Chapter 9
The OpenStudio Software Development Kit

9.1 Introduction

As discussed in Chap. 1, OpenStudio is not a single energy modeling tool. Rather, it is an SDK or platform, designed to reduce the cost and time to create a variety of energy efficiency assessment applications. The OpenStudio Application and PAT, presented in previous chapters, are intended as examples of using the SDK to create software in C++ and Electron/Angular respectively. A third example of creating new functionality with the SDK is the OpenStudio Measure introduced in Chap. 6. OpenStudio Measures are the most accessible means of creating new capability with OpenStudio and represent the "gateway" to more advanced application development. For this reason, the bulk of Chap. 9 is devoted to adapting existing Measures or creating new ones to add functionality to the OpenStudio Application or PAT.

9.2 OpenStudio Measure Overview

We have already explored the basic concept of an OpenStudio Measure along with the BCL as a convenient source for them in Chap. 6. In Chap. 7, we extended our understanding of OpenStudio Measures as a means of optimizing building energy performance and calibrating models against consumption data. Chapter 8 highlighted an advanced use of Measure scripting to combine Radiance and EnergyPlus analyses for more accurate daylighting design savings estimates. Another example

The original version of this chapter was revised. A correction to this chapter can be found at https://doi.org/10.1007/978-3-319-77809-9_10

Electronic Supplementary Material: The online version of this chapter (https://doi.org/10.1007/978-3-319-77809-9_9) contains supplementary material, which is available to authorized users.

L. Brackney et al., *Building Energy Modeling with OpenStudio*,
https://doi.org/10.1007/978-3-319-77809-9_9

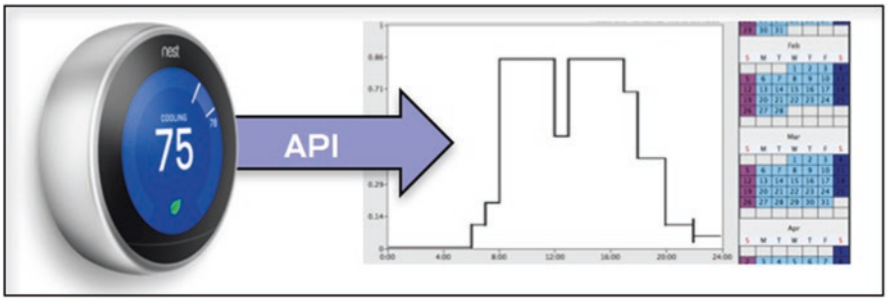


Fig. 9.1 NEST thermostat Measure concept

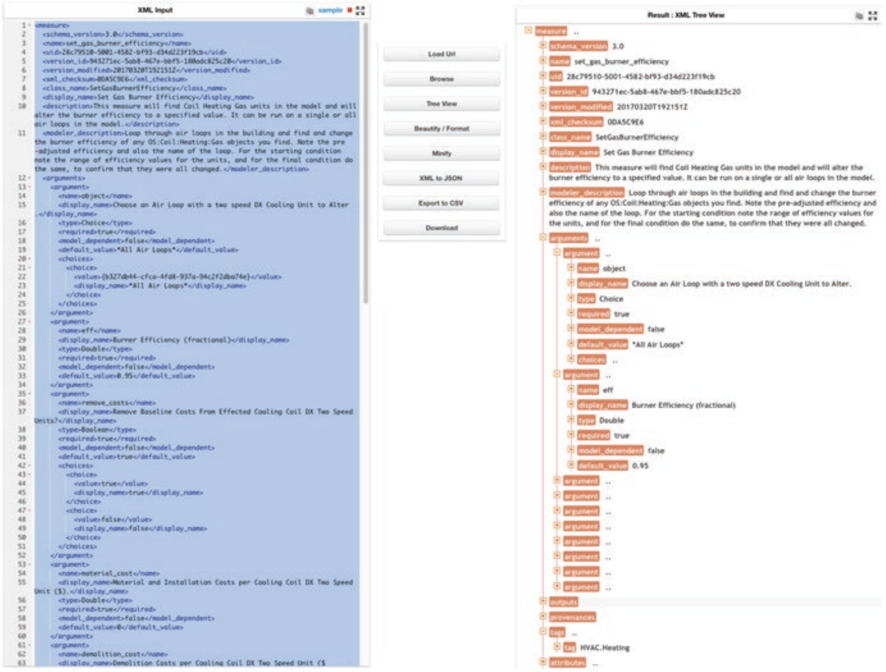
integrates[1] OpenStudio models with the GLHEPro[2] Ground Loop Heat Exchanger Design tool. Lastly, the authors know of one enterprising OpenStudio user who wrote a Measure to access measured occupancy data from his Nest thermostat as a means of creating Model schedules (Fig. 9.1). We now turn our attention to how Measures actually work and how to extend OpenStudio's functionality by creating new ones.

OpenStudio Measures are ZIP archives comprised of <u>at least</u> two files: measure.xml and measure.rb. The measure.xml file is an eXtensible Markup Language (XML) file containing descriptive information (metadata) about the measure. Measure metadata is used by the BCL and OpenStudio applications to identify where and how a measure might be used, what arguments the measure may accept, when it was last modified, etc. The measure.xml file, while written in plain text, can be difficult to read and edit. XML editor/viewers are widely available[3] and helpful in working with Measure metadata. Figure 9.2 illustrates codebeautify.org's web-based XML viewer inspecting a raw measure.xml (on left) and a convenient tree view of the same content (on right).

Readily apparent in the tree view are the Measure's name, when it was last modified, a high level description, a more detailed explanation of how the Measure works, its arguments, search tags, and more. In this case, the "*set_gas_burner_efficiency*" Measure contains ten arguments, many of which are optional. The first two named "*object*" and "*eff*" are required and tell the Measure which of the Model's air loops it should operate on, the default being all of them, and the fractional burner efficiency, which defaults to 0.95. Argument names, type definitions, etc. are of particular significance in describing how information is passed between OpenStudio applications and the measure.rb code itself.

If the OpenStudio Measure metadata standard defines how it can be connected with OpenStudio applications, much like a puzzle piece, then the measure.rb represents the picture on the puzzle piece. Measures may contain additional files such as design documents, test cases, and more as part of the Measure's "payload," but the measure.rb contains the actual Ruby code that is executed whenever the measure is invoked by an application. Figure 9.3 illustrates the overall "anatomy" of a Measure as it might appear on the BCL.

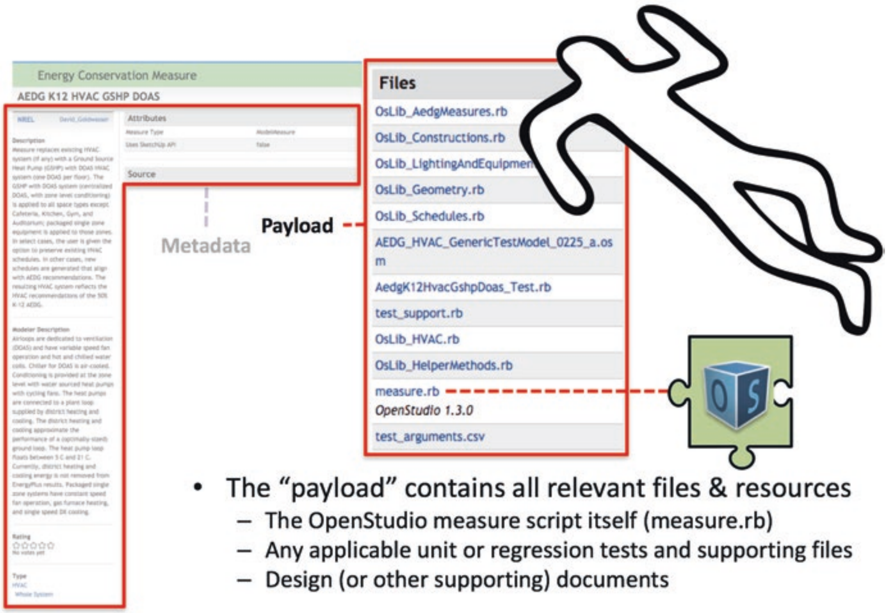
[1] https://bcl.nrel.gov/search/site/GLHEPro?f[0]=bundle%3Anrel_measure
[2] https://hvac.okstate.edu/glhepro/overview
[3] http://codebeautify.org/xmlviewer



```
<measure>
  <schema_version>3.0</schema_version>
  <name>set_gas_burner_efficiency</name>
  <uid>28c79510-5001-4582-bf93-d34d223f19cb</uid>
  <version_id>943271ec-5ab8-467e-bbf5-180adc825c20</version_id>
  <version_modified>20170320T192151Z</version_modified>
  <xml_checksum>0DA5C9E6</xml_checksum>
  <class_name>SetGasBurnerEfficiency</class_name>
  <display_name>Set Gas Burner Efficiency</display_name>
  <description>This measure will find Coil Heating Gas units in the model and will
alter the burner efficiency to a specified value. It can be run on a single or all
air loops in the model.</description>
  <modeler_description>Loop through air loops in the building and find and change
the burner efficiency of any OS:Coil:Heating:Gas objects you find. Note the pre
-adjusted efficiency and also the name of the loop. For the starting condition
note the range of efficiency values for the units, and for the final condition do
the same, to confirm that they were all changed.</modeler_description>
  <arguments>
    <argument>
      <name>object</name>
      <display_name>Choose an Air Loop with a two speed DX Cooling Unit to Alter
.</display_name>
      <type>Choice</type>
      <required>true</required>
      <model_dependent>false</model_dependent>
      <default_value>*All Air Loops*</default_value>
      <choices>
        <choice>
          <value>{b327db44-cfca-4fd8-937a-94c2f2dba74e}</value>
          <display_name>*All Air Loops*</display_name>
        </choice>
      </choices>
    </argument>
    <argument>
      <name>eff</name>
      <display_name>Burner Efficiency (fractional)</display_name>
      <type>Double</type>
      <required>true</required>
      <model_dependent>false</model_dependent>
      <default_value>0.95</default_value>
    </argument>
    <argument>
      <name>remove_costs</name>
      <display_name>Remove Baseline Costs From Effected Cooling Coil DX Two Speed
Units?</display_name>
      <type>Boolean</type>
      <required>true</required>
      <model_dependent>false</model_dependent>
      <default_value>true</default_value>
      <choices>
        <choice>
          <value>true</value>
          <display_name>true</display_name>
        </choice>
        <choice>
          <value>false</value>
          <display_name>false</display_name>
        </choice>
      </choices>
    </argument>
    <argument>
      <name>material_cost</name>
      <display_name>Material and Installation Costs per Cooling Coil DX Two Speed
Unit ($).</display_name>
      <type>Double</type>
      <required>true</required>
      <model_dependent>false</model_dependent>
      <default_value>0</default_value>
    </argument>
    <argument>
      <name>demolition_cost</name>
      <display_name>Demolition Costs per Cooling Coil DX Two Speed Unit ($
```

```
measure ..
  schema_version 3.0
  name set_gas_burner_efficiency
  uid 28c79510-5001-4582-bf93-d34d223f19cb
  version_id 943271ec-5ab8-467e-bbf5-180adc825c20
  version_modified 20170320T192151Z
  xml_checksum 0DA5C9E6
  class_name SetGasBurnerEfficiency
  display_name Set Gas Burner Efficiency
  description This measure will find Coil Heating Gas units in the model and will alter the
  burner efficiency to a specified value. It can be run on a single or all air loops in the model.
  modeler_description Loop through air loops in the building and find and change the burner
  efficiency of any OS:Coil:Heating:Gas objects you find. Note the pre-adjusted efficiency and
  also the name of the loop. For the starting condition note the range of efficiency values for
  the units, and for the final condition do the same, to confirm that they were all changed.
  arguments ..
    argument ..
      name object
      display_name Choose an Air Loop with a two speed DX Cooling Unit to Alter.
      type Choice
      required true
      model_dependent false
      default_value *All Air Loops*
      choices ..
    argument ..
      name eff
      display_name Burner Efficiency (fractional)
      type Double
      required true
      model_dependent false
      default_value 0.95
    argument ..
    argument ..
    argument ..
    argument ..
    argument ..
    argument ..
    argument ..
    argument ..
  outputs
  provenances
  tags ..
    tag HVAC.Heating
  attributes ..
```

Fig. 9.2 Code beautify used to inspect a Measure.xml file



Energy Conservation Measure

AEDG K12 HVAC GSHP DOAS

NREL David_Goldwasser

Description
Measure replaces existing HVAC system (if any) with a Ground Source Heat Pump (GSHP) with DOAS HVAC system (one DOAS per floor). The GSHP with DOAS system (centralized DOAS, with zone level conditioning) is applied to all space types except Cafeteria, Kitchen, Gym, and Auditorium; packaged single zone equipment is applied to those zones. In select cases, the user is given the option to preserve existing HVAC schedules. In other cases, new schedules are generated that align with AEDG recommendations. The resulting HVAC system reflects the HVAC recommendations of the 50% K-12 AEDG.

