



# **Systemdokumentation Projekt Symbolparser**

**Version 1.0**

S. Offenberger, S. Vogelhuber

Hagenberg, 12. November 2025

# Inhaltsverzeichnis

<b>1 Organisatorisches</b>	<b>3</b>
1.1 Team . . . . .	3
1.2 Aufteilung der Verantwortlichkeitsbereiche . . . . .	3
1.3 Aufwand . . . . .	4
<b>2 Anforderungsdefinition (Systemspezifikation)</b>	<b>5</b>
<b>3 Systementwurf</b>	<b>8</b>
3.1 Klassendiagramm . . . . .	8
3.2 Designentscheidungen . . . . .	9
<b>4 Dokumentation der Komponenten (Klassen)</b>	<b>10</b>
<b>5 Testprotokollierung</b>	<b>11</b>
<b>6 Quellcode</b>	<b>19</b>
6.1 Object.hpp . . . . .	19
6.2 Symbolparser.hpp . . . . .	20
6.3 Symbolparser.cpp . . . . .	22
6.4 ISymbolFactory.hpp . . . . .	24
6.5 Variable.hpp . . . . .	25
6.6 Variable.cpp . . . . .	27
6.7 Type.hpp . . . . .	28
6.8 Type.cpp . . . . .	29
6.9 Test.hpp . . . . .	30
6.10 SingletonBase.hpp . . . . .	33
6.11 JavaVariable.hpp . . . . .	34
6.12 JavaVariable.cpp . . . . .	35
6.13 JavaSymbolFactory.hpp . . . . .	36
6.14 JavaSymbolFactory.cpp . . . . .	37
6.15 IECVariable.hpp . . . . .	38
6.16 IECVariable.cpp . . . . .	39
6.17 IECSymbolFactory.hpp . . . . .	41
6.18 IECSymbolFactory.cpp . . . . .	42

# 1 Organisatorisches

## 1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: Simon.Offenberger@fh-hagenberg.at
- Susi Sorglos, Matr.-Nr.: yyyy, E-Mail: Susi.Sorglos@fh-hagenberg.at

## 1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
  - Design Klassendiagramm
  - Implementierung und Test der Klassen:
  - Implementierung des Testtreibers
  - Dokumentation
    - \* Object
    - \* Symbolparser
    - \* ISymbolFactory
    - \* Variable
    - \* Type
    - \* JavaVariable
    - \* JavaType
    - \* JavaSymbolFactory
    - \* IECVariable

- \* IECType
- \* IECSymbolFactory
- Simon Vogelhuber
  - Design Klassendiagramm
  - Implementierung des Testtreibers
  - Dokumentation
  - Implementierung und Komponententest der Klassen:
    - \* Object
    - \* Symbolparser
    - \* ISymbolFactory
    - \* Variable
    - \* Type
    - \* JavaVariable
    - \* JavaType
    - \* JavaSymbolFactory
    - \* IECVariable
    - \* IECType
    - \* IECSymbolFactory

### 1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich x Ph
- Simon Vogelhuber: geschätzt x Ph / tatsächlich x Ph

## 2 Anforderungsdefinition (Systemspezifikation)

Das Ziel ist es einen Symbolparser zu implementieren, der verschiedene Programmiersprachen unterstützt. Der Parser soll in der Lage sein Typen und Variablen zu erkennen und zu verarbeiten. Dazu wird eine Factory benötigt, die die entsprechenden Objekte für die verschiedenen Sprachen erzeugt.

### Funktionen des Symbolparsers:

- Auswählen der Programmiersprachen (auswählen der SymbolFactory)
- Speichern der erzeugten Objekte in einem Container.
- Erzeugen von Variablen und Typen über die SymbolFactory
- Überprüfung ob Typen und Variablen gültig sind.
- Beim Wechsel der SymbolFactory, werden alle Objekte der alten Factory in ein File gespeichert. Und die Objekte der neuen Factory werden aus dem File geladen.

### Funktionen der SymbolFactory:

- Erzeugen von Variablen und Typen der jeweiligen Programmiersprache.

### Funktionen der Variable:

- Speichern des Variablenamens
- Speichern des Variablentyps
- Auswerten der Variablen Deklaration (Syntaxprüfung)
- Zurückgeben des Variablenamens
- Zurückgeben des Variablentyps

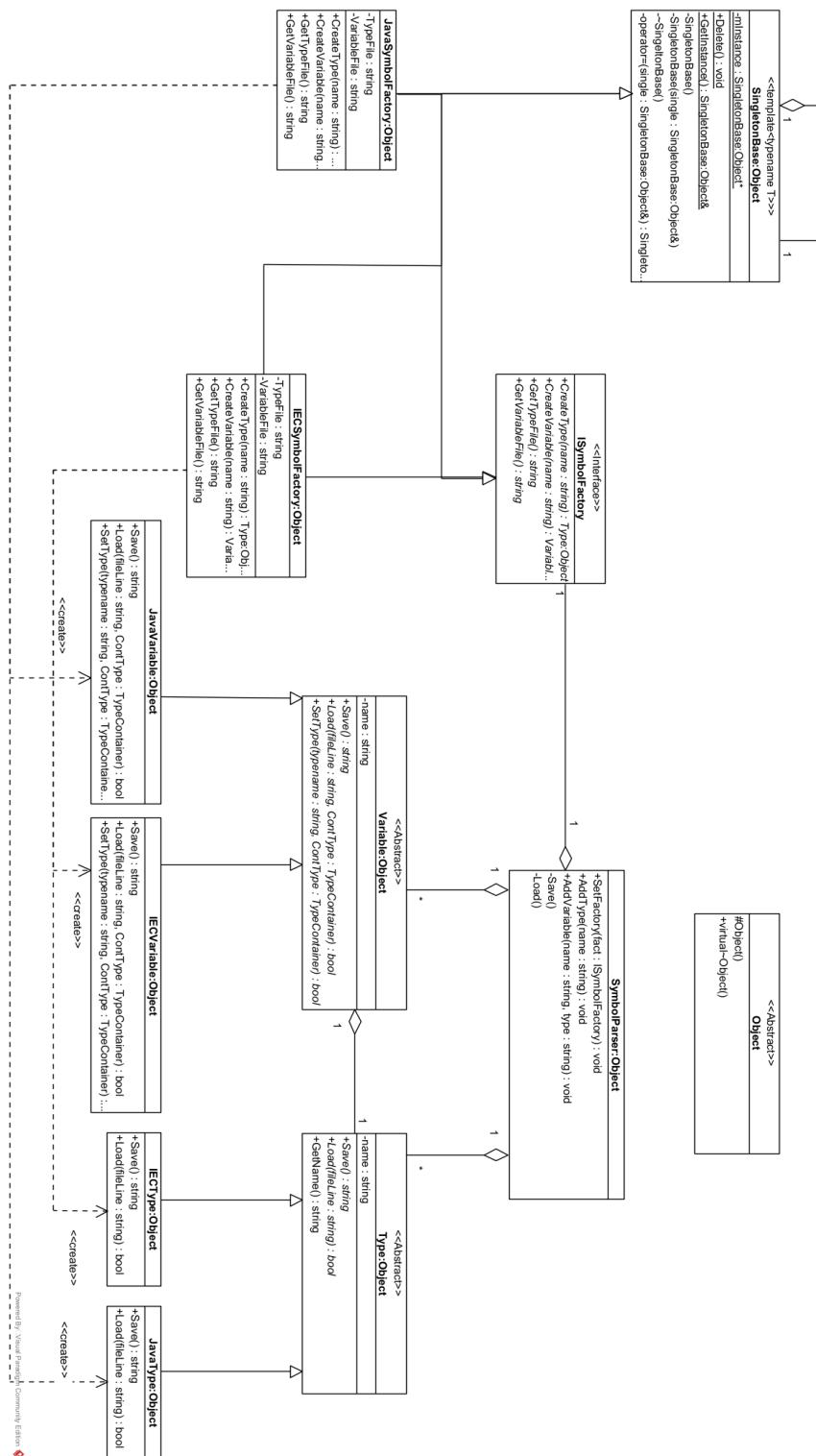
### Funktionen des Type:

