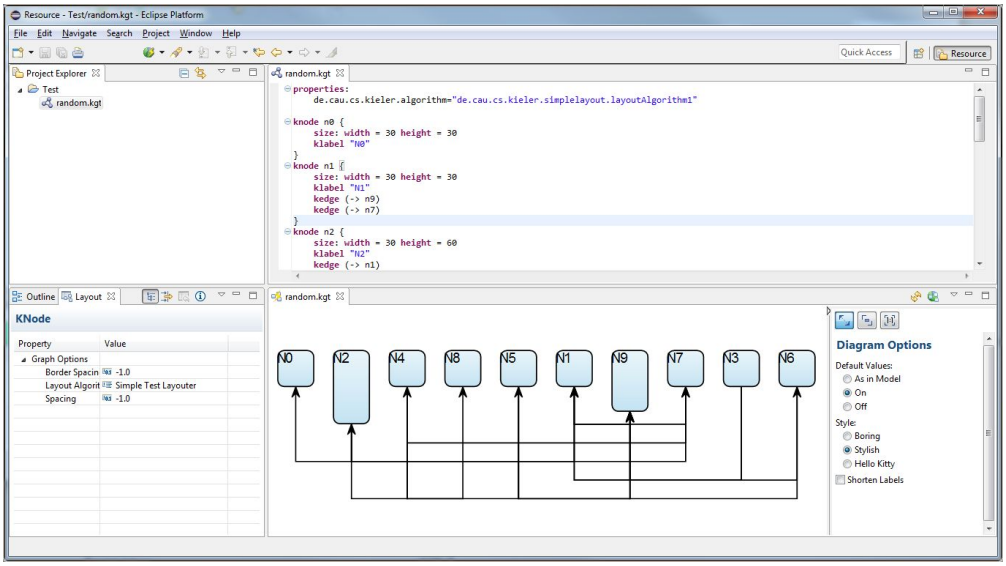
Layout Algorithms

Welcome to this tutorial! We will work our way through installing a proper Eclipse setup and developing a first very basic layout algorithm. The layout algorithm will integrate with ELK (Eclipse Layout Kernel), our very own framework that connects graphical editors with layout algorithms. Once you're finished, you should be able to write layout algorithms for ELK. And you should have a running Eclipse-based application that should look something like this:



Preliminaries

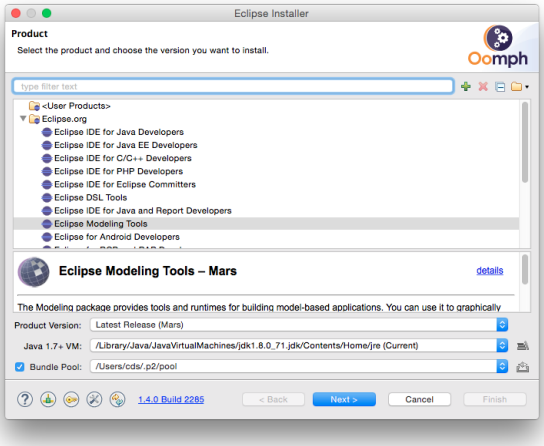
There's a few things to do before we dive into the tutorial itself. For example, to do Eclipse programming, you will have to get your hands on an Eclipse installation first. Read through the following sections to get ready for the tutorial tasks.

Required Software

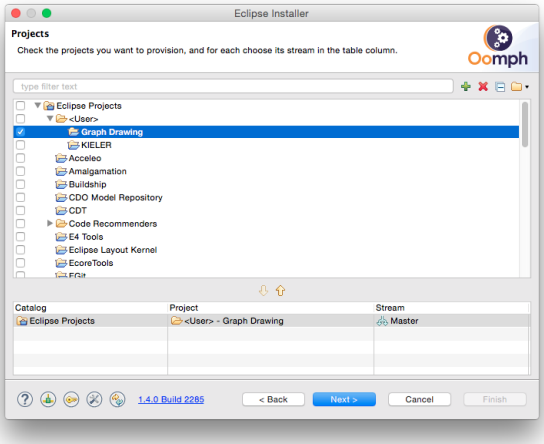
You will first need an Eclipse installation to hack away on layout algorithms with. Since we have a shiny Oomph setup available, this turns out to be comparatively painless:

1. Go to this site and download the Eclipse Installer for your platform. You will find the links at the bottom of the *Try the Eclipse Installer* box.
2. Start the installer. Click the Hamburger button at the top right corner and select Advanced Mode. Why? Because we're computer scientists, that's why!

