Environment Development Kit Middleware

User Guide

Version 0.1
June 2015

Environment Development Kit Middleware – User Guide

Contents

Disclaimers

The Environment Development Kit (EDK) middleware is currently in early alpha release. While efforts have been made to remove bugs from the system, it is possible some remain. Users of the EDK and its API do so at their own risk. Devices being controlled with the EDK should not be left unsupervised at any time. The EDK creators accept no responsibility for any damage to persons or property arising directly or indirectly as a result of using the system or any related software. By using the EDK you indicate your agreement to these terms.

The EDK was developed as part of ScaleUp, an international collaboration project between the University of Essex in the UK and King Abdulaziz University in Saudi Arabia.

1 Introduction

The Environment Development Kit (EDK) is a middleware system designed to allow external computer programs to access and potentially control a broad range of individual 'smart devices'. Entire intelligent environments, containing multiple devices of various types each connected via a common network, can also be represented using the mechanism API.

1.1 System Architecture

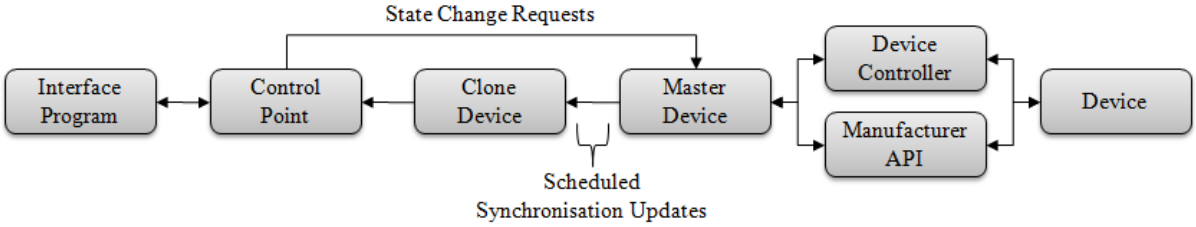

Fig. 1: The Environment Development Kit (EDK) Architecture

Figure 1 provides an overview of the entire EDK middleware architecture. In most cases the system described would be split at the 'Scheduled Synchronisation Updates' point, with each sub-section being deployable on different computers if desired. Master devices form the core of the EDK middleware architecture. Each master device includes a controller which links the virtual EDK representation to some physical hardware in the real-world. This controller may be bespoke or based upon a design provided by a specific device manufacturer. Once deployed on an EDK network, a master device will periodically broadcast details of itself and the current state of any associated hardware, to a specified multicast group.

EDK control points added to interface programs (e.g. agents), connect to the network and receive incoming update messages sent by master devices, via a subscribed multicast group. Upon first encountering a new master device, a control point clones it into a new instance that is stored locally the computer running the interface program. If several different control point instances existed on an EDK network, each would possess a different clone created from the same master device. Controllers are not passed on to clones when they are created, so master devices provide the only gateway to an associated piece of hardware.

1.2 Interfacing Platforms

The EDK currently features two distinctly different interfacing methods. Firstly, if enabled the middleware is capable of automatically generating a series of Web Services, which can be accessed in any web browser and used to remotely access and control the associated smart device. If the Web Services feature is enabled on an EDK control point, information about the entire network of a connected intelligent environment can be viewed.

Alternatively, when writing agents or other control programs that need to directly access smart devices, the EDK also offers an API written using the Java programming language. Through the API programs can create control points to search for any or specific smart devices connected to a network. Once discovered the same API and control point can be used to take readings from or make alterations to the state of these devices. Examples of how to implement these features are provided in Section 2.3.

1.3 Software Requirements

To operate the EDK requires Oracle's Java SDK version 6 or above. Compatibility with Java distributions provided by other organisations is not guaranteed. The current version of the middleware has been tested using Microsoft Windows and OSX.

When programming applications using the API it is highly recommended to import the EDK into an IDE software package such as NetBeans. A compatible network-enabled web browser application is required to access and view output from the Web Services interface method.

