Adding a Model Object to OpenStudio

This document explains how to add a model object to OpenStudio. It assumes that you have already checked out and built OpenStudio as described in the document “Configuring OpenStudio Build Environments.docx”.

Contents

[1.0 Getting Started 2](#_Toc348088751)

[1.1 Version Control 2](#_Toc348088752)

[1.2 Using SVN with OpenStudio 2](#_Toc348088753)

[1.3 Initial Build 4](#_Toc348088754)

[2.0 Design Your Work First 5](#_Toc348088755)

[3.0 Editing the OpenStudio.idd file 6](#_Toc348088756)

[4.0 Model Object Source Code Files 11](#_Toc348088757)

[4.1 ObjectName.hpp and the ObjectName\_Impl.hpp 11](#_Toc348088758)

[4.2 ObjectName.cpp 11](#_Toc348088759)

[4.3 Generating Source Code Files as a Starting Point 12](#_Toc348088760)

[4.4 Making Your Object Available to the Rest of OpenStudio 12](#_Toc348088761)

[4.5 Finishing the Source Code Files 13](#_Toc348088762)

[4.5.1 C++ Cheat Sheet 14](#_Toc348088763)

[5.0 Testing Your Code 17](#_Toc348088764)

[5.1 Creating a Test File 17](#_Toc348088765)

[5.2 Writing the Tests 17](#_Toc348088766)

[5.3 Running the Tests 18](#_Toc348088767)

[6.0 Forward Translation – OpenStudio to EnergyPlus 19](#_Toc348088768)

[6.1 Adding the ForwardTranslate File 19](#_Toc348088769)

[6.2 Invoking Translation for the New Object 20](#_Toc348088770)

[7.0 Show the New Object in OpenStudio App 21](#_Toc348088771)

1.0 Getting Started 2

1.1 Version Control 2

1.2 Using SVN with OpenStudio 2

1.3 Initial Build 4

2.0 Design Your Work First 5

3.0 Editing the OpenStudio.idd file 6

4.0 Model Object Source Code Files 10

4.1 ObjectName.hpp and the ObjectName\_Impl.hpp 10

4.2 ObjectName.cpp 10

4.3 Generating Source Code Files as a Starting Point 11

4.4 Making Your Object Available to the Rest of OpenStudio 11

4.5 Finishing the Source Code Files 12

4.5.1 C++ Cheat Sheet 13

5.0 Testing Your Code 15

5.1 Creating a Test File 15

5.2 Writing the Tests 15

5.3 Running the Tests 16

6.0 Forward Translation – OpenStudio to EnergyPlus 17

6.1 Adding the ForwardTranslate File 17

6.2 Invoking Translation for the New Object 18

7.0 Show the New Object in OpenStudio App 19

8.0 Create New Functional Tests 22

8.1 Create API Functional Tests 22

8.2 Create OSM File Functional Tests 23

# Getting Started

## Version Control

The OpenStudio source code files are stored using a version control system called SVN. This version control system stores previous versions of all the source code files. When you make edits to files, you do it locally (on your computer). Once you have tested the files and confirmed that your changes are working the way you want and didn’t break anything else, you Commit these changed files back to the version control system. Committing your files creates a backup point that you can revert back to and also allows other people to get your changes.

## Using SVN with OpenStudio

Here is the basic outline of the process used when working on the OpenStudio source code. More information about SVN is available online.

1. The Trunk is the official version of the source code. When official installers are made, the files on the Trunk are used to make them.
2. The first step in adding a new feature to OpenStudio is to create a Branch for your particular changes. A Branch is a copy of the Trunk source code dedicated to your new feature. This allows you to keep your files under version control without risking that your changes will break anything in the official Trunk source code yet. You will want a local checkout of your Branch to work with.
3. Make changes to the files in your Branch on your local computer. When you get to a point where you want the files backed-up, you Commit them to your Branch. If you later mess up the files on your computer, it’s OK; you simply Revert back to the last version that was saved to your Branch. Continue making changes to files on your computer and committing them to your Branch until you are satisfied that everything works the way you want it to.
4. Once you are satisfied that your Branch is good, ask someone to test it. They download your Branch and test it. They review the code and probably ask you to make some changes/fixes.
5. You make the changes the reviewer requested on your Branch.
6. Now, the Branch is ready for primetime. You Reintegrate your Branch into the Trunk. This takes the changes you have made on your Branch and puts them into the official Trunk source code. Sometimes there are Conflicts when you Reintegrate a Branch. For example, someone else could have been editing the same files as you, but on a different Branch. Whoever Reintegrates the Branch into the Trunk will need to resolve these Conflicts.

