Graphana - program extension

Several parts of *Graphana* can be extended. All extensions are done through java classes. This manual focusses on adding operations.

Contents

1 The framework

This section deals with using the framework in order to develop operations, especially algorithms. The basics of using *Graphana* as a framework are also described in 'graphana_manual.pdf', but a bit more detailed. To try out what is described in this section, a new java project with a main class should be created.
Firstly, the framework has to be imported. When writing a java application in eclipse, *Graphana* can be imported by clicking **Project → Properties → Java build path → Libraries → Add external jars** and choosing the 'graphana.jar' file.
The easiest way to initialize the framework and to execute operations is to create a `GraphanaAccess` instance. This class initializes *Graphana* and automatically registers all the default operations and libraries:

```
GraphanaAccess graphanaAccess = new GraphanaAccess();
```

This can be done within the main method. Now in order to add operations, firstly, a class which extends from a subclass of `Operation` must be created. Especially to add an algorithm, a class which extends `GraphAlgorithm` must be created. This should be done in a separate java file.
The following example is a graph algorithm, which sums up the edge weights of all incident edges of a given vertex. The implementation details of the sample are not important at the moment. They will become clear after reading the other subsections. The focus is on creating and using own algorithms:

```
public class AlgoEdgeWeightSum extends GraphAlgorithm{

  //Signature of the algorithm to call it during runtime
  @Override
  protected String getSignatureString() {
    return "getEdgeWeightSum (vertex:Vertex) : Float";
  }

  //Implementation of the algorithm
  @Override
  protected <VertexType, EdgeType> ExecutionReturn execute(
    GraphLibrary<VertexType, EdgeType> graph,
          UserInterface userInterface,
          OperationArguments args )
  {
    //Start with zero
    float weightSum = 0;
    //Extract the vertex given by the user
    VertexType vertex = args.getVertexArg(0);
    //Iterate over all incident edges of the given vertex
    for(EdgeType edge:graph.getIncidentEdges(vertex)) {
      //Add the weight of the current edge to the result
      weightSum += graph.getEdgeWeight(edge);
    }

    //Return the float wrapped within a GFloat
    return new GFloat(weightSum);
  }
}
```

Now back in the main class, the algorithm can be registered by calling the following `GraphanaAccess`-method:

```
public void registerOperation(Operation operation)
```

In the case of the example above, the call would look like this:

```
graphanaAccess.registerOperation(new AlgoEdgeWeightSum());
```

Afterwards, the algorithm can be called within a *Graphana* statement using the key defined in the signature. The following main method creates a constant testcase for the algorithm: After initialization and registration, the example creates a test graph instance by adding constant vertices and edges. Afterwards, the graph is visualized and the result is printed and finally the user interface main loop is started.

```
public static void main(String[] args)
{
  //Initialize
  GraphanaAccess graphanaAccess = new GraphanaAccess();
  //Register the algorithm
  graphanaAccess.registerOperation(new AlgoEdgeWeightSum());

  //Create test graph instance
  graphanaAccess.execute("createGraph(true,true);");
  //create vertices v0, v1, v2, v3, v4
  graphanaAccess.execute("addVertexRow(5,0,'v')");
  graphanaAccess.execute("addEdge($v1,$v0, 2)");
  graphanaAccess.execute("addEdge($v2,$v1, 3)");
  graphanaAccess.execute("addEdge($v1,$v4, 4)");
  graphanaAccess.execute("addEdge($v2,$v0, 1)");
  graphanaAccess.execute("addEdge($v3,$v4, 5)");

  //Visualize the graph, allow graph modification
  graphanaAccess.getUserInterface().showGraph(true);

  //Print the algorithm result
  System.out.println(
      "Sum of weights of incident edges of v1: "
      + graphanaAccess.executeFloat("getEdgeWeightSum($v1)")
      + " (expected: 9.0)"
      );

  //Start user interface main loop
  graphanaAccess.getUserInterface().mainLoop();
}
```

The example ignores error handling which is described in the *Graphana* manual. Therefore, if an error occurs, the stack trace will be printed and the program will be interrupted.

