CONTENTS

INTRODUCTION

This document provides information for software developers to clarify the source code of Debug flow. After reading this document the software structure can be better understood, design choices will be clearer and in the end software developers can contribute to the project.

When things are not clear an issue can be made on Github.

DEBUG FLOW

Why Debug flow?

In this decade software developers have powerful tools to build software, development environments make work easier and have a perfect integration with toolchains. Furthermore, debuggers will allow developers to find issues. However, that console thing that all programmer use is stupid simple and I'm happy when the console in an IDE supports colors. For the most developers that is enough, but some need better tools.

Some tools like debuggers, custom software applications, application specific log viewers, log files, and logging standards like log4j does the job. But the ultimate freeware solution for easier log viewing is still missing or did not become known.

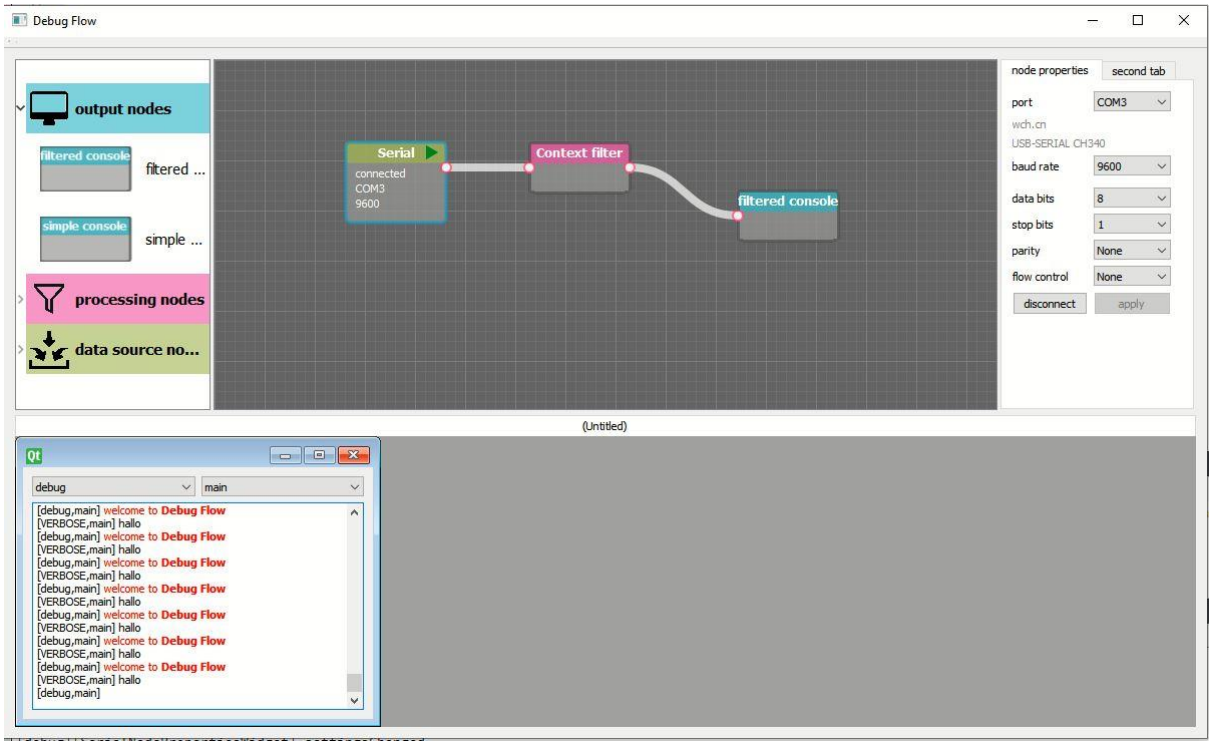
As embedded software developer I've worked on software with high data-rate sensor data, control algorithms and network protocols. These applications are a worst case scenario for logging and debugging, putting realtime sensor values in a control algorithm can result in unpredictable outputs during development, to find out what's going wrong sensor values and algorithm outputs must be logged. This sensor values are read with a high rate, so the logs are difficult to read when the application is running.

These situations also happen in general software development, much developers enables logs when they need them and disables them when they don't need them to clean up their logs. When a bug occurs, they compile the code again with the logs enabled, but unfortunately the bug doesn't appear again after that. Much developers know this phenomenon, much developers add a logging difficulty when asking them, some want data visualization, others just want to store their output, others want to combine logs or they want to process them, they want a user interface and much more.

