eps PLATFORM

DEVELOPER GUIDE

v1.5.8

Author: Frank Appiah
Project: StreamEPS
Version: v1.0.1

Table of Contents

Preface....3
0.1.Before Getting Started....3
0.2.The TestEvent Class....3
1. 1.Architectural Overview....6
2. 2.Event Assertions....7
2.1.Classic Event Operators....7
a)Writing a classic event assertion....7
2.2.Modal Assertions....8
a)Writing a modal assertion....8
2.3.Trends Assertions....9
a)Writing a trend assertion....10
3. 3.Event Aggregation....14
4. 4.Event Pattern Matching....16
5. 5.Event Filtering....17
5.1.Extending the filter processor....17
5.2.Writing an event filter....17
6. 6.StreamEPS Core Standalone....20
Summary....25

Preface

This is an open source event stream processing system or platform that provides an engine for processing segment-oriented, temporal-oriented, state-oriented and spatial-oriented event stream. It provides support to other processing systems via processing element adapter, PE. It supports distributed, scalable, partially fault-tolerant properties and allow developers to easily develop applications for processing continuous unbounded streams of data. This event processing project is modelled on an event processing model, which has a core base supporting some common features in event processing engines.
The platform currently supports the following processing pipelines that is provided by the engine. These processing include the evaluation of event assertion functions; the basic operators, logic operators, modal operators and trend assertions:- increasing, decreasing, non-increasing, non-decreasing and mixed trends; pattern match detects; event filtering :- comparison filter, range filter and in-not value set filter.

0.1. Before Getting Started

The minimum requirements to run the examples which are introduced in this chapters are only two; the latest version of StreamEPS libraries and JDK 1.6 or above. The latest version of StreamEPS is available in the project download page. To download the right version of JDK, please refer to your preferred JDK vendor's web site. As you read, you might have more questions about the classes introduced in this chapters. Please refer to the API reference whenever you want to know more about them. All class names in this document are linked to the online API reference for your convenience. Also, please don't hesitate to contact the StreamEPS project community and let us know if there's any incorrect information, errors in grammar and typo, and if you have a good idea to improve the documentation.

0.2. The TestEvent Class

The structure of this TestEvent class is used throughout this guide.

```
public class TestEvent extends EventObject {
   private String name;  /**The variable name*/          private Double value;    /**The variable value*/
   public TestEvent() {  }
   public TestEvent(String name, Double value) {
      this.name = name;
      this.value = value;}
   /** The rest are the setter/getter methods of the variables.*/
   @Override
   public String toString() {
      return "Name:" + name + "----Value:" + value;
   }
}
```

Source 1: The source code of the TestEvent

Every basic event atom show minimally implement or extend the EventObject. The event object is

an event with just a payload content with Payload class type and it also extends the header in order to inherit the properties/attributes of the header object. The header type is made up of a 9-tuple structure <*identifier*:String, *isComposable*:boolean, *chronon*: ChrononType, *occurrenceTime*:Date, *eventCertainty*:Float, *eventAnnotation*:String, *detectionTime*:Date, *eventSource*:String, *eventIdentity*:String>. The other events types in the StreamEPS project includes the following interface classes IStreamEvent, IDatabaseEvent, IPrimitiveEvent, an IEncrpytedEvent and a special event called ICurrentTimeEvent which carries along its payload the current time of the user system. The listeners used in the guide are the TestPatternMatchListener and TestUnPatternMatchListener which will explained in the subsequent sections. These are used to observe the events that are matched and unmatched with latter using the TestUnPatternMatchListener. The matched and unmatched events are stored in a IMatchEventMap and IUnMatchEventMap respectively.

This document guide is **not** fully complete and will deem it necessary that any errors should be forwarded to the author for immediate correction. It also does not adhere to any publishing house and it is solely the authors own work.

1. Architectural Overview

This section provides the architectural overview of the StreamEPS by a layer view approach as shown below. It depicts the building blocks of StreamEPS more importantly. The major architectural decisions taken will not be exhausted here. Please refer to the other document on the conceptual view of StreamEPS.