3. Select *Eclipse Modelling Tools* from the *Eclipse.org* section and click Next.



4. Next, we need to tell Oomph to get everything ready for layout algorithm development. Download our Oomph setup file, click the Plus button at the top right corner and add the setup file to the Github Projects catalog. Select the new Graph Drawing entry by clicking the check box to its left. This will cause an item to appear in the table at the bottom of the window. Once you're done, click Next.



5. Oomph now asks you to enter some more information. You can usually leave the settings as is, except for the Installation folder name. This will be the directory under which all your Eclipse installations installed with Oomph will appear, each in a separate sub-directory. Select a proper directory of your choice and click Next.

Finding Documentation

During the tutorial, we will cover each topic only briefly, so it is always a good idea to find more information online. Here's some more resources that may prove helpful:

- Java™ Platform, Standard Edition 8 API Specification
  As Java programmers, you will already know this one, but it's so important and helpful that it's worth repeating. The API documentation contains just about everything you need to know about the API provided by Java.
- Eclipse Help System
  Eclipse comes with its own help system that contains a wealth of information. You will be spending most of your time in the *Platform Plug-in Developer Guide*, which contains the following three important sections:
  - Programmer's Guide
    When you encounter a new topic, such as SWT or JFace, the Programmer's Guide often contains helpful articles to give you a first overview. Recommended reading.
  - References -> API Reference
    One of the two most important parts of the Eclipse Help System, the API Reference contains the Javadoc documentation of all Eclipse framework classes. Extremely helpful.
  - References -> Extension Points Reference
    The other of the two most important parts of the Eclipse Help System, the Extension Point Reference lists all extension points of the Eclipse framework along with information about what they are and how to use them. Also extremely helpful.
- Eclipsepedia
  The official Eclipse Wiki. Contains a wealth of information on Eclipse programming.
- Eclipse Resources
  Provides forums, tutorials, articles, presentations, etc. on Eclipse and Eclipse-related topics.
- Eclipse Layout Kernel 👍⚠️
  Documentation on how the layout infrastructure works and on how to write your own layout algorithms. This is our project, so if you find that something is unclear or missing, tell us about it!

You will find that despite of all of these resources Eclipse is still not as well commented and documented as we'd like it to be. Finding out how stuff works in the world of Eclipse can thus sometimes be a challenge. However, you are not alone: this also applies to many people who are conveniently connected by something called *The Internet*. It should go without saying that if all else fails, Google often turns up great tutorials or solutions to problems you may run into. And if it doesn't, your advisers will be happy to help.

As far as KIELER documentation is concerned, you will find documentation at the KIELER Confluence. The documentation is not as complete as we (and especially everyone else) would like it to be, however, so feel free to ask those responsible for help if you have questions that the documentation fails to answer.

Developing Your First Layout Algorithm

Now that the preliminaries are out of the way, it's time to develop your first layout algorithm! It will, however, be a very simple one. This tutorial focuses on creating Eclipse plug-ins and on learning how to develop with ELK.

Adding a New Plug-in

We need to create a new plug-in to implement the layout algorithm in. Switch back to the Java or Plug-in Development perspective and follow these steps:

1. Click *File* > *New* > *Other...* > *Plug-in Development* > *Plug-in Project*.
2. Enter `de.cau.cs.kieler.simplelayout` as the project name. Click *Next*.
3. Set the execution environment to *JavaSE-1.8*. (do this for all plug-ins that you create!)
4. Uncheck all checkboxes in the *Options* group and click *Finish*.
5. If Eclipse asks you whether the *Plug-in Development* perspective should be opened, choose either *Yes* or *No*. It doesn't make much of a difference anyway.

