HSD
FH-HAGENBERG

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:

```cpp
class SymbolParser : public Object
{
public:
    ...
    void AddType(std::string const& name);
    void AddVariable(std::string const& name, std::string const& type);
    void SetFactory(...);
protected:
    ...
private:
    ...
};
```

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*

1

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

JavaTypes.sym:

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

JavaVars.sym:

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

IECTypes.sym:

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

IECVars.sym:

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

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

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

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

Eine mögliche Anwendung im Hauptprogramm könnte so aussehen:

```
#include "SymbolParser.h"
#include "JavaSymbolFactory.h"
#include "IECSymbolFactory.h"


int main()
{
```

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

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

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

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

HSD

FH-HAGENBERG

Systemdokumentation
Projekt Symbolparser

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 16. November 2025

4

Inhaltsverzeichnis

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

        * 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

    - ∗ Variable

    - ∗ Type

    - ∗ JavaVariable

    - ∗ JavaType

    - ∗ IECVariable

    - ∗ IECType

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 Faktory 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 hierführ 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

<<Abstract>> Object

#Object()
+virtual~Object()

<<template<typename T>>> <<Abstract>> SingletonBase::Object

-mInstance : SingletonBase::Object*

#GetInstance() : SingletonBase::Object&
#SingletonBase()
#SingletonBase()
-SingletonBase(s : SingletonBase::Object&)
-operator=(s : SingletonBase::Object&) : SingletonBase::Object&

<<Interface>> ISymbolFactory

+virtual~ISymbolFactory()
+CreateType(name : string) : Type*
+CreateVariable(name : string) : Variable*
+GetTypeFileName() : string
+GetVariableFileName() : string

IECSymbolFactory::Object

-m_TypeFileName : const string
-m_VariableFileName : const string

IECSymbolFactory()
+CreateType(name : string) : Type*
+CreateVariable(name : string) : Variable*
+GetTypeFileName() : string
+GetVariableFileName() : string

JavaSymbolFactory::Object

-m_TypeFileName : const string
-m_VariableFileName : const string

-JavaSymbolFactory()
+CreateType(name : string) : Type*
+CreateVariable(name : string) : Variable*
+GetTypeFileName() : string
+GetVariableFileName() : string

SymbolParser::Object

+SymbolParser(fact : ISymbolFactory)
+SetFactory(Factory : ISymbolFactory) : void
+AddType(name : string) : void
+AddVariable(name : string, type : string) : void
-SaveState(type_file : string, var_file : string)
-LoadNewState(type_file : string, var_file : string)
+virtual~SymbolParser()

<<Abstract>> Identifier:Object

-m_name : string

#Identifier(name : string)
#Identifier()
+GetName() : string
+SetName(name : string)

<<Abstract>> Variable:Object

+GetSaveLine() : string
+LoadTypeName(fileLine : string) : string
+LoadVarName(fileLine : string) : string
+SetType(type : Type*)
+GetTypeName() : string
#Variable(name : string)
#Variable()

<<Abstract>> Type:Object

+GetSaveLine() : string
+LoadTypeName(fileLine : string) : string
+GetName() : string
+SetName(name : string)
#Type(name : string)
#Type()

IECVariable:Object

+IECVariable(name : string)
+IECVariable()
+GetSaveLine() : string
+LoadTypeName(fileLine : string) : string
+LoadVarName(fileLine : string) : string
+GetName() : string
+SetName(name : string)
+SetType(type : Type*)
+GetTypeName() : string

JavaVariable:Object

+JavaVariable(name : string)
+JavaVariable()
+GetSaveLine() : string
+LoadTypeName(fileLine : string) : string
+LoadVarName(fileLine : string) : string
+SetType(type : Type*)
+GetTypeName() : string
+GetName() : string
+SetName(name : string)

IECType:Object

+IECType(name : string)
+IECType()
+GetSaveLine() : string
+LoadTypeName(fileLine : string) : string
+GetName() : string
+SetName(name : string)

JavaType:Object

+JavaType(name : string)
+JavaType()
+GetSaveLine() : string
+LoadTypeName(fileLine : string) : string
+GetName() : string
+SetName(name : string)

<<use>>   <<create>>

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 speicher.n Die Datei ist folgendermaßen aufgebaut:

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

4.2 Datei: JavaVars.sym

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

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

4.3 Datei: IECTypes.sym

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

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

4.4 Datei: IECVars.sym

Diese Datei wird verwendet um die vom Symbolparser verwaltete IEC Varia-
blen 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!

```
*******************************************
              TESTCASE START
*******************************************



**** Test IEC Var Getter ****



*******************************************
              TESTCASE START
*******************************************

Test Variable Get Name
[Test OK] Result: (Expected: asdf == Result: asdf)

Test Variable Get Type
[Test OK] Result: (Expected: int == Result: int)

Test Variable Set Name
[Test OK] Result: (Expected: uint_fast_256_t == Result:
    ↪ uint_fast_256_t)

Check for Exception in Testcase
[Test OK] Result: (Expected: true == Result: true)

Test Exception in Set Name
[Test OK] Result: (Expected: ERROR: Empty String == Result:
    ↪ ERROR: Empty String)

Test Exception in Set Type with nullptr
[Test OK] Result: (Expected: ERROR: Passed in Nullptr! ==
    ↪ Result: ERROR: Passed in Nullptr!)

Test Variable Get Type after set with nullptr
[Test OK] Result: (Expected: int == Result: int)

```

```
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: S2peedController == Result:
     |   ↪ S2peedController)
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: S2peedController == Result:
        ↪ S2peedController)
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.
```

```
        ↪ == Result: ERROR: Provided string is empty.)
329
330 .AddVariable() - add empty var to factory
331 [Test OK] Result: (Expected: ERROR: Provided string is empty.
        ↪ == Result: ERROR: Provided string is empty.)
332
333 .AddVariable() - add variable with nonexisting type
334 [Test OK] Result: (Expected: ERROR: Provided type does not
        ↪ exist. == Result: ERROR: Provided type does not exist.)
335
336 .AddType() - add duplicate type
337 [Test OK] Result: (Expected: ERROR: Provided type already
        ↪ exists. == Result: ERROR: Provided type already exists.)
338
339 .AddVar() - add duplicate Var
340 [Test OK] Result: (Expected: ERROR: Provided Variable already
        ↪ exists. == Result: ERROR: Provided Variable already
        ↪ exists.)
341
342 Test Store and Load Java Fact with exeption Dup Type
343 [Test OK] Result: (Expected: ERROR: Provided type already
        ↪ exists. == Result: ERROR: Provided type already exists.)
344
345 Test Store and Load IEC Fact with exeption Dup Type
346 [Test OK] Result: (Expected: ERROR: Provided type already
        ↪ exists. == Result: ERROR: Provided type already exists.)
347
348
349 ******************************************
350
351 TEST OK!!
```

