**Author : Matthijs Piek, Stefan van den Oord**

**Version : 0.8**

**Date : 2015-07-27**

**Status : Draft**

TABLE OF CONTENTS

1. Introduction 4

1.1 Purpose and scope 4

1.2 Intended audience 5

1.3 Definitions, acronyms & abbreviations 5

1.4 References 5

2. Static design 5

2.1 Life cycle management 5

2.1.1 External interfaces 6

2.1.1.1 DICommApplianceManager 6

2.1.1.2 DINetworkNode 7

2.1.1.3 DICommApplianceBuilder 8

2.1.1.4 DICommApplianceSettings 8

2.1.1.5 DILocalNetworkNodeStore 9

2.1.1.6 Delegate protocol DICommApplianceManagerDelegate 9

2.1.2 Design-in 9

2.2 Local And remote control 10

2.2.1 External interfaces 10

2.2.1.1 DICommAppliance 10

2.2.1.2 DIPort and its subclasses 11

2.2.2 Internals: Connecting to a connected appliance 15

2.2.2.1 Local and Remote Control 17

2.2.3 Design-in 17

2.3 Discovery 18

2.3.1 External interfaces 18

2.3.2 Static design 18

2.4 Tests 18

3. Roadmap 18

Appendix A. Reuse from Smart Air iOS 19

A.1 Terminology: Appliances instead of Devices 20

A.2 Project-Specific Appliances 20

A.3 Settings 20

A.4 Configuration of DICommApplianceManager 21

A.5 Move Generic Logic to DICommAppliance 21

A.6 Air Purifier Specific Scheduling Logic 21

A.7 DIConnection 22

A.8 Discovery 22

A.9 Making Remote Control Configurable 22

A.10 General 23

Document data and history 24

LIST OF FIGURES

Figure 1: DI Comm Client Context 4

Figure 2: Classes involved in lifecycle management and discovery. DICommApplianceManager is the external interface. 6

Figure 3: Example of design-in on the lifecycle management layer. The EXExample classes are vertical-specific. The other classes are re-used as-is. 9

Figure 4: The classes involved in the external interface of local and remote control 10

Figure 5: DIPort and its subclasses. Functionality of DIPort will never be called directly but only through its subclasses. When designing-in a vertical specific DIPort subclass needs to be created. 12

Figure 6: The class diagram for the strategies. 16

Figure 7: Example of design-in on local and remote control. The EXExample classes are vertical-specific. All other classes are re-used as-is. 17

Figure 8: Reference implementation currently in Smart Air iOS. Some small changes needed to come to fully re-usable specification listed in this document. 19

# Introduction

More and more Philips apps support creating a connection with a connected appliance as defined in [DIPRA]. To this end a protocol has been specified in [DICOMM]. The DI Comm Client allows mobile apps to communicate with a connected appliance implementing the DI Comm protocol. The following diagram visualizes how the DI Comm Client fits into an app.



Figure 1: DI Comm Client Context

For iOS, the component can be derived from source code developed in the iOS Smart Air app. An appendix describes how to achieve this.

## Purpose and scope

This document defines the target SW architecture of the DI Comm Client component, including a definition of its interfaces.

## Intended audience

This document is targeted at SW architects and engineers who wish to implement the DI Comm Client component in their system, as well as SW engineers and architects developing and maintaining the component.

## Definitions, acronyms & abbreviations

DI Digital Innovation

UI User Interface

CPP Connected Products Platform

Vertical Relating to a specific product or value proposition.

Connected Appliance A (Philips) appliance that has been fitted with hardware that allows it to communicate with other devices. This communication is typically done over Wi-Fi or Bluetooth.

DI Comm Appliance A connected appliance that communicates using the DI Comm protocol [DICOMM]

## References

[DIPRA] DI Platform Reference Architecture, Bas Bergevoet (C07S01)

[DICOMM] DI Communications Protocol, Johan Sunter (C07S02)

[EWS] Easy Wi-Fi Setup Interface Specification, Stefan van den Oord (C07S04)

# Static design

The DI Comm Client component consists of several distinct but interrelated functionalities. Their design is discussed in this section. Only the DICommApplianceManager, DICommAppliance and DIPort classes are part of the external interface.

