Cytoscape Developer's Tutorial

Samad Lotia

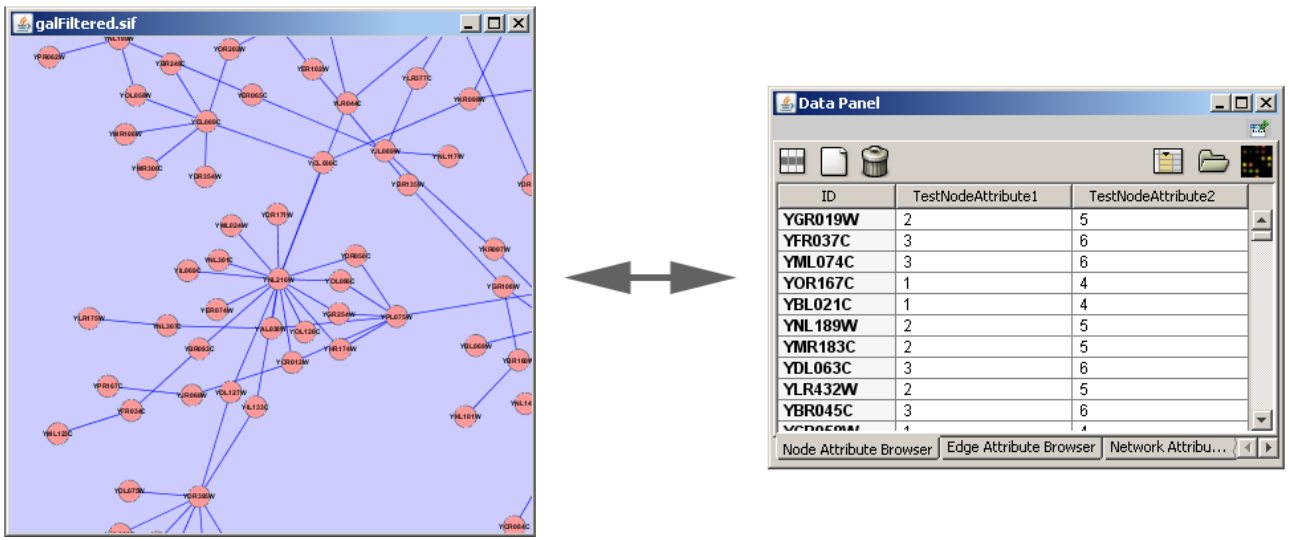
July 18, 2010

Why a developer's tutorial?

- Audience: people who want to learn how to program with Cytoscape
- Assumption: some familiarity with Java and programming in general. You should know:
  - What are classes, interfaces, methods, fields, et cætera
  - How to use the Java compiler and Apache Ant build files
- Provide a high level overview of Cytoscape's API
- Describe specifics on:
  - the network,
  - attributes,
  - the view,
  - programming the VizMapper,
  - writing tasks,
  - using tunables,
  - & writing a plugin
- *Feel free to ask questions during this presentation*

Networks and attributes

- Networks manipulated through `cytoscape.CyNetwork` interface
- Attributes manipulated through `cytoscape.CyAttributes` interface
- Networks and attributes are *decoupled*—one can be loaded without the other
- Networks and attributes are only associated if the attribute's key matches the unique identifier of a node, edge, or network