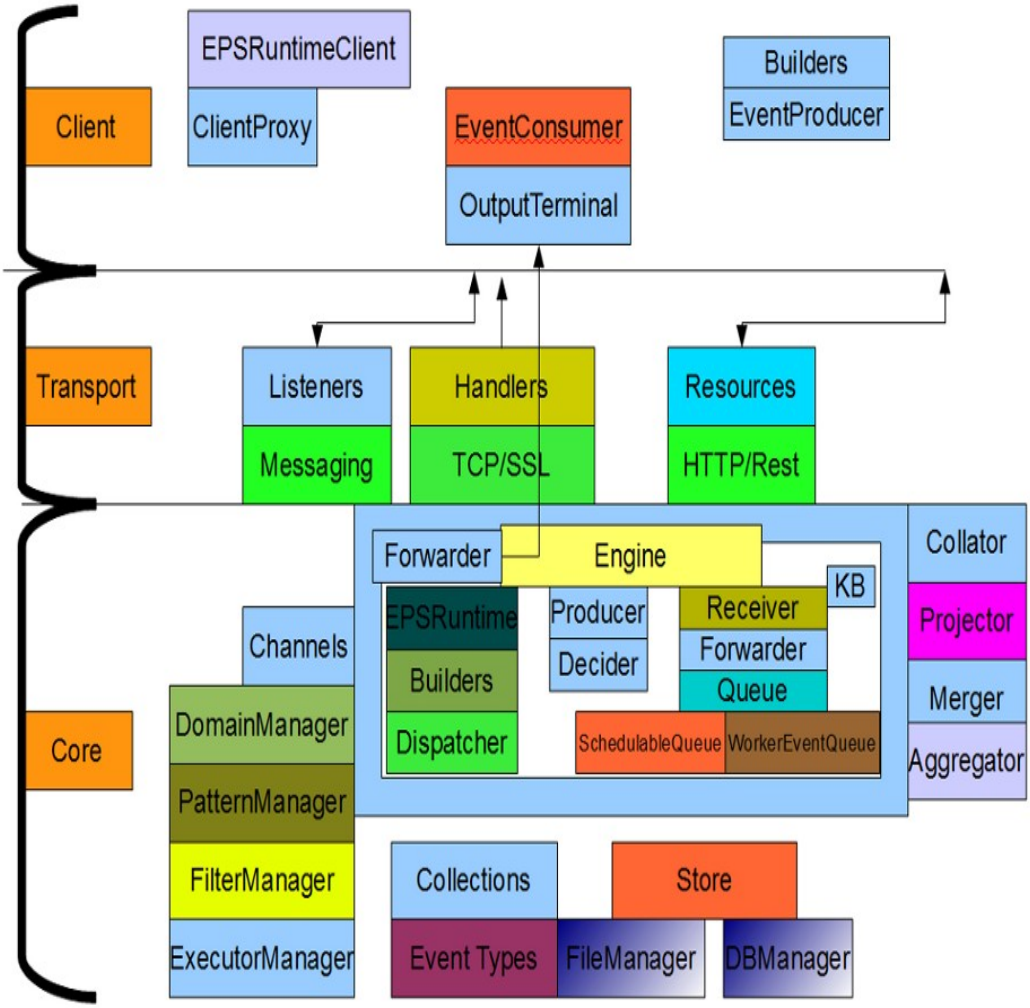

Figure 1: The building blocks of StreamEPS

2. Event Assertions

2.1. Classic Event Operators

Classic event operators that involves conjunction, disjunction, negation and a compound mix-in of any of the event operators:
A conjunction event operator simply asserts the logic of an event that satisfies a list of pattern parameters after the assertion valuations are performed on the event. The evaluation of an event will mean that a property on the event is compared with a *pattern parameter* that consists of <*property_name*, o*perator*, *value*>. The property name in the pattern parameter must be present in the structure of the event object otherwise an illegal argument exception is thrown. Basically, this is the classic logic symbols: ∨, ∧, and ¬ .

pattern parameter(s): <*value, > , 15*> , <*value, <, 90*>

event: <name:E1, value:80>

operator: ∧

evaluation: true

Text 2: A logical evaluation

Here, the logic compound mix-in includes the evaluation of two or more event expression of the same type or not. For example, if the relevant event types {E1, E2, E3} must satisfy the assertion: *E1.value* > *E2.value* **and** *E2.value* > *E3.value*.

a) Writing a classic event assertion

As shown in Source Listing 2 is the LogicExample code snippet that provides an introduction to the LogicalPattern in the below.

1. It creates an instance of the logicalPattern called logical with the parametrized type, TestEvent.
2. The TestPatternMatchListener implements the IPatternMatchListener which receives an instance of the IMatchEventMap class after the *processEvent* callback; the container, an instance of the Dispatchable class and an optional list of objects passed from the pattern processor. The TestUnPatternMatchListener implements the IPatternUnMatchListener which receives an instance of the IUnMatchEventMap class; the container - eventMap, an instance of the Dispatchable class and also an optional list of objects.
3. An instance of the pattern parameter with the property name “value” is added to the list of pattern parameters.
4. Here 40 events of the TestEvent instance is created iteratively and each at the same time is send to the logicalPattern to be processed. Finally, the pattern is outputted to forward both the matched event map and unmatched event map if any.

```
import org.streameps.dispatch.Dispatchable;
import org.streameps.operator.assertion.logic.AndAssertion;
import org.streameps.processor.pattern.LogicalPattern;

public class LogicExample{
public static void main(String[] args){
   LogicalPattern<TestEvent> logical=new LogicalPattern<TestEvent>();          1
   logical.setAssertion(new AndAssertion() );
   logical.getMatchListeners().add(new TestPatternMatchListener() );           2
   logical.getUnMatchListeners().add(new TestUnPatternMatchListener() );
    IPatternParameter pp = new PatternParameter("value", "N/A", 0);            3
     pe.getParameters().add(pp);
    Random rand=new Random(10);
      for (int i = 0; i < 40; i++) {                                           4
        TestEvent event = new TestEvent("E" + i, ((double) rand.nextDouble())+ 29-(1.987 * i) );
        logical.processEvent(event);
      }
      logical.output();
 }
}
```

