Operator Editor

User manual

Contact: **Dorian Gorgan**, dorian.gorgan@cs.utcluj.ro

Cluj-Napoca, 2012

Contents

1. Introduction

This platform has been created with the idea that individual users of the GreenLand platform may be able to customize their programs running on the Grid up to their smallest elements, which are the operators.

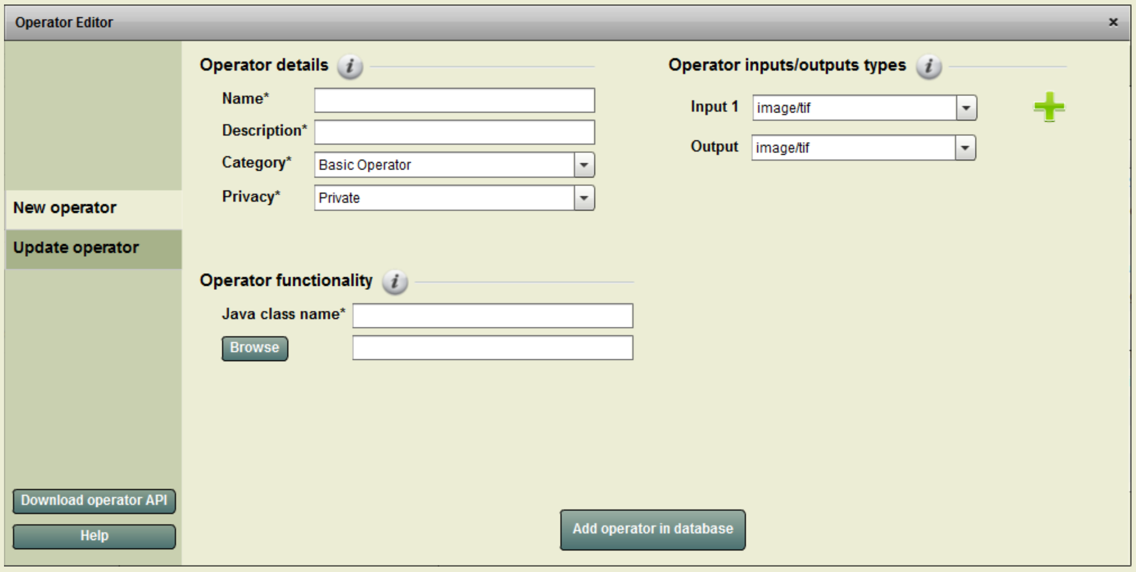
This document is an introduction into operator creation, and updating. It also highlights the rules for these operations. It presents in great detail the Graphical User Interface and Programming Interface used in this application.

2. Interface

The Operator Editor is part of the GreenLand application which may find at http://cgis2ui.mediogrid.utcluj.ro/GreenLandv2/. Once signed in, the user will select "*Add Workflows*" from the top menu bar, and then from the newly opened section select the "OperatorMng." button. At this moment a new modal widow will be displayed (Figure 1).
The user is able to create a new operator by using the "*New operator*" menu option. There is an approximately 1-2 days delay for releasing the operator for production usage (this time is required for the GreenLand team to validate it and to assure that it function correctly both on standalone machines and Grid infrastructure). If the operator is not valid, it will be not made accessible throughout the GreenLand application, not even for its owner.

Before using this option, there is the need to download an example (using the "*Download operator API*") that highlights how to extend basic functionalities in order to create your own operator. Within this page some mandatory fields are displayed to the users:

- **Name**: represents the operator's name throughout the GreenLand application;
- **Description**: it is useful to know what is the functionality provided by this operator,



Name*	
Description*	
Category*	Basic Operator
Privacy*	Private
Java class name*	
Input 1	image/tif
Output	image/tif