2 The EDK API

This section provides instructions on how to use the EDK middleware API to implement control systems for individual smart devices or more complex intelligent environments. Firstly, a guide on how to use the EDK API to implement and deploy a master device is presented. Following this, the next section details how to create a control point instance create clones of master devices in interface programs. The subsequent sections then explain how to read from a deployed device to learn its state, and finally, outlines how to send state action requests to an actuator, prompting the virtual representation, (and by extension any connected physical hardware), to adopt a new user specified state.

2.1 Implementing Master Devices

Any smart device requires three key components in order to exist on an EDK network;

1) A representation identifying the attributes of the device and declaring whether it is an actuator or sensor.
2) A gateway to the target EDK network from where the device will be accessible.
3) A controller connecting the virtual EDK representation with some counterpart hardware that exists in the real-world.

Together these components allow a 'master' device to be created using the EDK API. Once deployed on the network the virtual device representation will monitor or change the state of the real hardware based upon received user instructions. It will also periodically broadcast details of itself and the current state of the hardware to the EDK network, which is received by any active control points and used to create or synchronise associated 'clone' devices.

Master devices can be implemented and deployed remotely on one or several networked computers. Depending upon personal preference a master device application could also be used to generate one or several different actuators and sensors. They don't even need to be the same type of device. Multiple different control programs can be imported into the same implementation and if desired a group of devices can either share or each be allocated their own instance of the same controller. The remainder of this section will now explain the basics of how to implement a simple master device application.

2.1.1 Step One: Declaring Constants

```
private final int COMMUNICATIONS_PORT = 4446;
private final int WEBSERVICES_PORT = 8000;
private final String CONTROLLER_CLASS = "Controller";
private final String CONTROLLER_PACKAGE = "devicecontroller";
private final String CONTROLLER_DIRECTORY = "controllers";
private final String NETWORK_ADDRESS = "230.0.0.1";
```

Fig. 2: An example of communication and controller constants.

Firstly, several constants need to be declared to allow the communications and different control systems to operate correctly. Figure 2, provides examples of these and each attribute will be explained further when actually used later in the implementation process.

2.1.2 Step Two: Creating Device Instances

```
BooleanLight light1 =
        new BooleanLight("Light1", "Power", "Light Mk 1");
light1.addStateChangeListener(sl);
```

Fig. 3: Creating a new single variable smart device

```
DimmableLight dimmer1 = new DimmableLight("Dimmer1",
       new String[] { "Power", "Brightness" }, "Dimmer Light Mk 1");
dimmer1.addStateChangeListener(sl);
```

Fig. 4: Creating a new multi-variable smart device

The EDK API contains several different classes that can be used to create new representations of individual master devices. Which class is most appropriate for a given situation depends upon several factors, specifically, the number of state variables supported by the device and whether it is an actuator or a sensor. For common device types, the EDK API contains several dedicated classes, many of which contain bespoke convenience methods designed to allow better control of specific state variables. For example, Figures 3 and 4 show how to create instance of two different types of light emitting device. Both types have a state variable 'Power' declared, which is used to control whether the device is on or off. To support these actions, both the 'BooleanLight' and 'DimmableLight' classes contain two convenience methods 'turnOn' and turnOff' which allow the 'Power' state variable to be set without the need to directly declare or process any new values. The 'DimmableLight' class used in Figure 4 additionally contains 'getBrightness' and 'setBrightness' methods, which relate to the declared 'Brightness' state variable, used to control the emitted light level. Without these convenience methods the state variable name would need to be declared along with any new state value, (if applicable), in a formatted String in order to perform the same function. This procedure is discussed in more detail in Section 2.3.2.