Source 2: A logical pattern example

2.2. Modal Assertions

Modal event evaluations involve the assertion of *always* and *sometimes* operators. The *always* assertion must satisfy all the pattern parameter list on the event to be evaluated on. In evaluation terms, an always assertion operator is equivalent to the classic event conjuncture. Conversely, the sometimes assertion operator will evaluate the event with pattern parameter list to determine whether one or more of the parameter's is satisfied. Similarly, there are three modal compound event assertions namely the *andCompounModal*, *notCompoundModal* and the *orCoumpoundModal*. The compound modal is a mix-in of *always* and *sometimes* modal operators with a logic operators.

a) Writing a modal assertion

As shown in Source Listing 3 is the *ModalExample* code snippet that provides an introduction to the ModelPatternPE in the above.

```
import org.streameps.dispatch.Dispatchable;
import org.streameps.operator.assertion.modal.AlwaysAssertion;
import org.streameps.processor.pattern.ModalPatternPE;

public class ModalExample{
public static void main(String[] args){
   ModalPatternPE<TestEvent> modal=new ModalPatternPE<TestEvent>();                    1
   modal.setAssertion(new AndAssertion() );
   modalgetMatchListeners().add(new TestPatternMatchListener() );                      2
   modal.getUnMatchListeners().add(new TestUnPatternMatchListener() );
  PatternParameter pp = new PatternParameter("value", "N/A", 0);                       3
      pe.getParameters().add(pp);
   Random rand=new Random(10);
      for (int i = 0; i < 40; i++) {                                                   4
         TestEvent event = new TestEvent("E" + i, ((double) rand.nextDouble())+ 29-(1.987 * i) );
         modal.processEvent(event);
      }
      modal.output();
 }
}
```

Source 3: The source code of the ModalExample

1. It creates an instance of the ModalPatternPE called modal with the parametrized type TestEvent.
2. The TestPatternMatchListener implements the IPatternMatchListener which receives an instance of the IMatchEventMap class; the container, an instance of the Dispatchable class and an optional list of objects. The TestUnPatternMatchListener implements the IPatternUnMatchListener which receives an instance of the IUnMatchEventMap class; the container - eventMap, an instance of the Dispatchable class and an optional list of objects.
3. An instance of the pattern parameter with the property name “value” is added to the list of pattern parameters.
4. Here 40 events of the instance TestEvent is created iteratively and each at the same time is send to the logicalPattern to be processed. Finally, the pattern is outputted to forward both the matched event map and unmatched event map if any.

2.3. Trends Assertions

Trend Assertions provide the evaluation of events from a stream source to assert whether or not it is increasing, decreasing, non-increasing, non-decreasing or mixed which each will require that is the trend object is provided. The trend object interface, IT*rendObject* consists of an *objectID*: Time-stamp, *attribute:* String, a *trend list:* List which is the schema property of the set of participating events from the stream source extracted from the property attribute defined in the trend object. A constraint on the TrendObject specification (Source 1) is that the attribute value must exist exactly

as specified in the event structure to escape an illegal argument exception. The text as shown in Text 2 and Text 4 explains both the increasing assertion trend and stable assertion trend evaluations. The trend type includes an enumeration of string values: increasing, decreasing, non-decreasing, non-increasing, stable and mixed. On the other hand, the non-decreasing pattern is satisfied if the value of a given attribute does not decrease within the given context of provided set of participant events. It assess if e1 << e2 which implies that *e1.A <= e2.A* where A is a numeric attribute to be compared and E(e1,e2) is an event of a specific type. The non-increasing pattern is satisfied if the value of a given attribute does not increase within the given context. It assess if *e1 << e2 => e1.A >= e2.A* where A is a numeric attribute to be compared.

The increasing assertion is satisfied if the value of a given attribute increases strictly monotonically as we move forwards via the set of participant events.

It assess if e1 << e2 => e1.value < e2.value where value is a number attribute to be compared.

Text 4: An Increasing Assertion text

```
public interface ITrendObject<E> {


   public void setObjectId(String timestamp);
   public String getObjectId();
   public void setAttribute(String attribute);
   public String getAttribute();
   public List<ISchemaProperty<E>> getTrendList();
   public void   setTrendList ( List<ISchemaProperty<E>> trendList);
}
```

Source 1: The specification of the trend object.

The stable assertion is satisfied if the given attribute has the same value in all the participant events. The stable assertion is satisfied by an attribute value if for all the participant events, e1 << e2 ==> e1.value == e2.value.

