Katholieke
Universiteit
Leuven

**Department of
Computer Science**

Shared Internet Of Things Infrastructure Platform:

The Complete Architecture

Software Architecture (H09B5a and H07Z9a) – Part 2b

FILIPCIKOVA-HALILOVIC

Monika Filipcikova (r0683254)
Armin Halilovic(r0679689)

Contents

1. Architectural Decisions

> ℹ **Note:** This section discusses *all* your architectural decisions *in-depth*. First, *all* decisions related to the non-functionals are discussed in detail. Next, *all* other decisions are listed and discussed.

> ✔ **Hint:** Don't just say *what* you have done. Explain *why* you have done it.

1.1 ReqX: Requirement Name

> 📑 **TODO:** Use this section structure for each requirement

Key Decisions

> 📑 **TODO:** Briefly list your key architectural decisions. Pay attention to the solutions that you employed (in your own terms or using tactics and/or patterns).

- decision 1
- ...

Employed tactics and patterns: ...

Rationale

> 📑 **TODO:** Describe the design choices related to *ReqX* together with the rationale of why these choices where made.

Considered Alternatives

Alternative(s) for choice 1   Explain what alternative(s) you considered for this design choice and why they where not selected.

Deployment Decisions

...

Considered Deployment Alternatives

...

1.2 Other decisions

> 📑 **TODO:** *Optional* If you have made any other important architectural decisions that do not directly fit in the sections of the other qualities you can mention them here.
> Follow the same structure as above.

1.2.1 Decision 1

KeyDecisions

. . .

Rationale

. . .

Considered Alternatives

. . .

Deployment Decisions

. . .

Considered Deployment Alternatives

. . .

1.3 Discussion

> 📑 **TODO:** Use this section to discuss your architecture in retrospect. For example, what are the strong points of your architecture? What are the weak points? Is there anything you would have done otherwise with your current experience? Are there any remarks about the architecture that you would give to your customers? Etc.

2. Client-server view (UML Component diagram)

Figures

> ✔ **Hint:** No need to just repeat what we can see on the diagram.
> Don't do this: *As you can see on fig. x: comp A consists of B and C, and C connects to D.*
> But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

> ✔ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.
> If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

> ✔ **Hint:** If you have any doubts about the size of your figures, it is better to make your figure too large than too small. Alternatively, you can test the readability by printing it.

> ⚠ **Attention:** With regard to the context diagram, recall the lectures on what it means and should contain. Be sure not to miss any elements here. This is a frequent source of errors.

> ⚠ **Attention:** Make sure your main component-and-connector and context diagrams are consistent.

2.1 Context diagram

TODO: find a way to display the page horizontally with the image covering the whole page.

The context diagram of the client-server view is displayed in figure 2.1.

The external components are as follows.

- NotificationDeliveryService: blabla
- InfrastructureOwnerClient: blabla
- CustomerOrganisationClient: blabla

> 🔖 **TODO:** The context diagram of the client-server view: Discuss which components communicate with external components and what these external components represent.

2.2 Primary diagram

The primary diagram of the client-server view is displayed in figure 2.2.

> 🔖 **TODO:** The primary diagram and accompanying explanation.

Figure 2.1: Context diagram for the client-server view.

Figure 2.2: Primary diagram of the client-server view.

3. Decomposition view (UML Component diagram)

Figures

> ✔ **Hint:** No need to just repeat what we can see on the diagram.
> Don't do this: *As you can see on fig. x: comp A consists of B and C, and C connects to D.*
> But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

> ✔ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.
> If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

> ⚠ **Attention:** *Consistency between views!* Be sure to check for consistency between the client-server view and your decompositions.

> ⚠ **Attention:** *Consistency of a single decomposition!* Make sure that every interface provided or required by the decomposed component, is provided or required by a subcomponent in the decomposition.
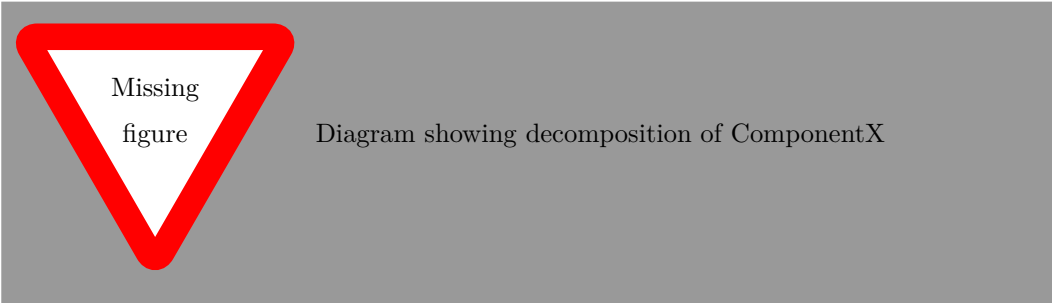
Figure 3.1: Decomposition of `ComponentX`
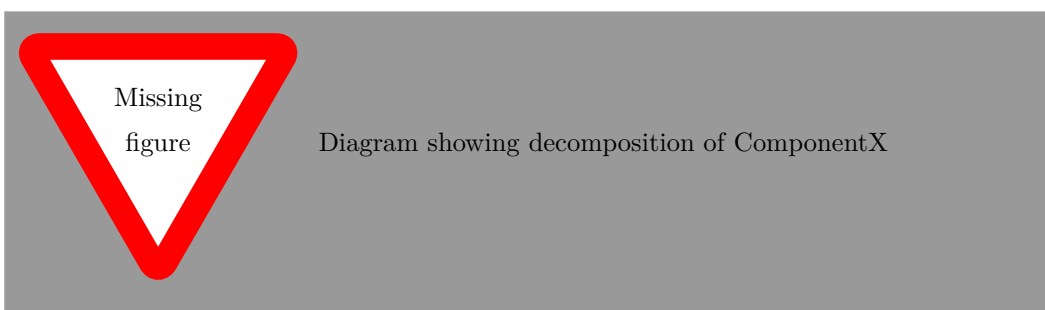
Figure 3.2: Decomposition of `ComponentY`.
This caption contains a longer explanation over multiple lines. This additional explanation is not shown in the list of figures.

4. Deployment view (UML Deployment diagram)

Figures

> ✔ **Hint:** No need to just repeat what we can see on the diagram.
> Don't do this: *As you can see on fig. x: components A and B are deployed on node C.*
> But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

> ✔ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.
> If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

> ⚠ **Attention:** Connect nodes on the deployment diagram, *not* components.

> ⚠ **Attention:** *Consistency between views!* Be sure to check for consistency between the client-server/decomposition view and your deployment view.

4.1 Context diagram

TODO: find a way to display the page horizontally with the image covering the whole page.

The context diagram for the deployment view is displayed in figure 4.1.

Components X, Y, Z are deployed on multiple nodes for bla bla bla.
Components A and B communicate using the C protocol...

> 📑 **TODO:** Describe the context diagram for the deployment view. For example, which protocols are used for communication with external systems and why?

4.2 Primary diagram

The primary diagram for the deployment view is displayed in figure 4.2.
TODO: add references to "architectural decisions" where we made some choices related to deployment of components.

> 📑 **TODO:** The primary deployment diagram itself. This discussion on the parts of the deployment diagram which are crucial for achieving certain non-functional requirements, and any alternative deployments that you considered, should be in the architectural decisions chapter.
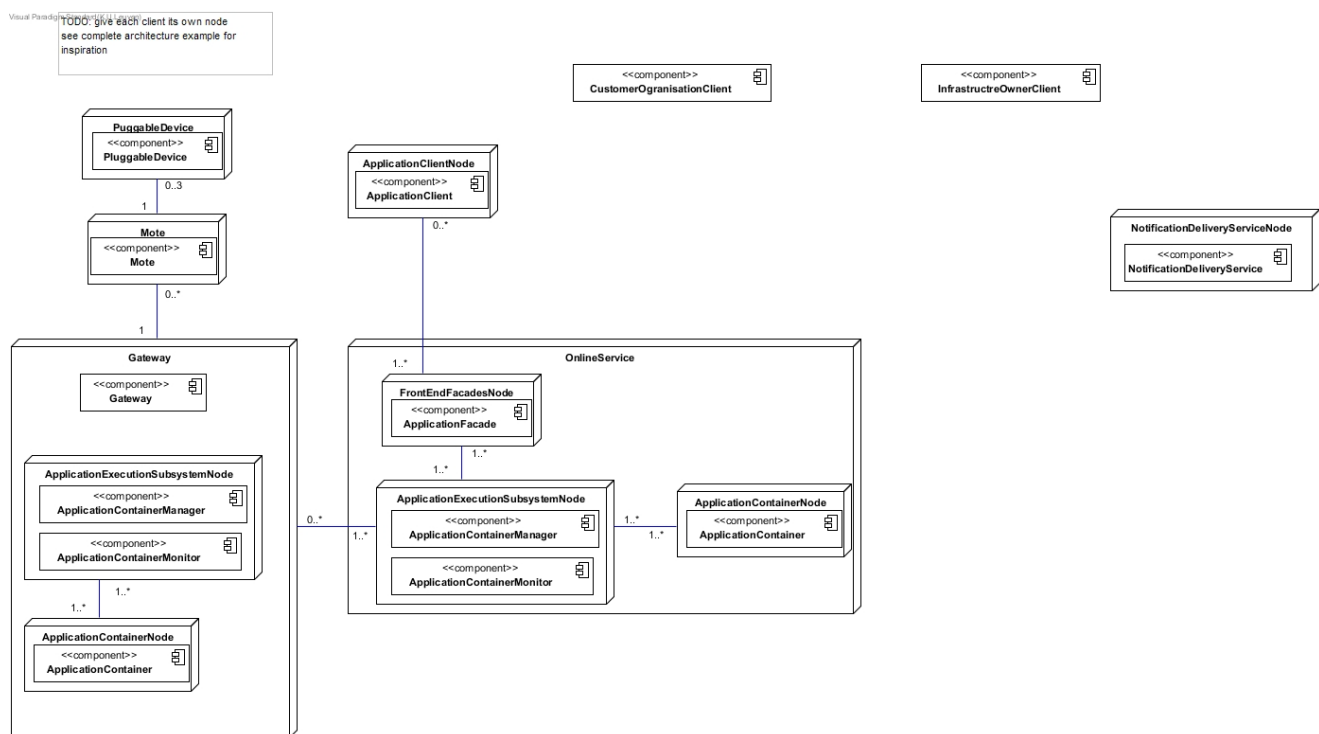
Figure 4.1: Context diagram for the deployment view.

No online service node here, because the primary diagram represents the whole online service.

ApplicationContainerMgmtNode

<<component>>
ApplicationContainerManager

<<component>>
ApplicationContainerMonitor

1

1..*

ApplicationContainerNode

<<component>>
ApplicationContainer

PluggableDeviceDatabase

<<component>>
PluggableDeviceDataDB

OtherDataDatabase

<<component>>
OtherDataDB

DeviceDatabase

<<component>>
DeviceDB

Figure 4.2: Primary diagram for the deployment view.

5. Scenarios

Figures

> ✔ **Hint:** No need to just repeat what we can see on the diagram.
> Don't do this: *As you can see on fig. x: component A calls operation b, next component C calls operation d.*
> But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

> ✔ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.
> If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

> ⚠ **Attention:** Do include a list of which sequence diagrams together illustrate a which scenario from the assignment.

> ✔ **Hint:** Don't only model the 'happy path' in your sequence diagrams. Take into account the quality attributes. For example, what happens when a certain component fails (Av) or overloads (P)? Use the sequence diagrams to illustrate how you have achieved the qualities in your architecture.

5.1 Scenarios

> 🔖 **TODO:** Illustrate how your architecture fulfills the most important data flows. As a rule of thumb, focus on the scenario of the assignment. Describe the scenario in terms of architectural components using UML Sequence diagrams and further explain the most important interactions in text. Illustrating the scenarios serves as a quick validation of the completeness of your architecture. If you notice at this point that for some reason, certain functionality or qualities are not addressed sufficiently in your architecture, it suffices to document this, together with a rationale of why this is the case according to you. You do not have to further refine you architecture at this point.

This section lists which sequence diagrams belong to which scenarios:

- UC11: Sensor data being processed by the system
  Figure 5.1
- UC19: Subscribing to an application
  Figure 5.2
- UC12: Applications issuing actuation commands
  Figure 5.3
- UC14, Av3, UC18: Sensors/actuators failing
  Figure 5.4
  This scenario displays the data flow when sensors/actuators fail, causing

- – deactivation of specific applications
- – a redundant sensor/actuator to take over in the context of a single application
- Av2: Application crash
  Figure 5.5
- U2, UC4: Plugging in a new pluggable device (sensor or actuator)
  Figure 5.6
- Av1, UC15: Detection and handling of communication channel failure
  Figure 5.7
- UC22, U1: Upgrading an application
  Figure 5.8
- UC26, UC27, UC12: Sending actuation commands via a mobile app
  Figure 5.9
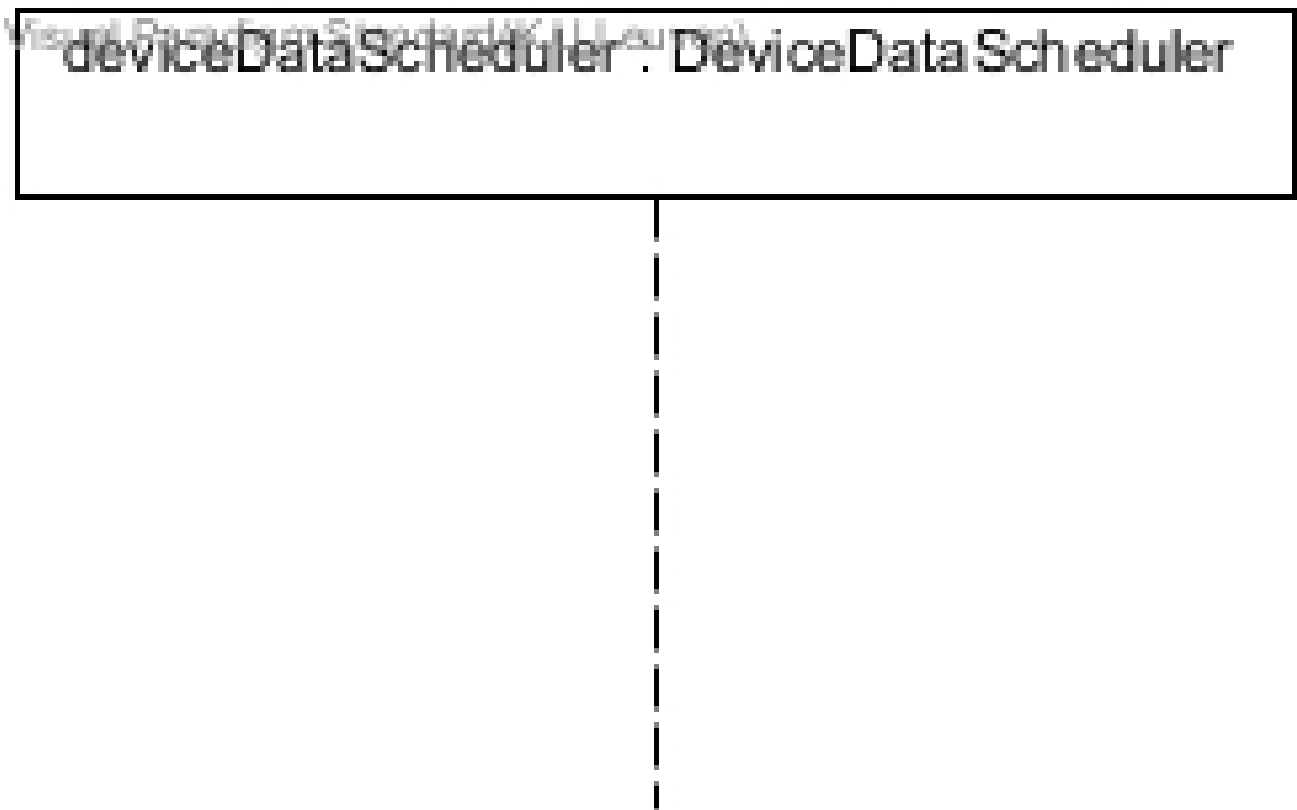
Figure 5.1: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

customerOrganisationFacade : CustomerOrganisationFacade
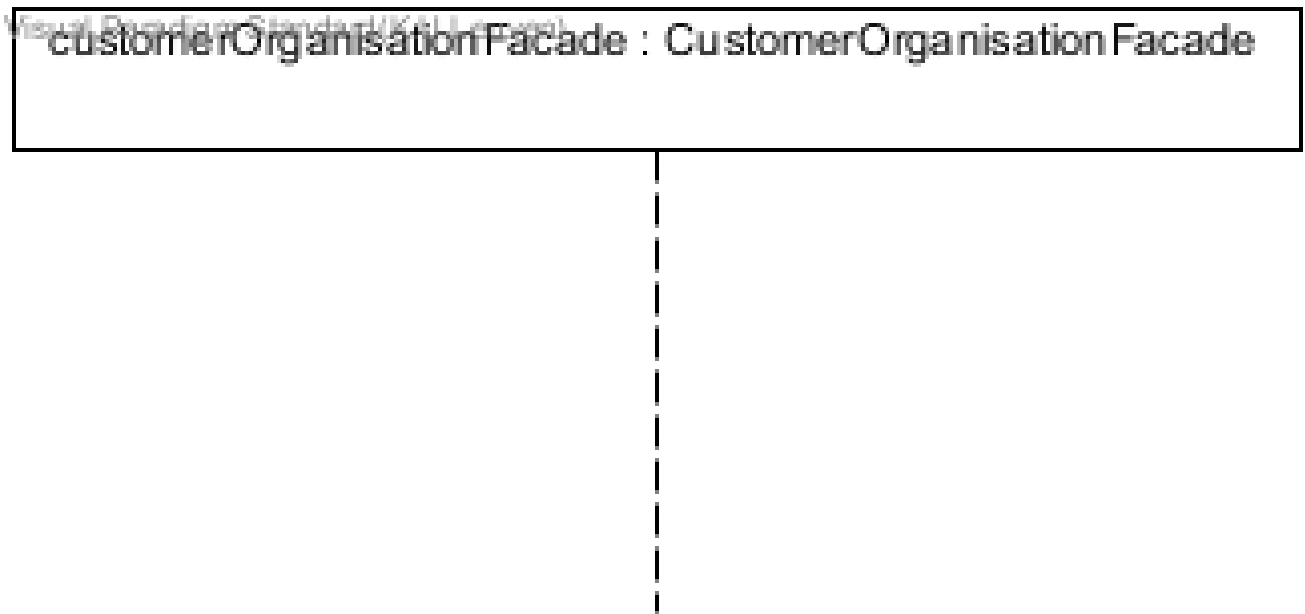
Figure 5.2: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

application Manager : ApplicationManager
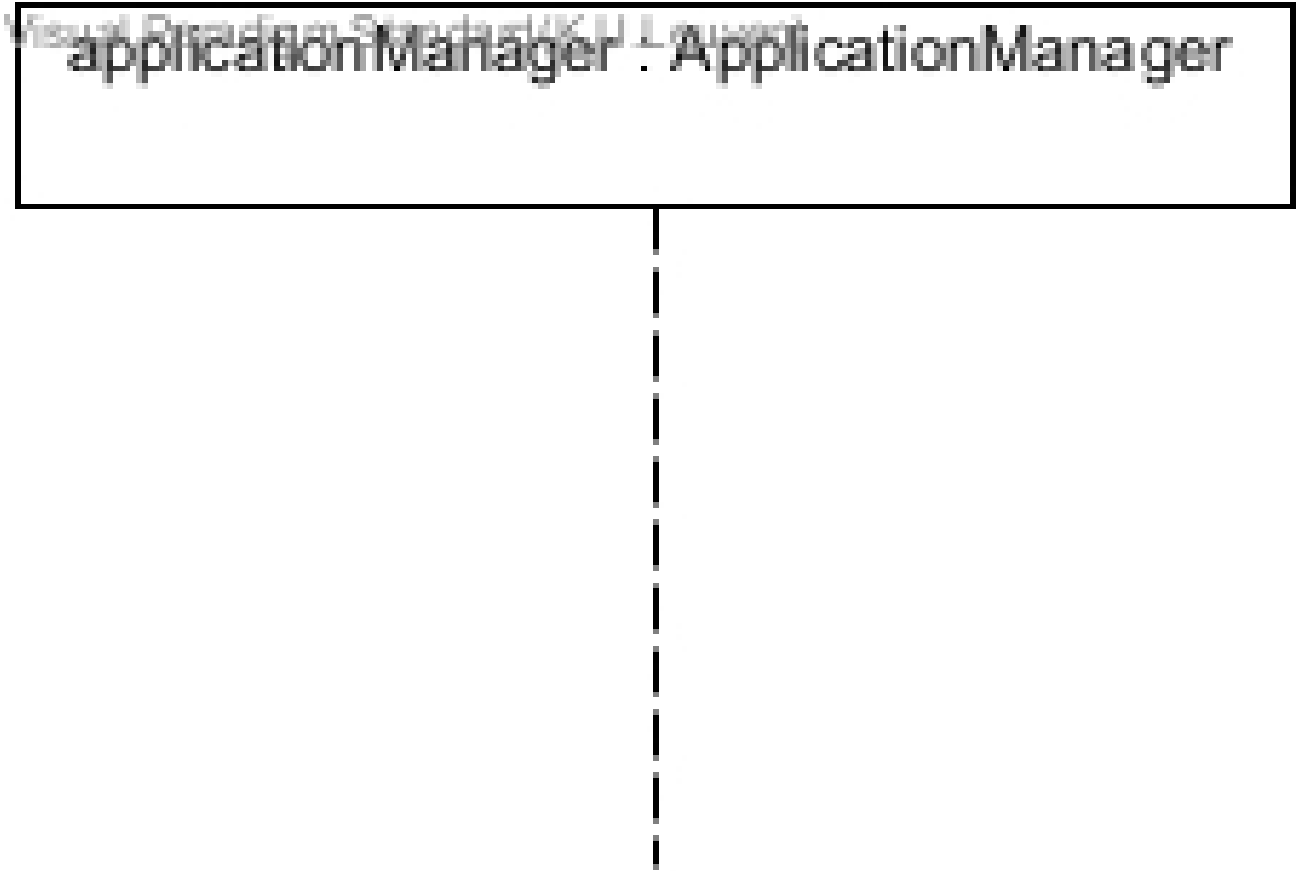
Figure 5.3: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

