



**HSD**

---

**FH-HAGENBERG**

# **Systemdokumentation Projekt Symbolparser**

**Version 1.0**

S. Offenberger, S. Vogelhuber

Hagenberg, 12. November 2025

# Inhaltsverzeichnis

<b>1</b>	<b>Organisatorisches</b>	<b>3</b>
1.1	Team . . . . .	3
1.2	Aufteilung der Verantwortlichkeitsbereiche . . . . .	3
1.3	Aufwand . . . . .	4
<b>2</b>	<b>Anforderungsdefinition (Systemspezifikation)</b>	<b>5</b>
<b>3</b>	<b>Systementwurf</b>	<b>8</b>
3.1	Klassendiagramm . . . . .	8
3.2	Designentscheidungen . . . . .	9
<b>4</b>	<b>Dokumentation der Komponenten (Klassen)</b>	<b>10</b>
<b>5</b>	<b>Testprotokollierung</b>	<b>11</b>
<b>6</b>	<b>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	SingetonBase.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 Variablennamens
- Speichern des Variablentyps
- Auswerten der Variablendeklaration (Syntaxprüfung)
- Zurückgeben des Variablennamens
- Zurückgeben des Variablentyps

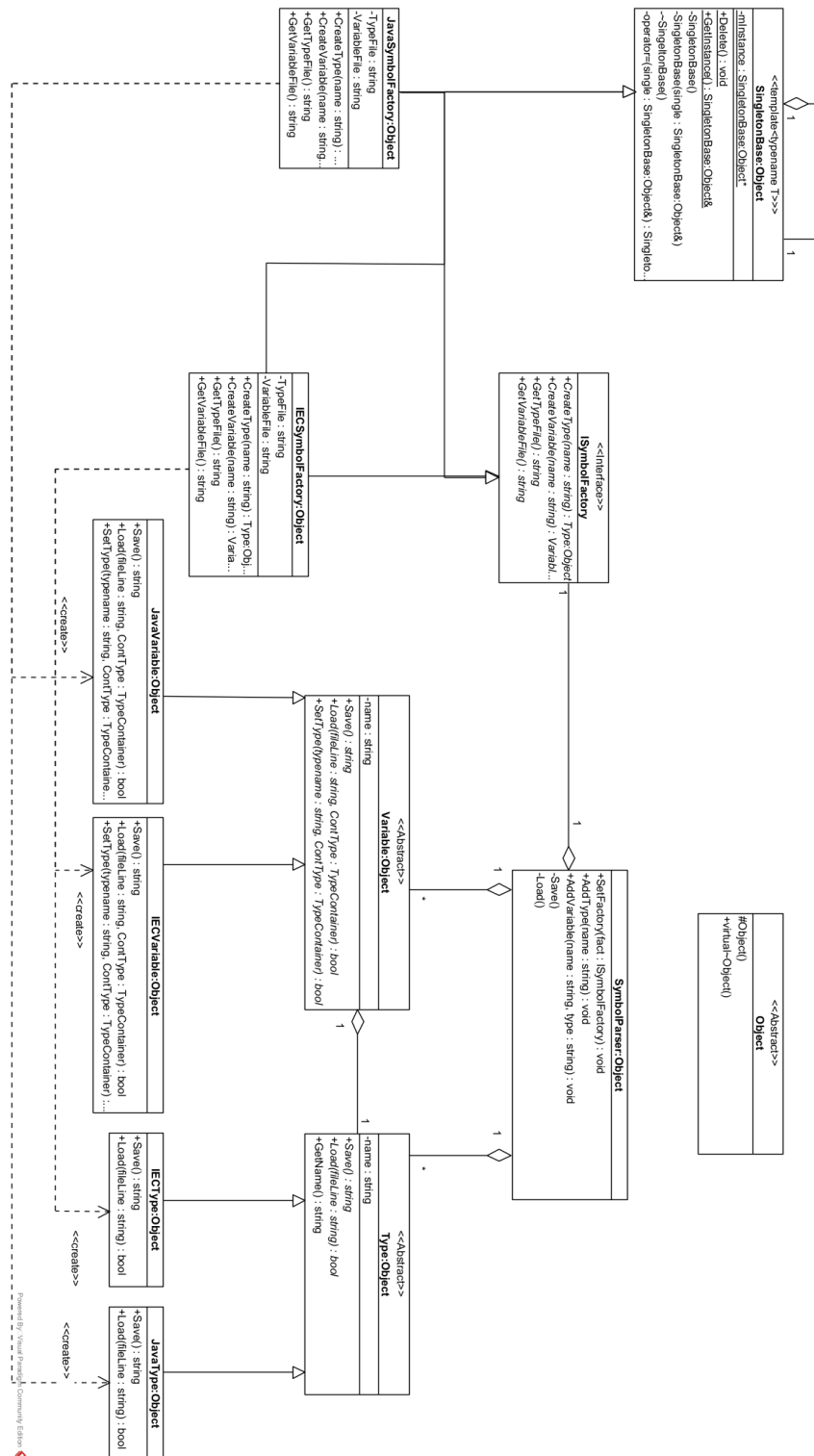
### Funktionen des Type:

- 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](http://doxy/html/index.html)

## 5 Testprotokollierung

```
1
2 *****
3 TESTCASE START
4 *****
5
6
7
8 **** Test IEC Var Getter ****
9
10
11 *****
12 TESTCASE START
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_fast256_t == Result: uint_fast256_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 *****
44
45
46
47 **** Test Java Var Getter ****
48
49
50 *****
51         TESTCASE START
52 *****
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_fast256_t == Result: uint_fast256_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 *****
90 TESTCASE START
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
    ↪ String)
101
102
103 *****
104
105
106
107 **** Test Java Type Getter ****
108
109
110 *****
111 TESTCASE START
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
    ↪ String)
122
123
124 *****
```

```
125
126
127 *****
128 TESTCASE START
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
170 Test Load Var Name IEC Var invalid Format
171 [Test OK] Result: (Expected:  == Result: )
172
173 Test Save LineFormat IEC Variable
174 [Test OK] Result: (Expected: VAR mCont : SpeedController; == Result: VAR
    ↪ mCont : SpeedController;)
175
176 Test Save LineFormat IEC Variable
177 [Test OK] Result: (Expected:  == Result: )
178
179
180 *****
181
182
183 *****
184 TESTCASE START
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: S2speedController == Result: S2speedController)
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 *****
302         TESTCASE START
303 *****
304
305 Normal Operating Parser
306 [Test OK] Result: (Expected: true == Result: true)
307
308 .AddType() - add empty type to factory
309 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
    ↪ ERROR: Provided string is empty.)
310
311 .AddVariable() - add empty type to factory
312 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
    ↪ ERROR: Provided string is empty.)
313
314 .AddVariable() - add empty var to factory
315 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
    ↪ ERROR: Provided string is empty.)
316
317 .AddVariable() - add variable with nonexisting type
318 [Test OK] Result: (Expected: ERROR: Provided type does not exist. == Result
    ↪ : ERROR: Provided type does not exist.)
319
320
321 *****
322
323 TEST OK!!
```

## 6 Quellcode

### 6.1 Object.hpp

```
1  /*****  
2  * \file   Object.hpp  
3  * \brief  common ancestor for all objects  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef OBJECT_HPP  
9  #define OBJECT_HPP  
10  
11 #include <string>  
12  
13 class Object {  
14 public:  
15  
16     // Exceptions constants  
17     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";  
18     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";  
19     inline static const std::string ERROR_NULLPTR = "ERROR:_Passed_in_Nullptr!";  
20  
21     // once virtual always virtual  
22     virtual ~Object() = default;  
23  
24 protected:  
25     Object() = default;  
26 };  
27  
28 #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.h"
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  /*****  
2  * \file    SymbolParser.cpp  
3  * \brief   A multi language parser for types and variables  
4  * \author  Simon  
5  * \date   Dezember 2025  
6  *****/  
7  
8  
9  #include <algorithm>  
10 #include <fstream>  
11 #include <iostream>  
12 #include "SymbolParser.hpp"  
13 #include "ISymbolFactory.hpp"  
14  
15 using namespace std;  
16  
17 void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)  
18 {  
19     ofstream type_File;  
20     ofstream var_File;  
21  
22     type_File.open(m_Factory.GetTypeFileName());  
23  
24     for_each(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& type) { type_File << type->GetSaveLine(); });  
25  
26     type_File.close();  
27  
28     var_File.open(m_Factory.GetVariableFileName());  
29  
30     for_each(m_variableCont.cbegin(), m_variableCont.cend(), [&](const auto& var) { var_File << var->GetSaveLine(); });  
31  
32     var_File.close();  
33 }  
34  
35 void SymbolParser::LoadNewState(const std::string& type_file, const std::string& var_file)  
36 {  
37     ifstream type_File;  
38     ifstream var_File;  
39  
40     m_typeCont.clear();  
41     m_variableCont.clear();  
42  
43     type_File.open(m_Factory.GetTypeFileName());  
44  
45     string line;  
46  
47     while (getline(type_File, line)) {  
48  
49         Type::Uptr pType = m_Factory.CreateType("");  
50  
51         pType->SetType(pType->LoadTypeName(line));  
52  
53         m_typeCont.push_back(move(pType));  
54     }  
55  
56     type_File.close();  
57  
58     var_File.open(m_Factory.GetVariableFileName());  
59  
60     while (getline(var_File, line)) {  
61  
62         auto pVar = m_Factory.CreateVariable("");  
63  
64         const string type = pVar->LoadTypeName(line);  
65         const string name = pVar->LoadVarName(line);  
66  
67         pVar->SetName(name);  
68  
69         // look up if type even exists if yes add to type container  
70         for (const auto& m_type : m_typeCont)  
71             if (m_type->LoadTypeName(line) == name)  
72                 m_variableCont.push_back(pVar);  
73     }
```

