Tutorial for plugin developers

Introduction

This guide is a beginners guide based on how to develop a plugin in order to apply new operations on metabolic systems using the MetaPlab software.
This is especially dedicated to people who are interested in biological systems and want to understand the functioning of the living cell through mathematical models and computational tools.

Essentially the theory on which these models are based on consists of membranes and multisets rewriting, in particular MP systems' dynamic is based on mass partition principle which defines the transformation rate of object populations according to some chemical laws.

The structure of MetaPlab is based on **extensible set of plugins**, written by Java programming language, and it was thought for solving specific tasks in the framework of MP systems, such as parameters estimation for regulative mechanisms of biological networks, simulation, visualization, graphical and statistical curve analysis, importation of biological networks from on-line databases, and hopefully other aspects which would result to be relevant for further investigations.

How to download MetaPlab

In order to develop a new plugin for processing biological models, the developer has firstly to download the source code from the **Download** section of MetaPlab site: **http://mplab.sci.univr.it**. Downloaded file is a zip archive containing all the source codes of MetaPlab. The used programming language is Java, version 1.6. MetaPlab development group usually adopt NetBeans IDE (version 5.5. and subsequent) as a development environment.

Some useful classes to know

Before starting to create a new plugin, some fundamental classes have to be known to understand the software architecture (Figure 1, 2, 3).

MPStoreExt package:

- *MPStoreExt*: it contains all the data related to an MP system, such as substances, parameters, reaction, regulation functions and constants. All these elements are contained in a MembraneExt object (Figure 3). MPStoreExt ensures data compatibility among all the modules of MetaPlab. It implements the *Cloneable* interface and the field called *extra* leaves the door open to every extension of the MP systems framework and also to other computational models, in order to make MetaPlab a shared platform (Figure 1, 2).

- *MembraneMPExt*: it is a class containing membrane's *name* and *description*, model's *time unit* and *mole unit*, and four *vectors*, collecting, respectively, membrane's substances, reactions, fluxes and parameters. It implements the *Cloneable* interface (Figure 3);

- *SubstanceExt*: it is a class that describes a substance in terms of its *name*, *variable name* (that used in *flux* evolution functions), *mole weight*. Moreover, an array called *values* enables to store dynamics of the substance by means of a time-series, which may be computed by simulations or got by experiments (Figure 3);

- *ParameterExt*: it is a class that describes a parameter in terms of its *name*, *variable name* (that used in *flux* evolution functions), *evolution function* and *extra* information. Moreover, an array called *values* enables to store the dynamics of the parameter by means of a time-series, which may be computed by simulations or defined by values (Figure 3). If a parameter is defined together with its evolution function the *functional* field must be set to true, while, if the parameter dynamics is defined by values, the *functional* field must be set to false. In this last case periodical dynamics can be achieved by setting the *periodical* field to true;

- *FluxExt*: it is a class that describes a flux in terms of its *name*, *evolution function*, an *extra* information. An array called *values* enables to store the dynamics of the flux by means of time-series, which may be computed the Log-gain theory (Figure 3);

- *ReactionExt*: it is a class that describes a reaction by its *name*, its flux (field called *reactive unit*) and two vectors of its *reactants* and *products* (Figure 3);

- *MultiplicityExt*: it is the class which stores a *substance* involved in a reaction with its *multiplicity* (Figure 3);

- *ConstantMPExt*: it is a class that define a *constant* used in the model (e.g. π) by its *name*, *variable name* (that used in *flux* evolution functions), *value* and *measure unit* (Figure 2).

