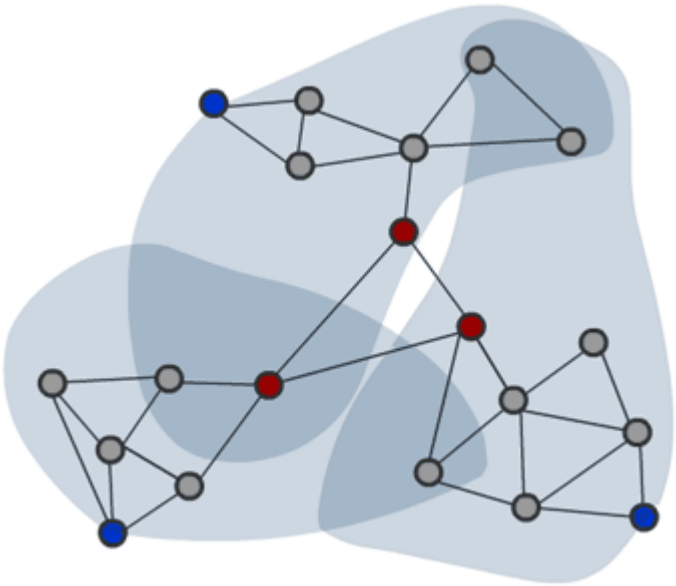


MITANDAO

Social network analyzer

DEVELOPER GUIDE

Content

What is Mitandao?

Mitandao is an open source software for social network analysis which can be used as a stand alone application or as a library. Mitandao is an extensible application, where you can add your own modules for network analysis. Mitandao library provides useful framework for creating social network analysis application. You can find all important information about Mitandao, as application, library or a framework for developers in our user guides.
Mitandao was developed by group of students of Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Slovakia.

How to create module into the Mitandao

Introduction

All the inputs, outputs, algorithms and filters in the Mitandao project are dynamically loaded modules. The user can create it's own module and let the Mitandao to load it. This module is then usable in the same way as any other module to participate in the work flow.

How to do this is described in this guide.

Modules types overview

There are four types of modules:

- *Input Module* used to load the social network from file, database etc.
- *Output Module* used to save the social network to file, database etc.
- *Filter Module* used to remove nodes from the loaded graph according to some condition
- *Algorithm Module* used to analyze the graph

How to create a module

Each of module types has defined its own interface named `InputModule`, `OutputModule`, `FilterModule` and `AlgorithmModule`.

Each of this modules extends the Module interface, which defines the methods:

```
public Graph analyze(Graph graph);
public String getName();
```

If you would like to create a module, you have to implement one of the interface and the methods analyze and `getName`.
The method `getName` is used to return some human readable name of the module. It is used for example in the Wizard in the Mitandao GUI to fill the combo boxes with set of available modules.
The method analyze is used to call the module to do, what it is for. It gets the currently last graph as input and it returns the new version of the graph (for example with assigned calculated values to the nodes).
If you would like to create for example an input module, you would have to create a class, implement the `InputModule` interface and implement the analyze and `getName` methods.

```
public class SomeFileReader implements InputModule {
```

```
        @Override
        public Graph apply(Graph graph) throws Exception {

                Graph readGraph;
                // read the graph
                return readGraph;
        }

        public String getName() {
                return("Example input module");
        }
}
```

The method apply throws Exception, if any exception occurs. This exception is then processed with the Mitandao core.

This module is compilable and loadable to the Mitandao and usable in the work flow but it has a problem. It takes a module as input and returns an another instance. It means, it ignores the input module, so if somebody creates a work flow, where this module is not the first part of the work flow, any analyze made before this module is used is lost. So all the time, when you create any module, copy all the created data to the instance of the graph you get. How to do this is described in the section How to use the user data storage.

One more thing to know is, that this module does not get any input from the user. If you need to create a module, which gets input from the user via the graphical user interface read the How to use the Mitandao UI framework section. If you need to create a module, which gets input via the MitandaoImpl's `setModuleParameters` method, read the How to create from variable module parameter.

