UML Profile for Dataflow Applications - User Manual

This documents describes a UML Profile which can be used to enriching standard UML Class Diagrams with the required semantics to model Dataflow applications. The profile offers a number o UML stereotypes to be applied on UML elements.

Background on Dataflow Applications

A Dataflow application can be represented as a direct acyclic graph (DAG) in which nodes represents either data sources, data transformations or data sinks, while edges represent the flows (or streams) of data. Typically a Dataflow application begins with one or more data sources that fed the applications with the input data streams. Transformations are then applied on such input data streams to eventually produce some output data streams that are then typically written to some external data sink. Data sources, transformations and sinks can be of different types. For instance a data sources can be text files, CSV files, databases, etc. Transformations can be averages, counts of distinct elements, sums, etc. Actually transformations are often user defined, meaning that is the user herself that is responsible to define how she want to transform a data stream into another data streams. For instance a transformation might just enrich each element of a data stream with some metadata, or it might count the number of occurrences of distinct elements of a data stream, update some machine learning model, etc.

Using the UML Profile for Dataflow Applications

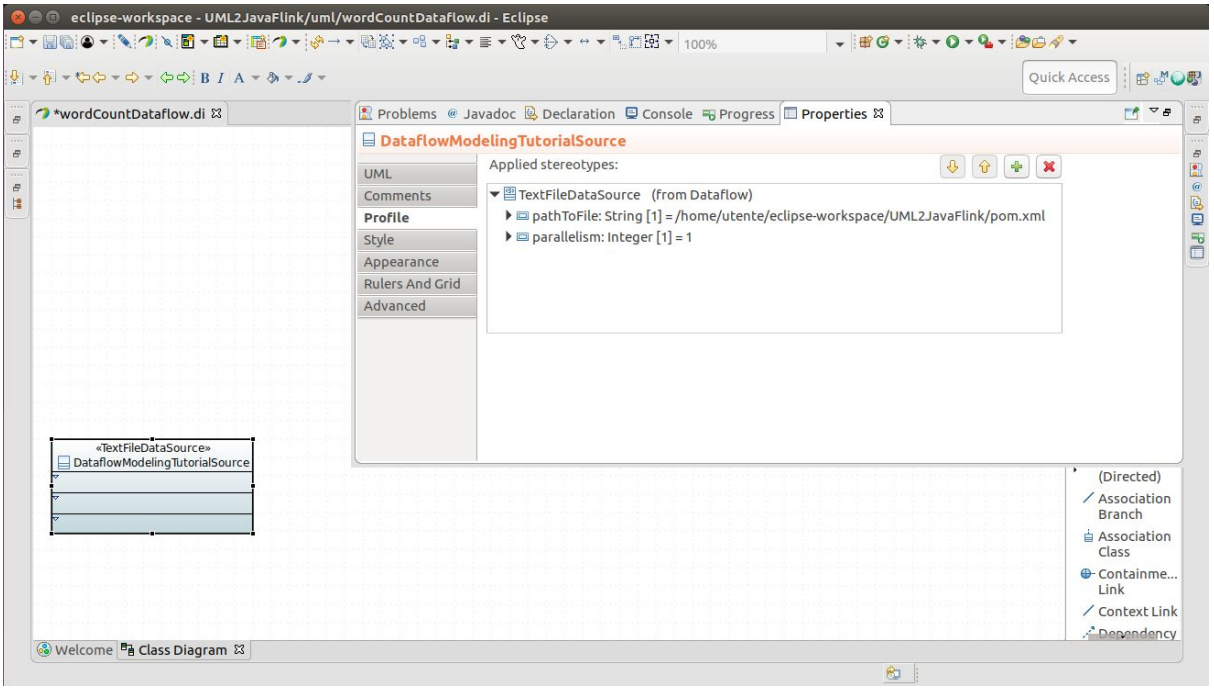
Our UML profile for Dataflow applications is intended to be applied on UML Class diagrams. It provides stereotypes for modeling different types of data sources, sinks and transformations, as well as different types of data streams, depending on how do we want the processing of a stream to be performed. For instance it is quite common to process a data stream by cutting it into subsequent chunks called windows, or in parallel by partitioning the tuples flowing into the stream.

The modeling method is such that UML Classes are used to model the nodes of the DAG (sources, transformations and sinks), while UML InformationFlow links are used to model the data flowing among the various nodes. Each InformationFlow link essentially represents a data stream, which conveys a specific type of data. The profile also allows to specify which data type is conveyed on a given data stream.

In order to understand how to concretely use our UML profile for Dataflow applications we will refer to a simple dataflow application named

