
Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku

Mat.Nr: S2410306027 / S2410306014

Punkte:

Übungsgruppe: 1

korrigiert:

Beispiel 1 (24 Punkte) Symbolparser: Entwerfen Sie aus der nachfolgenden Spezifikation ein Klassendiagramm, instanzieren Sie dieses und implementieren Sie die Funktionalität entsprechend:

Ein Symbolparser soll Symbole (Typen und Variablen) für verschiedene Programmiersprachen (Java, IEC,...) erzeugen und verwalten können! Dazu soll folgende öffentliche Schnittstelle angeboten werden:

```
1 class SymbolParser : public Object
2 {
3     public:
4         ...
5         void AddType(std::string const& name);
6         void AddVariable(std::string const& name, std::string const& type);
7         void SetFactory(...);
8     protected:
9         ...
10    private:
11        ...
12 };
```

Sowohl Typen als auch Variablen haben einen Namen und können jeweils in eine fix festgelegte Textdatei geschrieben bzw. von dieser wieder gelesen werden:

- Dateien für Java: *JavaTypes.sym* und *JavaVars.sym*
- Dateien für IEC: *IECTypes.sym* und *IECVars.sym*

Die Einträge in den Dateien sollen in ihrer Struktur folgendermaßen aussehen:

JavaTypes.sym:

```
class Button  
class Hugo  
class Window  
...
```

JavaVars.sym:

```
Button mBut;  
Window mWin;  
...
```

IECTypes.sym:

```
TYPE SpeedController  
TYPE Hugo  
TYPE Nero  
...
```

IECVars.sym:

```
VAR mCont : SpeedController;  
VAR mHu : Hugo;  
...
```

Variablen speichern einen Verweis auf ihren zugehörigen Typ. Variablen können nur erzeugt werden, wenn deren Typ im Symbolparser bereits vorhanden ist, ansonsten ist auf der Konsole eine entsprechende Fehlermeldung auszugeben! Variablen und Typen dürfen im Symbolparser nicht doppelt vorkommen! Variablen mit unterschiedlichen Namen können den gleichen Typ haben!

Der Parser hält immer nur Variablen und Typen einer Programmiersprache. Das bedeutet bei einem Wechsel der Programmiersprache sind alle Variablen und Typen in ihre zugehörigen Dateien zu schreiben und aus dem Symbolparser zu entfernen. Anschließend sind die Typen und Variablen der neuen Programmiersprache, falls bereits Symboldateien vorhanden sind, entsprechend in den Parser einzulesen.

Verwenden Sie zur Erzeugung der Typen und Variablen das Design Pattern *Abstract Factory* und implementieren Sie den Symbolparser so, dass er mit verschiedenen Fabriken (Programmiersprachen) arbeiten kann. Stellen Sie weiters sicher, dass für die Fabriken jeweils nur ein Exemplar in der Anwendung möglich ist.

Eine mögliche Anwendung im Hauptprogramm könnte so aussehen:

```
1 #include "SymbolParser.h"  
2 #include "JavaSymbolFactory.h"  
3 #include "IECSymbolFactory.h"  
4  
5  
6 int main()  
7 {
```

```

8    SymbolParser parser;
9
10   parser.SetFactory(JavaSymbolFactory::GetInstance());
11   parser.AddType("Button");
12   parser.AddType("Hugo");
13   parser.AddType("Window");
14   parser.AddVariable("mButton", "Button");
15   parser.AddVariable("mWin", "Window");
16
17   parser.SetFactory(IECSymbolFactory::GetInstance());
18   parser.AddType("SpeedController");
19   parser.AddType("Hugo");
20   parser.AddType("Nero");
21   parser.AddVariable("mCont", "SpeedController");
22   parser.AddVariable("mHu", "Hugo");
23
24   parser.SetFactory(JavaSymbolFactory::GetInstance());
25   parser.AddVariable("b", "Button");
26
27   parser.SetFactory(IECSymbolFactory::GetInstance());
28   parser.AddType("Hugo");
29   parser.AddVariable("mCont", "Hugo");
30
31   return 0;
32 }
```

Achten Sie darauf, dass im Hauptprogramm nur der Symbolparser und die Fabriken zu inkludieren sind! Das Design sollte so gestaltet werden, dass für eine neue Programmiersprache (wieder nur mit Variablen u. Typen) der Symbolparser und alle Schnittstellen unverändert bleiben!

Treffen Sie für alle unzureichenden Angaben sinnvolle Annahmen und begründen Sie diese. Verfassen Sie weiters eine Systemdokumentation (entsprechend den Vorgaben aus Übung1)!

Allgemeine Hinweise: Legen Sie bei der Erstellung Ihrer Übung großen Wert auf eine **saubere Strukturierung** und auf eine **sorgfältige Ausarbeitung!** Dokumentieren Sie alle Schnittstellen und versehen Sie Ihre Algorithmen an entscheidenden Stellen ausführlich mit Kommentaren! Testen Sie ihre Implementierungen ausführlich! Geben Sie den **Testoutput** mit ab!



Systemdokumentation Projekt Symbolparser

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 14. November 2025

Inhaltsverzeichnis

1 Organisatorisches	7
1.1 Team	7
1.2 Aufteilung der Verantwortlichkeitsbereiche	7
1.3 Aufwand	8
2 Anforderungsdefinition (Systemspezifikation)	9
3 Systementwurf	11
3.1 Designentscheidungen	11
4 Dateibeschreibung	13
4.1 Datei: JavaTypes.sym	13
4.2 Datei: JavaVars.sym	13
4.3 Datei: IECTypes.sym	14
4.4 Datei: IECVars.sym	14
5 Dokumentation der Komponenten (Klassen)	15
6 Testprotokollierung	16
7 Quellcode	25
7.1 Object.hpp	25
7.2 Symbolparser.hpp	26
7.3 Symbolparser.cpp	28
7.4 ISymbolFactory.hpp	32
7.5 Identifier.hpp	33
7.6 Identifier.cpp	34
7.7 Variable.hpp	35
7.8 Variable.cpp	37
7.9 Type.hpp	38
7.10 SingletonBase.hpp	39
7.11 JavaType.hpp	40
7.12 JavaType.cpp	41
7.13 JavaVariable.hpp	42
7.14 JavaVariable.cpp	43

7.15 JavaSymbolFactory.hpp	45
7.16 JavaSymbolFactory.cpp	46
7.17 IECType.hpp	47
7.18 IECType.cpp	48
7.19 IECVariable.hpp	49
7.20 IECVariable.cpp	50
7.21 IECSymbolFactory.hpp	52
7.22 IECSymbolFactory.cpp	53
7.23 main.cpp	54
7.24 Client.hpp	63
7.25 Client.cpp	64
7.26 Test.hpp	68
7.27 scanner.h	71

1 Organisatorisches

1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: Simon.Offenberger@fh-hagenberg.at
- Simon Vogelhuber, Matr.-Nr.: S2410306014@fhooe.at, E-Mail: s2410306014@fhooe.at

1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
 - Design Klassendiagramm
 - Implementierung und Test der Klassen:
 - * Object
 - * Symbolparser
 - * ISymbolFactory
 - * Variable
 - * Type
 - * JavaVariable
 - * JavaType
 - * JavaSymbolFactory
 - * IECVariable
 - * IECType
 - * IECSymbolFactory

- Implementierung des Testtreibers
- Dokumentation
- 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 11 Ph
- Simon Vogelhuber: geschätzt 10 Ph / tatsächlich 7 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 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 einen Pointer auf die aktuelle SymbolFactory, die zur Laufzeit gewechselt werden kann.

Verwendung des Singleton-Patterns:

Das Singleton- Pattern wurde für die konkreten SymbolFactories 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.

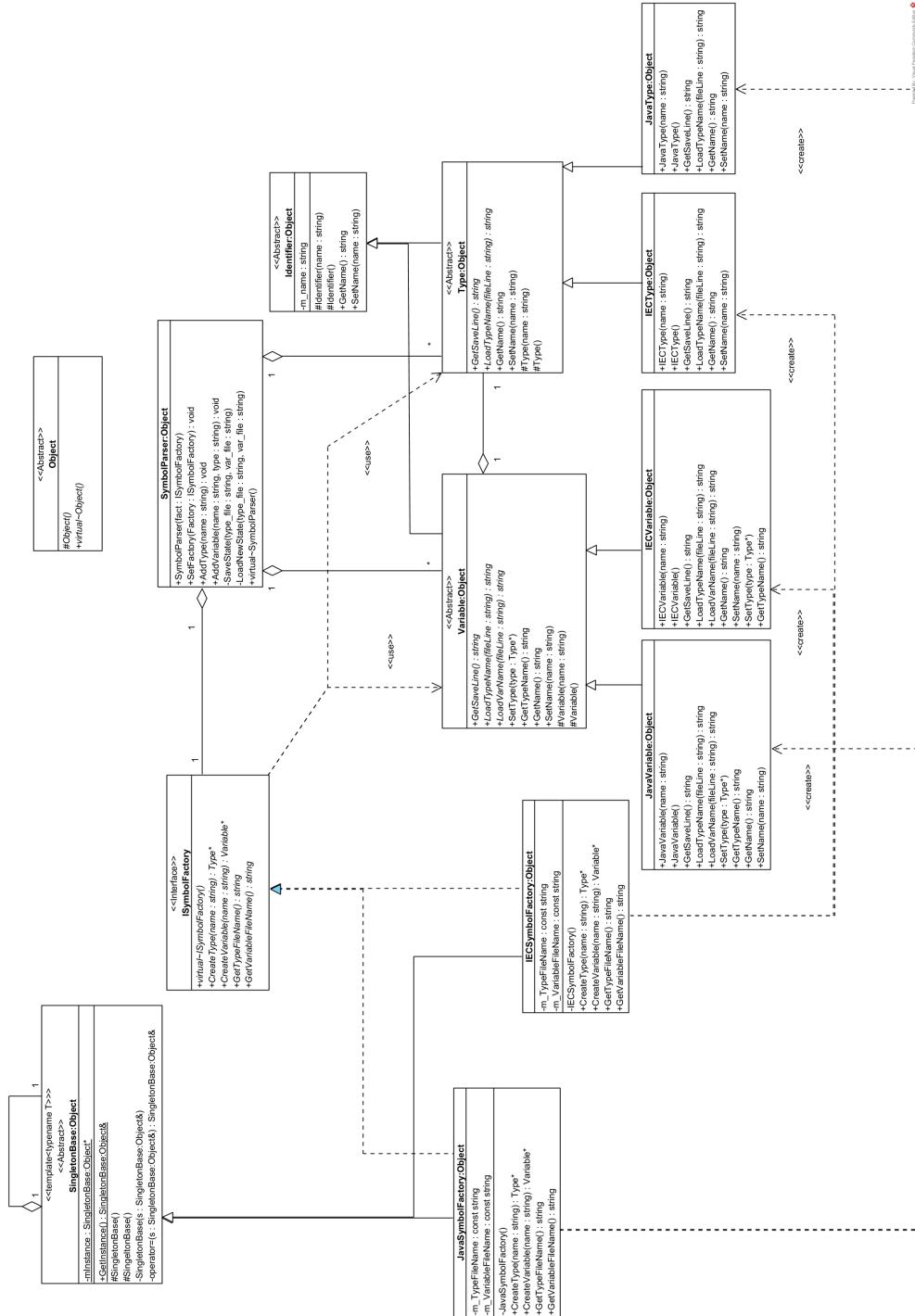
Container für Objekte:

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.

Klassendiagramm



4 Dateibeschreibung

Im folgenden Abschnitt werden die Formate der verwendeten Dateien beschrieben.

