WatchService` is not resolved.

org.apache.commons.io.monitor

- Good, because there are no observed issues.
- Good, because can handle huge amount of files without overflowing.
- Bad, because it uses a polling mechanism at fixed intervals, which can waste CPU cycles if no change occurs.

Use currently active tab in Select style (OO Panel) to decide style type

Context and Problem Statement

In the Select Style Dialog window of the OpenOffice Panel, in case a style is selected in both the CSL Styles Tab and JStyles Tab, how to decide which of the two will be activated for use?

Considered Options

- Use toggle in Select Style GUI
- Use toggle in Preferences
- Use Buttons in Select Style GUI
- Use Toggle in Main GUI
- Use currently active Tab in Select Style GUI and add a notification

Decision Outcome

Chosen option: “Use currently active Tab in Select Style GUI and add a notification”, because we already had two tabs indicating a clear separation of choices to the user. It was the most convenient way without adding extra steps to make the user choose “which style type to use” before selecting the style, which would be the case in the other options if chosen. The option is quite intuitive, extensible for working with multiple tabs and make only three to four clicks are necessary to select a style. Furthermore, the notification makes it clear to the user which style type as well as which style is selected.

Store Chats Alongside Database

Context and Problem Statement

Chats with AI should be stored somewhere. But where and how?

Considered Options

- Inside `.bib` file
- In local user folder
- Alongside `.bib` file

Decision Drivers

- Should work when shared with OneDrive, Dropbox or similar asynchronous services
- Should work on network drives
- Should be “easy” for users to follow
- Should be the same in a shared and non-shared setting (e.g., if Dropbox is used or not should make a difference)

Decision Outcome

Chosen option: “In local user folder”, because it’s very hard to work with a shared library, if two users will work simultaneously on one library, then AI chats file will be absolutely arbitrary and unmergable.

Pros and Cons of the Options

Inside `.bib` file

- Good, because we already have a machinery for managing the fields and other information of BIB entries
- Good, because chats are stored inside one file, and if the `.bib` file is moved, the chat history is preserved
- Bad, because there may be lots of chats and messages and `.bib` file become too cluttered and too big which slows down the processing of `.bib` file
- Bad, because if user shares a `.bib` file, they will also share chat messages, but chats are not ideal, so user may not want to share them

In local user folder

One can use `%APPDATA%`, where JabRef stores the Lucene index and other information. See `org.jabref.gui.desktop.os.NativeDesktop#getFulltextIndexBaseDirectory` for use in JabRef and https://github.com/harawata/appdirs for general information.

Concrete example for backup folder: `C:\Users\${username}\AppData\Local\org.jabref\jabref\backups`. Example filename: `4a070cf3--Chocolate.bib--2024-03-25--14.20.12.bak`.

- Good, because `.bib` file is kept clean
- Good, because chat messages are saved locally
- Neutral, because may be a little harder to implement
- Bad, because chat messages cannot be easily shared
- Bad, because when path of a `.bib` file is changed, the chats are lost

Alongside `.bib` file

- Good, because simple implementation
- Good, because, the user can send the chats file alongside the `.bib` file if they want to share the chats. If users do not want to share the messages, then they can omit the chats file
- Good, because `.bib` files is kept clean
- Bad, because user may not expect that a new file will be created alongside their `.bib` (or other LaTeX-related) files
- Bad, because, it may be not convenient to share both files (`.bib` file and chats file) in order to share chat history.
- Bad, because if `.bib` files are edited externally (meaning, not inside the JabRef), then chats file will not be updated correspondingly
- Bad, because if user moves `.bib` file, they should move the chats file too
- Bad, because if two persons work in parallel using a OneDrive share, the file is overwritten or a conflict file is generated. (Dropbox “conflicted copy”)

Store Chats in MVStore

Context and Problem Statement

This is a follow-up to ADR-032.

The chats with AI should be saved on exit from JabRef and retrieved on launch. We need to decide the format of the serialized messages.

Decision Drivers

- Easy to implement and maintain
- Memory-efficient (because JabRef is said to consume much memory)

Considered Options

- JSON
- MVStore
- Custom format

Decision Outcome

Chosen option: “MVStore”, because it is simple and memory-efficient.

Pros and Cons of the Options

JSON

- Good, because allows for easy storing and loading of chats
- Good, because cross-platform
- Good, because widely used and accepted, so there are lots of libraries for JSON format
- Good, because it is even possible to reuse the chats file for other purposes
- Good, because has potential for being mergeable by external tooling
- Bad, because too verbose (meaning the file size could be much smaller)

MVStore

- Good, because automatic loading and saving to disk
- Good, because memory-efficient

- Bad, because does not support mutable values in maps.
- Bad, because the order of messages need to be “hand-crafted” (e.g., by mapping from an Integer to the concrete message), since MVStore does not support storing list which update.
- Bad, because it stores data as key-values, but not as a custom data type (like tables in RDBMS)

Custom format

- Good, because we have the full control
- Bad, because involves writing our own language and parser
- Bad, because we need to implement optimizations found in databases on our own (storing some data in RAM, other on disk)

Use Citation Key for Grouping Chat Messages

Context and Problem Statement

As we store chat messages not inside a BibTeX entry in `.bib` file, the chats file is represented as a map to BibTeX entry and a list of messages. We need to specify the key of this map. Turns out, it is not that easy.

Decision Drivers

- The key should exist for every BibTeX entry
- The key should be unique along other BibTeX entries in one library file
- It is assumed that the key does not change at run-time, between launches of JabRef, and should be cross-platform (most important)

Considered Options

- `BibEntry` Java object
- `BibEntry`'s `id`
- `BibEntry`'s citation key
- `BibEntry`'s `ShareId`

Decision Outcome

Chosen option: "`BibEntry`'s citation key", because this is the only choice that complies to the third point in Decision Drivers.

Positive Consequences

- Easy to implement
- Cross-platform

Negative Consequences

- If the citation key is changed externally, then the chats file becomes out-of-sync
- Additional user interaction in order to make the citation key complain the first and second points of Decision Drivers

Pros and Cons of the Options

`BibEntry` Java object

Very bad, because it works only at run-time and is not stable.

`BibEntry`'s `id`

JabRef stores a unique identifier for each `BibEntry`. This identifier is created on each load of a library (and not stored permanently).

Very bad, for the same reasons as `BibEntry` Java object.

`BibEntry`'s citation key

- Good, because it is cross-platform, stable (meaning stays the same across launches of JabRef)
- Bad, because it is not guaranteed that citation key exists on `BibEntry`, and that it is unique across other BibTeX entries in the library

`BibEntry`'s `ShareId`

ADR-0027 describes the procedure of synchronization of a Bib(La)TeX library with a server. Thereby, also local and remote entries need to be kept consistent. The solution chosen there is that the **server** creates a UUID for each entry.

This approach cannot be used here, because there is no server running which we can ask for an UUID of an entry.

More Information

Refer to issue #160 in JabRef main repository

ADR-038 takes another option, because it re-generates the index at each start of JabRef.

Generate Embeddings Online

Context and Problem Statement

In order to perform a question and answering (Q&A) session over research papers with large language model (LLM), we need to process each file: each file should be converted to string, then this string is split into chunks, and for each chunk an embedding vector should be generated.

Where these embeddings should be generated?

Considered Options

- Local embedding model with `langchain4j`
- OpenAI embedding API

Decision Drivers

- Embedding generation should be fast
- Embeddings should have good performance (performance mean they “catch the semantics” good, see also MTEB)
- Generating embeddings should be cheap
- Embeddings should not be of a big size
- Embedding models and library to generate embeddings shouldn’t be big in distribution binary.

Decision Outcome

Chosen option: “OpenAI embedding API”, because the distribution size of JabRef will be nearly unaffected. Also, it’s fast and has a better performance, in comparison to available in `langchain4j`’s model `all-MiniLM-L6-v2`.

Pros and Cons of the Options

Local embedding model with `langchain4j`

- Good, because works locally, privacy saved, no Internet connection is required
- Good, because user doesn’t pay for anything

- Neutral, because how fast embedding generation is depends on chosen model. It may be small and fast, or big and time-consuming
- Neutral, because local embedding models may have less performance than OpenAI's (for example). *Actually, most embedding models suitable for use in JabRef are about ~50% performant)
- Bad, because embedding generation takes computer resources
- Bad, because the only framework to run embedding models in Java is ONNX, and it's very heavy in distribution binary

OpenAI embedding API

- Good, because we delegate the task of generating embeddings to an online service, so the user's computer is free to do some other job
- Good, because OpenAI models have typically have better performance
- Good, because JabRef distribution size will practically be unaffected
- Bad, because user should agree to send data to a third-party service, Internet connection is required
- Bad, because user pay for embedding generation (see also OpenAI embedding models pricing)

Use `TextArea` for Chat Message Content

Context and Problem Statement

This decision record concerns the UI component that is used for rendering the content of chat messages.

Decision Drivers

- Looks good (renders Markdown)
- User can select and copy text
- Has good performance

Considered Options

- Use `TextArea`
- Use a third-party package
- Use a Markdown parser and convert AST nodes to JavaFX TextFlow elements
- Use a Markdown parser to convert content into HTML and use a WebView for one message
- Use a Markdown parser and WebView for the whole chat history

Decision Outcome

Chosen option: “Use `TextArea`”. All other options require more time to implement. Some of the options do not support text selection and copying, which for now we value more than Markdown rendering.

Pros and Cons of the Options

Use TextArea

- Good, because it is easy to implement
- Good, because it supports text selection and copying
- Bad, because it does not offer rich text. Thus, Markdown can only be displayed in a plain text form.
- Bad, because default JavaFX’s `TextArea` shrinks

Use a third-party package

There seems to be only one package for JavaFX that provides a ready-to-use UI node for Markdown rendering.

- Good, because it is easy to implement
- Good, because it renders Markdown
- Good, because it renders Markdown to JavaFX nodes (does not use a `WebView`)
- Good, because complex elements from Markdown are supported (tables, code blocks, etc.)
- Bad, because it has very strange issues and architectural flaws with styling
- Bad, because it does not support text selection and copying (because of underlying JavaFX `Text` nodes)

Use a Markdown parser and convert AST nodes to JavaFX TextFlow elements

- Good, because we will support Markdown
- Good, because no need to write a Markdown parser from scratch
- Good, because does not use a WebView
- Good, because easy styling
- Bad, because we need some time to implement Markdown AST -> JavaFX nodes converter
- Bad, because rendering tables and code blocks may be hard
- Bad, because it will not support text selection and copying

Use a Markdown parser to convert content into HTML and use a WebView for one message

- Good, because there are libraries to convert Markdown to HTML
- Good, because may be easier to implement than other choices (except `TextArea`)
- Good, because it supports text selection and copying
- Bad, because it may be a problem to connect JavaFX CSS to `WebView`
- Bad, because one `WebView` for one message is resourceful

Use a Markdown parser and WebView for the whole chat history

- Good, because there are libraries to convert Markdown to HTML
- Good, because it supports text selection and copying
- Bad, because it may be a problem to connect JavaFX CSS to `WebView`
- Bad, because it may be a problem to correctly communicate with Java code and `WebView` to add new messages

More Information

This ADR is highly linked to ADR-0042.

Actually we used an `ExpandingTextArea` from `GemsFX` package so the content can occupy as much space as it needs in the `ScrollPane`.

About the selection and copying, this goes down to fundamental issue from JavaFX. `Text` and `Label` as a whole or a part cannot be selected and/or copied.

RAG Architecture Implementation

Context and Problem Statement

The current trend in questions and answering (Q&A) using large language models (LLMs) or other AI related technology is retrieval-augmented-generation (RAG).

RAG is related to Open Generative QA that means LLM (which generates text) is supplied with context (chunks of information extracted from various sources) and then it generates answer.

RAG architecture consists of these steps (simplified):

How source data is processed:

1. **Indexing**: application is supplied with information sources (PDFs, text files, web pages, etc.)
2. **Conversion**: files are converted to string (because LLM works on text data).
3. **Splitting**: the string from previous step is split into parts (because LLM has fixed context window, meaning it cannot handle big documents).
4. **Embedding generation**: a vector consisting of float values is generated out of chunks. This vector represents meaning of text and the main propety of such vectors is that chunks with similar meaning has vectors that are close to. Generation of such a vector is achieved by using a separate model called *embedding model*.
5. **Store**: chunks with relevant metadata (for example, from which document they were generated) and embedding vector are stored in a vector database.

How answer is generated:

1. **Ask**: user asks AI a question.
2. **Question embedding**: an embedding model generates embedding vector of a query.
3. **Data finding**: vector database performs search of most relevant pieces of information (a finite count of pieces). That’s performed by vector similarity: meaning how close are chunk vector with question vector.
4. **Prompt generation**: using a prompt template the user question is *augmented* with found information. Found information is not generally supplied to user, as it may seem strange that a user asked a question that was already supplied with found information. These pieces of text can be either totally ignored or showed separately in UI tab “Sources”.
5. **LLM generation**: LLM generates output.

This ADR concerns about implementation of this architecture.

Decision Drivers

- Prefer good and maintained libraries over self-made solutions for better quality.
- The usage of framework should be easy. It would seem strange when user wants to download a BIB editor, but they are required to install some separate software (or even Python runtime).
- RAG shouldn't provide any additional money costs. Users should pay only for LLM generation.

Considered Options

- Use a hand-crafted RAG
- Use a third-party Java library
- Use a standalone application
- Use an online service

Decision Outcome

Chosen option: mix of "Use a hand-crafted RAG" and "Use a third-party Java library".

Third-party libraries provide excellent resources for connecting to an LLM or extracting text from PDF files. For RAG, we mostly used all the machinery provided by `langchain4j`, but there were moments that should be hand-crafted:

- **LLM connection**: due to https://github.com/langchain4j/langchain4j/issues/1454 (https://github.com/InAnYan/jabref/issues/77) this was delegated to another library `jvm-openai`.
- **Embedding generation**: due to https://github.com/langchain4j/langchain4j/issues/1492 (https://github.com/InAnYan/jabref/issues/79), this was delegated to another library `djl`.
- **Indexing**: `langchain4j` is just a bunch of useful tools, but we still have to orchestrate when indexing should happen and what files should be processed.
- **Vector database**: there seems to be no embedded vector database (except SQLite with `sqlite-vss` extension). We implemented vector database using `MVStore` because that was easy.

Pros and Cons of the Options

Use a hand-crafted RAG

- Good, because we have the full control over generation
- Good, because extendable
- Bad, because LLM connection, embedding models, vector storage, and file conversion should be implemented manually

- Bad, because it’s hard to make a complex RAG architecture

Use a third-party Java library

- Good, because provides well-tested and maintained tools
- Good, because libraries have many LLM integrations, as well as embedding models, vector storage, and file conversion tools
- Good, because they provide complex RAG pipelines and extensions
- Neutral, because they provide many tools and functions, but they should be orchestrated in a real application
- Bad, because some of them are raw and undocumented
- Bad, because they are all similar to `langchain`
- Bad, because they may have bugs

Use a standalone application

- Good, because they provide complex RAG pipelines and extensions
- Good, because no additional code is required (except connecting to API)
- Neutral, because they provide not that many LLM integrations, embedding models, and vector storages
- Bad, because a standalone app running is required. Users may be required to set it up properly
- Bad, because the internal working of app is hidden. Additional agreement to Privacy or Terms of Service is needed
- Bad, because hard to extend

Use an online service

- Good, because all data is processed and stored not on the user’s machine: faster and no memory is used.
- Good, because they provide complex RAG pipelines and extensions
- Good, because no additional code is required (except connecting to API)
- Neutral, because they provide not that many LLM integrations, embedding models, and vector storages
- Bad, because requires connection to Internet
- Bad, because data is processed by a third party company
- Bad, because most of them require additional payment (in fact, it would be impossible to develop a free service like that)

Use `BibEntry.getId` for BibEntries at Indexing

Context and Problem Statement

The `BibEntry` class has `equals` and `hashCode` implemented on the content of the bib entry. Thus, if two bib entries have the same type, the same fields, and the same content, they are equal.

This, however, is not useful in the UI, where equal entries are not the same entries.

Decision Drivers

- Simple code
- Not changing much other JabRef code
- Working Lucene

Considered Options

- Use `BibEntry.getId` for indexing `BibEntry`
- Use `System.identityHashCode` for indexing `BibEntry`
- Rewrite `BibEntry` logic

Decision Outcome

Chosen option: “Use `BibEntry.getId` for indexing `BibEntry`”, because is the “natural” thing to ensure distinction between two instances of a `BibEntry` object - regardless of equality.

Use Apache Velocity as template engine

Context and Problem Statement

We need to choose a template engine for custom export filters and AI features.

A discussion of template engines was also in one of the JabRef repos.

A discussion was raised on StackOverflow "Velocity vs. FreeMarker vs. Thymeleaf".

Decision Drivers

- It should be fast.
- It should be possible to provide templates out of `String`s (required by the AI feature).
- It should have short and understandable syntax. Especially, it should work well with unset fields and empty `Optional`s.

Considered Options

- Apache Velocity
- Apache FreeMarker
- Thymeleaf

Decision Outcome

Chosen option: "Apache Velocity", because "Velocity's goal is to keep templates as simple as possible" (source). It is sufficient for our use case. Furthermore, Apache Velocity is lightweight, and it allows to generate text output. This is a good fit for the AI feature.

Pros and Cons of the Options

Apache Velocity

- Main page: https://velocity.apache.org/.
- User guide: https://velocity.apache.org/engine/devel/user-guide.html.
- Developer guide: https://velocity.apache.org/engine/devel/developer-guide.html.

Example:

```
You are an AI assistant that analyses research papers. You answer questions about papers.

Here are the papers you are analyzing:
#foreach( $entry in $entries )
${CanonicalBibEntry.getCanonicalRepresentation($entry)}
#end
```

- Good, because supports plain text templating.
- Good, because it is possible to use `String` as a template.
- Good, because it has simple syntax, and it is designed for simple template workflows.
- Good, because it has a stable syntax (source).
- Bad, because it is in maintenance mode.
- Bad, because removed from Spring 5.0.1

Apache FreeMarker

- Main page: https://freemarker.apache.org/index.html.
- User guide: https://freemarker.apache.org/docs/dgui.html.
- Developer guide: https://freemarker.apache.org/docs/pgui_quickstart.html.

Example:

```
You are an AI assistant that analyzes research papers. You answer questions about papers.

Here are the papers you are analyzing:
<#list entries as entry>
${CanonicalBibEntry.getCanonicalRepresentation(entry)}
</#list>
```

- Good, because supports plain text templating.
- Good, because it is possible to use `String` as a template.
- Good, because in active development.
- Good, because it is powerful and flexible.
- Good, because it has extensive documentation (source).
- Neutral, because it has received some API and syntax changes recently (source).
- Neutral, because FreeMarker is used for complex template workflow, which we do not need in JabRef.

Thymeleaf

- Main page: https://www.thymeleaf.org/.
- Documentation: https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html.

Example:

```
You are an AI assistant that analyzes research papers. You answer questions about papers.


Here are the papers you are analyzing:
[# th:each="entry : ${entries}"]
[(${CanonicalBibEntry.getCanonicalRepresentation(entry)})]
[/]
```

- Good, because supports plain text templating.
- Good, because it is possible to use `String` as a template.
- Good, because it has several template modes, that helps to make HTML, XML, and other templates.
- Good, because it is powerful and flexible.
- Neutral, because the API is a bit more complex than the other options.
- Bad, because the syntax is more complex than the other options. Especially for text output.

More Information

As stated in the template discussion issue, we should choose a template engine, and then slowly migrate previous code and templates to the chosen engine.

Other template engines are discussed at https://www.baeldung.com/spring-template-engines, especially `#other-template-engines`. We did not find any other engine there worth switching to.

Display front cover for book citations in the Preview tab

Context and Problem Statement

- Users have requested that the front covers of book citations are displayed in JabRef.
- This is discussed on the JabRef forum and raised as a feature request.
- We need to decide where the book cover should be placed.

Decision Drivers

- It should not be obtrusive or distracting since the main use of JabRef is for articles not books.
- It should not obstruct the view of existing GUI components, specifically the MainTable or the information in the EntryEditor's tabs.

Considered Options

Place the book cover in:

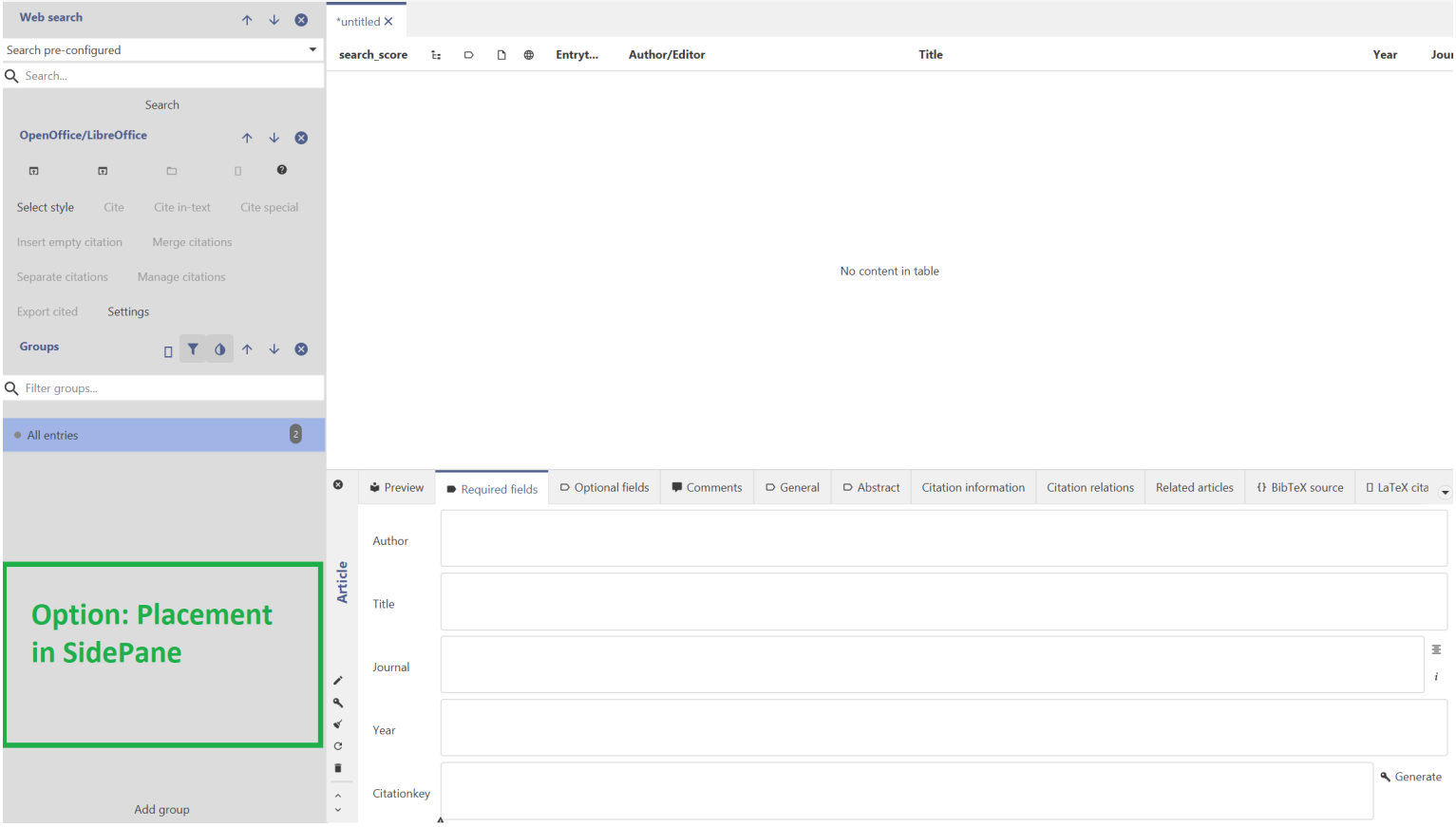
1. The existing SidePane
2. A new SidePane
3. The Preview panel of the EntryEditor
4. A SplitPane next to the MainTable

Decision Outcome

Chosen option: "The PreviewPanel of the EntryEditor".

