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 and 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

from GeoPHIRESUtils import \*

import logging

import traceback

from OptionList import \*

from Parameter import \*

from Result import \*

from Units import \*

from Parameter import \*

import numpy as np

from Economics import \*

class EconomicsAddOns(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):” – it must contain a reference to the parent class (in this case, Economics). Set it to the appropriate class for your needs.
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, 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.

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

1. For the read\_parameters 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\_\_, you should probably read the 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(reserv, wellbores, surfaceplant, model)

1. Note that for the Calculate method, all the other classes are passed in so you will have access to all the parameters and variables from those classes. Calculations tend to depend on 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: is it required to run? default value = False
   2. ErrMessage: what GEOPHIRES will report if the value provided is invalid. Default = "assume default value (see manual)")
   3. ToolTipText: when there is a GUI, this is the text that the user will see. Default = "This is ToolTip Text")
   4. UnitType: the type of units associated with this parameter (length, temperature, density, etc). Default = Units.NONE
   5. CurrentUnits: what the units are for this parameter (meters, Celcius, gm/cc, etc. Default = Units:NONE)
   6. PreferredUnits: 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 allows GEOPHIRESX to handle unit and currency conversions. If you don’t want In 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 (will be calculated with the Calculate method and will be available for 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, you will 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.