## Life cycle management

The DICommApplianceManager manages the life cycle of a connected appliance within the app life cycle.

It persistently stores the appliances that are connected as well as loads them from persistent storage on start-up. As such it acts as the central authority on which appliances are managed by the app[[1]](#footnote-1). It also makes sure the DI Comm Client is kept in a consistent state when the app goes to the background and resumes, as well as when network connections fail and resume.

The DICommApplianceManager plays a central role in an app using connected appliances and is created soon after launch.



Figure 2: Classes involved in lifecycle management and discovery. DICommApplianceManager is the external interface.

### External interfaces

#### DICommApplianceManager

The external interface consists of the DICommApplianceManager class. Please note that DICommApplianceManager is a singleton for historical reasons. Using singletons this way is discouraged, but changing it here would have too much impact.

DICommApplianceManager has the following methods and properties:

+ (DICommApplianceManager \*)sharedInstance;

You call this class method to acquire a singleton instance of DICommApplianceManager. Creates the singleton instance if it doesn’t exist yet.

+ (DICommApplianceManager\*) createSharedInstanceWithBuilder:(DICommApplianceBuilder\*)builder settings:(DICommApplianceSettings\*)settings

Class method to configure the shared instance that is returned by the sharedInstance: method. This method can only be called when no shared instance was created yet. So calling this after calling sharedInstance: will fail, and calling this more than once will also fail. Passing NULL for any of the parameters will cause the default instance of the corresponding parameter to be used.

- (NSMutableDictionary \*)discoveredAppliances;

The values of the dictionary form a list of discovered DINetworkNodes. The keys for the dictionary are the unique identifiers.

- (DICommAppliance \*)currentAppliance;

This contains the currently active connected appliance. The DICommApplianceManager will try to maintain a subscription with this appliance.

- (void)setCurrentAppliance:(DICommAppliance \*)purifier;

Sets the currently active connected appliance.

- (void)addApplianceWithNode:(DINetworkNode \*)node;

Creates a DICommAppliance (or subclass) instance. Adds appliance to the list of appliances managed.

- (void)removeAppliance:(DICommAppliance \*)purifier;

Remove appliance from the list of appliances managed.

- (BOOL)isManagingApplianceWithNode:(DINetworkNode \*)node;

This method indicates whether the manager is already managing an appliance with this node.

(nonatomic, weak) NSObject <DICommApplianceManagerDelegate> \* delegate;

The delegate protocol is used for communicating changes to the list of managed appliances. Refer to section 2.1.1.6 for details.

(nonatomic, strong) NSMutableArray \*addedAppliances;

The list of connected appliances managed by the DICommApplianceManager class.

(nonatomic, strong) DICommApplianceSettings \*settings;

Object containing logic to store and retrieve settings related to this component from persistent storage. When this is left unset, the manager will create its own DICommApplianceSettings, which will use NSUserDefaults.

(nonatomic, strong) DICommApplianceBuilder \* applianceBuilder;

Object containing logic to build a DICommAppliance (or subclass) instance. This will be used whenever a new DICommAppliance needs to be built. When designing-in this component, a subclass can be set here so the DICommAppliance creates subclasses.

#### DINetworkNode

This class stores information about network-related aspects of connected appliances. It is a data class has the following properties.

|  |  |  |
| --- | --- | --- |
| DINetworkNode (iOS) / com.philips.cl.di.comm.NetworkNode (Android) | | |
| name | String | The name that the user gave to the appliance |
| modelName | String | The model name of the appliance |
| ipAddress | String | The IP address on which the appliance can be reached |
| homeSsid | String | The network name of the network in which the appliance is configured |
| cppId | String | The identifier by which the appliance is known at the Philips CPP back-end |
| bootId | Integer | A number that can be used to detect that the appliance has rebooted[[2]](#footnote-2) |
| allowRemoteControl | Boolean | Indicates whether the appliance may be controlled remotely. This property is **not persistent**. |

The NetworkNode may store additional information such as the geolocation of the appliance, if desired for a specific application.