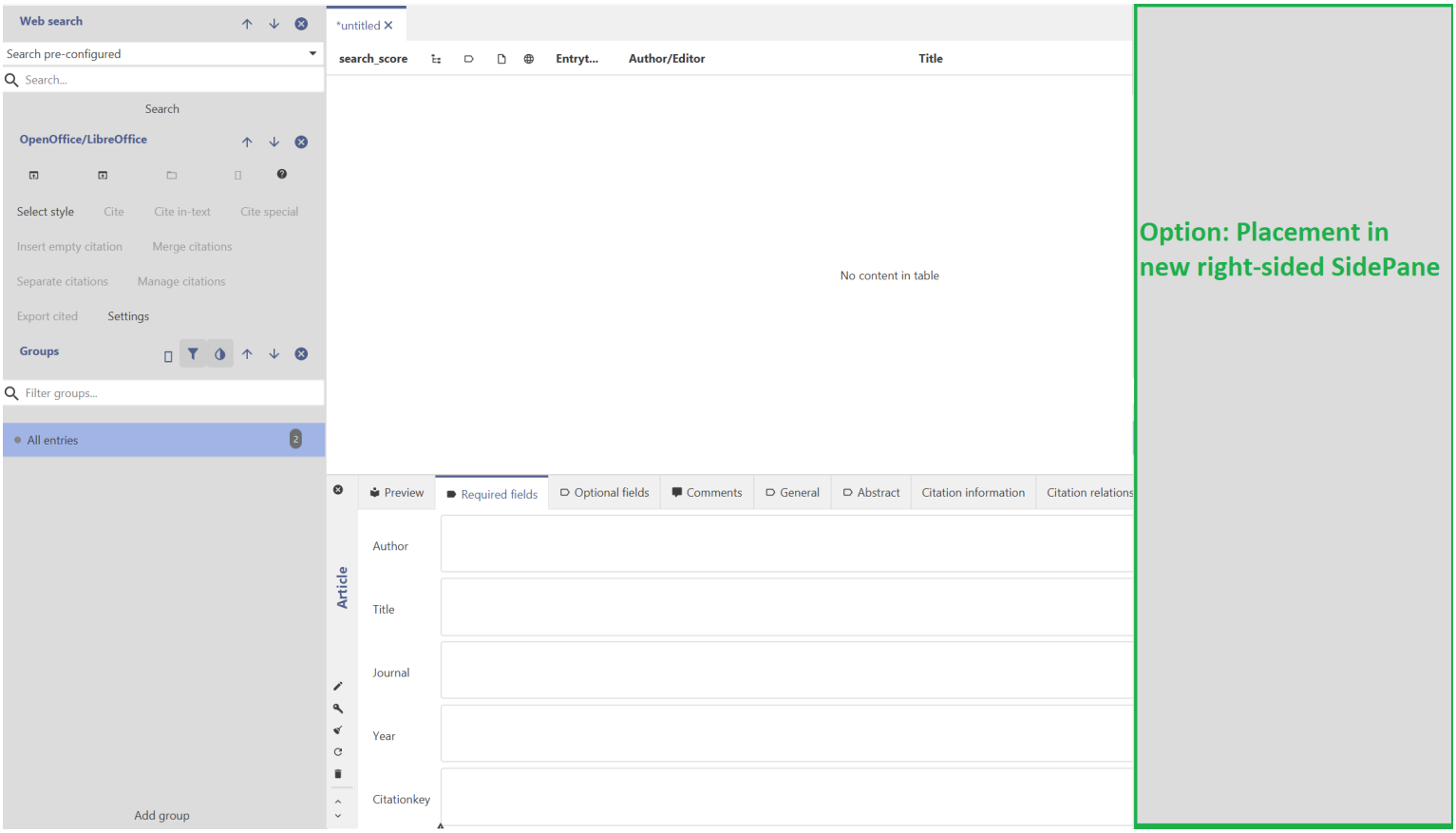
Pros and Cons of the Options

Existing SidePane



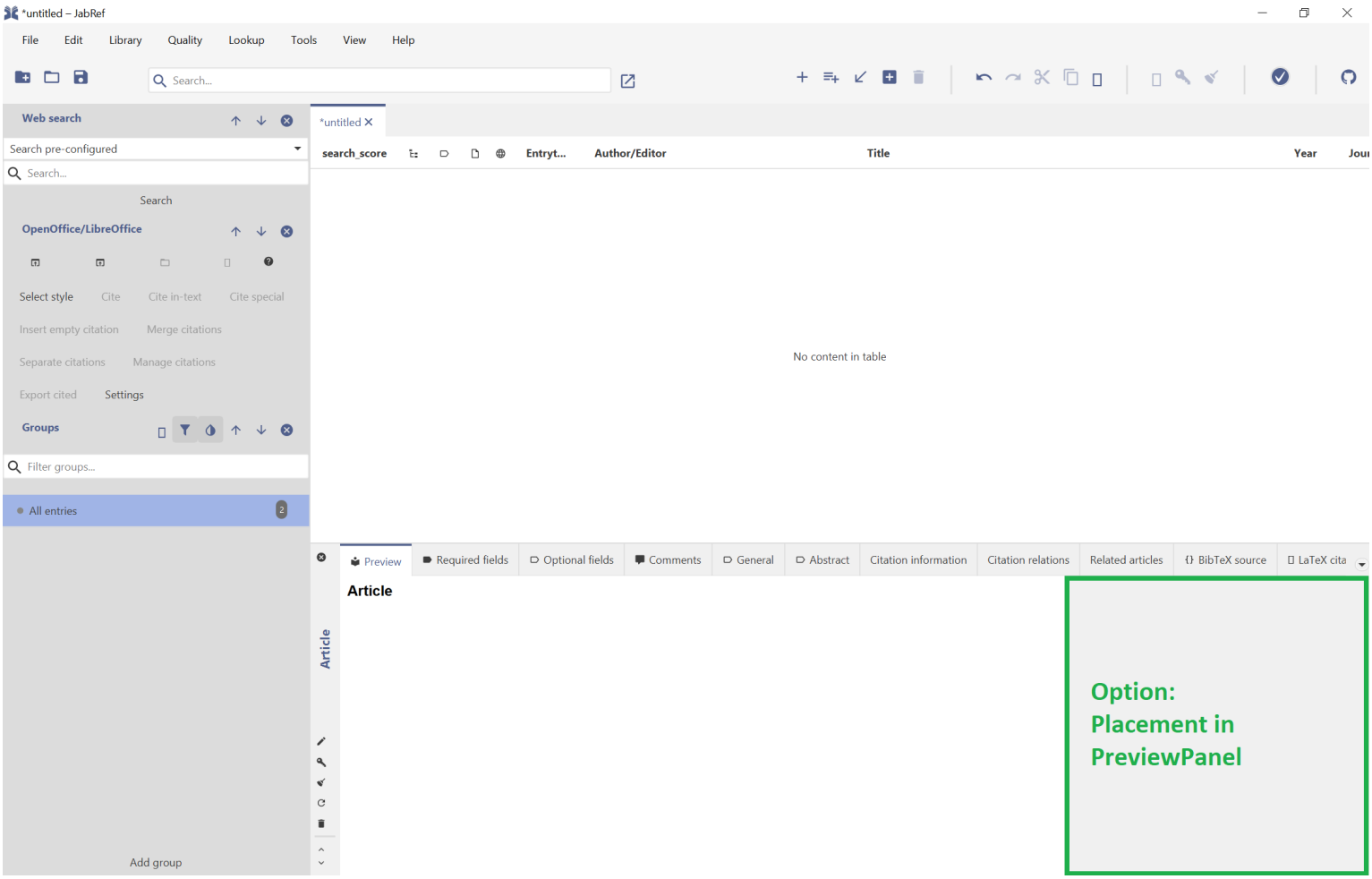
- Good, because it would be unobtrusive
- Bad, because it would crowd other panels in the SidePane
- Bad, because changing the size of the SidePane would affect both the MainTable and the EntryEditor.

New right-sided SidePane



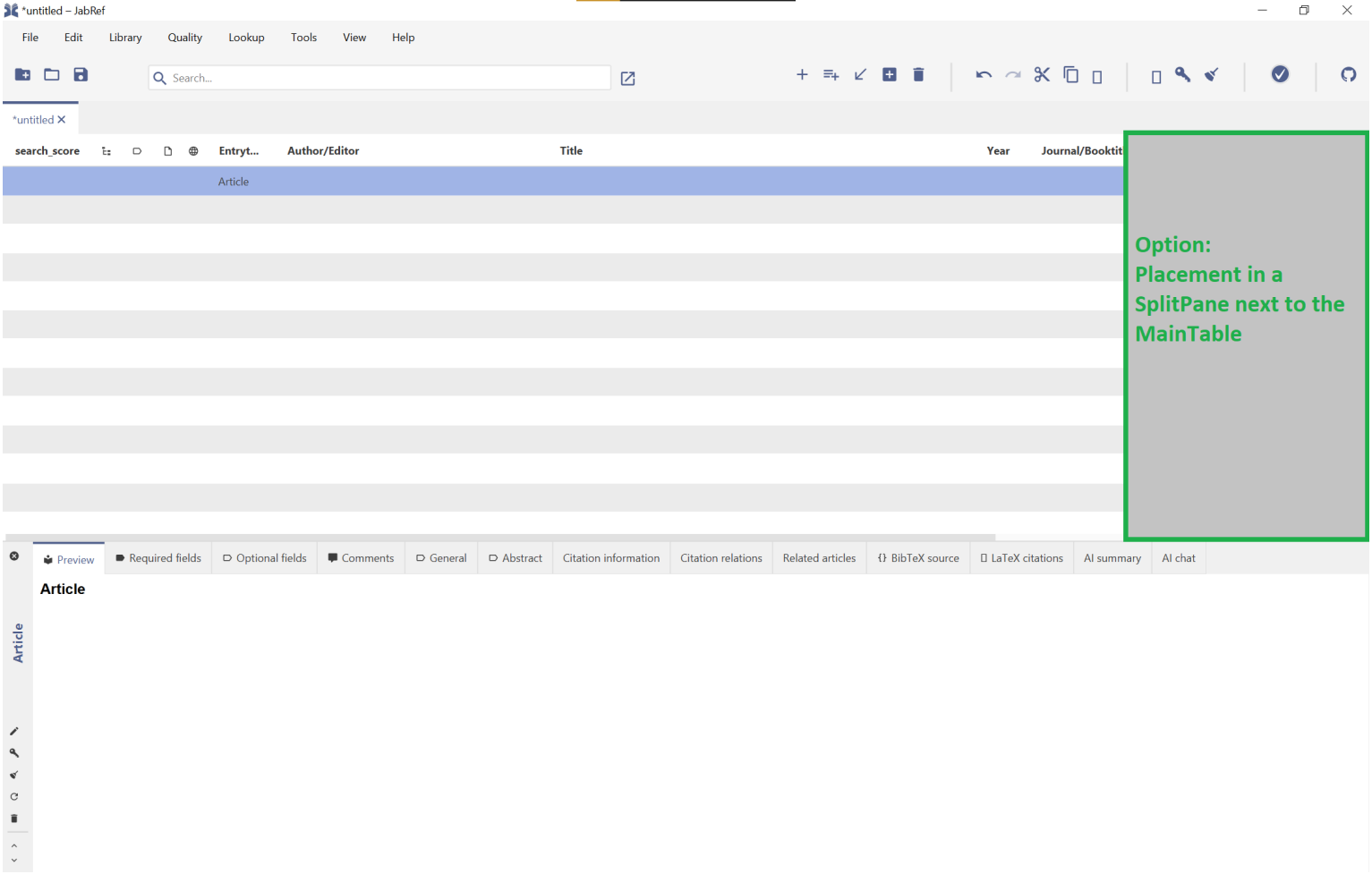
- Good, if integrated together with entry preview because it would make it easier to view a citation’s preview.
- Bad, because an extra SidePane would make the interface overly complex.

The PreviewPanel of the EntryEditor



- Good, because it would not be obtrusive or distracting.
- Bad, if the Entry Editor is closed, users will have to open the Entry Editor and navigate to the “Preview” or “Required fields” tab to see the cover.

SplitPane next to the MainTable



- Good, because changing the size of this SplitPane would only affect the MainTable.
- Bad, because it would obstruct some columns in the MainTable.

Use one language string for pluralization localization

Context and Problem Statement

For user-facing messages, sometimes, it needs to be counted: E.g., 1 entry updated, 2 entries updated, etc.

In some languages, there is not only “one” and “more than one”, but other forms:

- zero → “لم نزرع أي شجرة حتى الآن”
- one → “لقد زرعنا شجرة ١ حتى الآن”
- two → “لقد زرعنا شجرتين ٢ حتى الآن”
- few → “لقد زرعنا ٣ شجرات حتى الآن”
- many → “لقد زرعنا ١١ شجرة حتى الآن”
- other → “لقد زرعنا ١٠٠ شجرة حتى الآن”

(Example is from Pluralization: A Guide to Localizing Plurals)

How to localize pluralization?

Decision Drivers

- Good English language
- Good localization to other languages

Considered Options

- Use one language string for pluralization (no explicit pluralization)
- Use singular and plural
- Handling of multiple forms

Decision Outcome

Chosen option: “Use one form only (no explicit pluralization)”, because it is the most easiest to handle in the code.

Pros and Cons of the Options

Use one language string for pluralization (no explicit pluralization)

Example:

- `Imported 0 entry(s)`
- `Imported 1 entry(s)`
- `Imported 12 entry(s)`

There are sub alternatives here:

- `Imported %0 entry(ies)`.
- `Number of entries imported: %0` (always use “other” plural form)

These arguments are for the general case of using a single text for all kinds of numbers:

- Good, because easy to handle in the code
- Bad, because reads strange in English UI

Use singular and plural

Example:

- `Imported 0 entries`
- `Imported 1 entry`
- `Imported 12 entries`

- Good, because reads well in English
- Bad, because all localizations need to take an `if` check for the count
- Bad, because Arabic not localized properly

Handling of multiple forms

Example:

- `Imported 0 entries`
- `Imported 1 entry`
- `Imported 12 entries`

Code: `Localization.lang("Imported %0 entries", "Imported %0 entry.", "Imported %0 entries.", "Imported %0 entries.", "Imported %0 entries.", "Imported %0 entries.", count)`

- Good, because reads well in English
- Bad, because sophisticated localization handling is required
- Bad, because no Java library for handling pluralization is known
- Bad, because Arabic not localized properly

More Information

- Pluralization: A Guide to Localizing Plurals
- Language Plural Rules
- Unicode CLDR Project’s Plural Rules
- Implementation in Mozilla Firefox
- SX discussion on plural forms

Use `WebView` for Chat Message Content

Context and Problem Statement

This decision record concerns the UI component that is used for rendering the content of AI summaries.

Decision Drivers

Same as in ADR-0036.

Considered Options

Same as in ADR-0036.

Decision Outcome

Chosen option: “Use `WebView`”.

Some of the options does not support selecting and copying of text. Some options do not render Markdown.

However, in contrary to ADR-0036, we chose here a `WebView`, instead of `TextArea`, because there is only one summary content in UI (when user switches entries, no new components are added, rather old ones are *rebinding* to new entry). It would hurt the performance if we used `WebView` for messages, as there could be a lot of messages in one chat.

Pros and Cons of the Options

Same as in ADR-0036.

More Information

This ADR is highly linked to ADR-0036.

About the selection and copying, this goes down to fundamental issue from JavaFX. `Text` and `Label` as a whole or a part cannot be selected and/or copied.

Show merge dialog when importing a single PDF

Context and Problem Statement

PDF files are one of the main format for transferring various documents, especially scientific papers. However, by itself, PDF is like a picture, it contains commands solely for displaying the human-readable text, but it might not contain computer-readable metadata.

To overcome these problems various heuristics and AI models are used to "convert" a PDF into a BibTeX entry. However, it also introduces a level of problems, as heuristics are not ideal: sometimes it works perfectly, but on others it generates random output.

PDF importing in JabRef is done via `PdfImporter` abstract class and its descendants, and via `PdfMergeMetadataImporter`. `PdfImporter` is typically a single heuristics or method of extracting a `BibEntry` from PDF. `PdfMergeMetadataImporter` collects `BibEntry` candidates from all `PdfImporter`s and merges them automatically into a single `BibEntry`.

The specific problem JabRef has: should JabRef automate all heuristics (automatically merge all `BibEntry`ies from several `PdfImporter`s) when importing PDF files or should every file be analysed thoroughly by users?

Decision Drivers

- Option should provide a good-enough quality.
- It is desired to have a fine-grained controls of PDF importing for power-users.

Considered Options

- Automatically merge all `BibEntry` candidates from `PdfImporters`.
- Open a merge dialog with all candidates.
- Open a merge dialog with all candidates if a single PDF is imported.

Decision Outcome

Chosen option: "Open a merge dialog with all candidates if a single PDF is imported", because comes out best (see below).

Pros and Cons of the Options

Automatically merge all `BibEntry` candidates from `PdfImporters`

- Good, because minimal user interaction and disruption of flow. It also allows batch-processing.
- Bad, because heuristics are not ideal, and it is even harder to develop a “smarter” merging algorithm.

Open a merge dialog with all candidates

- Good, because allows for fine-grained import. Some correct field may be overridden by a wrong field from other importer, which is undesirable for power-users.
- Bad, because it is a dialog. If lots of PDFs are imported, then there will be lots of dialogs, which might be too daunting to process manually.

Open a merge dialog with all candidates if a single PDF is imported

Explanation:

- If a single PDF is imported, then open a merge dialog.
- If several PDFs are imported, merge candidates for each PDF automatically.

Outcomes:

- Good, because it combines the best of the other two options: Allow both for PDF batch-processing and for fine-grained control.

{short title, representative of solved problem and found solution}

Context and Problem Statement

{Describe the context and problem statement, e.g., in free form using two to three sentences or in the form of an illustrative story. You may want to articulate the problem in form of a question and add links to collaboration boards or issue management systems.}

Decision Drivers

- {decision driver 1, e.g., a force, facing concern, …}
- {decision driver 2, e.g., a force, facing concern, …}
- …

Considered Options

- {title of option 1}
- {title of option 2}
- {title of option 3}
- …

Decision Outcome

Chosen option: “{title of option 1}”, because {justification. e.g., only option, which meets k.o. criterion decision driver	which resolves force {force}	…	comes out best (see below)}.

Consequences

- Good, because {positive consequence, e.g., improvement of one or more desired qualities, …}
- Bad, because {negative consequence, e.g., compromising one or more desired qualities, …}
- …

Confirmation

{Describe how the implementation of/compliance with the ADR can/will be confirmed. Is the chosen design and its implementation in line with the decision? E.g., a design/code review or a test with a library such as ArchUnit can help validate this. Note that although we classify this element as optional, it is included in many ADRs.}

Pros and Cons of the Options

{title of option 1}

{example | description | pointer to more information | …}

- Good, because {argument a}
- Good, because {argument b}
- Neutral, because {argument c}
- Bad, because {argument d}
- …

{title of other option}

{example	description	pointer to more information	…}

- Good, because {argument a}
- Good, because {argument b}
- Neutral, because {argument c}
- Bad, because {argument d}
- …

More Information

{You might want to provide additional evidence/confidence for the decision outcome here and/or document the team agreement on the decision and/or define when/how this decision the decision should be realized and if/when it should be re-visited. Links to other decisions and resources might appear here as well.}

Decision Records

Below, all “architectural decision records” for JabRef are listed. This list uses the TOC functionality of the Just the Docs Jekyll template.

For new ADRs, please use adr-template.md as basis. More information on MADR is available at https://adr.github.io/madr/. General information about architectural decision records is available at https://adr.github.io/.

TABLE OF CONTENTS

- Use Markdown Architectural Decision Records
- Use Crowdin for translations
- Use SLF4J together with log4j2 for logging
- Use Gradle as build tool
- Use MariaDB Connector
- Fully Support UTF-8 Only For LaTeX Files
- Only translated strings in language file
- Provide a human-readable changelog
- Use public final instead of getters to offer access to immutable variables
- Use Plain JUnit5 for advanced test assertions
- Use H2 as Internal SQL Database
- Test external links in documentation
- Handle different bibentry formats of fetchers by adding a layer
- Add Native Support for BibLatex-Software
- Separate URL creation to enable proper logging
- Query syntax design
- Mutable preferences objects
- Allow org.jabref.model to access org.jabref.logic
- Use regular expression to split multiple-sentence titles
- Implement special fields as separate fields
- Use Jackson to parse study.yml
- Keep study as a DTO
- Remove stop words during query transformation
- Localized Preferences
- Use # as indicator for BibTeX string constants

- Reviewdog findings are code reviews
- Use Java Native Access to Determine Default Directory
- Store Chats Alongside Database
- Synchronization with remote databases
- Store Chats in MVStore
- Return BibTeX string and CSL Item JSON in the API
- Use Citation Key for Grouping Chat Messages
- Exporting multiple entries to CFF
- Generate Embeddings Online
- Use Apache Commons IO for directory monitoring
- Use TextArea for Chat Message Content
- Use currently active tab in Select style (OO Panel) to decide style type
- RAG Architecture Implementation
- Use WebView for Chat Message Content
- Use BibEntry.getId for BibEntry at indexing
- Use Apache Velocity as template engine
- Display front cover for book citations in the Preview tab
- Show merge dialog when importing a single PDF
- ADR Template

JabRef's development strategy

We aim to keep up to high-quality code standards and use code quality tools wherever possible.

To ensure high code-quality,

- We follow the principles of Java by Comparison.
- We follow the principles of Effective Java.
- We use Design Patterns when applicable.
- We document our design decisions using the lightweight architectural decision records MADR.
- We review each external pull request by at least two JabRef Core Developers.

Read on about our automated quality checks at Code Quality.

Continuous integration

JabRef has automatic checks using GitHub actions in place. One of them is checking for the formatting of the code. Consistent formatting ensures more easy reading of the code. Thus, we pay attention that JabRef's code follows the same code style.

Binaries are created using gradle and are uploaded to https://builds.jabref.org. These binaries are created without any checks to have them available as quickly as possible, even if the localization or some fetchers are broken. Deep link to the action: https://github.com/JabRef/jabref/actions?workflow=Deployment.

Branches

The branch main is the main development line and is intended to incorporate fixes and improvements as soon as possible and to move JabRef forward to modern technologies such as the latest Java version.

Other branches are used for discussing improvements with the help of pull requests. One can see the binaries of each branch at https://builds.jabref.org/. Releases mark milestones and are based on the `main` branch at a point in time.

How JabRef acquires contributors

- We participate in Hacktoberfest.
- We participate in Google Summer of Code.

Historical notes

JabRef 4.x

The main roadmap for JabRef 4.x was to modernize the UI, make the installation easier and reduce the number of opened issues.

JabRef 3.x

JabRef at the beginning of 2016 had a few issues:

- Most of the code is untested, non-documented, and contains a lot of bugs and issues.
- During the lifetime of JabRef, a lot of features, UI elements and preferences have been added. All of them are loosely wired together in the UI, but the UI lacks consistency and structure.
- This makes working on JabRef interesting as in every part of the program, one can improve something. :smiley:

JabRef 3.x is the effort to try to fix a lot of these issues. Much has been achieved, but much is still open.

We currently use two approaches: a) rewrite and put under test to improve quality and fix bugs, b) increase code quality. This leads to pull requests being reviewed by two JabRef developers to ensure i) code quality, ii) fit within the JabRef architecture, iii) high test coverage.

Code quality includes using latest Java features, but also readability.

Advanced: Eclipse as IDE

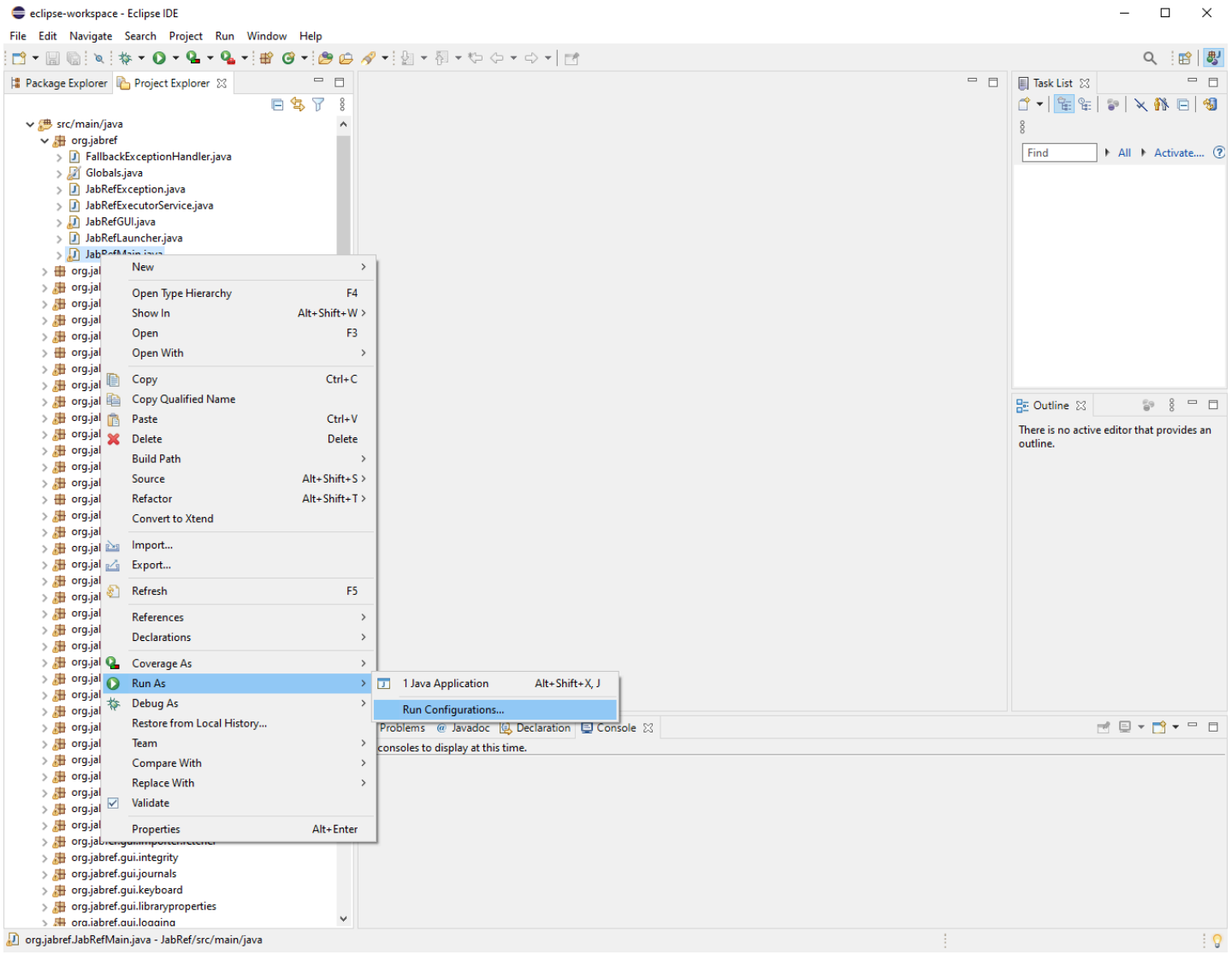
For advanced users, Eclipse (`2023-03` or newer) is also possible. On Ubuntu Linux, you can follow the documentation from the Ubuntu Community or the step-by-step guideline from Krizna to install Eclipse. On Windows, download it from www.eclipse.org and run the installer.

