SEVENTH FRAMEWORK PROGRAMME
THEME – Energy Efficient Buildings

EeB-ICT 2011.6.4. ICT for energy-efficient building and spaces of public use

Self learning Energy Efficient builDing and open Spaces

GA No. 285150

D5.4 - Specification and implementation of interfaces that have been integrated and tested

Work Package	WP5 - Self learning and global optimization		
Task	Task 5.4: Interfacing and Integration within SEEDS architecture		
Revision	0		
Due date	31/05/2013	**Submission date**	13/06/2013
Dissemination level	PU	**Deliverable type**	P

Authors	R. Meyer (FhG-EAS), Dr. Jürgen Haufe (FhG-EAS), F. Díaz (SOFTCRITS)
Verification	F. Díaz (SOFTCRITS), N. Jiménez-Redondo (CEMOSA)
Approval	N. Jimenez-Redondo (CEMOSA)

Table of Contents

List of Figures

List of Tables

1 Introduction

This document (D5.4 *"Interfacing and Integration within SEEDS architecture"*) is part of the Work Package 5 "*Self learning and global optimization*", within the FP7 project "*Self learning Energy Efficient builDings and open Spaces*".
Main objective of this deliverable is to describe the interfaces and interactions between the SEEDS components. The SEEDS architecture is divided into three layers.

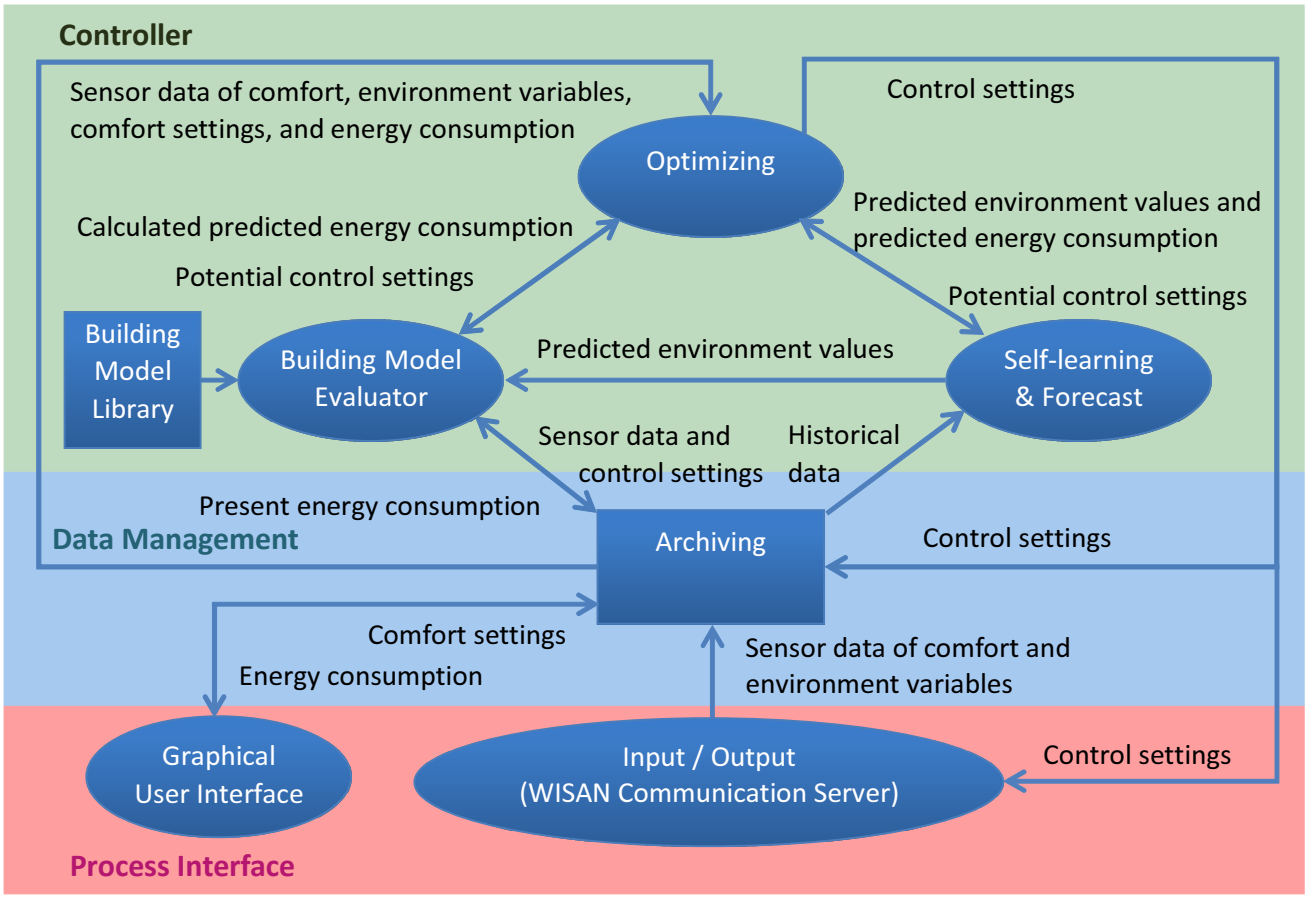

Figure 1: Layer architecture of SEEDS BEMS

The Controller Layer includes the core components of the building energy management system (BEMS) to compute and optimize the control settings for the facilities of the building. The Data Management Layer is a database which stores historical, present, and future data. The Process Layer includes the GUI to provide a graphical interface between the SEEDS BEMS and the users, and the WISAN for input and output sensor / actuator values.

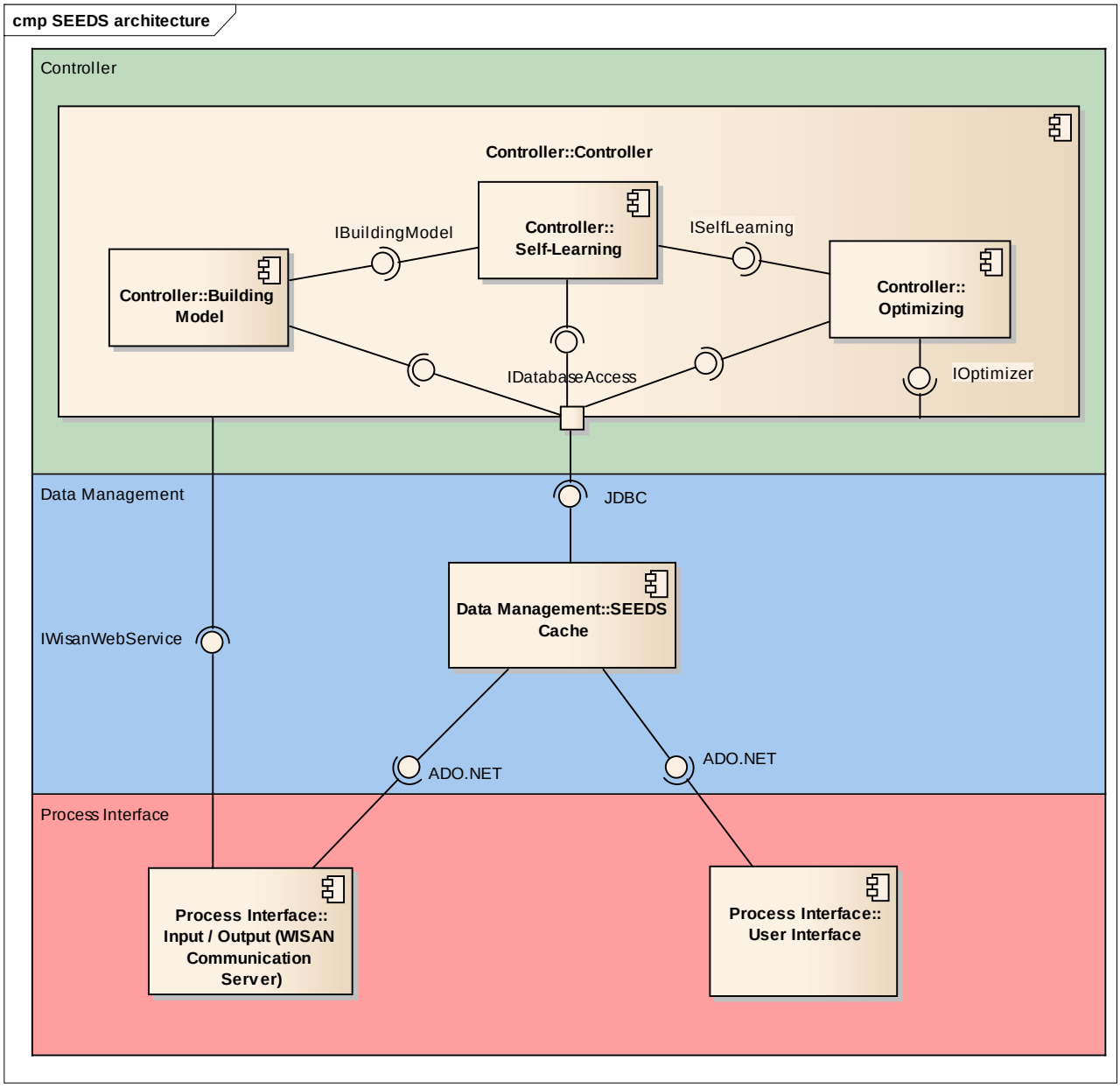

Figure 2: Component Diagram of SEEDS architecture

This document is a starting-point for the implementation of the SEEDS components. It describes the interfaces that have been implemented from the project engineers in textual form and UML Diagrams. In Section 3, all internal interfaces of the Controller are explained. That includes the interface definition of the core components Building Model, Self-Learning, Optimizer, and the Controller itself. In Section 4, all interfaces outside the controller are described. This mainly concerns the communication between the Data Management Layer (SEEDS Cache) and the interface between Controller, and WISAN to transfer optimized control settings.