#### DICommApplianceBuilder

This builder is used to create instance of DICommAppliance. Most projects will have a special-purpose subclass of DICommAppliance. In that case you also need a special-purpose subclass of DICommApplianceBuilder that can construct instances of the DICommAppliance subclass.

- (DICommAppliance\*)buildApplianceForNode:(DINetworkNode\*)node;

This method constructs an instance of DICommAppliance and configures it with the given node.

- (BOOL)canBuildApplianceForNode:(DINetworkNode\*)node;

Discovery may return multiple types of Philips connected appliances. Specific projects usually only support one (or a few) of those. The builder should implement this method to indicate whether it supports the given node (usually by checking the node’s model name). If YES, buildApplianceForNode: may be called. If NO, that method will return NULL.

#### DICommApplianceSettings

This class stores information that needs to persist between launches of the app. It has properties that store the list of CPP IDs of the appliances that the user has configured. Also, it stores the CPP ID of the one that the user has currently selected in the user interface:

@property (nonatomic, strong) NSArray \*myAppliances;

@property (nonatomic, strong) NSString \*selectedApplianceCppId;

Besides these properties, there is also a property that controls whether or not remote control is supported by the app. This controls whether the DICommApplianceManager will sign into CPP on startup. You can find more information about this in section 2.2.2.1.

@property (nonatomic, assign) BOOL remoteControlSupported;

This class has one public method:

- (void)registerDefaults;

This method is automatically called on initialization and can be used by subclasses to register the desired defaults with the OS defaults system. Subclasses must always call the implementation in DICommApplianceSettings if they override this method.

#### DILocalNetworkNodeStore

DILocalNetworkNodeStore persistently stores metadata about DI appliances that have been found on the network. This metadata takes the form of DINetworkNode instances (see 2.1.1.2). This store provides an interface that allows nodes to be stored and retrieved based on their CPP id or IP address. It should be considered internal to the DI Comm Client and is mainly used by DIServiceDiscovery and DICommApplianceManager.

This store allows for functions such as detecting that the IP address of an appliance has changed, detecting that its boot ID has changed, knowing what its home SSID is (so that you know whether to connect using CPP or directly).

DILocalNetworkNodeStore provides the following methods:

- (void)setLocalNetworkNode:(DINetworkNode \*)node forCppId:(NSString \*)cppId;  
- (DINetworkNode \*)localNetworkNodeForCppId:(NSString \*)cppId;  
- (DINetworkNode \*)localNetworkNodeForIpAddress:(NSString \*)address;  
- (void)removeLocalNetworkNodeForCppId:(NSString \*)string;

#### Delegate protocol DICommApplianceManagerDelegate

The delegate gets notified of changes to the list of managed appliances. This will probably be mainly used to dynamically manage the UI.

- (void)manager:(DICommApplianceManager \*)manager  
didAddNewAppliance:(DICommAppliance \*)appliance;

- (void)manager:(DICommApplianceManager \*)manager  
didAddExistingAppliance:(DICommAppliance \*)appliance;

- (void)manager:(DICommApplianceManager \*)manager  
didRemoveAppliance:(DICommAppliance \*)appliance;

### Design-in



Figure 3: Example of design-in on the lifecycle management layer. The EXExample classes are vertical-specific. The other classes are re-used as-is.

To design-in these classes, you need to subclass DICommApplianceBuilder to build the class for your vertical specific DICommAppliance subclass (see 2.2).

## Local And remote control

The component enables local and remote control of the connected appliance as explained in [DICOMM] and [DIPRA]. The component automatically selects between local and remote control by detecting whether it is on a local network where it found the connected appliance. In other cases it defaults to remote control. It is possible to configure the component to not use remote control.

### External interfaces



Figure 4: The classes involved in the external interface of local and remote control

The external interface consists of the DICommAppliance class and its ports, which are all instances of the DIPort subclass. These ports mirror the ports that are available in the [DICOMM] specification.

#### DICommAppliance

The most import properties are the following. They provide access to the various ports of the DICommAppliance. These ports provide the user of the component with the ability to read and write data to the ports of the connected appliance.