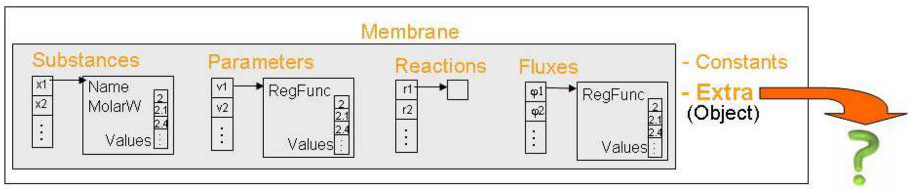
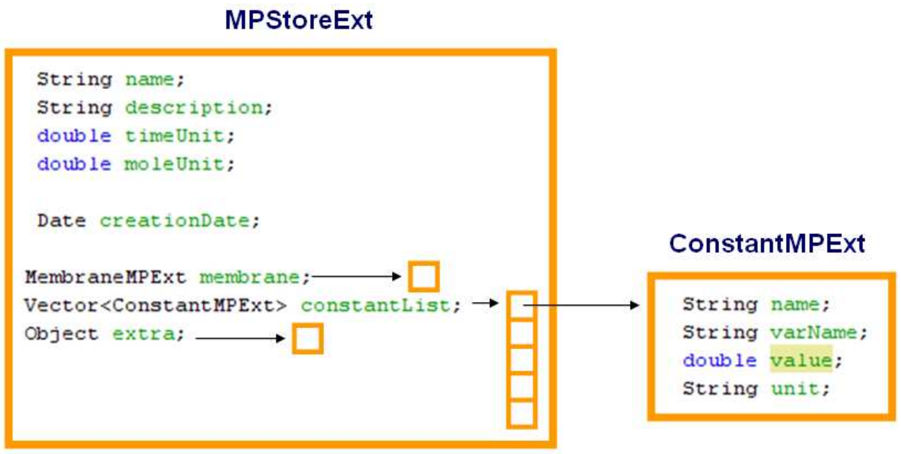


Figure 1: MPStoreExt data structure overview



```
String name;
String description;
double timeUnit;
double moleUnit;

Date creationDate;

MembraneMPExt membrane;
Vector<ConstantMPExt> constantList;
Object extra;
```

```
String name;
String varName;
double value;
String unit;
```

Figure 2: MPStoreExt class details

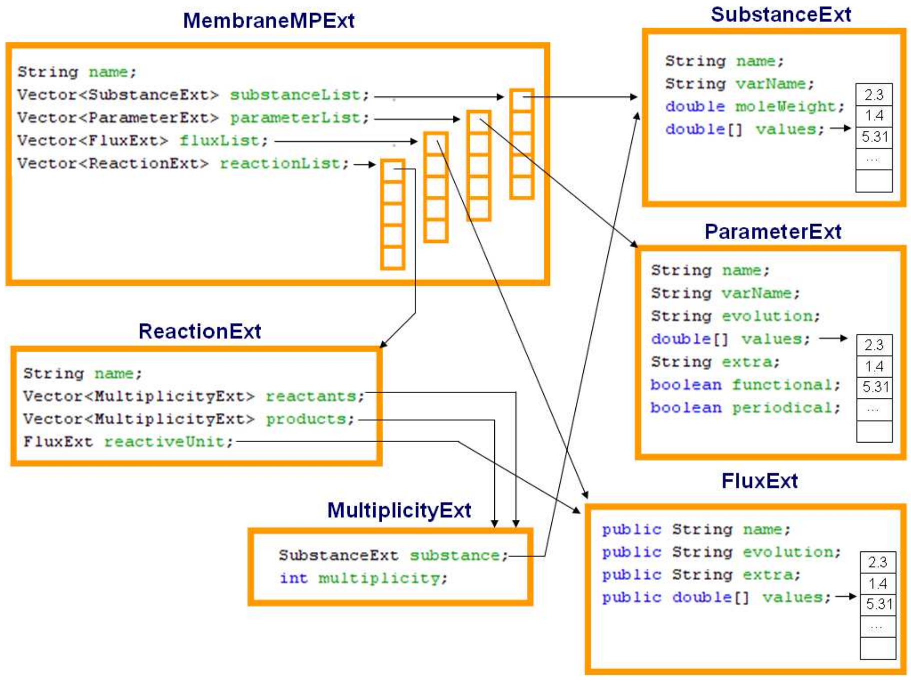
PluginExt package:

- *PluginExt*: it is the interface which must be implemented by every plugin. Figure 4 shows all the methods collected by this interface. For more information, see the code documentation;

- *PluginExtAbs*: it is an abstract class which implements the *PluginExt* interface (5). In particular, three important fields are defined:

  - *inMPStoreExt*: it is a vector of input MPStoreExt objects automatically received from the Plugin Manager when the plugin is called;
  - *outMPStoreExt*: it is a vector of output MPStoreExt objects automatically returned to the Plugin Manager when the plugin finishes its computation. These output can be displayed by the InputGUI or processed by other plugins;
  - *caller*: it refers to the object that calls the plugin.