7 Quellcode

7.1 Object.hpp

```cpp
/*******************************************************************//**
 * \file   Object.hpp
 * \brief  common ancestor for all objects
 *
 * \author Simon
 * \date   November 2025
 ***********************************************************************/
#ifndef OBJECT_HPP
#define OBJECT_HPP

#include <string>

class Object {
public:

        // Exceptions constants
        inline static const std::string ERROR_BAD_OSTREAM = "ERROR: Provided Ostream is bad";
        inline static const std::string ERROR_FAIL_WRITE = "ERROR: Fail to write on provided Ostream";
        inline static const std::string ERROR_NULLPTR = "ERROR: Passed in Nullptr!";

        // once virtual always virtual
        virtual ~Object() = default;


protected:
        Object() = default;
};

#endif // !OBJECT_HPP
```

7.2 **Symbolparser.hpp**

```cpp
/****************************************************************//**
 * \file   SymbolParser.hpp
 * \brief  A multi language parser for types and variables
 * \author Simon
 * \date   Dezember 2025
 ****************************************************************/

#ifndef SYMBOL_PARSER_HPP
#define SYMBOL_PARSER_HPP

#include <vector>

#include "Object.h"
#include "Variable.hpp"
#include "Type.hpp"
#include "ISymbolFactory.hpp"

class SymbolParser : public Object
{
public:
    inline static const std::string ERROR_EMPTY_STRING = "ERROR: Provided string is empty.";
    inline static const std::string ERROR_NONEXISTING_TYPE = "ERROR: Provided type does not exist.";
    inline static const std::string ERROR_DUPLICATE_TYPE = "ERROR: Provided type already exists.";
    inline static const std::string ERROR_DUPLICATE_VAR = "ERROR: Provided Variable already exists.";

    /**
     * \brief Polymorphic conatiner for saving variables
     */
    using TVariableCont = std::vector<Variable::Uptr>;

    /**
     * \brief Polymorphic conatiner for saving types
     */
    using TTypeCont = std::vector<Type::Sptr>;


    /**
     * \brief Sets Factory for parsing a language
     * \brief Previus variables and types of prior factory get saved,
     * \brief then the subsequent factories variables and types get loaded.
     * \param Reference to a SymbolFactory
     * \return void
     */
    void SetFactory(const ISymbolFactory& Factory);

    /**
     * \brief Adds a new type to the language
     * \param string of typename
     * \return void
     */
    void AddType(std::string const& name);

    /**
     * \brief Adds a new variable if type exists
     * \param string of variable, string of type
     * \return void
     */
    void AddVariable(std::string const& name, std::string const& type);

    /**
     * \brief CTOR of a Symbol Parser Object.
     * Loads the current state form the sym files
     * \param fact
     */
    SymbolParser(const ISymbolFactory& fact);

    /**
     * \brief DTOR of Symbol Parser.
     * Saves the current state to the sym Files
     */
    virtual ~SymbolParser();
```

```cpp
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**

```cpp
/****************************************************************//**
 * \file   SymbolParser.cpp
 * \brief  A multi language parser for types and variables
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#include <algorithm>
#include <fstream>
#include <iostream>
#include <cassert>
#include "SymbolParser.hpp"
#include "ISymbolFactory.hpp"

using namespace std;

void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)
{
    assert(m_Factory != nullptr);

    ofstream type_File;
    ofstream var_File;

    type_File.open(type_file);

    // check if file is good
    if (!type_File.good()) {
        type_File.close();
        return;
    }

    for_each(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& type) { type_File << type->
        GetSaveLine(); });

    type_File.close();


    var_File.open(var_file);

    // check if file is good
    if (!var_File.good()) {
        var_File.close();
        return;
    }

    for_each(m_variableCont.cbegin(), m_variableCont.cend(), [&](const auto& var) { var_File << var->
        GetSaveLine(); });

    var_File.close();
}

void SymbolParser::LoadNewState(const std::string& type_file, const std::string& var_file)
{
    assert(m_Factory != nullptr);

    ifstream type_File;
    ifstream var_File;

    m_typeCont.clear();
    m_variableCont.clear();

    type_File.open(type_file);

    // check if file is good
    if (!type_File.good()) {
        type_File.close();
        return;
    }

    string line;

    // Set Name can throw an exception
    // the file must be closed !!!
```

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

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

```
220        std::cerr << "Error_Type_for_Variable_does_not_exist_!!_\n";
221
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

```cpp
/*****************************************************************//**
 * \file   ISymbolFactory.hpp
 * \brief  A Interface for creating SymbolFactories
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#ifndef ISYMBOL_FACTORY_HPP
#define ISYMBOL_FACTORY_HPP

#include "Variable.hpp"
#include "Type.hpp"

class ISymbolFactory
{
public:
        /**
         * \brief Creates a variable
         *
         * \param string of variables name
         * \return uniqie pointer to variable
         */
        virtual Variable::Uptr CreateVariable(const std::string& name) const =0;

        /**
         * \brief Creates a type
         *
         * \param string of typename
         * \return unique pointer to type
         */
        virtual Type::Uptr CreateType(const std::string& name) const =0;

        /**
         * \brief Getter for file path of type file
         *
         * \return string of filePath
         */
        virtual const std::string& GetTypeFileName() const =0;

        /**
         * \brief Getter for file path of variable file
         *
         * \return string of filePath
         */
        virtual const std::string& GetVariableFileName() const =0;

        virtual ~ISymbolFactory() = default;

protected:
private:
};
#endif
```