- Auswerten der Typdeklaration (Syntaxprüfung)
- Speichern des Typnamens
- Zurückgeben des Typnamens



## 3 Systementwurf

## 3.1 Klassendiagramm



## 3.2 Designentscheidungen

### **Verwendung des Factory-Patterns:**

Das Factory-Pattern wurde verwendet, um die Erstellung von Objekten der verschiedenen Programmiersprachen zu kapseln. Das ermöglicht eine einfache Erweiterung des Systems um weitere Sprachen, ohne dass der Symbolparser angepasst werden muss. Der Parser Speichert hierfür eine Referenz auf die aktuelle SymbolFactory, die zur Laufzeit gewechselt werden kann.

### **Verwendung des Singleton-Patterns:**

Das Singleton- Pattern wurde für die SymbolFactory implementiert, um sicherzustellen, dass nur eine Instanz der Factory existiert.

### **Verwendung von Vererbung und Polymorphie:**

Die Klassen Variable und Type sind Basisklassen, von denen spezifische Implementierungen für jede Programmiersprache abgeleitet sind. Dadurch kann der Symbolparser generisch mit den Basisklassen arbeiten, ohne die spezifischen Implementierungen zu kennen.

### **Container für Objekte:**

Der Symbolparser verwendet einen Container (std::vector), um die erzeugten Objekte zu speichern. Dies ermöglicht eine einfache Verwaltung und Iteration über die Objekte. Für die Variablen werden unique- Pointer gespeichert, die Types werden jedoch als shared-Pointer gespeichert, da mehrere Variablen denselben Type referenzieren können.

### **SymbolParser:**

Der SymbolParser ist die zentrale Klasse, die die Interaktion mit dem Benutzer und die Verwaltung der Objekte übernimmt. Er bietet Methoden zum Setzen der aktuellen SymbolFactory, zum Erzeugen von Variablen und Typen sowie zum Speichern und Laden der Objekte. Der Parser überprüft ob eine eingegebene Variable oder ein Type gültig ist, indem er die entsprechenden Methoden der Objekte aufruft.