17
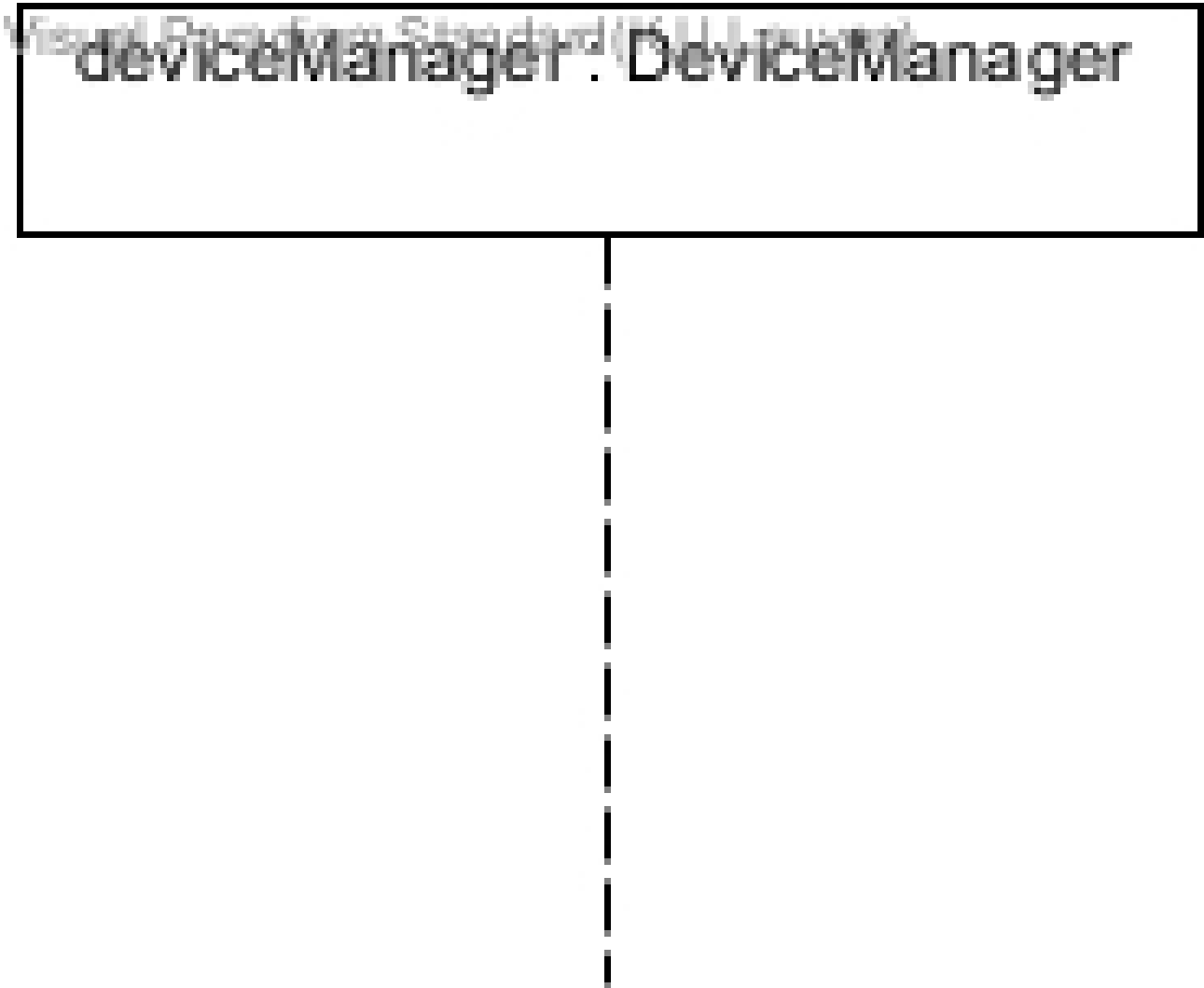
Figure 5.4: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.
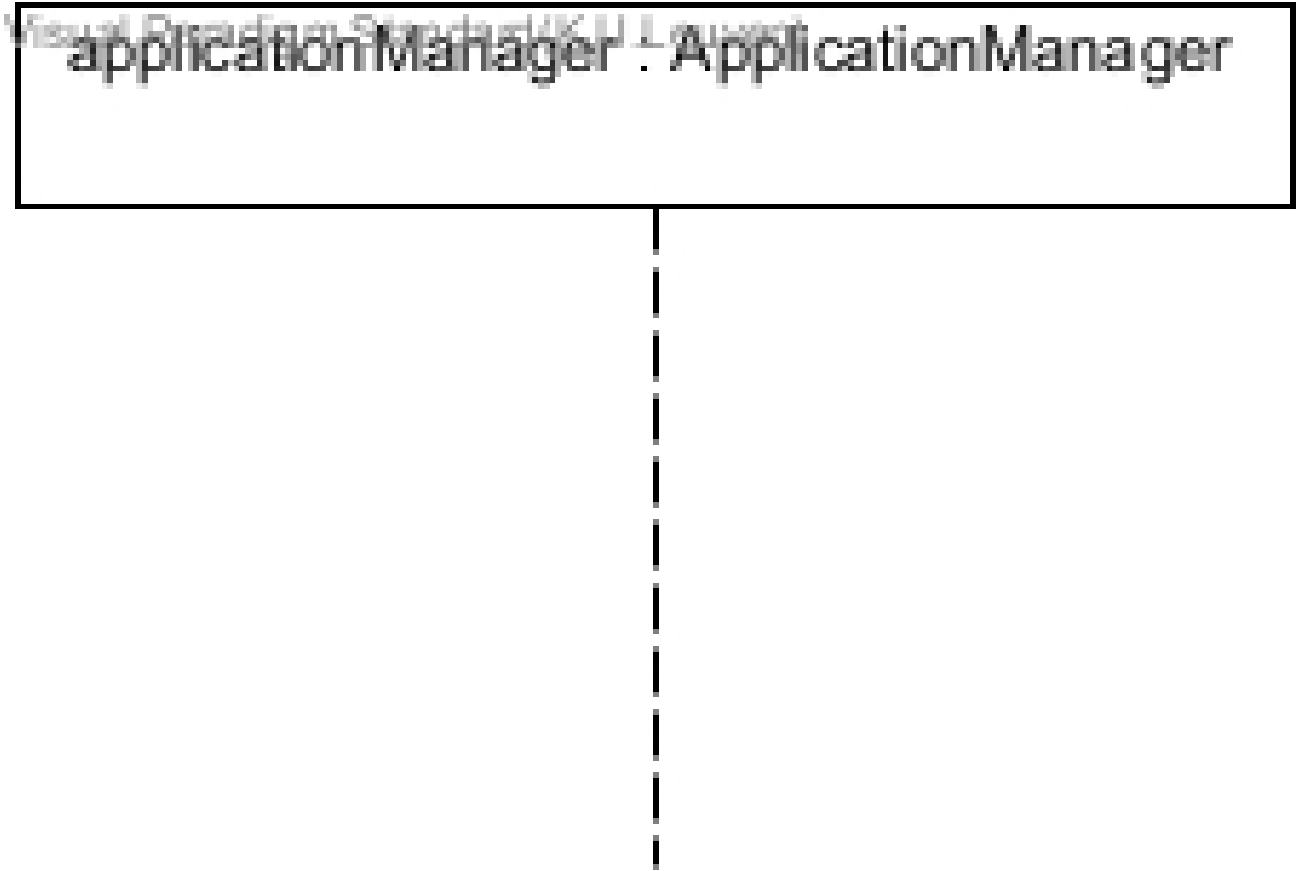
Figure 5.5: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

gatewayFacade : Gateway
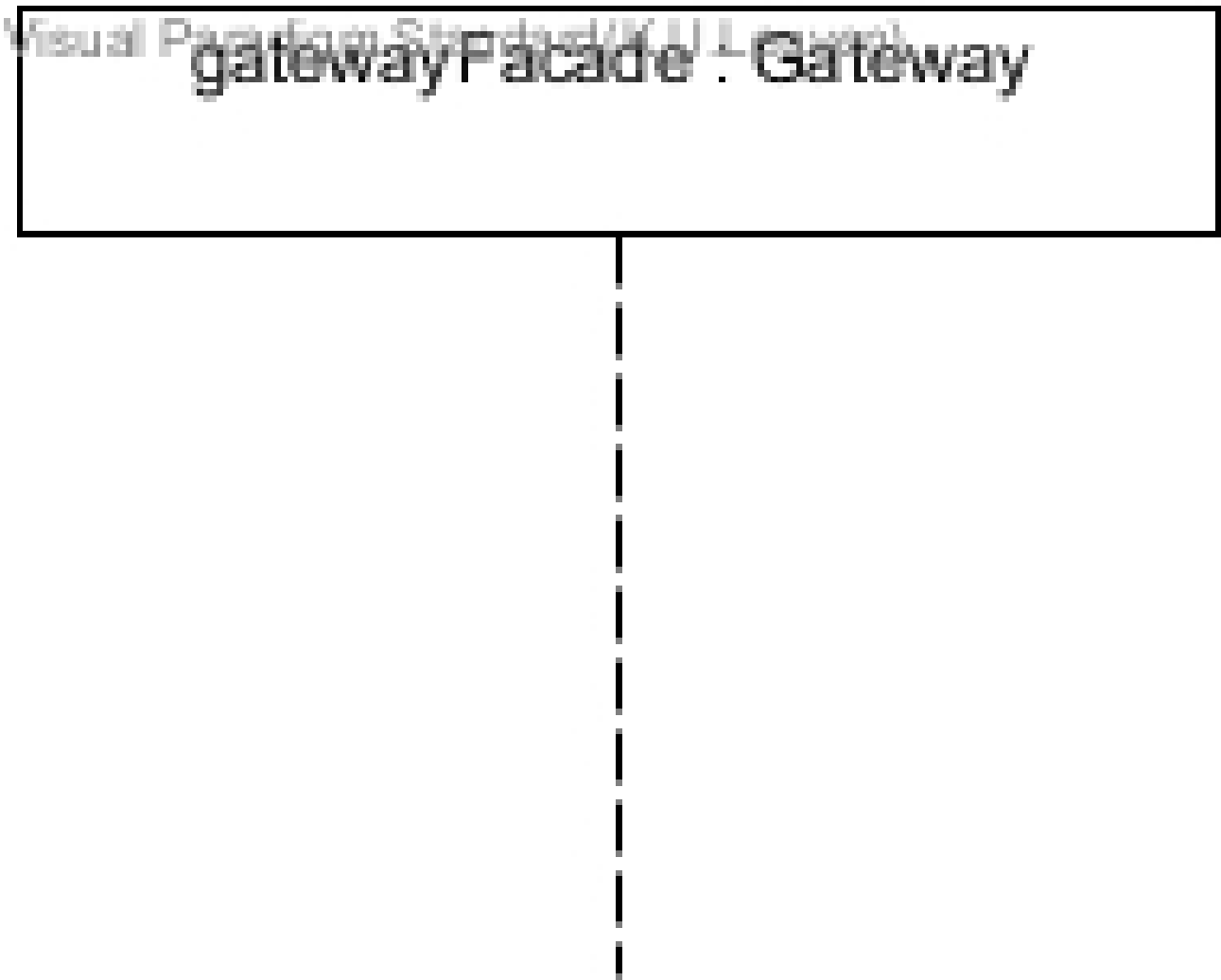
Figure 5.6: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

gatewayFacade : Gateway
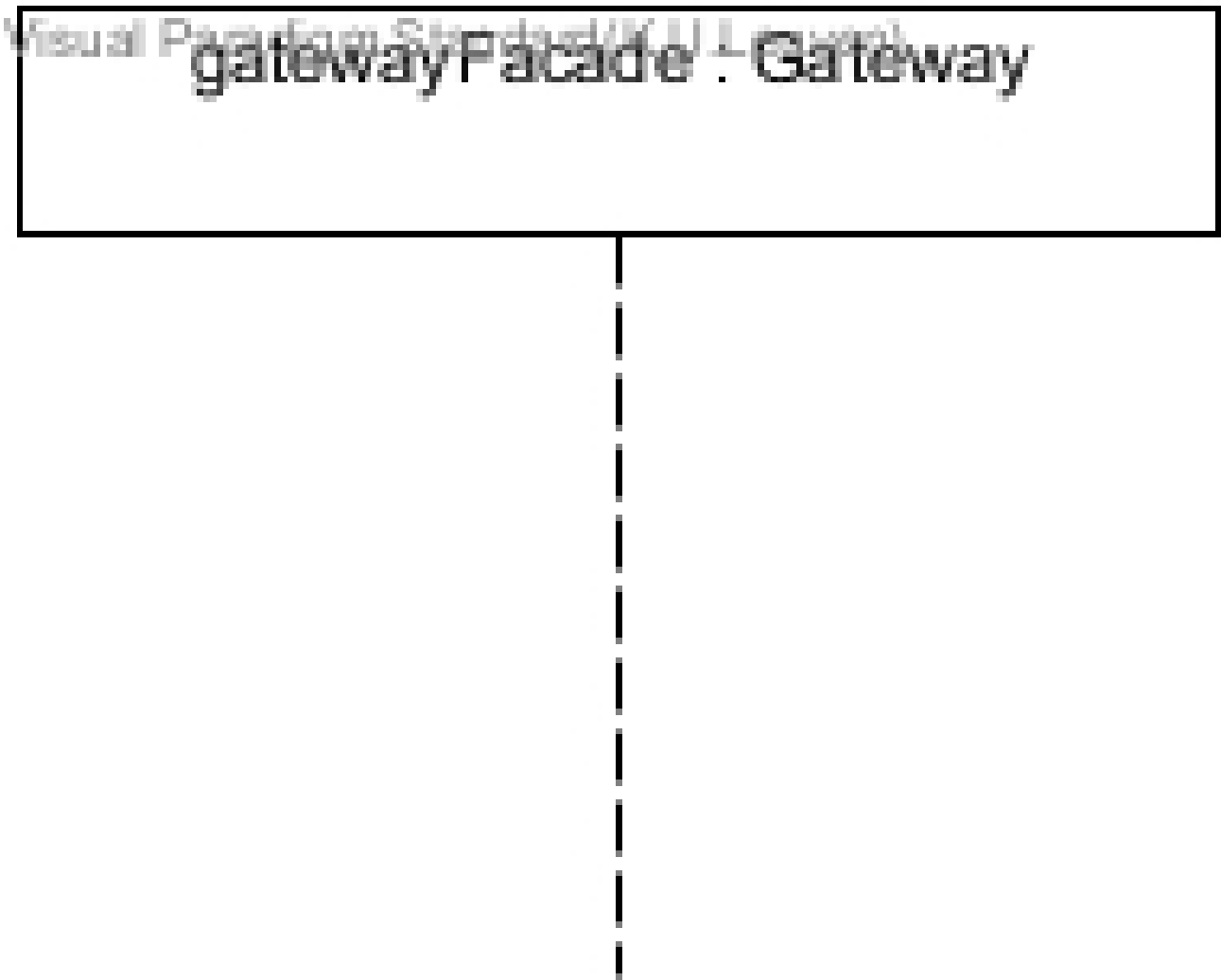
Figure 5.7: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.
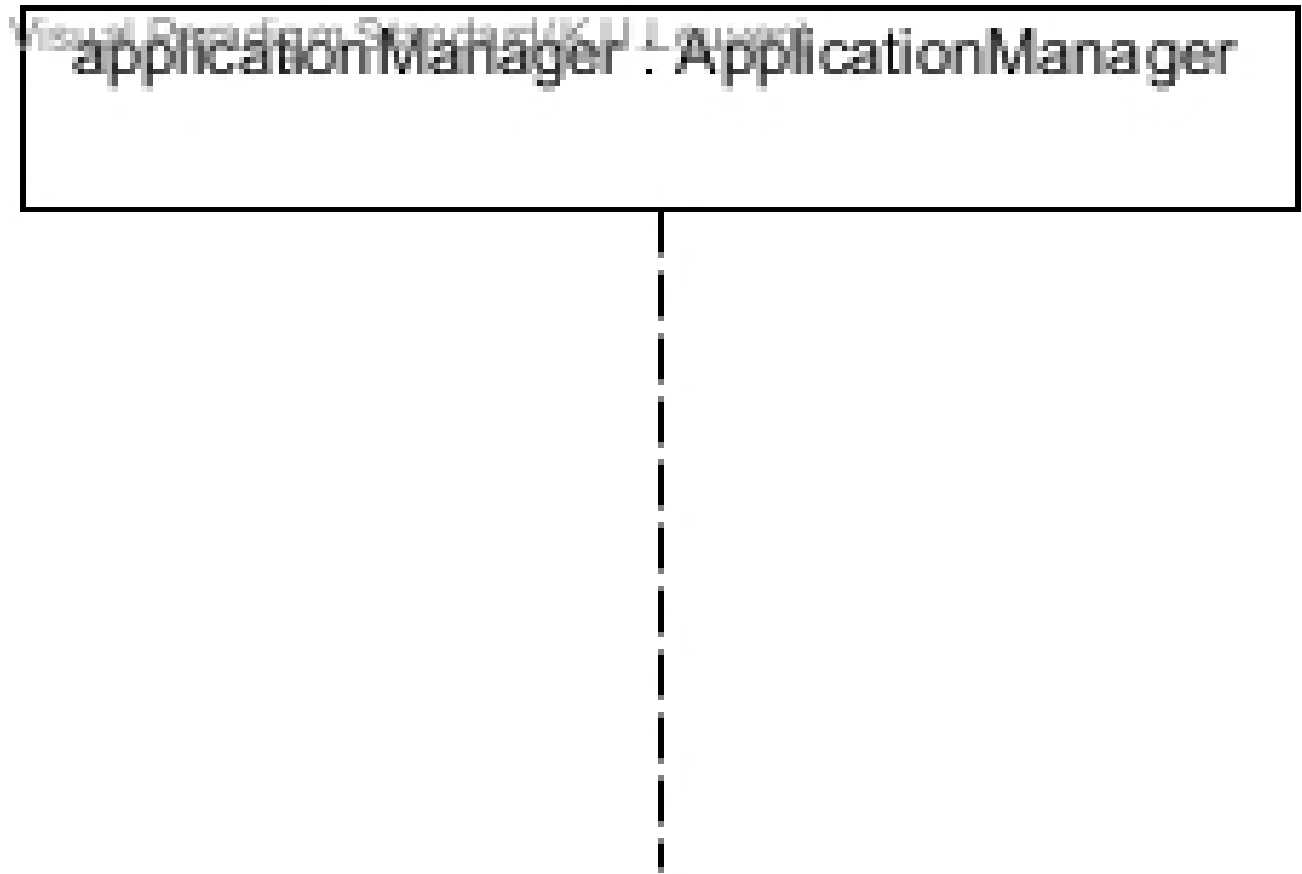
Figure 5.8: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

