Hardware Triggered Scanning: Course 2

Philip Taylor, Emma Arandjelovic

Observatory Sciences Limited

Course Aims

1. Understand the key concepts behind Malcolm
2. Get experience in setting up Malcolm from scratch
3. Understand the different strategies for hardware triggered scanning on beamlines
4. Learn how to configure Malcolm to run on beamlines

Course Content

1. Refresher and basic concepts
   - Blocks, parts, and controllers
2. Scanning components
   - Manager and Runnable controllers
   - Scan Point Generator
   - Designs
3. Scanning control
   - Scan layer
   - AreaDetector
4. From the classroom to the beamline
   - Scanning strategies
   - EPICS and motion control requirements
   - Configuring Malcolm to run on beamlines

Refresher: Software Stack

Data Analysis WorkbeNch
- Analysis and visualization

Generic Data Acquisition
- Experiment setup and supervision

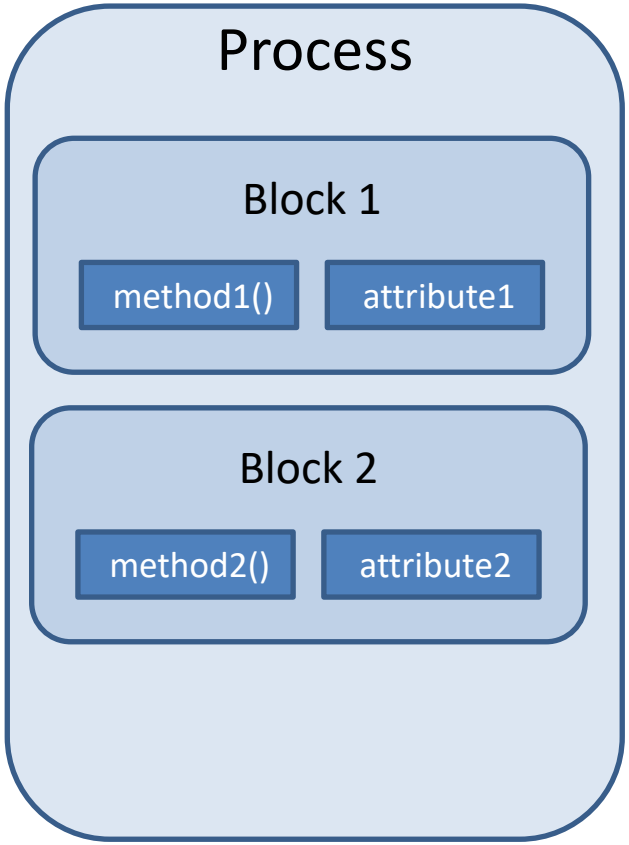
Malcolm
- Scan configuration

Experimental Physics & Industrial Control System
- Low level control of hardware

Refresher: What is Malcolm?

- Generic and extensible framework for scanning
- Middle layer between GDA and control system
- Implemented in python™
- Creates a s/w map of the h/w layer
- https://pymalcolm.readthedocs.io
- Web GUI called MalcolmJS
- https://malcolmjs.readthedocs.io

Refresher: Blocks



- A block is a user-centred view

  Examples:
  - Motion controller
  - PandA
  - Detector
  - ‘Hello World’ program

- It is an *interface* to an object’s:
  - Methods (actions)
  - Attributes (data)