Debug flow is supposed to be a user-friendly tool for almost all these situations, an ambitious goal.

The application

The possible use cases for this software are unlimited, making a user interface with all needed options will become huge and user unfriendly. Therefore, Debug flow works like flow-based programming where a desired application can be made with dragging and dropping blocks.



With debug flow all kinds of inputs can be used like output from a certain application which is in development, input from a serial port, input from a file or input from an internet connection. Because Debug flow works with nodes(also called blocks) multiple different inputs can be used at the same time.

Putting the data from an input to a processing node useful functions possible, think about filtering on a tag. Adding or removing colors, hiding data, translating error code and much more.

Output nodes present the data to the user like the standard simple console, Debug flow will offer much more advanced outputs like a console with filters, graphs, tables, html view and user interface components. Its also possible to have nodes that send data or nodes that save data to a file.

Using filter nodes data can be splitted into different windows which is useful for multiple tasks in one application.

Advanced functions

Debug flow will implement some advanced functions, one of this function is that the time when data is received at the output is saved, because of that the timestamp can be seen at the output console, with a tooltip for example. With the timestamp its also possible combine data from different input nodes base on time. Or more useful is using a timeline to view which events did occur at a certain time. The timeline can be dragged to a certain timestamp, all output nodes will show a cursor on place where the nearest event did occur, this is useful when comparing data in a graph with data in a log or in another graph.

The timeline feature is a challenging feature which is not implemented yet.

Another challenging feature of Debug flow is that the flow and all settings can be changed on-the-go while everything is still running. Even more challenging, when a setting is changed its possible to backwards update all logs and graphs. So, when a setting is changed all outputs will update their history. Therefore, huge buffers are needed, with will limit the amount of historical data.

Using this feature the problem that a bug occurs but certain needed logs are not enabled is belongs to the past.

This feature is not implemented yet and will challenge the developers of Debug flow.

Philosophy

Debug flow implements difficult functions with need a challenging architecture, in much applications the development method keep-it-stupid-simple or You-aren't-gonna-need-it is used, which is a good advice in some cases. Debug flow is made with an opposite development method. Debug flow is designed seen from user perspective.

In this decade computers are becoming faster, but software is becoming more inefficient due to developments like interpreted programming languages. Debug flow is written in C++ with efficiency as an important requirement. The architecture is made with efficiency in mind.

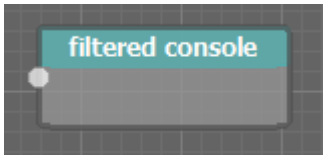
Progress

Debug flow is in the architecture stage, some fundamental features like memento, backward update, timeline and windows management are not fully done yet. After this is done more nodes will be made.

Making more nodes at this stage is not rational because all nodes must be refactored when something change in the architecture of the node.

THE NODE

overview

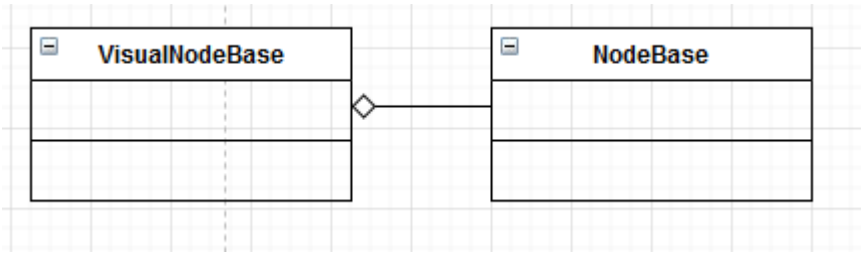


The node is a main object of debug flow, nodes must perform different complex functions in an efficient way, nodes must be visualized on the scene, nodes have different properties and there is the possibility to open and save the scene.

