Systemdokumentation
Projekt Symbolparser

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 12. November 2025

Inhaltsverzeichnis

1 Organisatorisches

1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: Simon.Offenberger@fh-hagenberg.at
- Susi Sorglos, Matr.-Nr.: yyyy, E-Mail: Susi.Sorglos@fh-hagenberg.at

1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
  - Design Klassendiagramm
  - Implementierung und Test der Klassen:
  - Implementierung des Testtreibers
  - Dokumentation
    - Object
    - Symbolparser
    - ISymbolFactory
    - Variable
    - Type
    - JavaVariable
    - JavaType
    - JavaSymbolFactory
    - IECVariable

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

1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich x Ph
- Simon Vogelhuber: geschätzt x Ph / tatsächlich x Ph

2 Anforderungsdefinition (Systemspezifikation)

Das Ziel ist es einen Symbolparser zu implementieren, der verschiedene Programmiersprachen unterstützt. Der Parser soll in der Lage sein Typen und Variablen zu erkennen und zu verarbeiten. Dazu wird eine Factory benötigt, die die entsprechenden Objekte für die verschiedenen Sprachen erzeugt.

Funktionen des Symbolparsers:

- Auswählen der Programmiersprachen (auswählen der SymbolFactory)
- Speichern der erzeugten Objekte in einem Container.
- Erzeugen von Variablen und Typen über die SymbolFactory
- Überprüfung ob Typen und Variablen gültig sind.
- Beim Wechsel der SymbolFactory, werden alle Objekte der alten 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:

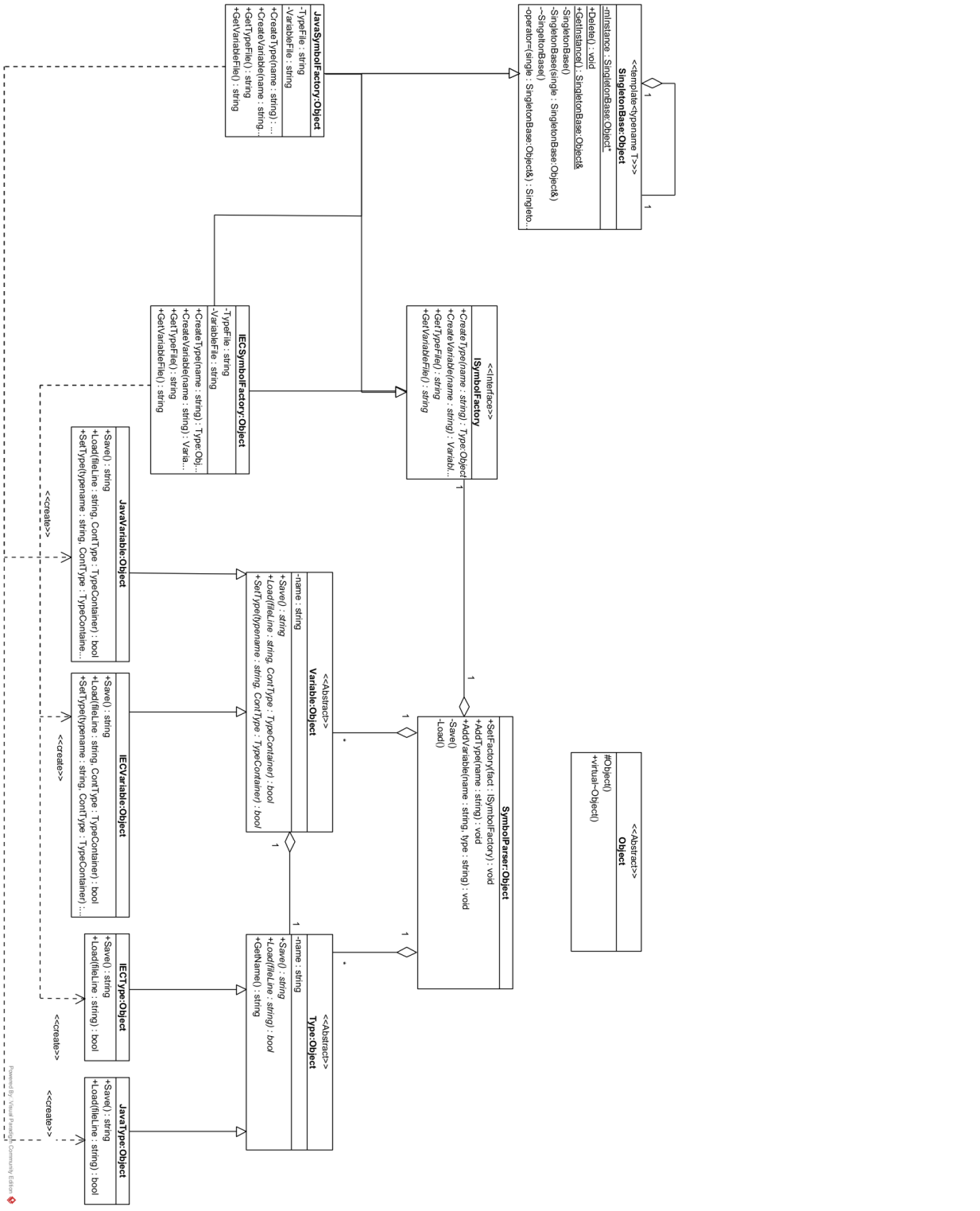
- Speichern des Variablennamens
- Speichern des Variablentyps
- Auswerten der Variablendeklaration (Syntaxprüfung)
- Zurückgeben des Variablennamens
- Zurückgeben des Variablentyps