Blocks Continued…

- A process hosts multiple blocks
- Blocks are given a unique **MRI**

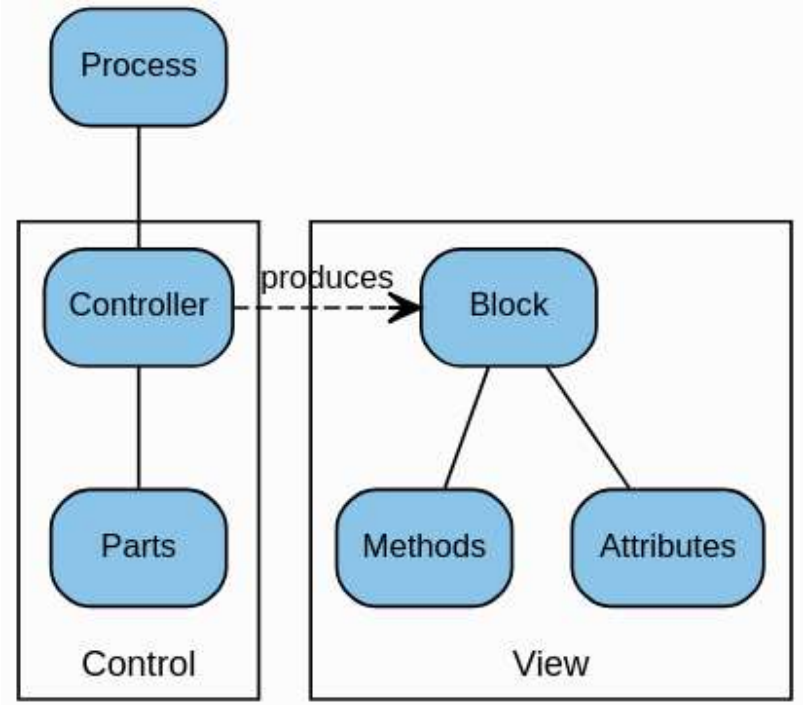
  (*Malcolm Resource Identifier*)

  – This is used by clients to address the block
- A block is just the interface

  – Contains no code!

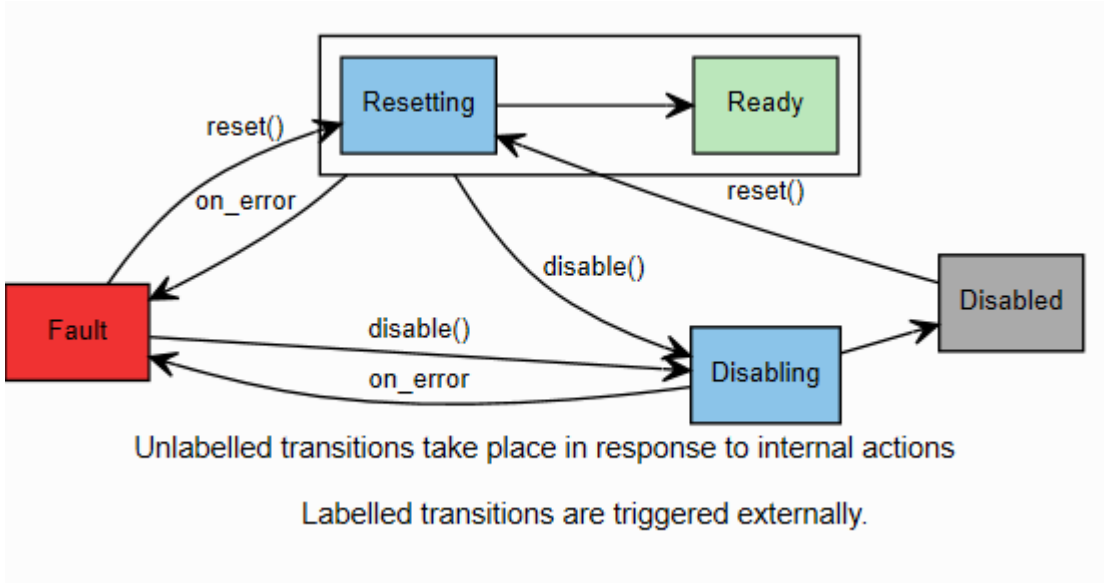
Parts and Controllers

- Methods and attributes are contributed in Python **Parts**
- A **Controller** provides a co-ordinating framework:
  - It creates a Block View that we interact with
  - It populates this with its own methods and attributes and also those from the contributed Parts



Stateful Controller

- Used by blocks in the Hardware Layer
- Implements a state machine:



Getting Started

- Stop BL4xP-ML-MALC-01 (Ctrl-T then Ctrl-X)
- Get the code:

git clone https://github.com/dls-controls/pymalcolm

- Run the Hello World example:

./malcolm/imalcolm.py malcolm/modules/demo/DEMO-HELLO.yaml

NB: The examples are based on tutorials which you can also try in your own time:
https://pymalcolm.readthedocs.io

