
# AShape™ Developer Manual

Release 1.1

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

**www.miginfocom.com**



Java is a trademark registered ® to Sun Microsystems.
`http://java.sun.com`

# Table of Contents

# MiG Calendar Tutorial

## Preface

This document aims to provide information on how to develop applications that uses `AShape`s.

The **AShape API JavaDoc** will provide 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.

Many IDE:s (Integrated Development Environment) of today have good support for inline help using J*avaDocs*. The standard HTML J*avaDocs* for the `AShape` framework is installed by default and can also be obtained from the site as described below. We highly recommend using this feature as it increases productively when creating applications with this component. Many things are only documented in the JavaDocs.

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 understand some of the details and 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 AShape product site

**Contacting Support via Email**

`support@miginfocom.com`

**Contacting support via online forums**

`www.miginfocom.com/forum/`

**Bug Reports**

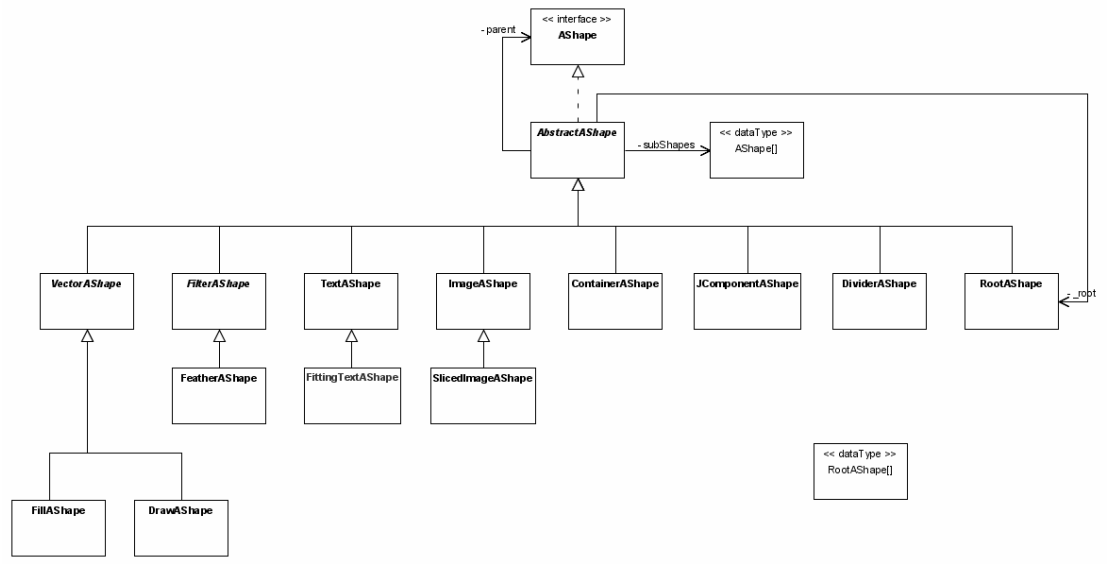
Via Email or forums as indicated above.

# AShape Structural Overview

### The Basics

To understand how this component works you must have an understanding on both the class hierarchy, which is really simple, and the tree like structure the objects are stored in when they are created. It is important to understand the distinction between *class inheritance* and *tree structure* the objects are ordered in.

### Class Hierarchy




As you can see the class hierarchy is simple to understand, yet very powerful. The objects should almost exclusively be handled by their interface type `AShape`. This ensures that the implementation can be exchanged at any time, say of you want to exchange the `TextAShape` for your own implementation at a later time.

`AbstractAShape` adds a lot of boiler plate functionality for the sub classes. It implements almost everything from the `AShape` interface and a lot more. This includes:

- Property handling with override support (explained below)
- Layout and paint cycle management

- Reference and actual bounds handling
- Hit shape storing and processing
- Layer and paint order handling
- Interactions

The concrete `AShape` classes, such as `FillAShape`, only have to implement the paint method that actually draws the shape in a provided `Graphics2D` object. This means that if you want to write your own shape classes it can easily be done, yet they will still have all the features that `AbstractAShape` provides.