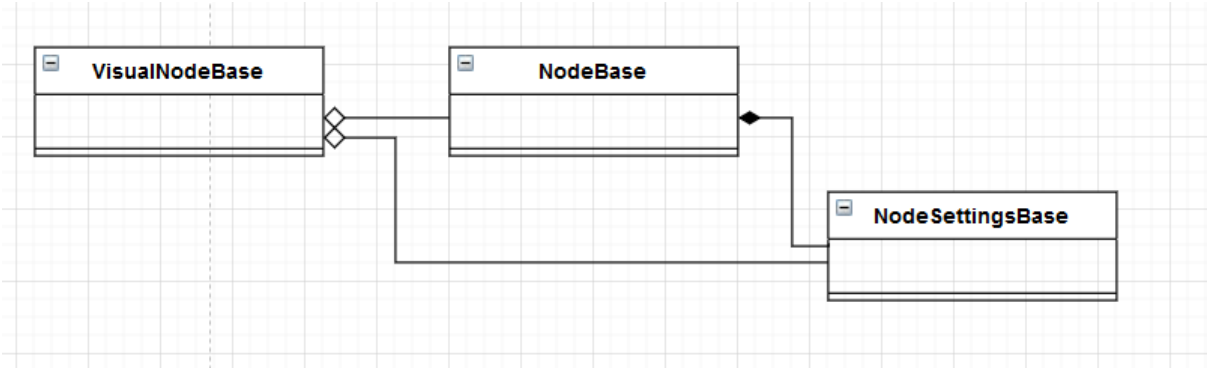
Each of these properties provoke structural challenges because nodes with very different functionalities need to communicate with each other so abstraction is important.

Although a node is one object from the user perspective, a node consists of different classes in the code.

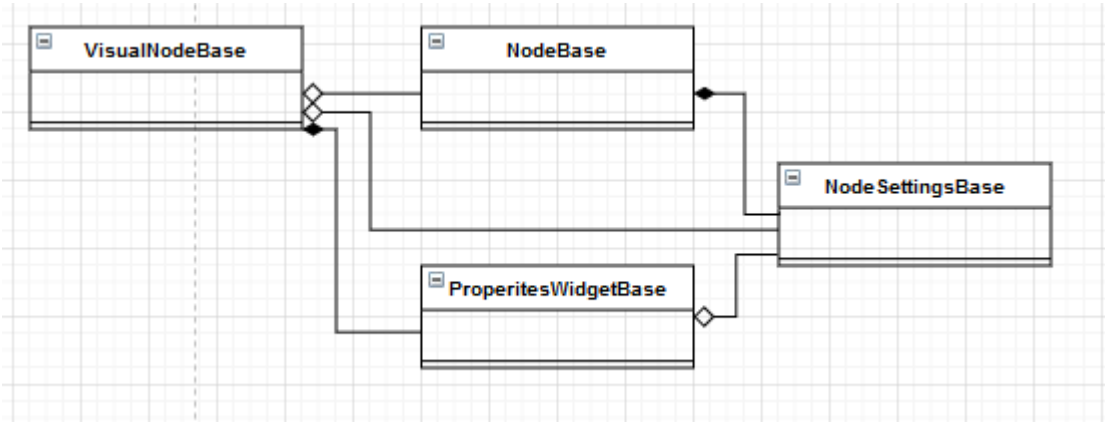
The visual representation and the low-level function are separated. For the visual representation all nodes are derived from the class *VisualNodeBase*, while the functional part of the node will be derived From *NodeBase*.

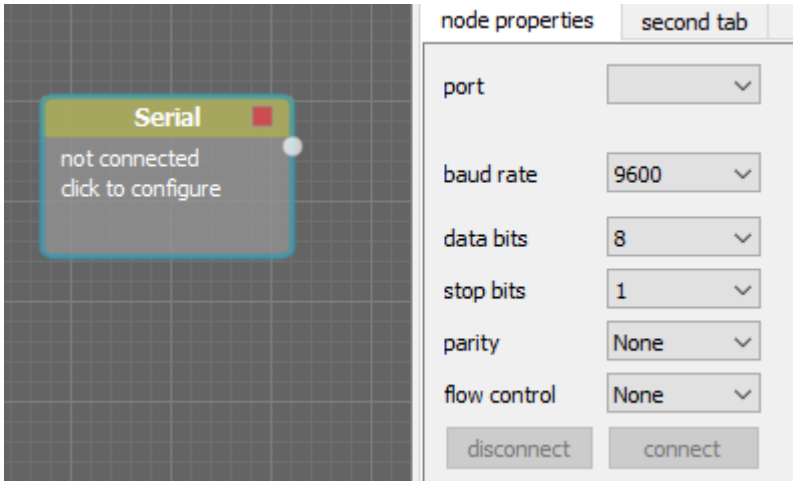


The *VisualNodeBase* have a pointer to *NodeBase*, these two objects have a shared object called *NodeSettingsBase,* all nodes have an object derived from this class.