application Manager : ApplicationManager
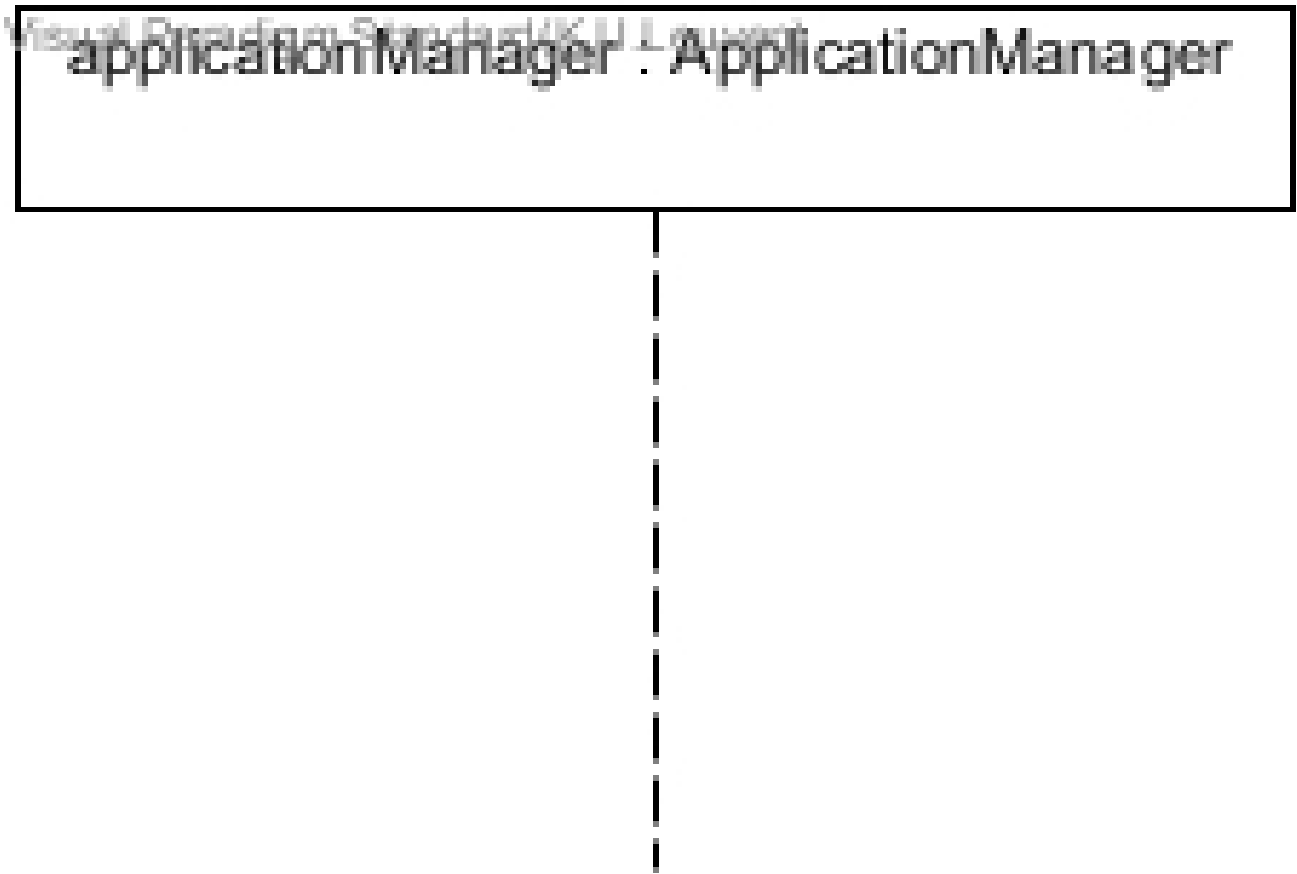
Figure 5.9: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

6. Element Catalog and Datatypes

Each method contains a short note on why the method was added (under "Created for"). This was done to keep track of our decisions and does not mean that the methods can only be used for the Quality Attribute/Use Case referenced in the "Created for" note.

7. Catalog

7.1 Components

7.1.1 AccessRightsManager

Responsibility: Responsible for all functionality related to access rights to pluggable devices. E.g. retrieving the access rights a customer organisation has for a device, updating access rights for customer organisations, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ AccessRightsMgmt
Required interfaces: ⊰ DBAccessRightsMgmt

7.1.2 ApplicationClient

Responsibility: Represents the client for applications. This is their external front end.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ⊰ AppDeviceData, ⊰ DeviceCommands

7.1.3 ApplicationContainer

Responsibility: This component contains a sandbox environment for an application instance to execute in. The container itself can detect when applications crash. In that case, the container tries to restart the application up to 3 times. If the application continues to crash, it is suspended and the ContainerMonitor is notified.
Super-components: ⊟ ApplicationManager
Sub-components: None
Provided interfaces: ⊶ AppInstanceMgmt, ⊶ AppMessages, ⊶ AppMonitoring
Required interfaces: ⊰ AppDeviceData, ⊰ AppMessages, ⊰ AppStatus, ⊰ DataConversion, ⊰ DeviceCommands

7.1.4 ApplicationContainerManager

Responsibility: Responsible for management of ApplicationContainers. E.g. creation, suspending containers, forwarding data to containers, etc. All communication that is destined to ApplicationContainers goes through here first, since the ApplicationContainerManager knows where exactly ApplicationContainers are deployed.
Super-components: ⊟ ApplicationManager
Sub-components: None
Provided interfaces: ⊶ AppDeviceData, ⊶ AppInstanceMgmt, ⊶ AppMessages, ⊶ AppMonitoring, ⊶ Commands, ⊶ DeviceCommands, ⊶ Monitoring
Required interfaces: ⊰ AppDeviceData, ⊰ AppInstanceMgmt, ⊰ AppMessages, ⊰ AppMonitoring, ⊰ AppStatus, ⊰ Commands, ⊰ Notify

7.1.5 ApplicationContainerMonitor

Responsibility: Monitors ApplicationContainers and keeps track of their ApplicationInstances.
Super-components: ⊟ ApplicationManager
Sub-components: None
Provided interfaces: ⊶ AppStatus, ⊶ Monitoring

Required interfaces: ⊰ AppMonitoring, ⊰ Notify

7.1.6   ApplicationExecutionSubsystemMonitor

Responsibility: Monitors the application execution subsystem (ApplicationContainerManager,ApplicationContainerMonitor
Super-components: ▤ ApplicationManager
Sub-components: None
Provided interfaces: None
Required interfaces: ⊰ Monitoring, ⊰ Notify

7.1.7   ApplicationFacade

Responsibility: Acts as an access point for ApplicationClients and handles all functionality that can be done by application's external front ends.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ AppDeviceData, ⊶ DeviceCommands
Required interfaces: ⊰ AppDeviceData, ⊰ DeviceCommands

7.1.8   ApplicationManagementLogic

Responsibility: The entry point for ApplicationManager. Responsible for tying together all components of the ApplicationManager and handling requests related to applications.
Super-components: ▤ ApplicationManager
Sub-components: None
Provided interfaces: ⊶ AppDeviceData, ⊶ AppMessages, ⊶ DeviceCommands, ⊶ ForwardData, ⊶ FrontEndAppRequests, ⊶ GWAppInstanceMgmt
Required interfaces: ⊰ AppDeviceMgmt, ⊰ AppInstanceMgmt, ⊰ AppMessages, ⊰ Commands, ⊰ DBAppMgmt, ⊰ InvoiceMgmt, ⊰ Notify, ⊰ RequestData, ⊰ RoleMgmt, ⊰ TopologyMgmt

7.1.9   ApplicationManager

Responsibility: Responsible for activating/deactivating applications, setting pluggable device redundancy requirements on DeviceManager components, and using NotificationHandler to send notifications to customer organisations.
Super-components: None
Sub-components: ▤ DeviceCommandConstructor, ▤ ApplicationContainerMonitor, ▤ ApplicationContainerManager, ▤ ApplicationManagementLogic, ▤ ApplicationExecutionSubsystemMonitor, ▤ DeviceDataConverter, ▤ ApplicationContainer
Provided interfaces: ⊶ AppDeviceData, ⊶ AppMessages, ⊶ DeviceCommands, ⊶ ForwardData, ⊶ FrontEndAppRequests, ⊶ GWAppInstanceMgmt
Required interfaces: ⊰ AppDeviceMgmt, ⊰ AppInstanceMgmt, ⊰ AppMessages, ⊰ DBAppDeviceMgmt, ⊰ DBAppMgmt, ⊰ DeviceCommands, ⊰ InvoiceMgmt, ⊰ Notify, ⊰ RequestData, ⊰ RoleMgmt, ⊰ TopologyMgmt

7.1.10   BrokerLogic

Responsibility: Undefined
Super-components: ▤ GatewayBroker
Sub-components: None
Provided interfaces: ⊶ AppDeviceMgmt, ⊶ AppInstanceMgmt, ⊶ AppMessages, ⊶ DeviceCommands, ⊶ GWCommunication
Required interfaces: ⊰ AppDeviceData, ⊰ AppMessages, ⊰ DBDeviceMgmt, ⊰ DeviceData, ⊰ GatewayUpdates, ⊰ GWAppInstanceMgmt, ⊰ Notify, ⊰ OSCommunication, ⊰ TopologyMgmt

7.1.11 BrokerLogic

Responsibility: Undefined
Super-components: ▤ OnlineServiceBroker ▷ ▤ Gateway
Sub-components: None
Provided interfaces: ⊸ AppDeviceData, ⊸ AppMessages, ⊸ DBDeviceMgmt, ⊸ DeviceData, ⊸ GWAppInstanceMgmt, ⊸ Notify, ⊸ OSCommunication, ⊸ OSMonitoring, ⊸ TopologyMgmt
Required interfaces: ⊰ AppDeviceMgmt, ⊰ AppInstanceMgmt, ⊰ AppMessages, ⊰ DeviceCommands, ⊰ GWCommunication, ⊰ OSUpdates, ⊰ Requests

7.1.12 CustomerOgranisationClient

Responsibility: Represents the client used by a customer organisation. This is the user's dashboard.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ⊰ SubscriptionMgmt

7.1.13 CustomerOrganisationFacade

Responsibility: Acts as an access point for CustomerOrganisationClients and handles all functionality that can be done by customer organisations.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ SubscriptionMgmt
Required interfaces: ⊰ FrontEndAppRequests, ⊰ RoleMgmt, ⊰ SubscriptionMgmt, ⊰ TopologyMgmt

7.1.14 DeviceCommandConstructor

Responsibility: Is responsible for verifying, constructing, and sending commands from applications for pluggable devices.
Super-components: ▤ ApplicationManager
Sub-components: None
Provided interfaces: ⊸ Commands
Required interfaces: ⊰ DBAppDeviceMgmt, ⊰ DeviceCommands

7.1.15 DeviceDataConverter

Responsibility: The DeviceDataConverter is responsible for conversions pluggable device data for applications. There is one in the Online Service for application instances running in the Online Service and one in Gateways for application instances running on gateways.
Super-components: ▤ ApplicationManager
Sub-components: None
Provided interfaces: ⊸ DataConversion
Required interfaces: None

7.1.16 DeviceDataScheduler

Responsibility: Responsible for scheduling incoming read and write requests for pluggable device data. Monitors throughput of requests and switches between normal and overload mode when appropriate. Avoids starvation of any type of request.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ DeviceData, ⊸ RequestData
Required interfaces: ⊰ DBDeviceData, ⊰ ForwardData

7.1.17 DeviceDB

Responsibility: Contains all information related to devices in the system, but not pluggable device data such as sensor data or actuation statuses. The data includes information about pluggable devices, motes, gateways, topologies, access rights, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ DBAccessRightsMgmt, ⊸ DBAppDeviceMgmt, ⊸ DBDeviceMgmt, ⊸ DBIODeviceMgmt, ⊸ DBTopologyMgmt
Required interfaces: None

7.1.18 DeviceManager

Responsibility: Monitors connected/operational devices on a gateway. Sends notifications in case of hardware failure. Can send a command to disable or reactivate applications when necessary.
Keeps a local cache of data about pluggable devices from the DeviceDB, such as specific formatting syntax, possible configuration parameters, etc. This data can then be used by the DeviceCommandConstructor on the DeviceManager to convert commands from ApplicationContainers on the gateway before they are sent to pluggable devices.
Super-components: ▣ Gateway
Sub-components: None
Provided interfaces: ⊸ AppDeviceMgmt, ⊸ DeviceCommands, ⊸ DeviceData, ⊸ DeviceMgmt, ⊸ Heartbeat
Required interfaces: ⊰ AppInstanceMgmt, ⊰ DBDeviceMgmt, ⊰ DeviceData, ⊰ DeviceMgmt, ⊰ GWAppInstanceMgmt, ⊰ Notify, ⊰ TopologyMgmt

7.1.19 Gateway

Responsibility: Main component on the gateway that allows different components to work with each other. E.g. transmits heartbeats from motes to DeviceManager, transmits commands to shut down applications, triggers notifications to be generated, ...
Super-components: None
Sub-components: ▣ DeviceManager, ▣ OnlineServiceBrokerMonitor, ▣ OnlineServiceBroker
Provided interfaces: ⊸ AppDeviceMgmt, ⊸ DeviceData, ⊸ Heartbeat, ⊸ OSCommunication
Required interfaces: ⊰ DeviceData, ⊰ DeviceMgmt, ⊰ DeviceMgmt, ⊰ GWCommunication

7.1.20 GatewayBroker

Responsibility: Undefined
Super-components: None
Sub-components: ▣ BrokerLogic, ▣ GatewayMonitor
Provided interfaces: ⊸ AppDeviceMgmt, ⊸ AppInstanceMgmt, ⊸ AppMessages, ⊸ DeviceCommands, ⊸ GWBrokerMonitoring, ⊸ GWCommunication
**Required interfaces:** ⊰ AppDeviceData, ⊰ AppMessages, ⊰ DBDeviceMgmt, ⊰ DeviceData, ⊰ GWAppInstanceMgmt, ⊰ Notify, ⊰ OSCommunication, ⊰ TopologyMgmt

7.1.21 GatewayBrokerMonitor

Responsibility: Undefined
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ⊰ DBDeviceMgmt, ⊰ GWBrokerMonitoring, ⊰ Notify

7.1.22   GatewayMonitor

Responsibility: Undefined
Super-components: ▤ GatewayBroker
Sub-components: None
Provided interfaces: ⊶ GatewayUpdates
Required interfaces: None

7.1.23   InfrastructreOwnerClient

Responsibility: Represents the client used by an infrastructure owner.   This is the user's dashboard.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ⊰ AccessRights

7.1.24   InfrastructureOwnerFacade

Responsibility: Acts as an access point for InfrastructureOwnerClients and handles all functionality that can be done by infrastructure owners.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ AccessRights
Required interfaces: ⊰ AccessRightsMgmt, ⊰ FrontEndAppRequests, ⊰ IOMgmt

7.1.25   InfrastructureOwnerManager

Responsibility: Responsible for all functionality related to infrastructure owners.   E.g.   looking up the devices they own, retrieving a list of customer organisations that they are associated to, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ IOMgmt
Required interfaces: ⊰ DBIODeviceMgmt, ⊰ DBIOMgmt

7.1.26   InvoiceManager

Responsibility: Responsible for all functionality related to access rights to invoicing.   E.g.   creating invoices.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ InvoiceMgmt
Required interfaces: ⊰ DBInvoiceMgmt

7.1.27   Mote

Responsibility: Sends heartbeats to the GatewayFacade.   Includes a list of connected pluggable devices in the heartbeats.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ DeviceData, ⊶ DeviceMgmt
Required interfaces: ⊰ Actuate, ⊰ Config, ⊰ DeviceData, ⊰ DeviceMgmt, ⊰ Heartbeat, ⊰ RequestData

7.1.28 NotificationDeliveryService

Responsibility: Represents a third party notification delivery service.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ NotificationDeliveryMgmt
Required interfaces: ⊰ DeliveryMgmt

7.1.29 NotificationHandler

Responsibility: Responsible for generation, storage, and delivery of notifications based on users' preferred communication channel.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ DeliveryMgmt, ⊸ Notify
Required interfaces: ⊰ DBNotificationMgmt, ⊰ NotificationDeliveryMgmt

7.1.30 OnlineServiceBroker

Responsibility: Undefined
Super-components: ▤ Gateway
Sub-components: ▤ BrokerLogic, ▤ OnlineServiceMonitor, ▤ RequestStore
Provided interfaces: ⊸ AppDeviceData, ⊸ AppMessages, ⊸ DBDeviceMgmt, ⊸ DeviceData, ⊸ GWAppInstanceMgmt, ⊸ Notify, ⊸ OSBrokerMonitoring, ⊸ OSCommunication, ⊸ TopologyMgmt
Required interfaces: ⊰ AppDeviceMgmt, ⊰ AppInstanceMgmt, ⊰ AppMessages, ⊰ DeviceCommands, ⊰ GWCommunication

7.1.31 OnlineServiceBrokerMonitor

Responsibility: Undefined
Super-components: ▤ Gateway
Sub-components: None
Provided interfaces: None
Required interfaces: ⊰ OSBrokerMonitoring

7.1.32 OnlineServiceMonitor

Responsibility: Undefined
Super-components: ▤ OnlineServiceBroker ▷ ▤ Gateway
Sub-components: None
Provided interfaces: ⊸ OSUpdates
Required interfaces: ⊰ OSMonitoring

7.1.33 OtherDataDB

Responsibility: General database for data. For example, storage of data about notifications.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ DBAppMgmt, ⊸ DBInvoiceMgmt, ⊸ DBIOMgmt, ⊸ DBNotificationMgmt, ⊸ DBSubscriptionMgmt, ⊸ DBUserRoleMgmt
Required interfaces: None

7.1.34 PluggableDevice

Responsibility: Responsible for sending pluggable device data to MoteFacade.   Needs to be initialised in order for the data to be used/stored.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ Actuate, ⊶ Config, ⊶ RequestData
Required interfaces: ⊰ DeviceData

7.1.35 PluggableDeviceDataDB

Responsibility: Database dedicated to pluggable device data only.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ DBDeviceData
Required interfaces: None

7.1.36 RequestStore

Responsibility: Undefined
Super-components: ▱ OnlineServiceBroker ▷ ▱ Gateway
Sub-components: None
Provided interfaces: ⊶ AppMessages, ⊶ DeviceData, ⊶ Requests
Required interfaces: ⊰ AppMessages, ⊰ DeviceData

7.1.37 SubscriptionManager

Responsibility: Responsible for all functionality related to access rights to subscriptions.   E.g. retrieving the applications that a customer organisation can subscribe to, creating new subscriptions to ApplicationInstances, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ SubscriptionMgmt
Required interfaces: ⊰ DBSubscriptionMgmt, ⊰ FrontEndAppRequests

7.1.38 TopologyManager

Responsibility: Responsible for all functionality related to topology.   E.g.   Adding a new mote to the topology of an infrastructure, checking whether or not all devices used by an application are active in the topology, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ TopologyMgmt
Required interfaces: ⊰ DBTopologyMgmt

7.1.39 UserRolesManager

Responsibility: Responsible for all functionality related to access rights to user roles.   E.g.   retrieving the user roles that are mandatory for a certain application, updating the roles assigned to users, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊶ RoleMgmt
Required interfaces: ⊰ DBUserRoleMgmt

7.2 Interfaces

7.2.1 AccessRights

Provided by: ▣ `InfrastructureOwnerFacade`
Required by: ▣ `InfrastructreOwnerClient`
Operations:

- configureDevice(**PluggableDeviceID** pID)
    - Effect: Returns a map of AccessRights and the IDs of customer organisations that have those AccessRights.
    - Created for: UC9.3 - UC9.4
- List<**PluggableDeviceInfo**> getAccessRights(int infrastructureOwnerID)
    - Effect: Returns a list of PluggableDeviceInfo to display so an infrastructure owner can select a device to configure access rights.
    - Created for: UC9.1
- void updateAccessRights()
    - Effect: Updates the access rights on a certain pluggable device for a group of customer organisations.
    - Created for: UC9.6

7.2.2 AccessRightsMgmt

Provided by: ▣ `AccessRightsManager`
Required by: ▣ `InfrastructureOwnerFacade`
Operations:

- getCustomerOrganisationsRights(**PluggableDeviceID** pID, List<int> custOrgIDs)
    - Effect: Returns a map of AccessRights and the IDs of customer organisations that have those AccessRights on a certain pluggable device.
    - Created for: UC9.4
- void updateAccessRights()
    - Effect: Updates the access rights on a certain pluggable device for a group of customer organisations.
    - Created for: UC9.7

7.2.3 Actuate

Provided by: ▣ `PluggableDevice`
Required by: ▣ `Mote`
Operations:

- void sendActuationCommand(string commandName)
    - Effect: Send an actuation command to the actuator.   Sending an unknown actuation command has no effect.

