Configuring Automatic Layout

This page describes how automatic layout can be configured for a given application. This includes how layout options can be set on graph elements, and how they are applied by KIML during the layout process. After having read this, you should be able to answer the following questions:

- What are *layout options*?
- How do layout algorithms support layout options?
- How do layout options end up in KGraph elements?
- How can I set layout options on elements programmatically?

This page does not list the available layout options, and neither does it explain any of them. You can find a list of layout options provided by KIML over here.

Contents

Layout Options and What They Are Good For

Even the most basic layout algorithm provides some settings for you to play with. This might be something as simple as the space left between different nodes, or something as complex as changing how node labels are placed and how that influences the size of each node. Each such setting must be registered with KIML as a *layout option*, and each algorithm must specify exactly which of these options it supports. Registering a layout option is done through one of KIML's extension points and can look like this:

```
<extension point="de.cau.cs.kieler.kiml.layoutProviders">
    <layoutOption
        id="de.cau.cs.kieler.nodeLabelPlacement"
        name="Node Label Placement"
        description="Hints for where node labels are to be placed; if empty, the node label's position is not
modified."
        advanced="true"
        appliesTo="nodes"
        type="enumset"
        class="de.cau.cs.kieler.kiml.options.NodeLabelPlacement"
        default="">
    </layoutOption>
</extension>
```

Such declarations are provided by layout algorithm developers, but not by tool developers who merely want to connect the layout infrastructure to their diagram viewers. Let's walk through the attributes available for layout options (not every available attribute appears in the example above):

- `id` – A unique identifier for this layout option. It is recommended that the identifier be prefixed by the plug-in name, to guarantee uniqueness.
- `type` – Defines the data type of this option; must be either `boolean`, `string`, `int`, `float`, `enum`, `enumset`, or `object`. The types `enum`, `enumset`, and `object` require the `class` attribute to be set.
- `name` – A user friendly name of this layout option, to be displayed in the UI.
- `description` – A user friendly description of this layout option, to be displayed in the UI. The description should contain all information needed to understand what this option does.
- `advanced` – Whether the option should only be shown in advanced mode in the layout view; default is `false`.
- `appliesTo` – A comma separated list of targets on which the layout option can be applied; a target can be either `parents` (for nodes that contain further nodes), `nodes` (for all nodes regardless of whether they contain further nodes or not), `edges`, `ports`, or `labels`. If omitted, the layout option is not shown to the user in the layout view, which is a good thing for options that will be set programmatically anyway.
- `class` – An optional Java class giving more detail on the data type. For `enum` and `enumset` options this attribute must hold the Enum class of the option. For `object` options it must hold the class name of an `IDataObject` implementation.
- `default` – The default value to use when no other value can be determined for this option.
- `lowerBound` – An optional lower bound on the values of this layout option.

- `upperBound` – An optional upper bound on the values of this layout option.
- `variance` – An optional variance for values of this layout option. The variance is taken as multiplier for Gaussian distributions when new values are determined. Options with uniform distibution, such as Boolean or enumeration types, do not need a variance value, since all values have equal probability. A variance of 0 implies that the option shall not be used in automatic configuration, regardless of its type.

The latter three attributes are used when a layout configuration is determined automatically, e.g. with an evolutionary algorithm. They are mainly meant for scientific experiments and can be ignored in most applications.

If a layout algorithm supports a particular layout option, it must tell KIML so. Here's an example:

```
<extension point="de.cau.cs.kieler.kiml.layoutProviders">
    <layoutAlgorithm ...>
        <knownOption
            option="de.cau.cs.kieler.borderSpacing"
            default="20">
        </knownOption>
    </layoutAlgorithm>
</extension>
```

This tells KIML that the defined layout algorithm supports the border spacing option. And even more, it overrides the default value declared by the layout option and sets it to 20.

The meta data gathered from the extension point are made available through LayoutMetaDataService. For direct programmatic access, some of that information is duplicated with constants in the class LayoutOptions. The layout option declared above, for example, is available as `LayoutOptions.NODE_LABEL_PLACEMENT`.