Furthermore, nodes have settings, this setting can be changed via the user interface, each node have its own settings widget so the user can view and change the settings. The settings widget is a separated class derived from property widget base.

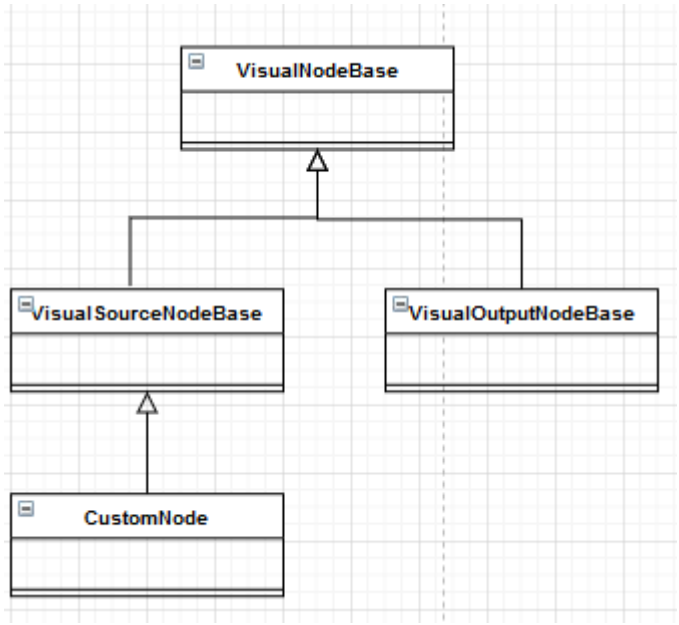




Visual node

DERIVING NODES

The *visualNodeBase* class is abstract, no *VisualNodeBase* instance can be made, instead you should derive an instance from it. As well there are general types like input and output nodes, this nodes are special because they share properties like having a window to visualize data. An input node should be derived from *VisualSourceNodeBase* and an output node should be derived from *VisualOutputNodeBase.*



FIELDS

Some fields of the *VisualBaseNode* will be explained in this paragraph.

The *VisualBaseNode* have some fields that will be filled in by the derived class, for example the name field will be used in the resource list and the shortDiscription will be used as tooltip.

```
//will be filled in by the derived class
QString name = "";
QString shortDiscription = "";
```

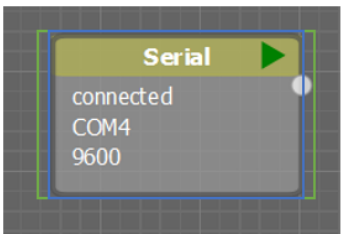
The centerX and centerY variables are the position of the node on the scene, the width and height are the size of the node.

```
//position on the scene
int centerX = graphicsViewOriginX;
int centerY = graphicsViewOriginY;
int width = 125;
int height = 50;
```

FUNCTIONS

The *VisualNodeBase* is derived from QGraphicsItem, therefore it implements the functions BoundingRect() and paint().

The BoundingRect() fuction returns the size of the node, this size includes the connectors that are sticking out. The InnerRect() function returns the size of the rectangle.



The paint function is abstract, it will be done by the derived class.

```
virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem
*option, QWidget *widget) = 0;
```

The derived class can later call functions that draws the standard functions that all nodes have, similarly classes like the **VisualOutputNodeBase** can also have additional paint functions like the functions to draw and running or stop button.

```
void drawConnectors(QPainter *painter, NodeStyleBase *nodeStyle);
void paintBase(QPainter *painter, NodeStyleBase *nodeStyle, QString name);
```

The abstract function active activates the derived class, the *NodeScene* class will activate the node when the node is placed, because while dragging its not desirable that the node already starts doing things. As result a window pops up when an outputnode is placed on the scene.

```
//activate the derived class
virtual void activate() = 0;
```

the following functions are being used to open and close the properties widget of an node, the functions will be called from *PropertyWidgetManager* which will be explained later. When ***loadPropertiesWidget*** is called a new PropertiesWidget will be constructed. When the node is not displayed anymore PropertyWidgetManager will call ***releasePropertiesWidget*** to delete the propertiesWidget from the memory.

```
virtual PropertyWidgetBase* loadPropertiesWidget(QWidget* parent) = 0;
virtual void releasePropertiesWidget() = 0;
```

read more about the node properties widget in the following paragraphs.

Other fields and functions will be explained in the following paragraphs.