How to use the user data storage

Introduction

The user data are stored in the graph under the specified key (usually, this is the qualified name of the module class). Mitandao library contains two data holder – `MitandaoVertexLabeller` and `MitandaoEdgeLabeller`.

How to work with labellers

You can get `MitandaoVertexLabeller` and `MitandaoEdgeLabeller` directly for the specific key.

```
MitandaoVertexLabeller labeller =
                                                    (MitandaoVertexLabeller)
graph.getUserDatum(key);
```

But the preferred way is to obtain it from `MitandaoVertexLabeller` directly, because it is type safe and if the datum under the key does not exist, new `MitandaoVertexLabeller` is created.

```
MitandaoVertexLabeller labeller = MitandaoVertexLabeller.getLabeller(graph,
key);
```

The nodes names are stored under the key

```
MitandaoVertexLabeller.DEFAULT_VERTEX_LABELER_KEY
```

You can access it in a simplified way without the key.

```
MitandaoVertexLabeller labeller = MitandaoVertexLabeller.getLabeller(graph);
```

All data are represented as `Strings`. You can access the data in labeller through method `getLabel`.

```
labeller.getLabel(vertex);
```

How to use this to copy user data

In the section How to create a module there was written that the given graph in the analyze method has not to be ignored and the data given in the specific method has to be copied into this graph. The following example demonstrates how to do this.
For example you would like to create an input module, which reads the pajek file using the Jung's load method. It is done like this.

```
public Graph apply(Graph graph) throws Exception {
                    // create instance of reader from the JUNG library
                    PajekNetReader reader = new PajekNetReader(true);
                    // loading the graph
                    Graph outputGraph = reader.load(getFilePath());

                    // getting the labeler under which are stored the data from
the loading
StringLabeller pajekLabeller = StringLabeller.getLabeller(outputGraph,
PajekNetReader.LABEL);

                    // getting the mitandao labeler, where the data I would like
to store
                    MitandaoVertexLabeller mitandaoLabeller =
MitandaoVertexLabeller.getLabeller(outputGraph);

                    // copy the labels from the JUNG's labels to the Mitandao
labels
                    MitandaoGraphUtils.copyVertexLabels(pajekLabeller,
mitandaoLabeller);

                    // removes the JUNG's labels
                    outputGraph.removeUserDatum(PajekNetReader.LABEL);

                    // copy the data from the graph loaded using the load method
to the graph given from the previous analysis
                    return MitandaoGraphUtils.union(outputGraph, graph);
          }
```

How to use the Mitandao UI framework

Introduction

The Mitandao UI framework is a part of Mitandao project, which makes it easy to build graphical modules into the Mitandao GUI.

There are three ways how to build graphical user interfaces using this framework:

- Let the framework generate the graphical panel from the module according to the annotations provided in this module
- Provide your own (not reusable) panel and connect it to the specific module using the panel's annotations
- Provide your own reusable panel and specify its name, so it can be used in other modules and you can access it through annotations. The example of this panel is an open or the save dialog.

In the next section we will describe all of this three ways in detail.

Automated panel generation

This is the simplest way how to use the Mitandao UI framework. To create a graphical panel, you need to create a module (how to do this is described in the How to create module into the Mitandao section).

For example this simple module:

```
public class SampleOutputModule implements OutputModule{

          private String example = "this is an example";

          private Integer exampleInteger = new Integer(12);


          public String getExample() {
                    return example;
          }

          public void setExample(String example) {
                    this.example = example;
          }

          public Integer getExampleInteger() {
                    return exampleInteger;
          }

          public void setExampleInteger(Integer exampleInteger) {
                    this.exampleInteger = exampleInteger;
          }
...
```