If the device being used is uncommon, or unknown for some reason, the EDK API also includes a series of generic classes which can create representations based purely on the number of declared state variables. For actuators these would be 'SingleVariableActuator' and 'MultiVariableActuator', whereas for sensors the appropriate generic classes would be 'SingleVariableSensor' and 'MultiVariableSensor'. Each class contains several different constructors that can be used depending upon how much information is available about the device being created. For instance, in the examples provided in Figures 3 and 4 the constructor is provided with a name for the device instance, the supported state variables (provided in a string array for multi-variable devices, as seen in Figure 4), and a description of the device itself. In this instance, all other required variables, such as a unique UUID are allocated to the new device by the EDK API. Figures 3 and 4 also show how a 'StateChangeListener' can be attached to individual master device instances, which flag whenever the value of any supported state variable is changed either by some attached hardware or due to a user request.

Note: When implementing master devices it is highly recommended that each instance is given a unique name value. It should always be ensured that device instances always have different uuid values, which can be generated randomly using the Java SDK UUID class.

Note: It is essential that all supported state variable names are declared when creating a new device instance. Several pre-formed device types are included within the EDK API, which often contain several convenience methods for performing device specific actions. To use these methods state variables must be declared using the specific names specified by the Javadoc information for the relevant class.

Note: If an application is intended to deploy multiple master device instances it is recommended to create each instance at this stage before continuing.

Note: To prevent processing errors individual state variables should never include the colon or tilde characters (i.e. : or ~) in their names or possible returnable values.

2.1.3 Step Three: Adding Communications

```
InetAddress environmentAddress =
        InetAddress.getByName(NETWORK_ADDRESS);
Communications communications =
        new Communications(environmentAddress, COMMUNICATIONS_PORT);
```

Fig. 5: Creating the communications system

An instance of the 'Communications' class must be created in order for master devices to be able to connect to an EDK network. Figure 5 shows how to do this using two of the variables from Figure 2 mentioned earlier to supply values for the network address and communications port variables. The EDK uses multicast communication to allow master devices to send updates to any running control points that have joined the same group. As a consequence, it is important to ensure that the value used for the 'NETWORK_ADDRESS' variable is a valid multicast address. It may also be necessary to open the value used for 'COMMUNICATIONS_PORT' on firewalls which may be blocking the sending or receiving of multicast communications packets on the network.

2.1.4 Step Four: Loading Device Controllers

```
File controllerJARFile = new File(CONTROLLER_DIRECTORY +
        File.separator + "DeviceController.jar");
Controller lightController = new Controller(
        controllerJARFile, CONTROLLER_PACKAGE, CONTROLLER_CLASS);
```

Fig. 6: Loading device control systems

Control system programs for individual smart devices are created independently to the EDK. Once implemented a device control system can be uploaded into a master device via the EDK API. Figure 6 shows how to do this. 'DeviceController.jar' is the filename of the controller being uploaded, from the designated 'CONTROLLER_DIRECTORY'. To integrate the controller program with the master device the EDK needs to know the package ('CONTROLLER_PACKAGE') and name of the main class ('CONTROLLER_CLASS'). As before, examples of these values are provided in Figure 2.

2.1.5 Step Five: Creating a Device Processing Hub

```
DeviceHub lightHub = new DeviceHub(communications);
lightHub.addDevice(light1, lightController);
lightHub.addDevice(dimmer1, dimmerController);
```

Fig. 7: Creating a hub and adding master devices

The next step is to create a hub to process devices. The 'communications' variable of the 'DeviceHub' constructor should be the same instance of the 'Communications' class created back in Step Three. Once the hub is initialised, each of the smart device instances created in Step Two need to be individually added. As these are master devices they also need to be associated with their respective controller programs, loaded during Step Four (Section 2.1.4). Figure 7, provides an example of how to implement this for the light devices used in the previous steps.

The purpose of the 'DeviceHub' class differs slightly depending upon whether it is used within an implementation for master devices or an EDK control point. For master devices, the hub liaises with the communication system created in Step Three, to have each of its stored devices access their associated controllers and perform a live update of their recorded state variables. Typically this would involve each device accessing the real world hardware to which it is connected by its control system. Once acquired the up-to-date state variable information is then passed back to the communications system, where it is transmitted to the EDK network and used synchronising any listening clone device representations. The role played by the hub when used with an EDK control point is explained in more detail in Section 2.3.