UTILS

The *visualNodeBase* use fields from VisualNodeConfig.h

Node styles

Its desirable that different nodes have different style properties, to realize this without losing the possibility to draw almost everything from the **VisualNodeBase** class there are nodes style classes. Via a node style class different style property can be parsed to the **VisualNodeBase**. There is a **NodeStyleBase** with the default style options. **OutputStyle, ProcessingStyle** and **SourceStyle** are derived to provide style differences between node types.

The styles are implemented using inheritance. example:

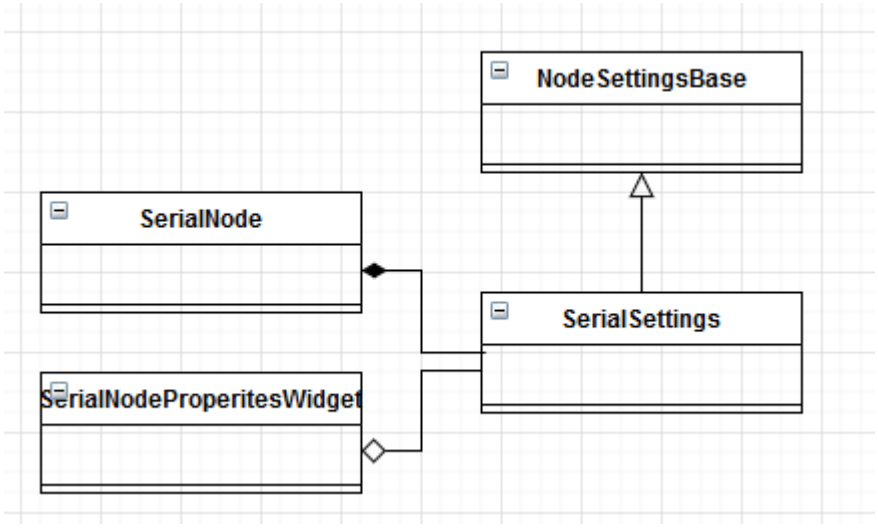
class **VisualFilteredConsole** : public VisualOutputNodeBase, public OutputStyle

Node settings

Nodes have setting, this setting can determine what the node does, the settings are visualized on the Visual node and the settings can be edited in the Properties widget.

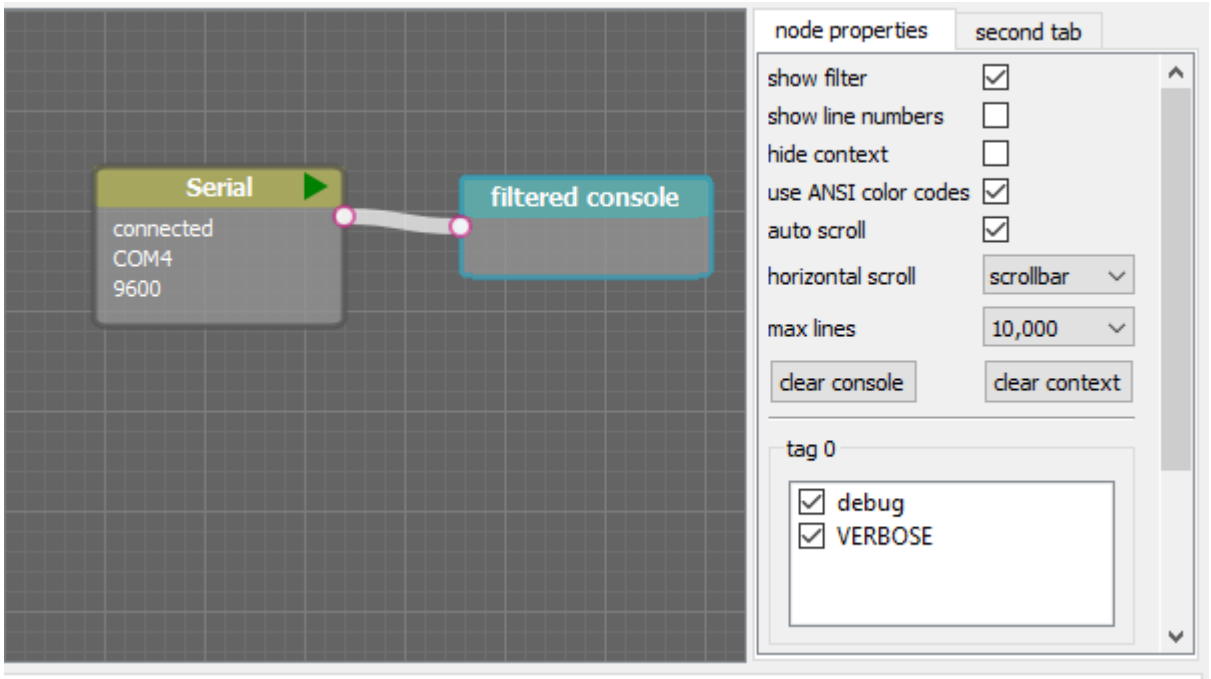
As described in a previous paragraph these functions are split into different classes.

