FH-OÖ Hagenberg/HSD
SDP3, WS 2025
Übung 4

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:

```
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*

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

JavaTypes.sym:

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

JavaVars.sym:

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

IECTypes.sym:

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

IECVars.sym:

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

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

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

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

Eine mögliche Anwendung im Hauptprogramm könnte so aussehen:

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

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

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

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

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

Systemdokumentation
Projekt Symbolparser

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 14. November 2025

Inhaltsverzeichnis

1 Organisatorisches **7**
1.1 Team 7
1.2 Aufteilung der Verantwortlichkeitsbereiche 7
1.3 Aufwand 8

2 Anforderungsdefinition (Systemspezifikation) **9**

3 Systementwurf **11**
3.1 Designentscheidungen 11

4 Dokumentation der Komponenten (Klassen) **13**

5 Testprotokollierung **14**

6 Quellcode **23**
6.1 Object.hpp 23
6.2 Symbolparser.hpp 24
6.3 Symbolparser.cpp 26
6.4 ISymbolFactory.hpp 29
6.5 Identifier.hpp 30
6.6 Identifier.cpp 31
6.7 Variable.hpp 32
6.8 Variable.cpp 34
6.9 Type.hpp 35
6.10 Type.cpp 36
6.11 SingetonBase.hpp 37
6.12 JavaVariable.hpp 38
6.13 JavaVariable.cpp 39
6.14 JavaSymbolFactory.hpp 41
6.15 JavaSymbolFactory.cpp 42
6.16 IECVariable.hpp 43
6.17 IECVariable.cpp 44
6.18 IECSymbolFactory.hpp 46
6.19 IECSymbolFactory.cpp 47
6.20 main.cpp 48

6.21 Client.hpp 57
6.22 Client.cpp 58
6.23 Test.hpp 62

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:
  - Implementierung des Testtreibers
  - Dokumentation
    - Object
    - Symbolparser
    - ISymbolFactory
    - Variable
    - Type
    - JavaVariable
    - JavaType
    - JavaSymbolFactory
    - IECVariable

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

1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich 8 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 eine Referenz auf die aktuelle SymbolFactory, die zur Laufzeit gewechselt werden kann.

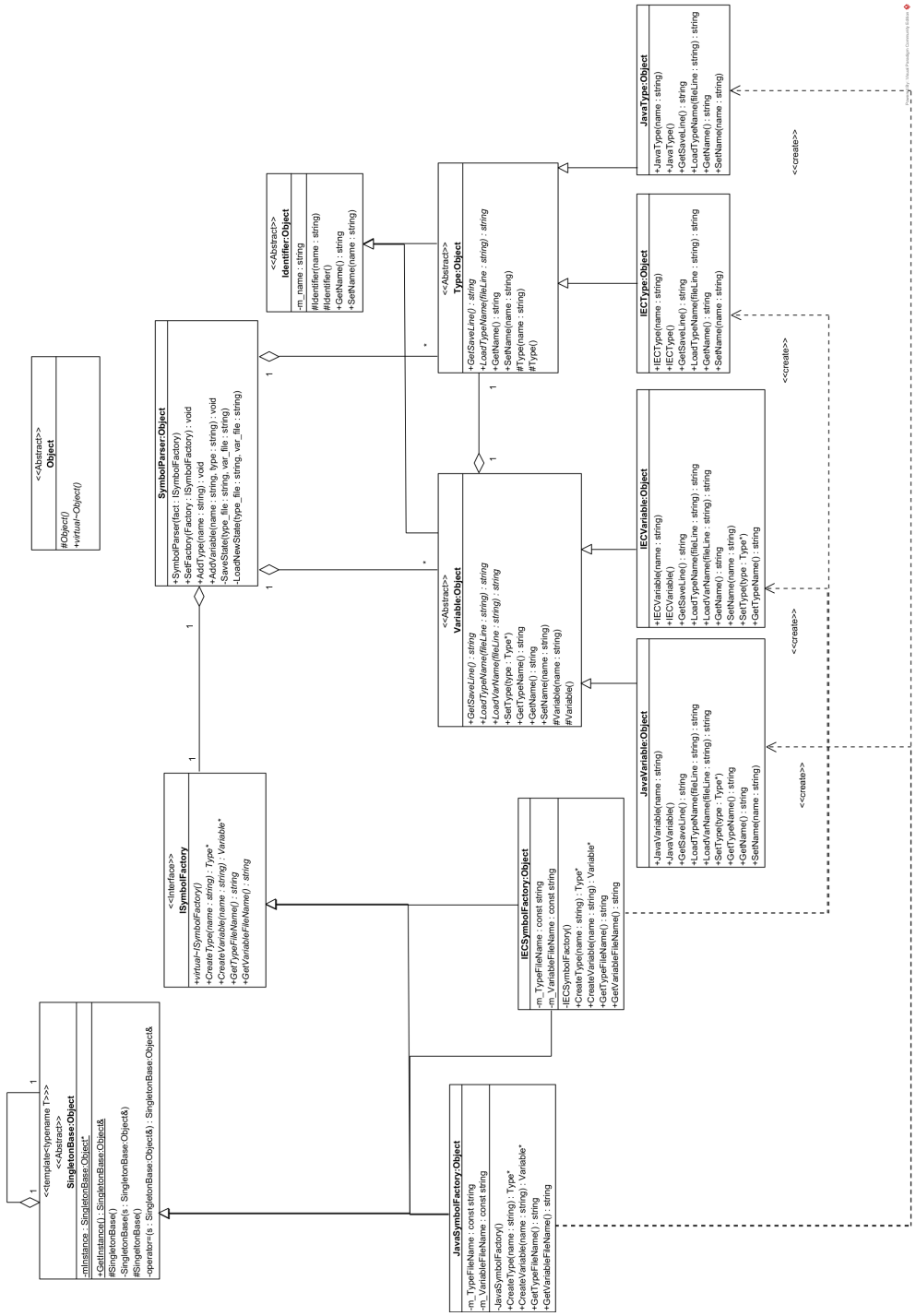
Verwendung des Singleton-Patterns:
Das Singleton- Pattern wurde für die SymbolFactory implementiert, um sicherzustellen, dass nur eine Instanz der Factory existiert.

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

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

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

Klassendiagramm



4 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis ./../doxy/html/index.html

5 Testprotokollierung

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

