Kamil Piotr Kaczorek - QuickAccess project

Software Engineer

http://kamil.scienceontheweb.net

e-mail: kamil.piotr.kaczorek@gmail.com

INTRODUCTION

QuickAccess library is a part of my private project called "**Chrono-Graphs**".

Chrono-Graphs is a concept that will allow user to design **LFOs (Low frequency oscillators)**, **ADSR envelopes** and **Arps** for the **hardware synthesizers** in a form of a graph state machine/work-flow with time-trigger based transitions.

The system is **soft-real-time** that is why my focus is **reading-access performance**, write/modification performance is not so important.

Chrono-Graphs activities should evaluate provided by the user expressions, that is why I had to implement expression parser/compiler.

First, the parser was implemented (as a current prototype).

Then I did extensive performance tests and profiling to find bottlenecks.

Then I started implementation of an infrastructure that I need to have a fast parser, starting from **QuickAccess.DataStructures** (added to this example).

The library consists of:

- **QuickAccess.Parser** – the parser prototype
- **QuickAccess.DataStructures** – the final data structures, where the focus is on a reading access performance.

The code is in initial phase of development.

THE PROTOTYPE OF A PARSING ENGINE WITH MATH EXPRESSION COMPILER.

Project: QuickAccess.Parser

Desing/Implementation time: 20h

The parser uses **left side recursive descent parsing algorithm.**

The implemented compiler parses the grammar defined by the following rules:

```
Expression = OperatorsChainExp
OperatorsChainExp = UnaryExpression, [BinaryOperator, OperatorsChainExp]
UnaryOperatorExpression = UnaryOperator, UnaryExpression
BinaryOperator = ? one of binary operators provided by ITermDefinitionsRepository instance ?
UnaryOperator = ? one of unary operators provided by ITermDefinitionsRepository instance ?
UnaryExpression = ('(', Expression, ')') | UnaryOperatorExpression | Function | Value | Variable
Function  = FunctionName, '(', [ArgList], ')'
FunctionName = Name & ? one of defined functions in IGrammarProductsFactory ?
Variable = Name & ? one of defined variables in IGrammarProductsFactory ?
Value = ? value parsed by the IGrammarProductsFactory, which is UnsignedNumber ?
Name = Letter, [{Letter|Digit}]
ArgList = [Expression, {separator, Expression}]
UnsignedNumber = ( ({Digit},'.', [{Digit}]) | ({Digit}, ['.']) | '.', {Digit} ), [('E'|'e'), ['+'|'-'], {Digit}]
```

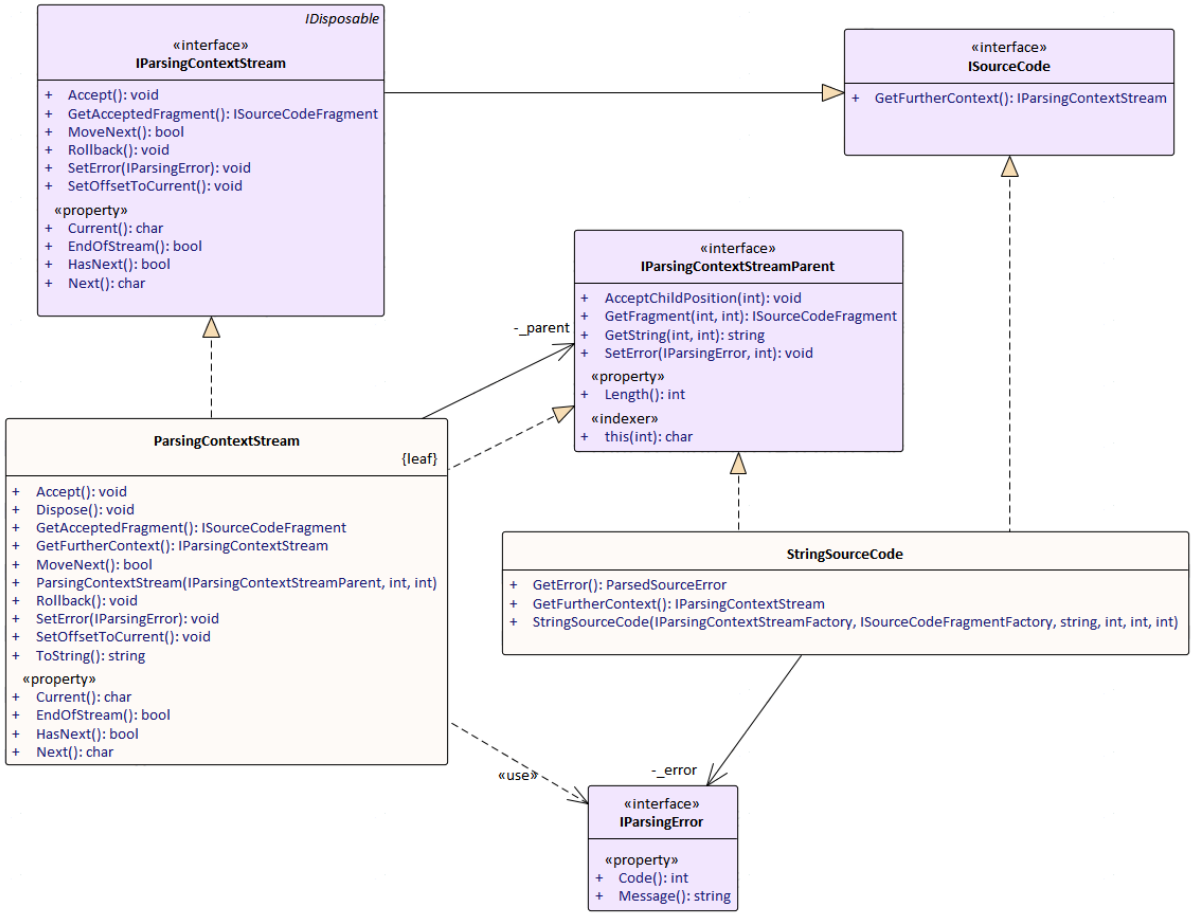
What was implemented:

