1. General Idea

The general idea of the project (Figure 1) is to transform raw/basic data that arrives from sensors into a semantically richer information that users can understand and take actions based on it. The engine’s architecture approach will to define two levels of transformation, where the first level transforms the event/stream into a stream of RDF facts using CEP and the second level takes these facts, and runs a continuous query in a stream reasoning system that deduces new facts.

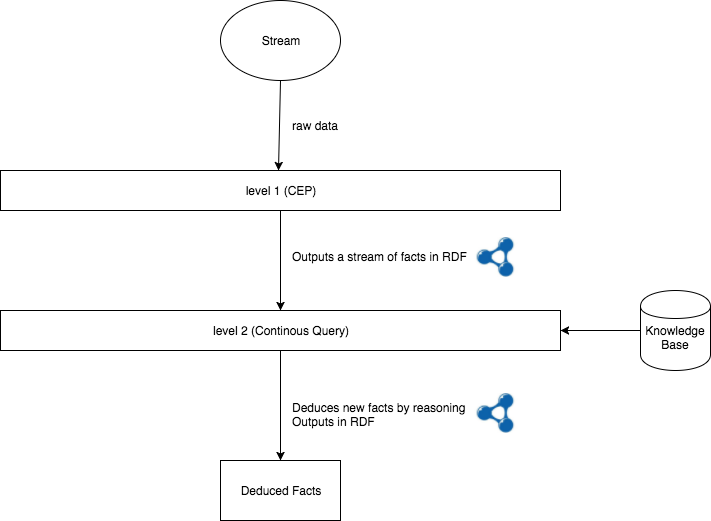


Figure 1 – General Idea

Initially, the first level of processing receives a) an UUID of the smart Thing's sensor and b) the related raw data that tell us if the thing is doing some action, experiencing a state change or any other transformation. For example, if the "Thing" is moving, the CEP engine will receive the UUID and the values of latitude and longitude. Otherwise, if we are monitoring a temperature sensor, it will receive the UUID of the sensor and the temperature value. Then, by matching a sequence of data in a pre-defined time window, it leads to a semantically annotated stream of triples (facts) like <UUID, hasType, HighTemperature>.

The second level of processing takes as input the facts generated by the first level, which was mentioned before. This level runs a continuous stream reasoning engine (e.g. C-SPARQL). The engine uses a Knowledge Base (KB) which provides a background knowledge (conceptual and assertional, T-Box and A-Box of an ontology) about the events and other non-events resources resources of the application domain, such as entities, relations, states. This means that events can be detected based on reasoning on their type hierarchy and their relationship to other objects in the application domain. For example, we have the fact that the temperature sensor with UUID x measured a high temperature value and, as we mentioned before, the first level of processing generates the triple <UUID, hasType, HighTemperature>. Adding the background knowledge that the sensor is connected – i.e. being worn by - called Bill and this person is currently a patient. Doing a reasoning in their type hierarchy leads that the patient Bill has high temperature. Also, we can extend this by adding a rule that high temperature is a sub class of fever, so the reasoner could infer that patient Bill has fever.

One of the benefits of using background knowledge with complex event processing is that we can have a higher expressiveness than if just using CEP. Expressiveness means that an event processing system can precisely express complex event patterns and reactions to events which can be directly translated into business operations. Returning to our example, adding another relationship, to the KB, that the temperature sensor was detected in hospital room numbered 123, it would be possible to precisely infer and inform a nurse or a doctor that patient Bill in room 123 has fever.

We choose C-SPARQL to our continous query module because it is one of the first contributions in stream reasoning. Its open source, easily extendable and well documented. Its source code can be downloaded at Github[[1]](#footnote-1).

The project was developed in Java, under the Spring-Boot framework, a framework for dependency injection. For the unit tests, were choose the jUnit and Mockito frameworks.

2. Scenario

Let us consider a hospital and a future application named Hospital 4.0 (H4.0). A common problem with an emergency sector of a hospital is to detect bottlenecks within the flow of patients. The detection of bottlenecks can help improve the quality of service and satisfaction of the patients.