After printing the result, the program is still running. To further test the algorithm during runtime, the following steps can be done for example:

- Right click on an empty area within the visualization window to create a new vertex.
- Right press onto the new vertex and release the right mouse button on 'v1' to add a new edge, which initially has weight 1.
- Switch to the console, type 'setEdgeWeight($v5->$v1, 6)' and press enter. This changes the weight of the new edge to 6.
- Type `getEdgeWeightSum($v1)` (or try any other vertex) and press enter. Verify the result with the visual output.

This visual way of testing algorithms works well for small test graphs. For larger graphs, java assertions or *Graphana* assertions should be preferred.

Now the implementation is finished and the algorithm is ready to be provided. When compiling the application, the file 'AlgoEdgeWeightSum.class' is generated. This file can now for example be copied into a 'plugins' folder in the graphana.jar directory. After starting the jar, the class can be imported. The following shell commands (in the graphana.jar directory) demonstrate this:

```
java -jar graphana.jar
>import("plugins/AlgoEdgeWeightSum.class")
```

With this, any *Graphana* user can call the operation by only retrieving the class and not the whole test application.

The test application can now be used to test out the concepts of the next section.

2 Operations

This section contains a detailed description on implementing operations.

2.1 Operation types

For a users view on operations, see ”graphana_manual.pdf”. This subsection deals with a programmers view.
The basic class for operations is `Operation`. As already mentioned in the user manual, there are three types of operations. All of them are subclasses of `Operation`:

- **Graph operations:** When derivating from `GraphOperation`, the main input is a graph. This graph can then be manipulated for example.
- **Graph algorithms:** When derivating from `GraphAlgorithm`, which is a subclass of `GraphOperation`, a graph is still the main input but is not supposed to be modified, but to calculate a result, like a graphparameter.
- **Commands:** Classes which derivate from `Command` usually do not operate on a graph, but do some general settings, like for example configure algorithms before executing them. Some of them calculate something which does not have to do anything with graphs, like for example the square root of a given number.

To add a new *Graphana* operation, a class, which extends one of the three `Operation`-subclasses must be implemented and then an instance must be registered in the program (registration will be explained later).

2.2 Signatures

At runtime, every operation is identified by an unique key, can receive arguments and can return a result. All `Operation`-subclasses have this in common. They must define a signature to determine key, parameters and return type. The following method must be implemented:

```
protected String getSignatureString();
```

The signature must be returned as a string in the following simplified syntax:

```
operationKey[|alias1|alias2...] [(
    parameterName:ParameterType1;
    paramName2:ParamType2;
    ...
    paramNameN:paramTypeN
  )] [: ReturnType]
```

Everything in square brackets is optional. Whitespaces, including linebreaks, are ignored when evaluating the signature. The signature will be shown to the user when he types `HELP` and the operation key or one of the aliases.
One example for a signature is the following one:

```
edgeExists|connected (vertex1:Vertex; vertex2:Vertex) : Boolean
```

The operation of the sample signature can be called either with ’edgeExists’ or with the alias ’connected’. It receives two vertices and returns a boolean.

2.3 Execute method

Every type of operation has an **execute** method, which is called, when the operation is called by the user or within a script. The `execute` method is not the same for every operation type. However, all types can receive arguments and can return a result. For example, the `execute` method of a command has the following signature:

```
protected ExecutionReturn execute(MainControl mainControl,
                                  OperationArguments args)
```

Commands will be explained later.

2.3.1 Operation arguments

An `execute` method receives an argument **args** of type `OperationArguments`. It contains the evaluated arguments of the operation call. The particular arguments can be read using the **get*X*Arg** methods, where ***X*** is the respective argument type, for example `getFloatArg`. The arguments are identified with their index whereas the first argument has the index 0. If the type is not determined in the signature (`Any`) then the argument can be accessed with **getArg(Integer index)** which returns the argument as a java `Object`.
So for example, to get the third argument, assuming that this argument is of type **Integer**, then the call within the execute method would look like this:

```
Integer arg = args.getIntArg(2);
```

The **get*X*Arg** call must be consistent to the signature. Otherwise a java exception is thrown.