7.5 Identifier.hpp

```cpp
/*****************************************************************//**
 * \file   Identifier.hpp
 * \brief  Generalization of Types and Variables
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#ifndef IDENTIFIER_HPP
#define IDENTIFIER_HPP

#include <string>
#include "Object.h"

class Identifier : public Object
{
public:

        inline static const std::string ERROR_EMPTY_STRING = "ERROR:_Empty_String";

        /**
         * \brief Getter for  name
         *
         * \return string of name
         */
        const std::string & GetName() const;

        /**
         * \brief Sets a  name
         *
         * \param string fileLine
         * \return string of type - SymbolParser has to check type for validity
         * \throw ERROR_EMPTY_STRING
         */
        void SetName(const std::string& name);

protected:
        Identifier(const std::string& name) : m_name{ name } {}
        Identifier() = default;

        std::string m_name;
private:
};

#endif
```

7.6 Identifier.cpp

```cpp
/****************************************************************//**
 * \file   Identifier.cpp
 * \brief  Generalization of Types and Variables
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/

#include "Identifier.hpp"


const std::string & Identifier::GetName() const{
        return m_name;
}

void Identifier::SetName(const std::string& name)
{
        if (name.empty()) throw Identifier::ERROR_EMPTY_STRING;

        m_name = name;
}
```

7.7 Variable.hpp

```cpp
/*****************************************************************//**
 * \file   Variable.hpp
 * \brief  Abstract class for parsing types
 * \author Simon Vogelhuber
 * \date   Dezember 2025
 *********************************************************************/

#ifndef VARIABLE_HPP
#define VARIABLE_HPP

#include <memory>
#include <string>

#include "Identifier.hpp"
#include "Type.hpp"

class Variable: public Identifier
{
public:
    /**
     * \brief Unique pointer type for variable
     */
    using Uptr = std::unique_ptr<Variable>;

    /**
     * \brief Returns formatted line of a variables declaration
     *
     * \return string of variable
     */
    virtual std::string GetSaveLine() const = 0;

    /**
     * \brief Loads the name of a variables type
     *
     * \param string fileLine
     * \return string of type - SymbolParser has to check type for validity
     */
    virtual std::string LoadTypeName(std::string const& fileLine) const  = 0;

    /**
     * \brief Loads name of a variable
     *
     * \param string fileLine
     * \return string of variables name
     */
    virtual std::string LoadVarName(std::string const& fileLine) const = 0;

    /**
     * \brief Sets the type of a variable
     *
     * \param shared pointer of type
     * \return void
     * \throw ERROR_NULLPTR
     */
    void SetType(Type::Sptr type);


    /**
     * \brief Name getter
     *
     * \return string of variable
     */
    const std::string & GetTypeName() const;


protected:
    Variable(const std::string& name) : Identifier{ name } {}
    Variable() = default;

    Type::Sptr m_type;
private:
};
```

```
73  #endif
```

7.8 Variable.cpp

```cpp
/****************************************************************//**
 * \file   Variable.cpp
 * \brief  Abstract class for parsing types
 * \author Simon Vogelhuber
 * \date   Dezember 2025
 ********************************************************************/

#include "Variable.hpp"

using namespace std;

void Variable::SetType(Type::Sptr type)
{
    if (type == nullptr) throw Type::ERROR_NULLPTR;

    m_type = std::move(type);
}

const std::string & Variable::GetTypeName() const
{
    return m_type->GetName();
}
```

7.9 Type.hpp

```cpp
/*****************************************************************//**
 * \file   Type.hpp
 * \brief  Abstract class for parsing types
 * \author Simon Vogelhuber
 * \date   Dezember 2025
 *********************************************************************/
#ifndef TYPE_HPP
#define TYPE_HPP

#include <memory>
#include <string>
#include "Identifier.hpp"

class Type : public Identifier
{
public:

        /**
         * \brief Unique pointer type for type
         */
        using Uptr = std::unique_ptr<Type>;

        /**
         * \brief Shared pointer type for type
         */
        using Sptr = std::shared_ptr<Type>;

        /**
         * \brief Loads a types name from a files line
         *
         * \param string fileLine
         * \return string of type
         */
        virtual std::string LoadTypeName(const std::string& fileLine) const = 0;

        /**
         * \brief Returns formatted line of a types declaration
         *
         * \return string of type declaration
         */
        virtual std::string GetSaveLine() const = 0;


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

7.10 SingetonBase.hpp

```cpp
/*****************************************************************//**
 * \file    SingetonBase.hpp
 * \brief   Base Class for creating singletons
 *
 * \author Simon
 * \date    November 2025
 *********************************************************************/
#ifndef SINGLETON_BASE_HPP
#define SINGLETON_BASE_HPP

#include "Object.h"
#include <memory>

template <typename T> class SingletonBase : public Object {
public:
        /**
         * \brief Getter for static member Singleton.
         *
         * \return
         */
        static T& GetInstance() {
                if (mInstance == nullptr) { mInstance = std::unique_ptr<T>{ new T{} }; };
                return *mInstance;
        }
protected:
        SingletonBase() = default;
        virtual ~SingletonBase() = default;

private:
        SingletonBase(SingletonBase const& s) = delete;
        SingletonBase& operator = (SingletonBase const& s) = delete;
        static std::unique_ptr<T> mInstance;
};

template <typename T> std::unique_ptr<T> SingletonBase<T>::mInstance = nullptr;

#endif // !SINGLETON_BASE_HPP
```

7.11 JavaType.hpp

```cpp
/*****************************************************************//**
 * \file   JavaType.hpp
 * \brief  A Class for parsing java types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/

#ifndef JAVA_TYPE_HPP
#define JAVA_TYPE_HPP

#include "Type.hpp"
#include <string>

class JavaType : public Type
{
public:
        /**
         * \brief Loads a types name from a files line
         *
         * \param string fileLine
         * \return string of type
         */
        virtual std::string LoadTypeName(const std::string& fileLine) const override;

        /**
         * \brief Returns formatted line of a types declaration
         *
         * \return string of type declaration
         */
        virtual std::string GetSaveLine() const override;

        JavaType(const std::string & name) : Type{ name } {}

        JavaType() = default;

protected:
private:
};
#endif
```

7.12 JavaType.cpp

```cpp
/*****************************************************************//**
 * \file   JavaType.cpp
 * \brief  A Class for parsing java types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/

#include "JavaType.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include "scanner.h"

using namespace pfc;
using namespace std;