Modeler Description
Airloops are dedicated to ventilation (DOAS) and have variable speed fan operation and hot and chilled water coils. Chiller for DOAS is air-cooled. Conditioning is provided at the zone level with water sourced heat pumps with cycling fans. The heat pumps are connected to a plant loop supplied by district heating and cooling. The district heating and cooling approximate the performance of a (optimally sized) ground loop. The heat pump loop floats between 5 C and 21 C. Currently, district heating and cooling energy is not removed from EnergyPlus results. Packaged single zone systems have constant speed fan operation, gas furnace heating, and single speed DX cooling.

Rating
☆☆☆☆☆
No votes yet

Type
HVAC
Whole System

Attributes	
Measure Type	ModelMeasure
Uses SketchUp API	false

Source

Files
- OsLib_AedgMeasures.rb
- OsLib_Constructions.rb
- OsLib_LightingAndEquipmen
- OsLib_Geometry.rb
- OsLib_Schedules.rb
- AEDG_HVAC_GenericTestModel_0225_a.osm
- AedgK12HvacGshpDoas_Test.rb
- test_support.rb
- OsLib_HVAC.rb
- OsLib_HelperMethods.rb
- measure.rb
- *OpenStudio 1.3.0*
- test_arguments.csv

- The "payload" contains all relevant files & resources
  - The OpenStudio measure script itself (measure.rb)
  - Any applicable unit or regression tests and supporting files
  - Design (or other supporting) documents

Fig. 9.3 Anatomy of a Measure

The following “snippet” of Measure Ruby code provides some flavor for what Measure scripts look like. In this example, the Measure loops across all surfaces in the Model. If a surface is an exterior wall, then a new construction is applied to that surface.

```
# Measure structure to replace exterior wall orig construction with new
model.getSurfaces.each do |s|
   if s.outsideBoundaryCondition == “Outdoors” and s.surfaceType == “Wall”
         if s.construction.name.get == orig
            s.setConstruction(new)
```

Sections 9.3 and 9.4 cover the rudiments of modifying existing measure.xml and measure.rb files or creating new ones. Section 9.5 presents the important topic of Measure testing, and additional files one might want to include as part of a Measure's payload to continuously verify that it works as intended. The reader is referred to the Measure Writer's Reference Guide on the OpenStudio website[4] as an additional resource for use alongside these sections of the text.

9.3 Adapting an Existing Measure

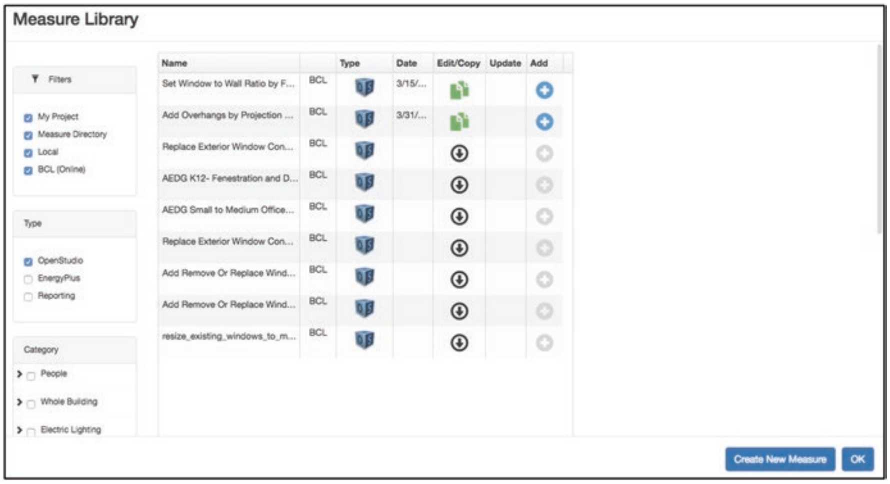
Recall from Chap. 6 that Measures frequently start out on the BCL and are then downloaded to the user's local library. In the case of PAT, Measures may then be added from that local library to a specific project. Figure 9.4 shows PAT's now familiar Measure management window with a range of fenestration Measures. Two of these have already been downloaded and are available to add to this PAT project.

Note that once a Measure has been downloaded, its Button changes to a Button. Clicking creates a copy of the measure in a separate directory designated as “MyMeasures.” This directory is distinct from the local library and is intended for use in creating new or customized Measures. Figure 9.5 illustrates a “flow” of Measures from the BCL to a user's local Measures directory, which can then be used directly in a PAT project or copied into MyMeasures for modification and use in PAT.

Both the OpenStudio Application and PAT have a Preferences Menu option that allows the user to specify (or change) the location of this directory. Measures in the MyMeasures directory will be denoted with a “My” label in PAT or an icon in the OpenStudio Application (Fig. 9.6).

Highlighting a Measure in the OpenStudio Application also enables a copy Measure option indicated by a Button. This behaves identically to PAT's Button and makes a copy of the Measure XML and .rb files to your MyMeasures directory for further editing (Fig. 9.7).

[4] http://nrel.github.io/OpenStudio-user-documentation/reference/measure_writing_guide/



Measure Library

Filters

- ☑ My Project
- ☑ Measure Directory
- ☑ Local
- ☑ BCL (Online)

Type

- ☑ OpenStudio
- ☐ EnergyPlus
- ☐ Reporting

Category

- ☐ People
- ☐ Whole Building
- ☐ Electric Lighting

Name		Type	Date	Edit/Copy	Update	Add
Set Window to Wall Ratio by F...	BCL	OS	3/15/...	✓		⊕
Add Overhangs by Projection ...	BCL	OS	3/31/...	✓		⊕
Replace Exterior Window Con...	BCL	OS		⬇		⊕
AEDG K12- Fenestration and D...	BCL	OS		⬇		⊕
AEDG Small to Medium Office...	BCL	OS		⬇		⊕
Replace Exterior Window Con...	BCL	OS		⬇		⊕
Add Remove Or Replace Wind...	BCL	OS		⬇		⊕
Add Remove Or Replace Wind...	BCL	OS		⬇		⊕
resize_existing_windows_to_m...	BCL	OS		⬇		⊕

Create New Measure | OK

Fig. 9.4 Measure library with BCL fenestration content shown

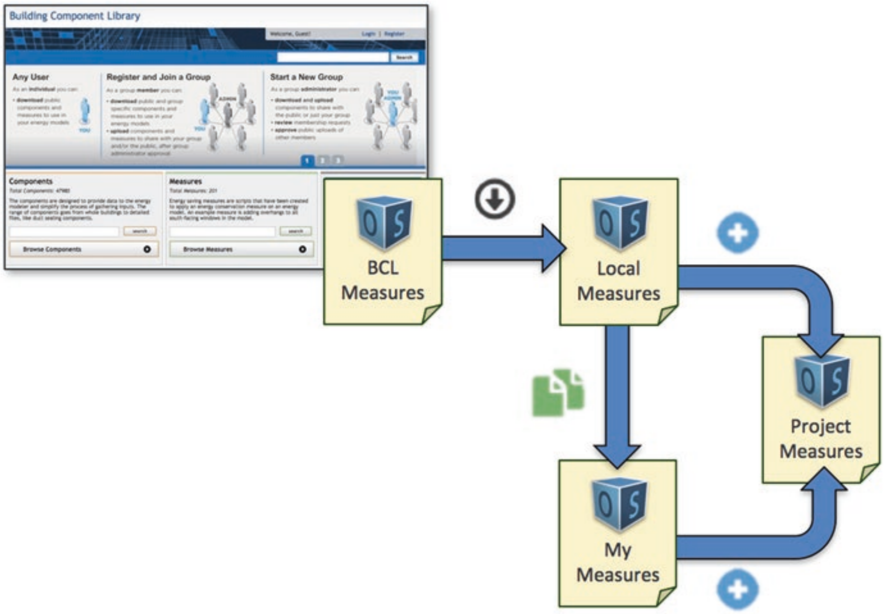


Fig. 9.5 Measure flow in PAT

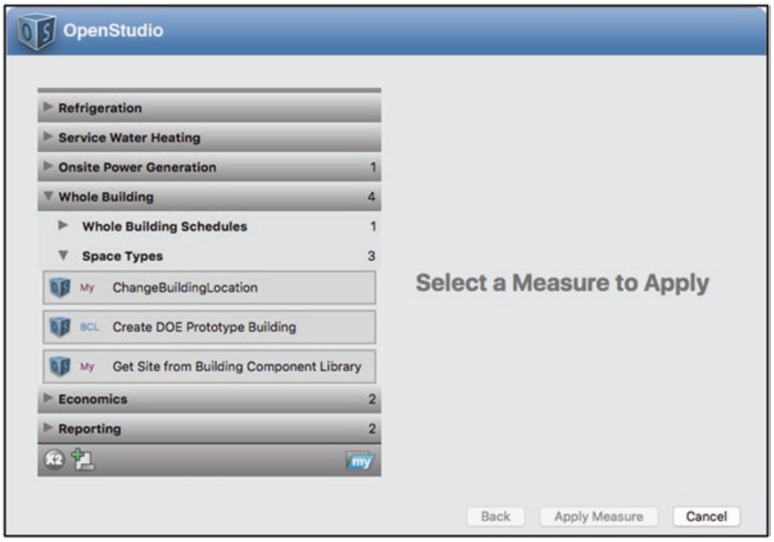
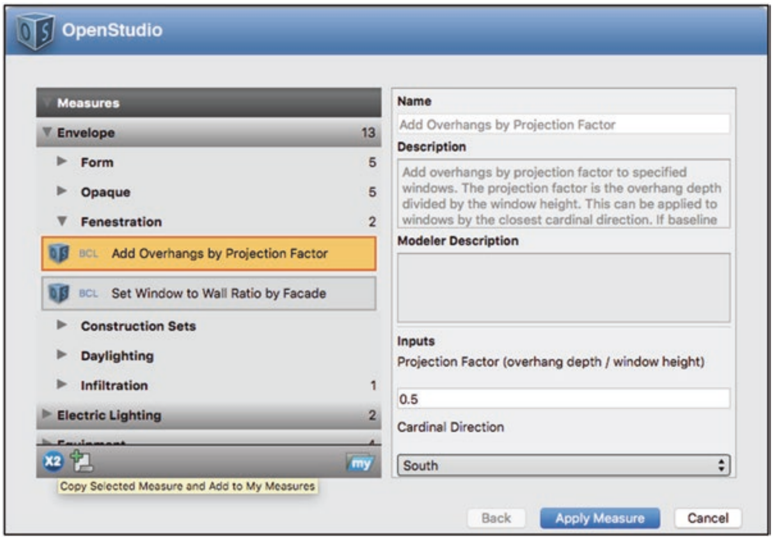

OpenStudio
Refrigeration
Service Water Heating
Onsite Power Generation 1
Whole Building 4
Whole Building Schedules 1
Space Types 3
My ChangeBuildingLocation
BCL Create DOE Prototype Building
My Get Site from Building Component Library
Economics 2
Reporting 2
Select a Measure to Apply
Back
Apply Measure
Cancel

Fig. 9.6 OpenStudio Apply Measure Now dialog highlighting BCL and MyMeasures content



Name
Add Overhangs by Projection Factor
Description
Add overhangs by projection factor to specified windows. The projection factor is the overhang depth divided by the window height. This can be applied to windows by the closest cardinal direction. If baseline
Modeler Description
Inputs
Projection Factor (overhang depth / window height)
0.5
Cardinal Direction
South

Fig. 9.7 Preparing to make a copy of a Measure in the OpenStudio Application

9.4 Checkpoint Thirteen: Adapting Measures

One of the best ways to learn how to write Measures is to modify an existing Measure. By taking this approach, users who are not programming experts can gain the confidence and skills to eventually develop their own Measures from scratch. In this exercise, we will copy the measure named "GasHeatingCoilEfficiency" and modify it to set the efficiency of gas fired hot water boilers instead.