Different nodes have different settings classes, a settings class must be derived from **NodeSettingsBase**. A derived setting class only need to implement a serialization method which will be explained in another paragraph. The rest of the content can be made by the developer.



Node properties

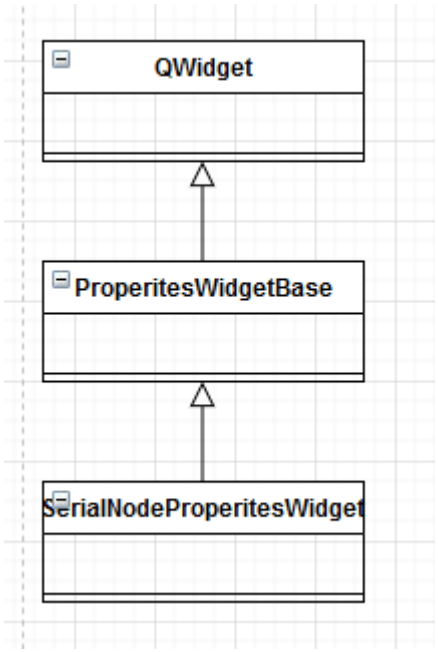
The node properties can be viewed and edited on the properties widget which is placed on the tab widget on the right side of the application.



The properties widget pops up when clicking on a node and it disappears when clicking on something else in the scene.

To save memory a property widget is constructed when it pops up, and it will be deleted from the memory when is disappears.

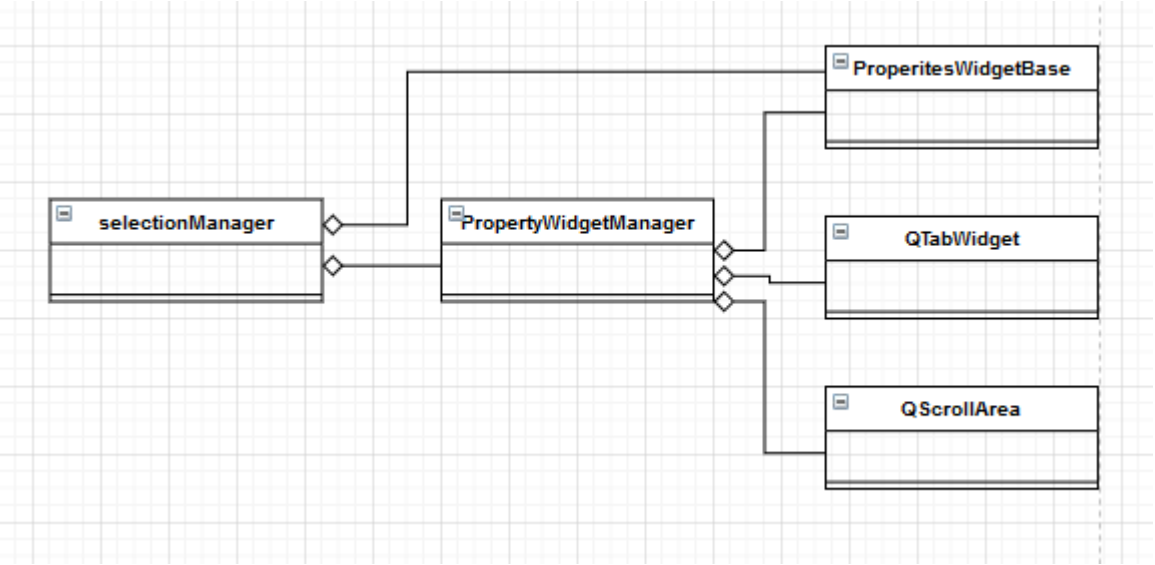
A property widget for a certain node is a class that is derived from **PropertyWidgetBase**, **PropertyWidgetBase** is derived from QWidget. Buttons, checkboxes and other components can be placed on the QWidget.