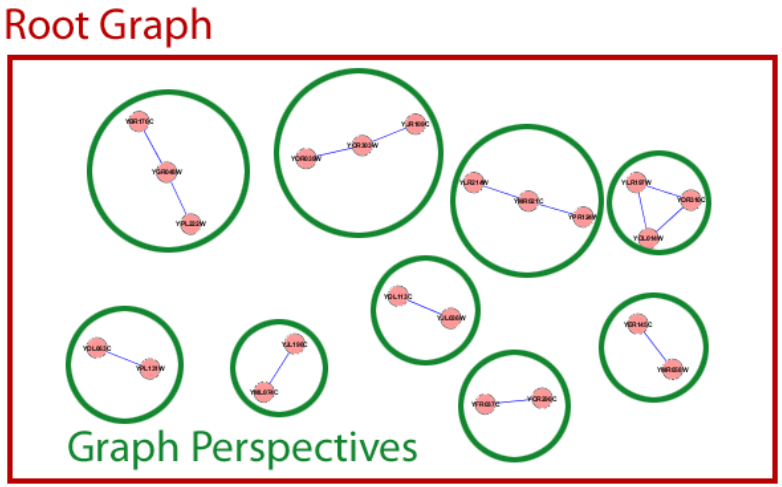


Interfaces versus classes in Java

- When programming with Cytoscape, you'll mostly be talking to Cytoscape thru interfaces, not classes
- An interface is somewhat like a C/C++ header file
- Interfaces allow Cytoscape to control how you obtain an instance
- Ensure that your code isn't aware of what goes on behind the scenes in Cytoscapeland
- × `CyNetwork myNetwork = new CyNetwork("My Network");`
- √ `CyNetwork myNetwork = Cytoscape.createNetwork("My Network");`
- Clear-cut separation between your code and Cytoscape's—you can't manœuvre into Cytoscape's code, and Cytoscape can't manœuvre into yours

Root graphs and graph perspectives

- All networks loaded in Cytoscape are stored in one super network called the *root graph*
  - Represented as `cytoscape.giny.CytoscapeRootGraph`
- A network is merely a subset of the root graph called a *graph perspective*
  - Represented as `CyNetwork`, which extends `giny.model.GraphPerspective`
- Multiple networks can have the same nodes and edges, because "networks" are just perspectives of the root graph



Cytoscape API preliminary

- `cytoscape.Cytoscape` is generally the starting point
- Singleton class—the interesting stuff are public static methods
- Lets you create nodes, edges, and networks and access them

Creating nodes, edges, and networks

- `Cytoscape.createNetwork(title)`—create an empty network
- `Cytoscape.createNetwork(nodes, edges, title)`—create a network from nodes and edges already created
- `Cytoscape.getCyNode(name, true)`—creates a node if a node doesn't already exist with that name. The node won't be in any network unless explicitly added to the network
- `CyNetwork.addNode(node)`—add a node made from `getCyNode`
- `Cytoscape.getCyEdge(source, name, target, interactionType, true)`—creates an edge if it doesn't already exist between source and target
- `CyNetwork.addEdge(edge)`—add an edge made from `getCyEdge`

Identifiers in Cytoscape

- IDs of a node, edge, or network never change even if the title or label changes
- All nodes, edges, and networks of the root graph have two types of IDs:
  - An integer ID
    - Used to query the topology of networks
    - Unique to the root graph
    - Accessable thru `GraphPerspective`, `Node`, and `Edge` interfaces
  - A String ID
    - Used for matching attributes to nodes, edges, and networks
    - Typically just a string version of the integer
    - Accessable thru `CyNetwork`, `CyNode`, and `CyEdge` interfaces

Accessing IDs of nodes, edges, and networks

- `CyNetwork.getIdentifier()`, `CyNode.getIdentifier()`, and `CyEdge.getIdentifier()`—retrieves String ID (useful for dealing with attributes)
- `CyNode.getRootGraphIndex()` and `CyEdge.getRootGraphIndex()`—retrieves integer ID (useful for querying the network topology)

Accessing existing nodes, edges, and networks

- `Cytoscape.getCurrentNetwork()`—get the network that has the current focus in the Control Panel
- `Cytoscape.getNetwork(title)`—retrieve a network based on its title
- `CyNetwork.nodesList()`—list of `giny.model.Nodes` that can be cast to `cytoscape.CyNode` & `CyNetwork.nodesIterator()`
- `CyNetwork.edgesList()`—list of `giny.model.Edges` that can be cast to `cytoscape.CyEdge` & `CyNetwork.edgesIterator()`
- `CyNetwork.getAdjacentEdgeIndicesArray(nodeIndex, true, true, true)`—list of integers that are integer IDs of edges