- Parser
- Math expression compiler
- Demo as an integration test

What is missing:

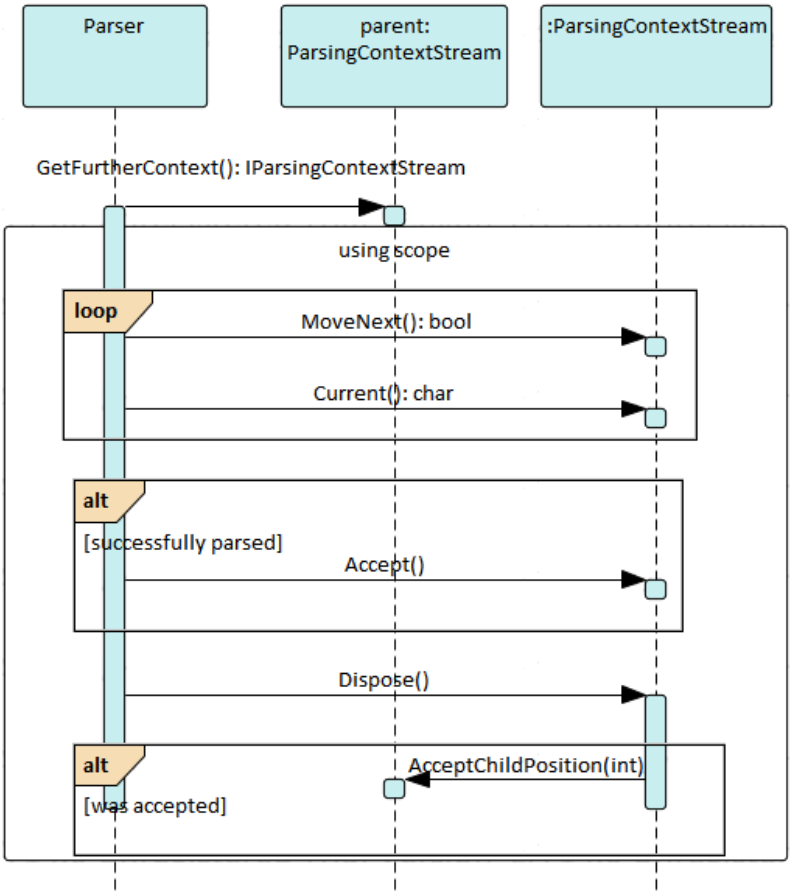
- Unit tests (because this is quick prototype - I did a lot of performance tests)

The parsing context



The above construction allows to provide the source stream for the specific parsing context.