7.2.4 AppDeviceData

Provided by: ▣ `ApplicationContainerManager`, ▣ `ApplicationFacade`, ▣ `ApplicationManagementLogic`, ▣ `ApplicationManager`, ▣ `BrokerLogic`, ▣ `OnlineServiceBroker`
Required by: ▣ `ApplicationClient`, ▣ `ApplicationContainer`, ▣ `ApplicationContainerManager`, ▣ `ApplicationFacade`, ▣ `BrokerLogic`, ▣ `GatewayBroker`
Operations:

- getDataForDevices(List<**PluggableDeviceID**> devices, **DateTime** from, **DateTime** to)
    - Effect: Returns DeviceData of pluggable devices over a specified time period.
    - Created for: UC24.1
- getDataForRoom(**RoomTopology** room, **DateTime** from, **DateTime** to)
    - Effect: Returns DeviceData of pluggable devices in a room over a specified time period.
    - Created for: UC24.1

- List<**RoomTopology**> getTopologyOverview(int applicationInstanceID, int customerOrganisationID)
    - Effect: Returns a list of RoomTopology containing devices that an ApplicationInstance has access to.
    - Created for: UC25.1

7.2.5   AppDeviceMgmt

Provided by: ▤ BrokerLogic, ▤ DeviceManager, ▤ Gateway, ▤ GatewayBroker
Required by: ▤ ApplicationManagementLogic, ▤ ApplicationManager, ▤ BrokerLogic, ▤ OnlineServiceBroker
Operations:
- bool areEssentialDevicesOperational(int applicationID)
    - Effect: Returns true if all essential devices for the application with id "applicationID" are operational.
    - Created for: UC18
- void setPluggableDevicesRequirements(int applicationID, List<**PluggableDeviceInfo**> devices)
    - Effect: Sets an application's requirements for pluggable devices.
    - Created for: Av3:   "Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices."
    TODO: update this with Relationship type?

7.2.6   AppInstanceMgmt

Provided by: ▤ ApplicationContainer, ▤ ApplicationContainerManager, ▤ BrokerLogic, ▤ GatewayBroker
Required by: ▤ ApplicationContainerManager, ▤ ApplicationManagementLogic, ▤ ApplicationManager, ▤ BrokerLogic, ▤ DeviceManager, ▤ OnlineServiceBroker
Operations:
- void activateApplicationInstance(int applicationInstanceID)
    - Effect: Activates an ApplicationInstance that is running on the gateway.
    - Created for: UC17.3, U2 - easy applications
- void deactivateApplicationInstance(int applicationInstanceID)
    - Effect: Deactivates a running instance of an application.
    - Created for: UC18, Av3:   automatic suspension/reactivation of applications.
- boolean isApplicationInstanceAvailable(int applicationInstanceID)
    - Effect: Returns true if a certain ApplicationInstance is available.
    - Created for: UC27.2

7.2.7   AppMessages

Provided by: ▤ ApplicationContainer, ▤ ApplicationContainerManager, ▤ ApplicationManagementLogic, ▤ ApplicationManager, ▤ BrokerLogic, ▤ BrokerLogic, ▤ GatewayBroker, ▤ OnlineServiceBroker, ▤ RequestStore
Required by: ▤ ApplicationContainer, ▤ ApplicationContainerManager, ▤ ApplicationManagementLogic, ▤ ApplicationManager, ▤ BrokerLogic, ▤ BrokerLogic, ▤ GatewayBroker, ▤ OnlineServiceBroker, ▤ RequestStore
Operations:
- void applicationInstanceFailed(int applicationInstanceID, int onOnlineService)
    - Effect: Lets the other parts of an application know that one of their parts has become unavailable.   This allowd the other parts to possibly run in a degraded mode.   Is called by an ApplicationContainer after its ApplicationInstance has crashed 3 times in a row.
    - Created for: Av1
- void sendCommandToApplicationInstance(int applicationInstanceID, string command, boolean forOnlineService)
    - Effect: Sends a command to an ApplicationInstance that is running on the Online Service or on a Gateway.
    - Created for: UC26.1
- void sendMessageToExternalFrontEnd(string hostName, int port, string message)

– Effect: Sends a message from an ApplicationInstance to an external front end. The ApplicationInstance provides the hostname and port.
– Created for: UC26.1

7.2.8 AppMonitoring

Provided by: ▣ `ApplicationContainer`, ▣ `ApplicationContainerManager`
Required by: ▣ `ApplicationContainerManager`, ▣ `ApplicationContainerMonitor`
Operations:
- **Echo** pingApplicationInstance(int applicationInstanceID)
    – Effect: Sends a ping request to an ApplicationContainer. The ApplicationContainer should respond with an Echo reply. The monitor uses this to check whether or not the container is still available.
    – Created for: The system is able to autonomously detect failures of its individual failing applications.

7.2.9 AppStatus

Provided by: ▣ `ApplicationContainerMonitor`
Required by: ▣ `ApplicationContainer`, ▣ `ApplicationContainerManager`
Operations:
- boolean isApplicationInstanceAvailable(int applicationInstanceID)
    – Effect: Returns true if a certain ApplicationInstance is available.
    – Created for: UC27.2
- void updateApplicationActive(int applicationInstanceID)
    – Effect: Updates an ApplicationInstance's status to 'active'.
- void updateApplicationInstanceSuspended(int applicationInstanceID)
    – Effect: Updates an ApplicationInstance's status to 'suspended'.
- void updateApplicationStatusUnavailable(int applicationInstanceID)
    – Effect: Effect: Notifies the ApplicationContainerMonitor that an ApplicationInstance has crashed.
    – Created for: Av2 - The system is able to autonomously detect failing applications

7.2.10 Commands

Provided by: ▣ `ApplicationContainerManager`, ▣ `DeviceCommandConstructor`
Required by: ▣ `ApplicationContainerManager`, ▣ `ApplicationManagementLogic`
Operations:
- void sendActuationCommand(List<**PluggableDeviceID**> device, string commandName) throws *UnknownCommandException*
    – Effect: Sends a command for a list of actuators to DeviceCommandConstructor for construction of actuation command messages according to the specific formatting syntax for the given actuators. The given command 'commandName' is a command that application developers use for a group of devices.
    – Created for: UC12 - commands from applications
- void verifyAndConstructConfigurationsForDevice(**PluggableDeviceID** pID, Map<string, string> config) throws *UnknownConfigurationParameterException*
    – Effect: Verifies the configuration parameters for a pluggable device. If the parameters have been successfuly verified, constructs a reconfiguration command according to the specific formatting syntax for the pluggable device and sends it to the DeviceManager if everything is correct.
    – Created for: UC13.2-3

7.2.11 Config

Provided by: ▣ `PluggableDevice`
Required by: ▣ `Mote`

Operations:
- Map<String, String> getConfig()
    - Effect: Returns the current configuration of a pluggable device as a parameter-value map.
- boolean setConfig()
    - Effect: Set the given configuration parameters of the pluggable device to the given values. Setting unknown parameters on a pluggable device (e.g., 'noise threshold' -> '3' on a light sensor) has no effect.
    - Created for: Given constraint, UC11: pluggable device needs to be initialised, M1: pluggable device must be able to be initialised

7.2.12 DataConversion

Provided by: ▣ DeviceDataConverter
Required by: ▣ ApplicationContainer
Operations:
- **DeviceData** convert(**DeviceData** data, string targetType)
    - Effect: Converts pluggable device data into other pluggable device data that contains the same information in a different measurement type.
    - Created for: M1: data processing subsystem should be extended with relevant data conversions

7.2.13 DBAccessRightsMgmt

Provided by: ▣ DeviceDB
Required by: ▣ AccessRightsManager
Operations:
- getCustomerOrganisationsRights(**PluggableDeviceID** pID, List<int> custOrgIDs)
    - Effect: Returns a map of AccessRights and the IDs of customer organisations that have those AccessRights on a certain pluggable device.
    - Created for: UC9.4
- void updateAccessRights()
    - Effect: Updates the access rights on a certain pluggable device for a group of customer organisations.
    - Created for: UC9.7

7.2.14 DBAppDeviceMgmt

Provided by: ▣ DeviceDB
Required by: ▣ ApplicationManager, ▣ DeviceCommandConstructor
Operations:
- Map<**PluggableDeviceID**, string> getFormattingSyntaxForDevices(List<**PluggableDeviceID**> devices)
    - Effect: Retrieves the specific formatting syntax for a list of pluggable devices.
    - Created for: UC12.2-3
- Map<**PluggableDeviceID**, **GatewayInfo**> getGatewaysForDevices(List<**PluggableDeviceID**> devices)
    - Effect: Returns a map of PluggableDevices with the gateways they are connected to.
    - Created for: UC12.2
- string getPossibleConfigurationParametersForDevice(**PluggableDeviceID** pID)
    - Effect: Retrieves the possible configuration parameters for a pluggable device.
    - Created for: UC13.2

7.2.15 DBAppMgmt

Provided by: ▣ OtherDataDB
Required by: ▣ ApplicationManagementLogic, ▣ ApplicationManager
Operations:
- void activateApplication(int applicationInstanceID, string status)

- – Effect: Sets an ApplicationInstance's status in the OtherDataDB to 'active'.
  - – Created for: UC17.4, U2 - easy applications
- int createNewApplicationInstance(int custOrgID, int applicationID)
  - – Effect: Creates a new ApplicationInstance for an application for a customer organisation and returns its id.
  - – Created for: UC19.4, U2 - easy applications
- **GatewayInfo** getApplicationInstanceGateway()
  - – Effect: Returns information about a gateway that an application instance is running on.
  - – Created for: Av2 - Need to know which gateways to send messages to for e.g. application commands.
- List<**Application**> getApplications()
  - – Effect: Returns a list of applications in the system.
  - – Created for: UC19.2, U2 - easy applications
- List<int> getApplicationsForDevice()
  - – Effect: Returns a list of applications that can use the device with id "pID".
  - – Created for: UC11: the system looks up the list of applications that use the pluggable device
- List<**PluggableDeviceID**> getDevicesForApplication(int applicationInstanceID)
  - – Effect: Returns a list of PluggableDeviceID of pluggable devices that an ApplicationInstance can use.
  - – Created for: UC17.2, U2 - easy applications
- string getInstallationInstructions(int applicationID)
  - – Effect: Returns the installation instructions of a certain application. If there are no installation instructions set, returns an empty string.
  - – Created for: UC17.6, U2 - easy applications
- List<**RoomTopology**> getNecessaryDevicesAndTopologyConfigurations(int applicationID)
  - – Effect: Returns a list of RoomTopology which is a minimal requirement for a certain application to run. This can used to display the requirements to a user or to check if requirements are fulfilled.
  - – Created for: UC19.5, U2 - easy applications
- void updateApplication(**ApplicationInstance** instance)
  - – Effect: Updates an application in the database (e.g. change state to 'inactive').
  - – Created for: UC18, Av3: automatic suspension/reactivation of applications.
- void updateApplicationDevicesSettings(int applicationInstanceID, List<**PluggableDeviceID**> devices, List<**Relationship**> relationships)
  - – Effect: Updates an ApplicationInstance's device settings. This includes which devices the instance can use and which relationships exist between those devices.
  - – Created for: UC19.6, U2 - easy applications
- void updateApplicationInstanceGateway(int applicationInstanceID, int gatewayID)
  - – Effect: Updates the gateway on which an application instance is running.
  - – Created for: Av2 - Need to know which gateways to send messages to for e.g. application commands.
- void updateCriticality(int applicationInstanceID, int isCritical)
  - – Effect: Updates the criticality of an ApplicationInstance.
  - – Created for: UC19.11, U2 - easy applications
- void updateSubscription(**Subscription** subscription)
  - – Effect: Updates a subscription in the database (e.g. change state to 'disabled').
  - – Created for: UC18

7.2.16 DBDeviceData

Provided by: ⌻ PluggableDeviceDataDB
Required by: ⌻ DeviceDataScheduler
Operations:
- List<**DeviceData**> getData(**PluggableDeviceID** pID, **DateTime** from, **DateTime** to)
  - – Effect: Returns data from a specific device in a certain time period.

– Created for: P2:   lookup queries
- getDataForDevices(List<**PluggableDeviceID**> devices, **DateTime** from, **DateTime** to)
    – Effect: Returns DeviceData of pluggable devices over a specified time period.
    – Created for: UC24.2
- void rcvData(**PluggableDeviceID** pID, **DeviceData** data)
    – Effect: Sends pluggable device data to the DB to be stored.
    – Created for: UC11, P2:   storing new pluggable data

7.2.17   DBDeviceMgmt

Provided by: ⌹ BrokerLogic, ⌹ DeviceDB, ⌹ OnlineServiceBroker
Required by: ⌹ BrokerLogic, ⌹ DeviceManager, ⌹ GatewayBroker, ⌹ GatewayBrokerMonitor
Operations:
- void addDevice(**PluggableDeviceID** pID, **PluggableDeviceType** type, Map<string, string> configurations, int moteID)
    – Effect: Adds a new pluggable device in the DeviceDB and adds a reference to a mote.   The device's status is 'uninitialised' by default and it's current configurations (which are now the default configurations) are stored as well.   If the device already exists, removes the data first (in case the device is plugged into a different mote or on a different network).
    – Created for: UC6.3, U2 - easy pluggable device installation
- int addMote(**MoteInfo** mote, int gatewayID, **IPAddress** moteIPAddress)
    – Effect: Adds a new mote in the DeviceDB along with an IP address and a reference to a gateway. Returns the DB id for the mote.
    – Created for: UC4.3, U2 - easy mote installation
- Map<string, string> getConfigDB(**PluggableDeviceID** pID)
    – Effect: Gets the last set configurations of a pluggable device from the DeviceDB.
    – Created for: UC6.3 - reintroduced device
- Map<string, string> getPluggableDeviceData(**PluggableDeviceID** pID)
    – Effect: Effect: Returns all data about a pluggable device in the DeviceDB, such as status, type information, specific formatting syntax (for applications), etc.
    – Created for: UC12 - construction of commands, UC13 - verification of configuration parameters
- void reactivateDevice(**PluggableDeviceID** pID)
    – Effect: Changes the status of a pluggable device to 'active'.
    – Created for: UC6.3 - reintroduced device
- void reactivateMote(int moteID)
    – Effect: Changes the status of the mote with DB id 'moteID' to 'active'.
    – Created for: U2 - Reintroducing a previously known mote should not require any con guration.
- void registerGateway(int gatewayID, **IPAddress** address)
    – Effect: Sets a gateway's status to 'active' and updates its IP address.
    – Created for: U2 - gateway installation

7.2.18   DBInvoiceMgmt

Provided by: ⌹ OtherDataDB
Required by: ⌹ InvoiceManager
Operations:
- void markActivatedApplication(int applicationInstanceID, int custOrgID, **DateTime** date)
    – Effect: Updates an ApplicationInstance's billing information: marks the start of a billing period.
    – Created for: UC17.4, U2 - easy applications

7.2.19   DBIODeviceMgmt

Provided by: ⌹ DeviceDB

Required by: OtherDataDB   InfrastructureOwnerManager
Operations:
- List<**PluggableDeviceInfo**> getDevices(int infrastructureOwnerID)
  - Effect: Returns a list of PluggableDeviceInfo of devices owned by an infrastructure owner.
  - Created for: UC9.2

7.2.20 DBIOMgmt

Provided by: OtherDataDB
Required by: InfrastructureOwnerManager
Operations:
- List<int> getCustomerOrganisations(int infrastructureOwnerID)
  - Effect: Returns a list of IDs of all customer organisations associated with an infrastructure owner.
  - Created for: UC9.4

7.2.21 DBNotificationMgmt

Provided by: OtherDataDB
Required by: NotificationHandler
Operations:
- int lookupNotificationChannelForUser()
  - Effect: Returns the id of the type of communication channel a user prefers.
  - Created for: UC15
- int storeNotification(**Notification** notification)
  - Effect: Stores a new notification entry in the database. Returns the id of the new notification.

  - Created for: UC15, Av3: notifications
- int updateNotification(**Notification** notification)
  - Effect: Updates an existing notification (e.g. change status to "sent").
  - Created for: UC15

7.2.22 DBSubscriptionMgmt

Provided by: OtherDataDB
Required by: SubscriptionManager
Operations:
- void createSubscription(int custOrgID, int applicationInstanceID)
  - Effect: Creates a subscription for a customer organisation to an ApplicationInstance. If the customer organisation is already subscribed to an older version of the the application, then the organisation is unsubscribed from that earlier version.
  - Created for: UC19.12-13, U2 - easy applications
- List<**Subscription**> getSubscriptions(int custOrgID)
  - Effect: Returns a list of subscriptions a customer organisation has.
  - Created for: UC19.2, U2 - easy applications

7.2.23 DBTopologyMgmt

Provided by: DeviceDB
Required by: TopologyManager
Operations:
- void addDevice(**PluggableDeviceID** pID, int moteID)
  - Effect: Adds a new pluggable device to the topology of the infrastructure owner and links it to a mote. The device gets the mote's location by default. If the device is already linked to another mote, overwrites that link.

- – Created for: UC6.3, U2 - easy pluggable device installation
- void addMote(int moteID, int infrastructureOwnerID, int gatewayID)
  - – Effect: Adds a new mote to a topology of an infrastructure owner. The mote is linked to a gateway and gets status 'unplaced' by default.
  - – Created for: UC4.3, U2 - easy mote installation
- boolean arePluggableDevicesPlaced(List<**PluggableDeviceID**> devices)
  - – Effect: Returns true if all pluggable devices in the given list have status 'placed' in the topology.
  - – Created for: UC17.2, U2 - easy applications
- List<**RoomTopology**> getTopology(int custOrgID)
  - – Effect: Returns a list of RoomTopology associated to a customer organisation.
  - – Created for: UC19.5, U2 - easy applications
- void reactivateDevice(**PluggableDeviceID** id)
  - – Effect: Changes the status of a pluggable device in the topology to 'placed'.
  - – Created for: UC6.3 - reintroduced device
- void reactivateMote(int moteID)
  - – Effect: Changes the status of the mote in the topology to 'placed'. The location of the mote is unchanged, it has already been set.
  - – Created for: U2 - Reintroducing a previously known mote should not require any con
    guration.