There are two parameters in this module:

```
private String example
```

and

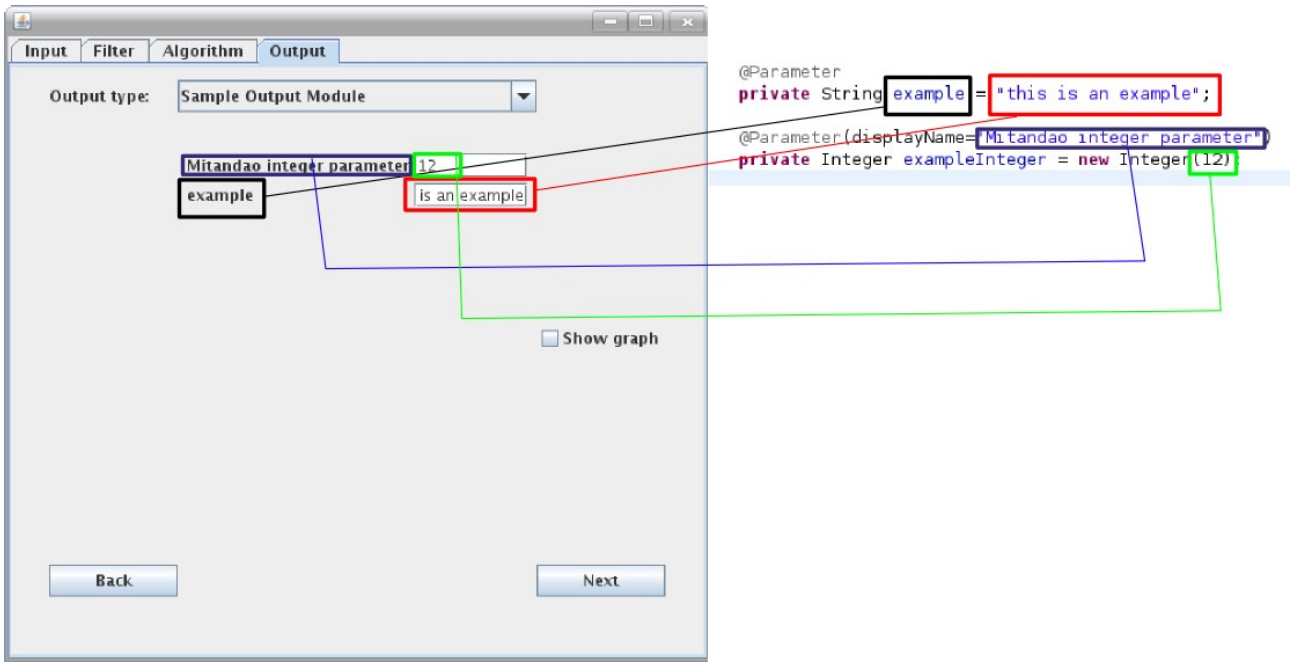
```
private Integer exampleInteger
```

What we need is to set these two parameters as input parameters of the module to have two text fields generated in the graphical panel. This is done via the `@Parameter` annotation. For example:

```
@Parameter
private String example = "this is an example";

@Parameter(displayName="Mitandao integer parameter")
private Integer exampleInteger = new Integer(12);
```

In the first example, there is set only the Parameter annotation, in the second there is set the display name too. It means, that in the first example the label before the text field will be exactly the same as the name of the parameter – '*example*'. In the second parameter, it will be the '*Mitandao integer parameter'* (as described in the display name). In the following figure there is shown how this module will look like and how are the parameters mapped to the screen.



The framework provides functionality to copy the values from the module to the panel and from the panel to the module automatically. It also provides functionality which takes care about the data type checking (e.g. if the parameter is type of double, the user can copy only the type of double into the generated text field).

Constraints

- Every parameter has to have the getter and setter method
- The supported parameter types are only
  - `java.lang.String`
  - `java.lang.Double`
  - `java.lang.Integer`

Providing not reusable panel

This is the second way how to create graphical modules into the Mitandao project. In the previous part it was described how to let the framework to generate the panel. In this section you will need to create your own panel. This can be useful if you need to create a more complex panel than a set of label - text field rows (e.g. buttons, radio buttons etc.).