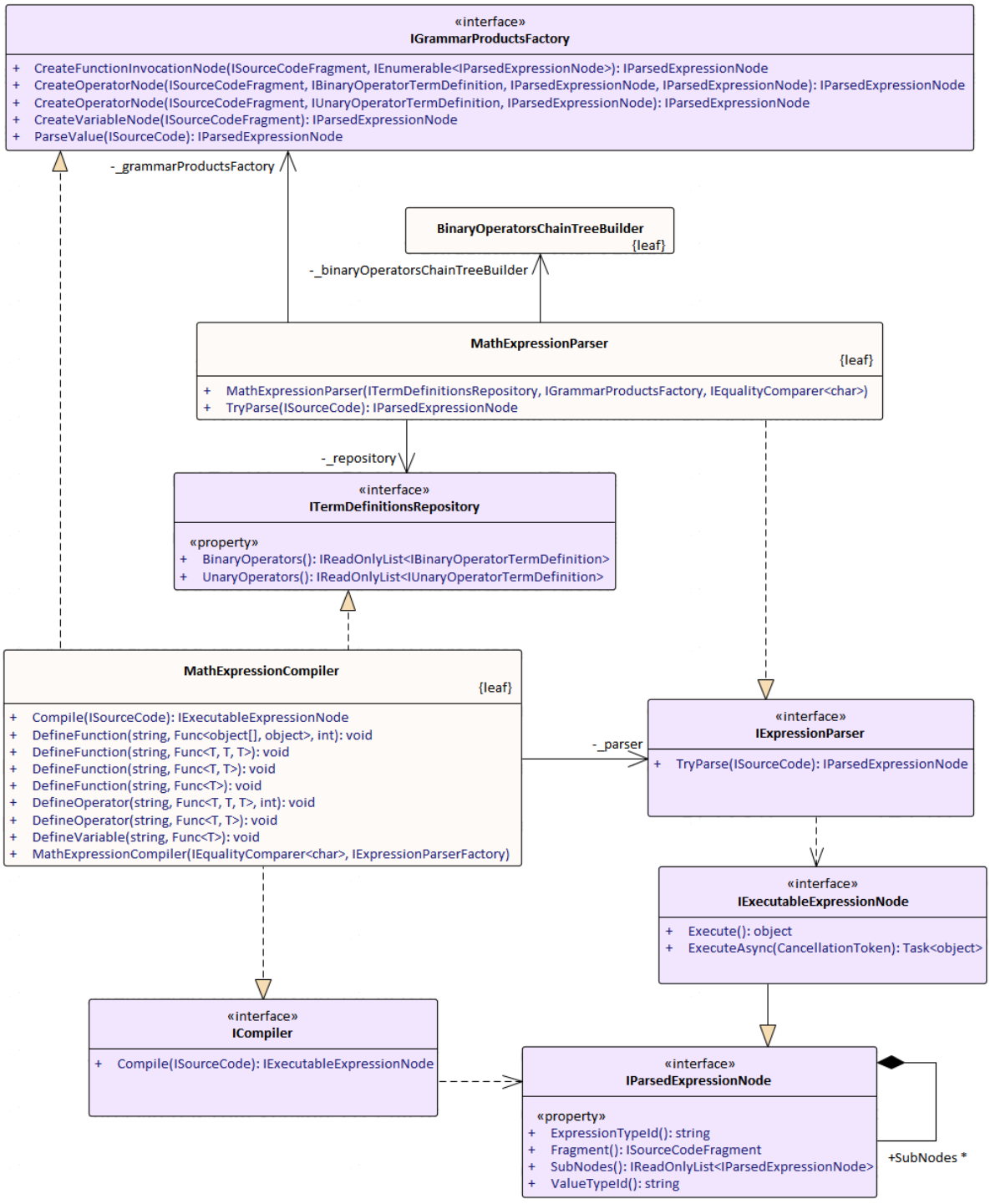
```
private IParsedFragment ParseUnsignedIntValue(IParsedSource src)
 {
     using (var ctx = src.GetFurtherContext())
     {
         if (ctx.ParseDigits() == 0)
         {
             ctx.SetError(ParsingErrors.DigitExpected);

             // on Dispose the current position will be ignored.
             return null;
         }

     ctx.Accept();

     //on Dispose the position of a parent
       //context will be advanced to the accepted position.
     return ctx.GetAcceptedFragment();
 }
```



Each parsing context stream gives the possibility to provide the child context, which enumerates the source code starting from the current parent position.

When the child context is accepted, then the child position is applied to the parent when child is disposed.

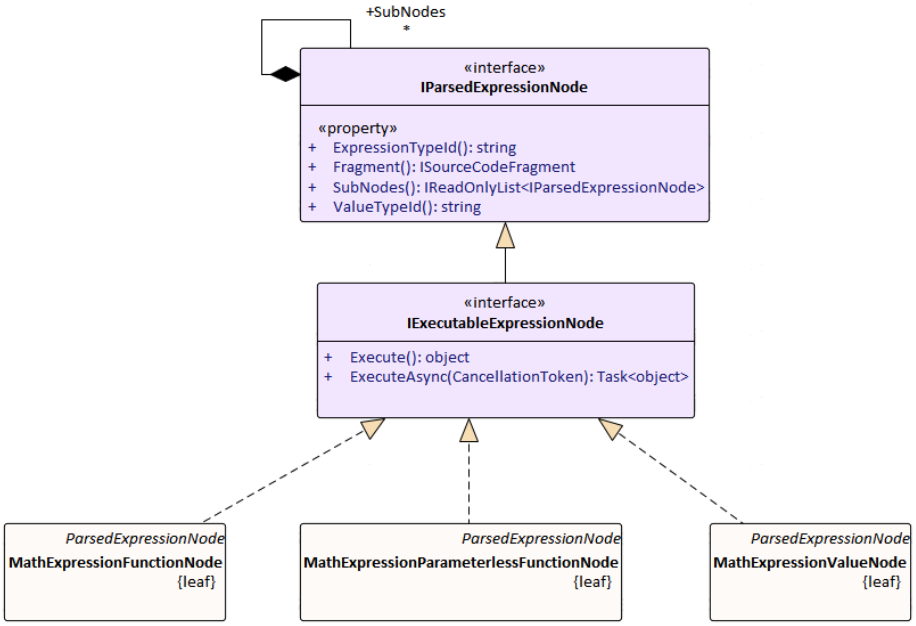
Math expression parser



MathExpressionParser parses the source code according to the grammar rules.
It builds the expression tree, respecting the operator's priorities.

Implemented compiler provides definition of final productions (operators, functions, variables, values) and builds the executable objects.