Demo 1: Hello World

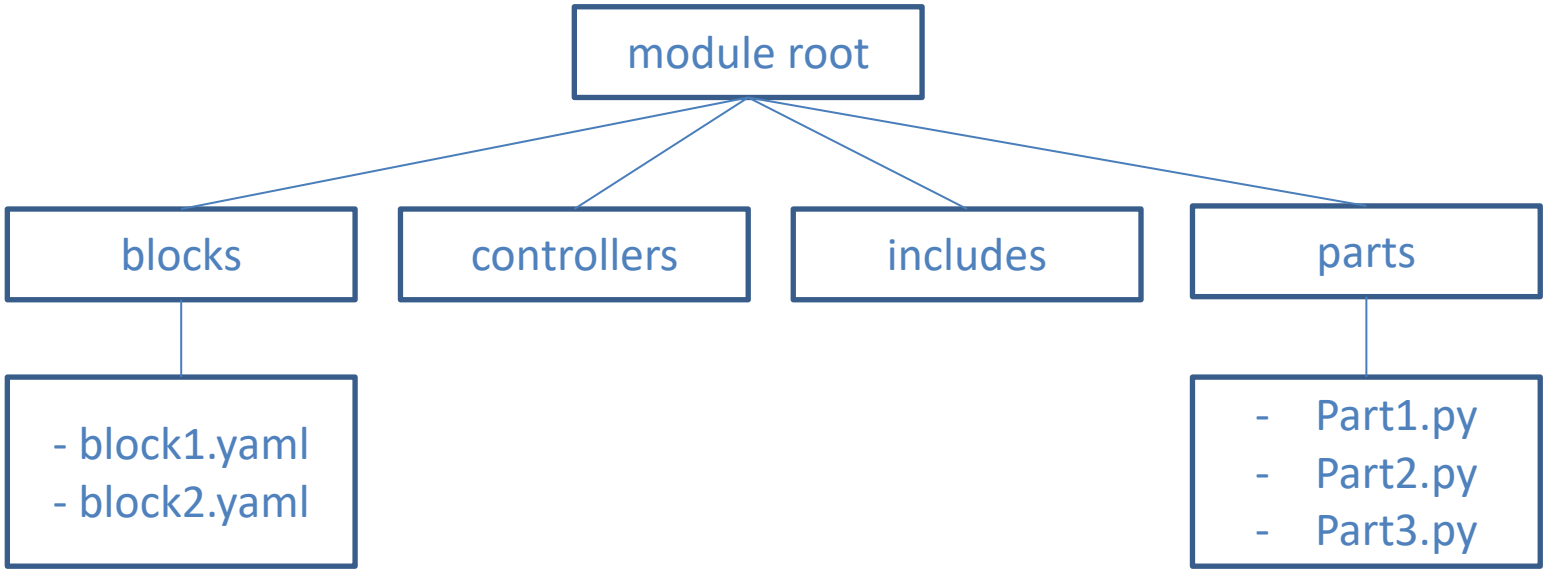
imalcolm launches an *IPython* interactive console

In the console, try:

```
>>> hello = self.block_view("HELLO")
>>> hello.greet("me")
Manufacturing greeting...
'Hello me'
>>> result = hello.greet(“Emma")
Manufacturing greeting...
>>> print(result)
Hello Emma
```

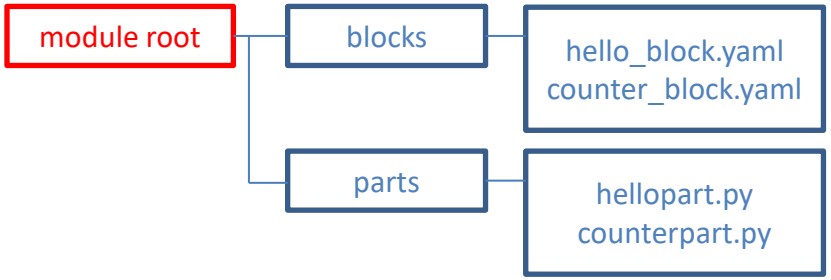
- *self* is a context we can use to get hold of block views
- *“HELLO”* is the MRI of our block
- The *hello* object is now a view of our HELLO block that we can use to call its methods
- The return value is assigned to the variable *result*

Module Structure



- Each item identified using the '.' separator, e.g. *demo.blocks.hello_block* and *demo.parts.HelloPart*