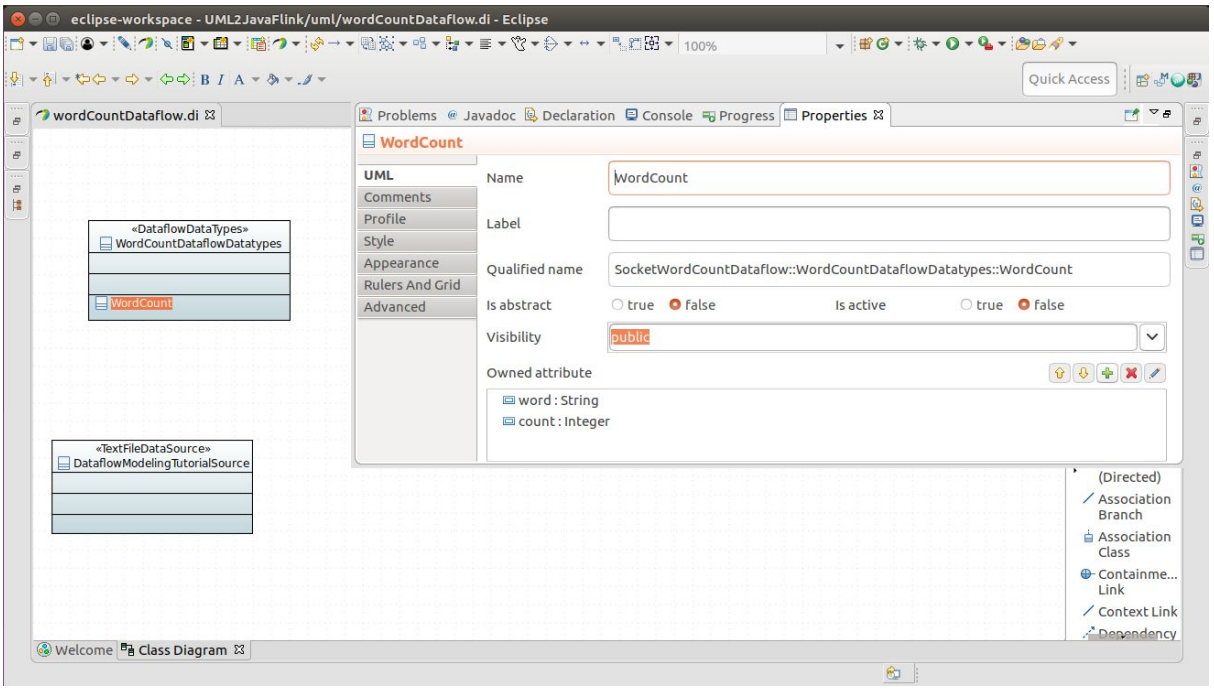
WordCountDataflow. The WordCountDataflow application takes as input a text file and counts the number of occurrences of each distinct word in the file. The applications write the result to an output file. In particular in this example we want to count word in this documents, which is assumed to be somewhere on the machine on which the dataflow application will be executed. In the following we are going to show the details of how to model the WordCountDataflow application in Eclipse using the provided ìEclipse Papyrus UML profile and how to generate the corresponding Apache Flink code using the provided Acceleo code generator.

Since our data source is a text file first of all we instantiate a Class (named DataflowModelingManualSource) and we apply on it the <<TextFileDataSource>> stereotype. This stereotype allows to specify the path to the text file to be used as input for the dataflow application.



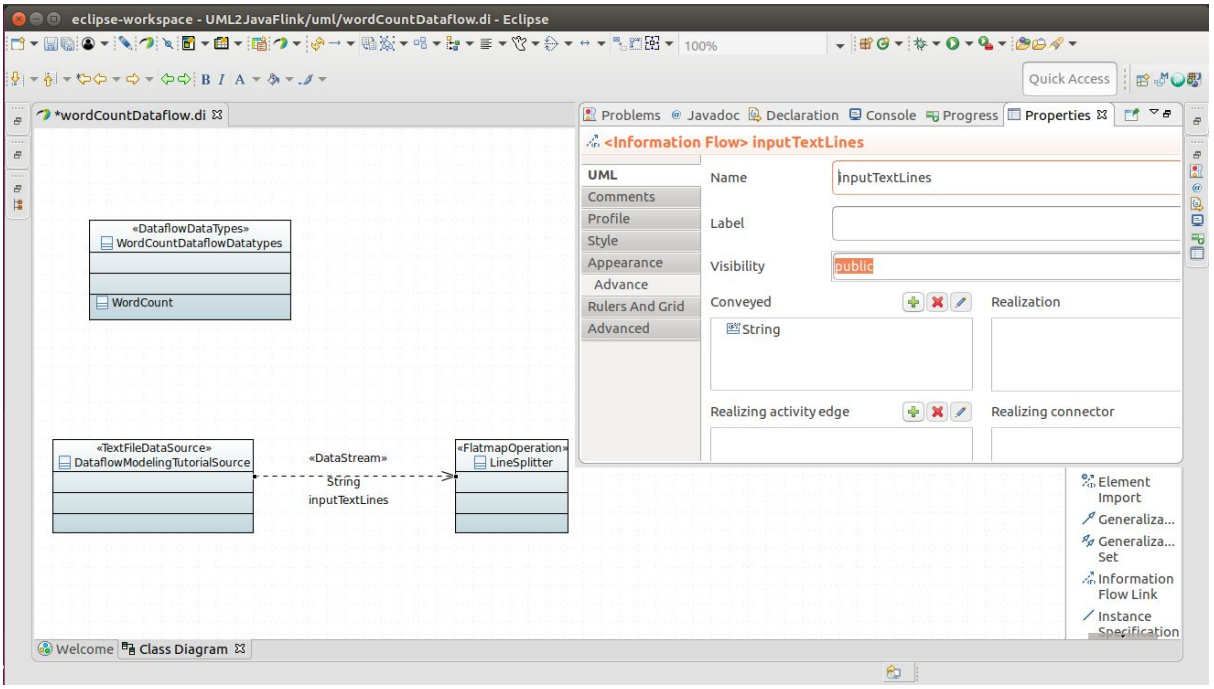
A <<TextFileDataSource>> reads a text file line by line and produces a data stream in which each tuple contains a line from the input text file. Since each tuple in such stream is just a text line, the type of data transmitted by this stream is simply “String”. Once we have such input data stream produced by a suitable data source we can start transforming it in order to reach our goal, i.e. counting the words in the source text file. One approach to do this is to split each text line into tokens representing single words, then grouping together tokens corresponding to the same word and finally counting the number of tokens in each group. Along this direction the first thing to do is to transform the stream of text lines into a stream of tokens. In particular we want to transform a line into a set of tokens of the form <word, 1>. In this case we are defining a custom data type (not primitive like integers or strings).