Funktionen des Type:

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

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Verwendung des Factory-Patterns:
Das Factory-Pattern wurde verwendet, um die Erstellung von Objekten der verschiedenen Programmiersprachen zu kapseln. Das ermöglicht eine einfache Erweiterung des Systems um weitere Sprachen, ohne dass der Symbolparser angepasst werden muss. Der Parser Speichert 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.

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
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: SpeedController == Result: SpeedController)
136
137 Test Load Type Name IEC Var invalid Format
138 [Test OK] Result: (Expected:  == Result: )
139
140 Test Load Var Name IEC Var invalid Format
141 [Test OK] Result: (Expected:  == Result: )
142
143 Test Load Type Name IEC Var invalid Format
144 [Test OK] Result: (Expected: mCont == Result: mCont)
145
146 Test Load Var Name IEC Var invalid Format
147 [Test OK] Result: (Expected:  == Result: )
148
149 Test Load Type Name IEC Var invalid Format
150 [Test OK] Result: (Expected:  == Result: )
151
152 Test Load Var Name IEC Var invalid Format
153 [Test OK] Result: (Expected:  == Result: )
154
155 Test Load Type Name IEC Var invalid Format
156 [Test OK] Result: (Expected: mCont == Result: mCont)
157
158 Test Load Var Name IEC Var invalid Format
159 [Test OK] Result: (Expected:  == Result: )
160
161 Test Load Type Name IEC Var invalid Format
162 [Test OK] Result: (Expected:  == Result: )
163
164 Test Load Var Name IEC Var invalid Format
165 [Test OK] Result: (Expected:  == Result: )
166
167 Test Load Type Name IEC Var invalid Format
168 [Test OK] Result: (Expected:  == Result: )
```