```
Test for Exception in TestCase
[Test OK] Result: (Expected: true == Result: true)


*****************************************


**** Test Java Var Getter ****


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

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

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)

Test Variable Get Type after set
[Test OK] Result: (Expected: uint_fast512_t == Result:
   ↪ uint_fast512_t)

Test for Exception in TestCase
```

```
[Test OK] Result: (Expected: true == Result: true)


*****************************************


**** Test IEC Type Getter ****


*****************************************
             TESTCASE START
*****************************************
Test Type Get Name after Set
[Test OK] Result: (Expected: unit_1024_t == Result:
   ↪ unit_1024_t)

Test Exception in Set Type
[Test OK] Result: (Expected: true == Result: true)

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


*****************************************


**** Test Java Type Getter ****


*****************************************
             TESTCASE START
*****************************************
Test Type Get Name after Set
[Test OK] Result: (Expected: unit_1024_t == Result:
   ↪ unit_1024_t)

Test Exception in Set Type
[Test OK] Result: (Expected: true == Result: true)

```

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


*****************************************


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

Test Load Type Name IEC Var
[Test OK] Result: (Expected: mCont == Result: mCont)

Test Load Var Name IEC Var
[Test OK] Result: (Expected: SpeedController == Result:
    ↪ SpeedController)

Test Load Type Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Var Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name IEC Var invalid Format
[Test OK] Result: (Expected: mCont == Result: mCont)

Test Load Var Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Var Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name IEC Var invalid Format
[Test OK] Result: (Expected: mCont == Result: mCont)

Test Load Var Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name IEC Var invalid Format
```

```
[Test OK] Result: (Expected:  == Result: )

Test Load Var Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Var Name IEC Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Save LineFormat IEC Variable
[Test OK] Result: (Expected: VAR mCont : SpeedController;
 == Result: VAR mCont : SpeedController;
)

Test Save LineFormat IEC Variable
[Test OK] Result: (Expected:  == Result: )

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


****************************************


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

Test Load Type Name Java Var
[Test OK] Result: (Expected: mCont == Result: mCont)

Test Load Var Name Java Var
[Test OK] Result: (Expected: mBut == Result: mBut)

Test Load Type Name Java Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Var Name Java Var invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name Java Var invalid Format
[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
```

```
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name Java Type invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name Java Type invalid Format
[Test OK] Result: (Expected: S2peedController == Result:
   ↪ S2peedController)

Test Load Type Name Java Type invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Load Type Name Java Type invalid Format
[Test OK] Result: (Expected:  == Result: )

Test Save LineFormat Java Type
[Test OK] Result: (Expected: class SpeedController
 == Result: class SpeedController
)

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


****************************************


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

Normal Operating Parser
[Test OK] Result: (Expected: true == Result: true)

.AddType() - add empty type to parser
[Test OK] Result: (Expected: ERROR: Provided string is empty.
   ↪ == Result: ERROR: Provided string is empty.)

.AddVariable() - add empty type to factory
[Test OK] Result: (Expected: ERROR: Provided string is empty.
   ↪ == Result: ERROR: Provided string is empty.)

.AddVariable() - add empty var to factory