9.4.1 Text Editor

For this exercise, you will need a text editor. Although a basic text editor like Notepad for Windows will work, a text editor with syntax highlighting for the Ruby programming language makes the process much easier. There are many programs available online. Notepad++ is a good free option for Windows, and Mac users might consider TextMate. The authors would not presume to suggest anything for Linux users, lest we provoke a VI/Emacs altercation.

9.4.2 Programming Background

Writing Measures involves basic computer programming with the Ruby scripting language. If basic programming concepts are unfamiliar to you, we recommend beginning with an online tutorial.[5] Spending an hour or two going through one of the many excellent online Ruby tutorials will be well worth the effort. Here are few key concepts you will need to comprehend, edit, or write most measures:

- Data types (String, Double, Integer, Array, Boolean),
- Variables,
- If & Case Statements, and
- For & For Each Loops.

9.4.3 Copy the Measure

The first step in adapting a Measure for a new purpose is to find an existing one that has similar functionality to what you are trying to achieve. For example, if your goal were to modify wall insulation, a Measure that modified roof insulation might be a good candidate. For the purpose of this exercise, the Measure to be copied will be "GasHeatingCoilEfficiency."

[5] https://www.ruby-lang.org/en/documentation/quickstart/

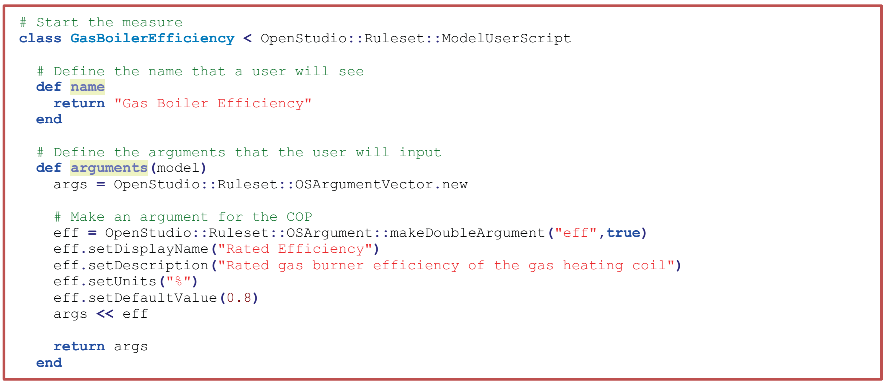
1. Open the *PrimarySchoolHVAC.osm* Model from Chap. 4 in the OpenStudio Application.
2. Open the “Apply Measure Now” dialog under “Components & Measures”
3. Find the “Gas Heating Coil Efficiency” Measure under HVAC > Heating
4. Make a copy of the Measure using the Button.
5. Change the name to: Gas Fired Boiler Efficiency.

9.4.4 Review the Measure

After the new Measure is created, a folder will be opened that contains a*/tests* directory, a *measure.xml file,* and the new *measure.rb* file. Open the *measure.rb* file in your text editor. It should look something like Fig. 9.8 below.

The first step is to get a general idea of how the existing code works. Well-written measures should have comments describing what the code is doing at each step. These comments are denoted by the # symbol at the beginning of the line. Read these comments in order, starting at the top of the file and moving down to the bottom. Don’t worry about the rest of the code in the file. You may need to repeat this process a few times to get the general idea. At the end of this process, you should have a general idea of the steps that are involved in the measure. If a measure is insufficiently commented or is very complex, it is not a good starting point. Rather than wasting time learning from a difficult example, look for a simpler measure to start with.

With the general idea in mind, start again at the first comment. Read the code immediately below this comment. Use your basic programming knowledge and energy modeling expertise to understand what is going on. You can add additional comments to the file on a line-by-line basis can help you document the progress you are making and keep the ideas straight. This process sounds tedious, but after doing it once or twice you will have a much better understanding of how the measure works.



```
# Start the measure
class GasBoilerEfficiency < OpenStudio::Ruleset::ModelUserScript

  # Define the name that a user will see
  def name
    return "Gas Boiler Efficiency"
  end

  # Define the arguments that the user will input
  def arguments(model)
    args = OpenStudio::Ruleset::OSArgumentVector.new

    # Make an argument for the COP
    eff = OpenStudio::Ruleset::OSArgument::makeDoubleArgument("eff",true)
    eff.setDisplayName("Rated Efficiency")
    eff.setDescription("Rated gas burner efficiency of the gas heating coil")
    eff.setUnits("%")
    eff.setDefaultValue(0.8)
    args << eff

    return args
  end
```

Fig. 9.8 Reading through a Measure

9.4.5 *Modify the Measure*

After gaining an understanding of the code, we are going to make our first edit to the file. In this case, we are simply changing the description and default value for the measure argument from gas burner efficiency to boiler efficiency. It is a good idea to start small, make minor changes, and then test before moving on. Find the section of the measures shown in Fig. 9.9 below, make the changes specified, and save the file.

Before making any other changes, run the measure using “Apply Measure Now” and review the resulting Model. Ensure that the changes you made are actually reflected in the Model. In this case, the changes to the argument can be seen in the dialog, as shown in Fig. 9.10.

If the measure didn’t run, go back and correct any typos or other errors you may have made. Generally, the line number for the error will be shown in the message. For example, Fig. 9.11 identifies the location of a typo.

Proceed in this fashion, making small changes and testing, until the measure is to your liking. Until you have performed this process a few times, resist the temptation to change multiple things at once. Think of the process as an experiment; if you change two variables and the measure doesn’t run, you won’t know which change was responsible for the failure.

Next, modify the allowable boiler efficiencies. Change the code as shown in Fig. 9.12 so that the boiler efficiency must be between 0.65 (65%) and 1 (100%). This range is acceptable for our purposes, but be aware that an existing building could contain an old boiler in disrepair that falls outside of these bounds these bounds.

Next, modify the code to change boilers instead of gas heating coils, as shown in Fig. 9.13.

Next, modify the code that reports whether or not the Measure was applicable, as shown in Fig. 9.14.

Next, modify the code that reports the initial and final condition of the Model, as shown in Fig. 9.15.

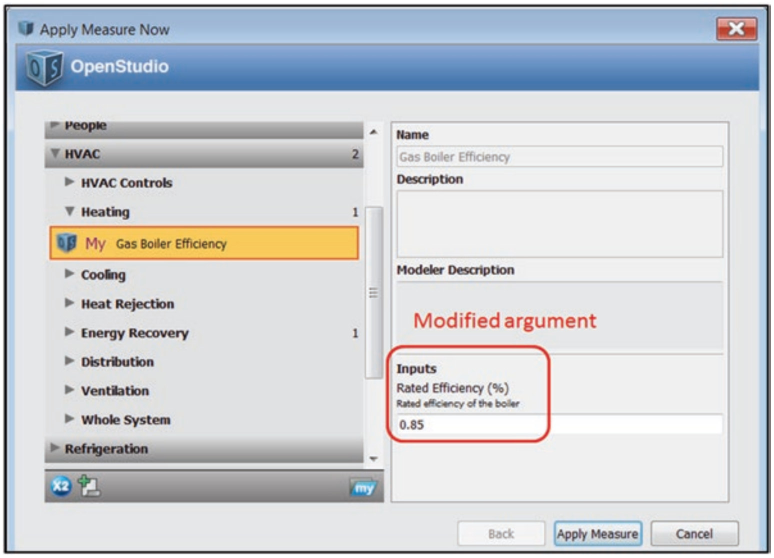
Change this:

```
    # Make an argument for the efficiency
    eff = OpenStudio::Ruleset::OSArgument::makeDoubleArgument("eff",true)
    eff.setDisplayName("Rated Efficiency")
    eff.setDescription("Rated gas burner efficiency of the gas heating coil")
    eff.setUnits("%")
    eff.setDefaultValue(0.8)
    args << eff
```

To this:

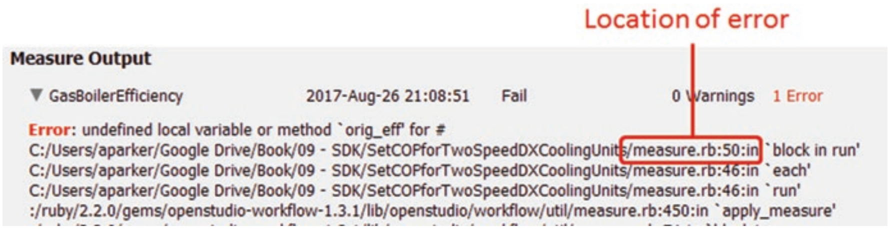
```
    # Make an argument for the efficiency
    eff = OpenStudio::Ruleset::OSArgument::makeDoubleArgument("eff",true)
    eff.setDisplayName("Rated Efficiency")
    eff.setDescription("Rated efficiency of the boiler")
    eff.setUnits("%")
    eff.setDefaultValue(0.85)
    args << eff
```

Fig. 9.9 Initial edits to the new Measure



Name: Gas Boiler Efficiency
Description:
Modeler Description:
Inputs
Rated Efficiency (%)
Rated efficiency of the boiler
0.85

Fig. 9.10 Verifying that the argument changes were made correctly



```
Measure Output
▼ GasBoilerEfficiency          2017-Aug-26 21:08:51     Fail          0 Warnings    1 Error
Error: undefined local variable or method `orig_eff' for #
C:/Users/aparker/Google Drive/Book/09 - SDK/SetCOPforTwoSpeedDXCoolingUnits/measure.rb:50:in `block in run'
C:/Users/aparker/Google Drive/Book/09 - SDK/SetCOPforTwoSpeedDXCoolingUnits/measure.rb:46:in `each'
C:/Users/aparker/Google Drive/Book/09 - SDK/SetCOPforTwoSpeedDXCoolingUnits/measure.rb:46:in `run'
:/ruby/2.2.0/gems/openstudio-workflow-1.3.1/lib/openstudio/workflow/util/measure.rb:450:in `apply_measure'
```