**Tree Structured**

The shapes are combined in a tree structure just like `Components` in Swing or SWT. There is always a `RootAShape` at the top. It has some additional information that is used for the whole tree. Then one just add sub shapes to the root shape, and probably sub shapes to the sub shapes as well. A sub shape node's bounds is always relative to it's parents bounds. Exactly what bounds a sub shape will get is dependent on the `ActivityLayout` set on the parent as well as its own `PlaceRect`. A `PlaceRect` is a rectangle that denotes a difference to another rectangle, in this case to its parent's bounds.

There are currently three different concrete implementations of a `PlaceRect`. And that is `AbsRect`, `AlignRect` and `AspectRatioRect`. They all have a little different approach on how to specify the relativeness to the parent's bounds.

Building simple as well as complex shapes this way is much like how it's done in Swing. There are just different types of layouts and Swing lacks the `PlaceRect` functionality.

In the demo directory created in the installation directory there is a class called `AShapeCreator`. It contains a lot of code that creates different types of `AShape`s. Some simple and some complex ones.

# Usage Patterns

**Pattern #1. Like a Normal Component**

This is a one-to-one relationship in that you create one

AShape (or rather one AShape hierarchy with a RootAShape at the top) for every entity you want to decorate i.e. a *UML package box* or an ActivityView in a date component.

This way is simple but if there are a great number of entities to paint there will be as many AShape hierarchies, which consumes resources. This is how Swing works.

If only one or only a very small number of entities should be painted, such as GUI buttons, this is the preferred method since each AShape has it's own reference handle and can be changed and managed directly without switching Interactors during the paint process (explained below).

Tip! The AShapes should be created with shared (same) graphics primitives that are immutable, such as Fonts, Colors, Paints and (usually) Images, to save resources.

**Pattern #2. Like a Stamp**

This is much like CellRenderers in Swing where you reuse the same Component for all cells in for instance a JTable. This reduces the resource strain especially regarding memory consumption and startup time. This approach is often called a Flyweight Pattern.

The drawback is that you have to manage a number of lightweight "peer" objects, called Interactors, which contains the specific information for the entity to paint. You don't have to do this if you don't want to but if you don't, every entity will be painted exactly the same (but they may have different reference bounds) with no possibility to interact with any one of them. They will be totally generic.

The Interactor contains information about the state of the entity to paint. For instance if the outline color of the currently painted AShape should be darker/brighter than the others because the mouse hovers over it. Basically you switch in the Interactors that belongs to the entity you are just about to paint.

See *Interactors* further down for information on how this works.

## The AShape Classes

Below is an overview of all concrete AShape classes. Almost

all non-paint functionality is implemented in `AbstractAShape` and thus not repeated for the subclasses below.

**RootAShape** – Must always be at the root of all `AShape` hierarchies, much like `Container` in Swing. Contains some extra methods and fields that applies to the whole tree.

**VectorAShape** – Wraps a normal *Java2D* `Shape` object and gives it attributes such as `Paint` and anti-aliasing hints. Is abstract.

**FillAShape** – Extends `VectorAShape` (above) to provide filling support for the shapes.

**DrawAShape** - Extends `VectorAShape` (above) to provide outline drawing support for the shapes.

**TextAShape** – For drawing text within some bounds. Can be rotated (+90 or -90 degrees) and set to wrap or not. A lot of attributes exist that makes it powerful and flexible to use.

**FittingTextAShape** – Subclass of `TextAShape` that checks the available strings one by one and draws the first one that fits the given bounds. Can be used if you have more than one possible text to draw, (e.g. "Sunday", "Sun" and "S") and want to paint the biggest one that fits.

**ImageShape** – For drawing and possibly stretching images. Can cache an arbitrary number of stretched versions of the image.

**SlicedImageAShape** – Extends `ImageAShape` and takes a normal `Image` and slice it up in a 3x3 slices. These slices are then stretched or tiled in a very flexible way to accommodate for the common case where you have a template image that you want to stretch to any size but where the border pixels should be treated in a different way than the center slice. This is a very common technique for themed GUI:s.

**FilterAShape** – A generic abstract base class for drawing filtered images, i.e. blurred or edge enhanced. Provides caching and slicing (through a `SlicedImage`) to improve performance in orders of magnitude, especially for rectangular, edge filtered, images/shapes such as shadows. Subclasses need only provide a `BufferedImageOp`, so creating enhanced image shapes is done in a snap.