For Eclipse, a working Java (Development Kit) 20 installation is required. In the case of IntelliJ, this will be downloaded inside the IDE (if you follow the steps below).

In the command line (terminal in Linux, cmd in Windows) run `javac -version` and make sure that the reported version is Java 20 (e.g., `javac 20`). If `javac` is not found or a wrong version is reported, check your `PATH` environment variable, your `JAVA_HOME` environment variable or install the most recent JDK. Please head to https://adoptium.net/de/temurin/releases to download JDK 20.

Always make sure your Eclipse installation us up to date.

1. Run `./gradlew run` to generate all resources and to check if JabRef runs.
   - The JabRef GUI should finally appear.
   - This step is only required once.
   - The directory `src-gen` is now filled.
2. Run `./gradlew eclipse`
   - **This must always be executed, when there are new upstream changes.**
3. Open or import the existing project in Eclipse as Java project.
   - Remark: Importing it as gradle project will not work correctly.
   - Refresh the project in Eclipse
4. Create a run/debug configuration for the main class `org.jabref.Launcher` and/or for `org.jabref.gui.JabRefMain` (both can be used equivalently)
   - Remark: The run/debug configuration needs to be added by right-clicking the class (e.g. `Launcher` or JabRefMain) otherwise it will not work.



- In the tab “Arguments” of the run/debug configuration, enter the following runtime VM arguments:

```
--add-exports javafx.controls/com.sun.javafx.scene.control=org.jabref
--add-exports org.controlsfx.controls/impl.org.controlsfx.skin=org.jabref
--add-exports javafx.graphics/com.sun.javafx.scene=org.controlsfx.controls
--add-opens javafx.graphics/javafx.scene=org.controlsfx.controls
--add-exports javafx.graphics/com.sun.javafx.scene.traversal=org.controlsfx.controls
--add-exports javafx.graphics/com.sun.javafx.css=org.controlsfx.controls
--add-exports javafx.controls/com.sun.javafx.scene.control.behavior=org.controlsfx.controls
--add-exports javafx.controls/com.sun.javafx.scene.control=org.controlsfx.controls
--add-exports javafx.controls/com.sun.javafx.scene.control.inputmap=org.controlsfx.controls
--add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
--add-exports javafx.base/com.sun.javafx.collections=org.controlsfx.controls
--add-exports javafx.base/com.sun.javafx.runtime=org.controlsfx.controls
--add-exports javafx.web/com.sun.webkit=org.controlsfx.controls
--add-exports javafx.graphics/com.sun.javafx.css=org.controlsfx.controls
--patch-module org.jabref=build/resources/main
```

- In the tab “Dependencies” of the run/debug configuration tick the checkbox “Exclude test code”

5. Optional: Install the e(fx)clipse plugin from the Eclipse marketplace: 1. Help -> Eclipse Marketplace... -> Search tab 2. Enter “e(fx)clipse” in the search dialogue 3. Click “Go” 4. Click “Install” button next to the plugin 5. Click “Finish”
6. Now you can build and run/debug the application by either using `Launcher` or `JabRefMain`. This is the recommended way, since the application starts quite fast.

Localization Test Configuration (Eclipse)

To run the `LocalizationConsistencyTest` you need to add some extra module information: Right-click on the file -> “Run/Debug as JUnit test”. Go to the Run/debug configuration created for that file and in the arguments tab under VM-configurations add:

```
--add-exports javafx.graphics/com.sun.javafx.application=ALL-UNNAMED
--add-exports javafx.graphics/com.sun.javafx.stage=ALL-UNNAMED
--add-exports javafx.graphics/com.sun.javafx.stage=com.jfoenix
```

Step 1: Get the code into IntelliJ

Start IntelliJ.

IntelliJ shows the following window:

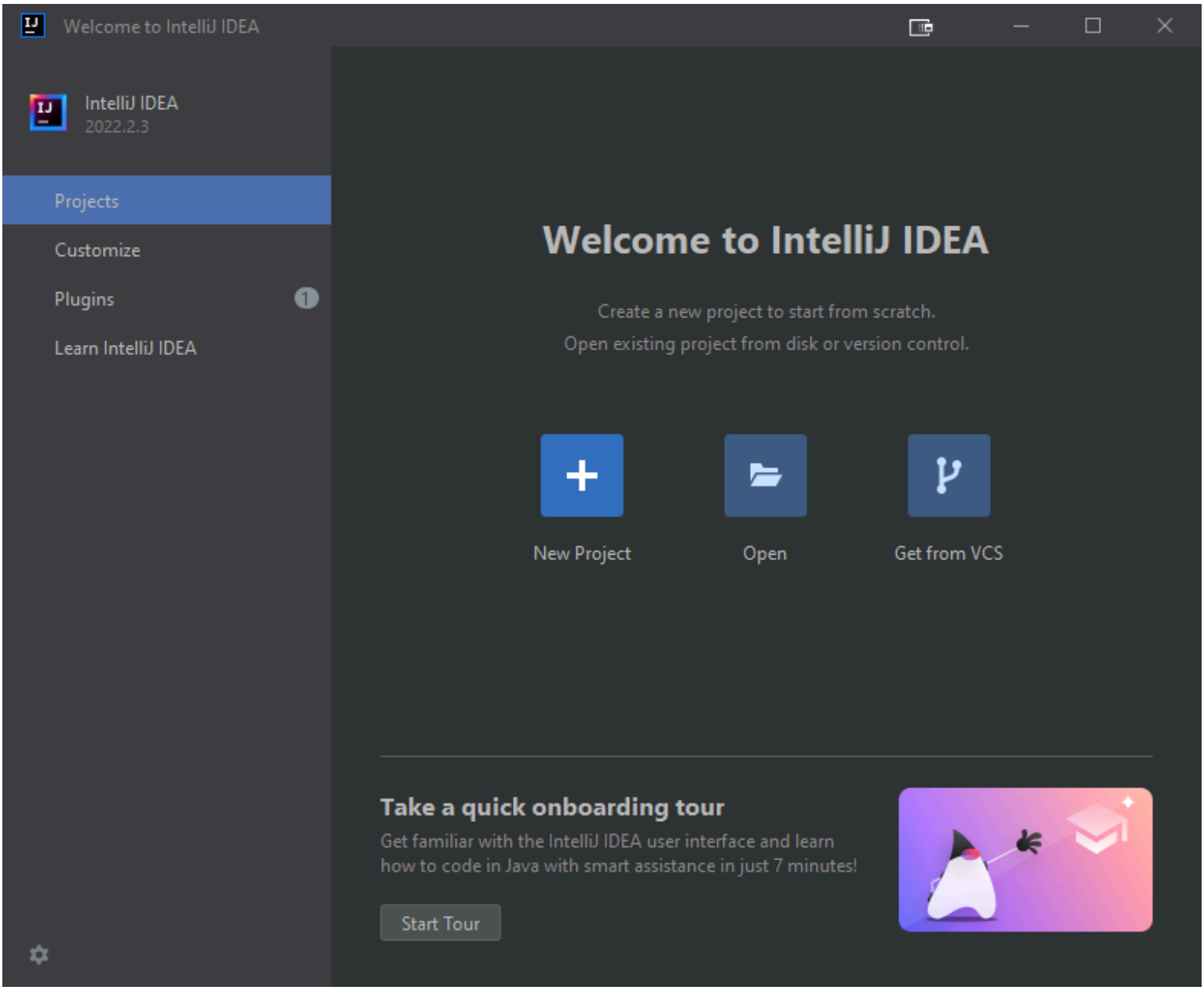


Figure: IntelliJ Start Window

Click on “Open”

Choose `build.gradle` in the root of the jabref source folder:

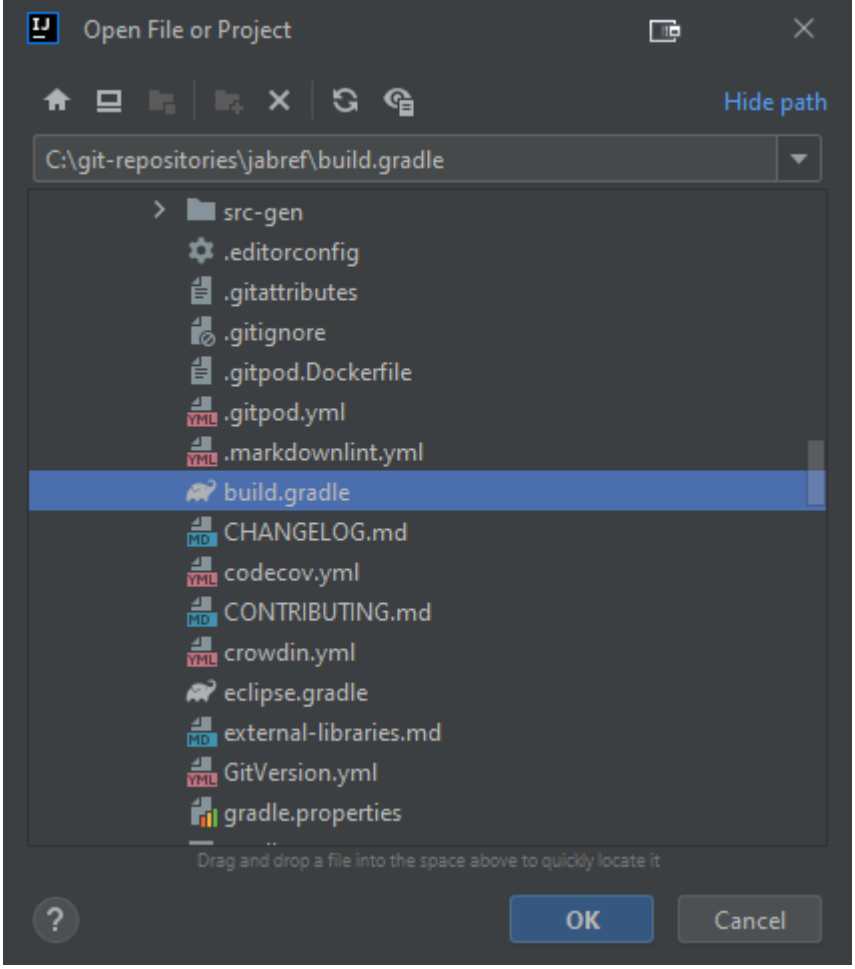


Figure: Choose `build.gradle` in the "Open Project or File" dialog

After pressing "OK", IntelliJ asks how that file should be opened. Answer: "Open as Project"

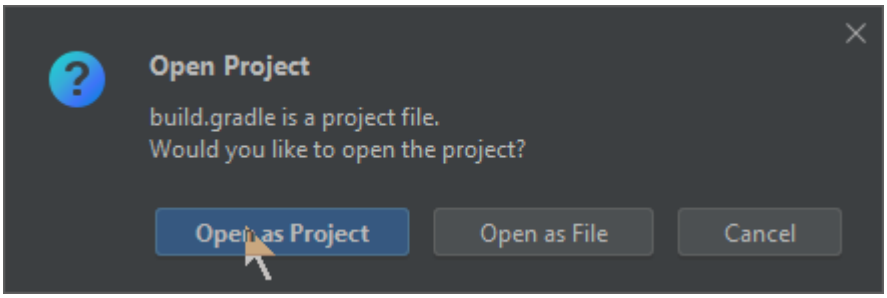


Figure: Choose "Open as Project" in the Open Project dialog

Then, trust the project:

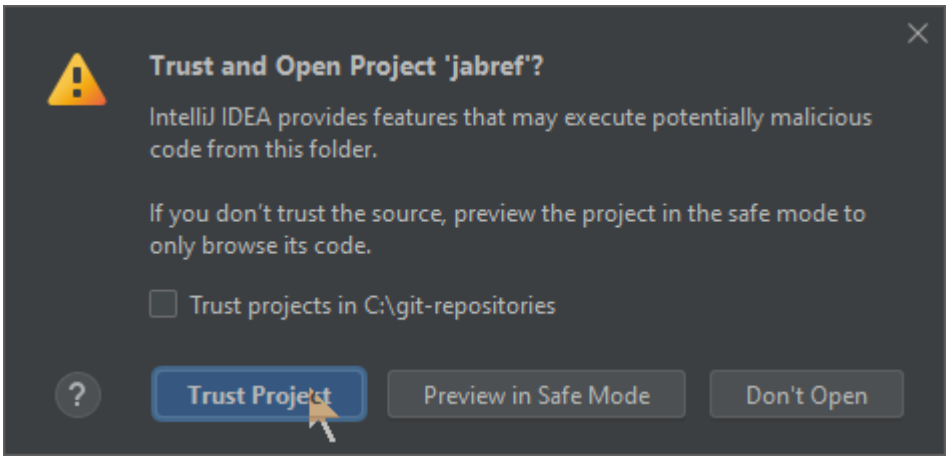


Figure: Choose "Trust Project" in the "Trust and Open Project" dialog

Ensure that committing via IntelliJ works

IntelliJ offers committing using the UI. Press `Alt+0` to open the commit dialog.

Unfortunately, IntelliJ has no support for ignored sub modules [IDEA-285237]. Fortunately, there is a workaround:

Go to **File > Settings… > Version Control > Directory Mappings**.
Note: In some MacBooks, `Settings` can be found at the “IntelliJ” button of the app menu instead of at “File”.

Currently, it looks as follows:

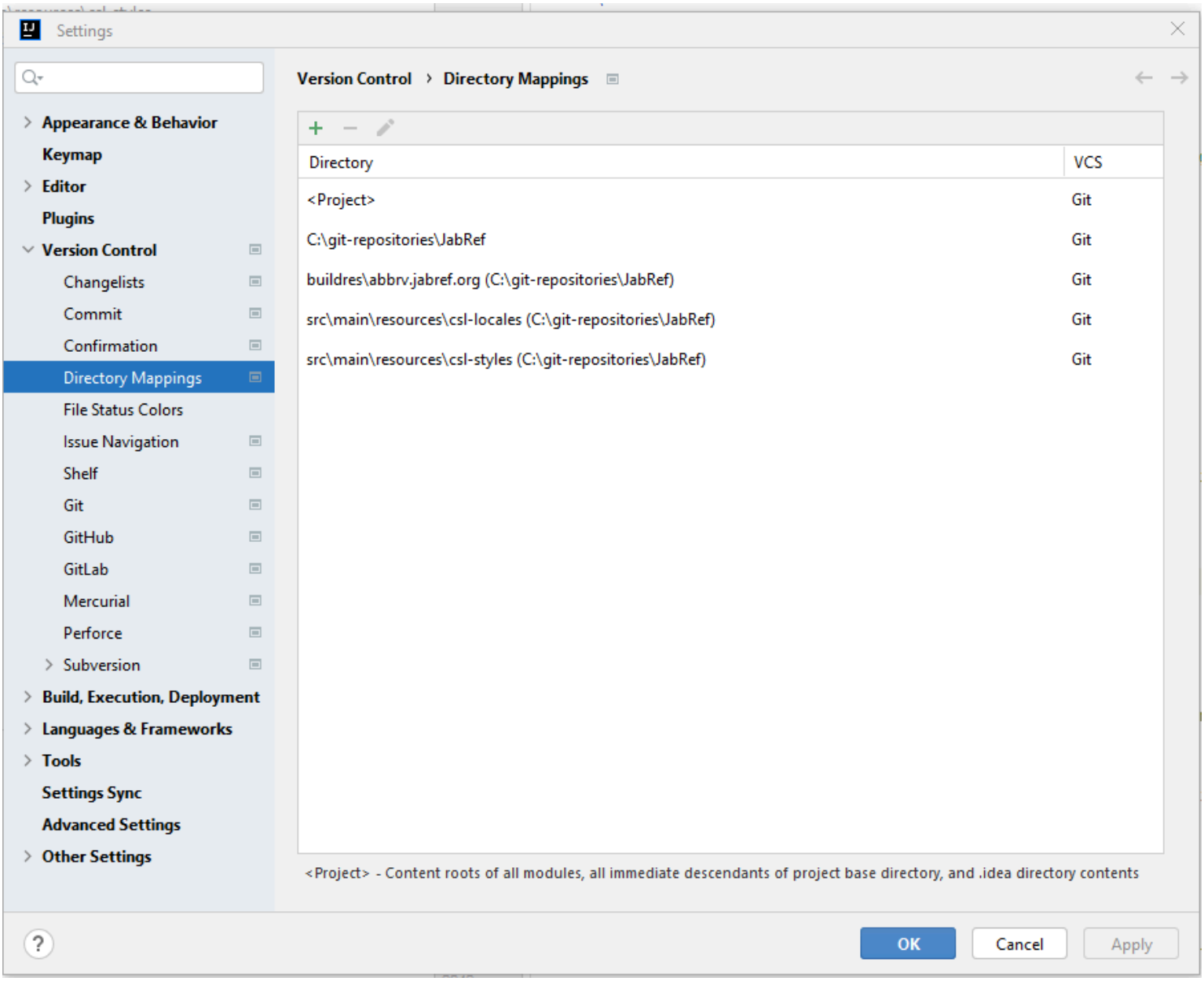


Figure: Directory Mappings unmodified

You need to tell IntelliJ to ignore the submodules `buildres\abbrv.jabref.org`, `src\main\resources\csl-locales`, and `src\main\resources\csl-styles`. Select all three (holding the `Ctrl` key). Then press the red minus button on top.

This will make these directories “Unregistered roots:”, which is fine.

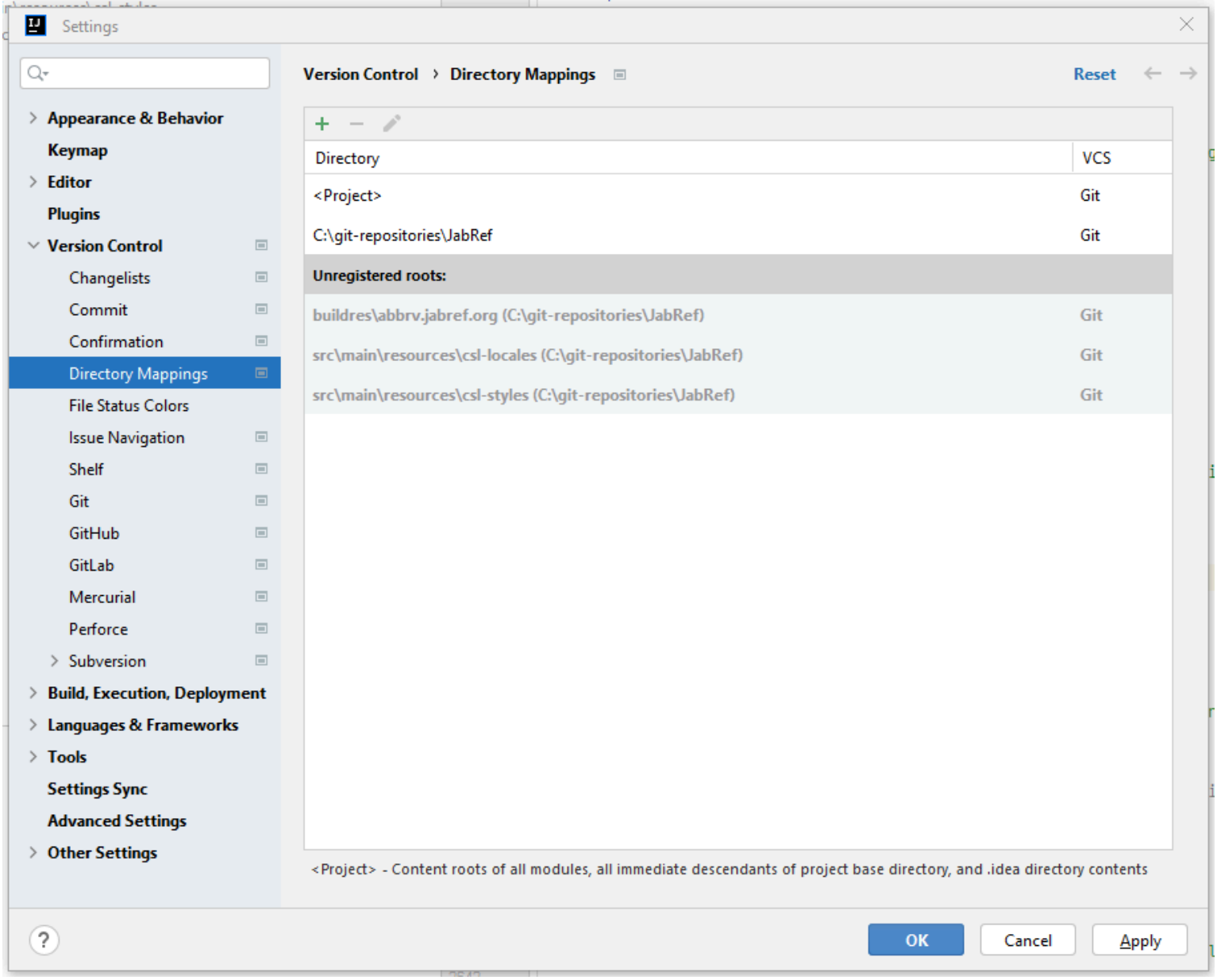


Figure: Directory Mappings having three unregistered roots

Ensure that committing with other tools work

Open a “git bash”. On Windows, navigate to `C:\git-repositories\JabRef`. Open the context menu of the file explorer (using the right mouse button), choose “Open Git Bash here”.

Execute following command:

```
git update-index --assume-unchanged buildres/abbrv.jabref.org src/main/resources/csl-styles src/main/res
```

If you do not see the context menu, re-install git following the steps given at StackOverflow.

Step 2: Set up the build system: JDK and Gradle

Ensure that JDK 23 is available to IntelliJ

Ensure you have a Java 23 SDK configured by navigating to **File > Project Structure… > Platform Settings > SDKs**.

Note: In some MacBooks, `Project Structure` can be found at the “IntelliJ” button of the app menu instead of at “File”.

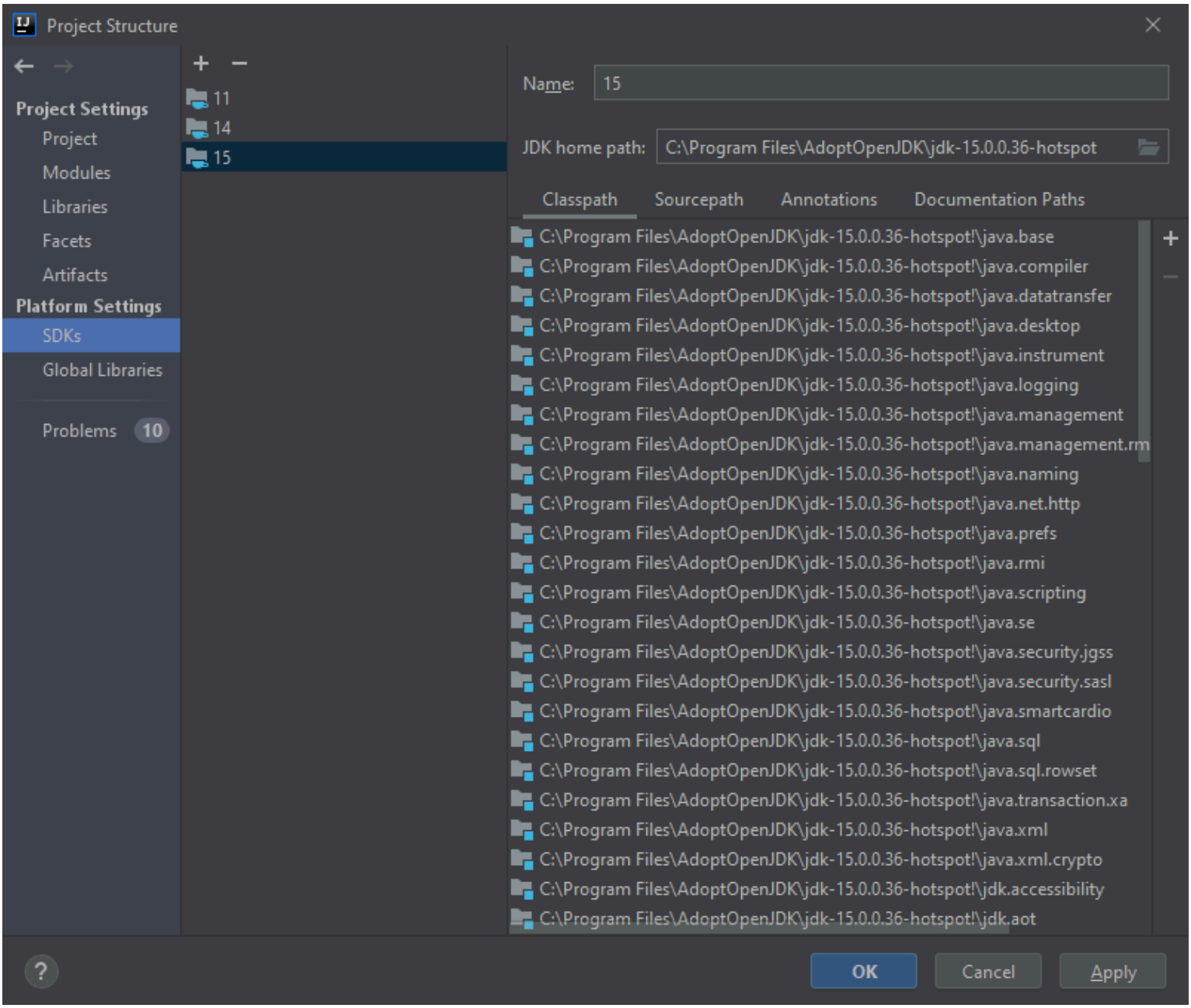


Figure: JDKs 11, 14, and 15 shown in available SDKs. JDK 23 is missing.