Text 5: A stable assertion text

a) Writing a trend assertion

In writing a trend assertion, there are four classes mainly needed to construct a meaningful runtime template. These classes include the specific trend assertion type (*increasing* with the class name : IncreasingAssertion, *decreasing* with the class name: DecreasingAssertion, *non-increasing* with its class type: NonIncreasingAssertion, *non-decreasing* which also has a class name: NonDecreasingAssertion, *mixed* trend with the class name: MixedAssertion and the *stable* class, StableAssertion), pattern *match listener* with the interface name: IPatternMatchListener, pattern *unmatched listener* interface with name: IPatternUnMatchListener and most importantly the list of

pattern parameters which each has the interface: IPatternParameter.

```
package org.streameps.example;
import org.streameps.operator.assertion.trend.DecreasingAssertion;
import org.streameps.processor.pattern.PatternParameter;
import org.streameps.processor.pattern.TrendPatternPE;
public class TrendExample {

public static void main(String[] args) {
    TrendPatternPE pe = new TrendPatternPE();                                    1
     pe.setDispatch(new TestDispatcher());
     pe.getMatchListeners().add(new TestPatternMatchListener());                 2
     pe.getUnMatchListeners().add(new TestUnPatternMatchListener());             3
     PatternParameter pp = new PatternParameter("value", "N/A", 0);              4
     pe.getParameters().add(pp);
     pe.setAssertion(new DecreasingAssertion());                                 5
     Random rand=new Random(10);
     for (int i = 0; i < 40; i++) {
       TestEvent event = new TestEvent("E" + i, ((double) rand.nextDouble())+ 29-(1.987 * i) );   6
       pe.processEvent(event);
     }
     pe.output();                                                                7
  }
}
```

Source 4: A trend example snippet

The main trend example, TrendExample as shown in Source 4 shows the combination of this classed in used. There is an optional class with class type, Dispatchable for the dispatcher service handling operation. In the snippet below, the source code depicts a trend pattern processing element used to detect a series of test events called TestEvent.

1	This is a trend pattern processing element (PE) which is initialised to create an instance, pe of the TrendPatternPE class.
2	The TestPatternMatchListener class is just an implementation of the IPatternMatchListener interface which is used to forward events during the trend processing on a matched event.
3	A new instance of the TestUnPatternMatchListener class is created and added to the list of pattern unmatched listeners which receives events when an event does not match the pattern parameter during the trend assertion evaluation.
4	A new instance of the IPatternParameter is created with the property name set to “value”. The property name is used to retrieved the property value of the event to get an instance

1 This is a trend pattern processing element (PE) which is initialised to create an instance, pe of the TrendPatternPE class.

of a ISchemaProperty which is then added to *property cache*, IEventPropertyCache. That new instance is then added to the list of pattern parameters.

5 The specific trend type which is the decreasing trend type is initialised and set to the assertion mutator of the TrendPatternPE.

6 A new TestEvent is created with its name "Ei" and a random value of type double where i is in [0,40]. The created event is then sent for processing by the TrendPatternPE, pe.

7 Finally the *output* method of the TrendPatternPE, pe is called to forward and immediately complete the pattern process.

As shown in Source 5 is the pattern match listener code snippet that provides a listener for the trendExample in the above.

```
package org.streameps.example;
import java.util.concurrent.LinkedBlockingQueue;
import org.streameps.core.util.SchemaUtil;
import org.streameps.dispatch.Dispatchable;
import org.streameps.processor.pattern.listener.IMatchEventMap;
import org.streameps.processor.pattern.listener.IPatternMatchListener;

public class TestPatternMatchListener<T> implements IPatternMatchListener<T> {  1

   public void onMatch(IMatchEventMap<T> eventMap, Dispatchable dispatcher, Object... optional) {  2
      System.out.println();
      System.out.println("Match event Listener");
      for (String eventname : eventMap.getKeySet()) {
         LinkedBlockingQueue<T> queue = eventMap.getMatchingEventAsObject(eventname);
         for (T event : queue) {  3
            System.out.println("Event:====Name:" +
                  SchemaUtil.getPropertyValue(event, "name") + "==Value:" +
                  SchemaUtil.getPropertyValue(event, "value"));
         }
      }
   }
}
```

Source 5: The test pattern match listener.

1 The TestPatternMatchListener implements the IPatternMatchListener which receives an instance of the IMatchEventMap class, an instance of the Dispatchable class and an optional list of objects.

2 The *onMatch* method is called with the *eventMap* from the TrendPatternPE processor and the other parameter.

3 The queue (blocked + linked) in the eventMap is then retrieved and iterated to get the property values <”name”, “value”>.

As shown in Source 7 is the pattern unmatched listener code snippet that provides a listener for the trendExample in the above.

```
package org.streameps.example;
import java.util.concurrent.LinkedBlockingQueue;
import org.streameps.dispatch.Dispatchable;
import org.streameps.processor.pattern.listener.IUnMatchEventMap;
import org.streameps.processor.pattern.listener.IPatternUnMatchListener;


public class TestUnPatternMatchListener<T> implements IPatternUnMatchListener<T> {        1

   public void onUnMatch(IUnMatchEventMap<T> eventMap, Dispatchable dispatcher, Object... optional) {  2
      System.out.println();
      System.out.println("Unpattern Match Listener....");
      for (String eventname : eventMap.getKeySet()) {  3
         LinkedBlockingQueue<T> queue = eventMap.getUnMatchingEventAsObject(eventname);
         for (T o : queue) {
            System.out.println("Event:=" + (o).toString());
         }
      }
   }
}
```

Source 7: The test pattern unmatched listener

1 The TestUnPatternMatchListener implements the IPatternUnMatchListener which receives an instance of the IMatchEventMap class; the container, an instance of the Dispatchable class and an optional list of objects.

