

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 (Programmier Sprachen) 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 Übung 1)!

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!



HSD

FH-HAGENBERG

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	SingetonBase.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 Variablennamens
- Speichern des Variablentyps
- Auswerten der Variablendeklaration (Syntaxprüfung)
- Zurückgeben des Variablennamens
- Zurückgeben des Variablentyps

Funktionen des Type:

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

3 Systementwurf

3.1 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.

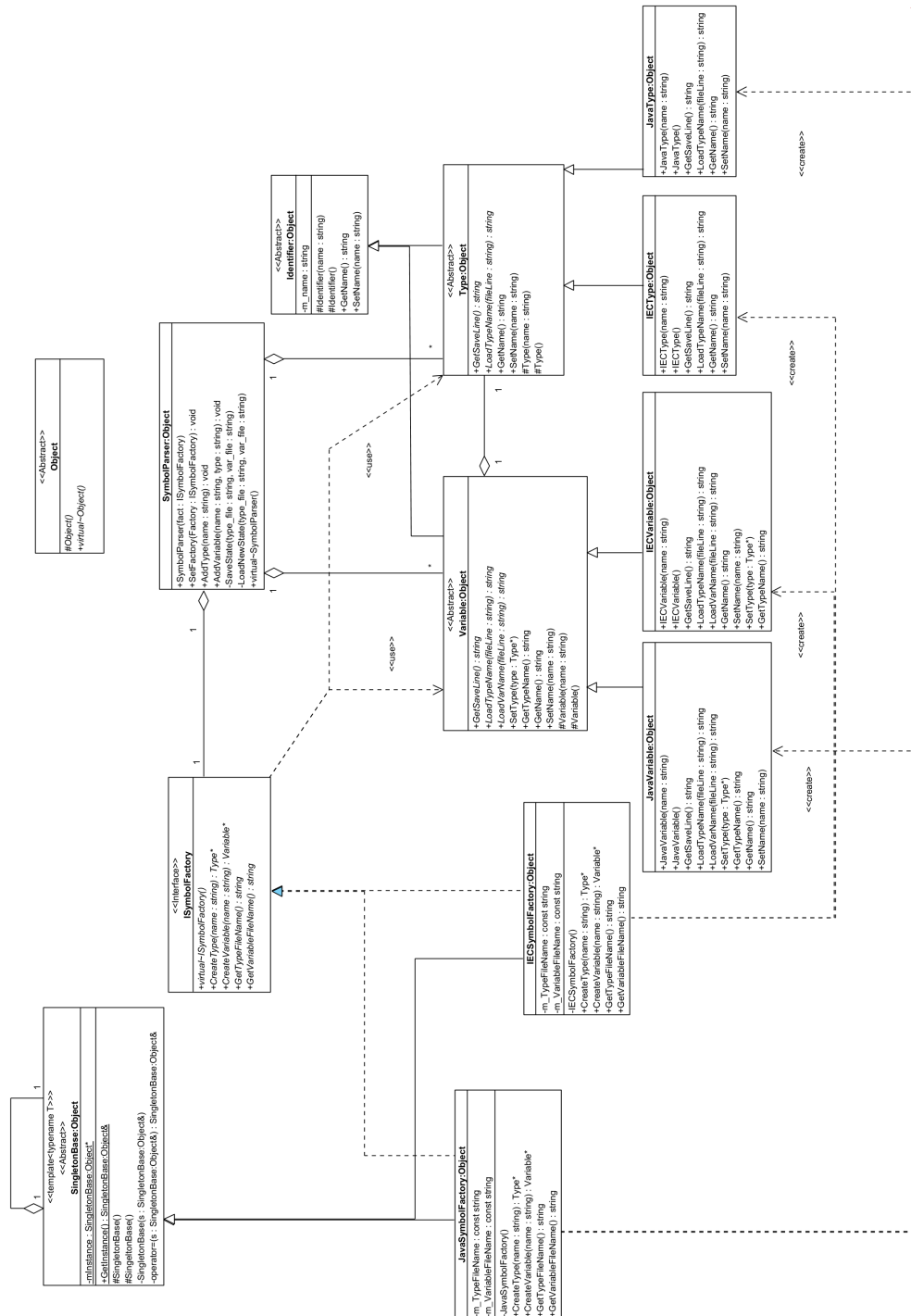
Container für Objekte:

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

```
36 Test Variable Get Type after set
37 [Test OK] Result: (Expected: uint_fast512_t == Result:
    ↪ uint_fast512_t)
38
39 Test for Exception in TestCase
40 [Test OK] Result: (Expected: true == Result: true)
41
42
43 *****
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:
    ↪ uint_fast_256_t)
62
63 Check for Exception in Testcase
64 [Test OK] Result: (Expected: true == Result: true)
65
66 Test Exception in Set Name
67 [Test OK] Result: (Expected: ERROR: Empty String == Result:
    ↪ ERROR: Empty String)
68
69 Test Exception in Set Type with nullptr
70 [Test OK] Result: (Expected: ERROR: Passed in Nullptr! ==
    ↪ Result: ERROR: Passed in Nullptr!)
71
72 Test Variable Get Type after set with nullptr
73 [Test OK] Result: (Expected: int == Result: int)
74
75 Test Variable Get Type after set
```

```
76 [Test OK] Result: (Expected: uint_fast512_t == Result:
    ↪ uint_fast512_t)
77
78 Test for Exception in TestCase
79 [Test OK] Result: (Expected: true == Result: true)
80
81
82 *****
83
84
85
86 **** Test IEC Type Getter ****
87
88
89 *****
90             TESTCASE START
91 *****
92
93 Test Type Get Name after Set
94 [Test OK] Result: (Expected: unit_1024_t == Result:
    ↪ unit_1024_t)
95
96 Test Exception in Set Type
97 [Test OK] Result: (Expected: true == Result: true)
98
99 Test Exception in Set Type
100 [Test OK] Result: (Expected: ERROR: Empty String == Result:
    ↪ ERROR: Empty String)
101
102
103 *****
104
105
106
107 **** Test Java Type Getter ****
108
109
110 *****
111             TESTCASE START
112 *****
113
114 Test Type Get Name after Set
115 [Test OK] Result: (Expected: unit_1024_t == Result:
    ↪ unit_1024_t)
```

```
116
117 Test Exception in Set Type
118 [Test OK] Result: (Expected: true == Result: true)
119
120 Test Exception in Set Type
121 [Test OK] Result: (Expected: ERROR: Empty String == Result:
    ↪ ERROR: Empty String)
122
123
124 *****
125
126
127 *****
128 TESTCASE START
129 *****
130
131 Test Load Type Name IEC Var
132 [Test OK] Result: (Expected: mCont == Result: mCont)
133
134 Test Load Var Name IEC Var
135 [Test OK] Result: (Expected: Speed_Controller == Result:
    ↪ Speed_Controller)
136
137 Test Load Type Name IEC Var invalid Format
138 [Test OK] Result: (Expected: == Result: )
139
140 Test Load Var Name IEC Var invalid Format
141 [Test OK] Result: (Expected: == Result: )
142
143 Test Load Type Name IEC Var invalid Format
144 [Test OK] Result: (Expected: mCont == Result: mCont)
145
146 Test Load Var Name IEC Var invalid Format
147 [Test OK] Result: (Expected: == Result: )
148
149 Test Load Type Name IEC Var invalid Format
150 [Test OK] Result: (Expected: == Result: )
151
152 Test Load Var Name IEC Var invalid Format
153 [Test OK] Result: (Expected: == Result: )
154
155 Test Load Type Name IEC Var invalid Format
156 [Test OK] Result: (Expected: mCont == Result: mCont)
157
```

```
158 Test Load Var Name IEC Var invalid Format
159 [Test OK] Result: (Expected: == Result: )
160
161 Test Load Type Name IEC Var invalid Format
162 [Test OK] Result: (Expected: == Result: )
163
164 Test Load Var Name IEC Var invalid Format
165 [Test OK] Result: (Expected: == Result: )
166
167 Test Load Type Name IEC Var invalid Format
168 [Test OK] Result: (Expected: == Result: )
169
170 Test Load Var Name IEC Var invalid Format
171 [Test OK] Result: (Expected: == Result: )
172
173 Test Save LineFormat IEC Variable
174 [Test OK] Result: (Expected: VAR mCont : 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 *****
250 TESTCASE START
251 *****
252
253 Test Load Type Name IEC Type
254 [Test OK] Result: (Expected: SpeedController == Result:
    ↪ 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: S2speedController == Result:
    ↪ S2speedController)
264
265 Test Load Type Name IEC Type invalid Format
266 [Test OK] Result: (Expected: == Result: )
267
268 Test Load Type Name IEC Type invalid Format
269 [Test OK] Result: (Expected: == Result: )
270
271 Test Save LineFormat IEC Type
272 [Test OK] Result: (Expected: TYPE SpeedController
273 == Result: TYPE SpeedController
274 )
275
276 Test for Exception in TestCase
277 [Test OK] Result: (Expected: true == Result: true)
278
279
280 *****
281
282
283 *****
284 TESTCASE START
285 *****
286
287 Test Load Type Name Java Type
```

```
288 [Test OK] Result: (Expected: SpeedController == Result:
    ↪ SpeedController)
289
290 Test Load Type Name Java Type invalid Format
291 [Test OK] Result: (Expected: == Result: )
292
293 Test Load Type Name Java Type invalid Format
294 [Test OK] Result: (Expected: == Result: )
295
296 Test Load Type Name Java Type invalid Format
297 [Test OK] Result: (Expected: S2speedController == Result:
    ↪ S2speedController)
298
299 Test Load Type Name Java Type invalid Format
300 [Test OK] Result: (Expected: == Result: )
301
302 Test Load Type Name Java Type invalid Format
303 [Test OK] Result: (Expected: == Result: )
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.
    ↪ == Result: ERROR: Provided string is empty.)
326
327 .AddVariable() - add empty type to factory
328 [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.
332         ↪ == Result: ERROR: Provided string is empty.)
333 .AddVariable() - add variable with nonexisting type
334 [Test OK] Result: (Expected: ERROR: Provided type does not
335         ↪ exist. == Result: ERROR: Provided type does not exist.)
336 .AddType() - add duplicate type
337 [Test OK] Result: (Expected: ERROR: Provided type already
338         ↪ exists. == Result: ERROR: Provided type already exists.)
339 .AddVar() - add duplicate Var
340 [Test OK] Result: (Expected: ERROR: Provided Variable already
341         ↪ exists. == Result: ERROR: Provided Variable already
342         ↪ exists.)
343 Test Store and Load Java Fact with exeption Dup Type
344 [Test OK] Result: (Expected: ERROR: Provided type already
345         ↪ exists. == Result: ERROR: Provided type already exists.)
346 Test Store and Load IEC Fact with exeption Dup Type
347 [Test OK] Result: (Expected: ERROR: Provided type already
348         ↪ exists. == Result: ERROR: Provided type already exists.)
349 *****
350
351 TEST OK!!
```

