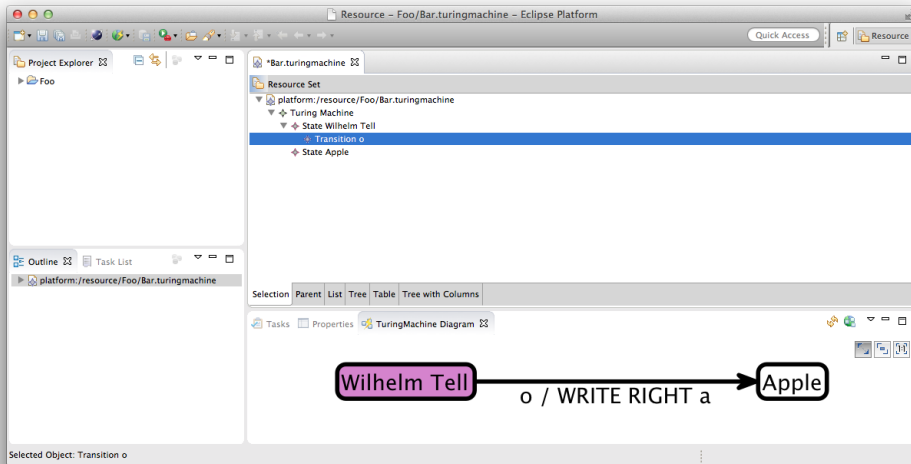


Lightweight Diagrams (KLighD)

KIELER Lightweight Diagrams (KLighD) allows you to develop visualizations for data structures quite easily. In this tutorial, we will install Eclipse and all the necessary components to develop KLighD visualizations before moving on to actually develop a visualization of a state machine:



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.

Suggested Reading

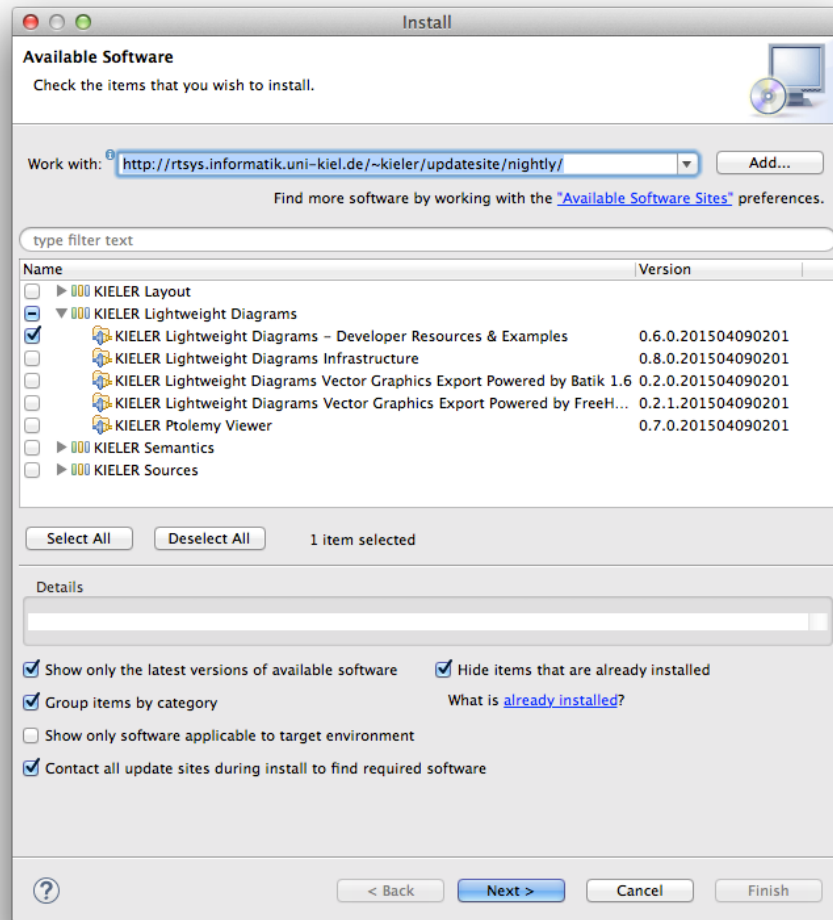
Before moving on with this tutorial, it is a good idea to familiarize yourself with the KGraph data structure, which we have [a tutorial about \(sort of\)](#).

