



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)

Academic year 2016–2017

# Contents

<b>1</b>	<b>Architectural Decisions</b>	<b>5</b>
1.1	ReqX: Requirement Name . . . . .	5
1.2	Other decisions . . . . .	5
1.2.1	Decision 1 . . . . .	6
1.3	Discussion . . . . .	6
<b>2</b>	<b>Client-server view (UML Component diagram)</b>	<b>7</b>
2.1	Context diagram . . . . .	7
2.2	Primary diagram . . . . .	7
<b>3</b>	<b>Decomposition view (UML Component diagram)</b>	<b>9</b>
<b>4</b>	<b>Deployment view (UML Deployment diagram)</b>	<b>11</b>
4.1	Context diagram . . . . .	11
4.2	Primary diagram . . . . .	11
<b>5</b>	<b>Scenarios</b>	<b>14</b>
5.1	Scenarios . . . . .	14
<b>6</b>	<b>Element Catalog and Datatypes</b>	<b>24</b>
<b>7</b>	<b>Catalog</b>	<b>25</b>
7.1	Components . . . . .	25
7.1.1	AccessRightsManager . . . . .	25
7.1.2	ApplicationContainer . . . . .	25
7.1.3	ApplicationManager . . . . .	25
7.1.4	CustomerOrganisationClient . . . . .	25
7.1.5	CustomerOrganisationFacade . . . . .	25
7.1.6	Database . . . . .	25
7.1.7	DeviceDataConverter . . . . .	26
7.1.8	DeviceDataScheduler . . . . .	26
7.1.9	DeviceDB . . . . .	26
7.1.10	DeviceManager . . . . .	26
7.1.11	GatewayFacade . . . . .	26
7.1.12	GWApplicationContainer . . . . .	27
7.1.13	InfrastructreOwnerClient . . . . .	27
7.1.14	InfrastructureOwnerFacade . . . . .	27
7.1.15	InfrastructureOwnerManager . . . . .	27
7.1.16	InvoiceManager . . . . .	27
7.1.17	MoteFacade . . . . .	27
7.1.18	NotificationDeliveryService . . . . .	28
7.1.19	NotificationHandler . . . . .	28
7.1.20	OtherFunctionality1 . . . . .	28
7.1.21	OtherFunctionality2 . . . . .	28
7.1.22	PluggableDeviceDataDB . . . . .	28
7.1.23	PluggableDeviceFacade . . . . .	28
7.1.24	SubscriptionManager . . . . .	28
7.1.25	TopologyManager . . . . .	29
7.1.26	UserRolesManager . . . . .	29
7.2	Interfaces . . . . .	29

7.2.1	AccessRights	29
7.2.2	AccessRightsMgmt	29
7.2.3	AccessRightsMgmt	29
7.2.4	Actuate	29
7.2.5	AppDeviceMgmt	29
7.2.6	AppDeviceMgmt	30
7.2.7	AppMgmt	30
7.2.8	AppMgmt	30
7.2.9	AppMgmt	30
7.2.10	AppMgmt	30
7.2.11	AppMgmt	31
7.2.12	Apps	31
7.2.13	Config	31
7.2.14	DataConversion	31
7.2.15	DeliveryMgmt	31
7.2.16	DeviceData	31
7.2.17	DeviceData	32
7.2.18	DeviceData	32
7.2.19	DeviceData	32
7.2.20	DeviceMgmt	32
7.2.21	DeviceMgmt	33
7.2.22	DeviceMgmt	33
7.2.23	DeviceMgmt	33
7.2.24	ForwardData	34
7.2.25	Heartbeat	34
7.2.26	InvoiceMgmt	34
7.2.27	InvoiceMgmt	34
7.2.28	IOAppMgmt	34
7.2.29	IODeviceMgmt	35
7.2.30	IOMgmt	35
7.2.31	IOMgmt	35
7.2.32	NotificationDeliveryMgmt	35
7.2.33	NotificationMgmt	35
7.2.34	Notify	35
7.2.35	Other	36
7.2.36	Other	36
7.2.37	Other	36
7.2.38	RequestData	36
7.2.39	RequestData	36
7.2.40	RoleMgmt	36
7.2.41	SubscriptionMgmt	36
7.2.42	SubscriptionMgmt	36
7.2.43	SubscriptionMgmt	37
7.2.44	TopologyMgmt	37
7.2.45	TopologyMgmt	37
7.2.46	UserRoleMgmt	37
7.3	Exceptions	37
7.4	Data types	37

<b>A</b>	<b>Attribute-driven design documentation</b>	<b>39</b>
A.1	Introduction	39
A.2	Adapted ADD process	39
A.2.1	Decomposition X: DRIVERS (Elements/Subsystem to decompose/expand)	39
A.2.2	Data type definitions and Interfaces for child modules	39
A.2.3	Verify and refine	39
A.3	Decomposition 1: Av3, UC14, UC15, UC18 (SIoTIP System)	40
A.3.1	Selected architectural drivers	40
A.3.2	Architectural design	40
A.3.3	Instantiation and allocation of functionality	41
A.3.4	Interfaces for child modules	43
A.3.5	Data type definitions	43
A.3.6	Verify and refine	43
A.4	Decomposition 2: M1, P2, UC11 (OtherFunctionality1)	46
A.4.1	Selected architectural drivers	46
A.4.2	Architectural design	46
A.4.3	Instantiation and allocation of functionality	47
A.4.4	Interfaces for child modules	47
A.4.5	Data type definitions	48
A.4.6	Verify and refine	49
A.5	Decomposition 3: U2, UC4, UC6, UC9, UC10, UC17, UC19 (Elements/Subsystem to decompose/expand)	52
A.5.1	Selected architectural drivers	52
A.5.2	Architectural design	52
A.5.3	Instantiation and allocation of functionality	55
A.5.4	Interfaces for child modules	55
A.5.5	Data type definitions	58
A.6	Decomposition 4: Av2, UC12, UC25, UC26, UC27 (application execution subsystem)	59
A.6.1	Selected architectural drivers	59
A.6.2	Architectural design	59
A.6.3	Instantiation and allocation of functionality	61
A.6.4	Interfaces for child modules	61
A.6.5	Data type definitions	62
A.7	Decomposition 5: Av1 (Elements/Subsystem to decompose/expand)	63
A.7.1	Selected architectural drivers	63
A.7.2	Architectural design	63
A.7.3	Instantiation and allocation of functionality	63
A.7.4	Interfaces for child modules	63
A.7.5	Data type definitions	64
A.8	Decomposition 6: P1, UC1, UC2, UC3, UC5, UC7, UC8, UC16, UC20 (Elements/Subsystem to decompose/expand)	65
A.8.1	Selected architectural drivers	65
A.8.2	Architectural design	65
A.8.3	Instantiation and allocation of functionality	66
A.8.4	Interfaces for child modules	66
A.8.5	Data type definitions	66
A.9	Decomposition 7: UC28, UC29 (Elements/Subsystem to decompose/expand)	67
A.9.1	Selected architectural drivers	67
A.9.2	Architectural design	67
A.9.3	Instantiation and allocation of functionality	67
A.9.4	Interfaces for child modules	67
A.9.5	Data type definitions	67
A.10	Decomposition 8: UC22, UC23 (Elements/Subsystem to decompose/expand)	68
A.10.1	Selected architectural drivers	68

A.10.2 Architectural design . . . . .	68
A.10.3 Instantiation and allocation of functionality . . . . .	68
A.10.4 Interfaces for child modules . . . . .	68
A.10.5 Data type definitions . . . . .	68
A.11 Decomposition 9: UC21 (Elements/Subsystem to decompose/expand) . . . . .	69
A.11.1 Selected architectural drivers . . . . .	69
A.11.2 Architectural design . . . . .	69
A.11.3 Instantiation and allocation of functionality . . . . .	69
A.11.4 Interfaces for child modules . . . . .	69
A.11.5 Data type definitions . . . . .	69

# 1. Architectural Decisions

**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 were made.

### Considered Alternatives

**Alternative(s) for choice 1** Explain what alternative(s) you considered for this design choice and why they were 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

2.1	Context diagram for the client-server view. . . . .	8
2.2	Primary diagram of the client-server view. . . . .	8

✓ **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.



## NotificationDeliveryService

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

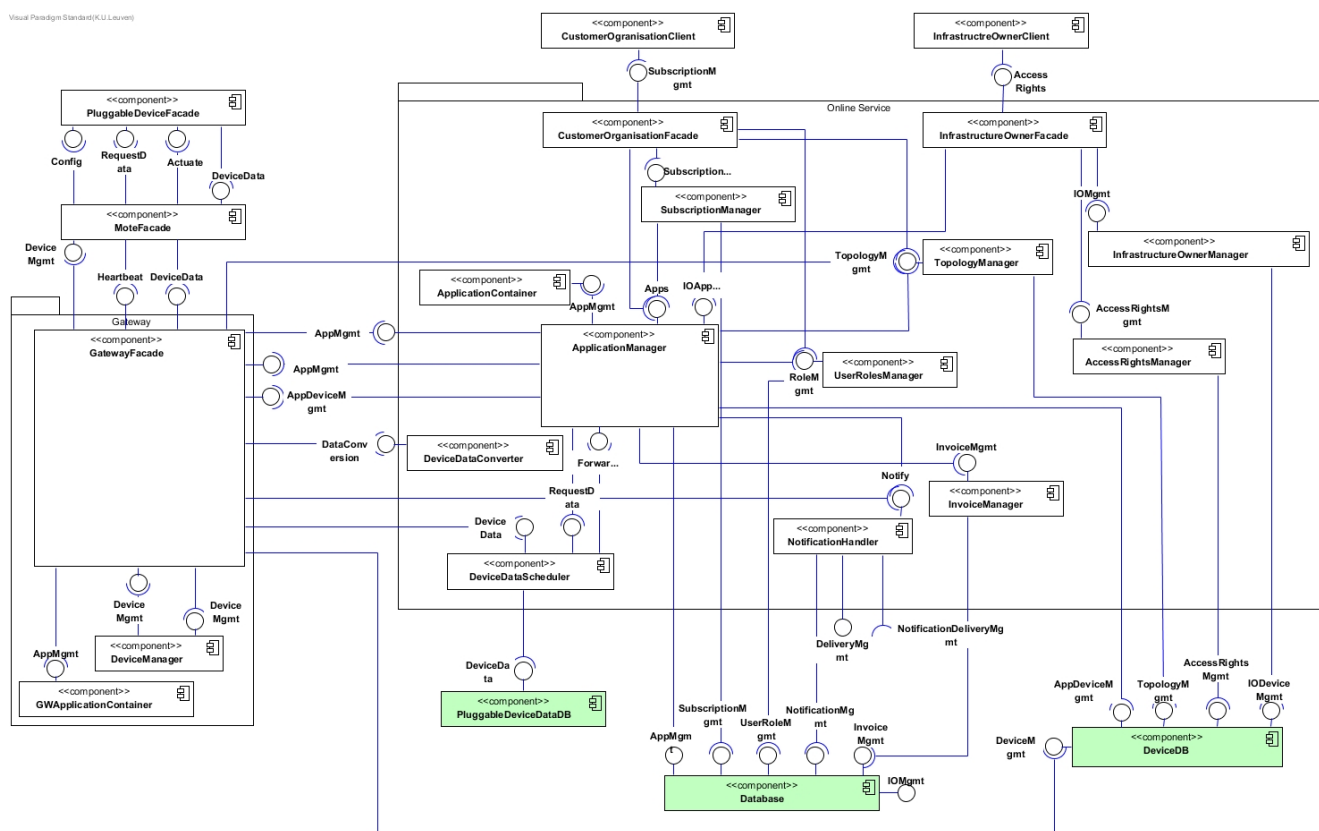


Figure 2.2: Primary diagram of the client-server view.

# 3. Decomposition view (UML Component diagram)

## Figures

3.1	Decomposition of <code>ComponentX</code> . . . . .	9
3.2	Decomposition of <code>ComponentY</code> . . . . .	10

✔ **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.

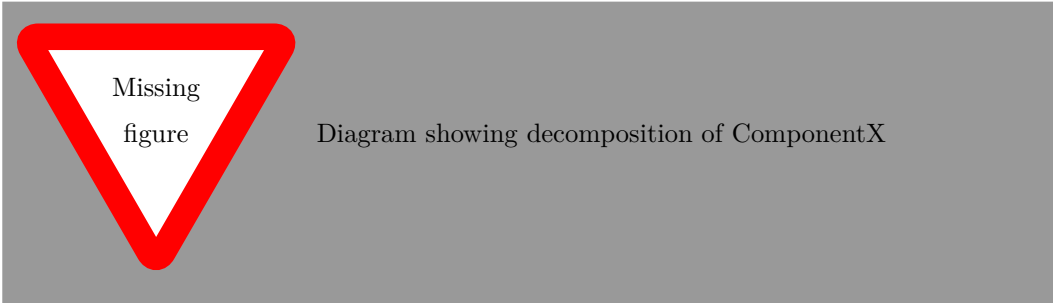


Figure 3.1: Decomposition of `ComponentX`

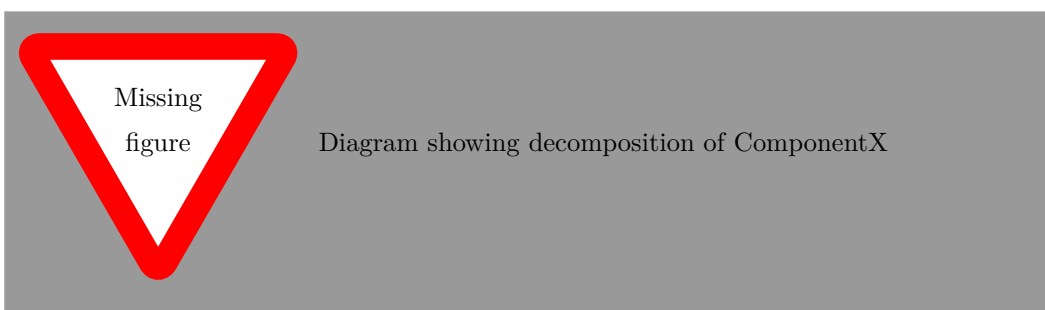


Figure 3.2: Decomposition of **ComponentY**.