First, the flow of patients starts at the Reception, where each patient registers his/her entrance. Second, at the Initial Diagnosis Room, a nurse makes a first evaluation of the patients and classifies them into different risk classes. Each risk class has a color and a maximum wait time for the patient to wait for a doctor. At the end of the evaluation, the nurse gives the patient a wrist with his/her corresponding risk color. Third, the patients wait in the General Waiting Room for a doctor to call him/her. Finally, after the meeting with the doctor, depending on the diagnosis, the patient can be sent to the Medication Room, to some of the Exams Rooms (X-Ray or Tomography) for further investigation or even sent back home.

For this scenario, we will consider only the flow of patients leaving/entering the General Waiting Room and the absence of a Doctor. The general organization of the rooms, is presented in figure 2 and the knowledge base class diagram is shown in figure 3.

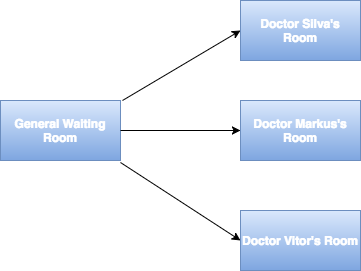


Figure 2 – Organization of the rooms

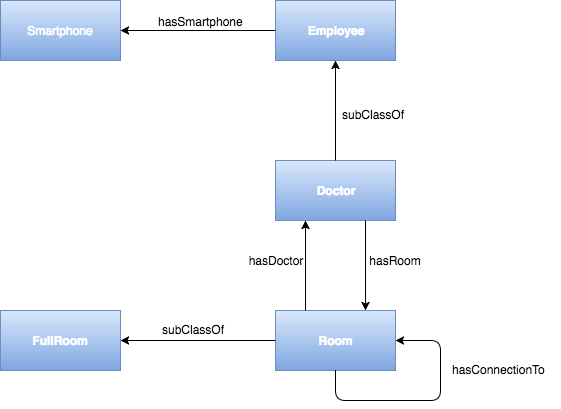


Figure 3 – Class Diagram

2.1 Assumptions

To better understand our two-phase reasoning model, let us consider the following characteristics to detect a bottleneck:

1. GWR become crowed, in other words, it has more patients than seats.
2. Doctor Silva, who is responsible for one of the rooms, did not come to work.
3. Each patient receives a device attached to his/her wrist when they finish the initial diagnosis.
4. Each Employee has a smartphone with Bluetooth technology and is responsible for only one room in the hospital.

2.2 CEP Approach

With these assumptions, we can develop a CEP strategy to serve as a RDF input to our continuous query engine. In the strategy, which is described in figure 4, a blue node represents a CEP operator and an arrow represents a flow of data. The operators are written in the Esper Event Processing Language.

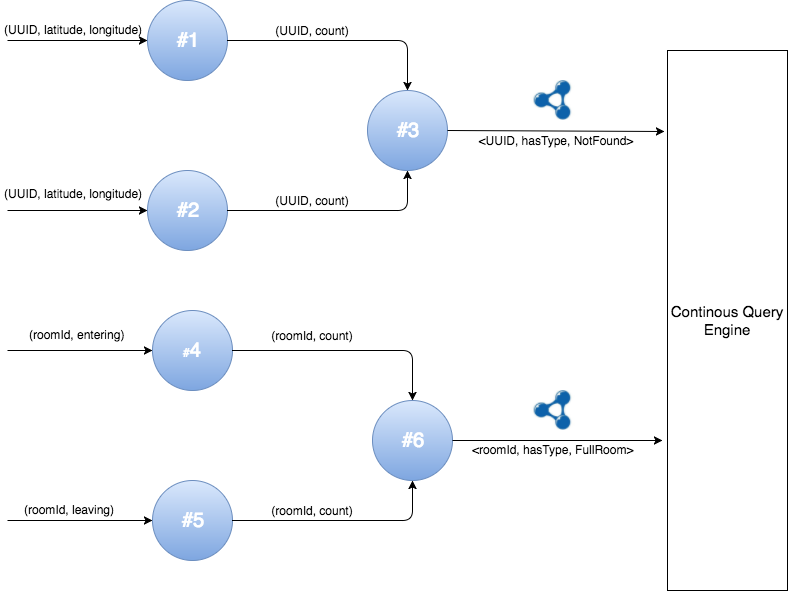


Figure 4 – CEP Strategy

2.3 Reasoning Approach

The second level of our approach, takes as input a stream of RDF facts generated by the CEP operators described in section 5.2. This level runs a continuous SPARQL engine (C-SPARQL) that integrates with a knowledge base. To detect a bottleneck, first we should define two axioms which are showed bellow.

