How to extend GEOPHIRES X

1. Decide which object(s) (Reservoir, Wellbores, Surface Plant, and/or Economics) you are going to extend. In this example, I will extend Economics.
2. Make a new file named the same as the class name you will use. In this case, I will create EcononomicsAddons. Add it to your project if you are using a development environment like PyCharm or Visual Studio.
3. In the Models class, add an import statement for the class you are making. In this case, the line looks like this:

from EconomicsAddons import \*

1. In the \_\_init\_\_ method of the Models class, initialize your new class. In this case, the line looks like this:

self.economics = EconomicsAddOns(self)

1. Fill that new file with this template, changing the class name and imports as appropriate:

import math

import sys

import numpy as np

import numpy\_financial as npf

import AdvModel

import Economics

from OptionList import EndUseOptions

from Parameter import intParameter, floatParameter, listParameter, OutputParameter

from Units import \*

class EconomicsAddOns(Economics.Economics):

def \_\_init\_\_(self, model):

model.logger.info("Init " + str(\_\_class\_\_) + ": " + sys.\_getframe( ).f\_code.co\_name)

#Set up all the Parameters that will be predefined by this class using

#the different types of parameter classes.

pass

#local variables that need initialization

pass

#results

pass

model.logger.info("Complete "+ str(\_\_class\_\_) + ": " + sys.\_getframe( ).f\_code.co\_name)

def \_\_str\_\_(self):

return "EconomicsAddOns"

def read\_parameters(self, model) -> None:

model.logger.info("Init " + str(\_\_class\_\_) + ": " + sys.\_getframe( ).f\_code.co\_name)

#Deal with all the parameter values that the user has provided.

pass

model.logger.info("complete "+ str(\_\_class\_\_) + ": " + sys.\_getframe( ).f\_code.co\_name)

def Calculate(self, reserv, wellbores, surfaceplant, model) -> None:

model.logger.info("Init " + str(\_\_class\_\_) + ": " + sys.\_getframe( ).f\_code.co\_name)

#This is where all the calculations are made using all the values that

#have been set.

pass

model.logger.info("complete "+ str(\_\_class\_\_) + ": " + sys.\_getframe( ).f\_code.co\_name)

1. Note the class definition: “class EconomicsAddOns(Economics.Economics):” – it must contain a reference to the parent class (in this case, Economics). Set it to the appropriate class for your needs. Note that multiple inheritances are also possible but not needed in this case.
2. Also note the import command “from Economics import \*” – it imports all the information about the parent class. Set it to the appropriate class for your needs.
3. Note that the “model” class is passed into all these methods. This is the wrapper class in which all the objects live. It contains values that are useful to all classes, like “logger”.
4. For the \_\_init\_\_method, you need to decide if you want to initialize the parent class (in this case, Economics), or not. Initializing it means that all the Parameters and variables in the parent class will be created and will be available for you to use in your methods. If you don’t want those variables and methods, don’t initialize the parent. The parent is initialized by adding the following line of code to the \_\_init\_\_ method right at the beginning, right after the logging is started, in the middle as needed, or right at the end, right before the logging stops. You would choose to initial at the beginning if some of the parent parameters/variables will be used in your initialization. If not, you can do it at the end. For initialization, it probably doesn’t matter when you call it.

super().\_\_init\_\_(model)

1. The read\_parameter method checks the list of parameters that the user has specified new values for in the text file and updates the class parameters with those values after validating them. It also allows programmers ti deal with any special cases that arise when the user changes a value – a change of value to one parameter might require an update to another unrelated Parameter. For the read\_parameters method, you need to make the same choice about running the parent class method of the same name, or not. If you initialized the parameters of the parent in \_\_init\_\_, you should probably read the user Parameters for any changes that the user wants to make to those parameters. Use this call to do that:

super().read\_parameters(model)

1. For the Calculate method, make the same choice about running the parent class method of the same name, or not. If you initialized the parameters of the parent in \_\_init\_\_, and read the parameters, you should probably Calculate the values based on those parameters. Those results and available to you in your calculations in this class if you do this. Use this call to do that:

super().Calculate(model)

