|  |  |
| --- | --- |
| Handläggare  Avdelning |  |

# VTL engine quick start

# Introduction

VTL20Engine is a standalone engine for execution of code written in the VTL programming language as it is defined in version 2.0. It is written in C# with .net standard 2.0 and uses Antlr 4 runtime. In this document the implementation of the VTL engine is described from a developer’s point of view. The purpose of this text is to describe the implementation and to help developers get going in extending the VTL engine.

# Overview

The execution of VTL code can be divided into three steps:

1. The code is parsed using Antlr and an abstract search tree Is built.
2. The abstract search tree is then traversed using an implementation of the visitor-pattern. Every node in the tree is visited and a chain of operator objects is built.
3. The calculation is performed as every step in the execution chain performed. Only the nodes that is needed for the computation of the desired result are executed.

# Parsing

The parsing of the VTL code is performed using the third-party component Antlr4. Using the language definition files vtl.g4 and vtlTokens.g4 compiled by Eurostat (VTL SDMX task force), Antlr4 generates code for parsing the VTL code. This code should be regenerated when updated language definition files are released using the generateCode.bat script. The script is located together with all automatically generated and externally written code in the directory \Parser\Antlr.

The following code block shows how the parser in Antlr is called:

var inputStream = new AntlrInputStream(inputVtlCode);

var lexer = new VtlLexer(inputStream);

var tokens = new CommonTokenStream(lexer);

var parser = new VtlParser(tokens) {BuildParseTree = true};

var context = parser.start();

First, a character stream is created. A lexer is then created that identifies all the individual characters in the stream. The characters are then grouped as keywords and symbols, also called tokens. Then a parser is created which builds up a tree with relationships between all tokens.

# Calculation object

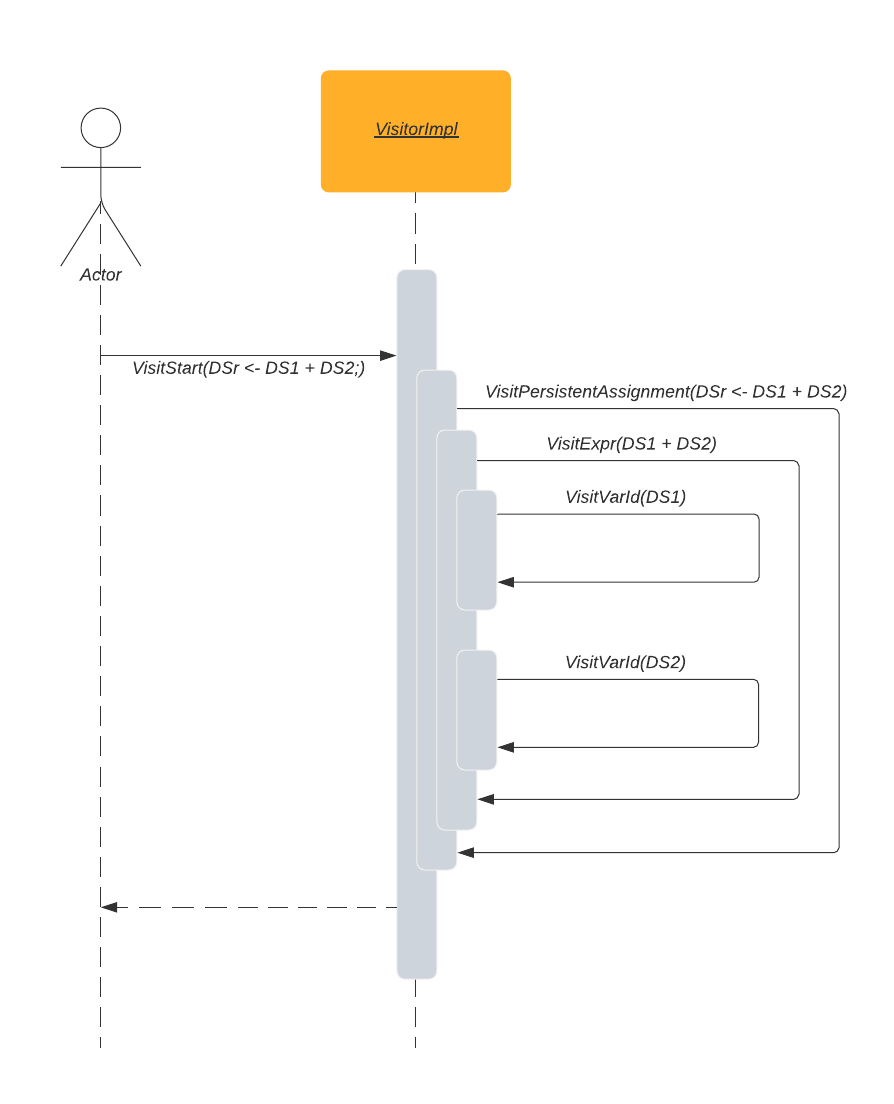
The tree built by Antlr is traversed using an implementation of the visitor pattern. Each node in the tree is visited and a calculation chain of operand objects is built. In practice, this has the same topology as the submitted tree, but it consists of objects of classes that can be executed to obtain a computation result. Below is the continuation of the code above.