4.1 Datei: JavaTypes.sym

Diese Datei wird verwendet um die vom Symbolparser verwaltete JavaTypes zu speichern. Die Datei ist folgendermaßen aufgebaut:

```
ClassDeclaration = "class" Identifier "\n" .
Identifier        = Letter , { Letter | Digit | "_" } .
Letter            = "A"..."Z" | "a"..."z" .
Digit             = "0"..."9" .
```

4.2 Datei: JavaVars.sym

Diese Datei wird verwendet um die vom Symbolparser verwaltete JavaVariablen zu speichern. Die Datei ist folgendermaßen aufgebaut:

```
ClassDeclaration = Identifier Identifier ";"\n ;
Identifier        = Letter , { Letter | Digit | "_" } .
Letter            = "A"..."Z" | "a"..."z" .
Digit             = "0"..."9" .
```

4.3 Datei: IECTypes.sym

Diese Datei wird verwendet um die vom Symbolparser verwaltete IECTypes zu speichern. Die Datei ist folgendermaßen aufgebaut:

```
ClassDeclaration = "TYPE" Identifier "\n" .
Identifier       = Letter , { Letter | Digit | "_" } .
Letter           = "A"..."Z" | "a"..."z" .
Digit            = "0"..."9" .
```

4.4 Datei: IECVars.sym

Diese Datei wird verwendet um die vom Symbolparser verwaltete IEC Variablen zu speichern. Die Datei ist folgendermaßen aufgebaut:

```
ClassDeclaration = "VAR" Identifier ":" Identifier ";"\n .
Identifier       = Letter { Letter | Digit | "_" } .
Letter           = "A"..."Z" | "a"..."z" .
Digit            = "0"..."9" .
```

5 Dokumentation der Komponenten (Klassen)

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

6 Testprotokollierung

Der Testtreiber für die Symbolfactory wurde im Client implementiert, um zu zeigen dass dieser nur vom Interface und den Factories abhängt!

```
1
2 **** TESTCASE START ****
3
4 ****
5
6
7 **** Test IEC Var Getter ****
8
9
10 **** TESTCASE START ****
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_fast_256_t == Result:
23   ↪ uint_fast_256_t)
24
25 Check for Exception in Testcase
26 [Test OK] Result: (Expected: true == Result: true)
27
28 Test Exception in Set Name
29 [Test OK] Result: (Expected: ERROR: Empty String == Result:
30   ↪ ERROR: Empty String)
31
32 Test Exception in Set Type with nullptr
33 [Test OK] Result: (Expected: ERROR: Passed in Nullptr! ==
34   ↪ Result: ERROR: Passed in Nullptr!)
35
36 Test Variable Get Type after set with nullptr
37 [Test OK] Result: (Expected: int == Result: int)
```

```
36 | Test Variable Get Type after set
37 | [Test OK] Result: (Expected: uint_fast512_t == Result:
38 |   ↪ uint_fast512_t)
39 |
40 | Test for Exception in TestCase
41 | [Test OK] Result: (Expected: true == Result: true)
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_fast_256_t == Result:
62 |   ↪ uint_fast_256_t)
63 |
64 | Check for Exception in Testcase
65 | [Test OK] Result: (Expected: true == Result: true)
66 |
67 | Test Exception in Set Name
68 | [Test OK] Result: (Expected: ERROR: Empty String == Result:
69 |   ↪ ERROR: Empty String)
70 |
71 | Test Exception in Set Type with nullptr
72 | [Test OK] Result: (Expected: ERROR: Passed in Nullptr! ==
73 |   ↪ Result: ERROR: Passed in Nullptr!)
74 |
75 | Test Variable Get Type after set with nullptr
76 | [Test OK] Result: (Expected: int == Result: int)
77 |
78 | Test Variable Get Type after set
```

```
76 [Test OK] Result: (Expected: uint_fast512_t == Result:  
77     ↪ uint_fast512_t)  
78  
79 Test for Exception in TestCase  
80 [Test OK] Result: (Expected: true == Result: true)  
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:  
95     ↪ unit_1024_t)  
96  
97 Test Exception in Set Type  
98 [Test OK] Result: (Expected: true == Result: true)  
99  
100 Test Exception in Set Type  
101 [Test OK] Result: (Expected: ERROR: Empty String == Result:  
102     ↪ ERROR: Empty String)  
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:  
116     ↪ 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:
122     ↪ ERROR: Empty String)
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: Speed_Controller == Result:
136     ↪ Speed_Controller)
137
138 Test Load Type Name IEC Var invalid Format
139 [Test OK] Result: (Expected: == Result: )
140
141 Test Load Var Name IEC Var invalid Format
142 [Test OK] Result: (Expected: == Result: )
143
144 Test Load Type Name IEC Var invalid Format
145 [Test OK] Result: (Expected: mCont == Result: mCont)
146
147 Test Load Var Name IEC Var invalid Format
148 [Test OK] Result: (Expected: == Result: )
149
150 Test Load Type Name IEC Var invalid Format
151 [Test OK] Result: (Expected: == Result: )
152
153 Test Load Var Name IEC Var invalid Format
154 [Test OK] Result: (Expected: == Result: )
155
156 Test Load Type Name IEC Var invalid Format
157 [Test OK] Result: (Expected: mCont == Result: mCont)
```

```
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 : Speed_Controller;
175 == Result: VAR mCont : Speed_Controller;
176 )
177
178 Test Save LineFormat IEC Variable
179 [Test OK] Result: (Expected: == Result: )
180
181 Test for Exception in TestCase
182 [Test OK] Result: (Expected: true == Result: true)
183
184 ****
185 ****
186
187
188 ****
189 TESTCASE START
190 ****
191
192 Test Load Type Name Java Var
193 [Test OK] Result: (Expected: mCont == Result: mCont)
194
195 Test Load Var Name Java Var
196 [Test OK] Result: (Expected: mBut == Result: mBut)
197
198 Test Load Type Name Java Var invalid Format
199 [Test OK] Result: (Expected: == Result: )
200
201 Test Load Var Name Java Var invalid Format
```

```
202 [Test OK] Result: (Expected: == Result: )
203
204 Test Load Type Name Java Var invalid Format
205 [Test OK] Result: (Expected: mCont == Result: mCont)
206
207 Test Load Var Name Java Var invalid Format
208 [Test OK] Result: (Expected: == Result: )
209
210 Test Load Type Name Java Var invalid Format
211 [Test OK] Result: (Expected: == Result: )
212
213 Test Load Var Name Java Var invalid Format
214 [Test OK] Result: (Expected: == Result: )
215
216 Test Load Type Name Java Var invalid Format
217 [Test OK] Result: (Expected: mCont == Result: mCont)
218
219 Test Load Var Name Java Var invalid Format
220 [Test OK] Result: (Expected: == Result: )
221
222 Test Load Type Name Java Var invalid Format
223 [Test OK] Result: (Expected: == Result: )
224
225 Test Load Var Name Java Var invalid Format
226 [Test OK] Result: (Expected: == Result: )
227
228 Test Load Type Name Java Var invalid Format
229 [Test OK] Result: (Expected: == Result: )
230
231 Test Load Var Name Java Var invalid Format
232 [Test OK] Result: (Expected: == Result: )
233
234 Test Save LineFormat IEC Variable
235 [Test OK] Result: (Expected: mCont mBut;
236 == Result: mCont mBut;
237 )
238
239 Test Save LineFormat IEC Variable
240 [Test OK] Result: (Expected: == Result: )
241
242 Test for Exception in TestCase
243 [Test OK] Result: (Expected: true == Result: true)
244
245
```

```
246 *****  
247  
248 *****  
249 ***** TESTCASE START  
250 *****  
251 *****  
252 Test Load Type Name IEC Type  
253 [Test OK] Result: (Expected: SpeedController == Result:  
254     ↪ SpeedController)  
255  
256 Test Load Type Name IEC Type invalid Format  
257 [Test OK] Result: (Expected: == Result: )  
258  
259 Test Load Type Name IEC Type invalid Format  
260 [Test OK] Result: (Expected: == Result: )  
261  
262 Test Load Type Name IEC Type invalid Format  
263 [Test OK] Result: (Expected: S2peedController == Result:  
264     ↪ S2peedController)  
265  
266 Test Load Type Name IEC Type invalid Format  
267 [Test OK] Result: (Expected: == Result: )  
268  
269 Test Load Type Name IEC Type invalid Format  
270 [Test OK] Result: (Expected: == Result: )  
271  
272 Test Save LineFormat IEC Type  
273 [Test OK] Result: (Expected: TYPE SpeedController  
274     == Result: TYPE SpeedController  
275 )  
276  
277 Test for Exception in TestCase  
278 [Test OK] Result: (Expected: true == Result: true)  
279  
280 *****  
281  
282 *****  
283 ***** TESTCASE START  
284 *****  
285 *****  
286 Test Load Type Name Java Type  
287
```

```
288 [Test OK] Result: (Expected: SpeedController == Result:  
289     ↢ SpeedController)  
290  
291 Test Load Type Name Java Type invalid Format  
292 [Test OK] Result: (Expected: == Result: )  
293  
294 Test Load Type Name Java Type invalid Format  
295 [Test OK] Result: (Expected: == Result: )  
296  
297 Test Load Type Name Java Type invalid Format  
298 [Test OK] Result: (Expected: S2peedController == Result:  
299     ↢ S2peedController)  
300  
301 Test Load Type Name Java Type invalid Format  
302 [Test OK] Result: (Expected: == Result: )  
303  
304  
305 Test Save LineFormat Java Type  
306 [Test OK] Result: (Expected: class SpeedController  
307     == Result: class SpeedController  
308 )  
309  
310 Test for Exception in TestCase  
311 [Test OK] Result: (Expected: true == Result: true)  
312  
313  
314 *****  
315  
316  
317 *****  
318         TESTCASE START  
319 *****  
320  
321 Normal Operating Parser  
322 [Test OK] Result: (Expected: true == Result: true)  
323  
324 .AddType() - add empty type to parser  
325 [Test OK] Result: (Expected: ERROR: Provided string is empty.  
326     ↢ == Result: ERROR: Provided string is empty.)  
327  
328 .AddVariable() - add empty type to factory  
329 [Test OK] Result: (Expected: ERROR: Provided string is empty.
```

```
329     ↳ == Result: ERROR: Provided string is empty.)  
330 .AddVariable() - add empty var to factory  
331 [Test OK] Result: (Expected: ERROR: Provided string is empty.  
    ↳ == Result: ERROR: Provided string is empty.)  
332 .AddVariable() - add variable with nonexisting type  
333 [Test OK] Result: (Expected: ERROR: Provided type does not  
    ↳ exist. == Result: ERROR: Provided type does not exist.)  
334 .AddType() - add duplicate type  
335 [Test OK] Result: (Expected: ERROR: Provided type already  
    ↳ exists. == Result: ERROR: Provided type already exists.)  
336 .AddVar() - add duplicate Var  
337 [Test OK] Result: (Expected: ERROR: Provided Variable already  
    ↳ exists. == Result: ERROR: Provided Variable already  
    ↳ exists.)  
338  
339 Test Store and Load Java Fact with exception Dup Type  
340 [Test OK] Result: (Expected: ERROR: Provided type already  
    ↳ exists. == Result: ERROR: Provided type already exists.)  
341  
342 Test Store and Load IEC Fact with exception Dup Type  
343 [Test OK] Result: (Expected: ERROR: Provided type already  
    ↳ exists. == Result: ERROR: Provided type already exists.)  
344  
345 *****  
346 TEST OK!!
```

7 Quellcode

7.1 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
```

7.2 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_NONEEXISTING_TYPE = "ERROR: Provided_type_does_not_exist.";
23     inline static const std::string ERROR_DUPLICATE_TYPE = "ERROR: Provided_type_already_exists.";
24     inline static const std::string ERROR_DUPLICATE_VAR = "ERROR: Provided_Variable_already_exists.";
25
26     /**
27     * \brief Polymorphic container for saving variables
28     */
29     using TVariableCont = std::vector<Variable::Uptr>;
30
31     /**
32     * \brief Polymorphic container for saving types
33     */
34     using TTypeCont = std::vector<Type::Sptr>;
35
36
37     /**
38     * \brief Sets Factory for parsing a language
39     * \brief Previous variables and types of prior factory get saved,
40     * \brief then the subsequent factories variables and types get loaded.
41     * \param Reference to a SymbolFactory
42     * \return void
43     */
44     void SetFactory(const ISymbolFactory& Factory);
45
46     /**
47     * \brief Adds a new type to the language
48     * \param string of typename
49     * \return void
50     */
51     void AddType(std::string const& name);
52
53     /**
54     * \brief Adds a new variable if type exists
55     * \param string of variable, string of type
56     * \return void
57     */
58     void AddVariable(std::string const& name, std::string const& type);
59
60     /**
61     * \brief CTOR of a Symbol Parser Object.
62     * \Load the current state from the sym files
63     * \param fact
64     */
65     SymbolParser(const ISymbolFactory& fact);
66
67     /**
68     * \brief DTOR of Symbol Parser.
69     * \Save the current state to the sym Files
70     */
71     virtual ~SymbolParser();
72

```

```
73 // Delete CopyCtor and Assign Op to prevent untestet behaviour.
74 SymbolParser(SymbolParser& s) = delete;
75 void operator=(SymbolParser s) = delete;
76
77 protected:
78 private:
79 /**
80 * \brief Saves the current state of a SymbolFactory to its file
81 * \param string of type files path, string of variable files path
82 * \return void
83 */
84 void SaveState(const std::string& type_file, const std::string& var_file);
85
86 /**
87 * \brief Loads a SymbolFactory's variables and types from file
88 * \param string of type files path, string of variable files path
89 * \return void
90 */
91 void LoadNewState(const std::string& type_file, const std::string& var_file);
92
93 TTypeCont m_typeCont;
94 TVariableCont m_variableCont;
95
96 const ISymbolFactory * m_Factory;
97 };
98 #endif
```

7.3 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 #include <algorithm>\n8 #include <fstream>\n9 #include <iostream>\n10 #include <cassert>\n11 #include "SymbolParser.hpp"\n12 #include "ISymbolFactory.hpp"\n13\n14 using namespace std;\n15\n16 void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)\n17 {\n18     assert(m_Factory != nullptr);\n19\n20     ofstream type_File;\n21     ofstream var_File;\n22\n23     type_File.open(type_file);\n24\n25     // check if file is good\n26     if (!type_File.good()) {\n27         type_File.close();\n28         return;\n29     }\n30\n31     for_each(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& type) { type_File << type->\n32             GetSaveLine(); });\n33\n34     type_File.close();\n35\n36     var_File.open(var_file);\n37\n38     // check if file is good\n39     if (!var_File.good()) {\n40         var_File.close();\n41         return;\n42     }\n43\n44     for_each(m_variableCont.cbegin(), m_variableCont.cend(), [&](const auto& var) { var_File << var->\n45             GetSaveLine(); });\n46\n47     var_File.close();\n48 }\n49\n50 void SymbolParser::LoadNewState(const std::string& type_file, const std::string& var_file)\n51 {\n52     assert(m_Factory != nullptr);\n53\n54     ifstream type_File;\n55     ifstream var_File;\n56\n57     m_typeCont.clear();\n58     m_variableCont.clear();\n59\n60     type_File.open(type_file);\n61\n62     // check if file is good\n63     if (!type_File.good()) {\n64         type_File.close();\n65         return;\n66     }\n67\n68     string line;\n69\n70     // Set Name can throw an exception\n    // the file must be closed !!!
```