It is also possible to define operations which can receive an arbitrary number of arguments by writing three dots ("...") at the end of the parameter list. In this case, the caller can pass arguments of the type of the last parameter in the list at any number. The **args.count()** method gives the overall number of given arguments.
For example, an operation with the signature

```
printObjects (separator:String; objects:Any ...)
```

can be called by the user for example with:

```
printObjects(";",  "A string", 56, "Another string")
```

The arguments can be accessed like this within the `execute` method:

```
String separator = args.getStringArg(0);
for(int i=1;i<args.count();i++)
    System.out.println(args.getArg(i).toString() + separator);
```

The output would be:

```
A string;56;Another string;
```

It is possible to define default values for the parameters of a signature by writing a `=` and the value in *Graphana*-syntax after the type of the parameter. If a parameter has a default value then the caller does not need to pass the respective argument and the default value is used instead. For example such a signature could look like this:

```
addEdge (vertex1:Vertex; vertex2:Vertex; weight:Integer=1)
```

With the sample signature, the caller can optionally give an edge weight.
If a parameter of the list has a default value then all following parameters also need a default value consequently. In the `execute` method when using the `args` instance, it is not distinguished between arguments which were explicitely passed in the call and arguments which were passed as default values. Especially, `args.count` also counts the default values in.

2.3.2 Operation return

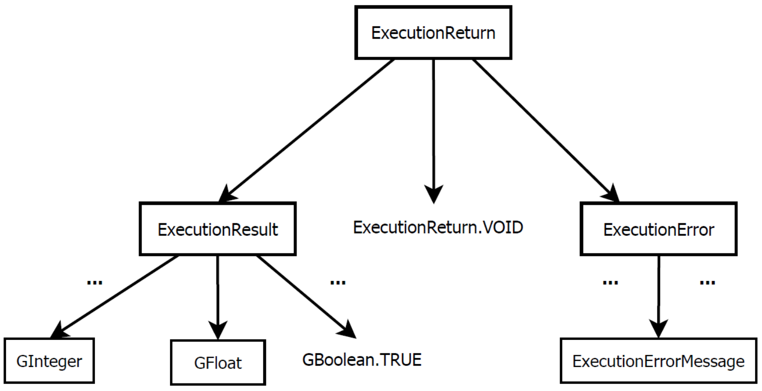
Every `execute` method must return a value of type `ExecutionReturn`. If the operation actually shall not return a result, then **ExecutionReturn.VOID** must be returned. If the operation computes a result, then an instance of a subclass of **ExecutionResult** must be returned. There are several predefined subclasses, one for each *Graphana* type. Examples are `GInteger`, `GBoolean` and `GString`. Every instance holds a value of the respective type.
For example, if an operation returns a **String** then the return statement of the `execute` method could look like this:

```
return new GString("This is a string.")
```

If an error occurs during execution then an instance of an `ExecutionError` or a subclass like `ExecutionErrorMessage` can be returned:

```
if(getFloatArg(0)>1)
        return new ExecutionErrorMessage(
                        "Only values smaller than 1 allowed.")
```

The following image depicts an extract of the structure of the `ExecutionReturn` subclasses:



The figure shows an extract of the relation between the `ExecutionReturn` subclasses. GInteger, GFloat etc. are just samples.

One special case is the `GBoolean` class: It cannot be instantiated directly, but by using `GBoolean.create(boolean)`, `GBoolean.TRUE` or `GBoolean.FALSE`.

2.4 Subclasses of Operation

This subsection deals with the different types of operations and with the differences between them.