Required Software

For this tutorial, we need you to have Eclipse installed:

1. Install [Eclipse](#). For what we do, we recommend installing the Eclipse **Modeling Tools** (not Classic, not Java Developers).
2. Install the *KIELER KGraph Editing and Visualization* for this tutorial.
 - a. Go to *Help > Install New Software*
 - b. Use either of our nightly update site at <http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/nightly/> or one of the [releases](#).

c. Check *KIELER Lightweight Diagrams - Developer Resources & Examples* and finish the wizard.



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

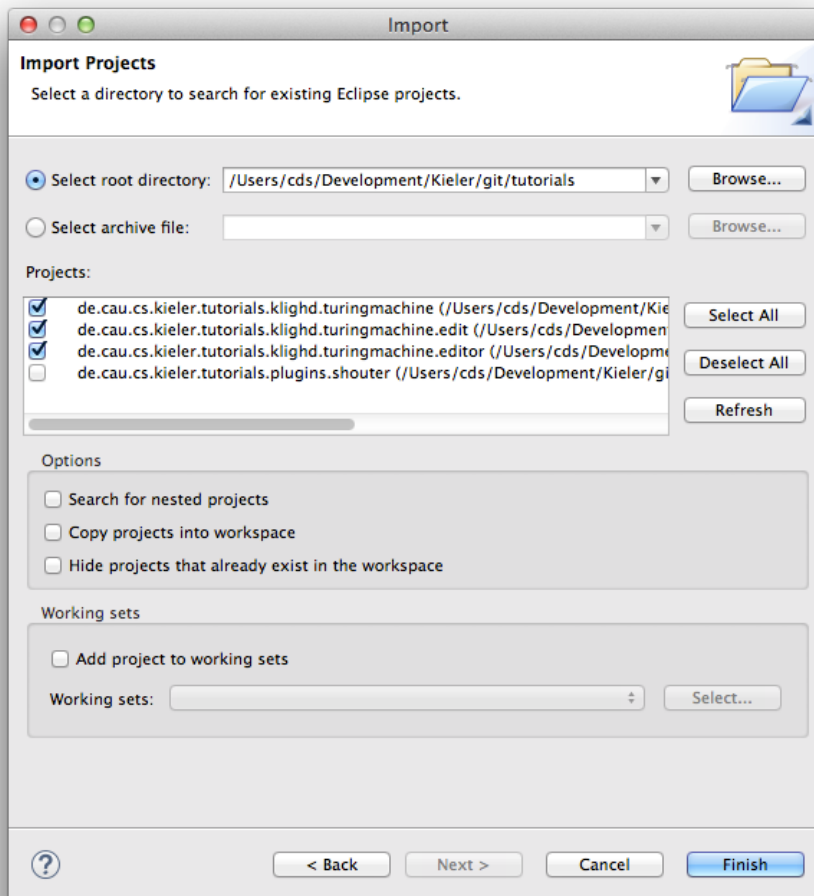
by something called *The Internet*. It should go without saying that if all else fails, [Google](#) often turns up great tutorials or solutions to problems you may run into. And if it doesn't, your advisers will be happy to help.

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.

Setting Up Your Workspace

Once you have started your (possibly brand new) Eclipse installation, your workspace is completely empty. For this tutorial, however, we need a bit of sample code to work with:

1. [Download](#) the zip file with all our prepared tutorial plugins from our Stash. Unzip the file.
2. Open the context menu within the *Package-Explorer* (on the very left, right-click the empty space).
3. Select *Import*. Then chose *General > Existing Projects into Workspace*.
4. Browse to the location where you unzipped the downloaded plug-ins. Check the check box in front of all the `de.cau.cs.kieler.tutorials.klighd.*` projects and press *Finish*.



