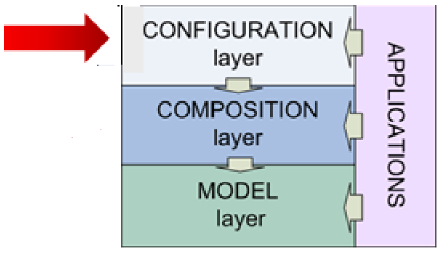
European Commission

BioMA

# Configuration Layer



# Reference Documentation

Joint Research Centre

| Release | Issue | Date |
|---|---|---|
| 1 | 3 | June 2019 |

## Copyright

## Disclaimer

# Contents

# About this document

1

This document describes the **BioMA Configuration layer**, one of the architectural layers of the BioMA Framework.

This document is targeted to software developers who want to create an application based on the configuration layer, or who want to transform a model into a configuration layer's compliant model.

Before reading this document, it is suggested to read these documents about the BioMA Framework:

- **BioMA Framework User Guide**, which describes the architecture and the components of the framework
- **Composition Layer Documentation**, describes the layer that allows composing modelling solutions and components

You can find all BioMA-related documents in the Agri4Cast Software Portal.

Normal users of the BioMA platform (e.g., the BioMA Spatial user, or an agronomical modeller) do not need to read this document in order to use the platform. We warmly suggest to the modellers to use the CLIC application in order to create a modelling solution. This way, CLIC will also create the configuration layer version of the modelling solution, without any effort by the user. (You can find the **CLIC User Guide** in the Agri4Cast Software Portal).

If, for some reasons, the modeller creates a composition layer modelling solution without using CLIC, and he/she wants to create the configuration layer version of the modelling solution, as well, please refer to section "How to transform a composition layer modelling solution into a configuration layer's one" on page 40. Even in this case, the modeller does not need to read this whole document.

In brief:

- The modeller who uses CLIC to create a modelling solution does not need to read this document.

- The modeller who creates a composition layer modelling solution without using CLIC, must read only the chapter “How to transform a composition layer modelling solution into a configuration layer’s one” on page 40.
- The developer who wants to transform a generic model (not using the composition layer) into a configuration layer modelling solution must read this document entirely.
- The developer who wants to create an application based on the configuration layer must read this document entirely.

**The topics are organized as follows:**

| Topic | Description |
|---|---|
| “About the Configuration layer” on page 5 | • What the Configuration layer is with respect to the BioMA framework architecture<br>• Main purposes of the Composition layer |
| “Configuration Layer Technical Documentation” on page 11 | • Terminology used in this chapter<br>• Main Interfaces and their relationships<br>• Other classes and their relationship<br>• BioMA Applications plugin architecture and IBiomaPlugin interface<br>• Persister mechanism<br>• Configuration files<br>• How to deploy a library within an application<br>• How to transform a composition layer modelling solution into a configuration layer’s one |

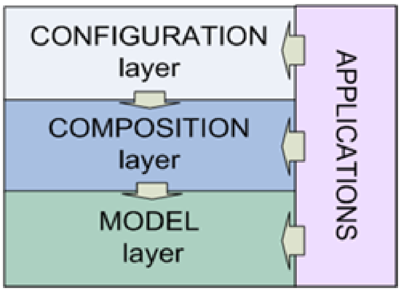
# About the Configuration layer

# 2

## The BioMA three layers architecture

The BioMA modelling framework simulation system has been discretized in layers, each with its own purposes.

The following diagram shows the framework layers:



As you can see, the configuration layer stays on top of the composition and the model layers. Please, note that there is no direct dependency between the configuration and the composition layer: it means that a configuration layer modelling solution does not need to be built on top of a composition layer modelling solution.

There is a dependency from the configuration layer to the model layer because some model layer classes (e.g., the VarInfo class) are used in the public interfaces of the configuration layer

- The **model layer** is where fine granularity models are implemented as discrete units.
- The **composition layer** is where basic models from different components are composed to build a modelling solution.
- The **configuration layer** is where information and data (configuration items) can be added to a modelling solution to make it usable in a specific context. (More details are provided in the next section).

Currently, the applications that use the configuration layer are **BioMA Spatial** and **Optimizer**.

The first allows running the simulations of the modelling solutions. The second allows calibrating the parameters of the modelling solutions versus some reference data.

You can find the **BioMA Spatial** and the **Optimizer** user guides in the Agri4Cast Software Portal.

## Purposes of the Configuration layer

The Configuration layer allows adding information to a modelling solution to make it usable in a specific context. The information consists of configuration items which specify the information needed, the information allowed, and possibly the tool to edit the information.

The Configuration layer also includes handles to run the model and to change model parameters.

By using the configuration layer it is possible to manage the input of the data into any modelling solution in the same way, so it is possible to build generic applications (like BioMA Spatial and Optimizer) that run any modelling solution. So, it is eliminated the need to create a new application whenever a new modelling solution is created or modified.

So, the configuration layer must be seen as a layer to standardize any modelling solution to be managed in the same way by the applications.

The functionalities that the configuration layer provides to the applications are:

- The possibility to fill the input data directly (e.g., the application user inputs a number as a variable value).
- The possibility to fill the input data by specifying the sources where to read the data from (e.g., the connection to a DB or the path to CSV file).
- The validation of the input data to fulfil certain requirements.
- The possibility to guide the application user in the process of filling the input data (e.g., to show in a list all the possible values for a certain variable).
- The possibility to define different ways to save the simulation outputs ("persister" objects) and configure them.
- The possibility to iterate the execution of a modelling solution in any dimension (for example on different years, or on different locations).
- The possibility to save in a configuration file all the inputs entered by the user and to reload the configuration file later.

The BioMA framework provides a specific Graphical User Interfaces (GUI) to manage the configuration layer into the applications. This GUI is already included in the BioMA Spatial and Optimizer applications.

**See also:**

- “The BioMA three layers architecture” on page 6
- “Configuration Layer Technical Documentation” on page 11

# Configuration Layer Technical Documentation

3

This chapter is organized into the following sections:

# Terminology used in this chapter

The C# object names and namespaces are written in *italic*.

The namespace of the configuration layer C# objects is *EC.JRC.MARS.ConfigurationLayer.Core*.

In this document, the classes having this namespace will be named only through their Name. Where there are references to classes having different namespace, it will be clearly written.

Where is written “the applications”, it refers to the applications built on the configuration layer. Currently, these are BioMA Spatial and Optimizer

## The library

The configuration layer is defined in the *EC.JRC.MARS.ConfigurationLayer.Core.dll* C# software library.