This diagram shows the basic process:

**Trunk**

**My Branch**

**My Computer**

**My Computer**

The following instructions assume you’ve already installed Tortoise SVN and have gotten a username and password for the OpenStudio source code SVN, following the instructions in the document on setting up your build environment.

1. Create a folder called “OpenStudio\_branch” on your computer. For this document, the folder is located at C:\Projects\OpenStudio\_branch
2. Right click on the folder, click “SVN Checkout”
3. In URL of repository, paste: <https://cbr.nrel.gov/openstudio/svn/trunk>.
4. Click OK. You will be prompted to enter your username and password. Check the box to have Tortoise SVN remember your password. Click OK to check out a copy of the Trunk source code.
5. Now, make your Branch. You will be given an identifier number for your Branch. Right-click on the OpenStudio\_branch folder, TotoiseSVN > Branch/Tag. Enter in:

From WC/URL: <https://cbr.nrel.gov/openstudio/svn/trunk>

To path: /branches/42076397\_model\_object\_adding\_guide

Log Message: [#42076397] starting a branch to demonstrate how to wrap a model object

(Note: replace the number with your Branch identifier and put a really short description of your work in the title. In the log message, put your branch identifier in the brackets with the # sign as shown, then put a brief description of what work will happen on this branch.)

1. Switch over to the Branch you just created by right-clicking on the OpenStudio\_branch folder and selecting TotoiseSVN > Switch.

To path: /branches/42076397\_model\_object\_adding\_guide

Congratulations! You have made a branch and you can now start working on changing OpenStudio by editing the source code files!

## Initial Build

Now that you’ve checked out the source code you will need to make an initial build of the software to work on. This initial build will take a long time because it will also build OpenStudio’s dependencies, you will not have to build these dependencies each time you build OpenStudio in the future. These instructions assume you have installed CMake (cmake-gui) and also Visual Studio per the instructions in “Configuring OpenStudio Build Environments.docx”.

1. Open: C:\Projects\OpenStudio\_branch and make a new folder called “build”
2. Open: CMake (cmake-gui)
   1. Where is the source code: C:/Projects/OpenStudio\_branch
   2. Where to build the binaries: C:/Projects/OpenStudio\_branch/build
3. Make sure that BUILD\_TESTING is checked.
4. Configure > Visual Studio 9 2008 > Use default native compilers > Finish
5. Keep clicking Configure until all red lines are gone.
6. Generate
7. Open: C:\Projects\OpenStudio\_branch\build\OpenStudio.sln with Visual Studio
8. In Visual Studio, Right-Click ALL\_BUILD > Build
9. Wait – the first time you build it takes about 4 hours. Do this overnight.
10. Once you have built OpenStudio.sln once, you don’t have to touch it again.
11. Open: C:\Projects\OpenStudio\_branch\build\OpenStudioCore-prefix\src\OpenStudioCore-build\OpenStudioCore.sln. This is what you will be building from now on.
12. In Visual Studio: Right-Click ALL\_BUILD > Build. Any build errors will show up in the output window of Visual Studio. Clicking on the error will show you the line that is failing to compile. The error description usually explains what is wrong. Most times, it is a missing semicolon or squiggly bracket.

# Design Your Work First

The first step in adding a new model object to OpenStudio is to think about what the object will do. Think about why the object is being added, what OpenStudio user interfaces the object can be accessed from, what other objects the new object will have to interact with, what special functionalities or operations will the new object need. The whole purpose of the OpenStudio object model is to provide intelligent access to the OpenStudio model, this step is where we need to think about what intelligence this new object needs. A design document should be developed which describes what fields the new object will have, what special methods it will have, and how the user will interact with the object in the OpenStudio user interfaces. This document does not have to be very formal or in depth, a member of the OpenStudio team can help to review and refine this design. Only after thinking through the initial design should you proceed to the next steps which implement that design.

# Editing the OpenStudio.idd file

EnergyPlus uses an .idd file that describes the fields that each object has, what can go in those fields, and provides defaults. OpenStudio also has an .idd file that serves the same purpose; however, the EnergyPlus and OpenStudio .idd files aren’t the same. Thinking through what fields each object has (e.g. the data that is saved in an OSM file) is an important part of the design of a new OpenStudio object. In many cases the new OpenStudio object will have very similar fields to an EnergyPlus object. However, there are also cases where the OpenStudio object does not need certain fields, needs additional fields, or the object does not correspond to an EnergyPlus object at all! Thinking about these concerns in the design is very important because if the fields of an OpenStudio object change the model object will need to be changed and code will have to be implemented to version translate old versions of the object to the new version.