```
71 |     try {
72 |         while (getline(type_File, line)) {
73 |
74 |             Type::Uptr pType = m_Factory->CreateType("");
75 |
76 |             assert(pType != nullptr);
77 |
78 |             pType->SetName(pType->LoadTypeName(line));
79 |
80 |             m_typeCont.push_back(move(pType));
81 |         }
82 |     }
83 |     catch (...) {
84 |         // file closes automatically due to RAII but here it is anyway
85 |         type_File.close();
86 |         throw; // rethrow
87 |     }
88 |
89 |
90 |     var_File.open(var_file);
91 |
92 |     // check if file is good
93 |     if (!var_File.good()) {
94 |         var_File.close();
95 |         return;
96 |     }
97 |
98 |     // Set Name can throw an exception
99 |     // the file must be closed !!!
100 |    try {
101 |        while (getline(var_File, line)) {
102 |
103 |            auto pVar = m_Factory->CreateVariable("");
104 |
105 |            assert(pVar != nullptr);
106 |
107 |            const string type = pVar->LoadTypeName(line);
108 |            const string name = pVar->LoadVarName(line);
109 |
110 |            pVar->SetName(name);
111 |
112 |            // look up if type even exists if yes add to type container
113 |            for (const auto& m_type : m_typeCont)
114 |            {
115 |                if (type == m_type->GetName())
116 |                {
117 |                    pVar->SetType(m_type);
118 |                }
119 |            }
120 |
121 |            if (pVar->GetTypeName() != "") {
122 |                m_variableCont.push_back(move(pVar));
123 |            }
124 |        }
125 |    }
126 |    catch (...) {
127 |        // file closes automatically due to RAII but here it is anyway
128 |        var_File.close();
129 |        throw; // rethrow
130 |    }
131 |
132 |    var_File.close();
133 |
134 |
135 |
136 void SymbolParser::SetFactory(const ISymbolFactory& Factory)
137 {
138     if (m_Factory == nullptr)
139         throw SymbolParser::ERROR_NULLPTR;
140 |
141     SaveState(m_Factory->GetTypeFileName(), m_Factory->GetVariableFileName());
142 |
143     m_Factory = &Factory;
144 |
145     LoadNewState(m_Factory->GetTypeFileName(), m_Factory->GetVariableFileName());
```

```
146 }
147
148 void SymbolParser::AddType(std::string const& name)
149 {
150     if (m_Factory == nullptr)
151         throw SymbolParser::ERROR_NULLPTR;
152
153     if (name.empty())
154         throw SymbolParser::ERROR_EMPTY_STRING;
155
156     // check if type already exists
157     auto it = find_if(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& t) { return t->GetName() == name; });
158
159     if (it != m_typeCont.cend()) {
160         std::cerr << "Error, type already exists!" <\n";
161         throw ERROR_DUPLICATE_TYPE;
162     }
163
164     Type::Uptr pType = m_Factory->CreateType(name);
165
166     if (pType == nullptr)
167         throw SymbolParser::ERROR_NULLPTR;
168
169     m_typeCont.push_back(move(pType));
170 }
171
172
173 void SymbolParser::AddVariable(std::string const& name, std::string const& type)
174 {
175     if (m_Factory == nullptr)
176         throw SymbolParser::ERROR_NULLPTR;
177
178     if (name.empty())
179         throw SymbolParser::ERROR_EMPTY_STRING;
180
181     if (type.empty())
182         throw SymbolParser::ERROR_EMPTY_STRING;
183
184     // check if variable already exists
185     auto it = find_if(m_variableCont.cbegin(), m_variableCont.cend(),
186                       [&](const auto& t) { return t->GetTypeName() == type && t->GetName() == name; });
187
188     // instead of a fixed output to the console
189     // an exception is thrown!!
190     // but here it is anyway
191     if (it != m_variableCont.cend()) {
192         std::cerr << "Error, variable already exists!" <\n";
193         throw ERROR_DUPLICATE_VAR;
194     }
195
196     // look up if type even exists if yes add to type container
197     for (const auto& m_type : m_typeCont)
198     {
199         if (type == m_type->GetName())
200         {
201             auto pVar = m_Factory->CreateVariable(name);
202
203             if (pVar == nullptr)
204                 throw SymbolParser::ERROR_NULLPTR;
205
206             pVar->SetType(m_type);
207
208             // Move ownership into container
209             m_variableCont.push_back(std::move(pVar));
210
211             // If each variable should only match one type, return early
212             return;
213         }
214     }
215
216     // Error is thrown instead of a console output!
217     // in our opinion this is more flexible than a
218     // fixed output to the console!!
219     // but here it is anyway because it is in the specification for the excersice
```

```
220     std::cerr << "Error_Type_for_Variable_does_not_exist_!!\n";
221     throw ERROR_NONEXISTING_TYPE;
222 }
223
224 SymbolParser::SymbolParser(const ISymbolFactory& fact) : m_Factory( &fact )
225 {
226     // Load State from previous parsing
227     LoadNewState(m_Factory->GetTypeFileName(), m_Factory->GetVariableFileName());
228 }
229
230 SymbolParser::~SymbolParser()
231 {
232     // Save Previous state on destruction of Object
233     SaveState(m_Factory->GetTypeFileName(), m_Factory->GetVariableFileName());
234 }
235 }
```

7.4 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) const =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) const =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() const =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() const =0;  
45  
46  
47     virtual ~ISymbolFactory() = default;  
48  
49 protected:  
50 private:  
51 };  
52  
53 #endif
```

7.5 Identifier.hpp

```
1 //*****  
2 * \file Identifier.hpp  
3 * \brief Generalization of Types and Variables  
4 * \author Simon  
5 * \date Dezember 2025  
6 *****  
7 #ifndef IDENTIFIER_HPP  
8 #define IDENTIFIER_HPP  
9  
10 #include <string>  
11 #include "Object.h"  
12  
13 class Identifier : public Object  
14 {  
15 public:  
16  
17     inline static const std::string ERROR_EMPTY_STRING = "ERROR: Empty String";  
18  
19     /**  
20      * \brief Getter for name  
21      *  
22      * \return string of name  
23      */  
24     std::string GetName() const;  
25  
26     /**  
27      * \brief Sets a name  
28      *  
29      * \param string fileLine  
30      * \return string of type - SymbolParser has to check type for validity  
31      * \throw ERROR_EMPTY_STRING  
32      */  
33     void SetName(const std::string& name);  
34  
35  
36 protected:  
37     Identifier(const std::string& name) : m_name{ name } {}  
38     Identifier() = default;  
39  
40     std::string m_name;  
41 private:  
42 };  
43  
44 #endif
```

7.6 Identifier.cpp

```
1 //*****\n2 * \file Identifier.cpp\n3 * \brief Generalization of Types and Variables\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8 #include "Identifier.hpp"\n9\n10 std::string Identifier::GetName() const{\n11     return m_name;\n12 }\n13\n14 void Identifier::SetName(const std::string& name)\n15 {\n16     if (name.empty()) throw Identifier::ERROR_EMPTY_STRING;\n17     m_name = name;\n18 }\n19\n20 }
```

7.7 Variable.hpp

```

1  /***** Variable.hpp ****/
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
11 #include <memory>
12 #include <string>
13
14 #include "Identifier.hpp"
15 #include "Type.hpp"
16
17 class Variable: public Identifier
18 {
19 public:
20     /**
21      * \brief Unique pointer type for variable
22      */
23     using Uptr = std::unique_ptr<Variable>;
24
25     /**
26      * \brief Returns formatted line of a variables declaration
27      *
28      * \return string of variable
29      */
30     virtual std::string GetSaveLine() const = 0;
31
32     /**
33      * \brief Loads the name of a variables type
34      *
35      * \param string fileLine
36      * \return string of type - SymbolParser has to check type for validity
37      */
38     virtual std::string LoadTypeName(std::string const& fileLine) const = 0;
39
40     /**
41      * \brief Loads name of a variable
42      *
43      * \param string fileLine
44      * \return string of variables name
45      */
46     virtual std::string LoadVarName(std::string const& fileLine) const = 0;
47
48     /**
49      * \brief Sets the type of a variable
50      *
51      * \param shared pointer of type
52      * \return void
53      * \throw ERROR_NULLPTR
54      */
55     void SetType(Type::Sptra type);
56
57     /**
58      * \brief Name getter
59      *
60      * \return string of variable
61      */
62     std::string GetTypeName() const;
63
64
65 protected:
66     Variable(const std::string& name) : Identifier{ name } {}
67     Variable() = default;
68
69     Type::Sptra m_type;
70
71 private:
72

```

73 };
74 **#endif**

7.8 Variable.cpp

```
1 //*****\n2 * \file Variable.cpp\n3 * \brief Abstract class for parsing types\n4 * \author Simon Vogelhuber\n5 * \date Dezember 2025\n6 *****\n7\n8 #include "Variable.hpp"\n9\n10 using namespace std;\n11\n12 void Variable::SetType(Type::Sptr type)\n13 {\n14     if (type == nullptr) throw Type::ERROR_NULLPTR;\n15\n16     m_type = std::move(type);\n17 }\n18\n19 std::string Variable::GetTypeName() const\n20 {\n21     return m_type->GetName();\n22 }
```

7.9 Type.hpp

```

1  /***** //**
2   * \file Type.hpp
3   * \brief Abstract class for parsing types
4   * \author Simon Vogelhuber
5   * \date Dezember 2025
6  *****/
7 #ifndef TYPE_HPP
8 #define TYPE_HPP
9
10 #include <memory>
11 #include <string>
12 #include "Identifier.hpp"
13
14 class Type : public Identifier
15 {
16 public:
17
18     /**
19      * \brief Unique pointer type for type
20      */
21     using Uptr = std::unique_ptr<Type>;
22
23     /**
24      * \brief Shared pointer type for type
25      */
26     using Sptr = std::shared_ptr<Type>;
27
28     /**
29      * \brief Loads a types name from a files line
30      *
31      * \param string fileLine
32      * \return string of type
33      */
34     virtual std::string LoadTypeName(const std::string& fileLine) const = 0;
35
36     /**
37      * \brief Returns formatted line of a types declaration
38      *
39      * \return string of type declaration
40      */
41     virtual std::string GetSaveLine() const = 0;
42
43
44 protected:
45     Type(const std::string& name) : Identifier{ name } {}
46     Type() = default;
47 private:
48 };
49 #endif

```

7.10 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     /**\n17      * \brief Getter for static member Singleton.\n18      *\n19      * \return\n20      */\n21     static T& GetInstance() {\n22         if (mInstance == nullptr) { mInstance = std::unique_ptr<T>{ new T{} }; }\n23         return *mInstance;\n24     }\n25 protected:\n26     SingletonBase() = default;\n27     virtual ~SingletonBase() = default;\n28\n29 private:\n30     SingletonBase(SingletonBase const& s) = delete;\n31     SingletonBase& operator = (SingletonBase const& s) = delete;\n32     static std::unique_ptr<T> mInstance;\n33 };\n34\n35 template <typename T> std::unique_ptr<T> SingletonBase<T>::mInstance = nullptr;\n36\n37 #endif // !SINGLETON_BASE_HPP
```

7.11 JavaType.hpp

```
1 //*****\n2 * \file JavaType.hpp\n3 * \brief A Class for parsing java types\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8 #ifndef JAVA_TYPE_HPP\n9 #define JAVA_TYPE_HPP\n10\n11 #include "Type.hpp"\n12 #include <string>\n13\n14 class JavaType : public Type\n15 {\n16 public:\n17     /**\n18      * \brief Loads a types name from a files line\n19      *\n20      * \param string fileLine\n21      * \return string of type\n22      */\n23     virtual std::string LoadTypeName(const std::string& fileLine) const override;\n24\n25     /**\n26      * \brief Returns formatted line of a types declaration\n27      *\n28      * \return string of type declaration\n29      */\n30     virtual std::string GetSaveLine() const override;\n31\n32     JavaType(const std::string name) : Type{ name } {}\n33\n34     JavaType() = default;\n35\n36 protected:\n37 private:\n38};\n#endif
```

7.12 JavaType.cpp

```
1 //*****  
2 * \file JavaType.cpp  
3 * \brief A Class for parsing java types  
4 * \author Simon  
5 * \date Dezember 2025  
6 *****  
7  
8 #include "JavaType.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 /**  
18 * \brief Scans an input string for the Type name of the Type.  
19 *  
20 * \param scan Reference to scanner object  
21 * \return empty string if no valid type name is found  
22 * \return name of type  
23 */  
24 static std::string ScanTypeName(scanner& scan) {  
25     try{  
26         string TypeName;  
27  
28         if (scan.get_identifier() == "class") {  
29             scan.next_symbol();  
30             TypeName = scan.get_identifier();  
31             scan.next_symbol();  
32             if (!scan.has_symbol()) {  
33                 return TypeName;  
34             }  
35         }  
36     }  
37     // catch Scanner Exceptions  
38     catch (...) {  
39         return "";  
40     }  
41  
42     return "";  
43 }  
44  
45  
46 std::string JavaType::LoadTypeName(const std::string& fileLine) const  
47 {  
48     stringstream sstream;  
49  
50     sstream << fileLine;  
51  
52     scanner scan;  
53  
54     scan.set_istream(sstream);  
55  
56     return ScanTypeName(scan);  
57 }  
58  
59 std::string JavaType::GetSaveLine() const  
60 {  
61     return "class_" + m_name + "\n";  
62 }
```

7.13 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};\n#endif
```