**See also:**

- “Main Interfaces” on page 13
- “BioMA Applications plugin architecture and IBiomaPlugin interface” on page 29

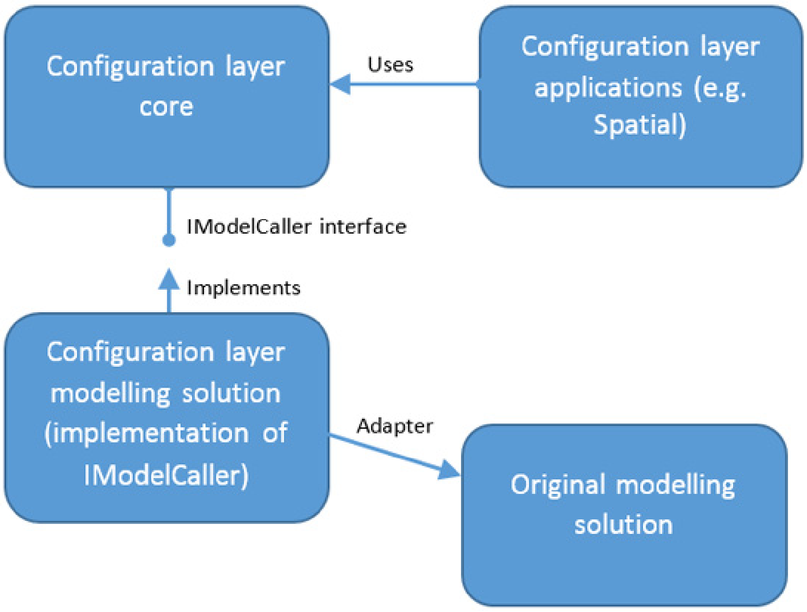
## Main Interfaces

The main interfaces of the configuration layer are: *IConfigurable* and *IModelCaller.*

*IConfigurable* represents an object that can be “configured”: it is possible to set the values of its internal properties using a configuration item.

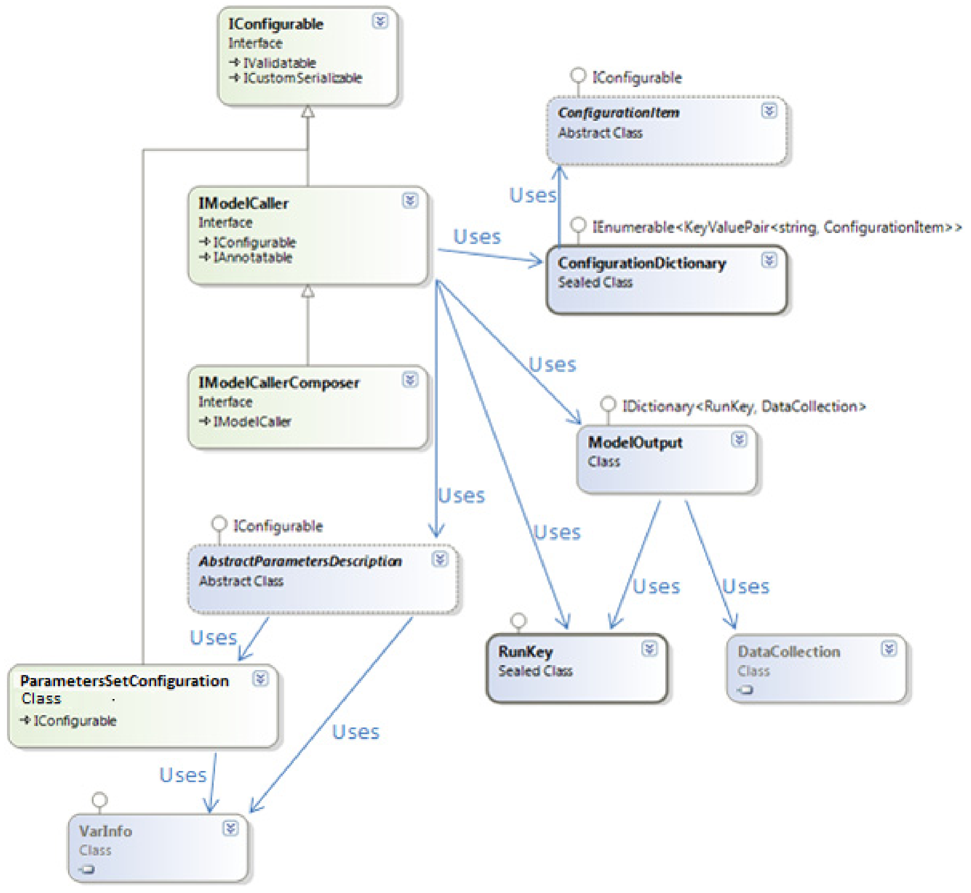
At configuration layer, a modelling solution is defined as a public class that implements the *IModelCaller* interface. The applications code interacts with the deployed modelling solutions through the *IModelCaller* interface. The *IModelCaller* interface extends the *IConfigurable* interface, so an implementation of *IModelCaller* is also *IConfigurable.*

The next figure shows the architectural structure of the configuration layer: a modelling solution (e.g., a single model object or a composition layer modelling solution) is referenced in the configuration layer through an adapter, which implements the *IModelCaller* interface. Once the adapter of the modelling solution is created, the modelling solution can be used by the applications:


Configuration layer core
Uses
Configuration layer applications (e.g. Spatial)
IModelCaller interface
Implements
Configuration layer modelling solution (implementation of IModelCaller)
Adapter
Original modelling solution

## Other classes and their relationship

The figure below shows the relations between the classes and the main interfaces of the configuration layer:



### Description:

- The **ModelOutput** class represents the output of the simulation of the model. It is defined as a dictionary of RunKeys and DataCollections.
- The **RunKey** represents a set of values that describe the configuration of the simulation in a unique way. It is used as a key to identify

different simulation results inside the **ModelOutput** class (for example, each key can represent the output of a different location in a location-iterated model execution).

- The **DataCollection** is a class belonging to the BioMA Model Layer and it is the container of the output of the simulation.
- The **ConfigurationDictionary** class contains the configuration values of the *IConfigurable* objects. It is defined as a set of ConfigurationItems.
- A **ConfigurationItem** describes one of the elements of the configuration. E.g., a configuration may contain one item that describes the kind of weather data, one item that describes the location data, etc. The set of these configuration items is the ConfigurationDictionary. Each item is provided with the rules to check the validity of its value. The ConfigurationDictionary has the functionality to check the validity of all its ConfigurationItems.
- The **AbstractParametersDescription** is an abstract class that defines a template to create the component specific "parameters description" objects. The "parameters description" is the object that contains the logics to retrieve the parameters of a component of a modeling solution. A "parameter description" is composed by several ParametersSetConfiguration objects, one for each sub-component of the component. So, we have a two levels architecture for managing the parameters: the modelling solution parameters are managed by different AbstractParametersDescriptions (one for each component). Each component parameters are managed by different ParametersSetConfigurations (one for each sub-component). In case the modeling solution is based on the BioMA Model/Composition Layer, the sub-components are the "strategies" (implementation of *IStrategy*), whereas the components are the composition layer components (*ISimulationComponent*).
- A **ParametersSetConfiguration** contains the logics to retrieve the parameters of a sub-component. E.g., a ParametersSetConfiguration may read the parameters from a XML file, or get the parameters using a query to a database, etc. A ParametersSetConfiguration can be configured, so it implements the *IConfigurable* interface.
- The parameters are defined as object of **VarInfo** type. The VarInfo class comes from the BioMA Model Layer and contains the object properties to store the value of the parameters and the logics to check the validity of the values (e.g., the value's validity boundaries).