**FeatherAShape** – Extends `FilterAShape` with a feather

(Gaussian Blur) algorithm. It takes another `AShape` object as the object to blur which makes it very flexible.

**`DividerAShape`** – A simple shape to use as a divider (horizontal or vertical). Made simple to be very quick and easy to create.

**`JComponentAShape`** – A special kind of shape that can contain a normal `JComponent`. There are no special magic here, it just sets the `JComponent`'s bounds. Repaints are handled automatically since its parent should be the same parent as the `JComponentShape` is drawn onto.

**`ContainerAShape`** – Does not paint anything itself, it is just to be used to group a number of `AShape`s. All `AShape`s can be container shapes but this one is simple and paints nothing.

## The Paint Process

The actual layout and paint process is much like the one in Sun's Swing architecture except that it happens every repaint and it doesn't invalidate any "dirty areas" as Swing does.

The steps as they normally happen in list form:

1. Some controller (maybe your code, maybe a `AShapeComponent` which is included) decides the *reference bounds* the `RootAShape` should have and sets it with a call to `setReferenceBounds(Rectangle)`.
2. If the `AShape` is to be used as a "stamp", and thus will be painting several entities, the `Interactor`(s) belonging to the currently decorated entity (i.e. an `ActivityView`) will be set on the `RootAShape` with `setInteractors(Interactor[])`.
3. `RootAShape.paint(..)` is called to start the layout + paint process.
4. The `RootAShape` calls `layout()` on itself to first do a layout run where all sub shapes' reference bounds are calculated and set.
5. The `AShapeLayout` installed will layout the children and then call `layout()` on them to make them layout them selves. This process makes sure that the whole tree will be

laid out. All AShapes in the tree can have their own AShapeLayout. DefaultAShapeLayout is the one used if none is set explicitly.

6. The ARootShape calls paintSubTree() on all its first level children. This paints that sub tree depth first. When all of the RootAShape's first level children has been called, the whole tree has been painted.
7. The hit areas for the just painted shapes (that have the hit area reporting turned on) are collected and stored in the Interactors that are of type MouseKeyInteractor (normally only one). The MouseKeyInteractor will make use of this information to listen for MouseEvents and/or KeyEvents and when a "hit/mouse hover/whatever" happens, maybe change something. See *Interactors* below.

   Important!

   The hit areas will only be reported for AShapes that has the property AShape.A_REPORT_HIT_AREA, which is Boolean.FALSE by default. If it is set to false, or not set at all, no mouse interaction will be possible with this sub shape. Only enable it for the sub shapes that needs it though, since calculating them uses some resources.
8. Repeat all for next entity or end if done.

# Layouts

### The Layout Cycle

The AShapes are laid out every time it is painted. How this cycle works is explained above.

### The Layouts

There are three AShapeLayouts delivered with this component.

### DefaultAShapeLayout

A very simple layout, which is also the default one. It offers the sub shapes the parent's actual bounds and let them place themselves relative to that. For instance if all sub shapes had the PlaceRect AbsRect.FILL set, they would all have the the same actual bounds as their parent, all overlapping each other. In short, the siblings (first level sub shapes) doesn't

depend on each other in any way, only on its parent.

**CutEdgeAShapeLayout**

Layout the sub shapes according to the following algorithm:

1. The bounds of the parent is offered to the shape in turn to get laid out.
2. The actual abounds that that shape will use (depending on its `PlaceRect` normally) is cut of from the parents bounds and the bounds left are offered to the next sub shape. This then repeats until all sub shapes has gotten laid out.

What this means is that no sub shape will overlap. This works much like a *DockingLayout* (sometimes called *EdgeLayout*) and is normally used as one. The sub shapes to be "docked" should have their `PlaceRect` cut of the correct piece. There is an optimized constructor for this in `AbsRect` (which implements `PlaceRect`).

**RowAShapeLayout**

A layout not unlike `BoxLayout` in Swing, but much more flexible. It lays out the sub shapes in one row, with the size in the non laid out dimension set to match that of the parent.

The size in the laid out dimension can be absolute or relative and have min/preferred/max sizes set. This makes it very flexible, especially since it asks the sub shapes what is their min/preferred/max size, if that information isn't set directly for the `RowAShapeLayout`.

It will not make the sum of the sub shapes larger that its own size.

This layout can be used for creating a list of icons for instance.

