MiG InfoCom AB

# MiG Calendar™ Tutorial

Release 6

MiG InfoCom AB
S:t Olofsgatan 28a
753 32 Uppsala
Sweden

**www.miginfocom.com**

# Table of Contents

# MiG Calendar Tutorial

## Preface

This document describes the main parts that makes out the MiG Calendar component. Is does not go into great detail of every function, it aims to give a thorough understanding how the different packages and classes relate to each other and what is their purpose.

The **Technical Specification** is more suited for details and should be used as a reference. It can be found at the web site indicated below and should also normally be installed adjacent to this document. Currently the UML JavaDoc is used a the technical specification. It will be converted to a PDF at a later time.

For a getting started fast please see the **Getting Started Guide** which is normally co-located at the same places as described above. The **Getting Stated Guide** has code examples on how to get up and running.

Many IDE:s (Integrated Development Environment) of today have good support for inline help using javadocs. The standard HTML javadocs for the MiG Calendar component is installed by default and can also be obtained from the site as described above and below. We highly recommend using this feature as it increases productively when creating applications with this component.

Although all developers independent of prior experience can benefit from reading this document, general knowledge of the standard Java API and OOP (Object Oriented Programming) will help understanding some of details, or why they are implemented in a certain way.

## Resources and Developer Support

MiG InfoCom AB provides support through email and the online forums. Information and updated tutorials will be made available on the MiG Calendar product site

### Contacting Support directly via Email

`support@miginfocom.com`

### Submit a support ticket

`http://www.migcalendar.com/support.php`

**MiG Calendar Product Site**

`www.migcalendar.com`

**Bug Reports**

Please submit a support ticket.

## Class hierarchy design notes

### Easy to Extend

MiG Calendar has been developed to be easy to extend and still easy to understand. Class design at large is similar to that of Swing's. Interfaces are used for places where the implementation should be able to be switched and abstract base classes normally implements boiler plate functionality giving the user the option to re-implement everything by using the interface or just what matters by extending the abstract base class.

### Method Signatures

You will notice that the method signatures normally are a bit longer that usual, with more options and boolean flags in them. Having one method with more options compared to one method per option will make extending that class easier since there is only one method to override rather than many chained ones.

There is usually more than one way to change the behavior of some class. There are the normal get/set methods that are used for simple properties. Often the classes can be overridden to change the behavior and for the bigger changes the implementations can be exchanged via 'add/remove concrete implementation' pattern.

### Immutability

Immutable objects are used extensively in MiG Calendar. Immutable objects can not change once created and are therefore thread safe and less bug prone due to accidental changes. The downside is that the constructors can be rather lengthy, but that is a price we are willing to pay for stability and simplicity.

### Return type philosophy

Defensive copying (cloning) of objects returned from method calls is normally the case, but not always. For instance getting the bounds of an `ActivityView` returns the 'live' object. The reason for this is performance. The bounds will be read thousands of times per second in high activity count date areas and making a copy for every call will produce too much garbage and the garbage collector will introduce ‘hiccups’. The `javadoc` documentation will clearly state when a returned object should be treated as read only.

**Theme Implementation**

Themes are implemented in MiG Calendar by sub classes to the themed classes. For instance there is a `ThemeDateArea` that extends `DefaultDateArea` and imposes the indicated properties in the provided theme on its base class. This is normally the design pattern for themes, extend and configure. This way the MiG Calendar component can be used just as easily with and without theme support.

**Rows vs Rows & Columns**

The term *rows* is used for denoting something one dimensional, it does necessarily mean that it is horizontal. The reason for this is the fact that for instance the `Grid` object can have either a horizontal or vertical primary dimension. Which dimension is the primary is a property of the Grid and if it would have been mutable the grid could be transposed just by changing the primary dimension. Therefore it would be very confusing to try to use the terms *rows* and *columns*. The terms *primary* and *secondary* are used to express which of the dimensions you mean.

**Date Ranges**

Date range and `DateRange` are used throughout this tutorial and everywhere else in the MiG Calendar nomenclature. Even though 'date' is the word used, time is also always included unless explicitly said otherwise. There are no classes in the MiG Calendar component that only handles whole dates and thus discards time-of-day.