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

4.1	Context diagram for the deployment view. . . . .	12
4.2	Primary diagram for the deployment view. . . . .	13

✓ **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.

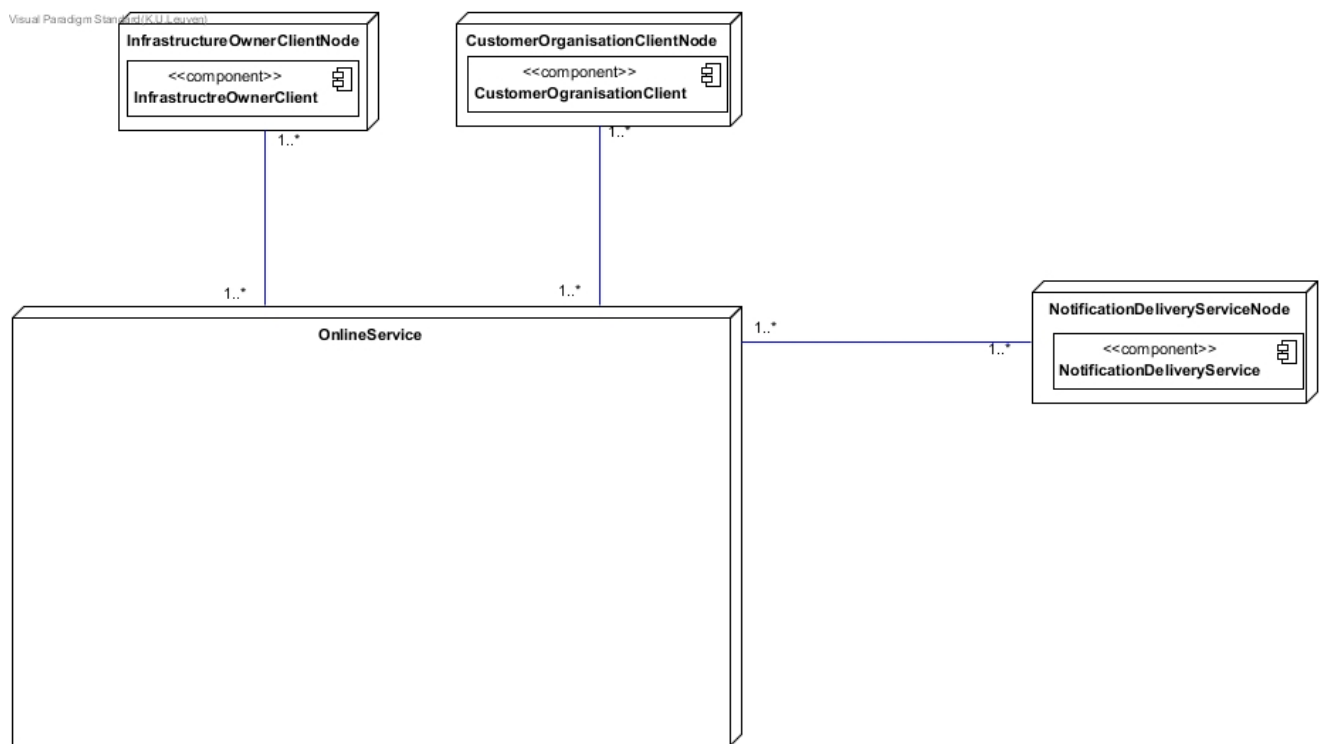


Figure 4.1: Context diagram for the deployment view.

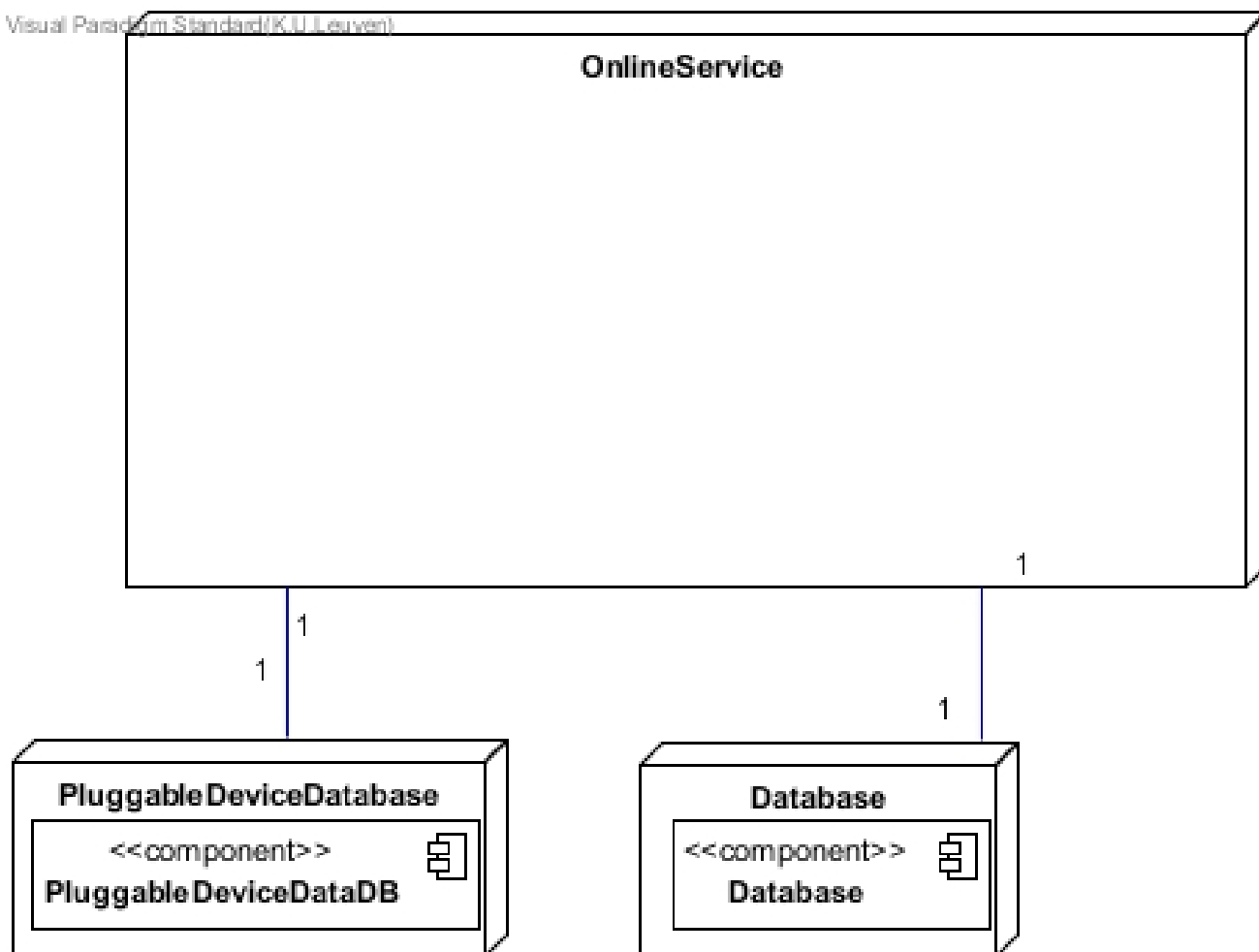


Figure 4.2: Primary diagram for the deployment view.

# 5. Scenarios

## Figures

5.1	Sensor data being processed by the system . . . . .	15
5.2	Subscribing to an application . . . . .	16
5.3	Applications issuing actuation commands . . . . .	17
5.4	Scenario . . . . .	18
5.5	Application crash . . . . .	19
5.6	Plugging in a new pluggable device (sensor or actuator) . . . . .	20
5.7	Detection and handling of communication channel failure . . . . .	21
5.8	Upgrading an application . . . . .	22
5.9	Sending actuation commands via a mobile app . . . . .	23

✓ **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 your 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

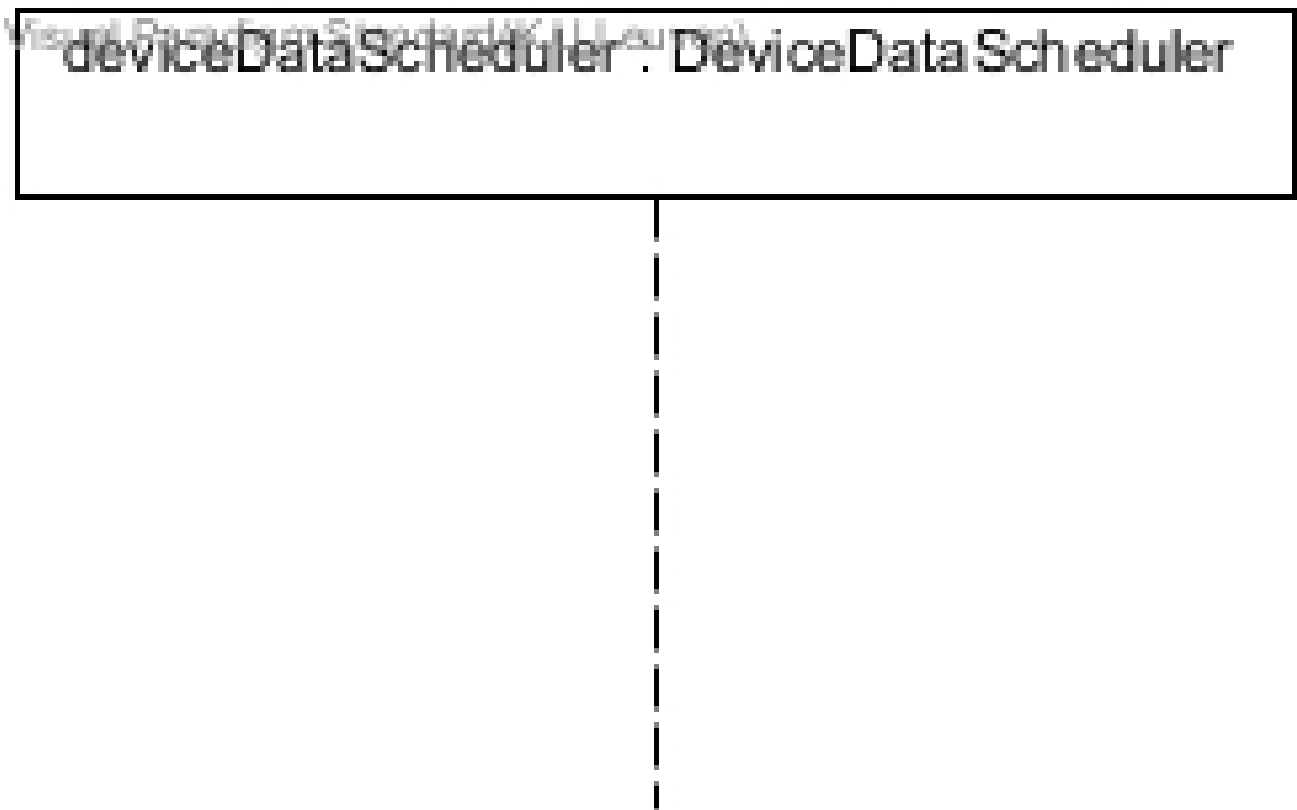


Figure 5.1: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.