**Rolling Your Own**

It is very easy to make your own `ShapeLayout`. You either implement `ShapeLayout` directly and write the two methods it specifies or if the the `AbstractAShapeLayout`'s size estimation is adequate you just extend it to implement the actual layout algorithm.

# Interactions

*Note! From v6.0 of the component there is a simpler and more flexible way to interact with the shapes. See the **Listening for MouseEvents** section below.*

### The Basics

Interactors are used to connect the `AShapes` to the rest of the application. They are very flexible and there exist many implementations to interact with different types of outside events.

The whole `util.interaction` package denotes a very extensible and highly decoupled way to say what you want to happened, to what, and when.

Tip! Also check the **Static Overrides** section below as it is a simpler way to change the properties on an `AShape` in dynamic way. It is a simpler and more flexible way to handle overrides in some ways but there are things you can't do with them, for instance starting and stopping animations on specific timer events.

### Usage Pattern

Normally you have an `Interactor` (or many) that is connected to the entity (the interacted) you are interacting with (such as a button or activity) and usually also to some source of events, such as a `JComponent` or a `Timer`. The `Interactor` contains a number of `Interaction`s which specify what to do when something happens. For instance: "*Change the color to blue when clicked on with the mouse*".

All Interactors will process the events sent to them and check if there are any `Interaction`s that needs attention (i.e. if it's a "trigger" for the `Interaction`). All Interactions that is triggered will be evaluated with the `Expression` set on it. That `Expression` can denote just about anything, for instance if the mouse is over it.

If the `Expression` is evaluated to `true` its associated `Command(s)` will be run using the `InteractionBroker` set on the `Interactor`.

The above algorithm is using a lot of different parts in a very decoupled framework. Almost any concrete implementation of an `InteractionBroker` can be connected to any type of `Interactor` for instance. This is very powerful but can be a bit hard to grasp since you can't count on the types of the classes to see what belongs to what.

TheJavaDocs of the difference classes will specify what it's used for and what information it needs to do that. There are also some static helper methods in `AShapeUtil` that helps to create some if the more used interactions, such as setting a `Cursor` for a mouse over event.

### AShape Properties

All properties for an `AShape` are stored as a name/value pair in a `Map`. This means that the `AShape` has very few dedicated get/set method pairs, it has instead `PropertyKey` constants for accessing the properties (you can also have your own key names of you want).

The downside if this is that type checking for the properties isn't possible and the API is rather sparse.

The upside is that this is very extensible and powerful when combined with *overrides*, as explained below. It is also very simple to subclass and make your own `AShape`s since you can use the already existing property constants or just provide new ones, which is a one-liner.

Another positive thing is that it's automatically persistable to XML through the API of the `AbstractAShape` and the beans XML persistence technology introduced in Java SDK 1.4. There are even convenience methods in `AShapeUtil` that makes loading and saving of `AShape`s a one-liner. As long as you store the `AShape`'s properties with the provided `putAttibute(..)/getAttribute()` methods, they will be possible to override (explained below) and automatically persistable to XML.

Please see the API JavaDocs for information on the properties and the expected types. Every `AShape` subclass has a list of properties it uses. Most properties (as are defined directly in the `AShape` interface. Some that are only interesting to a specific implementation, may be defined in that particular class.

### Interactor

The Interactor is the central place where everything is connected. It contains the `InteractionBroker`, `Interactions` and is the posting point for `InputEvents`. All other parts are retrievable from the `Interactor` or it's contained objects. Below are the components that is aggregated in the `Interactor`.

### Overrides

A unique feature of `AShape` is that of overrides. Since every property of an `AShape` is stored in a `PropertyKey`/value map it is very easy to intercept the getting and setting of every property. One such interception is the notion of *overrides*.

For every retrieval of a property, e.g. `Paint`, `PlaceRect`, `Font` and such, all `Interactor`s are asked if they have an overridden object to return for that particular property name. This means that is is very simple to exchange all, a sub set or just one of the properties for an `AShape` without actually changing it's state, and without changing properties that is unknown at the time of writing. The only thing you change is the `Interactor` since it contains the override `Map`.

This is for instance how normally you would set a different color for a mouse over:ed `AShape`, you set an override for the background paint on the corresponding `Interactor`. This will work both for the one-to-one pattern as well as the stamp pattern, as described above.

