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.
