# AiBench 1.9

*Operations Programmer Manual*

October 2006

## Credits

Daniel Glez-Peña.

Florentino Fdez-Riverola.

J. R. Méndez Reboredo.

## Table of Contents

Chapter 1. Basics....2
1.1. What is AiBench?....2
1.2. The AiBench Operation Model....3
1.2.1. The clipboard....4
1.3. The Workbench user interface....5
1.3.1. Dynamic generation of input dialogs....6
1.4. The plugins-based architecture of AiBench....8
Chapter 2. Programming Operations....11
2.1. Defining the operations with Annotations....11
2.1.1. @Operation annotation....11
2.1.2. @Port annotation....11
2.1.3. @Progress annotation....13
2.2. Connecting the operations to the AiBench's CORE....13
2.2.1. Connecting operations....13
2.2.2. Validating the user input....14
2.2.3. Providing progress information....15
2.2.4. Providing views and other GUI-related stuff....16
Chapter 3. Downloading AiBench and building plugins....18
3.1. Creating plugins with Ant....18
3.2. Using Eclipse....19

# Chapter 1. Basics

## 1.1. What is AiBench?

AiBench is a lightweight, non-intrusive, MVC-based Java application framework that eases the **connection**, **execution** and **integration** of operations with well defined input/output. This basic idea provides a powerful programming model to fast develop applications given that:

- The logic can be decoupled from the user interface.
- The interconnection of operations can also be decoupled based in the idea of "experiments".
- The programmer is forced to "think-before-program", easing the code reuse.

The new Operations can be plugged by only copying their .jar files in a specific directory and they will be available with just restarting the program.

AiBench was created focused in the easiness of building new operations. To achieve this objective, the design was made following these principles:

1. **Default-driven**. The programmer of Operations should see them running with the minimum amount of code. The framework should provide smart defaults for each option that can be fine-tuned in the future.
2. **Design independent of the problem**. AiBench was made inside a research group focused in the data-mining/AI field, but there is not any concept related with that field in the classes, interfaces or annotations of this framework. AiBench is, in fact, data-type agnostic, the programmer provides his data-types through his own classes.

The applications of AiBench are not limited, but it specially fits well in the data-mining/AI field, because every day there are new operations and data-types that need to be tested and integrated with the existing ones. In the next sections, you will find how to create operations for this simple, but powerful framework.

## 1.2. The AiBench Operation Model

AiBench defines an **Operation** in the following way:

**Operation**: *Unit of logic defined through a set of* ***ports****. A port is a point where some data (such as parameters, working data, etc.) can be received, produced or both. So, the direction of a port is one of: INPUT, OUTPUT or INPUT/OUTPUT.*

An Operation is a Java class where each port is associated with a method that:

- Receives the incoming data through a parameter (if the port is INPUT).
- Produces output data with the return value (if the port is OUPUT).
- Both (if the port is INPUT/OUPUT).

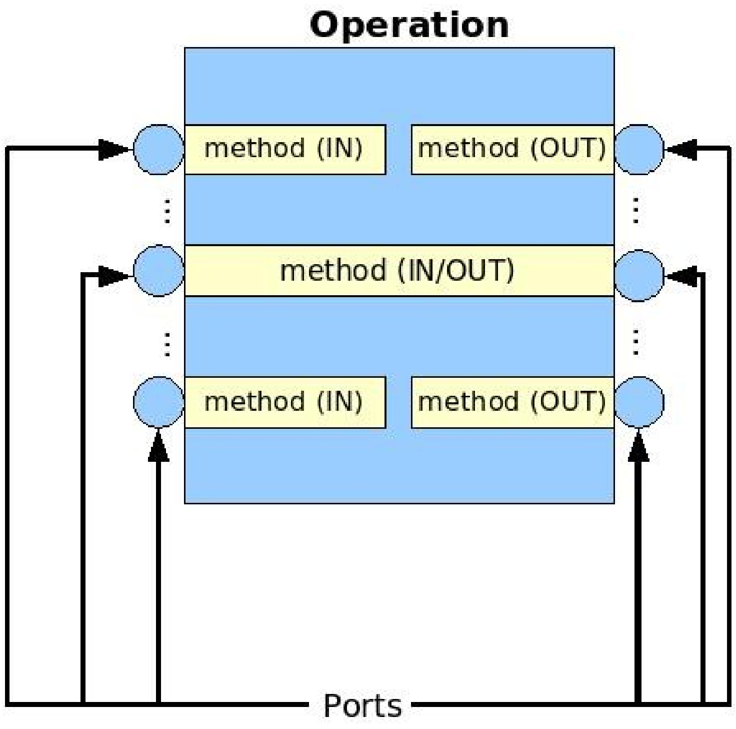


**Figure 1: An AiBench Operation**

The ports are defined with Java Annotations. A simple Operation file could look like this:

```
@Operation(description="this operation adds two numbers")
public class Sum{
	private int x,y;

	@Port(direction=Direction.INPUT, name="x param")
	public void setX(int x){
		this.x = x;
	}

	@Port(direction=Direction.INPUT, name="y param")
	public void setY(int y){
		this.y = y;
	}
	@Port(direction=Direction.OUTPUT)
	public int sum(){
		return this.x + this.y;
	}
}
```

The example defines an Operation with three ports: the first two are INPUT ports and the last one is an OUTPUT port.

The idea is to isolate the logic and only the logic in Operations. The AiBench Core will receive the user requests and start the execution of the Operation. The ports will be called with the correct parameters and the results will be saved (see clipboard), but the programmer doesn't need to do any of these tasks.

### 1.2.1. The clipboard

The "clipboard" is the mechanism that allows the integration between operations. It works in the following way: all the results generated through the execution of Operations will be saved in the clipboard, which is a structure that keeps these data objects classified by their Java classes. This structure allows the user to forward the data generated with an operation to the input of the next one.