Writing the Layout Algorithm

When writing our layout algorithm, we are going to need to be able to access code defined in several other plug-ins. To do that, we need to add dependencies to those plug-ins:

1. In your new plug-in, open the file `META-INF/MANIFEST.MF`. The plug-in manifest editor will open. Open its *Dependencies* tab.
2. Add dependencies to the following plug-ins:
    - `org.eclipse.elk.core`
    - `org.eclipse.elk.graph`
3. Save the editor.

Layout algorithms interface with ELK by means of a layout provider class that has to be created and registered with KIML.

1. Right-click the source folder of your plug-in and click *New* > *Class*.
2. Set the package to `de.cau.cs.kieler.simplelayout`, enter `SimpleLayoutProvider` as the class name, and select `org.eclipse.elk.core.AbstractLayoutProvider` as the superclass. (This will only be available through the *Browse* dialog if you have saved the plug-in manifest editor; if you haven't, Eclipse won't know about the new dependencies yet.)
3. Select *Generate comments* and click *Finish*.

Implement the layout provider class:

1. Add the following constants:

```
/** default value for spacing between nodes. */
private static final float DEFAULT_SPACING = 15.0f;
```

2. Use the following code as the skeleton of the `layout(...)` method:

```
progressMonitor.begin("Simple layouter", 1);
KShapeLayout parentLayout = layoutGraph.getData(KShapeLayout.class);

float objectSpacing = parentLayout.getProperty(CoreOptions.SPACING_NODE);
if (objectSpacing < 0) {
    objectSpacing = DEFAULT_SPACING;
}

float borderSpacing = parentLayout.getProperty(CoreOptions.SPACING_BORDER);
if (borderSpacing < 0) {
    borderSpacing = DEFAULT_SPACING;
}

// TODO: Insert actual layout code.

progressMonitor.done();
```

3. Press *CTRL+SHIFT+O* or select *Source > Organize Imports* from the context menu to add all required imports.
4. It is now time to write the code that places the nodes.Your code should place them next to each other in a row, as seen in the screenshot at the beginning of the tutorial.

Tips

The following tips might come in handy...

- Read the documentation of the KGraph and KLayoutData meta models. The input to the layout algorithm is a `KNode` that has child `KNodes` for every node in the graph. Iterate over these nodes by iterating over the `getChildren()` list of the `parentNode` argument.
- Retrieve the size of a node and set its position later using the following code:

```
KShapeLayout nodeLayout = node.getData(KShapeLayout.class);

// Retrieving the size
float width = nodeLayout.getWidth();
float height = nodeLayout.getHeight();

// Setting the position
nodeLayout.setXpos(x);
nodeLayout.setYpos(y);
```

- `objectSpacing` is the spacing to be left between each pair of nodes.
- `borderSpacing` is the spacing to be left to the borders of the drawing. The top left node's coordinates must therefore be at least `(borderSpacing, borderSpacing)`.
- At the end of the method, set the width and height of `parentLayout` such that it is large enough to hold the whole drawing, including borders.
- A complete layout algorithm will of course also route the edges between the nodes. Ignore that for now – you will do this at a later step.

Before you can test your layout code, you will have to register your new layout provider with ELK.

1. Right-click the `de.cau.cs.kieler.simplelayout` package and select *New > File.*
2. Create a file *simple.elkm* and double click it to open it.
3. When asked whether you want to add the *Xtext nature*, select *yes*.
4. The file is used to specify meta information for your layout algorithm. For this, copy the following code snippet into your editor:

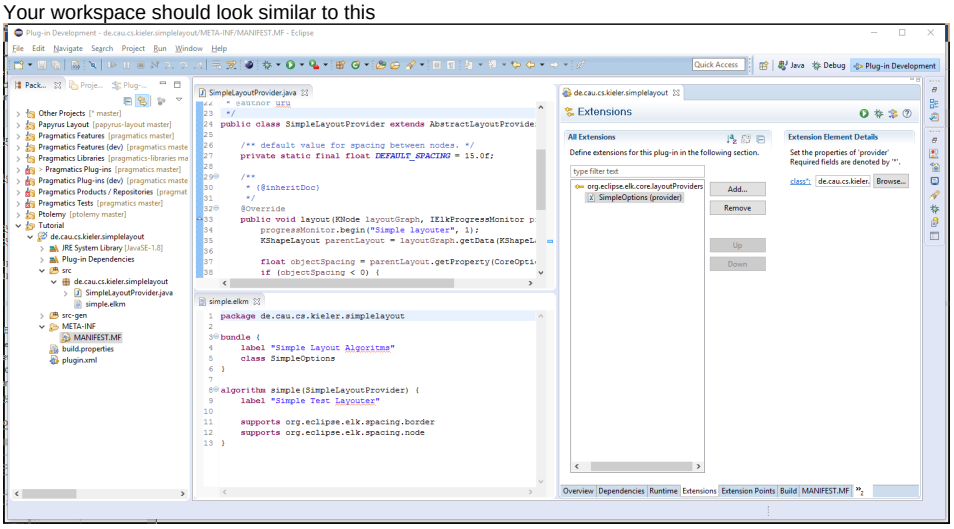
```
package de.cau.cs.kieler.simplelayout

bundle {
    label "Simple Layout Algoritms"
    metadataClass SimpleMetaDataProvider
}

algorithm simple(SimpleLayoutProvider) {
    label "Simple Test Layouter"
    metadataClass SimpleOptions

    supports org.eclipse.elk.spacing.border
    supports org.eclipse.elk.spacing.node
}
```

5. You still have to register the file with Eclipse. Open the `META-INF/MANIFEST.MF` file again and switch to the *Extensions* tab.
6. Add an extension for `org.eclipse.elk.core.layoutProviders`.
7. Right-click the extension and click *New* > *provider*.
8. Set the class to your bundle's meta data provider class name (use the browse button and enter `SimpleMetaDataProvider`). Note that `SimpleMetaDataProvider` is automatically generated from the *.elkm* file you created. Its name is specified by the `metadataClass` keyword in the `bundle` section. What is also created is the `SimpleOptions` class, which contains everything you need to access layout options from within your layout algorithm.
9. Save the editor
10. Your workspace should look similar to this



We will now have to add a new run configuration that will start an Eclipse instance with your layout code loaded into the application, ready to be used.

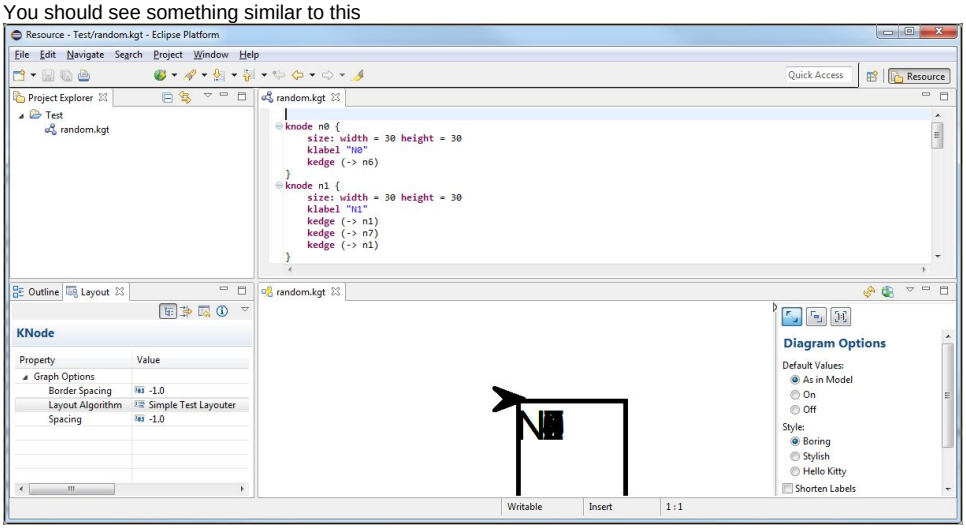
1. Click *Run* > *Debug Configurations...*
2. Right-click *Eclipse Application* and click *New*. Set the configuration's name to `Layout Test`.
3. In the *Arguments* tab, make sure the the program arguments include `-debug` and `-consoleLog`.
4. On the *Plug-ins* tab, set *Launch with* to *plug-ins selected below only*.
    a. Click *Deselect All*.
    b. Check the *Workspace* item in the tree.
    c. Check the following plugins under *Target Platform* are checked:
        - `de.cau.cs.kieler.kgraph.text.ui`
        - `de.cau.cs.kieler.klighd.xtext`
        - `org.eclipse.ui.ide.application`
        - `org.eclipse.platform`