@property (nonatomic, strong, readonly) DIDevicePort \*devicePort;

@property (nonatomic, strong, readonly) DIWiFiPort \*wifiPort;

@property (nonatomic, strong, readonly) DIWiFiUiPort \*wifiuiPort;

@property (nonatomic, strong, readonly) DIFirmwarePort \*firmwarePort;

@property (nonatomic, strong, readonly) DISecurityPort \*securityPort;

@property (nonatomic, strong, readonly) DIPairingPort \*pairingPort;

@property (nonatomic, strong, readonly) DIPushNotificationsPort \*pushNotificationsPort;

@property (nonatomic, strong, readonly) DIScheduleListPort \*scheduleListPort;

The manager will set the node property. It contains the information that the DICommAppliance needs to reach the appliance.

@property (nonatomic, strong) DINetworkNode \*node;

These methods deal with subscriptions on all ports that support subscription. See explanation about subscription for the DIPort class.

- (void)subscribe;

- (void)unsubscribe;

- (void)stopResubscribe;

- (void)resubscribe;

Debugging feature to give a nice string when printing a DICommAppliance:

- (NSString \*)description;

The name of the DICommAppliance, which identifies the appliance to the user in the UI:

- (NSString \*)name;

These methods concern pairing and push notifications:

- (void)checkPairingWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;

- (void)pairWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;

- (void)removeAllRelationshipsWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;

- (void)checkPushNotificationsEnabled;

#### DIPort and its subclasses



Figure 5: DIPort and its subclasses. Functionality of DIPort will never be called directly but only through its subclasses. When designing-in a vertical specific DIPort subclass needs to be created.

DIPort will never be instantiated directly. All access to its code will go via its subclasses. DIPort provides all the basic mechanisms for:

1. Connecting to a connected appliance via the [DICOMM] protocol.
2. Maintaining subscriptions
3. Handling the UI/UX performance flow explained in [DIPRA].
4. Caching the data received from the connected appliance.
5. Helper functions for converting the parsed JSON data to different formats.

The port-specific subclasses deal with making the parsed JSON data accessible to the rest of the app and as such they form the external interface. They will not be discussed in detail here, as they are mostly straightforward adaptations from [DICOMM]. Two exceptions are the DIPairingPort and the DIPushNotificationPort, which handle the entire flow for enabling pairing and push notifications, including connections to CPP.

##### Reading data

There are two ways to get information from a DIPort subclass. The first method is a “pull”-method: you can get information directly through the properties of the DIPort subclass. When you try to read a property for which the port does not have a value in the cache this will automatically trigger a reload of the port over the network. This is called “GETPROPS” in terms of [DICOMM]. You can force such a reload by calling the reloadProperties method.

The second method is a “push”-method: you can register instances of a DIPropertyUpdateHandler and DIErrorHandler. These will be called when the port’s properties get updated or when the port encounters an error respectively.

##### Subscriptions

Some ports allow subscriptions to updates on the appliance. The caller of the API can have the port subscribe to updates by calling the subscribe and resubscribe methods on the DIPort subclass or on DICommAppliance. In the latter case DICommAppliance will subscribe to updates on all ports that support subscription. The updates received through subscription will trigger calls to the registered DIPropertyUpdateHandlers.

As specified in [DICOMM] “SUBSCRIBE” requests will time-out after a certain time. DIPort will automatically try to refresh the subscription. Both the subscribe and resubscribe methods will start refreshing the subscription, but the subscribe method will always try to send a “SUBSCRIBE” request; the resubscribe method will only send one when the current subscription has timed-out. Similarly, the unsubscribe method will send “UNSUBSCRIBE” requests, but the stopResubscribe method will just stop refreshing the subscription. It’s normally preferred to use resubscribe and stopResubscribe instead of subscribe and unsubscribe in order to reduce the amount of network calls needed.

