HomePort Documentation Technical Report

Thibaut Le Guilly, Anders P. Ravn, Arne Skou

0.1 Problem Domain

As already mentioned, HomePort aims at providing a common interface to heterogeneous home automation networks. The environment of the system is thus a Smart Home.

A Smart Home contains a number of devices providing services that are used to create an intelligent environment in which the house reacts to the inhabitants needs, automating tasks or augmenting comfort. Examples of such automation are intelligent lights, that turn on when a person enters a room, intelligent heating, that adjust depending the temperature and humidity, or electric vehicles, that charge when the price of the electricity is low. An illustration of a smart house installation is shown in Figure 1. An example of a device in this illustration is a temperature and humidity sensor. This device then provide two services, one to read the temperature of a room, another to read the humidity.

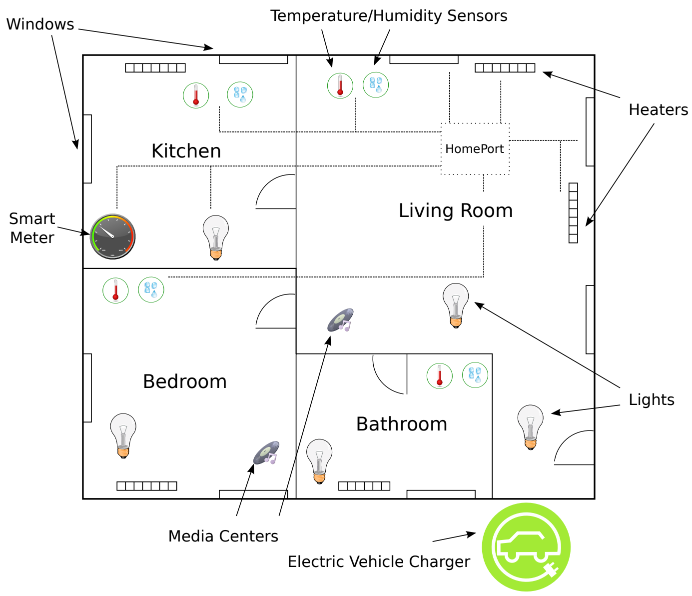


Figure 1: Example of a Smart House

The elements of the intelligent system belong to three categories. The first category is sensors; they are elements capable of observing the environment, reporting on the values of environment variables and detecting their changes. The temperature and humidity services, or the services provided by a smart meter are examples of such elements.

The second category is actuators; they control some part of the environment through devices they act upon such as heaters, lights or appliances. They can be set to different states (or set points) and can report their current states. Examples of such elements are heaters, EV charger or lights.

Finally, the third category is software clients, which access the values provided by the sensors and act upon the actuators. A client is an entity accessing the services available in the environment, either by reading sensors values or setting actuators. An important type of client is controllers that coordinate actuators with values reported by sensors, creating the intelligence in the system.

In order to coordinate actuators with sensors, the controllers need not only to get the values from the sensors and the possible states of the actuators, but also to know the configuration of the house. The configuration is information about the devices and the services they provide. The information may contain the type of service provided by a device, the granularity of a temperature sensor, or the location of a device.

For the coordination to take place, a controller also needs to be able to communicate with the devices. However, there are a tremendous number of communication technologies available for Smart Homes. Each of them have pros and cons, some are implemented using wireless technologies, such as ZigBee or Z-Wave, providing flexibility, others using wired technologies, such as PLC, providing reliability. By using different technologies in the same environment, one has a broader choice and may choose different technologies for different devices. However, this brings the problem that the controller needs to communicate with devices using different protocols. HomePort addresses this by providing a common interface to devices and the configuration of the environment. To communicate with subnetworks, HomePort uses adapters, that translate a client request to a format that the requested device can understand. There is therefore one adapter per subnetwork that associates the devices belonging to this subnetwork with the configuration. An adapter is identified by the name of the network it handles.