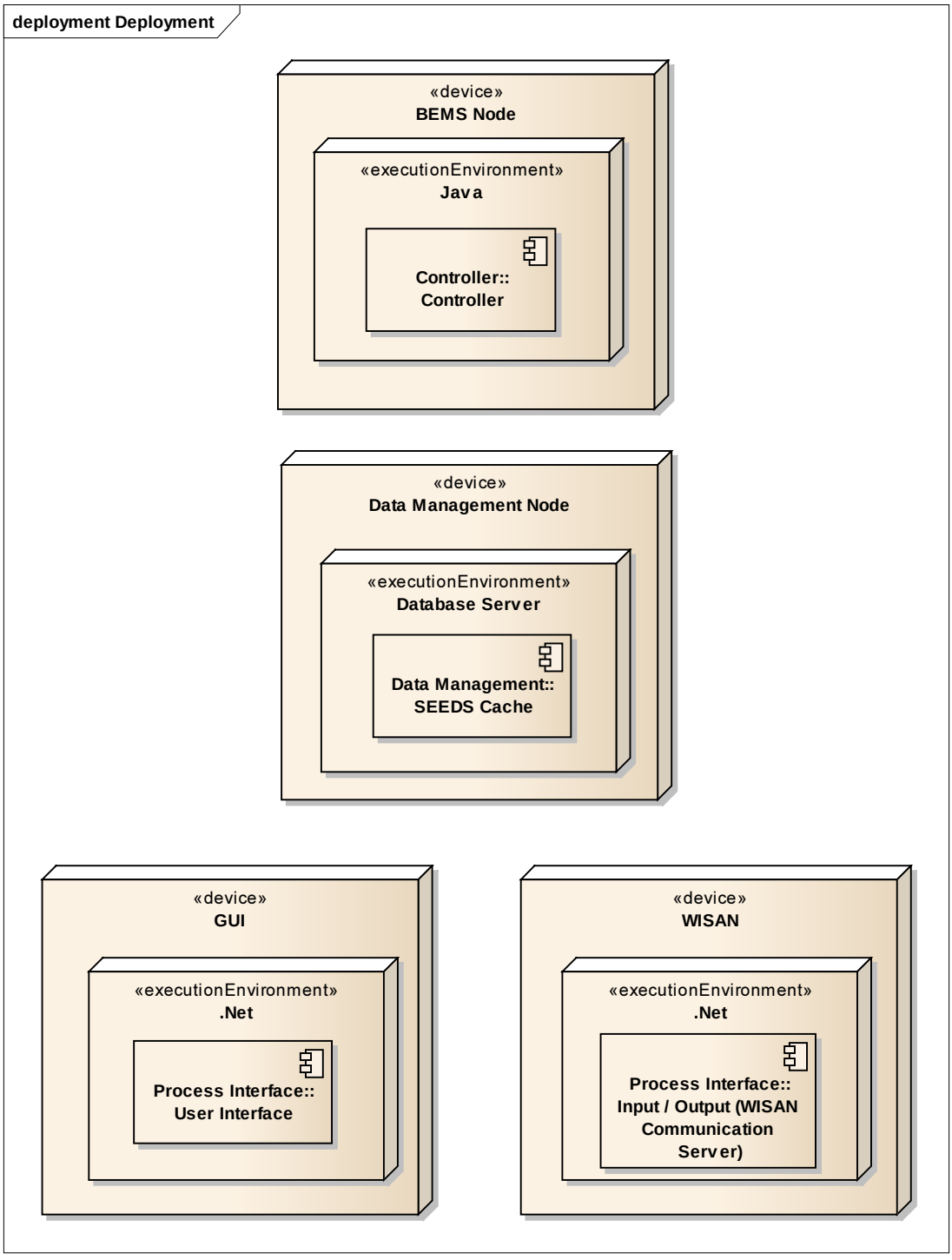

Figure 3: Deployment Diagram of the SEEDS architecture

The run-time architecture of the SEEDS system is divided into four possible nodes. The Controller and all of its core components are implemented in Java and executed on the same node. The Data Management Layer is represented by a Database Server on an addition node. The components of the Process Interface Layer GUI and WISAN are both implemented with Microsoft .NET and both executed on single nodes.

2 Test Bed

Since the SEEDS pilots were not ready by the time the present deliverable was produced, the implemented interfaces were tested on the Helicopter Garage example. To test the developed interfaces and the interaction between the various components of the SEEDS architecture a physical building model of the helicopter garage was developed. It calculates the process variables, especially the values of comfort, as response on the control settings, the comfort settings, weather, and occupancy. In SEEDS Controller implementations of the core components are instantiated and set up for controlling the helicopter garage. Furthermore, Archive and Front Panel are included in the test bed in the target implementation. Only the WISAN component is omitted; its communication is replaced by inputs and outputs of the physical building simulator.

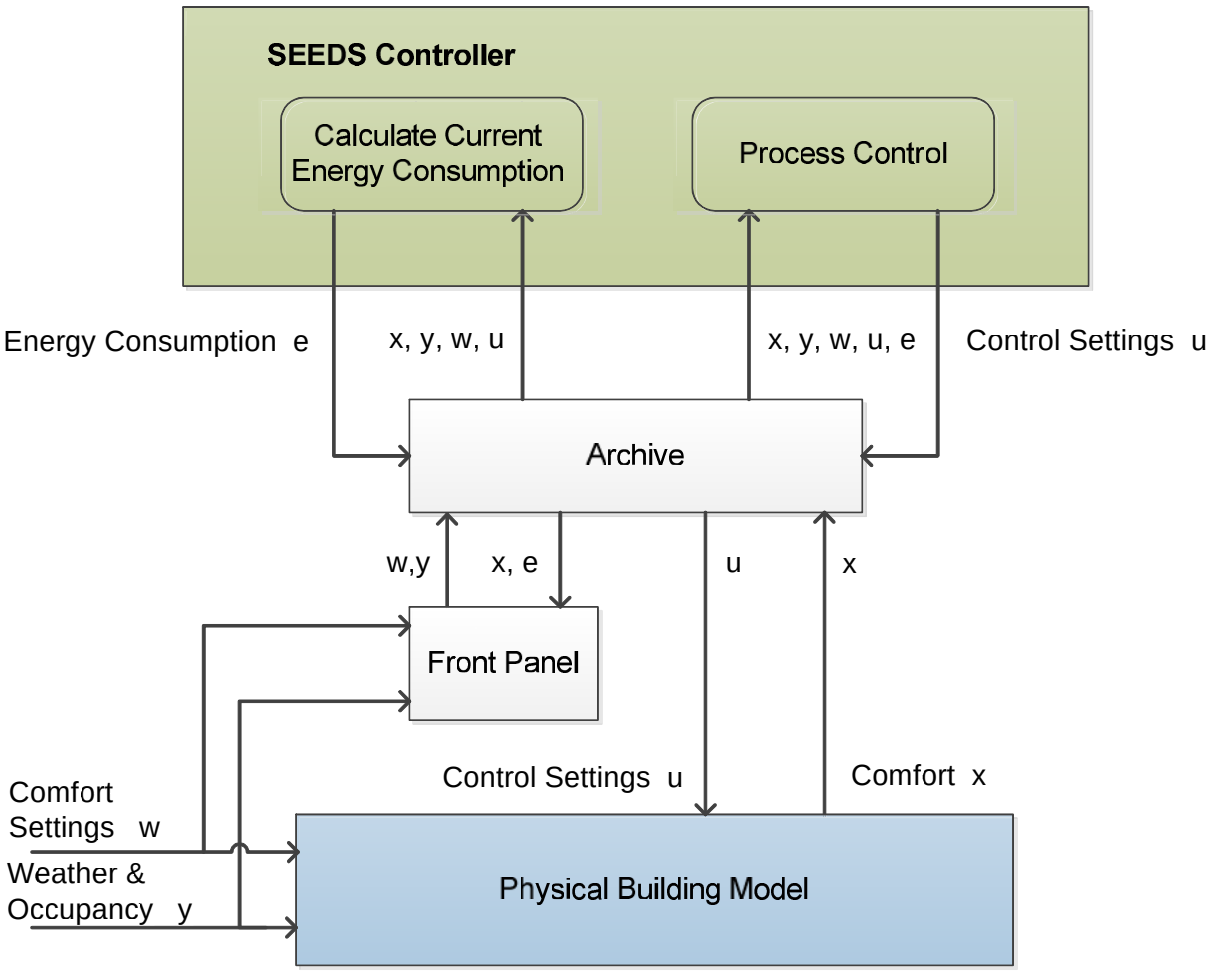

Figure 4: Architecture of the SEEDS Test Bed

Detailed information about the helicopter garage is given in the report [1] "*D2.3 - Modelling Methodology. ANNEX B - Application of SEEDS Modelling Methodology in an example (Helicopter Garage, HG)"* and report [2] *"D2.8 - Energy Control Strategy. First Version"* Chapter 4

3 Controller internal Interfaces

3.1 Controller

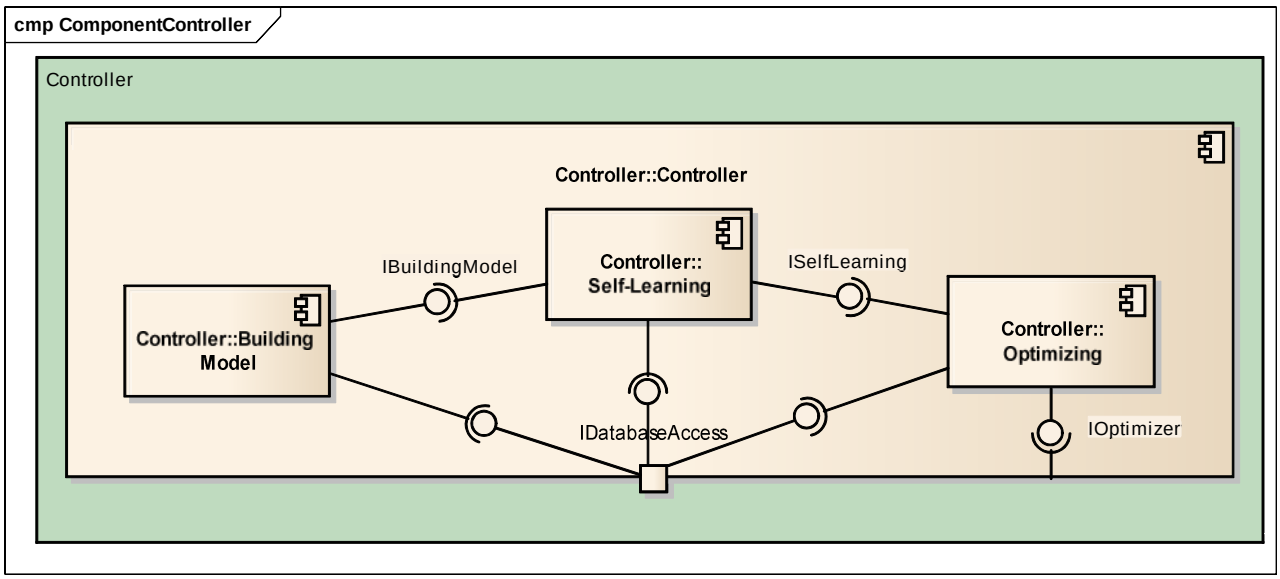

Figure 5: Component Diagram Controller

The Controller layer contains the three core components Building Model, Self-Learning and Optimizing. The Controller itself is represented by an own component (Controller::Controller). The Controller component fulfills several tasks and could be considered as the main start point of the BEMS. At startup, the Controller component takes care of the core components. It validates that all core components are available and instantiate them if required. All components are implemented in Java and executed in the same Java virtual machine (JVM).