Figure 1) Operator creation module

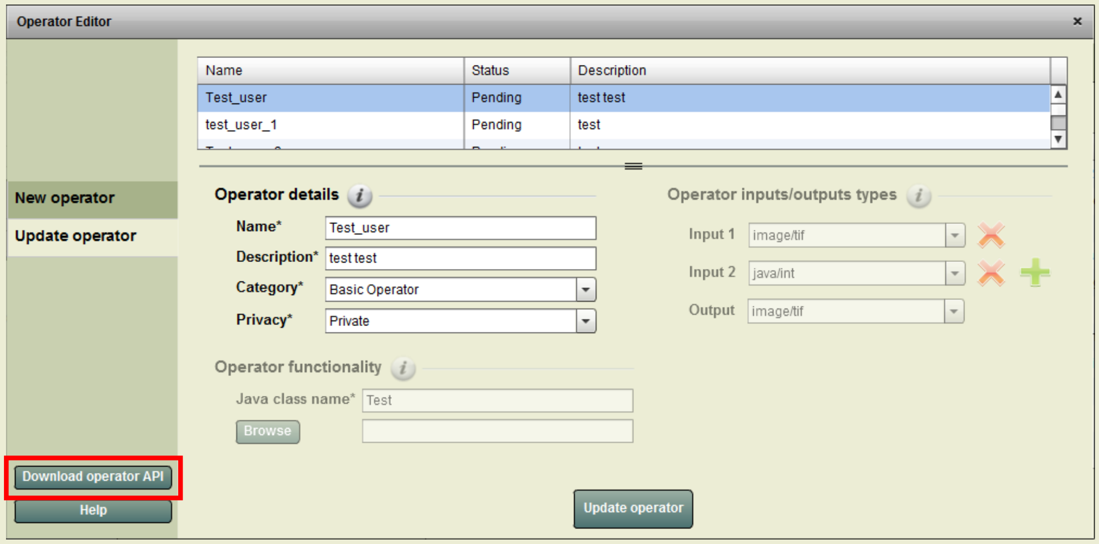
especially for users that do not own the module;

- **Category**: the best solution is to group operators that share the same functionality. Choose the best one for your operator;
- **Privacy**: public or private attributes will influence the operator's visibility throughout the GreenLand application. Private use means that the module could be used only by its own creator, while the public property allows its usage for the entire community;
- **Java class name**: this input should contain exactly the same name as the java class that was used to extend the API functionality;
- **Browse**: when creating a new operator, there is the need to extend the functionality provided by the existing API archive. This is a java example that needs to be further changed in order to implement your own type of actions. When inserting the operator into the GreenLand system, you should provide an archive with all the java classes, including external libraries and dependencies. The archive could have .rar, .zip, and .tar.gz formats;
- **Inputs/outputs types**: this section allows the users to specify the inputs and the outputs types, based on the java internal implementation.

When changing the module's properties, the "*Update operator*" menu option should be used (Figure 2). On the top of the page there is a table of operators, belonging to the current signed in user. Selecting one of them, will enable its properties to be displayed in the lower section of the page that has similar functionalities with the one presented in the operator creation paragraph.

There are three states available for an operator:

- **Pending**: set just after its creation. This means that the operator will be first analyzed by the GreenLand team, to verify if it adopts the principles expressed earlier. Name, description, category, and privacy are the only fields that could be update within this period;
- **Updated_requested**: the functionality provided by the operator contains syntactic error, or



Name	Status	Description
Test_user	Pending	test test
test_user_1	Pending	test

Operator details

Name* Test_user

Description* test test

Category* Basic Operator

Privacy* Private

Operator functionality

Java class name* Test

Operator inputs/outputs types

Input 1 image/tif

Input 2 java/int

Output image/tif

Figure 2) Update operator module

it does not perform correctly. At this stage, it is recommended to update the operator's implementation, by rewriting the java classes. Operators that share this status are not made accessible throughout the GreenLand application;

- **Valid**: the operator functions correctly and it is released to the users community.

3. Programming Interface

To the user interface there is also attached a programming interface, which allows the user to create the operators and to download a gProcess compatible framework for executing operators on the Grid.

Why is there the need for this programming interface? It is because gProcess has a unique way of passing parameters to its operators. This mode of passing constrains the operator to be implemented in Java and to have its arguments passed in a specific, predefined manner.

During the course of this chapter we will present the steps and requirements in order to build your own gProcess operator. We will show some tricks in order to embed already existing programs within the newly created operator. These may be implemented in any programming language as long as these programs can run within the Linux operating system. This constraint is imposed by the underlying Grid platform.