## The DateAreaContainer – keeps it all together

The `DateAreaContainer` is handling Headers, a date area and corner components. The four `header`s are on every side around the `DateArea` and typically contains dates, times or other labeling information. The headers can extend into zero, one or both of its adjacent corners. The four corners can be used for any `JComponent` making it easy for the user to customize the container.

**Scroll pane**

The scroll pane from the `DateArea` is the actual component residing in the middle of the date container, though it is rarely any need to address it explicitly for layout purposes. The scroll bars for that scroll pane is relocated from the scroll pane to outside the right and lower header respectively.

**Mix in components anywhere**

The container is using `GridBagLayout` and the javadoc for `DateAreaContainer` contains the grid rows and columns used for positioning the contained `header`s, corners and the `DateArea`. This makes it very easy to add you own components anywhere within the container, even between the header and `DateArea` if you like. You can put the anywhere.

## Calendar Combo Box

**Date picker**

A combo box that inherits from `JPanel` but look much like a normal `JComboBox`. It tries to tweak parts of it's visual appearance to blend in with the most common Look&Feels. It contains a `DateAreaContainer` and shows it in the popup window, just like a regular `JComboBox` would. It is created from scratch rather than extend `JComboBox` due to experienced troubles when trying to extend `JComboBox`.

The combo supports plus/minus buttons as well as a 'go to today' button. The date format for the editor is exchangeable and is implemented as a `JFormattedTextField`. The editor's text field can be used to edit a to/from date range.

## Date Spinners

The included date spinners vastly surpasses what is available in the standard J2SE SDK. Multiple spinners can be combined and coupled to express one date/time or one spinner can be used to edit a complex date format.

The spinners are simple to use and are located int the `calendar.spinner` package.

## Headers – labeling the Date Area

A `header` is comparable to a normal table header in that it decorates and labels the `DateArea`. The `Header` interface is very small, it only enables returning a `JComponent` and outlines how to dispose of the header. This means that you have complete control if you want to implement your own header. The following text explains the `AbstractDateHeader` and `ShapeDateHeader` which is the default implementation used.

**Primary and secondary dimension**

Technically it is like a small version of a `DateArea` in that it is based on a `Grid`. One dimension of the `header` is locked to the `DateArea`. For north and south `header`s the horizontal dimension is locked and for east and west the vertical dimension is matched to the `DateArea`. In the free (called secondary) dimension there can be one or more rows. Each row can have its own set of properties such as background, font, size, etc.

The cells in the primary (locked) dimensions can be automatically merged. This is for example useful if you want the week number in the header but are actually showing days in the date area. Every cell in the row that have the same text (e.g. '*Week 23*') will be merged.

**Themes**

Almost every aspect, and there are many, can be configured through `Theme`s. The header subclasses themselves are not `Theme` aware, but `ThemeDateAreaContainer` configures them accordingly at when they are created or recreated.

## DateArea

**The heart**

The `DateArea` is the heart of the MiG Calendar framework. It binds together almost all functionality. It doesn't actually implement that much functionality, it contains references to objects that does the hard work. It is a `JComponent` and as such it is dispatching mouse and key events among other things. The `javadoc` for `DefaultDateArea` will be a good way to start since explaining all functionality in text would be very abstract and hard to understand. Some of the responsibilities for a DateArea is:

- Catch and dispatch `InputEvent`s (e.g. Mouse- and `KeyEvents`) to the `Interaction` framework for further processing.
- Manage date ranges of different types, for instance *Selected, Pressed, MouseOver, Visible* and *Selectable.*

- Manage a `DateGrid` object that contains the date to pixel translations. The DateGrid object will be recreated when needed and the DateArea is maintaining the information that should be persistent, such as the primary dimension and selectable range.
- Managing the layout and paint process of everything that is visible on the date area. This is done thorough installable `Decorator`s. There are decorators for painting the grid, activities, cell labels, selection rubber band rectangle and so on. In all some 15 different types of decorators is delivered with MiG Calendar.
- Manage a subscription of the activities that should be shown in the `DateArea`. This includes checking for updates and taking the appropriate action when it comes to the visual part.