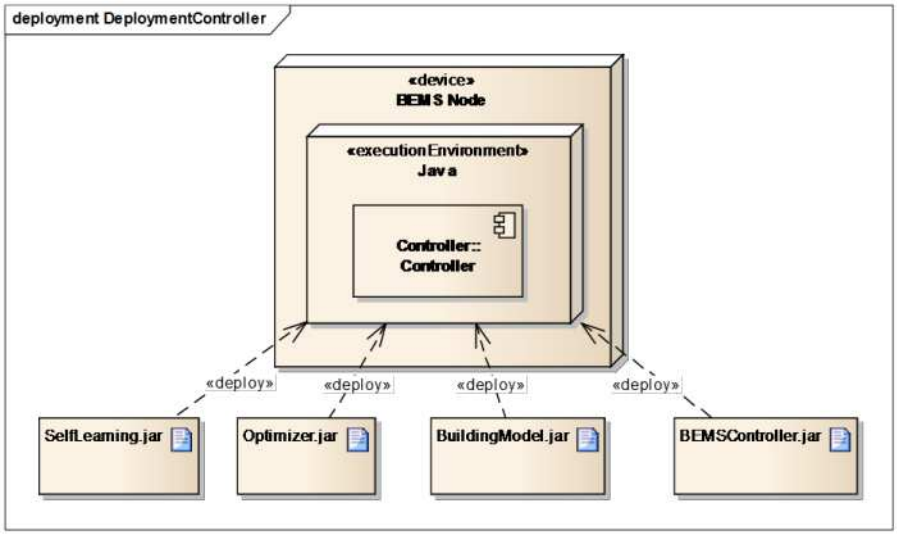

Figure 6: Deployment Diagram of the Controller

Each core component and the controller itself is represented by an own Java Archive (.jar) file. The `BEMSController.jar` is the main start point and has a main method to execute the controller loop.

After a successful start, the loop of process control is started and runs until the controller shutdown. The loop of process fulfills the following tasks:

- Calculation of the current energy consumption.
- Start of the optimization of control settings if necessary.

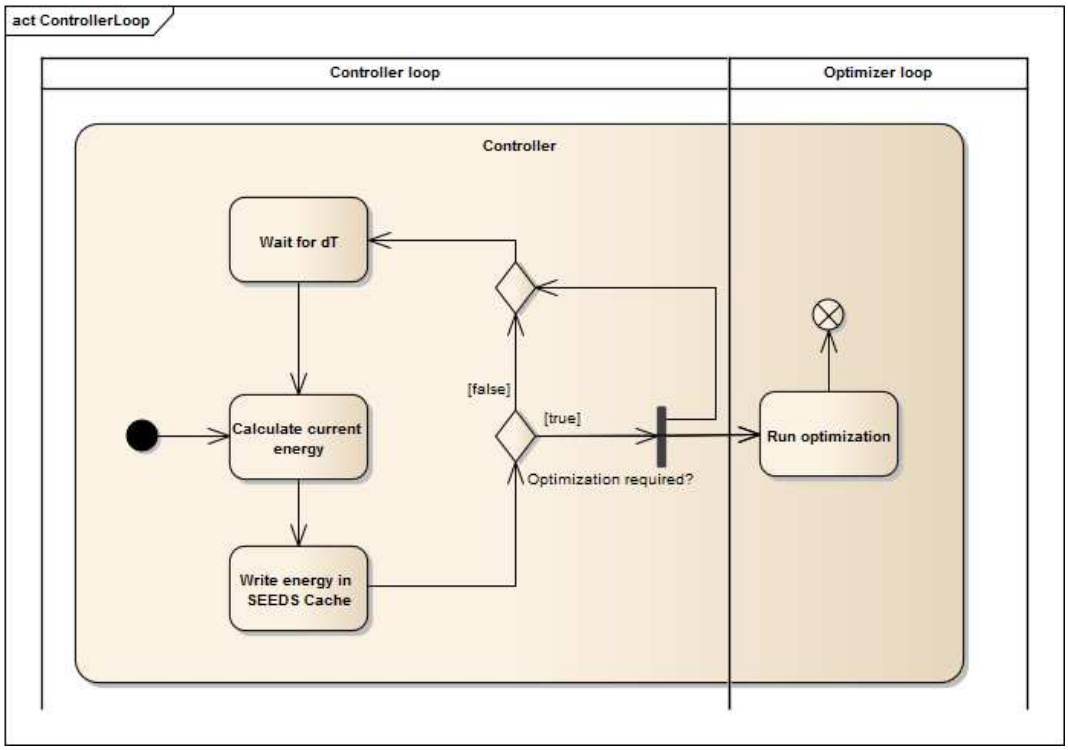

Figure 7: Control loop

All core components need access to the Data Management layer to read or write values from / to the SEEDS Cache. To simplify this access the controller provides an interface for easy access to the SEEDS Cache (`IDatabaseAccess` details in chapter 3.4).

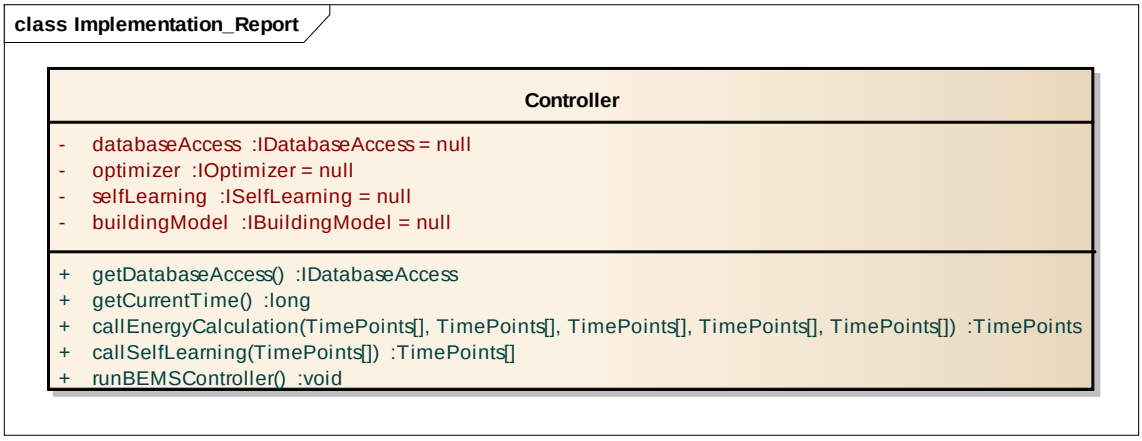

Figure 8: Controller Class Diagram

The method `getDatabaseAccess()` returns an Object that implements `IDatabaseAccess` which grant access to the SEEDS Cache.

```
public IDatabaseAccess getDatabaseAccess();
```

return:	IDatabaseAccess object which grant access to the SEEDS Cache

To provide a controller wide uniform time base the controller provides a method to query the current time.

```
public long getCurrentTime();
```

return:	Number of milliseconds since January 1, 1970, 00:00:00 GMT

The controller acts for the other core components as so called “mediator”. The mediator design pattern defines an object that encapsulates how a set of objects interacts. This promotes loose coupling for the core components and simplifies the dependencies between the components. To call the Building Model component, the controller provides the method `callEnergyCalculation(…)`. For detailed information about the interface of the Building Model see chapter 3.7

```
public TimePoints[] callEnergyCalculation( TimePoints[] temperatures,
                                           TimePoints[] controlSettings,
                                           TimePoints[] waterSupply,
                                           TimePoints[] deviceStates);
```

temperatures:	Array of current temperatures in °C
controlSettings:	Array of control settings
waterSupply:	Array of supply temperature in °C of the chiller and heat pump
deviceStates:	Array of device states
return:	Time points in kW of energy consumption from the rooms and building total consumption

To call the Self-Learning component the controller provides the method `callSelfLearning(…)`. For detailed information about the interface of the building model see chapter 3.6

```
public TimePoints[] callSelfLearning(TimePoints[] potentialControlSettings);
```

potentialControlSettings:	Array of potential control settings
return:	array of predicted room temperatures and energy consumption

3.2 OSGI

The controller consists of four different components (Controller itself, Building Model, Self-learning, Optimizer), all implemented in Java. The standard Java environment does not include a dynamic component model so OSGi (Open Services Gateway initiative framework) as module system and service platform was carried out. OSGi groups Java classes to so called bundles in .jar files and equip them with a MANIFEST.MF file with additional information (bundle name, versions, dependencies, service descriptions)

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Self-Learning
Bundle-SymbolicName: eu.seeds_fp7.controller.selflearning
Bundle-Version: 0.0.1.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Require-Bundle: eu.seeds_fp7.controller;bundle-version="0.0.1
Export-Package: eu.seeds_fp7.controller.selflearning
Service-Component: OSGi-INF/SelfLearning.xml
Bundle-ClassPath: .
```

Figure 9: MANIFEST.MF example

Each bundle is a tightly coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies.
In other words OSGi introduces a kind of modularization system on top of java core platform.
Each SEEDS Controller component is represented by its own OSGi bundle:

- `eu.seeds_fp7.controller.jar` Bundle with interface definitions and data classes.
- `eu.seeds_fp7.controller.impl.jar` Controller implementation.
- `eu.seeds_fp7.controller.buildingmodel.jar` Building Model implementation.
- `eu.seeds_fp7.controller.optimizing.jar` Optimizer implementation.
- `eu.seeds_fp7.controller.selflearning.jar` Self-learning implementation.

Bundle Name	Bundle dependencies
eu.seeds_fp7.controller.jar	
eu.seeds_fp7.controller.impl.jar	eu.seeds_fp7.controller.jar
eu.seeds_fp7.controller.selflearning.jar	eu.seeds_fp7.controller.jar
eu.seeds_fp7.controller.optimizing.jar	eu.seeds_fp7.controller.jar
eu.seeds_fp7.controller.buildingmodel.jar	eu.seeds_fp7.controller.jar

Table 1: Bundle Dependencies

The eu.seeds_fp7.controller.jar bundle contains only the interface definitions and the data classes, but no implementation. The other bundles contain the respective implementation and its only dependency of the `eu.seeds_fp7.controller.jar` bundle.
To populate the implementation of a functionality of a bundle, OSGi Declarative Services are used. A service is specified by a Java interface and can be registered with the OSGi Service Registry. Clients, who want to consume this functionality, ask the registry for this implementation, so that the client bundle has no direct dependency to the service provider bundle. To populate the service in the registry, an extra XML document containing the service description must exist in the bundle.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="eu.seeds_fp7.controller.selflearning">
   <implementation
class="eu.seeds_fp7.controller.selflearning.SelfLearning"/>
   <service>
      <provide
interface="eu.seeds_fp7.controller.selflearning.ISelfLearning"/>
   </service>
</scr:component>
```

Figure 10: Service Description example

Each SEEDS Controller component provides the following services:

- `eu.seeds_fp7.controller.IController` Service provider for Controller implementation.
- `eu.seeds_fp7.controller.buildingmodel.IBuildingModel` Service provider for Building Model implementation.
- `eu.seeds_fp7.controller.optimizing.IOptimizer` Service provider for Optimizer implementation.
- `eu.seeds_fp7.controller.selflearning.ISelfLearning` Service provider for Self-learning implementation.