## 4 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis [./doxy/html/index.html](#)

## 5 Testprotokollierung

```
1
2 **** TESTCASE START ****
3
4 ****
5
6
7 **** Test IEC Var Getter ****
8
9
10
11 **** TESTCASE START ****
12
13 ****
14
15 Test Variable Get Name
16 [Test OK] Result: (Expected: asdf == Result: asdf)
17
18 Test Variable Get Type
19 [Test OK] Result: (Expected: int == Result: int)
20
21 Test Variable Set Name
22 [Test OK] Result: (Expected: uint_fast_256_t == Result: uint_fast_256_t)
23
24 Check for Exception in Testcase
25 [Test OK] Result: (Expected: true == Result: true)
26
27 Test Exception in Set Name
28 [Test OK] Result: (Expected: ERROR: Empty String == Result: ERROR: Empty
   ↪ String)
29
30 Test Exception in Set Type with nullptr
31 [Test OK] Result: (Expected: ERROR: Passed in Nullptr! == Result: ERROR:
   ↪ Passed in Nullptr!)
32
33 Test Variable Get Type after set with nullptr
34 [Test OK] Result: (Expected: int == Result: int)
35
36 Test Variable Get Type after set
37 [Test OK] Result: (Expected: uint_fast512_t == Result: uint_fast512_t)
38
39 Test for Exception in TestCase
40 [Test OK] Result: (Expected: true == Result: true)
```

```
41
42
43 **** Test Java Var Getter ****
44
45
46
47 ***** TESTCASE START *****
48
49
50 ***** TESTCASE START *****
51
52 ***** TESTCASE START *****
53
54 Test Variable Get Name
55 [Test OK] Result: (Expected: jklm == Result: jklm)
56
57 Test Variable Get Type
58 [Test OK] Result: (Expected: int == Result: int)
59
60 Test Variable Set Name
61 [Test OK] Result: (Expected: uint_fast_256_t == Result: uint_fast_256_t)
62
63 Check for Exception in Testcase
64 [Test OK] Result: (Expected: true == Result: true)
65
66 Test Exception in Set Name
67 [Test OK] Result: (Expected: ERROR: Empty String == Result: ERROR: Empty
   ↪ String)
68
69 Test Exception in Set Type with nullptr
70 [Test OK] Result: (Expected: ERROR: Passed in Nullptr! == Result: ERROR:
   ↪ Passed in Nullptr!)
71
72 Test Variable Get Type after set with nullptr
73 [Test OK] Result: (Expected: int == Result: int)
74
75 Test Variable Get Type after set
76 [Test OK] Result: (Expected: uint_fast512_t == Result: uint_fast512_t)
77
78 Test for Exception in TestCase
79 [Test OK] Result: (Expected: true == Result: true)
80
81
82 *****
```

```
83
84
85
86 **** Test IEC Type Getter ****
87
88
89 **** TESTCASE START ****
90
91 ****
92
93 Test Type Get Name after Set
94 [Test OK] Result: (Expected: unit_1024_t == Result: unit_1024_t)
95
96 Test Exception in Set Type
97 [Test OK] Result: (Expected: true == Result: true)
98
99 Test Exception in Set Type
100 [Test OK] Result: (Expected: ERROR: Empty String == Result: ERROR: Empty
101   ↪ String)
102
103 ****
104
105
106
107 **** Test Java Type Getter ****
108
109
110 **** TESTCASE START ****
111
112 ****
113
114 Test Type Get Name after Set
115 [Test OK] Result: (Expected: unit_1024_t == Result: unit_1024_t)
116
117 Test Exception in Set Type
118 [Test OK] Result: (Expected: true == Result: true)
119
120 Test Exception in Set Type
121 [Test OK] Result: (Expected: ERROR: Empty String == Result: ERROR: Empty
122   ↪ String)
123
124 ****
```

```
125
126
127 **** TESTCASE START ****
128
129 ****
130
131 Test Load Type Name IEC Var
132 [Test OK] Result: (Expected: mCont == Result: mCont)
133
134 Test Load Var Name IEC Var
135 [Test OK] Result: (Expected: SpeedController == Result: SpeedController)
136
137 Test Load Type Name IEC Var invalid Format
138 [Test OK] Result: (Expected: == Result: )
139
140 Test Load Var Name IEC Var invalid Format
141 [Test OK] Result: (Expected: == Result: )
142
143 Test Load Type Name IEC Var invalid Format
144 [Test OK] Result: (Expected: mCont == Result: mCont)
145
146 Test Load Var Name IEC Var invalid Format
147 [Test OK] Result: (Expected: == Result: )
148
149 Test Load Type Name IEC Var invalid Format
150 [Test OK] Result: (Expected: == Result: )
151
152 Test Load Var Name IEC Var invalid Format
153 [Test OK] Result: (Expected: == Result: )
154
155 Test Load Type Name IEC Var invalid Format
156 [Test OK] Result: (Expected: mCont == Result: mCont)
157
158 Test Load Var Name IEC Var invalid Format
159 [Test OK] Result: (Expected: == Result: )
160
161 Test Load Type Name IEC Var invalid Format
162 [Test OK] Result: (Expected: == Result: )
163
164 Test Load Var Name IEC Var invalid Format
165 [Test OK] Result: (Expected: == Result: )
166
167 Test Load Type Name IEC Var invalid Format
168 [Test OK] Result: (Expected: == Result: )
```

```
169 Test Load Var Name IEC Var invalid Format
170 [Test OK] Result: (Expected: == Result: )
171
172 Test Save LineFormat IEC Variable
173 [Test OK] Result: (Expected: VAR mCont : SpeedController; == Result: VAR
174     ↪ mCont : SpeedController; )
175
176 Test Save LineFormat IEC Variable
177 [Test OK] Result: (Expected: == Result: )
178
179
180 ****
181
182
183 **** TESTCASE START ****
184
185 ****
186
187 Test Load Type Name Java Var
188 [Test OK] Result: (Expected: mCont == Result: mCont)
189
190 Test Load Var Name Java Var
191 [Test OK] Result: (Expected: mBut == Result: mBut)
192
193 Test Load Type Name Java Var invalid Format
194 [Test OK] Result: (Expected: == Result: )
195
196 Test Load Var Name Java Var invalid Format
197 [Test OK] Result: (Expected: == Result: )
198
199 Test Load Type Name Java Var invalid Format
200 [Test OK] Result: (Expected: mCont == Result: mCont)
201
202 Test Load Var Name Java Var invalid Format
203 [Test OK] Result: (Expected: == Result: )
204
205 Test Load Type Name Java Var invalid Format
206 [Test OK] Result: (Expected: == Result: )
207
208 Test Load Var Name Java Var invalid Format
209 [Test OK] Result: (Expected: == Result: )
210
211 Test Load Type Name Java Var invalid Format
```

```
212 [Test OK] Result: (Expected: mCont == Result: mCont)
213
214 Test Load Var Name Java Var invalid Format
215 [Test OK] Result: (Expected: == Result: )
216
217 Test Load Type Name Java Var invalid Format
218 [Test OK] Result: (Expected: == Result: )
219
220 Test Load Var Name Java Var invalid Format
221 [Test OK] Result: (Expected: == Result: )
222
223 Test Load Type Name Java Var invalid Format
224 [Test OK] Result: (Expected: == Result: )
225
226 Test Load Var Name Java Var invalid Format
227 [Test OK] Result: (Expected: == Result: )
228
229 Test Save LineFormat IEC Variable
230 [Test OK] Result: (Expected: mCont mBut == Result: mCont mBut)
231
232 Test Save LineFormat IEC Variable
233 [Test OK] Result: (Expected: == Result: )
234
235
236 ****
237
238
239 ****
240         TESTCASE START
241 ****
242
243 Test Load Type Name IEC Type
244 [Test OK] Result: (Expected: SpeedController == Result: SpeedController)
245
246 Test Load Type Name IEC Type invalid Format
247 [Test OK] Result: (Expected: == Result: )
248
249 Test Load Type Name IEC Type invalid Format
250 [Test OK] Result: (Expected: == Result: )
251
252 Test Load Type Name IEC Type invalid Format
253 [Test OK] Result: (Expected: S2peedController == Result: S2peedController)
254
255 Test Load Type Name IEC Type invalid Format
```

```
256 [Test OK] Result: (Expected: == Result: )
257
258 Test Load Type Name IEC Type invalid Format
259 [Test OK] Result: (Expected: == Result: )
260
261 Test Save LineFormat IEC Type
262 [Test OK] Result: (Expected: TYPE SpeedController
263 == Result: TYPE SpeedController
264 )
265
266
267 ****
268
269
270 ****
271 TESTCASE START
272 ****
273
274 Test Load Type Name Java Type
275 [Test OK] Result: (Expected: SpeedController == Result: SpeedController)
276
277 Test Load Type Name Java Type invalid Format
278 [Test OK] Result: (Expected: == Result: )
279
280 Test Load Type Name Java Type invalid Format
281 [Test OK] Result: (Expected: == Result: )
282
283 Test Load Type Name Java Type invalid Format
284 [Test OK] Result: (Expected: S2peedController == Result: S2peedController)
285
286 Test Load Type Name Java Type invalid Format
287 [Test OK] Result: (Expected: == Result: )
288
289 Test Load Type Name Java Type invalid Format
290 [Test OK] Result: (Expected: == Result: )
291
292 Test Save LineFormat Java Type
293 [Test OK] Result: (Expected: class SpeedController
294 == Result: class SpeedController
295 )
296
297
298 ****
299
```

```
300
301 **** TESTCASE START ****
302
303 Normal Operating Parser
304
305 [Test OK] Result: (Expected: true == Result: true)
306
307 .AddType() - add empty type to factory
308 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
309   ↪ ERROR: Provided string is empty.)
310
311 .AddVariable() - add empty type to factory
312 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
313   ↪ ERROR: Provided string is empty.)
314
315 .AddVariable() - add empty var to factory
316 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
317   ↪ ERROR: Provided string is empty.)
318
319 .AddVariable() - add variable with nonexisting type
320 [Test OK] Result: (Expected: ERROR: Provided type does not exist. == Result:
321   ↪ : ERROR: Provided type does not exist.)
322
323 TEST OK!
```