var heap = inputOperands.ToList();

var visitor = new VtlVisitorImpl(heap);

visitor.VisitStart(context);

Visitor.VisitStart visits the start node with the entire parser tree as context. The VisitStart method checks which nodes are at the next level in the context tree and then visits its visitor methods with the corresponding subtree as context. The tree is traversed by calls to nested VTL commands. Finally, the leaves of the tree will initiate calls to visitor methods that retrieve constant values ​​or values ​​stored in variables on the heap. The example in the figure below basically shows how the expression DSr <- DS1 + DS2; would be processed.



## Operand

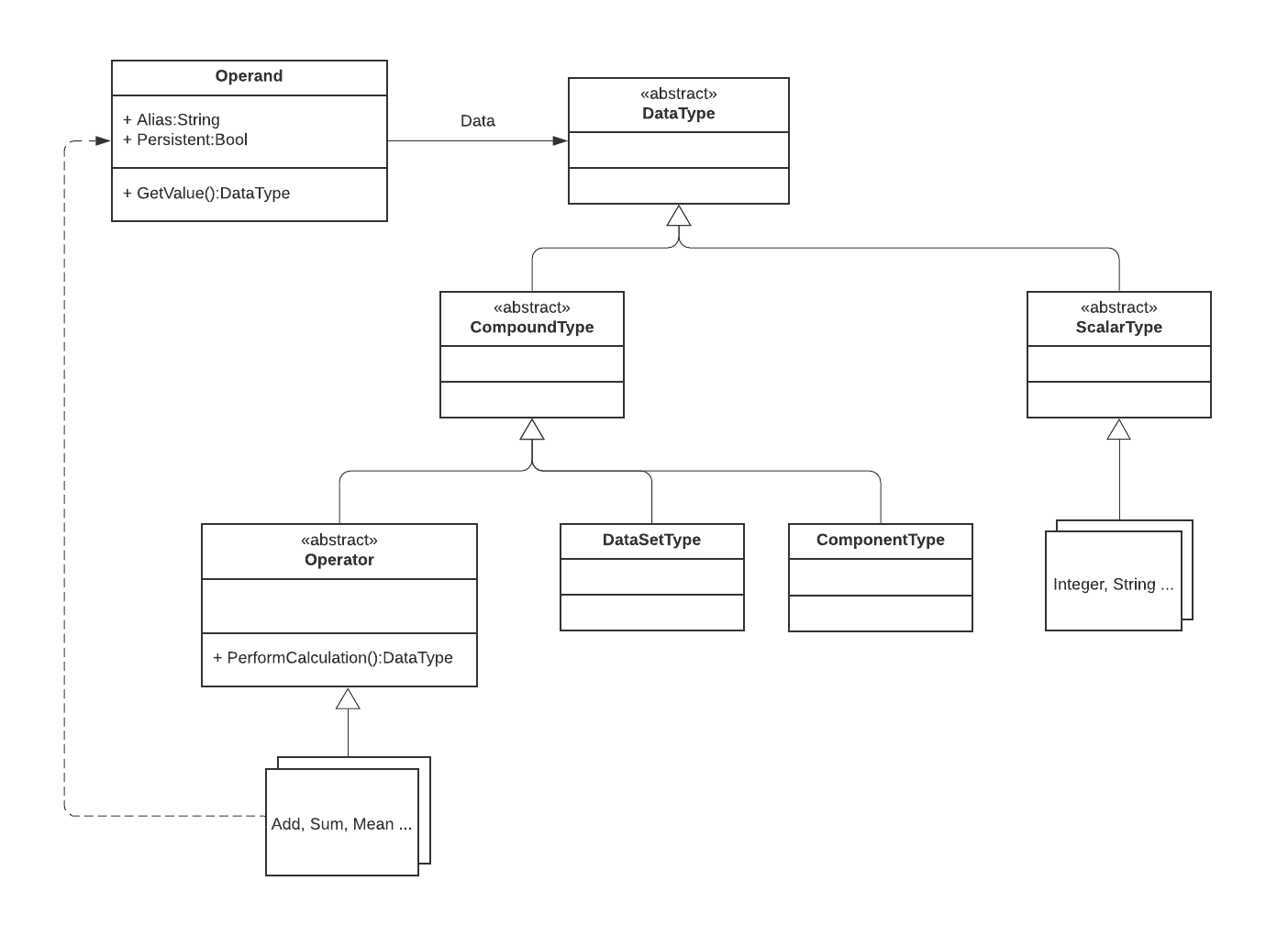
An operand is an object that can be passed as an argument to a VTL function, an operator. It encapsulates a data object and holds some metadata needed for the calculation. Operands have three important properties: Alias ​​is the name of the operand and is used to identify it. Persistent specifies whether the operand should be considered as a result of the VTL run. If it is not marked as persistent, it is not available outside the VLT engine. Finally, Operand has the Data property of the DataType data type. It can take the form of an Operator or a fixed value, either of a simple type (integer, string, etc.) or of a composite type (dataset, component, etc.).

## Operator

An operator performs a specific function, often a calculation. It always returns a data object (DataType) as a result, but depending on the function, it takes a varying number of operands and control parameters as arguments.