7 Quellcode

7.1 Object.hpp

```
1  /*****  
2  * \file   Object.hpp  
3  * \brief  common ancestor for all objects  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef OBJECT_HPP  
9  #define OBJECT_HPP  
10  
11 #include <string>  
12  
13 class Object {  
14 public:  
15  
16     // Exceptions constants  
17     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";  
18     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";  
19     inline static const std::string ERROR_NULLPTR = "ERROR:_Passed_in_Nullptr!";  
20  
21     // once virtual always virtual  
22     virtual ~Object() = default;  
23  
24 protected:  
25     Object() = default;  
26 };  
27  
28 #endif // !OBJECT_HPP  
29
```

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_NONEXISTING_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      * Loads 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      * Saves 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 SymbolFacotry to its file
81  * \param string of type files path, tring 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, tring 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  /***** SymbolParser.cpp *****/
2  * \file      SymbolParser.cpp
3  * \brief     A multi language parser for types and variables
4  * \author    Simon
5  * \date     Dezember 2025
6  *****/
7  #include <algorithm>
8  #include <fstream>
9  #include <iostream>
10 #include <cassert>
11 #include "SymbolParser.hpp"
12 #include "ISymbolFactory.hpp"
13
14 using namespace std;
15
16 void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)
17 {
18     assert(m_Factory != nullptr);
19
20     ofstream type_File;
21     ofstream var_File;
22
23     type_File.open(type_file);
24
25     // check if file is good
26     if (!type_File.good()) {
27         type_File.close();
28         return;
29     }
30
31     for_each(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& type) { type_File << type->
32         GetSaveLine(); });
33
34     type_File.close();
35
36     var_File.open(var_file);
37
38     // check if file is good
39     if (!var_File.good()) {
40         var_File.close();
41         return;
42     }
43
44     for_each(m_variableCont.cbegin(), m_variableCont.cend(), [&](const auto& var) { var_File << var->
45         GetSaveLine(); });
46
47     var_File.close();
48 }
49
50 void SymbolParser::LoadNewState(const std::string& type_file, const std::string& var_file)
51 {
52     assert(m_Factory != nullptr);
53
54     ifstream type_File;
55     ifstream var_File;
56
57     m_typeCont.clear();
58     m_variableCont.clear();
59
60     type_File.open(type_file);
61
62     // check if file is good
63     if (!type_File.good()) {
64         type_File.close();
65         return;
66     }
67
68     string line;
69
70     // Set Name can throw an exception
71     // 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     var_File.open(var_file);
90
91     // check if file is good
92     if (!var_File.good()) {
93         var_File.close();
94         return;
95     }
96
97     // Set Name can throw an exception
98     // the file must be closed !!!
99     try {
100         while (getline(var_File, line)) {
101
102             auto pVar = m_Factory->CreateVariable("");
103
104             assert(pVar != nullptr);
105
106             const string type = pVar->LoadTypeName(line);
107             const string name = pVar->LoadVarName(line);
108
109             pVar->SetName(name);
110
111             // look up if type even exists if yes add to type container
112             for (const auto& m_type : m_typeCont)
113             {
114                 if (type == m_type->GetName())
115                 {
116                     pVar->SetType(m_type);
117                 }
118             }
119
120             if (pVar->GetTypeName() != "") {
121                 m_variableCont.push_back(move(pVar));
122             }
123         }
124     }
125     catch (...) {
126         // file closes automatically due to RAII but here it is anyway
127         var_File.close();
128         throw; // rethrow
129     }
130
131     var_File.close();
132 }
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()
158         == name; });
159
160     if (it != m_typeCont.cend()) {
161         std::cerr << "Error_type_already_exists_!!_\n";
162         throw ERROR_DUPLICATE_TYPE;
163     }
164
165     Type::Uptr pType = m_Factory->CreateType(name);
166
167     if (pType == nullptr)
168         throw SymbolParser::ERROR_NULLPTR;
169
170     m_typeCont.push_back(move(pType));
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->GetType() == 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 exercise
```

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

7.5 Identifier.hpp

```
1  /***** Identifier.hpp *****/
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 protected:
36     Identifier(const std::string& name) : m_name{ name } {}
37     Identifier() = default;
38
39     std::string m_name;
40 private:
41 };
42
43
44 #endif
```

7.6 Identifier.cpp

```
1  /*****  
2  * \file   Identifier.cpp  
3  * \brief  Generalization of Types and Variables  
4  * \author  Simon  
5  * \date   Dezember 2025  
6  *****/  
7  
8  #include "Identifier.hpp"  
9  
10  
11  std::string Identifier::GetName() const {  
12      return m_name;  
13  }  
14  
15  void Identifier::SetName(const std::string& name)  
16  {  
17      if (name.empty()) throw Identifier::ERROR_EMPTY_STRING;  
18  
19      m_name = name;  
20  }
```

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::Sptr type);
56
57     /**
58      * \brief Name getter
59      *
60      * \return string of variable
61      */
62     std::string GetTypeName() const;
63
64 protected:
65     Variable(const std::string& name) : Identifier{ name } {}
66     Variable() = default;
67
68     Type::Sptr m_type;
69 private:

```

```
73 |;  
74 #endif
```

7.8 Variable.cpp

```
1  /*****  
2  * \file   Variable.cpp  
3  * \brief  Abstract class for parsing types  
4  * \author  Simon Vogelhuber  
5  * \date   Dezember 2025  
6  *****/  
7  
8  #include "Variable.hpp"  
9  
10 using namespace std;  
11  
12 void Variable::SetType(Type::Sptr type)  
13 {  
14     if (type == nullptr) throw Type::ERROR_NULLPTR;  
15  
16     m_type = std::move(type);  
17 }  
18  
19 std::string Variable::GetTypeName() const  
20 {  
21     return m_type->GetName();  
22 }
```

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 protected:
44     Type(const std::string& name) : Identifier{ name } {}
45     Type() = default;
46 private:
47 };
48 #endif
```

7.10 SingletonBase.hpp

```
1  /*****
2  * \file   SingletonBase.hpp
3  * \brief  Base Class for creating singletons
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef SINGLETON_BASE_HPP
9  #define SINGLETON_BASE_HPP
10
11 #include "Object.h"
12 #include <memory>
13
14 template <typename T> class SingletonBase : public Object {
15 public:
16     /**
17      * \brief Getter for static member Singleton.
18      *
19      * \return
20      */
21     static T& GetInstance() {
22         if (mInstance == nullptr) { mInstance = std::unique_ptr<T>{ new T{} }; };
23         return *mInstance;
24     }
25 protected:
26     SingletonBase() = default;
27     virtual ~SingletonBase() = default;
28
29 private:
30     SingletonBase(SingletonBase const& s) = delete;
31     SingletonBase& operator = (SingletonBase const& s) = delete;
32     static std::unique_ptr<T> mInstance;
33 };
34
35 template <typename T> std::unique_ptr<T> SingletonBase<T>::mInstance = nullptr;
36
37 #endif // !SINGLETON_BASE_HPP
```

7.11 JavaType.hpp

```
1  /*****  
2  * \file   JavaType.hpp  
3  * \brief  A Class for parsing java types  
4  * \author Simon  
5  * \date   Dezember 2025  
6  *****/  
7  
8  #ifndef JAVA_TYPE_HPP  
9  #define JAVA_TYPE_HPP  
10  
11 #include "Type.hpp"  
12 #include <string>  
13  
14 class JavaType : public Type  
15 {  
16 public:  
17     /**  
18     * \brief Loads a types name from a files line  
19     *  
20     * \param string fileLine  
21     * \return string of type  
22     */  
23     virtual std::string LoadTypeName(const std::string& fileLine) const override;  
24  
25     /**  
26     * \brief Returns formatted line of a types declaration  
27     *  
28     * \return string of type declaration  
29     */  
30     virtual std::string GetSaveLine() const override;  
31  
32     JavaType(const std::string name) : Type{ name } {}  
33  
34     JavaType() = default;  
35  
36 protected:  
37 private:  
38 };  
39 #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         // catch Scanner Exceptions
37         catch (...) {
38             return "";
39         }
40     }
41     return "";
42 }
43
44
45
46 std::string JavaType::LoadTypeName(const std::string& fileLine) const
47 {
48     stringstream sstream;
49     sstream << fileLine;
50     scanner scan;
51     scan.set_istream(sstream);
52     return ScanTypeName(scan);
53 }
54
55
56
57
58
59 std::string JavaType::GetSaveLine() const
60 {
61     return "class_" + m_name + "\n";
62 }
```

7.13 JavaVariable.hpp

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

7.14 JavaVariable.cpp