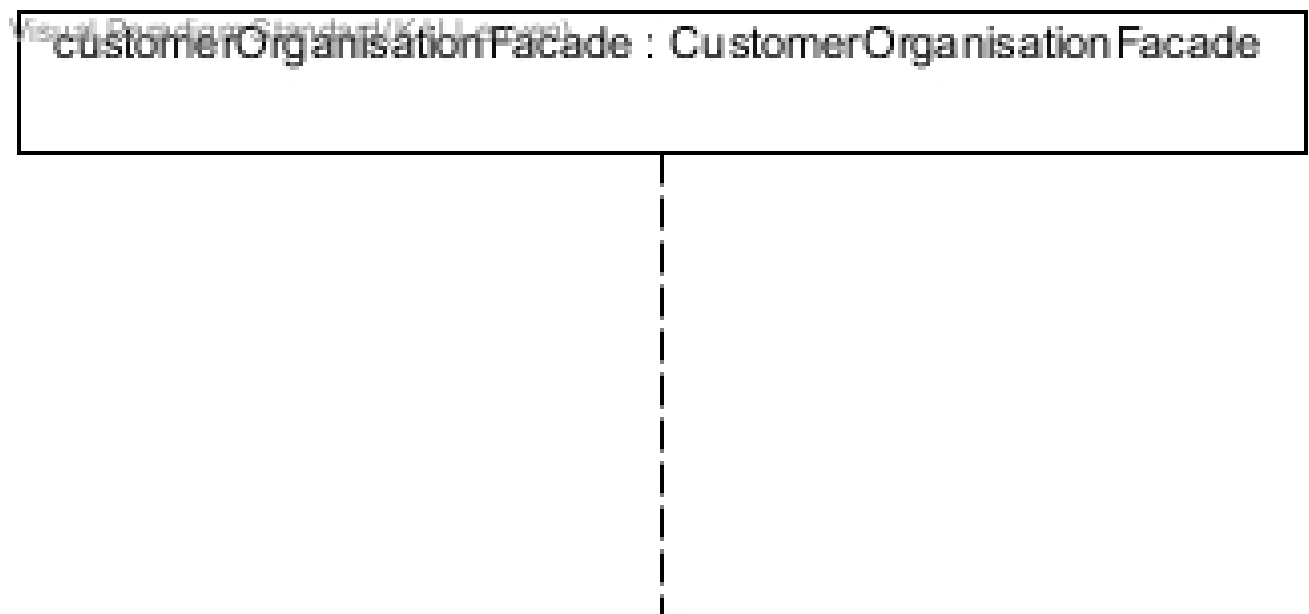


Figure 5.2: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

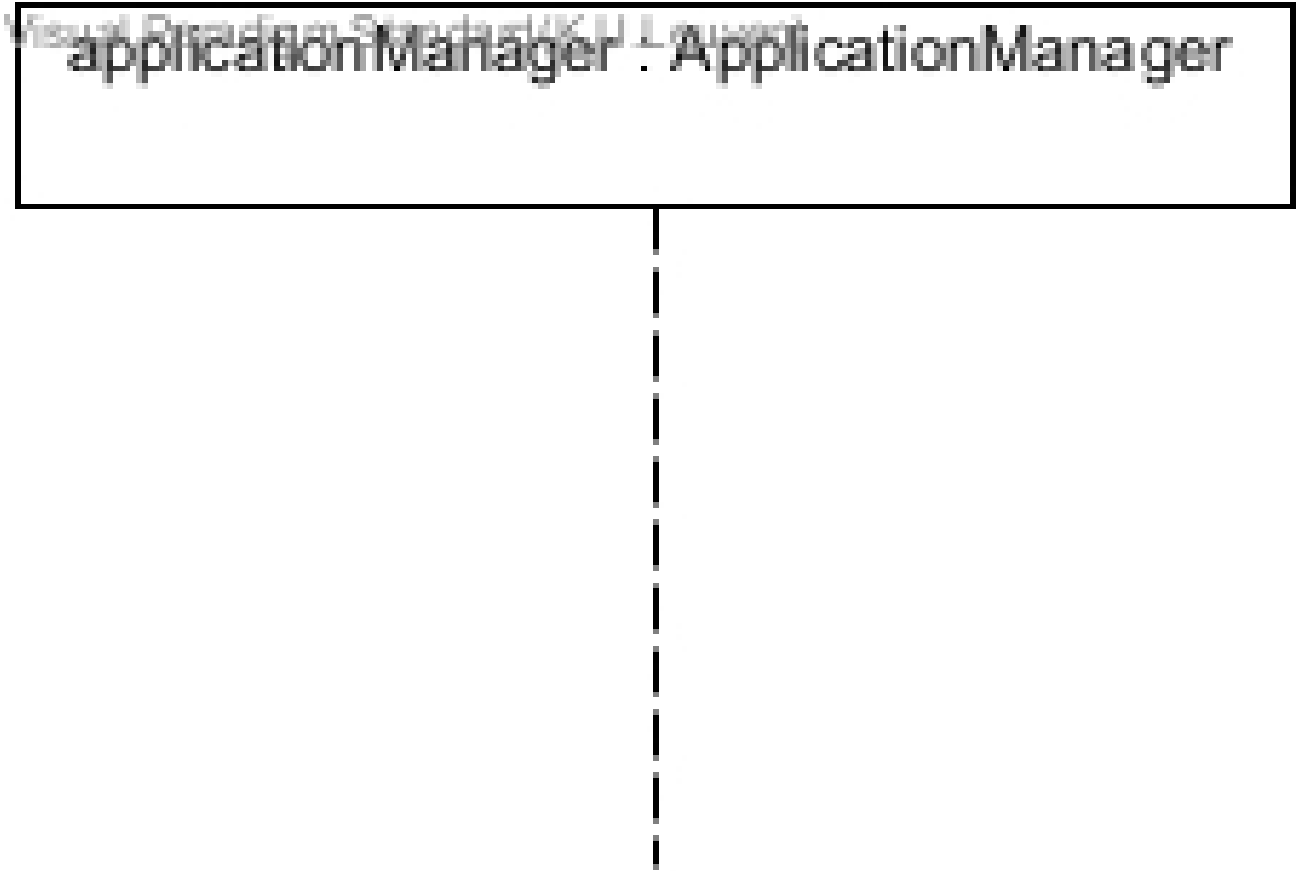


Figure 5.3: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

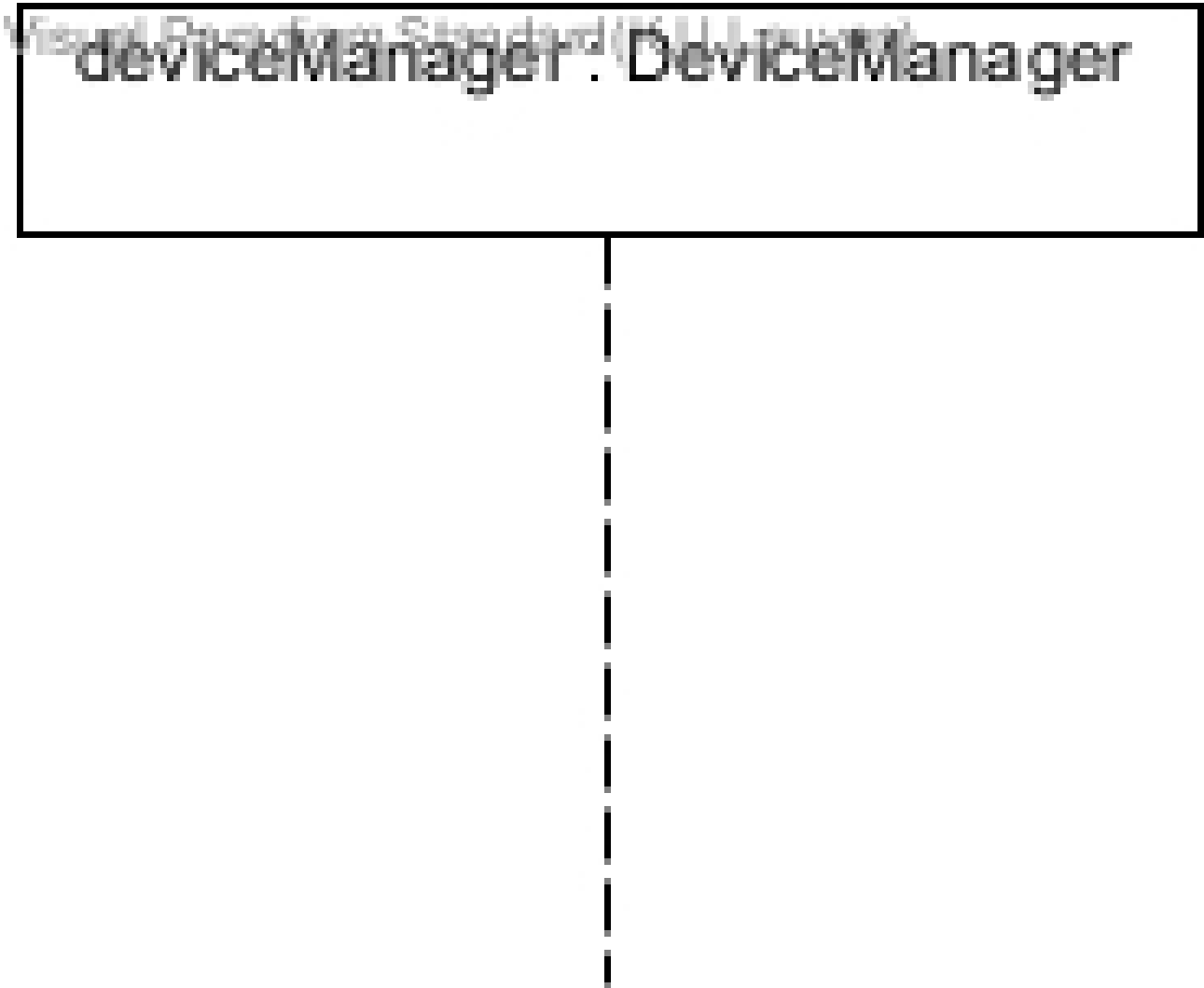


Figure 5.4: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

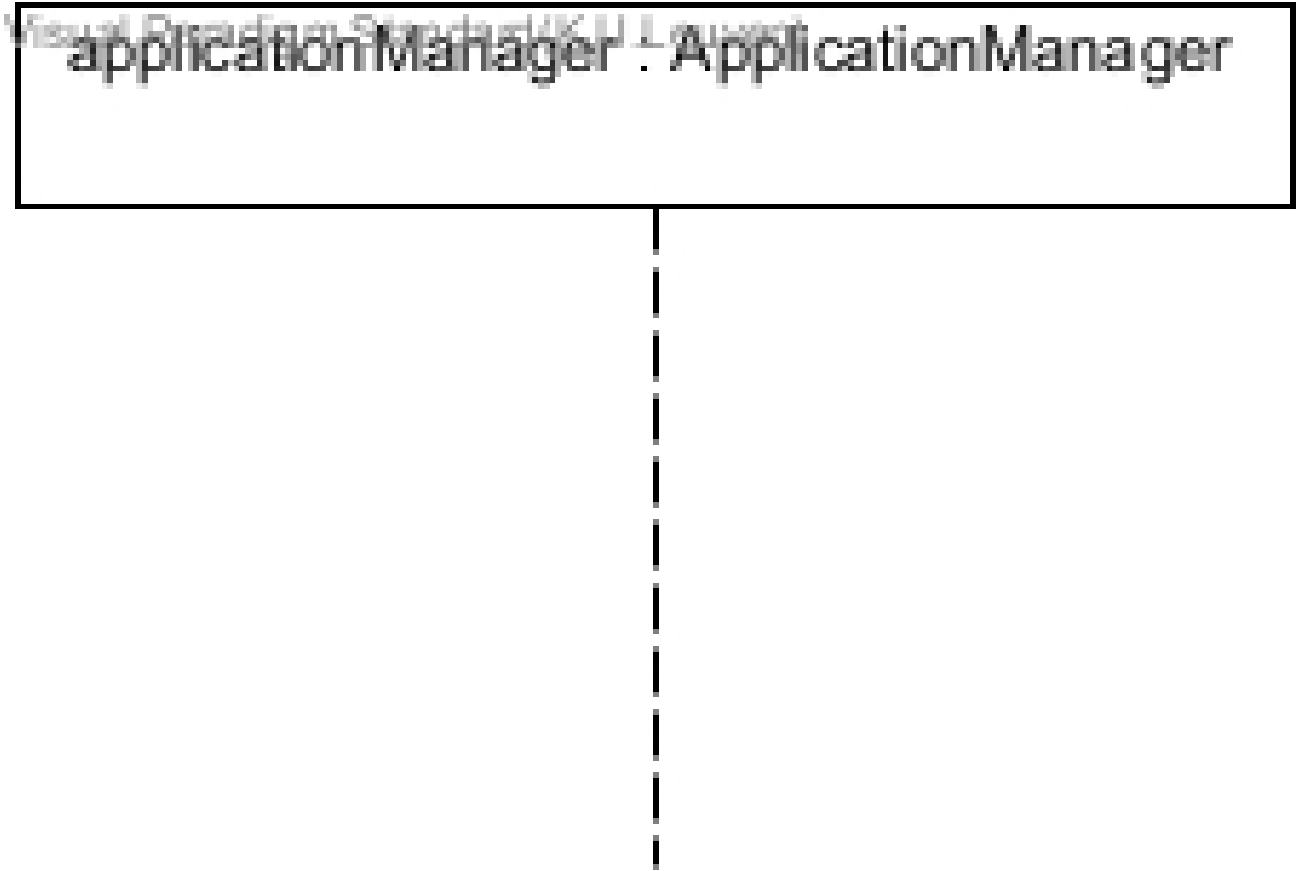


Figure 5.5: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