If there is another JDK than JDK 23 selected, click on the plus button and choose “Download JDK…”

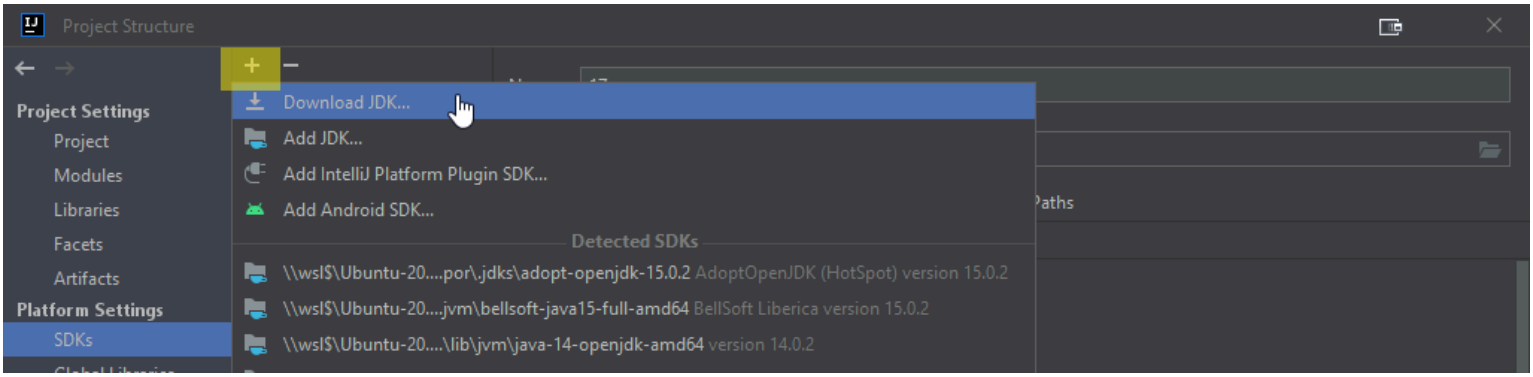


Figure: Download JDK…

Select JDK version 23 and then Eclipse Temurin.

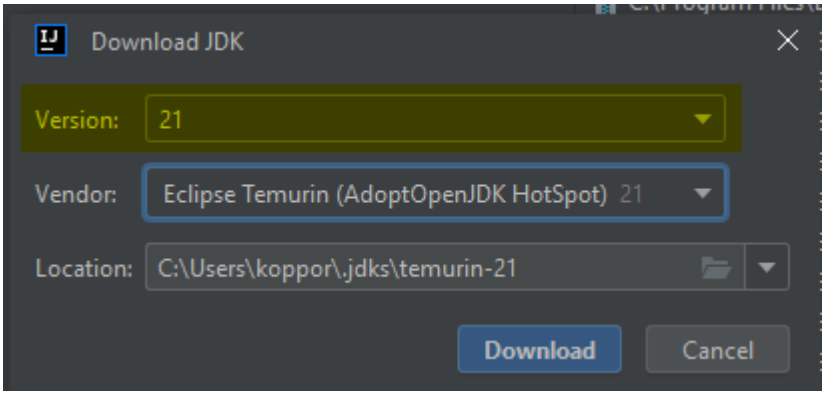


Figure: Example for JDK 23 - Choose Eclipse Temurin

After clicking “Download”, IntelliJ installs Eclipse Temurin:

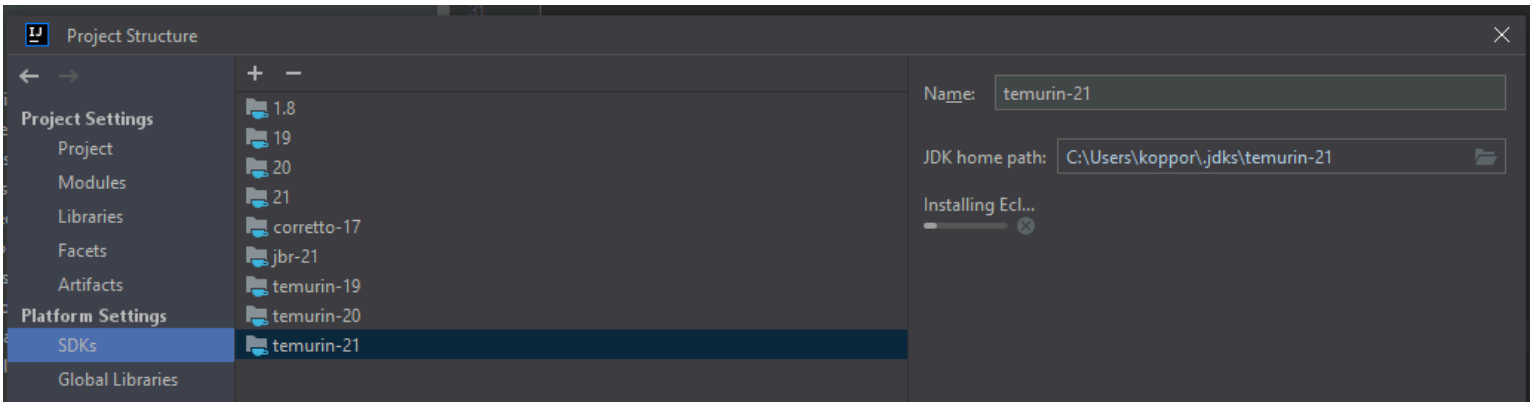


Figure: IntelliJ installs Eclipse Temurin

Navigate to **Project Settings > Project** and ensure that the projects’ SDK is Java 23.

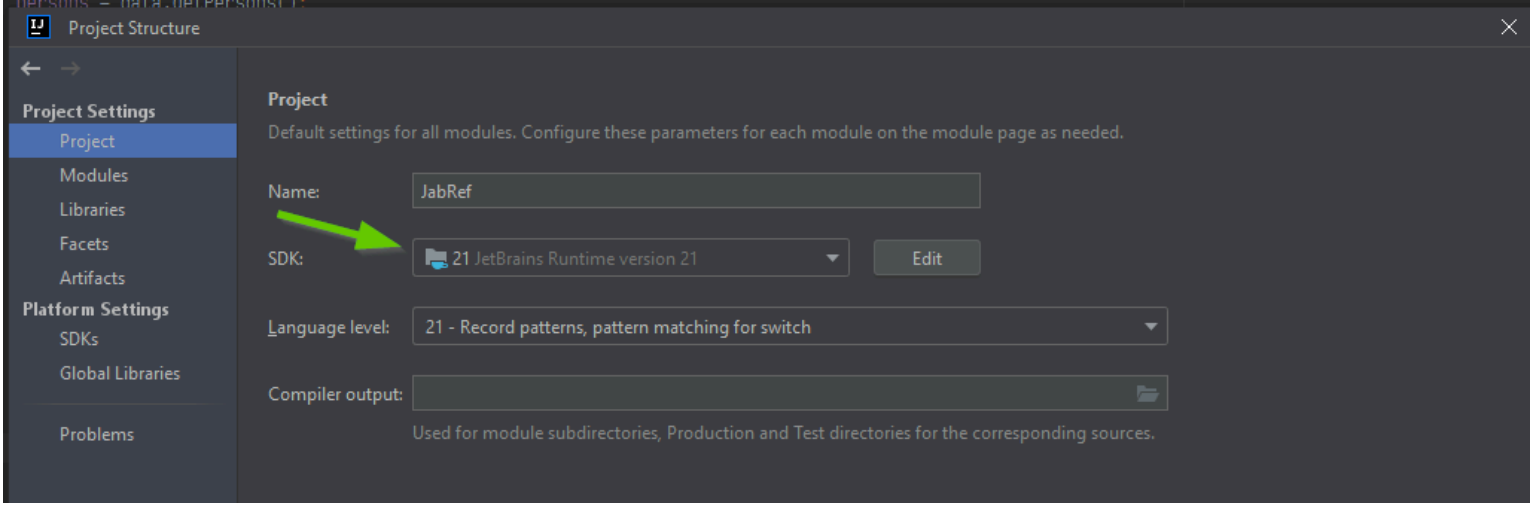


Figure: Project SDK is pinned to the downloaded SDK (showing JDK 23 as example)

Click “OK” to store the changes.

Ensure correct JDK setting for Gradle

Navigate to **File > Settings… > Build, Execution, Deployment > Build Tools > Gradle** and select the “Project SDK” as the Gradle JVM at the bottom. If that does not exist, just select JDK 23.

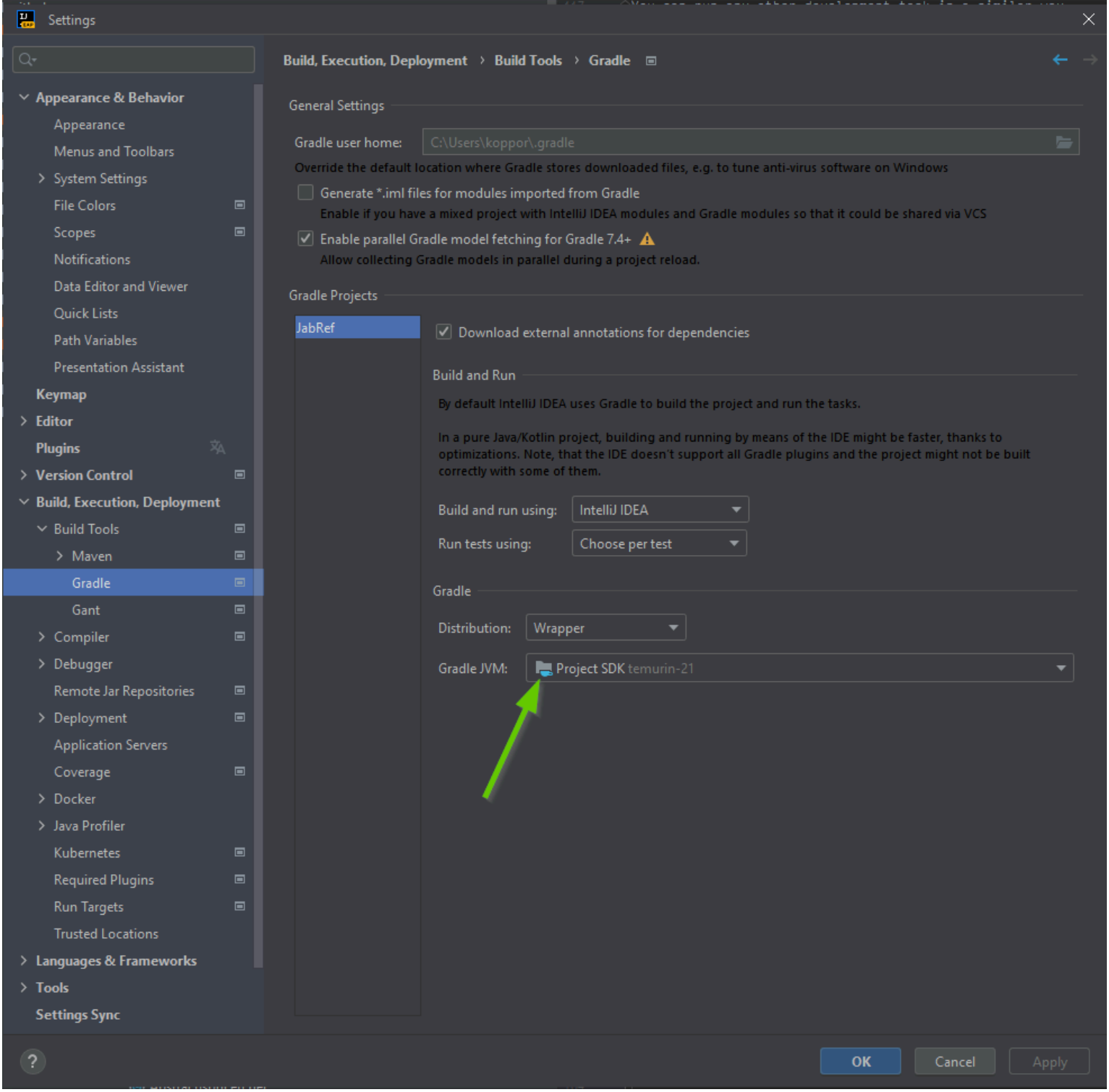


Figure: Gradle JVM is project SDK (showing

Enable compilation by IntelliJ

To prepare IntelliJ's build system additional steps are required:

Navigate to **Build, Execution, Deployment > Compiler > Java Compiler**, and under "Override compiler parameters per-module", click add ([+]) and choose `JabRef.main`:

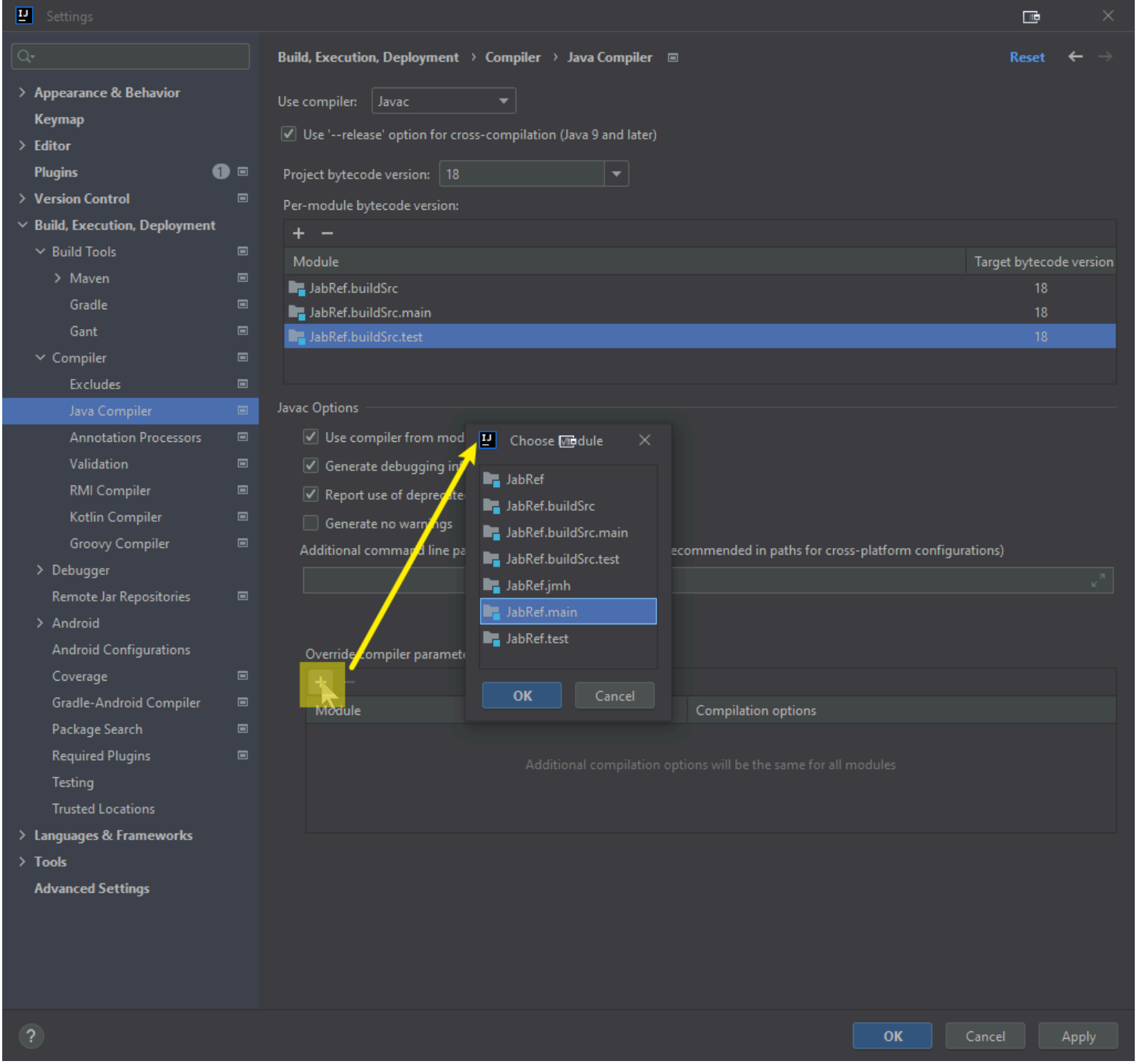


Figure: Choose JabRef.main

Copy following text into your clipboard:

```
--add-exports=javafx.controls/com.sun.javafx.scene.control=org.jabref
--add-exports=org.controlsfx.controls/impl.org.controlsfx.skin=org.jabref
--add-reads org.jabref=org.apache.commons.csv
--add-reads org.jabref=org.fxmisc.flowless
--add-reads org.jabref=langchain4j.core
--add-reads org.jabref=langchain4j.open.ai
```

Then double click inside the cell “Compilation options”. Press `Ctrl+A` to mark all text. Press `Ctrl+V` to paste all text. Press `Enter` to have the value really stored. Otherwise, it seems like the setting is stored, but it is not there if you reopen this preference dialog.

Note: If you use the expand arrow, you need to press `Shift+Enter` to close the expansion and then `Enter` to commit the value.

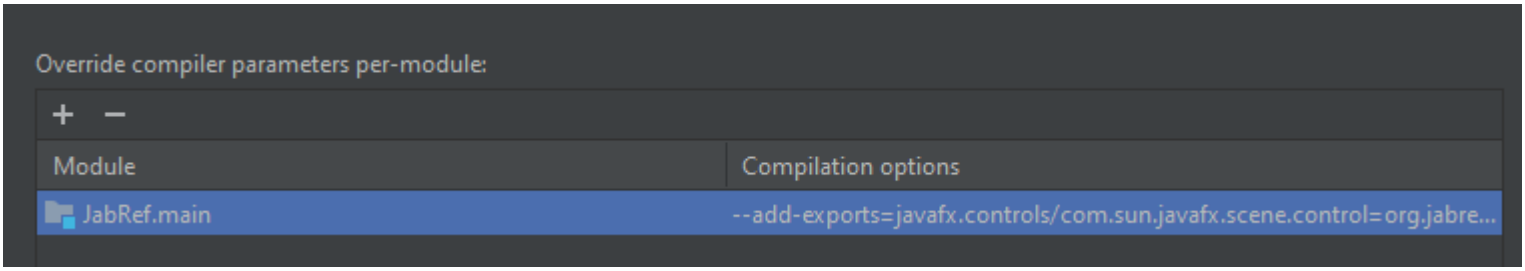


Figure: Resulting settings for module JabRef.main

Then click on “Apply” to store the setting.

Note: If this step is omitted, you will get: `java: package com.sun.javafx.scene.control is not visible (package com.sun.javafx.scene.control is declared in module javafx.controls, which does not export it to module org.jabref)`.

Enable annotation processors

Enable annotation processors by navigating to **Build, Execution, Deployment > Compiler > Annotation processors** and check “Enable annotation processing”

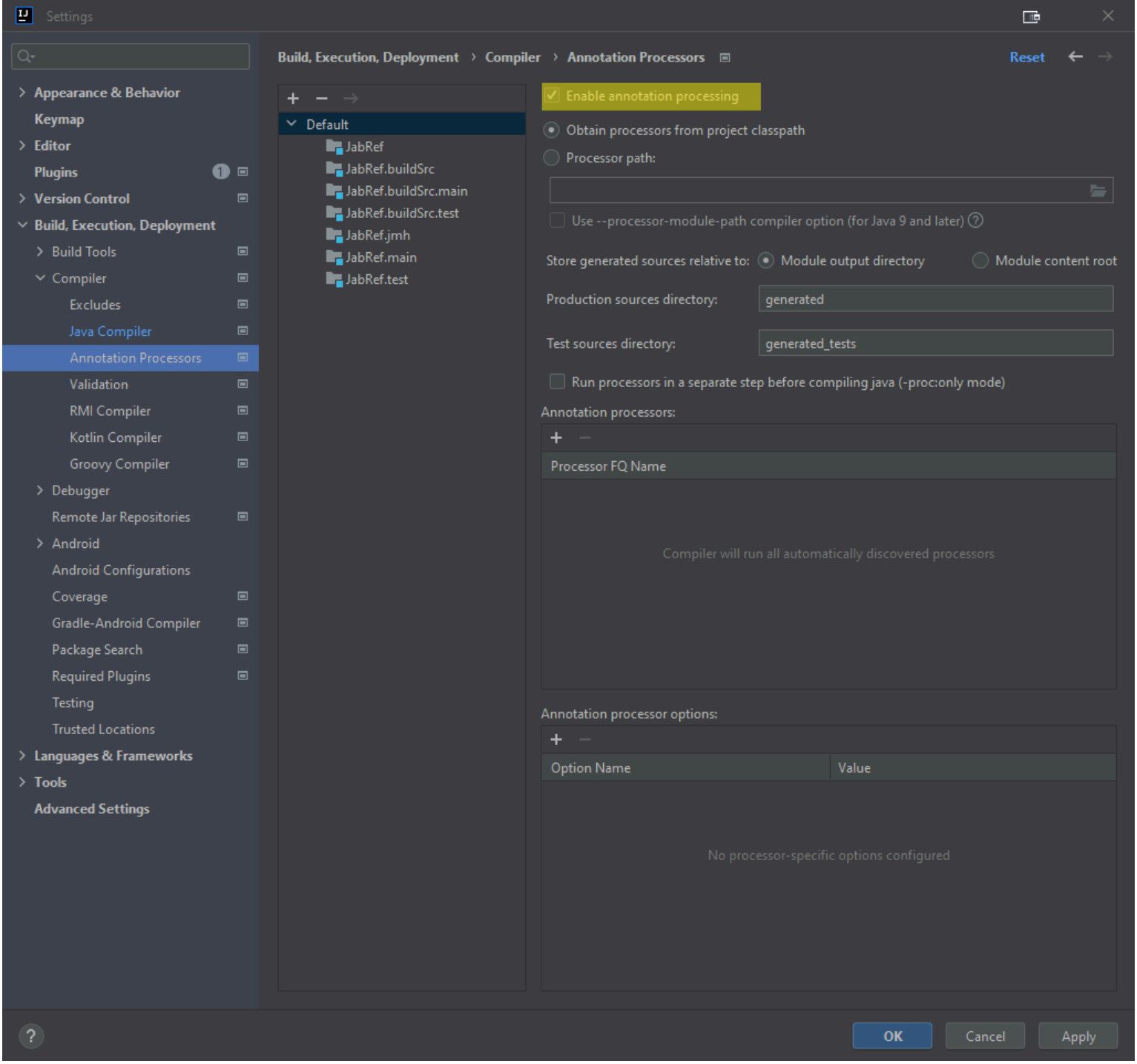


Figure: Enabled annotation processing

Using Gradle from within IntelliJ IDEA

> **NOTE**
> Ensuring JabRef builds with Gradle should always be the first step because, e.g. it generates additional sources that are required for compiling the code.

Open the Gradle Tool Window with the small button that can usually be found on the right side of IDEA or navigate to **View > Tool Windows > Gradle**. In the Gradle Tool Window, press the “Reload All Gradle Projects” button to ensure that all settings are up-to-date with the setting changes.

