PDP Developers Guide

(draft)

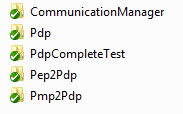
|  |  |  |
| --- | --- | --- |
| Revision history | | |
| Date | Author | Comment |
| 2013-08-10 | Stoimenov | Draft version |
| 2013-08-31 | Stoimenov | Document updated |
| 2013-11-24 | Stoimenov | Event handler dynamic reloading explained |

Notice:

This guide is intended for developers who want to understand PDP code in order to extend it, fix bugs, or modify it in some other way. Users who want to develop PEP and PMP modules and who want to communicate with PDP should read “PDP User Guide”.

# PDP Architecture

PDP stands for Policy Decision Point. The module is implemented in Java and it provides interfaces which can be used by PEP (Policy Enforcement Point) modules and PMP (Policy Management Point) modules. The code of PDP and related stubs is organized into the following projects:



Maven is used for project configuration. The projects have flat structure. Pdp is the parent project. Its pom.xml (maven descriptor file) resides at the same level in the file system hierarchy as its child projects.

CommunicationManager is part of the PDP and it is responsible for the communication between the PDP and other modules such as PMPs and PEPs.

Pep2Pdp contains a stub which can be used by PEP developers to make communication with PDP.

Pmp2Pdp contains a stub which can be used by PMP developers to communicate with PDP.

The communication between PEP and PDP and PMP and PDP is done via TCP. In addition to java socket programming, Google protocol buffers are used. Google protocol buffers enable an efficient way of serialization (fast and compact). Messages to be exchanged are defined in proto files (see CommunicationManager/src/main/resources/proto). Google protocol buffer tool (freely available from Google) is used to generate Java code in form of classes corresponding to these messages. In addition, there is support for programming languages such as C++ and Python. There is also support for other programming languages by third party vendors.

Figure 1 shows all message types used in PDP:

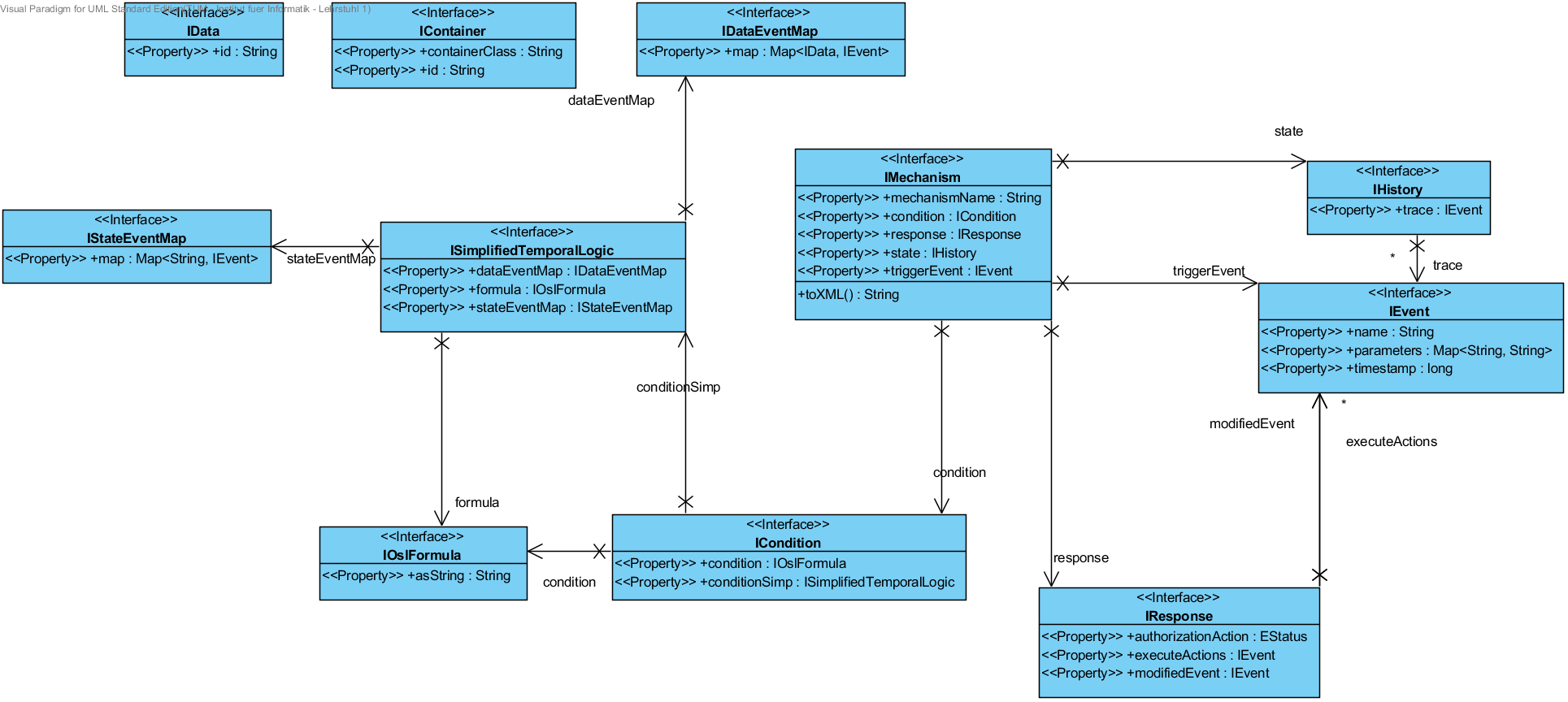


Figure Message Types

## Adding new message types and extending the old ones

To add a new message type, the proto file located in PdpCommunicationManager/src/main/resources/proto/PdpProtos.proto must be edited. This is where the definition of the new message type should be added. For details how to define a google protocol buffer message, refer to the online documentation of google protocol buffers. Google protocol buffer messages should be prefixed with Gp so that they can be easily distinguished from the rest of the code. After that, google protocol buffer compile tool should be used to generate corresponding java code. This is an example of the invocation of the tool (note that here absolute paths are used and they can differ from what is on your machine):

protoc --java\_out=D:\githiwi\CommunicationManager\src\main\java\ PdpProtos.proto

Program protoc is invoked in the directory where PdpProtos.proto is located. Google protocol buffer classes contain many methods which are not relevant to what is needed to program the PDP. Therefore, for each message type a corresponding java interface is created. For instance, if Google protocol buffer message type is Gp<name>, the corresponding java interface will be I<name>.

Examples are: GpData and IData, GpMechanism and IMechanism. The concrete implementations of the interfaces are named <name>Basic. For example, IMechanism interface is implemented by the java class MechanismBasic. MechanismBasic object can be created from a GpMechanism object by invoking the constructor of the MechanismBasic with GpMechanism object as the parameter. MechanismBasic also contains a static method:

**public** **static** GpMechanism createGpbMechanism(IMechanism m)

which can be used to convert IMechanism object instance to GpMechanism object.

The principles explained above hold for all message types.

Extending existing message types includes modifying the proto file, generating java classes from it, adding new methods to the corresponding java interface, implementing these methods in the basic class, and finally modifying the basic class to support the new attribute.

Notice that in basic class equals methods are redefined for the testing purposes. In the equals method two objects are compared by values of their private attributes. In addition, toString() methods are redefined as well for logging purposes.

# Scenario: Communication between a PEP and the PDP

Figure 2 represents class diagram of the communication manager. For the sake of simplicity, only the most important classes are shown.

Figure 3 shows interaction between a *PEP* client and the *PDP*.

Two threads are created during the *PDP* startup: PepFastServiceHandler and PmpFastServiceHandler. They wait for incoming client connections. Each thread listens to a different port. The port numbers can be specified in the file pdp.properties.

The sequence of method invocation will be explained for the case of communication between a *PEP* client and the *PDP*. Similar scenario applies to the case when a PMP communicates with the PDP.

PEP uses Pep2Pdp stub to establish connection to PDP. Connection parameters are IP address and port number of the PepFastServiceHandler. After establishing the connection, PepFastServiceHandler creates an instance of PepClientConnectionHandler which receives incoming messages from the PEP client. PEP then invokes notifyEvent() method on the stub with an instance of *IEvent* as a parameter. The stub will convert IEvent to corresponding instance of Google Protocol Buffer class GpEvent. The google protocol buffer object is received by the PepClientConnectionHandler. It is than converted to an instance of IEvent. The newly created IEvent is placed in the queue which is part of the RequestHandler class. The PepClientConnectionHandler is paused and it waits for the RequestHandler to process the event. The queue might already contain several requests that will be processed in FIFO order. Once the event is processed, the PepClientConnectionHandler is woken up and it replies to the *PEP* client by sending the response created by the RequestHandler.

RequestHandler ‘s queue elements contain all necessary information to uniquely identify the method invoked on the *PDP* and the thread which is responsible for sending the response to the client.

An example which shows how to communicate with PDP from the perspective of a PEP can be seen in Pep2Pdp/src/test/java/ TestPep2PdpCommunication.