## The Grid and its GridRows

To map a point in time to a specific position on the screen a `DateGrid` object is used. It doesn't matter if time progresses horizontally or vertically, the same grid class is used. The primary and secondary dimension (I.e. if time progresses horizontally and wraps vertically, like text, or the other way around) is a property of the grid object.

### Grid and DateGrid

A `Grid`, which is the base class for `DateGrid` and contains the actual cell handling methods, is always square (x * y). `DateGrid` adds the functionality to map a point in time to a certain position within the `Grid`.

### GridRows

A `GridRow` contains information about one row, or grid line, in the Grid. It can be either horizontal or vertical, though it doesn't contain that information itself. The size and position of every row and grid line, in both dimensions, is stored in these `GridRow`s. The grid's rows can be extremely flexibly sized. They can be absolute (pixels) or relative (percentage of bounds).

### GridSegments

`GridSegment`s are used to describe the size attributes for one or a group of rows. Remember that rows can be both horizontal or vertical. A `GridSegment` can  for instance describe something like this: 'Five rows with a minimum size of 10 pixels, max 25 pixels but preferably 20% of the total size (weighted)'.

To encapsulate all segments for a grid a `GridSegmentSpec` object is used. It is basically a group of `GridSegment`s, with some simple to use constructors. It describes one dimension of the `Grid`, for instance the primary dimension.

**Grid lines**

Quite similar are the specification for the grid lines. They also can be of any size (width), even individually. The `GridLineRepetitionSpec` interface extends the `RepetitionSpec` which describes a generic way to specify repetitions. There are two concrete classes that implements `GridLineRepetitionSpec`, one very flexible () and the other which are really simple to use. It is easy to create specifications like: 'Make every other grid line 2 pixels and the rest 1 pixel, but make the first and last 0 pixels. Set the Color to black for the thick ones and gray for the small ones.

**Sub grid rows**

If the grid is divided into `Category`s every `GridRow` can even contain sub rows (which is are also `GridRow`s). Every one of these sub-grid rows will filter which activities it accepts. This filter system is very flexible. See Categories below. Note that an `ActivityView` that exists in more than one `GridRow` (because more than one row accepts it) will have multiple bounds and is not implemented through multiple `ActivityView`s. The sub grid row implementation is really a tree structure. Rows can be in row, that can be in rows, and so on.

Sub grid rows don't have to be denoting categories, they can be used to filter on anything, but categories are the most common usage. Set the getting started guide for more information on sub rows and categories.

## Activities and ActivityViews

**Activity class**

An `Activity` is something that has a start and end date/time. It implements the functionality of the `Event` keyword in the iCalendar specification, but it is broader in context. It can have a number of properties, mapped with keys, much like a normal Map. It can be recurring and have some additional attributes such as creation time and last modified information.

**ActivityView class**

The `Activity` can be seen as the *model* of the Event (we mean Event in the context of iCalendar here), and `ActivityView` is a view of that model. Both are interfaces but they have both abstract base implementations as well as default concrete classes.

The `Activity` is what contains all persistent information and is the object to install listeners on to get updates when that information changes. `ActivityView`s are created and disposed of when convenient, they are used to show one side or recurrence of the actual `Activity`. Normally it is the `Activity` itself that creates the views for a certain context, it also pools them so they don't have to be recreated all the time.

### Activity views are expendables

Normally there are one `ActivityView` for every `DateArea` that shows the `Activity`. There are also one view per recurrence if the `Activity` is recurring. Even though the user are actually interacting with the view, the model gets all the updates and repaints/updates its views. This ensures that all views of an `Activity`, in all `DateArea`s, are updated if one view is being dragged.

### More than one bounds

One interesting aspect of an ActivityView is that it can have multiple bounds. This is because the view might span more than one row (say more than one week in a month view) and if so one rectangle can not describe the positions.

## Time Zones

Time zones are a fully supported in MiG Calendar. You can set the time zone for the date range when creating an `Activity`, and you can even change it later by setting a new date range. You can also set a time zone for the `DateArea` by specifying that in the visual date range,  though that might have less impact than you think.