The imported projects contain a meta model for [Turing machines](#). (You may notice that this tutorial thus also slips in a perfect opportunity to brush up on your knowledge of Turing machines. Consider it a public service and thank us later.) It does not model the tape or the head, only its states and transitions. It is these Turing machines that we will develop a visualization for over the course of this tutorial.

Fixing Problems

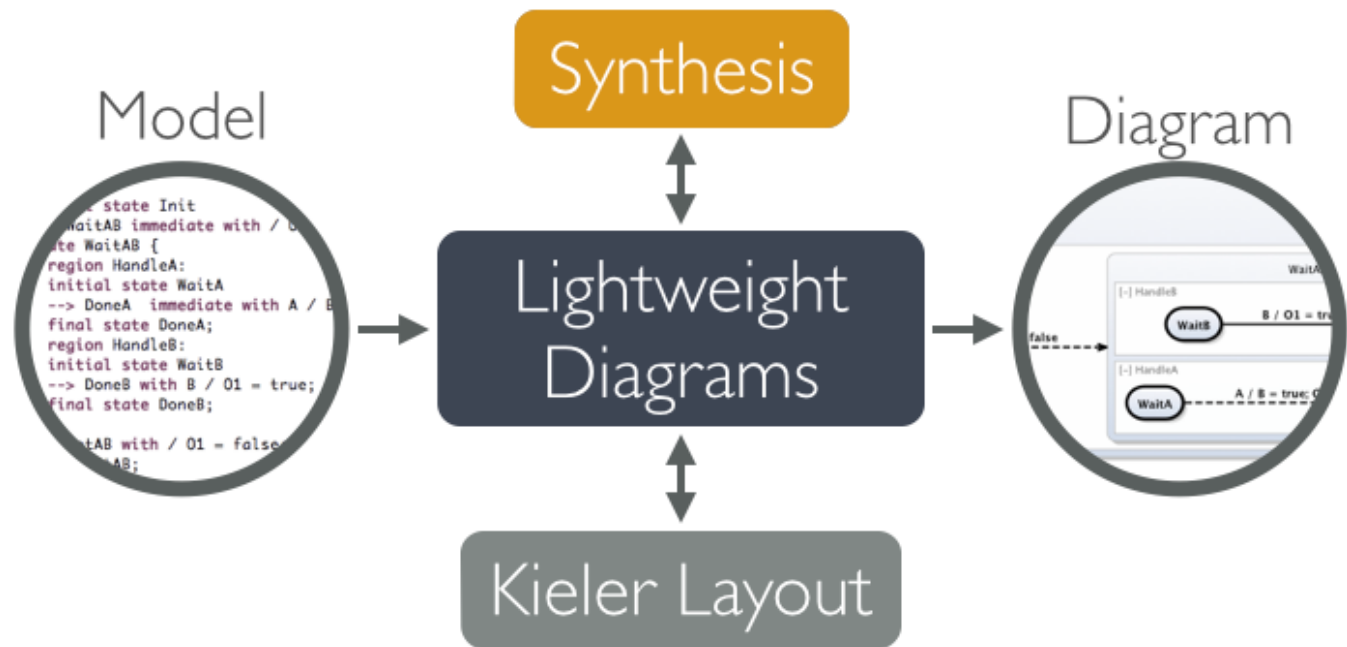
Should your projects be marked with a big red exclamation mark, there might be build path problems. To fix them, complete these steps with all such projects:

1. Right-click a project and select *Properties*.
2. In the dialog that opens up, navigate to the *Java Compiler* category.
3. Chances are that project-specific settings are enabled here. Disable them and click *OK*.

If that wasn't the source of the problem, ask your advisor.