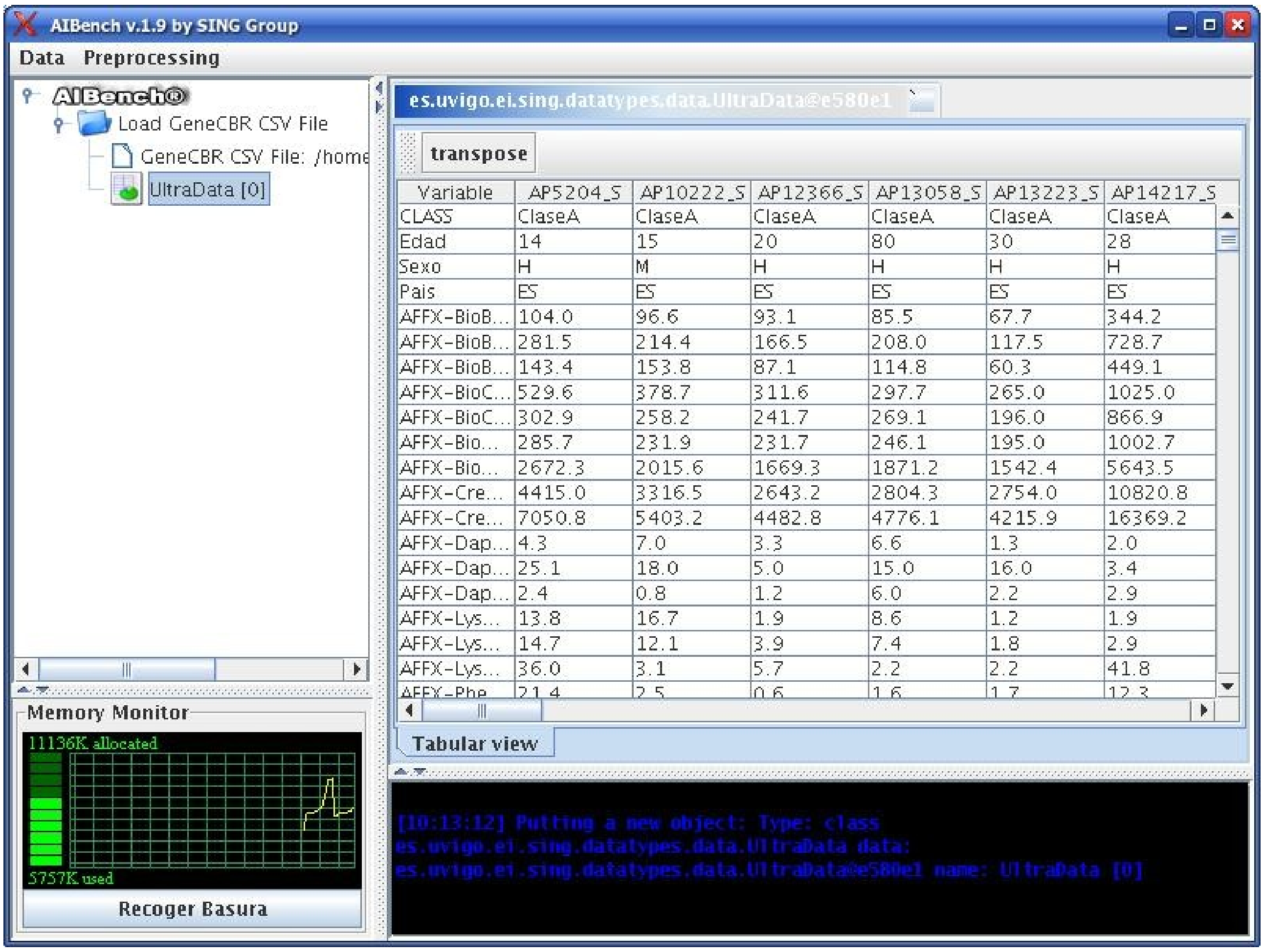
## 1.3. The Workbench user interface

AiBench provides a Java Swing GUI (*Graphical User Interface*), called Workbench, that allows the user to request the execution of operations. The main features of the Workbench are:

- **Deployment of the available Operations in menus**. The Operations also define a logical path such as */load/csv/loadCSVFile* used by the Workbench to create a menu hierarchy following those logical paths.
- **Dynamic generation of input dialogs**. When the user requests the execution of a given Operation, the Workbench generates an input dialog reflecting the input ports defined in that Operation. Depending of the data-type of each port, the control showed may change (see Dynamic generation of input dialogs).
- **User's input validation**. It uses the validating method provided in the operation (if there is one) to stop the user if the validation didn't succeed (see Validating the user input).
- **Monitoring the process of the Operation's execution**. The more monitoring information the

Operation provides, the more information will be displayed (see Providing progress information).

- **Display the results of an Operation**. The Workbench provides a default View of the results, but you can provide more sophisticated custom components associated with a Data-type to display its information.



| Variable | AP5204_S | AP10222_S | AP12366_S | AP13058_S | AP13223_S | AP14217_S |
|---|---|---|---|---|---|---|
| CLASS | ClaseA | ClaseA | ClaseA | ClaseA | ClaseA | ClaseA |
| Edad | 14 | 15 | 20 | 80 | 30 | 28 |
| Sexo | H | M | H | H | H | H |
| Pais | ES | ES | ES | ES | ES | ES |
| AFFX-BioB... | 104.0 | 96.6 | 93.1 | 85.5 | 67.7 | 344.2 |
| AFFX-BioB... | 281.5 | 214.4 | 166.5 | 208.0 | 117.5 | 728.7 |
| AFFX-BioB... | 143.4 | 153.8 | 87.1 | 114.8 | 60.3 | 449.1 |
| AFFX-BioC... | 529.6 | 378.7 | 311.6 | 297.7 | 265.0 | 1025.0 |
| AFFX-BioC... | 302.9 | 258.2 | 241.7 | 269.1 | 196.0 | 866.9 |
| AFFX-Bio... | 285.7 | 231.9 | 231.7 | 246.1 | 195.0 | 1002.7 |
| AFFX-Bio... | 2672.3 | 2015.6 | 1669.3 | 1871.2 | 1542.4 | 5643.5 |
| AFFX-Cre... | 4415.0 | 3316.5 | 2643.2 | 2804.3 | 2754.0 | 10820.8 |
| AFFX-Cre... | 7050.8 | 5403.2 | 4482.8 | 4776.1 | 4215.9 | 16369.2 |
| AFFX-Dap... | 4.3 | 7.0 | 3.3 | 6.6 | 1.3 | 2.0 |
| AFFX-Dap... | 25.1 | 18.0 | 5.0 | 15.0 | 16.0 | 3.4 |
| AFFX-Dap... | 2.4 | 0.8 | 1.2 | 6.0 | 2.2 | 2.9 |
| AFFX-Lys... | 13.8 | 16.7 | 1.9 | 8.6 | 1.2 | 1.9 |
| AFFX-Lys... | 14.7 | 12.1 | 3.9 | 7.4 | 1.8 | 2.9 |
| AFFX-Lys... | 36.0 | 3.1 | 5.7 | 2.2 | 2.2 | 41.8 |
| AFFX-Phe | 21.4 | 2.5 | 0.6 | 1.6 | 1.7 | 12.3 |

```
[10:13:12] Putting a new object: Type: class
es.uvigo.ei.sing.datatypes.data.UltraData data:
es.uvigo.ei.sing.datatypes.data.UltraData@e580e1 name: UltraData [0]
```

**Figure 2: Workbench Snapshot**

The Figure 2 shows a snapshot of the Workbench. On the left side, you can see an tree where the operations and results are showed. On the right side, you can see the components rendering results. In the bottom-left zone there is a memory monitor and in the bottom-right zone there is a log window capturing the Log4J output (http://logging.apache.org/log4j/docs).

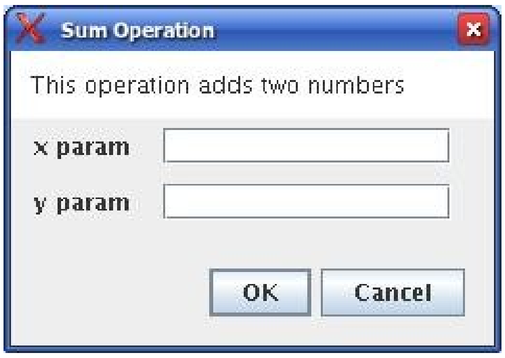
## 1.3.1. Dynamic generation of input dialogs

Currently the dynamic generation of dialogs, maps Data-types with controls following the policy of the following table.

| Data type | Control used |
|---|---|
| Primitive Type (int, float, double, char) | Text field |

| | |
|---|---|
| Boolean | Check-box |
| Enum type (Java 1.5) | Radio button with each option |
| A class with a constructor with one parameter of type String (primitive wrapper, String...) | Text field |
| java.io.File | Text field with a “Find...” button that brings an file chooser dialog |
| Other class (only can take the value from the CLIPBOARD) | Combo box with the instances of the same class available in the clipboard |
| Array | The control inferred with the bellow criteria, plus a list and an “add” button to put elements in the array |