Controlling what's selected

- `CyNetwork.setSelectedEdgeState(edge, boolean)`
- `CyNetwork.setSelectedEdgeState(list of edges, boolean)`
- `CyNetwork.setSelectedNodeState(node, boolean)`
- `CyNetwork.setSelectedNodeState(list of nodes, boolean)`
- `CyNetwork.getSelectedEdges()`
- `CyNetwork.getSelectedNodes()`
- `CyNetwork.isSelected(edge)`
- `CyNetwork.isSelected(node)`

Network views

- Networks merely store topology information. They do things like:
  - create or remove nodes and edges
  - lets you know the connectivity of nodes
  - change the selection of nodes or edges (seems kind of counter-intuitive)
- *Network views* let you control how a network looks, like:
  - Apply a visual style
  - Apply a layout algorithm
  - (x, y) position of nodes
  - The visibility of nodes
- Each node, edge, and network has an associated view
- A window in the Cytoscape Desktop will not show up when creating a network unless a view is created

Creating a network view

- `Cytoscape.createNetworkView(network)`
- `Cytoscape.createNetworkView(network, title, layout-algorithm)`—a network view can have a different title from a network

Getting a network view if one already exists

- `Cytoscape.getCurrentNetworkView()`
- `Cytoscape.getNetworkView(title)`

Getting node and edge views

- `CyNetworkView.getNodeView(node)`
- `CyNetworkView.getEdgeView(edge)`

Changing the appearance of nodes and edges

- `NodeView.setXPosition(double)`
- `NodeView.setYPosition(double)`
- `CyNetwork.hideNode(node)`
- `CyNetwork.hideNodes(list of nodes)`
- `CyNetwork.hideEdge(edge)`
- `CyNetwork.hideEdges(list of edges)`
- `CyNetworkView.updateView()`—changes you make won't be visible until you call this

Layers of Cytoscape API vis-à-vis networks

	Network Model	Corresponding View
Cytoscape API	cytoscape.jar	
	CyNetwork	CyNetworkView
	CyNode	CyNodeView
	CyEdge	CyEdgeView
Giny API	giny.jar	
	GraphPerspective	GraphView
	Node	NodeView
	Edge	EdgeView
Implementation	fing.jar	ding.jar
	FGraphPerspective	DGraphView
	FNode	DNodeView
	FEdge	DEdgeView

Layers of Cytoscape API vis-à-vis networks

- *Giny* is a third party API that was originally designed to provide layout algorithms for Cytoscape thru yFiles
- Giny does not provide any functionality, i.e., implementations of interfaces
- Giny will go away in Cytoscape 3
- *Cytoscape's API* adds additional methods on top of Giny
- The *implementation* actually provides the functionality in Cytoscape
- Implementation exists in the fing and ding libraries—this is apparent in stack traces

What do all those jars in the Cytoscape directory do?

cytoscape.jar	core application constituting the public API and the Swing application
lib	core libraries that perform basic functionality like network rendering; the core application requires these libraries
plugins	much of Cytoscape's built-in functionality is pulled off into individual plugins like layout algorithms and the selection filter; the core doesn't require these plugins
.cytoscape/2.7/plugins	user–installed plugins

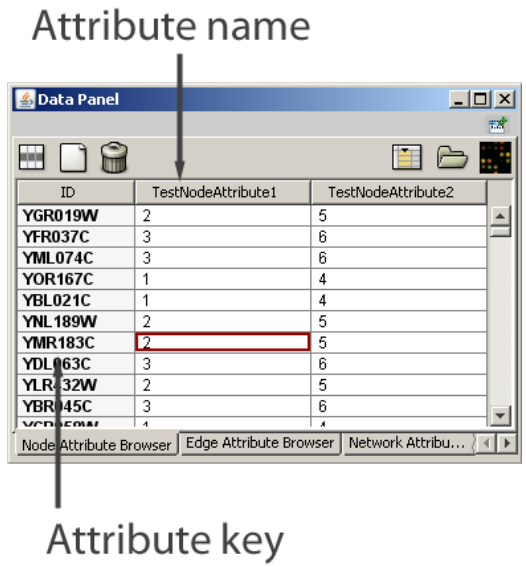
- You typically just compile against `cytoscape.jar`, rarely you should compile against libraries in the `lib` directory