1. DidNotComeToWork **EquivalentTo** Doctor **and** hasSmartphone **some** NotFound
2. BootleneckRoom **EquivalentTo** FullRoom **and** hasConnectionTo **some** (Room **and** hasDoctor **some** DidNotComeToWork)

The axiom (a) tells us that a doctor that did not come to work is deduced by two facts: (1) a doctor has a smartphone and (2) this smartphone was not found. The axiom (b) is equivalent to say that a bottleneck room is a Full Room that is connected to a Doctor’s room that did not come to work. With this information in our KB, we can now detect a bottleneck with the following query shown in figure 10:

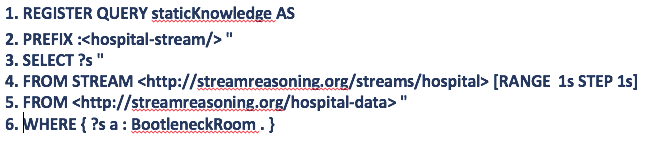


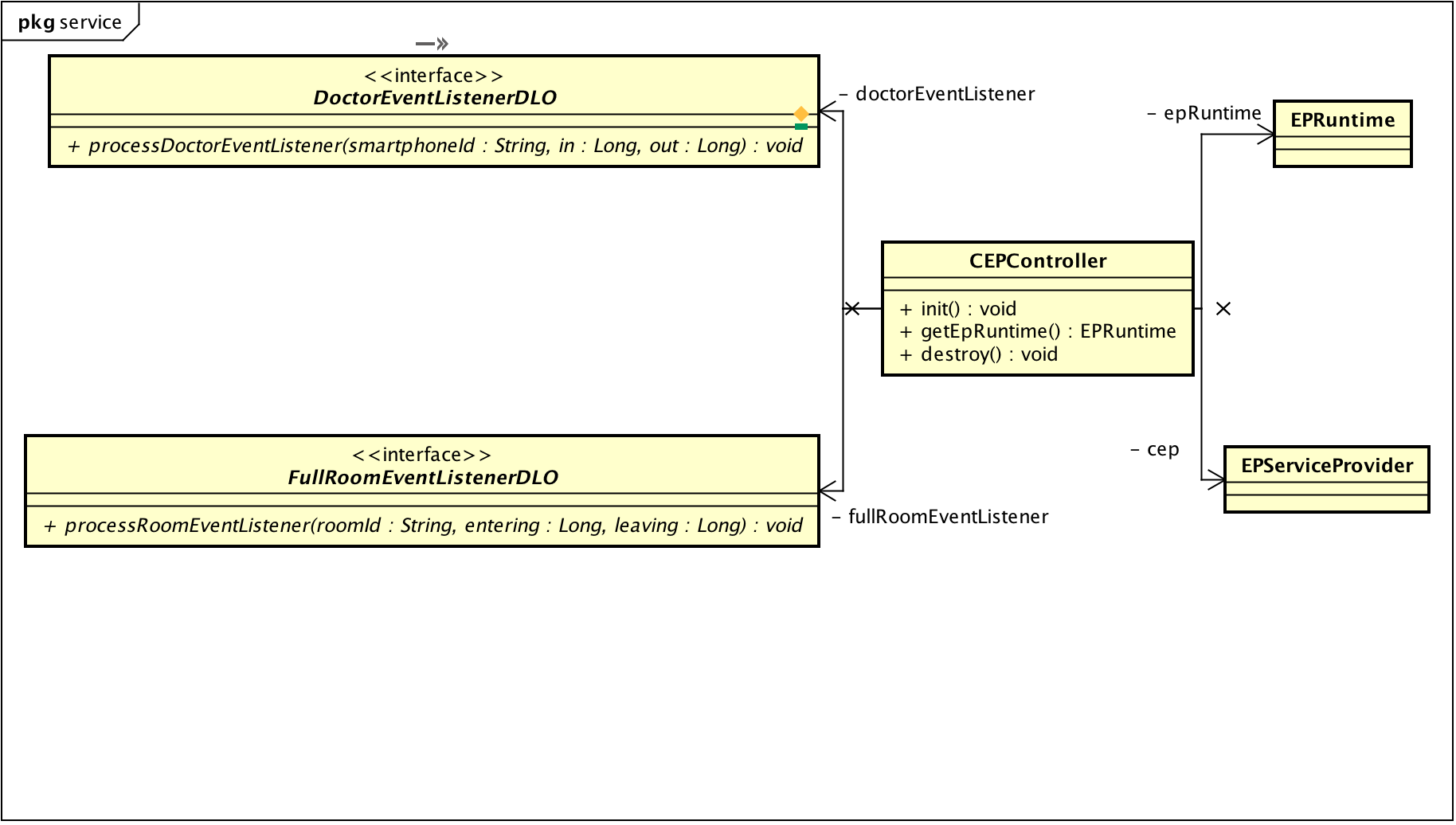
Figure 10 – C-Sparql Query

The REGISTER STREAM clause at line 1 register the continuous query named "staticKnowledge" in the engine. The query considers a sliding window of 1 second that slides 1 second (line 4) and opens the RDF stream (clause FROM STREAM at line 4). The clause FROM at line 5 opens the RDF graph containing the static data of the hospital, which considers the room connection and the doctor’s data. The line 6 matches the room which is a bottleneck room. Finally, line 3 constructs the RDF triples that are streamed out for down stream analysis.