7.14 JavaVariable.cpp

```
1 //*****\n2 * \file JavaVariable.cpp\n3 * \brief A Class for parsing java variables\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8 #include "JavaVariable.hpp"\n9 #include <iostream>\n10 #include <string>\n11 #include "scanner.h"\n12\n13 using namespace pfc;\n14 using namespace std;\n15\n16 /**\n17 * \brief Scans an input string for the Type name of the Var.\n18 *\n19 * \param scan Reference to scanner object\n20 * \return empty string if no valid type name is found\n21 * \return name of type\n22 */\n23 static std::string ScanTypeName(scanner& scan)\n24 {\n25     string typeName;\n26\n27     try{\n28         typeName = scan.get_identifier();\n29         scan.next_symbol();\n30     }\n31     // catch Scanner Exceptions\n32     catch (...) {\n33         return "";\n34     }\n35\n36     return typeName;\n37 }\n38\n39 /**\n40 * \brief Scans an input string for the Variable name of the Var.\n41 *\n42 * \param scan Reference to scanner object\n43 * \return empty string if no valid Variable name is found\n44 * \return name of Variable\n45 */\n46 static std::string ScanVarName(scanner& scan)\n47 {\n48     string varName;\n49\n50     try{\n51         varName = scan.get_identifier();\n52         scan.next_symbol();\n53\n54         // The line should be empty after the var Name!\n55         if (!scan.is(';')) varName = "";\n56     }\n57     // catch Scanner Exceptions\n58     catch (...) {\n59         return "";\n60     }\n61\n62     return varName;\n63 }\n64\n65 std::string JavaVariable::GetSaveLine() const\n66 {\n67     if (m_type == nullptr) return "";\n68\n69     return m_type->GetName() + " " + m_name + ";"\n70 }\n71\n72 std::string JavaVariable::LoadTypeName(std::string const& fileLine) const
```

```
73 {
74     stringstream lineStream;
75     lineStream << fileLine;
76     scanner scan(lineStream);
77
78     return ScanTypeName(scan);
79 }
80
81 std::string JavaVariable::LoadVarName(std::string const& fileLine) const
82 {
83     stringstream lineStream;
84     lineStream << fileLine;
85     scanner scan( lineStream );
86
87     string typeName = ScanTypeName(scan);
88     string varName = ScanVarName(scan);
89     if (typeName.empty()) varName = "";
90
91     return varName;
92 }
```

7.15 JavaSymbolFactory.hpp

```
1 //*****\n2 * \file JavaSymbolFactory.hpp\n3 * \brief A factory for creating java variables and types\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7 #ifndef JAVA_SYMBOL_FACTORY_HPP\n8 #define JAVA_SYMBOL_FACTORY_HPP\n9\n10 #include "ISymbolFactory.hpp"\n11 #include "SingletonBase.hpp"\n12\n13 class JavaSymbolFactory : public ISymbolFactory, public SingletonBase<JavaSymbolFactory>\n14 {\n15 public:\n16\n17     friend class SingletonBase<JavaSymbolFactory>;\n18\n19     /**\n20      * \brief Creates a java variable\n21      *\n22      * \param string of variables name\n23      * \return unique pointer to variable\n24      */\n25     virtual Variable::Uptr CreateVariable(const std::string& name) const override;\n26\n27     /**\n28      * \brief Creates a java type\n29      *\n30      * \param string of typename\n31      * \return unique pointer to type\n32      */\n33     virtual Type::Uptr CreateType(const std::string& name) const override;\n34\n35     /**\n36      * \brief Getter for file path of type file\n37      *\n38      * \return string of filePath\n39      */\n40     virtual const std::string& GetTypeFileName() const override;\n41\n42     /**\n43      * \brief Getter for file path of variable file\n44      *\n45      * \return string of filePath\n46      */\n47     virtual const std::string& GetVariableFileName() const override;\n48\n49     // delete CopyCtor and Assign operator to prevent untested behaviour\n50     JavaSymbolFactory(JavaSymbolFactory& fact) = delete;\n51     void operator=(JavaSymbolFactory fact) = delete;\n52\n53 protected:\n54 private:\n55     JavaSymbolFactory() = default;\n56     const std::string m_TypeFileName = "JavaTypes.sym";\n57     const std::string m_VariableFileName = "JavaVars.sym";\n58};\n59#endif
```

7.16 JavaSymbolFactory.cpp

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

7.17 IECType.hpp

```
1 //*****  
2 * \file IECType.hpp  
3 * \brief A Class for parsing IEC types  
4 * \author Simon  
5 * \date Dezember 2025  
6 //*****  
7  
8 #ifndef IEC_TYPE_HPP  
9 #define IEC_TYPE_HPP  
10 #include "Object.h"  
11 #include "Type.hpp"  
12  
13 class IECType: public Type  
14 {  
15 public:  
16     /**  
17      * \brief Loads a types name from a files line  
18      *  
19      * \param string fileLine  
20      * \return string of type  
21      */  
22     virtual std::string LoadTypeName(const std::string& fileLine) const override;  
23  
24     /**  
25      * \brief Returns formatted line of a types declaration  
26      *  
27      * \return string of type declaration  
28      */  
29     virtual std::string GetSaveLine() const override;  
30  
31     /**  
32      * \brief Constructs a Type with a specified name.  
33      *  
34      * \param name  
35      */  
36     IECType(const std::string name) : Type{ name } {}  
37  
38     IECType() = default;  
39  
40 protected:  
41 private:  
42};  
43 #endif
```

7.18 IECType.cpp

```
1 //*****\n2 * \file IECType.cpp\n3 * \brief A Class for parsing IEC types\n4 * \author Simon\n5 * \date Dezember 2025\n6 *****\n7\n8 #include "IECType.hpp"\n9 #include <iostream>\n10 #include <string>\n11 #include <iostream>\n12 #include "scanner.h"\n13\n14 using namespace pfc;\n15 using namespace std;\n16\n17 /**\n18 * \brief Scans an input string for the Type name of the Type.\n19 *\n20 * \param scan Reference to scanner object\n21 * \return empty string if no valid type name is found\n22 * \return name of type\n23 */\n24 static std::string ScanTypeName(scanner& scan) {\n25     try {\n26         string TypeName;\n27\n28         if (scan.get_identifier() == "TYPE") {\n29             scan.next_symbol();\n30             TypeName = scan.get_identifier();\n31             scan.next_symbol();\n32             if (!scan.has_symbol()) {\n33                 return TypeName;\n34             }\n35         }\n36     }\n37     // catch Scanner Exceptions\n38     catch (...) {\n39         return "";\n40     }\n41\n42     return "";\n43 }\n44\n45\n46 std::string IECType::LoadTypeName(const std::string& fileLine) const\n47 {\n48     stringstream sstream;\n49\n50     sstream << fileLine;\n51\n52     scanner scan;\n53\n54     scan.set_istream(sstream);\n55\n56     return ScanTypeName(scan);\n57 }\n58\n59\n60 std::string IECType::GetSaveLine() const\n61 {\n62     return "TYPE" + m_name + "\n";\n63 }
```

7.19 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     /**  
17      * \brief Returns formatted line of a Variable declaration  
18      *  
19      * \return string of variable  
20      */  
21     virtual std::string GetSaveLine() const override;  
22  
23     /**  
24      * \brief Loads the name of a variables type  
25      *  
26      * \param string fileLine  
27      * \return string of type - SymbolParser has to check type for validity  
28      */  
29     virtual std::string LoadTypeName(std::string const& fileLine) const override;  
30  
31     /**  
32      * \brief Loads name of a variable  
33      *  
34      * \param string fileLine  
35      * \return string of variables name  
36      */  
37     virtual std::string LoadVarName(std::string const& fileLine) const override;  
38  
39     IECVariable() = default;  
40  
41     IECVariable(const std::string& name) : Variable{ name } {}  
42  
43 protected:  
44 private:  
45 };  
46 #endif
```

7.20 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 <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     if (m_type == nullptr) return "";
20
21     return "VAR_" + m_type->GetName() + "_" + m_name + ";" + "\n";
22 }
23
24 /**
25 * \brief Scans an input string for the Type name of the Var.
26 *
27 * \param scan Reference to scanner object
28 * \return empty string if no valid type name is found
29 * \return name of type
30 */
31 static std::string ScanTypeName(scanner & scan) {
32     try{
33         string TypeName;
34
35         if (scan.get_identifier() == "VAR") {
36             scan.next_symbol();
37             TypeName = scan.get_identifier();
38             scan.next_symbol();
39             return TypeName;
40         }
41     }
42     // catch Scanner Exceptions
43     catch (...) {
44         return "";
45     }
46     return "";
47 }
48
49 /**
50 * \brief Scans an input string for the Variable name of the Var.
51 *
52 * \param scan Reference to scanner object
53 * \return empty string if no valid Variable name is found
54 * \return name of Variable
55 */
56 static std::string ScanVarName(scanner & scan) {
57     string VarName;
58
59     try{
60         if (scan.is(':')) {
61             scan.next_symbol();
62             VarName = scan.get_identifier();
63             scan.next_symbol();
64             if (!scan.is(';')) {
65                 VarName = "";
66             }
67         }
68     }
69     // catch Scanner Exceptions
70     catch (...) {
71         return "";
72     }

```

```
73     return VarName;
74 }
75
76
77 std::string IECVariable::LoadTypeName(std::string const& fileLine) const
78 {
79     stringstream converter;
80     converter << fileLine;
81     scanner Scan;
82
83     Scan.set_istream(converter);
84
85     return ScanTypeName(Scan);
86 }
87
88 std::string IECVariable::LoadVarName(std::string const& fileLine) const
89 {
90     stringstream converter;
91     converter << fileLine;
92     scanner Scan;
93
94     Scan.set_istream(converter);
95
96     string Typename = ScanTypeName(Scan);
97     string VarName = ScanVarName(Scan);
98
99     if (Typename.empty()) VarName = "";
100
101
102     return VarName;
103 }
```

7.21 IECSymbolFactory.hpp

```

1  ****
2  * \file   IECsymbolFactory.hpp
3  * \brief A factory for creating IEC variables and types
4  * \author Simon
5  * \date   Dezember 2025
6  ****
7  #ifndef IEC_SYMBOL_FACTORY_HPP
8  #define IEC_SYMBOL_FACTORY_HPP
9
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     // This class is a Singleton
19     friend class SingletonBase<IECSymbolFactory>;
20
21     /**
22      * \brief Creates a IEC variable
23      *
24      * \param string of variables name
25      * \return unique pointer to variable
26      */
27     virtual Variable::Uptr CreateVariable(const std::string& name) const override;
28
29     /**
30      * \brief Creates a IEC type
31      *
32      * \param string of typename
33      * \return unique pointer to type
34      */
35     virtual Type::Uptr CreateType(const std::string& name) const override;
36
37     /**
38      * \brief Getter for file path of type file
39      *
40      * \return string of filePath
41      */
42     virtual const std::string& GetTypeFileName() const override;
43
44     /**
45      * \brief Getter for file path of variable file
46      *
47      * \return string of filePath
48      */
49     virtual const std::string& GetVariableFileName() const override;
50
51     // delete CopyCtor and Assign operator to prevent untested behaviour
52     IECsymbolFactory(IECSymbolFactory& fact) = delete;
53     void operator=(IECSymbolFactory fact) = delete;
54
55 protected:
56 private:
57     IECSymbolFactory() = default;
58
59     const std::string m_TypeFileName = "IECTypes.sym";
60     const std::string m_VariableFileName = "IECVars.sym";
61 };
62
63 #endif

```

7.22 IECSymbolFactory.cpp

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

7.23 main.cpp

```

1 //*****\file Main.cpp*****
2 * \brief Testdriver for Symbol Parser and all connected Classes
3 * \author Simon
4 * \date November 2025
5 ****
6
7
8 // These Includes are needed because of the testcases !!
9 #include "IECVariable.hpp"
10 #include "JavaVariable.hpp"
11 #include "IECType.hpp"
12 #include "JavaType.hpp"
13
14 // The Client thests the SymbolParser and SymbolFactories
15 #include "Client.hpp"
16
17 // Testing Includes
18 #include "Test.hpp"
19 #include "vld.h"
20 #include <fstream>
21 #include <iostream>
22 #include <cassert>
23
24 #include <cstdio>
25
26 using namespace std;
27
28 #define WriteOutputFile ON
29
30 static bool TestVariable(Variable* var, const string & name, Type::SPtr typ, ostream & ost = cout);
31 static bool TestType(Type::SPtr typ, ostream & ost = cout);
32 static bool TestIECVar(ostream& ost = cout);
33 static bool TestJavaVar(ostream& ost = cout);
34 static bool TestIECType(ostream& ost = cout);
35 static bool TestJavaType(ostream& ost = cout);
36
37 /**
38 * \brief For deleting the sym files bevor the test cases!.
39 *
40 * \param path
41 */
42 static void EraseFile(const char* path) {
43     // Versucht, die Datei zu loeschen
44     if (std::remove(path) == 0) {
45         // Datei wurde erfolgreich geloescht
46         std::printf("Datei '%s' erfolgreich geloescht.\n", path);
47     }
48     else {
49         // Fehler beim Loeschen der Datei
50         std::perror("Fehler beim Loeschen der Datei");
51     }
52 }
53
54 int main()
55 {
56     // Erase previos Symbol files for test cases
57     EraseFile("IECTypes.sym");
58     EraseFile("IECVars.sym");
59     EraseFile("JavaTypes.sym");
60     EraseFile("JavaVars.sym");
61
62     bool TestOK = true;
63
64     ofstream output{ "output.txt" };
65
66
67     try {
68         Type::SPtr Itype{ make_shared<IECType>(IECType{ "int" }) };
69         Type::SPtr Jtyp{ make_shared<JavaType>(JavaType{ "int" }) };
70
71
72