Fig. 9.11 Identifying the location of a Measure error

```
Change this:

    # Check the efficiency for reasonableness
    # Efficiency must be between zero and one
    if eff <= 0 || eff >= 1
      runner.registerError("Efficiency must be between zero and one.  You entered #{eff}.")
      return false
    end

To this:

    # Check the efficiency for reasonableness
    # Efficiency must be between 0.65 and 1
    if eff <= 0.65 || eff >= 1
      runner.registerError("Efficiency must be between 0.65 and 1.  You entered #{eff}.")
      return false
    end
```

Fig. 9.12 Making a second change to the Measure

```
Change this:

    # Loop through the gas heating coils in the model
    num_htg_coils_changed = 0
    orig_effs = []
    model.getCoilHeatingGass.each do |htg_coil|

      # Get the original efficiency and store it
      eff_original = htg_coil.gasBurnerEfficiency
      orig_effs << eff_original

      # Change the efficiency to the new value
      htg_coil.setGasBurnerEfficiency(eff)

      # Report the change of efficiency
      runner.registerInfo("Changing the efficiency of #{htg_coil.name} from #{eff_original} to
#{eff} ")

      # Add to the number of coils changed
      num_htg_coils_changed += 1

    end

To this:

    # Loop through the boilers in the model
    num_blrs_changed = 0
    orig_effs = []
    model.getBoilerHotWaters.each do |boiler|

      # Get the original efficiency and store it
      eff_original = boiler.nominalThermalEfficiency
      orig_effs << eff_original

      # Change the efficiency to the new value
      boiler.setNominalThermalEfficiency(eff)

      # Report the change of efficiency
      runner.registerInfo("Changing the efficiency of #{boiler.name} from #{eff_original} to
#{eff} ")

      # Add to the number of boilers changed
      num_blrs_changed += 1

    end
```

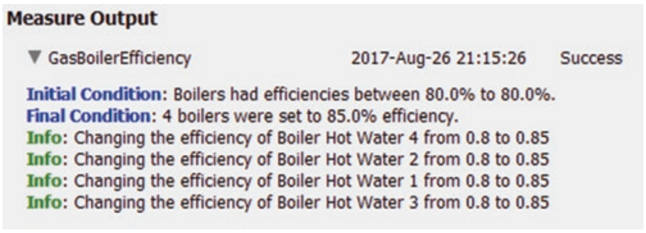
Fig. 9.13 Making a third change to the Measure

```
Change this:

    # Not applicable if no coils were changed
    if num_htg_coils_changed == 0
      runner.registerAsNotApplicable("Not applicable; model contains no gas heating coils.")
      return true
    end

To this:

    # Not applicable if no boilers were changed
    if num_blrs_changed == 0
      runner.registerAsNotApplicable("Not applicable; model contains no boilers.")
      return true
    end
```

Fig. 9.14 Making a fourth Measure change

```
Change this:

    # Record the initial efficiency range of the heating coils
    runner.registerInitialCondition("Gas coils had efficiencies between #{orig_effs.min * 100}% to
#{orig_effs.max * 100}%.")

    # Record the final efficiency of the heating coils
    runner.registerFinalCondition("#{num_htg_coils_changed} gas coils were set to #{eff * 100}%
efficiency.")

To this:

    # Record the initial efficiency range of the boilers
    runner.registerInitialCondition("Boilers had efficiencies between #{orig_effs.min * 100}% to
#{orig_effs.max * 100}%.")

    # Record the final efficiency of the boilers
    runner.registerFinalCondition("#{num_blrs_changed} boilers were set to #{eff * 100}%
efficiency.")
```

Fig. 9.15 Final Measure change



```
Measure Output

▼ GasBoilerEfficiency                 2017-Aug-26 21:15:26     Success

Initial Condition: Boilers had efficiencies between 80.0% to 80.0%.
Final Condition: 4 boilers were set to 85.0% efficiency.
Info: Changing the efficiency of Boiler Hot Water 4 from 0.8 to 0.85
Info: Changing the efficiency of Boiler Hot Water 2 from 0.8 to 0.85
Info: Changing the efficiency of Boiler Hot Water 1 from 0.8 to 0.85
Info: Changing the efficiency of Boiler Hot Water 3 from 0.8 to 0.85
```

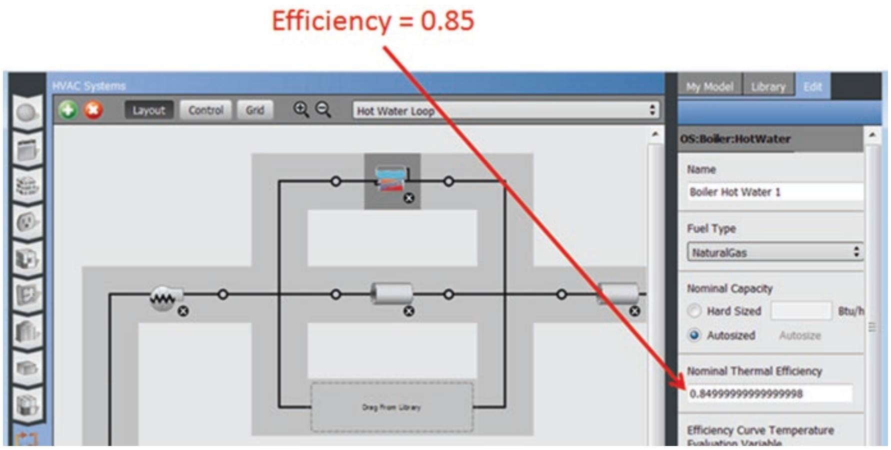
Fig. 9.16 Output from a successful application of our Measure

The final step is testing the Measure on a Model. Run the Measure via “Apply Measure Now” on our school Model. The result should be a dialog that shows the newly modified messages, which refer to boilers instead of gas heating coils, and show the newly modified default efficiency, as shown in Fig. 9.16.

If the Measure output looks good, the next step is to verify the Object values in the Model itself. Click Accept Changes in the “Apply Measure Now” dialog. Proceed to the HVAC ( ) Tab, and then navigate to the Hot Water Loop. Click on the boiler Object and ensure that the efficiency value matches expectations, as shown in Fig. 9.17. If it does not, use the “File/Revert to Saved” menu option to discard these changes and reload the Model as it was before the Measure was applied.

9.5 Creating a New Measure

Both the OpenStudio Application and PAT enable the creation of new Measures from scratch. Clicking the Create New Measure Button in PAT or the Button in the OpenStudio Application opens a window to capture information about the new Measure (Fig. 9.18). The astute reader will quickly realize that the contents of this dialog are used to create a new measure.xml file, and allow the author to specify the nature of the Measure, its intended use, and how it should show up within a Measure search.



OS:Boiler:HotWater

Name: Boiler Hot Water 1

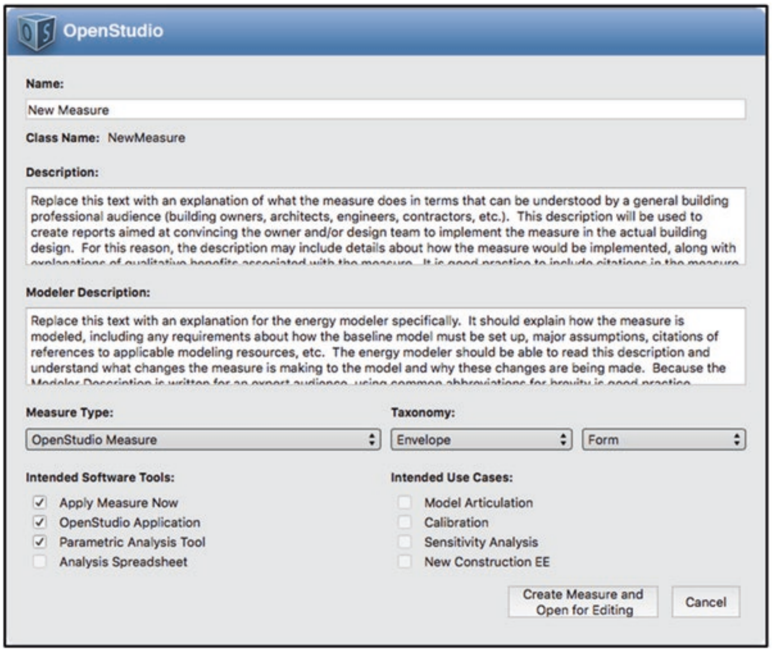
Fuel Type: NaturalGas

Nominal Capacity: Hard Sized Btu/h; Autosized Autosize

Nominal Thermal Efficiency: 0.84999999999999998

Efficiency Curve Temperature Evaluation Variable

Fig. 9.17 Boiler efficiency modified successfully by our new Measure


OpenStudio

Name:
New Measure

Class Name: NewMeasure

Description:
Replace this text with an explanation of what the measure does in terms that can be understood by a general building professional audience (building owners, architects, engineers, contractors, etc.). This description will be used to create reports aimed at convincing the owner and/or design team to implement the measure in the actual building design. For this reason, the description may include details about how the measure would be implemented, along with explanations of qualitative benefits associated with the measure. It is good practice to include citations in the measure

Modeler Description:
Replace this text with an explanation for the energy modeler specifically. It should explain how the measure is modeled, including any requirements about how the baseline model must be set up, major assumptions, citations of references to applicable modeling resources, etc. The energy modeler should be able to read this description and understand what changes the measure is making to the model and why these changes are being made. Because the Modeler Description is written for an expert audience, using common abbreviations for brevity is good practice

Measure Type: OpenStudio Measure

Taxonomy: Envelope; Form

Intended Software Tools:
- ✓ Apply Measure Now
- ✓ OpenStudio Application
- ✓ Parametric Analysis Tool
- Analysis Spreadsheet

Intended Use Cases:
- Model Articulation
- Calibration
- Sensitivity Analysis
- New Construction EE

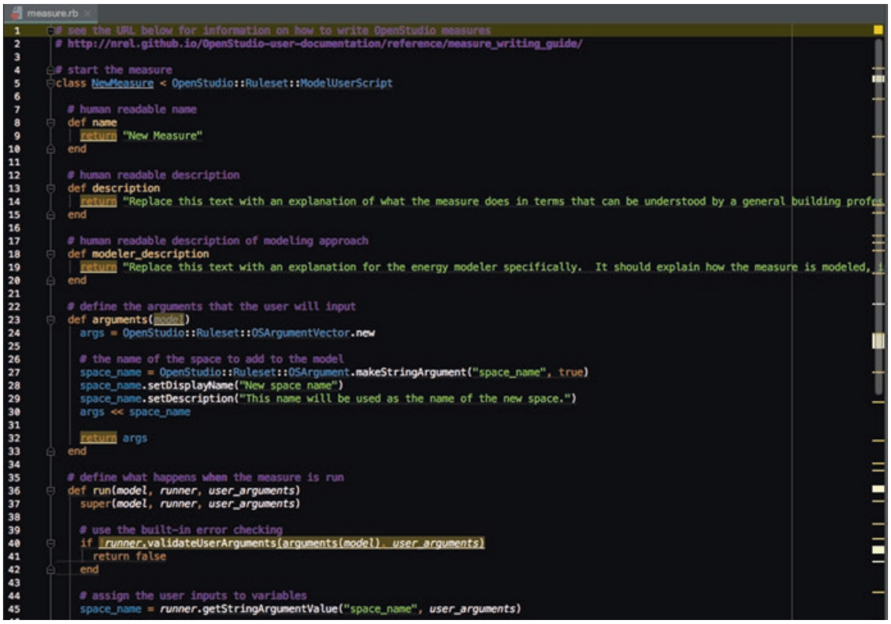
Fig. 9.18 OpenStudio Application Measure creation dialog

The Measure creation process also generates a measure.rb template file, which can be opened in a text editor or a Ruby Integrated Development Environment (IDE) like RubyMine (Fig. 9.19). The template file includes placeholders or "stubs" for required methods that are fully documented in the OpenStudio Measure Writer's Reference Guide. Many of these are self-explanatory – e.g. name, description, modeler_description, etc.