These classes are described in detail in the next sections.

**See:**

- "IConfigurable interface and Configuration Dictionary" on page 16
- "IConfigurable implementation sample" on page 19
- "IModelCaller interface" on page 23
- "ModelOutput and RunKey classes" on page 24
- "Model composition and ModelCallerIterator" on page 26
- "Data providers and IDataProvider interface" on page 28

## IConfigurable interface and Configuration Dictionary

The **IConfigurable** interface represents an object with a configuration.

A configuration is the handle to change or read the internal status of the object in a generic way (i.e., without using specific properties). Every object that can be configured should implement IConfigurable. The configuration is implemented as a ConfigurationDictionary object, which is a collection of ConfigurationItems objects.

The table below shows the main property of the IConfigurable interface:

| Icon | Member | Description |
|---|---|---|
| ▤ | Configuration | Returns the configuration of the object as a ConfigurationDictionary. |

A **ConfigurationItem** represents an element of the configuration of the model whose Value property can be set, read and verified against some validation logic. It is used together with the ConfigurationDictionary class. Users of this class (or of child classes) are of 2 types:

- **Consumer**: The user that needs to read the Value. It will instantiate this class (or child classes) and will provide, via the appropriate delegates, validation rules for the value contained. This corresponds to the rule that consumers know when a configuration element value is valid for their own purposes.
- **Provider**: The user who sets the Value property providing therefore configuration information to the Consumer. Once the Value has been set, it can call the validation methods, asking the Consumer if the value is suitable for its own purposes.

The Value property is a string property. Each configuration item contains the logics to transform the Value to the "parsed value".

For example, a particular ConfigurationItem can parse the Value to a number, or to a DateTime. The "parsed value" is obtained by calling the `GetParsedValue()` method. Hence the parsed value can be of any Type, it is defined as an object.

The parsing logics are contained in a specific instance of *IGenericValueConverter* interface defined in the constructor of the ConfigurationItem by the Consumer.

The table below shows the main methods of the ConfigurationItem class:

| Icon | Member | Description |
|---|---|---|
| ◆ | ConfigurationItem (ConfigurationDictionary, GenericVerifierDel, IGenericValueConverter<String>, Type) | Creates a new instance of the class. The created ConfiguratioItem refers to the ConfiguratioDictionary passed as the containing dictionary. The IGenericValueConverter contains the logics to convert the Value to the "parsed value". |
| ▤ | Configuration | Returns the configuration of the ConfigurationItem (composite pattern). |
| ▤ | ConfigurationItemType | The actual type of the Value property. |
| ▤ | GenericVerifier | Can be implemented to provide any further verification logic suitable for the configuration element. |
| ◆ | GetParsedValue() | Returns the parsed value, according to the IGenericValueConverter configured in the constructor, or to the one registered with StringConverterRegistry. |
| ◆ | IsErrorPresent(String) | Used during ConfigurationItems' validation in the ConfigurationDictionary.CheckConfigurationItemsValidity() method. |
| ⚡ | setHandler | Event that occurs when the property Value is set. |
| ◆ | Validate() | |
| ▤ | Value | The value (string) of this ConfigurationItem. |
| ◆ | Verify() | Must be implemented to call the delegate configured in IValue.GenericVerifier. |

The ConfigurationDictionary is a collection of ConfigurationItems, plus the functionalities to manage that collection (verify the consistency of the items against validation rules, save the collection, verify the correctness of the whole collection). Users of this class are of 2 types

- **Consumer**: The user who needs to read the values of the ConfigurationItems. It will instantiate this class providing the exact list of the ConfigurationItems' names it needs, and will provide, via the appropriate delegates, validation rules for each ConfigurationItem contained.

- **Provider**: The user who sets all the Values properties, providing therefore configuration information to the Consumer. It can, once it has set all the Values, call the verification method CheckConfigurationItemsValidity(), asking the Consumer if the values of the ConfigurationItems are suitable for its own purposes. This class is sealed, so it's not possible to redefine the containment and global validation logic, while it's possible to provide a custom ConfigurationItem, by sub classing it.

A particular subclass of ConfigurationItem is the LoadableConfigurationItem: its Value is a fully-qualified class name. Its parsed value is an instance of the class identified by the Value.

If the class implements *IConfigurable*, then a tree-like structure is created, where an *IConfigurable* creates another *IConfigurable*, and so on. We say that the LoadableConfigurationItem “implies” the newly create *IConfigurable*.

For example, an *IModelCaller* may have a LoadableConfigurationItem that creates the instance of another *IModelCaller*: this way, the first model caller can "use" the latter.

The table below shows the main methods of the ConfigurationDictionary class.

| Icon | Member | Description |
|---|---|---|
| ◆ | ConfigurationDictionary(String[]) | An instance of this class has the items described by the array of string passed to the constructor: after the creation, the user cannot add or remove any keys. |
| ⚡ | allSet | Event triggered when, after setting a Value for one of the items in this instance, all the ConfigurationItem(s) contained are set in the sense of CheckConfigurationItemsPresence(). |
| ◆ | CheckConfigurationItemsPresence () | Throws Exception if a ConfigurationItem value is not set. |
| ◆ | CheckConfigurationItemsValidity() | Throws Exception if verification is not passed. In this method, first of all CheckParametersPresence() is invoked to ensure the ConfigurationItem's values presence; then, further verification is performed by invoking the methods IsErrorPresent and Verify(). |

| Icon | Member | Description |
|---|---|---|
| ◆ | ConfigurationEqualsTo(ConfigurationDictionary) | Checks that this configuration is equals to the “otherConfiguration” parameter. It checks that:<br>• The two configurations contain the same set of keys (the order is not important).<br>• Each configuration item has a corresponding configuration item of the same type.<br>• For each couple of corresponding configuration items, it checks that:<br>- the two corresponding configuration items have the same value;<br>- if the two corresponding configuration items are LoadableConfigurationItem and their implicated object is a IConfigurable, it checks that the two configurations of the implicated objects are equals (recursively). |
| ◆ | GetConfigurationItem(String) | Gets the ConfigurationItem related to one key. |
| ◆ | GetConfigurationItemForImpliedObject(Object) | Searches for the ConfigurationItem that "implied" the IConfigurable passed as a parameter. The search is conducted recursively in all the ConfigurationDictionaries implied, starting with the root dictionary in the hierarchy containing this one. |
| ◆ | GetConfigurationItemParsedValue (ConfigurationDictionary, String) | Gets the parsed value for the ConfigurationItem identified by the key passed as parameter. |
| ◆ | GetConfigurationItemValue(String) | Gets the value related to one key. |
| ▤ | Keys | Retrieves the list of all ConfigurationItem’s names, in the order in which they were configured in the Constructor. |
| ◆ | ResetValues() | Resets the values for all ConfigurationItems in this dictionary, while leaving unchanged the verification logic provided by the delegates. |
| ◆ | Set(String, ConfigurationItem) | Sets the ConfigurationItem corresponding to a given ConfigurationItem name. This should not be invoked directly, but via an extension method normally called Bind, provided by each class extending ConfigurationItem. Its implementer should also provide this extension method to this class in order to bind properly the ConfigurationItem. The present method Set is left public because it must be called inside the Bind extension. |
| ⚡ | treeStructureChanged | Event triggered when the structure of the tree below this configuration dictionary changes. |
| ⚡ | validationRuleChanged | Event triggered when a validation rule (e.g., the acceptable values verifier or the generic verifier) of a configuration item changes. |

## IConfigurable implementation sample

In the following, we provide the code of an example of an *IConfigurable* valid implementation.