Looking Under the Hood



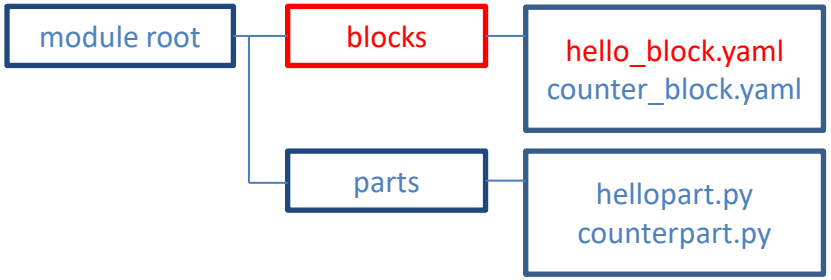
DEMO-HELLO.yaml

```
# Create some Blocks
- demo.blocks.hello_block:
    mri: HELLO

- demo.blocks.hello_block:
    mri: HELLO2

- demo.blocks.counter_block:
    mri: COUNTER

# Add a webserver
- web.blocks.web_server_block:
    mri: WEB
```

Instantiates 3 blocks. Each one defines a unique **M**alcolm **R**esource **I**dentifier (MRI)

Web server block starts an HTTP server on port 8008

Hello Block Definition



demo/blocks/hello_block.yaml:

```
- builtin.parameters.string:
    name: mri
    description: Malcolm resource id

- builtin.defines.docstring:
    value: Block with a greet() Method

- builtin.controllers.BasicController:
    mri: $(mri)
    description: $(docstring)

- demo.parts.HelloPart:
    name: hello
```

Defines parameters to be defined when instantiating the block. Value obtained using $(mri)

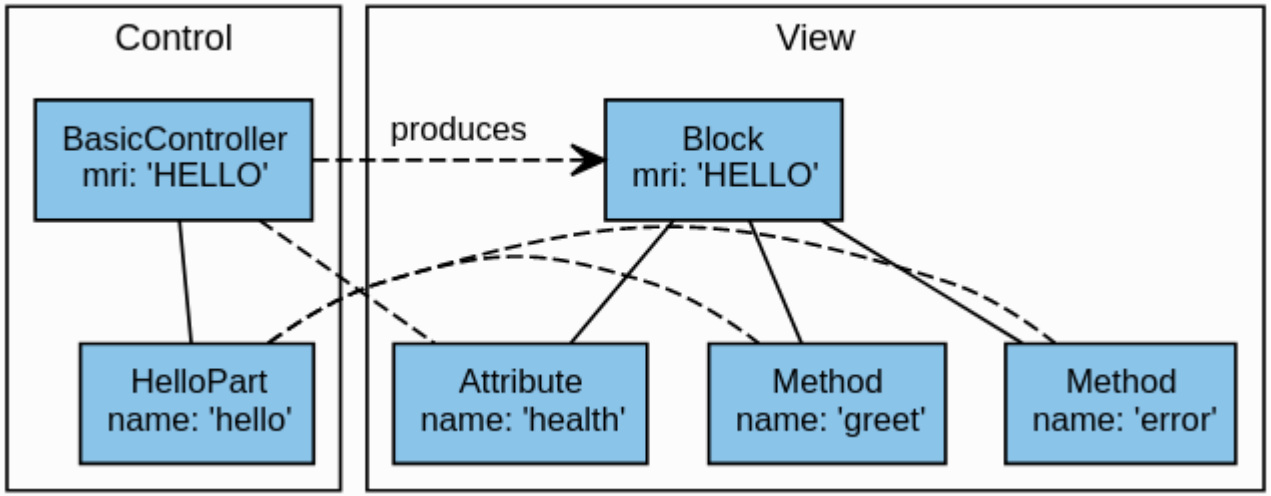
Defines the $(docstring) variable which describes the block's function

Defines the controller which will create the block for us. The *BasicController* is a simple container for parts

Defines the part which contributes the functionality of the block

- *name* is the (unique) name of the part within the controller

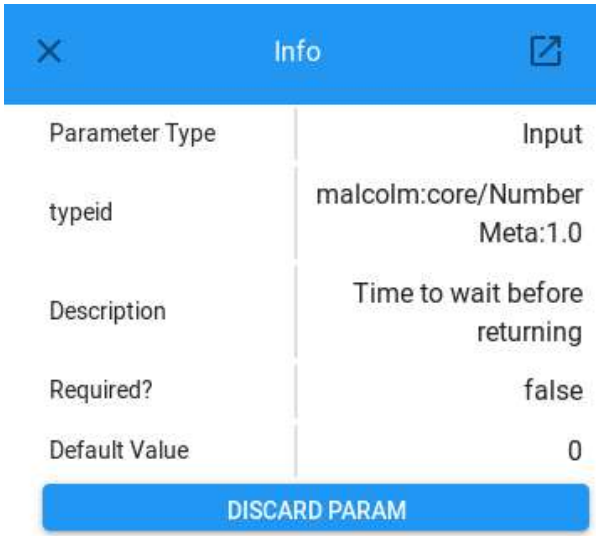
Hello World Structure



- The BasicController contributes a ‘health’ attribute
- The business logic methods are contributed by our HelloPart