```

```
[Test OK] Result: (Expected: ERROR: Provided string is empty.
    ↪ == Result: ERROR: Provided string is empty.)

.AddVariable() - add variable with nonexisting type
[Test OK] Result: (Expected: ERROR: Provided type does not
    ↪ exist. == Result: ERROR: Provided type does not exist.)

.AddType() - add duplicate type
[Test OK] Result: (Expected: ERROR: Provided type already
    ↪ exists. == Result: ERROR: Provided type already exists.)

.AddVar() - add duplicate Var
[Test OK] Result: (Expected: ERROR: Provided Variable already
    ↪ exists. == Result: ERROR: Provided Variable already
    ↪ exists.)

Test Store and Load Java Fact with exeption Dup Type
[Test OK] Result: (Expected: ERROR: Provided type already
    ↪ exists. == Result: ERROR: Provided type already exists.)

Test Store and Load IEC Fact with exeption Dup Type
[Test OK] Result: (Expected: ERROR: Provided type already
    ↪ exists. == Result: ERROR: Provided type already exists.)


****************************************

TEST OK!!
```

6 Quellcode

6.1 Object.hpp

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

6.2 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 <map>

#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(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.
     *
     * \param fact
     */
    SymbolParser(ISymbolFactory& fact);

    virtual ~SymbolParser();

    // Delete CopyCtor and Assign Op to prevent untestet behaviour.
    SymbolParser(SymbolParser& s) = delete;
    void operator=(SymbolParser s) = delete;
```

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

6.3 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;

    try {
        while (getline(type_File, line)) {
```

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

```
146
147     // check if type already exists
148     auto it = find_if(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& t) { return t->GetName()
            == name;});
149
150     if (it != m_typeCont.cend()) {
151         std::cerr << "Error type already exists !! \n";
152         throw ERROR_DUPLICATE_TYPE;
153     }
154
155     Type::Uptr pType = m_Factory->CreateType(name);
156     m_typeCont.push_back(move(pType));
157
158 }
159
160 void SymbolParser::AddVariable(std::string const& name, std::string const& type)
161 {
162     if (m_Factory == nullptr)
163         throw SymbolParser::ERROR_NULLPTR;
164
165     if (name.empty())
166         throw SymbolParser::ERROR_EMPTY_STRING;
167
168     if (type.empty())
169         throw SymbolParser::ERROR_EMPTY_STRING;
170
171     // check if variable already exists
172     auto it = find_if(m_variableCont.cbegin(), m_variableCont.cend(),
173         [&](const auto& t) { return t->GetTypeName() == type && t->GetName() == name;});
174
175     // instead of a fixed output to the console
176     // an exception is thrown!!
177     if (it != m_variableCont.cend()) {
178         std::cerr << "Error Variable already exists !! \n";
179         throw ERROR_DUPLICATE_VAR;
180     }
181
182     // look up if type even exists if yes add to type container
183     for (const auto& m_type : m_typeCont)
184     {
185         if (type == m_type->GetName())
186         {
187             auto pVar = m_Factory->CreateVariable(name);
188             pVar->SetType(m_type);
189
190             // Move ownership into container
191             m_variableCont.push_back(std::move(pVar));
192
193             // If each variable should only match one type, return early
194             return;
195         }
196     }
197
198     // Error is thrown instead of a console output!
199     // in our opinion this is more flexible than a
200     // fixed output to the console!!
201     // but here it is anyway
202     std::cerr << "Error Type for Variable does not exist !! \n";
203
204     throw ERROR_NONEXISTING_TYPE;
205 }
206
207 SymbolParser::SymbolParser(ISymbolFactory& fact) : m_Factory{ &fact }
208 {
209     // Load State from previos parsing
210     LoadNewState(m_Factory->GetTypeFileName(), m_Factory->GetVariableFileName());
211 }
212
213 SymbolParser::~SymbolParser()
214 {
215     SaveState(m_Factory->GetTypeFileName(), m_Factory->GetVariableFileName());
216 }
```