7.2.24 DBUserRoleMgmt

Provided by: ⧉ OtherDataDB
Required by: ⧉ UserRolesManager
Operations:
- boolean areMandatoryUserRolesAssigned(int applicationInstanceID)
  - – Effect: Returns true if all mandatory UserRoles for the application have been assigned to users. Finds the relevant customer organisations through the ApplicationInstance.
  - – Created for: UC17.1, U2 - easy applications
- List<**User**> getEndUsers(int custOrgID)
  - – Effect: Returns a list of Users which are associated to a customer organisation.
  - – Created for: UC19.8, U2 - easy applications
- List<**UserRole**> getMandatoryUserRoles(int applicationID)
  - – Effect: Returns a list of UserRoles that which need to be assigned in order for an ApplicationInstance to run.
  - – Created for: UC19.7, U2 - easy applications
- List<**UserRole**> getOptionalUserRoles(int applicationID)
  - – Effect: Returns a list of UserRoles which can optionally be assigned for an ApplicationInstance.
  - – Created for: UC19.7, U2 - easy applications
- List<**User**> getUsersWithRoles(int applicationInstanceID)
  - – Effect: Returns a list of Users associated to an ApplicationInstance that were assigned UserRoles.
  - – Created for: UC17.6, U2 - easy applications
- void updateUserRoles(int applicationInstanceID, Map<int, int> usersAndRoles)
  - – Effect: Updates the UserRoles assigned to Users for a certain ApplicationInstance. 'usersAndRoles' maps User IDs to UserRole IDs.
  - – Created for: UC19.9, U2 - easy applications

7.2.25 DeliveryMgmt

Provided by: ⧉ NotificationHandler
Required by: ⧉ NotificationDeliveryService
Operations:
- void acknowledgement(int notificationID)

- Effect: Sends an acknowledgement to the system for a certain notification to denote that a notification has been received.
- Created for: UC15

7.2.26 DeviceCommands

Provided by: ▣ BrokerLogic, ▣ DeviceManager, ▣ GatewayBroker
Required by: ▣ ApplicationManager, ▣ BrokerLogic, ▣ DeviceCommandConstructor, ▣ OnlineServiceBroker
Operations:
- Map<**PluggableDeviceID**, string> getFormattingSyntaxForDevices(List<**PluggableDeviceID**> devices)
  - Effect: Retrieves the specific formatting syntax for a list of pluggable devices from the DeviceManager. Used by the DeviceCommandConstructor on Gateways.
  - Created for: UC12 - commands from applications on gateways
- string getPossibleConfigurationParametersForDevice(**PluggableDeviceID** pID)
  - Effect: Retrieves the possible configuration parameters for a pluggable device. Used by the DeviceCommandConstructor on Gateways.
  - Created for: UC13 - configuration commands from applications on gateways
- void sendActuationCommand(Map<**PluggableDeviceID**, string> commandsForDevices)
  - Effect: Sends correctly constructed actuation commands for a group of actuators to the DeviceManager on a Gateway.
  - Created for: UC12 - send command message to the intended actuators
- void setConfiguration(**PluggableDeviceID** pID, Map<string, string> config)
  - Effect: Sets configuration parameters of a pluggable device. The DeviceManager first determines whether the pluggable device needs to be reconfigured. To do this, it checks the data it has about configurations set by other applications on the pluggable device. If the device can be reconfigured, then the configuration command is propagated to the pluggable device.
  - Created for: UC13.3

7.2.27 DeviceCommands

Provided by: ▣ ApplicationContainerManager, ▣ ApplicationFacade, ▣ ApplicationManagementLogic, ▣ ApplicationManager
Required by: ▣ ApplicationClient, ▣ ApplicationContainer, ▣ ApplicationFacade
Operations:
- void sendActuationCommand(List<**PluggableDeviceID**> devices, string command)
  - Effect: Sends a command for a list of actuators for construction of the actuation command messages.
  - Created for: UC12.1
- void setConfiguration(**PluggableDeviceID** pID, Map<string, string> config)
  - Effect: Sends a configuration command for a pluggable device for construction of the actual command messages.
  - Created for: UC13.1

7.2.28 DeviceData

Provided by: ▣ Mote
Required by: ▣ PluggableDevice
Operations:
- void rcvData()
  - Effect: Propagates pluggable device data to the connected gateway by calling rcvData on the gateway. (Initiated by the device).
  - Created for: UC11, P2: storing new pluggable data
- void rcvDataCallback(**PluggableDeviceID** pID, **DeviceData** data, int requestID)
  - Effect: Propagates pluggable device data to the connected gateway by calling rcvData on the gateway. (Callback of getDataAsync).

– Created for: UC11, P2: storing new pluggable data

7.2.29 DeviceData

Provided by: ▣ BrokerLogic, ▣ DeviceDataScheduler, ▣ OnlineServiceBroker, ▣ RequestStore
Required by: ▣ BrokerLogic, ▣ DeviceManager, ▣ Gateway, ▣ GatewayBroker, ▣ RequestStore
Operations:
- void rcvData(**PluggableDeviceID** pID, **DeviceData** data)
  - Effect: Sends pluggable device data to the scheduler to be processed.
  - Created for: UC11, P2: storing new pluggable data

7.2.30 DeviceData

Provided by: ▣ DeviceManager, ▣ Gateway
Required by: ▣ Mote
Operations:
- void rcvData(**PluggableDeviceID** pID, **DeviceData** data)
  - Effect: Provides pluggable device data to the gateway (Initiated by the device).
- void rcvDataCallback(**PluggableDeviceID** pID, **DeviceData** data, int requestID)
  - Effect: Provides device data to the gateway (Callback of getDataAsync).

7.2.31 DeviceMgmt

Provided by: ▣ Mote
Required by: ▣ DeviceManager, ▣ Gateway
Operations:
- List<**PluggableDeviceInfo**> getConnectedDevices()
  - Effect: Effect: Returns a list of information about devices that are connected to the mote.
  - Created for: UC18
  - Tradeoff: send PluggableDeviceID instead of DeviceInfo. If you send DeviceInfo, then ApplicationManager does not have to fetch this info. If you send PluggableDeviceID's, then less bandwidth is used and the Gateways do less work.
- void sendActuationCommand(**PluggableDeviceID** device, string commandName)
  - Effect: Sends an actuation command from to Mote for a pluggable device.
  - Created for: UC12.3
- void setConfig(**PluggableDeviceID** pID, Map<String, String> config)
  - Effect: Set the given configuration parameters of a PluggableDevice to the given values. Setting unknown parameters on a PluggableDevice has no effect.

  - Created for: UC11: pluggable device needs to be initialised, M1: pluggable device must be able to be initialised

7.2.32 DeviceMgmt

Provided by: ▣ DeviceManager
Required by: ▣ Gateway, ▣ Mote
Operations:
- bool areEssentialDevicesOperational(int applicationID)
  - Effect: Returns true if all essential devices for the application with id "applicationID" are operational.
  - Created for: UC18
- void heartbeat(int moteID, List<**PluggableDeviceInfo**> devicesmeter)
  - Effect: Sends a heartbeat from a mote to check/update timers for operational devices.
  - Created for: UC14, Av3: failure detection

- bool isDeviceInitialised(**PluggableDeviceID** pID)
  – Effect: Returns true if the device with id "pID" has been initialized.
  – Created for: UC11:   pluggable device needs to be initialised, M1:   pluggable device must be able to be initialised
  – TODO: need this check?   is 'initialized' status stored in DB or on gateways?   or both?
- void pluggableDevicePluggedIn(**MoteInfo** mInfo, **PluggableDeviceID** pID, **PluggableDeviceType** type)
  – Effect: Notify the gateway that a new PluggableDevice of the given type is connected to the mote.
- void pluggableDeviceRemoved(**PluggableDeviceID** pID)
  – Effect: Notify the gateway that a PluggableDevice is removed.
- void setPluggableDevicesRequirements(int applicationID, List<**PluggableDeviceInfo**> devices)
  – Effect: Sets an application's requirements for pluggable devices.
  – Created for: Av3:   "Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices."

7.2.33   ForwardData

Provided by: 🔲 `ApplicationManagementLogic`, 🔲 `ApplicationManager`
Required by: 🔲 `DeviceDataScheduler`
Operations:
- List<int> getApplicationsForDevice(**PluggableDeviceID** pID)
  – Effect: Returns a list of application instances that can use the device with id "pID".

  – Created for: UC11:   the system looks up the list of applications that use the pluggable device
- void rcvData(**PluggableDeviceID** pID, **DeviceData** data)
  – Effect: Sends pluggable device data to an application that wants to use it

  – Created for: UC11:   system relays data to applications

7.2.34   FrontEndAppRequests

Provided by: 🔲 `ApplicationManagementLogic`, 🔲 `ApplicationManager`
Required by: 🔲 `CustomerOrganisationFacade`, 🔲 `InfrastructureOwnerFacade`, 🔲 `SubscriptionManager`
Operations:
- void activateApplication(int applicationInstanceID)
  – Effect: Checks and activates an ApplicationInstance (UC17).
  – Created for: UC19.14, U2 - easy applications
- void checkApplicationsForActivationForCustomerOrganisations(List<int> custOrgIDs)
  – Effect: Checks and activates 'inactive' ApplicationInstances which can now execute again for a list of customer organisations.
  – Created for: UC9.7
- void checkApplicationsForDeactivationForCustomerOrganisations(List<int> custOrgIDs)
  – Effect: Checks for ApplicationInstances that require deactivation for a list of customer organisations.
  – Created for: UC9.7
- int createNewApplicationInstance(int custOrgID, int applicationID)
  – Effect: Creates a new ApplicationInstance for an application for a customer organisation and returns its id.
  – Created for: UC19.4, U2 - easy applications
- List<**Application**> getApplications()
  – Effect: Returns a list of applications in the system.
  – Created for: UC19.2, U2 - easy applications
- List<**RoomTopology**> getNecessaryDevicesAndTopologyConfigurations(int applicationID)
  – Effect: Returns a list of RoomTopology which is a minimal requirement for a certain application to run.   This can used to display the requirements to a user or to check if requirements are fulfilled.
  – Created for: UC19.5, U2 - easy applications

42

- void updateApplicationDevicesSettings(int applicationInstanceID, List<**PluggableDeviceID**> devices, List<**Relationship**> relationships)
  - Effect: Updates an ApplicationInstance's device settings.   This includes which devices the instance can use and which relationships exist between those devices.
  - Created for: UC19.6, U2 - easy applications
- void updateCriticality(int applicationInstanceID, boolean isCritical)
  - Effect: Updates the criticality of an ApplicationInstance.
  - Created for: UC19.11, U2 - easy applications

7.2.35   GatewayUpdates

Provided by: ⎙ GatewayMonitor
Required by: ⎙ BrokerLogic
Operations:

7.2.36   GWAppInstanceMgmt

Provided by: ⎙ ApplicationManagementLogic, ⎙ ApplicationManager, ⎙ BrokerLogic, ⎙ OnlineServiceBroker
Required by: ⎙ BrokerLogic, ⎙ DeviceManager, ⎙ GatewayBroker
Operations:
- void activateApplicationInstance(int applicationInstanceID)
  - Effect: Activates a new instance of an application.
  - Created for: UC18, Av3:   automatic suspension/reactivation of applications.
- void checkApplicationsForActivationForInfrastructureOwner(int infrastructureOwnerID)
  - Effect: Checks and activates applications which can now execute again.   The applications checked are those that are subscribed to by customers organisations associated to the given infrastructure owner.
  - Created for: UC17, UC6.3 - reintroduced device
- void deactivateApplicationInstance(int applicationInstanceID)
  - Effect: Deactivates a running instance of an application.
  - Created for: UC18, Av3:   automatic suspension/reactivation of applications.

7.2.37   GWBrokerMonitoring

Provided by: ⎙ GatewayBroker
Required by: ⎙ GatewayBrokerMonitor
Operations:

7.2.38   GWCommunication

Provided by: ⎙ BrokerLogic, ⎙ GatewayBroker
Required by: ⎙ BrokerLogic, ⎙ Gateway, ⎙ OnlineServiceBroker
Operations:

7.2.39   Heartbeat

Provided by: ⎙ DeviceManager, ⎙ Gateway
Required by: ⎙ Mote
Operations:
- void heartbeat(Map<string, string> moteInfo, List<Tuple<**PluggableDeviceID**, **PluggableDeviceType**>> pds)
  - Effect: Sends a heartbeat from a mote to a gateway, including a list of the pluggable devices and their device types (i.e.   those currently plugged into the mote)
  - Created for: Given constraint, UC14, Av3:   failure detection

7.2.40 InvoiceMgmt

Provided by: ⊞ InvoiceManager
Required by: ⊞ ApplicationManagementLogic, ⊞ ApplicationManager
Operations:
- void markActivatedApplication(int applicationInstanceID, int custOrgID, **DateTime** date)
  - Effect: Updates an ApplicationInstance's billing information: marks the start of a billing period.
  - Created for: UC17.4, U2 - easy applications

7.2.41 IOMgmt

Provided by: ⊞ InfrastructureOwnerManager
Required by: ⊞ InfrastructureOwnerFacade
Operations:
- List<int> getCustomerOrganisations(int infrastructureOwnerID)
  - Effect: Returns a list of IDs of all customer organisations associated with an infrastructure owner.
  - Created for: UC9.4
- List<**PluggableDeviceInfo**> getDevices(int infrastructureOwnerID)
  - Effect: Returns a list of PluggableDeviceInfo of devices owned by an infrastructure owner.
  - Created for: UC9.2

7.2.42 Monitoring

Provided by: ⊞ ApplicationContainerManager, ⊞ ApplicationContainerMonitor
Required by: ⊞ ApplicationExecutionSubsystemMonitor
Operations:
- **Echo** ping()
  - Effect: Sends a ping request to a component. The component should respond with an Echo reply. The monitor uses this to check whether or not the component is still available.
  - Created for: The system is able to autonomously detect failures of its individual application execution components

7.2.43 NotificationDeliveryMgmt

Provided by: ⊞ NotificationDeliveryService
Required by: ⊞ NotificationHandler
Operations:
- void notify(Map<string, string> data)
  - Effect: Delivers a notification to an end user using a specific delivery service.
  - Created for: UC15

7.2.44 Notify

Provided by: ⊞ BrokerLogic, ⊞ NotificationHandler, ⊞ OnlineServiceBroker
Required by: ⊞ ApplicationContainerManager, ⊞ ApplicationContainerMonitor, ⊞ ApplicationExecutionSubsystemMo
    ⊞ ApplicationManagementLogic, ⊞ ApplicationManager, ⊞ BrokerLogic, ⊞ DeviceManager,
    ⊞ GatewayBroker, ⊞ GatewayBrokerMonitor
Operations:
- void notify(int userID, string message)
  - Effect: Stores a new notification in the system and causes it to be sent to a user.
  - Created for: UC14, Av3: notifications
- void notifySystemAdministrator(string message)
  - Effect: Stores a new notification in the system and causes it to be sent to a SIoTIP system administrator.

7.2.45   OSBrokerMonitoring

Provided by: ▣ `OnlineServiceBroker`
Required by: ▣ `OnlineServiceBrokerMonitor`
Operations:

7.2.46   OSCommunication

Provided by: ▣ `BrokerLogic`, ▣ `Gateway`, ▣ `OnlineServiceBroker`
Required by: ▣ `BrokerLogic`, ▣ `GatewayBroker`
Operations:

7.2.47   OSMonitoring

Provided by: ▣ `BrokerLogic`
Required by: ▣ `OnlineServiceMonitor`
Operations:

7.2.48   OSUpdates

Provided by: ▣ `OnlineServiceMonitor`
Required by: ▣ `BrokerLogic`
Operations:

7.2.49   RequestData

Provided by: ▣ `PluggableDevice`
Required by: ▣ `Mote`
Operations:
- **DeviceData** getData()
  - Effect: Synchronously retrieve the device data of a device.

- void getDataAsync(int requestID)
  - Effect: Asynchronously retrieve the device data of a device (by calling rcvDataCallback).

7.2.50   RequestData

Provided by: ▣ `DeviceDataScheduler`
Required by: ▣ `ApplicationManagementLogic`, ▣ `ApplicationManager`
Operations:
- List<**DeviceData**> getData(**PluggableDeviceID** pID, **DateTime** from, **DateTime** to)
  - Effect: Requests data from a specific device in a certain time period.
  - Created for: P2:   requests from applications
- getDataForDevices(List<**PluggableDeviceID**> devices, **DateTime** from, **DateTime** to)
  - Effect: Returns DeviceData of pluggable devices over a specified time period.
  - Created for: UC24.2

7.2.51   Requests

Provided by: ▣ `RequestStore`
Required by: ▣ `BrokerLogic`
Operations:

7.2.52 RoleMgmt

Provided by: ⊞ UserRolesManager
Required by: ⊞ ApplicationManagementLogic, ⊞ ApplicationManager, ⊞ CustomerOrganisationFacade
Operations:

- boolean areMandatoryUserRolesAssigned(int applicationInstanceID)
  - Effect: Returns true if all mandatory UserRoles for the application have been assigned to users. Finds the relevant customer organisations through the ApplicationInstance.
  - Created for: UC17.1, U2 - easy applications
- List<**User**> getEndUsers(int custOrgID)
  - Effect: Returns a list of Users which are associated to a customer organisation.
  - Created for: UC19.8, U2 - easy applications
- List<**UserRole**> getMandatoryUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles that which need to be assigned in order for an ApplicationInstance to run.
  - Created for: UC19.7, U2 - easy applications
- List<**UserRole**> getOptionalUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles which can optionally be assigned for an ApplicationInstance.
  - Created for: UC19.7, U2 - easy applications
- List<**User**> getUsersWithRoles(int applicationInstanceID)
  - Effect: Returns a list of Users associated to an ApplicationInstance that were assigned UserRoles.
  - Created for: UC17.6, U2 - easy applications
- void updateUserRoles(int applicationInstanceID, Map<int, int> usersAndRoles)
  - Effect: Updates the UserRoles assigned to Users for a certain ApplicationInstance. 'usersAndRoles' maps User IDs to UserRole IDs.
  - Created for: UC19.9, U2 - easy applications

7.2.53 SubscriptionMgmt

Provided by: ⊞ CustomerOrganisationFacade
Required by: ⊞ CustomerOgranisationClient
Operations:

- Map<**Application**, **Subscription**> getApplicationsToSubscribe(int custOrgID)
  - Effect: Returns a map of Applications and Subscriptions a given customer organisation has to those applications
  - Created for: UC19.1, U2 - easy applications
- int subscribeToApplication(int custOrgID, int applicationID)
  - Effect: Creates a new ApplicationInstance for an application for a customer organisation and returns its id.
  - Created for: UC19.4, U2 - easy applications
- void updateApplicationDevicesSettings(int applicationInstanceID, List<**PluggableDeviceID**> devices, List<**Relationship**> relationships)
  - Effect: Updates an ApplicationInstance's device settings.   This includes which devices the instance can use and which relationships exist between those devices.
  - Created for: UC19.6, U2 - easy applications
- void updateCriticality(int applicationInstanceID, boolean isCritical)
  - Effect: Updates the criticality of an ApplicationInstance.
  - Created for: UC19.11, U2 - easy applications
- void updateUserRoles(int applicationInstanceID, Map<int, int> usersAndRoles)
  - Effect: Updates the UserRoles assigned to Users for a certain ApplicationInstance. 'usersAndRoles' maps User IDs to UserRole IDs.
  - Created for: UC19.9, U2 - easy applications

7.2.54 SubscriptionMgmt

Provided by: ▣ SubscriptionManager
Required by: ▣ CustomerOrganisationFacade
Operations:
- void createSubscription(int custOrgID, int applicationInstanceID)
  - Effect: Creates a subscription for a customer organisation to an ApplicationInstance.   If the customer organisation is already subscribed to an older version of the the application, then the organisation is unsubscribed from that earlier version.
  - Created for: UC19.12-13, U2 - easy applications
- Map<**Application**, **Subscription**> getApplicationsToSubscribe(int custOrgID)
  - Effect: Returns a map of Applications and Subscriptions a given customer organisation has to those applications
  - Created for: UC19.2, U2 - easy applications

7.2.55 TopologyMgmt

Provided by: ▣ BrokerLogic, ▣ OnlineServiceBroker, ▣ TopologyManager
Required by: ▣ ApplicationManagementLogic, ▣ ApplicationManager, ▣ BrokerLogic, ▣ CustomerOrganisationFacade, ▣ DeviceManager, ▣ GatewayBroker
Operations:
- void addDevice(**PluggableDeviceID** id, int moteID)
  - Effect: Adds a new pluggable device to the topology of the infrastructure owner and links it to a mote.   The device gets the mote's location by default.   If the device is already linked to another mote, overwrites that link.
  - Created for: UC6.3, U2 - easy pluggable device installation
- void addMote(int moteID, int gatewayID, int infrastructureOwnerID)
  - Effect: Adds a new mote to a topology of an infrastructure owner.   The mote is linked to a gateway and gets status 'unplaced' by default.
  - Created for: UC4.3, U2 - easy mote installation
- boolean arePluggableDevicesPlaced(List<**PluggableDeviceID**> devices)
  - Effect: Returns true if all pluggable devices in the given list have status 'placed' in the topology.
  - Created for: UC17.2, U2 - easy applications
- List<**RoomTopology**> getTopology(int custOrgID)
  - Effect: Returns a list of RoomTopology associated to a customer organisation.
  - Created for: UC19.5, U2 - easy applications
- void reactivateDevice(**PluggableDeviceID** id)
  - Effect: Changes the status of a pluggable device in the topology to 'placed'.
  - Created for: UC6.3 - reintroduced device
- void reactivateMote(int moteID)
  - Effect: Changes the status of the mote in the topology to 'placed'.   The location of the mote is unchanged, it has already been set.
  - Created for: U2 - Reintroducing a previously known mote should not require any con guration.

7.3 Exceptions

- *UnknownCommandException* Thrown if a command is sent by an application that is not recognised.
- *UnknownConfigurationParameterException* Thrown if an application tries to set a configuration parameter on a device that is not recognised.