Service Dependencies:

Bundle Name	Service dependencies
eu.seeds_fp7.controller	
eu.seeds_fp7.controller.impl	eu.seeds_fp7.controller.selflearning, eu.seeds_fp7.controller.optimizing, eu.seeds_fp7.controller.buildingmodel
eu.seeds_fp7.controller.selflearning	
eu.seeds_fp7.controller.optimizing	
eu.seeds_fp7.controller.buildingmodel	

3.3 Time-related Data Classes

The core components and the controller communicate directly through java method calls. For example the Optimizer component calls the Self-learning component with the evaluated control settings. The Self-learning component predicts the temperatures and returns it to the Optimizer component. An analysis of this parameter shows that all arguments are time-value data. To handle this data two data classes `TimePoint` and `TimePoints` are created. The following section explains the class `TimePoint`, a representation of a single time-value pair, and subsequently the class `TimePoints`, a representation of a sequence of time points.

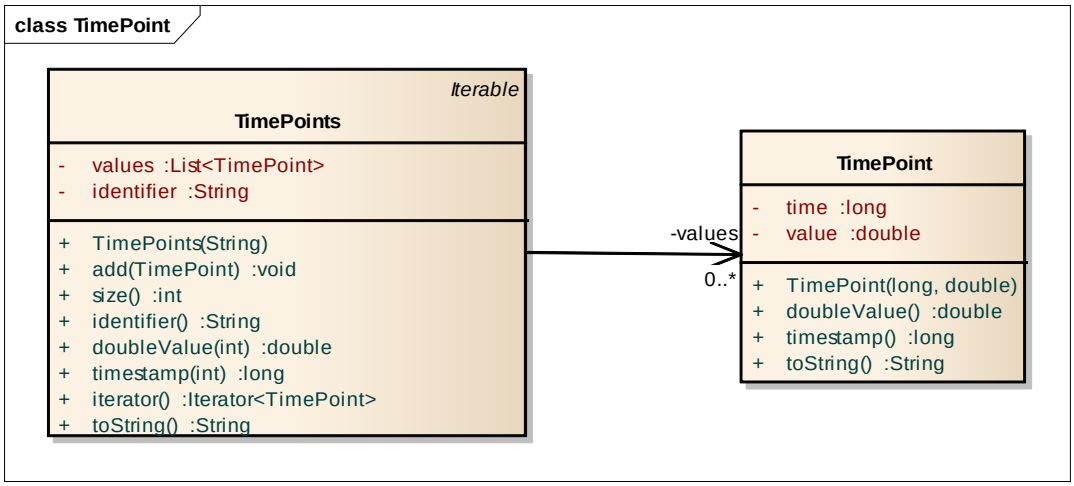

Figure 11: Time classes

`The class TimePoint` represents a single time-value pair. The time is stored as POSIX time in milliseconds and represents a `long` value of milliseconds since 1 January 1970. The value is stored as a `double` value.

The constructor creates a new `TimePoints` object with the specified time and value.

```
public TimePoint(long time, double value)
```

`time:`	Time in milliseconds since January 1, 1970, 00:00:00 GMT
`value:`	Value as double

The method `doubleValue()` returns the value of the `TimePoint` object as `double`.

```
public double doubleValue()
```

`returns:`	The value as double

The method `timestamp()` returns the time value of the `TimePoint` object as `long`.

```
public long timestamp()
```

`returns:`	Time in milliseconds since January 1, 1970, 00:00:00 GMT

The method `toString()` returns a human readable representation of the `TimePoint` object.

```
public String toString()
```

`returns:`	A string representation in form “Timestamp : Value”

The controller requires time curves e.g. a list of `TimePoint` objects. For this purpose the Class `TimePoints` was created who represents a sequence of time point. Addition to a list of `TimePoint` objects the `TimePoints` class has a unique identifier.

`TimePoints temperatureOfRoom1` represents:

TemperatureRoom1	
2012-07-12 10:00:00	24.123456
2012-07-12 10:30:00	26.654321
2012-07-12 11:00:00	28.112233
2012-07-12 11:30:00	27.987654
2012-07-12 12:00:00	25.332211
2012-07-12 12:30:00	23.000005
...	...
time	temperature

identifier

Figure 12: TimePoints data structure

The unique identifier represents a sensor or actuator value from the SEEDS Cache. The available identifiers depend on the controlled building and its facilities and are static configured before the controller starts.
The following tables show the identifiers of the control settings of the Helicopter Garage Example:

Identifier Group:	`DeviceControlSetting`

Description:	`FC_Speed – Speed level [1, 2, 3, 4] of the Fan-Coil1...Fan-Coil10`
	`CH_Load – Load level [0.25, 0.50, 0.75, 1.00] of a Chiller`
	`HP_Load – Load level [0.25, 0.50, 0.75, 1.00] of a Heat Pump`

Helicopter Garage Example:	*FANCOIL1, FANCOIL2, FANCOIL3, FANCOIL4, FANCOIL5, FANCOIL6, FANCOIL7, FANCOI8, FANCOI9, FANCOI10, CHILLER, HEATPUMP*

Table 3: Identifier Group – Device Control Settings

To provide a better overview the identifiers are grouped. The identifier group "DeviceControlSettings" represents the control settings of the devices and contains all controllable facilities. The Helicopter Garage Example contains ten fancoils, a heatpump, and a chiller for controlling.
A complete list of the identifiers for the helicopter garage example is to be found in the annex B.

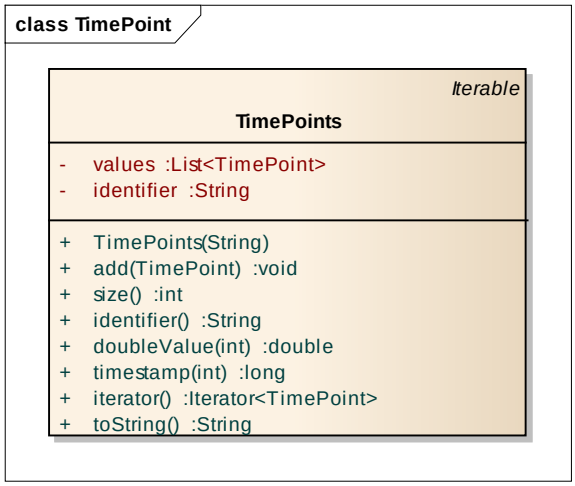

Figure 13: Class TimePoints

The public constructor creates a new `TimePoints` object with the specified identifier.

```
public TimePoints(String identifier)
```

`identifier:`	Unique identifier that labels a sensor or actuator from the SEEDS Cache

The method `size()` return the number of time – value – pairs for this object.

```
public int size()
```

`return:`	Number of TimePoint

The method `doubleValue(…)` returns the value for the specified index

```
public double doubleValue(int index)
```

`index:`	Index of the value
`Return:`	Value of the specified index

The method `timestamp(…)` returns the timestamp for the specified index

```
public long timestamp(int index)
```

`index:`	Index of the timestamp
`Return:`	Time on the specified index in milliseconds since January 1, 1970, 00:00:00 GMT

The method `identifier()` return the identifier of this TimePoints object

```
public String identifier()
```

Return:	Identifier of this object

The method `add(…)` adds a time – value – pair to this object.

```
public void add(TimePoint timePoint)
```

timePoint:	Unique name that labels a sensor or actuator from the SEEDS Cache

The method `iterator()` return an iterator over the `TimePoint` objects.

```
public Iterator<TimePoint> iterator()
```

Return: Return:	An iterator over the TimePoint elements.

The method `toString()` returns a human readable representation of the `TimePoints` object.

```
public String toString()
```

Return:	The identifier

3.4 Controller Archiving Interface

The SEEDS Cache is the central element in the SEEDS system. All components write their data to the cache and read the desired data from the cache. All Controller core components need access to the Data Management layer to read or write values from / to the SEEDS Cache. To unify the transfer, an interface is defined that includes all access methods to the database. The `IDatabaseAccess` interface is defined to create a database abstraction. This leaves the underlying database used from SEEDS Cache hidden for the other components and allows a database-independent communication and manipulation of the underlying database. This interface is able to handle the time classes `TimePoint` and `TimePoints`.

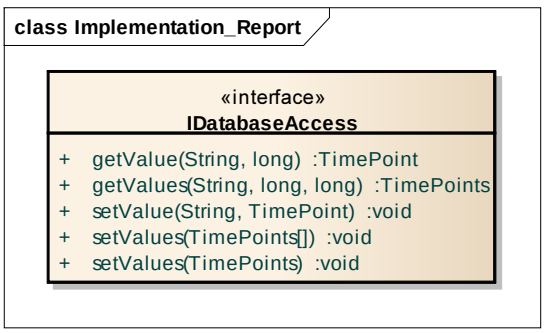

Table 4: IDatabaseAccess interface

The method `getValue(…)` queries the SEEDS Cache and returns a single legal time point to the specified time point. If the SEEDS Cache has no entry on the specified time, the next legal entry prior is returned.

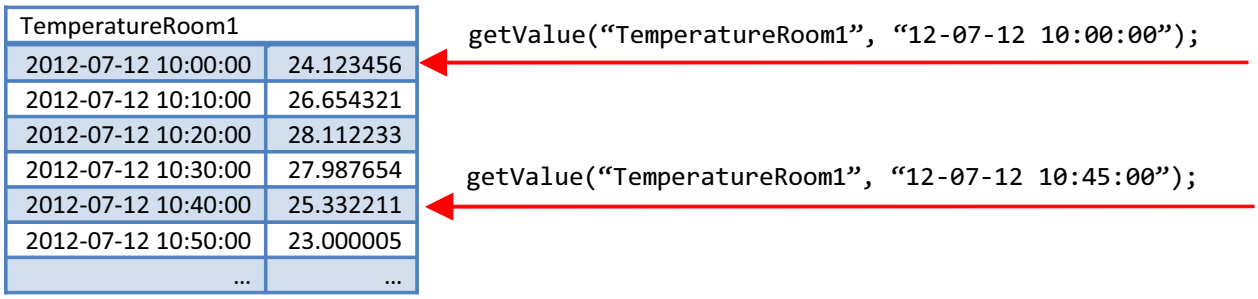

Figure 14: TimePoints data structure

```
TimePoint getValue(String identifier, long time);
```

identifier:	The identifier
time:	Time in milliseconds since January 1, 1970, 00:00:00 GMT
Returns:	Single time point (time, value) or NULL if identifier could not be found

The method `getValues(…)` queries the SEEDS Cache and returns a sequence of ascending continuing time points between the specified start and stop time. If the specified identifier could not be found, NULL is returned. If the SEEDS Cache contains no values for the specified time period, an empty TimePoints-Object is returned.

```
TimePoints getValues(String identifier, long startTime, long stopTime);
```

identifier:	The identifier
startTime:	Time in milliseconds since January 1, 1970, 00:00:00 GMT
stopTime:	Time in milliseconds since January 1, 1970, 00:00:00 GMT
Returns:	Sequence of ascending continuous data points or NULL if identifier could not be found

