Table des matières

[Introduction 1](#_Toc474423608)

[Template Actions 4](#_Toc474423609)

[TypeCobol Razor Templates 4](#_Toc474423610)

[Code Generator Implementation 8](#_Toc474423611)

[Source Document concept 9](#_Toc474423612)

[Linear Node Source Code Maper. 11](#_Toc474423613)

# Introduction

This document describes the implementation of the TypeCobol Code Generator: Codegen.

TypeCobol

TypeCobol is: Prototype of an incremental Cobol compiler front-end for IBM Enterprise Cobol 5.1 for zOS syntax.This prototype is a work in progress and is written in C#.

The Generation process follows the classical workflow:

TypeCobol Source File

Cobol85 Source file

Figure 1: Main workflow

Thus the Inputs of the Code Generation phase are: The Abstract Syntax tree formed by Nodes and the Input Text Lines of the original source file.

All Inputs can be obtained from an instance of the CompilationUnit class from which Input TokenLines and the RootNode can be retrieved.

The Code Generation phase steps like the following workflow.

1. Code template actions application
2. Node Linearization
3. Code generation in linear time.

Figure 2: Code generation workflow.

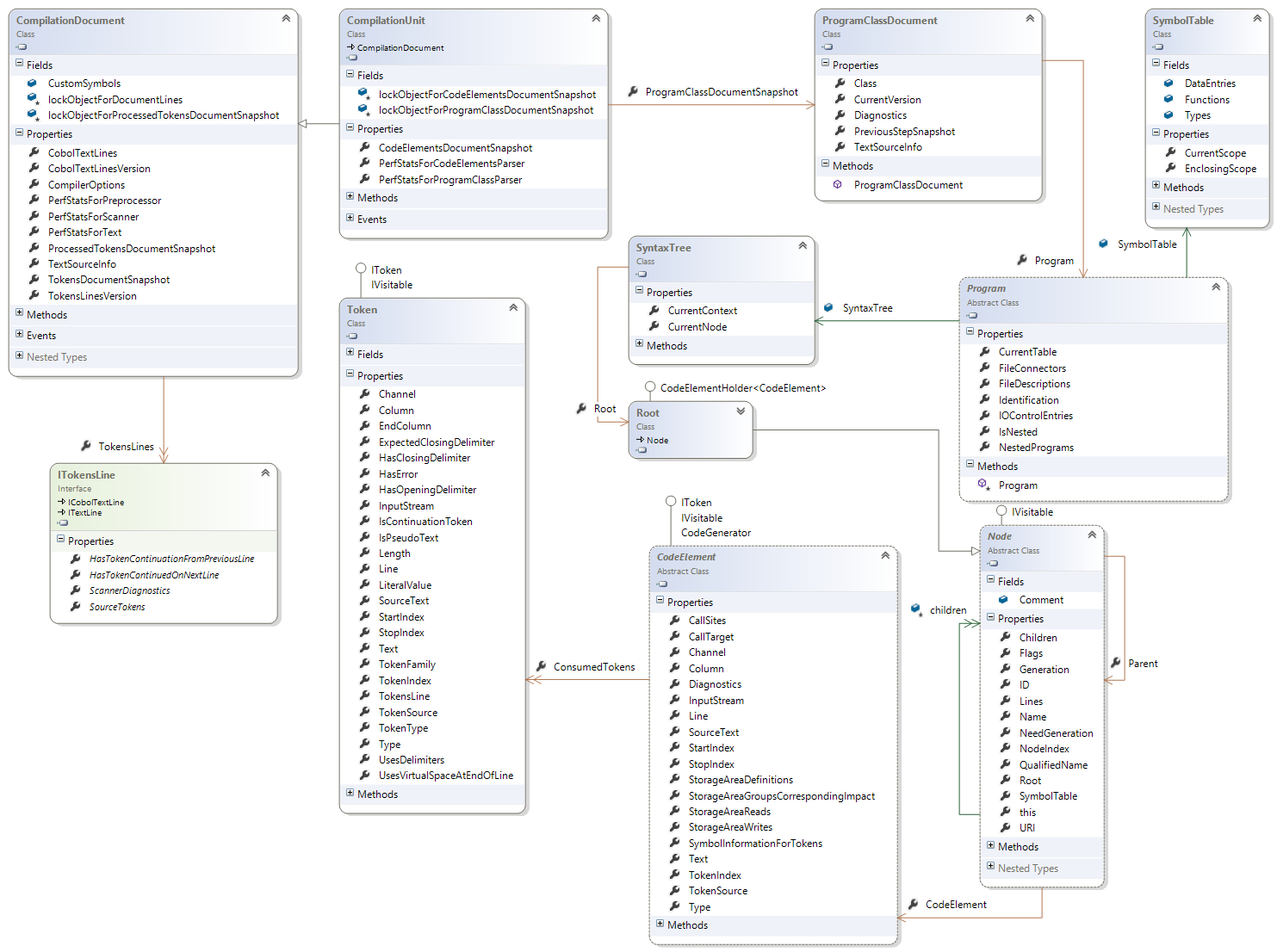


Figure 3: Class diagram: from CompilationUnit to TokenLines and Root Node.

# Template Actions

The Code generator uses code generation based on templates, by using a templating engine built upon Microsoft's Razor parsing technology:

<https://razorengine.codeplex.com/>

<https://github.com/Antaris/RazorEngine>

<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor>

The RazorEngine allows you to use Razor syntax to build robust templates. Currently we have integrated the vanilla Html + Code support, but we hope to support other markup languages in future.  
  
Using the templating engine couldn't be easier, using a simple syntax, we can do the following:

string template = "Hello @Model.Name! Welcome to Razor!";

string result = Razor.Parse(template, new { Name = "World" });

The templating engine supports strict and anonymous types, as well as customised base templates, for instance:

Razor.SetTemplateBase(typeof(HtmlTemplateBase<>));

string template =

@"<html>

<head>

<title>Hello @Model.Name</title>

</head>

<body>

Email: @Html.TextBoxFor(m => m.Email)

</body>

</html>";

var model = new PageModel { Name = "World", Email = "someone@somewhere.com" };

string result = Razor.Parse(template, model);

## TypeCobol Razor Templates

Razor Templates used by TypeCobol Generator are in the file:

*TypeCobol\CLI\src\config\skeletons.xml*