##### Sending Data

Sending data is achieved by setting the properties on the DIPort subclasses. This will trigger a “PUTPROPS” request in [DICOMM] terminology. To increase the perceived performance (see[DIPRA]) DIPort will act like the change succeeded when the method is called and update all DIPropertyUpdateHandlers. The “PUTPROPS” will be sent as soon as possible after. If it failed the requested change will be undone in DIPort and the DIPropertyUpdateHandlers and DIErrorHandlers will be notified. As long as DIPort is still trying to apply changes the applyingChanges method returns true. The applyingChanges method can be used to display a spinner or other activity indicator to the user.

##### Performance considerations

As seen in [DIPRA] it is important that the performance of the connection is perceived as being high. DIPort implements all the functionality necessary for the mechanism listed in [DIPRA] in a way that is mostly invisible to the caller of the API. Performance of sending data is also covered in 2.2.1.2.3.

An important performance consideration is that it is not allowed to send parallel requests to a DI Comm connected appliance. DIPort makes sure that doesn’t happen so that the callers of the API do not have to worry about this, but this does have a slight impact on the timing on the external interface.

DIPort tries to minimize the amount of network calls needed. When multiple changes are requested while DIPort is already performing a request for example, it will combine them in one call. Similarly, if reloadProperties is called but the current state of the connected appliance’s port is received as part of a “SUBSCRIBE” or “PUTPROPS” request it will treat the reloadProperties as successful without sending a “GETPROPS” request. If the same methods (with the same parameters) are called multiple times, DIPort will not necessarily perform the associated network calls several times.

The request types are prioritized in this order (high priority to low):

1. Changes (“PUTPROPS”) are prioritized highest because the user usually initiates changes. The time it takes for changes to reflect in reality will be the primary measurement for performance.
2. Subscribe calls are second because they also return the current state of the port, obsoleting possible reloadProperties calls.
3. Unsubscribe calls are third because the user will not care about their performance.
4. ReloadProperties (“GETPROPS”) are prioritized last because the other calls will also return the current state, making the “GETPROPS” network call superfluous.

##### DIPairingPort

DIPairingPort has more responsibilities than one might expect based on the fact that it is a DIPort subclass. One part of its responsibility is fulfilling the contract specified in [DICOMM], meaning that it allows for initiating a pairing procedure by telling the appliance that it wants to pair with that appliance. The second part of its responsibility is completing the paring procedure by communicating with the Philips back-end. This is handled transparently by DIPairingPort, meaning that a request to pair will handle both the local communication with the appliance, as well as the communication with the Philips back-end.

DIPairingPort has one public method.

- (BOOL)paired;

This method returns whether or not the appliance is paired with the app. This value is cached, so getting it will return the last-known value.

Besides that public method, DIPairingPort has the following methods. These methods are for internal use in the DI Comm client and should not be exposed on the public API. The public methods needed for this functionality are defined on the DICommAppliance interface.

- (void)pairWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;  
- (void)removePairWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;  
- (void)checkPairingWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;  
- (void)removeDataAccessWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;

For all the methods that have a success and failure handler, the implementation is so that *both*these handlers are called *and* the DIPropertyUpdateHandler/DIPropertyErrorHandler.

##### DIPushNotificationsPort

DIPushNotificationsPort is a bit of an odd duck in the sense that it is not a real port: there is no such thing in [DICOMM]. It was designed as a port, because the way it is used by an app is the same as other ports: get/set properties and getting notifications about them through DIPropertyUpdateHandler and DIPropertyErrorHandler.

DIPushNotificationsPort has one property:

@property (nonatomic) BOOL pushNotificationsEnabled;

This property indicates whether or not push notifications are enabled. This value is cached, so getting it will return the last-known value.

The public methods of this class are the following. Their function should be self-explanatory. As a result of calling these methods, the registered DIPropertyUpdateHandlers or DIPropertyErrorHandlers will be called.

- (void)enablePushNotifications;  
- (void)disablePushNotifications;  
- (void)checkPushNotificationsEnabled;

Besides these three methods, the DIPushNotificationsPort has the following four methods. These are intended for internal use by the DI Comm Client and must be used outside the component.

- (void)enablePushNotificationsWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;  
- (void)disablePushNotificationsWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;  
- (void)checkPushNotificationsEnabledWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;  
- (void)removePushNotificationsWithSuccess:(DISuccessHandler)success failure:(DIFailureHandler)failure;

### Internals: Connecting to a connected appliance

This section explains the internals of the local & remote control functionality. In principle this should be totally re-usable, so this is for background information only. Because this is not part of the external interface it will be explained in lesser detail.