2 The *onUnMatch* method is called with the *eventMap* from the TrendPatternPE processor and the other parameter.

3 The queue (blocked + linked) in the eventMap is then retrieved and iterated to get the property values <”name”, “value”>.

3. Event Aggregation

The implementation of an event stream aggregation provided by StreamEPS supports the following aggregation functions like count, minimum, maximum, mode, etc. Unimplemented functions can be implemented by extending the *IAggregation* interface. The aggregation interface is a parametrized class of two parameter types;

- *<T>* : The accumulator for the aggregation process
- *<S>:* The return type after the aggregation process.

There are two main process phases of the aggregation process that includes *process* callback which is basically consistent of a sub-task of two other processes: the *executeInitiator* process call and *executeIterator* process call and the *output* callback which just involves the *executeExpirator* process call. All these processes takes place in a processor called *EventAggregatorPE* which implements a policy called an AggregatePolicy. The aggregate policy is a special user-defined policy that is evaluated at each point in the three sub-processes namely the *execute** processes to serve as a further separation of concern to the user of this processor. At the *executeInitiator* point of execution, the specified *property name* of event is used to extract the *property value* of the event from the stream source and return it to the next executor. For the *executeIterator,* the return value extracted from the event in the earlier *execute* process is then populated into the *accumulator* which is sorted. Finally, the accumulated value map returns a list of only the value set, which is iteratively processed by the aggregation instance defined in the processor. After the aggregation process evaluation, the aggregate is then passed to a listener set to propagate the aggregate result to the client via the set listener. This listener is called the *AggregatorListener* that has only two methods with the most important called *onAggregate* callback. The second is a mutator for setting the aggregate context with the interface IAggregateContext. The populate-sorted accumulator is cleared and the *executeExpirator* of the aggregate policy then invoked. It is recommended that any further extension of the aggregation processing element follows the same functional breakdown of processes.

```
import org.stremeps.core.util.IDUtil;
import org.streameps.processor;
import org.streameps.aggregation.SumAggregation;

public class AggregatorExample {

public static void main(String[] args){
    IEventAggregatorPE aggregator=new EventAggregatorPE( IDUtil.getUniqueID("12345609eps"), "value");   1
    aggregator.setAggregatorListener(new TestAggregatorListener() ); //user must implement
    aggregator.setAggregation(new SumAggregation());
    aggregator.setAggregatorPolicy(new TestAggregatePolicy() ); //user must implement
    Random rand=new Random(10);
      for (int i = 0; i < 40; i++) {
        TestEvent event = new TestEvent("E" + i, ((double) rand.nextDouble())+ 29-(1.987 * i) );   2
        aggregator.process(event);
      }
     aggregator.output();                                                                          3
  }
}
```

Source 6: The Aggregation Function Example

1. The new instance of the processor, EventAggregatorPE is initialised with a unique identifier and the property name for the aggregation evaluation defaulted to “value”. A new instance of the TestAggregatorListener is created which is implemented from the AggregatorListener class and then set to the processor. The specified aggregation function is also then set which is in this case the SumAggregation implementing IAggregation. A user defined specific aggregate policy can also be passed on to the processor with the custom implementation. Here, the implementation of the listener is an instance of the TestAggregatorListener depicted earlier in the guide.
2. After the initialisation processes, what is needed is to send some events to the aggregation processor. A new instance of the TestEvent is created with an iterative name and a random generated value defined as shown above. Here, 40 events are send synchronously to the processor until an incremental count, i of 40 is reached.
3. Finally, on the account of sending events the output of the aggregation processor is called. This will then push the aggregate result value, Aggregation to the TestAggregatorListener by the *onAggregate* method call.

4. Event Pattern Matching

The pattern matching processor supports an enumerated list of patterns defined by the PatternType including highest subset pattern defined by the class, HighestSubsetPE; lowest subset pattern defined by the class, LowestSubsetPE, a threshold average pattern created by the class ThresholdAveragePE (this is part of the collection of aggregation pattern detect processors), a pattern detect for a trend with a thorough description in the earlier sections, a logical and modal patterns also described above. There is a special pattern processor called a function pattern, which provides an interface, IFunctor for the developers to create functions not defined in the platform. This function pattern processor is one of the other kind to the aggregation pattern which lets the developers to decide and make a pick to suit its need.
On the functional point, there is a so-called null pattern which is used to serve events from the stream source without any processing or so ever. This is defined in the org.streameps.processor.pattern.NullPattern class to provide this special functionality. This is really needed in the engine processing pipeline and you can find further information in the programmer guide, which provides more detail and important info on these pattern compositions. Each of these patterns extend the IBasePattern which provides the listeners (IPatternMatchListener and IPatternUnMatchListener) for clients to observe both the pattern matched and pattern unmatched activities, adding pattern parameters, getting access to the participant events/ matched events by a pull process, a define list of pattern policies and a user defined pre-post aware but defaulted to the class DefaultPrePostProcess. The pattern policies include the cardinality, consumption, evaluation, order and repeated policy.