The arguments method parses any required or optional arguments that are passed into the Measure, mapping them to variables that will be used by the Measure's code. As illustrated above, new Measure contains a brief code snippet that illustrates using a single string argument with a default value of "New space name."

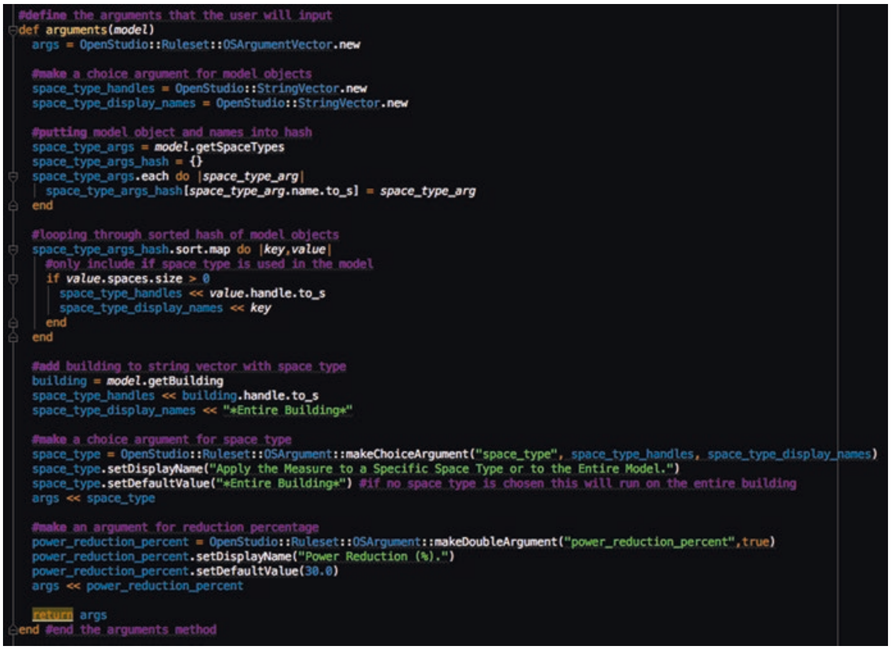
Figure 9.20 presents another sample arguments method, illustrating additional input data types. In this more sophisticated example, the Model that the Measure is being applied to is first queried to identify all of its space types. These space types are then used to populate a menu of choices that comprise the Measure's first argument dynamically. An additional (default) option "*Entire Building*" is also added, allowing the user to specify that the Measure should act on each space in the Model. A second floating point numerical argument related to "power reduction" is also expected by the Measure and defaults to 30. The reader is referred to the Measure Writing Guide for complete documentation on Measure arguments. Existing Measures in the BCL are also an excellent source of examples.

A Measure's run method is where the "real work" is performed. Logic to ensure that inputs are meaningful, perform transformations to the Model, and reportstatus are all contained within the run method. The run method created for a new Measure is shown in Fig. 9.21.



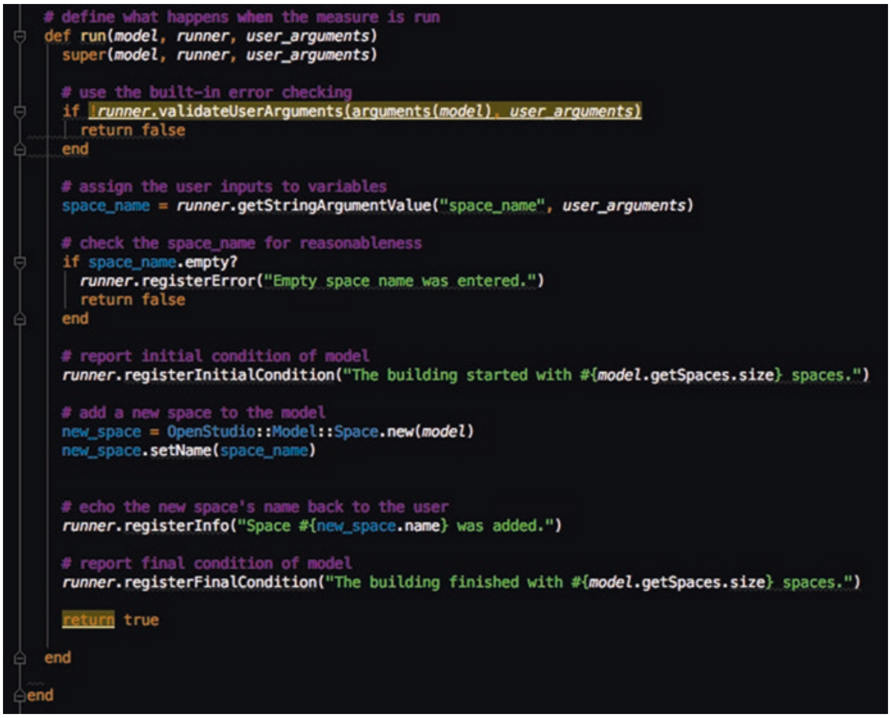
```
# see the URL below for information on how to write OpenStudio measures
# http://nrel.github.io/OpenStudio-user-documentation/reference/measure_writing_guide/

# start the measure
class NewMeasure < OpenStudio::Ruleset::ModelUserScript

  # human readable name
  def name
    return "New Measure"
  end

  # human readable description
  def description
    return "Replace this text with an explanation of what the measure does in terms that can be understood by a general building profe
  end

  # human readable description of modeling approach
  def modeler_description
    return "Replace this text with an explanation for the energy modeler specifically.  It should explain how the measure is modeled, i
  end

  # define the arguments that the user will input
  def arguments(model)
    args = OpenStudio::Ruleset::OSArgumentVector.new

    # the name of the space to add to the model
    space_name = OpenStudio::Ruleset::OSArgument.makeStringArgument("space_name", true)
    space_name.setDisplayName("New space name")
    space_name.setDescription("This name will be used as the name of the new space.")
    args << space_name

    return args
  end

  # define what happens when the measure is run
  def run(model, runner, user_arguments)
    super(model, runner, user_arguments)

    # use the built-in error checking
    if !runner.validateUserArguments(arguments(model), user_arguments)
      return false
    end

    # assign the user inputs to variables
    space_name = runner.getStringArgumentValue("space_name", user_arguments)
```

Fig. 9.19 New Measure template file edited within the RubyMine IDE



```ruby
#define the arguments that the user will input
def arguments(model)
  args = OpenStudio::Ruleset::OSArgumentVector.new

  #make a choice argument for model objects
  space_type_handles = OpenStudio::StringVector.new
  space_type_display_names = OpenStudio::StringVector.new

  #putting model object and names into hash
  space_type_args = model.getSpaceTypes
  space_type_args_hash = {}
  space_type_args.each do |space_type_arg|
    space_type_args_hash[space_type_arg.name.to_s] = space_type_arg
  end

  #looping through sorted hash of model objects
  space_type_args_hash.sort.map do |key,value|
    #only include if space type is used in the model
    if value.spaces.size > 0
      space_type_handles << value.handle.to_s
      space_type_display_names << key
    end
  end

  #add building to string vector with space type
  building = model.getBuilding
  space_type_handles << building.handle.to_s
  space_type_display_names << "*Entire Building*"

  #make a choice argument for space type
  space_type = OpenStudio::Ruleset::OSArgument::makeChoiceArgument("space_type", space_type_handles, space_type_display_names)
  space_type.setDisplayName("Apply the Measure to a Specific Space Type or to the Entire Model.")
  space_type.setDefaultValue("*Entire Building*") #if no space type is chosen this will run on the entire building
  args << space_type

  #make an argument for reduction percentage
  power_reduction_percent = OpenStudio::Ruleset::OSArgument::makeDoubleArgument("power_reduction_percent",true)
  power_reduction_percent.setDisplayName("Power Reduction (%).")
  power_reduction_percent.setDefaultValue(30.0)
  args << power_reduction_percent

  return args
end #end the arguments method
```

Fig. 9.20 Example Measure arguments method



```ruby
  # define what happens when the measure is run
  def run(model, runner, user_arguments)
    super(model, runner, user_arguments)

    # use the built-in error checking
    if !runner.validateUserArguments(arguments(model), user_arguments)
      return false
    end

    # assign the user inputs to variables
    space_name = runner.getStringArgumentValue("space_name", user_arguments)

    # check the space_name for reasonableness
    if space_name.empty?
      runner.registerError("Empty space name was entered.")
      return false
    end

    # report initial condition of model
    runner.registerInitialCondition("The building started with #{model.getSpaces.size} spaces.")

    # add a new space to the model
    new_space = OpenStudio::Model::Space.new(model)
    new_space.setName(space_name)


    # echo the new space's name back to the user
    runner.registerInfo("Space #{new_space.name} was added.")

    # report final condition of model
    runner.registerFinalCondition("The building finished with #{model.getSpaces.size} spaces.")

    return true

  end

end
```

Fig. 9.21 Run method from the new Measure template

This run method includes a call to OpenStudio's "runner.validateUserArguments" method that verifies that the user entered the required inputs with the correct data types. A Measure-specific check ensures that a non-empty space name was entered before proceeding. Measure authors are encouraged to qualify user inputs rigorously and invoke the "runner.registerError" method with a meaningful error message prior to executing "return false" code. This helps ensure that the Measure doesn't generate nonsensical models, providing meaningful feedback to the user.

Related to the topic of providing useful feedback to the user are a trio of OpenStudio methods; "runner.registerInitialCondition," "runner.registerInfo," and "runner.registerFinalCondition." We first experienced the products of these methods in Sect. 6.3.1 with an example Measure that reported the initial and final states of the Model when the ERV Measure was applied to a Model. The new Measure template reports the number of spaces in the initial Model, the name of the new space that was added, and the number of spaces in the final Model. The Measure's logic concludes with "return true," informing the program that called the Measure that it completed normally.

9.6 Checkpoint Fourteen: Creating a New Measure

In this exercise, we will create a new Measure from scratch, starting with just the basic template created by OpenStudio. This final activity assumes that you have read the OpenStudio Measure Writer's Reference Guide online. You will likely find the need to refer back to it often to understand the code that is being used.

The first step in writing a Measure is writing an outline of what it will do. This step may seem silly, but having an outline to refer back to when you get lost in the details of coding can be very helpful. For this example, we are going to make a Measure that adds a user-specified process load, of a user-specified fuel type to the largest Space in the Model. An outline of the steps includes:

1. Get user input for load type (electric or gas),
2. Get user input for the amount of load (W),
3. Identify the largest Space in the Model,
4. Create a process load of the appropriate type and wattage,
5. Assign it to the identified Space, and
6. Report out the load amount that was added and the associated Space.