7.4 Data types

- **Application**:
  Contains information about an application uploaded to the SIoTIP system.
- **ApplicationInstance**:
  Attributes: int id, int status, int customerOrganisationID, boolean isGatewayVersion
  Contains information on an application instance. When an ApplicationInstance is running on the Online Service and on a Gateway, the ApplicationIntances in the ApplicationContainers share the same id.
- **DateTime**:
  Represents an instant in time, expressed as a date and time of day.
- **DeviceData**:
  Data from a pluggable device. For sensors, this contains sensor values. For actuators, this contains the state of the actuator. The data is encapsulated within a JSON message, and should be converted into something meaningful based on the device type of the pluggable device that sent the data.
- **Echo**:
  Is a reply to an echo request (ping). Contains an Identifier and Sequence Number that matches the request that caused the Echo reply.
- **GatewayInfo**:
  Attributes: int gatewayID, **IPAddress** IPAddress, int infrastructureOwnerID
  An object containing information on a Gateway.
- **IPAddress**:
  Represents an Internet Protocol address.
- **MoteInfo**:
  Attributes: int moteID, int manufacturerID, int productID, int batteryLevel
  An object containing information on a Mote. This is a list of key-value pairs. The values depend on the type of mote. For example, only a battery-powered mote would include the batterylevel info.
- **Notification**:
  Attributes: int id, int recipientUserID, string message, int communicationChannelID, int notificationTypeID
  Contains information about a notification. The communicationChannellD represents the communication channel that will be used to send the notification to the user. The notificationTypeID denotes the type of the notification (normal / alarm / ...).
- **PluggableDeviceID**:
  A unique identifier of a pluggable device.
- **PluggableDeviceInfo**:
  Attributes: **PluggableDeviceID** id, **PluggableDeviceType** type, Map<string, string> config
  Contains information on a pluggable device.
- **PluggableDeviceType**:
  Attributes: int manufacturerID, int productID, int type, int measurementUnits
  Denotes the type of a pluggable device. It specifies the manufacturer, a model identifier (if any) of the pluggable (e.g. heat-o sensor 5000), the type of pluggable device (e.g. temperature sensor or power socket actuator) and the used measurement units (e.g. degrees Celsius, degrees Fahrenheit or decibel). Within MicroPnP this information is also used to retrieve and install the correct drivers (this is outside the scope of the system),
  This description is taken from the Discussion Board on Toledo.
- **Relationship**:
  Attributes: **PluggableDeviceID** pID1, **PluggableDeviceID** pID2, string relationship
  Represents a relationship between two devices. e.g. device2 can be used for the purpose that device1 fulfills, device2 is preferred over device1 for a responsibility, etc.
- **RoomTopology**:

Attributes: int roomID, List<**PluggableDeviceID**> devices, **Relationship** relationships, int buildingID

Represents a room in a topology. Contains the room's devices and the relationships between those devices.

- **Subscription**:

  Attributes: int id, int status, int customerOrganisationID, int applicationInstanceID

  Contains data about a subscription by a customer organisation for an application instance. Data about period/length of the subscription is stored in invoices.

- **User**:

  Represents a user in the SIoTIP system.

- **UserRole**:

  Attributes: int userID, int roleID, int applicationInstanceID, int customerOrganisationID

  Contains information on the role that a User fulfills for a certain ApplicationInstance.

A. Attribute-driven design documentation

A.1 Introduction

This chapter contains our ADD log. First, we list the changes we made the ADD process so it fits our workflow better. The remaining part of this chapter is the ADD log. Decompositions 1 and 2 have been changed relative to phase 2a of this project, because we forgot about the given interfaces for gateways and pluggable devices and made up our own (but similar) interfaces instead. The decompositions have been updated to use the given interfaces.

A.2 Adapted ADD process

We left off step a ("Pick an element that needs to be decomposed") since we never really chose a single Element to decompose. Instead, we chose the drivers for each decomposition first and then looked at which elements/subsystems would require changes or which new elements we would need to satisfy those drivers. For component, interfaces, datatypes: we list the new ones, but refer to the plugin exported catalog for descriptions

A.2.1 Decomposition X: DRIVERS (Elements/Subsystem to decompose/expand)

We changed these titles to reflect the architectural drivers we chose first and then denote which elements/subsystems needed changes to satisfy the drivers.

A.2.2 Data type definitions and Interfaces for child modules

For each decomposition, we have listed all new interfaces and data types that we added during the decomposition, but all details have been left out. We used the Visual Paradigm plugin provided by the SA team to generate the element catalog of chapter 6. All details can be found in there.

Also, since an ADD log was no longer a requirement for phase 2b, we have left out intermediary "OtherFunctionality" components and figures of diagrams. This was done to save time.

A.2.3 Verify and refine

We have skipped the verify and refine step because we chose to handle all chosen architectural drivers completely in every decomposition. We did not find this step to be useful after decompositions 1 and 2.

A.3   Decomposition 1: Av3, UC14, UC15, UC18 (SIoTIP System)

A.3.1   Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av3*:   Pluggable device or mote failure

The related functional drivers are:

- *UC14*:   Send heartbeat (Av3)
  This use case checks whether or not motes and pluggable devices are still operational.
- *UC15*:   Send notification (Av3)
  This use case sends a notification to a registered user.
- *UC18*:   Check and deactivate applications (Av3)
  This use case deactivates any application that requires deactivation, because of unavailability of essential pluggable devices or unassigned mandatory roles.

Rationale   Av3 was chosen first since it has high priority and it is more relevant to the core of the system than the other quality requirements with high priority (M1 and U2).   We believe that handling pluggable device failure/connectivity is more important to the whole of the system than M1 and U2, and that handling this first would give a stronger starting point for later ADD iterations than M1 or U2.

A.3.2   Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

Av3: Failure detection   Gateway need to be able to autonomously detect failure of one of its connected motes and pluggable devices.   This is achieved by making motes send heartbeats to their connected gateways. The gateways can then monitor their connected devices.   The heartbeats contain a list of devices that are connected/operational at the moment the mote sends the heartbeat.   Each gateway makes use of a `DeviceManager` component to monitor the devices.   This component uses timers to keep track of how long it has been since a device has sent a heartbeat or occured in a list of connected devices.   Once a timer expires, this is treated as a failure.

A mote has failed when 3 consecutive heartbeats do not arrive within 1 second of their expected arrival time. A pluggable device has failed when it does not occur in a heartbeat of the mote in which it is expected to be in.   This is is detected within 2 seconds after the arrival of the heartbeat.

Av3: Automatic application deactivation and redundancy settings   Applications should be automatically suspended when they can no longer operate due to failure of a pluggable device or mote and reactivated once the failure is resolved.   Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices.
This problem is tackled by the `DeviceManager`.   It stores the requirements for pluggable devices set by applications for all applications that use the gateway that the the `DeviceManager` runs on.   When it detects that an application can no longer operate due to failures, it will send a command to the `ApplicationManager` (via the `GatewayFacade`) to suspend that application.   When the required devices are operational again, the `DeviceManager` detects this and sends a command to reactivate the application.

Applications are suspended within 1 minute after detecting the failure of an essential pluggable device. Application are reactivated within 1 minute after the failure is resolved.

Av3: Notifications The infrastructure owner should be notified of any persistent pluggable device or mote failures.   Customer organisations should be notified if one or more of their applications is suspended or reactivated.   Applications using a failed pluggable device or any device on a failed mote should be notified. The `NotificationHandler` was put in place to deal with notifications.   Other components can use it to generate notifications for certain users in the system.   The `NotificationHandler` will then insert information relevant to the notification in the database (message, status, date and time, source, ...), and use an external delivery service to deliver the notification to users.   The used delivery medium is based on the user's preferences.
Since they are stored in the database, users can always view their notifications via their dashboard.   However, this funcionality is not expanded on in this decomposition yet.

Infrastructure owners are notified within 1 minute after detecting a mote outage lasting at least 10 seconds. Infrastructure owners are notified within 1 minute after the detection of the unavailability of a pluggable device for 30 seconds.
Applications are notified of the failure of relevant pluggable devices within 10 seconds.

Alternatives considered

Av3: Failure detection An alternative would have been to move the `DeviceManager` component from gateways to the Online Service.   This solution would make the gateways do less work, but would be very unscalable.   The reason is that as the customer base (and thus the amount of devices) increases, the Online Service would need to keep track of huge amounts of devices.   This would also flood the network to the Online Service with heartbeats.

Av3: Failure detection Another alternative for failure detection could have been the use of a Ping/Echo mechanism instead of Heartbeats.   Pings could then be used to check if a device is currently operational. However, as a device could not be operational for a moment because of e.g.   interference, timers would still be necessary to keep track of operational devices.   We opted to use heartbeats, as this would reduce the amount of data sent over the network used by the motes, and as motes would have to do slightly more work to process each Ping request in order to generate a reply.

Av3: Notifications Reliable and quick delivery of notifications is crucial to the system in order to solve problems should things go wrong.   Currently, the solution is to use a third party service for delivery of notifications.   In the case that no external services are found satisfactory, or if this dependency on an external service is unwanted, it is possible to build an internal solution for this.   For example, a `NotificationSender` component could make use of the `Factory pattern` for different message channels for different delivery methods (each with their own sendNotification method).   This solution allows us to easily add new message channels in the future with little effort.   The disadvantage of this is that an internal solution takes a lot more time to implement.

A.3.3 Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above.   For each component we note the quality attribute or use case that prompted us to create it.   Descriptions about the components can be found under chapter 6.

- ApplicationManager: Av3
- Database: /
- DeviceManager: Av3
- GatewayFacade: /
- Mote: UC14
- NotificationHandler: UC15

Decomposition Figure A.1 shows the components resulting from the decomposition in this run.
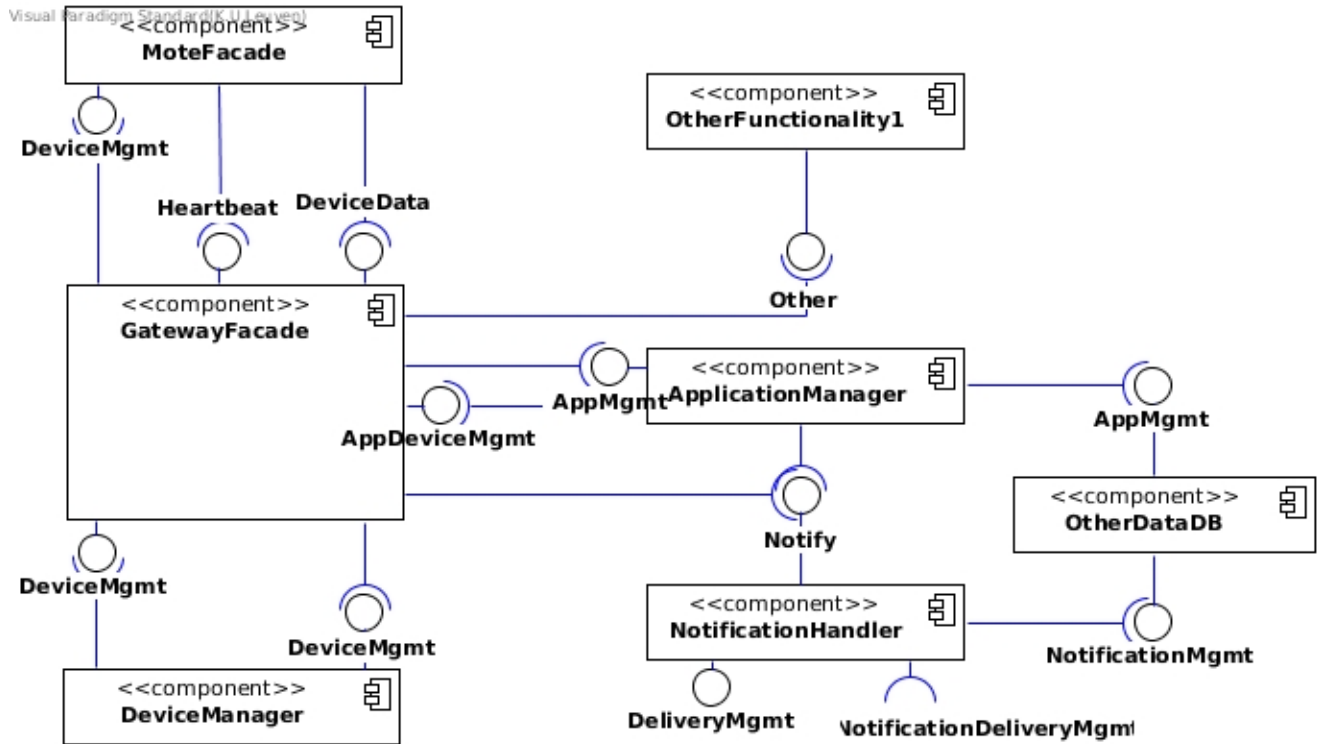
Figure A.1:   Component-and-connector diagram of this decomposition.

Deployment Figure A.2 shows the allocation of components to physical nodes.
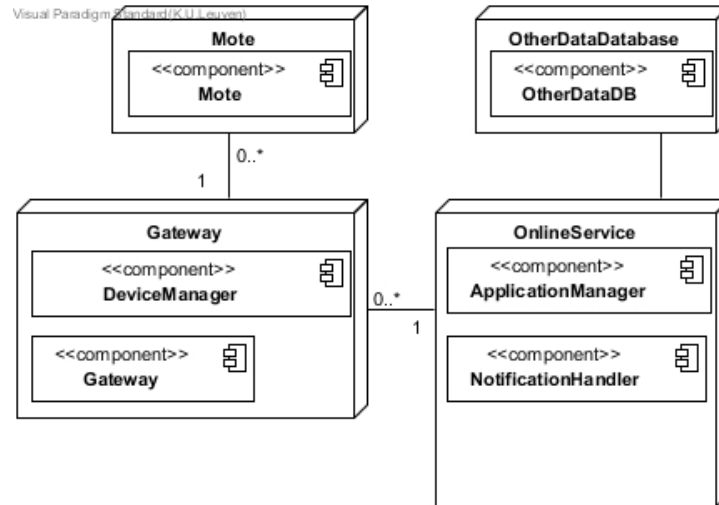
Figure A.2:   Deployment diagram of this decomposition.

A.3.4   Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above.   Detailed information about each interface and its methods can be found under chapter 6.

ApplicationManager

- AppMgmt

Database

- NotificationMgmt
- AppMgmt

GatewayFacade

- Heartbeat
- DeviceData
- DeviceMgmt
- AppDeviceMgmt

Mote

- DeviceMgmt

NotificationHandler

- Notify
- DeliveryMgmt

External notification delivery service

- NotificationDeliveryMgmt

DeviceManager

- DeviceMgmt

A.3.5   Data type definitions

This section lists the data types introduced in this decomposition.

- PluggableDeviceInfo
- Notification
- ApplicationInstance
- Subscription
- PluggableDeviceID
- PluggableDeviceType
- DeviceData
- Map<String,String>

A.3.6   Verify and refine

The selected architectural drivers have been handled completely in this decomposition.   This section describes per component which (parts of) the remaining requirements it is responsible for.   If requirements are split in multiple parts, this is indicated by the addition of a letter (or number, depending on the structure of the requirement) after their title.

ApplicationManager

- *Av2*: Application failure
  Prevention: a, b
  Detection: a, b, c
  Resolution: a, b, c
- *P1*: Large number of users: c
- *M1*: Integrate new sensor or actuator manufacturer: 1.c, 2.a
- *M2*: Big data analytics on pluggable data and/or application usage data: d, e
- *U1*: Application updates: a, b, c, d
- *U2*: Easy Installation: e
- *UC12*: Perform actuation command
- *UC17*: Activate an application: 3, 4

Database

- None

GatewayFacade

- *Av1*: Communication between SIoTIP gateway and Online Service
  Resolution: b, c, d
- *M1*: Integrate new sensor or actuator manufacturer: 1.a, 2.b
- *U2*: Easy Installation: a, c, d
- *UC11*: Send pluggable device data: 1

Mote

- *M1*: Integrate new sensor or actuator manufacturer: 1.a, 2.b
- *U2*: Easy Installation: b, c, d
- *UC04*: Install mote: 1, 2
- *UC05*: Uninstall mote: 1
- *UC06*: Insert a pluggable device into a mote: 2
- *UC07*: Remove a pluggable device from its mote: 2
- *UC11*: Send pluggable device data: 1

NotificationHandler

- *UC16*: Consult notification message: 5
- *UC17*: Activate an application: 5, 6

OtherFunctionality1

- *Av1*: Communication between SIoTIP gateway and Online Service
  Detection: a, b, c, d Resolution: a
- *P1*: Large number of users: a
- *P2*: Requests to the pluggable data database
- *M1*: Integrate new sensor or actuator manufacturer: 1.d
- *M2*: Big data analytics on pluggable data and/or application usage data: a
- *U2*: Easy Installation: e
- *UC01*: Register a customer organisation
- *UC02*: Register an end-user
- *UC03*: Unregister an end user
- *UC04*: Install mote: 3
- *UC05*: Uninstall mote: 2.b

- *UC06*: Insert a pluggable device into a mote: 3: topology part; alternative 3a.1.b
- *UC07*: Remove a pluggable device from its mote: 3.b
- *UC08*: Initialise a pluggable device: 1, 2, 4
- *UC09*: Configure pluggable device access rights
- *UC10*: Consult and configure the topology
- *UC11*: Send pluggable device data: 3
- *UC13*: Configure pluggable device
- *UC16*: Consult notification message: 1, 2, 3, 4
- *UC17*: Activate an application: 1, 2
- *UC19*: Subscribe to application
- *UC20*: Unsubscribe from application
- *UC21*: Send invoice
- *UC22*: Upload an application
- *UC23*: Consult application statistics
- *UC24*: Consult historical data
- *UC25*: Access topology and available devices
- *UC26*: Send application command or message to external front-end
- *UC27*: Receive application command or message to external front-end
- *UC28*: Log in
- *UC29*: Log out

DeviceManager

- *U2*: Easy Installation: c, d
- *UC04*: Install mote: 4
- *UC05*: Uninstall mote: 2
- *UC06*: Insert a pluggable device into a mote: 3: uninitialised part; alternative 3a.1 3a.2 3a.4; 4
- *UC07*: Remove a pluggable device from its mote: 3.a, 3.c
- *UC08*: Initialise a pluggable device: 3
- *UC11*: Send pluggable device data: 2, 3a

A.4 Decomposition 2: M1, P2, UC11 (OtherFunctionality1)

A.4.1 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *M1*:   Integrate new sensor or actuator manufacturer
- *P2*:   Requests to the pluggable data database

The related functional drivers are:

- *UC11*:   Send pluggable device data (P2)
  This use case stores pluggable device data in the pluggable device data storage.   This could be a sensor reading or an actuator status.

Rationale   We chose M1 as it was one of the remaining quality attributes with high priority.   M1's focus on easily introducing new types of devices to the system is very important because of the fast growing market for IoT and development of applications for IoT. Thus, we want to handle this quality attribute before U2 (the other remaining attribute with high priority), as we presume that customer organisations are more interested in using new devices than the effort it takes for infrastructure owners to install the devices.
We also chose P2 because it is strongly related to M1; the whole data flow from devices to storage/applications needs to exist before modifications can even be made.   This combination of M1 and P2 would force us to handle processing and storage of data while making the involved components as simple as possible to modify.

A.4.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

M1: Data conversion   With new types of devices, the pluggable data processing subsystem should be extended with relevant data conversions, e.g.   converting temperature in degrees Fahrenheit to degrees Celsius.
The `DeviceDataConverter` is put in place to handle the task of converting pluggable device data to data of a different type in the system.   This component can easily be modified for new types of data simply by adding a new conversion method for the new.

M1: Usage of new data by applications   The available applications in the system can be updated to use any new pluggable devices.
This is made possible by the RequestData interface provided by `DeviceDataScheduler`.   Data of the new type of device can be requested in the same way as for older devices: by using the device's unique id.   The application manager can get pluggable device data from the `PluggableDeviceDataDB` and return this data to applications in the DeviceData datatype.   This datatype can easily be updated for new types of pluggable devices.

P2: Scheduling   The pluggable data processing subsystem needs to be able to run in normal or overload mode, depending on whether or not the system can process requests within the deadlines given in the quality requirement.   Also, a mechanism should be in place to avoid starvation of any type of request.
The `DeviceDataScheduler` is used to deal with this problem.   It is responsible for scheduling requests that wish to interact with the `PluggableDeviceDataDB`. In normal mode, the system processes incoming requests in a FIFO order.   In overload mode, the requests are given a priority based on what the request is for and what the source of the request is.   The requests are then not simply processed in an order based on their priorities, but an aging technique is to be used such that starvation will be avoided.   Thus, in overload mode, requests are processed in an order based on a combination of the priorities of the requests and the age of the requests.