Finally we will present the GRASS framework for Grid execution and how by simply extending a class and overriding one methods, the user can have his or her own operator. The IDE's of choice for presenting this task will be Eclipse and Netbeans.

1. Downloading the API

Before the user can start implementing his or her operator it is advised to download the API (http://cgis2ui.mediogrid.utcluj.ro:8195/GreenLandv2/OperatorAPI.rar) for gProcess execution and start exploring it, in order to get a better grip of what is allowed and how parameters are passed to the program. Please see (Figure 2) for an example as to from where to download the file.

The content of the downloaded API is as follows:

a. JAR containing the API for gProcess execution and GRASS deployment: **Operators.jar**
b. JAR containing the dependencies for Grid execution: **apmon.jar**
c. GRASS application distributable archive: **grass.tar.gz**
d. GRASS dependency package: **dummy_location.tar.gz**
e. Main class of Operators.jar: **OperationExec.class**

In order to be able to use the framework all the files specified above must be included within the archive submitted as the operator files. Please see Sect. 2 .

Before continuing to the implementation phase of this tutorial it is advised for the user to have a general overview of the limitations this framework imposes:

a. Testing of any operators to be inserted within the gProcess platform will be done within the confines of a Linux operating system
b. All Operators which will be submitted to gProcess will be compiled using the JDK 1.5 or any previous version of the Java SE platform
c. When creating the operators, be it that the programmer uses the framework provided or starts from scratch. Either way the strict mode of passing parameters must be respected. More will be detailed in the next Section.

2. Running program locally

Before uploading your operators to gPorcess it is best that the user tests them locally. In order to be able to do this you should follow these steps:

a. Install Java SE 1.5 on your Linux machine
b. Install the development kit on your IDE
c. Select the default development kit for your project
d. Determine your working directory, and program arguments of your project

After completing these steps you will be able to run your application locally. The IDE's for which each of these steps will been discussed and detailed are Eclipse and Netbeans.

2.1. Install Java SE 1.5 on your Linux machine

First step is to download the Java Development Kit version SE 1.5 for Linux from http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase5-419410.html or any other source the user may wish.

It is required that you compile your program using this version due to the fact that certain nodes of the Grid have not been updated to use the latest version of the running environment.

2.2. Install the development kit on your IDE

In order to be able to make your IDE use the desired version of Java development kit you must first make it aware of it.

Eclipse:

Select from the menu bar **Run**-> **Run Configurations**

Select **Java Application** as the type you want to configure. If there are no configurations available please create one by right clicking on it and selecting **New**.

Go to the **JRE** Tab and select **Installed JREs**:

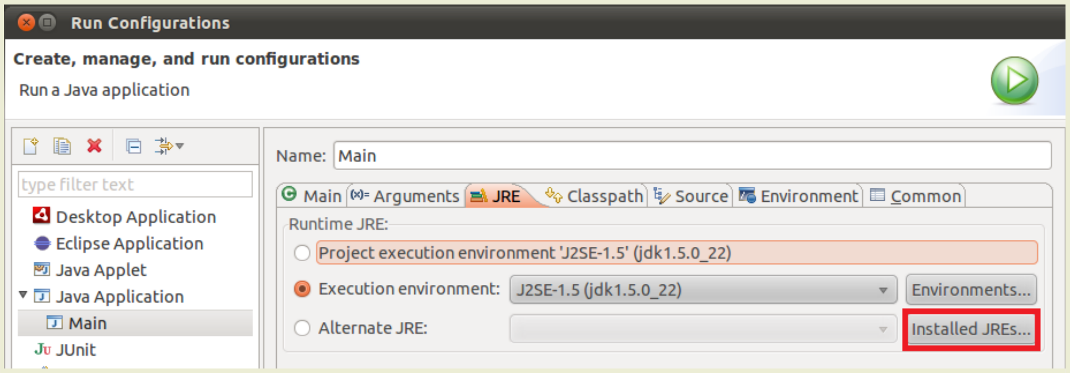
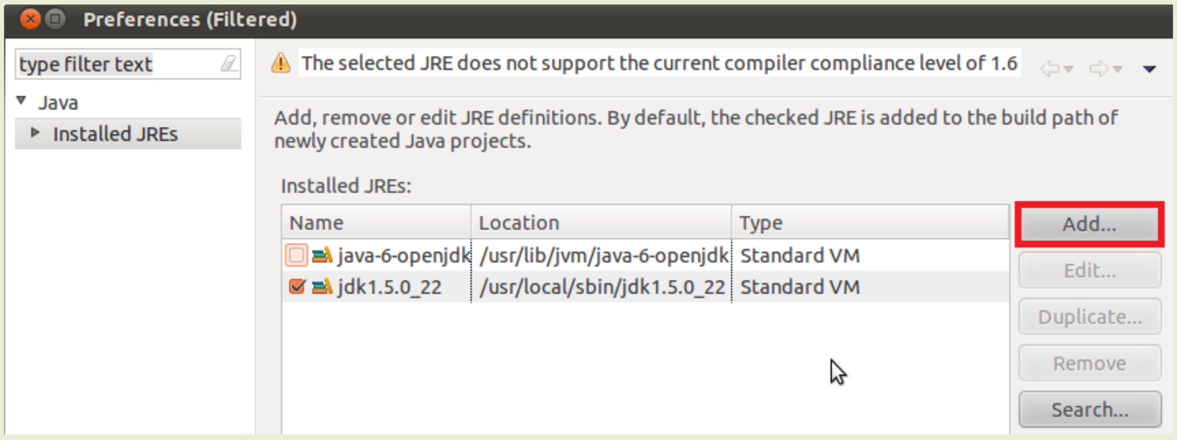