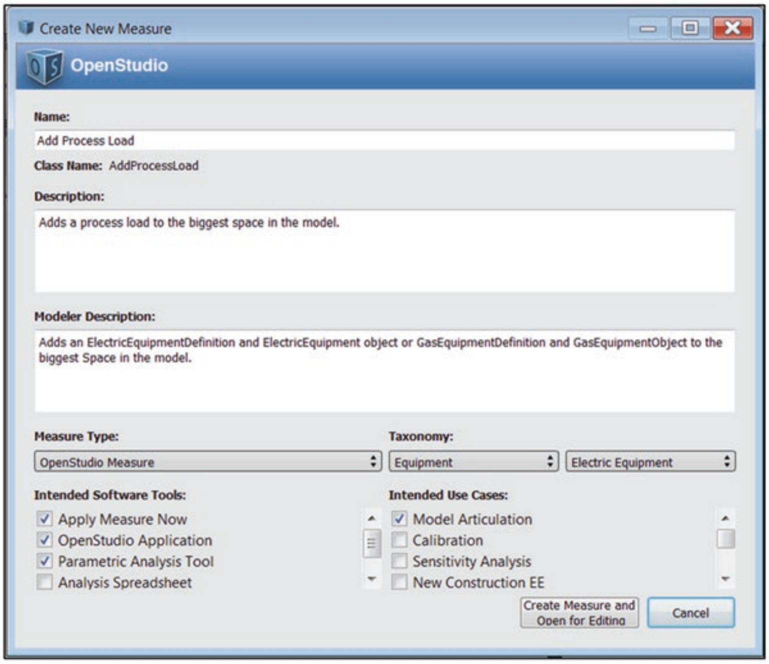
The next step in the process is to go through the OpenStudio GUI and walk through the outline steps. Expand upon the outline by writing down the OpenStudio Object types to be used. This will make it easier to look up the documentation for these objects while writing the Measure. A revised outline might look something like this:

1. Get user input for load type (electric or gas),
    - *Choice input (dropdown list)*

2. Get user input for the amount of load (W),
   - *Number input*
3. Identify the largest Space in the Model,
   - *OpenStudio Space objects*
4. Create a process load of the appropriate type and wattage,
   - *If electric, use OpenStudio Electric Equipment Definition and Electric Equipment objects*
   - *If gas, use OpenStudio Gas Equipment Definition and Gas Equipment objects*
5. Assign it to the identified Space, and
   - *OpenStudio Space objects*
6. Report out the amount of load that was added, and the associated Space.

Create a new Measure using the OpenStudio Application, as shown in Fig. 9.18. Use the inputs shown in Fig. 9.22.

Open the new measure.rb file in a text editor. Select this Measure in the "Apply Measure Now" dialog. It will be found under Equipment > Electric Equipment. You should see that default arguments match those in the measure.rb file. They must be



Name: Add Process Load

Class Name: AddProcessLoad

Description: Adds a process load to the biggest space in the model.

Modeler Description: Adds an ElectricEquipmentDefinition and ElectricEquipment object or GasEquipmentDefinition and GasEquipmentObject to the biggest Space in the model.

Measure Type: OpenStudio Measure

Taxonomy: Equipment, Electric Equipment

Intended Software Tools:
- ✓ Apply Measure Now
- ✓ OpenStudio Application
- ✓ Parametric Analysis Tool
- ☐ Analysis Spreadsheet

Intended Use Cases:
- ✓ Model Articulation
- ☐ Calibration
- ☐ Sensitivity Analysis
- ☐ New Construction EE

Fig. 9.22 Creating the new Measure XML

customized for our application. Create two Measure arguments: one for the load type and another for the wattage as shown in Fig. 9.23.

Add the code from Fig. 9.24 to assign the argument values to variables.

Figure 9.25 contains a code snippet that identifies the largest Space in a Model.

It is best practice to include error handling code and appropriate reporting in a Measure. Figure 9.26 is a simple example that reports if the Model contains no Spaces and exits. If Spaces do exist, then the Measure reports its name and size in ft^2.

The logic shown in Fig. 9.27 adds the appropriate load Object depending upon the fuel type and assigns it to the appropriate Space.

The Measure concludes in Fig. 9.28 with code to report the name of the load that was added to the largest Space.

Test the Measure on an example Model and ensure that it works as intended. If you have written the Measure correctly, the output should look like Fig. 9.29.

You've now written an entire Measure that may be used in the OpenStudio Application, PAT, or other OpenStudio-based applications. By writing an outline, planning the approach, and then dividing code up into small, manageable pieces; the Measure writing process is made less daunting. What tedious modeling tasks do you want to automate next?

```ruby
# Get user input for process load type
load_type_chs = OpenStudio::StringVector.new
load_type_chs << 'Electric'
load_type_chs << 'Natural Gas'
load_type = OpenStudio::Ruleset::OSArgument::makeChoiceArgument('load_type', load_type_chs, true)
load_type.setDisplayName('Load Type')
load_type.setDescription('The type of process load to add.')
load_type.setDefaultValue('Electric')
args << load_type

# Get user input for load wattage
load_wattage = OpenStudio::Ruleset::OSArgument::makeDoubleArgument('load_wattage', true)
load_wattage.setDisplayName('Load Wattage')
load_wattage.setUnits('W')
load_wattage.setDefaultValue(1000.0)
args << load_wattage
```

Fig. 9.23 Measure argument code

```ruby
# assign the user inputs to variables
load_type = runner.getStringArgumentValue("load_type", user_arguments)
load_wattage = runner.getDoubleArgumentValue("load_wattage", user_arguments)
```

Fig. 9.24 Assigning Measure arguments to Measure variables

```ruby
# Find the biggest space (by floor area) in the model
big_space = nil
biggest_area_m2 = 0
model.getSpaces.each do |space|
  # Go to the next space unless this one is bigger than previous spaces
  next if space.floorArea < biggest_area_m2
  # Record this as the biggest space
  big_space = space
  biggest_area_m2 = space.floorArea
end
```

Fig. 9.25 Identifying the largest Space in a Model

```
    # Not applicable if no spaces was found
    if big_space.nil?
      runner.registerAsNotApplicable("No spaces were found in the model.")
      return true
    end

    # Record the biggest space
    biggest_area_ft2 = OpenStudio.convert(biggest_area_m2, 'm^2', 'ft^2').get
    runner.registerInfo("The biggest space is #{big_space.name}, with a floor area of
#{biggest_area_ft2} ft2")
```

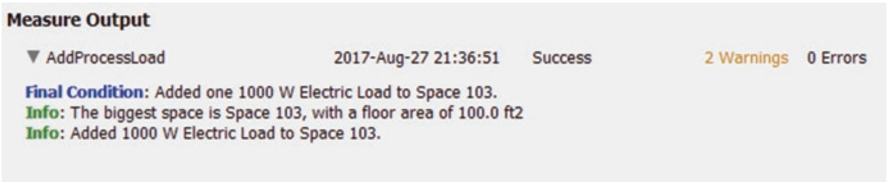
Fig. 9.26 Error trapping logic for models with no spaces

```
    # Make the correct type of load definition
    case load_type
    when 'Electric'
      # Create load definition
      load_def = OpenStudio::Model::ElectricEquipmentDefinition.new(model)
      load_def_name = "#{load_wattage.round} W Electric Load Definition"
      load_def.setName(load_def_name)
      load_def.setDesignLevel(load_wattage)

      # Create load instance
      load = OpenStudio::Model::ElectricEquipment.new(load_def)
      load_name = "#{load_wattage.round} W Electric Load"
      load.setName(load_name)
      load.setMultiplier(1.0)

      # Assign to biggest space
      load.setSpace(big_space)
      runner.registerInfo("Added #{load_name} to #{big_space.name}.")

    when 'Natural Gas'
      # Create load definition
      load_def = OpenStudio::Model::GasEquipmentDefinition.new(model)
      load_def_name = "#{load_wattage.round} W Gas Load Definition"
      load_def.setName(load_def_name)
      load_def.setDesignLevel(load_wattage)

      # Create load instance
      load = OpenStudio::Model::GasEquipment.new(load_def)
      load_name = "#{load_wattage.round} W Gas Load"
      load.setName(load_name)
      load.setMultiplier(1.0)

      # Assign to biggest space
      load.setSpace(big_space)
      runner.registerInfo("Added #{load.name} to #{big_space.name}.")

    end
```

Fig. 9.27 Ruby code that adds electric or gas equipment objects to the largest Space in the Model

```
    # Record the final condition
    runner.registerFinalCondition("Added one #{load.name} to #{big_space.name}.")

    return true
```

Fig. 9.28 Reporting the final state of the Model after the Measure completes



```
Measure Output

  ▼ AddProcessLoad                 2017-Aug-27 21:36:51    Success          2 Warnings   0 Errors

  Final Condition: Added one 1000 W Electric Load to Space 103.
  Info: The biggest space is Space 103, with a floor area of 100.0 ft2
  Info: Added 1000 W Electric Load to Space 103.
```

Fig. 9.29 Successful addition of a process load to the largest Space in a Model

9.7 Checkpoint Fifteen: Testing Measures

As shown in the previous two exercises, the most obvious way to test that a Measure is working correctly is to apply the Measure to a Model and check that the output messages and resulting changes occurred as expected. Some best practices for this step include testing on more than one Model, where differences in the Model's contents may cause issues. Testing on a Model where the Measure should not be applicable is equally valuable to ensure that the applicability is reported correctly.

Advanced users with large collections of their own Measures, or companies where employees share common libraries of Measures need to ensure that Measures continue to work over time. In these situations, it is impractical to manually re-test Measure functionality each time a change is made to OpenStudio itself or Measure code.

Fortunately, OpenStudio borrows from software engineering best practices to provide a solution. This solution is known as unit testing. A unit test is simply a script that runs a piece of code and checks that certain conditions, known as assertions are met. Unit tests for OpenStudio Measures programmatically assert that a Measure behaves correctly for one or more test Models. Assertion is not a guarantee of consistent performance but is a strong indicator.

Navigate to your *MyMeasures* directory to locate your GasHeatingCoil Measure, then open the file called GasHeatingCoil_Test.rb from the /tests subfolder. Figure 9.30 below shows this unit test for the Gas Heating Coil Measure.

Following the approach in the previous checkpoint and read through the file to get a high level understanding of how it works. A quick read of the comments shows that this test:

1. Creates an instance of the Measure,
2. Creates a runner to run the Measure,
3. Loads the test Model,
4. Sets the arguments for the Measure,
5. Runs the Measure on the test Model using the supplied arguments,
6. Shows the output including information, warning, & error messages,
7. Asserts that the Measure was applied successfully,
8. Asserts that the gas heating coils in the Model have the expected efficiencies, and
9. Saves the modified Model for later inspection,

Next, run the unit test for this Measure by using the following steps:

1. Open a terminal/command prompt
2. Navigate to the /tests directory for the Measure by running the command:

   - **On Windows** – *cd C:\path\to\GasHeatingCoilEfficiency\tests*
   - **On Mac** – *cd /Users/yourname/OpenStudio/MyMeasures/GasHeatingCoil Efficiency\tests*