Once the object design is complete and all the fields are known we can add a definition of the new object to the OpenStudio.idd file. Depending on how similar the object is to an existing EnergyPlus object it may be useful to copy the idd entry for the EnergyPlus object to use as a starting point for the OpenStudio idd object.

The OpenStudio.idd can be found at:

C:\Projects\OpenStudio\_branch\openstudiocore\resources\model\OpenStudio.idd

Some particular notes:

* + OpenStudio idd objects are prefixed with OS:
  + OpenStudio idd objects have a !handle field as the first field, you can copy this from another OpenStudio idd object to the top of the new idd object
  + If you used an existing EnergyPlus idd object as a starting point you will have to update field numbers as field indices change
  + OpenStudio tracks node names so you must add the \reference ConnectionObject to the !name field of HVAC components and \object-list ConnectionNames to nodes
  + You MUST leave **two** blank lines at the end of the OpenStudio.idd

From:

A4 , \field Air Outlet Node Name

\required-field

\type node

To:

A3 , \field Air Inlet Node Name

\required-field

\type object-list

\object-list ConnectionNames

Below is an example of the same object in the Energy+.idd and the OpenStudio.idd file.

The Energy+.idd.

Fan:ConstantVolume,

\min-fields 9

A1 , \field Name

\required-field

\reference Fans

\reference FansCV

\reference FansCVandOnOff

\reference FansCVandVAV

\reference FansCVandOnOffandVAV

A2 , \field Availability Schedule Name

\required-field

\type object-list

\object-list ScheduleNames

N1 , \field Fan Efficiency

\type real

\default 0.7

\minimum> 0.0

\maximum 1.0

N2 , \field Pressure Rise

\units Pa

\ip-units inH2O

\required-field

N3 , \field Maximum Flow Rate

\units m3/s

\minimum 0.0

\autosizable

N4 , \field Motor Efficiency

\type real

\default 0.9

\minimum> 0.0

\maximum 1.0

N5 , \field Motor In Airstream Fraction

\note 0.0 means fan motor outside of air stream, 1.0 means motor inside of air stream

\type real

\default 1.0

\minimum 0.0

\maximum 1.0

A3 , \field Air Inlet Node Name

\type node

\required-field

A4 , \field Air Outlet Node Name

\type node

\required-field

A5 ; \field End-Use Subcategory

\type alpha

\retaincase

\default General

The same object in the OpenStudio.idd

OS:Fan:ConstantVolume,

\min-fields 10

A1, \field Handle

\type handle

\required-field

A2, \field Name

\type alpha

\required-field

\reference Fans

\reference FansCV

\reference FansCVandOnOff

\reference FansCVandVAV

\reference FansCVandOnOffandVAV

\reference ConnectionObject

A3, \field Availability Schedule Name

\type object-list

\required-field

\object-list ScheduleNames

N1, \field Fan Efficiency

\type real

\minimum> 0

\maximum 1

\default 0.7

N2, \field Pressure Rise

\type real

\units Pa

\ip-units inH2O

\default 250.0

N3, \field Maximum Flow Rate

\type real

\autosizable

\units m3/s

\minimum 0

N4, \field Motor Efficiency

\type real

\minimum> 0

\maximum 1

\default 0.9

N5, \field Motor In Airstream Fraction

\note 0.0 means fan motor outside of air stream, 1.0 means motor inside of air stream

\type real

\minimum 0

\maximum 1

\default 1.0

A4, \field Air Inlet Node Name

\type object-list

\required-field

\object-list ConnectionNames

A5, \field Air Outlet Node Name

\type object-list

\required-field

\object-list ConnectionNames

A6; \field End-Use Subcategory

\type alpha

\retaincase

\default General

Once you have completed the idd entry for your OpenStudio model object you will need to build the OpenStudioCore project. This is because a code generation step parses the idd and creates source files that change whenever the OpenStudio.idd file is updated.

# Model Object Source Code Files

OpenStudio model objects are written using a “pointer to implementation” or pImpl idiom. This means that each model object has a public wrapper class (which is what other people will use when programming with your model object) and a hidden impl class that actually modifies the data for the model object. Multiple public wrappers may point to the same hidden impl as shown below:

|  |  |
| --- | --- |
|  | The source code for a model object is divided into 3 main files:  ObjectName.hpp – public wrapper interface declaration  ObjectName\_Impl.hpp – hidden impl interface declaration  ObjectName.cpp – implementation code for public wrapper and impl |

## ObjectName.hpp and the ObjectName\_Impl.hpp

ObjectName.hpp and the ObjectName\_Impl.hpp files contain the declarations of the methods that the object has. For example, FanConstantVolume has methods like this, which return the fan efficiency as a double (number).