Figure 3) Browse Installed JREs in Eclipse

Press **Add** to add a new Development Kit and specify the path toward your new **JRE folder**:



The selected JRE does not support the current compiler compliance level of 1.6

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

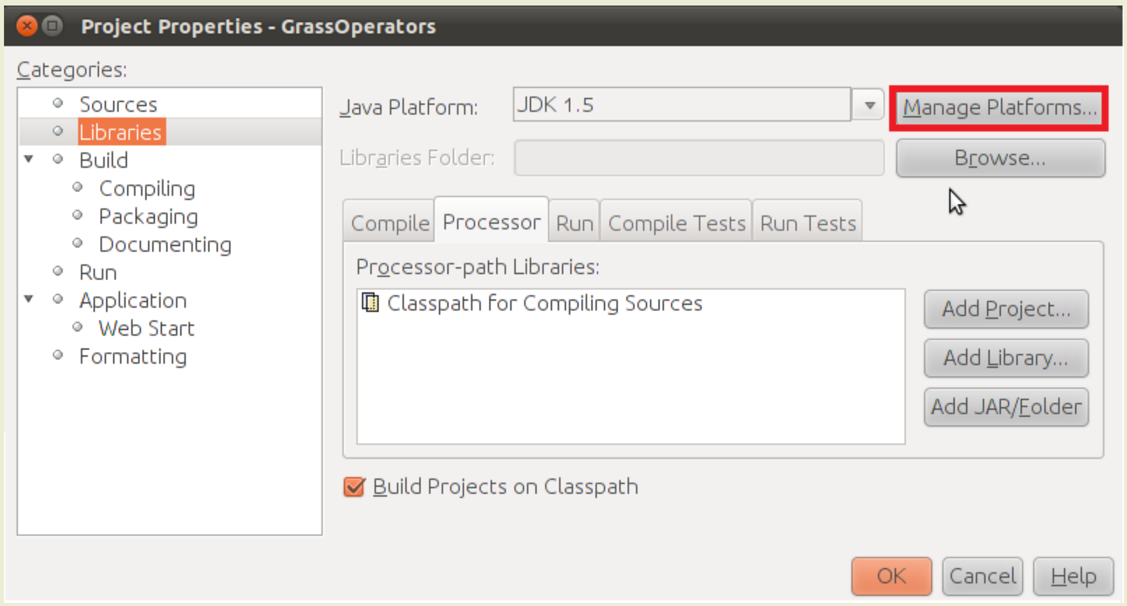
Name	Location	Type
☐ java-6-openjdk	/usr/lib/jvm/java-6-openjdk	Standard VM
☑ jdk1.5.0_22	/usr/local/sbin/jdk1.5.0_22	Standard VM