At first, you will need to create a module, then the panel and connect module with the panel via the `@Panel` annotation. For example:

```
@Panel("sk.fiit.mitandao.modules.inputs.ItsPanel")
public class SomeModule implements InputModule {
…
```

The parameters on the module side are marked with the `@Parameter` annotation (the same way, as described in the previous section). For example:

```
...
@Parameter
private String helloWorld = "Hello world";
...
```

The parameters on the panel side are marked with the `@RemoteParameter` annotation. They must have the same name as the parameters on the module side. For example, the panel for the module from above has to have the following parameter:

```
@RemoteParameter
private String helloWorld;
```

This means that the parameter `helloWorld` is the same parameter in the panel and in the module. The values between them are copied by the Mitandao UI framework.
The module side is from now the same as the module described in the first chapter. The next thing is to create the Panel side. The class has to extend `JPanel` and implement the `ParameterSetter` interface. The `ParameterSetter` interface defines the following two methods:

```
public void initialize();
public void setParameters();
```

The usage of this panel by the library is as follows:
At first, the parameters' values from the module instance are copied to it's remote parameters on the Panel. Then the `initialize()` method is called. There should be the creation of graphical components and setting their values according to the `@RemoteParameter` parameters.
The next step is the user's work with this panel (e.g. filling the values of the components etc.). Then, when the user finishes the work with the panel, the `setParameters()` method is called. There should be the values from the graphical components copied to the `@RemoteParameter` parameters.

Finally, the values from the `@RemoteParameter` parameters are copied to the Module. The example of the remote panel looks as follows:

```
public class ItsPanel extends JPanel implements ParameterSetter {

          @RemoteParameter
          private String helloWorld;

          private JTextField textField = new JTextField();
          @Override
          public void initialize() {
                    textField.setText(helloWorld);
                    add(textField);
          }

          @Override
          public void setParameters() {
                    helloWorld = textField.getText();

          }
…
```

Constraints

- Every parameter has to have the getter and setter method
- The supported parameters types are only
  - `java.lang.String`
  - `java.lang.Double`
  - `java.lang.Integer`

Providing reusable panels

In the previous section there was described the way how to create custom panels. These panels were connected to the module via the `@Panel` annotation. But these panels were usable only from modules, which has the exactly same parameters as the remote panel's remote parameters. In general, these panel are not reusable.

If you need to create custom panels that are reusable in other modules, you need to create them as follows.

For example, you would like to have a file chooser. It is a panel, which has a `JTextField`, `JButton` and `JFileChooser` which cooperates in a specific way. You wold like to use this panel in many other modules.

At first, you need to create a Panel (let's call it FileChooserPanel), that extends `JPanel` and implements the `CustomPanels` interface. The `CustomPanels` interface defines the following two methods:

```
public String getValue();
```

and

```
public void setValue(String value);
```

which corresponds to the input and output of specific `customPanel`. The module side is implemented in the same way, as in the first section of this document, and the reusable panels are accessible thought the `@SpecialParameter` annotation like this:

```
@SpecialParameter(SpecialParameterType.FILE_OPEN_DIALOG)
private String filePath = "/some/path/pajekFile.net";
```

So, what it means. At first we will explain the

```
private String filePath = "/some/path/pajekFile.net";
```

line. When the `FileChooserPanel` is created, the `setValue(String value)` method is called with input value from `filePath` variable. It means that you will need to copy this value in the method `setValue` to some graphical component. For example:

```
        public void setValue(String value) {
                if (value == null) return;

                pathField.setText(value);

                File currentFile = new File(pathField.getText());
                if (currentFile != null) {
                        getChooser().setSelectedFile(currentFile);
                }
        }
```

Then the user is able to work with this panel (e.g. choose some other file). After this, the method `getValue()` is called on this panel. It should return the value, which you want to have as an output value of this panel (e.g. the chosen file). It should look like this:

```
public String getValue() {
                return pathField.getText();
        }
```

The value returned from this method is then copied to the `filePath` field.
Now it will be described, how to make this panel accessible via the `@SpecialParameter` annotation.
You will need to register this panel to the `SpecialParameterType` enumeration. The registration looks like this:

```
FILE_OPEN_DIALOG("sk.fiit.mitandao.gui.modulepanels.predefinedpanels.FileOpenPa
nel")
```

where the *FILE_OPEN_DIALOG* is the symbolic name and the `"sk.fiit.mitandao.gui.modulepanels.predefinedpanels.FileOpenPanel"` is the full name of the panel.
Now, the `FILE_OPEN_DIALOG` is usable in any Mitandao module.