P2: Pluggable data separation   The processing of (large amounts of) requests concerning pluggable data has no impact on requests concerning other data, e.g.   available applications.

In order to statisfy this constraint, all data directly related to pluggable data has been separated into the `PluggableDeviceDataDB`. All requests concerning pluggable data will be handled by this new component. `PluggableDeviceDataDB` will run on a node different from the node that the `Datbase` component runs on. This way requests concerning pluggable will have no impact on requests concerning other data.

M1: Handling new types of pluggable devices   The new types of sensor or actuator data should be transmitted, processed and stored, and should be made available to applications.   The infrastructure managers must be able to initialize the new type of pluggable device, configure access rights for these devices, and view detailed information about the new type of pluggable device.

The components created thus far have been created with high cohesion in mind so that updating them for new devices would be relatively straightforward.   In order for this constraint to be satisfied, changes have to be made to the following elements:

- *PluggableDevice*:   This component needs to be updated so that the new type of device can be initialised and configured, and thus so that the device's data can be sent to the system.
- *DeviceData*:   Depending on how this data type is implemented, it might need an update in order for it to represent possible new data types (for example Temperature Filipcikova) and for the new data types to be serialized.
- *PluggableDeviceDataDB*: The database needs to be updated so that information can be retrieved about the new types of sensors and the new types of data.   Data related to the displaying of sensor data will also need to be updated.
- *PluggableDeviceConverter*:   see above.

A.4.3   Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above.   For each component we note the quality attribute or use case that prompted us to create it.   Descriptions about the components can be found under chapter 6.

- DeviceDataConverter: M1
- DeviceDataScheduler: P2
- PluggableDeviceDataDB: P2
- PluggableDevice: UC11

Decomposition   Figure A.3 shows the components resulting from the decomposition in this run.

Deployment   Figure A.4 shows the allocation of components to physical nodes.

A.4.4   Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above.   Detailed information about each interface and its methods can be found under chapter 6.

ApplicationManager

- ForwardData

Mote

- DeviceData
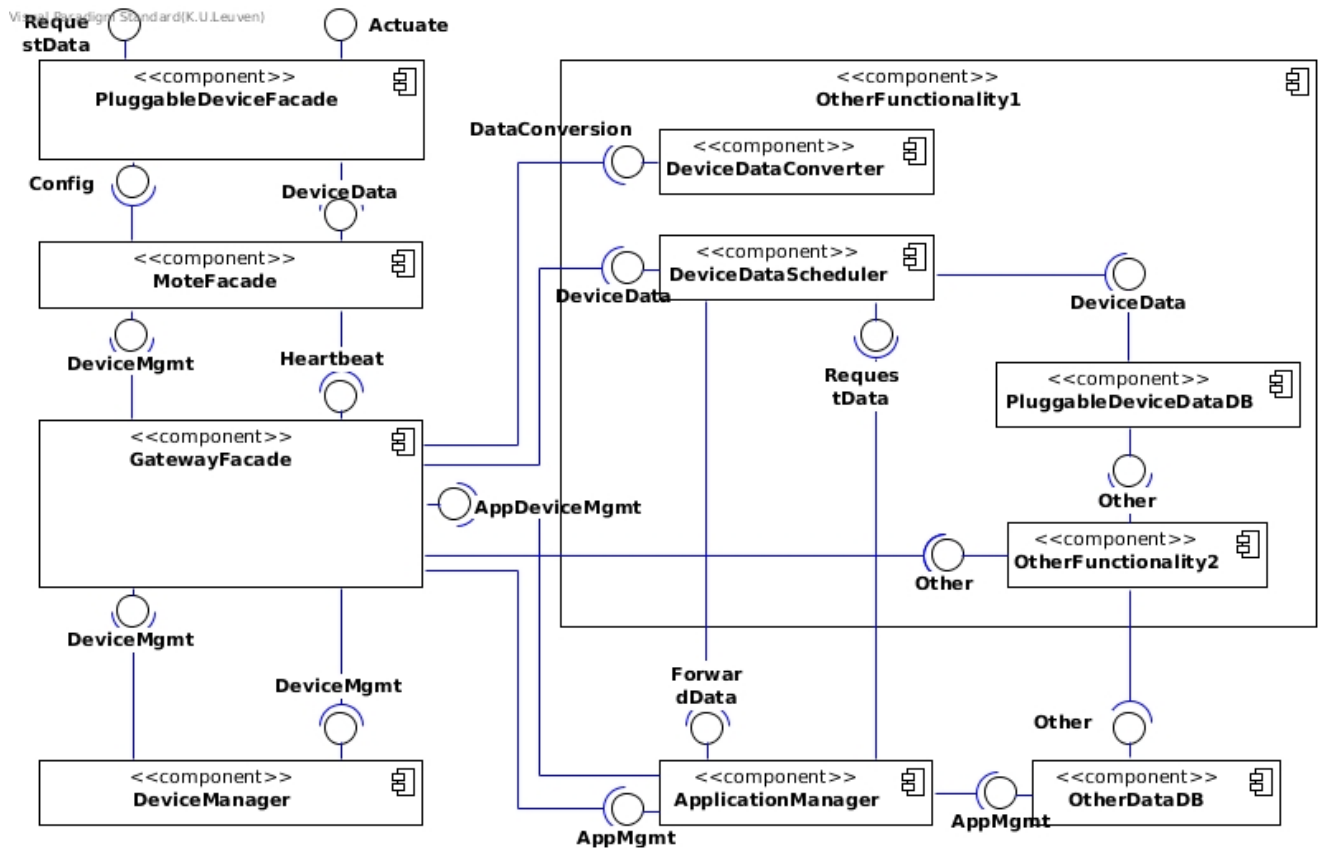
Figure A.3:   Component-and-connector diagram of this decomposition.

PluggableDevice

- Actuate
- Config
- RequestData

DeviceDataConverter

- DataConversion

DeviceDataScheduler

- RequestData
- DeviceData

PluggableDeviceDataDB

- DeviceData

A.4.5   Data type definitions

This section lists the new data types introduced during this decomposition.

- DateTime
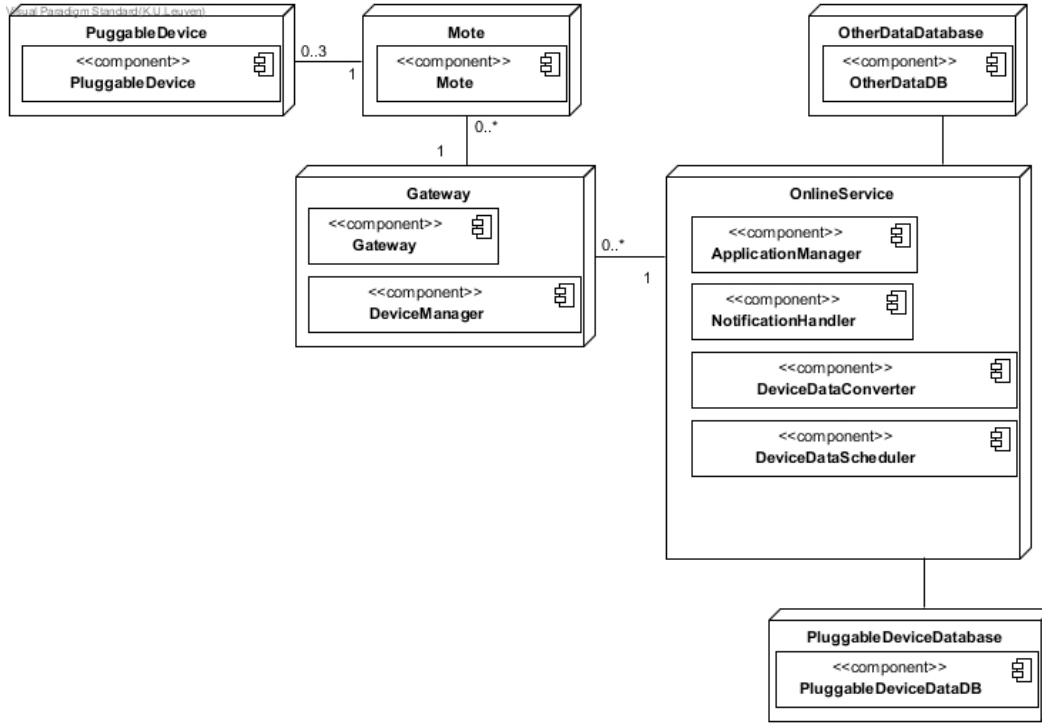
Figure A.4:   Deployment diagram of this decomposition.

A.4.6   Verify and refine

The selected architectural drivers have been handled completely in this decomposition.   This section describes per component which (parts of) the remaining requirements it is responsible for.   If requirements are split in multiple parts, this is indicated by the addition of a letter (or number, depending on the structure of the requirement) after their title.

ApplicationManager

- *Av2*:   Application failure
  Prevention: a, b
  Detection: a, b, c
  Resolution: a, b, c
- *P1*:   Large number of users: c
- *M2*:   Big data analytics on pluggable data and/or application usage data: d, e
- *U1*:   Application updates: a, b, c, d
- *U2*:   Easy Installation: e
- *UC12*:   Perform actuation command
- *UC17*:   Activate an application: 3, 4

OtherDataDB

- None

GatewayFacade

- *Av1*:   Communication between SIoTIP gateway and Online Service
  Resolution: b, c, d
- *U2*:   Easy Installation: a, c, d

Mote

- *U2*: Easy Installation: b, c, d
- *UC04*: Install mote: 1, 2
- *UC05*: Uninstall mote: 1
- *UC06*: Insert a pluggable device into a mote: 2
- *UC07*: Remove a pluggable device from its mote: 2

NotificationHandler

- *UC16*: Consult notification message: 5
- *UC17*: Activate an application: 5, 6

OtherFunctionality2

- *Av1*: Communication between SIoTIP gateway and Online Service
  Detection: a, b, c, d Resolution: a
- *P1*: Large number of users: a
- *M2*: Big data analytics on pluggable data and/or application usage data: a
- *U2*: Easy Installation: e
- *UC01*: Register a customer organisation
- *UC02*: Register an end-user
- *UC03*: Unregister an end user
- *UC04*: Install mote: 3
- *UC05*: Uninstall mote: 2.b
- *UC06*: Insert a pluggable device into a mote: 3: topology part; alternative 3a.1.b
- *UC07*: Remove a pluggable device from its mote: 3.b
- *UC08*: Initialise a pluggable device: 1, 2, 4
- *UC09*: Configure pluggable device access rights
- *UC10*: Consult and configure the topology
- *UC13*: Configure pluggable device
- *UC16*: Consult notification message: 1, 2, 3, 4
- *UC17*: Activate an application: 1, 2
- *UC19*: Subscribe to application
- *UC20*: Unsubscribe from application
- *UC21*: Send invoice
- *UC22*: Upload an application
- *UC23*: Consult application statistics
- *UC24*: Consult historical data
- *UC25*: Access topology and available devices
- *UC26*: Send application command or message to external front-end
- *UC27*: Receive application command or message to external front-end
- *UC28*: Log in
- *UC29*: Log out

PluggableDeviceDataDB

- *M2*: Big data analytics on pluggable data and/or application usage data: b

PluggableDevice

- *U2*: Easy Installation: d

DeviceManager

- *U2*:   Easy Installation: c, d
- *UC04*:   Install mote: 4
- *UC05*:   Uninstall mote: 2
- *UC06*:   Insert a pluggable device into a mote: 3:   uninitialised part; alternative 3a.1 3a.2 3a.4; 4
- *UC07*:   Remove a pluggable device from its mote: 3.a, 3.c
- *UC08*:   Initialise a pluggable device: 3,

DeviceDataScheduler

- *P1*:   Large number of users: b
- *M2*:   Big data analytics on pluggable data and/or application usage data: b, c

A.5 Decomposition 3: U2, UC4, UC6, UC9, UC10, UC17, UC19

A.5.1 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *U2*: Easy installation

The related functional drivers are:

- *UC4*: Install mote

- *UC6*: Insert a pluggable device into a mote

- *UC9*: Configure pluggable device access rights

- *UC10*: Consult and configure topology

- *UC17*: Activate an application

- *UC19*: Subscribe to application

A.5.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

Did you ever hear the tragedy of Darth Plagueis The Wise?   I thought not.   It's not a story the Jedi would tell you.   It's a Sith legend.   Darth Plagueis was a Dark Lord of the Sith, so powerful and so wise he could use the Force to influence the midichlorians to create life...   He had such a knowledge of the dark side that he could even keep the ones he cared about from dying.   The dark side of the Force is a pathway to many abilities some consider to be unnatural.   He became so powerful...   the only thing he was afraid of was losing his power, which eventually, of course, he did.   Unfortunately, he taught his apprentice everything he knew, then his apprentice killed him in his sleep.   Ironic.   He could save others from death, but not himself.

U2: Gateway installation The gateway should not require any configuration, other than being connected to the local wired or WiFi network, after it is plugged into an electrical socket.   An infrastructure owner should be able get the SIoTIP gateway up-and-running (connected) within 10 minutes given that the information (e.g.   WiFi SSID and passphrase) is available to the person responsible for the installation.

A connection to the internet is a constraint of the GatewayFacade.   After the gateway is connected to the internet (we don't model this), it connects to the gateway (we don't model this?)   and registers itself (we model this).

When an infrastructure owner orders a gateway, that gateway is linked to the IO. Gateway was already in the DeviceDB, but it was not linked to anyone.   It has a gatewayID => unique identifier gatewayID same like motes.

Important info related to gateways: GatewayID (new class), infrastructureOwnerID, IPAddress, status (active/inactive), location (in topology table) GatewayInfo(int gatewayID, int manufacturerID, int productID, int infrastructureOwnerID, IPAddress ip, int status)

gateway registers with online service: TODO: gateway is added in the infrastructure owners topology as 'unplaced'.   it will be visible to the infrastructure owner after the gateway connects and registers with the online service

We cannot link the gateway to an exact location for the infrastructure owner, because he might be managing multiple buildings an IP addresses can be dynamic.   If we let the IO choose for which building the gateway is, then this is bad for usability and he has to configure anyways.   Also in that case he cannot buy spare gateways, unless he buys spare gateways for every gateway in the building

U2: Mote installation Installing a new mote should not require more configuration than adding it to the topology.   Adding new motes, sensors or actuators should not involve more than just starting motes, and plugging devices into motes âĂŞ plug-and-play!   Reintroducing a previously known mote, with the same pluggable devices attached to it, should not require any configuration.   It is automatically re-added on its last known location on the topology.   The attached pluggable devices are automatically initialised and configured with their last known configuration and access rights.

Thing that need to happen automatically: *) mote should find the gateway (mote sends a broadcast message->ReceiveBroadcast) => this is done automatically?   see remarks of the use case *) gateway should register the mote (DeviceManager update, store entry in DB) *) on reintroduction of motes: DeviceManager notices this, makes the gateway send a message to online service to reuse some old topology

U2: Pluggable device installation Adding new sensors or actuators should require no further customer actions besides plugging it into the mote.   Configurable sensors and actuators should have a working default configuration.   Pluggable devices added to an already known mote are automatically added in the right location on the topology.   Making (initialised) sensors and actuators available to customer organisations and applications should not require more effort than configuring access rights (cf.   UC9).

)   After devices are plugged in: connect to mote, set up default configurations *) if the mote is already known, the device is added to the right location on the topology *) need something for configuration of access rights, can only happen for initialised devices

*) for reactivating last configurations: just set status to active and don't change configuration field, it will still be the same as in the past alternative: current_configuration and last_configuration in DB alternative: store all configurations on Gateway -> but it has bad resources alternative: store all versions on DeviceDB -> but lots of useless data then = extra work for db

*) Pluggable devices added to an already known mote are automatically added in "the right location" on the topology.   what exactly is a location?   => when a pluggable device is connected to a new mote, the pluggable device gets the location of the mote by default

U2: Easy applications Applications should work out of the box if the required sensors and actuators are available.   Only when mandatory end-user roles must be assigned, additional explicit configuration actions are required from a customer organisation (cf.   UC17, UC19).

)   if there is a subsription and new hardware is plugged in: need something to check if some application can be activated now => see UC6:   checkApplicationsForActivationForInfrastructureOwner *) need something to assign user roles to users during UC19

A.5.3   Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above.   For each component we note the quality attribute or use case that prompted us to create it.   Descriptions about the components can be found under chapter 6.

- AccessRightsManager: U2, UC9
- ApplicationContainerManager: U2, UC17
- CustomerOrganisationClient: U2, UC19
- CustomerOrganisationFacade: U2, UC19
- DeviceDB: U2, UC4, UC6
- InfrastructureOwnerClient: U2, UC9
- InfrastructureOwnerFacade: U2, UC9
- InfrastructureOwnerManager: U2, UC9
- InvoiceManager: U2, UC17
- TopologyManager: U2, UC4, UC6, UC17
- SubscriptionManager: U2, UC19
- UserRolesManager: U2, UC17, UC19

A.5.4   Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above.   Detailed information about each interface and its methods can be found under chapter 6.

AccessRightsManager

- AccessRightsMgmt

ApplicationContainerManager

- AppMgmt

ApplicationManager

- FrontEndAppMgmt
- IOAppMgmt

CustomerOrganisationFacade

- SubscriptionMgmt

DeviceDB

- AccessRightsMgmt
- DeviceMgmt
- TopologyMgmt
- IODeviceMgmt

InfrastructureOwnerFacade

- AccessRights

InfrastructureOwnerManager

- IOMgmt

InvoiceManager

- InvoiceMgmt

OtherDataDB

- InvoiceMgmt
- IOMgmt
- SubscriptionMgmt
- UserRoleMgmt

TopologyManager

- TopologyMgmt

SubscriptionManager

- SubscriptionMgmt

UserRolesManager

- RoleMgmt

A.5.5   Data type definitions

This section lists the new data types introduced during this decomposition.

- Application
- IPAddress
- Relationship
- RoomTopology
- User
- UserRole

A.6 Decomposition 4: Av2, UC12, UC25, UC26, UC27 (application execution subsystem)

A.6.1 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av2*: Application failure

The related functional drivers are:

- *UC12*: Perform actuation command
- *UC13*: Configure pluggable device
- *UC25*: Access topology and available devices
- *UC24*: Consult historical data
- *UC26*: Send application command or message to external front-end
- *UC27*: Receive application command or message from external front-end

Rationale At this point the remaining drivers were Av1, Av2, and P1, which all had medium priority. We chose decompositions 4, 5, and 6 based on the priorities of the use cases that are related to the quality attributes.

The related use cases from now on are the ones that would use components that are going to be changed in the decomposition.

A.6.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

If you read this, you are cool!

RATIONALE: ApplicationContainerManager and DeviceCommandConstructor need to be reconfigured ivm some used interfaces

Av2: Detection of failures The system is able to autonomously detect failures of its individual application execution components, failing applications, and failing application containers.

Upon detection, a SIoTIP system administrator is notified.

The failure of an internal application execution component is detected within 30 seconds. Detection of failed hardware or crashed software happens within 5 seconds. SIoTIP system administrators are notified within 1 minute.

To detect failures, we made use of the `Container` pattern. The application execution subsystem is composed of:

- `ApplicationContainer`
- `ApplicationContainerMonitor`
- `ApplicationContainerManager`
- `ApplicationExecutionSubsystemMonitor`

The `ApplicationContainer`s are deployed in groups on different nodes.

`ApplicationContainer`: is a container/sandbox that has 1 running application instance

`ApplicationContainerMonitor`: monitors the `ApplicationContainer` instances

To detect failing applications, `ApplicationContainer` and `ApplicationContainerMonitor` ApplicationContainer -> ApplicationContainerMonitor: void applicationCrashed(id applicationInstanceID)

To detect failling application containers, `ApplicationContainerMonitor` ApplicationContainerMonitor -> ApplicationContainer: Echo ping() -> we say container has crashed/failed when the following has no response:

To detect failures of individual application execution components, This means that one of `ApplicationContainer`, `ApplicationContainerMonitor`, `ApplicationContainerManager` crashed. If the `ApplicationContainer` failed, then the `ApplicationContainerMonitor` would detect this. If one of the other two components failed, the `ApplicationExecutionSubsystemMonitor` is put in place to detect this.