/**
 * \brief Scans an input string for the Type name of the Type.
 *
 * \param scan Reference to scanner object
 * \return emtpy string if no valid type name is found
 * \return name of type
 */
static std::string ScanTypeName(scanner& scan) {
        try{
                string TypeName;

                if (scan.get_identifier() == "class") {
                        scan.next_symbol();
                        TypeName = scan.get_identifier();
                        scan.next_symbol();
                        if (!scan.has_symbol()) {
                                return TypeName;
                        }
                }
        }
        // catch Scanner Exceptions
        catch (...) {
                return "";
        }

        return "";
}


std::string JavaType::LoadTypeName(const std::string& fileLine) const
{
        stringstream sstream;

        sstream << fileLine;

        scanner scan;

        scan.set_istream(sstream);

        return ScanTypeName(scan);
}

std::string JavaType::GetSaveLine() const
{
        return "class " + m_name + "\n";
}
```

7.13 JavaVariable.hpp

```cpp
/*****************************************************************//**
 * \file   JavaVariable.hpp
 * \brief  A Class for parsing java variables
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#ifndef JAVA_VARIABLE_HPP
#define JAVA_VARIABLE_HPP
#include "Object.h"
#include "Variable.hpp"

class JavaVariable :public Variable
{
public:
    /**
     * \brief Returns formatted line of a variables declaration
     *
     * \return string of variable
     */
    virtual std::string GetSaveLine() const override;

    /**
     * \brief Loads the name of a variables type
     *
     * \param string fileLine
     * \return string of type - SymbolParser has to check type for validity
     */
    virtual std::string LoadTypeName(std::string const& fileLine) const override;

    /**
     * \brief Loads name of a variable
     *
     * \param string fileLine
     * \return string of variables name
     */
    virtual std::string LoadVarName(std::string const& fileLine) const override;

    JavaVariable() = default;

    JavaVariable(const std::string& name) : Variable{ name } {}

protected:
private:
};
#endif
```

7.14 JavaVariable.cpp

```cpp
/****************************************************************//**
 * \file   JavaVariable.cpp
 * \brief  A Class for parsing java variables
 * \author Simon
 * \date   Dezember 2025
 ********************************************************************/

#include "JavaVariable.hpp"
#include <sstream>
#include <string>
#include "scanner.h"

using namespace pfc;
using namespace std;

/**
 * \brief Scans an input string for the Type name of the Var.
 *
 * \param scan Reference to scanner object
 * \return emtpy string if no valid type name is found
 * \return name of type
 */
static std::string ScanTypeName(scanner& scan)
{
        string typeName;

        try{
                typeName = scan.get_identifier();
                scan.next_symbol();
        }
        // catch Scanner Exceptions
        catch (...) {
                return "";
        }

        return typeName;
}

/**
 * \brief Scans an input string for the Variable name of the Var.
 *
 * \param scan Reference to scanner object
 * \return emtpy string if no valid Variable name is found
 * \return name of Variable
 */
static std::string ScanVarName(scanner& scan)
{
        string varName;

        try{
                varName = scan.get_identifier();
                scan.next_symbol();

                // The line should be emty after the var Name!
                if (!scan.is(';'))  varName = "";
        }
        // catch Scanner Exceptions
        catch (...) {
                return "";
        }

        return varName;
}

std::string JavaVariable::GetSaveLine() const
{
        if (m_type == nullptr) return "";

        return m_type->GetName() + "␣" + m_name + ";\n";
}

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

```cpp
/*****************************************************************//**
 * \file   JavaSymbolFactory.hpp
 * \brief  A factory for creating java variables and types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#ifndef JAVA_SYMBOL_FACTORY_HPP
#define JAVA_SYMBOL_FACTORY_HPP

#include "ISymbolFactory.hpp"
#include "SingetonBase.hpp"

class JavaSymbolFactory :public ISymbolFactory, public SingletonBase<JavaSymbolFactory>
{
public:

        friend class SingletonBase<JavaSymbolFactory>;

        /**
         * \brief Creates a java variable
         *
         * \param string of variables name
         * \return unique pointer to variable
         */
        virtual Variable::Uptr CreateVariable(const std::string& name) const override;

        /**
         * \brief Creates a java type
         *
         * \param string of typename
         * \return unique pointer to type
         */
        virtual Type::Uptr CreateType(const std::string& name) const override;

        /**
         * \brief Getter for file path of type file
         *
         * \return string of filePath
         */
        virtual const std::string& GetTypeFileName() const override;

        /**
         * \brief Getter for file path of variable file
         *
         * \return string of filePath
         */
        virtual const std::string& GetVariableFileName() const override;

        // delete CopyCtor and Assign operator to prevent untestet behaviour
        JavaSymbolFactory(JavaSymbolFactory& fact) = delete;
        void operator=(JavaSymbolFactory fact) = delete;

protected:
private:
        JavaSymbolFactory() = default;
        const std::string m_TypeFileName = "JavaTypes.sym";
        const std::string m_VariableFileName = "JavaVars.sym";
};
#endif
```

7.16 JavaSymbolFactory.cpp

```cpp
/*****************************************************************//**
 * \file   JavaSymbolFactory.cpp
 * \brief  A factory for creating java variables and types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#include "JavaSymbolFactory.hpp"
#include "JavaType.hpp"
#include "JavaVariable.hpp"


Variable::Uptr JavaSymbolFactory::CreateVariable(const std::string& name) const
{
    return std::make_unique<JavaVariable>( name );
}

Type::Uptr JavaSymbolFactory::CreateType(const std::string& name) const
{
    return std::make_unique<JavaType>(name);
}

const std::string& JavaSymbolFactory::GetTypeFileName() const
{
    return m_TypeFileName;
}

const std::string& JavaSymbolFactory::GetVariableFileName() const
{
    return m_VariableFileName;
}
```

7.17 **IECType.hpp**

```cpp
/*****************************************************************//**
 * \file   IECType.hpp
 * \brief  A Class for parsing IEC types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/

#ifndef IEC_TYPE_HPP
#define IEC_TYPE_HPP
#include "Object.h"
#include "Type.hpp"

class IECType: public Type
{
public:
        /**
         * \brief Loads a types name from a files line
         *
         * \param string fileLine
         * \return string of type
         */
        virtual std::string LoadTypeName(const std::string& fileLine) const override;

        /**
         * \brief Returns formatted line of a types declaration
         *
         * \return string of type declaration
         */
        virtual std::string GetSaveLine() const override;

        /**
         * \brief Constructs a Type with a specified name.
         *
         * \param name
         */
        IECType(const std::string & name) : Type{ name } {}

        IECType() = default;

protected:
private:
};
#endif
```