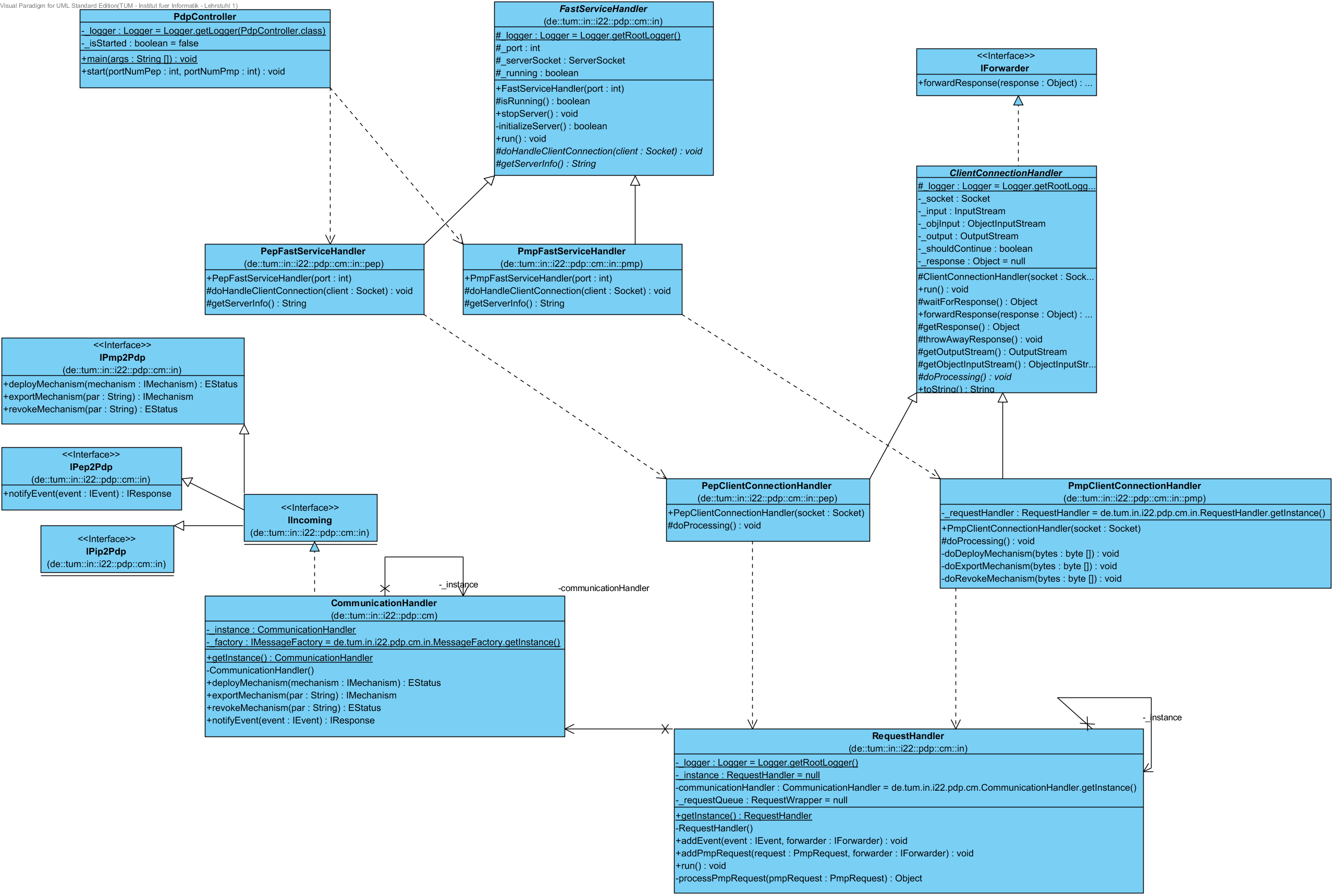


Figure Communication Module Class Diagram

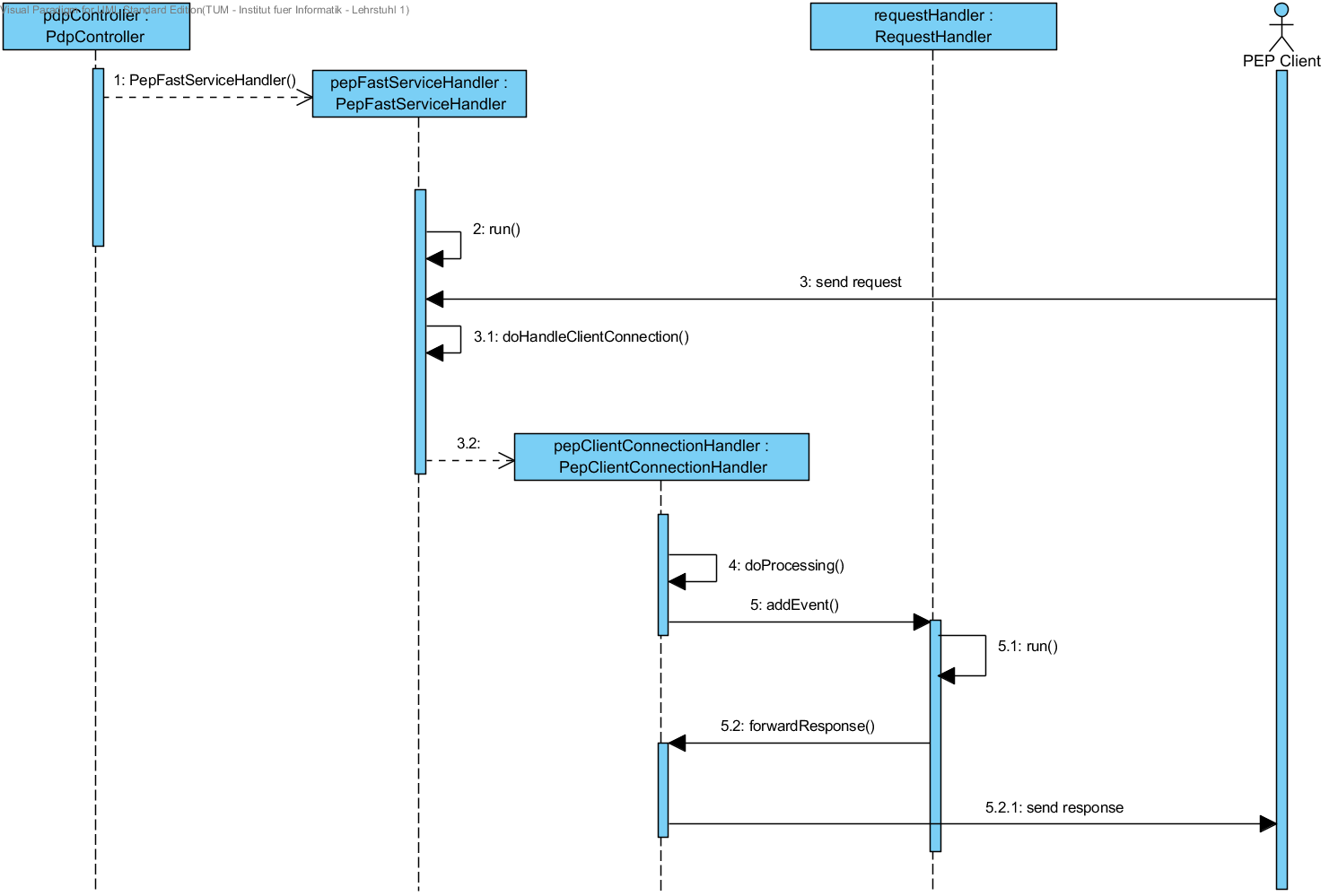


Figure PEP Client - PDP interaction

# Event handler reloading in PipCore

PIP contains the method notifyActualEvent(IEvent event). Based on the event name, the corresponding event handler is created which updates the information flow semantics (IFFLOW). Event handlers are dynamically loaded. An event handler is a class which must implement interface IEventHandler, which contains the method:

IStatus execute();

PIP entry point is PipHandler which implements interface IPdp2Pip. When notifyEvent() method is called on PipHandler, the handler asks EventHandlerManager to create EventHandler. PipHandler than invokes execute() method on EventHandler and returns the response to the invoker.

When PIP is started, event handler definitions are loaded from a derby file database. This database does not require database server and it is created on first application start in the folder where the application resides. PipManager uses the data access object (EventHandlerDao) to load definitions from the database. Each entry loaded from the database contains the full class id (class name plus package name), and the class definition represented as byte array.

A class with the same name can be reloaded by using the same class loader. Lets assume that class loader CL1 loads class A. If at later point we modify the code of class A, we cannot use the same class loader CL1 to reload already loaded class A. We must create another class loader.

Therefore, when the entries from the database are loaded, a hash map is created. The keys are class names, and the values are custom class loaders of type PipClassLoader.

PipClassLoader is initialized to keep the definition of the class as a byte array.

When updateInformationFlowSemantics() method is invoked on PIP, new class definitions are stored in the database, new class loaders are created and the hash map is updated.

EventHandler dynamic loading.emf

Figure 4 Event handler reloading

Contents

[1 PDP Architecture 3](#_Toc373092710)

[1.1 Adding new message types and extending the old ones 5](#_Toc373092711)

[2 Scenario: Communication between a PEP and the PDP 6](#_Toc373092712)

[3 Event handler reloading in PipCore 9](#_Toc373092713)