Figure 6: The class diagram for the strategies.

DIPort handles all connections via subclasses of DICommApplianceCommunicationStrategy. This follows both the Strategy design and the Composite design pattern. DICommApplianceCommunicationStrategy will never be used directly. These strategy subclasses exist:

1. Local strategy: connects over local Wi-Fi to the connected appliance. It is considered to be available when the currently connected SSID is equal to the homeSsid property of this DICommAppliance’s node.
2. Remote strategy: connects through CPP to the connected appliance. It is considered to be available when CPP Client has signed in and we are connected to the Internet.
3. Null strategy handles the unconnected case. It gracefully fails all the requests. It is never considered to be available.
4. Communication marshal: A strategy that dynamically selects between the other strategies. It will choose the highest priority available strategy. If none is available it will choose the lower priority strategy as a fallback. The priority order of the strategies is: Local, remote, null.

Almost all ports share the strategy instances and they all use the same communication marshal strategy instance. This instance contains a local, remote and a null strategy instance.

NOTE: The Smart Air reference implementation contains some dead code in the strategies. The property update handlers and error handlers in the strategies are not used and can be removed. (The property update handlers and error handlers in the ports are used so should be kept.)

#### Local and Remote Control

The DI Comm Client supports both local and remote connections. For local connections, the app and the appliance need to be on the same network. Remote connections don’t have that requirement, as they work through the Philips CPP back-end.

For some products, supporting remote control is a requirement, whereas for some other products it should not be supported. To control this aspect, you can use the remoteControlSupported property of DICommApplianceSettings. The correct value of this property must be set before DICommApplianceManager is used for the first time. DICommApplianceManager uses it to decide whether or not it signs into the Philips CPP backend. Also, the property allowRemoteControl in all newly created DINetworkNodes will be set to the same value, and for existing nodes it will be set to false if remoteControlSupported is false.

One of the considerations for storing and passing this information through DINetworkNode, is that it allows for future apps that operate multiple connected appliances, some of which do support remote control and some don’t.

### Design-in



Figure 7: Example of design-in on local and remote control. The EXExample classes are vertical-specific. All other classes are re-used as-is.

To design-in these classes, you need to subclass DICommAppliance to add your vertical specific functionality. It will probably need to add a vertical-specific subclass of DIPort to its list of ports. The DIPort subclass needs to implement methods to translate the values available from the DIPort methods to semantic values the remainder of the app can use and vice versa. These need to correspond with the vertical specific specification of the DICOMM protocol.

As an example, the Smart Air app contains the subclass APAirpurifier that adds an APAirPort instance to the list of ports. APAirPort translates the parsed JSON values to integers, enums and strings and makes them available as properties. When setting these integers, enums and strings the APAirPort calls the appropriate setProperty methods on the DIPort superclass that initiates network calls towards the connected appliance.

## Discovery

The component allows you to discover DI Comm connected appliances on your network as defined in section 3.1.3 of [DIPRA].

### External interfaces

The external interface to discovery is DICommApplianceManager, which is described in 2.1.

### Static design

DICommApplianceManager uses the DIServiceDiscovery class, which implements the SSDP discovery protocol. This listens for DI Comm implementations on the currently connected Wi-Fi network. The class performs discovery every 15 seconds.

## Tests

All functionality contains extensive unit and integration tests. It is important to maintain and run those, so that all the verticals making use of the horizontal component remain compatible with the component.

# Roadmap

1. Reuse from Smart Air iOS



Figure 8: Reference implementation currently in Smart Air iOS. Some small changes needed to come to fully re-usable specification listed in this document.

In Smart Air iOS an implementation of the DI Comm Client was made that is largely reusable. There are only a few tasks to be done in order to reuse it in other projects. This will of course cost some time, but it will save a huge development effort, because this code is non-trivial and the quality is well tested in Smart Air.