The Layout Option Manager

By now, we have an idea of what layout options do and why they are important in the first place. However, we haven't looked at how layout options end up on KGraph elements yet. This is where the `LayoutOptionManager` comes in. If you are not interested in the internal details, but want to configure automatic layout for your diagram viewer or editor, you may skip this section and proceed to programmatically setting layout options.

After a diagram layout manager has finished turning a given diagram into its KGraph representation, the layout option manager is asked to enrich the KGraph elements with layout options. The option values can come from different sources: the user might have set some using the layout view; there might be some defaults for certain kinds of diagrams; or the programmer might have decided to attach some layout options to certain elements for just this one layout run. Whatever the source, the options manager is in charge of collecting all these layout option values and making sure they find their way to the correct KGraph element. To start off with a clean plate, it first makes sure there are no layout options attached to the KGraph elements. It then does two things: collect every eligible source of layout options, and transfer values of layout options to the associated KGraph elements. Sounds easy enough.

The question remains how the layout options sources work. Each source is represented by a class that implements the ILayoutConfig interface, called a *lay out configurator*. KIML currently provides the following layout configurators, each representing a particular source of layout options, listed here in order of increasing priority:

- `DefaultLayoutConfig` – Applies fixed default values defined in the meta data of layout options. This is important for the Layout View, which displays the default values if nothing else has been specified.
- `EclipseLayoutConfig` – Users can define default layout options to be set on elements that meet certain criteria via the KIML preference page. This layout configurator takes these options and applies them. Furthermore, it also applies options configured through the extension point.
- `SemanticLayoutConfig` – An abstract superclass for configurators that base their computation of layout option values on the *semantic* model, a.k.a. *domain* model.
- `GmfLayoutConfig` / `GraphitiLayoutConfig` – These configurators apply layout option values set by the user in the Layout View. The values are stored in the notational model file of a diagram.
- `VolatileLayoutConfig` – A configurator for setting certain layout option values in one particular layout run. As the name says it, the values are volatile and thus they are not persisted.

The options manager collects all available and applicable layout configurators and sorts them by priority. For every graph element, each configurator is asked to provide layout options, starting with the one with lowest priority and working through the priority chain. Hereby configurators with higher priority are able to override values set by those with lower priority.

A Few Details on Layout Configurators

What we just learned is a bit of a simplification of what happens. Before we look at the details, let's take a look at the methods each layout configurator provides:

```
public interface ILayoutConfig {
    int getPriority();
    Object getOptionValue(LayoutOptionData optionData, LayoutContext context);
    Collection<IProperty<?>> getAffectedOptions(LayoutContext context);
    Object getContextValue(IProperty<?> property, LayoutContext context);
}
```

It is not hard to guess what `getPriority()` does: it returns the priority a given layout configuration has. If two layout configurations set a layout option to different values on a given graph element, the value set by the configuration with higher priority wins.

The interface discerns between *option* values and *context* values. Option values are what we have been talking about all the time, values assigned to layout options. Which particular values the configurator should apply depends on the given LayoutContext, which is a property holder with references to the diagram element currently in focus. For instance, the object representing an element in the diagram viewer is accessed with `context.getProperty(LayoutContext.DIAGRAM_PART)`. Similarly, the corresponding KGraph element is mapped to the property `LayoutContext.GRAPH_ELEM`, and the domain model element is mapped to `LayoutContext.DOMAIN_MODEL`. Each configurator is free to put additional information into the context, caching it for faster access and enabling to communicate it to other configurators. `getAffectedOptions(LayoutContext)` should return a collection of layout options for which the configurator yields non-null values with respect to the given context. The options can be referenced either with LayoutOptionData instances obtained from the LayoutMetaDataService or with Property instances from the constants defined in LayoutOptions. The actual value for a layout option is queried with `getOptionValue(LayoutOptionData, LayoutContext)`. The method `getContextValue(IProperty, LayoutContext)`, in contrast, is used to obtain more detailed information on the given context. For instance, the context may contain a reference to an element of the diagram viewer; only a specialized configurator made for that diagram viewer knows how to extract a reference to the corresponding domain model element from the given diagram element, so it can encode this knowledge in `getContextValue(…)` by returning the domain model element when the given property corresponds to LayoutContext.DOMAIN_MODEL.