```
169
170 Test Load Var Name IEC Var invalid Format
171 [Test OK] Result: (Expected:  == Result: )
172
173 Test Save LineFormat IEC Variable
174 [Test OK] Result: (Expected: VAR mCont : SpeedController; == Result: VAR
    ↪ mCont : SpeedController;)
175
176 Test Save LineFormat IEC Variable
177 [Test OK] Result: (Expected:  == Result: )
178
179
180 ****************************************
181
182
183 ****************************************
184              TESTCASE START
185 ****************************************
186
187 Test Load Type Name Java Var
188 [Test OK] Result: (Expected: mCont == Result: mCont)
189
190 Test Load Var Name Java Var
191 [Test OK] Result: (Expected: mBut == Result: mBut)
192
193 Test Load Type Name Java Var invalid Format
194 [Test OK] Result: (Expected:  == Result: )
195
196 Test Load Var Name Java Var invalid Format
197 [Test OK] Result: (Expected:  == Result: )
198
199 Test Load Type Name Java Var invalid Format
200 [Test OK] Result: (Expected: mCont == Result: mCont)
201
202 Test Load Var Name Java Var invalid Format
203 [Test OK] Result: (Expected:  == Result: )
204
205 Test Load Type Name Java Var invalid Format
206 [Test OK] Result: (Expected:  == Result: )
207
208 Test Load Var Name Java Var invalid Format
209 [Test OK] Result: (Expected:  == Result: )
210
211 Test Load Type Name Java Var invalid Format
```

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

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:  == 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:  == Result: )

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

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

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


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


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

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

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

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

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

Test Load Type Name IEC Type invalid Format
```

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

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

Test Save LineFormat IEC Type
[Test OK] Result: (Expected: TYPE SpeedController
 == Result: TYPE SpeedController
)


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


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

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

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
)


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

```

```
300
301 ****************************************
302              TESTCASE START
303 ****************************************
304
305 Normal Operating Parser
306 [Test OK] Result: (Expected: true == Result: true)
307
308 .AddType() - add empty type to factory
309 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
    ↪ ERROR: Provided string is empty.)
310
311 .AddVariable() - add empty type to factory
312 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
    ↪ ERROR: Provided string is empty.)
313
314 .AddVariable() - add empty var to factory
315 [Test OK] Result: (Expected: ERROR: Provided string is empty. == Result:
    ↪ ERROR: Provided string is empty.)
316
317 .AddVariable() - add variable with nonexisting type
318 [Test OK] Result: (Expected: ERROR: Provided type does not exist. == Result
    ↪ : ERROR: Provided type does not exist.)
319
320
321 ****************************************
322
323 TEST OK!!
```

6 Quellcode

6.1 Object.hpp

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

6.2 Symbolparser.hpp

```
/*****************************************************************//**
 * \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.";

    /**
     * \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);

    SymbolParser(ISymbolFactory & fact) : m_Factory{fact} {}

protected:
private:
    /**
     * \brief Saves the current state of a SymbolFacotry to its file
     * \param string of type files path, tring of variable files path
     * \return void
     */
    void SaveState(const std::string& type_file, const std::string& var_file);

    /**
     * \brief Loads a SymbolFactory's variables and types from file
     * \param string of type files path, tring of variable files path
     * \return void
     */
```

```
73      void LoadNewState(const std::string& type_file, const std::string& var_file);
74
75      TTypeCont m_typeCont;
76      TVariableCont m_variableCont;
77      ISymbolFactory & m_Factory;
78  };
79  #endif
```

6.3 Symbolparser.cpp

```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 "SymbolParser.hpp"
#include "ISymbolFactory.hpp"

using namespace std;

void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)
{
    ofstream type_File;
    ofstream var_File;

    type_File.open(m_Factory.GetTypeFileName());

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

    type_File.close();


    var_File.open(m_Factory.GetVariableFileName());

    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)
{
    ifstream type_File;
    ifstream var_File;

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

    type_File.open(m_Factory.GetTypeFileName());

    string line;

    while (getline(type_File, line)) {

        Type::Uptr pType = m_Factory.CreateType("");

        pType->SetType(pType->LoadTypeName(line));

        m_typeCont.push_back(move(pType));
    }

    type_File.close();


    var_File.open(m_Factory.GetVariableFileName());

    while (getline(var_File, line)) {

        auto pVar = m_Factory.CreateVariable("");

        const string type = pVar->LoadTypeName(line);
        const string name = pVar->LoadVarName(line);

        pVar->SetName(name);