Our custom data type, named WordCount, simply has two fields, namely “word”, which is a String and “count”, which is an Integer. The profile allows to list all the custom data types that will be used by the application into a Class tagged with the <<DataflowDataTypes>> stereotype.

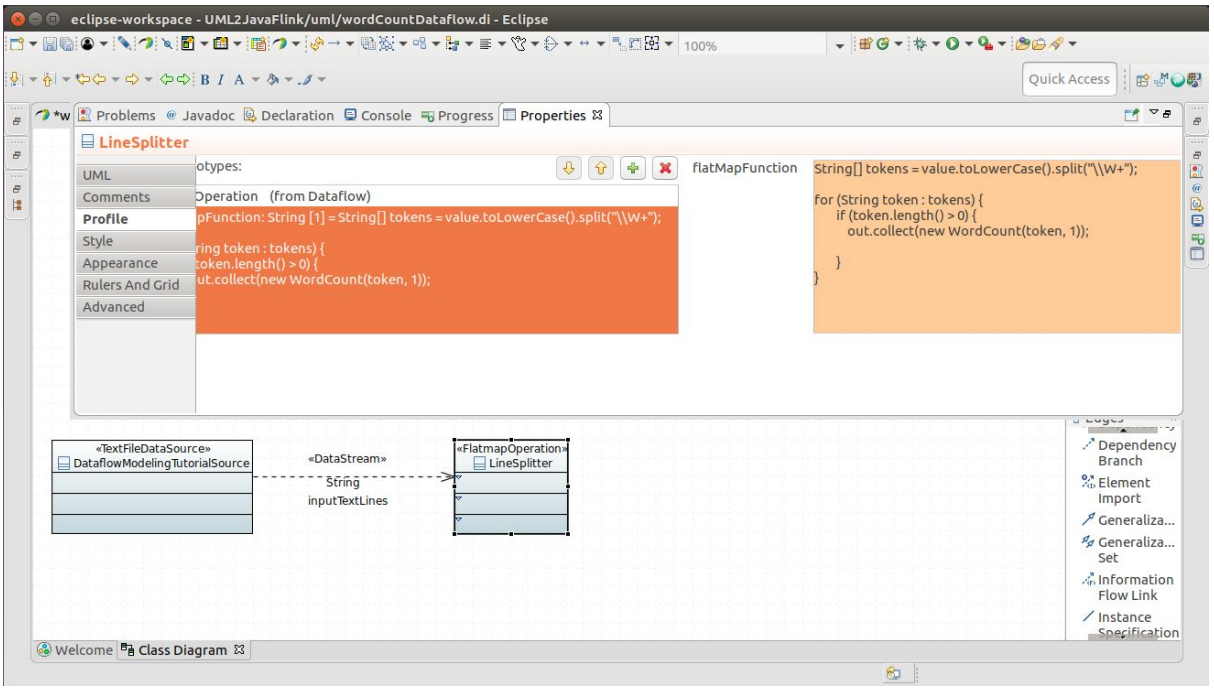


Once a custom data type has been defined in this way, it can be conveyed by any data stream (i.e. InformationFlow link tagged with the <<DataStream>> stereotype, as will be shown in a while) in the application.

Next, we want to model the fact that a specific transformation will take as input the stream of text lines produce and will produce as output a stream which conveys tuples of type WordCount.
The UML profile for dataflow applications provides a number of different transformations with different semantics. In this specific case we will use a FlatmapOperation, whose main characteristic is the following: for each incoming tuple of its input stream it can produce any number of tuples in the output stream, which is exactly what we need, since for every incoming text line we want to produce a variable number of tuples of type WordCount, one for each distinct word in the input text line. In order to model this aspect we can apply the <<FlatmapOperation>> stereotype on a UML Class and connect the data source with such class using an UML InformationFlow link tagged with the <<DataStream>> stereotype.