**Key points:**

- The using statement must include the main configuration layer namespace: **EC.JRC.MARS.ConfigurationLayer.Core.Configuration**.
- The ConfigurationDictionary must be a private field of the class, and must be available in GET by the public property Configuration.
- The ConfigurationDictionary must be instantiated in the constructor of the class and should not be re-instantiated later. In fact, this could lead to a change in the configuration (e.g., the set of configuration item changes) and this will cause problems in the management of the configuration by the applications.
- In this example the configuration contains 2 configuration items:
  - “Item one” contains a string value. The constraint on the value is defined in the ItemOneGenericVerifier method: the string length must be less than 10 characters.
  - “Item two” contains an integer value. The configuration item is of type “BoundedConfigurationItem”: this is a type of configuration item which is useful to check that the value is between a lower and a higher boundary. The boundaries are specified at the moment of the item definition (in this case, the value must be between 1 and 30).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using EC.JRC.MARS.ConfigurationLayer.Core.Configuration;

namespace MyNamespace
{
    public class TestConfigurable:IConfigurable
    {
        //constructor
        public TestConfigurable()
        {
            //instantiate the configuration dictionary field
defining two configuration items
            _configurationDictionary= new
ConfigurationDictionary("item one","item two");

            //define the configuration item 1
            ConfigurationItem i1 = new
UncheckedConfigurationItem(_configurationDictionary,
ItemOneGenericVerifier, typeof (string));
```

```
             //insert the configuration item 1 in the
configuration dictionary
            _configurationDictionary.Set("item one", i1);

            //define the configuration item 2
            ConfigurationItem i2 = new
BoundedConfigurationItem(_configurationDictionary,
null,()=>"1",()=>"30",ItemTwoBoundariesVerifier,typeof(int));
          //insert the configuration item 2 in the configuration
dictionary
            _configurationDictionary.Set("item two", i2);

        }



        /// <summary>
        /// Generic verifier (validator) of the value of
configuration item "item one"
        /// </summary>
        /// <param name="p"></param>
        /// <param name="d"></param>
        /// <returns></returns>
        private string ItemOneGenericVerifier(ConfigurationItem
p, ConfigurationDictionary d)
        {
           if (((string)p.GetParsedValue()).Length >= 10) return
"'item one' should be a string made by less than 10 characters";
            return null;//return null of no error
        }

        /// <summary>
        /// Boundaries verifier (validator) of the value of
configuration item "item two"
        /// </summary>
        /// <param name="p"></param>
        /// <returns></returns>
        private bool
ItemTwoBoundariesVerifier(BoundedConfigurationItem p)
        {
            return ((int)p.GetParsedValue() <=
Convert.ToInt32(p.HigherBound)) &&
                   ((int)p.GetParsedValue() >=
Convert.ToInt32(p.LowerBound));
        }


        //contains the configuration of the class
        private ConfigurationDictionary _configurationDictionary;

        public ConfigurationDictionary Configuration
        {
            //return the private field _configurationDictionary
that contains the configuration of the class
            get { return _configurationDictionary; }
        }

        public void Validate()
        {
            //validation is managed by the configuration
dictionary CheckConfigurationItemsValidity method, that checks
the validity of every configuration item

_configurationDictionary.CheckConfigurationItemsValidity();
        }

        public XElement CustomSerialize(SerializationContext
context)
        {
            return new XElement("empty");//no need of custom
serialization in this class
        }

        public void CustomDeserialize(XElement doc,
DeserializationContext context)
        {
           //no need of custom serialization in this class
        }

        public string Description
        {
            get { return "example class to demonstrate the
IConfigurable interface"; }
        }

        public string URL
        {
            get { return "http:\\myurl"; }
        }

    }
}
```

The following code is the code of the consumer that uses the TestConfigurable class (writes and reads the configuration item's values).

The handle to the configuration items is the GetConfigurationItem method:

```
 public void Test()
      {
        TestConfigurable obj= new TestConfigurable();
        obj.Configuration.GetConfigurationItem("item one").Value = "hi!";
        obj.Configuration.GetConfigurationItem("item two").Value = "14";
        obj.Validate();//no validation errors
        int res=(int)obj.Configuration.GetConfigurationItem("item
  two").GetParsedValue();//it will return the integer 14
                 }
```

## IModelCaller interface

IModelCaller interface must be implemented by all the modelling solutions of the framework.

It provides the methods to:

- Configure the run of a simulation of the modeling solution
- Set the parameters of the modeling solution
- Verify the correctness of the configuration and initialize the model (e.g., reset the internal status of the model)
- Execute the simulation
- Retrieve the result of the simulation

The table below shows the main method of the IModelCaller interface:

| Icon | Member | Description |
|---|---|---|
| ◆ | GetModelOutputSkeleton() | Returns the empty structure of the ModelOutput of the Modeling Solution, represented by one entry. This method should be implemented to return an empty ModelOutput reflecting the structure that it will have at an invocation of the IModelCaller.Call method, without data. This means that the IDictionary{RunKey, DataCollection that ModelOutput implements and that is returned here, should contain one entry pair, with a RunKey with all the run code descriptors as the key (but with fake values), and an empty DataCollection as the value. |
| ◆ | AddParametersAcceptable() | To be called to verify that the model parameters have acceptable values. E.g., they have values in their range, etc. |
| ◆ | Call() | Calls the execution of a Simulation Model. |
| ▤ | ComponentOutputs | For each component of the model, returns the list of the components outputs. |

| Icon | Member | Description |
|---|---|---|
| ▤ | Configuration | Returns the configuration of the model (inherited from IConfigurable). |
| ▤ | Description() | Returns the description of the model (inherited from IAnnotatable). |
| ◆ | Deserialize(IConfigurable, XElement, DeserializationContext) | Deserialize the configuration of the model: from an XML element, creates the configuration. (Inherited from ConfigurableSerializer). It is used to load a configuration saved as XML by the Serialize method. |
| ◆ | Finalization() | Called to perform “final” actions on this instance, after the Call method. |
| ◆ | Initialize() | Called to populate every data that needs to “survive” multiple calls to the Call method. |
| ▤ | Metadata() | Returns the metadata of the model. (Inherited from IAnnotatable). |
| ▤ | ParametersConfigurations | Returns the objects (one for each group of parameters set) that contains the configurations of the ParametersSetConfiguration classes. |
| ◆ | Serialize(IConfigurable, SerializationContext) | Serializes the configuration of the mode by saving the configuration in form of an XML element. (Inherited from ConfigurableSerializer). It is used to save the configuration of the model. |
| ▤ | URL() | Returns the URL associated with the metadata. (Inherited from IAnnotatable). |
| ◆ | Validate() | Validates the configuration of the model. |

## ModelOutput and RunKey classes

The execution of the method Call of *IModelCaller* returns a **ModelOutput** object that represents the output of the model. It inherits from an IDictionary<RunKey, DataCollection>, where the key is a RunKey and the value is the DataCollection returning from a single run.

DataCollection is a container of data, structured as a collection of “tables”. Every table contains a fixed set of columns and an extensible set of rows.

The RunKey represents the key to identify the output of the model. It has a dictionary structure, where the keys of the dictionary are called “run code descriptors”. Both the keys and the values are string objects. To use the RunKey, it is mandatory to set a value for each run code descriptor, using the Set method (or the this[string key] set property). For each run code descriptor, the value must be set only once.

The following table shows the main methods of ModelOutput:

| Icon | Member | Description |
|---|---|---|
| ◆ | ModelOutput(String[]) | Creates an instance of this class, with a fixed set of run code descriptor names. |
| ◆ | Add(RunKey, DataCollection) | Adds a RunKey, DataCollection pair. |
| ◆ | Add(KeyValuePair<RunKey, DataCollection>) | See IDictionary documentation. |
| ◆ | Clear() | See IDictionary documentation. |
| ◆ | Contains(KeyValuePair<RunKey, DataCollection>) | See IDictionary documentation. |
| ◆ | Contains(RunKey) | See IDictionary documentation. |
| ◆ | CopyTo(KeyValuePair<RunKey, DataCollection>[], Int32) | See IDictionary documentation. |
| ▤ | Count | Returns the number of RunKey, DataCollection couples in the dictionary. See IDictionary documentation. |
| ◆ | GetEnumerator() | Returns the enumerator for this instance. |
| 🔒◆ | IEnumerable.GetEnumerator() | Returns the enumerator for this instance. |
| ◆ | GetRunKey(String[]) | Builds a RunKey with the run code descriptor names configured at instantiation of this class and the values passed as parameter. |
| ◆ | GetRunKey() | Builds a RunKey with the run code descriptor names configured at instantiation of this class. |
| ▤ | IsReadOnly | See IDictionary documentation. |
| ▤ | Item[RunKey] | The DataCollection for the indexed RunKey. |
| ▤ | Keys | Returns the enumeration of the RunKeys. See IDictionary documentation. |
| ◆ | Remove(RunKey) | See IDictionary documentation. |
| ◆ | Remove(KeyValuePair<RunKey, DataCollection>) | Removes the couple RunKey, DataCollection from the ModelOutput dictionary. See IDictionary documentation. |
| ▤ | RunCodeDescriptors | Cloned array of the run code descriptor names. |
| ◆ | TryGetValue(RunKey, DataCollection) | Returns the DataCollection association with the specified RunKey. See IDictionary documentation. |
| ▤ | Values | Returns the enumeration of the DataCollections. See IDictionary documentation. |

The following table shows the main methods of RunKey:

| Icon | Member | Description |
|---|---|---|
| ◆ | Add(String, String) | Calls the Set method. See Set method documentation. |
| ◆ | Add(KeyValuePair<String, String>) | See IDictionary documentation. |

| Icon | Member | Description |
|---|---|---|
| | AllSet | Returns true if all the run code descriptors' values are set. |
| | Clear() | See IDictionary documentation. |
| | CompareTo(RunKey) | See IDictionary documentation. |
| | ComplementRunKey(String, String) | Creates a new RunKey that is equal to the current object but has a new run code descriptor. The new RunKey has all the current object descriptors plus the new one. This method does not modify the current object. |
| | Contains(KeyValuePair<String, String>) | See IDictionary documentation. |
| | ContainsKey(String) | See IDictionary documentation. |
| | CopyTo(KeyValuePair<String, String>[], Int32) | See IDictionary documentation. |
| | Count | See IDictionary documentation. |
| | GetEnumerator() | See IDictionary documentation. |
| | IEnumerable.GetEnumerator() | See IDictionary documentation. |
| | GetType() | (Inherited from Object). |
| | IsReadOnly | See IDictionary documentation. |
| | Item[String] | See IDictionary documentation. |
| | Keys | See IDictionary documentation. |
| | Remove(String) | Throws exception because it is not possible to remove a descriptor from a RunKey. |
| | Remove(KeyValuePair<String, String>) | See IDictionary documentation. |
| | RunCodeDescriptorsRepresentatio n | Returns a string representation of the descriptors of the RunKey. |
| | Set(String, String) | Sets the value of the specified run code descriptor if, and only if, the value had not already been set. If the value has already been set, an exception of type AttemptToModifyRunKeyException is thrown. |

## Model composition and ModelCallerIterator

An *IModelCaller* can be composed with other *IModelCallers* to create a more complex model.

This can be done via the *IModelCallerComposer* interface. It implements the *IModelCaller* interface to satisfy the "composite pattern", i.e., the composer can be in turn composed.

The following table, shows the method of the *IModelCallerComposer* interface:

| Icon | Member | Description |
|---|---|---|
| | Add (IModelCaller) | Adds and IModelCaller to the composer. |

A particular type of composer is the iterator: the **ModelCallerIterator** composes a model with itself, calling it several times as the number of desired iterations.

At each call, the value of one of the configuration items (the iteration field) is changed. It is used, for example, to run the same model on many locations, or over many time periods.

The following table shows the main methods of the ModelCallerIterator class:

| Icon | Member | Description |
|---|---|---|
| | ModelCallerIterator() | Creates a ModelCallerIterator. |
| | ModelCallerIterator(String) | Creates a ModelCallerIterator setting the path where the model to iterate is. |
| | Add (IModelCaller) | Adds and IModelCaller as iterated model of the iterator. |
| | AreParamtersAcceptable() | Checks the correctness of the parameters of the iterated model. |
| S | BuildModelCallerIterator(String, IModelCaller) | Static method that is used to create an iterator over the model passed as parameters. The first parameter is the name of the iteration field (it must be one of the model's configuration items). |
| | Call() | For each value of the iteration field, calls the Initialize - Call - Finalize methods of the iterated model. It also composes the ModelOutput returned at each call, to a single ModelOutput that has one couple RunKey-DataCollection for each iteration. It reacts to the stop signal, canceling the iterations not yet executed and returning the partial ModelOutput. At the end of each iteration, an IterationDone event is triggered. At the end of each successful iteration, an IterationSucceeded event is triggered. At the end of each iteration with errors, an IterationNotSucceeded event is triggered. |
| | Finalization() | This method does nothing: the Finalization of the iterated model is called, at each iteration, in the iterator Call method. |
| | GetModelOutputSkeleton () | Returns the iterated model GetModelOutputSkeleton. |
| | GutNumberOfIterations() | Returns the number of iterations, i.e. the number of values set in the iteration field. |
| | Initialize() | This method does nothing: the Initialize of the iterated model is called, at each iteration, in the iterator Call method. |
| | InnerOriginalModel | Returns the original model attached at the inner level of iteration. |

| Icon | Member | Description |
|---|---|---|
|  | ParametersConfigurations | Returns the iterated model's ParametersConfigurations. |
|  | SetStopSignal() | Activates the stop signal, checked at every iteration cycle. |

## Data providers and IDataProvider interface

The BioMA framework provides a set of data handlers created to provide data required by the model to the application. These classes are not part of the Configuration Layer, but are based on it, since they are configurable to be easily configured inside the BioMA applications.

This allows extracting from the model the logics to retrieve the input data, such as the weather information, and to put this logic in a separated and proper component. The data providers can be used with any modelling solutions, and can be deployed into the application installation separately. Each data provider defines an interface, which standardizes the data that the provider returns (e.g., the format of the weather data).

Each provider can be used directly from the code of the model, or can be referenced by the configuration of the modelling solution as a LoadableConfigurationItem. The latter approach is better, because it allows changing the instance of the provider at run time, giving a great flexibility to the simulation.

The data providers implement the *IConfigurable* interface (to be configurable) and the *IDataProvider* interface.

The *EC.JRC.MARS.ModelLayer.Core.IDataProvider* interface is an empty interface (no methods or properties) that is used only to distinguish the data providers.

These are the most commonly used data providers:

- **Weather** provider (interface *JRC.IPSC.MARS.WeatherProviders.Interfaces.IWeatherProvider*): provides weather daily data.
- **Soil data** provider (interface *JRC.IPSC.MARS.SoilDataProvider.Interfaces.ISoilDataProvider*): provides soil data.
- **Agromanagement data** provider (interface *JRC.IPSC.MARS.Agromanagement.Interfaces. IAgromanagementProvider*): provides agromanagement data.

# BioMA Applications plugin architecture and IBiomaPlugin interface

BioMA applications (e.g., BioMA Spatial) allow a third-party developer to build a plugin, that is, an external code that extends the functionality of the application.

The plugins are deployable in BioMA applications as DLLs. The deploy operation does not require neither a change in the application code nor the rebuild of the application.

To be a valid plugin, a class must implement the interface *IBiomaPlugin* defined in the **JRC.IPSC.MARS.BiomaPluginInterfacesDefiniton.dll** software library.

The following table shows the methods of the *IBiomaPlugin* interface:

| Icon | Member | Description |
|---|---|---|
| ◆ | Dispose() | For further use. |
| ◆ | Execute() | This method provides the logics of the plugin. It is called every time the plugin button is clicked in BioMA. |
| ▤ | Host | Set property. Sets the handle to the BioMA application through the IPluginHost interface. |
| ◆ | Initialize() | This method provides the initialization logics of the plugin. It is called every time the plugin button is clicked in BioMA. |
| ▤ | PluginAuthor | Returns the plugin author. |
| ▤ | PluginAvailabilityStatusSet | Returns the list of BiomaApplicationStatus in which the plugin is available (i.e., the plugin button is clickable). |
| ▤ | PluginDescription | Returns the plugin description. It will be displayed as the label of the plugin button in the BioMA GUI. |
| ▤ | PluginImagePath | Returns the path to the image that will be displayed on the plugin button in the BioMA GUI. Leave it null to only display text. |
| ▤ | PluginLongDescription | Returns the plugin long description. |
| ▤ | PluginName | Returns the plugin name. (This must be unique. It is not allowed to have two plugins with the same name). |
| ▤ | PluginVersion | Returns the plugin version Id. |

To be a valid plugin host (that is, an application that can host plugins) all the BioMA applications implement the *IPluginHost* interface, which is defined in the **JRC.IPSC.MARS.BiomaPluginInterfacesDefiniton.dll** software library.

Through the *IPluginHost* interface the application can give the data to the plugins (e.g., the last simulation output), or a handle to the current modelling solution.

**See also:**

- “Persister mechanism” on page 35
- “Configuration files” on page 31
- “How to deploy a library within an application” on page 34
- “How to transform a composition layer modelling solution into a configuration layer’s one” on page 40

# Configuration files

The configuration of the *IConfigurable* object can be saved in a .XML file, following the format defined in the BioMA Configuration layer.

Its file extension could be .BCF (Bioma Configuration File) or .PCF (Persisters Configuration File) being the file schema the same.

The BioMA configuration files are .XML files whose format and organization are as follows:

```
      <Serialization>
        <IConfigurable class="JRC.IPSC.MARS.BIOMA.CORE2.RootModelCaller">
            <Id>0</Id>
            <Set>
              <KeyDict>Modeling Solution</KeyDict>
              <ValueDict>
   EC.JRC.MARS.ConfigurationLayer.Core.IterationFramework.ModelCallerIter
   ator
              </ValueDict>
                  </Set>
                  <Set>
                    <KeyDict>Modeling Solution</KeyDict>
                    <KeyDict>Modeling Solution</KeyDict>
                    <ValueDict>
    EC.JRC.MARS.WOFOSTOperationalModelCaller.WOFOSTOperationalModelCaller
              </ValueDict>
            </Set>
              …
        </IConfigurable>
        <CustomSerializationsInTree>
                ….
          </CustomSerializationsInTree>
          <Events />
         </Serialization>
```

Under the `Serialization` tag, there are three tags: `IConfigurable`, `CustomSerializationInTree` and `Events`.

In common usage, the last two tags are always empty. So, in this document, only the `IConfigurable` tag is described.

The `IConfigurable` tag contains a list of `Set` tags, each one setting one characteristic (also called 'configuration item') of the simulation.

The configuration items are defined within a tree structure: each item is a leaf or a branch of the tree and belongs to one and only one parent in the tree. The root of the tree is one.

Each `Set` tag contains the identifier of the configuration item and the value to set. The identifier of the configuration item is made by the structure of the tree containing it, using the `KeyDict` tags for each level of the tree.

For example, if this is the structure of the tree (A is the root; C, D and H are leaves; B and E are branches)

```
A       ->      B       ->      C
        ->      D
        ->      E       ->      H
```

This is the identifier of item B:

```
<KeyDict>A</KeyDict>
<KeyDict>B</KeyDict>
```

This is the identifier of item H:

```
<KeyDict>A</KeyDict>
<KeyDict>E</KeyDict>
<KeyDict>H</KeyDict>
```

The value to set in the configuration item is defined within the `ValueDict` tag.

For example, to set the value 'v' in the item H,  the following is the piece of .XML:

```
<Set>
      <KeyDict>A</KeyDict>
      <KeyDict>E</KeyDict>
      <KeyDict>H</KeyDict>
      <ValueDict>v</ValueDict>
</Set>
```

If the value to set is a collection of values (e.g., the list of locations to run) the `ValueDict` tag must contain all the values, separated by a semicolon (';').

For example, to set the values 1, 2 and 3 in the item H:

```
<Set>
   <KeyDict>A</KeyDict>
   <KeyDict>E</KeyDict>
   <KeyDict>H</KeyDict>
   <ValueDict>1;2;3</ValueDict>
</Set>
```

Another example: The  sample  code below is the initial part of a standard BioMA modelling solution configuration.

There are two `Set` tags:

- The first allows setting a configuration item called '`Modeling solution`' to '`EC.JRC.MARS.ConfigurationLayer.Core.IterationFramework.ModelCallerIterator`'.
- The second tag sets the item called '`Modeling solution/ Modeling solution`' to '`EC.JRC.MARS.WOFOSTOperationalModelCaller.WOFOSTOperationalModelCaller`':

```
         <IConfigurable
class="JRC.IPSC.MARS.BIOMA.CORE2.RootModelCaller">
           <Id>0</Id>
           <Set>
             <KeyDict>Modeling Solution</KeyDict>
             <ValueDict>
               EC.JRC.MARS.ConfigurationLayer.Core.IterationFramewo
       rk.ModelCallerIterator
               </ValueDict>
           </Set>
           <Set>
             <KeyDict>Modeling Solution</KeyDict>
             <KeyDict>Modeling Solution</KeyDict>
             <ValueDict>
               EC.JRC.MARS.WOFOSTOperationalModelCaller.WOFOSTOpera
       tionalModelCaller
               </ValueDict>
           </Set>
       …

</IConfigurable>
```

## See also:

- “How to deploy a library within an application” on page 34
- “How to transform a composition layer modelling solution into a configuration layer’s one” on page 40

## How to deploy a library within an application

In order to be used inside an application, the *IConfigurable* object must be registered within the application's deployment registry file. For most of the BioMA applications, the file is called by default "`DeployedDlls.xml`". In BioMA Spatial, the name of the deployment file is configured in the application configuration main file.

The application writes in the deployment registry file the name and the path of the libraries (DLL or EXE files) containing the deployed *IConfigurable* objects.

When a library is registered, all the *IConfigurable* objects that are defined in the library are deployed and can be used by the application.

A library can contain more than one *IConfigurable* object, all of them become usable after the library is registered. If the user registers a library that do not contain any *IConfigurable* object, the library will be ignored.

It is also possible to register a set of libraries at the same time. To do so, the libraries must be zipped in a unique ZIP file, renaming the extension to ".bpkg" (BioMA Package file).

Most of the BioMA applications (e.g., BioMA Spatial) have an automatic procedure to help the user in deploying the libraries. (For further information, see the applications documentation here https://agri4cast.jrc.ec.europa.eu/DataPortal/Index.aspx?o=s).

The workflow to use an *IConfigurable* object is as follows:

1. The modeler writes the *IConfigurable* object (whatever function it has. It could be a data provider, a modelling solution, a persister, and so forth).
2. The modeler builds the library containing the *IConfigurable* object.
3. If there are many objects and libraries that should be used together, the user might create a BioMA Package file containing all the libraries.
4. The modeler deploys the libraries in the application following the application's specific procedure.
5. The modeler can use the *IConfigurable* object in the application.

### See also:

- "How to transform a composition layer modelling solution into a configuration layer's one" on page 40

## Persister mechanism

In the BioMA Framework, a persister is an *IConfigurable* object that has the function of saving the simulation results to a whatever form of persistence (a database, a CSV file, and so forth).

A valid persister class must implement the *IPersister* interface. This interface already extends the *IConfigurable* interface.

Like the other configurable objects, the persisters can be deployed into the BioMA applications and can be used from the application user interface. For example, in BioMA Spatial, it is possible to configure up to 4 persisters that allow saving the simulation results into 4 different forms of persistence. For example, the results can be saved simultaneously to a database and to a file.

A persister includes the logics to save both the simulation results and the simulation metadata to the form of persistence. The metadata saving is optional and can be left not implemented.

When saving the metadata, the model is passed to the persister using the *IAnnotatable* interface, which defines the metadata of the model. Furthermore, a textual description of the simulation and the serialization XML content are passed to the persister, as well. This way, it is possible to save in the form of persistence all the information that is needed to trace and recreate the characteristics of the simulation.

A persister can be declared "acceptable" for a certain kind of *ModelOutput*. Although a developer should try to develop a "universal" persister that works for every possible *ModelOutput*, the system offers the possibility of creating a customized persister that works only with a specific *ModelOutput*, as well as of defining the persister suitable only for that *ModelOutput*, using the *IsAcceptableFor* interface method.

The table below shows the methods of the *IPersister* interface:

| Icon | Member | Description |
|---|---|---|
| ◆ | StartingNewSavingSession() | Triggers the start of a new "saving session". A saving session must be identified by a unique simulation ID. When this method is invoked, a new simulation UD must replace the current one. The new simulation ID must be used until the *StartNewSavingSession* method is called again. The method returns the simulation ID. |
| ◆ | SaveOutputs(ModelOutput) | Saves the specified ModelOutput to the form of persistence. |
| ▤ | CurrentSimulationId | Returns the current simulation ID. |

| Icon | Member | Description |
|---|---|---|
| | SaveMedatada(IAnnotable, string, string) | Saves to the form of persistence the metadata of the specified *IAnnotable* class, the specified descriptive string, and the configuration string. This latter is a string that contains the XML content of the configuration serialization. |
| | IsAcceptableFor(ModelOutput output) | Returns True if the persister is able to manage the persistence of the specified ModelOutput. Otherwise, returns False. |
| | persisterMessageEventi | Event triggered when the persister must launch a message to the application that is using it. |

A set of common usage persisters are already available in the BioMA Framework applications:

### EC.JRC.MARS.DBPersister.DBPersister

This persister allows saving a *ModelOutput* to a database. It also saves the metadata relative to the model output. It uses the data layer classes for saving both the metadata and the data (see the **Data Layer Documentation**). Its configuration contains 3 items:

- **Mapping file path**: Path to the mapping file of the data layer.
- **Connection**: A *ConnectionConfigurationItem* that contains the configuration of the connection of the DB (connection string + connection provider).
- **Save rules**: An object that implements *ISaveRule.ISaveRule* for managing the rules to save the data of a table. *DBPersister* will only save the table's rows that match the configured save rule.

### EC.JRC.MARS.Persister.CSVPersister

This persister save the *ModelOutput* to a set of CSV files. It saves to a different file each table that is contained in the *ModelOutput*. It does not save the simulation metadata, and its simulation ID is always 0. Its configuration has 2 configuration items:

- **Directory**: It specifies the directory where to save the CSV file(s).
- **File name prefix**: It specifies the prefix for the CSV file name(s).

### EC.JRC.MARS.Persister.XmlPersisterAllDataMetadataList

This persister saves to a set of XML files. The name of the files are automatically created on the basis of the configured file name prefix and the current timestamp.

The persister must be configured with the path where to save the file and the prefix to be used to create the name of the file. The persister creates a file containing the metadata (.MET) and a file containing the XML serialization of a *System.Data.DataSet* object. The *DataSet* is extracted from the ModelOutput through the *DataCollection* structure. Moreover, the persister creates an index file (.BRI) that indicates, in case of iteration, the list of the XML files created.

The following shows an example of metadata file (.MET):