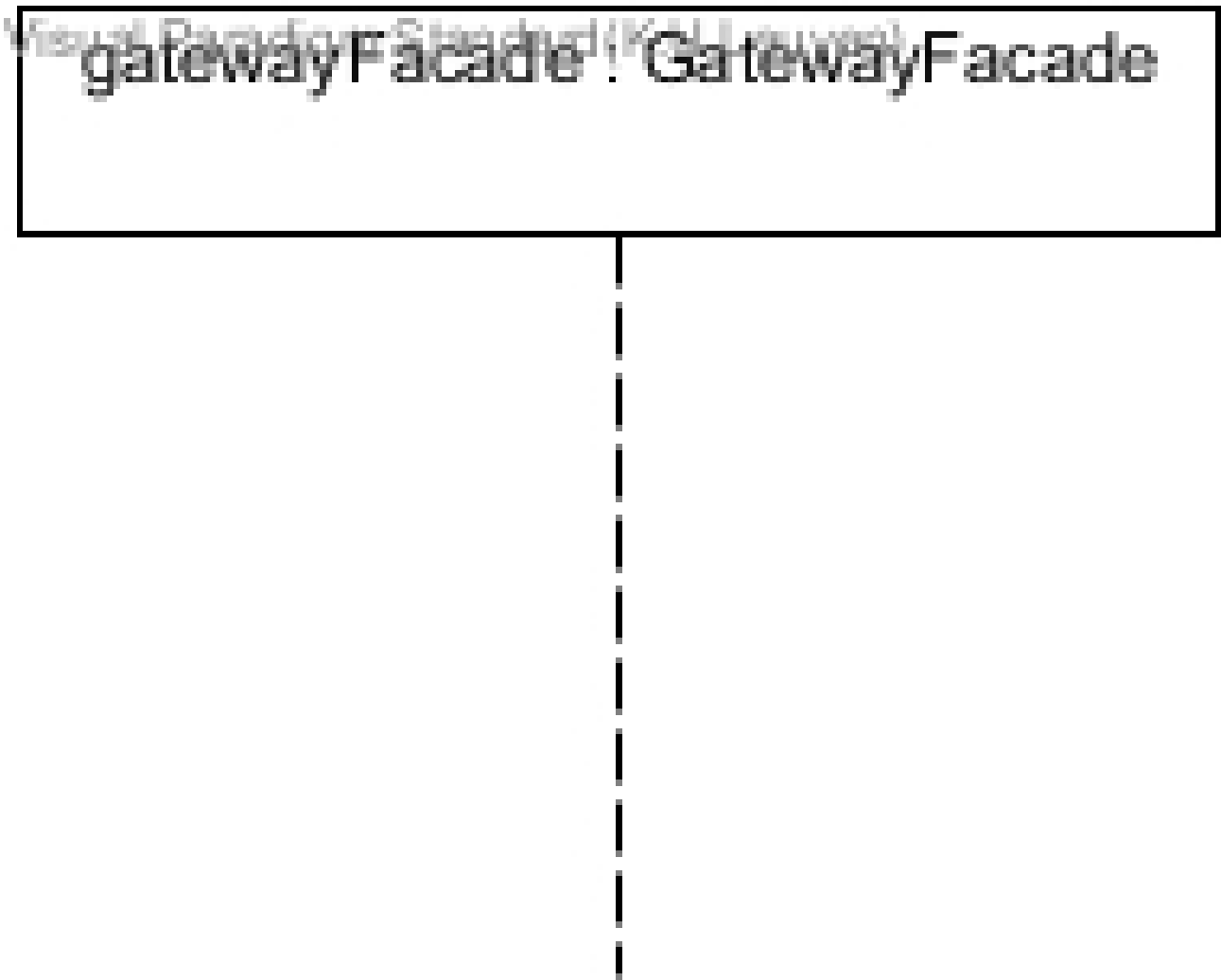


Figure 5.6: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

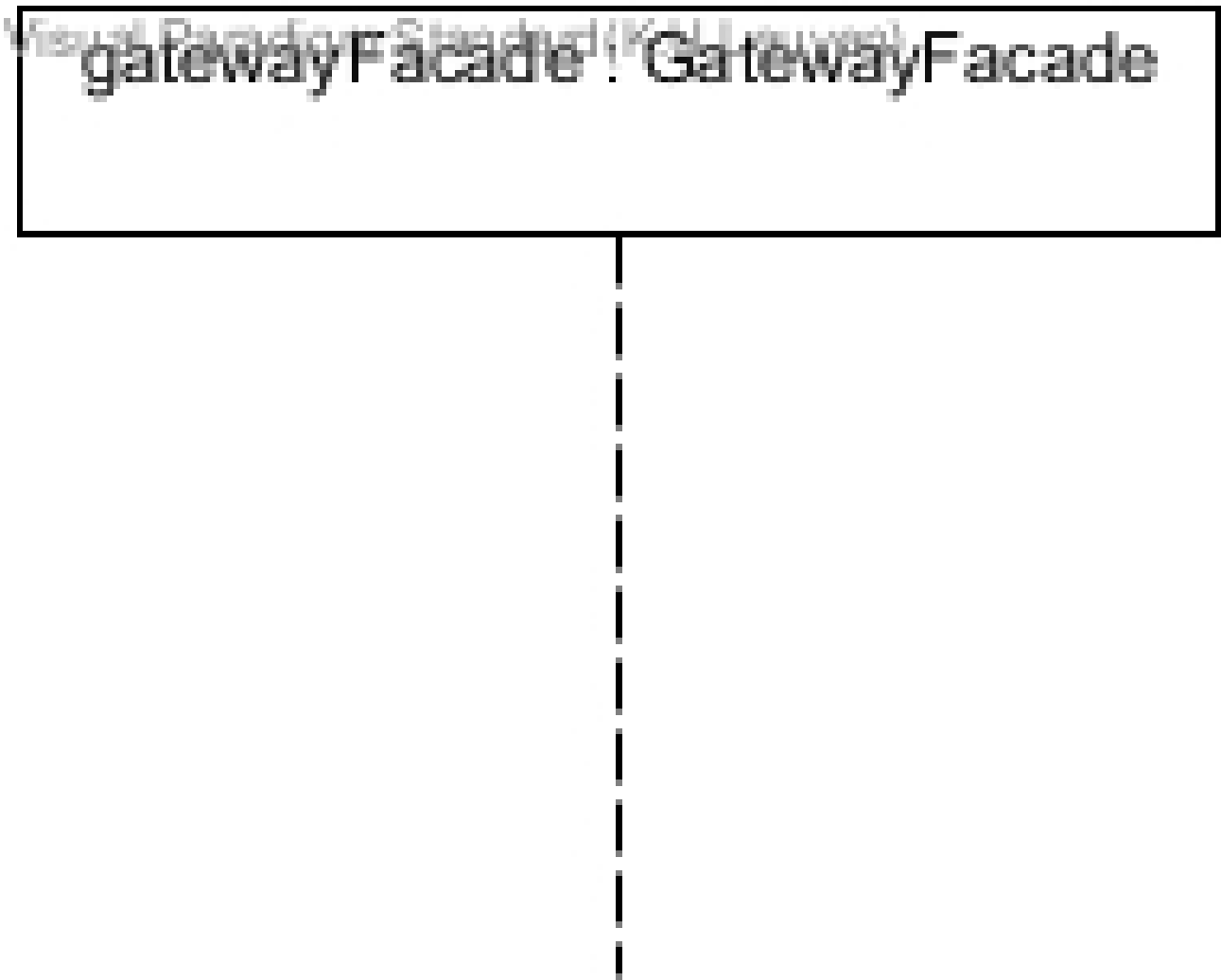


Figure 5.7: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

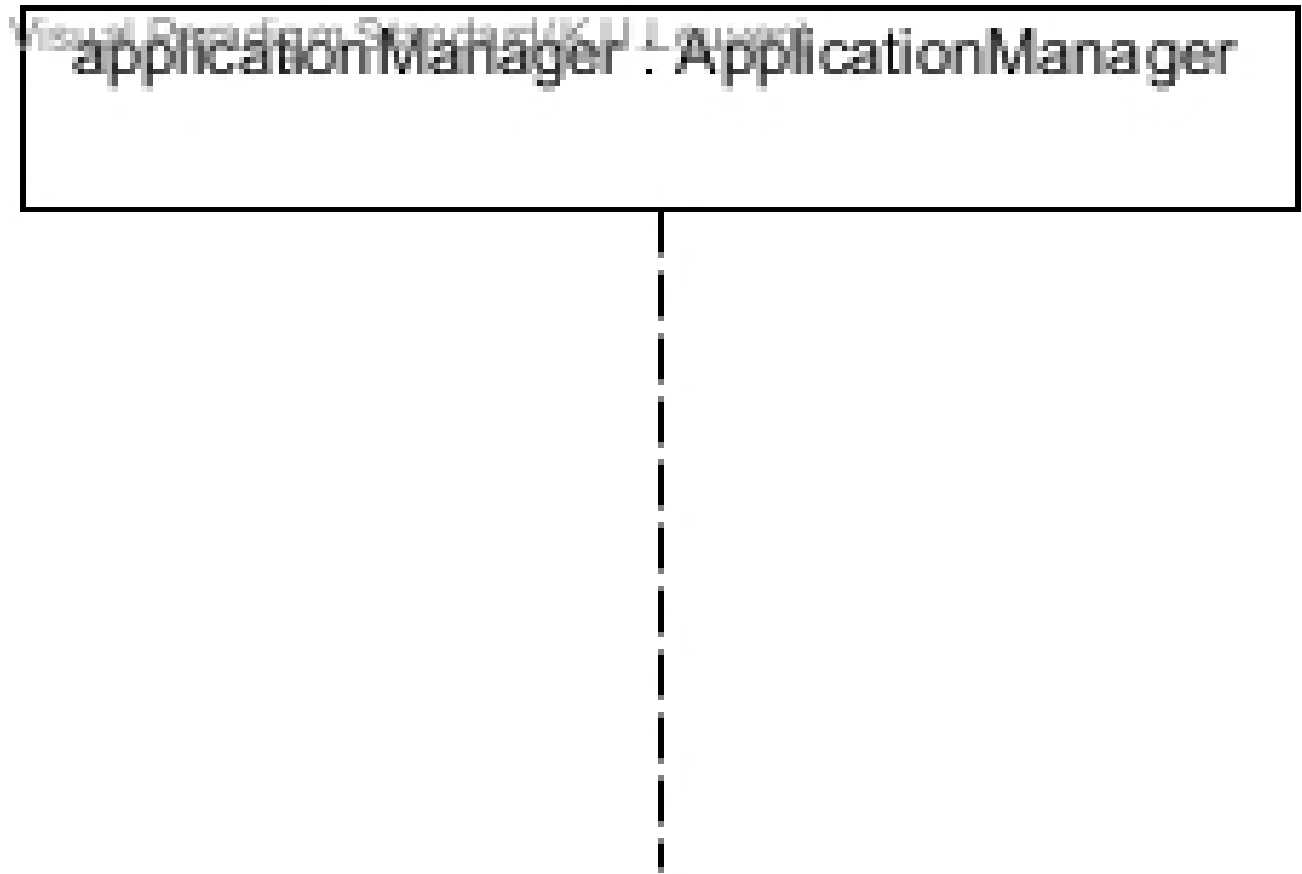


Figure 5.8: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.

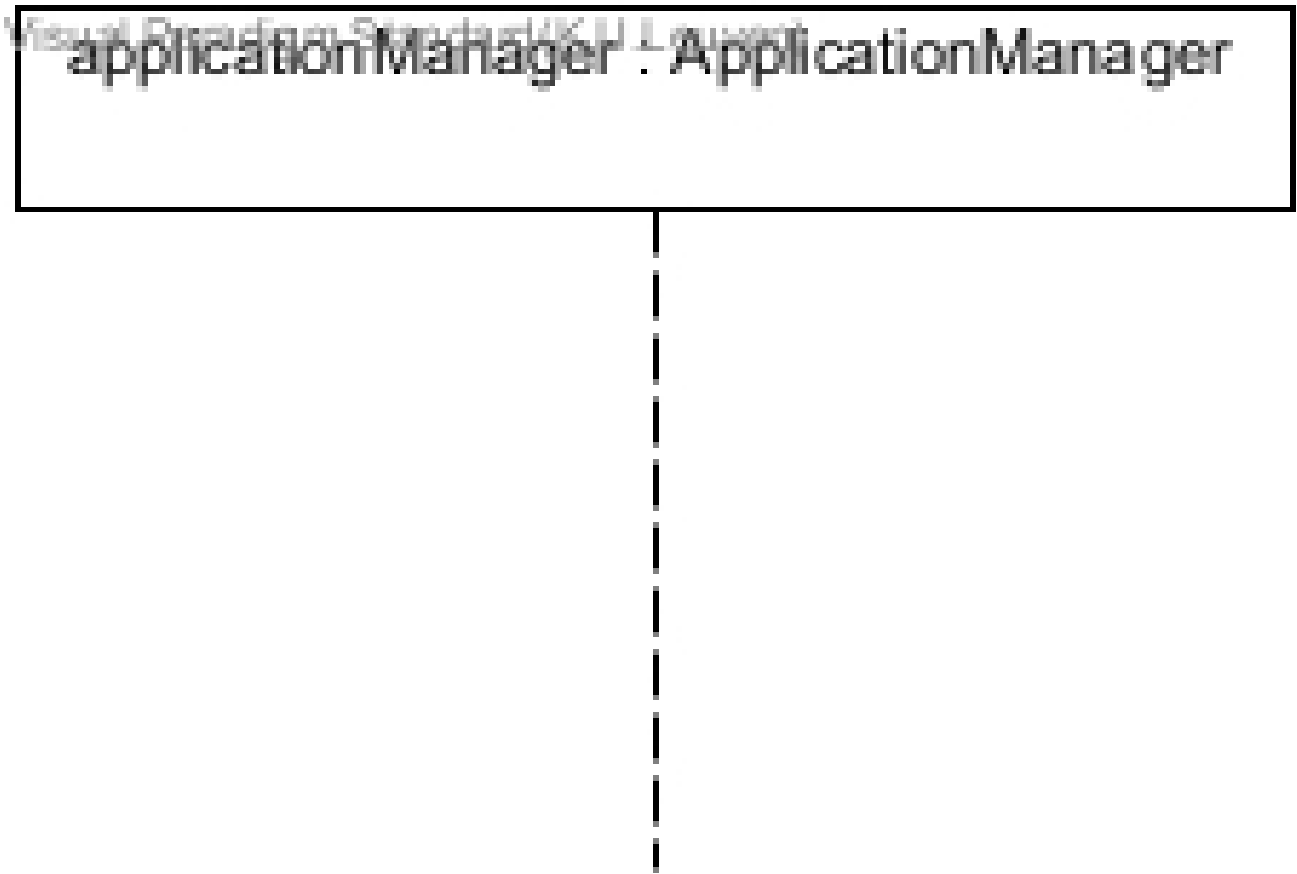


Figure 5.9: EXPLAIN WHAT HAPPENS IN THE SCENARIO.  
ADD COMMENTS.  
LINK TO OTHER RELEVANT SCENARIO'S.