```

```

73 IECVariable IECVar{ "asdf" };
74 IECVar.SetType(Itype);
75
76 JavaVariable JavaVar{ "jklm" };
77 JavaVar.SetType(Jtyp);
78
79 cout << "\n*****TestIECVarGetter****\n";
80 TestOK = TestOK && TestVariable(&IECVar, "asdf", Itype);
81
82 cout << "\n*****TestJavaVarGetter****\n";
83 TestOK = TestOK && TestVariable(&JavaVar, "jklm", Jtyp);
84
85 cout << "\n*****TestIECTypeGetter****\n";
86 TestOK = TestOK && TestType(Itype);
87
88 cout << "\n*****TestJavaTypeGetter****\n";
89 TestOK = TestOK && TestType(Jtyp);
90
91 TestOK = TestOK && TestIECVar();
92
93 TestOK = TestOK && TestJavaVar();
94
95 TestOK = TestOK && TestIECType();
96
97 TestOK = TestOK && TestJavaType();
98
99 TestOK = TestOK && TestSymbolParser();
100
101 if (WriteOutputFile) {
102
103     // Erase previous Symbol files for test cases
104     EraseFile("IECTypes.sym");
105     EraseFile("IECVars.sym");
106     EraseFile("JavaTypes.sym");
107     EraseFile("JavaVars.sym");
108
109
110     Type::Sptr Itypel{ make_shared<IECType>(IECType{ "int" }) };
111
112     Type::Sptr Jtyp1{ make_shared<JavaType>(JavaType{ "int" }) };
113
114     IECVariable IECVar1{ "asdf" };
115     IECVar1.SetType(Itypel);
116
117     JavaVariable JavaVar1{ "jklm" };
118     JavaVar1.SetType(Jtyp1);
119
120     output << TestStart;
121
122     output << "\n*****TestIECVarGetter****\n";
123     TestOK = TestOK && TestVariable(&IECVar1, "asdf", Itypel, output);
124
125     output << "\n*****TestJavaVarGetter****\n";
126     TestOK = TestOK && TestVariable(&JavaVar1, "jklm", Jtyp1, output);
127
128     output << "\n*****TestIECTypeGetter****\n";
129     TestOK = TestOK && TestType(Itypel, output);
130
131     output << "\n*****TestJavaTypeGetter****\n";
132     TestOK = TestOK && TestType(Jtyp1, output);
133
134     TestOK = TestOK && TestIECVar(output);
135
136     TestOK = TestOK && TestJavaVar(output);
137
138     TestOK = TestOK && TestIECType(output);
139
140     TestOK = TestOK && TestJavaType(output);
141
142     TestOK = TestOK && TestSymbolParser(output);
143
144     if (TestOK) {
145         output << TestCaseOK;
146     }
147     else {

```

```
148         output << TestCaseFail;
149     }
150
151     output.close();
152 }
153
154 if (TestOK) {
155     cout << TestCaseOK;
156 }
157 else {
158     cout << TestCaseFail;
159 }
160 }
161 catch (const string& err) {
162     cerr << err << TestCaseFail;
163 }
164 catch (bad_alloc const& error) {
165     cerr << error.what() << TestCaseFail;
166 }
167 catch (const exception& err) {
168     cerr << err.what() << TestCaseFail;
169 }
170 catch (...) {
171     cerr << "Unhandelt_Exception" << TestCaseFail;
172 }
173
174 if (output.is_open()) output.close();
175
176 return 0;
177 }
178
179 bool TestVariable(Variable* var, const string& name, Type::Sptr typ, ostream& ost)
180 {
181     assert(ost.good());
182     assert(var != nullptr);
183     assert(typ != nullptr);
184
185     ost << TestStart;
186
187     bool TestOK = true;
188     string error_msg;
189
190     try {
191
192         TestOK = TestOK && check_dump(ost, "Test_Variable_Get_Name", name, var->GetName());
193         TestOK = TestOK && check_dump(ost, "Test_Variable_Get_Type", typ->GetName(), var->GetTypeName()
194             );
195
196         const string var_name = "uint_fast_256_t";
197
198         var->SetName(var_name);
199
200         TestOK = TestOK && check_dump(ost, "Test_Variable_Set_Name", var_name, var->GetName());
201
202     }
203     catch (const string& err) {
204         error_msg = err;
205     }
206     catch (bad_alloc const& error) {
207         error_msg = error.what();
208     }
209     catch (const exception& err) {
210         error_msg = err.what();
211     }
212     catch (...) {
213         error_msg = "Unhandelt_Exception";
214     }
215
216     TestOK = TestOK && check_dump(ost, "Check_for_Exception_in_TestCase", true, error_msg.empty());
217     error_msg.clear();
218
219     try {
220         var->SetName("");
221     }
```

```
222     catch (const string& err) {
223         error_msg = err;
224     }
225     catch (bad_alloc const& error) {
226         error_msg = error.what();
227     }
228     catch (const exception& err) {
229         error_msg = err.what();
230     }
231     catch (...) {
232         error_msg = "UnhandeltedException";
233     }
234
235     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Set_Name", Variable::ERROR_EMPTY_STRING,
236                                     error_msg);
237     error_msg.clear();
238
239     try {
240         var->SetType(nullptr);
241     }
242     catch (const string& err) {
243         error_msg = err;
244     }
245     catch (bad_alloc const& error) {
246         error_msg = error.what();
247     }
248     catch (const exception& err) {
249         error_msg = err.what();
250     }
251     catch (...) {
252         error_msg = "UnhandeltedException";
253     }
254
255     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Set_Type_with_nullptr", Variable::
256                                     ERROR_NULLPTR, error_msg);
257     error_msg.clear();
258
259     try {
260         TestOK = TestOK && check_dump(ost, "Test_Variable_Get_Type_after_set_with_nullptr", typ->
261                                         GetName(), var->GetTypeName());
262
263         typ->SetName("uint_fast512_t");
264         var->SetType(typ);
265
266         TestOK = TestOK && check_dump(ost, "Test_Variable_Get_Type_after_set", typ->GetName(), var->
267                                         GetTypeName());
268     }
269     catch (const string& err) {
270         error_msg = err;
271     }
272     catch (bad_alloc const& error) {
273         error_msg = error.what();
274     }
275     catch (const exception& err) {
276         error_msg = err.what();
277     }
278     catch (...) {
279         error_msg = "UnhandeltedException";
280     }
281
282     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
283     error_msg.clear();
284
285     ost << TestEnd;
286
287     return TestOK;
288 }
289
290 bool TestType(Type::Sptr typ, ostream& ost)
291 {
292     assert(ost.good());
293     assert(typ != nullptr);
```

```
293     ost << TestStart;
294
295     bool TestOK = true;
296     string error_msg;
297
298     try {
299         typ->SetName("unit_1024_t");
300         TestOK = TestOK && check_dump(ost, "Test_Type_Get_Name_after_Set", static_cast<string>("unit_1024_t"), typ->GetName());
301     }
302     catch (const string& err) {
303         error_msg = err;
304     }
305     catch (bad_alloc const& error) {
306         error_msg = error.what();
307     }
308     catch (const exception& err) {
309         error_msg = err.what();
310     }
311     catch (...) {
312         error_msg = "Unhandelt_Exception";
313     }
314
315     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Set_Type", true, error_msg.empty());
316     error_msg.clear();
317
318
319     try {
320         typ->SetName("");
321     }
322     catch (const string& err) {
323         error_msg = err;
324     }
325     catch (bad_alloc const& error) {
326         error_msg = error.what();
327     }
328     catch (const exception& err) {
329         error_msg = err.what();
330     }
331     catch (...) {
332         error_msg = "Unhandelt_Exception";
333     }
334
335     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Set_Type", Type::ERROR_EMPTY_STRING,
336                                     error_msg);
337     error_msg.clear();
338
339     ost << TestEnd;
340
341     return TestOK;
342 }
343
344 bool TestIECVar(ostream& ost)
345 {
346     assert(ost.good());
347
348     ost << TestStart;
349
350     bool TestOK = true;
351     string error_msg;
352
353     try {
354
355         IECVariable var;
356
357         const string LineToDecode = "VAR_mCont:_Speed_Controller;\n";
358         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var", static_cast<string>("mCont"),
359                                         var.LoadTypeName(LineToDecode));
360         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var", static_cast<string>("Speed_Controller"),
361                                         var.LoadVarName(LineToDecode));
362
363         const string InvLineToDecode = "1VAR_mCont:_SpeedController;";
```

```

363     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
364         string>(""), var.LoadTypeName(InvLineToDecode));
365     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
366         string>(""), var.LoadVarName(InvLineToDecode));
367     const string Inv2LineToDecode = "VAR_mCont_:SpeedController";
368     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
369         string>("mCont"), var.LoadTypeName(Inv2LineToDecode));
370     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
371         string>("mCont"), var.LoadVarName(Inv2LineToDecode));
372     const string Inv3LineToDecode = "Var_mCont_:a12343";
373     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
374         string>("mCont"), var.LoadTypeName(Inv3LineToDecode));
375     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
376         string>("mCont"), var.LoadVarName(Inv3LineToDecode));
377     const string Inv4LineToDecode = "VAR_123_:a12343";
378     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
379         string>(""), var.LoadTypeName(Inv4LineToDecode));
380     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
381         string>(""), var.LoadVarName(Inv4LineToDecode));
382     const string Inv5LineToDecode = "VAR_123_:a12343";
383     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
384         string>(""), var.LoadTypeName(Inv5LineToDecode));
385     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
386         string>(""), var.LoadVarName(Inv5LineToDecode));
387     Type::SPtr IECTyp = make_shared<IECType>();
388     var.SetName(var.LoadVarName(LineToDecode));
389     IECTyp->SetName(var.LoadTypeName(LineToDecode));
390     var.SetType(IECTyp);
391     TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", LineToDecode, var.
392         GetSaveLine());
393     IECVariable IVar;
394
395     TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", static_cast<string>(""
396         ), IVar.GetSaveLine());
397 }
398 catch (const string& err) {
399     error_msg = err;
400 }
401 catch (bad_alloc const& error) {
402     error_msg = error.what();
403 }
404 catch (const exception& err) {
405     error_msg = err.what();
406 }
407 catch (...) {
408     error_msg = "UnhandeltedException";
409 }
410
411 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
412 error_msg.clear();
413
414 ost << TestEnd;
415
416 return TestOK;
417 }
418
419 bool TestJavaVar(ostream& ost)
420 {
421     assert(ost.good());
422
423     ost << TestStart;