```
1  /*****
2  * \file   JavaVariable.cpp
3  * \brief  A Class for parsing java variables
4  * \author Simon
5  * \date   Dezember 2025
6  *****/
7
8  #include "JavaVariable.hpp"
9  #include <sstream>
10 #include <string>
11 #include "scanner.h"
12
13 using namespace pfc;
14 using namespace std;
15
16 /**
17  * \brief Scans an input string for the Type name of the Var.
18  *
19  * \param scan Reference to scanner object
20  * \return empty string if no valid type name is found
21  * \return name of type
22  */
23 static std::string ScanTypeName(scanner& scan)
24 {
25     string typeName;
26
27     try{
28         typeName = scan.get_identifier();
29         scan.next_symbol();
30     }
31     // catch Scanner Exceptions
32     catch (...) {
33         return "";
34     }
35
36     return typeName;
37 }
38
39 /**
40  * \brief Scans an input string for the Variable name of the Var.
41  *
42  * \param scan Reference to scanner object
43  * \return empty string if no valid Variable name is found
44  * \return name of Variable
45  */
46 static std::string ScanVarName(scanner& scan)
47 {
48     string varName;
49
50     try{
51         varName = scan.get_identifier();
52         scan.next_symbol();
53
54         // The line should be empty after the var Name!
55         if (!scan.is(';')) varName = "";
56     }
57     // catch Scanner Exceptions
58     catch (...) {
59         return "";
60     }
61
62     return varName;
63 }
64
65 std::string JavaVariable::GetSaveLine() const
66 {
67     if (m_type == nullptr) return "";
68
69     return m_type->GetName() + "_" + m_name + ";\n";
70 }
71
72 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  /*****
2  * \file   JavaSymbolFactory.hpp
3  * \brief  A factory for creating java variables and types
4  * \author Simon
5  * \date   Dezember 2025
6  *****/
7  #ifndef JAVA_SYMBOL_FACTORY_HPP
8  #define JAVA_SYMBOL_FACTORY_HPP
9
10 #include "ISymbolFactory.hpp"
11 #include "SingletonBase.hpp"
12
13 class JavaSymbolFactory :public ISymbolFactory, public SingletonBase<JavaSymbolFactory>
14 {
15 public:
16
17     friend class SingletonBase<JavaSymbolFactory>;
18
19     /**
20     * \brief Creates a java variable
21     *
22     * \param string of variables name
23     * \return unique pointer to variable
24     */
25     virtual Variable::Uptr CreateVariable(const std::string& name) const override;
26
27     /**
28     * \brief Creates a java type
29     *
30     * \param string of typename
31     * \return unique pointer to type
32     */
33     virtual Type::Uptr CreateType(const std::string& name) const override;
34
35     /**
36     * \brief Getter for file path of type file
37     *
38     * \return string of filePath
39     */
40     virtual const std::string& GetTypeFileName() const override;
41
42     /**
43     * \brief Getter for file path of variable file
44     *
45     * \return string of filePath
46     */
47     virtual const std::string& GetVariableFileName() const override;
48
49     // delete CopyCtor and Assign operator to prevent untestet behaviour
50     JavaSymbolFactory(JavaSymbolFactory& fact) = delete;
51     void operator=(JavaSymbolFactory fact) = delete;
52
53 protected:
54 private:
55     JavaSymbolFactory() = default;
56     const std::string m_TypeFileName = "JavaTypes.sym";
57     const std::string m_VariableFileName = "JavaVars.sym";
58 };
59 #endif
```

7.16 JavaSymbolFactory.cpp

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

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  /*****
2  * \file    IECType.cpp
3  * \brief   A Class for parsing IEC types
4  * \author  Simon
5  * \date    Dezember 2025
6  *****/
7
8  #include "IECType.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() == "TYPE") {
29             scan.next_symbol();
30             TypeName = scan.get_identifier();
31             scan.next_symbol();
32             if (!scan.has_symbol()) {
33                 return TypeName;
34             }
35         }
36         // catch Scanner Exceptions
37         catch (...) {
38             return "";
39         }
40     }
41     return "";
42 }
43
44
45
46 std::string IECType::LoadTypeName(const std::string& fileLine) const
47 {
48     stringstream sstream;
49     sstream << fileLine;
50
51     scanner scan;
52     scan.set_istream(sstream);
53
54     return ScanTypeName(scan);
55 }
56
57
58
59
60 std::string IECType::GetSaveLine() const
61 {
62     return "TYPE_" + m_name + "\n";
63 }
```

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 }
```

```
73         return VarName;
74     }
75
76
77
78     std::string IECVariable::LoadTypeName(std::string const& fileLine) const
79     {
80         stringstream converter;
81         converter << fileLine;
82         scanner Scan;
83
84         Scan.set_istream(converter);
85
86         return ScanTypeName(Scan);
87     }
88
89     std::string IECVariable::LoadVarName(std::string const& fileLine) const
90     {
91         stringstream converter;
92         converter << fileLine;
93         scanner Scan;
94
95         Scan.set_istream(converter);
96
97         string Typename = ScanTypeName(Scan);
98         string VarName = ScanVarName(Scan);
99
100         if (Typename.empty()) VarName = "";
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 untestet 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  /*****  
2  * \file    IECSymbolFactory.cpp  
3  * \brief   A factory for creating IEC variables and types  
4  * \author  Simon  
5  * \date    Dezember 2025  
6  *****/  
7  
8  #include "IECSymbolFactory.hpp"  
9  #include "IECType.hpp"  
10 #include "IECVariable.hpp"  
11  
12  
13 Variable::Uptr IECSymbolFactory::CreateVariable(const std::string& name) const  
14 {  
15     return std::make_unique<IECVariable>(name);  
16 }  
17  
18 Type::Uptr IECSymbolFactory::CreateType(const std::string& name) const  
19 {  
20     return std::make_unique<IECType>(name);  
21 }  
22  
23 const std::string& IECSymbolFactory::GetTypeFileName() const  
24 {  
25     return m_TypeFileName;  
26 }  
27  
28 const std::string& IECSymbolFactory::GetVariableFileName() const  
29 {  
30     return m_VariableFileName;  
31 }
```

7.23 main.cpp

```
1  /*****
2  * \file    Main.cpp
3  * \brief   Testdriver for Symbol Parser and all connected Classes
4  * \author   Simon
5  * \date    November 2025
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 tests 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
63     bool TestOK = true;
64
65     ofstream output{ "output.txt" };
66
67
68     try {
69         Type::Sptr Itype{ make_shared<IECType>(IECType{ "int" }) };
70
71         Type::Sptr Jtyp{ make_shared<JavaType>(JavaType{ "int" }) };
72     }
```