There are two processor entries in the pattern matching detects for getting the functionality as specified by the pattern processor. The pattern processor with the interface, IPatternProcessor provides the interface/contract for specifying pattern matching processes defined by the contract method, *process*. There are two process methods one to iteratively add events to the participant event set provided by the class *IParticipantEventSet* and the other with the instance of the *SortedAccumulator*. These collections are provided in the org.streameps.aggregation.collection package. The second parameter of the *process* method for each process takes the implementation of the IBasePattern. The flexibility of this type of processor is that it allows the developer to set the listeners which is not possible in the IPatternManager. On the contrary, the IPatternManager interface provides the same functionality with additional pattern chain pipe but the matched event and unmatched events are to be pulled by the developer if not listeners are added to the list of pattern listeners.
@see The package org.streameps.test.HighestPatternTest for sample code in the project page.

5. Event Filtering

The core of event filtering in StreamEPS is made up of these three components namely the filter expression, the value set and the filter processor. The filter expression component is provided by the following interface classes IEventContentFilterExprn, IEventHeaderFilterExprn and IEventTypeFilterExprn with each extending an abstract core interface IPredicateExpr defined in the package org.streameps.filter. There are three value sets which implements the interface IValueSet to provide the unique identifier. The comparison value set provided by the interface IComparisonValueSet consists of a partition window of events sorted in an accumulator. The range value set with has an interface class defined by the IRangeValueSet with similar structure content as the comparison value set. The third value set is the in-not value set with its interface class InNotValueSet. The architectural decision is to place each value set into its bounded-context latter explained. Lets move on to the filter processing of events in the platform, StreamEPS. A filter in StreamEPS has a common *filter* method for the filtering process of the context value set provided. The filter process is provided by the interface IEPSFilter which has an abstract implementation called AbstractEPSFilter. It provides the developer an opportunity to implement custom filters with ease. It is recommend for the developer to extend the abstract filter class, AbstractEPSFilter instead of the IEPSFilter interface. The *filter* method takes a special context value set parameter called the IExprEvaluatorContext. The context value set has a filter type for determining the abstract filter type to use, a filter operator that is used in the evaluation of events , the context entry which provides the predicate expression and terms and most importantly, the parametrized event container which is defaulted to the sorted accumulator provided by the interface ISortedAccumulator. The filter type is an enumeration of string group values ; comparison, range and in-not values and the filter operator is made up of similar enumerated string values that includes the comparison group; *less than*(lt), *greater than*(gt), *equal*(eq), *greater than or equal* (geq) and on and on, the range group; *range close*([a,b]), *range open*((a,b)), *range half open* and *half closed*, *in list values* and finally the in-not value group that provides *not range*(half-open, half-closed, close, open).

5.1. Extending the filter processor

In order to implement new filters for the platform, the developer has to look at these

- The filter expression component to use.
- The specific value set as the context holder with its event container which the platform has provided more. A referent to the context holder draws the developer's attention to the IExprEvaluatorContext.
- Extending the AbstractEPSFilter for your custom filter.

5.2. Writing an event filter

The entry point to compositing a functional filter context is provided by the filter manager with its provided interface called IFilterManager. In constructing a filter for the filter manager, the developer can choose to use the FilterContextBuilder in the builder package to build it standing rules as shown below. Let step through this example.

```
import org.streameps.engine.builder.FilterContextBuilder;
import org.streameps.filter.FilterType;
import org.streameps.filter.IEPSFilter;
import org.streameps.filter.eval.ComparisonContentEval;
import org.streameps.context.PredicateOperator;

FilterContextBuilder filterContextBuilder = new FilterContextBuilder();  1
      filterContextBuilder.buildPredicateTerm("value",
PredicateOperator.GREATER_THAN_OR_EQUAL, 18)
            .buildContextEntry("TestEvent", new ComparisonContentEval())
            .buildEvaluatorContext(FilterType.COMPARISON).buildEPSFilter()
            .buildFilterListener(new TestFilterObserver())                2
            .buildFilterContext();
IEPSFilter filter = filterContextBuilder.getFilter();          3

IFilterManager manager =new IFilterManager();          4
manager.processFilter(filter);
```

Source 8: An event filter template with the filter context builder

1. An instance, *filterContextBuilder* of the filter context builder is created with the class FilterContextBuilder.
2. The instance builder is then used to build the filter context and importantly the filter object itself. A predicate term with its property name set to “value”, it operator to greater than and it property value set to 18. A comparison content expression defined by the class ComparisonContextEval is used with the context entry name set to “TestEvent” to build a new instance of the IContextEntry interface. The filter listener used to received the filter value set is then also built with a test listener called TestFilterObserver shown in the source listing 9 below.
3. Finally on the build process, the filter context is built and then the filter object itself retrieved for the filter manager processing.
4. A new instance , *manager* of the filter manager is created and the created instance of the filter object is then passed to the *processFilter* method which processes the filter context

provided. The matched filter events and unmatched are then sent to the observers. Much of concern is the matched filter events which is forwarded to the instance of the registered test observer in the list of filter listeners.

The listing below depicts an implementation of the filter event observer called TestFilterObserver.