The method `setValue(…)` stores a single time point object in the SEEDS Cache.

```
void setValue(String identifier, TimePoint point);
```

identifier:	The identifier for the time point
point:	Time value pair to store in the SEEDS Cache

The method `setValues(TimePoints)` stores a single TimePoints Object in the SEEDS Cache.

```
void setValues(TimePoints timePoints);
```

`timePoints:`	time points object to store in the SEEDS Cache

For easy storing of `TimePoints` arrays the `setValues(TimePoints[])` was created. It stores all `TimePoints` from the array in the SEEDS Cache.

```
void setValues(TimePoints[] timePoints);
```

`timePoints:`	Array of time points to store in the SEEDS Cache

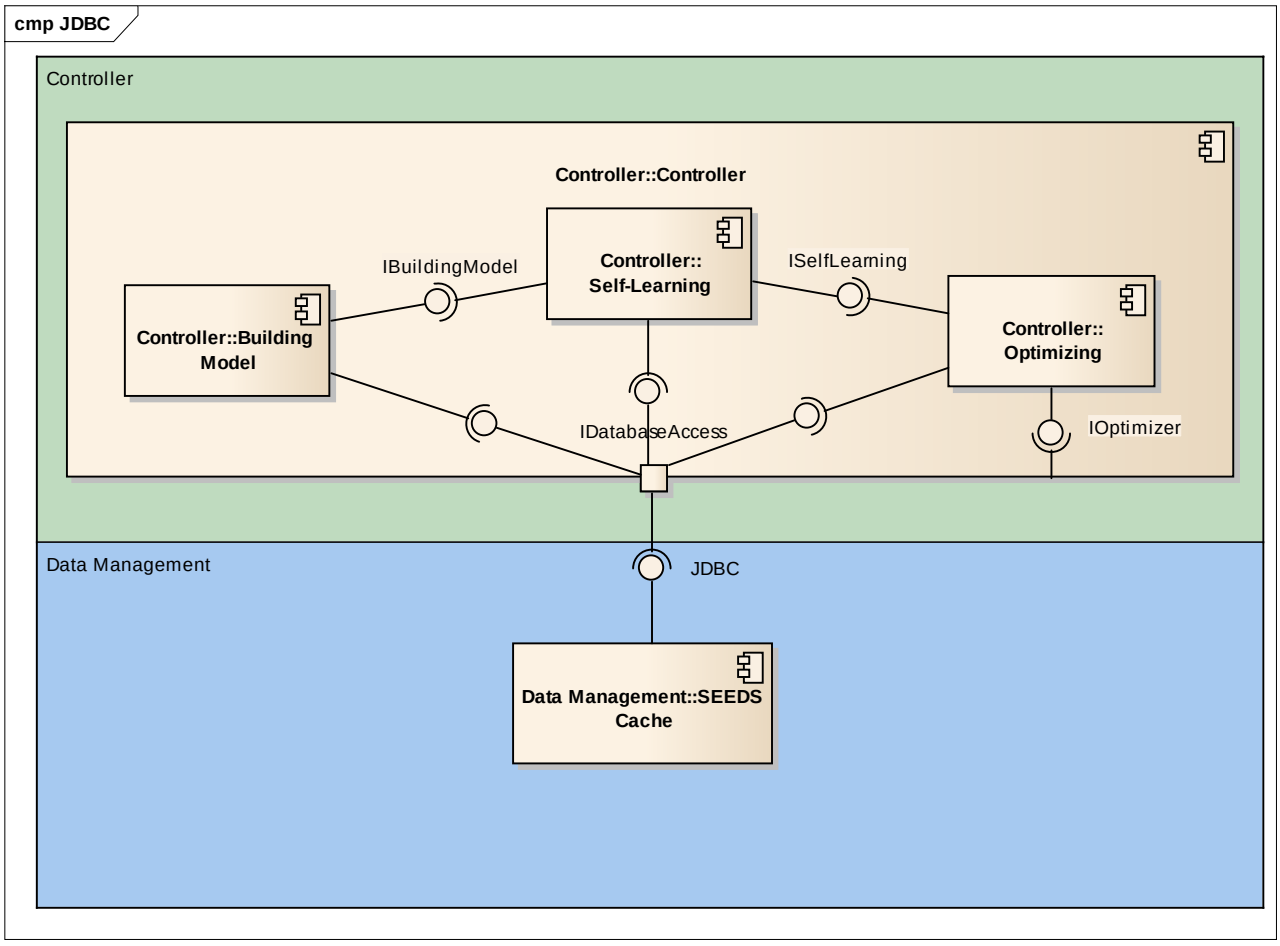

Figure 15: Java Database Connectivity

The IDatabaseAccess implementation translates the identifiers to a storage location in the SEEDS Cache. Depending on the IDatabaseAccess – method, the implementation read or writes data from or to this location. For this purpose the Java Database Connectivity (JDBC) is used. JDBC is a Java-based data access technology for querying and updating relational databases.

3.5 Optimization Interface

This component encapsulates the optimization algorithm. It creates a set of potential control settings and transfers them to the Self-learning component. The Self-learning component predicts the temperatures and the energy consumption of the building and returns these values. The optimizer checks whether the temperatures are in the limitations of the comfort and the energy consumption is decreased. In negative case the potential control settings are discarded. In positive case they are registered and used as reference for a next trial. If the optimizer found a minimum of energy consumption or a fixed number of trails is reached, the results have to be written to the SEEDS Cache.

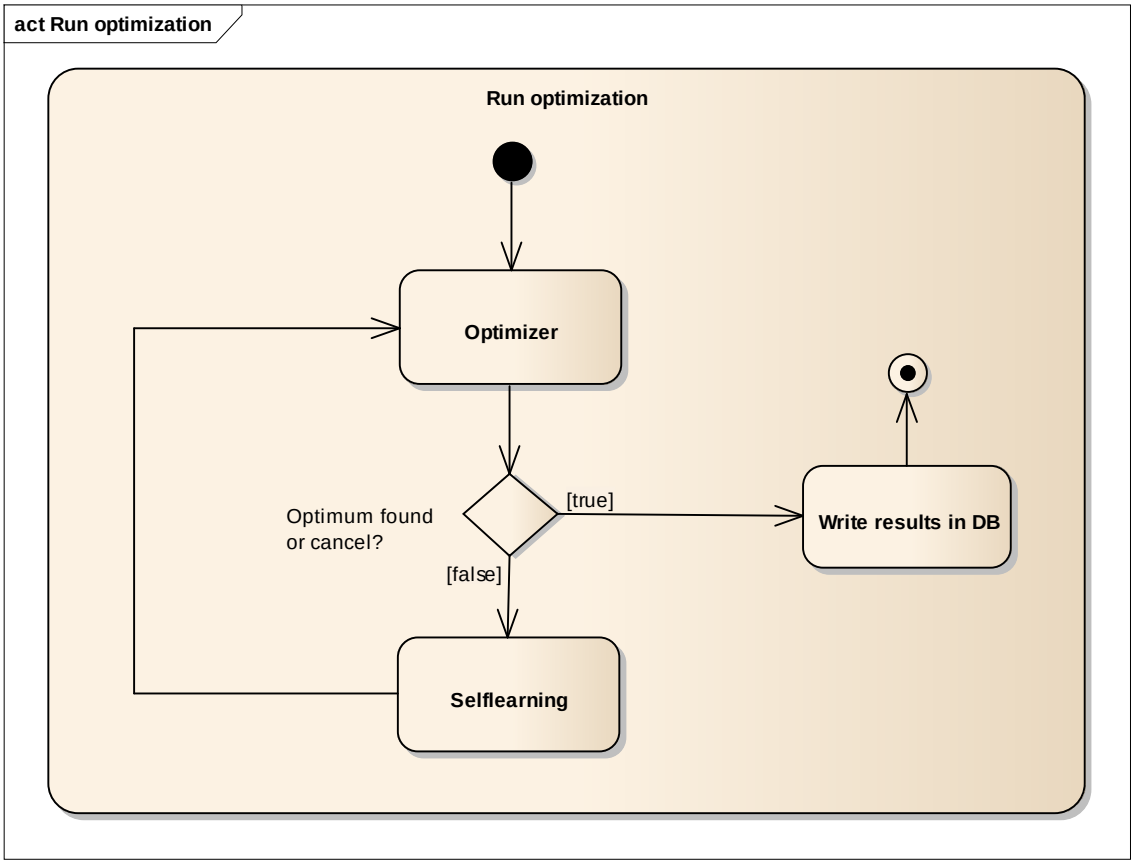

Figure 16: Activity Diagram of Optimizer run

The Optimizer has to implement the interface `IOptimizer`. This interface has only one method called `doOptimizing(…)` with the Controller Object as argument. The Method returns the optimized control settings as array of `TimePoints` objects.

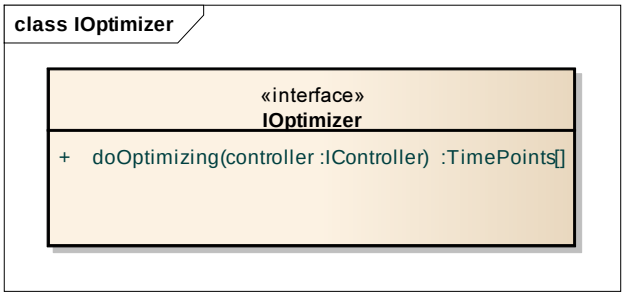

Figure 17: Class Diagram of IOtimizer Interface

The Controller object provides access to the SEEDS Cache and to the Self-learning component. Over the controller method `getDatabaseAccess()` the Optimizer can query the database and store the optimized control settings. To call the Self-learning component the controller object provides the `callSelfLearing(…)` method. This method requires the potential control settings as argument and returns the predicted temperatures and energy consumption.