With this override pattern it is also very easy to restore the original value, you just restore (remove) the override and the original value is visible once again. This means that there is no need to save the original value and late restore it.

The reason the you can't just set the new value on the `AShape` itself is that it is normally used as a rubber stamp and for instance the shape can be used to paint all Activities in a date area (this is even the normal case). If for instance a new color was set on the shape when it was mouse over:ed then **all** activities would change color. The override is connected to the *interacted* through the `Interactor` so it is overriding that property only for one paint (the mouse over:ed one).

### Interaction

An interaction is a concrete class that describes a trigger that, when it happens, the framework should check the validity of an `Expression`. If that `Expression` is evaluated to `true` it also contains a `Command(s)` that should be executed. It isn't more complicated than that since it is very generic, yet is is powerful just because it can describe almost any `if-then` constellation.

**InteractionBroker**
It is the interpreter of `Command`s. Since `Command`s are very generic something is needed to interpret what that `Command` wants to do, if that is not contained in the `Command` itself.

**Command**
The `Command` is a concrete class and contains a generic (again) description of something **to do**. It consists of the *command string*, a *property/value pair* that may be use to explain it further, and a *target*.

**CommandSet**
A collection of link `Command`s. Is an interface and is implemented by `DefaultCommand`. It is used to chain multiple commands together for one interaction. You can use `CompositeCommand` for this.

**Expression**
An `Expression` is something that can be evaluated to `true` or `false`. The only thing you have for sure is a `PropertyProvider` that you can use to *get* values for certain properties. `Expression` is an interface and the concrete implementations provides some context to which properties is interesting and how they should be evaluated.

For instance the `LogicalExpression` denotes a comparison algorithm where you provide a *property name* and a *value* that is should relate to with anyone of it's ten operators (e.g. *equals*, *is_null* and *in_collection*). The actual object to compare the give value to is gotten from the mandatory `PropertyProvider`.

Since the `PropertyProvider` can be different for every evaluation this makes for a great way to describe expressions like: "If the object that the mouse is over right now is named *background*". Currently `ActivityInteractor` and `MouseKeyInteractor` implement the `PropertyProvider` interface and will thus pass them self as the `PropertyProvider` to the `Expression`'s evaluate method. This means they can return property values for things like a list of named areas the mouse was over when it was pressed (`MouseKeyInteractor.PROP_MOUSE_PRESSED_LIST`).

**Static Overrides**

Static overrides is a simple and very extensible way to control overrides without the need for the Command pattern described above. It means that you can call a static method on the `ActivityInteractor` class to set an `OverrideFilter` that contains code that either override a property or not. Here is some example code:

```
String shapeName = AshapeUtil.DEFAULT_OUTLINE_SHAPE_NAME;
ActivityInteractor.setStaticOverride(shapeName, AShape.A_PAINT, new OverrideFilter() {
   public Object getOverride(Object subject, Object defaultObject)
   {
      Activity activity = ((ActivityView) subject).getModel();

      if (activity.getStates().isStateSet(GenericStates.SELECTED_BIT))
         return Color.RED;

      return defaultObject;
   }
});
```

The code adds an override filter that is checking if the Activity is selected, and if it is, returns the color red. If the activity isn't selected it returns the default object which is the normal color in this case.

The code above can be altered in many ways to programatically alter how `AShape`s look depending on some condition of the "subject" it is decorating.

### Putting it All Together

This package has a lot of different parts, and what parts that can be combined together to make a meaningful interaction specification is not entirely obvious. Therefore you should probably look at the demo application and the source code for `AShapeUtil`, which contains a lot of static `AShape` creation methods.

### Listening for Mouse Events

This is new for v6.0 if MiG Calendar component. It offers the same observer pattern for MouseEvents that Swing has. You can easily just listen for instance for mouse over events and change the color (or whatever) using your custom code. This is much easier than specifyin this with `Interactor`s as you had to do pre 6.0.