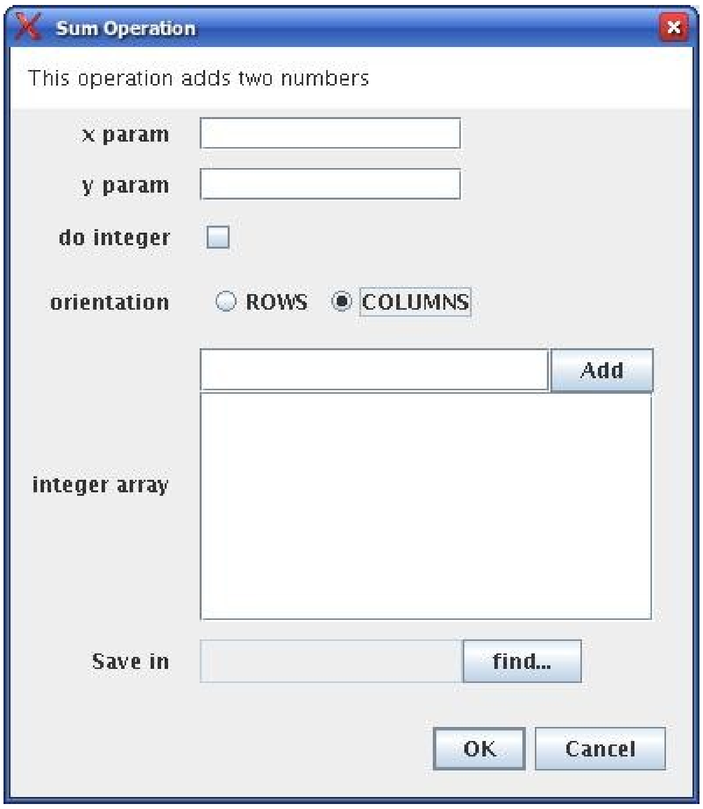
With the *Sum* example showed before, the Workbench generates the input dialog like this one:



This operation adds two numbers

**Figure 3: A dynamic input dialog**

The generation of dialogs is very powerful and can generate complex dialogs like this one:


Sum Operation

This operation adds two numbers

x param

y param

do integer

orientation ROWS [✓ COLUMNS]

integer array

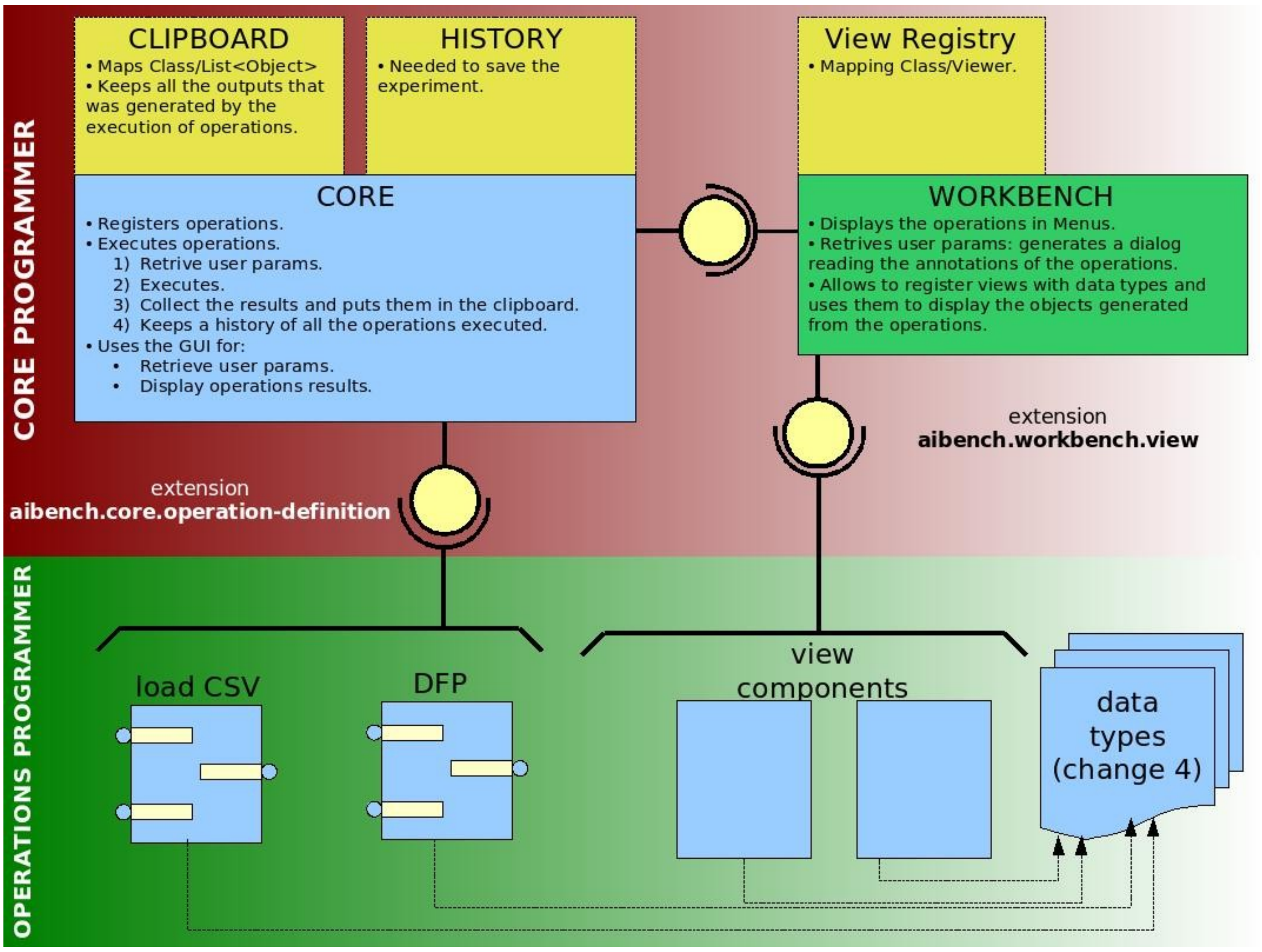
Save in

**Figure 4: Another dynamic dialog**

## 1.4. The plugins-based architecture of AiBench

Figure 5 shows an overview of the AiBench framework. The green zone indicates that the *Operations programmer* role has to develop his own Operations and his own Data-types. The red zone shows that the *core programmer* role maintains the Core and the Workbench. The scope of this document is related with the *Operations programmer*.

In summary, the *Operations programmer* builds this type of artefacts:


CORE PROGRAMMER
CLIPBOARD
• Maps Class/List<Object>
• Keeps all the outputs that was generated by the execution of operations.
HISTORY
• Needed to save the experiment.
View Registry
• Mapping Class/Viewer.
CORE
• Registers operations.
• Executes operations.
1) Retrive user params.
2) Executes.
3) Collect the results and puts them in the clipboard.
4) Keeps a history of all the operations executed.
• Uses the GUI for:
• Retrieve user params.
• Display operations results.
WORKBENCH
• Displays the operations in Menus.
• Retrives user params: generates a dialog reading the annotations of the operations.
• Allows to register views with data types and uses them to display the objects generated from the operations.
extension
aibench.workbench.view
extension
aibench.core.operation-definition
OPERATIONS PROGRAMMER
load CSV
DFP
view components
data types (change 4)
**Figure 5: AiBench's architecture overview**

1. **Operations**. The units of logic.

2. **Data-types**. Normal Java classes used as input and output of the Operations.

3. **Views**. Classes that inherits from *JComponent* and are used to display the Data-types inside the Workbench.

AiBench uses a **plugin engine** behind the scenes to provide advanced capabilities such as dynamic discover and load of new Operations by only restarting the application. You need to know the basics of this engine in order to develop Operations for AiBench.

A plugin engine is used to create applications based in software modules, called plugins. This paradigm follows these rules:

- A plugin is a set of classes, isolated (by default) from the rest classes belonging to other plugins.

- A plugin can define **extension points** (the yellow circles in Figure 5). An extension point is a place where other plugins can be connected to and then discovered and used at run-time. Optionally, an extension point could establish a Java interface that must be implemented by the plugins connected to it.

- A plugin can use the classes from other plugin only if a **dependency** from the first to the second has been defined.

AiBench contains two basic plugins: the **CORE** and the **WORKBENCH**, communicated internally by an extension point (the top one in the Figure 5), but this is out of the scope of this document.