## 6 Quellcode

### 6.1 Object.hpp

```
1 //*****\n2 * \file Object.hpp\n3 * \brief common ancestor for all objects\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8 #ifndef OBJECT_HPP\n9 #define OBJECT_HPP\n10\n11 #include <string>\n12\n13 class Object {\n14 public:\n15\n16     // Exceptions constants\n17     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:\\Provided\\Ostream\\is\\bad";\n18     inline static const std::string ERROR_FAIL_WRITE = "ERROR:\\Fail\\to\\write\\on\\provided\\Ostream";\n19     inline static const std::string ERROR_NULLPTR = "ERROR:\\Passed\\in\\Nullptr!";\n20\n21     // once virtual always virtual\n22     virtual ~Object() = default;\n23\n24\n25 protected:\n26     Object() = default;\n27 };\n28\n29 #endif // !OBJECT_HPP
```

## 6.2 Symbolparser.hpp

```

1  /**************************************************************************/ 
2  * \file   SymbolParser.hpp
3  * \brief  A multi language parser for types and variables
4  * \author Simon
5  * \date   Dezember 2025
6  /**************************************************************************/
7
8 #ifndef SYMBOL_PARSER_HPP
9 #define SYMBOL_PARSER_HPP
10
11 #include <vector>
12
13 #include "Object.hpp"
14 #include "Variable.hpp"
15 #include "Type.hpp"
16 #include "ISymbolFactory.hpp"
17
18 class SymbolParser : public Object
19 {
20 public:
21     inline static const std::string ERROR_EMPTY_STRING = "ERROR: Provided string is empty.";
22     inline static const std::string ERROR_NONEXISTING_TYPE = "ERROR: Provided type does not exist.";
23
24     /**
25      * \brief Polymorphic container for saving variables
26      */
27     using TVariableCont = std::vector<Variable::Uptr>;
28
29     /**
30      * \brief Polymorphic container for saving types
31      */
32     using TTypeCont = std::vector<Type::Sptr>;
33
34     /**
35      * \brief Sets Factory for parsing a language
36      * \brief Previous variables and types of prior factory get saved,
37      * \brief then the subsequent factories variables and types get loaded.
38      * \param Reference to a SymbolFactory
39      * \return void
40      */
41     void SetFactory(ISymbolFactory& Factory);
42
43     /**
44      * \brief Adds a new type to the language
45      * \param string of typename
46      * \return void
47      */
48     void AddType(std::string const& name);
49
50     /**
51      * \brief Adds a new variable if type exists
52      * \param string of variable, string of type
53      * \return void
54      */
55     void AddVariable(std::string const& name, std::string const& type);
56
57     SymbolParser(ISymbolFactory & fact) : m_Factory{fact} {}
58
59 protected:
60 private:
61     /**
62      * \brief Saves the current state of a SymbolFactory to its file
63      * \param string of type files path, string of variable files path
64      * \return void
65      */
66     void SaveState(const std::string& type_file, const std::string& var_file);
67
68     /**
69      * \brief Loads a SymbolFactory's variables and types from file
70      * \param string of type files path, string of variable files path
71      * \return void
72      */

```

```
73     void LoadNewState(const std::string& type_file, const std::string& var_file);
74
75     TTypeCont m_typeCont;
76     TVariableCont m_variableCont;
77     ISymbolFactory & m_Factory;
78 };
79 #endif
```

## 6.3 Symbolparser.cpp

```
1 //*****\n2 * \file SymbolParser.cpp\n3 * \brief A multi language parser for types and variables\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8\n9 #include <algorithm>\n10 #include <fstream>\n11 #include <iostream>\n12 #include "SymbolParser.hpp"\n13 #include "ISymbolFactory.hpp"\n14\n15 using namespace std;\n16\n17 void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)\n18 {\n19     ofstream type_File;\n20     ofstream var_File;\n21\n22     type_File.open(m_Factory.GetTypeFileName());\n23\n24     for_each(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& type) { type_File << type->GetSaveLine(); });\n25\n26     type_File.close();\n27\n28\n29     var_File.open(m_Factory.GetVariableFileName());\n30\n31     for_each(m_variableCont.cbegin(), m_variableCont.cend(), [&](const auto& var) { var_File << var->GetSaveLine(); });\n32\n33     var_File.close();\n34 }\n35\n36 void SymbolParser::LoadNewState(const std::string& type_file, const std::string& var_file)\n37 {\n38     ifstream type_File;\n39     ifstream var_File;\n40\n41     m_typeCont.clear();\n42     m_variableCont.clear();\n43\n44     type_File.open(m_Factory.GetTypeFileName());\n45\n46     string line;\n47\n48     while (getline(type_File, line)) {\n49\n50         Type::Uptr pType = m_Factory.CreateType("");\n51\n52         pType->SetType(pType->LoadTypeName(line));\n53\n54         m_typeCont.push_back(move(pType));\n55     }\n56\n57     type_File.close();\n58\n59\n60     var_File.open(m_Factory.GetVariableFileName());\n61\n62     while (getline(var_File, line)) {\n63\n64         auto pVar = m_Factory.CreateVariable("");\n65\n66         const string type = pVar->LoadTypeName(line);\n67         const string name = pVar->LoadVarName(line);\n68\n69         pVar->SetName(name);\n70\n71         // look up if type even exists if yes add to type container\n72         for (const auto& m_type : m_typeCont)
```

```
73     {
74         if (type == m_type->GetType())
75         {
76             pVar->SetType(m_type);
77
78             // If each variable should only match one type, break early
79             break;
80         }
81     }
82 }
83
84 if (pVar->GetType() != "") {
85     m_variableCont.push_back(std::move(pVar));
86 }
87 }
88 var_File.close();
89 }
90 }
91
92 void SymbolParser::SetFactory(ISymbolFactory& Factory)
93 {
94
95     SaveState(m_Factory.GetTypeFileName(), m_Factory.GetVariableFileName());
96
97     m_Factory = Factory;
98
99     LoadNewState(m_Factory.GetTypeFileName(), m_Factory.GetVariableFileName());
100 }
101
102 void SymbolParser::AddType(std::string const& name)
103 {
104     if (name.empty())
105         throw SymbolParser::ERROR_EMPTY_STRING;
106
107     Type::Uptr pType = m_Factory.CreateType(name);
108     m_typeCont.push_back(std::move(pType));
109 }
110
111 void SymbolParser::AddVariable(std::string const& name, std::string const& type)
112 {
113     if (name.empty())
114         throw SymbolParser::ERROR_EMPTY_STRING;
115
116     if (type.empty())
117         throw SymbolParser::ERROR_EMPTY_STRING;
118
119     // look up if type even exists if yes add to type container
120     for (const auto& m_type : m_typeCont)
121     {
122         if (type == m_type->GetType())
123         {
124             auto pVar = m_Factory.CreateVariable(name);
125             pVar->SetType(m_type);
126
127             // Move ownership into container
128             m_variableCont.push_back(std::move(pVar));
129
130             // If each variable should only match one type, return early
131             return;
132         }
133     }
134 }
135
136 throw ERROR_NONEXISTING_TYPE;
137 }
```

