Porting MiGLayout to other GUI toolkits

`MiGLayout`, henceforth called ML, is made for porting to different GUI tookits. In fact, the whole core part of the library is not using a single class from Swing or any other GUI toolkit, it is plain Java 1.4/1.5.

If you have created a port of ML, please let us know so we can link to it from the web site. Nothing would make us more happy than if ML is spread to other GUI toolkits, even outside the JVM!

General Considerations

The connection to GUIs is made through two interfaces and one class, `ComponentWrapper`, `ContainerWrapper` and `MigLayout`. The first two are adapter interfaces to the corresponding components in the GUI toolkit and typically just contain transfer code directly to them. `MigLayout.java` is the layout coordination class. It is the actual class that plugs into the layout system of the GUI toolkit. In Swing and SWT it is implementations of `LayoutManager2` and `Layout`, respectively, and for JavaFX it's a panel as far as I know.

If you are porting ML to a new toolkit all you need to do is to provide implementations for these three classes and you're done. Of course, if you are targeting a non-JVM compatible language you also need to translate the core classes. This should be easy since a lot of effort has been made to not use exclusive Java features but to use simple statements that are available in all languages, some collection classes being the exception.

If you are porting to a JVM language you have the option to link to the core of MigLayout (`java.net.miginfocom.layout.*`) and not convert that code to the language you are porting to. Whether this is preferable to rewriting the core code is your choice. Rewriting the core has the disadvantage that it is effectively a branch and you will manually have to port future bug fixes made to the ML core.

Some of the code in the core is pretty complex. I would expect few would have the time to actually learn what is happening in every step of the layout. Row by row translation would therefore probably be the best way to port the core to a new language. Many of the seemingly complex code snippets are made this way to keep the library small, fast and memory efficient. This is especially true for `Grid.java`, the heart of the layout calculations.

Tests

Unfortunately there are no unit tests for ML today. If you create tests please send them to `miglayout@miginfocom.com` and they will be included in the source. They should probably be written for JUnit 4.

ContainerWrapper interface

This interface contains only a few methods, basically all handling how to get the child components of the container and how they are arranged. The code to adapt a toolkit specific container should be trivial.

ComponentWrapper interface

This is the adapter interface to the base toolkit component. Most methods are for getting the different types of sizes (minimum, preferred and maximum). Even if the toolkit doesn't have a notion of this, like SWT, which only have a preferred size, the minimum and maximum sizes can usually be set to sensible values.

For minimum size one have to choose whether to set it to zero or the same as the preferred size. If the toolkit component doesn't have a notion of minimum size there is no way to always get it to be automatically correct. If zero is returned components will collapse more often and if it is set to the preferred size they will be rigid and never collapse beyond their natural size.

Of course the user can always specify this manually but Swing makes this simpler since every component knows both its minimum and preferred size. Maximum size is usually set to a really high value. ML uses `LayoutUtil.INF` for large numbers, which is less likely to wrap to negative but still be really large.

By looking at the Swing and SWT implementations of this interface it should be fairly trivial to implement.

MigLayout class

This class is in the Swing and SWT ports the coordination class, and it is toolkit dependent. Much of what it does is to figure out when there's need to do a re-layout and then ask the `Grid` class to do the layout calculations and setting the sizes on the components.

Even though the Swing version of this class is quite long, almost 700 SLOC including comments, the part that does the actual layout is just a couple of lines. The Swing version has some extra features that aren't really core, like the ability to serialize the layout and read/change the constraints in many ways.

This class also handles the debug painting coordination. Since it is not usually a layout manager's responsibility to paint something, how exactly to paint the debug information is toolkit specific. What to paint is available from the Grid object after a layout cycle.

IDEUtil

This is a class that is mostly for IDE vendors. After setting design time to true, either directly on the `LayoutUtil` class or through the normal JavaBeans procedure, extra information is available by calling the static methods on this class. Most methods are for getting the original constraint string that was used when adding the component to the layout and to get the grid position and size for components.

PlatformDefaults

This class contains information about how components are placed relative to each other. If the toolkit has different values for gaps and button ordering than the provided defaults, which are different for every platform, this class might need to be appended or replaced.