/\*\* Returns the value of the FanEfficiency field. \*\*/

double fanEfficiency**();**

And methods like this, which set the fan efficiency when given a number (double).

/\*\* Sets the value of the FanEfficiency field. \*\*/

void setFanEfficiency**(**double value**);**

## ObjectName.cpp

ObjectName.cpp is where the methods declared in the other files are actually written. For example, here is how setFanEfficiency is defined in FanConstantVolume.cpp.

void FanConstantVolume\_Impl**::**setFanEfficiency**(**double val**)**

**{**

**this->**setDouble**(**OS\_Fan\_ConstantVolumeFields**::**FanEfficiency**,**val**);**

**}**

## Generating Source Code Files as a Starting Point

The files that are needed to describe a model object can be written by hand. However, there is a script that can be used to generate files as a starting point. These files won’t have all of the functionality that needs to be included, but will start you off of the right path. The project will need to be built using the latest OpenStudio.idd file before running this script in order to correctly generate classes for your object.

Note: the first “ObjectName” as an EnergyPlus object doesnot have any colon in between. The second “OS:ObjectName”. For information on class inheritance, go to <http://openstudio.nrel.gov/latest-c-sdk-documentation/model>, click a HVAC component class on the left window and then click *inheritance diagram* in the right window.

cd C:\\OpenStudio\developer\ruby

ruby -I "C:\Projects\OpenStudio\_branch\build\OSCore-prefix\src\OSCore-build\ruby\Debug" GenerateClass.rb -s "model" -c "ObjectName " -b "BaseClassName" -o " C:\Projects\OpenStudio\_branch\openstudiocore\src\model" -p -q -i "OS:ObjectName"

example:

ruby -I "C:\OpenStudio\build\OSCore-prefix\src\OSCore-build\ruby\Debug" GenerateClass.rb -s "model" -c "GeneratorMicroTurbine" -b "Generator" -o "C: \OpenStudio\openstudiocore\src\model" -p -q -i "OS:Generator:MicroTurbine"

1. Open a command prompt
2. Paste first command above and press Enter.
3. Paste second command above, replacing ObjectName with the actual object name and BaseClassName with the desired base class name. The OS:ObjectName should match the OpenStudio.idd. Press Enter.
4. Open: C:\Projects\OpenStudio\_branch\openstudiocore\src\model. You should see 3 new files for your component.
5. Commit these files to your Branch by Right-Clicking on the files and selecting SVN Add then clicking SVN Commit. Remember, you’ll need the [#identifier] in the commit message, along with a description of what you are doing. In this case, something like “adding new files for ObjectName.”

## Making Your Object Available to the Rest of OpenStudio

At this point, the rest of OpenStudio doesn’t know about the new objects you have added. In order to tell it about these objects, you need to add references in the following files (model objects are added in alphabetical order):

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\model\model.cpp – this allows the model to create objects of this type

REGISTER\_CONSTRUCTOR**(**ObjectName**);**

…

REGISTER\_COPYCONSTRUCTORS**(**ObjectName**);**

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\model\ConcreteModelObjects.hpp – this provides a list of all model objects

#include <model/ObjectName.hpp>

**…**

#include <model/ObjectName\_Impl.hpp>

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\model\CMakeLists.txt – this adds your files to the Visual Studio project

ObjectName.hpp

ObjectName\_Impl.hpp

ObjectName.cpp

…

ObjectName\_Impl.hpp

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\model\ModelHVAC.i – this exports your model object to other languages using SWIG. The exact .i file will vary based on the object you are using. There are also slightly different macros to use if your object is unique (e.g. at most one per model).

MODELOBJECT\_TEMPLATES(ObjectName);

…

SWIG\_MODELOBJECT(ObjectName);

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\model\ScheduleTypeRegistry.cpp

**{**"ZoneHVACPackagedTerminalAirConditioner"**,**"Availability"**,**"availabilitySchedule"**,false,**"Availability"**,**0.0**,**1.0**},**

Once you have made all of these changes you will need to build the project again (at this stage you can build only the openstudio\_model project to save time). The first time you try to build Visual Studio will detect the change to the CMakeLists,txt file and prompt you to stop the build and reload the project including the new files. Press No at the prompt to begin building the entire project after reloading. After the files have been added to the Visual Studio project (they will show up under openstudio\_model) you can build the openstudio\_model project.