The most important part of the source code for this illustration is the *handleFilteredEvent* which is received from the filter manager, an instance of the filter value set from the filter matching process. In this example, only the value identifier of each filter value set is retrieved to emphasis the point in the guide.

```
public class TestFilterObserver implements IFilteredEventObserver{

   public void addFilteredEventObserver(IFilteredEventObserver eventObserver) ...
   public void removeFilteredEventObserver(IFilteredEventObserver  eventObserver) ...
   public void notifyAllObservers() ...
   public void update(Observable o, Object arg) ...

   public void handleFilteredEvent(IFilterValueSet filterValueSet) {
      System.out.println("Filter Observer:"+filterValueSet.getValueIdentifier());
   }
```

Source 9: A test filter event observer with the handleFilteredEvent method

6. StreamEPS Core Standalone

In using StreamEPS as a standalone software will require just the eps-core-0.5.8.jar on your classpath or the maven dependency configuration as shown below:

<groupId>org.streameps</groupId>

<artifactId>eps-core</artifactId>

<version>0.5.8</version>

The standalone software with StreamEPS is built around the builder pattern and this makes it use to be easy.

Step 1: Once the configuration is done, initialise the four interfaces by choosing the specific type of engine to used in the software. The engines are under the package name: org.streameps.engine. These snippets of source code in the example is using the segment engine. The full details of the class, *EngineTest* is in the test package of the *core* or *client* sub-projects.

```
IEPSDecider<IContextPartition<ISegmentContext>> decider = new SegmentDecider();
IEPSReceiver<IContextPartition<ISegmentContext>, TestEvent> receiver = new SegmentReceiver();
IEPSEngine<IContextPartition<ISegmentContext>, TestEvent> engine = new SegmentEngine();
IEPSProducer producer = new EPSProducer();
```

1

Step 2: The engine builder can then be constructed with the decider, engine and receiver. The producer is also set to the engine builder.

```
EngineBuilder engineBuilder;
    {
      //2: set the engine, decider and receiver properties.
      engineBuilder = new EngineBuilder(decider, engine, receiver);
      engineBuilder.setProducer(producer);
    }
```

2

Step 3: The aggregate context builder is used to define both a decider aggregate context and a producer aggregate context. The aggregate context basically will define the event attribute, the aggregate function, assertion value and the assertion operator type. In this example, builder is preparing a decider aggregate context with its event attribute: value, function: sum, assertion value: 20 and the assertion operator: greater than. The aggregate context is also set along with the

aggregate listener defined with the interface: org.streameps.processor.AggregatorListener by setting the aggregate enabled flag. You may choose not build any aggregate context for a particular implementation.

```
//decider aggregate context
    AggregateContextBuilder aggregatebuilder = new AggregateContextBuilder();
    aggregatebuilder.buildDeciderAggregateContext("value", new SumAggregation(), 20,
AssertionType.GREATER);
    // engineBuilder.setAggregatedDetectEnabled(aggregatebuilder.getAggregateContext(), new
TestAggregateListener());
    //producer aggregate context
    aggregatebuilder.buildProducerAggregateContext("value", new SumAggregation());
    engineBuilder.setAggregatedEnabled(aggregatebuilder.getAggregateContext(), new
TestAggregateListener(), true);
```

3

Step 4: During the normal running of the standalone, you want to keep the event received by the engine to a permanent store for audit/security purposes. The store context builder class provided in the *org.streameps.engine.builder* is used to construct such functionality. An instance of the store property is created with the store location. After you create an instance of the historyStore with the store property instance and the provided executor manager.

```
String location = "C:/store";

    StoreContextBuilder storeContextBuilder = new StoreContextBuilder();

    IStoreProperty isp = new StoreProperty("comp", IEPSFileSystem.DEFAULT_SYSTEM_ID, location);
    IHistoryStore historyStore = new HistoryStore(StoreType.FILE,
        new AuditEventStore(isp, engine.getExecutorManager()));
    storeContextBuilder.addStoreProperty(isp).addHistoryStore(historyStore);
    engineBuilder.buildAuditStore(storeContextBuilder.getHistoryStore());
```

4

Step 5: If you are interested in listening to pattern decisions during the engine processing, then implement the interface org.streameps.decider.IDeciderContextListener. The most important aspect of the engine include determining the event count/sequence size before sending out the event set to the engine pipeline, whether queue or not, whether asynchronous or not, whether to save on receive or not and whether to save at decision point. Secondly, the size of core pool and the thread name for the executor manager. The dispatcher used in dispatching the event will require to know the size of the dispatch queue, the initial delay for the new dispatch, the periodic delay which sets the delay for the next dispatch process and the time unit to used in the dispatch process.