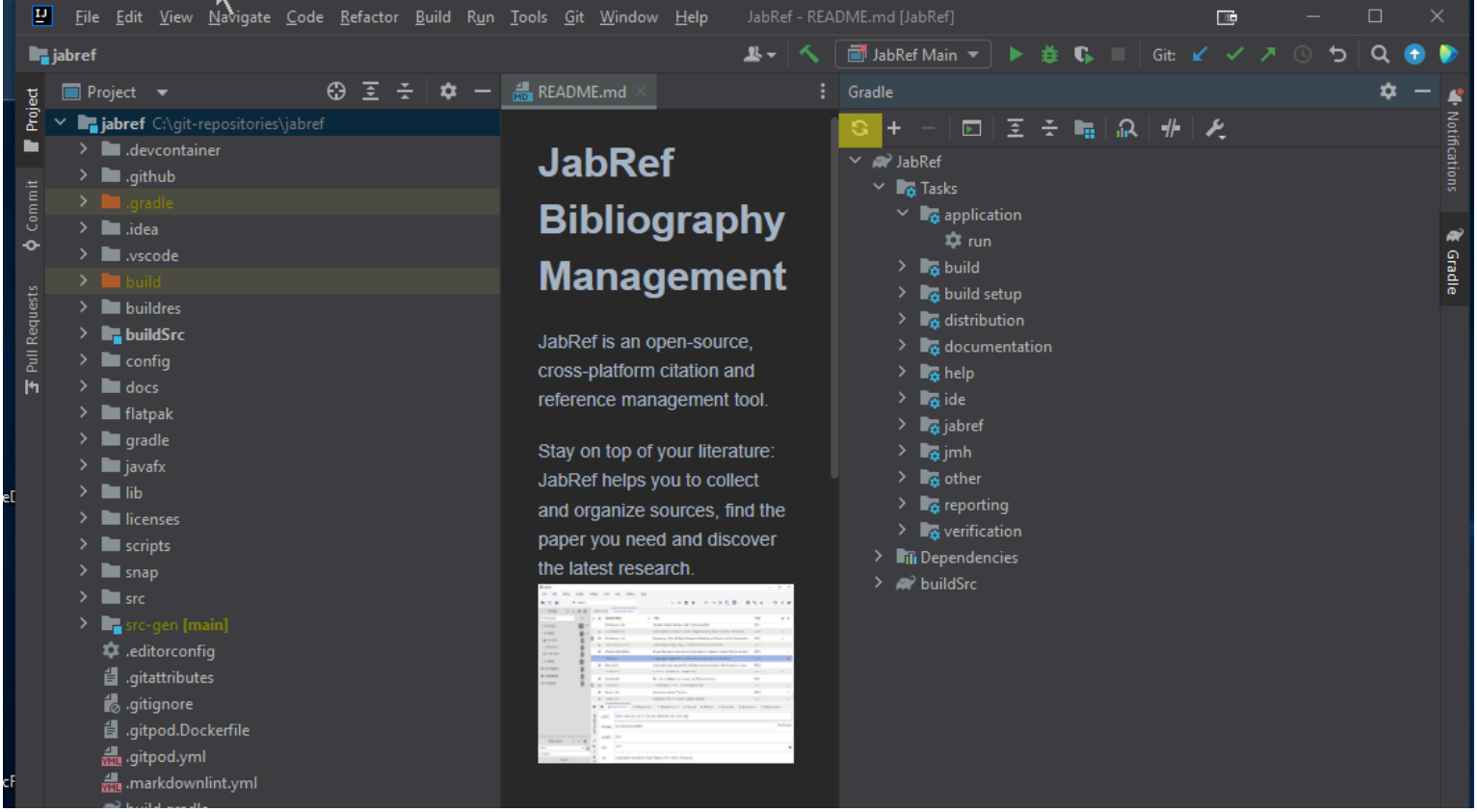


Figure: Reload of Gradle project

After that, you can use the Gradle Tool Window to build all parts of JabRef and run it. To do so, expand the JabRef project in the Gradle Tool Window and navigate to Tasks. From there, you can build and run JabRef by double-clicking **JabRef > Tasks > application > run**.

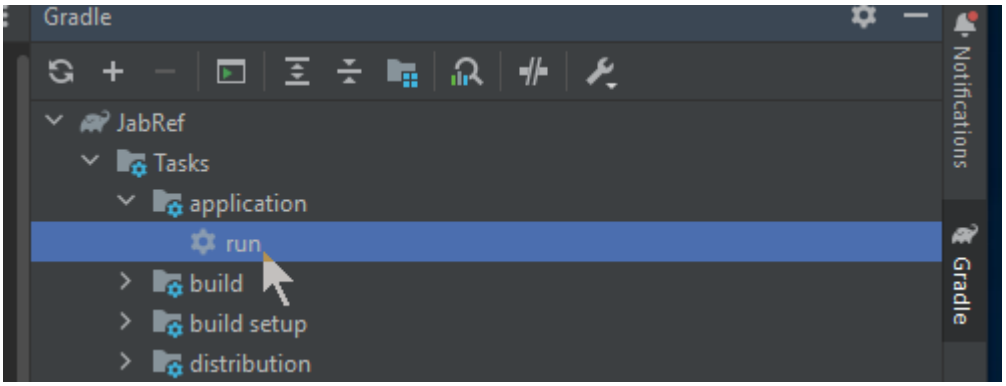


Figure: JabRef > Tasks > application > run

The Gradle run window opens, shows compilation and then the output of JabRef. The spinner will run as long as JabRef is open.

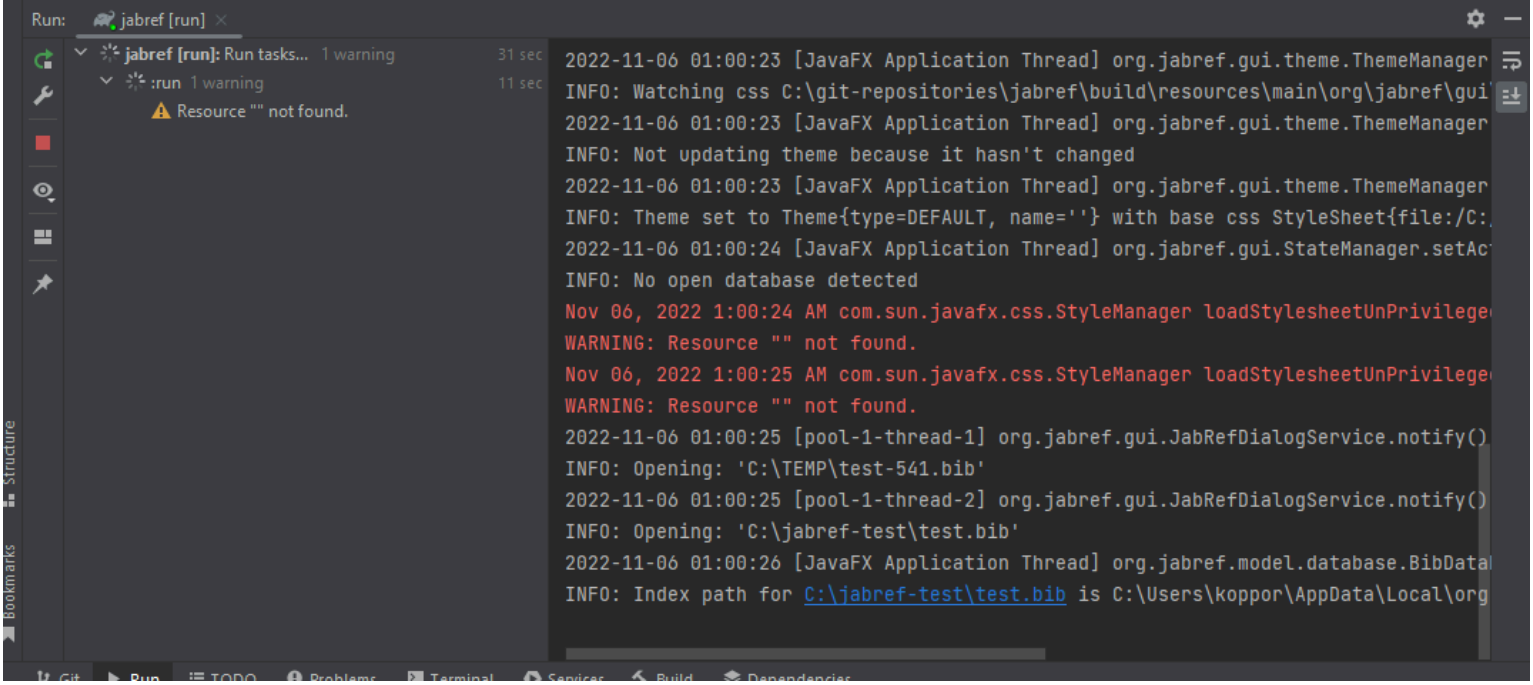


Figure: Gradle run Window

You can close JabRef again.

After that a new entry called “jabref [run]” appears in the run configurations. Now you can also select “jabref [run]” and either run or debug the application from within IntelliJ.

> **NOTE**
> You can run any other development task similarly.

Using IntelliJ’s internal build system for tests

In **File > Settings… > Build, Execution, Deployment > Build Tools > Gradle** the setting “Run tests using:” is set to “IntelliJ IDEA”.

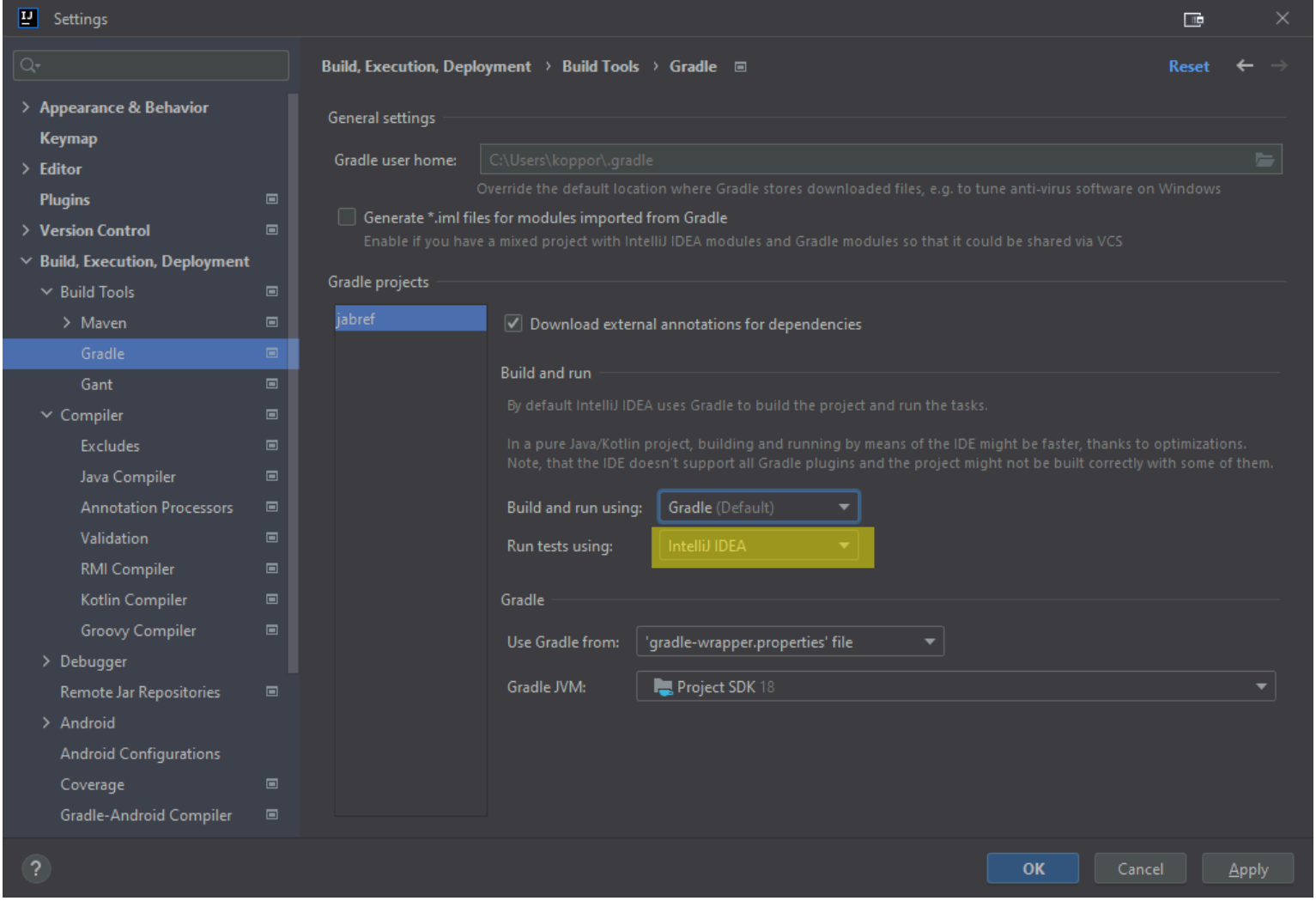


Figure: IntelliJ setting: Run tests using IntelliJ

> **NOTE**
> In case there are difficulties later, this is the place to switch back to gradle.

Click “OK” to close the preference dialog.

In the menu bar, select **Build > Rebuild project**.

IntelliJ now compiles JabRef. This should happen without any error.

Now you can use IntelliJ IDEA’s internal build system by using **Build > Build Project**.

Final build system checks

To run an example test from IntelliJ, we let IntelliJ create a launch configuration:

Locate the class `BibEntryTest`: Press `Ctrl+N`. Then, the “Search for classes dialog” pops up. Enter `bibentrytest`. Now, `BibEntryTest` should appear first:

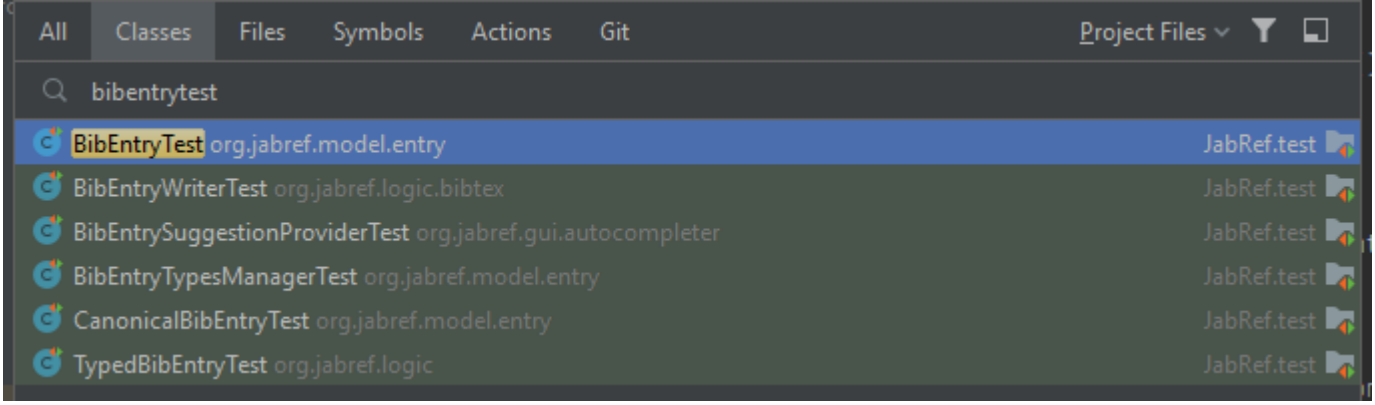


Figure: IntelliJ search for class “BibEntryTest”

Press `Enter` to jump to that class.

Hover on the green play button on `defaultConstructor`:

Figure: However on green play button

Then, click on it. A popup menu opens. Choose the first entry “Run testDefaultConstructor” and click on it.

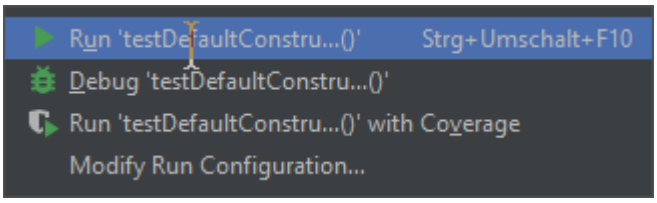


Figure: Run testDefaultConstructor

Then, the single test starts.

You also have an entry in the Launch configurations to directly launch the test. You can also click on the debug symbol next to it to enable stopping at breakpoints.

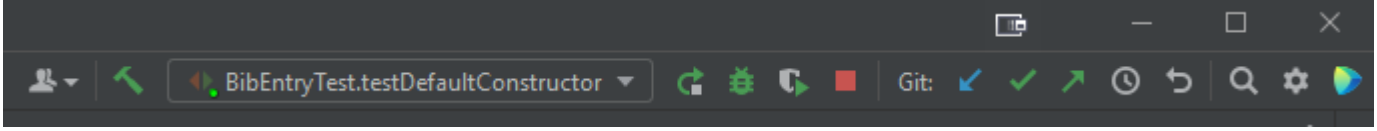


Figure: Launch menu contains BibEntry test case

The tests are green after the run. You can also use the play button there to re-execute the tests. A right-click on “BibEntryTests” enables the debugger to start.

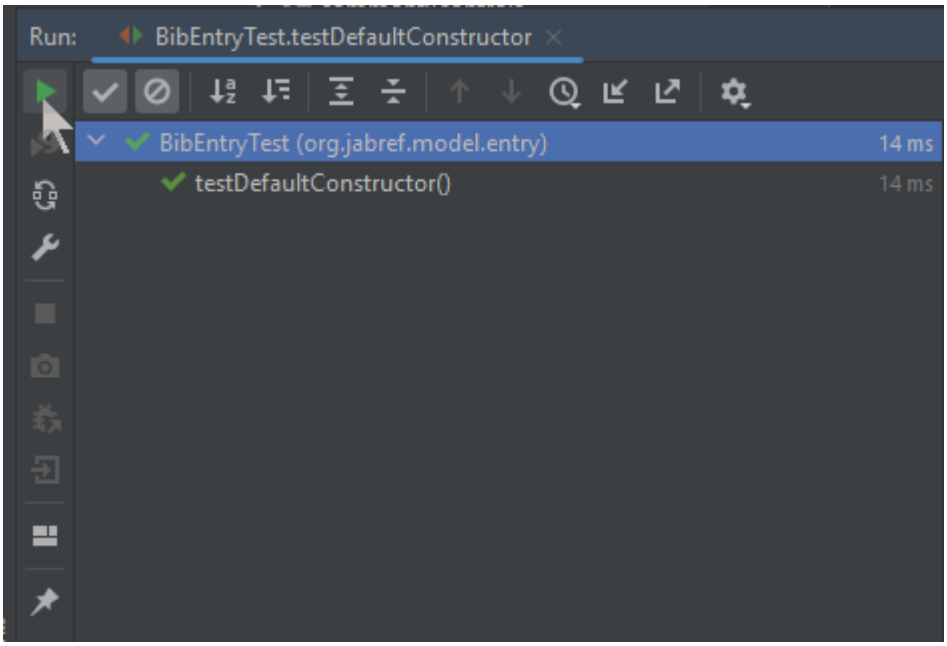


Figure: Run window for the BibEntry test case

Step 3: Set up JabRef's code style

Contributions to JabRef's source code need to have a code formatting that is consistent with existing source code. For that purpose, JabRef provides code-style and check-style definitions.

Install the CheckStyle-IDEA plugin, it can be found via the plug-in repository: Navigate to **File > Settings… > Plugins"**. On the top, click on "Marketplace". Then, search for "Checkstyle". Click on "Install" choose "CheckStyle-IDEA".
Note: In some MacBooks, `Settings` can be found at the "IntelliJ" button of the app menu instead of at "File".

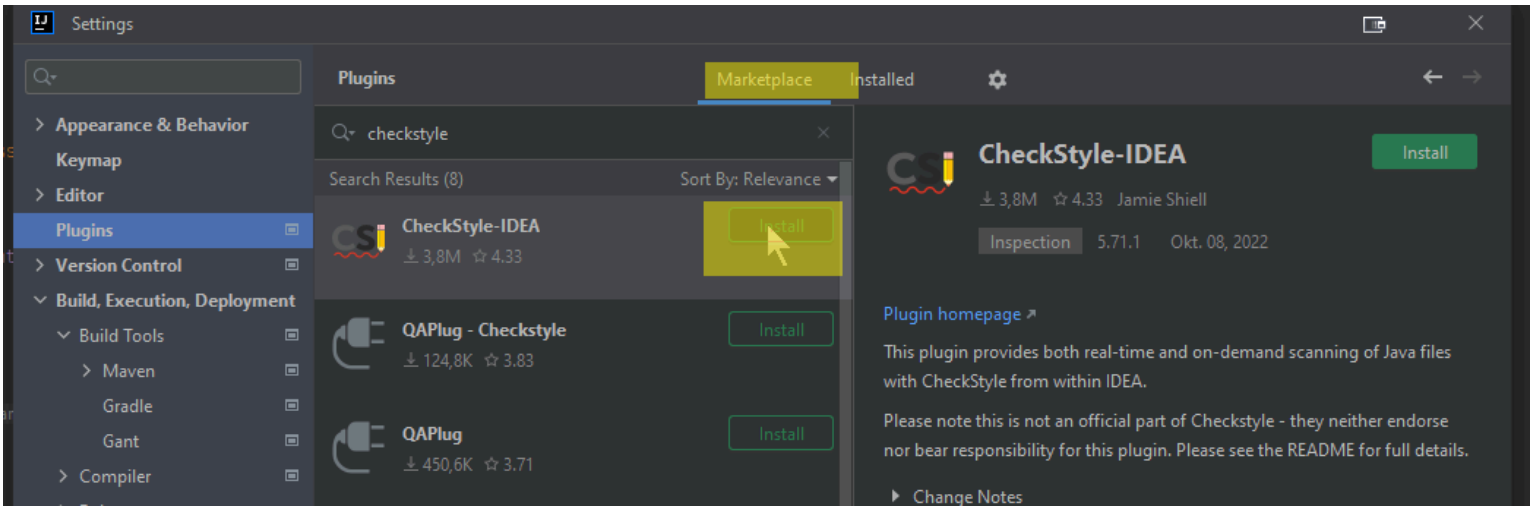


Figure: Install CheckStyle

After clicking, IntelliJ asks for confirmation:

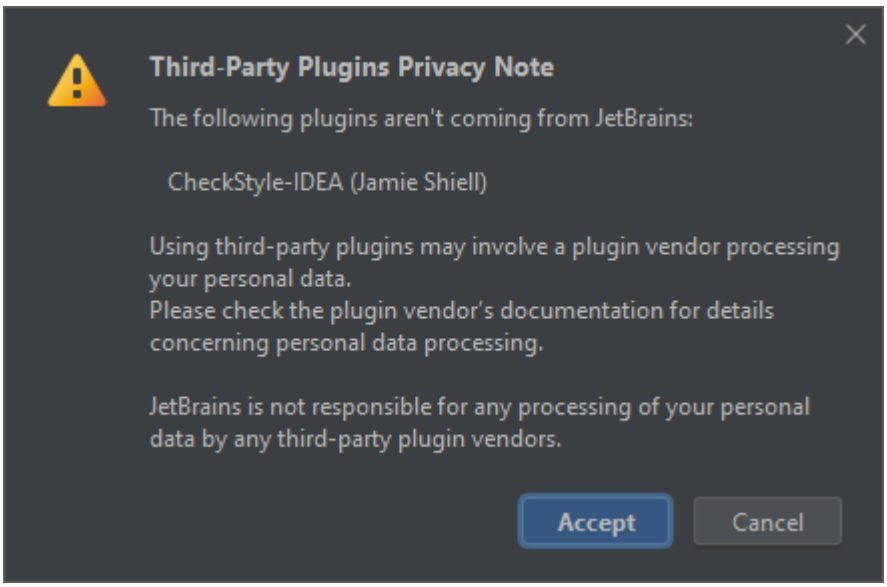


Figure: Third Party Plugin Privacy Notice

If you agree, click on "Agree" and you can continue.

Afterwards, use the "Restart IDE" button to restart IntelliJ.

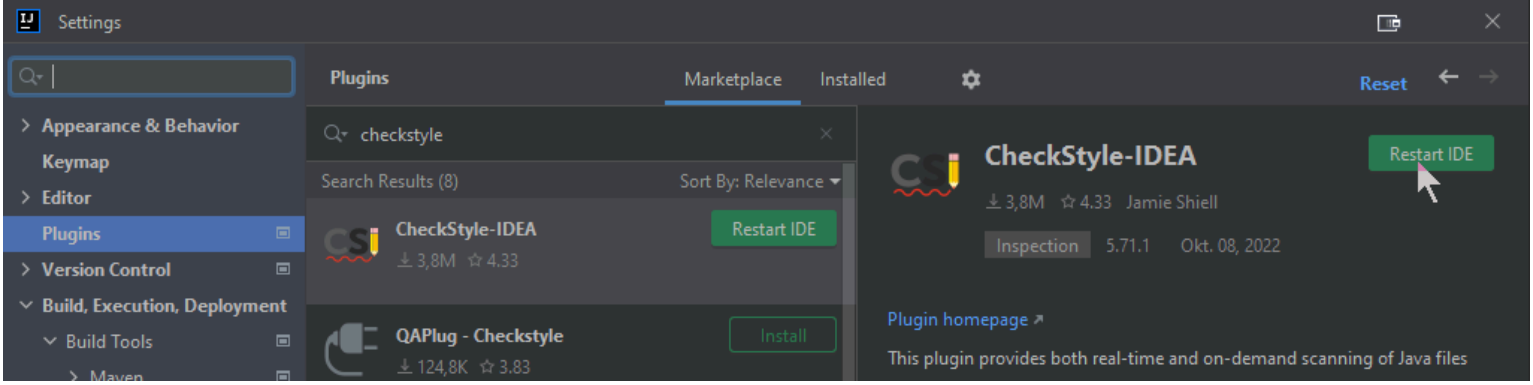


Figure: IntelliJ restart IDE

Click on “Restart” to finally restart.

Wait for IntelliJ coming up again.