Time zones can be a bit hard to grasp since they actually present a view of the actual time. Here are a few concrete notes and pointers on time zones:

- An activity's time zone will not change the position for it in the date area. It will only change the time shown in its title or text (if for instance the template text `$startTime$` is used).
- Activities' time zones will not affect whether they overlap in time or not.

- Setting a different time zone for the date area will generally only change the times on the date/time headers. However if you create a date range by saying it should span a day or a week and provide a different time zone that will mean the date area will span another physical time since the day/week boundary is probably different in the other time zone.
- The time zone generally affects calculations that deal with hour-of-day and such properties of the time. Rounding to a date boundary will for instance give different physical date/times in different time zones.
- When printing a `DateRange` to the command line the time zone will be honored. Printing a `Date` object will always use in the default time zone for the machine. By using a `SimpleDateFormat` you can choose the time zone used when printing a `Date` by setting a different `Calendar` object on it or changing the time zone for the current one.
- If no time zone is specified (set to null) in an activity's date range, that activity will get the default time zone. The default time zone will not be set at creation time though, it will be re-evaluated for every use, so changing the default time zone will affect all activities with a `null` time zone set.

## ActivityDepository

### Where Activities are found

One problem with activities is that they are usually stored not withing the calendaring component, in fact that would be a bad design. Also they can be scattered over more than one storage medium, possibly at least one that are not connected continuously. Since the *can* be stored in multiple places they can also come out of sync, i.e. the same activity exists in different versions on different storage mediums. These situations have to be addressed by the `ActivityDepository`.

### Subscriptions

From the MiG Calendar's positions things are quite simple. Every `DateArea`, or at least `DefaultDateArea`, only registers itself as a listener on the depository and will as such get notified of changes in the depository. This is normally handled automatically.

### ActivityIOFilters

When you write your own importer of activities, or create a factory that creates them you should implement `ActivityIOPlugin`. It contains some informative methods where you can provide an `ID` and a human readable name.

### Resolving version conflicts

The reason you have to implement `ActivityIOPlugin` is so that every `Activity` has a parent that can take care of its life cycle. When you add the `Activity` to the depository you also have to provide a reference to the plugin that read or created the `Activity`. If there already exists an `Activity` with the same ID as the one you are adding a version conflict will arise, no matter if they actually are equal, that is for the `ActivityResolver` so decide.

If no `ActivityResolver` is installed for the the `ActivityDepository`, the `Activity` with the most recent modification date will be retained and the other one discarded. If another synchronization algorithm is to be used, the `ActivityResolver` can easily be replaced for a more advanced or better suited one.

## Laying out the ActivityViews

### Point to time to pixels

How to map a certain date range to some bounds in a pixel based date area is not trivial one. MiG Calendar has a very flexible solution and a very powerful default implementation that should be sufficient for almost any need.

### More than one layout simultaneously

The `DefaultDateArea` has support for multiple layouts at the same time. The layouts are ordered after how keen they are to layout a certain `ActivityView`. The installed layout that reports the highest integer value for a `ActivityView` will, during the layout run, be asked to layout that view.

During the layout run the layout that is first (layouts are sortable and sorted in their natural order, which is changeable) in line will be asked to layout its views. It does so and returns all rectangles that are now 'occupied'. The next layout will have to, or should, consider these occupied rectangles as forbidden areas and have to layout around them.

### More then one level

The above paragraph describes what happens within one level. Since activities can be divided into layers, a lower level will be laid completely before the next level is laid out. The occupied rectangles will be cleared between layers making them visually overlap. In short one layout cycle will be made for every layer (see `Activity.getLayerIndex()`) that has at least one `Activity`.

This flexibility is needed to accommodate for the possibility to have, for instance, background images for Christmas, normal date range rectangles for most activities and maybe icons for alarms and reminders.

Implementations of interface `ActivityLayoutBroker` (`DefaultDateArea` is one) will decide which activities is laid out by which `ActivityLayout`s. The default implementation decides upon whether the activity's layout context (`Activity.getLayoutContext()`) matches the installed layout's. The `ActivityLayoutBroker` and the `Activity` thus cooperates in this process.