Constraints

- Every parameter has to have the getter and setter method
- The reusable custom parameter (panel) has to have only one input and only one output.
- The reusable custom parameter (panel) has to have only String input and output.

How to create from variable module parameter

Introduction

The `MitandaoImpl` class, which is the main interface to the Mitandao library, has for this purpose this to interesting methods:

```
public void setModuleParameters(Module module, Map<String, Object> parameters)
public Map<String, Object> getModuleParameters(Module module)
```

This methods sets and gets the values from and to the modules' parameters. How to mark the module's parameter to be usable this two methods is described in this section.

Variables versus module parameters

If you create a module, which has for example a variable declared like this

```
private String someParameter = "This is some parameter";
```

you can not use the methods described above to maintain the content of this variable. For the library it is only some private variable, which is ignored. To make from this variable module parameter use the Parameter annotation and create standard getter and setter methods to it. An example of a `String` module parameter is as follows.

```
@Parameter
private String someParameter = "This is some parameter";
public String getSomeParameter() {
        return someParameter;
}

public void setSomeParameter(String someParameter) {
        this.someParameter = someParameter;
}
```

Now the `someParameter` is a module parameter and the methods `setModuleParameters` and `getModuleParameters` are usable to maintain it's content.

Constraints

- Every parameter has to have the getter and setter method
- The supported parameter types are only
  - `java.lang.String`
  - `java.lang.Double`
  - `java.lang.Integer`

Creating and adding new modules

After deciding what type of module you want to create you have implement the appropriate interface. Like described in library guide there are 4 possible module types. This types have following interface names:

- `InputModule`
- `AlgorithmModule`
- `FilterModule`
- `OutputModule`

The corresponding package names are:

- `package sk.fiit.mitandao.modules.inputs;`
- `package sk.fiit.mitandao.modules.algorithms;`
- `package sk.fiit.mitandao.modules.filters;`
- `package sk.fiit.mitandao.modules.filters;`

A simple example of an module implementation:

```
package sk.fiit.mitandao.modules.algorithms;

import sk.fiit.mitandao.modules.interfaces.AlgorithmModule;

public class Betweeness implements AlgorithmModule {

          @Override
          public Graph apply(Graph graph) throws Exception {
                    return graph;
          }

          @Override
          public String getName() {
                    return "JUNG Betweeness Centrality";
          }

}
```

After implementing the appropriate interface you have to override two methods, which come from the parent interface – Module. All 4 interface mentioned above are directly inherited from this interface.

```
public interface InputModule extends Module{

}
```

The `apply()` method is the executive method which work with the Graph object /create graph from some data source, calculate measurement, change structure, show the result/.

The `getName()` method is used for getting the name so it can be showed in the list of available modules in appropriate wizard tab.

Constraints

- Every module has to implements one of the interfaces and override mentioned methods.

Linking modules

The Mitandao uses dynamical loading of the module classes in order to give you the list of available modules. Therefore it does not search classes on the standard classpath, but on the paths specified in the special XML file named modules_paths.xml. This XML file has to be placed somewhere on the classpath (the easiest way is to put it next to the mitandao.jar file).
The sample XML file looks like this:

```
<modulePaths>
  <modulePathToClass>../MITANDAO/bin</modulePathToClass>
  <modulePathToClass>./modules</modulePathToClass>
  <modulePathToClass>file:///home/lula/programs/mitandao/modules</modulePathToClass>
</modulePaths>
```

You can use absolute or relative paths in this XML. This path has to lead to the first directory of the package structure. If you use the absolute path, you have to write the protocol prefix file:// at the beginning. If you use relative paths, they are relative to the directory returned by the system property user.dir (by System.getProperty("user.dir")). This is the directory where the application was run.
Notice that even in Windows you can use the "/" in the relative paths.

Example:
If your directory looks like this:

```
/home/lula/programs/mitandao/modules/sk/fiit/mitandao/modules/MyModule.class
```

And sk.fiit.mitandao.modules.MyModule.class is the full name of your class, than the path specified in the XML would look like this:

```
file:///home/lula/programs/mitandao/modules
```

If you starts your application in the mitandao directory, the relative path would look like this:

```
./modules
```