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

## 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 };  
49 #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 Returns 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  
12  using namespace std;  
13  
14  
15  void Variable::SetType(Type::Sptr type)  
16  {  
17      if (type == nullptr) throw Type::ERROR_NULLPTR;  
18  
19      m_type = std::move(type);  
20  }  
21  
22  std::string Variable::GetName() const  
23  {  
24      return m_name;  
25  }  
26  
27  std::string Variable::GetType() const  
28  {  
29      return m_type->GetType();  
30  }  
31  
32  void Variable::SetName(const std::string& name)  
33  {  
34      if (name.empty()) throw Variable::ERROR_EMPTY_STRING;  
35  
36      m_name = name;  
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      * \return string of type  
33      */  
34     std::string GetType() const;  
35  
36     /**  
37      * \brief Sets a types name  
38      * \param string fileLine  
39      * \return string of type - SymbolParser has to check type for validity  
40      */  
41     void SetType(const std::string& name);  
42  
43     /**  
44      * \brief Loads a types name from a files line  
45      * \param string fileLine  
46      * \return string of type  
47      */  
48     virtual std::string LoadTypeName(const std::string& fileLine) const = 0;  
49  
50     /**  
51      * \brief Returns formatted line of a types declaration  
52      * \return string of type declaration  
53      */  
54     virtual std::string GetSaveLine() const = 0;  
55  
56     Type(const std::string& name) : m_name{ name } {}  
57     Type() = default;  
58  
59 protected:  
60     std::string m_name;  
61 private:  
62 };  
63 #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  
11 std::string Type::GetType() const{  
12     return m_name;  
13 }  
14  
15 void Type::SetType(const std::string& name)  
16 {  
17     if (name.empty()) throw Type::ERROR_EMPTY_STRING;  
18  
19     m_name = name;  
20 }
```

## 6.9 Test.hpp

```
1  /*****  
2  * \file   Test.hpp  
3  * \brief  File that provides a Test Function with a formatted 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 << "*****" << std::endl;  
50         ost << "TESTCASE_START" << std::endl;  
51         ost << "*****" << 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 << "*****" << 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:_(Expected:_" << std::boolalpha << expected << std::noboolalpha << std::endl << std::endl;
109         }
110         else {
111             ostr << testcase << std::endl << colorRed << "[Test_FAILED]" << colorWhite << "Result:_(Expected:_" << std::boolalpha << expected << std::noboolalpha << std::endl << std::endl;
112         }
113 #else
114         if (expected == result) {
115             ostr << testcase << std::endl << "[Test_OK]" << "Result:_(Expected:_" << std::boolalpha << expected << "_" << "==" << "Result:_" << result << std::endl << std::endl;
116         }
117         else {
118             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:_(Expected:_" << std::boolalpha << expected << "_" << "!=" << "Result:_" << result << std::endl << std::endl;
119         }
120 #endif
121     }
122     if (ostr.fail()) {
123         std::cerr << "Error:_Write_Ostream" << std::endl;
124     }
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
147 template <typename T>
148 std::ostream& operator<< (std::ostream& ost, const std::list<T> & cont) {
149     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
150     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "_"));
151     return ost;
152 }
153
154 template <typename T>
155 std::ostream& operator<< (std::ostream& ost, const std::deque<T> & cont) {
156     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
157     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "_"));
158     return ost;
159 }
160
161 template <typename T>
162 std::ostream& operator<< (std::ostream& ost, const std::forward_list<T> & cont) {
163     if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
164     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "_"));
165     return ost;
166 }
167
168
169 #endif // !TEST_HPP
```



## 6.10 SingletonBase.hpp

```
1  /*****  
2  * \file   SingletonBase.hpp  
3  * \brief  Base Class for creating singletons  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef SINGLETON_BASE_HPP  
9  #define SINGLETON_BASE_HPP  
10  
11 #include "Object.h"  
12 #include <memory>  
13  
14 template <typename T> class SingletonBase : public Object {  
15 public:  
16     static T& GetInstance() {  
17         if (mInstance == nullptr) { mInstance = std::unique_ptr<T>{ new T{} }; };  
18         return *mInstance;  
19     }  
20 protected:  
21     SingletonBase() = default;  
22     virtual ~SingletonBase() = default;  
23  
24 private:  
25     SingletonBase(SingletonBase const& s) = delete;  
26     SingletonBase& operator = (SingletonBase const& s) = delete;  
27     static std::unique_ptr<T> mInstance;  
28 };  
29  
30 template <typename T> std::unique_ptr<T> SingletonBase<T>::mInstance = nullptr;  
31  
32 #endif // !SINGLETON_BASE_HPP
```

## 6.11 JavaVariable.hpp