This may seem complicated, and it is, but the good news is that the vast majority of developers will not need to dig that deep into the layout configuration infrastructure. There are easier ways to specify configurations, as described in the following section.

Programmatically Setting Layout Options

So with all these layout configurators available, how do you actually go about setting values for layout options programmatically? Well, as always: it depends.

Using the Extension Point

The recommended way to configure your layout is to use the `layoutConfigs` extension point. It offers three different kinds of configurations, explained in the following.

staticConfig

A `staticConfig` element can set one value for one layout option in the context of a particular diagram element type. Let's see an example:

```
      <staticConfig
            class="org.eclipse.emf.ecore.EReference"
            option="de.cau.cs.kieler.edgeType"
            value="ASSOCIATION">
      </staticConfig>
```

Here `class` refers to a domain model class, in this case the `EReference` class from the Ecore meta model defined by EMF, and `option` refers to a layout option through its identifier. The meaning of this declaration is that whenever automatic layout is requested for an Ecore class diagram, the `edgeType` option is set to `ASSOCIATION` for all edges linked to instances of `EReference`. Since the domain model (*abstract syntax*) is independent of the specific diagram viewer (*concrete syntax*), this configuration is applied to all diagram viewers that use the Ecore meta model.

Alternatively to domain model elements, `staticConfig` may also reference concrete syntax elements:

```
      <staticConfig
            class="org.eclipse.emf.ecoretools.diagram.edit.parts.EClassESuperTypesEditPart"
            option="de.cau.cs.kieler.edgeType"
            value="GENERALIZATION">
      </staticConfig>
```

This layout option value is applied only to edges linked to instances of `EClassESuperTypesEditPart`, which is a concrete syntax element of the Ecore Tools class diagram editor. Other Ecore meta model editors are not affected by this declaration. This distinction is particularly useful for meta models that are accessed with multiple different editors, as is often the case for UML tools.

A third variant is the use of *diagram types*, as in this example:

```
      <diagramType
            id="de.cau.cs.kieler.layout.diagrams.classDiagram"
            name="Class Diagram">
      </diagramType>
      <staticConfig
            class="org.eclipse.emf.ecore.EPackage"
            option="de.cau.cs.kieler.diagramType"
            value="de.cau.cs.kieler.layout.diagrams.classDiagram">
      </staticConfig>
      <staticConfig
            class="de.cau.cs.kieler.layout.diagrams.classDiagram"
            option="de.cau.cs.kieler.edgeRouting"
            value="SPLINES">
      </staticConfig>
```

A diagram type can be declared with a `diagramType` element and can be associated with an abstract syntax or concrete syntax class using the `diagramType` option, as shown in the first `staticConfig` declaration in the example above. The second `staticConfig` sets an option for the declared diagram type by using its identifier in the `class` attribute. This kind of indirection is very useful when you have n model classes and you want to set m layout options for each of those classes. Instead of writing $n \times m$ static declarations, you assign a diagram type t to each of the n classes and then declare the m layout options for t, resulting in $n + m$ option declarations (in many cases $n + m < n \times m$).

A further use of diagram types is for the selection of layout algorithms: a layout algorithm may declare that is is especially suited to process diagrams of certain type t. If the diagram type t is assigned to a diagram viewer, the most suitable layout algorithm is chosen automatically for that viewer.

The following diagram types are predefined in KIML:

- de.cau.cs.kieler.layout.diagrams.stateMachine – All kinds of state machines, statecharts, etc.
- de.cau.cs.kieler.layout.diagrams.dataFlow – All kinds of data flow diagrams, e.g. actor diagrams, block diagrams, certain component diagrams, etc.
- de.cau.cs.kieler.layout.diagrams.classDiagram – Class diagrams as defined by the UML, but also meta model diagrams such as the Ecore format.
- de.cau.cs.kieler.layout.diagrams.usecaseDiagram – UML use case diagrams.
- de.cau.cs.kieler.layout.diagrams.boxes – Unconnected boxes (graphs with no edges), e.g. parallel regions in statecharts.

semanticConfig

A `semanticConfig` element registers a subclass of SemanticLayoutConfig:

```
      <semanticOption
            class="de.cau.cs.kieler.synccharts.Scope"
            config="de.cau.cs.kieler.synccharts.diagram.custom.AnnotationsLayoutConfig">
      </semanticOption>
```

Similarly to `staticConfig` entries, the `class` attribute refers to which model elements the configuration is applied. However, only domain model (a.k.a. *semantic* model) classes may be referenced. The `config` attribute names a concrete implementation of the semantic layout configurator.