Three `ActivityLayout` implementations are included by default.

- `FlexGridLayout` lays out in a grid or flow like formation. It is hight customizable, even trough a Theme.
- `TimeBoundsLayout` lays out the views according to their exact or rounded dates and times in the `DateGrid`. Overlapping date ranges are handled and the overlap can be specified both absolute or relative. The size in the secondary dimension is also very flexible with min/preferred/max size that can be absolute or relative to the available bounds for the `GridRow`.
- `HideLayout` can be used to hide views with certain properties. Currently the duration is evaluated. It can for instance hide activity views that spans more than one day or is less than on hour.

The layout framework in MiG Calendar is very flexible though you will probably just install the layout that you need and that's it. It it probably even simpler, you just configure one or more layouts in the GUI Theme Editor and everything is handled automatically!

## Decorators – layered painting

Decorators is used to paint in both the `DefaultDateArea` and `AbstactDateHeader` and its sub classes. `Decorator`s are very simple to understand, they are objects that gets some bounds to paint within and they do just that. Since everything (yes everything!) that is painted in the `DefaultDateArea` is painted with `Decorator`s the look is truly pluggable, even without sub classing it.

This also means that it is possible to decide on the order that the different installed decorators should be called. This is done through the mutable *layer index*, much like for layouts explained above. Decorators will be sorted according to their layer index and then the painting will be done in that order.

This means you can have the grid lines drawn above or under the `ActivityView`s and so on. Some decorators will only make sens in a certain order, for instance the background decorator should probably be invoked first.

MiG Calendar is delivered with around 15 different decorators, all decorating some part of the `DateArea` GUI. Background, activity views, grid lines, cell labels and the no-fit shape are some of them.

Also `Header`s, through the `AbstractDateHeader`, are painting its contents with decorators.

The `Decorator` class hierarchy is somewhat extensive and checking the UML diagram for is probably a must if you intend to write your own decorators, at least if you want to get a some free boiler plate from one of the current decorators.

## AShape – attributed graphics graph

The `AShape` framework is one of the more extensive sub components within the MiG Calendar component. It is powerful enough to write really innovative GUIs in, for instance be vector based user interfaces with animations triggered by user interactions.

AShape is a complete stand alone component in its own right, with use cases outside that of the MiG Calendar component. In this calendar component it is used by the default activity painter to draw the `ActivityView`s. It is also used by one type of `Header Decorator` to draw text and backgrounds.

`AShape` is an API for hierarchical graphics. Sub shapes relate to their parent in that it will use its parent's bounds as a base for getting its own bounds. Much of `JComponent`'s design ideas are used in `AShape`, but `AShape` is a lot more light weight and more aimed at fast Java2D graphics. There are also some similarities between the

SVG (Scalable Vector Graphics) format and `AShape.` SVG is broader in context but `AShape` has more flexible layout support and is more geared towards Java and Java2D. `AShape` has the ability for non uniform scaling, a functionality that SVG currently lacks.

There are many concrete classes that implements the `AShape` interface. `VectorShape`, `TextShape`, `ImageShape` and `FeatherShape` (for blurring) are some of them. It is also very easy to write your own `AShape`s if the built in ones don't cover your particular use case. The `AbstractShape` implements most of the boiler plate and all you have to do is extend it and provide the painting code.

The layout idea behind `AShape` is that you provide it (actually the `ARootShape`) with a reference rectangle and then tell it to paint itself relative to that rectangle. The `AShape` itself decides how it should relate to that rectangle, but it will probably in most cases just cover the whole of it. Sub shapes of a shape are then placed according to the bounds that the parent shape actually used. The placement relative to the `Rectangle` bounds is normally specified with a `PlaceRect`, or rather one of the concrete classes that implement that interface (`AbsRect` or `AlignRect`).

A `PlaceRect` describes how one `Rectangle` relates to another `Rectangle`, optionally with a reference `Dimension` (i.e. size). The `PlaceRect` handling and a lot of other boiler plate is done in `AbstractShape` so you don't have to bother with it normally. That is also why almost every `xxxShape` takes a `PlaceRect` as argument in its constructor.