The pattern builder is used to create pattern matching processes in the engine. There are several patterns as described in Chapter 4 and the developer is left to the implementation of the listeners, and of course the pattern matching process if the ones provided does not support your needs.

```
//producer decider listener
    engineBuilder.setDeciderListener(new TestDeciderListener());

    //set the properties: sequence size, asynchronous flag, queue flag, saveonReceive flag, saveonDecide flag.
    engineBuilder.buildProperties(20, false, true, true, true);
    engineBuilder.buildExecutorManagerProperties(2, "EPS");
    engineBuilder.buildDispatcher(3, 0, 1, TimeUnit.MILLISECONDS, new DispatcherService());

    //3: set up a pattern detector for the decider.
    PatternBuilder patternBuilder = new PatternBuilder(new NullPatternPE())
//patternBuilder.buildParameter("value", 16);/*No comparison operator needed.*/
            .buildPatternMatchListener(new TestPatternMatchListener())
            .buildPatternUnMatchListener(new TestUnPatternMatchListener());

    //add the pattern 1 detector built to the engine/decider.
    engineBuilder.buildPattern(patternBuilder.getBasePattern());
patternBuilder = new PatternBuilder(new TrendPatternPE(new IncreasingAssertion()));
    patternBuilder.buildParameter("value");

    //pattern 3: repeated pattern detector process.
    patternBuilder = new PatternBuilder(new HighestSubsetPE<TestEvent>());
    patternBuilder.buildParameter("value", 12);
```

5

Step 6: After all the pattern detector/matching building process, you have to create the receiver context using its builder provided by ReceiverContextBuilder. In this example, it first creates a unique identifier with the IDUtil, then builds the context detail with the dimension being SEGMENT_ORIENTED. Further it builds the segment parameter attribute used to get assertion value which is name in the example and finally the context parameter by retrieving the segment parameter from the chain process. The receiver instance created in step 1 is set with the segment context from the receiver builder chain build process.

```
ReceiverContextBuilder contextBuilder = new ReceiverContextBuilder(new SegmentParam());


      contextBuilder.buildIdentifier(IDUtil.getUniqueID(new
Date().toString())).buildContextDetail(IDUtil.getUniqueID(new Date().toString()),
ContextDimType.SEGMENT_ORIENTED) // .buildSegmentParameter(new ComparisonContentEval(),
            // new PredicateTerm("value", PredicateOperator.GREATER_THAN_OR_EQUAL, 18))
            .buildSegmentParamAttribute("name") //.buildSegmentParamAttribute("name")
            .buildContextParameter("Test Event", contextBuilder.getSegmentParam());


      receiver.setReceiverContext(contextBuilder.getContext());
```

6

Step 7: The receiver context building process is done, lets move on to the filter context builder process. The filter context builder also uses the same chain process for the filter context creation. This snippet depicts a filter context with first, a predicate term of structure : <property-name, assertion-operator, assertion-value> corresponding to <value, GREATER_THAN_OR_EQUAL, 18>, then builds the context entry of ComparisonContextEval expression evaluator and the comparison ExprEvaluatorContext. Further, the above process calls is used to create a filter and then a user-defined filter listener/observer instance is set to build the filter listener. Finally, the filter context is built and then set to the producer.

```
FilterContextBuilder filterContextBuilder = new FilterContextBuilder();
      filterContextBuilder
         .buildPredicateTerm("value", PredicateOperator.GREATER_THAN_OR_EQUAL,18)
         .buildContextEntry("TestEvent", new ComparisonContentEval())
         .buildEvaluatorContext(FilterType.COMPARISON)
         .buildEPSFilter()
         .buildFilterListener(new TestFilterObserver())
         .buildFilterContext();
      IEPSFilter filter = filterContextBuilder.getFilter();
producer.setFilterContext(filterContextBuilder.getFilterContext());
```

7

Step 8: The runtime client called IEPSRuntimeClient is created to hold all the instances of the builders which includes the aggregate builder, filter context builder, pattern builder, store context builder and the receiver context builder. It provides two important functionalities which includes the ability to start and restart the engine after any changes to any builder instance of the four components; decider, engine, producer, receiver.

```
IEPSRuntimeClient epsRuntimeClient = new EPSRuntimeClient(engineBuilder,
        aggregatebuilder,
        filterContextBuilder,
        patternBuilder,
        storeContextBuilder,
        contextBuilder);
    epsRuntimeClient.restartEngine();
    engine = epsRuntimeClient.getEngine();
```

8

Step 9: In conclusion, the runtime client is has restarted the engine and retrieved that instance from the runtime client. Use the restart-engine instance to send some events to the engine pipeline for the aggregation, filtering, projection, collation processes. Have a seat and see the your dashboard popping out event if there matches whether pattern or filter and more.

```
Random rand = new Random(50);
  for (int i = 0; i < 20; i++) {
     TestEvent event = new TestEvent("E" + i, ((double) rand.nextDouble()) + 29 - (2 * i));
     engine.sendEvent(event, true);
  }
```

9

Summary

This guide has provided the developer a detailed introduction to the StreamEPS platform. This platform provides an engine for processing segment-oriented, temporal-oriented, state-oriented and spatial-oriented event stream from a stream source defined by the event producer of the client in the processing pipeline. This document did look at the event assertions, event aggregation, event pattern matching and event filtering process. Importantly, a developer walk-through of sample source code for the illustration of the functional components provided in this platform. An information on the essential points in the design is exposed to the developer to implement custom implementations whether aggregation functions, pattern matching or event filters.
In future documents , the author will provide more detail explanation of the design and development processes taken to reach this implementation, more specifically the context-driven design approach to software development.