2.4.1 Commands

The signature of the `execute` method of a command is the following:

```
protected ExecutionReturn execute(MainControl mainControl,
                                  OperationArguments args)
```

The parameter `mainControl` gives access to the configuration and dates of the program. Especially, it gives access to the `UserInterface` by calling `mainControl.getUserInterface()`. Using the returned interface a string can be printed with the method `UserInterface.userOutput(String message)`.
For example, a simple command which devides an **Integer** with a **Float** could look like this:

```
private class Division extends Command
{
  @Override
  protected String getSignatureString()
  {
    //The signature as a constant String
    return "divide|div␣(dividend:Integer;␣divisor:Float)␣:␣Float";
  }

  @Override
  protected ExecutionReturn execute(MainControl mainControl,
                                    OperationArguments args)
  {
    int dividend = args.getIntArg(0);
    float divisor = args.getFloatArg(1);

    //Check, whether the divisor is zero
    if(divisor == 0)
      return new ExecutionErrorMessage("Division␣by␣0.");
    else
      return new GFloat(dividend / divisor);
  }
}
```

After registration, which was explained in the very first section, the sample command can either be called with `divide` or with `div`. The signature ensures, that the operation can be called only if an integer is given as the first and a float is given as the second argument. So for example the command can be used like this in the running program:

```
res = divide(78, 0.5);
```

2.4.2 Graph operations

Graph operations operate on graphs to modify it or to extract informations. Therefore the `execute` method receives a `GraphLibrary` including the generic types:

```
protected <VertexType,EdgeType> ExecutionReturn execute(
    GraphLibrary<VertexType,EdgeType> graph,
    UserInterface userInterface, OperationArguments args)
```

The `userInterface` and the `args` parameters are already explained in the prior subsection. The given `graph` contains a set of basic graph operations like insertion, deletion and iteration of vertices. If the graph operation is written for a special graph library, then the internal graph can be accessed using `graph.getInternalGraph()`, which returns an `Object` so a type cast to the respective graph class must be done explicitely. Alternatively, a graph operation can use the higher level methods of the given `graph` to be independent of the internal graph library.

This sample graph operation simply adds a vertex with the given identifier to the graph:

```
private class ExecAddVertex extends GraphOperation
{
  @Override
  protected String getSignatureString()
  {
    //The signature string of the algorithm
    return "addVertex (identifier:String) : void";
  }

  @Override
  protected <VertexType,EdgeType> ExecutionReturn execute(
      GraphLibrary<VertexType,EdgeType> graph,
      UserInterface userInterface,
      OperationArguments args)
  {
    //add a vertex to the given graph
    graph.addVertex(args.getStringArg(0));

    //This operation does not return a result
    return ExecutionReturn.VOID;
  }
}
```

The operation works for every graph library, because it only uses the higher level method `GraphLibary.addVertex` and therefore never accesses the internal graph directly.

Graph preconditions

A graph operation can set up some graph preconditions. That means, that, in addition to the fact that the operation cannot be called with the invalid argument types, a graph operation cannot be called if the respective graph does not fullfill the graph preconditions. Therefore, there is no need to check the conditions within the `execute` method and to do error handling. Furthermore, the graph preconditions are automatically listed in the documentation of the operation. The graph preconditions of a graph operation can be configured by calling the respective methods in the super class, which will be explained in the following.

The following method determines, with which graph libraries the operation can be called:

```
protected final void setCompatibleLibs(
    GraphLibrary<?,?>[] compatibleLibs)
```

If `setCompatibleLibs` is never called then the operation is assumed to be compatible with every graph library, which means that it does not access the internal graph directly but work with the higher level methods offered by `GraphLibary`.

In addition, an operation can allow and disallow graph configurations by calling the following method:

```
protected final void setAllowedGraphConfig(
                                    boolean directedAllowed,
                                    boolean notSimpleAllowed,
                                    boolean emptyAllowed)
```

If `directedAllowed` is false then the given graph is always undirected. If `notSimpleAllowed` is false then the graph does not have loops. If `emptyAllowed` is false then the graph has at least one vertex.