6.4 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
```

6.5 Identifier.hpp

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

#include <memory>
#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
         */
        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
```

6.6 Identifier.cpp

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

#include "Identifier.hpp"


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;
}
```

6.7 Variable.hpp

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

```
73 };
74 #endif
```

6.8 Variable.cpp

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

#include "Variable.hpp"
#include <cassert>


using namespace std;


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

    m_type = std::move(type);
}

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

6.9 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
```

6.10 Type.cpp

```
1 /*****************************************************************//**
2  * \file   Type.cpp
3  * \brief  Abstract class for parsing types
4  * \author Simon Vogelhuber
5  * \date   Dezember 2025
6  *********************************************************************/
7
8 #include "Type.hpp"
```

6.11 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:
	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
```

6.12 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
```

6.13 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 = scan.get_identifier();
        scan.next_symbol();
        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;
        varName = scan.get_identifier();
        scan.next_symbol();

        // The line should be emty after the var Name!
        if (scan.is(';'))       return varName;
        else                                    return "";
}

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
{
        stringstream lineStream;
        lineStream << fileLine;
        scanner scan{lineStream};

        return ScanTypeName(scan);
}

std::string JavaVariable::LoadVarName(std::string const& fileLine) const
{
        stringstream lineStream;
        lineStream << fileLine;
        scanner scan{ lineStream };

        string typeName = ScanTypeName(scan);
        string varName = ScanVarName(scan);
        if (typeName.empty()) varName = "";
```

```
73
74          return varName;
75  }
```

6.14 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
```

6.15 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;
}
```

6.16 IECVariable.hpp

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

6.17 IECVariable.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) {
	string TypeName;

	if (scan.get_identifier() == "VAR") {
		scan.next_symbol();
		TypeName = scan.get_identifier();
		scan.next_symbol();
		return TypeName;
	}

	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;

	if (scan.is(':')) {
		scan.next_symbol();
		VarName = scan.get_identifier();
		scan.next_symbol();
		if (!scan.is(';')) {
			VarName = "";
		}
	}

	return VarName;
}


std::string IECVariable::LoadTypeName(std::string const& fileLine) const
{
	stringstream converter;
	converter << fileLine;
	scanner Scan;

```

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

6.18 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
```

6.19 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;
}
```

6.20 main.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);



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" }) };