Attributes

- As stated before: attributes are only tenuously associated with network objects by matching the ID of a node, edge, or network with the attribute's key
- No guarantees made that an attribute is mapped to a node, edge, or network
- Attributes have types: booleans, strings, numbers, lists, maps/dictionaries
- One attribute must have only one type; in other words, all attributes values under a single attribute must have the same type

Attributes

- In order to access an attribute value, you must specify:
  - the attribute key (which would be the string ID)
  - the attribute's type
  - the name of the attribute (the column name)
- Cytoscape can tell you the type of an attribute



Attributes—important caveat

- If there's a node, edge, or network that has the same ID as another, it will have the same attributes!
- Direct implication of the decoupled quality of attributes and of the root graph
- This affects the visual style—two nodes with same ID will have same visual style even in different networks

Obtaining attributes

- Networks, nodes, and edges each have their own world of attributes
  - `Cytoscape.getNetworkAttributes()`
  - `Cytoscape.getNodeAttributes()`
  - `Cytoscape.getEdgeAttributes()`

Retrieving attribute values

- `CyAttributes.hasAttribute(ID, attribute-name)`
- Separate methods for retrieving attribute values for each attribute type:
  - `CyAttributes.getBooleanAttribute(ID, attribute-name)`
  - `CyAttributes.getDoubleAttribute(ID, attribute-name)`
  - `CyAttributes.getIntegerAttribute(ID, attribute-name)`
  - `CyAttributes.getListAttribute(ID, attribute-name)`
  - `CyAttributes.getMapAttribute(ID, attribute-name)`
- Assuming the wrong attribute type causes the above methods to return null
- Check the type of the method by calling: `CyAttributes.getType(attribute-name)`

Assigning attribute values

- `CyAttributes.setAttribute(ID, attribute-name, value)`—only works for strings, integers, doubles, booleans
- `CyAttributes.setListAttribute(ID, attribute-name, list)`
- `CyAttributes.setMapAttribute(ID, attribute-name, map)`

Looping thru attribute values

- To loop thru each attribute name: `CyAttributes.getAttributeNames()`
- To loop thru attribute values of each node in a network:
  1. Obtain the node attributes: `CyAttributes nodeAttrs = Cytoscape.getNodeAttributes();`
  2. Obtain list of Nodes: `List nodes = myCyNetwork.nodesList();`
  3. Cast each element in the list to a CyNode: `CyNode cyNode = (CyNode) nodes.next();`
  4. Obtain string ID of node: `String id = cyNode.getIdentifier();`
  5. Get the node attribute: `cyAttrs.getXAttribute(id, "myAttribute")`

Introducing the VizMapper

- The VizMapper takes an attribute and correspondingly changes the appearance of a node or an edge ("visual property")
- It can affect things like
  - node shape
  - node size
  - node & edge color
  - node & edge label
  - node & edge tooltip
- A "visual style" is a bag of visual properties

A who's who of interfaces for the VizMapper

VisualMappingManager manages the currently selected visual style

```
VisualMappingManager manager =
Cytoscape.getVisualMappingManager();
```

Mapping converts an attribute value into something the VizMapper can use

Calculator specifies what a `Mapping` affects—the node shape, edge tooltip, et cætera

AppearanceCalculator manages the appearance of nodes, edges, or the global appearance based on `Calculators`

CalculatorCatalog repository of calculators available to all of Cytoscape

```
CalculatorCatalog catalog =
manager.getCalculatorCatalog();
```

Creating a mapping

- First step in programmatically making a visual style is making a mapping
- Types of mappings:

  Pass Through the result is exactly what the attribute value is

  Discrete individually specify the result for each possible attribute value (equivalent to a `Map`, where keys are attribute values, and values are possible results)

  Continuous provide a mathematical function that computes the result based on the attribute value (example: $f(x) = 2x + 1$), only works on numerical attributes