This file is an XML file which use to specify template actions that can use Razor template. The schema of the skeletons.xml file is the following and is in the file:

*TypeCobol\Codegen\src\Skeletons\Skeleton.xsd*

|  |  |
| --- | --- |
| **Schema Elements** | **Code / Description** |
| This is the Type that enumerates the available actions that can be applied to a Node.   * Comment * Create * Erase * Expand * replace | <!-- Enumeration Type of the action attribute in a pattern element-->  <xs:simpleType name="ActionEnumType">  <xs:restriction base="xs:string">  <!-- Action to comment the original source code for this node-->  <xs:enumeration value="comment"/>  <!-- Action to create a New Generate Node-->  <xs:enumeration value="create"/>  <!-- Delete words in node input -->  <xs:enumeration value="erase"/>  <!-- The Expand action : Comment the node, remove all children add extra nodes, generate code-->  <xs:enumeration value="expand"/>  <!-- Action to replace a Node by a new Generated one -->  <xs:enumeration value="replace"/>  </xs:restriction>  </xs:simpleType> |
|  | A List of skeletons. A skeleton represents a code generation scheme that can be applied to a node that matches or verifies a set of conditions on whose nodes have attributes values:   * node : the node’s type (à la C# namespace) * name : The node’s name * level : The Cobol Level * type : A Cobol Type * sender: condition-name on SET for condition-names (to true) * receiver: For statements using items for receiving data, this is the name where the value is written. * unsafe: For statements using items for receiving data, are unsafe writes allowed ? * function: relative to function call information ??? (Not Used ???) * definitions: Get all Types and functions definitions of the current node, this definitions will be passed to the **create** and **replace** actions. * variables: not used. * Typecobol: Get all TypeCobol Qualified Symbol Reference * Visibility: Get the visibility of a function declaration if the node is a function declaration. * Copyname: The name of the first copy if one exists "?TCRFUN\_LIBRARY\_COPY?" otherwise.   Pattern is used to describe razor template to be applied to an action.   * Name: is the pattern’s identifier * Group: This * Location: COBOL paragraph where the generated code must be but * Action: the action to be performed. * Var: the variable that are used in the pattern in the format “name=value, …, name=value” * Position: Index position in the target node parent, of insertion. |

The following actions are available:

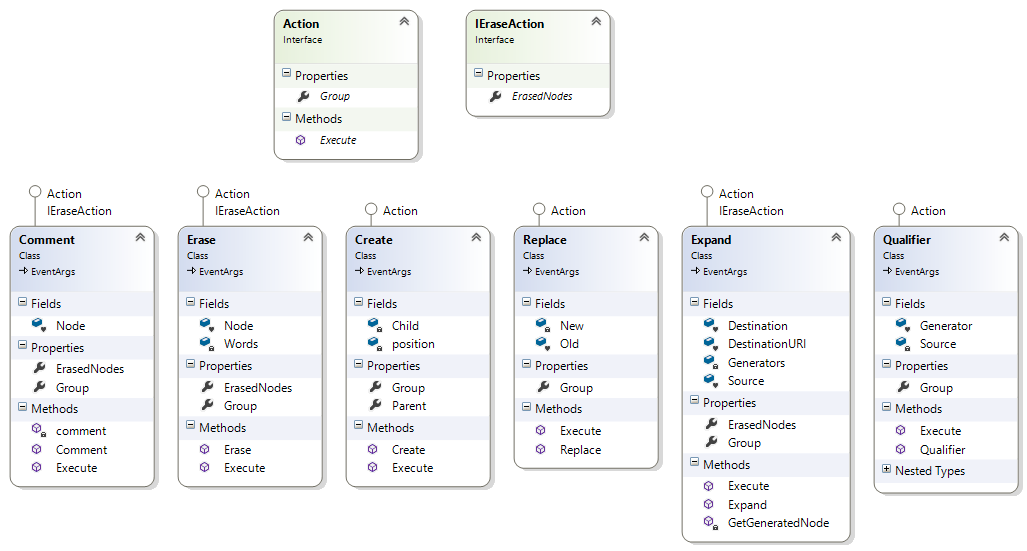
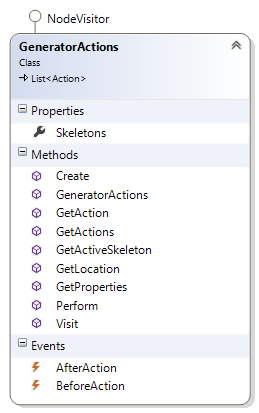


Figure 4: Class Diagram; Available Actions.

|  |  |
| --- | --- |
| **Action** | **Description.** |
| Comment | Mark a node to be commented with all its children. |
| Erase | Action to remove a Node from the generated code. The Removed node will be commented, all its children will be cleared.  If the erase action contains words to be erased from the input template, a new GeneratedNode is built and its output is the input without the word erased. |
| Create | Action to create a new Generate Node |
| Replace | Action to replace a Node by a New Generated one. The Old one is commented but not its children. The children are copied in the new Generated Node. |
| Expand | Action that created a new node by expanding the current node with new generated node. The source node will be commented in its children cleared.  The expanded node will be added in the Destination parent node as child at the right index. |
| Qualifier | Action used to detect if a node is subject to Type Cobol Qualifier Symbol Reference Style, and to Generate Node to perform the permutation and the replacement of :: by OF. |

All actions are performed by the class GeneratorActions.



# Code Generator Implementation

The main process of the Code Generator: is to create a structure that maps Nodes to their corresponding lines in the source code. Each node to be generated is associated a target source code buffer which contents the original source code corresponding to the Node. Generating the code for a Node is then replacing in the associated source code buffer is original code by the generated one, this implies the calculation of the positions of the Node within the associated source code buffer.



Figure 5: Code Generator sequence diagram

The sequence diagram above, illustrates the sequence of actions.

## Source Document concept

The concept of a source document is to be able to store a source text inside a structure that can allows to track positions inside the source code document. So that when portions of text are inserted or removed, these positions are automatically updated (moved) to always reference the tracked position.

The Implementation of such structure implies 6 classes:

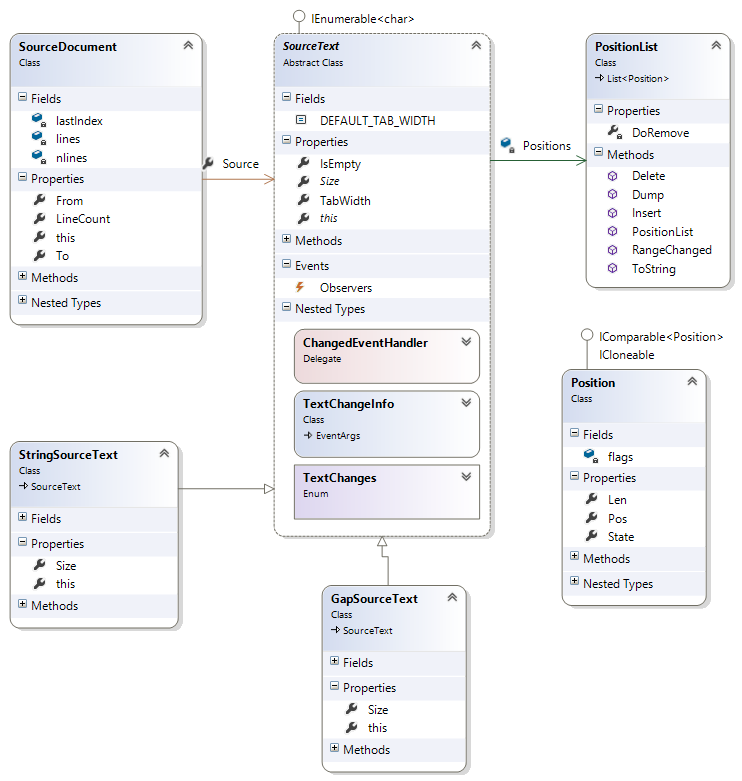


Figure 6: Class Diagram of the Source Document concept

|  |  |
| --- | --- |
| **Class** | **Description** |
| Position | Represents a moveable position inside a source code. |
| PositionList | A List to represent a collection of Positions. On Insertion or deleting the positions inside the list are recalculated. |
| SourceText | An abstract representation of a Source Text. The class Text defines methods to manage a buffer for holding source text. Together with the source text is a list of robust pointers (called Positions) into the source text.  Dependents of the Source Text are noticed by the observer mechanism and receive a change event. |
| StringSourceText | A Source Text which is represented as a String of characters. |
| GapSourceText | A Gap Source Text, with the concept of <https://en.wikipedia.org/wiki/Gap_buffer>: that allows efficient insertion and deletion operations clustered near the same location. |
| SourceDocument | Class that represents a source document, with lines tracking on insertion and deleting. |

## 

## Linear Node Source Code Maper.

The Code generator, generates code by injecting (inserting, replacing, deleting) text in the original source code at specific Node’s position.

**Why injecting code in the original source code**? The answer to this question is that, the code generator must preserve all whitespaces, comments and preprocessor directives that were in the original source at their positions, this because such lexical items are ignored during lexical analysis.

In order to inject text at a right position for a given Node whose code is being generated, it is necessary to calculate the range of positions covered by the node within the original source code.

From the positions of a node, it is then possible to determine which lines in the source code covered the node, and from the target lines we can extract the corresponding source code in a separate buffer.

We can then create the following Mappings.

|  |  |  |
| --- | --- | --- |
| **Domain** |  | **CoDomain** |
| * Node 🡪 Positions 🡪(Line \* Source code) * Node | 🡺 |  |
| * Line | 🡺 | (List<Node> \* ) |

These mappings say that:

* Given a Node we can associate it a Source Code generation buffer in which code injection can be performed based on Node’s positions inside the buffer.
* Given a Line (Line number) we can associate to it which Nodes are generated on the line, and the target Source Code buffer.

Thus it is possible to generate linearly all node, by code injection in target line buffers, injection location are tracked using Node’s positions in the target buffer.

**BUT it still one question: how to determine Node’s positions?**

The answer to this question, is that given a Node it is possible to know its CodeElement and from its CodeElement the lexical tokens consumed by the CodeElement. Each lexical token has: Line number, a StartColum and EndColum values.