        IECVariable IECVar{ "asdf" };
        IECVar.SetType(Itype);

```

```
73          JavaVariable JavaVar{ "jklm" };
74          JavaVar.SetType(Jtyp);
75
76          cout << "\n␣\n****␣Test␣IEC␣Var␣Getter␣****\n␣\n␣";
77          TestOK = TestOK && TestVariable(&IECVar, "asdf", Itype);
78
79          cout << "\n␣\n****␣Test␣Java␣Var␣Getter␣****\n␣\n␣";
80          TestOK = TestOK && TestVariable(&JavaVar, "jklm", Jtyp);
81
82          cout << "\n␣\n****␣Test␣IEC␣Type␣Getter␣****\n␣\n␣";
83          TestOK = TestOK && TestType(Itype);
84
85          cout << "\n␣\n****␣Test␣Java␣Type␣Getter␣****\n␣\n␣";
86          TestOK = TestOK && TestType(Jtyp);
87
88          TestOK = TestOK && TestIECVar();
89
90          TestOK = TestOK && TestJavaVar();
91
92          TestOK = TestOK && TestIECType();
93
94          TestOK = TestOK && TestJavaType();
95
96          TestOK = TestOK && TestSymbolParser();
97
98          if (WriteOutputFile) {
99
100             // Erase previos Symbol files for test cases
101             EraseFile("IECTypes.sym");
102             EraseFile("IECVars.sym");
103             EraseFile("JavaTypes.sym");
104             EraseFile("JavaVars.sym");
105
106
107             Type::Sptr Itype1{ make_shared<IECType>(IECType{ "int" }) };
108
109             Type::Sptr Jtyp1{ make_shared<JavaType>(JavaType{ "int" }) };
110
111             IECVariable IECVar1{ "asdf" };
112             IECVar1.SetType(Itype1);
113
114             JavaVariable JavaVar1{ "jklm" };
115             JavaVar1.SetType(Jtyp1);
116
117             output << TestStart;
118
119             output << "\n␣\n****␣Test␣IEC␣Var␣Getter␣****\n␣\n␣";
120             TestOK = TestOK && TestVariable(&IECVar1, "asdf", Itype1, output);
121
122             output << "\n␣\n****␣Test␣Java␣Var␣Getter␣****\n␣\n␣";
123             TestOK = TestOK && TestVariable(&JavaVar1, "jklm", Jtyp1, output);
124
125             output << "\n␣\n****␣Test␣IEC␣Type␣Getter␣****\n␣\n␣";
126             TestOK = TestOK && TestType(Itype1, output);
127
128             output << "\n␣\n****␣Test␣Java␣Type␣Getter␣****\n␣\n␣";
129             TestOK = TestOK && TestType(Jtyp1, output);
130
131             TestOK = TestOK && TestIECVar(output);
132
133             TestOK = TestOK && TestJavaVar(output);
134
135             TestOK = TestOK && TestIECType(output);
136
137             TestOK = TestOK && TestJavaType(output);
138
139             TestOK = TestOK && TestSymbolParser(output);
140
141             if (TestOK) {
142                 output << TestCaseOK;
143             }
144             else {
145                 output << TestCaseFail;
146             }
147
```

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

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

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

```
362
363         const string Inv2LineToDecode = "VAR␣mCont␣:␣SpeedController";
364         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>("mCont"), var.LoadTypeName(Inv2LineToDecode));
365         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv2LineToDecode));
366
367         const string Inv3LineToDecode = "Var␣mCont␣:␣SpeedController;";
368         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(Inv3LineToDecode));
369         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv3LineToDecode));
370
371         const string Inv4LineToDecode = "VAR␣mCont␣:␣12343;";
372         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>("mCont"), var.LoadTypeName(Inv4LineToDecode));
373         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv4LineToDecode));
374
375         const string Inv5LineToDecode = "VAR␣123␣:␣a12343;";
376         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(Inv5LineToDecode));
377         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv5LineToDecode));
378
379         const string Inv6LineToDecode = "";
380         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(Inv6LineToDecode));
381         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣IEC␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv6LineToDecode));
382
383         Type::Sptr IECTyp = make_shared<IECType>( IECType{} );
384         var.SetName(var.LoadVarName(LineToDecode));
385         IECTyp->SetName(var.LoadTypeName(LineToDecode));
386         var.SetType(IECTyp);
387
388         TestOK == TestOK && check_dump(ost, "Test␣Save␣LineFormat␣IEC␣Variable", LineToDecode, var.
                GetSaveLine());
389
390         IECVariable IVar;
391
392         TestOK == TestOK && check_dump(ost, "Test␣Save␣LineFormat␣IEC␣Variable", static_cast<string>(""
                ), IVar.GetSaveLine());
393
394     }
395     catch (const string& err) {
396         error_msg = err;
397     }
398     catch (bad_alloc const& error) {
399         error_msg = error.what();
400     }
401     catch (const exception& err) {
402         error_msg = err.what();
403     }
404     catch (...) {
405         error_msg = "Unhandelt␣Exception";
406     }
407
408     TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣TestCase", true, error_msg.empty());
409     error_msg.clear();
410
411     ost << TestEnd;
412
413     return TestOK;
414 }
415
416 bool TestJavaVar(ostream& ost)
417 {
418     assert(ost.good());
419
420     ost << TestStart;
421
422
423     bool TestOK = true;
424     string error_msg;
```