## 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:** Undefined  
**Super-components:** None  
**Sub-components:** None  
**Provided interfaces:** ◊ AccessRightsMgmt  
**Required interfaces:** < AccessRightsMgmt

### 7.1.2 ApplicationContainer

**Responsibility:** Undefined  
**Super-components:** None  
**Sub-components:** None  
**Provided interfaces:** ◊ AppMgmt  
**Required interfaces:** None

### 7.1.3 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:** None  
**Provided interfaces:** ◊ AppMgmt, ◊ Apps, ◊ ForwardData, ◊ IOAppMgmt  
**Required interfaces:** < AppDeviceMgmt, < AppDeviceMgmt, < AppMgmt, < AppMgmt, < AppMgmt, < InvoiceMgmt, < Notify, < RequestData, < RoleMgmt, < TopologyMgmt

### 7.1.4 CustomerOrganisationClient

**Responsibility:** Undefined  
**Super-components:** None  
**Sub-components:** None  
**Provided interfaces:** None  
**Required interfaces:** < SubscriptionMgmt

### 7.1.5 CustomerOrganisationFacade

**Responsibility:** Undefined  
**Super-components:** None  
**Sub-components:** None  
**Provided interfaces:** ◊ SubscriptionMgmt  
**Required interfaces:** < Apps, < RoleMgmt, < SubscriptionMgmt, < TopologyMgmt

### 7.1.6 Database

**Responsibility:** General database for data. For example, storage of data about notifications.  
**Super-components:** None

**Sub-components:** None

**Provided interfaces:**   ⊖ AppMgmt,   ⊖ InvoiceMgmt,   ⊖ IOMgmt,   ⊖ NotificationMgmt,   ⊖ Other,  
                          ⊖ SubscriptionMgmt, ⊖ UserRoleMgmt

**Required interfaces:** None

### 7.1.7 DeviceDataConverter

**Responsibility:** The DeviceDataConverter is responsible for converting pluggable device data in the data processing subsystem.

**Super-components:** None

**Sub-components:** None

**Provided interfaces:**   ⊖ DataConversion

**Required interfaces:** None

### 7.1.8 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:**   < DeviceData, < ForwardData

### 7.1.9 DeviceDB

**Responsibility:** Everything related to pluggable devices, except for the pluggable device data

Gateways

Motes

Pluggable devices

Topology

Access rights

**Super-components:** None

**Sub-components:** None

**Provided interfaces:**   ⊖ AccessRightsMgmt,   ⊖ AppDeviceMgmt,   ⊖ DeviceMgmt,   ⊖ IODeviceMgmt,  
                          ⊖ TopologyMgmt

**Required interfaces:** None

### 7.1.10 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.

**Super-components:** None

**Sub-components:** None

**Provided interfaces:**   ⊖ DeviceMgmt

**Required interfaces:**   < DeviceMgmt

### 7.1.11 GatewayFacade

**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:** None

**Provided interfaces:** ⊖ AppDeviceMgmt, ⊖ AppMgmt, ⊖ DeviceData, ⊖ DeviceMgmt, ⊖ Heartbeat

**Required interfaces:** ⋖ AppMgmt, ⋖ AppMgmt, ⋖ DataConversion, ⋖ DeviceData, ⋖ DeviceMgmt, ⋖ DeviceMgmt, ⋖ DeviceMgmt, ⋖ Notify, ⋖ Other, ⋖ TopologyMgmt

### 7.1.12 GWApplicationContainer

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ⊖ AppMgmt

**Required interfaces:** None

### 7.1.13 InfrastructreOwnerClient

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** None

**Required interfaces:** ⋖ AccessRights

### 7.1.14 InfrastructureOwnerFacade

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ⊖ AccessRights

**Required interfaces:** ⋖ AccessRightsMgmt, ⋖ IOAppMgmt, ⋖ IOMgmt

### 7.1.15 InfrastructureOwnerManager

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ⊖ IOMgmt

**Required interfaces:** ⋖ IODeviceMgmt

### 7.1.16 InvoiceManager

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ⊖ InvoiceMgmt

**Required interfaces:** ⋖ InvoiceMgmt

### 7.1.17 MoteFacade

**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, ⋖ Heartbeat, ⋖ RequestData

### 7.1.18 NotificationDeliveryService

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** None

**Required interfaces:** None

### 7.1.19 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:** < NotificationDeliveryMgmt, < NotificationMgmt

### 7.1.20 OtherFunctionality1

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ◊ Other

**Required interfaces:** < Other, < Other

### 7.1.21 OtherFunctionality2

**Responsibility:** Undefined

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ◊ Other

**Required interfaces:** < Other, < Other

### 7.1.22 PluggableDeviceDataDB

**Responsibility:** Database dedicated to pluggable device data only.

**Super-components:** None

**Sub-components:** None

**Provided interfaces:** ◊ DeviceData, ◊ Other

**Required interfaces:** None

### 7.1.23 PluggableDeviceFacade

**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:** ⓘ SubscriptionManager

**Provided interfaces:** ◊ Actuate, ◊ Config, ◊ RequestData

**Required interfaces:** < DeviceData

### 7.1.24 SubscriptionManager

**Responsibility:** Undefined

**Super-components:** ⓘ PluggableDeviceFacade

Sub-components: None  
Provided interfaces: ∘ SubscriptionMgmt  
Required interfaces: ∙ Apps, ∙ SubscriptionMgmt

### 7.1.25 TopologyManager

Responsibility: Undefined  
Super-components: None  
Sub-components: None  
Provided interfaces: ∘ TopologyMgmt  
Required interfaces: ∙ TopologyMgmt

### 7.1.26 UserRolesManager

Responsibility: Undefined  
Super-components: None  
Sub-components: None  
Provided interfaces: ∘ RoleMgmt  
Required interfaces: ∙ UserRoleMgmt

## 7.2 Interfaces

### 7.2.1 AccessRights

Provided by: ⓘ InfrastructureOwnerFacade  
Required by: ⓘ InfrastructreOwnerClient  
Operations:

### 7.2.2 AccessRightsMgmt

Provided by: ⓘ AccessRightsManager  
Required by: ⓘ InfrastructureOwnerFacade  
Operations:

### 7.2.3 AccessRightsMgmt

Provided by: ⓘ DeviceDB  
Required by: ⓘ AccessRightsManager  
Operations:

### 7.2.4 Actuate

Provided by: ⓘ PluggableDeviceFacade  
Required by: ⓘ MoteFacade  
Operations:

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

### 7.2.5 AppDeviceMgmt

Provided by: ⓘ DeviceDB  
Required by: ⓘ ApplicationManager  
Operations:

### 7.2.6 AppDeviceMgmt

Provided by:  GatewayFacade

Required by:  ApplicationManager

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?

### 7.2.7 AppMgmt

Provided by:  Database

Required by:  ApplicationManager

Operations:

- `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
- `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 updateSubscription(Subscription subscription)`
  - Effect: Updates a subscription in the database (e.g. change state to 'disabled').
  - Created for: UC18

### 7.2.8 AppMgmt

Provided by:  ApplicationManager


Required by:  GatewayFacade

Operations:

- `void activateApplicationInstance(int applicationInstanceID)`
  - Effect: Activates a new instance of an application.
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.
- `void deactivateApplicationInstance(int applicationInstanceID)`
  - Effect: Deactivates a running instance of an application.
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.


### 7.2.9 AppMgmt


Provided by:  ApplicationContainer

Required by:  ApplicationManager

Operations:

### 7.2.10 AppMgmt

Provided by:  GatewayFacade

Required by:  ApplicationManager

Operations:

### 7.2.11 AppMgmt

Provided by: [¶](#) GWApplicationContainer

Required by: [¶](#) GatewayFacade

Operations:

### 7.2.12 Apps

Provided by: [¶](#) ApplicationManager

Required by: [¶](#) CustomerOrganisationFacade, [¶](#) SubscriptionManager

Operations:

### 7.2.13 Config

Provided by: [¶](#) PluggableDeviceFacade

Required by: [¶](#) MoteFacade

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

### 7.2.14 DataConversion

Provided by: [¶](#) DeviceDataConverter

Required by: [¶](#) GatewayFacade

Operations:

- `DeviceData convert(PluggableDeviceID plD, DeviceData data, string type)`
  - Effect: Converts pluggable device data into other pluggable device data that contains the same information in a different type.
  - Created for: M1: data processing subsystem should be extended with relevant data conversions

### 7.2.15 DeliveryMgmt

Provided by: [¶](#) NotificationHandler

Required by: None

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

### 7.2.16 DeviceData

Provided by: [¶](#) PluggableDeviceDataDB

Required by: [¶](#) DeviceDataScheduler


Operations:


- `List<DeviceData> getData(PluggableDeviceID plD, DateTime from, DateTime to)`
  - Effect: Returns data from a specific device in a certain time period.
  - Created for: P2: lookup queries
- `void rcvData(PluggableDeviceID plD, DeviceData data)`



- Effect: Sends pluggable device data to the DB to be stored.
- Created for: UC11, P2: storing new pluggable data

### 7.2.17 DeviceData

Provided by:  MoteFacade

Required by:  PluggableDeviceFacade

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** plD, **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

### 7.2.18 DeviceData


Provided by:  DeviceDataScheduler


Required by:  GatewayFacade

Operations:

- void rcvData(**PluggableDeviceID** plD, **DeviceData** data)
  - Effect: Sends pluggable device data to the scheduler to be processed.
  - Created for: UC11, P2: storing new pluggable data

### 7.2.19 DeviceData


Provided by:  GatewayFacade

Required by:  MoteFacade

Operations:

- void rcvData(**PluggableDeviceID** plD, **DeviceData** data)
  - Effect: Provides pluggable device data to the gateway (Initiated by the device).
- void rcvDataCallback(**PluggableDeviceID** plD, **DeviceData** data, int requestID)
  - Effect: Provides device data to the gateway (Callback of getDataAsync).

### 7.2.20 DeviceMgmt

Provided by:  MoteFacade

Required by:  GatewayFacade

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 setConfig(**PluggableDeviceID** plD, 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

### 7.2.21 DeviceMgmt

Provided by: [DeviceDB](#)

Required by: [GatewayFacade](#)

Operations:

### 7.2.22 DeviceMgmt

Provided by: [DeviceManager](#)

Required by: [GatewayFacade](#)

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 motelID, 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 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."