2.1.6 Step Six: Adding Web Services, (Optional)

```
Webserver webServer = new Webserver(WEBSERVICES_PORT);
lightHub.addWebServices(webServer);
```

Fig. 8: Enabling the EDK Web Services

Using the Web Services interface method with an EDK middleware implementation is optional, so this step can be ignored if desired. However, enabling the Web Services system only requires the code shown in Figure 8 to be added after the creation of the device hub. The variable 'WEBSERVICES_PORT' is the last of the declared constants create back in Step One (Section 2.1.1), which in the case of this example is listed in Figure 2.

Once added to an instance of the 'DeviceHub' class the EDK mechanism will auto-generate a Web Services control interface for each of the declared smart devices and add them to the internal HTTP Server. Currently two different acceptable commands are implemented for sensors, (i.e. about and get), while actuators have three, (i.e. about, get and set). The Web Services interface can be loaded using any standard Web Browser, including on most mobile devices, such as smart phones and tablet computers. The syntax for an EDK Web Service is naturally bespoke to each situation where it is used, but the basic URL structure is as follows;

http://<Computer IP Address>:<Web Server Port>/<Device Name>/<Command>

So based upon the 'BooleanLight' and 'DimmableLight' smart device examples used throughout this tutorial, some acceptable URLs would be;

http://127.0.0.1:8000/Light1/about
http://127.0.0.1:8000/Light1/get
http://127.0.0.1:8000/Light1/set?Power:1
http://127.0.0.1:8000/Dimmer1/get?Brightness
http://127.0.0.1:8000/Dimmer1/set?Brightness:75

'127.0.0.1'	The IP address of the computer running the master device representation being accessed, (likely to be different to localhost).
'8000'	The port number for the Web Server, as declared by the value of the 'WEBSERVICES_PORT' constant created back in Step One.
'Light1' **'Dimmer1'**	The names of the smart devices being accessed, as declared when their representations were created in Step Two.
'about'	A command to display general information about the specified device.
'get'	A command to return the name of each state variable supported by the specified device, along with its currently recorded value.
'set?Power:1'	A command to set state variable 'Power' to a value of '1'. **Note:** 0 = OFF, 1 = ON
'get?Brightness'	A command to return the current value of state variable 'Brightness'.
'set?Brightness:75'	A command to set state variable 'Brightness' to a value of '75'.

2.2 Creating a Control Point

```
InetAddress environmentAddress =
        InetAddress.getByName(NETWORK_ADDRESS);
Communications communications =
        new Communications(environmentAddress, COMMUNICATIONS_PORT);

ControlPoint controlPoint =
        new ControlPoint(communications, "ControlPoint1");
```

Fig. 9: Code for creating an EDK Control Point instance

An instance of the 'ControlPoint' class is required to allow external client programs to access devices via an EDK network. Figure 9 shows the API code required to perform this task. The String variable used in the 'ControlPoint' constructor is a bespoke name for the specific instance being created. If more than one control point is being used within a client program, (i.e. to access different EDK networks), this variable can be used to identify specific instances if required.

A control point effectively acts as a portal into the EDK middleware system, allowing users to search for groups or specific master devices on the associated network. The details of the network which is accessed by a control point are provided by its associated communications system, as described in Section 2.1.3.

In addition to providing details of an EDK network, the communications system supplied to a control point is also responsible for processing state update messages for the master devices. The control point itself automatically generates an internal 'DeviceHub' instance, which is subsequently used to create and store clone device instances, based off information received by the communications system from the EDK network. The control point accesses the stored clone devices and uses their information to provide returnable results for user searches, as described in the next section.