Figure 2 shows a class diagram representing the elements of the environement and their relations, and Figure 3 shows their behaviors. As depicted on the class diagram, HomePort has a configuration that is associated with adapters. An adapter has an identifier that is unique among the adapters of a configuration. It owns devices, and each device provides at least one service. A device has an identifier that is unique in the system, so that client do not have to know about adapters to find a device in the system. A services has a state, representing its current state, and an identifier that is unique among the services provided by the device they belong to. A service is thus uniquely identified by its identifier, and the identifier of the device it belongs to. Services are specialized in two categories. Actuators, whose state can be set, and sensors whose state one can only get. Looking at the behavior of the objects, we have the device that can be attached or detached to an adapter. The adapter can then register the device services in the configuration, enabling a client to get the state of a service or set it, if the service is an actuator. Finally, the configuration can be obtained or updated by clients.

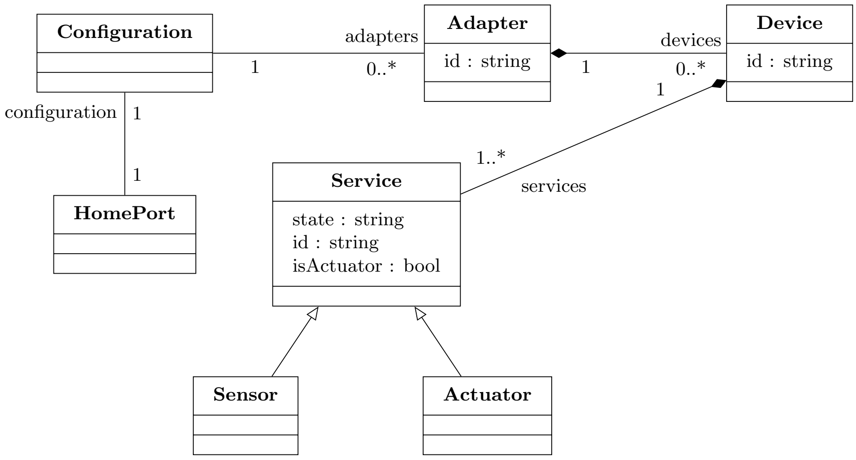


Figure 2: Class Diagram

0.2 Actors

After analyzing the problem domain, we move on to identify the actors of the system.

Clients are the users of the system. They access the services provided by the environment by reading sensor values and setting actuator set points.

Controllers are specialized clients, that send control commands to actuators based on the values returned by sensors.

Configurator is responsible for attaching or detaching devices to adapters and thus indirectly updating the configuration.

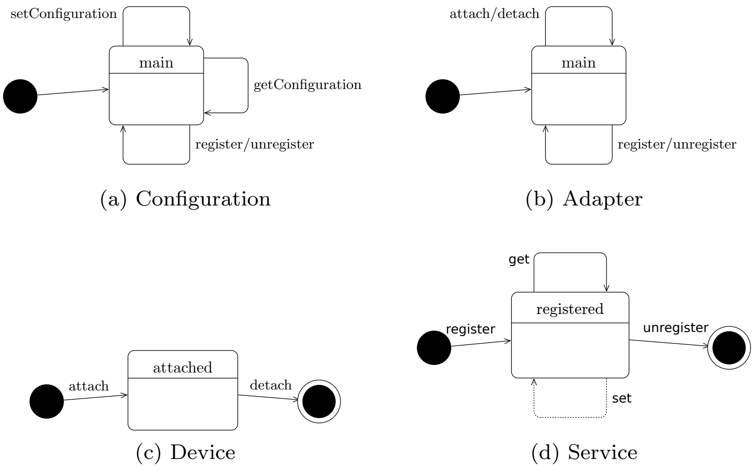
Now that the actors are identified, the functionalities of the system can be defined.

1 Functional Requirements

The functional requirements of the system are specified through a use case driven modeling phase. In this phase, each use case is described with short natural language paragraphs in a schematic way. First the purpose is outlined, then the actors. The pre- and post-conditions of the functions of the use case are then formulated, as well as one or more triggering event. The standard and alternative processes are then detailed by the execution steps of the use case, where each step corresponds to an action of the system or an actor. Finally, interactions between the system and actors are depicted by sequence diagrams, using the triggering events. In our running case, three use cases are presented.