```

73     IECVariable IECVar{ "asdf" };
74     IECVar.SetType(Itype);
75
76     JavaVariable JavaVar{ "jklm" };
77     JavaVar.SetType(Jtyp);
78
79     cout << "\n\n****_Test_IEC_Var_Getter_****\n\n";
80     TestOK = TestOK && TestVariable(&IECVar, "asdf", Itype);
81
82     cout << "\n\n****_Test_Java_Var_Getter_****\n\n";
83     TestOK = TestOK && TestVariable(&JavaVar, "jklm", Jtyp);
84
85     cout << "\n\n****_Test_IEC_Type_Getter_****\n\n";
86     TestOK = TestOK && TestType(Itype);
87
88     cout << "\n\n****_Test_Java_Type_Getter_****\n\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 previos 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\n****_Test_IEC_Var_Getter_****\n\n";
123         TestOK = TestOK && TestVariable(&IECVar1, "asdf", Itypel, output);
124
125         output << "\n\n****_Test_Java_Var_Getter_****\n\n";
126         TestOK = TestOK && TestVariable(&JavaVar1, "jklm", Jtyp1, output);
127
128         output << "\n\n****_Test_IEC_Type_Getter_****\n\n";
129         TestOK = TestOK && TestType(Itypel, output);
130
131         output << "\n\n****_Test_Java_Type_Getter_****\n\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->GetType());
194
195         const string var_name = "uint_fast_256_t";
196
197         var->SetName(var_name);
198
199         TestOK = TestOK && check_dump(ost, "Test_Variable_Set_Name", var_name, var->GetName());
200
201     }
202     catch (const string& err) {
203         error_msg = err;
204     }
205     catch (bad_alloc const& error) {
206         error_msg = error.what();
207     }
208     catch (const exception& err) {
209         error_msg = err.what();
210     }
211     catch (...) {
212         error_msg = "Unhandelt_Exception";
213     }
214
215     TestOK = TestOK && check_dump(ost, "Check_for_Exception_in_Testcase", true, error_msg.empty());
216     error_msg.clear();
217
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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
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->GetType());
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             GetType());
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 = "Unhandelt_Exception";
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
351     bool TestOK = true;
352     string error_msg;
353
354     try {
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"), var.LoadVarName(LineToDecode));
361
362         const string InvLineToDecode = "1VAR_mCont_:SpeedController;";

```

```

363     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadTypeName(InvLineToDecode));
364     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadVarName(InvLineToDecode));
365
366     const string Inv2LineToDecode = "VAR_mCont_:SpeedController";
367     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
        string>("mCont"), var.LoadTypeName(Inv2LineToDecode));
368     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadVarName(Inv2LineToDecode));
369
370     const string Inv3LineToDecode = "Var_mCont_:SpeedController;";
371     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadTypeName(Inv3LineToDecode));
372     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadVarName(Inv3LineToDecode));
373
374     const string Inv4LineToDecode = "VAR_mCont_:12343;";
375     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
        string>("mCont"), var.LoadTypeName(Inv4LineToDecode));
376     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadVarName(Inv4LineToDecode));
377
378     const string Inv5LineToDecode = "VAR_123_:a12343;";
379     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadTypeName(Inv5LineToDecode));
380     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadVarName(Inv5LineToDecode));
381
382     const string Inv6LineToDecode = "";
383     TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadTypeName(Inv6LineToDecode));
384     TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_IEC_Var_invalid_Format", static_cast<
        string>(""), var.LoadVarName(Inv6LineToDecode));
385
386     Type::Sptr IECType = make_shared<IECType>( IECType{} );
387     var.SetName(var.LoadVarName(LineToDecode));
388     IECType->SetName(var.LoadTypeName(LineToDecode));
389     var.SetType(IECType);
390
391     TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", LineToDecode, var.
        GetSaveLine());
392
393     IECVariable IVar;
394
395     TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", static_cast<string>("")
        ), IVar.GetSaveLine());
396
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 = "Unhandelt_Exception";
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>(""), 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>(""), 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>(""), 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>(""), var.LoadTypeName(Inv5LineToDecode));
466         TestOK == TestOK && check_dump(ost, "Test_Load_Var_Name_Java_Var_invalid_Format", static_cast<
467                                         string>(""), 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 JType = make_shared<JavaType>(JavaType{});
476         var.SetName(var.LoadVarName(LineToDecode));
477         JType->SetName(var.LoadTypeName(LineToDecode));
478         var.SetType(JType);
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 = "Unhandelt_Exception";
499     }

```

```
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
500     bool TestOK = true;
501     string error_msg;
502
503     try{
504         IECType typ;
505
506         const string LineToDecode = "TYPE_SpeedController\n";
507         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type", static_cast<string>("
SpeedController"), typ.LoadTypeName(LineToDecode));
508
509         const string InvLineToDecode = "1TYPE_SpeedController";
510         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type_invalid_Format", static_cast<
string>(""), typ.LoadTypeName(InvLineToDecode));
511
512         const string Inv2LineToDecode = "TYPE_1SpeedController";
513         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type_invalid_Format", static_cast<
string>(""), typ.LoadTypeName(Inv2LineToDecode));
514
515         const string Inv3LineToDecode = "TYPE_S2peedController";
516         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type_invalid_Format", static_cast<
string>("S2peedController"), typ.LoadTypeName(Inv3LineToDecode));
517
518         const string Inv4LineToDecode = "TYPE_SpeedController;";
519         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type_invalid_Format", static_cast<
string>(""), typ.LoadTypeName(Inv4LineToDecode));
520
521         const string Inv6LineToDecode = "";
522         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_IEC_Type_invalid_Format", static_cast<
string>(""), typ.LoadTypeName(Inv6LineToDecode));
523
524         typ.SetName(typ.LoadTypeName(LineToDecode));
525
526         TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Type", LineToDecode, typ.
GetSaveLine());
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 = "lclass_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_lSpeedController";
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_S2peedController";
574         TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast
<string>("S2peedController"), 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 = "Unhandelt_Exception";
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 = "Unhandelt_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     }
```