Expression node



The structure that represents the compilation result and allows to calculate compiled expression

Demo as an integration test:

```
// Arrange
var sourceCode = "sin(90*PI/180.0)*2^(1+1)*-1e-1";
var compiler = new MathExpressionCompiler(CharComparer.CaseSensitive, new
MathExpressionParserFactory());
compiler.DefineOperator<double>("+", (x, y) => x + y, 0);
compiler.DefineOperator<double>("-", (x, y) => x - y, 0);
compiler.DefineOperator<double>("*", (x, y) => x * y, 10);
compiler.DefineOperator<double>("/", (x, y) => x / y, 10);
compiler.DefineOperator<double>("^", Math.Pow, 20);
compiler.DefineOperator<double>("+", x => x);
compiler.DefineOperator<double>("-", x => -x);
compiler.DefineFunction<double>("pow", Math.Pow);
compiler.DefineFunction<double>("sin", Math.Sin);
compiler.DefineFunction<double>("cos", Math.Cos);
compiler.DefineFunction<double>("ceiling", Math.Ceiling);
compiler.DefineFunction<double>("floor", Math.Floor);
compiler.DefineFunction<double>("abs", Math.Abs);
compiler.DefineVariable("PI", () => Math.PI);

var source = new StringSourceCode(new ParsingContextStreamFactory(), new SourceCodeFragmentFactory(),
sourceCode);
// Act
var res = compiler.Compile(source);
// Assert
Assert.IsNotNull(res);
var error = source.GetError();
Assert.IsNull(error);
var calcRes = (double)res.Execute();
Assert.AreEqual(-0.4, calcRes, 0.0000001);
```

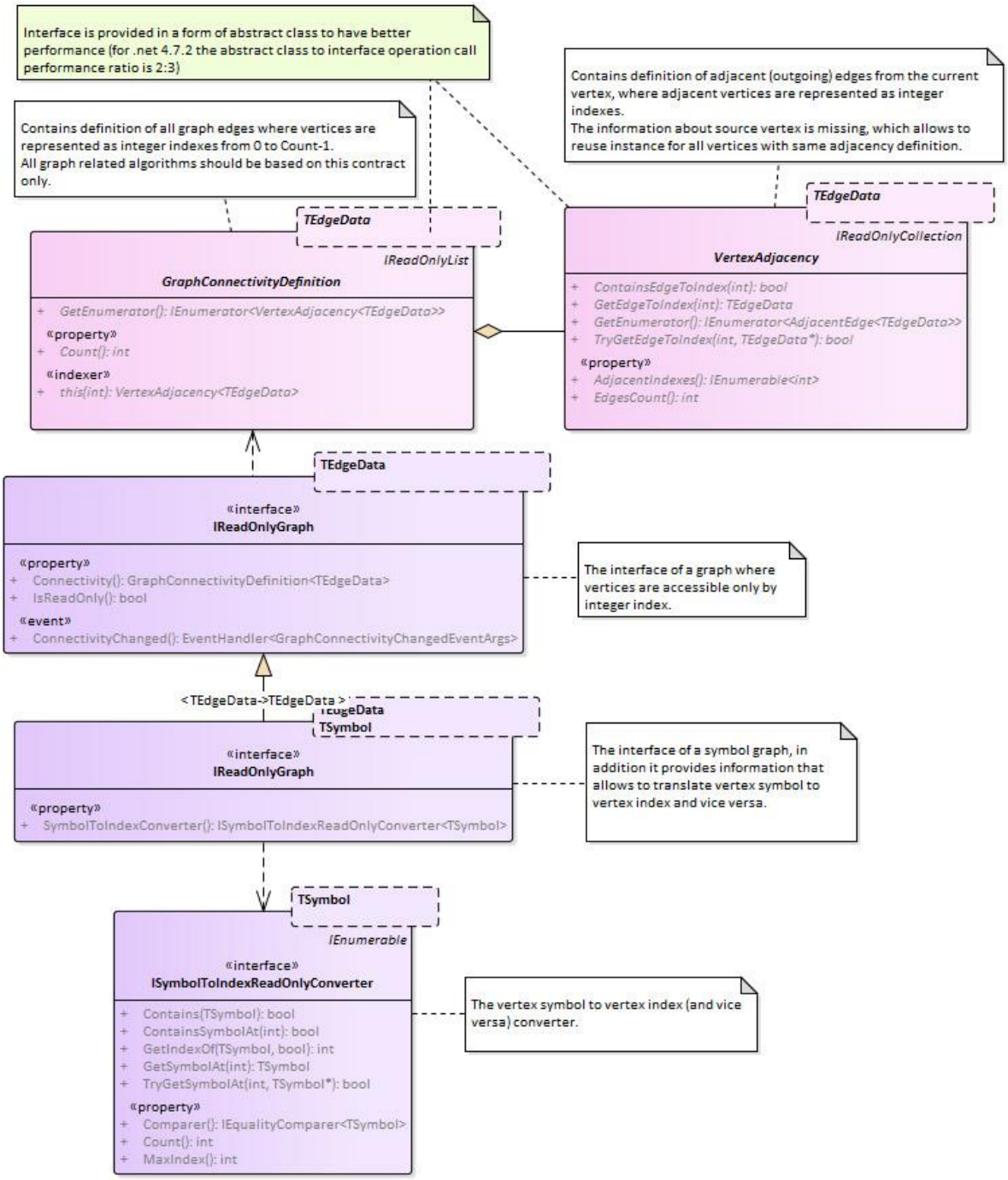
DATA STRUCTURES

Implementation time: 50h, Unit tests – work in progress.