## Finishing the Source Code Files

The initial source code files written by the generate class script should compile. However, as noted before they are not a final solution because they do not implement any of the intelligence we want our model objects to have. Depending on the design you may need to go in and add methods, remove methods, or modify existing methods. Additionally, some areas in the code will be marked with “TODO” comments pointing out code that needs to be modified. Other model objects can serve as good examples for how to implement certain types of functionality.

### C++ Cheat Sheet

This document is not an appropriate place to explain C++; however, a few basic pieces of information may help. For more information, search online. There are

#### Creating a Variable

//type of variable //variable name

double someNumber**;**

someNumber = 5;

#### Creating a Variable and Assigning it a Value on one Line

//type of variable //variable name //value to set variable

double someNumber **=** 5**;**

#### Optionals

A boost::optional is a type of variable that is either false (ie the method didn’t return anything) or true. If an optional is true, in order to get the thing that the method returned, it must be De-Referenced. There are 2 ways to do this:

//This optional variable

boost**::**optional**<**ThermalZone**>** optThermalZone **=** modelObject**.**thermalZone**();**

//can be de-referenced with

**if** **(**optThermalZone**)** **{**

ThermalZone thermalZone **=** **(\***optThermalZone**);**

**}**

//or, if you are calling a method on the object immediately after de-reference

**if** **(**optThermalZone**)** **{**

std**::**vector**<**Space**>** spacesInZone **=** **(\***optThermalZone**).**spaces**();**

**}**

//you can also do it this way, with the -> replacing the \* before

**if** **(**optThermalZone**)** **{**

std**::**vector**<**Space**>** spacesInZone **=** optThermalZone**->**spaces**();**

**}**

#### Declaring a Method

The following example (in a .hpp file) declares a method that can be called on a thermal zone object to set the volume of the zone. This method must be implemented in a .cpp file.

//what type of object is returned //name of the method //what inputs are needed, and what type of object each input is

void setVolume**(**double volume**);**

#### Implementing a Method

The following example (in a .cpp file) implements the setVolume method for ThermalZone and ThermalZone\_Impl. Notice that the wrapper class simply passes the call on to the impl class.

void ThermalZone\_Impl::setVolume(boost::optional<double> volume) {

bool result = false;

result = setDouble(OS\_ThermalZoneFields::Volume, volume.get());

BOOST\_ASSERT(result);

}

void ThermalZone::setVolume(boost::optional<double> volume) {

getImpl<detail::ThermalZone\_Impl>()->setVolume(volume);

}

#### Vectors

The standard vector is used to hold 0 or more of a certain type of item. They are declared as follows.

//type of variable //variable name

std**::**vector**<**Thing**>** vectorOfThings**;**

//put things in the vector

double firstNum **=** 4**;**

vectorOfThings**.**push\_back**(**firstNum**);**

double secondNum **=** 5**;**

vectorOfThings**.**push\_back**(second**Num**);**

#### For Each Loop

In order to loop through a vector of things, OpenStudio code uses the following code.

//loop through the vector of things

BOOST\_FOREACH**(**Thing**&** thing**,** vectorOfThings**)** **{**

//do something to each thing in the vector

**}**

# Testing Your Code

OpenStudio contains thousands of files. In order to ensure that a change in one file doesn’t break something in another file, tests are written for each new addition. These tests are part of the OpenStudio source code and need to be run on your Branch before you Reintegrate back into the Trunk.

## Creating a Test File

The test framework that OpenStudio uses is called gtest. To make a new test:

1. Open: C:\Projects\OpenStudio\_branch\openstudiocore\src\model\test
2. Copy an existing test file to a new file called: ObjectName\_GTest.cpp
3. Open C:\Projects\OpenStudio\_branch\openstudiocore\src\model\CMakeLists.txt

test/ObjectName\_GTest.cpp

1. and add a reference to ObjectName\_GTest.cpp along with the other gtest files.

## Writing the Tests

The test file should contain tests for each method in the object, (all setters, getters, etc.) as well as test that it connects and disconnects from a loop, can be removed, etc. Each test is declared using either the TEST or TEST\_F macros and should have a unique name. The test body is all the code between the curly braces defining the test, this code is run each time the test is invoked. Within the test body we can create and use model objects and make test assertions. There are two primary types of assertions we can make. Assertions starting with ASSERT halt the entire test if the test condition fails while those starting with EXPECT register a failure but allow the test continue. Like the source code, the best way to understand how to write your own tests is to look at and copy tests from similar, existing files. Also, it is good practice to comment your test code to state what you are trying to test. This makes code review faster and easier for everyone.

A particularly well-commented example test file can be found at:

C:\Projects\OpenStudio\_branch\openstudiocore\Openstudiocore\src\model\test\CoilCoolingDXTwoSpeed\_GTest.cpp

## Running the Tests

Running Tests from Visual Studio

This method assumes you will run the tests with VS rather than at the command line.