7.18 IECType.cpp

```cpp
/****************************************************************//**
 * \file   IECType.cpp
 * \brief  A Class for parsing IEC types
 * \author Simon
 * \date   Dezember 2025
 ********************************************************************/

#include "IECType.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include "scanner.h"

using namespace pfc;
using namespace std;

/**
 * \brief Scans an input string for the Type name of the Type.
 *
 * \param scan Reference to scanner object
 * \return emtpy string if no valid type name is found
 * \return name of type
 */
static std::string ScanTypeName(scanner& scan) {
	try {
		string TypeName;

		if (scan.get_identifier() == "TYPE") {
			scan.next_symbol();
			TypeName = scan.get_identifier();
			scan.next_symbol();
			if (!scan.has_symbol()) {
				return TypeName;
			}
		}
	}
	// catch Scanner Exceptions
	catch (...) {
		return "";
	}

	return "";
}


std::string IECType::LoadTypeName(const std::string& fileLine) const
{
	stringstream sstream;

	sstream << fileLine;

	scanner scan;

	scan.set_istream(sstream);

	return ScanTypeName(scan);
}


std::string IECType::GetSaveLine() const
{
	return "TYPE_" + m_name + "\n";
}
```

7.19 IECVariable.hpp

```cpp
/*****************************************************************//**
 * \file   IECVariable.hpp
 * \brief  A Class for parsing IEC variables
 * \author Simon Vogelhuber
 * \date   Dezember 2025
 *********************************************************************/

#ifndef IEC_VARIABLE_HPP
#define IEC_VARIABLE_HPP

#include "Variable.hpp"

class IECVariable : public Variable
{
public:
    /**
     * \brief Returns formatted line of a Variable declaration
     *
     * \return string of variable
     */
    virtual std::string GetSaveLine() const override;

    /**
     * \brief Loads the name of a variables type
     *
     * \param string fileLine
     * \return string of type - SymbolParser has to check type for validity
     */
    virtual std::string LoadTypeName(std::string const& fileLine) const override;

    /**
     * \brief Loads name of a variable
     *
     * \param string fileLine
     * \return string of variables name
     */
    virtual std::string LoadVarName(std::string const& fileLine) const override;

    IECVariable() = default;

    IECVariable(const std::string & name) : Variable{ name } {}

protected:
private:
};
#endif
```

7.20 IECVariable.cpp

```cpp
/*******************************************************************//**
 * \file   IECVariable.cpp
 * \brief  A Class for parsing IEC variables
 * \author Simon Vogelhuber
 * \date   Dezember 2025
 ***********************************************************************/

#include "IECVariable.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include "scanner.h"

using namespace pfc;
using namespace std;

std::string IECVariable::GetSaveLine() const
{
        if (m_type == nullptr) return "";

        return  "VAR " + m_type->GetName() + " : " + m_name + ";\n";
}

/**
 * \brief Scans an input string for the Type name of the Var.
 *
 * \param scan Reference to scanner object
 * \return emtpy string if no valid type name is found
 * \return name of type
 */
static std::string ScanTypeName(scanner & scan) {
        try{
                string TypeName;

                if (scan.get_identifier() == "VAR") {
                        scan.next_symbol();
                        TypeName = scan.get_identifier();
                        scan.next_symbol();
                        return TypeName;
                }
        }
        // catch Scanner Exceptions
        catch (...) {
                return "";
        }
        return "";
}

/**
 * \brief Scans an input string for the Variable name of the Var.
 *
 * \param scan Reference to scanner object
 * \return emtpy string if no valid Variable name is found
 * \return name of Variable
 */
static std::string ScanVarName(scanner & scan) {
        string VarName;

        try{
                if (scan.is(':')) {
                        scan.next_symbol();
                        VarName = scan.get_identifier();
                        scan.next_symbol();
                        if (!scan.is(';')) {
                                VarName = "";
                        }
                }
        }
        // catch Scanner Exceptions
        catch (...) {
                return "";
        }
```

```
73
74          return VarName;
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

```cpp
/****************************************************************//**
 * \file   IECSymbolFactory.hpp
 * \brief  A factory for creating IEC variables and types
 * \author Simon
 * \date   Dezember 2025
 ********************************************************************/
#ifndef IEC_SYMBOL_FACTORY_HPP
#define IEC_SYMBOL_FACTORY_HPP

#include "Object.h"
#include "ISymbolFactory.hpp"
#include "SingetonBase.hpp"

class IECSymbolFactory : public ISymbolFactory , public SingletonBase<IECSymbolFactory>
{
public:

        // This class is a Singleton
        friend class SingletonBase<IECSymbolFactory>;

        /**
         * \brief Creates a IEC variable
         *
         * \param string of variables name
         * \return uniqie pointer to variable
         */
        virtual Variable::Uptr CreateVariable(const std::string& name) const override;

        /**
         * \brief Creates a IEC type
         *
         * \param string of typename
         * \return unique pointer to type
         */
        virtual Type::Uptr CreateType(const std::string& name) const override;

        /**
         * \brief Getter for file path of type file
         *
         * \return string of filePath
         */
        virtual const std::string& GetTypeFileName() const override;

        /**
         * \brief Getter for file path of variable file
         *
         * \return string of filePath
         */
        virtual const std::string& GetVariableFileName() const override;

        // delete CopyCtor and Assign operator to prevent untestet behaviour
        IECSymbolFactory(IECSymbolFactory& fact) = delete;
        void operator=(IECSymbolFactory fact) = delete;

protected:
private:
        IECSymbolFactory() = default;

        const std::string m_TypeFileName = "IECTypes.sym";
        const std::string m_VariableFileName  =  "IECVars.sym";
};

#endif
```

7.22 IECSymbolFactory.cpp

```cpp
/*****************************************************************//**
 * \file   IECSymbolFactory.cpp
 * \brief  A factory for creating IEC variables and types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/
#include "IECSymbolFactory.hpp"
#include "IECType.hpp"
#include "IECVariable.hpp"


Variable::Uptr IECSymbolFactory::CreateVariable(const std::string& name) const
{
    return std::make_unique<IECVariable>(name);
}