The sub shapes can freely relate to its parent's bounds and since the `PlaceRect` implementations have a very flexible coordinating system they can do this in a very advanced and dynamic fashion. It is for instance quite easy to describe something like: 'use the right 50% of the parent's bounds but no more than 10 pixels, right justified if constrained'.

But how does siblings interact? With `ShapeLayout`s. Every `AShape` has a `ShapeLayout` that actually gives the sub shapes theirs reference rectangle. If no layout is specified explicitly the `DefaultShapeLayout` is used which will layout all sub shapes with the exact bounds of the parent (on which it is installed). All sibling will thus get the same reference bounds to used for placing themselves and there are no sibling cooperation.

`RowShapeLayout` and `DockingShapeLayout` are delivered by default and how they work and are supposed to be used is specified in the javadocs.

All built in `AShape` implementations are serializable to XML with the default Java beans persistence using the standard get/set naming convention. There is no special `Delegate` needed for this.

`AShape`s can be used in two ways. There is the normal one-to-one usage pattern which means that *one* shape will be used to paint *one* thing. This would not be scalable if there are 1.000 or maybe 10.000 things to paint as it would mean that many `AShape`s would have to be created. That would consume an unnecessary amount of resources. Therefore all `AShape` types are *stampable*. This means that the same `AShape` instance can be used to paint multiple, normally all, entities (e.g. `ActivityView`s). An `Interactor` is used as the peer between the 'thing' to paint/decorate and the `AShape`. That `Interactor` is more light weight than the `AShape` and should contain the per-'thing' information, if any. It also has other functionality, see the Interactions section below.

Attributes, such as `Paint`, `Font` and `PlaceRect`, of the `AShape`s are stored in a `Map` in the `AbstractShape` class. A key is used for getting and setting the attributes. Every `AShape` subclass documents which keys to use to access the attributes interesting for that class.

For further details on how to build and use `AShape`s see the `AShape` tutorial and the Getting Started Guide.

## Interactions – how the user interacts

The Interaction framework is a Action/Command-like pattern optimized to fit the MiG Calendar component.

To start with we have the `Interactor` hierarchy of classes, which includes `ActivityInteractor`, `MouseKeyInteractor` and `TimerInteractor`. An `Interactor` is the peer to some functionality that need to be monitored and interacted with. For instance the `MouseKeyInteractor` monitors mouse and key events and for every one of them it evaluates if there is a registered `Interaction` that should be triggered.

An `Interaction`, not to be intermixed with `Interactor`, mainly consists of three parts:

1. A trigger of some kind (e.g. mouse move)
2. An `Expression` that needs to be evaluated to `true`. E.g. `MouseButton == 1`

3. The `Command` or `CommandSet` that should be executed. E.g. `StartAnimation`.

A `Command` is a generic specification of an action that is to be taken and what is the target. It can also be sub classes to support any action that can be written in Java code, without the need for a specific `InteractionBroker`.

An `InteractionBroker` is an executor/interpreter of `Command`s. A broker will normally handle a certain type of commands, for example the `ActivityViewInteractionBroker` can handle commands that will affect an `ActivityView`, it has specific knowledge on how to manipulate it.

The `Interaction` framework is very loosely coupled. The different parts, Interactor, Interaction, Command, Expression and InteractionBroker have few demands on each other's types. This can be confusing at first but is very powerful once understood. See the examples in the AShape Tutorial for hints on how to use this powerful and flexible framework.

One important functionality that the `Interactor` framework supports is the notion of overriding attributes. If, for instance, an `AShape`'s background color is normally blue, it can be overridden to be yellow when the mouse is hovering over it. The original property of the `AShape` is never actually changed, it is just exchanged for the overridden value when the attribute is fetched, for instance during the paint process. The `Interactor` keeps track of the overrides that is currently active for its peer object, e.g. an `ActivityView`.