1.1 UC 1 - Operate Services

Purpose: The core functionality of the system is to perform operations on services. Two operations can be performed: getting the state of a service or



(a) Configuration (b) Adapter (c) Device (d) Service

Figure 3: Statechart of the elements

setting it.

Involved Actors: Clients.

Precondition: The service to operate is registered and is capable of performing the desired operation.

Trigger: The system receives a request from a client.

Postcondition: The state of the service is updated according to the performed operation and the client receives the updated state of the service.

Standard Process

1. The client requests the system to perform an operation on a service.
2. The system looks-up the requested service.
3. The system performs the desired operation on the service.
4. The system returns the updated state of the service to the client.

Exceptional Processes

- In step 2: the requested service is not registered in the system.
  - The system returns an error message to the client, indicating that the service is not registered.
- In step 3: invalid request if the requested service is not capable of performing the desired operation.
  - The system returns an error message to the client, indicating that the requested operation is not applicable on the requested service.
- In step 3: the device providing the requested service is not available.
  - The system returns an error message to the client, informing it that the device is currently unavailable.

The sequence diagram is shown in Figure 4.

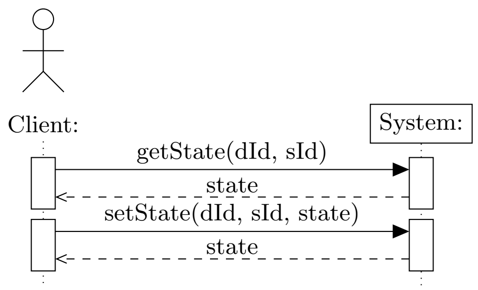


Figure 4: UC 1 Sequence Diagram

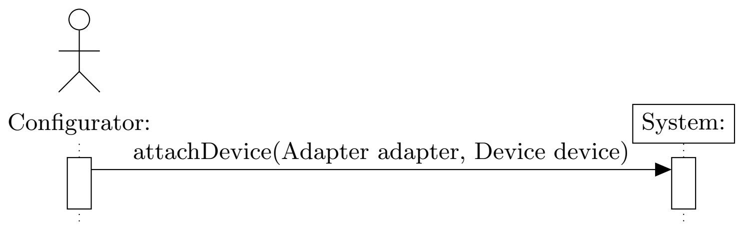


Figure 5: UC 2 Sequence Diagram

1.2 UC 2 Attaching a Device

Purpose: Handles when a new device is connected, “attached” to the system.

Primary Actors: Configurator.

Precondition: None.

Trigger: The configurator installs a new device.

Postcondition: The device and its services are added to the configuration through an adapter.

Standard Process:

1. The configurator installs a new device.
2. The system receives the device and the adapter it belongs to.
3. The system assigns the device a new identifier.
4. The system attaches the device to its adapter.
5. The system assigns each service an identifier.
6. The system adds the device to the adapter.

Exceptional Process: None.

The sequence diagram is shown in Figure 4.

1.3 UC 3 Access Configuration

Purpose: A client, or controller, needs to discover the services available. To do so, it accesses the configuration.

Primary Actors: Client, Configurator.

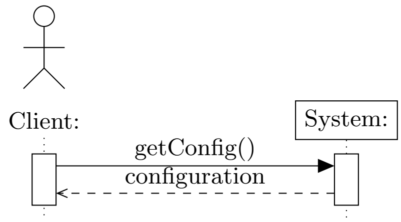


Figure 6: UC 3 - Sequence Diagram

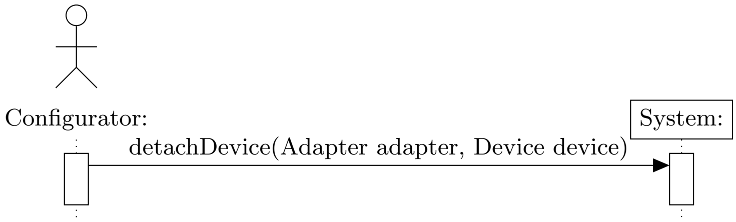


Figure 7: UC 4 - Sequence Diagram

Precondition: None.
Trigger: A configuration request.
Postcondition: The configuration is returned to the requestor.
Standard Process:

1. The client requests to know the configuration.
2. The system returns the configuration to the client.

Exceptional Process: None.
The sequence diagram is shown in Figure 6.

1.4 UC 4 Detaching a Device

Brief Description: Handles when a device is disconnected, “detached” from the system.
Primary Actors: Configurator.
Precondition: None.
Trigger: The configurator removes a new device in the environment.
Postcondition: The services provided by the device are removed from the configuration.
Standard Process:

1. The configurator detaches a device from a network handled by an adapter.
2. The system received the device to detach and the adapter it belongs to.
3. The device is removed from its adapter.

The sequence diagram is shown in Figure 7.

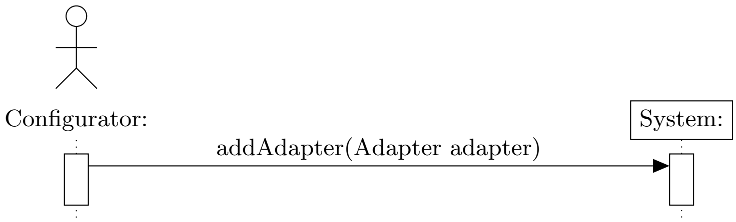


Figure 8: UC 5 - Sequence Diagram

1.5 UC 5 Add an Adapter

Brief Description: Handles when an adapter is added to the system.

Primary Actors: Configurator.

Precondition: None.

Trigger: The configurator adds a new adapter to the system.

Postcondition: The adapter is added to the list of adapter in the configuration

Standard Process:

1. The configurator gives the system an adapter to add.

2. The adapter is added to the configuration.

The sequence diagram is shown in Figure 9.

1.6 UC 6 Remove an Adapter

Brief Description: Handles when an adapter is removed from the system.

Primary Actors: Configurator.

Precondition: None.

Trigger: The configurator removes an adapter from the system.

Postcondition: The adapter is removed from the list of adapter in the configuration

Standard Process:

1. The configurator gives the system an adapter to remove.

2. The adapter is removed from the configuration.

The sequence diagram is shown in Figure 9.

2 Functionality Specification

After defining the functional requirements of the system, we specify the functionalities in details, by defining the content of the functions. We make the following assumptions, based on the class diagram shown in Figure 2:

1. There exists an object $homePort$ defined as $homePort : HomePort$.

2. The object $homePort$ access the configuration via an association $configuration$.

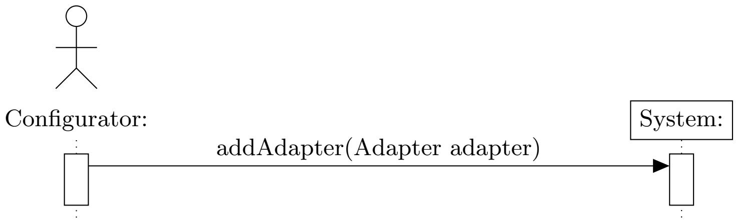


Figure 9: UC 5 - Sequence Diagram

3. The configuration is associated with a set of *Adapter* objects, $configuration.adapters$. It is realized by the attribute $adapters : set(Adapter)$.

4. An adapter owns a set of devices, denoted by $adapter.devices$. It is realized by the attribute $devices : set(Device)$.

5. A device owns a set of services, denoted by $device.services$. It is realized by the attribute $services : set(Service)$.

6. A service provides an attribute $isActuator$ which is used to determine if the service is an actuator or a sensor.

Given these assumptions, we start by specifying the functionalities of UC1.