## 6.4 ISymbolFactory.hpp

```

1 /*****  

2 * \file ISymbolFactory.hpp  

3 * \brief A Interface for creating SymbolFactories  

4 * \author Simon  

5 * \date Dezember 2025  

6 *****  

7 #ifndef ISYMBOL_FACTORY_HPP  

8 #define ISYMBOL_FACTORY_HPP  

9  

10 #include "Variable.hpp"  

11 #include "Type.hpp"  

12  

13 class ISymbolFactory  

14 {  

15 public:  

16     /**  

17      * \brief Creates a variable  

18      *  

19      * \param string of variables name  

20      * \return unique pointer to variable  

21      */  

22     virtual Variable::Uptr CreateVariable(const std::string& name)=0;  

23  

24     /**  

25      * \brief Creates a type  

26      *  

27      * \param string of typename  

28      * \return unique pointer to type  

29      */  

30     virtual Type::Uptr CreateType(const std::string& name)=0;  

31  

32     /**  

33      * \brief Getter for file path of type file  

34      *  

35      * \return string of filePath  

36      */  

37     virtual const std::string& GetTypeFileName()=0;  

38  

39     /**  

40      * \brief Getter for file path of variable file  

41      *  

42      * \return string of filePath  

43      */  

44     virtual const std::string& GetVariableFileName()=0;  

45  

46 protected:  

47 private:  

48 };  

#endif

```

## 6.5 Variable.hpp

```

1 /*****
2 * \file Variable.hpp
3 * \brief Abstract class for parsing types
4 * \author Simon Vogelhuber
5 * \date Dezember 2025
6 *****/
7
8 #ifndef VARIABLE_HPP
9 #define VARIABLE_HPP
10 #include <memory>
11 #include <vector>
12 #include <string>
13
14 #include "Object.h"
15 #include "Type.hpp"
16
17 class Variable: public Object
18 {
19 public:
20     /**
21      * \brief Unique pointer type for variable
22      */
23     using Uptr = std::unique_ptr<Variable>;
24
25
26     inline static const std::string ERROR_EMPTY_STRING = "ERROR:_Empty_String";
27
28 /**
29  * \brief Returns formatted line of a variables declaration
30  *
31  * \return string of variable
32  */
33     virtual std::string GetSaveLine() const = 0;
34
35 /**
36  * \brief Loads the name of a variables type
37  *
38  * \param string fileLine
39  * \return string of type - SymbolParser has to check type for validity
40  */
41     virtual std::string LoadTypeName(std::string const& fileLine) const = 0;
42
43 /**
44  * \brief Loads name of a variable
45  *
46  * \param string fileLine
47  * \return string of variables name
48  */
49     virtual std::string LoadVarName(std::string const& fileLine) const = 0;
50
51 /**
52  * \brief Sets the type of a variable
53  *
54  * \param shared pointer of type
55  * \return void
56  */
57     void SetType(Type::Sptr type);
58
59 /**
60  * \brief Retuns the name of a variable
61  *
62  * \return string name
63  */
64     std::string GetName() const;
65
66 /**
67  * \brief Name getter
68  *
69  * \return string of variable
70  */
71     std::string GetType() const;
72

```

```
73     /**
74      * \brief Sets name of variable
75      *
76      * \return void
77      */
78     void SetName(const std::string & name);
79
80 protected:
81     Variable(const std::string& name) : m_name{ name } {}
82     Variable() = default;
83     std::string m_name;
84     Type::Sptr m_type;
85 private:
86 };
87 #endif
```

## 6.6 Variable.cpp

```
1 //*****  
2 * \file Variable.cpp  
3 * \brief Abstract class for parsing types  
4 * \author Simon Vogelhuber  
5 * \date Dezember 2025  
6 *****  
7  
8 #include "Variable.hpp"  
9 #include <cassert>  
10  
11 using namespace std;  
12  
13  
14 void Variable::SetType(Type::Sptr type)  
15 {  
16     if (type == nullptr) throw Type::ERROR_NULLPTR;  
17     m_type = std::move(type);  
18 }  
19  
20 std::string Variable::GetName() const  
21 {  
22     return m_name;  
23 }  
24  
25 std::string Variable::GetType() const  
26 {  
27     return m_type->GetType();  
28 }  
29  
30 void Variable::SetName(const std::string& name)  
31 {  
32     if (name.empty()) throw Variable::ERROR_EMPTY_STRING;  
33     m_name = name;  
34 }  
35  
36 }  
37 }
```