## Geometry Package and Coordinates

Since the MiG Calendar requires a lot of flexibility when it comes to positioning graphics on screen a new type of coordinates was developed. All sub components within the component uses this coordinating system. Classes that could denote statements like: '10 pixels from the end' and '20% in, but no less than 10 pixels from the right edge'. These kind of statements is easy to do in Java code, but quite hard to specify in a persistable object, such as a coordinate. The `util.gfx.geometry` packages has these main parts:

- Numbers. A number can relate to 0, 1, or 2 other numbers. 0 means it is a value on it's own, such as an `Integer` normally is. 1 means it needs one reference value in order to produce a number, for instance when a coordinate denotes an offset or percentage. 2

means that it needs two reference values, normally start and end values, and the coordinate would be in reference to one or both of them. Those types are specified by the interfaces `AtNumber`, `AtRefNumber` and `AtRefRangeNumber` respectively.

- Links. If a coordinate should be in reference to another coordinate, for instance if you would like to dock the right edge of one rectangle to the left edge of another, a link links them together. Links are one-way and the link to coordinate must be set before the link can be resolved. I.e. it must be layout out before the linked coordinate. A link implements `AtNumber` so it can in effect be used at a coordinate like any other number.
- Filters. They could also be called constraints or operators, but they are more generic, that is why they got the name Filters. They wraps a coordinate in some logic, it might be a minimum constraint, a multiplication or a filter that rounds it to the nearest integer. These filters are themselves `AtXxxx` numbers which means that a filter can wrap another filter and so on to create quite advanced layered expressions. Normally that is not the usage though, they should be used for imposing constraints or simple adjustments. Filters is an implementation based on the decorator pattern.

The flexibility of this geometry package might seem daunting but it's really quite simple, and you seldom have to use any other types than those listed under Numbers above.

The geometry package gives MiG Calendar the much needed possibility to express coordinates relative to one another or relative to other boundaries, such as available bounds. See the examples in the getting Started guide to get acquainted to using this geometry package.

## Categories – hierarchical tags for Activities

All activities can be connected (in a loose manner) to one or many categories (`Category`). The categories themselves are hierarchical and can denote just about anything. In one JVM there are always exactly one category *root* and all sub categories are mounted somewhere below that root just like a normal tree structure or file system. Categories can have multiple parents. This means that a category sub tree can be mounted on more than one cranch for instance.

What the categories mean are totally up to the user of this component. They can indicate owner of the `Activity`, a difficulty level of the it or just what type of activity it is (E.g. *Home*, *Work* and *Imported*).

Since one activity can belong (reference) more than one `Category` and Categories are hierarchical the possibilities are endless. In the simplest case they can be used to differentiate calendars by their owners. In a more complex context they can be used to create a complete role tree with hierarchical dependencies.

`GridRow`s in the `Grid` can filter on, among other things, one or more categories. This opens up for having multiple calendars show in line side-by-side. It also means the MiG Calendar supports activities that belongs to more than one calendar and as such shares all information.

If there are sub grid rows in the date area there can also be a header that shows the names of the grid rows. Since categories is actually only one use for sub grid rows, the can filter on anything, the header is actually called SubRowGridHeader. This header works much the same way as the date headers, which means the can have more than one row in the header and cells in different rows can be merged.

## Themes – advanced hierarchical properties

A `Theme` is comparable to a configuration file, only it is very flexible and supports one-to-one, many-to-one and one-to-many mappings. The keys are hierarchically arranged and can be constrained to one or many different types. For `Comparable` types they can even be constrained in it's range with a min/max.

The simplest way to learn about Themes is probably to start the Theme Editor and play around with it. The Theme Editor is a GUI for changing any type of `Theme`, and the `Theme` used in the MiG Calendar component is `CalendarTheme`. Note that the Theme Editor is generic and can be used to visually edit themes that have nothing to do with calendaring.

Themes are based on a key/value map where the key is a String that looks like a file name including path. E.g. `”Decorators/CellLabel/installInLayer”` is a key. The first slash is always omitted, though it is always mounted on the the `Theme` *root*.