  All the methods involved in the communication between the Plugin Manager and a plugin are implemented, leaving just three abstract methods to be implemented by the plugin developer:



```
MembraneMPExt
String name;
Vector<SubstanceExt> substanceList;
Vector<ParameterExt> parameterList;
Vector<FluxExt> fluxList;
Vector<ReactionExt> reactionList;

SubstanceExt
String name;
String varName;
double moleWeight;
double[] values;

ParameterExt
String name;
String varName;
String evolution;
double[] values;
String extra;
boolean functional;
boolean periodical;

ReactionExt
String name;
Vector<MultiplicityExt> reactants;
Vector<MultiplicityExt> products;
FluxExt reactiveUnit;

MultiplicityExt
SubstanceExt substance;
int multiplicity;

FluxExt
public String name;
public String evolution;
public String extra;
public double[] values;
```

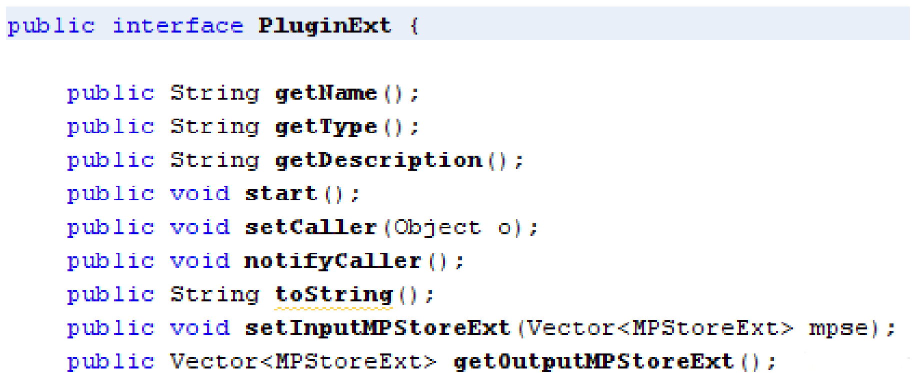
Figure 3: MembraneMPExt, SubstanceExt, parameterExt, FluxExt, ReactionExt, MultiplicityExt classes details

  - *getName()*: it has to return the name of the plugin as a *String*;
  - *getDescription()*: it has to return a description of the plugin features as a *String*;
  - *start()*: it is automatically called by the Plugin Manager when the user selects the plugin from the plugin list and launches it. The code included in this method should process the input received by the *inMPStoreExt* vector and store the processing data into the *outMPStoreExt* vector, to make it available to the Plugin Manager. Moreover, this method can call other methods, graphical user interfaces and even external libraries.

The PluginExtAbs class should be extended by the plugin developer in order to implement new plugins without wasting too much time on the data communication between the Plugin Manager and the Plugin.

Creation and compilation of a new plugin

To create a new plugin, called for instance *NewPlugin*, using the integrated development environment (IDE) *NetBeans*, one has to create a project called



```
public interface PluginExt {

    public String getName();
    public String getType();
    public String getDescription();
    public void start();
    public void setCaller(Object o);
    public void notifyCaller();
    public String toString();
    public void setInputMPStoreExt(Vector<MPStoreExt> mpse);
    public Vector<MPStoreExt> getOutputMPStoreExt();
```