Go to **File > Settings… > Editor > Code Style**

Click on the settings wheel (next to the scheme chooser), then click “Import Scheme >”, then click “IntelliJ IDEA code style XML”

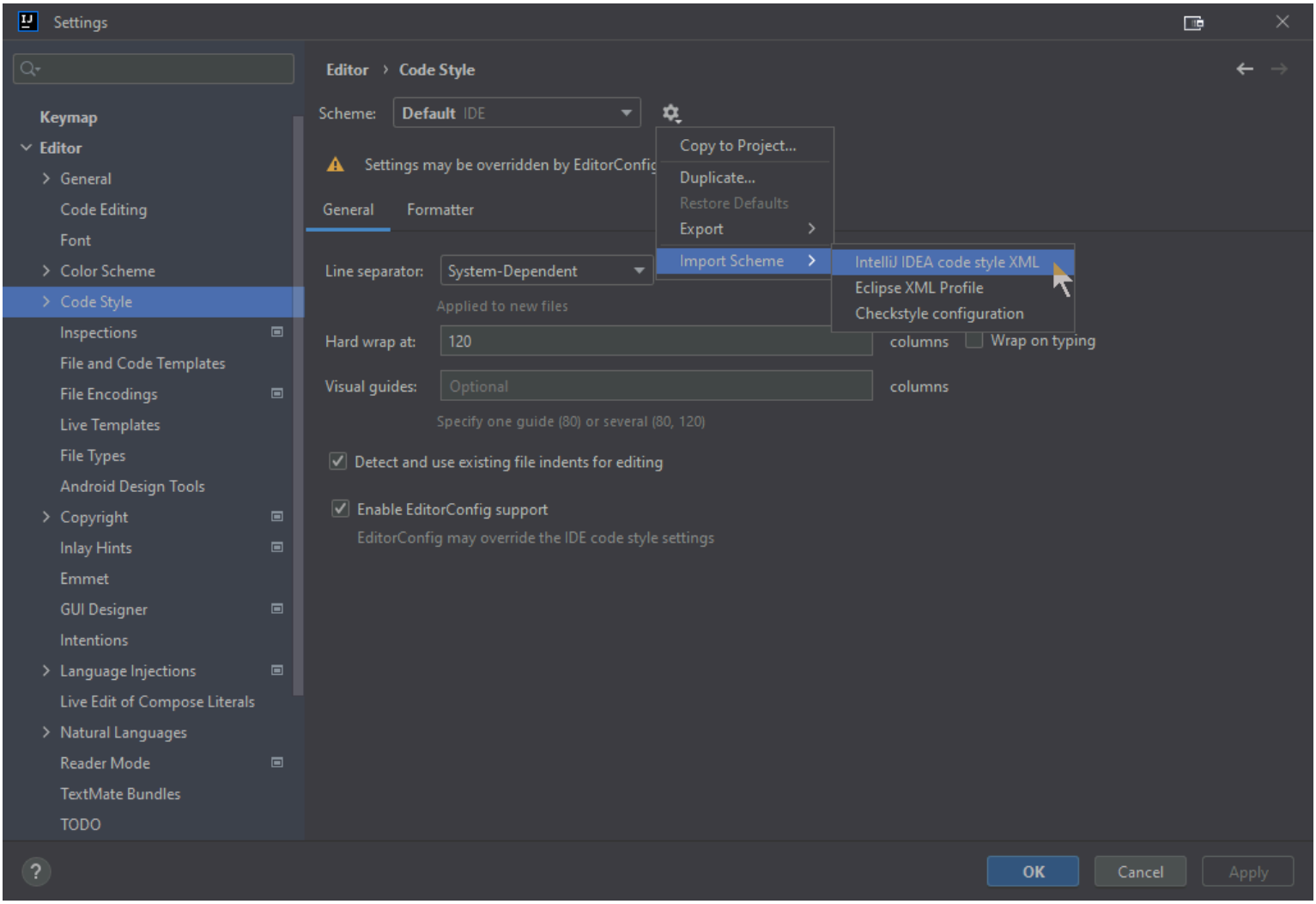


Figure: Location of “Import Scheme > IntelliJ IDEA code style XML”

You have to browse for the directory `config` in JabRef’s code. There is an `IntelliJ Code Style.xml`.

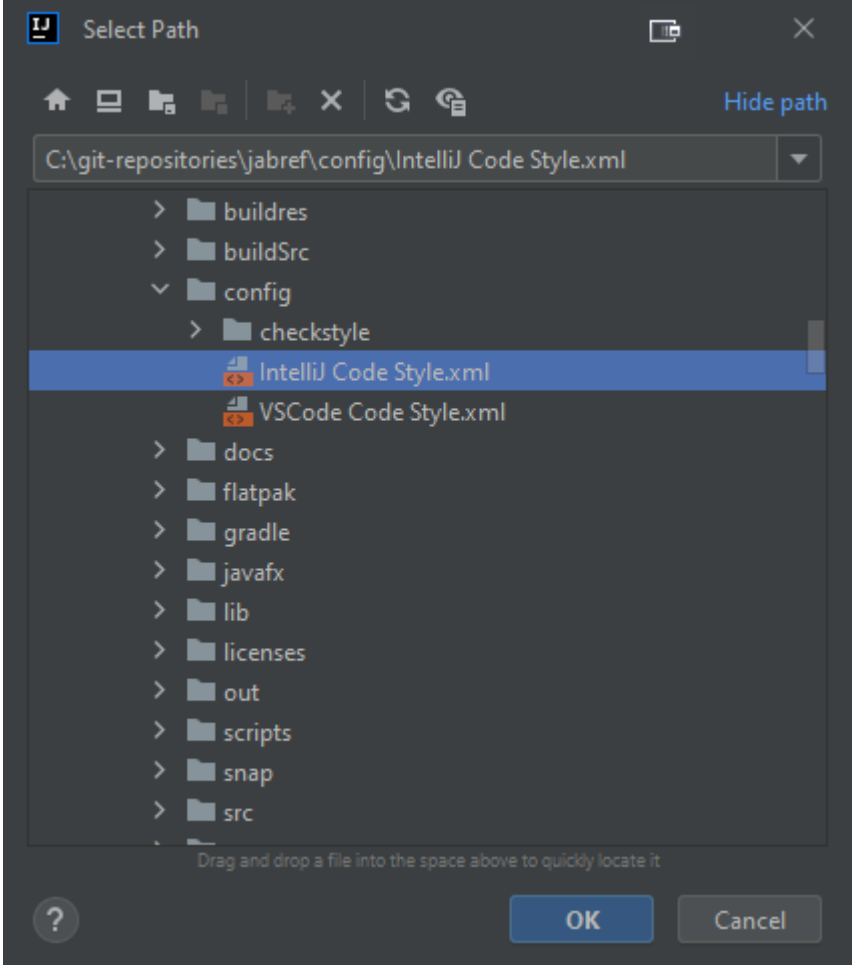


Figure: Browsing for `config/IntelliJ Code Style.xml`

Click “OK”.

At following dialog is “Import Scheme”. Click there “OK”, too.

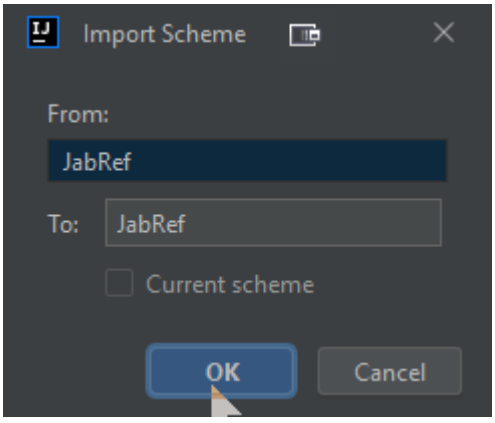


Figure: Import to JabRef

Click on “Apply” to store the preferences.

Put JabRef’s checkstyle configuration in place

Now, put the checkstyle configuration file is in place:

Go to **File > Settings… > Tools > Checkstyle > Configuration File**

Trigger the import dialog of a CheckStyle style by clicking the [+] button:

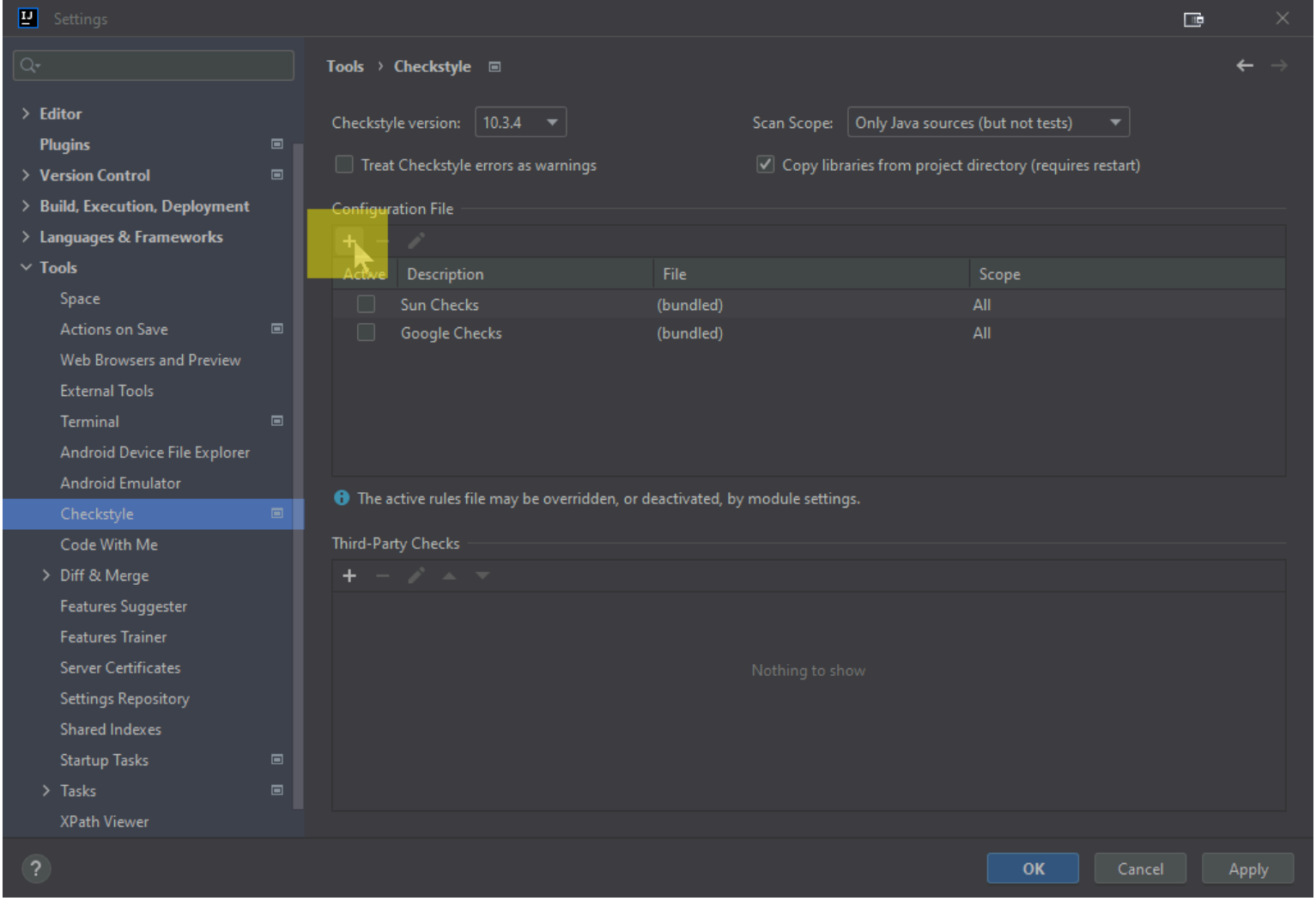


Figure: Trigger the rule import dialog

Then:

- Put “JabRef” as description.
- Browse for `config/checkstyle/checkstyle.xml`
- Tick “Store relative to project location”
- Click “Next”

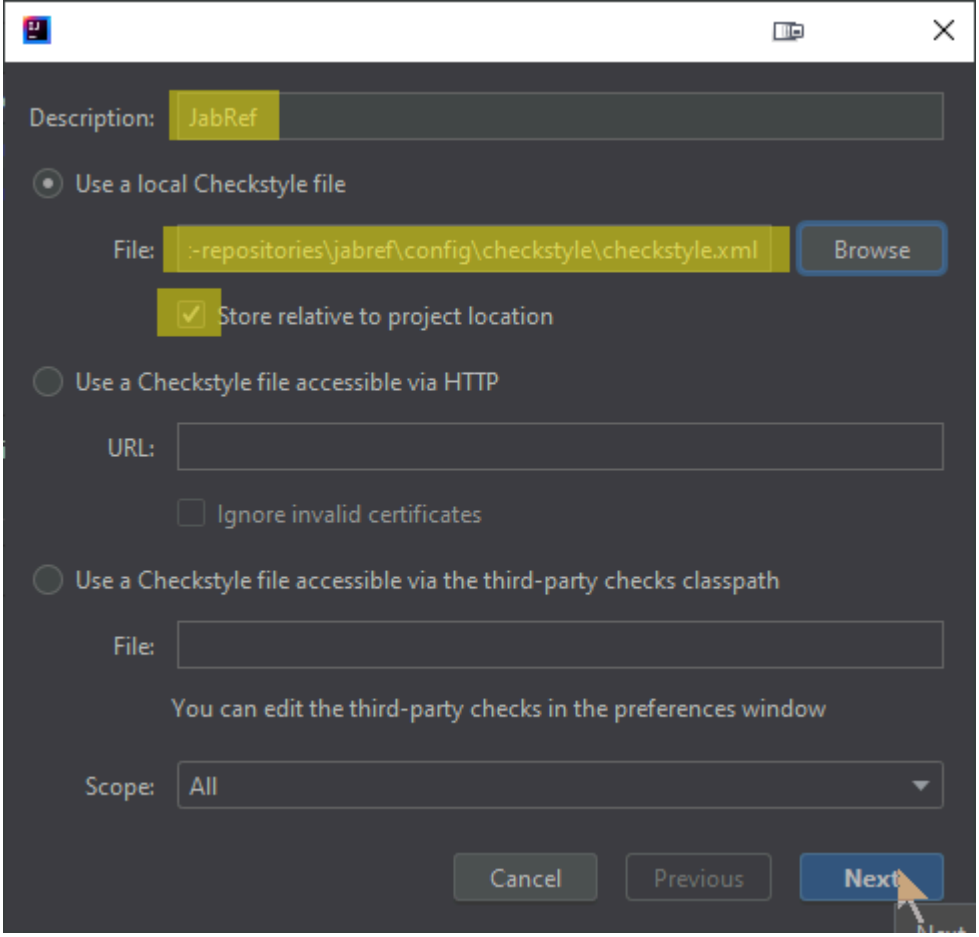


Figure: Filled Rule Import Dialog

Click on “Finish”

Activate the CheckStyle configuration file by ticking it in the list

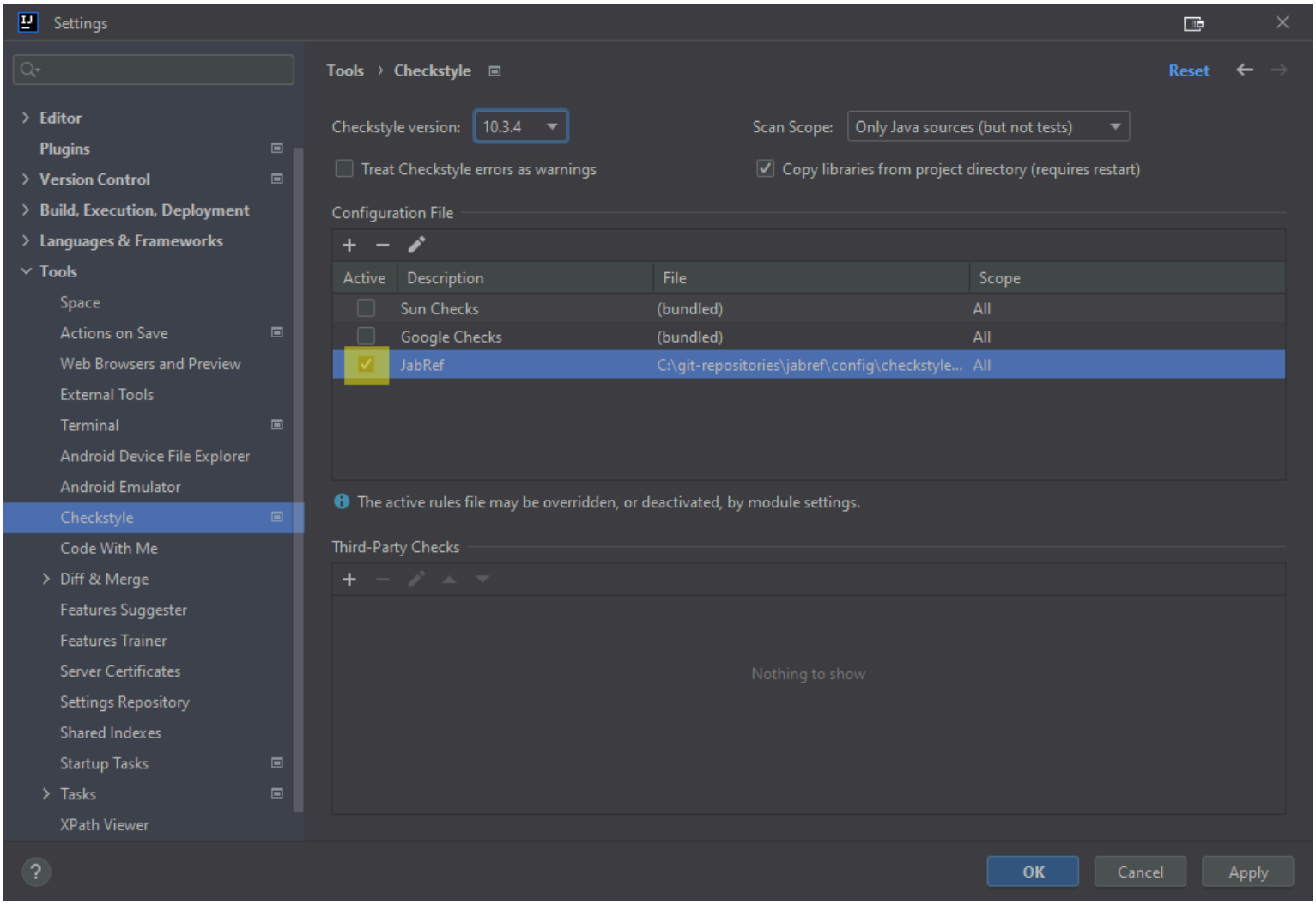


Figure: JabRef's checkstyle config is activated

Ensure that the latest CheckStyle version is selected (10.21.0 or higher). Also, set the "Scan Scope" to "Only Java sources (including tests)".

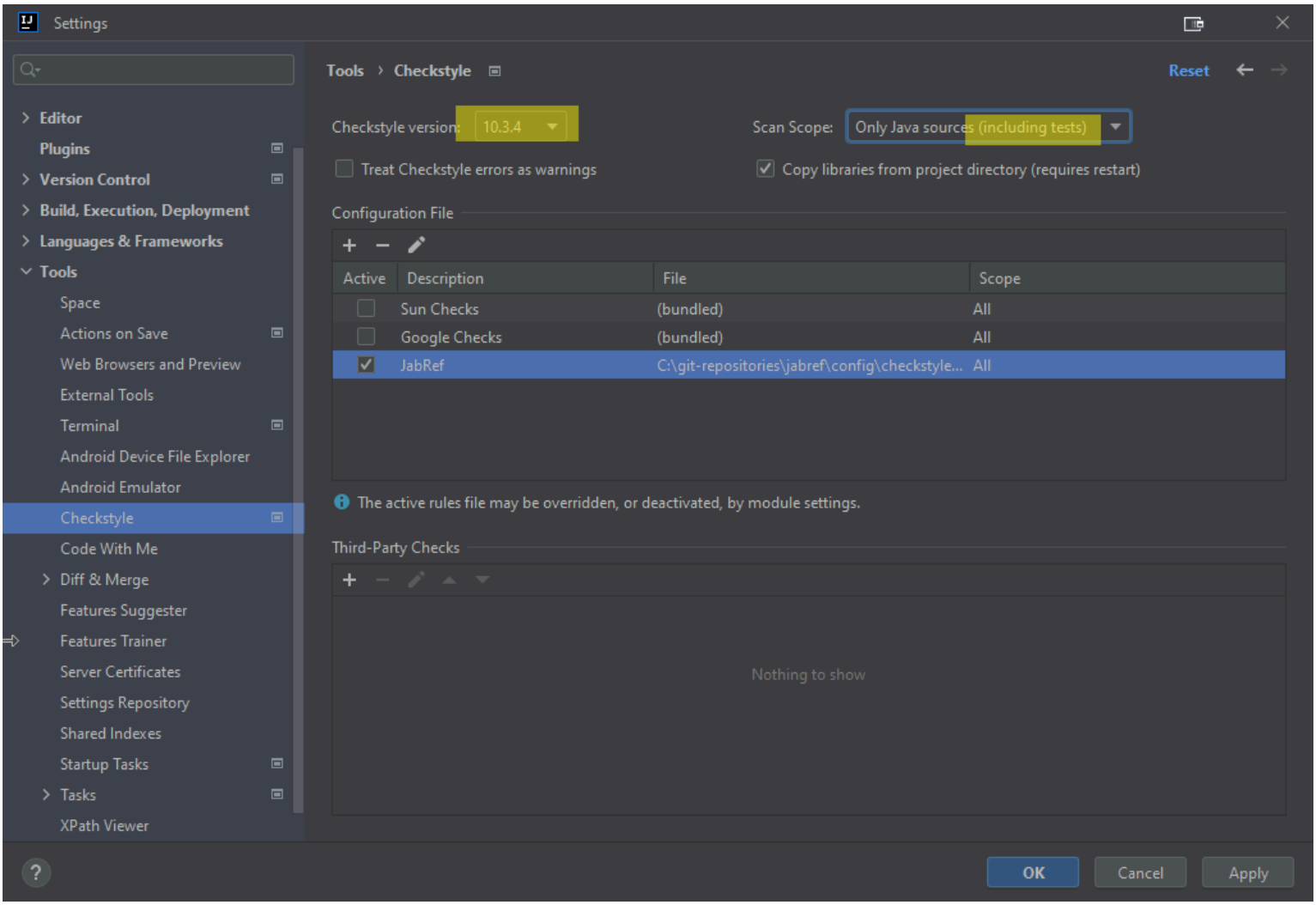


Figure: Checkstyle is the highest version - and tests are also scanned

Save settings by clicking "Apply" and then "OK"

Run checkstyle

In the lower part of IntelliJ's window, click on "Checkstyle". In "Rules", change to "JabRef". Then, you can run a check on all modified files.

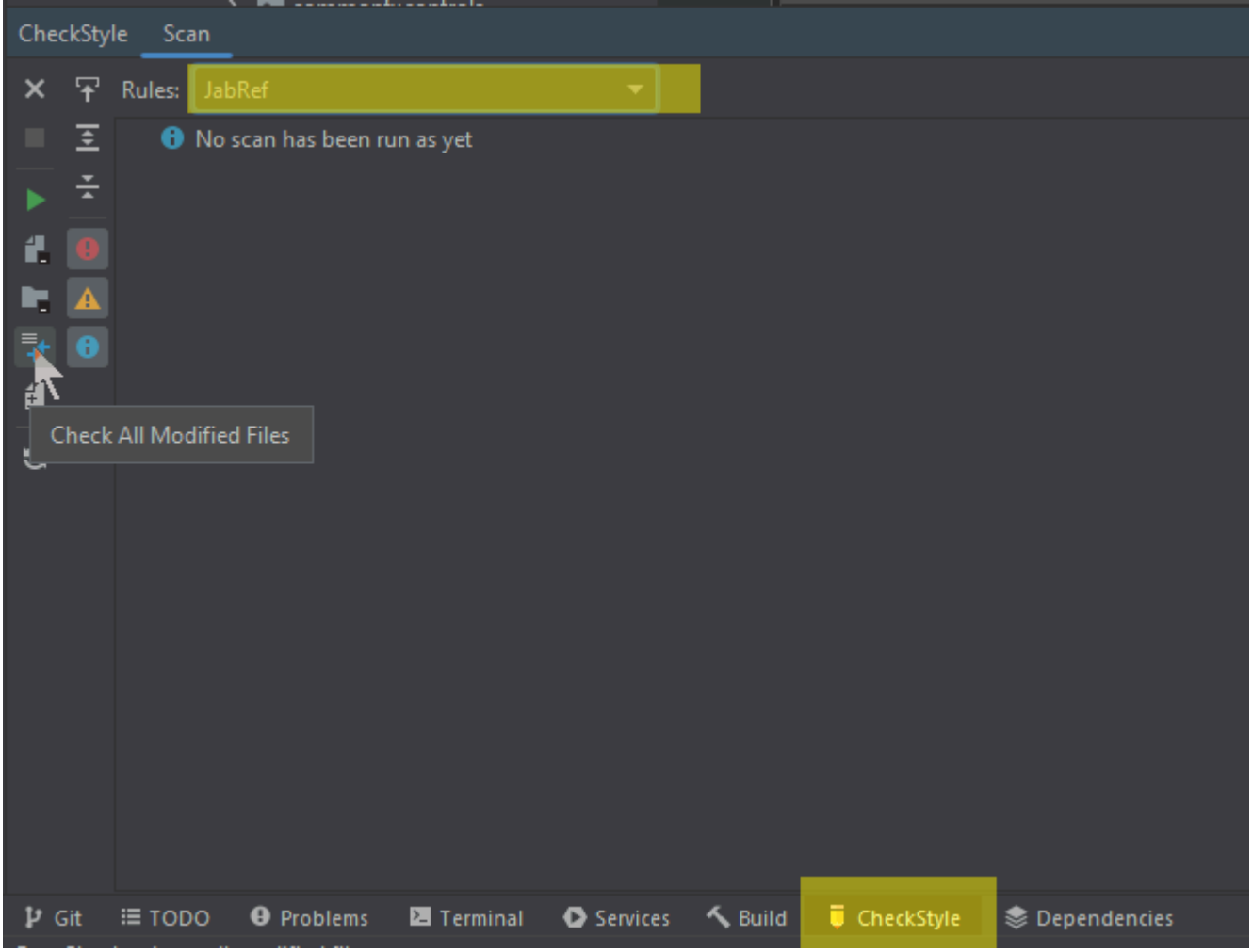


Figure: JabRef's style is active - and we are ready to run a check on all modified files

Have auto format working properly in JavaDoc

To have auto format working properly in the context of JavaDoc and line wrapping, "Wrap at right margin" has to be disabled. Details are found in IntelliJ issue 240517.