UC1 incorporates two functions, getting the state of a service and setting it. To use one of these functions, a client provides the identifiers of the adapter and the device the service belongs to, as well as the identifier of the service itself. To access the service, the system thus needs to first find the correct adapter, device and service. Thus the first step is to ensure that the information provided by the client is valid, meaning that the identifier correspond to existing objects in the system. Once the system has located the correct service, it either sets the return value to the state of the service, if the client requests to get the service, or modify it before, if the client requests to set it. The detailed functionalities are specified as follows:

Use Cases	**UC1 - Operate Services**
Class	*HomePort*
Method	getState(string dId, string sId; string state) **pre:** ∃ Adapter a ∈ configuration.adapters • a.devices.find(dId) ≠ null ∃ Adapter a ∈ configuration.adapters, Device d ∈ a.devices • d.services.find(sId) ≠ null **post:** state' = d.services.find(sId).state
Method	setState(Id sId, State stateIn; State stateOut) **pre:** ∃ Adapter a ∈ configuration.adapters • a.devices.find(dId) ≠ null ∃ Adapter a ∈ configuration.adapters, Device d ∈ a.devices • d.services.find(sId) ≠ null d.find(sId).isActuator = true **post:** d.services.find(sId).state' = stateIn stateOut = d.services.find(sId).state
Invariants	configuration ≠ null

UC2 provides only one functionality, that is attaching a device to an adapter. The system is provided with the device to attach and the adapter it should be attached to. The system first needs to ensure the provided adapter exists in the configuration. As mentioned in the requirements, a device has an identifier that is unique in the system. The system thus needs to generate an identifier for the new device, that is the assigned to the device. In a similar manner, the device generates a unique identifier for each of its services. The functionalities are detailed:

Use Cases	**UC2 - Attach a Device**
Class	*HomePort*
Method	attachDevice(Adapter adapter, Device device)
	pre: configuration.adapters.find(adapter.id) $\neq$ null
	post: adapter.devices.add(device)
	$\forall$ Adapter a $\in$ configuration.adapters, Device d $\in$ a $\neq$ device $\bullet$ device.id $\neq$ d.id
Invariants	configuration $\neq$ null

UC3 also provides a single functionality, that is returning the configuration to a client. The details are provided:

Use Cases	**UC3 - Operate Configuration**
Class	*HomePort*
Method	getConfig(;Configuration config)
	pre: true
	post: config' = configuration
Invariants	configuration $\neq$ null

UC4 allows a configurator to detach a device from the system. The system receives the device to detach, the adapter it belongs to, checks that they are in the configuration and removes the device from the adapter.

Use Cases	**UC4 - Detach a Device**
Class	*HomePort*
Method	DetachDevice(Adapter adapter, Device device)
	pre: configuration.adapters.find(adapter.id) $\neq$ null
	configuration.adapters.find(adapter.id).find(device.id) $\neq$ null
	post: adapter.devices.remove(device)
Invariants	configuration $\neq$ null

UC5 allows a configurator to add an adapter to the system. The configurator provides the system with the adapter to add. The system assigns an identifier to the adapter and adds it to the configuration. We assume the configuration provides a generateAdapterId() function that generate an identifier that is unique among the adapters of the configuration.

Use Cases	**UC5 - Add an adapter**
Class	*HomePort*
Method	addAdapter(Adapter adapter)
	pre: none.
	post: adapter.id = configuration.generateAdapterId()
	configuration.adapters.add(adapter);
Invariants	configuration $\neq$ null

UC6 allows a configuratot to remove an adapter from the system. The configurator provides the system with the adapter to remove. The system checks that the adapter exists in the system, and removes it from the configuration if it does.

Use Cases	**UC6 - Remove an adapter**
Class	*HomePort*
Method	removeAdapter(Adapter adapter)
	pre: configuration.find(adapter) $\neq$ null
	post: adapter.id = configuration.generateAdapterId()
	configuration.adapters.add(adapter);
Invariants	configuration $\neq$ null