Also the CORE implements the extensible **AiBench Operation Model** by the definition of an extension point called “aibench.core.operation-definition”. To add Operations to AiBench, you have to put them in a plugin and connect it to this extension point. There is not any required interface to implement.

The WORKBENCH implements the **Workbench** user interface and also defines an extension point called “aibench.core.workbench.view”. To register a view component associated with a Data-type, you have to put your component in a plugin and connect it to this extension point. There is not any required interface to implement.

The user Data-types have to be placed inside a plugin, but this plugin doesn't need to be connected to any extension point. (Please note that if some of your Operations use these Data-types and reside in other plugins, they must depend on this plugin).

You can put your Operations, Views and Data-types in one, two or more separated plugins, based on your own design decision. The only rules that have to be followed are:

1. If there are Operations and/or Views in a plugin, this plugin must be connected to the properly extension points as it was explained before.
2. If the Views and/or Operations in a plugin make use of the Data-types located in other plugin, this plugin must depend on the Data-types plugin.

The connection and dependency between plugins is made through the **plugin.xml** file present in every plugin. Here is an example of that file.

```
<plugin start="false">
   <uid>geneCBR.preprocessing.dfp</uid>
   <name>GeneCBR's DFP</name>
   <version>1.0.0</version>

<!-- DEPENDENCIES: This plugin depends in other that define some data-types needed by the
operations present in this plugin -->
   <dependencies>
              <dependency uid="sing.datatypes"/>
   </dependencies>
```

```
<!-- EXTENSIONS: The extensions that this plugin in connected to -->
   <extensions>


       <!-- EXTESION 1. Each Operation is plugged in the CORE with its aibench.core.operation-
       definition extension -->
       <extension
               uid="aibench.core"
               name="aibench.core.operation-definition"
               class="es.uvigo.ei.sing.geneCBR.dfp.DFPOperation">


               <!-- Additional operation info -->
               <operation-description
                               name="Discriminant Fuzzy Patterns Filtering"
                               uid= "geneCBR.preprocessing.dfp"
                               path="3@Preprocessing/1@Feature selection/"
               />
       </extension>


       <!--EXTENSION 2. The Graphical-related information is given extending the Workbench     with
       its aibench.workbench.view extension -->
               <extension
                       uid="aibench.workbench"
                       name="aibench.workbench.view" >
                       <icon-operation operation="geneCBR.preprocessing.dfp"
                               icon="icons/patterns.png"/>
                       <view name="Feature Selection Results View"
datatype="es.uvigo.ei.sing.datatypes.featureselection.FeatureSelectionResults"
class="es.uvigo.ei.sing.datatypes.gui.FeatureSelectionResultsViewer"/>
                       <icon-datatype
datatype="es.uvigo.ei.sing.jcbr.casebase.ExemplarsModelCaseBase" icon="icons/cbase.gif"/>


               </extension>


   </extensions>
</plugin>
```

In the example, you can see that:

1. The plugin has an **unique identifier** (uid), that is useful to reference this plugin from others.
2. The plugin defines its **dependencies** on others with the <dependencies> and <dependency> tags.
3. The plugin is connected to **extension points** using the <extensions> and <extension> tags.

The last point is defined in more detail in the next chapters.

# Chapter 2. Programming Operations

## 2.1. Defining the operations with Annotations

In order to create a new Operation, you need to define a class, annotate it and connect it to AiBench through the plugin.xml file that must be present in you plugins in the path described in the chapter Downloading AiBench and building plugins. The annotations needed to create your own operations are described bellow.

### 2.1.1. @Operation annotation

This annotation is a class annotation, that must be present in all Operation classes. The attributes of this annotation are showed in the following table.

| Attribute name | Type | Description | Default |
|---|---|---|---|
| name | String | The name of the Operation. This will be used, for example, in the menus. This value also can be established in the plugin.xml | <empty string> |
| description | String | A briefly description of the Operation. This text appears, for example, in the header of the dynamic dialogs generated by the Workbench. | <empty string> |

### 2.1.2. @Port annotation

With this annotation you can define all the things related with a port. This annotation appears before the method that will be associated with the port. The attributes of this annotation are showed in the following table.

| Attribute name | Type | Description | Default |
|---|---|---|---|
| name | String | The name of the port. This text will be used, for example, to render a label in the dynamic dialogs. | <empty string> |
| description | String | A briefly description of the port. This text appears, for example, near the respective control in the dynamic dialogs. | <empty string> |
| direction | Direction (enumerated type) | The data flow direction. One of:<br>INPUT<br>OUTPUT<br>BOTH | Direction.BOTH |
| defaultValue | String | The default value of the port (must be | <empty string> |

| | | | |
|---|---|---|---|
| | | an INPUT or BOTH port). This value will be used, for example, to populate the input dialog with default values. The value cannot be used to set CLIPBOARD values, only primitives, and for classes with a String constructor. | |
| validateMethod | String | The name with an existing method in the same class that receives the same parameter as the method of this @Port. This validate method should throw an exception if the input is not valid, and do nothing otherwise. (see Validating the user input) | <empty string> |
| order | int | The ports will be called by the CORE in this order. If two ports has the same order, an INPUT/BOTH port is called before an OUTPUT port. If the two ports have the INPUT/BOTH direction, there is not any determined behaviour. | -1 |

### 2.1.3. @Progress annotation

This optional annotation can be used to give a Java bean that keeps in its properties the information related with the actual progress of the Operation. The Operation should call the setter methods of the bean during its process, and the Workbench, reads them in “real-time” showing them to the user. The annotation has no attributes, it should be used with the method that returns that bean. An example of this annotation can be found in Providing progress information.

## 2.2. Plugging operations to the AiBench's CORE

This section shows real examples with the use of the annotations and the configuration of the plugin.xml files to connect Operations to the AiBench Core and how to provide GUI components to the Workbench to render your Data-types.

### 2.2.1. Connecting operations

As it was explained before, an Operation must be connected to the core's “aibench.core.operation-definition” extension. This can be done in the plugin.xml file of the plugin where the Operation class resides. For example:

```
<extension
	uid="aibench.core"
	name="aibench.core.operation-definition"
	class="es.uvigo.ei.sing.geneCBR.dfp.DFPOperation">
```

```
    <!-- Additional operation info -->
    <operation-description
        name="Discriminant Fuzzy Patterns Filtering"
        uid= "geneCBR.preprocessing.dfp"
        path="3@Preprocessing/1@Feature selection/"
    />
</extension>
```

The relevant things are this:

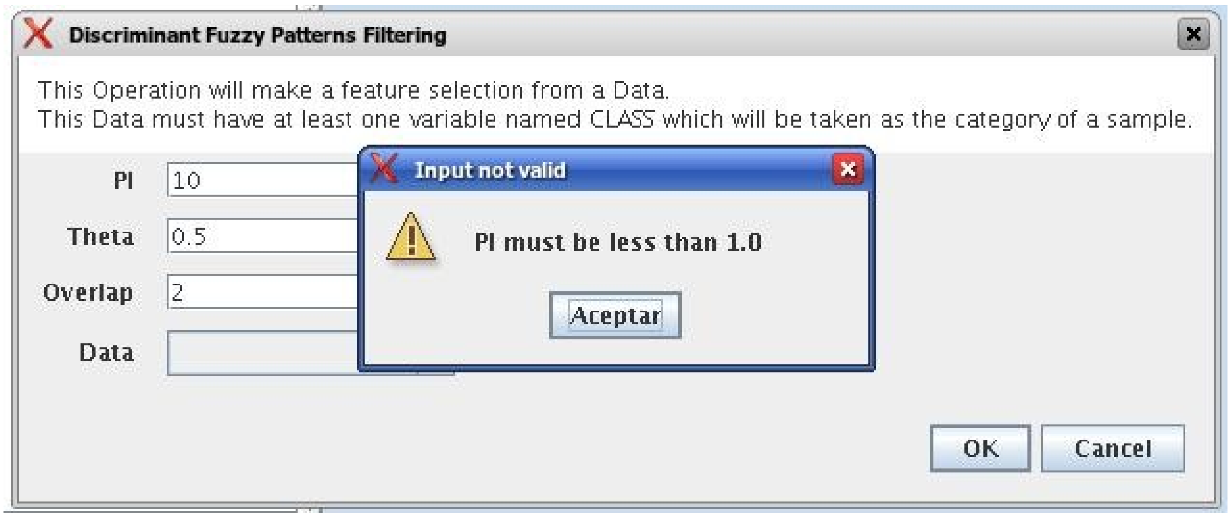
- **class**. Is the class of the Operation.
- **<operation-description>** tag. Gives more information about the operation.
    - **name**. The name of the operation.
    - **uid**. An identifier useful to reference this operation from other places.
    - **path**. The location in the user interface where the user can find this operation (think in a menu). This path is defined like a file-system path, but each item can be preceded with a number@, that establishes a desired order of the option relative to others. For example: if a operation is in *@1Data/* and other in *@2Preprocessing/*, the Workbench will create two menus in its main window, placing Data before of Preprocessing.