GRAPHS

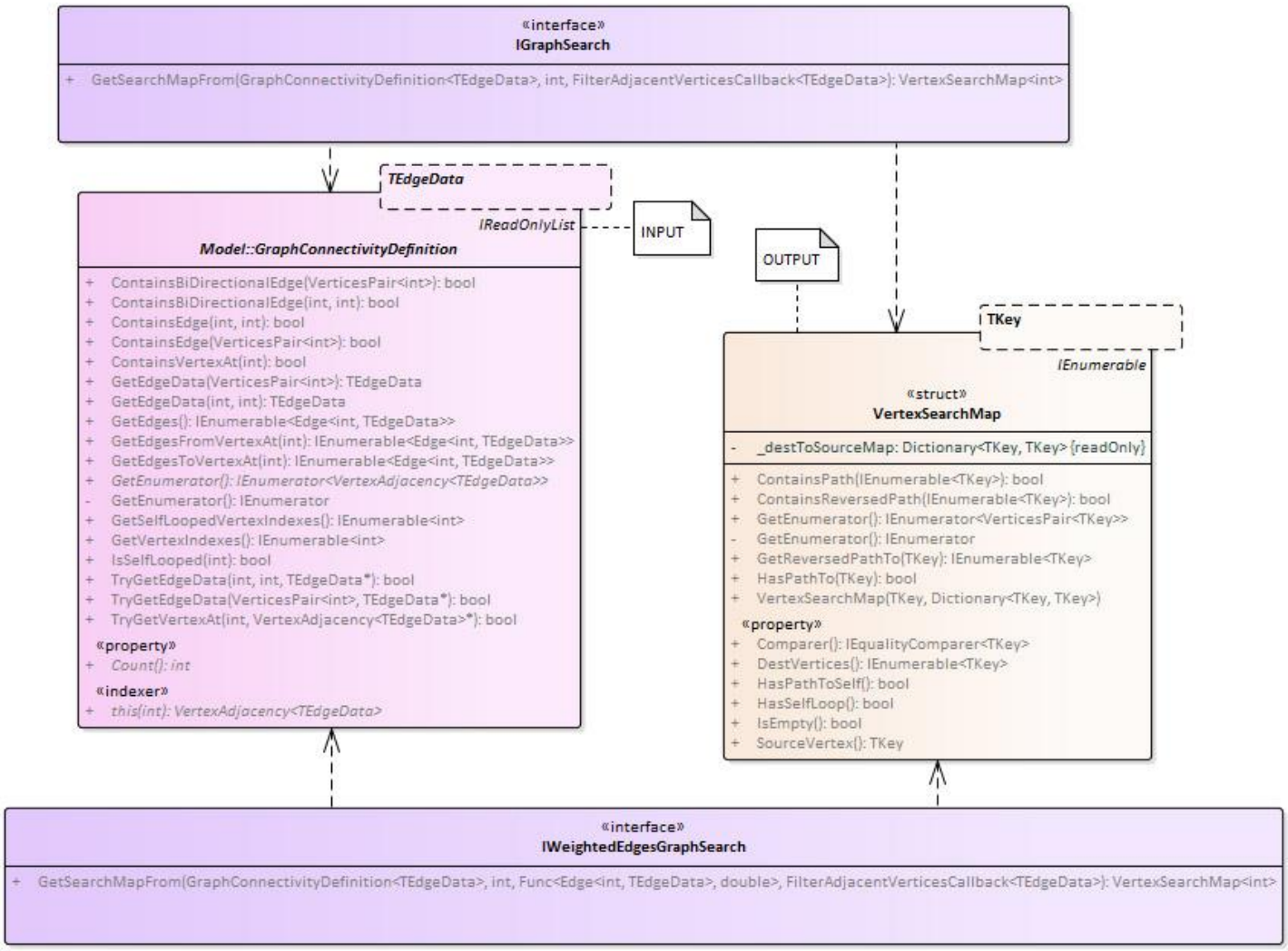
The graphs are needed for the "Chrono-Graph" representation and to provide type conversion in parsed expressions.