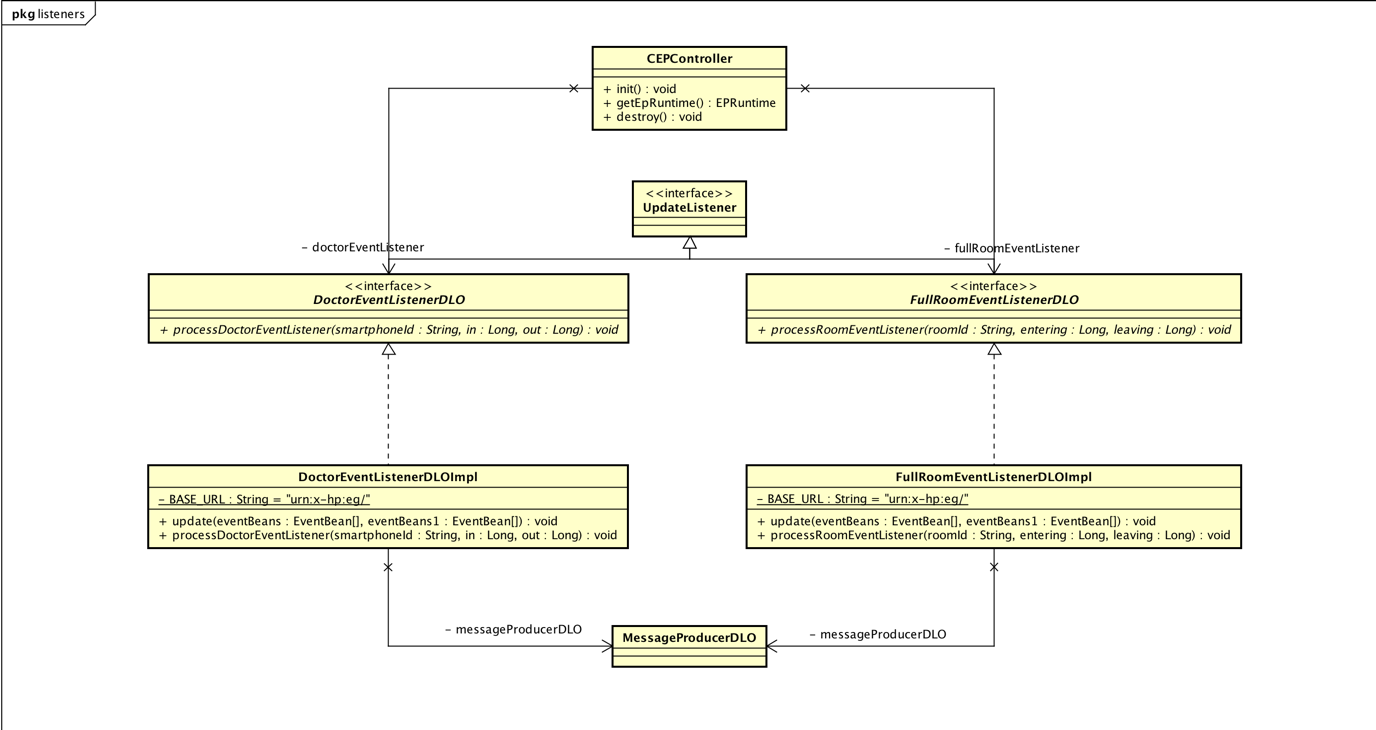
3. Project Structure

The project is divided into three modules. The first, that is called esmocyp-cep, is responsible to run a service that takes incoming streams and apply CEP rules to generate a new stream of semantically annotated RDF statements. The second module is the esmocyp-reasoning. This module takes the stream of RDF statements and insert then into a continous query engine that runs a C-SPARQL query to identify a bottleneck. And the last one, but not less important, is a module that is responsible for the communication between the esmocyp-cep and esmocyp-reasoning modules. This module uses the Apache Kafka Broker for message delivering.