Figure 4) Add Reference to JRE in Eclipse

<u>**Netbeans**</u>:

Select from **Project Explorer** any project and select **Properties.**

Select **Libraries** category and choose Manage Platforms


Project Properties - GrassOperators

Java Platform: JDK 1.5

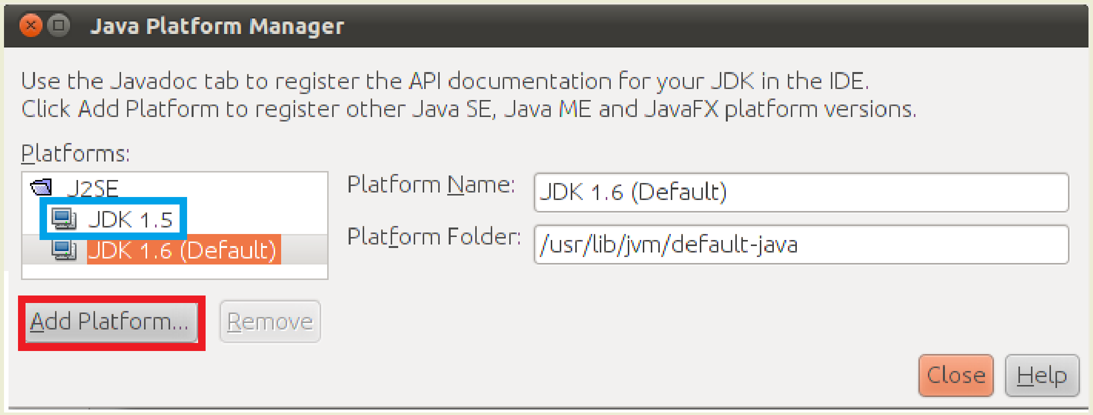
Libraries Folder:

Processor-path Libraries: Classpath for Compiling Sources

[✓] Build Projects on Classpath

Figure 5) Select JRE in Netbeans

Specify the path towards the Java SE 1.5 development kit.


Java Platform Manager

Use the Javadoc tab to register the API documentation for your JDK in the IDE.
Click Add Platform to register other Java SE, Java ME and JavaFX platform versions.

Platform Name: JDK 1.6 (Default)

Platform Folder: /usr/lib/jvm/default-java

Figure 6) Add reference to JRE in Netbeans

2.3. Select the default development kit for your project

In order to compile your project using the Java SE 1.5 Development Kit, you will have to tell your IDE which platform to specifically use.

At this moment we start from the premise that the user has already created a project.

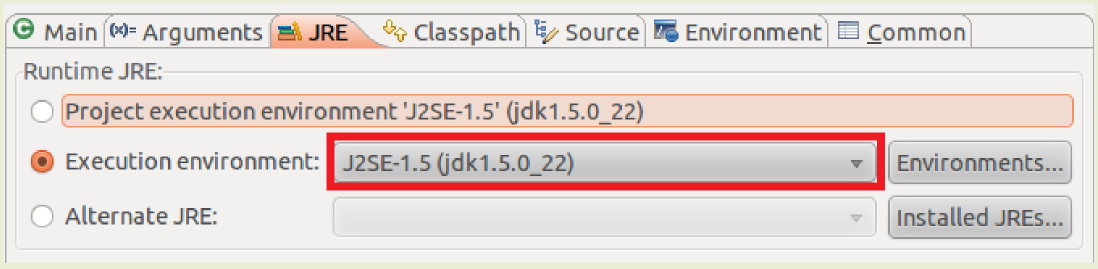
Eclipse:

When working with Eclipse it is important to note that the **Run Configurations** are not attached to a given project in a one to one fashion as in Netbeans.