This is easier to understand with some example code:

```
myAShape.addMouseListener(new MouseInteractionListener() {
   public void mouseInteracted(MouseInteractionEvent e) {

      if (e.getEventKey() == MouseKeyInteractor.MOUSE_OVER_CHANGE) {

         String shapeName = AShapeUtil.DEFAULT_OUTLINE_SHAPE_NAME;
         MouseKeyInteractor inter = e.getMouseKeyInteractor();
         PropertyKey moListKey = MouseKeyInteractor.PROP_MOUSE_OVER_LIST;

         if (MouseInteractionEvent.isShapeInList(shapeName, moListKey)) {
            inter.addOverride(shapeName, AShape.A_PAINT, "myid", Color.RED);
         } else {
            inter.removeOverrideById("myid");
         }
      }
   }
});
```

The code checks if the outline shape for the default shape is in the "mouse over:ed" list. If it is; an override for the `Paint` is is set to `Red`. If not; the override is removed (if there) to expose the original paint again effectively making a mouse over effect. The "`myid`" is just the id of the override so we can remove it later.

The `MouseKeyInteractor` contains references to the the keys for the different event types as well as the keys for retrieving the `List`(s) and values that contains state information such as which is the topmost mouse over:ed `AShape` name.

There is a lot of state information contained in the MouseInteractionEvent. You can listen for just about anything and react to this with a reference to the `Interactor` that can adjust the state (by adding an override) for that particular `AShape` "stamp".

# Animations

### The Basics

The `ashapes.animation.*` package contains all that is needed to create animated `Ashape`s. They can even be animated when for instance the user mouse-overs it. It consists of an `Animator` that controls an `Animation`s over a `TimeLine` to produce the result.

The framework is based on the absolute time and not delay between frames. This but gives much better quality since even a very slow target environment will play the animations

in absolute time, they will never lag behind. If the computer can't keep up, for whatever reason, frames are dropped to keep up rather than played in turn.

Time lines can also progress non-linear. This means that you can provide any custom `Function` that will outline the progress of the animation. For instance it can slow down or accelerate. It can even have advance sinus-like equations to make animations oscillate.

**Animation**

`Animation` is an interface that describes how one object fades into another, given a value between `0.0` and `1.0` (`float`). Subclasses provides the actual implementation that does the transformation. For instance `ColorAnimation` can "animate" between two `Color`s.

The `AbstractAnimation` class that all current `Animation`s subclass provides for caching the values if the frame count is provided. This is normally a must for animating `Image` transitions for instance but it can of course be turned off.

**The Animator**

The `Animator` class is an abstract class that provides basic boiler plate functionality for controlling an animation. The `OverrideAnimator` is the default implementation and it provides everything needed to animate some aspect of an `AShape` through overrides (as explained above). It does the animating in an indirect manner, by overriding properties on the `AShape`. It uses its own `Thread` to set overrides in the `Interactor`, which means that for instance one activity can (optionally) be animated when mouse over:ed. Setting the override triggers a repaint and the next frame will be shown.

This method of animation means that **any** property of `AShape` that can be overridden can be animated, and since all properties can be overridden, they can all be animated.

There are seven different types of Animations delivered with the framework and it is very simple to write your own. You just have to extend `AbstractAnimation` and write one method that returns the object that corresponds to the `float` value (`0..0` to `1.0`).

**Time Lines**

A `TimeLine` denotes how frame numbers should relate to

absolute time. It has duration (time), tick count, repetition type/count and a `Function`. It manages this information including the ability to *pause*, *stop*, *reverse* and *resume* animations.

Time lines are ever changing objects since there state is a function of time + earlier state, which progresses outside our control. To be able to read multiple properties from a `TimeLine` its state is "freezable" and when frozen they will not change and you can read multiple values that will be correct in relation to each other. You will probably not have to bother with this though, but it is a feature needed to write precise `Animator`s for instance.

### Functions

This framework has the possibility to map frames to absolute time non-linearly. This means that it is easy to do accelerated fade ins and outs and to, for instance, create oscillating animations. The default Function is `LinearFunction` which maps time and frame in a linear fashion. Also provided is a class `ExpFunction` which maps frames and time in exponentially, making it simple to make accelerated or retarding animations.

## Swing Interoperability

### The Basics

Swing is the standard GUI framework provided by Sun Microsystems and is used to create user interfaces in Java. The `AShape` component interacts with Swing both ways. It can be used within a Swing `JComponent` and any JComponent can be contained in a `AShape` and still be used the normal way.