## 6.7 Type.hpp

```

1 /***** //**
2 * \file Type.hpp
3 * \brief Abstract class for parsing types
4 * \author Simon Vogelhuber
5 * \date Dezember 2025
6 *****/
7
8 #ifndef TYPE_HPP
9 #define TYPE_HPP
10 #include <memory>
11 #include <string>
12 #include "Object.h"
13
14 class Type : public Object
15 {
16 public:
17
18     inline static const std::string ERROR_EMPTY_STRING = "ERROR:_Empty_String";
19
20     /**
21      * \brief Unique pointer type for type
22      */
23     using Uptr = std::unique_ptr<Type>;
24
25     /**
26      * \brief Shared pointer type for type
27      */
28     using Sptr = std::shared_ptr<Type>;
29
30     /**
31      * \brief Getter for type name
32      *
33      * \return string of type
34      */
35     std::string GetType() const;
36
37     /**
38      * \brief Sets a types name
39      *
40      * \param string fileLine
41      * \return string of type - SymbolParser has to check type for validity
42      */
43     void SetType(const std::string& name);
44
45     /**
46      * \brief Loads a types name from a files line
47      *
48      * \param string fileLine
49      * \return string of type
50      */
51     virtual std::string LoadTypeName(const std::string& fileLine) const = 0;
52
53     /**
54      * \brief Returns formatted line of a types declaration
55      *
56      * \return string of type declaration
57      */
58     virtual std::string GetSaveLine() const = 0;
59
60     Type(const std::string& name) : m_name(name) {}
61     Type() = default;
62
63 protected:
64     std::string m_name;
65 private:
66 };
67#endif

```

## 6.8 Type.cpp

```
1 //*****  
2 * \file  Type.cpp  
3 * \brief Abstract class for parsing types  
4 * \author Simon Vogelhuber  
5 * \date  Dezember 2025  
6 *****  
7  
8 #include "Type.hpp"  
9  
10 std::string Type::GetType() const{  
11     return m_name;  
12 }  
13  
14 void Type::SetType(const std::string& name)  
15 {  
16     if (name.empty()) throw Type::ERROR_EMPTY_STRING;  
17     m_name = name;  
18 }  
19 }
```

## 6.9 Test.hpp

```

1 /***** //**
2 * \file Test.hpp
3 * \brief File that provides a Test Function with a formated output
4 *
5 * \author Simon
6 * \date April 2025
7 *****/
8 #ifndef TEST_HPP
9 #define TEST_HPP
10
11 #include <string>
12 #include <iostream>
13 #include <vector>
14 #include <list>
15 #include <queue>
16 #include <forward_list>
17
18 #define ON 1
19 #define OFF 0
20 #define COLOR_OUTPUT OFF
21
22 // Definitions of colors in order to change the color of the output stream
23 const std::string colorRed = "\x1B[31m";
24 const std::string colorGreen = "\x1B[32m";
25 const std::string colorWhite = "\x1B[37m";
26
27 inline std::ostream& RED(std::ostream& ost) {
28     if (ost.good()) {
29         ost << colorRed;
30     }
31     return ost;
32 }
33 inline std::ostream& GREEN(std::ostream& ost) {
34     if (ost.good()) {
35         ost << colorGreen;
36     }
37     return ost;
38 }
39 inline std::ostream& WHITE(std::ostream& ost) {
40     if (ost.good()) {
41         ost << colorWhite;
42     }
43     return ost;
44 }
45
46 inline std::ostream& TestStart(std::ostream& ost) {
47     if (ost.good()) {
48         ost << std::endl;
49         ost << "*****TESTCASE_START*****" << std::endl;
50         ost << "_____TESTCASE_START_____" << std::endl;
51         ost << "*****TESTCASE_START*****" << std::endl;
52         ost << std::endl;
53     }
54     return ost;
55 }
56
57 inline std::ostream& TestEnd(std::ostream& ost) {
58     if (ost.good()) {
59         ost << std::endl;
60         ost << "*****TESTCASE_END*****" << std::endl;
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67
68 #if COLOR_OUTPUT
69     if (ost.good()) {
70         ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;
71     }
72 #else

```

```
73     if (ost.good()) {
74         ost << "TEST_OK!!" << std::endl;
75     }
76 #endif // COLOR_OUTPUT
77
78     return ost;
79 }
80
81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83 #if COLOR_OUTPUT
84     if (ost.good()) {
85         ost << colorRed << "TEST_FAILED!!" << colorWhite << std::endl;
86     }
87 #else
88     if (ost.good()) {
89         ost << "TEST_FAILED!!" << std::endl;
90     }
91 #endif // COLOR_OUTPUT
92
93     return ost;
94 }
95
96 /**
97 * \brief function that reports if the testcase was successful.
98 *
99 * \param testcase String that indicates the testcase
100 * \param successful true -> reports to cout test OK
101 * \param successful false -> reports test failed
102 */
103 template <typename T>
104 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
105     if (ostr.good()) {
106 #if COLOR_OUTPUT
107         if (expected == result) {
108             ostr << testcase << std::endl << colorGreen << "[Test_OK]" << colorWhite << "Result:" << std::boolalpha << expected
109             << std::noboolalpha << std::endl << std::endl;
110         }
111         else {
112             ostr << testcase << std::endl << colorRed << "[Test_FAILED]" << colorWhite << "Result:" << std::boolalpha << expected
113             << std::noboolalpha << std::endl << std::endl;
114         }
115 #else
116         if (expected == result) {
117             ostr << testcase << std::endl << "[Test_OK]" << "Result:" << std::boolalpha << expected << "==" << "Result:" <<
118         }
119         else {
120             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:" << std::boolalpha << expected << "!=" << "Result:" <<
121         }
122 #endif
123         if (ostr.fail()) {
124             std::cerr << "Error: Write_Ostream" << std::endl;
125         }
126     }
127     else {
128         std::cerr << "Error: Bad_Ostream" << std::endl;
129     }
130     return expected == result;
131 }
132
133 template <typename T1, typename T2>
134 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
135     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
136     ost << "(" << p.first << "," << p.second << ")";
137     return ost;
138 }
139
140 template <typename T>
141 std::ostream& operator<< (std::ostream& ost, const std::vector<T> & cont) {
142     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
143     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
144     return ost;
145 }
```

```
146 template <typename T>
147 std::ostream& operator<< (std::ostream& ost, const std::list<T> & cont) {
148     if (!ost.good()) throw std::exception( "Error_bad_Ostream! " );
149     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
150     return ost;
151 }
152 }
153 template <typename T>
154 std::ostream& operator<< (std::ostream& ost, const std::deque<T> & cont) {
155     if (!ost.good()) throw std::exception( "Error_bad_Ostream! " );
156     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
157     return ost;
158 }
159 }
160 template <typename T>
161 std::ostream& operator<< (std::ostream& ost, const std::forward_list<T> & cont) {
162     if (!ost.good()) throw std::exception( "Error_bad_Ostream! " );
163     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
164     return ost;
165 }
166 }
167 }
168 
```

```
#endif // !TEST_HPP
```