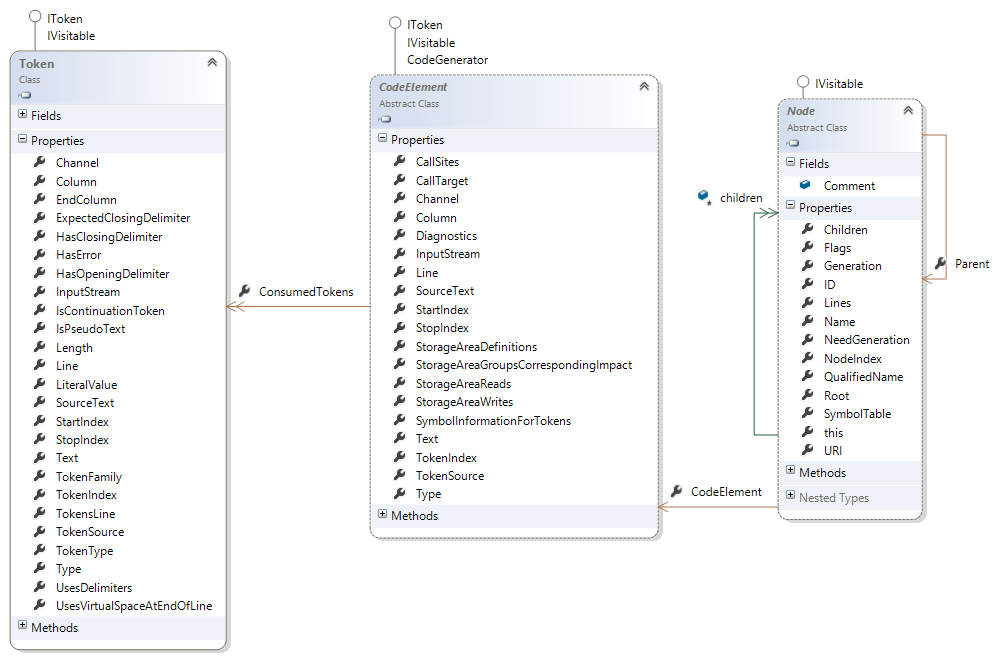


Figure 7: From Node to CodeElement, to Consumed Tokens

The following method:

public Tuple<int, int, int, List<int>, List<int>> FromToPositions(Node node)

In the class ***TypeCobol.Codegen.Generator*** calculates all position information for a given Node. Calculated positions are related to the Original Source Document. The method return a tuple whose items are:

|  |  |
| --- | --- |
| **Items** | **Description** |
| Item1 | The ***From*** column which is the column of the first valid Token |
| Item2 | The ***To*** column which is the ending column in the last line covered by Node’s tokens. |
| Item3 | The **Span** which is the indentation on the last line covered by the Node’s tokens up to the ending token’s column. |
| Item4 | The list of Line Numbers covered by the Node’s tokens. |
| Item5 | The list of offsets of each line from the first line inside the target source code buffer. |

Given these items, it is possible by translating these positions, to the portion of text covered by the Node’s lines, to calculate the (**From**, **To**) positions of the Node within the target source code buffer.

To Construct the Linear Node Source Code Mapper we can distinguish the following Steps:

1. Node Linearization Phase
2. Removed Nodes Phase
3. Node’s Buffer Source Code association
4. Function Declaration lines relocation
5. Node without position attachment.

### Data Structure

The Linear Node Source Code Mapper uses the following class as Data Structure.

|  |  |
| --- | --- |
| Classes | Description. |
|  | The class LinearNodeSourceCodeMapper is the main class whose members are described below. This class maintains the array of Line information *LineData* and the indexed list of nodes *Nodes*.     * The Inner class LineInfo is used to store data related to a line:  (*Buffer* the target source code buffer, *FunctionBodyBuffer* if the line is related to a function body buffer, *LineNodes* the list of Node index associated to the line).      * The Inner class NodeData is used to store information related to a Node.      * The Inner class NodeFunctiondata is a specialization of the class NodeData used to store specific information related to a Function Declaration Node.      * The Inner class LinearGeneratedNode is used to represent additional Nodes created to the Linearization, these Nodes are nodes that represents the code source of text lines that are relocated when the function declaration is relocated at the tail of the generated source document.      * The Inner Enumeration *Phase* enumerates ALL phases performed during the linearization process. |

### Node Linearization Phase

The Linearization is the process to associate to each line in the original source code its nodes, and to associate to each node its target Source Code Buffer. So that we can establish the following relations.

* Node -> SourceText
* Line -> (List<Node> \* SourceText)

The first relation says that given a Node we can associate its corresponding Source Code Buffer, and the second relation says that given a Line (Line Number) we can associate its list of nodes and its Source Code Buffer.

The Linearization Phase starts from the RootNode and traverse all its children recursively.

The following method of the class LinearNodeSourceCodeMapper is executed:

|  |
| --- |
| /// <summary>  /// The Linearization Phase.  /// <param name="node">The node to linearize</param>  /// <param name="functionBody">true if the node belongs to a function body, false otherwise</param>  /// <returns>True if children of the given node must be visited, false otherwise</returns>  /// </summary>  private bool ProcessLinearization(Node node, bool functionBody = false); |

### Removed Nodes Phase

The Removed Nodes Phase is the phase that marks all node that must be removed from the generation process. Their field Removed in the NodeData instance is set to true.

### Node’s Buffer Source Code association