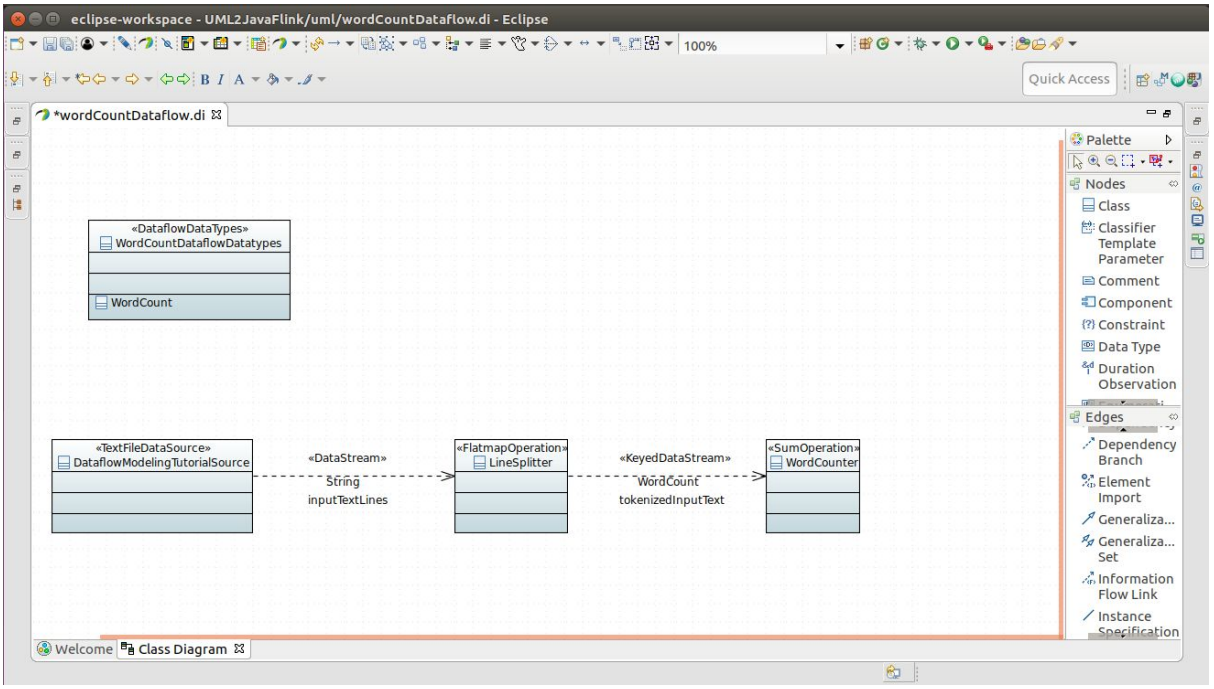
Some generic transformations, such as the FlatmapOperation, require the core transformation logic to be explicitly specified by the user. This means the user has to write a small fragment of code that specify how to process individual tuple in the input data stream to produce tuples in some output data stream. The profile allows to specify this logic using a Java syntax and a set of conventions. In the case of the <<FlatmapOperation>>, when writing the required code fragment the user can rely on a variable named “value”, which represent a generic tuple processed by the operator and that is of the same data type conveyed by the input stream of transformation (String, in our specific case) and on an variable name “out”, which can be used to collect objects to be emitted on the output stream, according to the conveyed type. In the case of the LineSplitter operator, the code fragment just has to split the “value” variable (i.e. a String coming from the “inputTextLines” data stream) into word and for each word create and collect (using the “out” collector variable) an object of type WordCount with the count attribute set to 1.



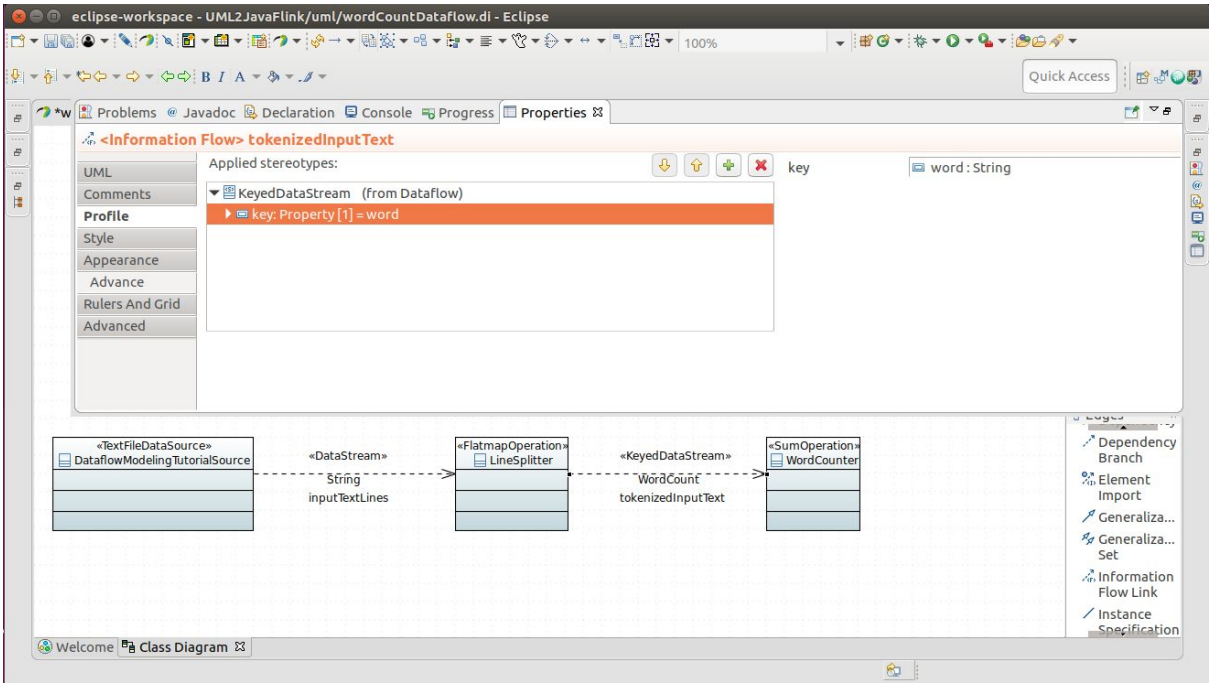
The LineSplitter operator will produce a stream of tuples of type WordCount. As previously stated, we want now to group all the tuples of such stream having the same value for the field “word” and then count the number of tuples in each group. The output of such operation have to be a new stream which convey tuples of type WordCount, but now the “count” field of each tuple reports the actual count for the corresponding word (rather than the token “1”). In order to do this we first have to :