```

```

424
425
426     bool TestOK = true;
427     string error_msg;
428
429     try {
430
431         JavaVariable var;
432
433         const string LineToDecode = "mCont_mBut;\n";
434         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var", static_cast<string>("mCont"),
435                                         var.LoadTypeName(LineToDecode));
436         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var", static_cast<string>("mBut"),
437                                         var.LoadVarName(LineToDecode));
438
439         const string InvLineToDecode = "1mCont_mBut;";
440         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var_invalid_Format", static_cast<
441                                         string>(""), var.LoadTypeName(InvLineToDecode));
442         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
443                                         string>(""), var.LoadVarName(InvLineToDecode));
444
445         const string Inv2LineToDecode = "mCont_mBut;";
446         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var_invalid_Format", static_cast<
447                                         string>("mCont"), var.LoadTypeName(Inv2LineToDecode));
448         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
449                                         string>("mBut"), var.LoadVarName(Inv2LineToDecode));
450
451         const string Inv3LineToDecode = "2mCont_mBut;";
452         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var_invalid_Format", static_cast<
453                                         string>("mCont"), var.LoadTypeName(Inv3LineToDecode));
454         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
455                                         string>("mBut"), var.LoadVarName(Inv3LineToDecode));
456
457         const string Inv4LineToDecode = "mCont_123;";
458         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var_invalid_Format", static_cast<
459                                         string>("mCont"), var.LoadTypeName(Inv4LineToDecode));
460         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
461                                         string>("123"), var.LoadVarName(Inv4LineToDecode));
462
463         const string Inv5LineToDecode = "123_a12343;";
464         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var_invalid_Format", static_cast<
465                                         string>("123"), var.LoadTypeName(Inv5LineToDecode));
466         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
467                                         string>("a12343"), var.LoadVarName(Inv5LineToDecode));
468
469         const string Inv6LineToDecode = "";
470         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Var_invalid_Format", static_cast<
471                                         string>(""), var.LoadTypeName(Inv6LineToDecode));
472         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
473                                         string>(""), var.LoadVarName(Inv6LineToDecode));
474
475         Type::SPtr JTyp = make_shared<JavaType>(JavaType{});
476         var.SetName(var.LoadVarName(LineToDecode));
477         JTyp->SetName(var.LoadTypeName(LineToDecode));
478         var.SetType(JTyp);
479
480         TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", LineToDecode, var,
481                                         GetSaveLine());
482
483         JavaVariable JVar;
484
485         TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", static_cast<string>(""
486                                         ), JVar.GetSaveLine());
487     }
488     catch (const string& err) {
489         error_msg = err;
490     }
491     catch (bad_alloc const& error) {
492         error_msg = error.what();
493     }
494     catch (const exception& err) {
495         error_msg = err.what();
496     }
497     catch (...) {
498         error_msg = "UnhandeltedException";
499     }
500 }
```

```
483     }
484 
485     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
486     error_msg.clear();
487 
488     ost << TestEnd;
489 
490     return TestOK;
491 }
492 
493 bool TestIECType(ostream& ost)
494 {
495     assert(ost.good());
496 
497     ost << TestStart;
498 
499     bool TestOK = true;
500     string error_msg;
501 
502     try{
503         IECType typ;
504 
505         const string LineToDecode = "TYPE_SpeedController\n";
506         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type", static_cast<string>("SpeedController"), typ.LoadTypeName(LineToDecode));
507 
508         const string InvLineToDecode = "1TYPE_SpeedController";
509         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_InvalidFormat", static_cast<string>(""), typ.LoadTypeName(InvLineToDecode));
510 
511         const string Inv2LineToDecode = "TYPE_1SpeedController";
512         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_InvalidFormat", static_cast<string>(""), typ.LoadTypeName(Inv2LineToDecode));
513 
514         const string Inv3LineToDecode = "TYPE_S2speedController";
515         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_InvalidFormat", static_cast<string>("S2speedController"), typ.LoadTypeName(Inv3LineToDecode));
516 
517         const string Inv4LineToDecode = "TYPE_SpeedController";
518         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_InvalidFormat", static_cast<string>(""), typ.LoadTypeName(Inv4LineToDecode));
519 
520         const string Inv6LineToDecode = "";
521         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_InvalidFormat", static_cast<string>(""), typ.LoadTypeName(Inv6LineToDecode));
522 
523         typ.SetName(typ.LoadTypeName(LineToDecode));
524 
525         TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_InvalidFormat", LineToDecode, typ.GetSaveLine());
526 
527     }
528 
529     catch (const string& err) {
530         error_msg = err;
531     }
532     catch (bad_alloc const& error) {
533         error_msg = error.what();
534     }
535     catch (const exception& err) {
536         error_msg = err.what();
537     }
538     catch (...) {
539         error_msg = "Unhandelt_Exception";
540     }
541 
542     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
543     error_msg.clear();
544 
545     ost << TestEnd;
546 
547     return TestOK;
548 }
549 
550 bool TestJavaType(ostream& ost)
```

```
551 {
552     assert(ost.good());
553
554     ost << TestStart;
555
556
557     bool TestOK = true;
558     string error_msg;
559
560     try{
561
562         JavaType typ;
563
564         const string LineToDecode = "class_SpeedController\n";
565         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type", static_cast<string>("SpeedController"), typ.LoadTypeName(LineToDecode));
566
567         const string InvLineToDecode = "1class_SpeedController";
568         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast<string>(""), typ.LoadTypeName(InvLineToDecode));
569
570         const string Inv2LineToDecode = "class_1SpeedController";
571         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast<string>(""), typ.LoadTypeName(Inv2LineToDecode));
572
573         const string Inv3LineToDecode = "class_S2speedController";
574         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast<string>("S2speedController"), typ.LoadTypeName(Inv3LineToDecode));
575
576         const string Inv4LineToDecode = "class_SpeedController";
577         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast<string>(""), typ.LoadTypeName(Inv4LineToDecode));
578
579         const string Inv6LineToDecode = "";
580         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast<string>(""), typ.LoadTypeName(Inv6LineToDecode));
581
582         typ.SetName(typ.LoadTypeName(LineToDecode));
583
584         TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_Java_Type", LineToDecode, typ.GetSaveLine());
585
586     }
587     catch (const string& err) {
588         error_msg = err;
589     }
590     catch (bad_alloc const& error) {
591         error_msg = error.what();
592     }
593     catch (const exception& err) {
594         error_msg = err.what();
595     }
596     catch (...) {
597         error_msg = "UnhandeltedException";
598     }
599
600     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
601     error_msg.clear();
602
603     ost << TestEnd;
604
605     return TestOK;
606 }
```

7.24 Client.hpp

```
1 //*****
2 * \file Client.hpp
3 * \brief Test File to show that you only need to include Symbol Parser
4 * \brief plus factories to work with the parser!
5 *
6 * \author Simon
7 * \date November 2025
8 *****/
9
10 #ifndef CLIENT_HPP
11 #define CLIENT_HPP
12
13 #include <iostream>
14 /**
15 * \brief TestDriver for the SymbolParser.
16 */
17 bool TestSymbolParser(std::ostream& ost = std::cout);
18
19 #endif // !1
```

7.25 Client.cpp

```

1 //***** Client.cpp ****
2 * \file Client.cpp
3 * \brief Test File to show that you only need to include Symbol Parser
4 * \brief plus factories to work with the parser!
5 *
6 * \author Simon
7 * \date November 2025
8 ****
9
10 // The client is only dependent on these classes!!
11 #include "SymbolParser.hpp"
12 #include "JavaSymbolFactory.hpp"
13 #include "IECSymbolFactory.hpp"
14
15 // Testing Includes
16 #include "Test.hpp"
17 #include <fstream>
18 #include <cassert>
19 #include "Client.hpp"
20
21 using namespace std;
22
23 bool TestSymbolParser(std::ostream& ost)
24 {
25     bool TestOK = true;
26     string error_msg;
27
28     if (!ost.good()) throw "Error_Ostream_bad!";
29
30     ost << TestStart;
31
32     // normal operating mode - no exception should be thrown
33     try {
34         SymbolParser parser{ JavaSymbolFactory::GetInstance() };
35         parser.AddType("Button");
36         parser.AddVariable("mButton", "Button");
37         parser.SetFactory(IECSymbolFactory::GetInstance());
38         parser.AddType("TYPE");
39         parser.AddVariable("VARIABLE", "TYPE");
40         parser.SetFactory(JavaSymbolFactory::GetInstance());
41         parser.AddVariable("mButton2", "Button"); // <- this is only possible if the loading of the
42             vars was successful
43     }
44     catch (const string& err) {
45         error_msg = err;
46     }
47     catch (bad_alloc const& error) {
48         error_msg = error.what();
49     }
50     catch (const exception& err) {
51         error_msg = err.what();
52     }
53     catch (...) {
54         error_msg = "Unhandled_Exception";
55     }
56
57     TestOK = TestOK && check_dump(ost, "Normal_Operating_Parser", true, error_msg.empty());
58     error_msg.clear();
59
60     // addtype - adding empty type - throws error
61     try {
62         SymbolParser parser{ JavaSymbolFactory::GetInstance() };
63         parser.AddType("");
64     }
65     catch (const string& err) {
66         error_msg = err;
67     }
68     catch (bad_alloc const& error) {
69         error_msg = error.what();
70     }
71     catch (const exception& err) {
72         error_msg = err.what();
73     }
74
75     cout << "Test OK: " << (TestOK ? "true" : "false") << endl;
76 }

```

```
72     }
73     catch (...) {
74         error_msg = "UnhandeltedException";
75     }
76
77     TestOK = TestOK && check_dump(ost, ".AddType()_-_add_empty_type_to_parser", SymbolParser::
78                                     ERROR_EMPTY_STRING, error_msg);
79     error_msg.clear();
80
81     // addVariable add empty type - throws error
82     try {
83         SymbolParser parser{ JavaSymbolFactory::GetInstance() };
84         parser.AddVariable("VarName", "");
85     }
86     catch (const string& err) {
87         error_msg = err;
88     }
89     catch (bad_alloc const& error) {
90         error_msg = error.what();
91     }
92     catch (const exception& err) {
93         error_msg = err.what();
94     }
95     catch (...) {
96         error_msg = "UnhandeltedException";
97     }
98
99     TestOK = TestOK && check_dump(ost, ".AddVariable()_-_add_empty_type_to_factory", SymbolParser::
100                                ERROR_EMPTY_STRING, error_msg);
101    error_msg.clear();
102
103   // addVariable add empty var - throws error
104   try {
105       SymbolParser parser{ JavaSymbolFactory::GetInstance() };
106       parser.AddVariable("", "Type");
107   }
108   catch (const string& err) {
109       error_msg = err;
110   }
111   catch (bad_alloc const& error) {
112       error_msg = error.what();
113   }
114   catch (const exception& err) {
115       error_msg = err.what();
116   }
117   catch (...) {
118       error_msg = "UnhandeltedException";
119   }
120
121   TestOK = TestOK && check_dump(ost, ".AddVariable()_-_add_empty_var_to_factory", SymbolParser::
122                                 ERROR_EMPTY_STRING, error_msg);
123   error_msg.clear();
124
125   // addVariable add variable for non existing type
126   try {
127       SymbolParser parser{ JavaSymbolFactory::GetInstance() };
128       parser.AddVariable("Var", "Type");
129   }
130   catch (const string& err) {
131       error_msg = err;
132   }
133   catch (bad_alloc const& error) {
134       error_msg = error.what();
135   }
136   catch (const exception& err) {
137       error_msg = err.what();
138   }
139   catch (...) {
140       error_msg = "UnhandeltedException";
141   }
142
143   TestOK = TestOK && check_dump(ost, ".AddVariable()_-_add_variable_with_nonexisting_type",
144                                 SymbolParser::ERROR_NONEEXISTING_TYPE, error_msg);
```

```
143 // addVariable add variable for non existing type
144 try {
145     SymbolParser parser( JavaSymbolFactory::GetInstance() );
146     parser.AddType("uint65536_t");
147     parser.AddType("uint65536_t");
148 }
149 catch (const string& err) {
150     error_msg = err;
151 }
152 catch (bad_alloc const& error) {
153     error_msg = error.what();
154 }
155 catch (const exception& err) {
156     error_msg = err.what();
157 }
158 catch (...) {
159     error_msg = "UnhandeltedException";
160 }
161
162 TestOK = TestOK && check_dump(ost, ".AddType()", "SymbolParser::",
163                                 ERROR_DUPLICATE_TYPE, error_msg);
164 error_msg.clear();
165
166 // addVariable add variable for non existing type
167 try {
168     SymbolParser parser( JavaSymbolFactory::GetInstance() );
169     parser.AddType("uint1096_t");
170     parser.AddVariable("Large_int", "uint4096_t");
171     parser.AddVariable("Large_int", "uint4096_t");
172 }
173 catch (const string& err) {
174     error_msg = err;
175 }
176 catch (bad_alloc const& error) {
177     error_msg = error.what();
178 }
179 catch (const exception& err) {
180     error_msg = err.what();
181 }
182 catch (...) {
183     error_msg = "UnhandeltedException";
184 }
185
186 TestOK = TestOK && check_dump(ost, ".AddVar()", "SymbolParser::",
187                                 ERROR_DUPLICATE_VAR, error_msg);
188 error_msg.clear();
189
190 // Test Load and Store of the SymbolParser
191 try {
192     SymbolParser parser( JavaSymbolFactory::GetInstance() );
193     parser.AddType("uint8192_t");
194     parser.AddVariable("Large_int", "uint8192_t");
195     parser.SetFactory( IECSymbolFactory::GetInstance());
196     parser.AddType("ui32");
197     parser.AddVariable("Hello", "ui32");
198     parser.SetFactory(JavaSymbolFactory::GetInstance());
199     parser.AddType("uint8192_t"); // <-- this should throw exception type already exists!
200 }
201 catch (const string& err) {
202     error_msg = err;
203 }
204 catch (bad_alloc const& error) {
205     error_msg = error.what();
206 }
207 catch (const exception& err) {
208     error_msg = err.what();
209 }
210 catch (...) {
211     error_msg = "UnhandeltedException";
212 }
213
214 TestOK = TestOK && check_dump(ost, "Test_Store_and_Load_Java_Fact_with_Exeption_Dup_Type",
215                                 SymbolParser::ERROR_DUPLICATE_TYPE, error_msg);
```

```
215     error_msg.clear();
216
217
218 // Test Load and Store of the SymbolParser
219 try {
220
221     SymbolParser parser{ IECSymbolFactory::GetInstance() };
222
223     parser.AddType("ui32");
224
225 }
226 catch (const string& err) {
227     error_msg = err;
228 }
229 catch (bad_alloc const& error) {
230     error_msg = error.what();
231 }
232 catch (const exception& err) {
233     error_msg = err.what();
234 }
235 catch (...) {
236     error_msg = "Unhandelt_Exception";
237 }
238
239 TestOK = TestOK && check_dump(ost, "Test_Store_and_Load_IEC_Fact_with_Exeption_Dup_Type",
240                                 SymbolParser::ERROR_DUPLICATE_TYPE, error_msg);
241 error_msg.clear();
242
243 ost << TestEnd;
244 return TestOK;
245 }
```