Creating a Visualization

Now that we have the model of what we want to visualize, it's time to program the actual visualization. But first, let's take a minute to think about what we need to do here. Take a look at the following diagram:

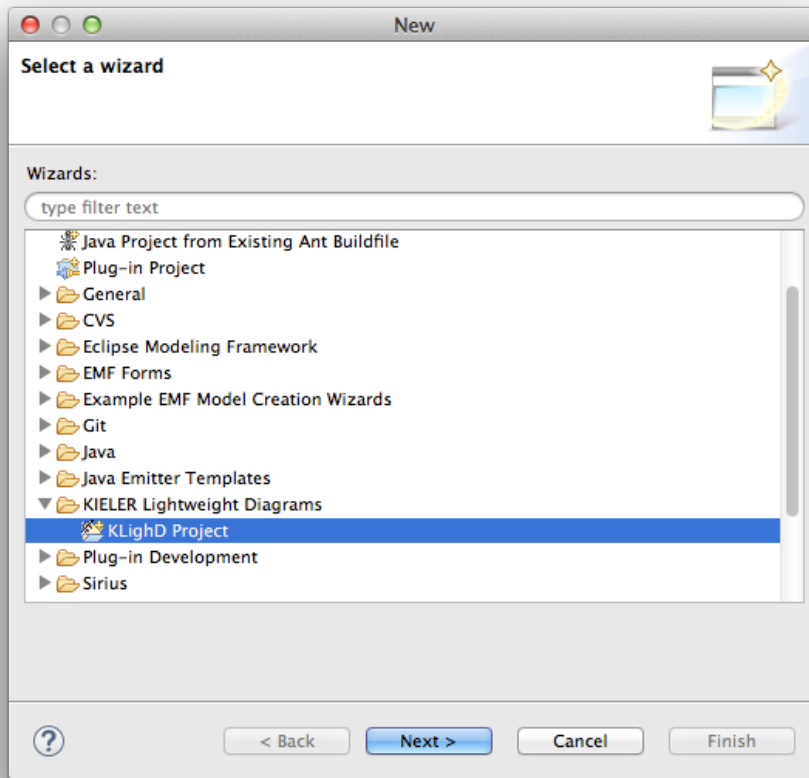


In our visualization, what we have is an instance of a Turing machine model, and what we want to end up with is a proper diagram that visualizes it. Of course, Lightweight Diagrams has not the slightest clue about what we want our diagrams to look like. That is where the *synthesis*, highlighted above, comes into play. The synthesis is what transforms an instance of our source model into a KGraph that Lightweight Diagrams knows how to render. (Along the way, KIELER Layout is invoked by KLighD to compute positions for all diagram elements.) However, the KGraph itself does not suffice to render the diagram: it only specifies its structure, but not its appearance. To specify the latter, we also need to augment our KGraph by KRendering information. See [its documentation](#) to learn more about what it can do.

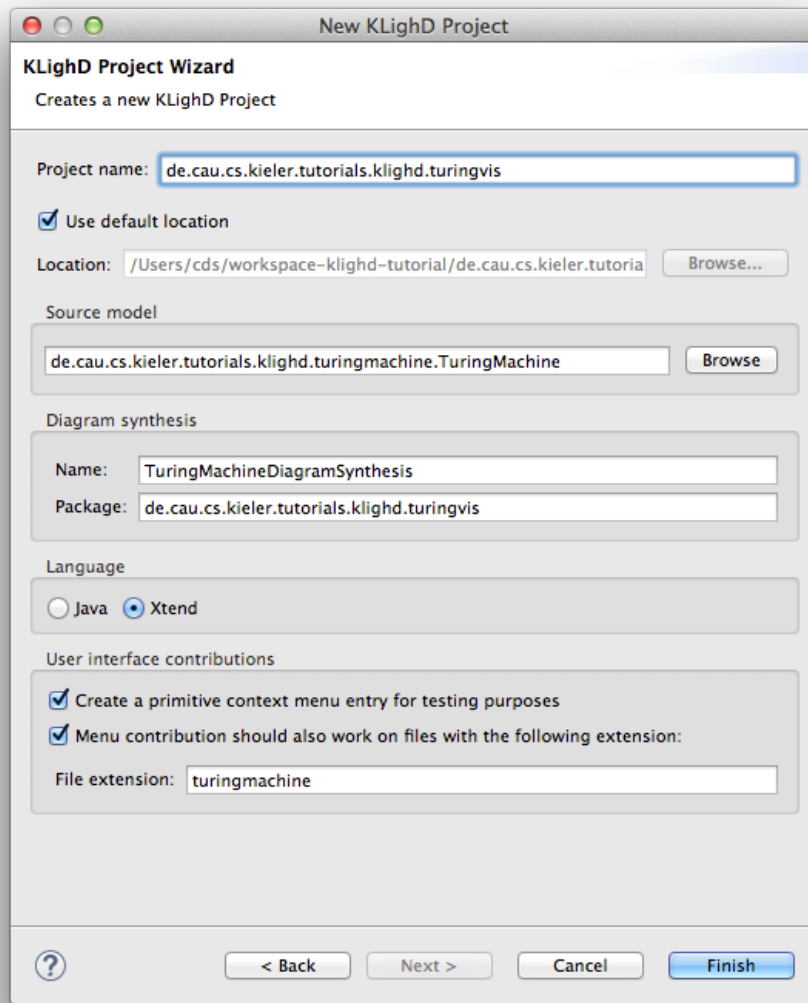
Right, let's dive right in and develop our synthesis.