Brief Aside: Annotypes

- Library for annotating Python types with metadata
- Allow clients to discover parameter type information at runtime



Brief Aside: Annotypes

In the Python part:

```
from annotypes import Anno, add_call_types
with Anno("my variable"):
    AMyVar = str
```

Variable description

Variable name

Variable type

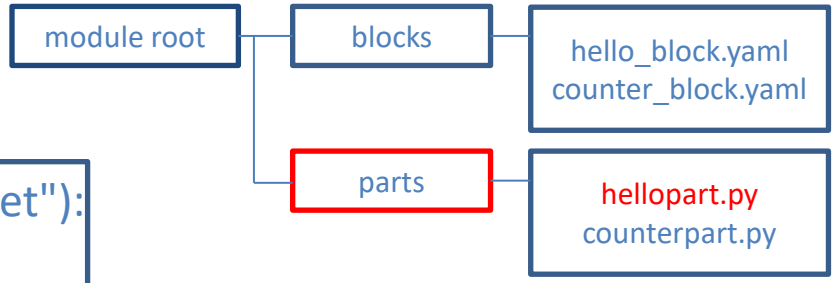
add_call_types is a Python decorator which uses our Annotype definitions to provide introspection information

Brief Aside: Annotypes

```
with Anno("Description of parameter 1 (a string)"):
    AParam1 = str
with Anno("Description of parameter 2 (an int)"):
    AParam2 = int
with Anno("Description of the return value (a float)"):
    AReturn = float

@add_call_types
    def myMethod(self, param1, param2):
        # type: (AParam1, AParam2) -> AReturn
```

See: https://github.com/dls-controls/annotypes

Hello Part Definition



demo/parts/hellopart.py:

```
with Anno("The name of the person to greet"):
    AName = str
with Anno("Time to wait before returning"):
    ASleep = float
with Anno("The manufactured greeting"):
    AGreeting = str

class HelloPart(Part):

    def setup(self, registrar):
        # type: (PartRegistrar) -> None
        super(HelloPart, self).setup(registrar)
        registrar.add_method_model(self.greet)
        registrar.add_method_model(self.error)
```

- Type definition information
- Class must extend *Part* and provide a *setup* method
- The *PartRegistrar* is a utility object used to register methods and attributes with the block

Hello Part Definition

demo/parts/hellopart.py (continued…):

```
@add_call_types
  def greet(self, name, sleep=0):
    # type: (AName, ASleep) -> AGreeting
    """Optionally sleep <sleep> seconds, then
       return a greeting to <name>"""
    print("Manufacturing greeting...")
    sleep_for(sleep)
    greeting = "Hello %s" % name
    return greeting

  def error(self):
    """Raise an error"""
    raise RuntimeError("You called error()")
```

- *greet* method contains the 'business logic' of the part
- *error* method can be used to raise an error if something goes wrong
- The three Annotypes we defined earlier are used to provide introspection information