Type::Uptr IECSymbolFactory::CreateType(const std::string& name) const
{
    return std::make_unique<IECType>(name);
}

const std::string& IECSymbolFactory::GetTypeFileName() const
{
    return m_TypeFileName;
}

const std::string& IECSymbolFactory::GetVariableFileName() const
{
    return m_VariableFileName;
}
```

7.23 main.cpp

```cpp
/******************************************************************//**
 * \file   Main.cpp
 * \brief  Testdriver for Symbol Parser and all connected Classes
 * \author Simon
 * \date   November 2025
 *********************************************************************/

// These Includes are needed because of the testcases !!
#include "IECVariable.hpp"
#include "JavaVariable.hpp"
#include "IECType.hpp"
#include "JavaType.hpp"

// The Client thests the SymbolParser and SymbolFactories
#include "Client.hpp"

// Testing Includes
#include "Test.hpp"
#include "vld.h"
#include <fstream>
#include <iostream>
#include <cassert>

#include <cstdio>

using namespace std;

#define WriteOutputFile ON

static bool TestVariable(Variable* var, const string & name, Type::Sptr typ, ostream & ost = cout);
static bool TestType(Type::Sptr typ, ostream & ost = cout);
static bool TestIECVar(ostream& ost = cout);
static bool TestJavaVar(ostream& ost = cout);
static bool TestIECType(ostream& ost = cout);
static bool TestJavaType(ostream& ost = cout);

/**
 * \brief For deleting the sym files bevor the test cases!.
 *
 * \param path
 */
static void EraseFile(const char* path) {
    // Versucht, die Datei zu loeschen
    if (std::remove(path) == 0) {
        // Datei wurde erfolgreich geloescht
        std::printf("Datei_'%s'_erfolgreich_geloescht.\n", path);
    }
    else {
        // Fehler beim Loeschen der Datei
        std::perror("Fehler_beim_Loeschen_der_Datei");
    }
}

int main()
{
    // Erase previos Symbol files for test cases
    EraseFile("IECTypes.sym");
    EraseFile("IECVars.sym");
    EraseFile("JavaTypes.sym");
    EraseFile("JavaVars.sym");


    bool TestOK = true;

    ofstream output{ "output.txt" };


    try {
        Type::Sptr Itype{ make_shared<IECType>(IECType{ "int" }) };

        Type::Sptr Jtyp{ make_shared<JavaType>(JavaType{ "int" }) };
```

```cpp
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 Itype1{ make_shared<IECType>(IECType{ "int" }) };
111
112             Type::Sptr Jtyp1{ make_shared<JavaType>(JavaType{ "int" }) };
113
114             IECVariable IECVar1{ "asdf" };
115             IECVar1.SetType(Itype1);
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", Itype1, 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(Itype1, 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 {
```

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

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

```cpp
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
320        try {
321            typ->SetName("");
322        }
323        catch (const string& err) {
324            error_msg = err;
325        }
326        catch (bad_alloc const& error) {
327            error_msg = error.what();
328        }
329        catch (const exception& err) {
330            error_msg = err.what();
331        }
332        catch (...) {
333            error_msg = "Unhandelt_Exception";
334        }
335
336        TestOK = TestOK && check_dump(ost, "Test_Exception_in_Set_Type", Type::ERROR_EMPTY_STRING,
                   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
356            IECVariable var;
357
358            const string LineToDecode = "VAR_mCont_:_Speed_Controller;\n";
359            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_IEC_Var",static_cast<string>("mCont"),
                   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 IECTyp = make_shared<IECType>( IECType{} );
387         var.SetName(var.LoadVarName(LineToDecode));
388         IECTyp->SetName(var.LoadTypeName(LineToDecode));
389         var.SetType(IECTyp);
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"),
                    var.LoadTypeName(LineToDecode));
435            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var",static_cast<string>("mBut"),
                    var.LoadVarName(LineToDecode));
436
437            const string InvLineToDecode = "1mCont mBut;";
438            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadTypeName(InvLineToDecode));
439            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadVarName(InvLineToDecode));
440
441            const string Inv2LineToDecode = "mCont ;mBut;";
442            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_Java_Var_invalid_Format",static_cast<
                    string>("mCont"), var.LoadTypeName(Inv2LineToDecode));
443            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadVarName(Inv2LineToDecode));
444
445            const string Inv3LineToDecode = "2mCont mBut;";
446            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadTypeName(Inv3LineToDecode));
447            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadVarName(Inv3LineToDecode));
448
449            const string Inv4LineToDecode = "mCont 123;";
450            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_Java_Var_invalid_Format",static_cast<
                    string>("mCont"), var.LoadTypeName(Inv4LineToDecode));
451            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadVarName(Inv4LineToDecode));
452
453            const string Inv5LineToDecode = "123 : a12343;";
454            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadTypeName(Inv5LineToDecode));
455            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadVarName(Inv5LineToDecode));
456
457            const string Inv6LineToDecode = "";
458            TestOK == TestOK && check_dump(ost,"Test_Load_Type_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadTypeName(Inv6LineToDecode));
459            TestOK == TestOK && check_dump(ost,"Test_Load_Var_Name_Java_Var_invalid_Format",static_cast<
                    string>(""), var.LoadVarName(Inv6LineToDecode));
460
461            Type::Sptr JTyp = make_shared<JavaType>(JavaType{});
462            var.SetName(var.LoadVarName(LineToDecode));
463            JTyp->SetName(var.LoadTypeName(LineToDecode));
464            var.SetType(JTyp);
465
466            TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", LineToDecode, var.
                    GetSaveLine());
467
468            JavaVariable JVar;
469
470            TestOK == TestOK && check_dump(ost, "Test_Save_LineFormat_IEC_Variable", static_cast<string>(""
                    ), JVar.GetSaveLine());
471        }
472        catch (const string& err) {
473            error_msg = err;
474        }
475        catch (bad_alloc const& error) {
476            error_msg = error.what();
477        }
478        catch (const exception& err) {
479            error_msg = err.what();
480        }
481        catch (...) {
482            error_msg = "Unhandelt Exception";
```