Creating a pass through mapping

- `PassThroughMapping pm = new PassThroughMapping(default-value, attribute-name);`
- Default value also serves as an indicator to the attribute type of the result

Creating a discrete mapping

```
DiscreteMapping disMapping = new
DiscreteMapping(NodeShape.RECT,
ObjectMapping.NODE_MAPPING);

disMapping.setControllingAttributeName("attr1",
network, false);

disMapping.putMapValue(new Integer(1),
NodeShape.DIAMOND);

disMapping.putMapValue(new Integer(2),
NodeShape.ELLIPSE);

disMapping.putMapValue(new Integer(3),
NodeShape.TRIANGLE);
```

Creating a continuous mapping

```
// Continuous Mapping - set node color
   ContinuousMapping continuousMapping = new ContinuousMapping(Color.WHITE,
                                                               ObjectMapping.NODE_MAPPING);
   continuousMapping.setControllingAttributeName("attr3", network, false);

   Interpolator numToColor = new LinearNumberToColorInterpolator();
   continuousMapping.setInterpolator(numToColor);

   Color underColor = Color.GRAY;
   Color minColor = Color.RED;
   Color midColor = Color.WHITE;
   Color maxColor = Color.GREEN;
   Color overColor = Color.BLUE;

   // Create boundary conditions                    less than,   equals,  greater than
   BoundaryRangeValues bv0 = new BoundaryRangeValues(underColor, minColor, minColor);
   BoundaryRangeValues bv1 = new BoundaryRangeValues(midColor, midColor, midColor);
   BoundaryRangeValues bv2 = new BoundaryRangeValues(maxColor, maxColor, overColor);

   // Set the attribute point values associated with the boundary values
   continuousMapping.addPoint(0.0, bv0);
   continuousMapping.addPoint(1.0, bv1);
   continuousMapping.addPoint(2.0, bv2);

   Calculator nodeColorCalculator = new BasicCalculator("Example Node Color Calc",
                                                        continuousMapping,
                            VisualPropertyType.NODE_FILL_COLOR);
   nodeAppCalc.setCalculator(nodeColorCalculator);
```

Programmatically creating a visual style

1. Create appearance calculators for nodes, edges, and the global appearance

```
NodeAppearanceCalculator nodeAppCalc = new NodeAppearanceCalculator();

EdgeAppearanceCalculator edgeAppCalc = new EdgeAppearanceCalculator();

GlobalAppearanceCalculator globalAppCalc = new GlobalAppearanceCalculator();
```

2. Create the mapping
3. Create a calculator based on the mapping

```
Calculator calc = new BasicCalculator("name", mapping, VisualPropertyType.NODE_LABEL);
```

4. Tell the appropriate appearance calculator about the calculator you made

```
nodeAppCalc.setCalculator(calc);
```

5. Create a visual style based on all three appearance calculators

```
VisualStyle visualStyle = new VisualStyle(name, nodeAppCalc, edgeAppCalc, globalAppCalc);
```

Setting the visual style

1. Obtain a visual style: `VisualStyle vs = catalog.getVisualStyle(vsName);`
2. Assign the visual style to the network view: `networkView.setVisualStyle(vs.getName());`
3. Tell the visual mapping manager about the visual style: `manager.setVisualStyle(vs);`
4. Redraw the network view: `networkView.redrawGraph(true,true);`

Why tasks?

- Allows execution of a piece of code that takes a long time to complete—seconds to hours
- Gets executed in its own thread so the Cytoscape UI is still responsive while code is executing
- Provides visual feedback to the user about the progress of the task in two ways: status messages and a progress bar
- Allows the user to cancel the task
  - It is easy to overlook cancellation, but this is vitally important to providing a good user interface

A who's who of tasks

Task an interface where you'd write an implementation of this (`Task` is analogous to `Runnable`)

TaskMonitor a class given to your `Task` so that your task can communicate its status to the user

- Lets you set the status message, the progress, et cætera

TaskManager you "submit" your task to the `TaskManager`, so it'll execute your task and manage the UI during your task's execution