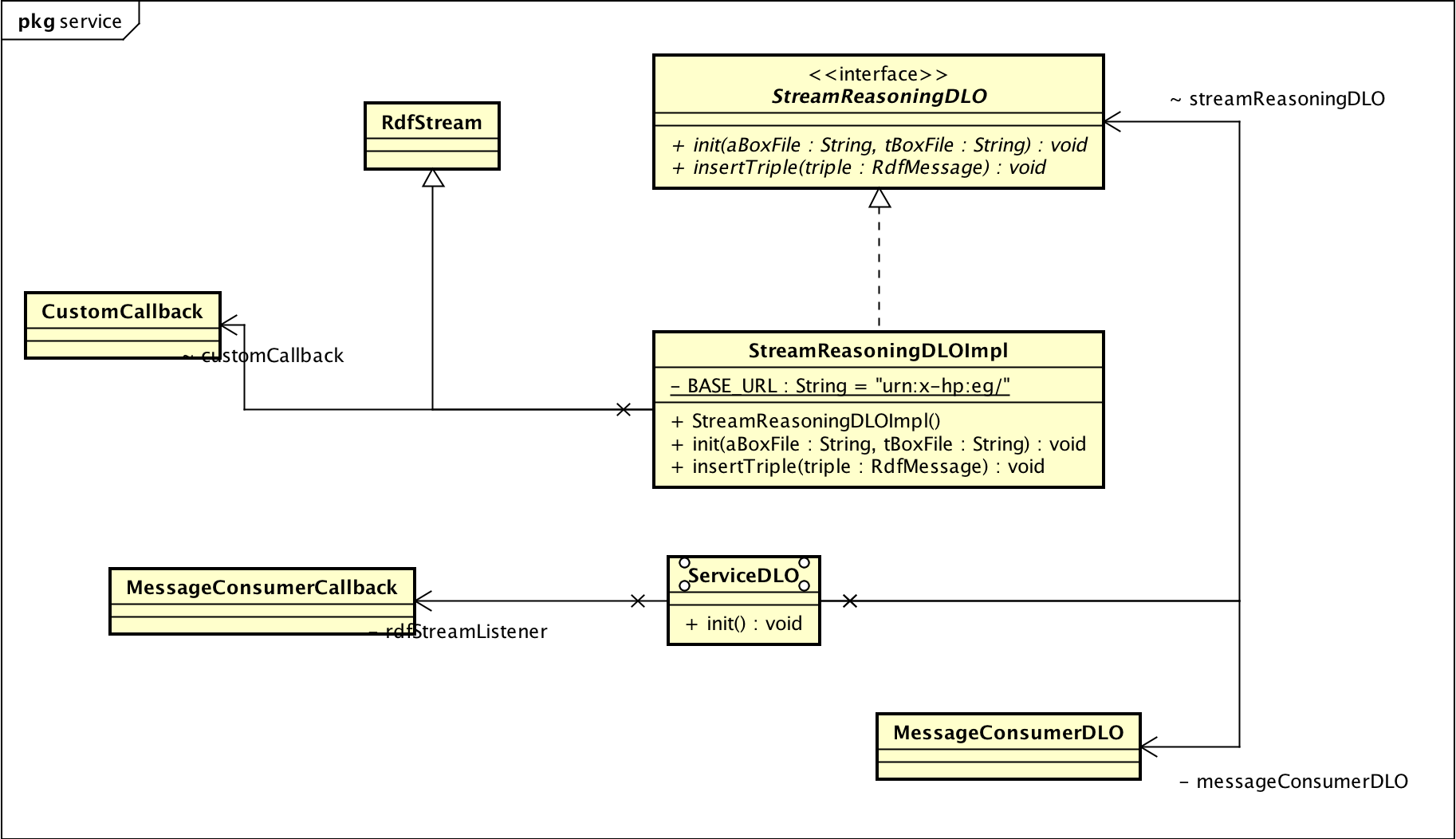
3.1 Diagrams



esmocyp-cep module | package service



esmocyp-cep | package listeners



esmocyp-reasoning | package service

4. Test Cases

|  |  |
| --- | --- |
| Name | TestCEPRuleDoctor1 |
| Description | Simulates that a doctor is not in the hospital area. |
| Input | A stream of geo-location and smartphone UUID. The geo-location is simulated so it is not in the hospital area.  UUID: SMARTPHONE\_TEST  Latidude: 1  Longitude: 1  This sample is generated 10 times |
| Expected Result | The cep rule listener must be called 1 time and the count of “not in hospital area” must be 10. |

|  |  |
| --- | --- |
| Name | TestCEPRuleDoctor2 |
| Description | Simulates that a doctor is in the hospital area. |
| Input | A stream of geo-location and smartphone UUID. The geo-location is simulated so it is not in the hospital area.  UUID: SMARTPHONE\_TEST  Latidude: -22.86691  Longitude: -43.255070  This sample is generated 10 times |
| Expected Result | The cep rule listener must not be called. |

|  |  |
| --- | --- |
| Name | TestCEPRuleDoctor3 |
| Description | Simulates that a doctor is initially in the hospital area but in some moment he is not. |
| Input | A stream of geo-location and smartphone UUID. The geo-location is simulated so it is not in the hospital area.  UUID: SMARTPHONE\_TEST  Latidude: -22.86691  Longitude: -43.255070  This sample is generated 5 times  UUID: SMARTPHONE\_TEST  Latidude: 1  Longitude: 1  This sample is generated 5 times |
| Expected Result | The cep rule listener must be called at least 2 times. The count of “in hospital” must be 5 and the count of “not in hospital” must be 5. |

|  |  |
| --- | --- |
| Name | TestCEPRuleFullRoom1 |
| Description | Simulates a flow of people entering the general waiting room. |
| Input | A stream of entering people.  ROOM ID: ROOM\_TEST  This sample is generated 10 times |
| Expected Result | The cep rule listener must be called 1 time and count of people who entered the room must be 10. |

|  |  |
| --- | --- |
| Name | TestCEPRuleFullRoom2 |
| Description | Simulates a flow of people leaving the general waiting room. |
| Input | A stream of leaving people.  ROOM ID: ROOM\_TEST  This sample is generated 10 times |
| Expected Result | The cep rule listener must not be called. |

|  |  |
| --- | --- |
| Name | TestCEPRuleFullRoom3 |
| Description | Simulates a flow of people leaving and entering the general waiting room. |
| Input | A stream of leaving people.  ROOM ID: ROOM\_TEST  This sample is generated 5 times  A stream of entering people  ROOM ID: ROOM\_TEST  This sample is generated 5 times |
| Expected Result | The cep rule listener must be called at least 1 time. The entering count must be 5 and the leaving count must be 5. |