```
1  /*****  
2  * \file   JavaVariable.hpp  
3  * \brief  A Class for parsing java variables  
4  * \author Simon  
5  * \date   Dezember 2025  
6  *****/  
7  #ifndef JAVA_VARIABLE_HPP  
8  #define JAVA_VARIABLE_HPP  
9  #include "Object.h"  
10 #include "Variable.hpp"  
11  
12 class JavaVariable :public Variable  
13 {  
14 public:  
15     /**  
16     * \brief Returns formatted line of a variables declaration  
17     *  
18     * \return string of variable  
19     */  
20     virtual std::string GetSaveLine() const override;  
21  
22     /**  
23     * \brief Loads the name of a variables type  
24     *  
25     * \param string fileLine  
26     * \return string of type - SymbolParser has to check type for validity  
27     */  
28     virtual std::string LoadTypeName(std::string const& fileLine) const override;  
29  
30     /**  
31     * \brief Loads name of a variable  
32     *  
33     * \param string fileLine  
34     * \return string of variables name  
35     */  
36     virtual std::string LoadVarName(std::string const& fileLine) const override;  
37  
38     JavaVariable() = default;  
39  
40     JavaVariable(const std::string& name) : Variable{ name } {}  
41  
42 protected:  
43 private:  
44 };  
45 #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  /*****  
2  * \file   JavaSymbolFactory.hpp  
3  * \brief  A factory for parsing java variables and types  
4  * \author Simon  
5  * \date   Dezember 2025  
6  *****/  
7  
8  #ifndef JAVA_SYMBOL_FACTORY_HPP  
9  #define JAVA_SYMBOL_FACTORY_HPP  
10 #include "Object.h"  
11 #include "ISymbolFactory.hpp"  
12 #include "SingletonBase.hpp"  
13  
14 class JavaSymbolFactory :public ISymbolFactory, public SingletonBase<JavaSymbolFactory>  
15 {  
16 public:  
17  
18     friend class SingletonBase<JavaSymbolFactory>;  
19  
20     /**  
21      * \brief Creates a java 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 java 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     JavaSymbolFactory() = default;  
53     const std::string m_TypeFileName = "IECTypes.sym";  
54     const std::string m_VariableFileName = "IECVars.sym";  
55 };  
56 #endif
```

## 6.14 JavaSymbolFactory.cpp

```
1  /*****  
2  * \file   JavaSymbolFactory.cpp  
3  * \brief  A factory for parsing java variables and types  
4  * \author Simon  
5  * \date   Dezember 2025  
6  *****/  
7  
8  
9  #include "JavaSymbolFactory.hpp"  
10 #include "JavaType.hpp"  
11 #include "JavaVariable.hpp"  
12  
13  
14 Variable::Uptr JavaSymbolFactory::CreateVariable(const std::string& name)  
15 {  
16     return std::make_unique<JavaVariable>(JavaVariable{ name });  
17 }  
18  
19 Type::Uptr JavaSymbolFactory::CreateType(const std::string& name)  
20 {  
21     return std::make_unique<JavaType>(JavaType{name});  
22 }  
23  
24 const std::string& JavaSymbolFactory::GetTypeFileName()  
25 {  
26     return m_TypeFileName;  
27 }  
28  
29 const std::string& JavaSymbolFactory::GetVariableFileName()  
30 {  
31     return m_VariableFileName;  
32 }
```

## 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  /***** IECVariable.cpp *****/
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 <sstream>
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 |
74 |     Scan.set_istream(converter);
75 |
76 |     string Typename = ScanTypeName(Scan);
77 |     string VarName = ScanVarName(Scan);
78 |
79 |     if (Typename.empty()) VarName = "";
80 |
81 |     return VarName;
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  
13 Variable::Uptr IECSymbolFactory::CreateVariable(const std::string& name)  
14 {  
15     return std::make_unique<IECVariable>(IECVariable{name});  
16 }  
17  
18 Type::Uptr IECSymbolFactory::CreateType(const std::string& name)  
19 {  
20     return std::make_unique<IECType>(IECType{name});  
21 }  
22  
23 const std::string& IECSymbolFactory::GetTypeFileName()  
24 {  
25     return m_TypeFileName;  
26 }  
27  
28 const std::string& IECSymbolFactory::GetVariableFileName()  
29 {  
30     return m_VariableFileName;  
31 }
```