Adding a Visualization Project

1. Right-click in the *Package Explorer* and select *New -> Other*. From the list, select *KIELER Lightweight Diagrams -> KLighD Project*.



2. Give your new project a proper name and select the base element of the model you want to visualize. Here's what you should enter here before clicking *Finish*:



Note that we ask the wizard to create a menu item for files with the `.turingmachine` extension. Those will be the files we save our Turing machines in.

If you have followed these steps, the wizard will have created a new project. Let's take a minute to examine a few of the files in that project.

There are two source files in the `de.cau.cs.kieler.tutorials.klighd.turingvis` package. The first, `OpenDiagramHandler`, implements the logic required to open a KLighD view on a file or model element. The second, `TuringMachineDiagramSynthesis`, implements the skeleton of the synthesis we will be writing.

So now we have these two files lying around, which is nice. But KLighD doesn't know that it's supposed to invoke the synthesis to turn Turing machines into diagrams, and Eclipse doesn't know that it's supposed to display commands in context menus. That's what the `plugin.xml` file in the project's root directory is for. In there, the synthesis is registered with KLighD, and the menu command is registered with Eclipse.

Programming the Visualization

We will now start filling in the synthesis skeleton, so open the `TuringMachineDiagramSynthesis` class:

```
/* Package and import statements... */

class TuringMachineDiagramSynthesis extends AbstractDiagramSynthesis<TuringMachine> {

    @Inject extension KNodeExtensions
    @Inject extension KEdgeExtensions
    @Inject extension KPortExtensions
    @Inject extension KLabelExtensions
    @Inject extension KRenderingExtensions
    @Inject extension KContainerRenderingExtensions
```

```

@Inject extension KPolylineExtensions
@Inject extension KColorExtensions
extension KRenderingFactory = KRenderingFactory.eINSTANCE

override KNode transform(TuringMachine model) {
    val root = model.createNode().associateWith(model);

    // Your dsl element <-> diagram figure mapping goes here!!

    return root;
}
}

```

Note that this is not Java, but [Xtend](#) code. Xtend is a language that compiles to Java and has a bunch of nifty little features that make writing a synthesis easier. The main entry point of our synthesis is the `transform` method. As you can see, the method gets a `TuringMachine` instance and returns a `KNode`. That `KNode` will contain our actual diagram.

Visualizing States

Let's start by adding nodes for all states in the Turing machine.

1. Add a new method to your synthesis that transforms a `State` into a corresponding `KNode`:

```

private def KNode transform(State state) {
    val stateNode = state.createNode().associateWith(state);

    return stateNode;
}

```

2. While this method does indeed create a node for the state passed to it, `KLighD` wouldn't know how to render it yet. Let's draw the node as a rounded rectangle by adding the following line before the `return` statement:

```

stateNode.addRoundedRectangle(4, 4, 2);

```

3. The only thing missing now is a label with the state's name:

```

stateNode.addInsideCenteredNodeLabel(state.name,
    KlighdConstants.DEFAULT_FONT_SIZE,
    KlighdConstants.DEFAULT_FONT_NAME);
stateNode.addLayoutParam(
    CoreOptions.NODE_SIZE_CONSTRAINTS,
    EnumSet.of(SizeConstraint.MINIMUM_SIZE, SizeConstraint.NODE_LABELS));