Figure 4: Extract of PluginExt interface code

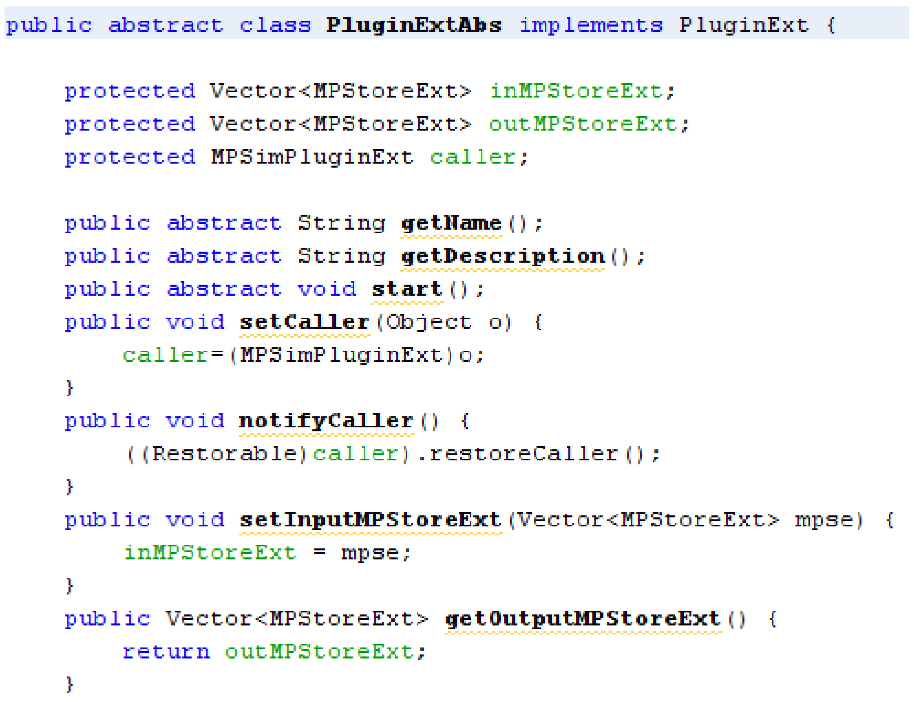
NewPlugin having a class *NewPlugin.java* inside its default package (i.e., in the main plugin directory). This class must implement the *PluginExt* interface and it is recommended that it extends the *PluginExtAbs* abstract class. In this last case, indeed, all the needed data structures will be automatically inherited, the communication between the plugin and the Plugin Manager will be suitably managed and just methods *getName()*, *getDescription()* and *start()* will have to be implemented, as we explained above. The *NewPlugin.java* header must include the importation of packages *MPStoreExt.** and *PluginExt.**. For getting a plugin example, please, download the *EmptyPlugin* source code from http://mplab.sci.univr.it/plugins/Plugins.php and start to develop your own plugin just implementing the three methods described above.

Before compiling the project, one has to set the project properties adding a link to MetaPlab library: select the *Properties* of *NewPlugin* project and add MetaPlab project (folder *MetaPlab-devel/MetaPlab*) to the project *Libraries*. To add external libraries one has to remember to add the library even to the MetaPlab project properties.

Testing the new plugin

To test a new plugin:

- compiling the plugin project obtaining a *jar* file,
- run the project following one of the next two steps:
  - copying the jar file into the source code folder *MetaPlab-devel/MetaPlab-/PluginExt*, and then running the MetaPlab project by clicking on the *run* button from NetBeans;



```java
public abstract class PluginExtAbs implements PluginExt {

    protected Vector<MPStoreExt> inMPStoreExt;
    protected Vector<MPStoreExt> outMPStoreExt;
    protected MPSimPluginExt caller;

    public abstract String getName();
    public abstract String getDescription();
    public abstract void start();
    public void setCaller(Object o) {
        caller=(MPSimPluginExt)o;
    }
    public void notifyCaller() {
        ((Restorable)caller).restoreCaller();
    }
    public void setInputMPStoreExt(Vector<MPStoreExt> mpse) {
        inMPStoreExt = mpse;
    }
    public Vector<MPStoreExt> getOutputMPStoreExt() {
        return outMPStoreExt;
    }
```

Figure 5: Extract of PluginExtAbs abstract class code

- copying the jar file into the binary code folder: *MetaPlab-dist/PluginExt*, and then running the MetaPlab project by clicking on *MetaPlab.jar* into the *MetaPlab-dist* folder.

The new plugin will be automatically detected and loaded by the Plugin Manager which will display a related entry in the plugin list. To run the new plugin, one has to select it and click on the *Run* button.

Publishing the new plugin

Once a plugin has been fully tested showing interesting results, it can be published on the *Plugin* section of the MetaPlab web-site. For more information, see the instructions at the page http://mplab.sci.univr.it/plugins/Plugins.php.