```
425
426     try {
427
428         JavaVariable var;
429
430         const string LineToDecode = "mCont␣mBut;\n";
431         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var",static_cast<string>("mCont"),
                var.LoadTypeName(LineToDecode));
432         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var",static_cast<string>("mBut"),
                var.LoadVarName(LineToDecode));
433
434         const string InvLineToDecode = "1mCont␣mBut;";
435         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(InvLineToDecode));
436         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(InvLineToDecode));
437
438         const string Inv2LineToDecode = "mCont␣;mBut;";
439         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>("mCont"), var.LoadTypeName(Inv2LineToDecode));
440         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv2LineToDecode));
441
442         const string Inv3LineToDecode = "2mCont␣mBut;";
443         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(Inv3LineToDecode));
444         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv3LineToDecode));
445
446         const string Inv4LineToDecode = "mCont␣123;";
447         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>("mCont"), var.LoadTypeName(Inv4LineToDecode));
448         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv4LineToDecode));
449
450         const string Inv5LineToDecode = "123␣:␣a12343;";
451         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(Inv5LineToDecode));
452         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv5LineToDecode));
453
454         const string Inv6LineToDecode = "";
455         TestOK == TestOK && check_dump(ost,"Test␣Load␣Type␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadTypeName(Inv6LineToDecode));
456         TestOK == TestOK && check_dump(ost,"Test␣Load␣Var␣Name␣Java␣Var␣invalid␣Format",static_cast<
                string>(""), var.LoadVarName(Inv6LineToDecode));
457
458         Type::Sptr JTyp = make_shared<JavaType>(JavaType{});
459         var.SetName(var.LoadVarName(LineToDecode));
460         JTyp->SetName(var.LoadTypeName(LineToDecode));
461         var.SetType(JTyp);
462
463         TestOK == TestOK && check_dump(ost, "Test␣Save␣LineFormat␣IEC␣Variable", LineToDecode, var.
                GetSaveLine());
464
465         JavaVariable JVar;
466
467         TestOK == TestOK && check_dump(ost, "Test␣Save␣LineFormat␣IEC␣Variable", static_cast<string>(""
                ), JVar.GetSaveLine());
468     }
469     catch (const string& err) {
470         error_msg = err;
471     }
472     catch (bad_alloc const& error) {
473         error_msg = error.what();
474     }
475     catch (const exception& err) {
476         error_msg = err.what();
477     }
478     catch (...) {
479         error_msg = "Unhandelt␣Exception";
480     }
481
482     TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣TestCase", true, error_msg.empty());
483     error_msg.clear();
```