Graph Contract



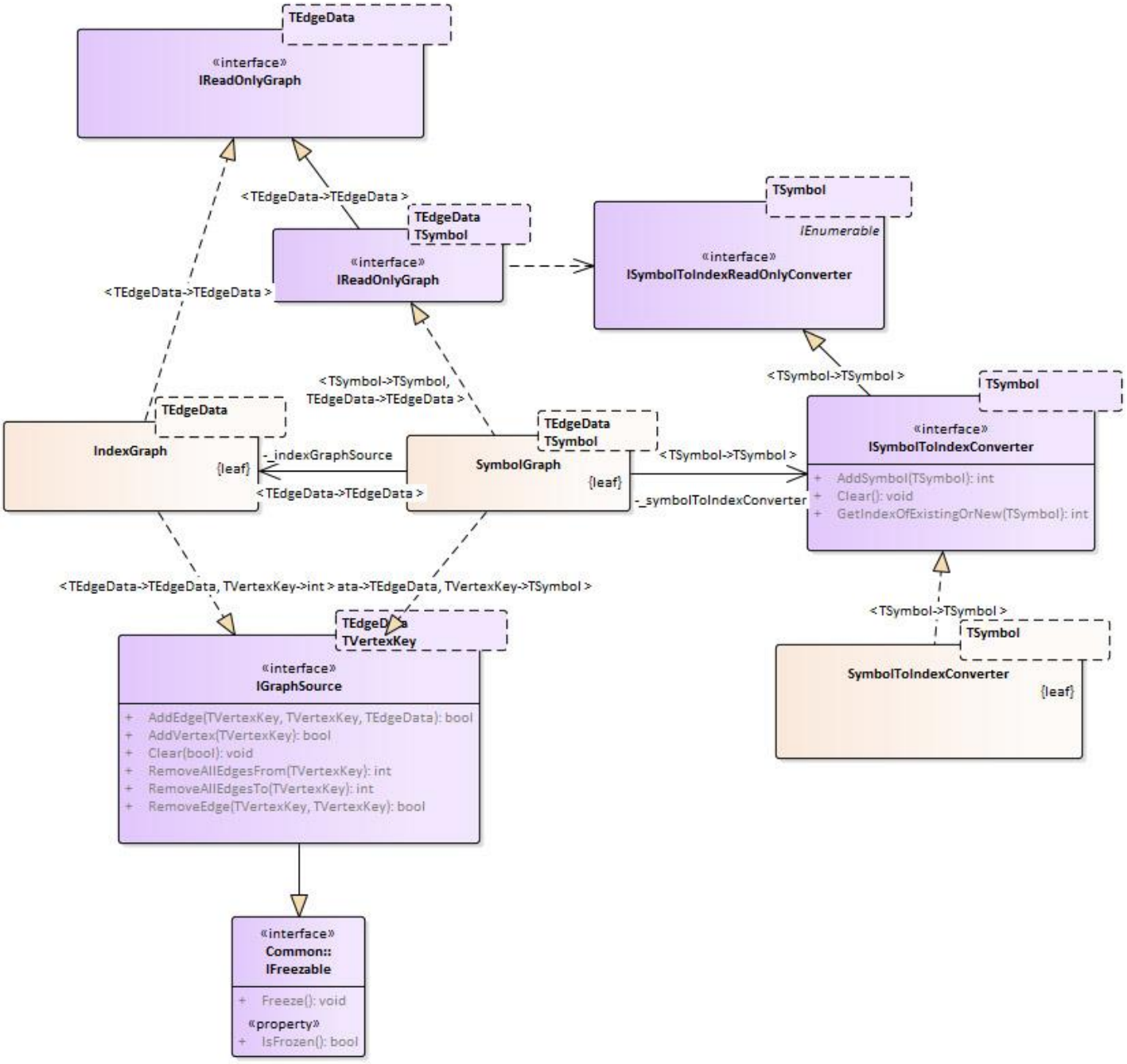
The contract above allows to decuple any graph implementation from the graphs algorithms implementation

Graph algorithm contract



Every graph algorithm should work on GraphConnectivityDefinition only.
Search algorithm contract allows to filter edges by provided delegate.

Symbol Graph Implementation



The graph reading contract is not coupled with graph definition contract, which allows to provide application specific graph definition API.

Above implementation of **SymbolGraph** will be used to store type converters of parsed expressions.

The **IGraphSource** interface is just a graph definition contract for regular graphs.

IFreezable

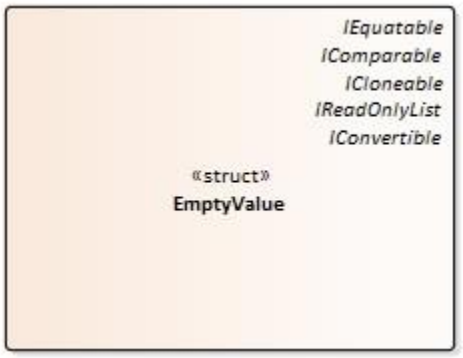
IGraphSource extends **IFreezable** interface that allows to freeze a graph (make it read-only).

Frozen graph is optimized and has removed reference to the vertices pool what reduces a memory usage by the graph. When graph is frozen it stays frozen till the end of its lifetime.

Execution of not pure operation on frozen graph will cause **ReadOnlyException**.