Connecting a Client

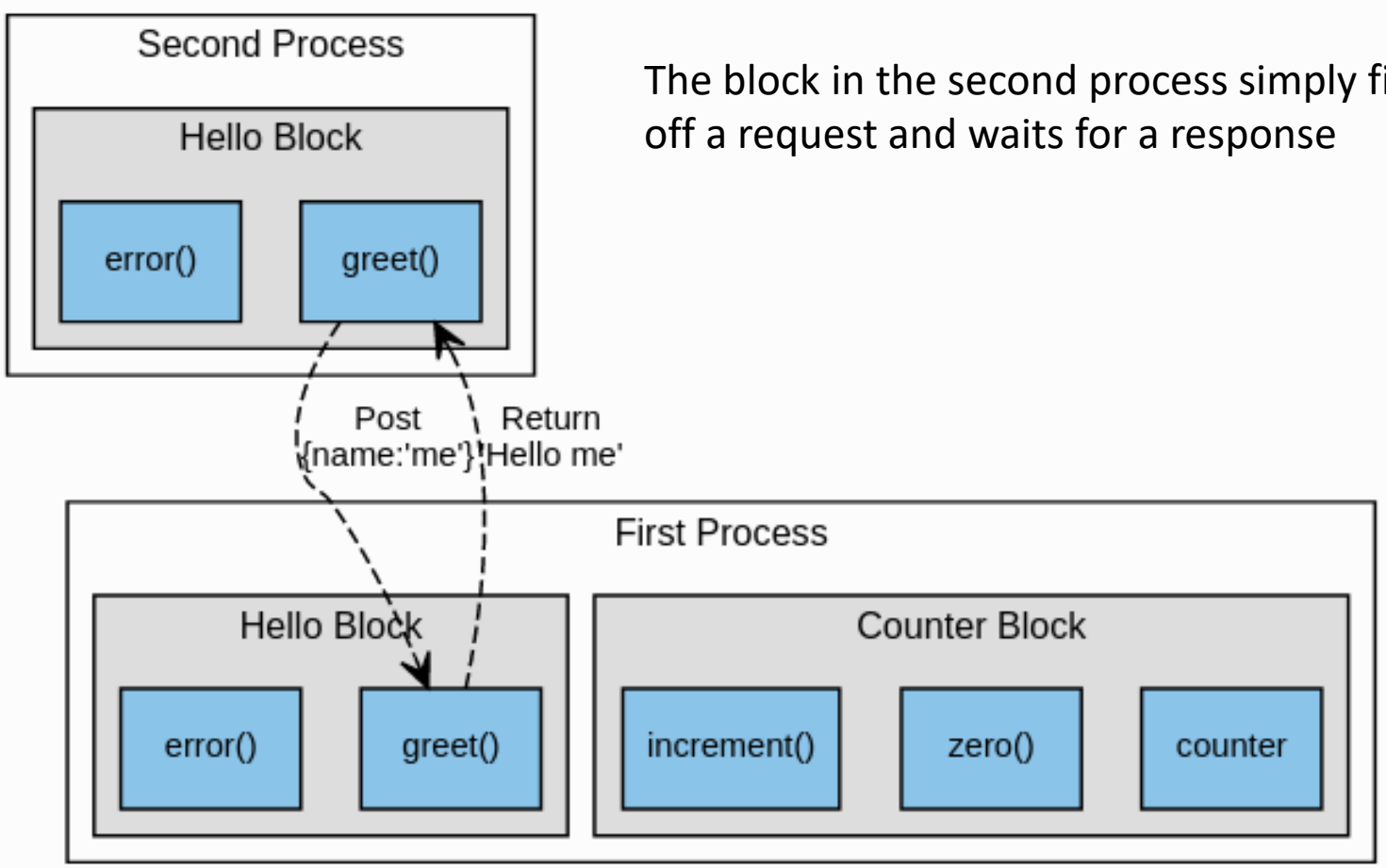
Start up a second process to connect to the first:

./malcolm/imalcolm.py –c ws://localhost:8008

```
>>> self.make_proxy("localhost:8008", "HELLO")
>>> self.block_view("HELLO").greet("me")
u'Hello me'
```

Check the output from the first process to see this is the one doing the actual “work”

Connecting a Client



The block in the second process simply fires off a request and waits for a response