## 6.10 SingletonBase.hpp

```
1 //*****\n2 * \file  SingletonBase.hpp\n3 * \brief  Base Class for creating singletons\n4 *\n5 * \author Simon\n6 * \date   November 2025\n7 *****\n8 #ifndef SINGLETON_BASE_HPP\n9 #define SINGLETON_BASE_HPP\n10\n11 #include "Object.h"\n12 #include <memory>\n13\n14 template <typename T> class SingletonBase : public Object {\n15 public:\n16     static T& GetInstance() {\n17         if (mInstance == nullptr) { mInstance = std::unique_ptr<T>{ new T{} } }; }\n18         return *mInstance;\n19     }\n20 protected:\n21     SingletonBase() = default;\n22     virtual ~SingletonBase() = default;\n23\n24 private:\n25     SingletonBase(SingletonBase const& s) = delete;\n26     SingletonBase& operator = (SingletonBase const& s) = delete;\n27     static std::unique_ptr<T> mInstance;\n28 };\n29\n30 template <typename T> std::unique_ptr<T> SingletonBase<T>::mInstance = nullptr;\n31\n32 #endif // !SINGLETON_BASE_HPP
```

## 6.11 JavaVariable.hpp

```
1 //*****\n2 * \file JavaVariable.hpp\n3 * \brief A Class for parsing java variables\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7 #ifndef JAVA_VARIABLE_HPP\n8 #define JAVA_VARIABLE_HPP\n9 #include "Object.h"\n10 #include "Variable.hpp"\n11\n12 class JavaVariable : public Variable\n13 {\n14 public:\n15     /**\n16      * \brief Returns formatted line of a variables declaration\n17      *\n18      * \return string of variable\n19      */\n20     virtual std::string GetSaveLine() const override;\n21\n22     /**\n23      * \brief Loads the name of a variables type\n24      *\n25      * \param string fileLine\n26      * \return string of type - SymbolParser has to check type for validity\n27      */\n28     virtual std::string LoadTypeName(std::string const& fileLine) const override;\n29\n30     /**\n31      * \brief Loads name of a variable\n32      *\n33      * \param string fileLine\n34      * \return string of variables name\n35      */\n36     virtual std::string LoadVarName(std::string const& fileLine) const override;\n37\n38     JavaVariable() = default;\n39\n40     JavaVariable(const std::string& name) : Variable{ name } {}\n41\n42 protected:\n43 private:\n44};\n45#endif
```

## 6.12 JavaVariable.cpp

```
1 //*****  
2 * \file JavaVariable.cpp  
3 * \brief A Class for parsing java variables  
4 * \author Simon  
5 * \date Dezember 2025  
6 *****  
7  
8 #include "JavaVariable.hpp"  
9 #include <sstream>  
10 #include <string>  
11 #include "scanner.h"  
12  
13 using namespace pfc;  
14 using namespace std;  
15  
16 static std::string ScanTypeName(scanner& scan)  
17 {  
18     string typeName = scan.get_identifier();  
19     scan.next_symbol();  
20     return typeName;  
21 }  
22  
23 static std::string ScanVarName(scanner& scan)  
24 {  
25     string varName;  
26     varName = scan.get_identifier();  
27  
28     return varName;  
29 }  
30  
31 std::string JavaVariable::GetSaveLine() const  
32 {  
33     stringstream line;  
34  
35     if (m_type != nullptr) {  
36         line << m_type->GetType() << "_" << m_name;  
37     }  
38  
39     return line.str();  
40 }  
41  
42 std::string JavaVariable::LoadTypeName(std::string const& fileLine) const  
43 {  
44     stringstream lineStream;  
45     lineStream << fileLine;  
46     scanner scan{lineStream};  
47  
48     return ScanTypeName(scan);  
49 }  
50  
51 std::string JavaVariable::LoadVarName(std::string const& fileLine) const  
52 {  
53     stringstream lineStream;  
54     lineStream << fileLine;  
55     scanner scan{ lineStream };  
56  
57     string typeName = ScanTypeName(scan);  
58     string varName = ScanVarName(scan);  
59     if (typeName.empty()) varName = "";  
60  
61     return varName;  
62 }
```

## 6.13 JavaSymbolFactory.hpp

```
1 //*****\n2 * \file JavaSymbolFactory.hpp\n3 * \brief A factory for parsing java variables and types\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8 #ifndef JAVA_SYMBOL_FACTORY_HPP\n9 #define JAVA_SYMBOL_FACTORY_HPP\n10 #include "Object.h"\n11 #include "ISymbolFactory.hpp"\n12 #include "SingletonBase.hpp"\n13\n14 class JavaSymbolFactory : public ISymbolFactory, public SingletonBase<JavaSymbolFactory>\n15 {\n16 public:\n17\n18     friend class SingletonBase<JavaSymbolFactory>;\n19\n20     /**\n21      * \brief Creates a java variable\n22      *\n23      * \param string of variables name\n24      * \return unique pointer to variable\n25      */\n26     virtual Variable::Uptr CreateVariable(const std::string& name) override;\n27\n28     /**\n29      * \brief Creates a java type\n30      *\n31      * \param string of typename\n32      * \return unique pointer to type\n33      */\n34     virtual Type::Uptr CreateType(const std::string& name) override;\n35\n36     /**\n37      * \brief Getter for file path of type file\n38      *\n39      * \return string of filePath\n40      */\n41     virtual const std::string& GetTypeFileName() override;\n42\n43     /**\n44      * \brief Getter for file path of variable file\n45      *\n46      * \return string of filePath\n47      */\n48     virtual const std::string& GetVariableFileName() override;\n49\n50 protected:\n51 private:\n52     JavaSymbolFactory() = default;\n53     const std::string m_TypeFileName = "IECTypes.sym";\n54     const std::string m_VariableFileName = "IECVars.sym";\n55 };\n#endif
```

## 6.14 JavaSymbolFactory.cpp

```
1 //*****\n2 * \file JavaSymbolFactory.cpp\n3 * \brief A factory for parsing java variables and types\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8\n9 #include "JavaSymbolFactory.hpp"\n10 #include "JavaType.hpp"\n11 #include "JavaVariable.hpp"\n12\n13\n14 Variable::Uptr JavaSymbolFactory::CreateVariable(const std::string& name)\n15 {\n16     return std::make_unique<JavaVariable>(JavaVariable{ name });\n17 }\n18\n19 Type::Uptr JavaSymbolFactory::CreateType(const std::string& name)\n20 {\n21     return std::make_unique<JavaType>(JavaType{name});\n22 }\n23\n24 const std::string& JavaSymbolFactory::GetTypeFileName()\n25 {\n26     return m_TypeFileName;\n27 }\n28\n29 const std::string& JavaSymbolFactory::GetVariableFileName()\n30 {\n31     return m_VariableFileName;\n32 }
```