```
<?xml version="1.0" encoding="utf-8"
standalone="yes"?><Index><Title>Index of results logs for
simulation started at 2012-10-26_16-14-38. Description:'test'</
Title>
<Metadata>
<MetadataValue>
<field>Check preconditions</field>
<value>Disable</value>
</MetadataValue>
<MetadataValue>
<field>Simulation Configuration</field>
<value>EC.JRC.MARS.SimulationControl.CompositionLayerLibrary.Day
ByDayStartYearFromListSimulationControlProvider</value>
</MetadataValue>
<MetadataValue>
<field>WeatherProviderSimulationComponent Configuration</field>
<value>JRC.IPSC.MARS.WeatherProviders.Library.CgmsOracleEuropeWe
atherProvider</value>
</MetadataValue>
<MetadataValue>
<field>AgroManagementSimulationComponent Configuration</field>
<value>JRC.IPSC.MARS.CommonAgromanagementProviders.XMLFileAgroma
nagementProvider</value>
</MetadataValue>
<MetadataValue>
<field>SoilDataProviderSimulationComponent Configuration</field>
<value>JRC.IPSC.MARS.SoilDataProvider.Library.MPEFilesSoilDataP
rovider</value>
</MetadataValue>
<MetadataValue>
<field>CropSystPot Switch UseVernalization</field>
<value>false</value>
</MetadataValue>
<MetadataValue>
<field>CropSystPot Switch UseCO2</field>
<value>false</value>
</MetadataValue>
<MetadataValue>
<field>CropSystPot Switch IsC3</field>
```

```
<value>true</value>
</MetadataValue>
<MetadataValue>
<field>CropSystPot Switch UsePhotoPeriod</field>
<value>false</value>
</MetadataValue>
<MetadataValue>
<field>CropSystPot Switch UseTemperature</field>
<value>true</value>
</MetadataValue>
<MetadataValue>
<field>Parameters configuration</field>
<value>Parameters configuration updated</value>
</MetadataValue>
</Metadata>
```

The following shows an example of a data file (.XML). It is the serializations of a *System.Data.DataSet* object:

```
<NewDataSet>
  <WeatherProvider_Grid_weather>
    <Step>0</Step>
    <LocationID>65219</LocationID>
    <Date>01/01/1997</Date>
    <DOY>1</DOY>
    <Year>1997</Year>
    <Calculated_radiation>8215</Calculated_radiation>
    <Day>1997-01-01T00:00:00+01:00</Day>
    <E0>0.68445062</E0>
    <Grid_no>65219</Grid_no>
    <Maximum_temperature>13.40</Maximum_temperature>
    <Minimum_temperature>5.80</Minimum_temperature>
    <Minimum_temperatureDayAfter>0</Minimum_temperatureDayAfter>
    <Rainfall>0</Rainfall>
    <Snow_depth>0</Snow_depth>
    <Vapour_pressure>7.69</Vapour_pressure>
    <Windspeed>1.70</Windspeed>
    <Hmax>0</Hmax>
    <Hmin>0</Hmin>
    <IsHumidityPresent>false</IsHumidityPresent>
    <IsVpdPresent>false</IsVpdPresent>
    <IsE0Present>true</IsE0Present>
  </WeatherProvider_Grid_weather>
…etc…
```

### EC.JRC.MARS.Persister.ModelOutputCumulator

This persister cumulates many *ModelOutput* into a single *ModelOutput* object. The final *ModelOutput* is exposed as a public property of the class, which makes it available to other parts of the application. It does not save the simulation metadata and its simulation ID is always 0.

## How to transform a composition layer modelling solution into a configuration layer's one

A common issue when creating a new modelling solution is to transform an existing composition layer modelling solution (see **Composition Layer Documentation**) into a configuration layer modelling solution, in order to run it in the BioMA applications.

This chapter describes how to do this through a simple procedure.

**Note:**

If you are creating the composition layer modelling solution using the CLIC application, CLIC itself generates the configuration layer modelling solution too, meaning that no other effort is required by the user.

A composition layer modelling solution is represented by a valid implementation of the `EC.JRC.MARS.CompositionLayer.Core.ModelRunner` abstract class.

A configuration layer modelling solution is represented by a valid implementation of the *IModelCaller* interface.

An adapter between the two implementations is made available within the BioMA Framework. Its name is: `EC.JRC.MARS.ConfigurationLayer.ModelCallerAdapter4CompositionLayer.AModelCallerAdapter4Composition`.

This adapter class implements *IModelCaller*, so it is a valid configuration layer modelling solution. It is built using the implementation of the *ModelRunner*: the *ModelRunner* instance is set into the adapter instance during the call of the adapter constructor.

Automatically, all the aspects of the *ModelRunner* that need a configuration (e.g., the switches) are transformed into configuration items.

Using this adapter class, the conversion between the two layers is automatic. The following code snippet, shows the code to transform a *ModelRunner* (`CropSystModelRunner`) into an *IModelCaller* (`CropSystModelCaller`):

```
/// <summary>
/// Transform CropSystModelRunner into CropSystModelCallerb using
the composition layer - configuration layer adapter.
```

```
/// CropSystModelCaller must extend
EC.JRC.MARS.ConfigurationLayer.ModelCallerAdapter4CompositionLaye
r.AModelCallerAdapter4Composition
/// </summary>
    public class CropSystModelCaller :
AModelCallerAdapter4Composition
    {
        /// <summary>
        /// In the constructor, instantiate the modelToRun
protected property
        /// (of type ModelRunner) to the concrete model runner
(type CropSystModelRunner).
       /// Then call the InitializeConfigurationStructures method.
        /// </summary>
        public CropSystModelCaller()
        {

            modelToRun= new CropSystModelRunner();
            this.InitializeConfigurationStructures();

        }

        /// <summary>
        /// The OutputStructure of the model runner is exposed
through this public property
        /// </summary>
        public OutputStructure ModelRunnerOutputStructure
        {
            get { return modelToRun.OutputStructure; }
        }

        #region Overrides of AModelCallerAdapter4Composition

        public override string Description
        {
            get { return "CRopSyst model caller"; }
        }

        public override string URL
        {
            get { return "http://"; }
        }
        public override XmlElement Metadata
        {
            get { throw new NotImplementedException(); }
        }

        #endregion
    }
```