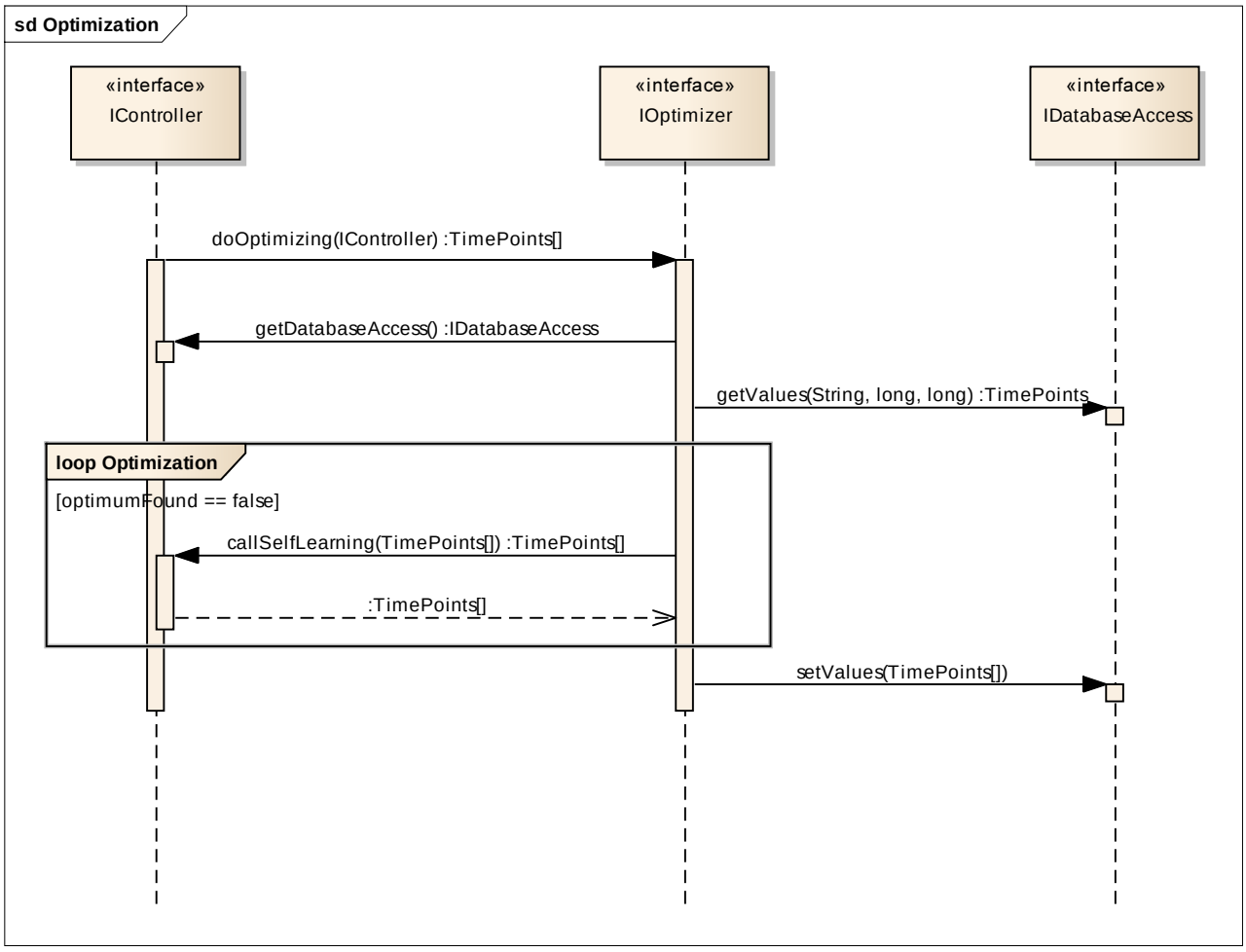

Figure 18: Sequence diagram of Optimizer loop

```
public TimePoints[] doOptimizing(IController controller);
```

controller:	The controller object
Returns:	The optimized control settings as array of `TimePoints` objects.

To provide the Optimizer implementation as OSGi Service, the bundle has to register the implemented class in the OSGi Service Registry under the `eu.seeds_fp7.controller.optimizer.IOptimizer` interface. The OSGI Service component description of the Helicopter Garage Optimizer implementation can be found in annex C.

3.6 Self-learning Interface

This component forecasts the comfort conditions and the energy consumption of the building. It is called from the Optimizer component with potential control settings. The predicted comfort settings and the predicted energy consumption are returned.

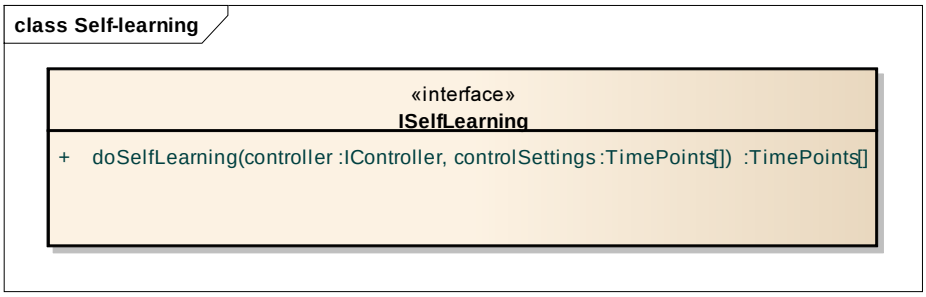

Figure 19: Class Diagram of ISelfLearning Interface

The Self-learning component has to implement the `ISelfLearning` interface. This interface defines one method called `doSelfLearning(…)` with two arguments. The first argument is the `IController` object which provides access to the SEEDS Cache and the Building Model component. The second argument is a `TimePoints` - array of potential control settings. The method returns the predicted comfort conditions and energy consumption as an array of `TimePoints`.

```
public TimePoints[] doSelfLearning(IController controller,
                                   TimePoints[] potentialControlSettings);
```

controller:	The controller object
potentialControlSettings:	Array of potential control settings
Returns:	The predicted comfort conditions and energy consumption as an array of `TimePoints`

An example call of the `doSelfLearning` method with detailed list of the identifiers for the Helicopter Garage example can be found in annex D.
It depends on the internal implementation of the Self-learning component if it predicts the energy with forecast technics or if the Building Model component is called.

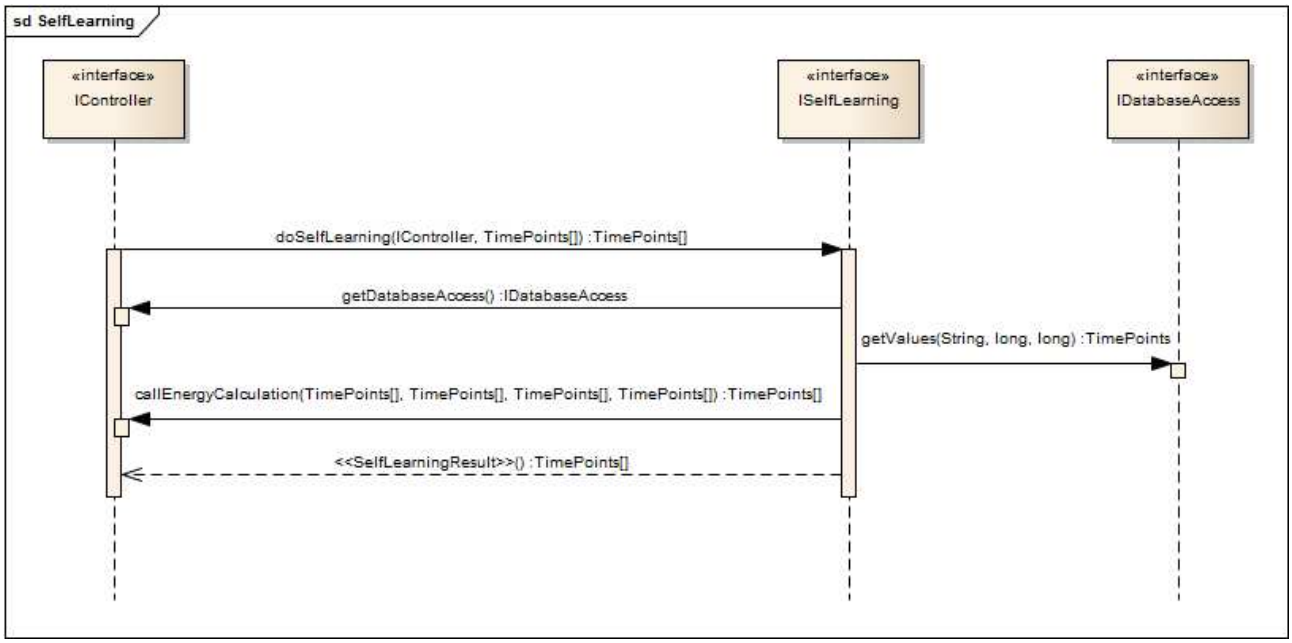

Figure 20: Self-learning example with calling the Building Model component

To provide the Self-learning implementation as OSGi Service the bundle has to register the implemented class in the OSGi Service Registry under the `eu.seeds_fp7.controller.selflearning.ISelfLearning` interface.

The OSGI Service component description of the Helicopter Garage Self-learning implementation can be found in annex C.

3.7 Building Model Interface

This component is used in twofold concern:

- Firstly, calculation of the present energy consumption.
  In doing this, sensor data from the facilities and present control settings are taken into account.
- Secondly, it is used to calculate the future energy consumption of the building.
  For that, forecasted comfort values by self-learning and potential control settings by optimizing are used.

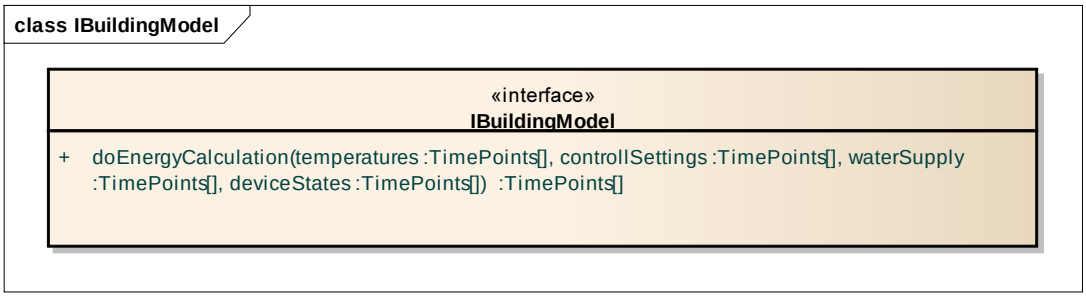

Figure 21: Class Diagram of IBuildingModel interface

The Building Model component has to implement the interface `IBuildingModel`. This interface defines the method `doEnergyCalculation(…)` which is called from the controller component to calculate the current energy consumption and depending on the implementation from the Self-learning component to predict the energy consumption.

```
public TimePoints[] doEnergyCalculation(TimePoints[] temperatures,
                                        TimePoints[] controlSettings,
                                        TimePoints[] waterSupply,
                                        TimePoints[] deviceStates);
```

`temperatures:`	Array of room and outside temperatures in °C
`controlSettings:`	Array of control settings
`waterSupply:`	Array of water supply temperatures in °C
`deviceStates:`	Array of device states
`Return:`	The energy consumption in kWh of the rooms and the total consumption of the building as an array of `TimePoints`

An example call of the `doEnergyCalculation` method with detailed list of the identifiers for the Helicopter Garage example can be found in annex D.

To provide the Building Model implementation as OSGi Service the bundle has to register the implemented class in the OSGi Service Registry under the `eu.seeds_fp7.controller.buildingmodel.IBuildingModel` interface. The OSGI Service component description of the Helicopter Garage Example Building Model implementation can be found in annex C.