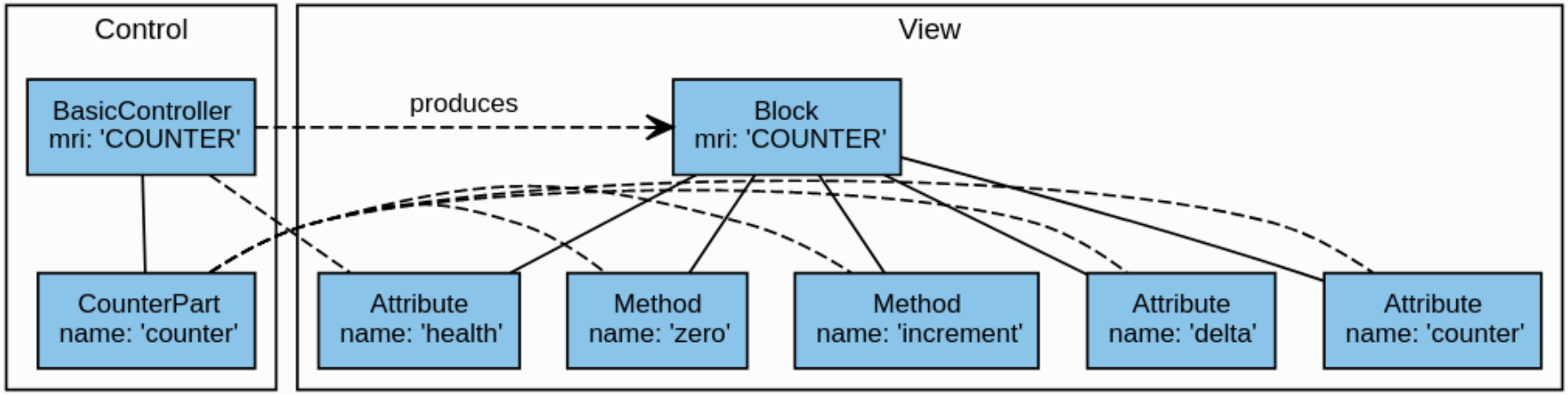
Demo 2: Adding Attributes

- We have seen how to add a method to a block
  - registrar.**add_method_model**(<method>)

- To add attributes we need a little more information:
  - registrar.**add_attribute_model**(
    "<attribute_name>",
    <attribute_model_instance>, → Contains type info for validation
    <method>) → Method called when someone tries to set the value

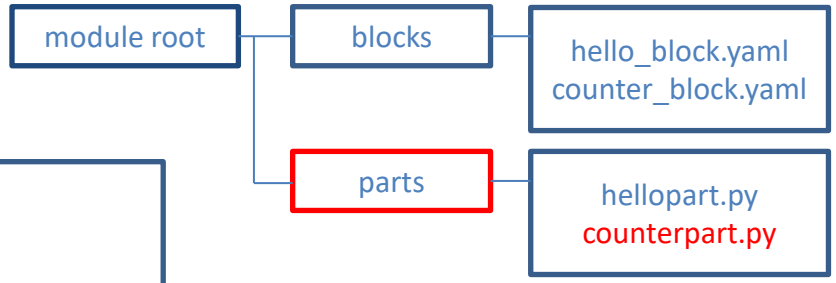
Counter Demo Structure

- a writeable attribute counter which keeps the current counter value
- a writeable attribute delta which stores the amount to increment by
- a method zero() which will set counter = 0
- a method increment() which will set counter = counter + delta



➢ Notice the additional attribute 'health' contributed by the BasicController

Counter Part Definition



demo/parts/counterpart.py:

```
class CounterPart(Part):

    counter = None  # type: AttributeModel
    delta = None    # type: AttributeModel

    def zero(self):
        """Zero the counter attribute"""
        self.counter.set_value(0)

    def increment(self):
        """Add delta to the counter attribute"""
        self.counter.set_value(self.counter.value +
                self.delta.value)
```

- Define the attributes as type *AttributeModel*
- Use the AttributeModel's *set_value* method to update the counter's value
- Use its *value* attribute to get its value

Counter Part Definition

demo/parts/counterpart.py (continued):

```
def setup(self, registrar):
    # type: (PartRegistrar) -> None
    super(CounterPart, self).setup(registrar)
    self.counter  = NumberMeta(
        "float64", "The current counter value",
        tags=[config_tag(),
        Widget.TEXTINPUT.tag()]
    ).create_attribute_model()

    self.delta  =  NumberMeta(
        "float64", "Amount to increment by",
        tags=[config_tag(),
        Widget.TEXTINPUT.tag()]
    ).create_attribute_model(
        initial_value=1)
```

- Call the parent class *setup* method
- Create a *NumberMeta* object:
  - Type, description
  - Tags indicating usage info
- Create the *AttributeModel* from the meta object, supplying a default initial value if desired