1) split the output stream of the LineSplitter operator into a number of substreams, one for each distinct word to be counted. We say this word to be the “key” associated to a substream.
2) for each substream, add an operator that takes as input the substream itself and sums over its tuples the “count” field, obtaining the count for the “key” of the substream. The operator finally emits such count into a new stream as an object of type WordCount.

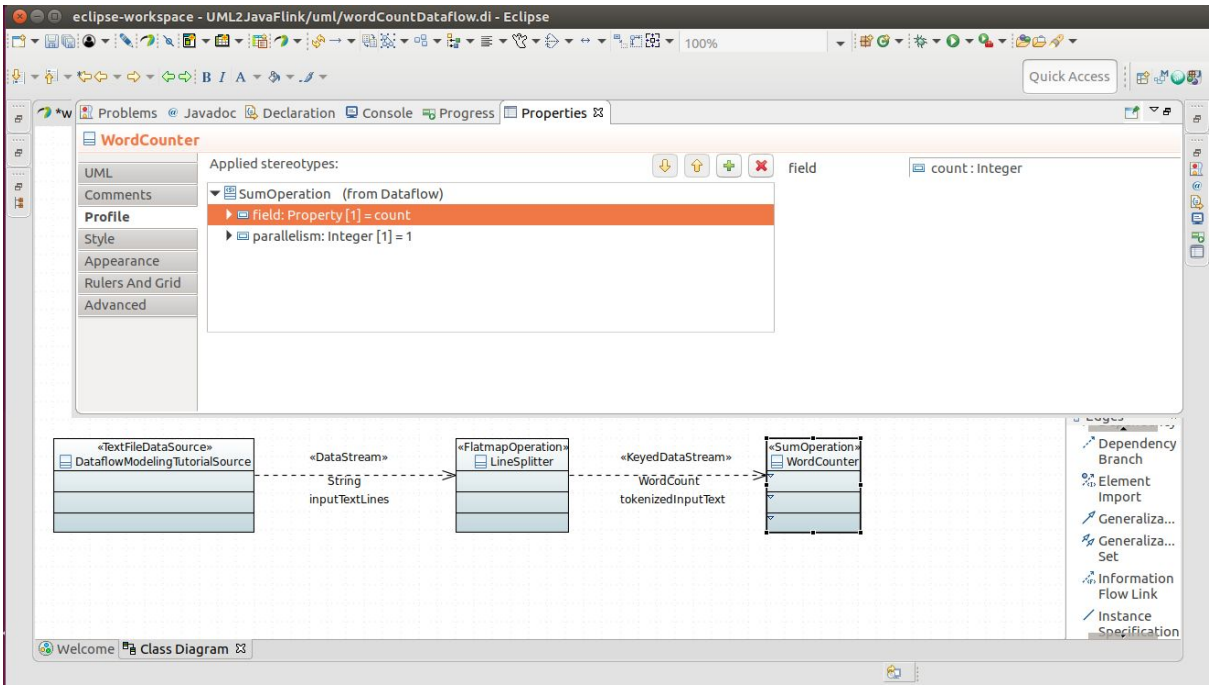
The following picture show how we can model the above aspects using the provided UML profile:



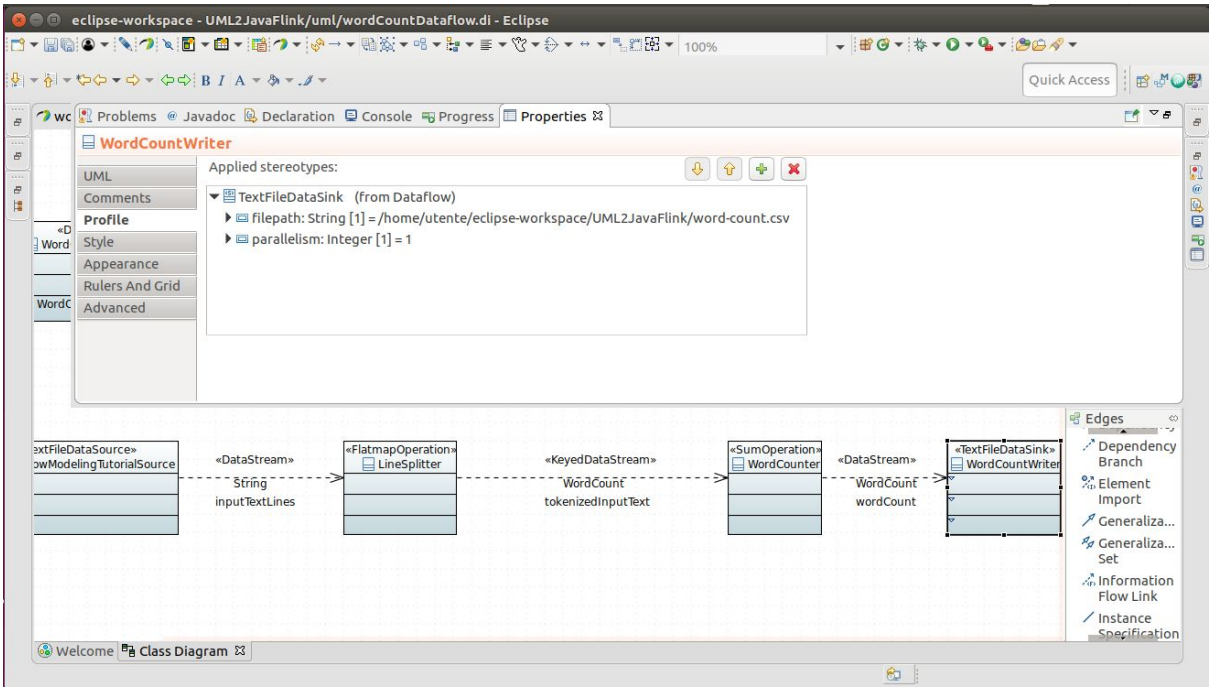
Basically when we say we want to create from the TokenizedInputText data stream a substream for each distinct word, we are saying we want to split the stream into multiple substreams based on some key. Tuple with the same value for the specified key will belong to the same substream. To model this aspect we can apply the KeyedDataStream stereotype, which allow us to specify which field on the stream is the key to be considered when splitting the stream into multiple substreams.



Such “keyed” stream becomes the input of an operator that have to count the number of tuples in each group. To do so, we can actually simply sum the “count” field over the tuples of a given substream. Along this direction, we can apply to this operator the <<SumOperator>> stereotype. Such stereotype represents a transformation with a specified semantics (i.e. summing numbers over a stream dimention), such that the user doesn’t have to specify it writing the corresponding piece of code to be executed for each incoming tuple of the input stream. The stereotype allows to specify which is the field to be summed up, in our case the “count” field of the WordCount datatype.

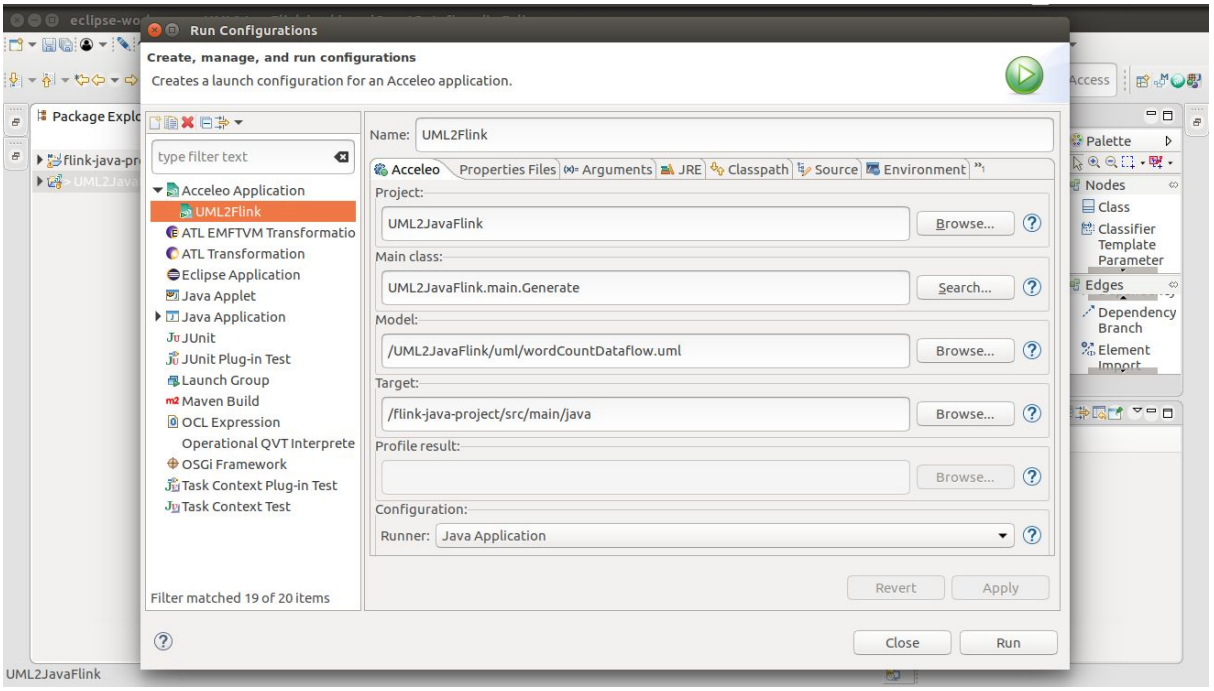


Finally, the output stream of the WordCounter operator will be exactly the intended output of the whole dataflow application, which we want to write on some external sink, in particular on a text file. In order to model this aspect we can instantiate a Class, apply the <<TextFileDataSink>> stereotype and specify the output of the LineSplitter operator to be the input of such sink operator.