Create new Run Configurations for your Project:

Run->**Run Configurations**-> **Java Application**-> **New**

JRE->Alternate JRE->JSE 1.5



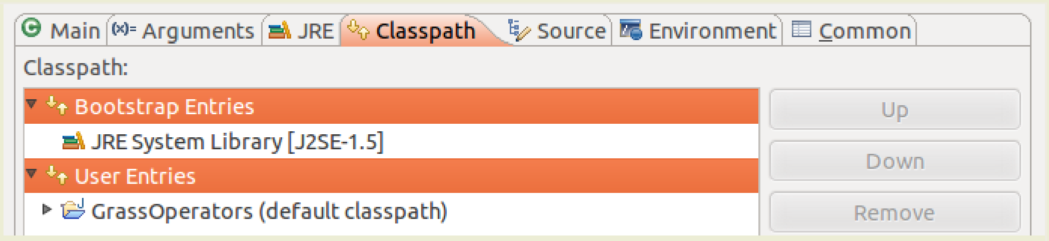
Runtime JRE:
- ○ Project execution environment 'J2SE-1.5' (jdk1.5.0_22)
- ● Execution environment: J2SE-1.5 (jdk1.5.0_22)
- ○ Alternate JRE:

Figure 7) Eclipse Project Java Running Environment

Set Java Running Environment Libraries for your Project:

Classpath->**Bootstrap entries**-> **Add** JRE System Library [J2SE-1.5]

-> Remove JRE System Library [J2SE-1.6] or [J2SE-1.7]



Classpath:
- Bootstrap Entries
  - JRE System Library [J2SE-1.5]
- User Entries
  - GrassOperators (default classpath)

Figure 8) Add Eclipse Classpath Entries

<u>**Netbeans**</u>:

Each Netbeans project has exactly one running configuration attached to it.

Select Project -> **Project Properties** -> **Libraries** -> **Java Platform**

Please see (Figure 5).

2.4. Determine your working directory and program arguments of your project

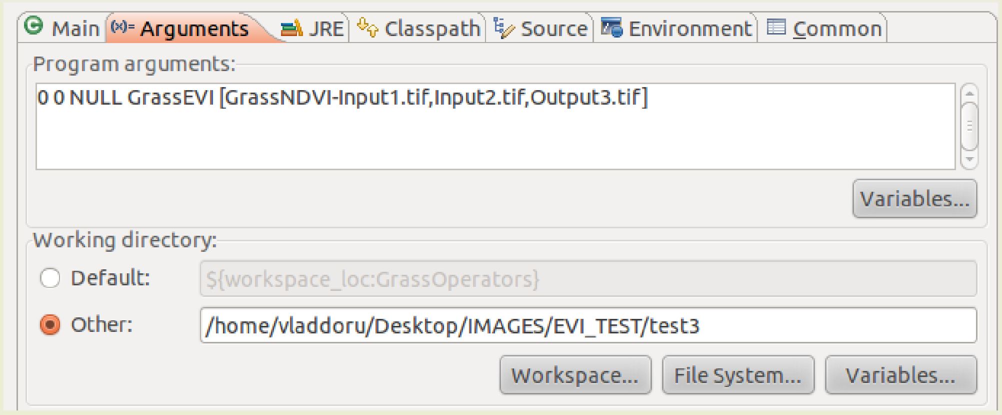
The last thing the user has to do before he can start developing his own code is establish the working directory which must contain all the needed files. If the user decides to use GRASS he will have to include both GRASS program archive and GRASS dependency package.

This section will also present how to test his program using defined input parameters.

<u>**Eclipse**</u>**:**

Run-> **Run Configurations**->**Arguments**

a) Program Arguments
b) Working Directory



Program arguments:

```
0 0 NULL GrassEVI [GrassNDVI-Input1.tif,Input2.tif,Output3.tif]
```

Working directory:

Default: ${workspace_loc:GrassOperators}

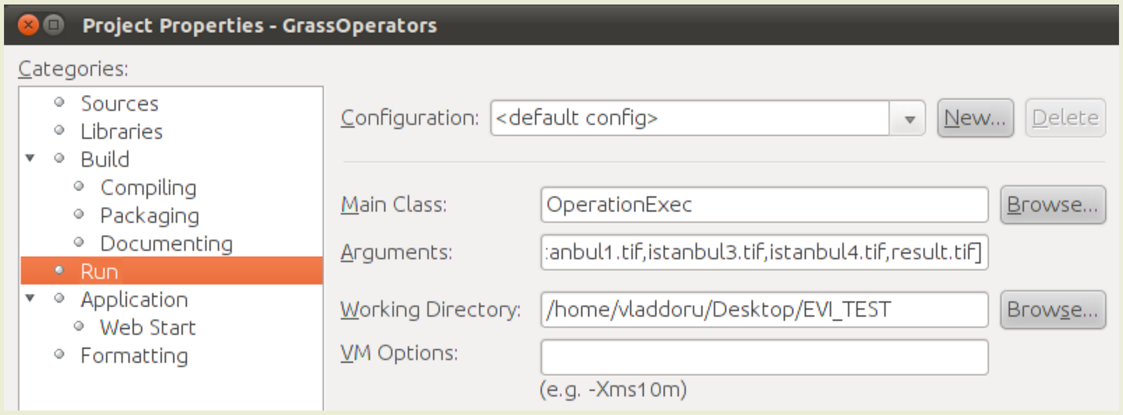
[✓ Other:] /home/vladdoru/Desktop/IMAGES/EVI_TEST/test3

Figure 9) Passing Arguments and Working Directory in Eclipse

<u>Netbeans</u>:

Select Project->Project Properties->Run

- **a)** Arguments
- **b)** Working Directory



Configuration: <default config>

Main Class: OperationExec

Arguments: :anbul1.tif,istanbul3.tif,istanbul4.tif,result.tif]

Working Directory: /home/vladdoru/Desktop/EVI_TEST

VM Options:

Figure 10) Passing Arguments and Working Directory in Netbeans

The Grid program takes a series of 5 arguments, from which it extracts information about how to execute it.

0 0 NULL Grass [GrassNDVI-input1.tif,input2.tif,output.tif]

The first represents whether the operator should be monitored or not on the Grid.

- 0- not monitored
- 1- monitored

This information is handled by gProcess and the user must not concern himself with it anymore. The default value of 0 should be passed when executing locally.

The second parameter represents the identifier of the iPDG sent to the platform. Again the user must not be concerned with it, the value 0 will do.

The third parameter represents an execution name given to the operator. The value of NULL will suffice for a local execution.

Forth is the name of the operator being executed and has no real meaning when it is executed locally. Any value not containing a space will suffice.

The fifth argument is the only one with importance when the operator is not on the Grid. It is surrounded by right brackets '[' ']' and it contains the name of the class implementing the required functionality and the arguments which that class takes.

[**GrassNDVI**-input1.tif,input2.tif,output.tif] : name of class

Note: The name of the executing class is separated from the rest of the parameters by a '-' minus sign.

[GrassNDVI-**input1.tif,input2.tif,output.tif**]: arguments

Note: Each argument is separated by a ',' comma from its neighbors.

Note: Every argument except for the last is considered to be input for the operator.

Note: The last operator must be a file, because if it is not it cannot be communicated over the Grid platform.

3. API Description

In order to create a class, which can be executed using the provided API, the user must extend the class **Operator** and override the method **Execute().** In order to work this class must be part of the **gPOperators** package.

```
package gPOperators;

import java.util.Vector;

public class GrassGeneric extends Operator{
    @Override
    public void Execute(){
        int i=0;
        String inputArg1=this.getParams().get(i);//First Input Argument
        i++;
        String inputArg2=this.getParams().get(i);//Second Input Argument
        …
        String outputArg2=this.getParams().get(i);//Last Output Argument
    }
}
```

To be able to access the parameters sent to a operator the user must access them by calling the method **getParams().get(X);** where **X** is the number of the parameter. This function is defined in class **Operator** and is available to all subclasses extending it.

In order to create a class, which uses GRASS to solve the satellite image processing problems the user must extend the class GrassGeneric, which in its turn extends the class Operator, thus satisfying the before mentioned requirement. The same constraints as before apply.

```
package gPOperators;

import java.util.Vector;

public class GrassNDVI extends GrassGeneric{

    @Override
    public void grassExecute(StringBuilder builder){
        builder.append(…);
        builder.append(…);
       …
    }

    @Override
    public void Execute(){
       try{
              generic("grassExecute");
       }catch(Exception e){…}
    }
}
```

The only method which has to be overridden in order to be able to write GRASS scripts is **grassExecute() .** The script needs to be written within the StringBuilder object passed as input. Each GRASS or BASH command must be followed by a “\n”, in order to signal next line of the command code.