```cpp
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)
```

```cpp
551  {
552      assert(ost.good());
553
554      ost << TestStart;
555
556
557      bool TestOK = true;
558      string error_msg;
559
560      try{
561
562          JavaType typ;
563
564          const string LineToDecode = "class_SpeedController\n";
565          TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type", static_cast<string>("
                 SpeedController"), typ.LoadTypeName(LineToDecode));
566
567          const string InvLineToDecode = "1class_SpeedController";
568          TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast
                 <string>(""), typ.LoadTypeName(InvLineToDecode));
569
570          const string Inv2LineToDecode = "class_1SpeedController";
571          TestOK == TestOK && check_dump(ost, "Test_Load_Type_Name_Java_Type_invalid_Format", static_cast
                 <string>(""), typ.LoadTypeName(Inv2LineToDecode));
572
573          const string Inv3LineToDecode = "class_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

```cpp
/****************************************************************//**
 * \file   Client.hpp
 * \brief  Test File to show that you only need to include Symbol Parser
 * \brief  plus factories to work with the parser!
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#ifndef CLIENT_HPP
#define CLIENT_HPP

#include <iostream>
/**
 * \brief TestDriver for the SymbolParser.
 */
bool TestSymbolParser(std::ostream& ost = std::cout);

#endif // !1
```

7.25 Client.cpp

```cpp
/*****************************************************************//**
 * \file   Client.cpp
 * \brief  Test File to show that you only need to include Symbol Parser
 * \brief  plus factories to work with the parser!
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

// The client is only dependent on these classes!!
#include "SymbolParser.hpp"
#include "JavaSymbolFactory.hpp"
#include "IECSymbolFactory.hpp"

// Testing Includes
#include "Test.hpp"
#include <fstream>
#include <cassert>
#include "Client.hpp"

using namespace std;

bool TestSymbolParser(std::ostream& ost)
{
    bool TestOK = true;
    string error_msg;

    if (!ost.good()) throw "Error Ostream bad!";

    ost << TestStart;

    // normal operating mode - no exception should be thrown
    try {
        SymbolParser parser{ JavaSymbolFactory::GetInstance() };
        parser.AddType("Button");
        parser.AddVariable("mButton", "Button");
        parser.SetFactory(IECSymbolFactory::GetInstance());
        parser.AddType("TYPE");
        parser.AddVariable("VARIABLE", "TYPE");
        parser.SetFactory(JavaSymbolFactory::GetInstance());
        parser.AddVariable("mButton2", "Button"); // <- this is only possible if the loading of the
            vars was successful
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt Exception";
    }

    TestOK = TestOK && check_dump(ost, "Normal Operating Parser", true, error_msg.empty());
    error_msg.clear();

    // addtype - adding empty type - throws error
    try {
        SymbolParser parser{ JavaSymbolFactory::GetInstance() };
        parser.AddType("");
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
```

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

```
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_exeption_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_exeption_Dup_Type",
240            SymbolParser::ERROR_DUPLICATE_TYPE, error_msg);
         error_msg.clear();
241
242        ost << TestEnd;
243        return TestOK;
244    }
```

7.26 Test.hpp

```cpp
/****************************************************//**
 * \file   Test.hpp
 * \brief  File that provides a Test Function with a formated output
 *
 * \author Simon
 * \date   April 2025
 *********************************************************/
#ifndef TEST_HPP
#define TEST_HPP

#include <string>
#include <iostream>
#include <vector>
#include <list>
#include <queue>
#include <forward_list>

#define ON 1
#define OFF 0
#define COLOR_OUTPUT OFF

// Definitions of colors in order to change the color of the output stream.
const std::string colorRed = "\x1B[31m";
const std::string colorGreen = "\x1B[32m";
const std::string colorWhite = "\x1B[37m";

inline std::ostream& RED(std::ostream& ost) {
        if (ost.good()) {
                ost << colorRed;
        }
        return ost;
}
inline std::ostream& GREEN(std::ostream& ost) {
        if (ost.good()) {
                ost << colorGreen;
        }
        return ost;
}
inline std::ostream& WHITE(std::ostream& ost) {
        if (ost.good()) {
                ost << colorWhite;
        }
        return ost;
}

inline std::ostream& TestStart(std::ostream& ost) {
        if (ost.good()) {
                ost << std::endl;
                ost << "*****************************************" << std::endl;
                ost << "_____TESTCASE_START_____" << std::endl;
                ost << "*****************************************" << std::endl;
                ost << std::endl;
        }
        return ost;
}

inline std::ostream& TestEnd(std::ostream& ost) {
        if (ost.good()) {
                ost << std::endl;
                ost << "*****************************************" << std::endl;
                ost << std::endl;
        }
        return ost;
}

inline std::ostream& TestCaseOK(std::ostream& ost) {

#if COLOR_OUTPUT
        if (ost.good()) {
                ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;
        }
#else
```

```cpp
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          }
88  #else
89          if (ost.good()) {
90                  ost << "TEST FAILED !!" << std::endl;
91
92          }
93  #endif // COLOR_OUTPUT
94
95          return ost;
96  }
97
98  /**
99          * \brief function that reports if the testcase was successful.
100         *
101         * \param testcase      String that indicates the testcase
102         * \param succsessful true -> reports to cout test OK
103         * \param succsessful false -> reports test failed
104         */
105 template <typename T>
106 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
107         if (ostr.good()) {
108 #if COLOR_OUTPUT
109                 if (expected == result) {
110                         ostr << testcase << std::endl <<  colorGreen << "[Test OK] " << colorWhite <<"
                                Result: (Expected: " << std::boolalpha << expected << " ==" << " Result: "
                                 << result << ")"  << std::noboolalpha << std::endl << std::endl;
111                 }
112                 else {
113                         ostr << testcase << std::endl << colorRed << "[Test FAILED] " << colorWhite <<
                                "Result: (Expected: " << std::boolalpha << expected << " !=" << " Result: 
                                " << result << ")"  << std::noboolalpha << std::endl << std::endl;
114                 }
115 #else
116                 if (expected == result) {
117                         ostr << testcase << std::endl << "[Test OK] "  << "Result: (Expected: " << std
                                ::boolalpha << expected << " ==" << " Result: " << result << ")" << std::
                                noboolalpha << std::endl << std::endl;
118                 }
119                 else {
120                         ostr << testcase << std::endl  << "[Test FAILED] "  << "Result: (Expected: " <<
                                 std::boolalpha << expected << " !=" << " Result: " << result << ")" <<
                                std::noboolalpha << std::endl << std::endl;
121                 }
122 #endif
123                 if (ostr.fail()) {
124                         std::cerr << "Error: Write Ostream" << std::endl;
125                 }
126         }
127         else {
128                 std::cerr << "Error: Bad Ostream" << std::endl;
129         }
130         return expected == result;
131 }
132
133 template <typename T1, typename T2>
134 std::ostream& operator<< (std::ostream& ost,const std::pair<T1,T2> & p) {
135         if (!ost.good()) throw std::exception{ "Error bad Ostream!" };
136         ost << "(" << p.first << "," << p.second << ")";
137         return ost;
138 }
139
```

```cpp
140   template <typename T>
141   std::ostream& operator<< (std::ostream& ost,const std::vector<T> & cont) {
142           if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
143           std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>{ost, "_"});
144           return ost;
145   }
146
147   template <typename T>
148   std::ostream& operator<< (std::ostream& ost,const std::list<T> & cont) {
149           if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
150           std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>{ost, "_"});
151           return ost;
152   }
153
154   template <typename T>
155   std::ostream& operator<< (std::ostream& ost,const std::deque<T> & cont) {
156           if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
157           std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>{ost, "_"});
158           return ost;
159   }
160
161   template <typename T>
162   std::ostream& operator<< (std::ostream& ost,const std::forward_list<T> & cont) {
163           if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
164           std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>{ost, "_"});
165           return ost;
166   }
167
168
169   #endif // !TEST_HPP
```