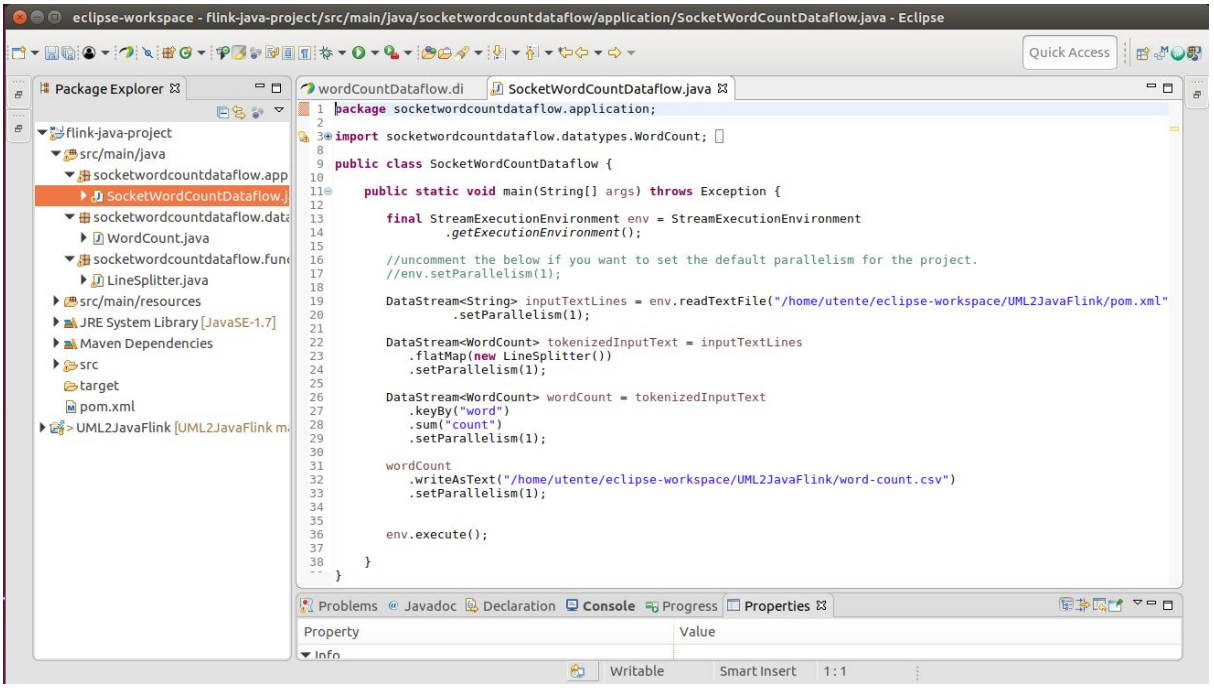


Generating the Application Code

Once we are done with the modeling activities, the designed model can be used to automatically generate the corresponding code written using the Apache Flink framework. To do so, our UML profile comes with an Acceleo module that realizes the code generation. The code generation can be launched using an Acceleo run configuration as shown in the following picture.

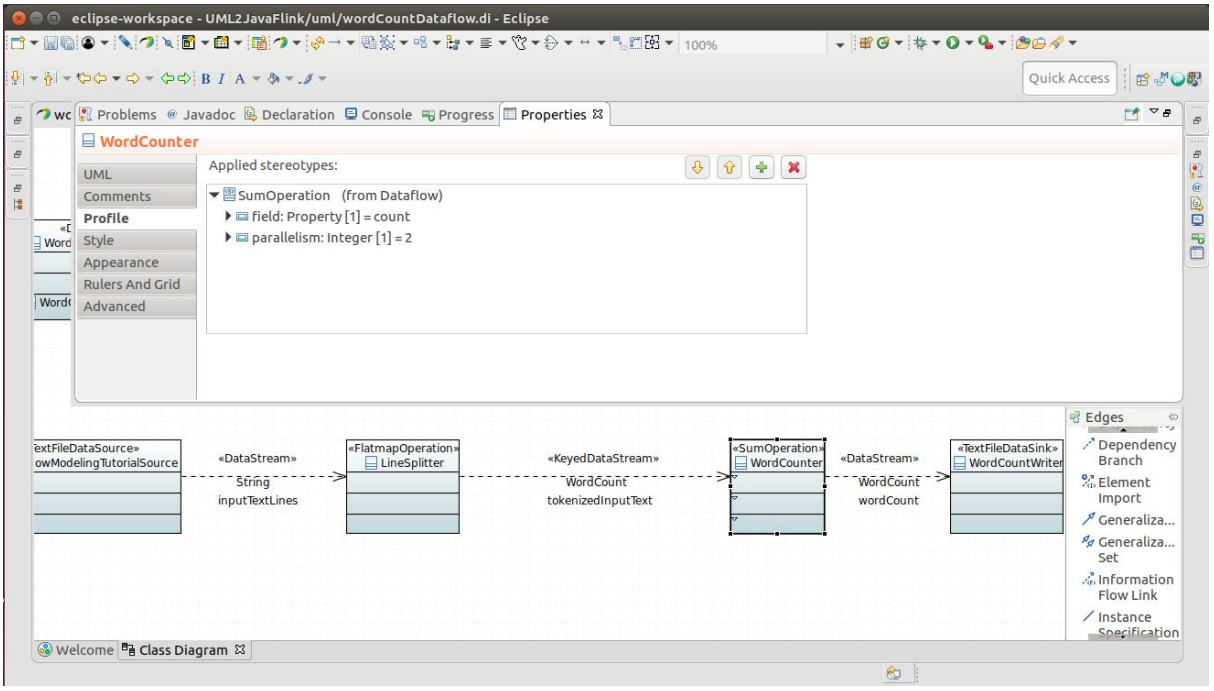


Finally we can visualize and execute the generated application code.