The advantage of this kind of configuration compared to `staticConfig` declarations is that it may perform arbitrary analyses of the domain model. For instance, different option values may be computed depending on certain properties of the domain model elements. This approach can be used to enable annotations of domain model elements. When the domain model is stored with a textual format, e.g. defined with Xtext, such annotations can be written in the source file that specifies the model:

```
@portConstraints FIXED_SIDE
@minWidth 20.0
@minHeight 15.0
entity IdentityActor
{
    @portSide WEST
    port Input;

    @portSide EAST
    port Output;
}
```

The source file annotations can be translated to KIML layout options with a semantic layout configurator, which is registered to each domain model class where annotations can occur.

customConfig

This extension element can be used to register arbitrary implementations of ILayoutConfig. However, this is required only for some experimental configurators used in research. Tool developers normally do not need to use this kind of extension element.

Using Volatile Configurators

The class VolatileLayoutConfig is meant for programmatic layout configuration. It stores layout option values in a hash map. Values are either set globally, that means they are applied to all graph elements, or with a specific context. Global values are easy to configure:

```
DiagramLayoutEngine.INSTANCE.layout(workbenchPart, diagramPart,
        new VolatileLayoutConfig()
            .setValue(LayoutOptions.ALGORITHM, "de.cau.cs.kieler.klay.layered")
            .setValue(LayoutOptions.SPACING, 30.0f)
            .setValue(LayoutOptions.ANIMATE, true));
```

If multiple configurators are passed to the DiagramLayoutEngine, the layout is computed multiple times: once for each configurator. This behavior can be used to apply different layout algorithms one after another, e.g. first a node placer algorithm and then an edge router algorithm, as in this example:

```
DiagramLayoutEngine.INSTANCE.layout(workbenchPart, diagramPart,
        new VolatileLayoutConfig()
            .setValue(LayoutOptions.ALGORITHM, "de.cau.cs.kieler.klay.force"),
        new VolatileLayoutConfig()
            .setValue(LayoutOptions.ALGORITHM, "de.cau.cs.kieler.kiml.libavoid"));
```

If you want to use multiple configurators in the same layout computation, use a CompoundLayoutConfig:

```
DiagramLayoutEngine.INSTANCE.layout(workbenchPart, diagramPart,
        CompoundLayoutConfig.of(config1, config2, ...));
```

Setting layout option values with a specific context is done with this method of VolatileLayoutConfig:

```
public <T, C> VolatileLayoutConfig setValue(final IProperty<? super T> option, final C contextObj,
            final IProperty<? super C> contextKey, final T value)
```

Don't be scared by the rather cryptic declaration. The arguments `contextKey` and `contextObj` determine in which context the option value is to be applied. For instance, using `LayoutContext.DOMAIN_MODEL` as context key and a specific domain model element as context object, the value is applied exactly to the graph element that is linked to that model element. If you want to refer to an element of the diagram viewer, i.e. the concrete syntax, use `LayoutContext.DIAGRAM_PART` as context key. The return value is the volatile layout configurator itself, allowing for a builder pattern.