2.2.1 Searching for Devices

```
Device[] deviceList = controlPoint.searchForDevices();
```

Fig. 10: Searching for known master devices on an EDK network

To search for known master devices present on a EDK network, a 'ControlPoint' instance can use its 'searchForDevices' method, as shown in Figure 10. Instances of 'ControlPoint' will only become aware of master devices upon receiving an update packet from them. Therefore, upon initially starting, there may be a delay before all master devices present on a network are discovered, depending upon where they are in their individual update cycles when the control point joins the EDK multicast group. It is typically a good idea to enclose the search command in a 'for', 'do-while' or 'while' loop to keep the control point scanning the network until a non-empty array is returned or a desired device is found. Alternatively, this command could also be repeatedly called from within an isolated thread, which runs continuously in the background, allowing devices that start broadcasting after the control point's initial search to also be detected.

2.2.2 Searching for Specific Devices

```
// Search for all instances of a specific device type
Device[] deviceList = controlPoint.
        searchForDevicesByType(Actuator.BOOLEAN_LIGHT);

// Search for a single device with the specified name
Device device = controlPoint.searchForDeviceByName("Light1");

// Search for a single device with the specified UUID
UUID uuid =
        UUID.fromString("2deb19fc-5d41-4ac9-a0ea-4e86ff7a121e");
Device device = controlPoint.searchForDeviceByUUID(uuid);
```

Fig. 11: Three methods for finding specific devices

If details of the target device or devices are known ahead of time, a client program can alternatively use one of the more specific search methods of the 'ControlPoint' class. As shown in Figure 11, there are currently three such methods, each targeting different attributes of master devices. Firstly, the 'searchForDevicesByType' method can be used to filter the known list of master devices by their type, returning an array of any matching instances stored in the control point 'DeviceHub'. The topmost example in Figure 11 uses this method to search for all instances of a 'BooleanLight'. The 'Actuator' and 'Sensor' classes in the API both contain numerous other declared variables that can be used with this method each representing one of the pre-formed smart device types included in the EDK. Alternatively, to search for a device type not included within the standard API, programs can use the 'SINGLE_VARIABLE_DEVICE' and 'MULTI_VARIABLE_DEVICE' variables, (also found in the 'Actuator' and 'Sensor' classes), or a bespoke device name entered as a String.

The two remaining search methods, shown in Figure 11, are each designed to only return a single device, matching either a specified name or uuid criterion. If no matching device was found as a result of a completed search then a 'null' value is returned. If for some reason, two different master devices existed on an EDK network and both were called 'Light1', the 'searchForDeviceByName' method will only return the first instance it encounters when scanning the contents of the control point device hub, with any other potentially matching clone simply being ignored. This also applies for the 'searchForDeviceByUUID' method if both devices shared the same uuid value.

Note: When implementing master devices it is highly recommended that each instance is given a unique name value. It should always be ensured that device instances always have different uuid values, which can be generated randomly using the Java SDK UUID class.

2.3 Processing EDK Smart Devices

As mentioned earlier the instance of 'Device Hub' created by a control point is used to store clones created to locally represent networked master devices. It is the device hub itself that creates the clones, which is the automatic response whenever receiving information from the communications system that doesn't match any previously known representation. Any clone devices are stored locally on the same computer as the control point used to create them. Aside from not possessing controllers for hardware, they are identical to the master devices

that spawned them in every way. Even the unique attributes of the original master device are copied, including its name and uuid value. As such, by using the associated classes within the EDK API, the details and states of each clone can be read, and in the case of actuators manipulated, in the same manner as if connecting directly to any master device instance.

2.3.1 Reading from Smart Devices

```
String deviceState = device.getState();

String powerState = device.getState("Power");

int brightnessState =
        Integer.parseInt(device.getState("Brightness"));

int brightnessState =
        ((DimmableLight)device).getBrightness();
```

Fig. 12: Methods for reading the values of device state variables

The EDK API contains several different methods of reading the state of smart devices. A selection of the possible options is listed in Figure 12. Which method is best largely depends upon what information is required about the state of a device and how it is subsequently used.