1. Open: OpenStudioCore.sln in VS
2. Right click openstudio\_model\_tests : > Set As Startup Project
3. Right click openstudio\_model\_tests: > Properties

Configuration Properties: > Debugging

Command Arguments: > --gtest\_filter=\*ObjectName\*

Working Directory: C:\OpenStudio\build\OpenStudioCore-prefix\src\OpenStudioCore-build\Products\Debug

1. VS > Play (green triangle at top of main VS screen)

You can insert break points into the source code or the gtest to debug while the tests are running. This can help you debug if a test is failing.

1. The debug logs will come out as ModelFixtureTest.log at:

C:\OpenStudio\build\OpenStudioCore-prefix\src\OpenStudioCore-build\Products\Debug

# Forward Translation – OpenStudio to EnergyPlus

The OpenStudio model (.osm) is different than the EnergyPlus model (.idf). Not a lot different in most cases, but different enough that in order to create a working EnergyPlus file, translation must be done. This is done via the EnergyPlus Forward Translator.

## Adding the ForwardTranslate File

1. Open: C:\Projects\OpenStudio\_branch\openstudiocore\src\energyplus\ForwardTranslator
2. Create: ForwardTranslateObjectName.cpp

This goal of this file is to take an OpenStudio model object and copy data into the EnergyPlus fields. For the most part, this simply means using the model object’s getter methods, and using the resulting values to populate an EnergyPlus object’s input fields. A well-commented example is: …\src\energyplus\ForwardTranslator\ForwardTranslateCoilCoolingDXTwoSpeed.cpp

boost**::**optional**<**IdfObject**>** ForwardTranslator**::**translateCoilCoolingDXTwoSpeedWithoutUnitary**(** model**::**CoilCoolingDXTwoSpeed **&** modelObject **)**

**{**

//setup two boost optionals to use to store get method returns

boost**::**optional**<**std**::**string**>** s**;**

boost**::**optional**<**double**>** d**;**

//create the IdfObject that will be the coil

IdfObject idfObject**(**IddObjectType**::**Coil\_Cooling\_DX\_TwoSpeed**);**

//Name

m\_idfObjects**.**push\_back**(**idfObject**);**

s **=** modelObject**.**name**();**

**if(** s **)**

**{**

idfObject**.**setName**(\***s**);**

**}**

// A2 , \field Availability Schedule Name

Schedule sched **=** modelObject**.**getAvailabilitySchedule**();**

translateAndMapModelObject**(**sched**);**

idfObject**.**setString**(**Coil\_Cooling\_DX\_TwoSpeedFields**::**AvailabilityScheduleName**,** sched**.**name**().**get**()** **);**

// N1 , \field Rated High Speed Total Cooling Capacity

d **=** modelObject**.**getRatedHighSpeedTotalCoolingCapacity**();**

**if(** d **)**

**{** idfObject**.**setDouble**(**Coil\_Cooling\_DX\_TwoSpeedFields**::**RatedHighSpeedTotalCoolingCapacity**,\***d**);**

**}**

**else**

Once again, the best thing to do is look at other ForwardTranslate files as examples. Most objects being added will be very similar to something already in OpenStudio.

## Invoking Translation for the New Object

Now, OpenStudio knows how to translate an ObjectName object. It is time to make OpenStudio translate this object whenever it is encountered in a file. Register the new object in the following files.

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\energyplus\CMakeLists.txt

ForwardTranslator/ForwardTranslateObjectName.cpp

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\energyplus\ForwardTranslator.hpp

class ObjectName**;**

…

boost**::**optional**<**IdfObject**>** translateObjectName**(** model**::**ObjectName **&** modelObject **);**

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\energyplus\ForwardTranslator.cpp

**case** openstudio**::**IddObjectType**::**OS\_ObjectName **:**

**{**

model**::**ObjectName modelObject **=** modelObject**.**cast**<**ObjectName**>();**

retVal **=** translate ObjectName**(**modelObject**);**

**break;**

**}**

…

result**.**push\_back**(**IddObjectType**::**OS\_ObjectName**);**

# Show the New Object in OpenStudio App

Most model objects show up in the OpenStudio Application in customized interfaces. However, for HVAC component objects, the following instructions should work.