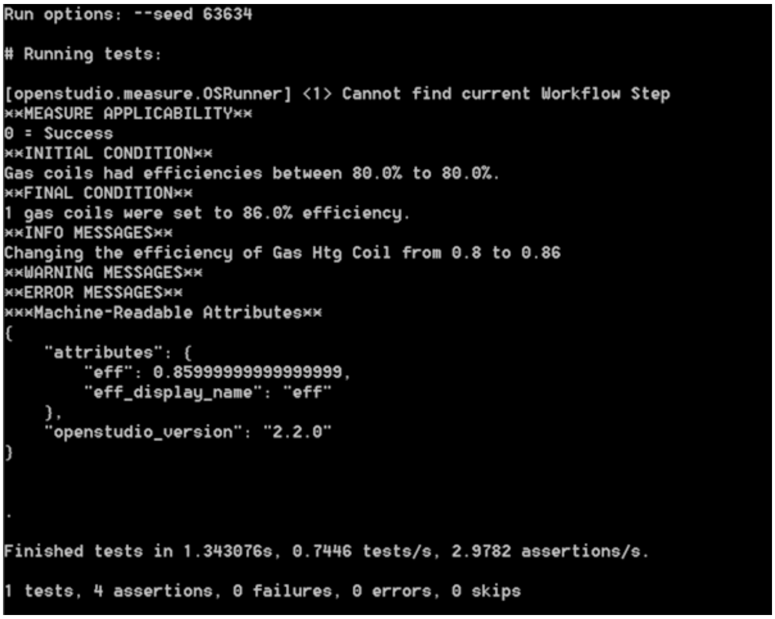
```
require 'openstudio'
require 'openstudio/ruleset/ShowRunnerOutput'

require "#{File.dirname(__FILE__)}/../measure.rb"

require 'minitest/autorun'

class GasHeatingCoilEfficiency_Test < MiniTest::Unit::TestCase

   def test_change_efficiency
    # create an instance of the measure
    measure = GasHeatingCoilEfficiency.new

    # create an instance of a runner
    runner = OpenStudio::Measure::OSRunner.new(OpenStudio::WorkflowJSON.new)

    # load the test model
    translator = OpenStudio::OSVersion::VersionTranslator.new
    path = OpenStudio::Path.new(File.dirname(__FILE__) + "/MyPrimarySchoolHVACTest.osm")
    model = translator.loadModel(path)
    assert(model.is_initialized)
    model = model.get

    # get arguments
    arguments = measure.arguments(model)
    argument_map = OpenStudio::Ruleset.convertOSArgumentVectorToMap(arguments)

    # create hash of argument values.
    # If the argument has a default that you want to use, you don't need it in the hash
    args_hash = {}
    args_hash["eff"] = 0.86
    # using defaults values from measure.rb for other arguments

    # populate argument with specified hash value if specified
    arguments.each do |arg|
      temp_arg_var = arg.clone
      if args_hash.has_key?(arg.name)
        assert(temp_arg_var.setValue(args_hash[arg.name]))
      end
      argument_map[arg.name] = temp_arg_var
    end

    # run the measure
    measure.run(model, runner, argument_map)
    result = runner.result

    # show the output
    show_output(result)

    # assert that it ran correctly
    assert_equal("Success", result.value.valueName)

    # assert that all boilers in the model now have an efficiency of 0.86
    model.getCoilHeatingGass.each do |htg_coil|
      assert_equal(0.86, htg_coil.gasBurnerEfficiency, "Efficiency was not set correctly.")
    end

    # save the model to test output directory
    output_file_path = OpenStudio::Path.new(File.dirname(__FILE__) + "/output/test_output.osm")
    model.save(output_file_path,true)
  end

end
```

Fig. 9.30 A typical Measure unit test

3. Run the command:

   - **On Windows** – C:\openstudio-2.3.0\bin\openstudio GasHeatingCoilEfficiency_Test.rb
   - **On Mac** – /Applications/OpenStudio-2.3.0/bin/openstudio GasHeatingCoilEfficiency_Test.rb

The output in Fig. 9.31 shows the initial condition, the final condition, info, warning, and error messages. The last line shows that there were four assertions



```
Run options: --seed 63634

# Running tests:

[openstudio.measure.OSRunner] <1> Cannot find current Workflow Step
**MEASURE APPLICABILITY**
0 = Success
**INITIAL CONDITION**
Gas coils had efficiencies between 80.0% to 80.0%.
**FINAL CONDITION**
1 gas coils were set to 86.0% efficiency.
**INFO MESSAGES**
Changing the efficiency of Gas Htg Coil from 0.8 to 0.86
**WARNING MESSAGES**
**ERROR MESSAGES**
***Machine-Readable Attributes**
{
    "attributes": {
        "eff": 0.8599999999999999,
        "eff_display_name": "eff"
    },
    "openstudio_version": "2.2.0"
}


.

Finished tests in 1.343076s, 0.7446 tests/s, 2.9782 assertions/s.

1 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

Fig. 9.31 Typical Measure unit test output

made, and that there were zero test failures. This means that all of the checks passed and that the Measure is asserted to work correctly.

Now, modify the unit test for the Gas Boiler Efficiency Measure that was created in the previous checkpoint. The modified code is shown on the next page with the changes highlighted (Fig. 9.32). Try to avoid looking at this code while making the modifications yourself and see how far you can get. Remember the strategy of making small changes and re-running. Remember that error messages typically show the line number where an error exists.

When you have successfully modified the unit test, the output will look like Fig. 9.33.

In order for unit tests to be useful, they must incorporate meaningful checks. By itself, checking that the Measure completed successfully is a poor assertion of success. The Measure could be setting values incorrectly or missing other key behaviors entirely. For this reason, best practice is to include all the checks you would perform when reviewing the output of a Measure successfully in the unit test. Obviously, writing these tests takes time, so the effort needs to be weighed against the cost. For those with large collections of Measures to manage, the upfront cost of developing these tests quickly pays for itself in avoided issues later on.

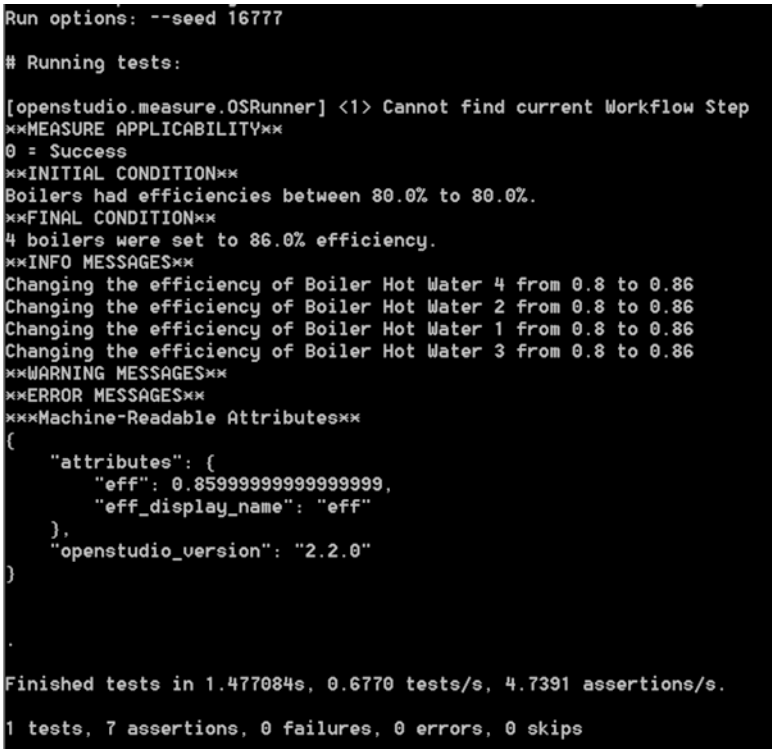
```
require 'openstudio'
require 'openstudio/ruleset/ShowRunnerOutput'

require "#{File.dirname(__FILE__)}/../measure.rb"

require 'minitest/autorun'

class GasBoilerEfficiency_Test < MiniTest::Unit::TestCase

   def test_change_efficiency
    # create an instance of the measure
    measure = GasBoilerEfficiency.new

    # create an instance of a runner
    runner = OpenStudio::Measure::OSRunner.new(OpenStudio::WorkflowJSON.new)

    # load the test model
    translator = OpenStudio::OSVersion::VersionTranslator.new
    path = OpenStudio::Path.new(File.dirname(__FILE__) + "/MyPrimarySchoolHVACTest.osm")
    model = translator.loadModel(path)
    assert(model.is_initialized)
    model = model.get

    # get arguments
    arguments = measure.arguments(model)
    argument_map = OpenStudio::Ruleset.convertOSArgumentVectorToMap(arguments)

    # create hash of argument values.
    # If the argument has a default that you want to use, you don't need it in the hash
    args_hash = {}
    args_hash["eff"] = 0.86
    # using defaults values from measure.rb for other arguments

    # populate argument with specified hash value if specified
    arguments.each do |arg|
      temp_arg_var = arg.clone
      if args_hash.has_key?(arg.name)
        assert(temp_arg_var.setValue(args_hash[arg.name]))
      end
      argument_map[arg.name] = temp_arg_var
    end

    # run the measure
    measure.run(model, runner, argument_map)
    result = runner.result

    # show the output
    show_output(result)

    # assert that it ran correctly
    assert_equal("Success", result.value.valueName)

    # assert that all boilers in the model now have an efficiency of 0.86
    model.getBoilerHotWaters.each do |boiler|
      assert_equal(0.86, boiler.nominalThermalEfficiency, "Efficiency was not set correctly.")
    end

    # save the model to test output directory
    output_file_path = OpenStudio::Path.new(File.dirname(__FILE__) + "/output/test_output.osm")
    model.save(output_file_path,true)
  end

end
```

Fig. 9.32 Modified unit test for the Gas Boiler Efficiency Measure

9.8 The OpenStudio Command Line Interface

The OpenStudio CLI is a compact, cross-platform executable that processes OpenStudio "Workflow" (OSW) files[6] to create and simulate building energy models. Workflows chain together Model files (OSMs) and Measures to automate common modeling tasks. An OSW is a JSON (JavaScript Object Notation) file that

[6] https://nrel.github.io/OpenStudio-user-documentation/reference/command_line_interface/



```
Run options: --seed 16777

# Running tests:

[openstudio.measure.OSRunner] <1> Cannot find current Workflow Step
**MEASURE APPLICABILITY**
0 = Success
**INITIAL CONDITION**
Boilers had efficiencies between 80.0% to 80.0%.
**FINAL CONDITION**
4 boilers were set to 86.0% efficiency.
**INFO MESSAGES**
Changing the efficiency of Boiler Hot Water 4 from 0.8 to 0.86
Changing the efficiency of Boiler Hot Water 2 from 0.8 to 0.86
Changing the efficiency of Boiler Hot Water 1 from 0.8 to 0.86
Changing the efficiency of Boiler Hot Water 3 from 0.8 to 0.86
**WARNING MESSAGES**
**ERROR MESSAGES**
***Machine-Readable Attributes**
{
    "attributes": {
        "eff": 0.8599999999999999,
        "eff_display_name": "eff"
    },
    "openstudio_version": "2.2.0"
}


.

Finished tests in 1.477084s, 0.6770 tests/s, 4.7391 assertions/s.