From the Figure 12 examples, the topmost method is a general 'getState' command which is common to every EDK device. When called the 'getState' command will return a single string representation of the entire device, or more specifically its state variable values. Responses are always sent in pairs, with the name of the state variable and its current value. For instance, in the case of the 'Light1' 'BooleanLight' device used in the implementation examples earlier, a 'getState' request could result in either of the following String responses;

Power:0
Power:1

Where, 'Power' is the name of the only state variable included in a 'BooleanLight' object, while zero (off) and one (on) are the current values of that state, according to the response. The colon separating the two values has been provided to act as a key in a split command to allow easy separation of variable name and its value.

In the case of multi-variable devices, which contain more than one state variable, an additional tilde key ('~') is added to separate the individual attributes. So for the 'Dimmer1' 'DimmableLight' used in earlier examples, a 'getState' request could return;

Power:0~Brightness:100

Where, 'Power' is the first state variable and 'Brightness' is the second, with current recorded values of zero (off) and one hundred (percentage of maximum illumination) respectively.

Note: To prevent processing errors individual state variables should never include the colon or tilde characters (i.e. : or ~) in their names or possible returnable values.

Note: When implementing control programs for smart devices it is essential that the methods of the controller return responses to state action requests formatted correctly, as described in this section.

The three remaining methods in Figure 12, each show ways that the API can be used to return only the current value of a specific state variable, rather than the name/value pairs shown above. Generally this is achieved by specifying the name of the state variable whose value is required as a variable in the method. The second and third down examples in Figure 12 demonstrate this process for a 'BooleanLight' and 'DimmableLight' respectively. Typically, the returned state values are in a String format, but can easily be converted as is shown with the 'Brightness' variable example, where the value is converted into an integer once returned.

In many cases the String to integer conversion performed in the third example of Figure 12 can often be avoided as many of the EDK API devices include convenience methods, which return state variable values in their most appropriate format automatically. This is demonstrated by the bottommost method in Figure 12, which uses a 'getBrightness' method found in the 'DimmableLight' class which automatically returns the state value as an integer. All that is required to use these bespoke convenience methods is to cast the generic 'Device' object returned by a control point search into the appropriate device type class, as is shown in the example.

2.3.2 Writing to Smart Devices

```
for (Device device : deviceList) {
    if (device.getName().equalsIgnoreCase("Dimmer1") &&
            device.getType().equals("DimmableLight")) {
        if (device.getState("Power").equalsIgnoreCase("0")) {

            ((DimmableLight)device).turnOn();

        }

        else {

            ((DimmableLight)device).turnOff();

        }
    }
}
```

Fig. 13: An example of sending action state change requests to a master device.

In the example provided by Figure 13, an array returned by a control point search (as described in Section 2.2), is scanned for a specific device called 'Dimmer1', which is also a 'DimmableLight'. If found the value of the state variable 'Power' is requested from the device. If the value returned for 'Power' is zero (off) then a command is sent to turn the light on. For any other circumstance the command is to turn the light off.

Note: In the Figure 13 example, 'Actuator.DIMMABLE_LIGHT' could also have been used instead of the String value "DimmableLight", as was described in Section 2.2.2.

In the example the generic 'device' variable taken from the 'Device' array is cast into a 'DimmableLight' object. This allows the programmer to access the two convenience methods 'turnOff' and 'turnOn', which are actually contained within the Actuator class. The EDK API contains several models for intelligent devices which can be used in place of the more generic actuator and sensor classes. Some of these classes also contain further convenience methods specific to that device type. For example, the 'DimmableLight' class also contains a 'setBrightness' method to allow a specific value to be entered for a 'Brightness' state variable, controlling the amount of light emitted by the device.

3 Device Control Systems

Creating the control systems for individual smart devices is typically a bespoke process dependent upon the interface requirements of the hardware being used. Therefore, this guide won't go into specifics of how to implement the actual specifics of command programs. This section highlights the steps that should be taken to integrate an existing smart device control program with an EDK middleware implementation.