To manage opening and closing, constructing and destructing the properties widget there is a **PropertyWidgetManager**. The **PropertyWidgetManager** is constructed by **FlowWidget** and its's passed

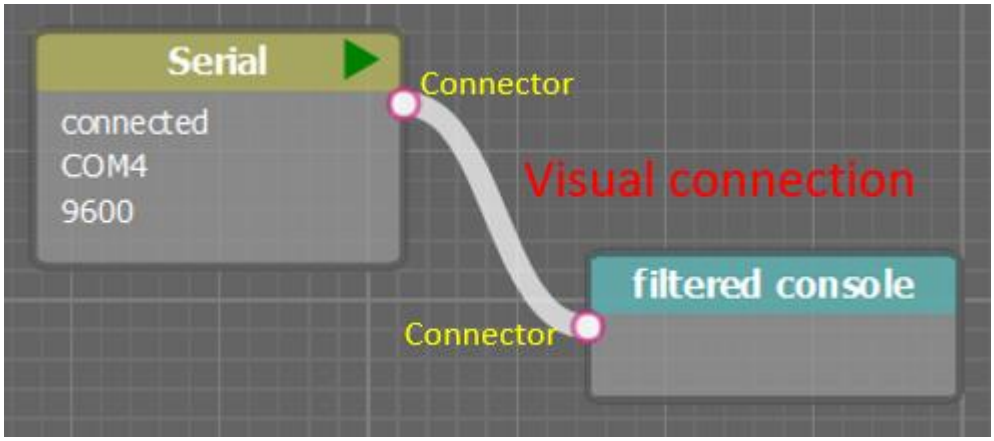
to the **SelectionManager**. The **SelectionManager** will be discusses in a later paragraph, in short, the **SelectionManager** will call functions of the **PropertyWidgetManager** like **notifyNodeSelected**, **notifyMultipleSelected** and **notifyNoneSelected**.

When **notifyNodeSelected** is called the **PropertyWidgetManager** will construct and show the property widget the selected node. When **notifyMultipleSelected** and **notifyNoneSelected** is called the **PropertyWidgetManager** will close and delete the property widget if it was open.



Connections

A visual connection is a visual line between two nodes. It doesn't have a function for transmitting or receiving data, that is done with subscriptions with will be discusses later. The visual connection is a way to visualize that there is a kind of connection between nodes.



Visual nodes have connectors, a connector indicate that a connection can be made from or to that point. When clicking on the connector a connection will be made between the connector and the mouse position. When clicking on a second connector the connection is made between the two nodes and the nodes will make the low-level subscription for data transfer.

A **Connector** is owned by the **VisualNodeBase**, the **VisualNodeBase** have a list with connectors. A

More connectors can be added on the go when settings are changed. The **Connector** has fields like and x and y location, an angle for drawing the line, a name which is a identifier for the serialization and deserialization, a type like INPUT or OUTPUT and a list with connected **VisualConnection** ‘s because a connector can have multiple nodes connected.

A **Connector** is not derived from QGraphicsItem, instead the **VisualNodeBase** paints the connectors. Beside the painting the **VisualNodeBase** will animate when hovering over the connector and will give visual feedback when a connection cannot be made.

A **VisualConnection** is owned by both nodes, is derived from QGraphicsItem and paints itself. The way the **VisualConnection** is painted can be configured with the **ConnectionStyle** class. The line between the nodes can be a straight line or a nice belzier curve. The belzier curve will only be calculated when connected nodes are moved, therefore the Boolean settingsChanged tracks if there is anything changed.

A **VisualConnection** is usually connected between two nodes, except when making the node then one side of the connection follows the mouse, therefore to Booleans connection1Set and connection1Set must be checked before using functions like **getConnector1**.

When to make a low-level subscription the **makeConnection** function of **VisualNodeBase** can be used. But before calling this function make sure to call **requestConnection** and **recursiveCircularDependencyCheck** first. This these functions will prevent making a connection between two inputs or two outputs. Also, it’s important to use the circular dependency check so no data loops will appear in the flow.

The **VisualNodeBase** internally uses this functions to give a red or green symbol to show if a connection is possible or not.

Node base

Circular buffer

Subscriptions

Scene

Itemlist

Window management

Node serialization

Selection

Memento

Construction

destruction

building a node

tests

FLOW EN UI

TODO

Features