### JComponents in AShapes

There is a special `AShape` thatcan contain a `JComponent`, namely `JComponentAShape`. It will set the bounds of the "wrapped" `JComponent` to the same as it would get itself and as such provides a transparent glue to have a `JComponent` in an `AShape`. There are no special requirements on the `JComponent` other than it still, as all `JComponent`s, need a valid parent. Normally this should be

the same Component that the AShape itself is painted on.

### AShapes in JComponents

Also for this, reversed, use case there exist a special class, AShapeComponent. It is a JComponent that:

- Forwards InputEvents (MouseEvents and KeyEvents to the "wrapped" AShape's Interactor(s) for processing before itself handles them (if not consumed).
- Adapts the Swing repaint cycle to migrate to that of the AShape's.
- Registers itself to listen for InteractionEvents on the AShape and when they occur re-dispatches them to listeners added to the AShapeComponent.
- Installs the normal types of Interactors on the wrapped AShape, if desired.
- Handles the actual JComponent to be positioned relative to. This means that AShapeComponent can be used as any type of renderer component, for instance a Component returned by TableCellRenderer or TreeCellRenderer. This opens up for having AShapes in JTrees and JTables!

### Cursors

Mouse cursors is a property of the Component class. Every AShape can have a Cursor attribute set on it with the key AShape.A_MOUSE_CURSOR.

Interactions have to be installed on the AShapes to actually transfer this set Cursor to the Component on which it is rendered. It can be done manually but calling AShapeUtil.enableMouseOverCursor(RootAShape root) is the easiest way. It installs Interactions to set the Cursor depending of which sub shape the mouse is over.

Note!

Since there is no way in Swing to know which is the default Cursor for a certain point in the Component the cursor can not be restored when it leaves the shape. You must listen for MouseEntered events in the Component that is drawing the AShapes and set the Cursor to the desired one. The AShape framework will send synthesized MouseEntered and MouseExited events to the Component when the mouse exit and enter the hit area of a AShape.

### Events

Interactors are the objects responsible for dispatching and processing InputEvents. This is normally done by the MouseKeyInteractor with a DefaultInteractionBroker as the receiver of the Commands run when an Interaction should occur.

How does the MouseKeyInteractor get the events? The simple fact is that it doesn't. The Container in which the AShape is to be painted (for instance the AShapeComponent as explained above) has to provide them to the Interactors.

This *could* be done automatically by the Interactor but it would break some applications that also listens on the Events very early in the dispatching process so this way is more compatible, but you will have to remember to do it if you provide your own Component container. Here is the code from AShapeComponent that forwards the InputEvents to the Interactors of the AShape. Note that **all** Interactors of all decorated entities must be notified. This method will override the processEvent() of the Component class to first re-dispatch them to the Interactors.

```
/** Overridden to let all activity views have fist chance to interact with the events
and
 * if they are consumed disregard them for further processing.
 * @param e The event.
 */
protected void processEvent(AWTEvent e)
{
  if (e instanceof InputEvent) {
    Interactor[] interactors = rootShape.getInteractors();
    if (interactors != null) {
      InputEvent ie = (InputEvent) e;
      for (int i = 0; i < interactors.length; i++)
          interactors[i].processEvent(ie);

      if (ie.isConsumed())
        return;

      // Resets the Cursor. Only needed if the AShape changes it in the first place
      if (e instanceof MouseEvent) {
        if((MouseEvent) e).getID() == MouseEvent.MOUSE_ENTERED)
          setCursor(null);   // Whatever cursor that is to be used
      }
    }
  }

  super.processEvent(e);
}
```

## Continued Reading

This document has given you the basics for experimenting on your own. There are currently no GUI tool, such as a vector paint application, to experiment with all the different aspects of the `AShape`s. It is therefore suggested that you set up a basic Swing application that just shows an `AShapeComponent` and set a `RootAShape` with `AShape`s that you can easily change and experiment with.

If you have requirements that can not be met by the properties for the built in `AShape` types you must resort to writing code in order to customize it further. Almost all aspects of an `AShape` can be exchanged and/or overridden to extend just about everything. This is by design. To be able to do this you will need to have a thorough understanding of how the different parts fits together. The **API JavaDoc** is a must read for doing this. Also, the support forums at http://www.miginfocom.com/forum/ can be used to ask questions.

When you have gotten acquainted with the component you are welcome to make feature requests. Suggestions for the `AShape` component are appreciated and those should be posted in the forums.