### 2.2.2. Validating the user input

To validate the input of a port, you can use the *validateMethod* attribute of the @Port annotation as it was explained before. Here you can see a simple example:

```
... //inside the code of an operation
@Port(name="PI", direction=Direction.INPUT, description="ex: 0.9", defaultValue="0.9",
validateMethod="validatePI")
public void setPI(float pi){
        this.pi=pi;
}
public void validatePI(float pi){
        if (pi>1.0)
            throw new IllegalArgumentException("PI must be less than 1.0");
}
...
```

The Workbench GUI uses this information to guide the user to the correct parameters. Figure 6 shows the *validatePI* method in action.



This Operation will make a feature selection from a Data.
This Data must have at least one variable named CLASS which will be taken as the category of a sample.

PI 10
Theta 0.5
Overlap 2
Data

PI must be less than 1.0

**Figure 6: Validating user's input**

### 2.2.3. Providing progress information

To show detailed information about the progress of a given Operation you can use the @Progress annotation, as it was explained before. Here you can see a simple example:

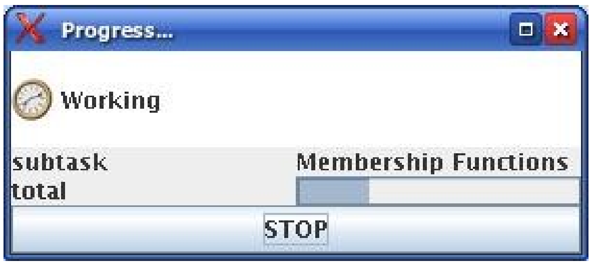
```
@Progress
public DFPStatus getStatus(){
		return this.status;
}
@Port(direction=Direction.OUTPUT)
public void process(){
	//in the process we make changes in the progress bean
	this.status.setSubtask(“doing this”);
	//...
	this.status.setSubtask(“doing that”);
}
```

*DFPOperation* is an user-defined Java bean which its code could look like this:

```
public class DFPStatus {
	private String subtask;
	private float total=0.0f;

	public String getSubtask() { return this.subtask; }
	public void setSubtask(String subtask) { this.subtask = subtask; }

	public float getTotal() { return this.total; }
	public void setTotal(float total) { this.total = total; }
}
```

The Workbench GUI will show in the progress monitor all the properties of the bean with text labels. Please note, that the float properties will be managed in a special manner: they will be displayed as progress bars where their position is empty if the float value is 0.0 or less, and full if the float value is 1.0 or greater.

Figure 7 shows a progress monitor that displays a bean with two properties: one String called "subtask" and one float called "total".



subtask Membership Functions
total

**Figure 7: Progress monitor**

## 2.2.4. Providing views and other GUI-related stuff

As it was explained before, you can connect Views to show your Data-types instances by extending the "aibench.workbench.view" extension. This can be done in the plugin.xml file of the plugin where the components classes resides. You can also give icons for the operations and Data-types. For example:

```
<extension uid="aibench.workbench" name="aibench.workbench.view" >


      <view name="Feature Selection Results View"
      datatype="es.uvigo.ei.sing.datatypes.featureselection.FeatureSelectionResults"
      class="es.uvigo.ei.sing.datatypes.gui.FeatureSelectionResultsViewer"/>


      <icon-datatype datatype="es.uvigo.ei.sing.jcbr.casebase.ExemplarsModelCaseBase"
      icon="icons/cbase.gif"/>


      <icon-operation operation="geneCBR.preprocessing.dfp" icon="icons/patterns.png"/>


</extension>
```

The code in the example does this:

Defines a View which is defined with the **<view>** tag. This tag must provide:

- **name**: A symbolic name.
- **data-type**. The class of the data-type that this View can render.
- **class**: The class of the visual component. It must inherit from JComponent. At run-time, when an instance of the Data-type is created, the Workbench will create an instance of the component. The data instance is passed to the component in one of these ways:
  - If one field of the component class is annotated with the @Data annotation. In this case the Workbench will look for the setter method and use it to inject the data instance.
  - If @Data is not present, the Workbench will look for a constructor with an only one parameter that must be of the same class (or superclass) of the data-type.

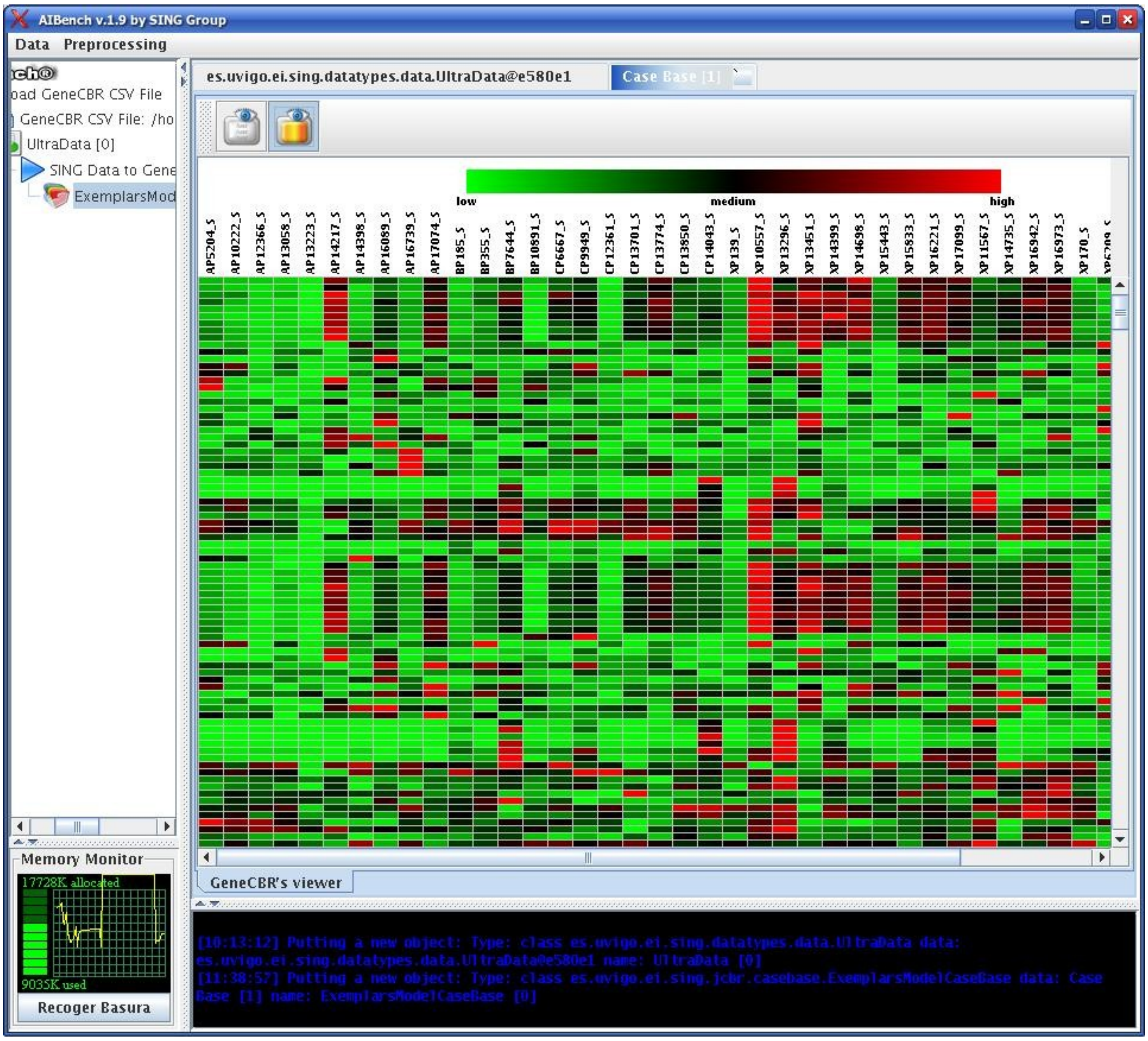
Defines one icon for a Data-type. This can be done with the **<icon-datatype>** tag which has these attributes:

- **datatype**. The class of the data-type.
- **icon**. A path inside the plugin's .jar pointing to the icon's image.

Defines one icon for an Operation. This can be done with the **<icon-operation>** tag which has these attributes:

- **operation**. The uid of the operation.
- **icon**. A path inside the plugin's .jar pointing to the icon's image.

Figure 8 shows an sophisticated View used to display a Data-type.



```
[10:13:12] Putting a new object: Type: class es.uvigo.ei.sing.datatypes.data.UltraData data:
es.uvigo.ei.sing.datatypes.data.UltraData@e580e1 name: UltraData [0]
[11:38:57] Putting a new object: Type: class es.uvigo.ei.sing.jcbr.casebase.ExemplarsModelCaseBase data: Case
Base [1] name: ExemplarsModelCaseBase [0]
```

**Figure 8: An example of a sophisticated View**

# Chapter 3. Downloading AiBench and building plugins

To develop new plugins for AiBench, you have to download the Aibench-sdk-version.zip and then you can choose one of these ways:

1. Create the plugins using the build.xml file with Ant (http://ant.apache.org).
2. Use Eclipse.

## 3.1. Creating plugins with Ant

If you have chosen the first option, you need to know the directory tree. The green folders are those you have to create with each new plugin. The blue ones are generated when you run Ant.

| | |
|---|---|
| / | |
| ○ run.bat. | Script to start AiBench in Windows platforms. |
| ○ run.sh. | Script to start AiBench in Unix platforms. |
| ○ build.xml. | Ant script to build the project. |
| ● /lib/ | .jar files with a distribution to boot Aibench |
| ○ aibench.jar | |
| ○ otherjars.jar | |
| ● /plugins_src | Source code of plugins. |
| ○ **/yourpluginfolder/** | Source code of each new plugin. |
| ■ **/your/package/yourfile.java** | A Java package with Java files. |
| ■ **plugin.xml** | Your plugin descriptor. |
| ● /plugins_bin | Plugins .jar and .class files |
| ○ core.jar | AiBench CORE plugin (mandatory) |
| ○ workbench.jar | AiBench Workbench plugin (mandatory) |
| ○ otherplugin.jar | Other plugins that you want to use (optional) |
| ○ **/yourplugin/** | Your compiled plugin. |

To create a new plugin you need to follow these steps:

1. Create a folder under /plugins_src, say */yourplugin*.
2. Code your Java files in that folder.
3. Create your plugin.xml in the root of this new folder.
4. Run Ant. The default task will compile your .java files under your source folder and put them

in the */plugins_bin/yourplugin/* folder. You should see this output:

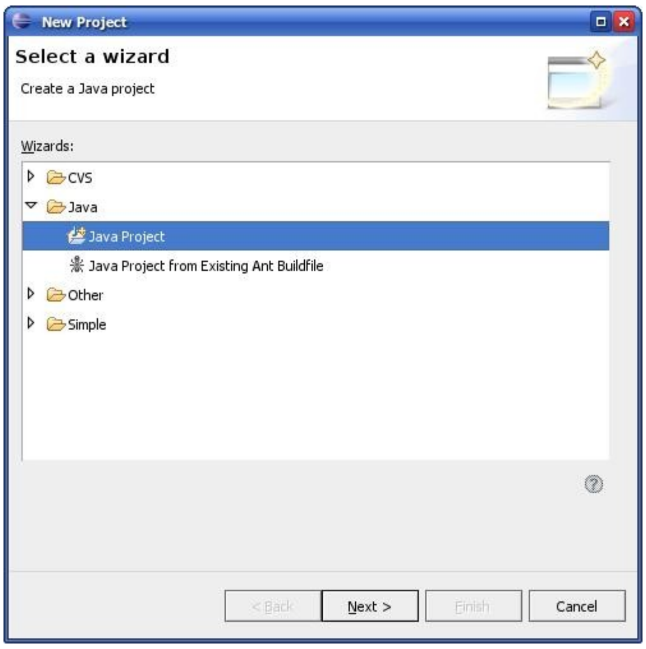
```
Buildfile: build.xml


compile:
    [mkdir] Created dir: /home/user/AiBench_sdk/plugins_bin/yourplugin
    [javac] Compiling 1 source file to /home/user/AiBench_sdk/plugins_bin/yourplugin
     [copy] Copying 1 file to /home/lipido/workspace/AiBench_sdk/plugins_bin/yourplugin


BUILD SUCCESSFUL
Total time: 1 second
```

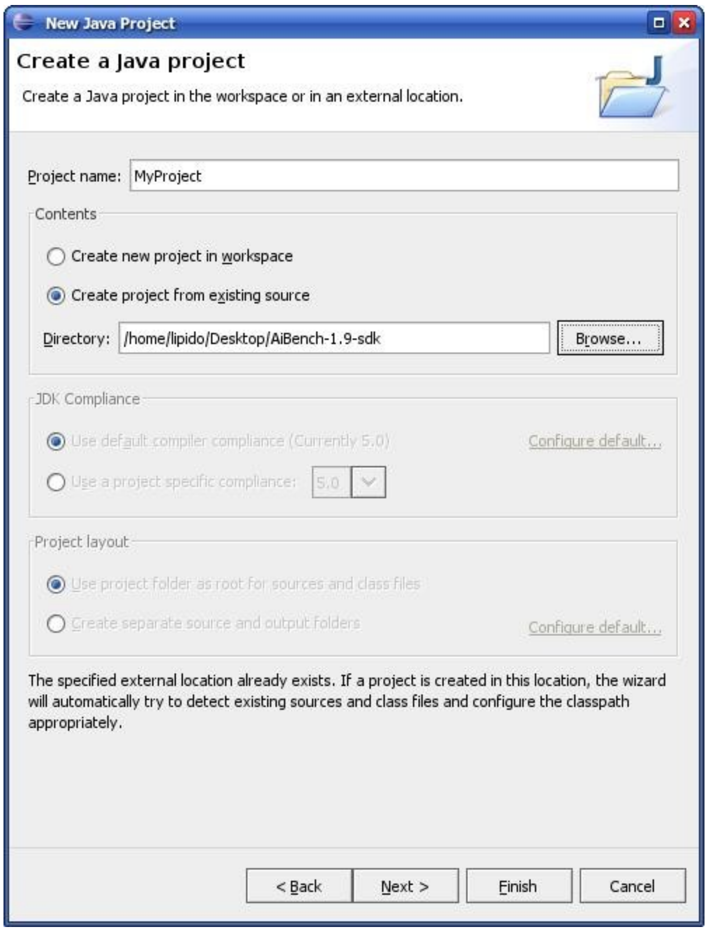
5. Run AiBench with run.sh (Unix) or run.bat (Windows).