Go to **File > Settings… > Editor > Code Style > Java > JavaDoc**.

At "Other", disable "Wrap at right margin"

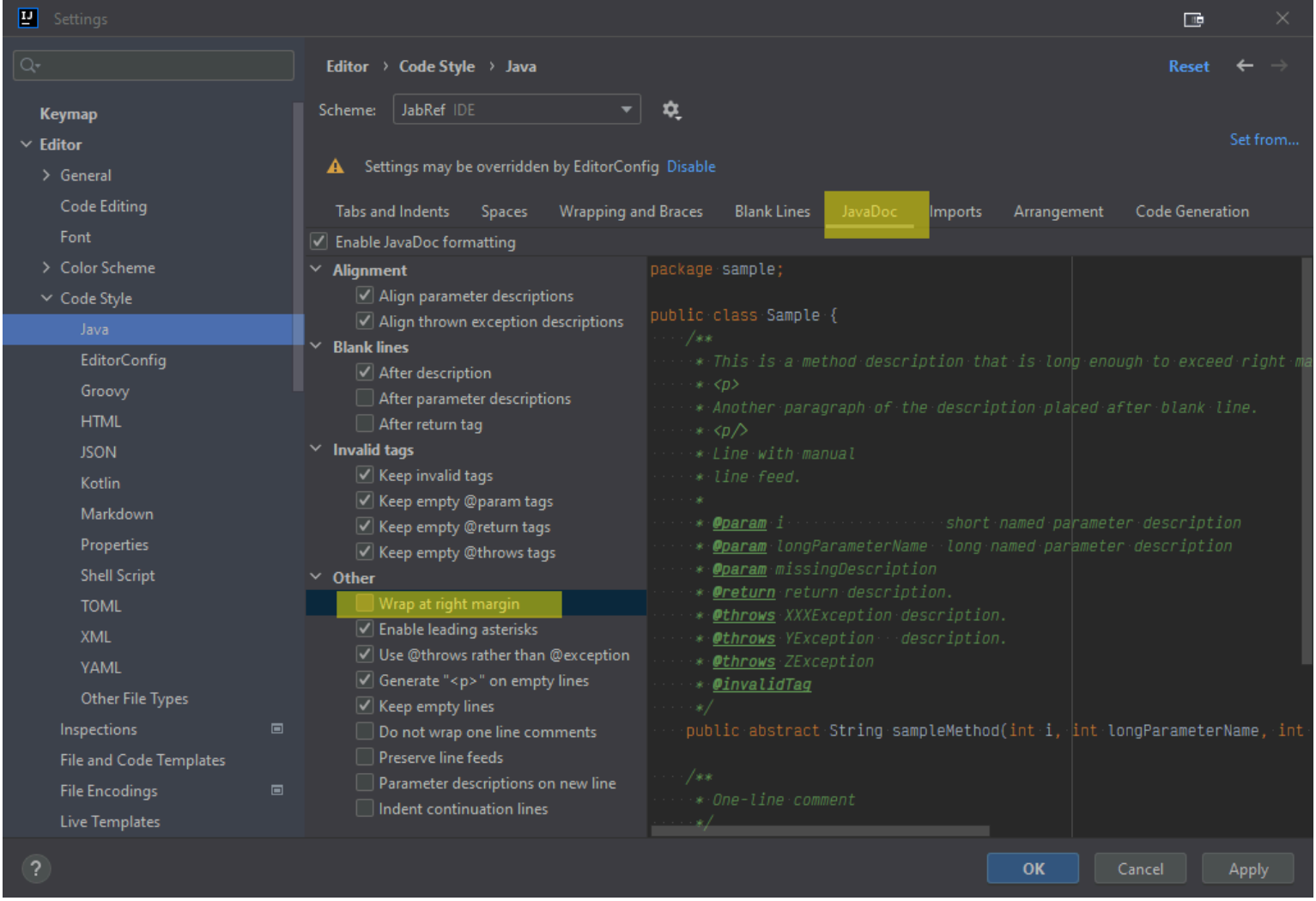


Figure: ”Wrap at right margin” disabled

Enable proper import cleanup

To enable “magic” creation and auto cleanup of imports, go to **File > Settings… > Editor > General > Auto Import**. There, enable both “Add unambiguous imports on the fly” and “Optimize imports on the fly” (Source: JetBrains help).

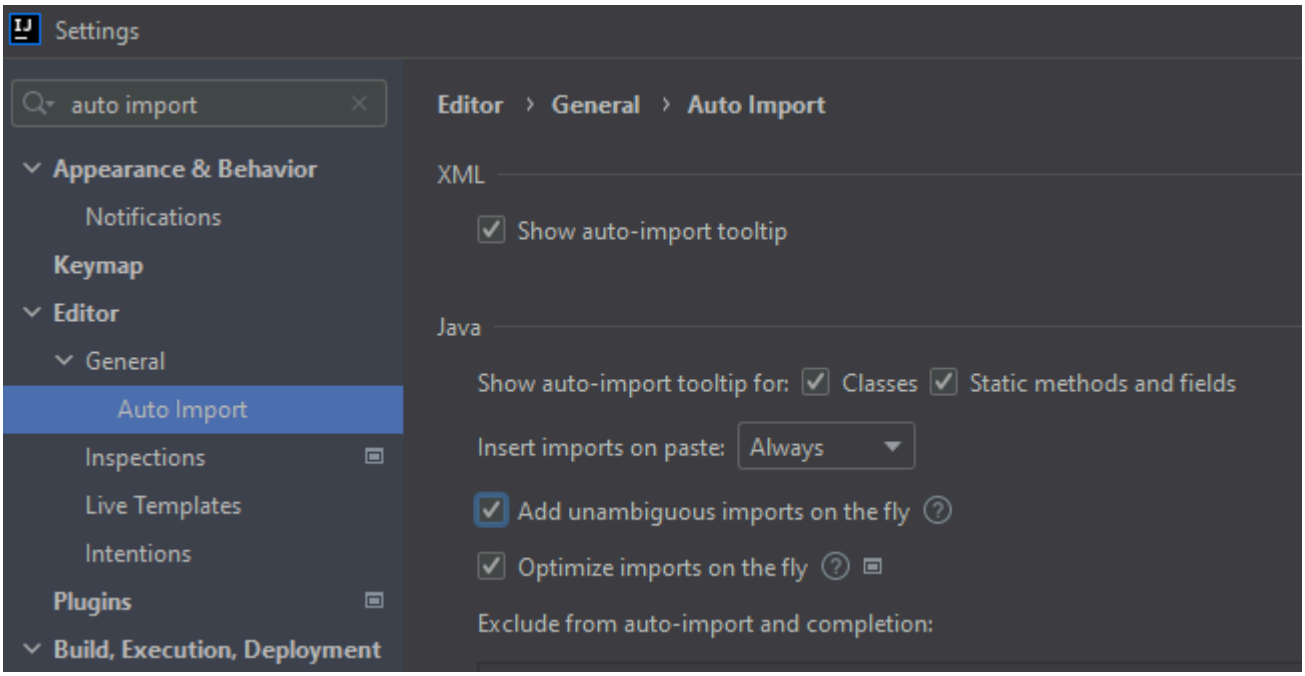


Figure: Auto import enabled

Press “OK”.

Disable too advanced code folding

Go to **File > Settings… > Editor > General > Code Folding**. At section “General”, disable “File header” and “Imports”. At section “Java”, disable “One-line methods”.

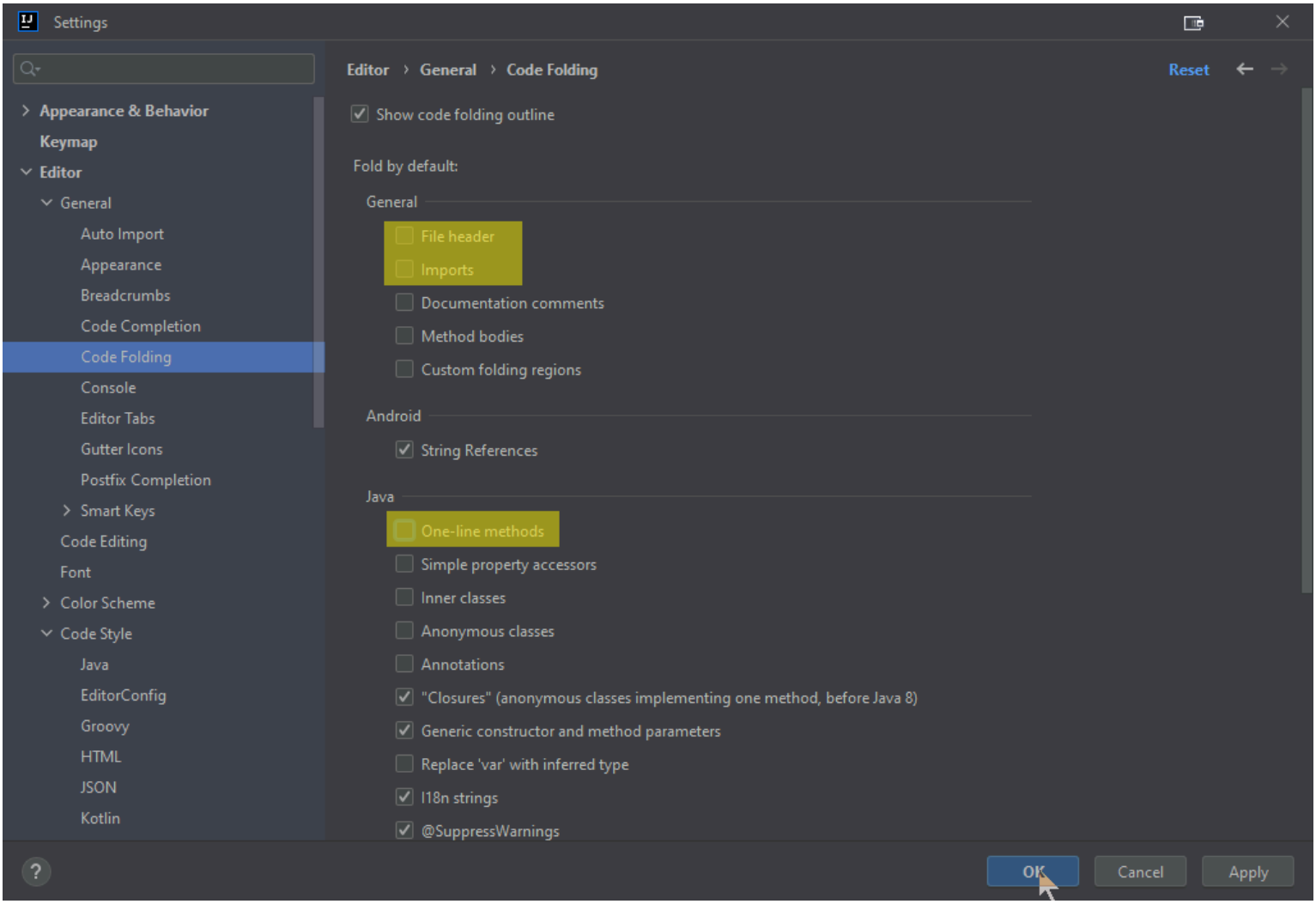


Figure: Code foldings disabled

Press “OK”.

Final comments

SUMMARY

Now you have configured IntelliJ completely. You can run the main application using Gradle and the test cases using IntelliJ. The code formatting rules are imported - and the most common styling issue at imports is automatically resolved by IntelliJ. Finally, you have Checkstyle running locally so that you can check for styling errors before submitting the pull request.

Got it running? GREAT! You are ready to lurk the code and contribute to JabRef. Please make sure to also read our contribution guide.

Advanced: Build and run using IntelliJ IDEA

In “Step 2: Setup the build system: JDK and Gradle”, IntelliJ was configured to use Gradle as tool for launching JabRef. It is also possible to use IntelliJ’s internal build and run system to launch JabRef. Due to IDEA-119280, it is a bit more work.

1. Navigate to **File > Settings… > Build, Execution, Deployment > Build Tools > Gradle**.
2. Change the setting “Build and run using:” to “IntelliJ IDEA”.
3. Navigate to **File > Settings… > Build, Execution, Deployment > Compiler**.
4. Uncheck `Clear output directory on rebuild`.
5. Navigate to **File > Settings… > Build, Execution, Deployment > Compiler > Java Compiler**.
6. Uncheck `--Use 'release' option for cross-compilation`.
7. Click “OK” to store the preferences and close the dialog.
8. **Build > Build Project** (`Ctrl+F9`)
9. Open the project view (`Alt+1`, on macOS `cmd+1`)
10. Copy all build resources to the folder of the build classes
    a. Navigate to the folder `build/resources/main`
    b. Right click -> “Open In” -> “Explorer (Finder on macOS)”
    c. Navigate into directory “main”
    d. Select the folder `out/production/classes`
    e. Right click -> “Open In” -> “Explorer (Finder on macOS)”
    f. Navigate into directory “classes”
    g. Now you have two Explorer windows opened. Copy all files and directories from the first one to the second one.
11. Locate the class `Launcher` (e.g., by `ctrl+N` and then typing `Launcher`). Press `Enter` to jump to that class.

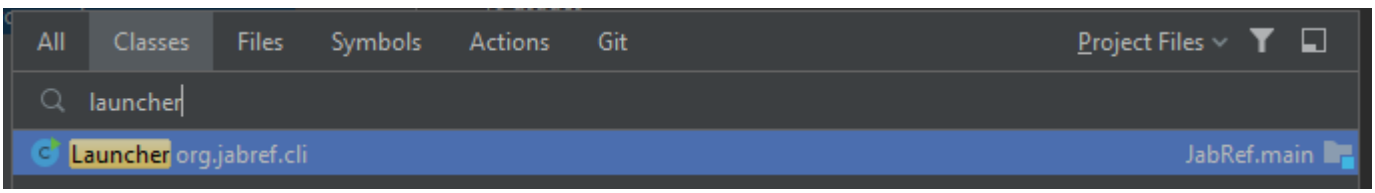


Figure: IntelliJ search for class “Launcher”

1. Click on the green play button next to the `main` method to create a Launch configuration. IntelliJ will fail in launching.

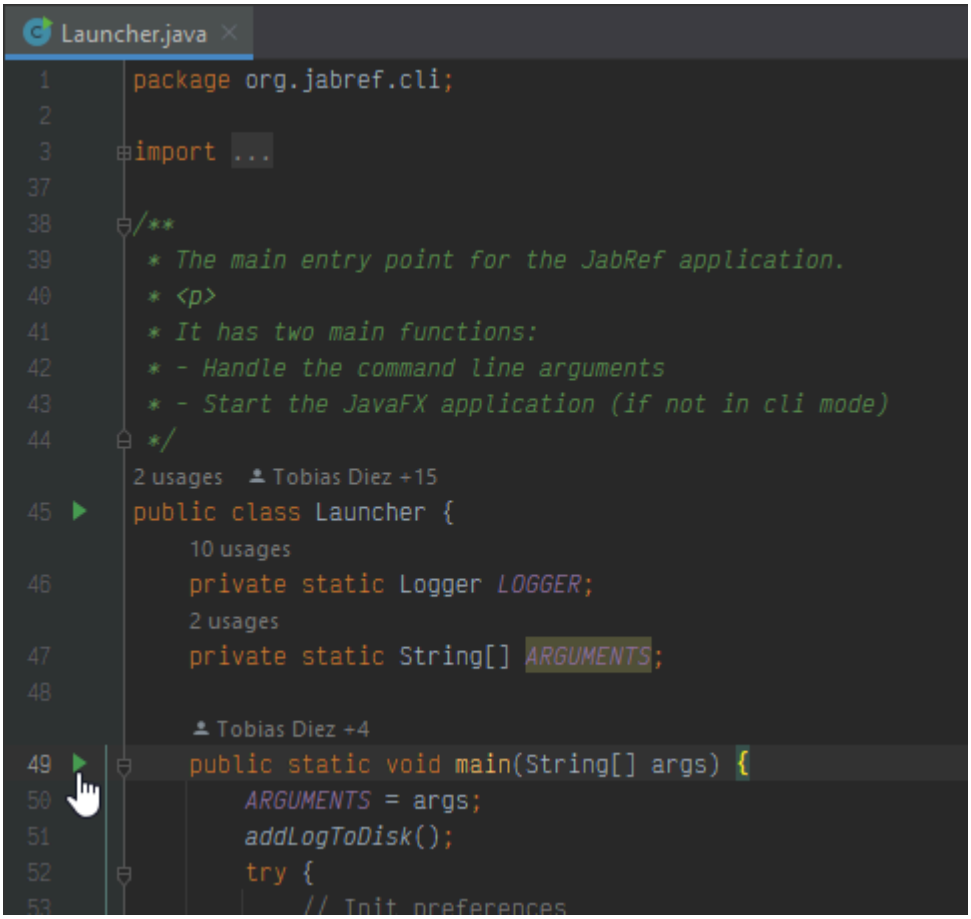


Figure: However on green play

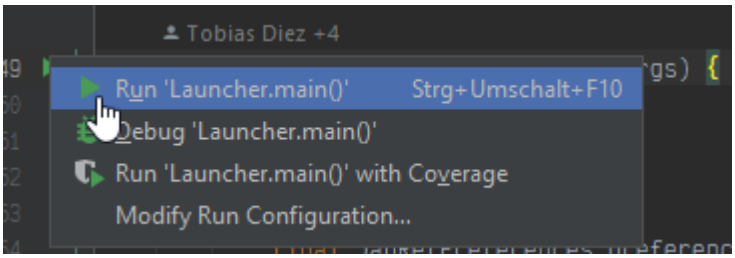


Figure: Run JabRef via launcher

1. On the top right of the IntelliJ window, next to the newly created launch configuration, click on the drop down
2. Click on “Edit Configurations…”
3. On the right, click on “Modify options”
4. Ensure that “Use classpath of module” is checked
5. Select “Add VM options”
6. In the newly appearing field for VM options, insert:

```
--add-exports=javafx.controls/com.sun.javafx.scene.control=org.jabref
--add-opens=org.controlsfx.controls/org.controlsfx.control.textfield=org.jabref
--add-exports=org.controlsfx.controls/impl.org.controlsfx.skin=org.jabref
--add-exports javafx.controls/com.sun.javafx.scene.control=org.jabref
--add-exports org.controlsfx.controls/impl.org.controlsfx.skin=org.jabref
--add-exports javafx.graphics/com.sun.javafx.scene=org.controlsfx.controls
--add-opens javafx.graphics/javafx.scene=org.controlsfx.controls
```

```
--add-exports javafx.graphics/com.sun.javafx.scene.traversal=org.controlsfx.controls
--add-exports javafx.graphics/com.sun.javafx.css=org.controlsfx.controls
--add-exports javafx.controls/com.sun.javafx.scene.control.behavior=org.controlsfx.controls
--add-exports javafx.controls/com.sun.javafx.scene.control=org.controlsfx.controls
--add-opens=javafx.controls/javafx.scene.control.skin=org.controlsfx.controls
--add-exports javafx.controls/com.sun.javafx.scene.control.inputmap=org.controlsfx.controls
--add-exports javafx.base/com.sun.javafx.event=org.controlsfx.controls
--add-exports javafx.base/com.sun.javafx.collections=org.controlsfx.controls
--add-exports javafx.base/com.sun.javafx.runtime=org.controlsfx.controls
--add-exports javafx.web/com.sun.webkit=org.controlsfx.controls
--add-exports javafx.graphics/com.sun.javafx.css=org.controlsfx.controls
--add-reads org.jabref=org.fxmisc.flowless
--add-reads org.jabref=org.apache.commons.csv
```

7 Click “Apply”

8 Click “Run”. You can also click on the debug symbol next to it to enable stopping at breakpoints.

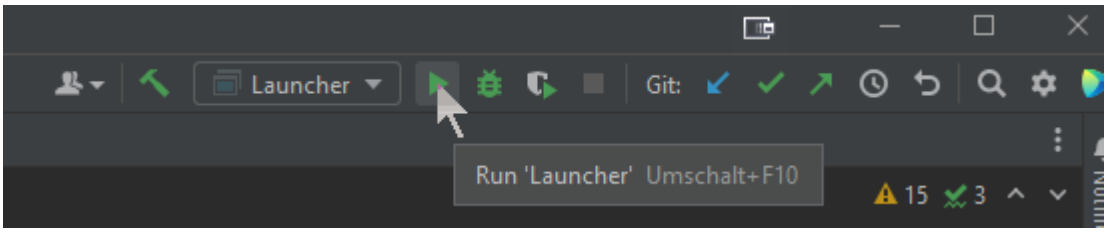


Figure: Launch menu contains “Launcher”

Pre Condition 1: GitHub Account

If you do not yet have a GitHub account, please create one.

Proposals for account names:

- Login similar to your university account. Example: `koppor`
- Use your last name prefixed by the first letter of your first name. Example: `okopp`
- Use `firstname.lastname`. Example: `oliver.kopp`

You can hide your email address by following the recommendations at https://saraford.net/2017/02/19/how-to-hide-your-email-address-in-your-git-commits-but-still-get-contributions-to-show-up-on-your-github-profile-050/.

Most developers, though, do not hide their email address. They use one which may get public. Mostly, they create a new email account for development only. That account then be used for development mailing lists, mail exchange with other developers, etc.

Examples:

- Same login as in GitHub (see above). Example: `koppor@gmail.com`
- “`it`” in the name. Example: `kopp.it@gmail.com`
- Use the university login. Example: `st342435@stud.uni-stuttgart.de`

Pre Condition 2: Required Software

git

It is strongly recommended that you have git installed.

- On Debian-based distros: `sudo apt-get install git`
- On Windows: Download the installer and install it. Using chocolatey, you can run `choco install git.install -y --params "/GitAndUnixToolsOnPath /WindowsTerminal` to a) install git and b) have Linux commands such as `grep` available in your `PATH`.
- Official installation instructions

Installed IDE

We highly encourdage IntelliJ IDEA, because all other IDEs work less good. Especially using VS.Code has issues.

IntelliJ's Community Edition works well. Most contributors use the Ultimate Edition, because they are students getting that edition for free.

Other Tooling

We collected some other tooling recommendations. We invite you to read on at our tool recommendations.

Pre Condition 3: Code on the local machine

This section explains how you get the JabRef code onto your machine in a form allowing you to make contributions.

Fork JabRef into your GitHub account

1. Log into your GitHub account
2. Go to https://github.com/JabRef/jabref
3. Create a fork by clicking at fork button on the right top corner
4. A fork repository will be created under your account `https://github.com/YOUR_USERNAME/jabref`.

A longer explanation is available at https://help.github.com/en/articles/fork-a-repo.

Clone your forked repository on your local machine

In a command line, navigate to the folder where you want to place the source code (parent folder of `jabref`). To prevent issues along the way, it is strongly recommend choosing a path that does not contain any special (non-ASCII or whitespace) characters. In the following, we will use `c:\git-repositories` as base folder:

```
cd \
mkdir git-repositories
cd git-repositories
git clone --recurse-submodules https://github.com/JabRef/jabref.git JabRef
cd JabRef
git remote rename origin upstream
git remote add origin https://github.com/YOUR_USERNAME/jabref.git
git fetch --all
git branch --set-upstream-to=origin/main main
```

IMPORTANT

`--recurse-submodules` is necessary to have the required files available to JabRef. (Background: It concerns the files from citation-style-language/styles and more).

Note that putting the repo JabRef directly on `C:\` or any other drive letter on Windows causes compile errors (**negative example**: `C:\jabref`).

Please really ensure that you pass `JabRef` as parameter. Otherwise, you will get `java.lang.IllegalStateException: Module entity with name: jabref should be available`. See IDEA-317606 for details.

BACKGROUND

Initial cloning might be very slow (`27.00 KiB/s`).

To prevent this, first the `upstream` repository is cloned. This repository seems to live in the caches of GitHub.

Now, you have two remote repositories, where `origin` is yours and `upstream` is the one of the JabRef organization.

You can see it with `git remote -v`:

```
c:\git-repositories\jabref> git remote -v
origin     https://github.com/YOURUSERNAME/jabref.git (fetch)
origin     https://github.com/YOURUSERNAME/jabref.git (push)
upstream     https://github.com/jabref/jabref.git (fetch)
upstream     https://github.com/jabref/jabref.git (push)
```

Trouble shooting

Changes in `src/main/resources/csl-styles` are shown

You need to remove these directories from the “Directory Mappings” in IntelliJ. Look for the setting in preferences. A long how-to is contained in Step 1: Get the code into IntelliJ.

Issues with `buildSrc`

1. Open the context menu of `buildSrc`.
2. Select “Load/Unload modules”.
3. Unload `jabRef.buildSrc`.

Issues with generated source files

In rare cases you might encounter problems due to out-dated automatically generated source files. Running gradle task “clean” (Command line: `./gradlew clean`) deletes these old copies. Do not forget to run at least `./gradlew assemble` or `./gradlew eclipse` afterwards to regenerate the source files.

Issue with “Module org.jsoup” not found, required by org.jabref

Following error message appears:

```
Error occurred during initialization of boot layer
java.lang.module.FindException: Module org.jsoup not found, required by org.jabref
```

This can include different modules.