1. Make an icon and a mini-icon. For testing, simply copy and rename one the existing icons in C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\images.
2. Add the renamed icon and mini-icon files to:

C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\images

C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\images\mini\_icons

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\IconLibrary.cpp

m\_icons**[**openstudio**::**IddObjectType**(**openstudio**::**IddObjectType**::**OS\_ObjectName**).**value**()]** **=** **new** QPixmap**(**":images/objectname.png"**);**

…

m\_miniIcons**[**openstudio**::**IddObjectType**(**openstudio**::**IddObjectType**::**OS\_ObjectName**).**value**()]** **=** **new** QPixmap**(**":images/mini\_icons/objectname.png"**);**

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\openstudio.qrc

<file>images/objectname.png</file>

…

<file>images/mini\_icons/objectname.png</file>

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\library\OpenStudioPolicy.xml

<POLICY IddObjectType=**"OS\_Component\_Name"**>

<rule IddField=**"Hot Water Inlet Node Name"** Access=**"HIDDEN"**/>

<rule IddField=**"Hot Water Outlet Node Name"** Access=**"HIDDEN"**/>

</POLICY>

This section will need to be customized for your particular object. It determines which fields are hidden (so user can’t mess them up) in the right column interface in the OpenStudio application. Generally, node names are hidden because the GUI takes care of hooking them up correctly.

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_lib\MainRightColumnController.cpp.

libraryWidget**->**addModelObjectType**(**IddObjectType**::**OS\_ObjectName**,**"Object Name"**);**

MyModelList**->**addModelObjectType**(**IddObjectType**::**OS\_ObjectName**,**"Object Name"**);**

MyLibraryList**->**addModelObjectType**(**IddObjectType**::**OS\_ObjectName**,**"Object Name"**);**

This section tells OpenStudio to display all objects of this type in the right hand library of objects that can be dragged into the HVAC scene. Add object names where applicable.

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_app\OpenStudioApp.cpp

#include <model/ObjectName.hpp>

1. C:\Projects\OpenStudio\_branch\openstudiocore\src\openstudio\_app\Resources\hvaclibrary\hvac\_library.osm
   1. Open the hvac\_libary.osm file in the version of the OpenStudio application that is located in your build directory. C:\Projects\OpenStudio\_branch\build\OpenStudioCore-prefix\src\OpenStudioCore-build\Products\Debug\OpenStudio.exe
   2. The OpeStudio application will use the version translator to automatically update the model to the latest version when you open it.
   3. Save the hvac\_library.osm file after it has been opened and updated to the latest version. It is important that the version of the library osm file is current. You will add at least one instance of your new objects to this file, making them available to the application. If the version of the osm file is older than the version of OpenStudio that introduced the new model object, OpenStudio will be confused and report that the objects are not valid.
   4. Generate new content for the hvac\_library.osm file using the ruby bindings.
      1. Create a new temporary ruby file similar to the following example and save it to C:\Projects\add\_to\_hvac\_library.rb. This file does not need to be added to the project or committed to version control. You may generate this file in any text editor that you are comfortable with.

#Test the methods that set and get all the fields

TEST(CoilCoolingDXTwoSpeed,CoilCoolingDXTwoSpeed\_SetGetFields)

{

#create a model to use in testing this code.

require 'C:/Projects/OpenStudio\_branch/build/OpenStudioCore-prefix/src/OpenStudioCore-build/ruby/Debug/openstudio.rb'

model= OpenStudio::Model m;::Model.new()

#create a avail\_schedule and the curves to use in the constructor= OpenStudio::Model::ScheduleConstant.new(model)

model::ScheduleCompact avail\_schedule(m);.setValue(1.0)

model::CurveBiquadratic ccfot1(m);

model::CurveCubic ccfof2(m);

model::CurveBiquadratic eirfot3(m);

model::CurveQuadratic eirfof4(m);

model::CurveCubic plf5(m);

model::CurveBiquadratic lsccfot6(m);

model::CurveBiquadratic lseirfot7(m);

#make a coil to do the testing on

model::CoilCoolingDXTwoSpeed coil(m,

schedule,

ccfot1,

ccfof2,

eirfot3,

eirfof4,

plf5,

lsccfot6,

lseirfot7);

# A2 , \field Availability Schedule Name

model::ScheduleCompact availSch(m);

coil.setAvailabilitySchedule(availSch);

ASSERT\_EQ(availSch,coil.getAvailabilitySchedule());

coilheatbb = OpenStudio::Model::CoilHeatingWaterBaseboard.new(model)

baseboard = OpenStudio::Model::ZoneHVACBaseboardConvectiveWater.new(model,avail\_schedule,coilheatbb)

f = File.new('C:/Projects/add\_to\_hvac\_libary.osm','w+')

f << model

f.close

}

* + 1. Open the Command Prompt from the Start Menu -> All Programs -> Accessories -> Command Prompt
    2. Execute Ruby on the add\_to\_hvac\_library.rb file.