Empty edge data

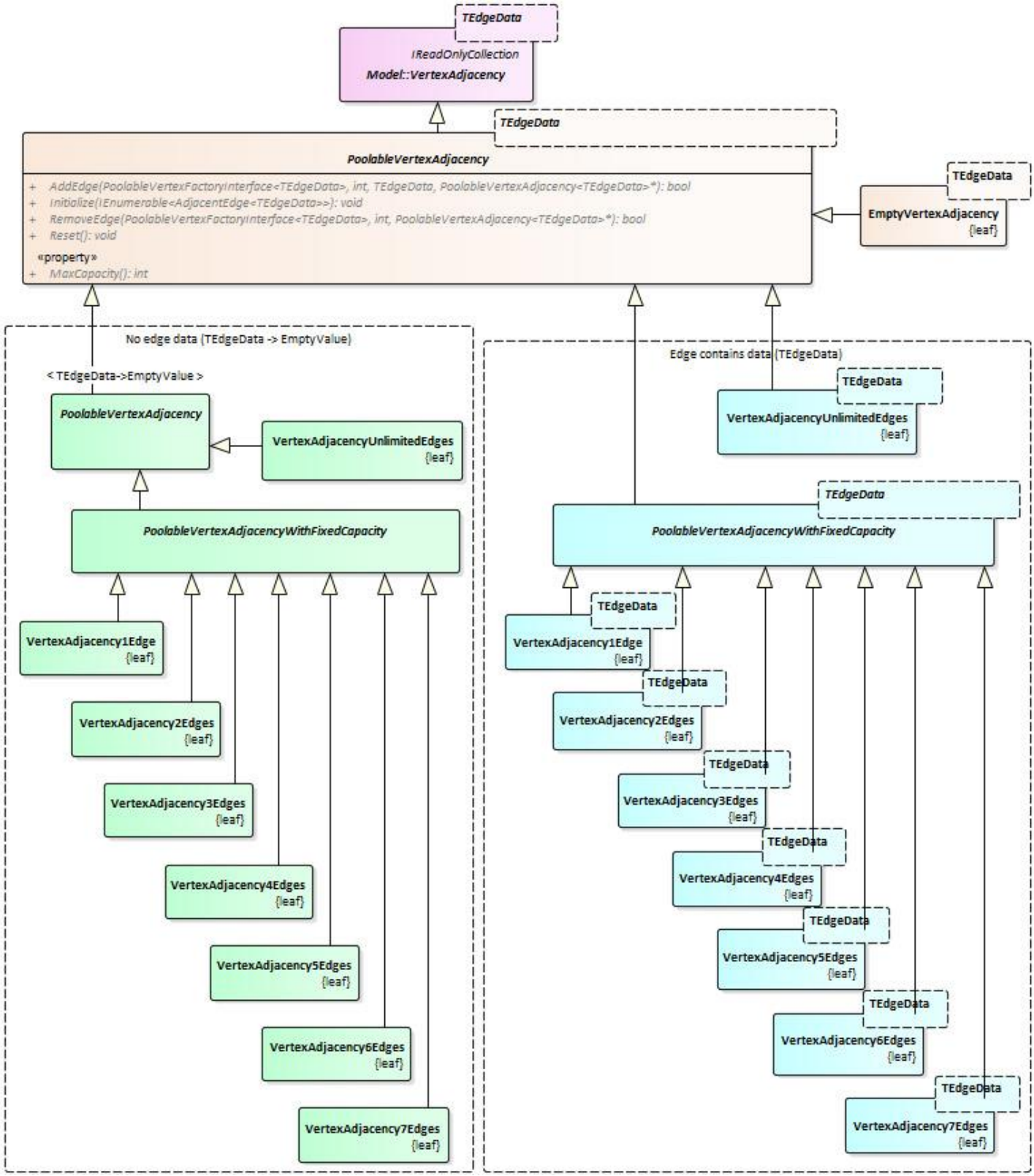
When graph doesn't contain edge data then **EmptyValue** structure type should be used as **TEdgeData** generic parameter.



EmptyValue is a structure that doesn't contain neither any fields nor properties.