3.1.1 Implementing EDK Compatible Device Controllers

```
package devicecontroller;

public class Controller {

    public Controller() {

    }

    public String getState() {

    }

    public String setState(String state) {

    }
}
```

Fig. 14: A controller template for a single variable smart device

Recalling back to Section 2.1.1, two of the constants that needed to be declared when implementing a master device, (as shown in Figure 2) were 'CONTROLLER_CLASS' and 'CONTROLLER_PACKAGE'. The origins of the values used in the Figure 2 example, can be seen in Figure 14. The value that should be used for the 'CONTROLLER_CLASS' constant is the name of the main class of the control system program, which in the example is simply 'Controller'. Additionally, the value for the 'CONTROLLER_PACKAGE' is the name of the package containing the declared main class, in this case 'devicecontroller'.

When implementing a control program for smart devices for use with an EDK middleware system, Figure 14 shows the minimum classes required for integration. More specifically, the 'getState' and 'setState' methods are both essential and should be used to directly return or update the current state of the associated hardware respectively. If additional code is required to create a link with the associated hardware, such as using a third-party software package (e.g. RXTX or a manufacturer API), then all this code should be placed into a constructor within the main class, as shown by the 'Controller' constructor in Figure 14. If necessary, the constructor code should establish a connection with the smart device hardware and then maintain it as a global variable that can be accessed directly by the 'getState' and 'setState' methods. Alternatively, the constructor could be used to start an isolated thread, which uses locally declared variables, also accessible by the 'getState' and 'setState' methods, to handle state action requests. It is essential that no code necessary to directly establish a connection

with hardware is included in either the ‘getState’ or ‘setState’ method as doing so could lead to overall instability in the EDK middleware system.

Note: It is essential that methods used in device controllers return state values in as a single String using the expected EDK formats, as discussed in Section 2.3.1.

Note: It is recommended that the ‘setState’ method returns the same value as an action request to ‘getState’ once completing its operation.

3.1.2 Controllers for Multi-Variable Devices

```
package devicecontrollermultiple;

public class Controller {

    public Controller() {

    }

    public String getBrightness() {

    }

    public String getState() {

    }

    public String setBrightness(String value) {

    }

    public String setState(String state) {

    }
}
```

Fig. 15: A controller template for a dimmable light, (multi-variable smart device)

Figure 15 highlights how the controller code in Figure 14 can be extended to handle smart devices with multiple state variables. The code presented in Figure 14, could function with a multi-variable smart device, but it is often desirable to separate certain state variables to better structure program code, or for efficiency, etc. Figure 15 shows a template for the controller used with the ‘DimmableLight’ device ‘Dimmer1’ example mentioned throughout this guide. Notice how the ‘getState’ and ‘setState’ methods have been retained, (for handling the ‘Power’ state variable), although the ‘Brightness’ state variable has been separated and given its own handling methods, namely ‘getBrightness’ and ‘setBrightness’. To add additional ‘get’ and ‘set’ methods it is necessary to ensure that their suffix is named exactly the same as the state variable they are expected handle, (e.g. ‘getPower’, ‘getBrightness’, ‘setPower’, ‘setBrightness’, etc).

Note: Any ‘set’ methods should also only expect to receive a single variable containing the new state value to be processed, as shown in both Figures 14 and 15. No other variables must be added for the methods to be compatible.

Note: As with ‘setState’, it is recommended that any ‘set’ method returns the same value as an action request to ‘getState’ once completing its operation.

When accessing a loaded device controller an EDK implementation will first search through the methods of the declared controller class (i.e. ‘CONTROLLER_CLASS’) to see if it contains a bespoke match for the current state variable is needs to process. If no appropriate method matching the state variable can be found the system will then automatically default to either the ‘getState’ or ‘setState’ method, depending upon which action is being performed. Additionally, notice how in Figure 15 the package name has changed, reflecting that it is a different device controller program from the Figure 14 example. To be loaded correctly into the EDK and used with a master device, the Figure 15 example would need to specify ‘devicecontrollermultiple’ as the value for ‘CONTROLLER_PACKAGE’ during the first implementation step, (see Section 2.1.1 and Figure 2).