    d. Click *Add Required Plug-ins*. Press it twice (just to be sure!).
5. Click *Apply* to save your changes and then *Debug* to start an Eclipse instance to test with.

Test the layouter in your new Eclipse instance:

1. Click *New* > *Project...* > *General* > *Project* and set the project name to something like `Test`.
2. Right-click the new project and click *New* > *Other* > *KGraph* > *Random KGraph*. Enter a meaningful name and click *Finish*.
3. Open the .kgt file. To show up the Diagram vie, select *Window* > *Show View* > Other... > Other > Diagram
4. Open the *Layout* view through *Window* > *Show View* > *Other...* > *Eclipse Diagram Layout* > *Layout*. Move the view somewhere such that you can see the view and the diagram simultaneously.
5. Chose your *Simple Test Layouter* in the *Layout Algorithm* section of the *Layout View*. (If the *Layout View* shows no properties, click the white background in the *Diagram View* once.)

6. You should see something similar to this



Once you're satisfied with your node placement code, it's time to take care of edge routing.

1. Add a new method that will implement the edge routing using the following skeleton code:

```
/**
 * Routes the edges connecting the nodes in the given graph.
 *
 * @param parentNode the graph whose edges to route.
 * @param yStart y coordinate of the start of the edge routing area.
 * @param objectSpacing the object spacing.
 * @return height used for edge routing.
 */
private float routeEdges(final KNode parentNode, final float yStart, final float objectSpacing) {
    // TODO: Implement edge routing

    return 0;
}
```

2. Add a call to `routeEdges(...)` in your `doLayout(...)` method and implement the latter.

Tips

Here's a few tips for implementing the edge routing:

- Each edge shall be drawn with three orthogonal line segments: one vertical segment below the start node, one vertical segment below the target node, and a horizontal segment that connects the two.
- The horizontal segments of two different edges shall not have the same y-coordinate. Two neighboring horizontal segments shall be placed at a distance of objectSpacing.
- See the screenshot at the top of the tutorial for an example.
- Find the edges in a graph by calling `getOutgoingEdges()` or `getIncomingEdges()` on the nodes.
- You can add bend points to edges through the edge's edge layout:

```
KEdgeLayout edgeLayout = edge.getData(KEdgeLayout.class);
KPoint bendPoint = KLayoutDataFactory.eINSTANCE.createKPoint();
edgeLayout.getBendPoints().add(bendPoint);
```

- You will want to clear the list of bend points of each edge layout before adding bend points to it. This will remove all bend points the edge had prior to invoking your layout algorithm.
- Set the values of the points returned by `getSourcePoint()` and `getTargetPoint()` according to the positions where an edge leaves its source node and reaches its target node.
- If you want, you can improve the edge routing code by allowing horizontal segments to share the same y-coordinate if that doesn't make them overlap. Your goal could be to produce an edge routing that uses as little space as possible.
- If that's not enough yet: can you find a way to find an order of the horizontal segments such that as few edge crossings as possible are produced?

The drawing framework does something *intelligent*. Or at least it thinks it is intelligent. When the source point or target point of an edge does not touch the boundary of a node, it moves them such that they touch the boundary. This can be confusing when you calculate and specify positions in the code that are not reflected in the diagram you see.

Once you're done implementing the edge routing code, test it by running your debug configuration again, as before.