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.