* 1. Terminology: Appliances instead of Devices

In order to be consistent with the system architecture, what is currently called DIConnectedDevice in Smart Air, should be renamed to DICommAppliance. This also goes for all classes that have this as part of their name, for example DIConnectedDeviceLocalCommunicationStrategy should be renamed to DICommApplianceLocalCommunicationStrategy.

* 1. Project-Specific Appliances

In Smart Air, the class APAirPurifierManager is responsible to create instances of the APAirPurifier class. The APAirPurifier class is the subclass of DICommAppliance that adds air purifier specific logic, such as the APAirPort. For this to be reusable, there needs to be a builder class that can create the correct DICommAppliance subclass instances, for example MCMultiCooker.

Steps to take:

1. APAirPurifierManager contains code to create instances of APAirPurifier. Refactor this code by moving it to a new class APAirPurifierBuilder. Call this new method buildApplianceForNode:.
2. Add method canBuildApplianceForNode: and replace modelName checking in APAirPurifierManager (nodeIsAirPurifier:) by a call to this.
3. Extract the superclass DICommApplianceBuilder from APAirPurifierBuilder.
4. Move the geo-location functionality to APAirPurifierBuilder.
5. Rename APAirPurifierManager to DICommApplianceManager.
6. Make interface consistent with external interface specification earlier in document.
   1. Settings

DICommApplianceManager (previously called APAirPurifierManager) uses an instance of APSettings to interact with NSUserDefaults. APSettings is specific to Smart Air, so the following steps need to be taken to make it generic:

1. Extract a superclass of APSettings and call it DICommApplianceSettings.
2. Move the storage property and init to DICommApplianceSettings, and add an empty registerDefaults to DICommApplianceSettings.
3. Move myAirPurifiers and selectedAirPurifier to the superclass and rename them to myAppliances and selectedApplianceCppId.
4. Declare registerDefaults in the header file of DICommApplianceSettings so that subclasses can override it.
   1. Configuration of DICommApplianceManager

The DICommApplicanceManager is now reusable, but it needs a reference to a DICommApplianceBuilder and a DICommApplianceSettings instance in order to do its job.

1. Add a parameter for the DICommApplianceBuilder to the initWithSettings:localNetworkNodeStore: method
2. Have the default init method create a default DICommApplianceBuilder instance for this parameter
3. Create a static method createSharedInstanceWithBuilder:settings: in DICommApplianceManager. This method may only be called once; it has to NSAssert that the sharedInstance is not created yet. It should create the shared instance and initialize it with the provided builder and settings instances.

The result is that you now have a reusable, generic DICommApplianceManager that can be configured to use project-specific subclasses of DICommApplianceBuilder (e.g. APAirPurifierBuilder) and DICommApplianceSettings (e.g. APSettings). The initial setup of the DICommApplianceManager (i.e. the call to createSharedInstanceWithBuilder:settings:) has to be done in the application delegate.

* 1. Move Generic Logic to DICommAppliance

Now that DICommApplianceManager can be reused, the next step is to move a bit of code from APAirPurifier into DICommAppliance, because it is already generic.

1. Move the cppId property to DICommAppliance, as well as the isEqual and description methods
2. Move the methods related to pairing to DICommAppliance
3. Move the method removeAllRelationshipsWith[…] to DICommAppliance
4. Move checkPushNotifications to DICommAppliance
   1. Air Purifier Specific Scheduling Logic

Scheduling is mostly already reusable; only a few adjustments are needed.

1. APPortConstants contains constants that are specific to Smart Air (e.g. fan speed etc.), and constants that are reusable (e.g. related to pairing and scheduling). Split this file in two and place all the reusable (non Smart Air specific) constants in a new file DIPortConstants.

The following steps apply to DIScheduleEntryPort:

1. Rename the property command and methods command and setCommand: to airPortCommand and setAirPortCommand: and move them to a new category DIScheduleEntryPort+AirPurifier. Make sure that the code in DIScheduleEntryPort+TableViewable uses the new airPortCommand property. Everything should still be working at this point.
2. Rename the property realCommand to command (also rename the getter and setter)
   1. DIConnection

DIConnection contains a reference to the Smart Air specific air port. The rule here is that all standard ports except the device and wifiui port have product ID 0, and other ports have port number 1. In other words, the implementation should be changed so that it returns 0 for a given list of ports, and 1 in all other cases.