The user is not obliged to extend **Execute()** method as previously stated, only if there are additional operations to be performed outside what GRASS and Linux BASH scripts can offer.

Most often would be the case when the user wants to execute two or more scripts sequentially or in parallel.

4. GRASS Description

In the previous subsection there was a section on how to integrate your operator with GRASS scripts, but there was no mention of how to create an operator using this type of program.

Note: Previous Grass knowledge is required.

As it turns out there are a few limitations inherent within the GRASS executables. The first limitation is the fact that when the user tries to output a georeferenced image using **r.out.gdal** an error occurs (**Segmentation Fault**). The only supported output is GeoTiff.

In order to circumvent this the user may change the Grass distributable, but it is advised not to do so since testing locally differs from executing on the Grid. This facility is for Grass experts only, which are familiar with Grass development and internal workings of it.

To solve the segmentation fault problem it is required to export the file using **r.out.tiff** first and that the extension of the outputted file be **.tif**. Only afterwards is it possible to export using **r.out.gdal** by using the same name.

```
r.in.gdal input=input.tif output=internal_repr1 location=intermediary
 #import Tiff image into location intermediary
g.gisenv set=LOCATION_NAME=intermediary
 #change current location to the one where the input image has been
created
#your code here
r.out.tiff input=internal_repr1 output=file.tif
#export Tiff without geographic reference
r.out.gdal input=internal_repr1 output=file.tif
#overwrite existing Tiff file in order to obtain a georeferenced file
```

4. Examples

The following section will detail two examples, one of creating a simple operator and one which deals with using GRASS as in order to achieve satellite image processing.

1. Adding two numbers

This operator deals with the simple function of adding two numbers.
Inputs: Constant Integer, Constant Integer
Output: File containing the result of the addition.

```
package gPOperators;

import java.util.Vector;

public class GrassNDVI extends Operator{

    @Override
    public void Execute(){
       int i1=Integer.parseInt(this.getParams().get(0));
       int i2=Integer.parseInt(this.getParams().get(1));
       Integer i3=i1+i2;
       Writer output=null;
       //Integers cannot be passed as output, only files.
       File f=new File(this.getParams().get(2));
       try {
             f.createNewFile();
             output=new BufferedWriter(new FileWriter(f));
             output.write(i3.toString());
       } catch (IOException ex) {
             Logger.getLogger(Add2Ints.class.getName()).log(…);
       } finally {
             if(output!=null){
                   try {
                         output.close();
                   } catch (IOException ex) {
                         …;
                   }
             }
       }
    }
}
```

2. Grass NDVI

This operator deals with creating a NDVI cover of the area of a Red and Near Infrared Image

Inputs: Red Image Band, Near Infrared Image Band

Output: NDVI Image

Note: This operator uses the Grass API provided in the distributable.

```
package gPOperators;

public class GrassNDVI extends GrassGeneric{

       @Override
       public void grassExecute(StringBuilder builder){
          builder.append("\n");
          //GRASS Import begin
          builder.append("r.in.gdal input=").append(this.getParams().get(0));
          builder.append(" output=image"+0);
          builder.append(" location=intermediary \n");
          builder.append("g.gisenv set=LOCATION_NAME=intermediary \n");
          //GRASS Import end
          builder.append("r.in.gdal input=").append(this.getParams().get(1));
          builder.append(" output=image"+1).append("\n");
          //Calculation of NDVI Image.
          builder.append("r.mapcalc \"output\"=\"(float(image1-image0)”
          +”/float(image1+image0))\"\n");
          builder.append("r.colors map=output rules=ndvi\n");
          String output=this.getParams().get(2);
          //GRASS Export. Please note the order and limitations.
          builder.append("r.out.tiff input=output output="+output+"\n");
          builder.append("r.out.gdal input=output output="+output+"\n");
       }
}
```

5. Conclusion

The advantage of using this platform for operator submission is much faster than the traditional way of development where the user asks the administrators to integrate the developed program into the ESIP platform.

If a certain operator is not satisfactory, the user may choose to replace it with his/her own version, thus creating a solution which much better copes with his/her everyday needs rather than relying on a general solution offered by the system.