Some of the more noteworthy features of the Theme framework are:

- Key's can be linked to other keys, and even chained. This means that there can be one edit point for many related properties, say colors for week days.
- Default values for every key, or key hierarchy, are mandatory. Though they can be `null`.
- Keys can contain a ordered list of values. The values in the list can have constraints, rather than the list object itself, which would be the case for just using a `ArrayList` as the value.
- All value change can optionally be validated (default) against the key's capabilities (constraints). This means that an erratic value can never end up in the `Theme`.
- Both default values and capabilities (constraints) can be set recursively on a key branch. This decreases the verboseness and increases correctness when building themes.
- Themes can be listened on for changes to keys using the normal Java `xxxListener` pattern.
- Loading and saving (serializing and deserializing) themes are supported directly by the theme and is very simple.
- Multiple themes, even of different types, can be loaded simultaneously and are handled by the `Themes` class. They are indexed with a *context* `String` as the key and are accessible from the whole JVM since the `Themes` class is a *Singleton*.

For more information on themes, including how to build you own and/or extend the CalendarTheme see the javadocs for `Theme`.

## Theme Editor – GUI for Themes

The Theme Editor is a GUI application for editing themes. All information needed to edit the theme is included in the theme class. Theme properties are serialized to XML for storage in a file or database. The `Theme` class implementation, e.g. `CalendarTheme`, needs to be in the *classpath* for the Theme Editor to be able to edit it. This is because the default values and key capabilities (i.e. constraints) of the theme is not saved in the XML properties file, but is located in the actual `xxxTheme` class.

The Theme Editor can edit any type of value that it recognizes. The most common value types already have an associated editor, but it is very easy to write your own, you only subclass `AbstractPropertyEditor` and registers it to the editor. You can even register the editor from the command line as long as it is in the class path.

The default AShape used to visualize demo can be exchanged if the Theme Editor is started from the command line.

## Utility Classes – things you may find useful

To defer code duplication a lot of functionality is lifted out to utility classes. `DateUtil` is such a class. `DateRange` is another. Here is a list over some of the classes available to the user of the MiG Calendar Component.

- `DateUtil`. Static methods for different kinds of date manipulation. `Calendar` rounding to boundaries (e.g. day or hour), flexible and fast duration string formatting and week bases helper functions are some of the things this class can do.
- `GfxUtil`. Contains a lot of geometry calculation methods, mostly based on `Rectangle` or `Rectangle2D`. Color manipulation, lots of string drawing methods, image and texture blending and improved image loading are examples of what this class provides.
- DateRange. This is a class, or rather class hierarchy, used extensively throughout the MiG Calendar component, but it has uses outside communicating with the component. It encapsulates a start and end time and a lot of operations on that period. Iterating over it at specified multiples of a boundary is one example. It exists both as a truly immutable and a mutable version as well as it implement both a mutable and immutable interfaces.
- Misc. graphics classes. `UIColor`, `SlicedImage`, `XtdTexturePaint` and `ArrowButton` are some of the usable classes in the `util.gfx` package.
- Base64 encoder/decoder. The fastest Java bases base64 encoder and decoder is included. It is developed by MiG InfoCom AB and Open Sourced. It can be downloaded from http://migbase64.sourceforge.net where there also are some performance comparison charts.
- IO. The `util.io.IOUtil` class adds support for aggregating XML delegates (`java.beans.PersistenceDelegate`) used for saving and loading unsupported object types to XML with the standard 1.4+ `XMLEncoder` and `XMLDecoder`. If you want to add a object type of your own and that type doesn't follow the

standard get/set naming convention for all properties (and an empty constructor) a the delegate for that object could be added to this class and it will be used by the framework.

## Putting it all together

All major parts of the MiG Calendar component have been explained above, but there are a lot of details left out for you to find out. It is easy to get started with MiG Calendar but the advanced user will also be able to tweak, exchange, extend or replace just about anything in the framework.

The default implementations of all functions are targeted to fill the need for most situations. Just by creating a Theme with the Visual Theme Editor you can almost make the calendar look like any of the main calendaring applications available today (Outlook 2003, Apple's iCal 1.5 and Mozilla's Calendar).

Creating an `AShape` to be used for painting the activities are maybe the hardest part if you want to create you own custom calendaring application, so we have made a couple of default ones accessible through static methods in the `ShapeFactory` class for you to use.