If the following method is called then the received graph is always directed:

```
protected final void setAlwaysConvertToDirected()
```

A graph will automatically be converted into an equivalent directed graph when the operation is called with an undirected graph.

If the following method is called then the received graph is always a deep copy of the original graph:

```
protected final void setAlwaysCopy()
```

This is useful, if the graph operation for example modifies the graph temporally to compute something. The original graph will not be affected.

The configuration methods can be called within the `getSignature` method or within the constructor, but never within the `execute` method. An example will be shown in the next subsection.

2.4.3 Graph algorithms

Algorithms are special graph operations so everything that was explained in the previous subsection is also true for algorithms. The `execute` method has the same signature as the `execute` method of graph operations and the graph preconditions are working the same way. There are only few differences. For example the algorithm counter does not measure the time of graph operations but only of algorithms.

The following sample is the vertex cover size `GraphAlgorithm`. Basically, the class only calls the vertex cover algorithm of JGraphT:

```
public class AlgoVertexCoverSize extends GraphAlgorithm {
    @Override
    public String getSignatureString()
    {

        //GRAPH PRECONDITIONS
        //The algorithm can only be called with JGraphT
        this.setCompatibleLib(new JGraphTLib());
        //The graph must be undirected and simple.
        //Theres no restriction for weighted/unweighted
        this.setAllowedGraphConfig(false, true, false);

        //The signature string of the algorithm
        return
            "vertexCoverSize|vertexCover (useGreedy:Boolean) : Integer";
    }

    @Override
    protected <VertexType,EdgeType> ExecutionReturn execute(
        GraphLibrary<VertexType,EdgeType> graph,
        UserInterface userInterface,
        OperationArguments args)
    {
        //The internal graph can only be of the JGraphT type
        //UndirectedGraph because of the graph preconditions
        UndirectedGraph<VertexType,EdgeType> internalGraph =
            (UndirectedGraph<VertexType,EdgeType>)graph.getInternalGraph();
        Set<VertexType> cover;
        //The argument determines, which heuristic shall be used
        if(args.getBoolArg(0))
            cover = VertexCovers.findGreedyCover(internalGraph);
        else
            cover = VertexCovers.find2ApproximationCover(internalGraph);
        //Return the result as an Integer
        return new GInteger(cover.size());
    }
}
```

2.5 Operation groups

Operations can be gathered in groups by derivating from `OperationGroup`. The following method must be implemented:

```
public Operation[] getOperations();
```

The method must return an array of instances of the operations which shall be registered when registering the group. The registration of a group works the same way as with single operations.

For example the commands `startCounter` and `getCounter` are subclasses gathered in one group so they can easily share a variable:

```
public class CmdsCounter extends OperationGroup{

  long timer;

  private class ExecStartCounter extends Command
  {
    @Override
    protected ExecutionReturn execute(MainControl mainControl,
                                       OperationArguments args)
    {
      timer = System.currentTimeMillis();
      return ExecutionReturn.VOID;
    }

    @Override
    protected String getSignatureString()
    {
      return "startCounter|restartCounter|setCounter␣:␣void";
    }
  }

  private class ExecGetCounter extends Command
  {
    @Override
    protected ExecutionReturn execute(MainControl mainControl,
                                       OperationArguments args)
    {
      if(timer<=0)
        return new ExecutionErrorMessage
          ("Counter␣not␣set␣yet.␣Call␣'startcounter'␣first.");
      else
        return new GInteger
          ((int)(System.currentTimeMillis()-timer));
    }

    @Override
    protected String getSignatureString()
    {
      return "getCounter␣:␣Integer";
    }
  }

  @Override
  public Command[] getOperations()
  {
    return new Command[]{
        new ExecStartCounter(),
            new ExecGetCounter()
          };
  }
```

Both commands are inner classes and share the variable timer. To register the two commands, only the group `CmndsCounter` has to be registered. When using `GraphanaAccess`, the registration looks like this:

```
graphanaAccess.registerOperations(new CmdsCounter());
```