## 3.2. Using Eclipse

If you want to use Eclipse to create your plugins, you need to follow these steps:

1. Create a new Java Project. Give a name for it, and **select the Aibench folder** as the existing folder for the project.





New Java Project

Create a Java project

Create a Java project in the workspace or in an external location.

Project name: MyProject

Contents

- ○ Create new project in workspace
- ● Create project from existing source

Directory: /home/lipido/Desktop/AiBench-1.9-sdk

JDK Compliance

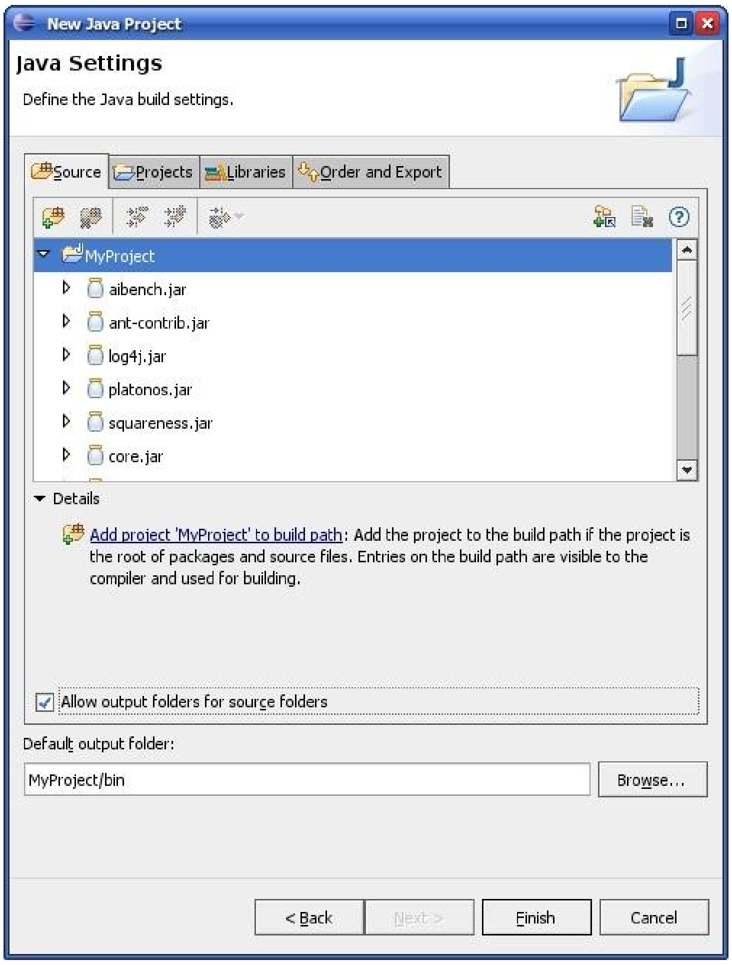
- ● Use default compiler compliance (Currently 5.0)
- ○ Use a project specific compliance: 5.0

Project layout

- ● Use project folder as root for sources and class files
- ○ Create separate source and output folders

The specified external location already exists. If a project is created in this location, the wizard will automatically try to detect existing sources and class files and configure the classpath appropriately.

2. Check the option **Allow output folders for source folders.**


New Java Project

Java Settings

Define the Java build settings.

Source | Projects | Libraries | Order and Export

- MyProject
  - aibench.jar
  - ant-contrib.jar
  - log4j.jar
  - platonos.jar
  - squareness.jar
  - core.jar

Details

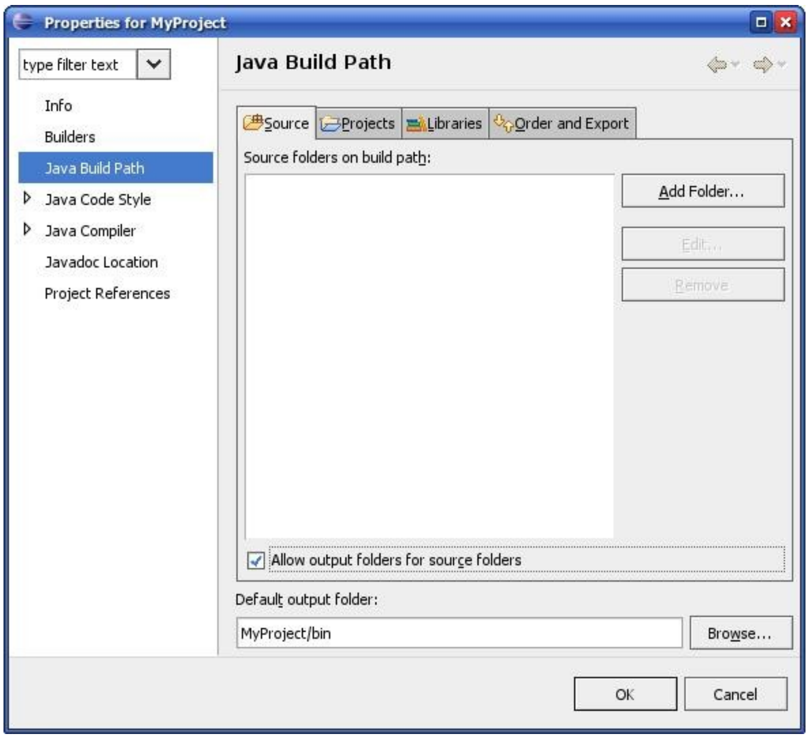
Add project 'MyProject' to build path: Add the project to the build path if the project is the root of packages and source files. Entries on the build path are visible to the compiler and used for building.

[✓] Allow output folders for source folders

Default output folder:

MyProject/bin

3. Now, you need to create the source folder for the new plugin. Go to the project properties, under “Java Build Path”, “Source” tab. Check the option **Allow output folders for source folders.**


Properties for MyProject

type filter text

- Info
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Javadoc Location
- Project References

Java Build Path

Source | Projects | Libraries | Order and Export

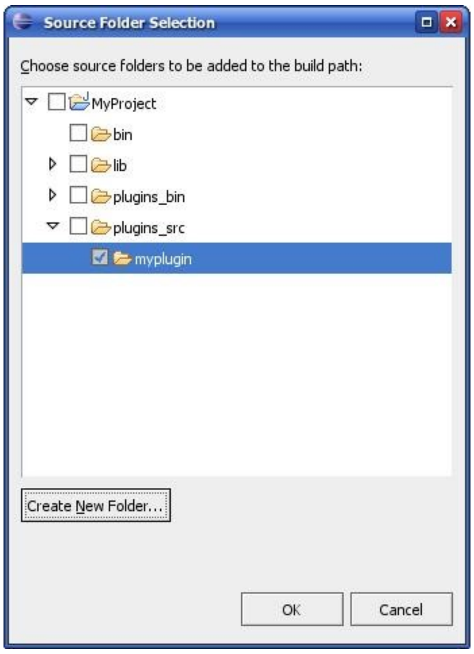
Source folders on build path:

[✓] Allow output folders for source folders

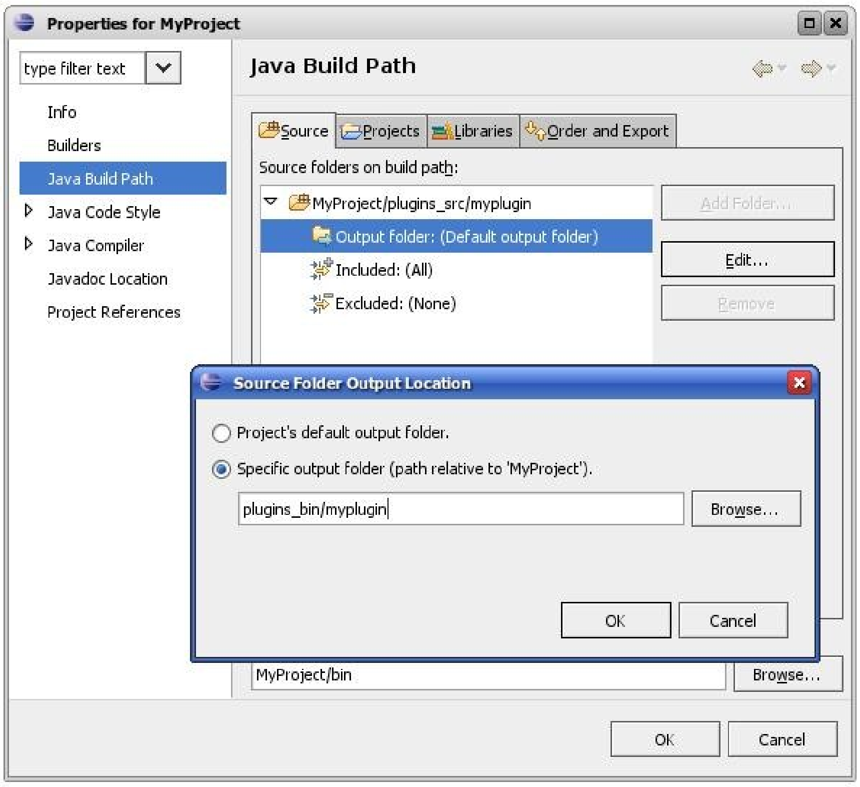
Default output folder:

MyProject/bin

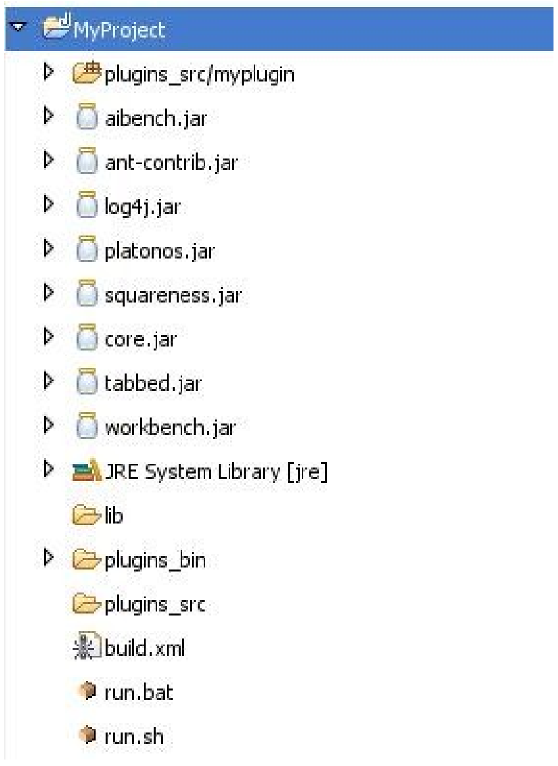
4. Add a new source folder for your plugin, say *myplugin* and click OK.



5. Change the output folder of the new source folder. Specify the name *plugins_bin/myplugin.*



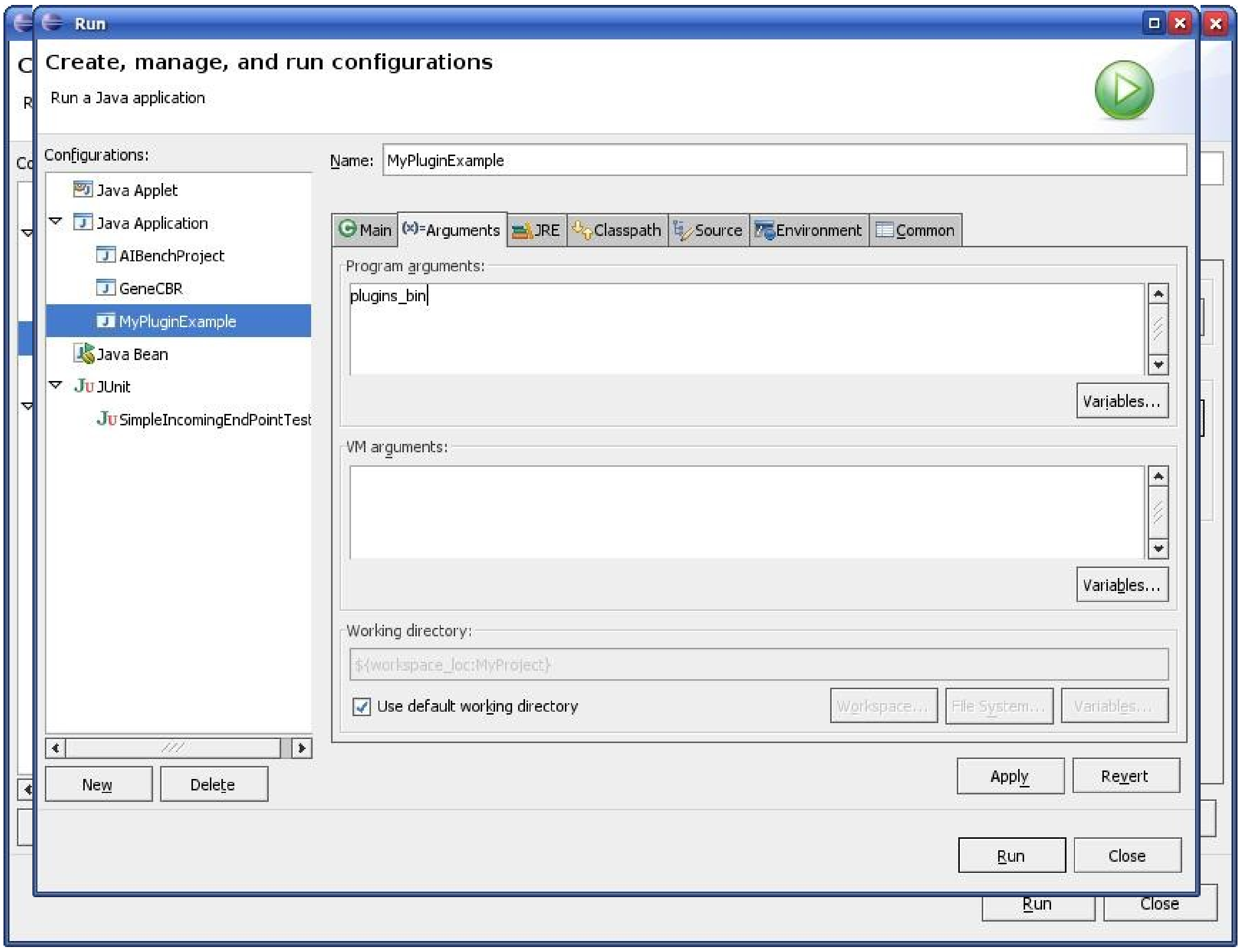
6. Your project tree should look similar to this:



7. Now you are ready to start coding your plugin. Remember, you need at least your plugin.xml file.

8. In order to launch AiBench, you need to create a **Launch Configuration,** going into Run... and creating a new configuration for a Java Application. In the "Main" tab, check **Include libraries when searching for a main class** and select the class *es.uvigo.ei.aibench.Launcher*.

9. In the "Arguments" tab add a new parameter which is where all the plugin live, say *plugins_bin*. Finally, click "Run". AiBench will start and always will read your changes in your code between launches.



Create, manage, and run configurations

Run a Java application

Name: MyPluginExample

Program arguments:

```
plugins_bin
```

VM arguments:

Working directory:

${workspace_loc:MyProject}

☑ Use default working directory