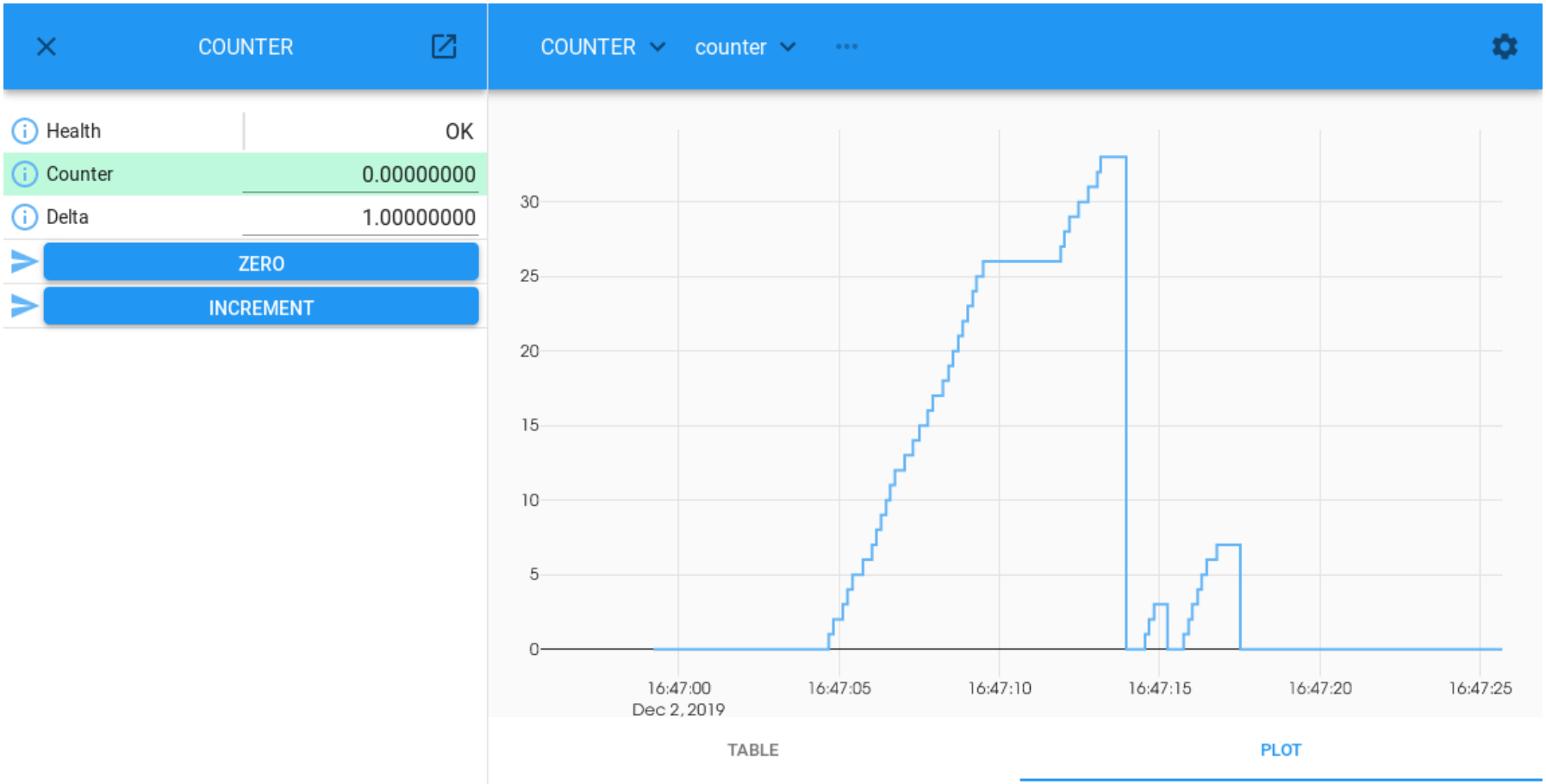
Counter Part Definition

demo/parts/counterpart.py (continued):

```
registrar.add_attribute_model(
        "counter",
        self.counter,
        self.counter.set_value)

registrar.add_attribute_model(
        “delta",
        self.delta,
        self.delta.set_value)

registrar.add_method_model(self.zero)

registrar.add_method_model(self.increment)
```

- Register the attribute models:
  - Name,
  - Model,
  - Method name
    - If not supplied, the attribute is read-only
- Register the method models as before

Running the Example



Practical Exercises

1. Modify the *Hello* example to throw an error saying “No name supplied!” if the name is an empty string.
2. Try setting the counter attribute to a non numeric value. What happens?
3. Modify the *Counter* example to add a new *decrement* method which decrements the counter by the delta value.