### 7.2.23 DeviceMgmt

Provided by: [GatewayFacade](#)

Required by: [DeviceManager](#)


Operations:

- `void deactivateApplicationInstance(int applicationInstanceID)`
  - Effect: Deactivates a certain application. This could happen when mandatory pluggable devices for the application are missing.
  - Created for: Av3: automatic suspension/reactivation of applications.
- `List<PluggableDeviceInfo> getConnectedDevices()`
  - Effect: Returns a list of information about devices that are connected to the gateway.
  - 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 pluggableDevicePersistentFailure()`
  - Effect: Lets the gateway know that a timer for pluggable device or mote has expired. This will generate a notification for an infrastructure owner.
  - Created for: Av3: The infrastructure owner should be notified of any persistent pluggable device or mote failures.
- `void pluggableDevicePluggedIn(Map<string, string> mInfo, PluggableDeviceID pID, PluggableDeviceType type)`
  - Effect: Notifies the gateway that a new pluggable device of the given type is connected to the mote.
- `void pluggableDeviceRemoved(PluggableDeviceID pID)`
  - Effect: Notifies the gateway that a pluggable device is removed.
- `void reactivateApplicationInstance(int applicationInstanceID)`

- Effect: Reactivate an application instance. This could happen automatically after a broken sensor has been replaced.
- Created for: Av3: automatic suspension/reactivation of applications.
- void setConfig(**PluggableDeviceID** plD, 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

### 7.2.24 ForwardData

Provided by:  ApplicationManager


Required by:  DeviceDataScheduler

Operations:

- List<int> getApplicationsForDevice(**PluggableDeviceID** plD)
  - Effect: Returns a list of application instances that can use the device with id "plD".
  - Created for: UC11: the system looks up the list of applications that use the pluggable device
- void rcvData(**PluggableDeviceID** plD, **DeviceData** data)
  - Effect: Sends pluggable device data to an application that wants to use it
  - Created for: UC11: system relays data to applications

### 7.2.25 Heartbeat


Provided by:  GatewayFacade


Required by:  MoteFacade

Operations:

- void heartbeat(Map<string, string> motelInfo, 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


### 7.2.26 InvoiceMgmt

Provided by:  Database

Required by:  InvoiceManager

Operations:

### 7.2.27 InvoiceMgmt


Provided by:  InvoiceManager

Required by:  ApplicationManager

Operations:

### 7.2.28 IOAppMgmt

Provided by:  ApplicationManager

Required by:  InfrastructureOwnerFacade

Operations:

### 7.2.29 IODeviceMgmt

Provided by: [🔗](#) DeviceDB

Required by: [🔗](#) InfrastructureOwnerManager

Operations:

### 7.2.30 IOMgmt

Provided by: [🔗](#) Database

Required by: None

Operations:

### 7.2.31 IOMgmt

Provided by: [🔗](#) InfrastructureOwnerManager

Required by: [🔗](#) InfrastructureOwnerFacade

Operations:

### 7.2.32 NotificationDeliveryMgmt

Provided by: None

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

### 7.2.33 NotificationMgmt

Provided by: [🔗](#) Database

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

### 7.2.34 Notify

Provided by: [🔗](#) NotificationHandler

Required by: [🔗](#) ApplicationManager, [🔗](#) GatewayFacade

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

### 7.2.35 Other

Provided by: [PluggableDeviceDataDB](#)

Required by: [OtherFunctionality1](#), [OtherFunctionality2](#)

Operations:

### 7.2.36 Other

Provided by: [Database](#)

Required by: [OtherFunctionality1](#), [OtherFunctionality2](#)

Operations:

### 7.2.37 Other

Provided by: [OtherFunctionality1](#), [OtherFunctionality2](#)

Required by: [GatewayFacade](#)

Operations:

### 7.2.38 RequestData

Provided by: [PluggableDeviceFacade](#)

Required by: [MoteFacade](#)

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`).

### 7.2.39 RequestData

Provided by: [DeviceDataScheduler](#)

Required by: [ApplicationManager](#)

Operations:

- `List<DeviceData> getData(int applicationID, PluggableDeviceID pID, DateTime from, DateTime to)`
  - Effect: Requests data from a specific device in a certain time period.
  - Created for: P2: requests from applications

### 7.2.40 RoleMgmt

Provided by: [UserRolesManager](#)

Required by: [ApplicationManager](#), [CustomerOrganisationFacade](#)

Operations:

### 7.2.41 SubscriptionMgmt

Provided by: [Database](#)

Required by: [SubscriptionManager](#)

Operations:

### 7.2.42 SubscriptionMgmt

Provided by: [CustomerOrganisationFacade](#)

Required by: [CustomerOrganisationClient](#)

Operations:

### 7.2.43 SubscriptionMgmt

Provided by: [SubscriptionManager](#)

Required by: [CustomerOrganisationFacade](#)

Operations:

### 7.2.44 TopologyMgmt

Provided by: [TopologyManager](#)

Required by: [ApplicationManager](#), [CustomerOrganisationFacade](#), [GatewayFacade](#)

Operations:

### 7.2.45 TopologyMgmt

Provided by: [DeviceDB](#)

Required by: [TopologyManager](#)

Operations:

### 7.2.46 UserRoleMgmt

Provided by: [Database](#)

Required by: [UserRolesManager](#)

Operations:

## 7.3 Exceptions

No exceptions

## 7.4 Data types

- **ApplicationInstance:**  
Attributes: int id, int status, int customerOrganisationID  
Contains information on an application instance.
- **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.
- **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 communicationChannelID 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:**  
Denotes the type of a pluggable device.
- **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.

# 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 occurred 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 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 **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 functionality 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`: /
- `MoteFacade`: UC14
- `NotificationHandler`: UC15

**Decomposition** Figure A.1 shows the components resulting from the decomposition in this run.

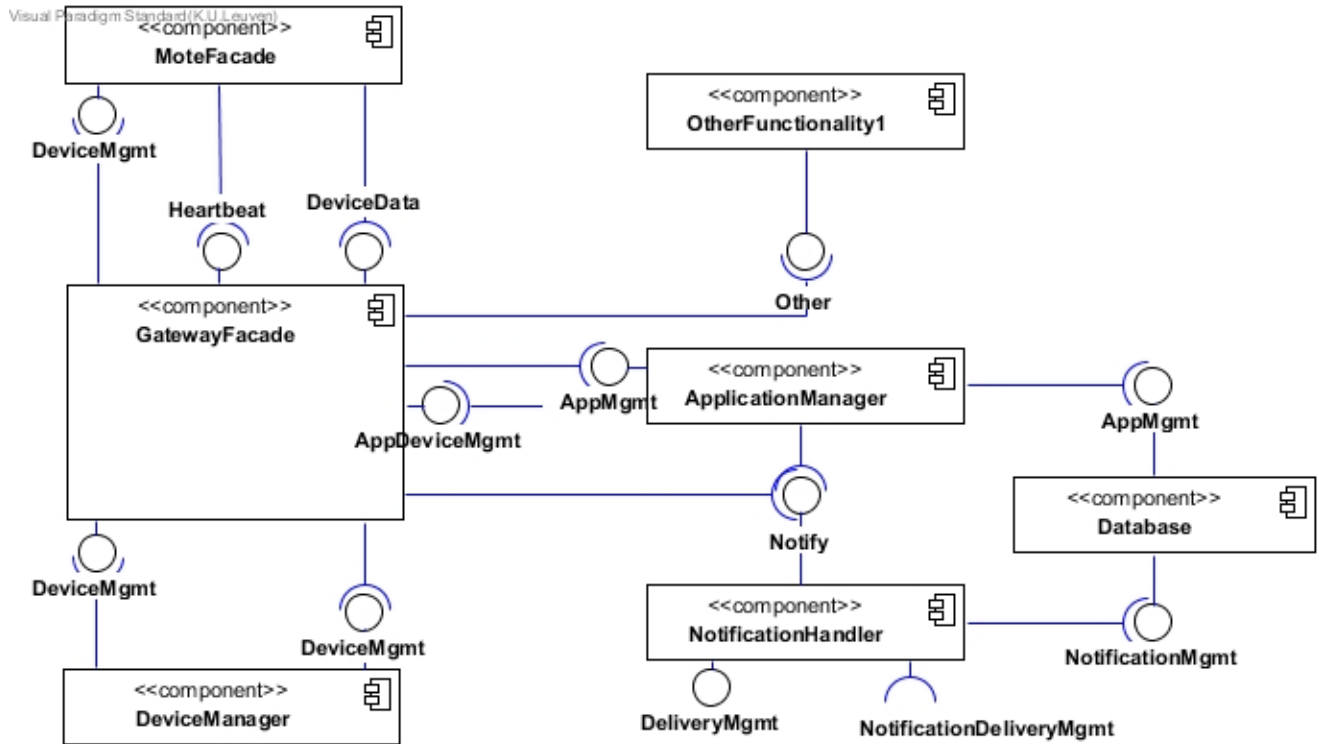


Figure A.1: Component-and-connector diagram of this decomposition.

**Deployment** Figure A.2 shows the allocation of components to physical nodes.

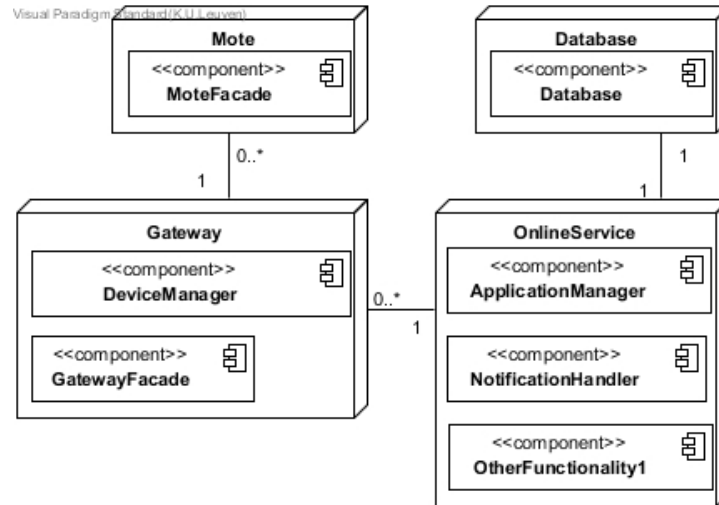


Figure A.2: Deployment diagram of this decomposition.

### 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

#### MoteFacade

- 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

## MoteFacade

- *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 satisfy 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 `Database` 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:

- *PluggableDeviceFacade*: 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
- `PluggableDeviceFacade`: 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`

#### MoteFacade

- `DeviceData`



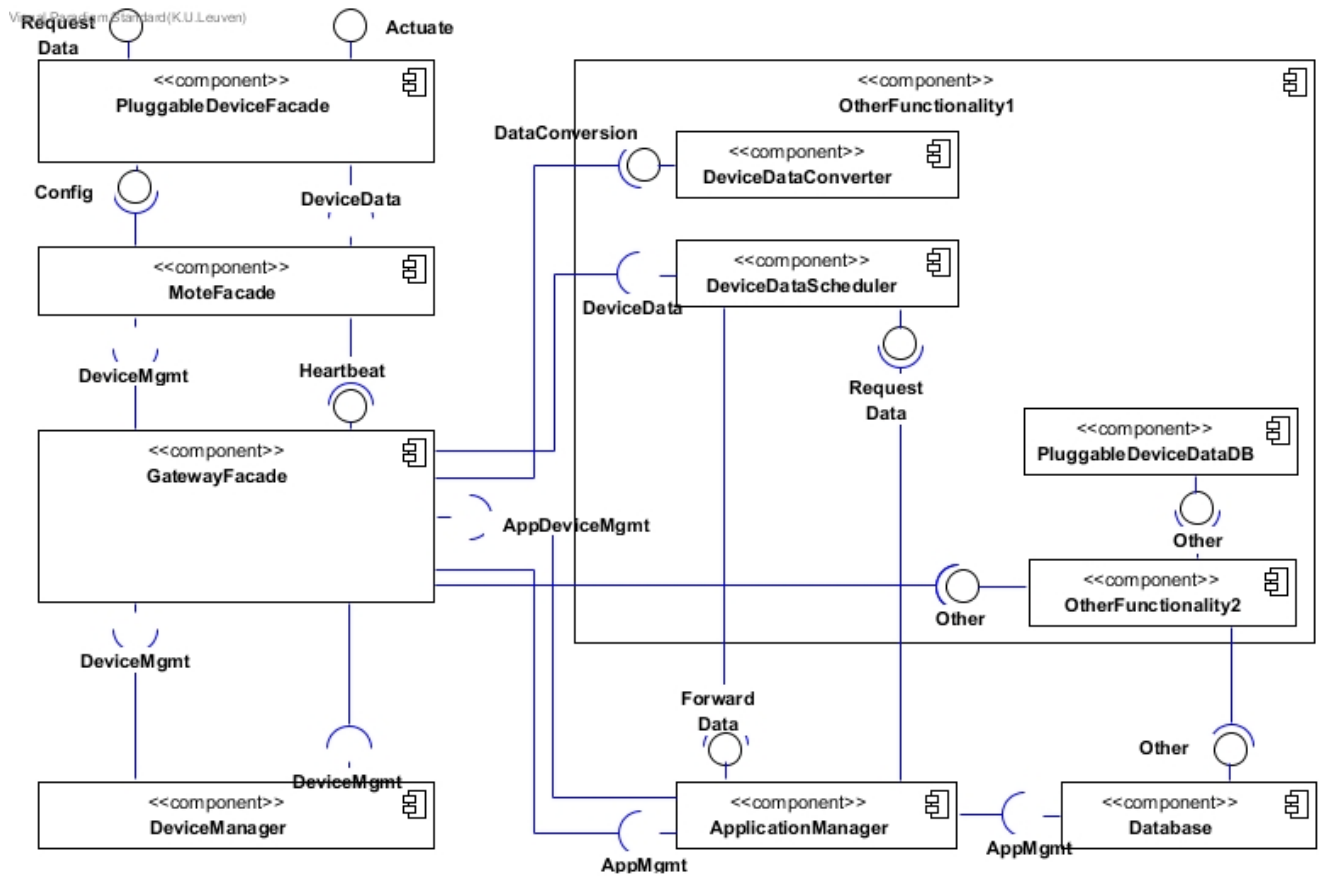


Figure A.3: Component-and-connector diagram of this decomposition.

