Project Guidelines

We employ a set of project guidelines that developers are expected to adhere to. They are meant to keep our project manageable as opposed to slowly spiraling into chaos and certain oblivion. If you're following the Getting Started guide, you've already encountered some of the guidelines, disguised as settings on the Configuring Eclipse page. They won't be mentioned here again.

The project guidelines can be divided into two broad categories: management guidelines, and coding guidelines.

Content

Management Guidelines

We know the term sounds business-y, but just bear with us for a minute. These are our guidelines:

- We have a weekly project meeting to discuss KIELER issues and to check up on what everyone else is doing. KIELER developers are expected to attend the meeting, if at all possible. You're also welcome to give a short demo of cool new features you've developed.
- We have a developer mailing list that all KIELER developers are expected to be on. If you haven't signed up for it yet, do so now. The mailing list is used to announce changes that affect other KIELER developers.
- We have an IRC channel that you should join. Point your favourite IRC client (XChat is installed on our thin clients) to the Freenode network (irc.freenode.org) and join #kieler.
- We use Git as source code management system. In order to understand the high-level workflows involving clones and branches, read the Source Code Management page.
- We want our code to be well-designed and well-written. To help ensure that, we have a review process, designed to have other people look at your code and suggest improvements as well as learn from it. Take the opportunity to get some valuable input!
- KIELER is an experimental and fast-changing research project, so we don't document it with anything approaching professional standards. However, there are two sources for information: bachelor / master / diploma / doctoral theses about specific parts of KIELER, and this Wiki. If you have developed some complex feature, please consider adding documentation for it to the Wiki. For development work done as part of a thesis, this is pretty much mandatory and will go into the thesis's appendix.

Coding Guidelines

The coding guidelines are guidelines that we cannot enforce solely with the Eclipse configuration:

- When contributing to the KIELER user interface, be sure to respect the Eclipse User Interface Guidelines. By the way, *Hallway Usability Testing* is a great way to make sure users understand the UI you're designing. Just draw a sketch of your dialog design on a piece of paper, visit a few offices and ask people what they think of your design, how they would use it, and whether they understand it. You're almost certain to discover a few problems with your design that you wouldn't have discovered until after you've written a whole lot of code.
- Feel free to use our utility classes. They are found in KIELER's core projects, particularly `kieler.core` and `kieler.core.ui`. If you have written code that you feel could well be part of the KIELER core, tell your supervisor about it.
- When creating new plug-ins, we have special conventions to be adhered to, summarized on the Naming and Metadata page.

Exception Handling

Default handling:

```
IStatus status = new Status(IStatus.ERROR, MyPlugin.PLUGIN_ID, message, exception);
StatusManager.getManager().handle(status);
```

If you want to force the status manager to display the error in a dialog, pass the option `StatusManager.SHOW`.

How to Throw an Exception

The throwable classes are Error, Exception, and RuntimeException. You should choose to throw either an Exception or a RuntimeException. In the first case you must specify the exception in the method's throws declaration (*checked exception*), in the latter case this is not required (*unchecked exception*). In most cases it is advisable to throw a more specific subclass in order to give a hint on what has happened.

- Checked exceptions
    - `KielerModelException`
      Thrown when a problem occurs with a certain model element.
    - `TransformException`
      Thrown when a problem occurs during a model transformation.

- Unchecked exceptions
  - `IllegalStateException`
    Thrown when the class is in an illegal state, e.g. some class variables have incorrect values or are not compatible with the method parameters.
  - `UnsupportedOperationException`
    Thrown when a method is not implemented by a concrete subclass or a specific feature given by parameters is not supported.
  - `IllegalArgumentException`
    Thrown when method parameters have invalid values.
  - `WrappedException`
    Used to wrap a checked exception into an unchecked one.
  - `UnsupportedPartException`
    Thrown when no graphical framework bridge is found for an editor part or edit part instance.
  - `UnsupportedGraphException`
    Thrown when a layout algorithm is executed on a graph that is not supported.

Another hint: often some cleanup code is required that needs to be executed in case of an exception as well as in the normal case. Use a finally block to do this! You can also make a try ... finally block without any catch block, which is very useful.

Checked or Unchecked?

Checked exceptions have the advantage that it is explicitly visible that such an exception can occur and must be handled in some way. They should be used when specific problems are likely to occur and it is known in advance what kind of problems must be covered. A good example is IOException.

Unchecked exceptions have the advantage that it is not required to attach throws declarations for any type of exception that can occur in any part of the call tree. They should be used when it is not known in advance what kind of problems can occur or the problems may occur only in a very deep part of the call tree. This applies to programming errors, which may invoke unchecked exceptions such as NullPointerException or IndexOutOfBoundsException and should always be made visible to the user. Often it is also useful to wrap exceptions into unchecked exceptions:

```
try {
    // ...
} catch (IOException e) {
    throw new WrappedException("Oh no, the input file wasn't found!", e);
}
```

Assertions

Feel free to use the Java assertions mechanism:

```
private void makeTreeRoot(final Tree tree, final Node newRoot) {
    // the new root element is expected to be part of the tree already
    assert tree.contains(newRoot) : "new root not part of the tree";

    ...
}
```

However, bear in mind that assertions capture invariants and assumptions about your code. They thus concern the inner workings of your algorithms and have no business in validating user input. This is because of two things:

1. Assertions throw errors if they fail, but these errors are not declared to be thrown by methods. And what's more, they only throw `AssertionError`s instead of exceptions that are more explicit about what went wrong exactly.
2. Assertions are usually disabled and will thus fail to capture malformed user input in productive installations. They are only meant to help you debug your code and document assumptions and invariants that you would otherwise implicitly assume (or hope) to be true.

To enable assertions in your run configurations, add the `-ea` parameter to the VM arguments.

For more information about Java assertions, see the official documentation.

Platform Independence

Always try to separate user interface code from conceptual or algorithmic code. This avoids dependencies on GUI stuff in pure algorithm projects, such as KLay. A big concern is to avoid Eclipse dependencies in KIELER's base plug-ins to make the code as portable as possible. Other people might want to use it, but might not be developing Eclipse-based projects.

That being said, Eclipse is still the main platform for KIELER, and many projects are tightly integrated with it.

Special KIELER Issues

In order to stay platform-independent, KIELER handles preferences and algorithm progress monitoring independently of Eclipse. Thus, don't use the standard mechanisms, but the KIELER ones:

- `IKielerProgressMonitor`
  This is just the same idea for a progress monitor, which is very useful when executing long running algorithms. Such also the algorithm (with no dependencies on UI elements or Eclipse) need to access the progress monitor in order to report any progress. The KIELER implementation adds neat features such as execution time measurement.