JTaskConfig a class that lets you specify how you want your task's UI

How to write tasks

1. Caveat: make sure to include `cytoscape-task.jar` (in Cytoscape's `lib` directory) in the classpath to compile
2. Write an implementation of `cytoscape.task.Task`
3. Create a UI configuration of the task: `JTaskConfig config = new JTaskConfig();`
4. Modify `config` to configure the UI; things that can be changed are whether to display stuff like:
   - A cancel button (a good programmer would always provide cancellation for tasks)
   - A status message
   - A time elapsed field
   - An estimated time remaining field (typically only useful if your task is downloading something and the ETA can be accurately guessed)
5. Execute the task: `TaskManager.executeTask(myTask, config);` (this method will only return once the task is finished)

Example task: Calculation of Pi

$$\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdots\right)$$

```
public class PiCalculator implements Task {
 private final int iterations;
 public PiCalculator(int iterations) {
  this.iterations = iterations;
 }

 protected double sum = 4.0;
 private TaskMonitor taskMonitor = null;
 private void cancel = false;

 public void run() {
  boolean negative = true;
  for (int i = 1; (i < iterations) && (!cancel); i++)
  {
   sum += (negative ? -4.0 : 4.0) / (i * 2.0 + 1.0);
   negative = !negative;
   if (taskMonitor != null)
    taskMonitor.setProgress(i * 100.0 / iterations);
  }
 }

 public void halt() {
  cancel = true;
 }

 public void setTaskMonitor(TaskMonitor taskMonitor) {
  this.taskMonitor = taskMonitor;
 }
}
```

Introduction to tunables

- You have an algorithm that takes a series of parameters
- Tunables are an easy way to manage those parameters:
  - UI generation for those parameters without having to roll your own Swing dialog
  - Saving those parameters to disk so the user doesn't have to re-enter the parameters each time Cytoscape is fired up
  - Lets other plugins programmatically modify the parameters
- A *Tunable* represents a single parameter (a single knob to configure) that an algorithm says it needs to be specified
- A *ModuleProperties* represents a collection of `Tunables`

Types of tunables

- The usual suspects:
  - `BOOLEAN`
  - `STRING`
  - `INTEGER`
  - `DOUBLE`
- The unusual suspects:
  - `LIST`
  - `NODEATTRIBUTE & EDGEATTRIBUTE`
  - `GROUP`
- A tunable can have flags assigned to them, like lists can allow multiselection or bounds for numerical tunables

Overview of working with tunables

- Create a `ModuleProperties`: `ModuleProperties props = new ModulePropertiesImpl(property-prefix, module-name)`
- Tell the `ModuleProperties` about all the tunables you want by creating a `Tunable` object and adding it
- Tell the `ModuleProperties` that you want it to obtain the tunables' values from the user: `props.updateValues()`
- Read in all the tunables' values

Tunables example

```
props.add(new Tunable("partition", "Partition graph before layout",
                          Tunable.BOOLEAN, new Boolean(true)));

props.add(new Tunable("selected_only", "Only layout selected nodes",
                          Tunable.BOOLEAN, new Boolean(false)));

props.add(new Tunable("defaultSpringCoefficient", "Default Spring Coefficient",
                          Tunable.DOUBLE, new Double(defaultSpringCoefficient)));

props.updateValues();

Tunables t;
t = props.get("partition");
if ((t != null) && (t.valueChanged() || force))
     setPartition(partition.getValue().toString());

t = layoutProperties.get("selected_only");
  if ((t != null) && (t.valueChanged() || force))
    selectedOnly = ((Boolean) t.getValue()).booleanValue();

t = layoutProperties.get("defaultSpringLength");
if ((t != null) && (t.valueChanged() || force))
  defaultSpringLength = ((Double) t.getValue()).doubleValue();
```