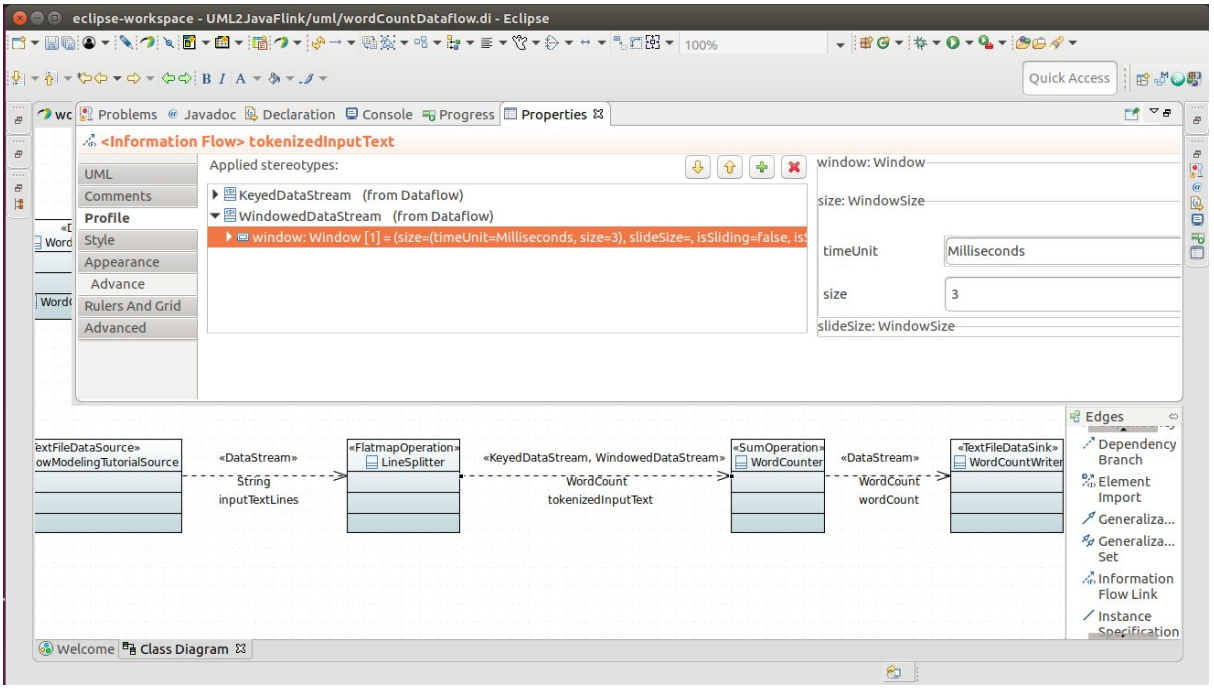
Parallel Execution

One of the main feature of dataflow applications is that they can be easily parallelized by exploiting data parallelism, i.e. partitioning data so that it can be processed in parallel by multiple instances of the application. The way data are partitioned in order to gain from a parallel execution depends on the application logic. For instance data can be partitioned randomly or based on some key, as we have seen in the example above. Let us assume, for instance, that at some point we want to parallelize the execution of the WordCounter operator, i.e. we want multiple instances of the operator to be running at the same time as distinct partitions of the “tokenizedInputText” data stream to be assigned to different instance of the WordCounter operator so that they can be processed in parallel. Since the “tokenizedInputText” is already a keyed data stream (i.e. partitioned into multiple substreams based on a key field), by using the UML profile for dataflow applications it is sufficient to change the parallelism of the WordCounter operator.



Windowing Operations

Another important feature of dataflow applications is called windowing, which is at the heart of processing infinite streams. Windows split a given data stream into “buckets” of finite size, over which we can apply computations.
https://flink.apache.org/news/2015/12/04/Introducing-windows.html


eclipse-workspace - UML2JavaFlink/uml/wordCountDataflow.di - Eclipse
Quick Access
Problems
Javadoc
Declaration
Console
Progress
Properties
<Information Flow> tokenizedInputText
UML
Comments
Profile
Style
Appearance
Advance
Rulers And Grid
Advanced
Applied stereotypes:
KeyedDataStream (from Dataflow)
WindowedDataStream (from Dataflow)
window: Window [1] = (size=(timeUnit=Milliseconds, size=3), slideSize=, isSliding=false, is
window: Window
size: WindowSize
timeUnit
Milliseconds
size
3
slideSize: WindowSize
«DataStream»
String
inputTextLines
«FlatmapOperation»
LineSplitter
«KeyedDataStream, WindowedDataStream»
WordCount
tokenizedInputText
«SumOperation»
WordCounter
«DataStream»
WordCount
wordCount
«TextFileDataSink»
WordCountWriter
Edges
Dependency Branch
Element Import
Generaliza...
Generaliza... Set
Information Flow Link
Instance Specification
Welcome
Class Diagram