4 Controller external Interfaces

4.1 User Interface SEEDS Cache Interface

The Graphical User Interface provides the interface between the user and the SEEDS system. The GUI displays to the user the SEEDS system data, and gives him the ability to change data into the system. The GUI communicates with the individual SEEDS components by means of the SEEDS Cache and enables data exchange between the GUI and the individual system components.

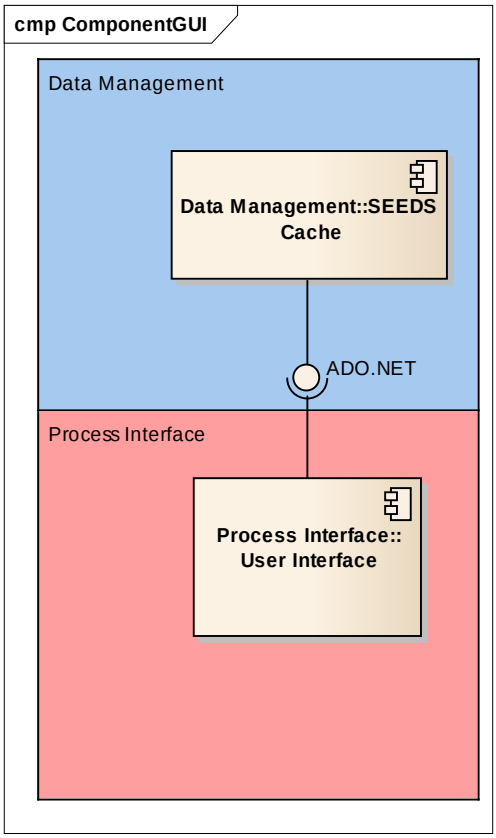

Figure 22: Interface between SEEDS Cache and GUI

The GUI communicates directly with the SEEDS Cache over ADO.Net. ADO.Net is a set of software components included with the Microsoft .NET Framework to access and modify data stored in relational database systems. The system parameters that are displayed on the GUI will be periodically read from the SEEDS cache. This will be realized by periodically sending a SELECT query to the SEEDS Cache. The read data that has been changed is visualized in the GUI. If the user changes data on the GUI, the GUI executes an insert command to the SEEDS Cache. Detailed information about the GUI implementation is given in the report [3] *D6.3 – "Specification of Hardware and Software Platform".*

4.2 Controller to WISAN Interface

This section describes the interface between the Controller and the WISAN Communication Server. If the controller has finished an optimization run and new control settings are available, the WISAN component must be directly and immediate informed about the new actuator commands.

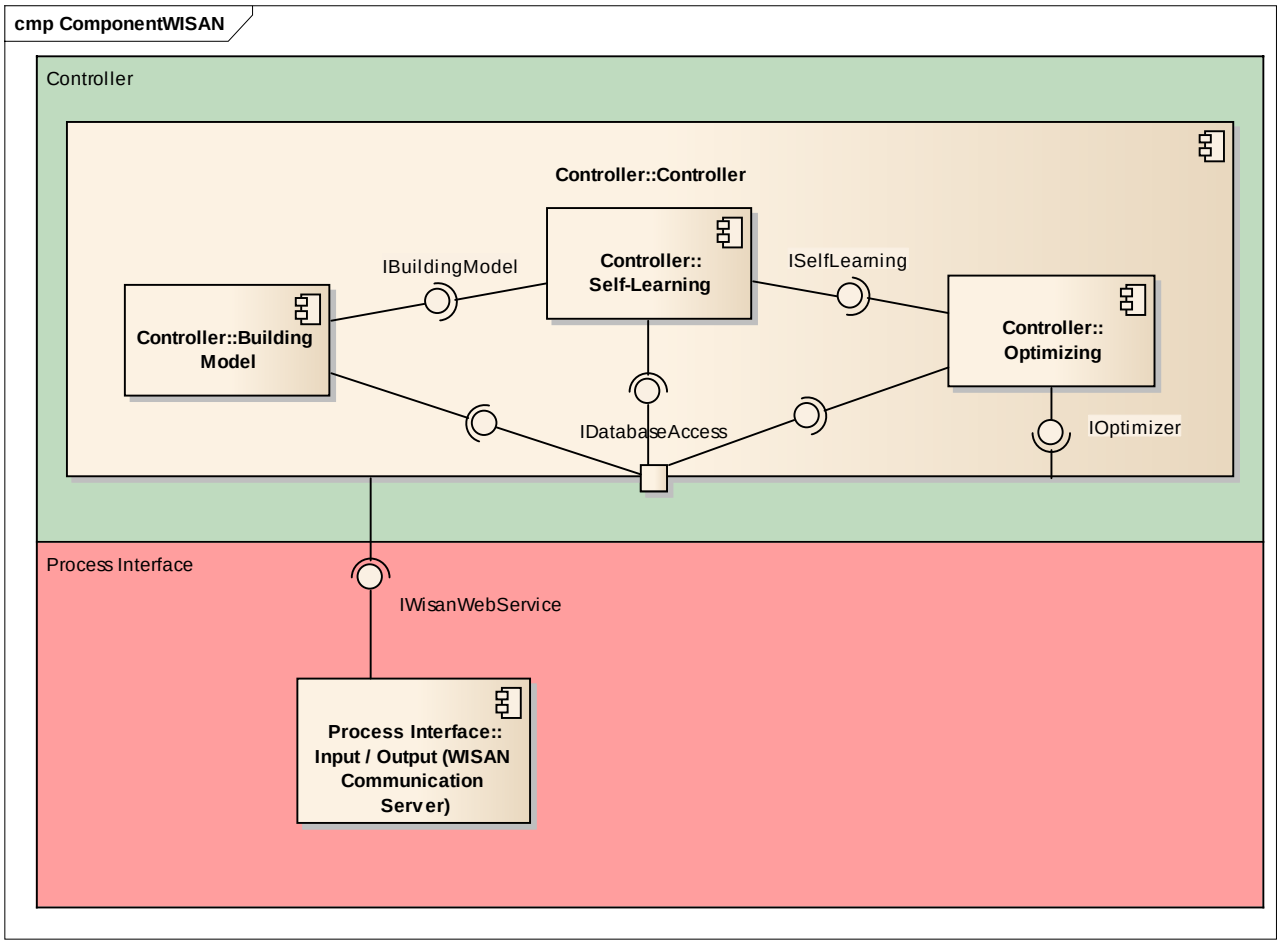

Figure 23: Controller WISAN interface

Detailed information about the WISAN communication infrastructure is given in the report [4] *D4.3 – "Plug & Play conformance requirements: API, Integration Webservices and libraries".*
The WISAN component provides two communication elements for external excess: The "WISAN Client library" and the "Web Service".

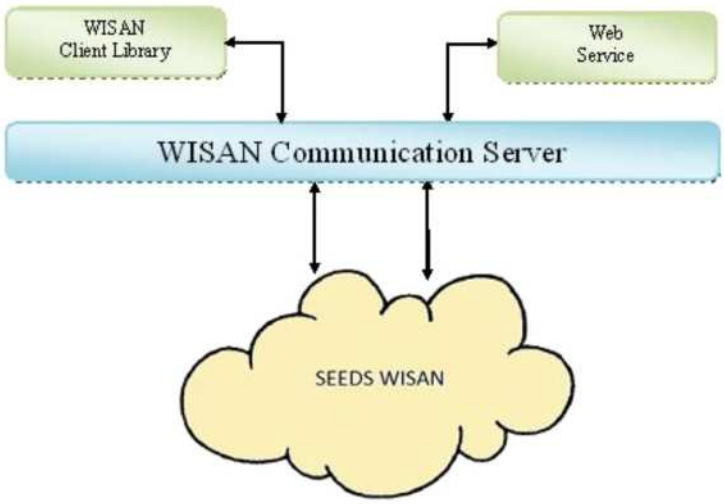

Figure 24: WISAN communication infrastructure

Because of the interoperable interaction over a network, the "Web Service" communication element was chosen to transfer the new control settings to WISAN. For this purpose the `IWisanWebService` interface defines the method `SetActuatorValueByNodeNameAndActuatorName`:

```
bool SetActuatorValueByNodeNameAndActuatorName(string nodeName, string actuatorName,
double value);
```

The actuator is identified by is name and node name so that this information are stored in the controller.

4.3 WISAN SEEDS Cache Interface

This section describes the interface between the WISAN Communication Server and the SEEDS Cache.

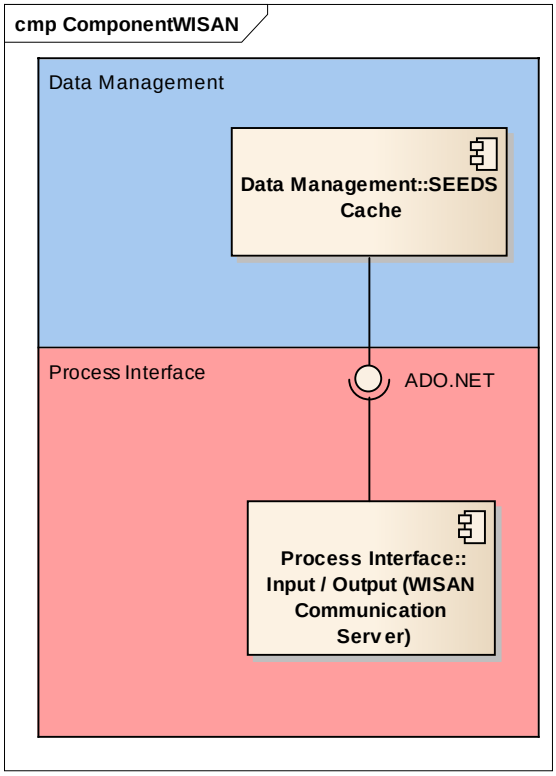

Figure 25: WISAN SEEDS Cache

The WISAN stores the last value of each sensor/actuator in the SEEDS Cache. The WISAN Server uses the ADO .NET Entity framework to update the values of the sensors/actuators in the SEEDS Cache. Detailed information about the WISAN Server implementation is given in the report D4.4 – "Communication infrastructure and tool support implementation".

The ADO.NET Data provider model provides a common managed interface in the .NET Framework for connecting to and interacting with the data store. The ADO.NET Entity framework builds on top of the ADO.NET Data Provider model to allow for use of the Entity Framework with any data source for which a supported provider is available.

5 Summary

Deliverable D5.4 *"Interfacing and Integration within SEEDS architecture"* describes the interfaces and interactions between the SEEDS components. The first part of this deliverable describes the interfaces of the SEEDS controller and its core components. Since all core components are implemented in JAVA and run on the same JAVA virtual machine, a fast data exchange between the core components was realized. By using the OSGi framework, the controller provides a complete and dynamic component model. All implementation of the core components are populated with OSGi Services and therefore are simply to exchange with potentially other implementations.
The second part of this deliverable describes the interface outside the controller, mainly the communication between the central point of the SEEDS architecture: the SEEDS Cache. All SEEDS components have access to it to store or read values. The usage of a standard database as SEEDS Cache provides the option to simply extend the SEEDS system with existing other system components. If these components are in a position to use standard database access techniques like ADO.Net or JDBC they could be easily integrated into the SEEDS System.