```

4. Now that we know how to transform states, we have to call our new method from the main transformation method. Replace the comment in its body with the following line of code:

```

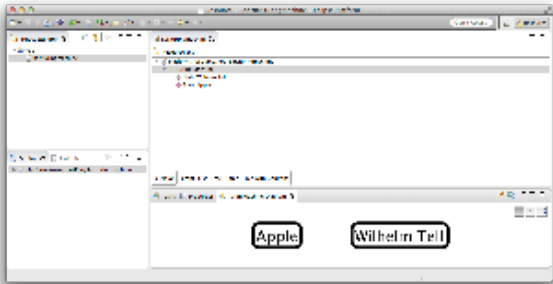
model.states.forEach[ s | root.children += transform(s) ]

```

Let's see if our visualization works. Start your program (if you don't know how to do that, check out our [Eclipse Plug-ins and Extension Points](#) tutorial) and follow these steps:

1. Right-click the *Package Explorer* and select *New -> Project*.
2. In the dialogue, select *General -> Project* and click *Next*.
3. Give the project a meaningful name (*Foo* is a classic) and click *Finish*.
4. Right-click your new project in the *Package Explorer* and select *New -> Other*.
5. In the dialogue, select *Example EMF Model Creation Wizards -> Turingmachine Model* and click *Next*.
6. Give your Turing machine file a meaningful name (*Bar.turingmachine* is a classic) and click next.
7. As the *Model Object*, select *Turing Machine*, which is the root element of our model. Click *Finish*.
8. In the editor that opens, expand the first row. Make sure the *Properties* view is open by right-clicking in the editor and selecting *Show Properties View*.
9. Right-click *Turing Machine* and select *New Child -> State*. Give the state a name through the properties view, such as *Wilhelm Tell*. Also mark it as the initial state of the Turing machine.
10. Add another, non-initial state named *Apple*.
11. Save the file.

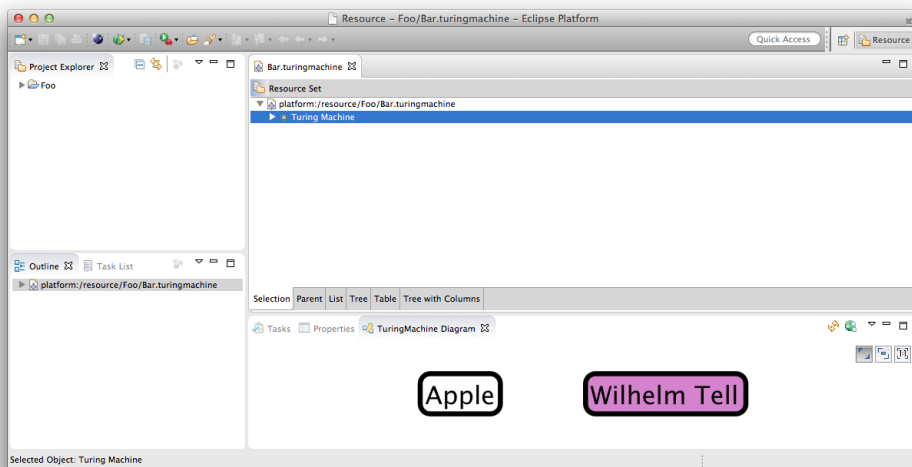
To test your visualization, you can right-click either the file in the *Package Explorer* or the *Turing Machine* row in the editor and select *Open TuringMachine diagram*. A KLighD view should open and display something like this:



That's quite nice already, but we have no way to determine which of these states is the default. Let's add a hideously ugly background colour to initial states by changing the line that adds a rounded rectangle:

```
stateNode.addRoundedRectangle(4, 4, 2) => [ rect |  
  if (state.initial) {  
    rect.setBackgroundColor(210, 130, 210);  
  }  
];
```

The result should look something like this (I told you it was going to be hideous):



Visualizing Transitions

Now that we have our states, it's time to visualize transitions as well. It would be nice to just transform transitions leaving a given state while we're transforming that state. Let's do that:

1. Add the following code right before the `return` statement in the state transformation method:

```
stateNode.outgoingEdges.addAll(state.outgoingTransitions.map[ t | transform(t) ])
```

2. Of course, we will have to add a `transform` method for transitions now:

```
private def KEdge transform(Transition trans) {  
  val transEdge = trans.createEdge().associateWith(trans)  
  
  return transEdge;  
}
```

3. Again, we need to tell KLighD how to render the label:


```
transEdge.addPolyline(2).addHeadArrowDecorator();
```

4. And the edge also needs a label that describes the transition:

```
// Add a label to the edge
val label = KGraphUtil.createInitializedLabel(transEdge);
val labelText = trans.trigger + " / "
                + trans.action + " "
                + trans.direction + " "
                + trans.newChar;
label.configureCenterEdgeLabel(labelText,
    KlighdConstants.DEFAULT_FONT_SIZE,
    KlighdConstants.DEFAULT_FONT_NAME);
```

5. What is missing now is to set the edge target to the node that represents the target state. There's a problem here, though: we don't even know if the target state already had a node created for it. Thus, we need to make sure it has:

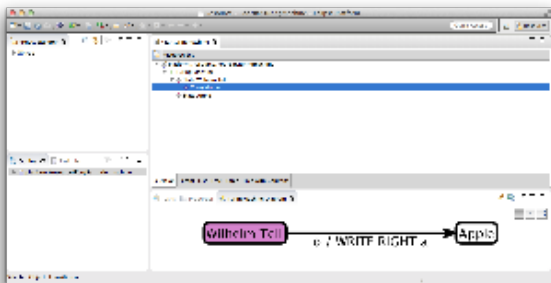
```
transEdge.target = transform(trans.targetState);
```

6. This, however, results in a different problem: if the target state already had a node created for it, we now create another node. What we would need the state transformation method to do would be to first check if a node has already been created for a given state, and, if so, return that instead of creating a new one. It turns out that Xtend already supports this pattern. Change the method's declaration and its first line to the following:

```
private def create stateNode : state.createNode() transform(State state) {
    stateNode.associateWith(state);
```

Also, remove the return statement. Xtend now checks if we already created a node for the given state and, if not, execute the code in the method.

Start your program again, open your Turing machine file and add an outgoing transition to your *Wilhelm Tell* state. Change the properties 'New Char' and 'Trigger' to arbitrary characters (but not 0, since that translates to the ASCII char null and will kill your Label). Set the *Apple* state as the transition's target state. Save the model and fire up a KLighD view. It should look something like this:



Going Further

This concludes our little KLighD tutorial, but of course there's a lot you haven't seen yet. To go further, take a look at the [KLighD pages](#) in our KIELER Confluence space and the examples described there. Also, try solving the assignment below, if you like.

Assignment

In text files, lists can be written like this:

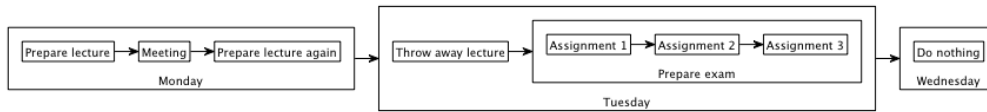
```
* Monday
* Prepare lecture
* Meeting
* Prepare lecture again
* Tuesday
* Throw away lecture
* Prepare exam
* Assignment 1
```

```

    * Assignment 2
    * Assignment 3
* Wednesday
* Do nothing

```

Create a new project that provides a menu item that fires up a visualization for this kind of list format. The visualization should look something like this:



1. Create a new KLighD Project that visualizes `String` instances. Let the wizard create a context menu item for files with the extension `textlist`.
2. The generated command handler will try to let the Eclipse Modeling Framework load the file. Replace this code with code that simply opens the file and reads its content into a `String`.
3. Write a synthesis that turns a `String` into a `KGraph`. You can use the synthesis from our tutorial as a template.

When parsing the string, you can simply assume that each level of the list gets indented by exactly two spaces, and that the text of each list item starts after an asterisk followed by a space. No need to build fancy error checking into your solution.