#### PluggableDeviceFacade

- 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

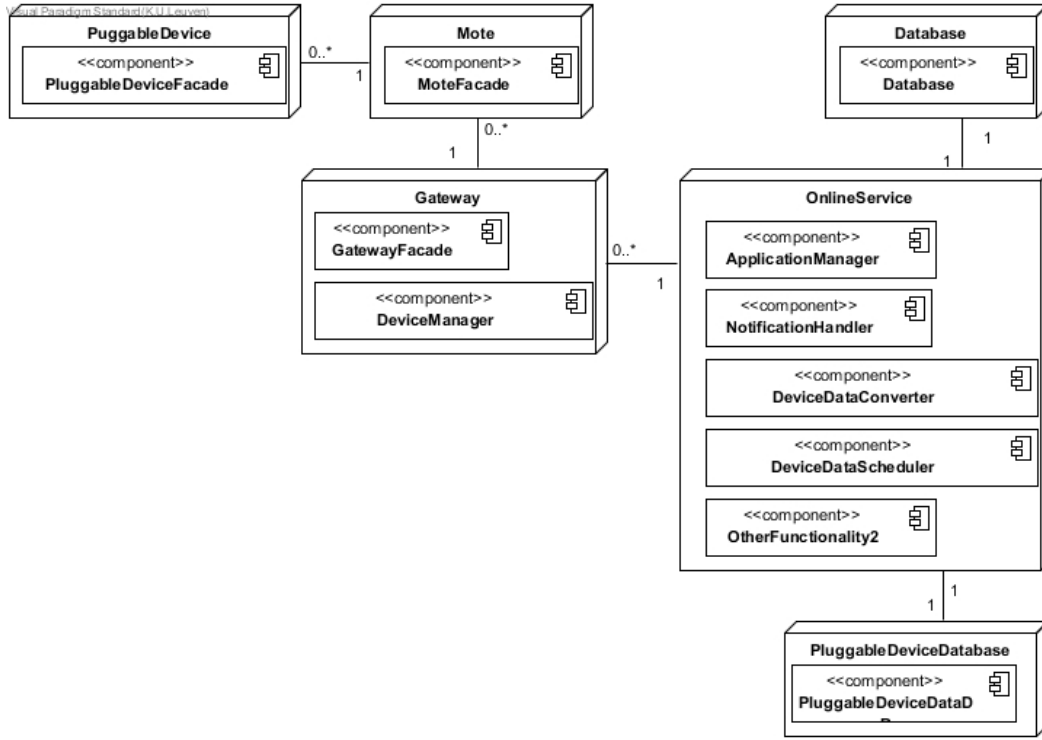


Figure A.4: Deployment diagram of this decomposition.

#### 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

##### Database

- None

##### GatewayFacade

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

## MoteFacade

- *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

## PluggableDeviceFacade

- *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 (Elements/Subsystem to decompose/expand)

### 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  
Short description
- *UC6*: Insert a pluggable device into a mote  
Short description
- *UC9*: Configure pluggable device access rights  
Short description
- *UC10*: Consult and configure topology  
Short description
- *UC17*: Activate an application  
Short description
- *UC19*: Subscribe to application  
Short description

### 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.

**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. Important info related to gateways: GatewayID (new class), infrastructureOwnerID, IPAddress, status (active/inactive), location (in topology table) gateway registers with online service: sets status to 'active' and updates IPAddress if different  
GatewayFacade -> DeviceDB: interface DeviceMgmt: registerGateway(GatewayID id, IPAddress address)

**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 - 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

UC4: Remark : The mote is pre-con

gured to connect to a specific gateway by the hardware manufacturer. This linking process is out of scope for this assignment. Likewise, the automatic assignment of an IPv6 address to the mote is out of scope.

if new mote: for step 2., we can use the heartbeat system. heartbeat is sent from mote to gateway, the DeviceManager in the gateway notices that this is from a new mote and starts the registerMote procedure

FOR RATIONALE: The IPAddress of the mote can be parsed out of the 6lowpan header in the heartbeat messages

ALTERNATIVE IS GatewayFacade: registerMote, but this is more work (battery power and implementation)

FOR RATIONALE: be careful with int moteID. when we register, we send MoteInfo to the DB, DB returns a DIFFERENT int moteID we use the other moteID in the rest of the system

if reintroduced mote: 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. The attached pluggable devices are automatically initialised and configured with their last known access rights. Already done by DeviceManager, it detects the devices, updates DB, and configures the devices

**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

UC6: insert a pluggable device into a mote mote is already installed

when device is plugged into a mote: 2. The system receives a message from the mote, informing it of the new pluggable device. This message specifies the identity

and type of the new pluggable device. => for registration, we can use the heartbeat system. heartbeat is sent from mote to gateway, the DeviceManager in the gateway notices that there is a new pluggable device Status of new pluggable devices will be 'uninitialised' by default in the DB, and 'unplaced' in topology New pluggable devices get the location of the mote in the topology, but have status unplaced

if the device is a known previously active device (ON THE SAME MOTE):

â marks the pluggable device as âactiveâ:

â updates the topology:

â configures the pluggable device with the last known access rights: RATIONALE: nothing needs to happen here, permission information will just not be used if the device is inactive if the device is reactivated, the permissions are already there

\* configures the pluggable device with the last known configuration:

â checks and activates applications which can now execute again: RATIONALE: gateway sends id of IO instead of fetching customers of IO. the online service will do that extra work = less work for gateway.

\* send notification

UC9: configure pluggable device access rights Map<int, AccessRights> : maps customerOrganisationIDs to their AccessRights

1. The primary actor indicates that they want to configure the access rights to pluggable devices.
2. The system retrieves the list of pluggable devices associated with the primary actor and presents these to the primary actor.

3. The primary actor indicates for which pluggable device they want to configure the access rights.
4. The system retrieves all customer organisations associated with the primary actor and presents these to the primary actor, thereby indicating which of these customer organizations already have access to the pluggable device.
5. The system asks to indicate which of these should have access to the pluggable device.
6. The primary actor selects the customer organisations that may use the pluggable device and submits the selection.
7. The system updates the roles of the selected customer organisations, giving them access rights to the pluggable device. checks and activates 'inactive' applications for the customer organisations with updated access rights (Include: UC17: Activate an application). checks for applications that require deactivation because of the unavailability of pluggable devices

**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 subscription 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

UC17: Activate an application ApplicationManager is triggered by something else do to the following: activateApplication(int applicationInstanceID) could be triggered because of: new pluggable device detected, new subscription, changed topology, new version of application

1. The system checks that all mandatory roles have been assigned to end-users. // returns true if all mandatory user roles for the application have been assigned to users // can find the customer organisation through the ApplicationInstance, because instances are for cust orgs
2. If all mandatory roles have been assigned, the system checks that all necessary pluggable devices are available in the topology. // pluggable device status has to be 'active' in the DB, and status has to be 'placed/available' in the topology
3. If all necessary pluggable devices are available, the system activates all necessary parts of the application on gateways and in the Online Service. FOR RATIONALE: we add ApplicationContainer component that will contain application INSTANCES in some kind of sandbox environment. The container can run/pause/stop these instances
4. The system marks the application as 'active' and updates the billing information.
5. The system sends a notification to the customer organisation subscribed to the application, to inform them that the application is running (Include: UC15: Send notification).

6. The system sends an SMS or e-mail to the end-users that were assigned roles. Possibly, this contains instructions for the end-user on how to install a mobile app linked to the application.

ALTERNATIVE SCENARIOS: 2a. If end-users are not assigned to each mandatory role, the application is added as an 'inactive' application and the subscription info is set accordingly. The system notifies the customer organisation subscribed to the application (Include: UC15: Send notification). The use case ends. This happens if checkMandatoryUserRoles returned false. FOR

RATIONALE: the UC is called "activate application", thus we assume the system already contains the application instance and subscription information, but the instance is 'inactive' at the start of the UC => OK  
 3a. If not all pluggable devices necessary for the application are available, the application is added as an 'inactive' application and the subscription info is set accordingly. The system notifies the customer organisation subscribed to the application (Include: UC15: Send notification). The use case ends. This happens if arePluggableDevicesActive returned false. Same as 2a. => OK

UC19: Subscribe to application A customer organisation subscribes to an application via the dashboard.

REMARKS: \*) This use case concerns both subscribing to a new application, as well as upgrading an existing application to a newer version (if this is not an automatic update, as indicated in UC22 : Upload an

application). \*) Roles in an application can be used to indicate which end-user(s) should be notified in case of specific events. \*) It is possible to reassign end-users to roles later on. For simplicity, no separate use case is provided for this.

1. The primary actor indicates they want to subscribe to an application.
2. The system looks up the available applications (i.e., those to which the primary actor is not yet subscribed, or applications for which a newer version is available that requires a new subscription).
3. The system presents the available applications to the primary actor, e.g. as a list or table, thereby indicating whether it is a new application or an update.
4. The primary actor selects the desired application.
5. The system checks which topology configuration and selection of pluggable devices for the application are necessary (e.g., to indicate in which rooms a heating control application should run) and presents a topology allowing the primary actor to indicate their configuration.
6. The primary actor carries out the topology configuration.
7. The system loads the (mandatory and optional) roles to which end-users have to be (or can be) assigned for that application. UserRole contains isMandatory member
8. The system loads the end-users associated with the primary actor, presents them to the primary actor, and requests to assign end-users to the roles in the application.
9. The primary actor provides end-users for the roles. can users have multiple roles? => no, only one
10. The system asks the primary actor if this application should be considered a critical application (cf. P2 : Requests to the pluggable data database).
11. The primary actor indicates the criticality of the application.
12. The system registers the subscription and criticality of the application.
13. If the selected application is a newer version of an application to which the primary actor was previously subscribed, the primary actor is automatically unsubscribed from that earlier version (cf. step 4 or UC20 : Unsubscribe from application).
14. The system activates the application (Include: UC17 : Activate an application).

### 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.

- Component: (Relevant QA or UC)

### 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.

#### ApplicationManager

- AppMgmt, last defined in section A.3.4
  - void checkApplicationsForActivationForInfrastructureOwner(int infrastructureOwnerID)
    - \* Effect: Checks and activates applications which can now execute again. Finds relevant applications through the customers of the given infrastructureOwnerID.
    - \* Created for: UC17
    - UC6.3 - reintroduced device
- IOAppMgmt
  - void checkApplicationsForActivationForCustomerOrganisations(List<int> customerOrganisationIDs)



- \* Effect: Checks and activates applications which can now execute again. Finds relevant applications through the given customerOrganisationIDs.
- \* Created for: UC9.7 - checks and activates 'inactive' applications for the customer organisations with updated access rights  
UC17
- void checkApplicationsForDeactivationForCustomerOrganisations(List<int> customerOrganisationIDs)
  - \* Effect: Checks and deactivates applications which that require deactivation because of the unavailability of pluggable devices. Only checks applications that are linked to the given customerOrganisationIDs.
  - \* Created for: UC9.7 - checks for applications that require deactivation  
UC18

## DeviceDB

- DeviceMgmt

- int addMote(MoteInfo mote, int gatewayID, IPAddress moteIPAddress)
  - \* Effect: Schedules a DB query to add a new mote to the .
  - \* Created for: UC4.3
- void reactivateMote(int moteID)
  - \* Effect: Schedules a DB query to change the status of the mote to active.
  - \* Created for: UC4.3 - reintroduced mote
- void addDevice(PluggableDeviceID id, PluggableDeviceType type, Map<String, String> defaultConfig)
  - \* Effect: Schedules a DB query to add a new pluggable device to the .
  - \* Created for: UC6.3
- void reactivateDevice(PluggableDeviceID id)
  - \* Effect: Schedules a DB query to change the status of the pluggable device to active.
  - \* Created for: UC6.3 - reintroduced device
- Map<String, String> getConfigDB(PluggableDeviceID pID)
  - \* Effect: Schedules a DB query to get the last set configuration of a pluggable device.
  - \* Created for: UC6.3 - reintroduced device

- TopologyMgmt

- void addMoteInTopology(int moteID, int infrastructureOwnerID, int gatewayID)
  - \* Effect: Schedules a DB query to add a new mote in the topology.
  - \* Created for: UC4.3
- void reactivateMoteInTopology(int moteID)
  - \* Effect: Schedules a DB query to change the status of the mote in the topology to 'placed'.
  - \* Created for: UC4.3 - reintroduced mote
- void addDevice(PluggableDeviceID id, int moteID)
  - \* Effect: Schedules a DB query to add a new pluggable device to the topology.
  - \* Created for: UC6.3
- void reactivateDevice(PluggableDeviceID id)
  - \* Effect: Schedules a DB query to change the status of the pluggable device to 'placed'.
  - \* Created for: UC6.3 - reintroduced device

## GatewayFacade

- DeviceMgmt, last defined in section A.3.4

- int addMote(MoteInfo mote, int gatewayID, IPAddress moteIPAddress)
  - \* Effect: Sends a DB query to the DeviceDataScheduler to add the new mote.  
Sends a request to the TopologyManager to add the new mote to the topology of the infrastructure owner.