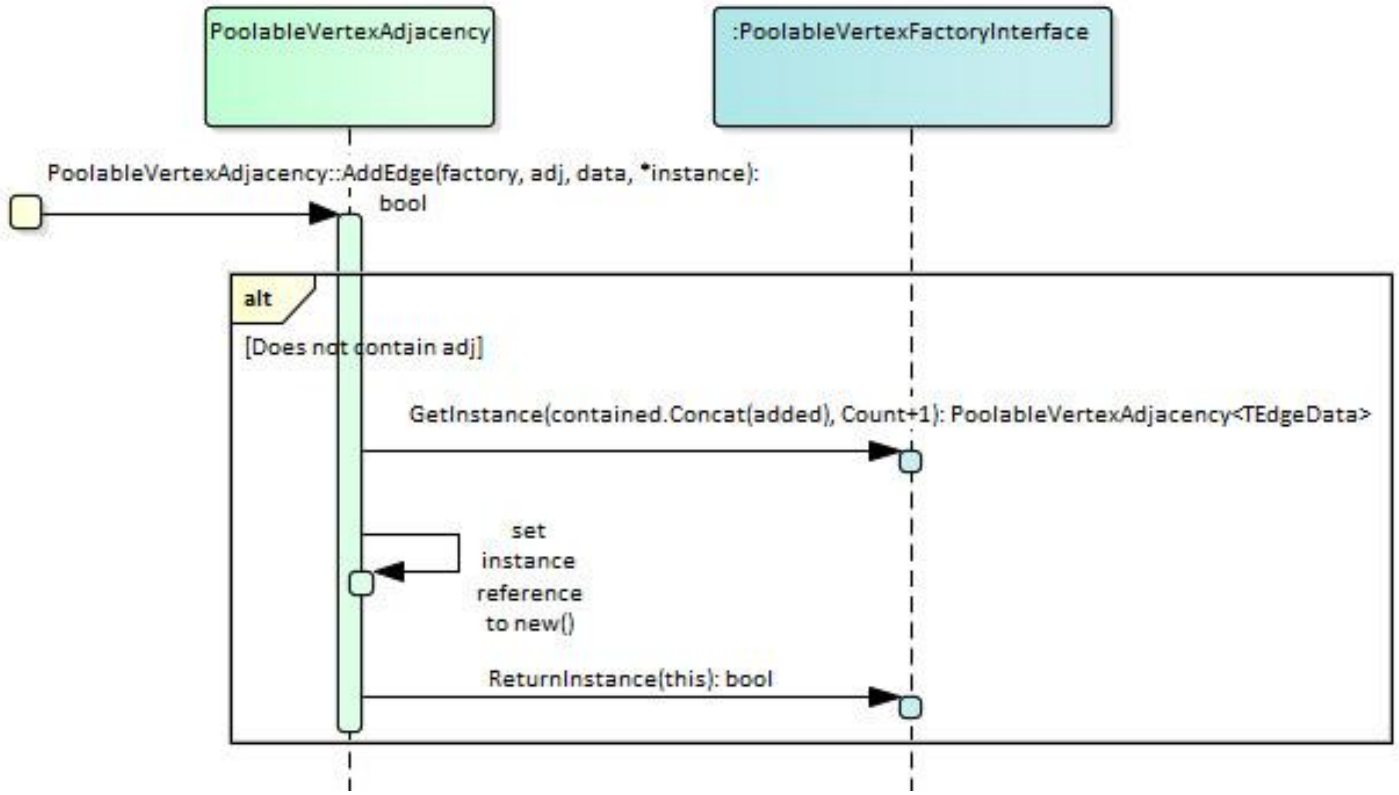
For EmptyValue as TEdgeData different **VertexAdjacency** types are instantiated and simplified graph extension methods are provided (e.g. AddEdge without data parameter) .

Poolable vertex adjacency



Performance tests shows that up to 7 vertices there is significant difference in accessing (for reading) adjacency information between HashSet/Dictionary based construction vs plain (index, data) fields-based construction.
Up to 7 adjacent vertices the HashSet/Dictionary based VertexAdjacency is much slower.

Vertex adjacency with fixed capacity classes were provided to increase reading performance and reduce memory usage of vertices with a number of adjacent edges less than eight.
Vertex adjacency with fixed capacity instance can store only specified constant number of adjacent edges.
When the edge is added then the current vertex instance must be replaced by instance with higher capacity, as it is shown in diagram below:



The vertex factory is implemented as an object pool, what **reduces garbage collection** effort.
On GetInstance factory returns instance of vertex type determined based on number of edges:

- 0 – Empty vertex adjacency (singleton – instance is reused for each empty vertex)
- 1-7 – Specific for each number of edges that can contain only fixed number of edges.
- >7 – HashSet/Dictionary based vertex that can contain unlimited number of edges.

Empty vertex adjacency doesn't contain any specific information that is why is provided as a singleton.

Graph compactization

IGraphConnectivityDefinitionFactory exposes CreateCompacted method:

```
[Pure]
ReindexedDataResult<GraphConnectivityDefinition<TEdgeData>> CreateCompacted<TEdgeData>(
                    GraphConnectivityDefinition<TEdgeData> source,
                    IVertexAdjacencyFactory<TEdgeData> verticesFactory,
                    bool sharedVerticesInstances,
                    IEqualityComparer<VertexAdjacency<TEdgeData>> vertexEqualityComparer = null);
```

CreateCompacted method returns compacted graph with re-indexing information.

The compactization operation sorts vertices by number of edges, putting vertices with lower number of adjacent edges at the end of the vertex list.

It doesn't allocate memory for empty vertices but provides virtual access to them.

It allows to reuse instances of vertices with same information to reduce memory usage.

Compacted graph is read-only.

There are extension methods like ToCompacted or GetCompacted that allows to execute compactization on IReadOnlyGraph instance.

The result of an operation contains information that allows to translate old vertex indexes into new order.

Example of compactization:

Original graph adjacency memory allocation:

0 -> 1

1 -> 0, 2, 3

2 -> 0, 2, 3

3 -> (empty vertex singleton instance)

4 -> (empty vertex singleton instance)

5 -> 1, 2, 3, 4, 5

Symbol map: 0 -> “A”, 1 -> “B”, 2 -> “C”, 3 -> “D”, 4 -> “E”, 5 -> “F”

Allocation after compactization with reusing of vertex instances:

0 -> 1, 2, 4, 5, 0

1 -> 3, 2, 4

2 (same instance as 1) -> 3, 2, 4

3 -> 1

Symbol map: 0 -> “F”, 1 -> “B”, 2 -> “C”, 3 -> “A”, 4 -> “D”, 5 -> “E”

The memory is not allocated for the vertices 3 and 4, but vertices are included in a result view (Vertices count is 6, ContainsVertexAt(4) returns true).

Please refer to the code for more details.