* 1. Discovery

DIServiceDiscovery is reusable except for one little detail: it filters the SSDP replies using a string that contains “AirPurifier/”. To change this, introduce a string property serverFilter on DIServiceDiscovery. In deviceReplied:, check whether this property is set, and if yes, filter for the contents of that substring.

Now the filter has to be configured properly. Normally we would do this by making DIServiceDiscovery a parameter to DICommApplianceManager instead of a singleton, but that is probably too big a change. Instead, do it in the application delegate. Right before the call to createSharedInstanceWithBuilder:settings: in the application delegate, obtain the sharedInstance of DIServiceDiscovery and set the filter to the value it already had before this refactoring.

* 1. Making Remote Control Configurable

In Smart Air, remote control was a requirement. In some projects, it should not be used, so this must be made configurable:

1. Add the property remoteControlSupported to DICommApplianceSettings.
2. Make the code in DICommApplianceManager that performs CPP operations dependent on that setting: if it is false, don’t do the CPP operations.
3. Add the property allowRemoteControl to DINetworkNode.
4. Change the code that creates DINetworkNode objects so that the allowRemoteControl property initially has the same value as remoteControlSupported. Also, modify the code that restores DINetworkNode objects from persistent storage so that it sets the allowRemoteControl property to the same value as remoteControlSupported.
5. In DICommApplianceRemoteCommunicationStrategy, modify the isAvailable method so that it returns false if the node’s allowRemoteControl property is false. This check should be done before the checks that are already there in the isAvailable method.
   1. General

Some of the existing code in Smart Air is not compliant with above reference architecture. When extracting the DI Comm component from Smart Air, relevant code should be modified so that it is compliant with this reference architecture. Just to give one example: the four non-public methods of DIPushNotificationsPort should be removed from DIPushNotificationsPort.h and moved into a separate header file DIPushNotificationsPort+Protected.h.

Document data and history

|  |  |  |  |
| --- | --- | --- | --- |
| **Document data** | | | |
| **Project name** | Connected Digital Propositions | | |
| **Document name** | DI Comm Client – App Reference Design iOS | | |
| **Document id** | C07S02 | **Date** | 2015-07-27 |
| **Version** | 0.8 | **Status** | Draft |

|  |  |  |  |
| --- | --- | --- | --- |
| **Document history** | | | |
| **Version** | **Date** | **Author** | **Reason** |
| 0.1 | 2014-10-16 | Matthijs Piek | Very early rough draft in order to help multi-cooker project along |
| 0.2 | 2014-10-20 | Matthijs Piek, Stefan van den Oord | Added method to configure DIConnectedApplianceManager shared instance; added contents for appendix about reuse from Smart Air |
| 0.3 | 2014-12-11 | Stefan van den Oord | Added documentation regarding controlling whether remote control is supported by an app. |
| 0.4 | 2015-01-07 | Jeroen Mols | Added instructions to reuse AirPurifier Android DIComm architecture. |
| 0.5 | 2015-01-14 | Jeroen Mols | Added Parcelable interface for NetworkNode |
| 0.6 | 2015-01-14 | Jeroen Mols | Split Reference architecture into iOS and Android version |
| 0.7 | 2015-02-13 | Stefan van den Oord | Fixed inconsistencies in terminology (‘device’ was used instead of ‘appliance’). Changed some terminology:   * DIConnectedAppliance\* -> DICommAppliance\* |
| 0.8 | 2015-07-27 | Stefan van den Oord | Rebranded DI to CDP |

**Open issues.**

|  |  |
| --- | --- |
| **Date** | **Subject** |
|  |  |
|  |  |

1. An appliance is considered to be managed when the user has added it to the user interface of the app. This is usually done through Easy Wi-Fi Setup [EWS] or by selecting it from a list of discovered appliances. [↑](#footnote-ref-1)
2. Please refer to [DIPRA] for more information. [↑](#footnote-ref-2)