Writing plugins

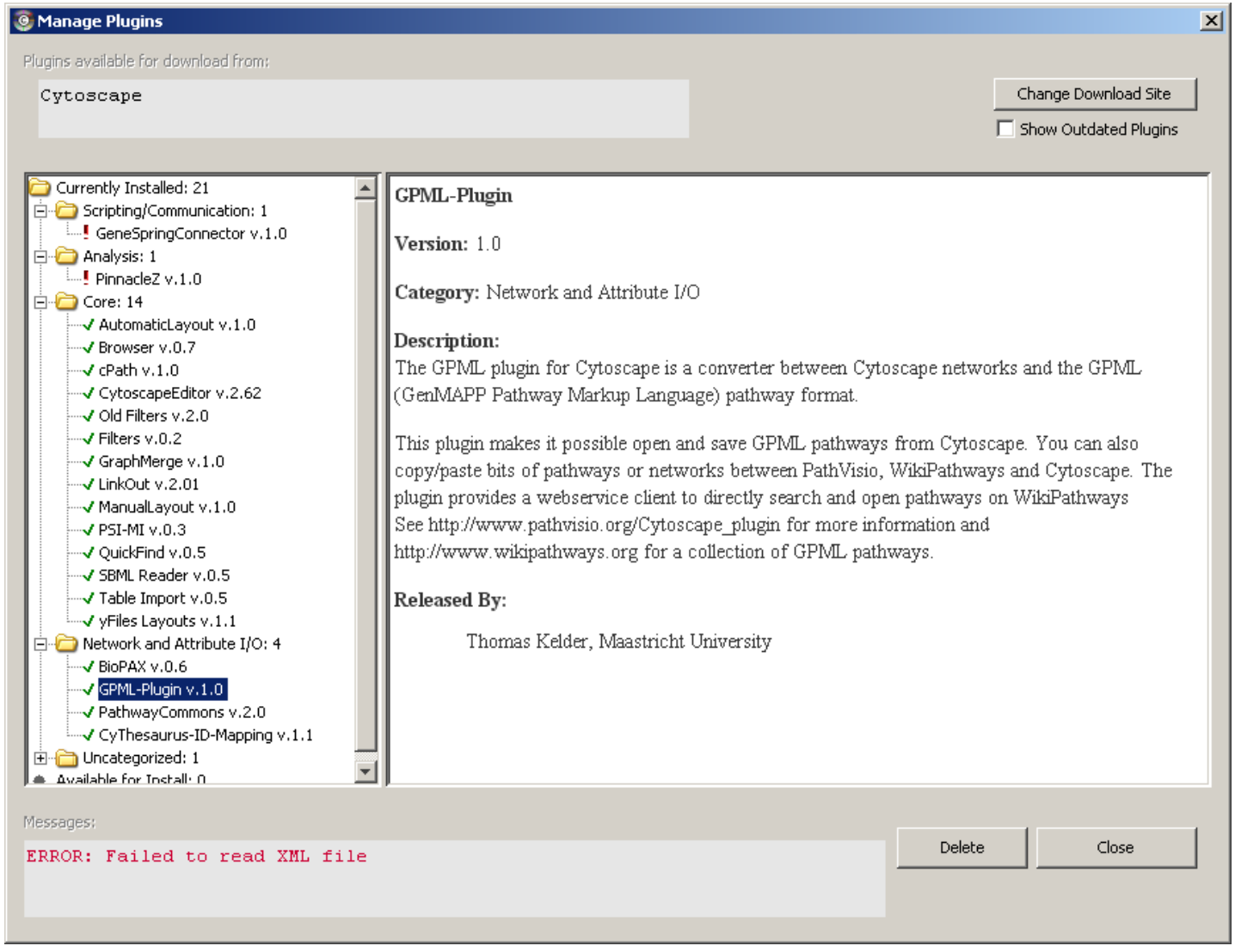
- All plugins are made up of the following pieces:
  1. Your code and a class that extends `CytoscapePlugin`
  2. A `plugin.props` file that gives basic information about the plugin
  3. An Apache Ant `build.xml` file that compiles and packages your code into a jar that can be put into the plugins directory
- Versioning of the plugin must follow this convention otherwise Cytoscape won't recognize the plugin:
  major-version`.`*minor-version* (examples: `1.0`, `4.2`)

