# IBM Functions

IBM functions implement the interface wts.models.DisMELS.framework.IBMFunctions.IBMFunctionInterface. The abstract class AbstractIBMFunctions in the same package provides a convenience implementation of many of the methods required by the interface, as well as some “plumbing”, to ease the task of creating a new IBMFunction. Classes that implement the interface should probably extend the abstract class as discussed below.

Create a java class that extends the abstract superclass wts.models.DisMELS.framework.IBMFunctions.AbstractIBMFunction and implements all abstract methods, including

* 1. Object clone(): this should return a copy of the IBMFunction object on which it is called.
  2. String getFunctionType(): should return the user-friendly type (category) of IBM function this class is used for.
  3. String getFunctionName(): should return a user-friendly name for the function.
  4. String getDescription(): should return a user-friendly description of the function.
  5. void setParameterValue(String name, Object value): should set the local value (if any) of the parameter with the given name. Call super.setParameterValue(name,value) in the overridden method to set the value of the IBMParameter in the superclass.
  6. Object calculate(Object vars): should calculate the function based on the input variables (vars) and return the output as an Object (subclasses can limit the scope of the return class further; e.g. as a Double).

As a convenience, override void setParameterValue(String name, Object value) to set the local value (if any) of the parameter with the given name. If you do this, be sure to call the overridden superclass method super.setParameterValue(name, value) to set the value of the IBMParameter in the superclass. Otherwise the value will not be saved or displayed correctly.

In the java file,

* 1. copy the lines

@ServiceProviders(value={

@ServiceProvider(service=IBMFunctionInterface.class)}

)

before the class definition (e.g. ‘public class myClass extends AbstractIBMFunction {‘, and add the associated imports.

* 1. Create two public static final int’s ‘numParams’ (the number of parameters defined for the function) and ‘numSubFuncs’ (the number of subfunctions defined).
  2. In the constructor for the class, initialize the superclass (AbstractIBMFunction) using ‘super(numParams, numSubFuncs);’ to set the number of parameters and subfunctions in the superclass.
  3. Create and add each parameter using the superclass function ‘addParameter(String name, Class clazz, String description)’ or ‘addParameter(String name, Object value, String description)’.
  4. Create and add each subfunction using the superclass function ‘setSubfunction(String name, IBMFunctionInterface subfunction)’.

Create the java beans info class corresponding to the IBMFunction java class (in Netbeans, right click on the filename in the Projects window and select “BeanInfo Editor…”

* 1. Edit the resulting BeanInfo java file and add the following line below the comment “// Here you can add code for customizing the BeanDescriptor.”:

beanDescriptor.setValue("persistenceDelegate",new AbstractIBMFunctionPersistenceDelegate());

# Life Stage Class

Life stage classes implement the wts.models.DisMELS.framework.LifeStageInterface. The AbstractLHS class in the same package provides a convenience implementation of many of the methods required by the interface, as well as some “plumbing”, to ease the task of creating a new life stage. Classes that implement the interface should probably extend the abstract class as discussed below.

Life stage classes encapsulate two related classes: a life stage parameters class and a life stage attributes class. The former is concerned with information related to ALL individuals within a given life stage (i.e., parameters and functions for processes describing all individuals within the stage) while the latter is concerned with information related to EACH particular (super) individual (i.e., characteristics of the individual’s physiology and environment). Life stage parameter classes implement the wts.models.DisMELS.framework.LifeStageParametersInterface and life stage attributes classes implement the wts.models.DisMELS.framework.LifeStageAttributesInterface. The tasks of developing these classes for a new life stage are discussed in other sections.

Life stage classes must also reference one other class, a “PointFeatureType” class. This class is required to visualize results and interact with the MapViewer user interface. Its development is discussed in the “PointFeatureType” section.

Life stage classes may also (but don’t have to) reference other life stage classes as either “next” life stage classes or “spawned” life stage classes. The former consist of life stage classes that members of a life stage class can metamorphose into (e.g., from “egg” to “larva”) while the latter consist of classes that members of a life stage class can create and release during a model run (e.g., an “adult” releasing an “egg”).

Creating a new life stage class by copying an existing one

This is the simplest approach to developing a new life stage class. Assuming all the files associated with a life stage class are grouped in the same java package, copy the

# PointFeatureType Class

A PointFeatureType class is required to visualize model results and to interact with the Map Viewer through the user interface. Its class name is given as part of the definition of a new life stage class.

The PointFeatureType class associated with a given life stage should extend the superclass wts.models.DisMELS.framework.LHSPointFeatureType. Two constructors for the subclass should be defined, one which takes a life stage attributes object of the type associated with the life stage class in question, and one which takes the name of an LHS\_Type.