```
72     }
73     catch (...) {
74         error_msg = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
141     }
142
143     TestOK = TestOK && check_dump(ost, ".AddVariable()_-add_variable_with_nonexisting_type",
144         SymbolParser::ERROR_NONEXISTING_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 = "Unhandelt_Exception";
160 }
161
162 TestOK = TestOK && check_dump(ost, ".AddType()_add_duplicate_type", SymbolParser::
    ERROR_DUPLICATE_TYPE, error_msg);
163 error_msg.clear();
164
165 // addVariable add variable for non existing type
166 try {
167     SymbolParser parser{ JavaSymbolFactory::GetInstance() };
168     parser.AddType("uint4096_t");
169     parser.AddVariable("Large_int", "uint4096_t");
170     parser.AddVariable("Large_int", "uint4096_t");
171 }
172 catch (const string& err) {
173     error_msg = err;
174 }
175 catch (bad_alloc const& error) {
176     error_msg = error.what();
177 }
178 catch (const exception& err) {
179     error_msg = err.what();
180 }
181 catch (...) {
182     error_msg = "Unhandelt_Exception";
183 }
184
185 TestOK = TestOK && check_dump(ost, ".AddVar()_add_duplicate_Var", SymbolParser::
    ERROR_DUPLICATE_VAR, error_msg);
186 error_msg.clear();
187
188
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 = "Unhandelt_Exception";
212 }
213
214 TestOK = TestOK && check_dump(ost, "Test_Store_and_Load_Java_Fact_with_exception_Dup_Type",
    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_exception_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  /*****  
2  * \file   Test.hpp  
3  * \brief  File that provides a Test Function with a formatted output  
4  *  
5  * \author Simon  
6  * \date   April 2025  
7  *****/  
8  #ifndef TEST_HPP  
9  #define TEST_HPP  
10  
11 #include <string>  
12 #include <iostream>  
13 #include <vector>  
14 #include <list>  
15 #include <queue>  
16 #include <forward_list>  
17  
18 #define ON 1  
19 #define OFF 0  
20 #define COLOR_OUTPUT OFF  
21  
22 // Definitions of colors in order to change the color of the output stream.  
23 const std::string colorRed = "\x1B[31m";  
24 const std::string colorGreen = "\x1B[32m";  
25 const std::string colorWhite = "\x1B[37m";  
26  
27 inline std::ostream& RED(std::ostream& ost) {  
28     if (ost.good()) {  
29         ost << colorRed;  
30     }  
31     return ost;  
32 }  
33 inline std::ostream& GREEN(std::ostream& ost) {  
34     if (ost.good()) {  
35         ost << colorGreen;  
36     }  
37     return ost;  
38 }  
39 inline std::ostream& WHITE(std::ostream& ost) {  
40     if (ost.good()) {  
41         ost << colorWhite;  
42     }  
43     return ost;  
44 }  
45  
46 inline std::ostream& TestStart(std::ostream& ost) {  
47     if (ost.good()) {  
48         ost << std::endl;  
49         ost << "*****" << std::endl;  
50         ost << "      TESTCASE_START      " << std::endl;  
51         ost << "*****" << std::endl;  
52         ost << std::endl;  
53     }  
54     return ost;  
55 }  
56  
57 inline std::ostream& TestEnd(std::ostream& ost) {  
58     if (ost.good()) {  
59         ost << std::endl;  
60         ost << "*****" << std::endl;  
61         ost << std::endl;  
62     }  
63     return ost;  
64 }  
65  
66 inline std::ostream& TestCaseOK(std::ostream& ost) {  
67     #if COLOR_OUTPUT  
68         if (ost.good()) {  
69             ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;  
70         }  
71     #else  
72
```

```

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         }
112         else {
113             ostr << testcase << std::endl << colorRed << "[Test_FAILED]" << colorWhite <<
114                 "Result:(Expected:" << std::boolalpha << expected << "!=" << "Result:" <<
115                 result << ")" << std::noboolalpha << std::endl << std::endl;
116         }
117 #else
118         if (expected == result) {
119             ostr << testcase << std::endl << "[Test_OK]" << "Result:(Expected:" << std::
120                 boolalpha << expected << "==" << "Result:" << result << ")" << std::
121                 noboolalpha << std::endl << std::endl;
122         }
123         else {
124             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:(Expected:" <<
125                 std::boolalpha << expected << "!=" << "Result:" << result << ")" <<
126                 std::noboolalpha << std::endl << std::endl;
127         }
128 #endif
129
130         if (ostr.fail()) {
131             std::cerr << "Error:_Write_Ostream" << std::endl;
132         }
133     }
134     else {
135         std::cerr << "Error:_Bad_Ostream" << std::endl;
136     }
137     return expected == result;
138 }
139
140 template <typename T1, typename T2>
141 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
142     if (!ost.good()) throw std::exception("Error:_bad_Ostream!");
143     ost << "(" << p.first << "," << p.second << ")";
144     return ost;
145 }

```