Writing a CytoscapePlugin class

- The constructor of your class that extends `CytoscapePlugin` must take no arguments
- Cytoscape will invoke the constructor of this class
- This is the starting invocation of your plugin (analogous to `main`)
- Typically most plugins insert one or more menu items; this is accomplished by:
  1. Create a `JMenuItem` with an action that you want to be invoked when the menu item is selected
  2. Obtain the `JMenu` into which you want to have your menu item:
     `Cytoscape.getDesktop().getCyMenus().getFileMenu()`
     (More menus are available in `CyMenus`)
  3. Add the menu item to the menu: `menu.add(myMenuItem)`

Writing a plugin.props file

```
# can't have whitespace
pluginName=MyPlugin

# can have html tags
pluginDescription=Information about the plugin

pluginVersion=1.0

# list of versions of Cytoscape plugin is compatible with
cytoscapeVersion=2.5,2.6

pluginCategory=Scripting/Communication

# optional properties
projectURL=http://my-lab-site.org/myCytoscapePlugin

pluginAuthorsInstitutions=Sarah and Victor:ISB;Mike, Kei and Peng:UCSD

releaseDate=May 1, 2008
```


Manage Plugins
Plugins available for download from:
Cytoscape
Change Download Site
Show Outdated Plugins
Currently Installed: 21
Scripting/Communication: 1
GeneSpringConnector v.1.0
Analysis: 1
PinnacleZ v.1.0
Core: 14
AutomaticLayout v.1.0
Browser v.0.7
cPath v.1.0
CytoscapeEditor v.2.62
Old Filters v.2.0
Filters v.0.2
GraphMerge v.1.0
LinkOut v.2.01
ManualLayout v.1.0
PSI-MI v.0.3
QuickFind v.0.5
SBML Reader v.0.5
Table Import v.0.5
yFiles Layouts v.1.1
Network and Attribute I/O: 4
BioPAX v.0.6
GPML-Plugin v.1.0
PathwayCommons v.2.0
CyThesaurus-ID-Mapping v.1.1
Uncategorized: 1
Available for Install: 0
GPML-Plugin
Version: 1.0
Category: Network and Attribute I/O
Description:
The GPML plugin for Cytoscape is a converter between Cytoscape networks and the GPML (GenMAPP Pathway Markup Language) pathway format.
This plugin makes it possible open and save GPML pathways from Cytoscape. You can also copy/paste bits of pathways or networks between PathVisio, WikiPathways and Cytoscape. The plugin provides a webservice client to directly search and open pathways on WikiPathways
See http://www.pathvisio.org/Cytoscape_plugin for more information and http://www.wikipathways.org for a collection of GPML pathways.
Released By:
Thomas Kelder, Maastricht University
Messages:
ERROR: Failed to read XML file
Delete
Close

The Apache Ant build.xml file

- Compiling: include `cytoscape.jar` into your classpath, and the jars in the `lib` directory only if you need them
- Creating the jar:
  - Manifest must have `Cytoscape-Plugin` attribute that specifies class path location of your class that extends `CytoscapePlugin` (extremely important)
  - `plugin.props` must be packaged into the same directory as your class that extends `CytoscapePlugin`

```
<target name="jar" description="Create MyPlugin jar file" depends="compile">
    <copy file="${basedir}/plugin.props" todir="{$classes.dir}/my/package" />
    <jar destfile="${build.dir}/MyPlugin.jar">
         <manifest>
         <attribute name="Cytoscape-Plugin"
                    value="my.package.MyPlugin" />
         </manifest>

         <fileset dir="${classes.dir}" />
    </jar>
</target>
```

- Once your jar file is ready, you can copy it to the `plugins` directory to test it out, share it with others, or upload it to the Cytoscape website to make it available in the Plugin Manager

Shout-outs

- Thanks to Scooter Morris for providing his slides, and Gary Bader for the plugin tutorial on the Cytoscape Wiki