7.26 Test.hpp

```

1  /***** Test.hpp ****/
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 << "*****" << 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 <<
109             "Result:(Expected:" << std::boolalpha << expected << "!=" << "Result:" <<
110             << result << ")" << std::noboolalpha << std::endl << std::endl;
111     } else {
112         ostr << testcase << std::endl << colorRed << "[Test_FAILED]" << colorWhite <<
113             "Result:(Expected:" << std::boolalpha << expected << "!=" << "Result:" <<
114             << result << ")" << std::noboolalpha << std::endl << std::endl;
115     }
116 #else
117         if (expected == result) {
118             ostr << testcase << std::endl << "[Test_OK]" << "Result:(Expected:" <<
119             std::boolalpha << expected << "!=" << "Result:" << result << ")" << std::endl;
120     } else {
121         ostr << testcase << std::endl << "[Test_FAILED]" << "Result:(Expected:" <<
122             std::boolalpha << expected << "!=" << "Result:" << result << ")" <<
123             std::noboolalpha << std::endl << std::endl;
124     }
125 #endif
126     if (ostr.fail()) {
127         std::cerr << "Error: Write_Ostream" << std::endl;
128     }
129     else {
130         std::cerr << "Error: Bad_Ostream" << std::endl;
131     }
132     return expected == result;
133 }
134 template <typename T1, typename T2>
135 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
136     if (!ost.good()) throw std::exception( "Error_bad_Ostream!" );
137     ost << "(" << p.first << "," << p.second << ")";
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
```

7.27 scanner.h

```
1 //      $Id: scanner.h 46519 2022-12-04 12:29:04Z p20068 $
2 //      $URL: https://svn01.fh-hagenberg.at/bin/cepheden/pfc/trunk/pfc/inc/pfc/scanner.h $
3 // $Revision: 46519 $
4 //      $Date: 2022-12-04 13:29:04 +0100 (So., 04 Dez 2022) $
5 //      $Author: p20068 $
6 //      Creator: Peter Kulczycki
7 //      Creation: November 14, 2020
8 // Copyright: (c) 2021 Peter Kulczycki (peter.kulczycki<AT>fh-hagenberg.at)
9 // License: This document contains proprietary information belonging to
10 //          University of Applied Sciences Upper Austria, Campus Hagenberg.
11 //          It is distributed under the Boost Software License (see
12 //          https://www.boost.org/users/license.html).
13
14 #pragma once
15
16 #undef PFC_SCANNER_VERSION
17 #define PFC_SCANNER_VERSION "2.6.2"
18
19 #include <cassert>
20 #include <format>
21 #include <functional>
22 #include <iostream>
23 #include <locale>
24 #include <sstream>
25 #include <stdexcept>
26 #include <string_view>
27 #include <string>
28 #include <unordered_map>
29 #include <variant>
30
31 using namespace std::string_literals;
32 using namespace std::string_view_literals;
33
34 namespace pfc { namespace scn { namespace details {
35
36     template <typename V> void negate (V & v) {
37     }
38
39     template <typename T, typename ...R, typename V> void negate (V & v) {
40         if (std::holds_alternative <T> (v))
41             std::get <T> (v) *= -1;
42         else
43             negate <R...> (v);
44     }
45
46     struct symbol_kind final {
47         enum class tag_t {
48             keyword, terminal_class, terminal_symbol, undefined
49         };
50
51         friend std::ostream & operator << (std::ostream & lhs, symbol_kind const & rhs) {
52             return lhs << std::format ("({},{},{}]", rhs.str, to_string_view (rhs.tag), rhs.num);
53         }
54
55         static constexpr std::string_view to_string_view (tag_t const tag) {
56             switch (tag) {
57                 case tag_t::keyword:           return "kw"sv; break;
58                 case tag_t::terminal_class:   return "tc"sv; break;
59                 case tag_t::terminal_symbol:  return "ts"sv; break;
60                 case tag_t::undefined:       return "ud"sv; break;
61             }
62
63             return "??"sv;
64         }
65
66         constexpr bool operator == (symbol_kind const & rhs) const {
67             return (tag == rhs.tag) && (num == rhs.num);
68         }
69
70         constexpr bool operator != (symbol_kind const & rhs) const {
71             return !operator == (rhs);
72         }
73     }
74 }}
```

```

73     char           chr {};
74     std::size_t    num {};
75     std::string_view str {"undefined"sv};
76     tag_t          tag {tag_t::undefined};
77 };
78
79
80 constexpr symbol_kind sk_keywd {'_',
81     0, "keyword"sv,
82     symbol_kind::tag_t::keyword};
83 constexpr symbol_kind sk_ident {'_',
84     1, "identifier"sv,
85     symbol_kind::tag_t::terminal_class};
86 constexpr symbol_kind sk_int {'_',
87     2, "integer"sv,
88     symbol_kind::tag_t::terminal_class};
89 constexpr symbol_kind sk_real {'_',
90     3, "real"sv,
91     symbol_kind::tag_t::terminal_class};
92 constexpr symbol_kind sk_str {'_',
93     4, "string"sv,
94     symbol_kind::tag_t::terminal_class};
95 constexpr symbol_kind sk_assign {'=',
96     1, "assign"sv,
97     symbol_kind::tag_t::terminal_symbol};
98 constexpr symbol_kind sk_caret {'^',
99     2, "caret"sv,
100    symbol_kind::tag_t::terminal_symbol};
101   constexpr symbol_kind sk_colon {':',
102     3, "colon"sv,
103     symbol_kind::tag_t::terminal_symbol};
104   constexpr symbol_kind sk_comma {',',
105     4, "comma"sv,
106     symbol_kind::tag_t::terminal_symbol};
107   constexpr symbol_kind sk_div {'/',
108     5, "division"sv,
109     symbol_kind::tag_t::terminal_symbol};
110   constexpr symbol_kind sk_dquote {'"',
111     6, "double_quote"sv,
112     symbol_kind::tag_t::terminal_symbol};
113   constexpr symbol_kind sk_eof {'\0',
114     7, "end_of_file"sv,
115     symbol_kind::tag_t::terminal_symbol};
116   constexpr symbol_kind sk_eol {'\n',
117     8, "end_of_line"sv,
118     symbol_kind::tag_t::terminal_symbol};
119   constexpr symbol_kind sk_lpar {'(',
120     9, "left_parenthesis"sv,
121     symbol_kind::tag_t::terminal_symbol};
122   constexpr symbol_kind sk_minusus {'-',
123     10, "minus"sv,
124     symbol_kind::tag_t::terminal_symbol};
125   constexpr symbol_kind sk_mult {'*',
126     11, "multiply"sv,
127     symbol_kind::tag_t::terminal_symbol};
128   constexpr symbol_kind sk_period {'.',
129     12, "period"sv,
130     symbol_kind::tag_t::terminal_symbol};
131   constexpr symbol_kind sk_plus {'+'},
132     13, "plus"sv,
133     symbol_kind::tag_t::terminal_symbol};
134   constexpr symbol_kind sk_rpar {')',
135     14, "right_parenthesis"sv,
136     symbol_kind::tag_t::terminal_symbol};
137   constexpr symbol_kind sk_semi {';',
138     15, "semicolon"sv,
139     symbol_kind::tag_t::terminal_symbol};
140   constexpr symbol_kind sk_space {'_',
141     16, "space"sv,
142     symbol_kind::tag_t::terminal_symbol};
143   constexpr symbol_kind sk_tab {'\t',
144     17, "tabulator"sv,
145     symbol_kind::tag_t::terminal_symbol};
146   constexpr symbol_kind sk_uscore {'_',
147     18, "underscore"sv,
148     symbol_kind::tag_t::terminal_symbol};
149
150
151 inline symbol_kind to_symbol_kind (char const c) {
152     static std::unordered_map <char, symbol_kind> const map {
153         {'=', sk_assign}, {'^', sk_caret}, {':', sk_colon}, {'/', sk_div },
154         {'"', sk_dquote}, {'\0', sk_eof}, {'\n', sk_eol}, {'(', sk_lpar}, {'-', sk_minus},
155         {'*', sk_mult}, {'.', sk_period}, {'+', sk_plus}, {')', sk_rpar}, {';', sk_semi },
156         {'_', sk_space}, {'\t', sk_tab}, {'_', sk_uscore}
157     };
158
159     if (auto const pos {map.find (c)}; pos != std::end (map))
160         return pos->second;
161
162     return sk_undef;
163 }
164
165 inline auto to_string (char const c) {
166     return std::string {to_symbol_kind (c).str};
167 }
168
169 template <typename T> constexpr bool is_alpha (T const c, std::locale const & locale = std::locale::classic ())
170 {
171     return std::isalpha (c, locale) || std::char_traits <T>::eq (c, T (sk_uscore.chr));
172 }
173
174 constexpr bool is_digit (auto const c, std::locale const & locale = std::locale::classic ()) {
175     return std::isdigit (c, locale);
176 }
177
178 } // namespace details
179
180 struct exception final : std::runtime_error {
181     explicit exception (std::string const & m) : std::runtime_error {m} {
182     }
183 };
184
185 } // namespace scn
186
187 struct symbol final {
188     constexpr symbol () = default;
189
190     template <typename attr_t = int> constexpr symbol (scn::details::symbol_kind kind, attr_t attr = {})
191     : m_kind {std::move (kind)}
192     , m_attr {std::move (attr)} {
193 }