```
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, "\n"));
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, "\n"));
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, "\n"));
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, "\n"));
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/cephelden/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 }
```

```

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

```

```

147
148 constexpr bool operator == (symbol const & rhs) const {
149     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> () : "";
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> () : "";
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> () : "";
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
297     symbol const & next_symbol (char const chr) {
298         if (!is (chr))
299             throw scn::exception {
300                 "Expected_' "s + scn::details::to_string (chr) + "'_but_have_" + to_string (
301                     m_current_symbol) + '.'.
302             };
303         return next_symbol ();
304     }
305
306     symbol const & next_symbol (std::string const & name) {
307         if (!is_keyword (name))
308             throw scn::exception {
309                 "Expected_keyword_' "s + name + "'_but_have_" + to_string (m_current_symbol) + '.'.
310             };
311         return next_symbol ();
312     }
313
314     template <typename attr_t> constexpr attr_t const & get_attribute () const {
315         return m_current_symbol.get_attribute <attr_t> ();
316     }
317
318     std::string get_identifier () const {
319         return m_current_symbol.get_identifier ();
320     }
321
322     constexpr int get_integer () const {
323         return m_current_symbol.get_integer ();
324     }
325
326     constexpr double get_number () const {
327         return m_current_symbol.get_number ();
328     }
329
330     constexpr double get_real () const {
331         return m_current_symbol.get_real ();
332     }
333
334     std::string get_string () const {
335         return m_current_symbol.get_string ();
336     }
337
338     constexpr bool is (char const c) const {
339         return m_current_symbol.is (c);
340     }
341
342     constexpr bool is_eof () const {
343         return m_current_symbol.is_eof ();
344     }
345
346     constexpr bool is_identifier () const {
347         return m_current_symbol.is_identifier ();
348     }
349
350     constexpr bool is_integer () const {
351         return m_current_symbol.is_integer ();
352     }
353
354     constexpr bool is_keyword () const {
355         return m_current_symbol.is_keyword ();
356     }
357
358     constexpr bool is_keyword (std::size_t const num) const {
359         return m_current_symbol.is_keyword (num);
360     }
361
362     bool is_keyword (std::string const & name) const {
363         return m_current_symbol.is_keyword (name);
364     }
365
366     constexpr bool is_number () const {
367         return m_current_symbol.is_number ();
368     }
369 }
```

```

370
371     constexpr bool is_real () const {
372         return m_current_symbol.is_real ();
373     }
374
375     constexpr bool is_string () const {
376         return m_current_symbol.is_string ();
377     }
378
379     constexpr bool has_symbol () const {
380         return !is_eof ();
381     }
382
383     std::size_t register_keyword (std::string const & name) {
384         if (std::empty (name))
385             return 0;
386
387         if (auto const f {m_keywords.find (name)}; f != std::end (m_keywords))
388             return f->second.m_kind.num;
389
390         auto const sym {make_keyword_symbol (std::size (m_keywords) + 1, name)};
391
392         m_keywords[name] = sym;
393
394         if (is_identifier () && (current_symbol ().get_identifier () == name))
395             m_current_symbol = sym;
396
397         return sym.m_kind.num;
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
446 constexpr bool is_whitespace_chr () const {
447     return (m_current_chr == scn::details::sk_space.chr) || (m_current_chr == scn::details::sk_tab
448         .chr) || is_eol_chr ();
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         } else {
511             m_current_symbol = scn::details::to_symbol_kind (m_current_chr);
512
513             if ((scanned = m_current_symbol.is_terminal_symbol ()) == false)
514                 throw scn::exception {"Unknown_character_' "s + m_current_chr + "'_encountered."};
515
516             read_next_chr ();
517         }
518     }
```

```
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
550 bool scan_keyword_or_identifier () {
551     assert (scn::details::is_alpha (m_current_chr));
552
553     std::string text;
554
555     while (scn::details::is_alpha (m_current_chr) || scn::details::is_digit (m_current_chr)) {
556         text += m_current_chr; read_next_chr ();
557     }
558
559     if (auto const num {keyword_is_registered (text)}; num > 0) {
560         m_current_symbol = {scn::details::sk_keywd, text}; m_current_symbol.m_kind.num = num;
561     } else
562         m_current_symbol = {scn::details::sk_ident, text};
563
564     return true;
565 }
566
567 bool scan_number (bool have_period = false, bool const is_negative = false) {
568     assert (scn::details::is_digit (m_current_chr));
569
570     int integer_part {0};
571
572     if (!have_period) {
573         while (scn::details::is_digit (m_current_chr)) {
574             integer_part = integer_part * 10 + (m_current_chr - '0'); read_next_chr ();
575         }
576
577         if (m_current_chr == scn::details::sk_period.chr) {
578             have_period = true; read_next_chr (scn::details::sk_period.chr);
579         }
580
581         m_current_symbol = {scn::details::sk_int, integer_part};
582     }
583
584     if (have_period) {
585         int fractional_part {0};
586         double exponent {1};
587
588         while (scn::details::is_digit (m_current_chr)) {
589             fractional_part = fractional_part * 10 + (m_current_chr - '0');
```

```

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_" + 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
```