All interfaces (except the WISAN to SEEDS Cache) described by this document have been successfully tested on the Helicopter Garage example. Detailed information about the Test Bed and optimization results are given in the report [2] *D2.8 – "Development of energy control strategy"* chapter 4.

6 Bibliography

[1] Márquez, F.; Jiménez, N.; Barragán, A.: D2.3 Modelling Methodology. 2013

[2] Donath U.; Haufe, J; Esteves R.; Montague S.: D2.8 Energy Control Strategy. First Version 2013

[3] Peneva, R.: D6.3 Specification of Hardware and Software Platform. 2013

[4] España, J.; Díaz, F.: D4.3 Plug & Play conformance requirements: API, Integration Webservices and libraries. 2012

Annex A: Abbreviations and acronyms

ADO.Net	ActiveX Data Objects .NET
API	Application Programming Interface
BEMS	Building Energy Management System
D	Deliverable
DoW	Description of Work
FP7	Seventh Framework Programme
GMT	Greenwich Mean Time
GUI	Graphical User Interface
HVAC	Heating, Ventilation Air Condition
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
JDBC	Java Database Connectivity
JVM	Java virtual machine
OSGi	Open Services Gateway initiative
POSIX	Portable Operating System Interface
SEEDS	Self learning energy efficient building and open spaces
UML	Unified Modeling Language
WISAN	Wireless intelligent sensors and actuators network
WP	Work Package
XML	Extensible Markup Language

Annex B: Helicopter Garage Identifiers

Identifier Group:	DeviceControlSetting
Description:	FC_Speed – Speed level [1, 2, 3, 4] of the Fan-Coil1...Fan-Coil10 CH_Load – Load level [0.25, 0.50, 0.75, 1.00] of a Chiller HP_Load – Load level [0.25, 0.50, 0.75, 1.00] of a Heat Pump
Helicopter Garage Example:	*FANCOIL1, FANCOIL2, FANCOIL3, FANCOIL4, FANCOIL5, FANCOIL6, FANCOIL7, FANCOI8, FANCOI9, FANCOI10, CHILLER, HEATPUMP*

Table 5: Identifier Group – Device Control Settings

Identifier Group:	DeviceSettings
Description:	AT – Set point of the air return temperature in °C of the Fan-Coil WTCH – Set point of the water supply temperature in °C of the Chiller WTHP – Set point of the water supply temperature in °C of the Heat Pump
Helicopter Garage Example:	*FANCOIL1, FANCOIL2, FANCOIL3, FANCOIL4, FANCOIL5, FANCOIL6, FANCOIL7, FANCOI8, FANCOI9, FANCOI10, CHILLER, HEATPUMP*

Table 6: Identifier Group - Device Settings

Identifier Group:	EnergyConsumption
Description:	E - Total energy consumption of the Building in kWh over the last 24 hours ERi - Total energy consumption of Rooms in kWh
Helicopter Garage Example:	*ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6, ROOM7, ROOM8, ROOM9, ROOM10, CHILLER, HEATPUMP, PUMP_CHILLER, PUMP_HEATPUMP, TOTAL_24H, TOTAL_SAMPLE*

Table 7: Identifier Group – Energy Consumption

Identifier Group:	DeviceState
Description:	On – device states OnFC – On/Off of the Fan-Coil OnCH – On/Off of the Chiller OnHP – On/Off of the Heat Pump
Helicopter Garage Example:	*FANCOIL1, FANCOIL2, FANCOIL3, FANCOIL4, FANCOIL5, FANCOIL6, FANCOIL7, FANCOI8, FANCOI9, FANCOI10, CHILLER, HEATPUMP*

Table 8: Identifier Group – Device State

Identifier Group:	DeviceWaterReturn
Description:	Water return temperature in °C of the chiller and heat pump
Helicopter Garage Example:	*CHILLER, HEATPUMP*

Table 9: Identifier Group – Device Water Return

Identifier Group:	DeviceWaterSupply
Description:	Water supply temperature in °C of the chiller and heat pump
Helicopter Garage Example:	*CHILLER, HEATPUMP*

Table 10: Identifier Group – Device Water Supply

Identifier Group:	Occupancy
Description:	O – Occupancy of the Room in Number of Persons
Helicopter Garage Example:	*ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6, ROOM7, ROOM8, ROOM9, ROOM10*

Table 11: Identifier Group – Occupancy

Identifier Group:	SolarRadiation
Description:	SR – Solar Radiation Intensity of the Room in the range 0 to 1
Helicopter Garage Example:	*ROOM3, ROOM4, ROOM5, ROOM6, ROOM7, ROOM8, ROOM9, ROOM10*

Table 12: Identifier Group – Solar Radiation

Identifier Group:	TemperatureComfort
Description:	CT – Comfort temperature in °C of the room
Helicopter Garage Example:	*ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6, ROOM7, ROOM8, ROOM9, ROOM10*

Table 13: Identifier Group – Comfort Temperature

Identifier Group:	TemperatureComfortDelta
Description:	Delta Tmax – Maximum delta of the comfort temperatures in °K
Helicopter Garage Example:	*ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6, ROOM7, ROOM8, ROOM9, ROOM10*

Table 14: Identifier Group – Comfort Temperature Delta

Identifier Group:	TemperatureCurrent
Description:	RT – Current temperature in °C of the Room OT – Outdoor Temperature in °C
Helicopter Garage Example:	*ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6, ROOM7, ROOM8, ROOM9, ROOM10, OUTSIDE*

Table 15: Identifier Group – Current Temperature

Annex C: Helicopter Garage OSGi Service Components

OSGI Service component description of the Controller implementation:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="eu.seeds_fp7.controller.impl">
   <implementation class="eu.seeds_fp7.controller.impl.Controller"/>
   <service>
      <provide interface="eu.seeds_fp7.controller.IController"/>
   </service>
</scr:component>
```

OSGI Service component description of the Optimizer implementation:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="eu.seeds_fp7.controller.optimizing">
   <implementation class="eu.seeds_fp7.controller.optimizing.Optimizer"/>
   <service>
      <provide interface="eu.seeds_fp7.controller.optimizer.IOptimizer"/>
   </service>
</scr:component>
```

OSGI Service component description of the Self-learning implementation:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="eu.seeds_fp7.controller.selflearning">
   <implementation class="eu.seeds_fp7.controller.selflearning.SelfLearning"/>
   <service>
      <provide interface="eu.seeds_fp7.controller.selflearning.ISelfLearning"/>
   </service>
</scr:component>
```

OSGI Service component description of the Building Model implementation

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="eu.seeds_fp7.controller.buildingmodel">
   <implementation class="eu.seeds_fp7.controller.buildingmodel.CalculateEnergy"/>
   <service>
      <provide interface="eu.seeds_fp7.controller.buildingmodel.IBuildingModel"/>
   </service>
</scr:component>
```

Annex D: Helicopter Garage method call examples

List of identifiers for calling the Self-learning component in the Helicopter Garage example:

```
public TimePoints[] doSelfLearning(IController controller,
                                   TimePoints[] potentialControlSettings);
```

controller:	Controller object with database access
potentialControlSettings:	Array of potential control settings with the following identifiers: DeviceControlSetting.FANCOIL1, DeviceControlSetting.FANCOIL2, DeviceControlSetting.FANCOIL3, DeviceControlSetting.FANCOIL4, DeviceControlSetting.FANCOIL5, DeviceControlSetting.FANCOIL6, DeviceControlSetting.FANCOIL7, DeviceControlSetting.FANCOIL8, DeviceControlSetting.FANCOIL9, DeviceControlSetting.FANCOIL10, DeviceControlSetting.CHILLER, DeviceControlSetting.HEATPUMP
Returns:	Array of predicted room temperatures and energy consumption with the following identifiers: TemperatureCurrent.ROOM1, TemperatureCurrent.ROOM2, TemperatureCurrent.ROOM3, TemperatureCurrent.ROOM4, TemperatureCurrent.ROOM5, TemperatureCurrent.ROOM6, TemperatureCurrent.ROOM7, TemperatureCurrent.ROOM8, TemperatureCurrent.ROOM9, TemperatureCurrent.ROOM10, EnergyConsumption.TOTAL

List of identifiers for calling the Building Model component in the Helicopter Garage example:

```
public TimePoints[] doEnergyCalculation(TimePoints[] temperatures,
                                        TimePoints[] controlSettings,
                                        TimePoints[] waterSupply,
                                        TimePoints[] deviceStates);
```

temperatures:	Array of current room and outside temperatures with the following identifiers: TemperatureCurrent.ROOM1, TemperatureCurrent.ROOM2, TemperatureCurrent.ROOM3, TemperatureCurrent.ROOM4, TemperatureCurrent.ROOM5, TemperatureCurrent.ROOM6, TemperatureCurrent.ROOM7, TemperatureCurrent.ROOM8, TemperatureCurrent.ROOM9, TemperatureCurrent.ROOM10, TemperatureCurrent.OUTSIDE
controlSettings:	Array of control settings with the following identifiers: DeviceControlSetting.FANCOIL1, DeviceControlSetting.FANCOIL2, DeviceControlSetting.FANCOIL3, DeviceControlSetting.FANCOIL4, DeviceControlSetting.FANCOIL5, DeviceControlSetting.FANCOIL6, DeviceControlSetting.FANCOIL7, DeviceControlSetting.FANCOIL8, DeviceControlSetting.FANCOIL9, DeviceControlSetting.FANCOIL10, DeviceControlSetting.CHILLER, DeviceControlSetting.HEATPUMP
waterSupply:	Array of water supply temperature in °C of the chiller and heat pump with the following identifiers: DeviceWaterSupply.CHILLER,DeviceWaterSupply.HEATPUMP
deviceStates:	Array of device states with the following identifiers: DeviceState.FANCOIL1, DeviceState.FANCOIL2, DeviceState.FANCOIL3, DeviceState.FANCOIL4, DeviceState.FANCOIL5, DeviceState.FANCOIL6, DeviceState.FANCOIL7, DeviceState.FANCOIL8, DeviceState.FANCOIL9, DeviceState.FANCOIL10, DeviceState.CHILLER, DeviceState.HEATPUMP
Return:	Calculated energy consumption in kw/h with the following identifiers: EnergyConsumption.ROOM1, EnergyConsumption.ROOM2, EnergyConsumption.ROOM3, EnergyConsumption.ROOM4, EnergyConsumption.ROOM5, EnergyConsumption.ROOM6, EnergyConsumption.ROOM7, EnergyConsumption.ROOM8, EnergyConsumption.ROOM9, EnergyConsumption.ROOM10, EnergyConsumption.CHILLER, EnergyConsumption.HEATPUMP, EnergyConsumption.TOTAL