Configuration During Layout Graph Construction

Volatile configurators are also useful for the implementation of diagram layout managers (IDiagramLayoutManager). These implementations are responsible for creating layout graphs following the KGraph meta model from a given diagram viewer (method `buildLayoutGraph(…)`). For some layout options it is reasonable to determine concrete values while the layout graph is built, e.g. for the minimal width and height of nodes:

```
KNode childLayoutNode = KimlUtil.createInitializedNode();
KShapeLayout nodeLayout = childLayoutNode.getData(KShapeLayout.class);
Dimension minSize = nodeEditPart.getFigure().getMinimumSize();
nodeLayout.setProperty(LayoutOptions.MIN_WIDTH, (float) minSize.width);
nodeLayout.setProperty(LayoutOptions.MIN_HEIGHT, (float) minSize.height);
```

The problem is that the layout option manager that applies all configurators to the layout graph removes any option values that have been set directly on the graph elements, hence the configuration done in the previous example has no effect on the layout process. But do not fear, for salvation is near:

```
mapping.getLayoutConfigs().add(VolatileLayoutConfig.fromProperties(mapping.getLayoutGraph(), PRIORITY));
```

The variable `mapping` refers to the LayoutMapping instance created in `buildLayoutGraph(…)`. The static method `fromProperties(…)` offered by VolatileLayoutConfig creates a configuration that contains all the layout option values that have previously been seen directly on the graph elements. By adding this configuration to the layout mapping, we make sure it is considered by the layout option manager and the options are applied to the graph elements exactly as we have specified. Happy end.

If you are uncertain about which value to use for `PRIORITY`, take something like 25.

Adding Support for the Layout View

The Layout View empowers users to directly modify the layout configuration for the currently viewed diagram. This power, however, comes with a price. Tool developers implementing a IDiagramLayoutManager additionally have to provide an implementation of IMutableLayoutConfig that loads and saves layout option values in a way that they can be stored persistently with the respective diagram. Good examples of such configurators are GmfLayoutConfig a nd GraphitiLayoutConfig for GMF and Graphiti diagrams, respectively. The GMF implementation stores option values as *styles* in the GMF *Notation* model, while the Graphiti implementation stores the values as *properties* in the Graphiti *Pictogram* model. If you are developing an editor based on GMF or Graphiti, simply reuse these implementations and you're fine. Otherwise, read this section to learn how to implement a suitable configurator.

A *mutable* layout configurator is one that can not only read option values, but also write them. Most interface methods are rather self-explanatory, therefore we will consider only the `getContextValue(IProperty, LayoutContext)` method here. This method receives a LayoutContext and should return a value that corresponds to the given property, if possible, and `null` otherwise. The starting point is usually the current value of `LayoutContext.DIAGRAM_PART` in the given context, called the *diagram part*, which refers to the currently selected diagram element in the viewer (the abstract syntax element). From this the method should extract more information considering the following other context properties:

- `LayoutContext.DOMAIN_MODEL` – The domain model element linked to the current diagram part.
- `LayoutContext.CONTAINER_DIAGRAM_PART` – The diagram part that corresponds to the graph or subgraph that contains the current diagram part. This is called the *container*. If The current diagram part is already the top-level element of the diagram, then there is no container and `null` should be returned.
- `LayoutContext.CONTAINER_DOMAIN_MODEL` – The domain model element linked to the container.
- `LayoutContext.OPT_TARGETS` – A set containing the kind of graph element that corresponds to the current diagram part, referenced with the enumeration LayoutOptionData.Target. If the diagram part is a node, for example, the set should contain the value `NODES`. If the node is also a container for a subgraph, the set should additionally contain the value `PARENTS`.
- `DefaultLayoutConfig.HAS_PORTS` – If the current diagram part is a node, the returned value for this property should be `true` or `false` depending on whether the node has any ports or not. Ports are explicit connection points for edges; they occur frequently in data flow diagrams.
- `DefaultLayoutConfig.CONTENT_HINT` – If the diagram contains an annotation about which layout algorithm to use for the content of the current diagram part, the returned value for this property should be the identifier of that algorithm. This is the same kind of annotation that is accessed through `getOptionValue(…)`, i.e. a value set by the user with the Layout View.
- `DefaultLayoutConfig.CONTAINER_HINT` – The same as for `CONTENT_HINT`, but referring to the container.
- `EclipseLayoutConfig.EDITING_DOMAIN` – If your diagram editor needs an EMF editing domain in order to modify annotations of layout options, then such an editing domain should be returned for this property.

An instance of your self-made configurator should be returned by the `getDiagramConfig()` method of your diagram layout manager.