1 tests, 7 assertions, 0 failures, 0 errors, 0 skips
```

Fig. 9.33 Unit test output for Gas Boiler Efficiency Measure

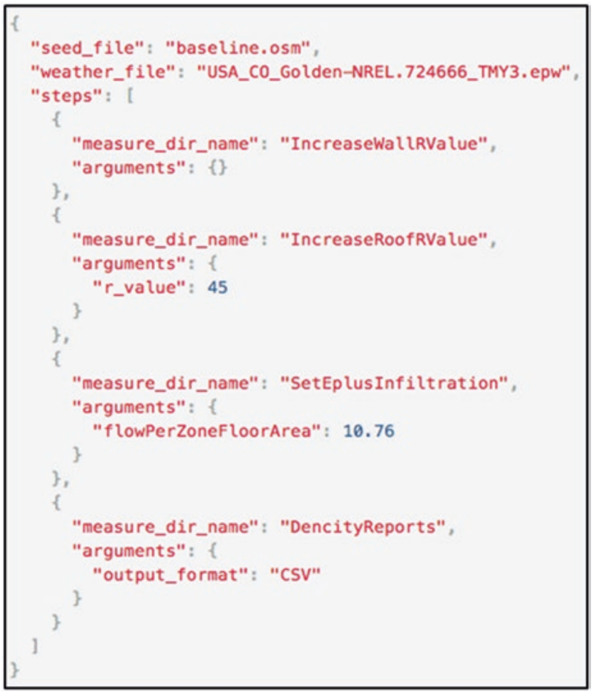
completely specifies a modeling workflow including the initial (seed) Model, weather file, and specific Measures that are to be applied sequentially, along with any arguments the Measures may require (Fig. 9.34).[7]

If Measures are akin to ingredients, then an OSW is the recipe, and the CLI is the cook who uses both to create part of a tasty meal. Figure 9.35 illustrates the basic concept of using the CLI to execute a simple workflow.

As its name would suggest, the CLI is invoked from a Windows, Mac, or Linux command line prompt. Calling the openstudio executable with a -h or --help argument produces the information shown in Fig. 9.36.

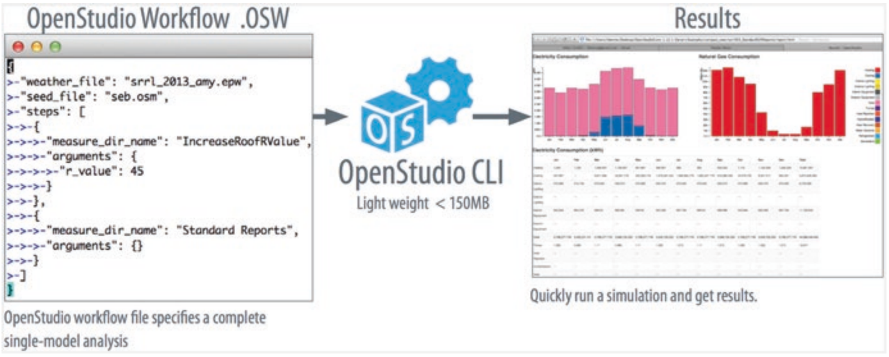
Using the CLI, the implementation details of proposed building energy modeling application become software agnostic. Any programming language that can be used to create an OSW and call the CLI can utilize OpenStudio and its supported engines to perform detailed energy analysis.

[7] https://nrel.github.io/OpenStudio-user-documentation/reference/command_line_interface/#osw-structure



```
{
  "seed_file": "baseline.osm",
  "weather_file": "USA_CO_Golden-NREL.724666_TMY3.epw",
  "steps": [
    {
      "measure_dir_name": "IncreaseWallRValue",
      "arguments": {}
    },
    {
      "measure_dir_name": "IncreaseRoofRValue",
      "arguments": {
        "r_value": 45
      }
    },
    {
      "measure_dir_name": "SetEplusInfiltration",
      "arguments": {
        "flowPerZoneFloorArea": 10.76
      }
    },
    {
      "measure_dir_name": "DencityReports",
      "arguments": {
        "output_format": "CSV"
      }
    }
  ]
}
```

Fig. 9.34 Example OSW file



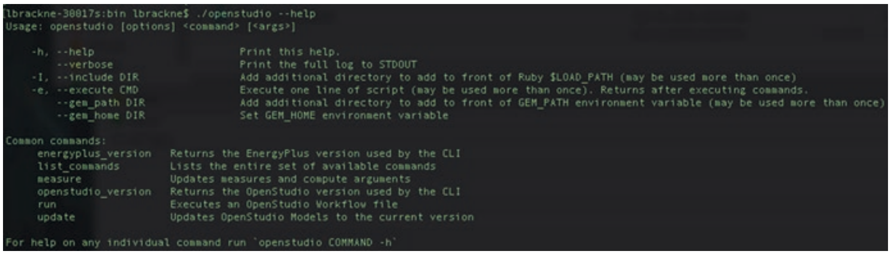
```
{
>-"weather_file": "srrl_2013_amy.epw",
>-"seed_file": "seb.osm",
>-"steps": [
>->-{
>->->-"measure_dir_name": "IncreaseRoofRValue",
>->->-"arguments": {
>->->->-"r_value": 45
>->->-}
>->-},
>->-{
>->->-"measure_dir_name": "Standard Reports",
>->->-"arguments": {}
>->-}
>-]
}
```

OpenStudio Workflow .OSW
OpenStudio workflow file specifies a complete single-model analysis
OpenStudio CLI
Light weight < 150MB
Results
Quickly run a simulation and get results.

Fig. 9.35 OpenStudio CLI enabled analysis (*Credit: Marjorie Schott*)

9.9 The OpenStudio Meta Command Line Interface

The CLI is a powerful tool for creating and simulating individual building models, but it requires a stream of OSW files to drive it. In Chap. 6, we saw this in action as we used PAT to manually generate a series of building Design Alternatives that were essentially individual OSWs, each processed by the OpenStudio CLI.



```
lbrackne-38017s:bin lbrackne$ ./openstudio --help
Usage: openstudio [options] <command> [<args>]

    -h, --help                  Print this help.
        --verbose               Print the full log to STDOUT
    -I, --include DIR           Add additional directory to add to front of Ruby $LOAD_PATH (may be used more than once)
    -e, --execute CMD           Execute one line of script (may be used more than once). Returns after executing commands.
        --gem_path DIR          Add additional directory to add to front of GEM_PATH environment variable (may be used more than once)
        --gem_home DIR          Set GEM_HOME environment variable

Common commands:
     energyplus_version   Returns the EnergyPlus version used by the CLI
     list_commands        Lists the entire set of available commands
     measure              Updates measures and compute arguments
     openstudio_version   Returns the OpenStudio version used by the CLI
     run                  Executes an OpenStudio Workflow file
     update               Updates OpenStudio Models to the current version

For help on any individual command run `openstudio COMMAND -h`
```

Fig. 9.36 OpenStudio CLI help text

In Chap. 7, we explored PAT's ability to generate hundreds or thousands of OSWs automatically. While each individual OSW was still processed by the CLI, they were generated by another part of the SDK called the OpenStudio "Meta" CLI – so-called because it is a CLI that generates inputs for another CLI. If the CLI is the cook, then the Meta CLI is the chef, overseeing all of the individual cooks in the kitchen to satisfy a restaurant full of hungry diners.

Whereas the CLI takes instruction from an OSW, the Meta CLI uses an OpenStudio Analysis (OSA) file. An OSA describes the characteristics of an algorithm that will autonomously create individual OSWs. Example analyses include the use of sampling or optimization algorithms that dynamically generate combinations of seeds, Measures, and Measure arguments. Such algorithms enable rapid evaluation of efficiency performance spaces, optimized designs based on multiple objectives, or calibration of energy models against metered consumption data.

OSAs are also JSON files, but specify one or more seed models, one or more weather files, an algorithm and its parameters, and a set of OpenStudio Measures along with their parameters. Unlike workflow Measure arguments that are prescribed in an OSW, an OSA can describe ranges, distributions, or sets of arguments that can be explored by the algorithm. The schema for the OSA is currently documented in https://github.com/NREL/OpenStudio-analysis-gem. In Chap. 7, we utilized PAT to generate a number of different OSAs.

The Meta CLI is a compact, cross-platform executable that consumes OSA files and executes them using local, cluster, or cloud computing resources.[8] Like the OpenStudio CLI, the Meta CLI can be used to create (more advanced) applications. Figure 9.37 describes the basic concept of using the Meta CLI to execute an analysis. It also illustrates the relationship between the Meta CLI, OpenStudio Server,[9] and parallel computing "workers" running individual instances of the OpenStudio CLI.

As with the CLI, the Meta CLI is invoked from the command line. Invoking it with a -h or --help argument returns basic guidance on using it to start or stop an OpenStudio Server and run an OSA on it (Fig. 9.38).

[8] https://github.com/NREL/OpenStudio-server/blob/develop/bin/openstudio_meta

[9] https://github.com/NREL/OpenStudio-server

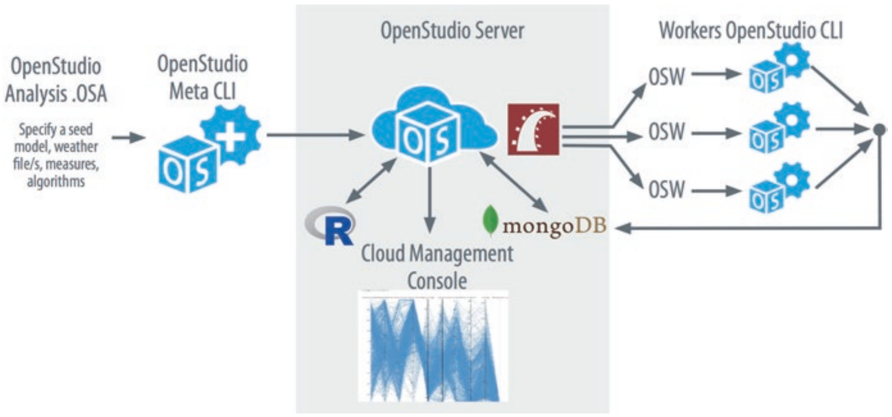
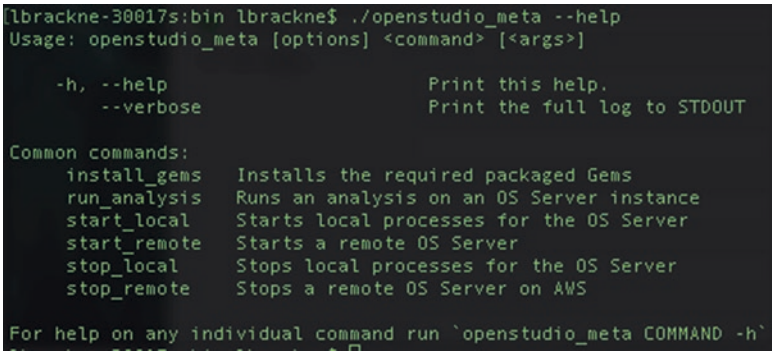


Fig. 9.37 OpenStudio Meta CLI enabled analysis (*Credit: Marjorie Schott*)



```
[lbrackne-30017s:bin lbrackne$ ./openstudio_meta --help
Usage: openstudio_meta [options] <command> [<args>]

    -h, --help                       Print this help.
        --verbose                    Print the full log to STDOUT

Common commands:
     install_gems   Installs the required packaged Gems
     run_analysis   Runs an analysis on an OS Server instance
     start_local    Starts local processes for the OS Server
     start_remote   Starts a remote OS Server
     stop_local     Stops local processes for the OS Server
     stop_remote    Stops a remote OS Server on AWS

For help on any individual command run `openstudio_meta COMMAND -h`
```

Fig. 9.38 OpenStudio Meta CLI help text

Use of the OpenStudio Meta CLI outside of PAT is an advanced software engineering topic that can best be explored via documentation located at https://github.com/NREL/OpenStudio-server.

9.10 Additional Exercises (for Advanced Users)

1) Continue learning about OpenStudio Measures by creating additional variations of existing Measures or write an entirely new one.
2) Explore the OpenStudio CLI by creating some simple calculators that take input from the user, write one or more OSW files, and call the CLI. A few simple ideas include:

- A before and after retrofit calculator that involves two OSW files – both with the same seed Model and weather file. The two workflow files would only differ in the EE Measures and/or Measure arguments applied in the retrofit case.
- A simple calculator that assesses the energy savings potential of a new EE technology across many building types by utilizing an empty seed model and calling the DOE Prototype Building Measure as the first step in a workflow.

References

https://bcl.nrel.gov/search/site/GLHEPro?f[0]=bundle%3Anrel_measure
http://codebeautify.org/xmlviewer
https://hvac.okstate.edu/glhepro/overview
https://github.com/NREL/OpenStudio-analysis-gem
https://github.com/NREL/OpenStudio-server
https://github.com/NREL/OpenStudio-server/blob/develop/bin/openstudio_meta
https://nrel.github.io/OpenStudio-user-documentation/reference/command_line_interface
https://nrel.github.io/OpenStudio-user-documentation/reference/command_line_interface/#osw-structure
http://nrel.github.io/OpenStudio-user-documentation/reference/measure_writing_guide
https://www.ruby-lang.org/en/documentation/quickstart/