1. Note that for the Calculate method, the model class is passed in to give access to the logger but also to all the other classes (reserve, surfaceplant, etc) since they are attributes of the Model wrapper class. Calculations tend to depend on the other classes. In the case, my Economic AddOns use information for nearly all the other classes.
2. Now start coding your methods. In the \_\_Init\_\_ method, you need to decide what your Parameters will be. For each one, you need to use the appropriate class constructor; for an integer, intParameter; for a float, floatParameter; etc. For each Parameter, you must specify its name, value, default value, and valid range (if int or float). Optionally, you can specify:
   1. Required (Boolean): is it required to run? default value = False
   2. ErrMessage (string): what GEOPHIRES will report if the value provided is invalid. Default = "assume default value (see manual)")
   3. ToolTipText (string): when there is a GUI, this is the text that the user will see. Default = "This is ToolTip Text")
   4. UnitType (Unit Type enumeration): the type of units associated with this parameter (length, temperature, density, etc). Default = Units.NONE
   5. CurrentUnits (Unit enumeration): what the units are for this parameter (meters, Celcius, gm/cc, etc. Default = Units:NONE)
   6. PreferredUnits (units: usually equal to CurrentUnits, but these are the units that the calculations assume when running. Default - Units.NONE
3. UnitType, CurrentUnits, and PreferredUnits are the attributes that allow GEOPHIRESX to handle unit and currency conversions. If you don’t want to use that functionality, don’t use them. If you do, see the code examples to see how this works.
4. In the \_\_Init\_\_ method, you must also decide what your local variables will be, and what values they will start with.
5. In the \_\_Init\_\_ method, you need to decide what your OutputParameters will be (they will be calculated with your Calculate method and will be available to other classes for use and output). For each one, you need to use the class constructor OutputParameter. You must set its name and value. Note that value is of type “Any” – that means it can be assigned an int, float, bool, list, etc. Optionally, you can set:
   1. ToolTipText: see above
   2. UnitType: see above
   3. PreferredUnits: see above
   4. CurrentUnits: See above
6. In the \_\_init\_\_ method, note the use of two dictionaries: ParameterDict and OutputParameterDict. When a Parameter or OutputParameter is created, it is also added to the dictionary. These dictionaries are publicly available and give access to all the parameters. These get used in several ways, so stick to the convention of using them as you see them used in the parent classes.
7. In the read\_parameter method, you need to decide if any of your parameters need special processing once they have been read in and modified by a user value change. The ReadParameter() utility function should be used to deal with all the parameters read in that apply to your Object, but if a change to any of your parameters triggers other actions, insert code here to handle those actions – see parent classes for how that is done.
8. In the Calculate method, insert the code you need to make your calculations. You can use the input parameters, local variables, all parameters (input and output) from other classes in your calculations, but note:
   1. Think carefully about the ordering of the calculations, and when the values you wish to use are valid. If you are extending the Reservoir object, note that the parent Reservoir output parameters are only valid after the parent class Calculate method has been run. It may also be possible that output values from one class may be altered later by the Calculate method on other classes. GEOPHIRES-X core code tries to avoid this, as it is confusing, but it is possible, so know your variables!
   2. The parent class as input parameters which will be set to valid default values after the parent \_\_init\_\_ method is called, but note that any of these values could be changed when the read\_parameter method for that class is called. And other unrelated parameters might also change due to dependencies, so don’t rely on the input parameters to be finalized until after read\_parameter on the parent has run. Normally, input parameters for a class don’t change after read\_parameter for that class has run, but it does happen sometimes. GEOPHIRES-X core code tries to avoid this, as it is confusing, but it is possible, so know your variables!
   3. Be careful how you modify the class variables. If you modify a parent variable or parameter by referring to it using the “self.” construct, then you are modifying the local copy of it associated with it in your class, just like if you run a method of a class using the self.method\_name() construction, you are running the local class copy of your method (running any changes you made as well). If you don’t override the method, then you will run the parent method, even if you refer to it with self. If you want to access or modify the parent variables, Parameters, or methods, you can refer to them explicitly – recall that the model class is passed into your class and its mthods, so you can access model.reserv, which would give you direct access to the parent Reservoir model, model.surfaceplant to refer to SurfacePlant, and so on.
9. Once you are done with your Calculations, you also are likely to want to show them to your users. This is usually accomplished by creating an OuputClass that has the sole job of writing your results to the output file. In this case, look at the class OutputsAddOns. Note that its parent is Outputs, in which the outputs for the base classes are integrated and reported. The method PrintOutputs open the output file (HDR.out) and uses formatted text strings to write values into the file. Note that you can write single values, or loop thru arrays of values. You can also access and report values from other classes and parents – especially if your Calculate modified them. You should assume that all the outputs from the other classes were reported before you modified them. For example, the Net Present Value (NPV) of the project is recalculated in the EconomicAddons method of my extension because my economic AddOns changes the income, expenses, and profits of the project. I assume that the NPV value has been written to the output file value already (and it represents the NPV of the project before the AddOns). I report the NPV again when I report the outputs of my class, and I note in the text that this is an update to Project NPV based on the AddOns. To make sure of that logic, I have a local output parameter called NPV and I modify and report that without change the NPV output parameter in the Economics class.