1. Go to File -> Invalidate caches...
2. Check “Clear file system cache and Local History”.
3. Check “Clear VCS Log caches and indexes”.
4. Uncheck the others.
5. Click on “Invalidate and Restart”.
6. After IntelliJ restarted, you have to do the “buildSrc”, “Log4JAppender”, and “src-gen” steps again.

Issues with OpenJFX libraries in local maven repository

There might be problems with building if you have OpenJFX libraries in local maven repository, resulting in errors like this:

```
> Could not find javafx-fxml-20-mac.jar (org.openjfx:javafx-fxml:20).
    Searched in the following locations:
        file:<your local maven repository path>/repository/org/openjfx/javafx-fxml/20/javafx-fxml-20-ma
```

As a workaround, you can remove all local OpenJFX artifacts by deleting the whole OpenJFX folder from specified location.

Issues with `JournalAbbreviationLoader`

In case of a NPE at `Files.copy` at `org.jabref.logic.journals.JournalAbbreviationLoader.loadRepository(JournalAbbreviationLoader.java:30) ~[classes/:?]`, invalidate caches and restart IntelliJ. Then, Build -> Rebuild Project.

If that does not help:

1. Save/Commit all your work
2. Close IntelliJ
3. Delete all non-versioned items: `git clean -xdf`. This really destroys data
4. Execute `./gradlew run`
5. Start IntelliJ and try again.

Java installation

An indication that `JAVA_HOME` is not correctly set or no JDK 21 is installed in the IDE is following error message:

```
compileJava FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':compileJava'.
> java.lang.ExceptionInInitializerError (no error message)
```

Another indication is following output

```
java.lang.UnsupportedClassVersionError: org/javamodularity/moduleplugin/ModuleSystemPlugin has been comp
```

Attempts to open preferences panel freezes application

This is likely caused by improper integration of your OS or Desktop Environment with your password prompting program or password manager. Ensure that these are working properly, then restart your machine and attempt to run the program.

In an ideal scenario, a password prompt should appear when the program starts, provided the keyring your OS uses has not already been unlocked. However, the implementation details vary depending on the operating system, which makes troubleshooting more complex.

For Windows and macOS users, specific configurations may differ based on the password management tools and settings used, so ensure your OS's password management system is properly set up and functioning.

For Linux users, ensure that your xdg-desktop-portal settings refer to active and valid portal implementations installed on your system. However, there might be other factors involved, so additional research or guidance specific to your distribution may be necessary.

For reference, see the discussion at issue #11766.

Advanced: VS Code as IDE

We are working on supporting VS Code for development. There is basic support, but important things such as our code conventions are not in place. Thus, use at your own risk.

Quick howto:

1. Start VS Code in the JabRef directory: `code .`.
2. There will be a poup asking “Reopen in Container”. Click on that link.
3. VS Code restarts. Wait about 3 minutes until the dev container is build. You can click on “Starting Dev Container (show log)” to see the progress.
4. Afterwards, the Java project is imported. You can open the log (Click on “Check details”). Do that.
5. The terminal (tab “Java Build Status”) will show some project synchronization and hang at `80% [797/1000]`. It keeps hanging at `Importing root project: 80% Refreshing '/jabref'`. Just wait. Then it hangs at `Synchronizing Gradle build at /workspaces/jabref: 80%`. Just wait. Then it takes long for `Refreshing workspace:`. Just wait. **Note:** If you had the project opened in IntelliJ before, this might cause issues (as outlined at https://issuetracker.google.com/issues/255903901?pli=1). Close everything, ensure that you committed your changes (if any), then execute `git clean -xdf` to wipe out all changes and created files - and start from step 1 again.
6. On the left, you will see a gradle button.
7. Click on the gradle button and open **JabRef -> Tasks -> application**.
8. Double click on **run**.
9. In the terminal, a new tab “run” opens.
10. On your desktop machine, open http://127.0.0.1:6080/ in a web browser. Do not open the proposed port `6050`. This is JabRef’s remote command port.
11. Use `vscode` as password.
12. You will see an opened JabRef.

Alternative to steps 9 to 10:

In case interaction using the web browser is too slow, you can use a VNC connection:

1. Install VNC Connect
2. Use `vscode` as password

Trouble shooting

In case there are reading errors on the file system, the docker container probably is out of order. Close VS Code. Stop the docker container, kill docker process in the Task Manager (if necessary). Start docker again. Start VS Code again.

Background

We use VS Code's Dev Containers feature. Thereby, we use desktop-lite to enable viewing the JabRef app.

Set up a local workspace

> **IMPORTANT**
> These steps are very important. They allow you to focus on the content and ensure that the code formatting always goes well.

This guide explains how to set up your environment for development of JabRef. It includes information about prerequisites, configuring your IDE, and running JabRef locally to verify your setup. Please follow the steps one-by-one.

First, we work on prerequisites (software, account, code fork) you need to get started to develop JabRef. Then, we work on a proper IDE setup.

TABLE OF CONTENTS

- Pre Condition 1: GitHub Account
- Pre Condition 2: Required Software
- Pre Condition 3: Code on the local machine
- Step 1: Get the code into IntelliJ
- Step 2: Set up the build system: JDK and Gradle
- Step 3: Set up JabRef’s code style
- Advanced: Build and run using IntelliJ IDEA
- Advanced: Eclipse as IDE
- Advanced: VS Code as IDE
- Trouble shooting

High-level documentation

This page describes relevant information about the code structure of JabRef precisely and succinctly. Closer-to-code documentation is available at Code HowTos.

We have been successfully transitioning from a spaghetti to a more structured architecture with the `model` in the center, and the `logic` as an intermediate layer towards the `gui` which is the outer shell. There are additional utility packages for `preferences` and the `cli`. The dependencies are only directed towards the center. We have JUnit tests to detect violations of the most crucial dependencies (between `logic`, `model`, and `gui`), and the build will fail automatically in these cases.

The `model` represents the most important data structures (`BibDatases`, `BibEntries`, `Events`, and related aspects) and has only a little bit of logic attached. The `logic` is responsible for reading/writing/importing/exporting and manipulating the `model`, and it is structured often as an API the `gui` can call and use. Only the `gui` knows the user and their preferences and can interact with them to help them solving tasks. For each layer, we form packages according to their responsibility, i.e., vertical structuring. The `model` should have no dependencies to other classes of JabRef and the `logic` should only depend on `model` classes. The `cli` package bundles classes that are responsible for JabRef's command line interface. The `preferences` package represents all information customizable by a user for her personal needs.

We use an event bus to publish events from the `model` to the other layers. This allows us to keep the architecture but still react upon changes within the core in the outer layers. Note that we are currently switching to JavaFX's observables, as this concepts seems as we aim for a stronger coupling to the data producers.

Package Structure

Permitted dependencies in our architecture are:

```
gui --> logic --> model

gui ------------> model

gui ------------> preferences

gui ------------> cli

gui ------------> global classes


logic ------------> model


global classes ------------> everywhere
```

```
cli ------------> model
cli ------------> logic
cli ------------> global classes
cli ------------> preferences
```

All packages and classes which are currently not part of these packages (we are still in the process of structuring) are considered as gui classes from a dependency stand of view.

Most Important Classes and their Relation

Both GUI and CLI are started via the `JabRefMain` which will in turn call `JabRef` which then decides whether the GUI (`JabRefFrame`) or the CLI (`JabRefCLI` and a lot of code in `JabRef`) will be started. The `JabRefFrame` represents the Window which contains a `SidePane` on the left used for the fetchers/groups Each tab is a `BasePanel` which has a `SearchBar` at the top, a `MainTable` at the center and a `PreviewPanel` or an `EntryEditor` at the bottom. Any right click on the `MainTable` is handled by the `RightClickMenu`. Each `BasePanel` holds a `BibDatabaseContext` consisting of a `BibDatabase` and the `MetaData`, which are the only relevant data of the currently shown database. A `BibDatabase` has a list of `BibEntries`. Each `BibEntry` has an ID, a citation key and a key/value store for the fields with their values. Interpreted data (such as the type or the file field) is stored in the `TypedBibentry` type. The user can change the `JabRefPreferences` through the `PreferencesDialog`.

Getting into the code

TABLE OF CONTENTS

- JabRef’s development strategy
- Set up a local workspace
- High-level documentation

AI

User Interface

Chatting with AI

`req~ai.chat.new-message-based-on-previous~1`

To enable simple editing and resending of previous messages, `Cursor Up` should show last message. This should only happen if the current text field is empty.

Needs: impl

Requirements

This part of the documentation collects requirements using OpenFastTrace.

Specifying requirements

One writes directly below a Markdown heading a requirement identifier.

Example:

```
### Example
`req~ai.example~1`
```

It is important that there is no empty line directly after the heading.

One needs to add `<!-- markdownlint-disable-file MD022 -->` to the end of the file, because the ID of the requirement needs to follow the heading directly.

Linking implementations

Then, one writes down at the requirement. Directly at the end, one writes that it requires an implementation:

```
Needs: impl
```

One can also state that there should be detailed design document (`dsn`). However, typically in JabRef, we go from the requirement directly to the implementation.

Then, at the implementation, a comment is added this implementation is covered:

```
// [impl->req~ai.example~1]
```

When executing the gradle task `traceRequirements`, `build/tracing.txt` is generated. In case of a tracing error, one can inspect this file to see which requirements were not covered.

More Information

- User manual of OpenFastTrace
- We cannot copy and paste real examples here, because of openfasttrace#280.

TABLE OF CONTENTS

- AI

JabRef and Software Engineering Training

By using JabRef as training object in exercises and labs, students can level-up their coding and project management skills. When taking part in JabRef development, one will learn modern Java coding practices, how code reviews work and how to properly address reviewing feedback.

Why university instructors should cooperate with us?

- High-quality student education due to real-world tooling and real-world code base
- Sustainability of student works: No more thrown-away solved exercises: They now are incorporated in a real-world product
- No need to provision infra structure
- Visibility of your research groups
- No need to think about basic software engineering exercises anymore: JabRef cooperation partners have them.

How to integrate JabRef in your class

1. Choose task from the board Candidates for university projects. There, new functionality is categorized in small, medium, and large effort. Moreover, categorization on the main focus (UI, logic, or both), implementation effort, testing effort, and "issue understanding effort". The latter category is important, because some issues are "quick wins" and others need thorough thinking.

   In general, all issues of JabRef are free to take. Be aware that the difficulty of bugs and feature vary. For the brave, the Bug Board or the Feature Board provide other issue sources. Especially for Master students, these are excellent boards to find issues that train maintenance knowledge (which is essential for industry work). Finally, there is a collection of good first issues, if you search for something to start guiding you though a focused aspect of JabRef's code.

2. Get in touch with the JabRef team to reserve issues for your student group and possibly to discuss details. We offer email, skype, gitter.im, discord. Get in touch with @koppor to find the right channel and to start forming the success of your course.
3. Schedule tasks with students
4. Students implement code
5. Students review other student's code (recommended: students of a previous year's project review current year's project code)

6 Students address review feedback
7 Students submit pull request
8 Code reviews by JabRef maintainers
9 Students address feedback and learn more about good coding practices by incorporating feedback
10 Students update their pull request
11 Pull request is merged

For a near-to-perfect preparation and effect of the course, we ask you to get in touch with us **four weeks** in advance. Then, the JabRef team can a) learn about the starting skill level of the students, b) the aimed skill level at the end of the course, c) the amount of time the students are given to learn about and contribute to JabRef, d) check the Candidates for university projects for appropriate tasks (and fill it as needed), e) recommend appropriate features.

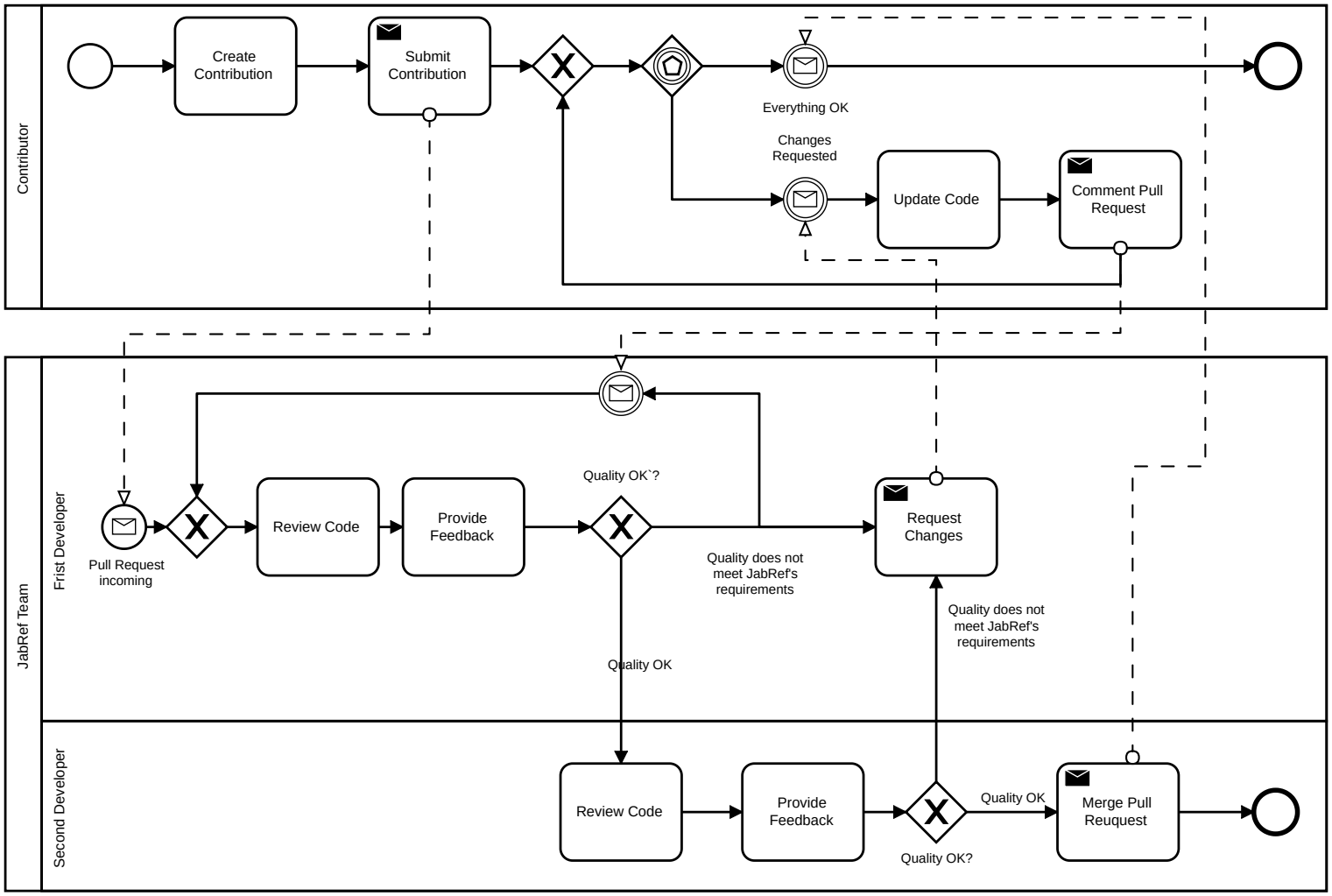
It is also possible to just direct students to our Contribution Guide. The learning effect may be lower as the time of the students has to be spent to a) learn about JabRef and b) select an appropriate issue.

Since a huge fraction of software costs is spent on software maintenance, adding new features also educates in that aspect: perfective maintenance 2 is trained. When fixing bugs, corrective maintenance 2 is trained.

Process for contributions

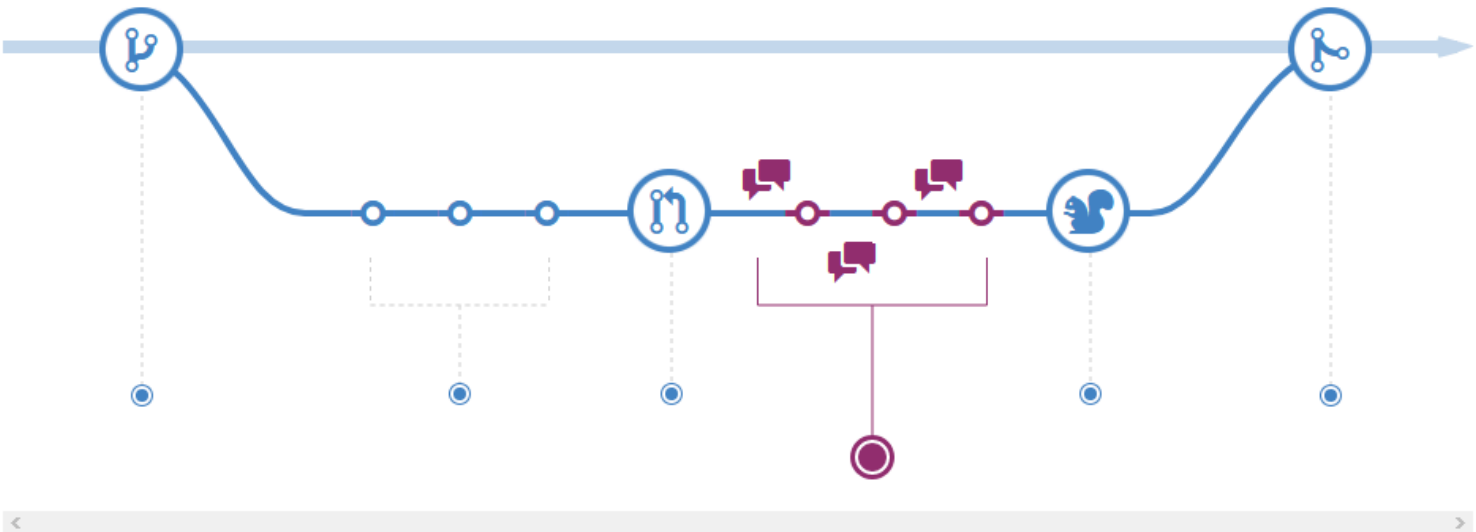
There is no special process for student contributions. We want to discuss it nevertheless to increase awareness of the time required from starting the contribution until the inclusion in a release of JabRef.

The process for accepting contributions is as below. The syntax is BPMN modeled using bpmn.io.



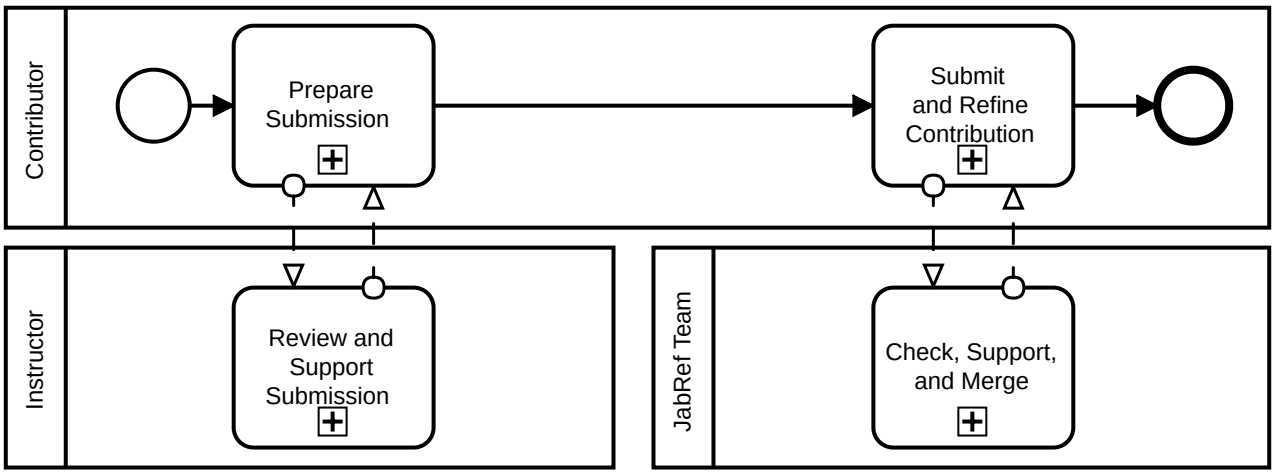
In short, the contribution is **reviewed by two JabRef developers**. Typically, they have constructive feedback on their contribution. This means, that the contributors get comments on their contribution enabling them to level-up their coding skill. Incorporating improvements takes time, too. The benefit is two-fold: a) contributors improve their coding skills and b) JabRef's code quality improves. All in all, we ask to respect the aims of the JabRef team and to reserve time to incorporate the reviewer's comments.

GitHub describes that in their page Understanding the GitHub flow:



Process for Java newcomers

Newcomers contributing in the context of a university teaching experience are invited to follow the process described above. In case the capacity of the instructing university allows, we propose a three-step approach. First, the contributors prepare their contribution as usual. Then, they submit the pull request *to a separate repository*. There, the instructor reviews the pull request and provides feedback. This happens in a loop until the instructor shows the green light. Then, the pull request can be submitted to the main JabRef repository. This will help to reduce the load on the JabRef team and improve the quality of the initial pull request.



Past courses

> In case your course is missing, feel free to add it.

English as course language

HARBIN INSTITUTE OF TECHNOLOGY (HIT), CHINA

Course: Open Source Software Development, 2018, 2019

> In this course, students will be introduced to the processes and tools specific to Open Source Software development, and they will analyze existing projects to understand the architecture and processes of these projects. Besides, students will attempt to contribute source code to a large existing Open Source Software project.

KING'S COLLEGE LONDON

Course: BSc Computer Science Individual Project, 2022/2023

> Students experience the procedure of finding and fixing small and medium issues in an open source project.

NORTHERN ARIZONA UNIVERSITY (NAU), USA

Course CS499 - Open Source Software Development, 2018

> Students experience the process of getting involved in an Open Source project by engaging with a real project. Their goal is to make a "substantial" contribution to a project.

UNIVERSITY OF TENNESSEE, KNOXVILLE, USA

Diversity awareness course by Vandana Singh, 2022

UNIVERSITY OF VICTORIA, CANADA

Course SENG371: Software Evolution by Roberto A. Bittencourt

> Introduces problems and solutions of long-term software maintenance/evolution and large-scale, long-lived software systems. Topics include software engineering techniques for programming-in-the-large, programming-in-the-many, legacy software systems, software architecture, software evolution, software maintenance, reverse engineering, program understanding, software visualization, advanced issues in object-oriented programming, design patterns, antipatterns, and client-server computing. Culminates in a team project.
>
> During the course, students work in small groups to solve three assignments, each assignment handling three JabRef issues. First assignment deals with small bug fixes. Second assignment handles testing and refactoring. Third assignment handles features or bug fixes that deal with both the GUI and the business logic.

German as course language

UNIVERSITÄT BASEL, SWITZERLAND

Course 10915-01: Software Engineering, 2019 to 2023

- Lecture Materials: https://github.com/unibas-marcelluethi/software-engineering
- Excercise touching JabRef:
  - General idea: identify a feature missing in JabRef and develop the specification, system design, and implementation of the feature.
  - Introduction to JabRef's code: Exercise 5: Introduction into JabRef code.
  - Prominent feature implemented: Parse full-text references using Grobid. PR #5614.

UNIVERSITY OF STUTTGART, GERMANY

Course "Softwarepraktikum" as part of the BSc Informatik, 2012

> A group of three students experienced the full software engineering process within one semester. They worked part-time for the project.

Course Studienprojekt as part of the BSc Software Engineering, 2015/2016

> A group of nine students experienced the full software engineering process within one year. They worked part-time for the project.

Course "Programming and Software Development" as part of the BSc Software Engineering, 2018

> One exercise to contribute a minor fix or feature to JabRef. Goal: learn contribution to an open-source project using git and GitHub.

Swedish

KTH ROYAL INSTITUTE OF TECHNOLOGY, SWEDEN

Course DD2480 Software Engineering Fundamentals, 2020, 2024

> Groups of students from three to five persons experienced the whole software engineering process within a week: From the requirements' specification to the final pull request.

Portuguese

FEDERAL UNIVERSITY OF TECHNOLOGY, PARANÁ, BRAZIL
Course Open Source Software, 2013 to 2016

> Students are requested to contribute to an Open Source project to learn about the maintenance and evolution of software projects. This project is the predecessor of NAU's CS499.

Praises and media coverage

- O. Kopp et al.: JabRef: BibTeX-based literature management software, TUGboat 44(3) - explains the motivation and the concept of the curated issues.
- "I have learnt more from this single pull request regarding production-ready code than I ever have from my three years of CS degree."
- JabRef mentioned as one of "Top 8 Open Source GitHub Projects to Level-Up Your Coding" by CodeGym.

(Please send us your praise if you enjoyed the experience)

Notes

1. JabRef tries to achieve high code quality. This ultimately leads to improved software engineering knowledge of contributors. After contributing for JabRef, both coding and general software engineering skills will have increased. Our development strategy provides more details.
2. We recommend to start early and constantly, since students working earlier and more often produce projects that are more correct and completed earlier at the same overall invested time 1.
3. Be aware that JabRef is run by volunteers. This implies that the development team cannot ensure to provide feedback on code within hours.
4. Be aware that from the first pull request to the final acceptance the typical time needed is two weeks.
5. Be aware that JabRef tries to achieve high code quality. This leads to code reviews requiring actions from the contributors. This also applies for code of students. Read on at our Development Strategy for more details.

References

[1]: @ayaankazerouni: Developing Procrastination Feedback for Student Software Developers

[2]: Lientz B., Swanson E., 1980: Software Maintenance Management. Addison Wesley, Reading, MA.