> ruby C:\Projects\add\_to\_hvac\_library.rb

* + 1. Copy the contents of the new file at C:\Projects\add\_to\_hvac\_library.osm into the end of the hvac\_library.osm file. Exclude the version object from the copy. The hvac\_library.osm file should already contain the correct version object based on the previous steps.

# Create New Functional Tests

In OpenStudio there are functional tests to make sure that the feature set works properly and continues to work as the software evolves. There are two distinct types of functional tests. One type of functional test ensures that the OpenStudio API remains stable, and that class and method names remain constant. These API tests are based on ruby files that contain the necessary API calls to construct and simulate a complete model. The results of the simulations are analyzed to make sure they are reasonable. The other type of functional tests is used to make sure that saved osm files, utilizing specific features, continue to work as OpenStudio evolves. These tests are based on saved osm models that demonstrate each feature. The osm test files are added to the project and fixed at the OpenStudio version when the feature they test was introduced. The files are simulated and the results are analyzed to make sure they are reasonable. Because the osm files remain unchanged and OpenStudio continues to evolve, the testing workflow exercises and verifies the correctness of the OpenStudio version translator.

The functional tests are not part of the main OpenStudio repository. Instead they are located at <https://cbr.nrel.gov/openstudio-resources/svn/trunk>. To extend and run the functional tests you will need to checkout this repository. These instructions will assume you have a working copy of openstudio-resources located at C:\Projects\openstudio-resources.

## Create API Functional Tests

The API tests are located in ruby files in the openstudio-resources\model\simulationtests directory.

1. Use one of the existing ruby files such as baseline\_sys01.rb as a starting point.
2. Make a copy of the starting ruby file and rename it according to the HVAC feature that will be tested.
3. Edit the new ruby file to create a model that demonstrates the new feature. If the new feature is related to HVAC, you should remove the model.add\_hvac method call, and replace it with detailed API calls that demonstrate the new feature.
4. Check the new ruby file into version control.
5. Open the file openstudio-resources\model\_tests\ModelSimulation\_GTest.cpp and add a new test fixture following the pattern of TEST\_F(ModelSimulationFixture, baseline\_sys01\_rb).
6. Create a build directory located at C:\Projects\openstudio-resources\build.
7. Configure the openstudio-resources project using cmake and the same method used to configure the main openstudio project. Change OPENSTUDIO\_BUILD\_DIR value to point to the build directory under your working branch.
8. Open the Visual Studio solution located at C:\Projects\openstudio-resources\build\ OpenStudioRegression.sln and build the solution.
9. Open the Command Prompt from the Start Menu -> All Programs -> Accessories -> Command Prompt
10. Change to the openstudio-resources build directory.

> cd C:\Projects\openstudio-resources\build

1. Run ctest on the model tests. This command will run EnergyPlus simulations and check the results. The argument “-N” should be replaced with the number of simulations (processors) to run in parallel. Note: -N (don’t run, show me what would be run) ; -R regular expression. Also note: > Ctest -help will show all the options. Filter for just the nexts we want.

> ctest -jN -R “ModelSimulationFixture.\*rb” -N

1. If you add another test you must repeat these steps. It is important that you rebuild the Visual Studio solution before running ctest.
2. Error debug: go to Openstudio-resources/build/testing/temporary/lasttest.log

## Create OSM File Functional Tests

1. Complete steps to add and run a new ruby API test.
2. Running the API test will create an osm file that can be used to form a new osm file test. You can find the osm file at openstudio-resources/build/testruns/baseline\_sys01.rb/Ruby/out.osm, replacing baseline\_sys01.rb with your test name. Copy the generated osm file to openstudio-resources/model/simulationtests/baseline\_sys01.osm, replacing baseline\_sys01.osm with the name of your test.
3. Check the new osm file into version control.
4. Open the file openstudio-resources\model\_tests\ModelSimulation\_GTest.cpp and add a new test fixture following the pattern of TEST\_F(ModelSimulationFixture, baseline\_sys01\_osm).
5. Rebuild the Visual Studio Solution.
6. Open the Command Prompt from the Start Menu -> All Programs -> Accessories -> Command Prompt
7. Change to the openstudio-resources build directory.

> cd C:\Projects\openstudio-resources\build

1. Run ctest on the model tests. This command will run EnergyPlus simulations and check the results. The argument “N” should be replaced with the number of simulations to run in parallel.

> ctest -jN -R “ModelSimulationFixture.\*osm”

To run both the osm and rb tests together use the command.

> ctest -jN -R “ModelSimulationFixture.\*”