```
484
485     ost << TestEnd;
486
487     return TestOK;
488 }
489
490 bool TestIECType(ostream& ost)
491 {
492     assert(ost.good());
493
494     ost << TestStart;
495
496
497     bool TestOK = true;
498     string error_msg;
499
500     try{
501         IECType typ;
502
503         const string LineToDecode = "TYPE␣SpeedController\n";
504         TestOK == TestOK && check_dump(ost, "Test␣Load␣Type␣Name␣IEC␣Type", static_cast<string>("
                SpeedController"), typ.LoadTypeName(LineToDecode));
505
506         const string InvLineToDecode = "1TYPE␣SpeedController";
507         TestOK == TestOK && check_dump(ost, "Test␣Load␣Type␣Name␣IEC␣Type␣invalid␣Format", static_cast<
                string>(""), typ.LoadTypeName(InvLineToDecode));
508
509         const string Inv2LineToDecode = "TYPE␣1SpeedController";
510         TestOK == TestOK && check_dump(ost, "Test␣Load␣Type␣Name␣IEC␣Type␣invalid␣Format", static_cast<
                string>(""), typ.LoadTypeName(Inv2LineToDecode));
511
512         const string Inv3LineToDecode = "TYPE␣S2peedController";
513         TestOK == TestOK && check_dump(ost, "Test␣Load␣Type␣Name␣IEC␣Type␣invalid␣Format", static_cast<
                string>("S2peedController"), typ.LoadTypeName(Inv3LineToDecode));
514
515         const string Inv4LineToDecode = "TYPE␣SpeedController;";
516         TestOK == TestOK && check_dump(ost, "Test␣Load␣Type␣Name␣IEC␣Type␣invalid␣Format", static_cast<
                string>(""), typ.LoadTypeName(Inv4LineToDecode));
517
518         const string Inv6LineToDecode = "";
519         TestOK == TestOK && check_dump(ost, "Test␣Load␣Type␣Name␣IEC␣Type␣invalid␣Format", static_cast<
                string>(""), typ.LoadTypeName(Inv6LineToDecode));
520
521         typ.SetName(typ.LoadTypeName(LineToDecode));
522
523         TestOK == TestOK && check_dump(ost, "Test␣Save␣LineFormat␣IEC␣Type", LineToDecode, typ.
                GetSaveLine());
524
525     }
526     catch (const string& err) {
527         error_msg = err;
528     }
529     catch (bad_alloc const& error) {
530         error_msg = error.what();
531     }
532     catch (const exception& err) {
533         error_msg = err.what();
534     }
535     catch (...) {
536         error_msg = "Unhandelt␣Exception";
537     }
538
539     TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣TestCase", true, error_msg.empty());
540     error_msg.clear();
541
542     ost << TestEnd;
543
544     return TestOK;
545 }
546
547 bool TestJavaType(ostream& ost)
548 {
549     assert(ost.good());
550
551     ost << TestStart;
```

```
552
553
554     bool TestOK = true;
555     string error_msg;
556
557     try{
558
559         JavaType typ;
560
561         const string LineToDecode = "class SpeedController\n";
562         TestOK == TestOK && check_dump(ost, "Test Load Type Name Java Type", static_cast<string>("
                SpeedController"), typ.LoadTypeName(LineToDecode));
563
564         const string InvLineToDecode = "1class SpeedController";
565         TestOK == TestOK && check_dump(ost, "Test Load Type Name Java Type invalid Format", static_cast
                <string>(""), typ.LoadTypeName(InvLineToDecode));
566
567         const string Inv2LineToDecode = "class 1SpeedController";
568         TestOK == TestOK && check_dump(ost, "Test Load Type Name Java Type invalid Format", static_cast
                <string>(""), typ.LoadTypeName(Inv2LineToDecode));
569
570         const string Inv3LineToDecode = "class S2peedController";
571         TestOK == TestOK && check_dump(ost, "Test Load Type Name Java Type invalid Format", static_cast
                <string>("S2peedController"), typ.LoadTypeName(Inv3LineToDecode));
572
573         const string Inv4LineToDecode = "class SpeedController;";
574         TestOK == TestOK && check_dump(ost, "Test Load Type Name Java Type invalid Format", static_cast
                <string>(""), typ.LoadTypeName(Inv4LineToDecode));
575
576         const string Inv6LineToDecode = "";
577         TestOK == TestOK && check_dump(ost, "Test Load Type Name Java Type invalid Format", static_cast
                <string>(""), typ.LoadTypeName(Inv6LineToDecode));
578
579         typ.SetName(typ.LoadTypeName(LineToDecode));
580
581         TestOK == TestOK && check_dump(ost, "Test Save LineFormat Java Type", LineToDecode, typ.
                GetSaveLine());
582
583     }
584     catch (const string& err) {
585         error_msg = err;
586     }
587     catch (bad_alloc const& error) {
588         error_msg = error.what();
589     }
590     catch (const exception& err) {
591         error_msg = err.what();
592     }
593     catch (...) {
594         error_msg = "Unhandelt Exception";
595     }
596
597     TestOK = TestOK && check_dump(ost, "Test for Exception in TestCase", true, error_msg.empty());
598     error_msg.clear();
599
600     ost << TestEnd;
601
602     return TestOK;
603 }
```

6.21 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>

bool TestSymbolParser(std::ostream& ost = std::cout);

#endif // !1
```

6.22 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
 *********************************************************************/


#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;
    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();
    }
    catch (...) {
        error_msg = "Unhandelt Exception";
```

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

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

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

6.23 Test.hpp

```
/*****************************************************************//**
 * \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
```

```
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
```

```
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
```