## 6.15 IECVariable.hpp

```
1 //*****  
2 * \file IECVariable.hpp  
3 * \brief A Class for parsing IEC variables  
4 * \author Simon Vogelhuber  
5 * \date Dezember 2025  
6 *****  
7  
8 #ifndef IEC_VARIABLE_HPP  
9 #define IEC_VARIABLE_HPP  
10  
11 #include "Variable.hpp"  
12  
13 class IECVariable : public Variable  
14 {  
15 public:  
16     virtual std::string GetSaveLine() const override;  
17  
18     /**  
19      * \brief Loads the name of a variables type  
20      *  
21      * \param string fileLine  
22      * \return string of type - SymbolParser has to check type for validity  
23      */  
24     virtual std::string LoadTypeName(std::string const& fileLine) const override;  
25  
26     /**  
27      * \brief Loads name of a variable  
28      *  
29      * \param string fileLine  
30      * \return string of variables name  
31      */  
32     virtual std::string LoadVarName(std::string const& fileLine) const override;  
33  
34     IECVariable() = default;  
35  
36     IECVariable(const std::string& name) : Variable( name ) {}  
37  
38 protected:  
39 private:  
40 };  
41 #endif
```

## 6.16 IECVariable.cpp

```
1 //*****  
2 * \file IECVariable.cpp  
3 * \brief A Class for parsing IEC variables  
4 * \author Simon Vogelhuber  
5 * \date Dezember 2025  
6 *****  
7  
8 #include "IECVariable.hpp"  
9 #include <iostream>  
10 #include <string>  
11 #include <iostream>  
12 #include "scanner.h"  
13  
14 using namespace pfc;  
15 using namespace std;  
16  
17 std::string IECVariable::GetSaveLine() const  
18 {  
19     stringstream line;  
20  
21     if (m_type != nullptr) {  
22         line << "VAR" << m_type->GetType() << ":" << m_name << ";"  
23     }  
24  
25     return line.str();  
26 }  
27  
28 static std::string ScanTypeName(scanner & scan) {  
29     string TypeName;  
30  
31     if (scan.get_identifier() == "VAR") {  
32         scan.next_symbol();  
33         TypeName = scan.get_identifier();  
34         scan.next_symbol();  
35         return TypeName;  
36     }  
37  
38     return "";  
39 }  
40  
41 static std::string ScanVarName(scanner & scan) {  
42     string VarName;  
43  
44     if (scan.is(':')) {  
45         scan.next_symbol();  
46         VarName = scan.get_identifier();  
47         scan.next_symbol();  
48         if (!scan.is(';')) {  
49             VarName = "";  
50         }  
51     }  
52  
53     return VarName;  
54 }  
55  
56  
57 std::string IECVariable::LoadTypeName(std::string const& fileLine) const  
58 {  
59     stringstream converter;  
60     converter << fileLine;  
61     scanner Scan;  
62  
63     Scan.set_istream(converter);  
64  
65     return ScanTypeName(Scan);  
66 }  
67  
68 std::string IECVariable::LoadVarName(std::string const& fileLine) const  
69 {  
70     stringstream converter;  
71     converter << fileLine;  
72     scanner Scan;
```

```
73     Scan.set_istream(converter);
74
75     string Typename = ScanTypeName(Scan);
76     string VarName = ScanVarName(Scan);
77
78     if (Typename.empty()) VarName = "";
79
80     return VarName;
81
82 }
```

## 6.17 IECSymbolFactory.hpp

```
1 //*****  
2 * \file IECSymbolFactory.hpp  
3 * \brief A factory for parsing IEC variables and types  
4 * \author Simon  
5 * \date Dezember 2025  
6 *****  
7  
8 #ifndef IEC_SYMBOL_FACTORY_HPP  
9 #define IEC_SYMBOL_FACTORY_HPP  
10 #include "Object.h"  
11 #include "ISymbolFactory.hpp"  
12 #include "SingletonBase.hpp"  
13  
14 class IECSymbolFactory:public ISymbolFactory , public SingletonBase<IECSymbolFactory>  
15 {  
16 public:  
17  
18     friend class SingletonBase<IECSymbolFactory>;  
19  
20     /**  
21      * \brief Creates a IEC variable  
22      *  
23      * \param string of variables name  
24      * \return unique pointer to variable  
25      */  
26     virtual Variable::Uptr CreateVariable(const std::string& name) override;  
27  
28     /**  
29      * \brief Creates a IEC type  
30      *  
31      * \param string of typename  
32      * \return unique pointer to type  
33      */  
34     virtual Type::Uptr CreateType(const std::string& name) override;  
35  
36     /**  
37      * \brief Getter for file path of type file  
38      *  
39      * \return string of filePath  
40      */  
41     virtual const std::string& GetTypeFileName() override;  
42  
43     /**  
44      * \brief Getter for file path of variable file  
45      *  
46      * \return string of filePath  
47      */  
48     virtual const std::string& GetVariableFileName() override;  
49  
50 protected:  
51 private:  
52     IECSymbolFactory() = default;  
53     const std::string m_TypeFileName = "IECTypes.sym";  
54     const std::string m_VariableFileName = "IECVars.sym";  
55 };  
56  
57 #endif
```

## 6.18 IECSymbolFactory.cpp

```
1 //*****  
2 * \file IECSymbolFactory.cpp  
3 * \brief A factory for parsing IEC variables and types  
4 * \author Simon  
5 * \date Dezember 2025  
6 *****  
7  
8 #include "IECSymbolFactory.hpp"  
9 #include "IECType.hpp"  
10 #include "IECVariable.hpp"  
11  
12 Variable::Uptr IECSymbolFactory::CreateVariable(const std::string& name)  
13 {  
14     return std::make_unique<IECVariable>(IECVariable{name});  
15 }  
16  
17 Type::Uptr IECSymbolFactory::CreateType(const std::string& name)  
18 {  
19     return std::make_unique<IECType>(IECType{name});  
20 }  
21  
22 const std::string& IECSymbolFactory::GetTypeFileName()  
23 {  
24     return m_TypeFileName;  
25 }  
26  
27 const std::string& IECSymbolFactory::GetVariableFileName()  
28 {  
29     return m_VariableFileName;  
30 }  
31 }
```