7.27 **scanner.h**

```
1   //        $Id: scanner.h 46519 2022-12-04 12:29:04Z p20068 $
2   //       $URL: https://svn01.fh-hagenberg.at/bin/cepheiden/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 &) {
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     }
```

```cpp
 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::
           classic ()) {
124      return std::isalpha (c, locale) || std::char_traits <T>::eq (c, T {sk_uscore.chr});
125  }
126
127  constexpr bool is_digit (auto const c, std::locale const & locale = std::locale::classic ()) {
128      return std::isdigit (c, locale);
129  }
130
131  }   // namespace details
132
133  struct exception final : std::runtime_error {
134      explicit exception (std::string const & m) : std::runtime_error {m} {
135      }
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      }
```

```cpp
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> () : ""s;
220     }
```

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

```
296
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 (
                            m_current_symbol) + ‘.’
301                };
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
312            return next_symbol ();
313        }
314
315        template <typename attr_t> constexpr attr_t const & get_attribute () const {
316            return m_current_symbol.get_attribute <attr_t> ();
317        }
318
319        std::string get_identifier () const {
320            return m_current_symbol.get_identifier ();
321        }
322
323        constexpr int get_integer () const {
324            return m_current_symbol.get_integer ();
325        }
326
327        constexpr double get_number () const {
328            return m_current_symbol.get_number ();
329        }
330
331        constexpr double get_real () const {
332            return m_current_symbol.get_real ();
333        }
334
335        std::string get_string () const {
336            return m_current_symbol.get_string ();
337        }
338
339        constexpr bool is (char const c) const {
340            return m_current_symbol.is (c);
341        }
342
343        constexpr bool is_eof () const {
344            return m_current_symbol.is_eof ();
345        }
346
347        constexpr bool is_identifier () const {
348            return m_current_symbol.is_identifier ();
349        }
350
351        constexpr bool is_integer () const {
352            return m_current_symbol.is_integer ();
353        }
354
355        constexpr bool is_keyword () const {
356            return m_current_symbol.is_keyword ();
357        }
358
359        constexpr bool is_keyword (std::size_t const num) const {
360            return m_current_symbol.is_keyword (num);
361        }
362
363        bool is_keyword (std::string const & name) const {
364            return m_current_symbol.is_keyword (name);
365        }
366
367        constexpr bool is_number () const {
368            return m_current_symbol.is_number ();
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
                      .chr) || is_eol_chr ();
448        }
449
450        std::size_t keyword_is_registered (std::string const & name) const {
451           if (auto const pos {m_keywords.find (name)}; pos != std::end (m_keywords))
452              return pos->second.m_kind.num;
453
454           return 0;
455        }
456
457        void read_next_chr () {
458           m_current_chr = scn::details::sk_eof.chr;
459
460           if (m_p_in && *m_p_in)
461              m_p_in->get (m_current_chr);
462        }
463
464        void read_next_chr (char const c) {
465           if (m_current_chr != c)
466              throw scn::exception {"Expected ’"s + c + "’ but have ’" + m_current_chr + "’."};
467
468           read_next_chr ();
469        }
470
471        void read_next_symbol () {
472           m_current_symbol.clear (); bool scanned {false};
473
474           while (!scanned) {
475              if (is_eof_chr ())
476                 scanned = true;
477
478              else if (is_whitespace_chr ())
479                 read_next_chr ();
480
481              else if (scn::details::is_alpha (m_current_chr))
482                 scanned = scan_keyword_or_identifier ();
483
484              else if (m_current_chr == scn::details::sk_dquote.chr)
485                 scanned = scan_string ();
486
487              else if (scn::details::is_digit (m_current_chr))
488                 scanned = try_read_number ();
489
490              else if (m_current_chr == scn::details::sk_period.chr)
491                 scanned = try_read_number ();
492
493              else if (m_current_chr == scn::details::sk_plus.chr)
494                 scanned = try_read_number ();
495
496              else if (m_current_chr == scn::details::sk_minus.chr)
497                 scanned = try_read_number ();
498
499              else if (m_current_chr == scn::details::sk_div.chr) {
500                 read_next_chr (scn::details::sk_div.chr);
501
502                    if (m_current_chr == scn::details::sk_div .chr) scan_comment_single_line (scn::
                         details::sk_div.chr);
503                 else if (m_current_chr == scn::details::sk_mult.chr) scan_comment_multi_line  (scn::
                      details::sk_div.chr);
504
505                 else {
506                    m_current_symbol = scn::details::sk_div; scanned = true;
507                 }
508
509              } else {
510                 m_current_symbol = scn::details::to_symbol_kind (m_current_chr);
511
512                 if ((scanned = m_current_symbol.is_terminal_symbol ()) == false)
513                    throw scn::exception {"Unknown character ’"s + m_current_chr + "’ encountered."};
514
515                 read_next_chr ();
516              }
```

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

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

```
663   }   // namespace pfc
```