- \* Created for: UC4.3
- `void reactivateMote(int moteID)`
  - \* Effect: Sends a DB query to the DeviceDataScheduler to reactivate the mote.  
Sends a request to the TopologyManager to update the mote's topology status.
  - \* Created for: UC4.3 - reintroduced mote
- `void addDevice(PluggableDeviceID id, PluggableDeviceType type, Map<String, String> defaultConfig, int moteID)`
  - \* Effect: Sends a DB query to the DeviceDataScheduler to add the new device. If the device already exists, removes the data first. The device's status is 'uninitialised' by default.  
Sends a request to the TopologyManager to add the new device to the topology and link it to a mote.
  - \* Created for: UC6.3
- `void reactivateDevice(PluggableDeviceID id)`
  - \* Effect: Sends a DB query to the DeviceDataScheduler to reactivate the device. Marks the pluggable device as 'active'.  
Sends a request to the TopologyManager to update the device's topology status.
  - \* Created for: UC6.3 - reintroduced device
- `void notifyNewMote(int moteID, MoteInfo mote)`
  - \* Effect: Lets the gateway know that a new mote has been installed in the system. This will generate a notification for an infrastructure owner.
  - \* Created for: UC4.4
- `void notifyReactivatedMote(int moteID, MoteInfo mote)`
  - \* Effect: Lets the gateway know that a mote has been reactivated in the system. This will generate a notification for an infrastructure owner.
  - \* Created for: UC4.4
- `void notifyNewPluggableDevice(PluggableDeviceID id, PluggableDeviceType type, Map<String, String> defaultConfigurations)`
  - \* Effect: Lets the gateway know that a new device has been installed in the system. This will generate a notification for an infrastructure owner.
  - \* Created for: UC6.3
- `void notifyReactivatedPluggableDevice(PluggableDeviceID id, PluggableDeviceType type, Map<String, String> defaultConfigurations)`
  - \* Effect: Lets the gateway know that a pluggable device has been reactivated in the system. This will generate a notification for an infrastructure owner.
  - \* Created for: UC6.3
- `void setConfig(PluggableDeviceID pID, Map<String, String> config)`
  - \* Now also used for UC6.3 - reintroduced device
- `Map<String, String> getConfigDB(PluggableDeviceID pID)`
  - \* Effect: Sends a DB query to the DeviceDataScheduler to get the last set configuration of a pluggable device.
  - \* Created for: UC6.3 - reintroduced device
- AppMgmt, last defined in section A.3.4
  - `void checkApplicationsForActivationForInfrastructureOwner(int infrastructureOwnerID)`
    - \* Effect: Forwards this request to the ApplicationManager. Used by DeviceManager after connecting a pluggable device.
    - \* Created for: UC17
    - UC6.3 - reintroduced device

## MoteFacade

- DeviceMgmt, last defined in section A.3.4

- `void setConfig(PluggableDeviceID pID, Map<String, String> config)`  
     \* Now also used for UC6.3 - reintroduced mote

### PluggableDeviceFacade

- Config, last defined in section A.3.4
  - `boolean setConfig(Map<String, String> config)`  
     \* Now also used for UC6.3 - reintroduced mote

### TopologyManager

- TopologyMgmt
  - `void addMote(int moteID, int infrastructureOwnerID, int gatewayID)`  
     \* Effect: Sends a DB query to the DeviceDataScheduler to add a mote to the topology. Its status is 'unplaced' by default.  
     \* Created for: UC4.3
  - `void reactivateMote(int moteID)`  
     \* Effect: Sends a DB query to the DeviceDataScheduler to change the status of the mote to 'placed'.  
     \* Created for: UC4.3 - reintroduced mote
  - `void addDevice(PluggableDeviceID id, int moteID)`  
     \* Effect: Sends a DB query to the DeviceDataScheduler to add a device in the topology and link it to a mote. The device gets the mote's location by default. If the device is already linked to another mote, overwrite that link.  
     \* Created for: UC6.3
  - `void reactivateDevice(PluggableDeviceID id)`  
     \* Effect: Sends a DB query to the DeviceDataScheduler to change the status of the device to 'placed'.  
     \* Created for: UC6.3 - reintroduced device

### A.5.5 Data type definitions

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

- None

## 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  
Short description of the UC.
- *UC13*: Configure pluggable device  
Short description of the UC.
- *UC25*: Access topology and available devices  
Short description of the UC.
- *UC24*: Consult historical data  
Short description of the UC.
- *UC26*: Send application command or message to external front-end  
Short description of the UC.
- *UC27*: Receive application command or message from external front-end  
Short description of the UC.

**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.

**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.

\*) APPLICATION CRASH -> Monitor

ApplicationContainer becomes: ApplicationManager: that does management of instances and communication with other components

ApplicationInstance: is a container/sandbox that has 1 running application instance = application execution component ?? run on different hardware maybe?

ContainerMonitor: that monitors the ContainerInstance instances

ContainerMonitor -> ApplicationInstance: boolean check() ApplicationInstance -> ContainerMonitor: void applicationCrashed(id applicationInstanceID)

\*) OR APPLICATION EXECUTION COMPONENT CRASH -> we say container has crashed/failed when the following has no response: ContainerMonitor -> ApplicationInstance: boolean check() is ok????

\*) Detection of failed hardware TODO ask what hardware?? separate hardware that runs application instances?

\*) Send notification

**Av2: Resolution of application 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.

Application fails -> ContainerMonitor detects this -> ApplicationInstance: restart 3 failed restarts -> suspend app and send notification

\*) Graceful degradation: -> send message to other parts to notify of failed part we can have component responsible for controlling state of application and can start, stop or restart application.

**Av2: Resolution of application execution component failures** 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.

Solution for the problem. ApplicationInstance failure -> send notification to system administrator SIoTIP system administrators are notified within 1 minute. application return different metrics

(usage of memory, number of served requests and so on) and some monitoring system (component), that monitors container sends regularly request to this endpoint and collect this information about applications

**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.

-> we already have this, the ApplicationContainer component.

**Av2: Up-time of application execution subsystem** The subsystem for executing applications in the Online Service must have a guaranteed minimal up-time. The subsystem for executing applications in the Online Service must be available 99% of the time, measured per month.

Solution for the problem. Subsystem = ApplicationManager + ApplicationContainer + new components ??? containers can have replicas, for example every container has 3 replicas and when one crash there has to be something that find out that there are just 2 replicas now and it is needed to create new replica.

There is also needed component for load balancing.

UC12: 1. An application indicates that it wants one or more pluggable devices to perform an actuation command (e.g., sound all buzzers on the first floor). ApplicationClient -> ApplicationFacade: interface AppMgmt: sendCommand(DeviceDataID id, MoteID id, Map<String,String> data) ApplicationFacade -> CommandHandler: interface AppMgmt: sendCommand(DeviceDataID id, MoteID id, Map<String,String> data)

2. The system constructs the actuation command message according to the specific formatting syntax for the involved pluggable device(s) CommandHandler -> ApplicationManager: void sendCommand(DeviceDataID id, MoteID id, Map<String,String> data) Json message createData(DeviceDataID id, MoteID id, Map<String,String> data)

-sends the command message to the intended pluggable device(s).

ApplicationManager -> GatewayFacade: interface AppDeviceMgmt: void sendCommand(List<int> actuators, Json message)

3. The pluggable device(s) receive(s) the actuation command message and perform(s) the contained actuation command. GatewayFacade -> MoteFacade: interface DeviceMgmt: sendCommand(Json message) MoteFacade -> PluggableDeviceFacade: interface Actuate: void sendActuationCommand(String commandName)

Remarks: The topology may be used to determine the set of actuators that need to perform the actuation command. ApplicationManager -> TopologyManager: interface TopologyMgmt: List<int> determineActuators(Json message).

Note that an explicit acknowledgement is not sent by the pluggable device. However, the pluggable device will regularly send its data, including state information, to the system (cf. UC11:Send

pluggable device data). This state will reflect the outcome of the actuation command (e.g., indicating that a buzzer is on). Furthermore, the application will be notified when pluggable devices fail (cf. UC14: Send heartbeat). PluggableDeviceData -> MoteFacade: interface DeviceData: void rcvData(PluggableDeviceID pID, DeviceData data) MoteFacade -> GatewayFacade: interface DeviceData: void rcvData(PluggableDeviceID pID, DeviceData data) GatewayFacade -> ApplicationManager: interface AppMgmt: sendDeviceStatus(PluggableDeviceID pID, String status)

UC13: 1. The primary actor specifies that it wants to set a configuration parameter of a pluggable device. ApplicationClient -> ApplicationFacade: interface AppMgmt: void setConfiguration(PluggableDeviceID id, Map<String,String> parameters) ApplicationFacade -> ApplicationManager: interface DeviceMgmt: void setConfiguration(PluggableDeviceID id, Map<String,String> parameters)

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). ApplicationManager -> DeviceDB: interface AppDeviceMgmt: bool checkDeviceParameter(PluggableDeviceID):

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. ???how

4. The system updates the internal configuration of the pluggable device. ApplicationManager -> Database: interface AppMgmt: void setConfiguration(PluggableDeviceID id, Map<String,String> parameters)

5. The system informs the primary actor that the reconfiguration was done successfully. ApplicationManager -> NotificationManager: interface Notify: notify(int userID, string message)

Alternative scenarios: 2a. If the value is invalid for the pluggable device, the system informs the application via an exception. The use case ends. - first exception Remarks: Note that different applications may have different preferences for a single pluggable device

UC24: 1. The primary actor indicates that it wants to consult a specified collection of historical data in a specified timeframe. ApplicationClient -> ApplicationFacade: interface AppData: void getHistoricalData(DateTime from, DateTime to, int custOrgID) ApplicationFacade -> ApplicationManager: interface Apps: void getHistoricalData(DateTime from, DateTime to, int custOrgID) 2. The system determines from which pluggable devices the data is required and looks up the data. ??? 3. The system presents the primary actor with the requested historical overview, e.g. as a table. ApplicationManager -> DeviceDataScheduler: interface RequestData: HistoricalData getHistoricalData(List<PluggableDeviceID> id,DateTime from, DateTime to) DeviceDataScheduler -> PluggableDeviceDB: interface DeviceData: HistoricalData getHistoricalData(List<PluggableDeviceID> id,DateTime from, DateTime to)

UC25: 1. The primary actor indicates that it wants an overview of the topology. ApplicationClient -> ApplicationFacade: interface TopologyOverview void getTopologyOverview(int custOrgID) 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. ApplicationFacade -> ApplicationManager: interface Apps: List<RoomTopology> getTopology(int custOrgID) TopologyManager -> DeviceDB: interface TopologyMgmt: List<RoomTopology> getTopology(int custOrgID) 3. The system presents the topology view to the primary actor.

UC26:

### 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: (Relevant QA or UC)

### 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.

## **Component**

- Interface

### **A.6.5 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.7 Decomposition 5: Av1 (Elements/Subsystem to decompose/expand)

### A.7.1 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av1*: Communication between SIO TIP gateway and Online Service

### 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.

**Av1: Problem title** Short description of the problem.

Solution for the problem.

Communication pattern: broker ?

Prevention: \*) SLA 99.9% availability, can't model this \*) Local networks 90% availability

Detection: \*) Online Service can detect failures of its individual internal communication components: Ping/Echo, Monitor, Heartbeat, Timestamp The failure of an internal SIO TIP Online Service component is detected within 30 seconds.

\*) SIO TIP gateway can detect failures of its individual internal communication components: Ping/Echo, Monitor, Heartbeat, Timestamp

\*) Online service can detect that a SIO TIP gateway is not sending data anymore based on the expected synchronisation interval: Monitor: GWMonitor An outage is defined as 3 consecutive expected synchronisations that do not arrive within 1 minute of their expected arrival time.

\*) Online Service should acknowledge each message sent by the SIO TIP gateway so that the gateway can detect failures.

CommunicationModule on gateway and Online Service. Sends all messages to Online Service and waits for acknowledgements for all sent messages. If no acknowledgement within X seconds -> Online Service failure

Resolution: \*) gateway outage (= complete failure) detected -> notify IO and SIO TIP system administrator The infrastructure owner is notified within 5 minutes after the detection of an outage of their gateway. A SIO TIP system administrator should be notified within 1 minute after the detection of a simultaneous outage of more than 1% of the registered gateways.

\*) gateway component failure detected -> first, restart component. If still fails, reboot gateway entirely.

\*) Online Service / communication channel failure detected -> -> temporarily store all incoming pluggable data and issued application commands internally The SIO TIP gateway can store at least 3 days of pluggable data before old data has to be overwritten.

\*) When Online Service = unreachable, applications on SIO TIP gateway can still operate normally, TODO ask Response Measure: \*) The SIO TIP gateway will start synchronising with the Online service within 1 minute after the communication channel becomes available.

### 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: (Av1 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 6: P1, UC1, UC2, UC3, UC5, UC7, UC8, UC16, UC20 (Elements/Subsystem to decompose/expand)

### A.8.1 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  
Short description of the UC.
- *UC2*: Register an end-user  
Short description of the UC.
- *UC3*: Unregister an end-user  
Short description of the UC.
- *UC5*: Uninstall mote  
Short description of the UC.
- *UC7*: Remove a pluggable device from its mote  
Short description of the UC.
- *UC8*: Initialise a pluggable device  
Short description of the UC.
- *UC16*: Consult notification message  
Short description of the UC.
- *UC20*: Unsubscribe from application  
Short description of the UC.

### 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.

**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.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: (P1 or 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 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.9.1 Selected architectural drivers

The functional drivers are:

- *UC28*: Log in  
Short description of the UC.
- *UC29*: Log out  
Short description of the UC.

### 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 8: UC22, UC23 (Elements/Subsystem to decompose/expand)

### A.10.1 Selected architectural drivers

The functional drivers are:

- *UC22*: Upload an application  
Short description of the UC.
- *UC23*: Consult application statistics  
Short description of the UC.

### 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.

## A.11 Decomposition 9: UC21 (Elements/Subsystem to decompose/expand)

### A.11.1 Selected architectural drivers

The functional drivers are:

- *UC21*: Send invoice  
Short description of the UC.

### A.11.2 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.11.3 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.11.4 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.11.5 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.