These components will be deployed on the Online Service and on gateways. Since gateways are weaker machines than the ones on the Online Service, the ApplicationContainer can be confgured differently for gateways. The ApplicationContainers will then have stricter limits on resources used of the node they are working on. DeviceCommandConstructor and ApplicationContainerManager interfaces need to begin re-routed depening on whether they are in a Gateway or on the Online Service.

Av2: Resolution of application failures and application execution component failures In case of application crash, the system autonomously restarts failed applications. If part of an application fails, the remaining parts remain operational, possibly in a degraded mode (graceful degradation).

After 3 failed restarts the application is suspended, and the application developer and customer organisation are notified within 5 minutes.

In case of failure of application execution components or an application container, a system administrator is notified.

SIoTIP system administrators are notified within 1 minute.

When an application instance fails, the ApplicationContainerMonitor detects this and sends a command to the ApplicationContainerManager to restart the application instance. The ApplicationContainerMonitor keeps track of how many times the application instance has been restarted after a failure. After 3 failed restarts, the monitor send a command to the ApplicationContainerManager to suspend the application instance and send a notification to the application developers of the application and to the affected customer organisaiton. Also, to achieve graceful degradation, the ApplicationContainerManager notifies other parts of the application instance of its suspension.

If one of the components of the application execution subsystem fails, a SIoTIP system administrator is notified.

Av2: Failures do not impact other applications or other functionality of the system This does not affect other applications that are executing on the Online service or SIoTIP gateway. This does not affect the availability of other functionality of the system, such as the dashboards.

Applications fail independently: they are executed within their own container to avoid application crashes to affect other applications.

Each ApplicationContainer contains one application instance. If an application fails, then this will be handled by the application execution subsystem so this does not affect any other application or other functionality of the system. The ApplicationContainers are constructed such that failures of applications do not affect the containers. The ApplicationContainers are to be deployed on different nodes alone or grouped with other containers. Write something here.

UC12: DEVELOPERS WRITE THIS: command = "on" actuators = getActuatorsOfType("lightswitch") foreach (actuator) actuator.command("on")

WE NEED TO CONVERT "on" TO A COMMAND THAT THE ACTUATOR UNDERSTANDS "on" => "turnOn" "on" => "lightOn" "on" => "switch"

1. An application indicates that it wants one or more pluggable devices to perform an actuation command from client application: ApplicationClient -> ApplicationFacade: interface DeviceCommands: void sendActuationCommand(List<PluggableDeviceID> devices, string command) ApplicationFacade -> ApplicationManagementLogic: interface DeviceCommands: void sendActuationCommand(List<PluggableDeviceID> devices, string command)

from applicationContainer: ApplicationContainer -> ApplicationContainerManager: interface DeviceCommands: void sendActuationCommand(List<PluggableDeviceID> devices, string command)

68

ApplicationContainerManager -> ApplicationManagementLogic: interface DeviceCommands: void sendActuationCommand(List<PluggableDeviceID> devices, string command)

from application on gateway: SKIP STEP 2 GW/ApplicationContainer -> GW/DeviceCommandConstructor: interface Commands: void sendActuationCommand(List<PluggableDeviceID> device, string commandName) GW/DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: Map<PluggableDeviceID, string> getFormattingSyntaxForDevices(List<PluggableDeviceID> devices) GW/DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: void sendActuationCommand(Map<PluggableDeviceID, string> commandsForDevices)

2. The system -constructs the actuation command message according to the specific formatting syntax for the involved pluggable device(s) ApplicationManagementLogic -> DeviceCommandConstructor: interface Commands: void sendActuationCommand(List<PluggableDeviceID> device, string commandName) DeviceCommandConstructor -> DeviceDB: interface DBAppDeviceMgmt: Map<PluggableDeviceID, string> getFormattingSyntaxForDevices(List<PluggableDeviceID> devices)

-sends the command message to the intended pluggable device(s). DeviceCommandConstructor -> DeviceDB: interface DBAppDeviceMgmt: Map<PluggableDeviceID, GatewayInfo> getGatewaysForDevices(List<PluggableDeviceID> devices) DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: void sendActuationCommand(Map<PluggableDeviceID, string> commandsForDevices)

3. The pluggable device(s) receive(s) the actuation command message and perform(s) the contained actuation command. DeviceManager -> Mote: interface DeviceMgmt: void sendActuationCommand(PluggableDeviceID device, string commandName) Mote -> PluggableDevice: interface Actuate: void sendActuationCommand(string commandName)

UC13: 1. The primary actor specifies that it wants to set a configuration parameter of a pluggable device. from client application: ApplicationClient -> ApplicationFacade: interface DeviceCommands: void setConfiguration(PluggableDeviceID pID, Map<string, string> config) ApplicationFacade -> ApplicationManagementLogic: interface DeviceCommands: void setConfiguration(PluggableDeviceID pID, Map<string, string> config)

from applicationContainer: ApplicationContainer -> ApplicationContainerManager: interface DeviceCommands: void setConfiguration(PluggableDeviceID pID, Map<string, string> config) ApplicationContainerManager -> ApplicationManagementLogic: interface DeviceCommands: void setConfiguration(PluggableDeviceID pID, Map<string, string> config)

from application on gateway: SKIP STEPS 2 AND 3 GW/ApplicationContainer -> GW/DeviceCommandConstructor: interface Commands: void verifyAndConstructConfigurationsForDevice(PluggableDeviceID pID, Map<string, string> config) GW/DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: string getPossibleConfigurationParametersForDevice(PluggableDeviceID pID) GW/DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: Map<PluggableDeviceID, string> getFormattingSyntaxForDevices(List<PluggableDeviceID> devices) GW/DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: void setConfiguration(PluggableDeviceID pID, Map<string, string> config)

2. The system verifies that the value of the configuration parameter is valid for the device (for example, a sensor which provides temperature information may have hardware limits on the sampling frequency). ApplicationManagementLogic -> DeviceCommandConstructor: interface Commands: void verifyAndConstructConfigurationsForDevice(PluggableDeviceID pID, Map<string, string> config) DeviceCommandConstructor -> DeviceDB: interface DBAppDeviceMgmt: string getPossibleConfigurationParametersForDevice(PluggableDeviceID pID) DeviceCommandConstructor -> DeviceDB: interface DBAppDeviceMgmt: Map<PluggableDeviceID, string> getFormattingSyntaxForDevices(List<PluggableDeviceID> devices)

3. The system determines whether the pluggable device needs to be reconfigured, and if so, constructs a reconfiguration command according to the specific formatting syntax for the pluggable device and sends it to the pluggable device.

DeviceCommandConstructor -> DeviceManager: interface DeviceCommands: void setConfiguration(PluggableDeviceID pID, Map<string, string> config)

4. The system updates the internal configuration of the pluggable device. DeviceManager -> Mote: interface DeviceMgmt: void setConfig(...)

UC24: 1. The primary actor indicates that it wants to consult a specified collection of historical data in a specified timeframe. from client application: ApplicationClient -> ApplicationFacade: interface

DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForDevices(List<PluggableDeviceID> devices, DateTime from, DateTime to) ApplicationClient -> ApplicationFacade: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForRoom(RoomTopology room, DateTime from, DateTime to)

ApplicationFacade -> ApplicationManagementLogic: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForDevices(List<PluggableDeviceID> devices, DateTime from, DateTime to) ApplicationClient -> ApplicationFacade: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForRoom(RoomTopology room, DateTime from, DateTime to)

from ApplicationContainer: ApplicationContainer -> ApplicationContainerManager: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForDevices(List<PluggableDeviceID> devices, DateTime from, DateTime to) ApplicationContainerManager -> ApplicationManagementLogic: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForDevices(List<PluggableDeviceID> devices, DateTime from, DateTime to) ApplicationContainer -> ApplicationContainerManager: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForRoom(RoomTopology room, DateTime from, DateTime to) ApplicationContainerManager -> ApplicationManagementLogic: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForRoom(RoomTopology room, DateTime from, DateTime to)

2. The system determines from which pluggable devices the data is required and looks up the data. ApplicationManagementLogic parses the RoomTopology object and uses the PluggableDeviceIDs in the RoomTopology

ApplicationManagementLogic -> DeviceDataScheduler: interface RequestData: Map<PluggableDeviceID, List<DeviceData» getDataForDevices(List<PluggableDeviceID> devices, DateTime from, DateTime to) DeviceDataScheduler -> PluggableDeviceDB: interface DeviceData: Map<PluggableDeviceID, List<DeviceData» getDataForDevices(List<PluggableDeviceID> devices, DateTime from, DateTime to)

3. The system presents the primary actor with the requested historical overview, e.g. as a table. Return value of first call.

UC25: 1. The primary actor indicates that it wants an overview of the topology. applicationClient wants overview: ApplicationClient -> ApplicationFacade: interface DeviceData: List<RoomTopology> getTopologyOverview(int applicationInstanceID, int customerOrganisationID) ApplicationFacade -> ApplicationManagementLogic: interface DeviceData: List<RoomTopology> getTopologyOverview(int applicationInstanceID, int customerOrganisationID)

applicationContainer wants overview: ApplicationContainer -> ApplicationContainerManager: interface DeviceData: List<RoomTopology> getTopologyOverview(int applicationInstanceID, int customerOrganisationID) ApplicationContainerManager -> ApplicationManagementLogic: interface DeviceData: List<RoomTopology> getTopologyOverview(int applicationInstanceID, int customerOrganisationID)

2. The system looks up the pluggable devices that are available to the customer organisation that owns the primary actor, and composes a view on the topology including these pluggable devices. ApplicationManagementLogic -> TopologyManager: interface TopologyMgmt: List<RoomTopology> getTopology(int custOrgID) TopologyManager -> DeviceDB: interface TopologyMgmt: List<RoomTopology> getTopology(int custOrgID)

ApplicationManagementLogic -> OtherDataDB: interface AppMgmt: List<PluggableDeviceID> getDevicesForApplication(int applicationInstanceID)

3. The system presents the topology view to the primary actor. = return value of getTopologyOverview

UC26: AppDevelopers want to send a message to: Gateway: sendMessageToGateway(string message) OS: sendMessageToOnlineService(string message) AppClient: sendMessageToExternalClient(string message, IPAddress host, int port)

Some deployment rationale: ApplicationManagementLogic <— HTTP —> ApplicationClient ApplicationManagementLogic <— —> GW/ApplicationContainer

we need to store on which gateways ApplicationContainers are running. ApplicationManagementLogic -> OtherDataDB: void updateApplicationInstanceGateway(int applicationInstance, int gatewayID)

Alternative 3b. NotAllowedException

1. The primary actor indicates it wants to send an application command or message to an external front-end and specifies the destination (e.g., as an application identifier for SIoTIP applications, or a hostname and port for external systems).

FROM APP INSTANCE TO FRONT END: ApplicationContainer -> ApplicationContainerManager: interface AppMessages: void sendMessageToExternalFrontEnd(string hostName, int port, string message) ApplicationContainerManager -> ApplicationManagementLogic: interface AppMessages: void sendMessageToExternalFrontEnd(string hostName, int port, string message)

FROM APP INSTANCE TO APP INSTANCE: ApplicationContainer -> ApplicationContainerManager: interface AppMessages: void sendCommandToApplicationInstance(int applicationInstanceID, string command) ApplicationContainerManager -> ApplicationManagementLogic: interface AppMessages: void sendCommandToApplicationInstance(int applicationInstanceID, string command)

FROM FRONT END TO APP INSTANCE: ApplicationClient -> ApplicationFacade: interface AppMgmt: void sendCommandToApplicationInstance(int applicationInstanceID, string command) ApplicationFacade -> ApplicationManagementLogic: interface FrontEndAppRequests: void sendCommandToApplicationInstance(int applicationInstanceID, string command)

2. The system checks that the primary actor is allowed to send to the specified destination. THE MESSAGE HAS ARRIVED IN ApplicationManagementLogic

Don't need to check, since an application can only send commands to another part of the same application. If the command comes from a gateway -> send it to the online service instance If the command comes from the online service -> send it to the gateway instance If the command is for an external front end -> just send the message to the hostname and port

Maybe a check can be done here to do some kind of rate limiting, if developers want to send more requests, they pay more MOOONNEEYYYYY

3. If the primary actor is allowed to send to the destination, and if the destination is another application, the system delivers the application command to that destination (Include: UC27: Receive application command or message from external front-end).

UC27: Comment from professor: UC27 (just as UC26) deals with communication between different parts of the same application (including messages coming from front-ends).

1. The system receives an application command or message for a SIoTIP application. ApplicationManagementLogic has received an application command (SEE UC26)

2. The system checks that the destination is available. message is for Gateway/ApplicationContainer: Destination = GW/ApplicationContainer find where the application container is, -> OtherDataDB: gatewayOfApplicationContainer then send it to the correct GW/ApplicationContainerManager the applicationContainerMAnager will send the message to the correct ApplicationContainer ApplicationContainerManagers keep track of ApplicationContainer IDs

ApplicationManagementLogic -> OtherDataDB: interface AppMgmt: Gateway getGatewayForApplicationInstance(int applicationInstanceID) ApplicationManagementLogic -> GW/ApplicationContainerManager: interface AppInstanceManagement: boolean isApplicationInstanceAvailable(int applicationInstanceID) GW/ApplicationContainerManager -> GW/ApplicationContainerMonitor: interface AppStatus: boolean isApplicationInstanceAvailable(int applicationInstanceID)

message is for ApplicationContainer: The same, but on Online Service, so you don't need to search for the gateway just send the message to the OS ApplicationContainerManager, it will do the rest

ApplicationManagementLogic -> ApplicationContainerManager: interface AppInstanceManagement: boolean isApplicationInstanceAvailable(int applicationInstanceID) ApplicationContainerManager -> ApplicationContainerMonitor: interface AppStatus: boolean isApplicationInstanceAvailable(int applicationInstanceID)

message is for ApplicationClient: just PING the hostname and port, if the ping succeeds, send the message ApplicationManagementLogic: ping the (host, port) and check reply

3. If the destination is available, the system delivers the message to the destination application. message = string

message is for Gateway/ApplicationContainer: ApplicationManagementLogic -> GW/ApplicationContainerManager: interface AppInstanceManagement: void sendMessageToApplicationInstance(int applicationInstanceID, string message) GW/ApplicationContainerManager -> GW/ApplicationContainer: interface AppInstanceManagement: void sendMessageToApplicationInstance(int applicationInstanceID, string message)

message is for ApplicationContainer: ApplicationManagementLogic -> ApplicationContainerManager: interface AppInstanceManagement: void sendMessageToApplicationInstance(int applicationInstanceID, string

message) ApplicationContainerManager -> ApplicationContainer: interface AppInstanceManagement: void
sendMessageToApplicationInstance(int applicationInstanceID, string message)
message is for ApplicationClient: ApplicationManagementLogic: Open socket (host and port are given) ->
send message -> close socket

A.6.3   Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above.   For
each component we note the quality attribute or use case that prompted us to create it.   Descriptions about
the components can be found under chapter 6.

- Component: ApplicationClient
- Component: ApplicationContainer
- Component: ApplicationContainerManager
- Component: ApplicationContainerMonitor
- Component: ApplicationExecutionSubsystemMonitor
- Component: ApplicationFacade
- Component: ApplicationManagementLogic
- Component: DeviceCommandConstructor

A.6.4   Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above.   Detailed
information about each interface and its methods can be found under chapter 6.

ApplicationContainer

- AppMessages
- AppInstanceMgmt
- AppMonitoring

ApplicationContainerManager

- AppMessages
- AppInstanceMgmt
- DeviceCommands
- DeviceData

ApplicationContainerMonitor

- Monitoring
- AppStatus

ApplicationFacade

- DeviceData
- DeviceCommands

ApplicationManagementLogic

- DeviceData
- DeviceCommands
- FrontEndAppRequests
- GWAppInstanceMgmt

- AppMessages

DeviceCommandConstructor

- Commands

DeviceManager

- DeviceCommands

DeviceDB

- AppDeviceMgmt

A.6.5   Data type definitions

This section lists the new data types introduced during this decomposition.

- Echo the response to a ping message

A.7 Decomposition 6: P1, UC1, UC2, UC3, UC5, UC7, UC8, UC16, UC20 (Elements/Subsystem to decompose/expand)

A.7.1 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *P1*: Large number of users

The related functional drivers are:

- *UC1*: Register a customer organisation

- *UC2*: Register an end-user

- *UC3*: Unregister an end-user

- *UC5*: Uninstall mote

- *UC7*: Remove a pluggable device from its mote

- *UC8*: Initialise a pluggable device

- *UC16*: Consult notification message

- *UC20*: Unsubscribe from application

A.7.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.
https://www.alertra.com/blog/2010/improve-availability-performance-using-database-replication https://serverfault.com/ques are-the-performance-implications-for-using-sql-server-replication " Ik dacht aan misschien de meeste van de componenten gewoon duplicaten maar dan is de DB wel u bottleneck en dan moet ge zo'n systeem van distributed systems gebruiken om ook de databases juist te kunnen replicaten. Maar daarnaast echt geen idee ivm performance. Ge kunt wel bullshit van de "tactics" in u rationale zetten zoals the developers have to "increase computation efficiency" and "reduce computational overhead". Manage event rate en scheduling policy kunt ge misschien wel gebruiken om er voor te zorgen dat bepaalde taken gebeuren op een moment dat de load op de online service wat lager is maar ik weet Ni echt welke taken "

P1: Problem title The SIoTIP Online Service replies to the service requests of the infrastructure owner and customer organisations.

P1: Problem title The Online Service processes the data received from the gateways.

P1: Problem title The application execution subsystem should be able to execute an increasing number of active applications.

P1: Problem title The initial deployment of SIoTIP should be able to deal with at least 5000 gateways in total, and should be provisioned to service at least 3000 registered users simultaneously connected to SIoTIP. -> 5000 gateways * 4 motes per gateway * 3 devices per mote = 60000 devices -> Keep communication with gateways at a minimum e.g. gateway messages are of type "send data", "new device connected", ... -> LOAD BALANCING

P1: Problem title Scaling up to service an increasing amount of infrastructure owners, customers organisations and applications should (in worst case) be linear ; i.e. it should not require proportionally more resources (machines, etc.) than the initial amount of resources provisioned per customer organisation/infrastructure owner and per gateway.

A.7.3   Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- Component: (P1 or UC)

A.7.4   Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

Component

- Interface

A.7.5   Data type definitions

This section lists the new data types introduced during this decomposition.

- DateTime: Represents an instant in time, typically expressed as a date and time of day.

A.8 Decomposition 7: UC28, UC29 (Elements/Subsystem to decompose/expand)

At this point, all quality attributes have been handled. The remaining decompositions handle all of the use cases that are left. The order is based on the priority of the use cases.

A.8.1 Selected architectural drivers

The functional drivers are:

- *UC28*: Log in

- *UC29*: Log out

A.8.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

UC: Problem title Short description of the problem.
Solution for the problem.
WHen you check credentials, return a string sessionID to the client. The client sends this sessionID with every request. The Online Service checks the sessionID to determine if the user is logged in. Monika: store sessionID in otherDataDB PMS: store sessionID in sessionDB Check what is better for us http://shiflett.org/articles/storing-sessions-in-a-database

A.8.3 Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- Component: (Relevant UC)

A.8.4 Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

Component

- Interface

A.8.5 Data type definitions

This section lists the new data types introduced during this decomposition.

- DateTime: Represents an instant in time, typically expressed as a date and time of day.

A.9 Decomposition 8: UC22, UC23 (Elements/Subsystem to decompose/expand)

A.9.1 Selected architectural drivers

The functional drivers are:

- *UC22*:   Upload an application

- *UC23*:   Consult application statistics

A.9.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

UC: Problem title   Short description of the problem.
Solution for the problem.

A.9.3 Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above.   For each component we note the quality attribute or use case that prompted us to create it.   Descriptions about the components can be found under chapter 6.

- Component: (Relevant UC)

A.9.4 Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above.   Detailed information about each interface and its methods can be found under chapter 6.

Component

- Interface

A.9.5 Data type definitions

This section lists the new data types introduced during this decomposition.

- DateTime: Represents an instant in time, typically expressed as a date and time of day.

A.10 Decomposition 9: UC21 (Elements/Subsystem to decompose/expand)

A.10.1 Selected architectural drivers

The functional drivers are:

- *UC21*: Send invoice

A.10.2 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

UC: Problem title Short description of the problem.
Solution for the problem.

A.10.3 Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- Component: (Relevant UC)

A.10.4 Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

Component

- Interface

A.10.5 Data type definitions

This section lists the new data types introduced during this decomposition.

- DateTime: Represents an instant in time, typically expressed as a date and time of day.