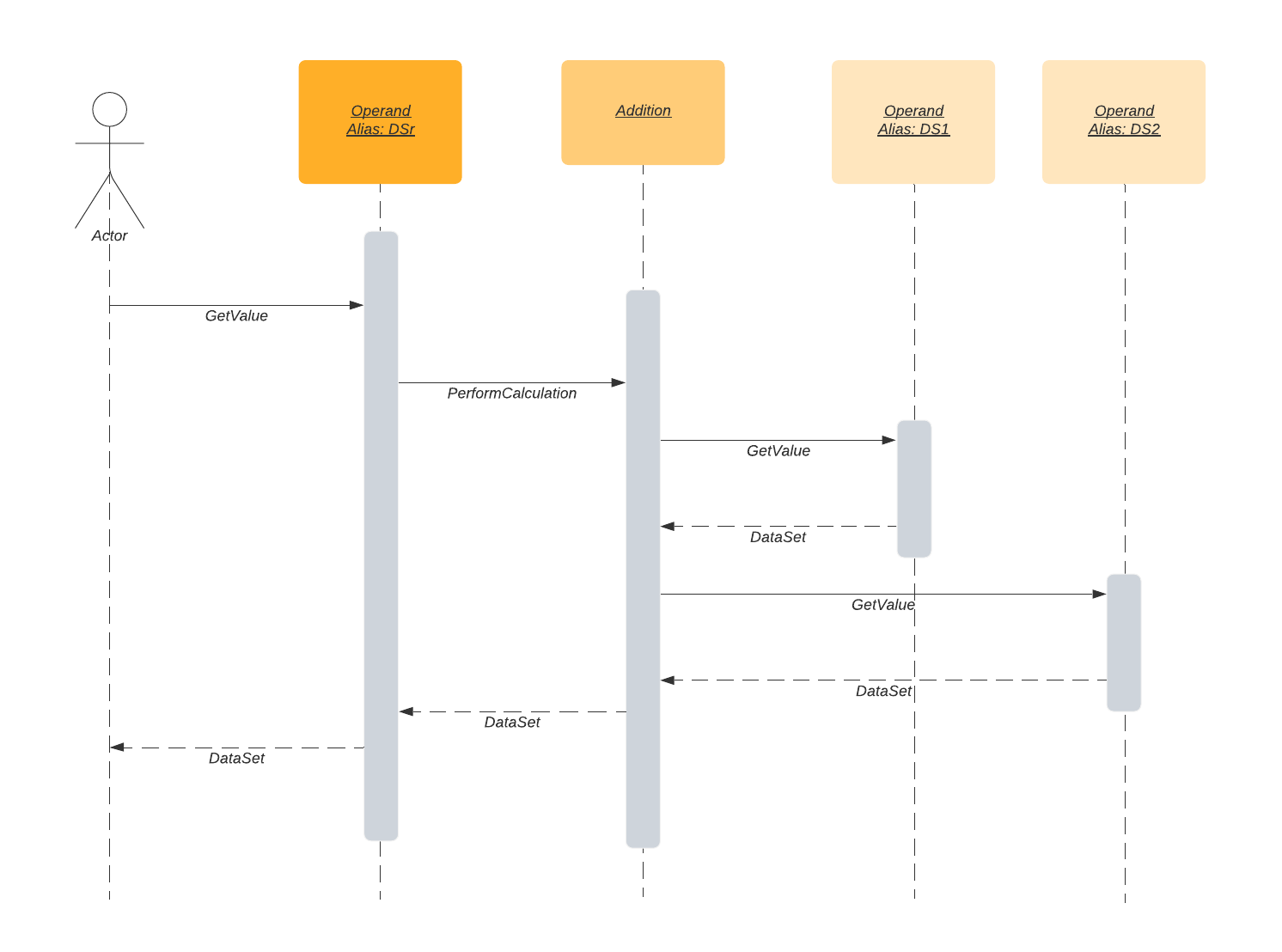
## DataType

DataType is the base class for all data handling in the VTL engine. It enables a very flexible handling of nested calls. Arguments to an operator can just as well be other operators such as simple data types or datasets. This structure is taken directly from the VTL documentation, see VTL user manual page 49. See also the figure below.



# Execution

When calling the engine, in addition to the VTL code, a set of named input parameters is also sent along. They can be of scalar type (eg numbers, text strings) or composite type (eg datasets, components). These are nested into operand objects, supplemented with aliases and put into a list called the heap. From this list, input parameters and partial results are accessed during the execution of the calculation.

The calculation is initiated by requesting the result for one of the nodes of the calculation chain. All nodes are made up of operands. The requested node requests its parameters, which in case of nested calculations are also operands. This request is made using the GetValue() method. If the data in the operand is an operator, the operator's PerformCalculation method is called, and if it is a data-bearing type, the data is simply returned. The figure below shows a summary sequence diagram for the execution of the VTL code DSr <- DS1 + DS2;

## Operators

VTL defines many operators and to keep them organized they are sorted into different categories. Each operator is implemented in a class except in some complex cases where helper classes are necessary. In the development project's directory structure, these classes are grouped in the same way as the operators are divided into chapters in the VTL document "Library of Operators". Many operators have similar behaviors, thus, to facilitate maintenance and further development as well as to avoid duplicated code, inheritance is used liberally in the implementation of the operators.