```

```
147  constexpr bool operator == (symbol const & rhs) const {
148      return (m_kind == rhs.m_kind) && (m_attr == rhs.m_attr);
150  }
151
152  constexpr bool operator != (symbol const & rhs) const {
153      return !operator == (rhs);
154  }
155
156  void clear () {
157      m_kind = scn::details::sk_eof;
158      m_attr = 0;
159  }
160
161  template <typename attr_t> constexpr attr_t const & get_attribute () const {
162      return std::get <attr_t> (m_attr);
163  }
164
165  template <typename attr_t> constexpr bool holds_attribute () const {
166      return std::holds_alternative <attr_t> (m_attr);
167  }
168
169  constexpr bool is (char const c) const {
170      return (m_kind.tag == scn::details::symbol_kind::tag_t::terminal_symbol) && (m_kind.chr == c);
171  }
172
173  constexpr bool is_eof () const {
174      return (m_kind == scn::details::sk_eof);
175  }
176
177  constexpr bool is_identifier () const {
178      return (m_kind == scn::details::sk_ident) && holds_attribute <std::string> ();
179  }
180
181  constexpr bool is_integer () const {
182      return (m_kind == scn::details::sk_int) && holds_attribute <int> ();
183  }
184
185  constexpr bool is_keyword () const {
186      return (m_kind.tag == scn::details::symbol_kind::tag_t::keyword) && holds_attribute <std::string>
187          ();
188  }
189
190  constexpr bool is_keyword (std::size_t const num) const {
191      return is_keyword () && (num > 0) && (m_kind.num == num);
192  }
193
194  bool is_keyword (std::string const & name) const {
195      return !std::empty (name) && (get_keyword () == name);
196  }
197
198  constexpr bool is_real () const {
199      return (m_kind == scn::details::sk_real) && holds_attribute <double> ();
200  }
201
202  constexpr bool is_number () const {
203      return is_integer () || is_real ();
204  }
205
206  constexpr bool is_string () const {
207      return (m_kind == scn::details::sk_str) && holds_attribute <std::string> ();
208  }
209
210  constexpr bool is_terminal_class () const {
211      return m_kind.tag == scn::details::symbol_kind::tag_t::terminal_class;
212  }
213
214  constexpr bool is_terminal_symbol () const {
215      return m_kind.tag == scn::details::symbol_kind::tag_t::terminal_symbol;
216  }
217
218  std::string get_identifier () const {
219      return is_identifier () ? get_attribute <std::string> () : ""s;
220  }
```

```
221 |     constexpr int get_integer () const {
222 |         return is_integer () ? get_attribute <int> () : 0;
223 |
224 |
225 |     std::string get_keyword () const {
226 |         return is_keyword () ? get_attribute <std::string> () : ""s;
227 |     }
228 |
229 |     constexpr double get_number () const {
230 |         if (is_integer ()) return get_attribute <int> ();
231 |         if (is_real ()) return get_attribute <double> ();
232 |
233 |         return 0.0;
234 |     }
235 |
236 |     constexpr double get_real () const {
237 |         return is_real () ? get_attribute <double> () : 0.0;
238 |     }
239 |
240 |     std::string get_string () const {
241 |         return is_string () ? get_attribute <std::string> () : ""s;
242 |     }
243 |
244 |     std::ostream & write (std::ostream & out) const {
245 |         out << '(' << m_kind;
246 |
247 |         if (is_keyword () || is_terminal_class ())
248 |             std::visit ([&out] (auto const & v) { out << std::format ("{}'", v); }, m_attr);
249 |
250 |         return out << ')';
251 |     }
252 |
253 |     friend std::ostream & operator << (std::ostream & lhs, symbol const & rhs) {
254 |         return rhs.write (lhs);
255 |     }
256 |
257 |     scn::details::symbol_kind m_kind {scn::details::sk_eof};
258 |     std::variant <int, double, std::string> m_attr {0};
259 | };
260 |
261 | inline std::string to_string (symbol const & sym) {
262 |     static std::ostringstream out; out.str (""); out << sym; return out.str ();
263 | }
264 |
265 | class scanner final {
266 |     public:
267 |         static char const * version () {
268 |             return PFC_SCANNER_VERSION;
269 |         }
270 |
271 |         scanner () {
272 |             set_istream ();
273 |         }
274 |
275 |         explicit scanner (std::istream & in, bool const use_stream_exceptions = false) {
276 |             set_istream (in, use_stream_exceptions);
277 |         }
278 |
279 |         scanner (std::istream &&) = delete;
280 |
281 |         scanner (scanner const &) = delete;
282 |         scanner (scanner &&) = default;
283 |
284 |         ~scanner () = default;
285 |
286 |         scanner & operator = (scanner const &) = delete;
287 |         scanner & operator = (scanner &&) = default;
288 |
289 |         constexpr symbol const & current_symbol () const {
290 |             return m_current_symbol;
291 |         }
292 |
293 |         symbol const & next_symbol () {
294 |             read_next_symbol (); return current_symbol ();
295 |         }
```

```
296     symbol const & next_symbol (char const chr) {
297         if (!is (chr))
298             throw scn::exception {
299                 "Expected_"s + scn::details::to_string (chr) + "' but have'" + to_string (
300                     m_current_symbol) + '.';
301         };
302         return next_symbol ();
303     }
304
305     symbol const & next_symbol (std::string const & name) {
306         if (!is_keyword (name))
307             throw scn::exception {
308                 "Expected_keyword_"s + name + "' but have'" + to_string (m_current_symbol) + '.';
309         };
310         return next_symbol ();
311     }
312
313     template <typename attr_t> constexpr attr_t const & get_attribute () const {
314         return m_current_symbol.get_attribute <attr_t> ();
315     }
316
317     std::string get_identifier () const {
318         return m_current_symbol.get_identifier ();
319     }
320
321     constexpr int get_integer () const {
322         return m_current_symbol.get_integer ();
323     }
324
325     constexpr double get_number () const {
326         return m_current_symbol.get_number ();
327     }
328
329     constexpr double get_real () const {
330         return m_current_symbol.get_real ();
331     }
332
333     std::string get_string () const {
334         return m_current_symbol.get_string ();
335     }
336
337     constexpr bool is (char const c) const {
338         return m_current_symbol.is (c);
339     }
340
341     constexpr bool is_eof () const {
342         return m_current_symbol.is_eof ();
343     }
344
345     constexpr bool is_identifier () const {
346         return m_current_symbol.is_identifier ();
347     }
348
349     constexpr bool is_integer () const {
350         return m_current_symbol.is_integer ();
351     }
352
353     constexpr bool is_keyword () const {
354         return m_current_symbol.is_keyword ();
355     }
356
357     constexpr bool is_keyword (std::size_t const num) const {
358         return m_current_symbol.is_keyword (num);
359     }
360
361     bool is_keyword (std::string const & name) const {
362         return m_current_symbol.is_keyword (name);
363     }
364
365     constexpr bool is_number () const {
366         return m_current_symbol.is_number ();
367     }
368
369 }
```

```
370      constexpr bool is_real () const {
371          return m_current_symbol.is_real ();
372      }
373
374      constexpr bool is_string () const {
375          return m_current_symbol.is_string ();
376      }
377
378      constexpr bool has_symbol () const {
379          return !is_eof ();
380      }
381
382      std::size_t register_keyword (std::string const & name) {
383          if (std::empty (name))
384              return 0;
385
386          if (auto const f {m_keywords.find (name)}; f != std::end (m_keywords))
387              return f->second.m_kind.num;
388
389          auto const sym {make_keyword_symbol (std::size (m_keywords) + 1, name)};
390
391          m_keywords[name] = sym;
392
393          if (is_identifier () && (current_symbol ().get_identifier () == name))
394              m_current_symbol = sym;
395
396          return sym.m_kind.num;
397      }
398
399
400      void set_istream () {
401          m_p_in = nullptr; read_next_chr (); read_next_symbol ();
402      }
403
404      void set_istream (std::istream & in, bool const use_stream_exceptions = false) {
405          if (use_stream_exceptions) {
406              in.exceptions (std::ios::badbit /*| std::ios::eofbit*/ | std::ios::failbit);
407          }
408
409          m_p_in = &in; read_next_chr (); read_next_symbol ();
410      }
411
412      void set_istream (std::istream &&) = delete;
413
414      void signed_numbers (bool const set = true) {
415          m_signed_numbers = set;
416      }
417
418      std::ostream & write (std::ostream & out) const {
419          out << std::format ("{}registered_keywords\n", std::empty (m_keywords) ? "no_\\" : "");
420
421          for (auto const & k : m_keywords)
422              out << "_\\" << k.first << ":"_\" << k.second << '\n';
423
424          return out <<
425              "current_char:_\" << scn::details::to_symbol_kind (m_current_chr) << "\n"
426              "current_symbol:_\" << m_current_symbol << '\n';
427      }
428
429      friend std::ostream & operator << (std::ostream & lhs, scanner const & rhs) {
430          return rhs.write (lhs);
431      }
432
433  private:
434      static symbol make_keyword_symbol (std::size_t const num, std::string const & name) {
435          auto kind {scn::details::sk_keywd}; kind.num = num; return {kind, name};
436      }
437
438      constexpr bool is_eof_chr () const {
439          return m_current_chr == scn::details::sk_eof.chr;
440      }
441
442      constexpr bool is_eol_chr () const {
443          return m_current_chr == scn::details::sk_eol.chr;
444      }
```

```
445     constexpr bool is_whitespace_chr () const {
446         return (m_current_chr == scn::details::sk_space.chr) || (m_current_chr == scn::details::sk_tab
447             .chr) || is_eol_chr ();
448     }
449
450     std::size_t keyword_is_registered (std::string const & name) const {
451         if (auto const pos {m_keywords.find (name)}; pos != std::end (m_keywords))
452             return pos->second.m_kind.num;
453
454         return 0;
455     }
456
457     void read_next_chr () {
458         m_current_chr = scn::details::sk_eof.chr;
459
460         if (m_p_in && *m_p_in)
461             m_p_in->get (m_current_chr);
462     }
463
464     void read_next_chr (char const c) {
465         if (m_current_chr != c)
466             throw scn::exception {"Expected \"s + c + '\" but have '\"' + m_current_chr + '\".'"};
467
468         read_next_chr ();
469     }
470
471     void read_next_symbol () {
472         m_current_symbol.clear (); bool scanned {false};
473
474         while (!scanned) {
475             if (is_eof_chr ())
476                 scanned = true;
477
478             else if (is_whitespace_chr ())
479                 read_next_chr ();
480
481             else if (scn::details::is_alpha (m_current_chr))
482                 scanned = scan_keyword_or_identifier ();
483
484             else if (m_current_chr == scn::details::sk_dquote.chr)
485                 scanned = scan_string ();
486
487             else if (scn::details::is_digit (m_current_chr))
488                 scanned = try_read_number ();
489
490             else if (m_current_chr == scn::details::sk_period.chr)
491                 scanned = try_read_number ();
492
493             else if (m_current_chr == scn::details::sk_plus.chr)
494                 scanned = try_read_number ();
495
496             else if (m_current_chr == scn::details::sk_minus.chr)
497                 scanned = try_read_number ();
498
499             else if (m_current_chr == scn::details::sk_div.chr) {
500                 read_next_chr (scn::details::sk_div.chr);
501
502                 if (m_current_chr == scn::details::sk_div .chr) scan_comment_single_line (scn::
503                     details::sk_div.chr);
504                 else if (m_current_chr == scn::details::sk_mult.chr) scan_comment_multi_line (scn::
505                     details::sk_div.chr);
506
507                 else {
508                     m_current_symbol = scn::details::sk_div; scanned = true;
509                 }
510
511             } else {
512                 m_current_symbol = scn::details::to_symbol_kind (m_current_chr);
513
514                 if ((scanned = m_current_symbol.is_terminal_symbol ()) == false)
515                     throw scn::exception {"Unknown character \"s + m_current_chr + '\"' encountered."};
516
517                 read_next_chr ();
518             }
519         }
520     }
```

```
517     }
518 }
519
520 void scan_comment_multi_line (char prev_chr) {
521     assert (prev_chr == scn::details::sk_div .chr);
522     assert (m_current_chr == scn::details::sk_mult.chr);
523
524     auto depth {0};
525     std::string text {prev_chr};
526
527     do {
528         text += m_current_chr; auto annihilate {false};
529
530         if (prev_chr == scn::details::sk_div .chr) { if (m_current_chr == scn::details::
531             sk_mult.chr) { ++depth; annihilate = true; } }
532         else if (prev_chr == scn::details::sk_mult.chr) { if (m_current_chr == scn::details::sk_div
533             .chr) { --depth; annihilate = true; } }
534
535         prev_chr = annihilate ? scn::details::sk_eof.chr : m_current_chr; read_next_chr ();
536     } while ((depth > 0) && !is_eof_chr ());
537 }
538
539 void scan_comment_single_line (char const prev_chr) {
540     assert (prev_chr == scn::details::sk_div.chr);
541     assert (m_current_chr == scn::details::sk_div.chr);
542
543     std::string text {prev_chr};
544
545     while (!is_eol_chr () && !is_eof_chr ()) {
546         text += m_current_chr; read_next_chr ();
547     }
548
549     bool scan_keyword_or_identifier () {
550         assert (scn::details::is_alpha (m_current_chr));
551
552         std::string text;
553
554         while (scn::details::is_alpha (m_current_chr) || scn::details::is_digit (m_current_chr)) {
555             text += m_current_chr; read_next_chr ();
556         }
557
558         if (auto const num {keyword_is_registered (text)}; num > 0) {
559             m_current_symbol = {scn::details::sk_keywd, text}; m_current_symbol.m_kind.num = num;
560         } else
561             m_current_symbol = {scn::details::sk_ident, text};
562
563         return true;
564     }
565
566     bool scan_number (bool have_period = false, bool const is_negative = false) {
567         assert (scn::details::is_digit (m_current_chr));
568
569         int integer_part {0};
570
571         if (!have_period) {
572             while (scn::details::is_digit (m_current_chr)) {
573                 integer_part = integer_part * 10 + (m_current_chr - '0'); read_next_chr ();
574             }
575
576             if (m_current_chr == scn::details::sk_period.chr) {
577                 have_period = true; read_next_chr (scn::details::sk_period.chr);
578             }
579
580             m_current_symbol = {scn::details::sk_int, integer_part};
581         }
582
583         if (have_period) {
584             int fractional_part {0};
585             double exponent {1};
586
587             while (scn::details::is_digit (m_current_chr)) {
588                 fractional_part = fractional_part * 10 + (m_current_chr - '0');
589                 exponent *= 10;
```

```
590         read_next_chr ();
591     }
592 
593     m_current_symbol = {scn::details::sk_real, integer_part + fractional_part / exponent};
594 }
595 
596 if (is_negative)
597     scn::details::negate <int, double> (m_current_symbol.m_attr);
598 
599 return true;
600 }
601 
602 bool scan_string () {
603     read_next_chr (scn::details::sk_dquote.chr);
604 
605     bool scanned {false};
606     std::string text {};
607 
608     while ((m_current_chr != scn::details::sk_dquote.chr) && !is_eol_chr () && !is_eof_chr ()) {
609         text += m_current_chr; read_next_chr ();
610     }
611 
612     if ((scanned = (m_current_chr == scn::details::sk_dquote.chr)) == true) {
613         m_current_symbol = {scn::details::sk_str, text}; read_next_chr ();
614     } else
615         throw scn::exception {
616             "Expected_terminating_quote_in_string_but_have_"s + scn::details::to_string (
617                 m_current_chr) + ".";
618         };
619 
620     return scanned;
621 }
622 
623 bool try_read_number () {
624     if (scn::details::is_digit (m_current_chr))
625         return scan_number ();
626 
627     else if (m_current_chr == scn::details::sk_period.chr)
628         return try_read_number_from_period ();
629 
630     bool const have_minus {m_current_chr == scn::details::sk_minus.chr};
631     bool const have_plus {m_current_chr == scn::details::sk_plus .chr};
632 
633     read_next_chr ();
634 
635     if (m_signed_numbers)
636         if (scn::details::is_digit (m_current_chr))
637             return scan_number (false, have_minus);
638 
639     else if (m_current_chr == scn::details::sk_period.chr)
640         return try_read_number_from_period (have_minus);
641 
642     if (have_minus) m_current_symbol = scn::details::sk_minus; else
643     if (have_plus ) m_current_symbol = scn::details::sk_plus;
644 
645     return true;
646 }
647 
648 bool try_read_number_from_period (bool const is_negative = false) {
649     read_next_chr (scn::details::sk_period.chr);
650 
651     if (scn::details::is_digit (m_current_chr))
652         return scan_number (true, is_negative);
653 
654     m_current_symbol = scn::details::sk_period; return true;
655 }
656 
657 char m_current_chr {scn::details::sk_eof.chr}; //
658 symbol m_current_symbol {}; //
659 std::unordered_map <std::string, symbol> m_keywords {}; //
660 std::istream * m_p_in {nullptr}; // non
661     owning
662     bool m_signed_numbers {false}; //
```

663 } // namespace pfc