|  |  |
| --- | --- |
| Name | TestCEPRuleFullRoom2 |
| Description | Simulates a flow of people leaving the general waiting room. |
| Input | A stream of leaving people.  ROOM ID: ROOM\_TEST  This sample is generated 10 times |
| Expected Result | The cep rule listener must not be called. |

|  |  |
| --- | --- |
| Name | TestListenerDoctor1 |
| Description | Tests the generation of “not in hospital” RDF triples |
| Input | UUID: SMARTPHONE\_TEST  In: 0  Out: 10 |
| Expected Result | Must generate the rdf triple:  <urn:x-hp:eg/SMARTPHONE\_TEST,  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,  urn:x-hp:eg/NaoEncontrado > |

|  |  |
| --- | --- |
| Name | TestListenerDoctor2 |
| Description | Tests the generation of “not in hospital” RDF triples |
| Input | UUID: SMARTPHONE\_TEST  In: 10  Out: 0 |
| Expected Result | Must not generate any rdf triple |

|  |  |
| --- | --- |
| Name | TestListenerDoctor2 |
| Description | Tests the generation of “not in hospital” RDF triples |
| Input | UUID: SMARTPHONE\_TEST  In: 10  Out: 10 |
| Expected Result | Must not generate any rdf triple |

|  |  |
| --- | --- |
| Name | TestListenerRoom1 |
| Description | Tests the generation of “full room” RDF triples |
| Input | ROOM ID: ROOM\_ID\_TEST  In: 10  Out: 0 |
| Expected Result | <urn:x-hp:eg/ROOM\_ID\_TEST,  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,  urn:x-hp:eg/SalaCheia > |

|  |  |
| --- | --- |
| Name | TestListenerRoom2 |
| Description | Tests the generation of “full room” RDF triples |
| Input | ROOM ID: ROOM\_ID\_TEST  In: 0  Out: 10 |
| Expected Result | Must not generate any rdf triple |

|  |  |
| --- | --- |
| Name | TestListenerRoom3 |
| Description | Tests the generation of “full room” RDF triples |
| Input | ROOM ID: ROOM\_ID\_TEST  In: 10  Out: 10 |
| Expected Result | Must not generate any rdf triple |

|  |  |
| --- | --- |
| Name | TestReasoningEngine1 |
| Description | Simulates a flow of “smartphone not found” RDF data. |
| Input | The rdf triple:  <urn:x-hp:eg/smartphoneDoRuhan,  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,  urn:x-hp:eg/NaoEncontrado > |
| Expected Result | The cep rule listener must not be called. |

|  |  |
| --- | --- |
| Name | TestReasoningEngine2 |
| Description | Simulates a flow of “full room” RDF data. |
| Input | The rdf triple:  <urn:x-hp:eg/saladeEspera1,  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,  urn:x-hp:eg/SalaCheia > |
| Expected Result | The cep rule listener must not be called. |

|  |  |
| --- | --- |
| Name | TestReasoningEngine3 |
| Description | Simulates a flow of “full room” and “smartphone not found” RDF data. |
| Input | The rdf triples:  <urn:x-hp:eg/smartphoneDoRuhan,  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,  urn:x-hp:eg/NaoEncontrado >  <urn:x-hp:eg/saladeEspera1,  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,  urn:x-hp:eg/SalaCheia > |
| Expected Result | The cep rule listener must not be called at least 1 time and the result must be:  < urn:x-hp:eg/saladeEspera1> |

5. Instructions

The user must have Apache Maven (<https://maven.apache.org/)> installed

Download Apache kafka (<https://kafka.apache.org/downloads>) and install. Follow the instructions under <https://kafka.apache.org/quickstart>

Run mvn install in the core module of the C-SPARQL source code (https:github.com/mobbyd1/c-sparql)

In the esmocyp source code first, run mvn install in the source root. After this, run the Main class under the esmocyp-reasoning module to initialize the continous query module. In the esmocyp-cep module, run the Main class to initialize the cep rules and the test streams.

The console under the esmocy-reasoning must continously output:

urn:x-hp:eg/saladeEspera1 urn:x-hp:eg/salaDoRuhan

1. www.github.com [↑](#footnote-ref-1)