        // look up if type even exists if yes add to type container
        for (const auto& m_type : m_typeCont)
```

```
        {
            if (type == m_type->GetType())
            {

                pVar->SetType(m_type);

                // If each variable should only match one type, break early
                break;
            }
        }

        if (pVar->GetType() != "") {
            m_variableCont.push_back(move(pVar));
        }
    }

    var_File.close();
}


void SymbolParser::SetFactory(ISymbolFactory& Factory)
{

    SaveState(m_Factory.GetTypeFileName(), m_Factory.GetVariableFileName());

    m_Factory = Factory;

    LoadNewState(m_Factory.GetTypeFileName(), m_Factory.GetVariableFileName());
}
void SymbolParser::AddType(std::string const& name)
{
    if (name.empty())
        throw SymbolParser::ERROR_EMPTY_STRING;

    Type::Uptr pType = m_Factory.CreateType(name);
    m_typeCont.push_back(move(pType));
}

void SymbolParser::AddVariable(std::string const& name, std::string const& type)
{
    if (name.empty())
        throw SymbolParser::ERROR_EMPTY_STRING;

    if (type.empty())
        throw SymbolParser::ERROR_EMPTY_STRING;

    // look up if type even exists if yes add to type container
    for (const auto& m_type : m_typeCont)
    {
        if (type == m_type->GetType())
        {
            auto pVar = m_Factory.CreateVariable(name);
            pVar->SetType(m_type);

            // Move ownership into container
            m_variableCont.push_back(std::move(pVar));

            // If each variable should only match one type, return early
            return;
        }
    }

    throw ERROR_NONEXISTING_TYPE;
}
```

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)=0;

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

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

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

protected:
private:
};
#endif
```

6.5 Variable.hpp

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

#ifndef VARIABLE_HPP
#define VARIABLE_HPP
#include <memory>
#include <vector>
#include <string>

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

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


    inline static const std::string ERROR_EMPTY_STRING = "ERROR: Empty String";

    /**
     * \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
     */
    void SetType(Type::Sptr type);

    /**
     * \brief Retuns the name of a variable
     *
     * \return string name
     */
    std::string GetName() const;

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

```

```
73      /**
74       * \brief Sets name of variable
75       *
76       * \return void
77       */
78      void SetName(const std::string & name);
79
80  protected:
81      Variable(const std::string& name) : m_name{ name } {}
82      Variable() = default;
83      std::string m_name;
84      Type::Sptr m_type;
85  private:
86  };
87  #endif
```

6.6 Variable.cpp

```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::GetName() const
{
    return m_name;
}

std::string Variable::GetType() const
{
    return m_type->GetType();
}

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

    m_name = name;
}
```

6.7 Type.hpp

```
/*****************************************************************//**
 * \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 "Object.h"

class Type : public Object
{
public:

   inline static const std::string ERROR_EMPTY_STRING = "ERROR: Empty String";

   /**
    * \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 Getter for type name
    *
    * \return string of type
    */
   std::string GetType() const;

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

   /**
    * \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;

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

protected:
   std::string m_name;
private:
};
#endif
```

6.8 Type.cpp

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

#include "Type.hpp"


std::string Type::GetType() const{
   return m_name;
}

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

   m_name = name;
}
```

6.9 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 << expect
   << std::noboolalpha << std::endl << std::endl;
111       }
112       else {
113          ostr << testcase << std::endl << colorRed << "[Test␣FAILED]␣" << colorWhite << "Result:␣(Expected:␣" << std::boolalpha << expe
   << 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:␣"
118       }
119       else {
120          ostr << testcase << std::endl  << "[Test␣FAILED]␣"  << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣!=" << "␣Resul
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
```

6.10 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:
   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
```

6.11 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
```

6.12 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;

static std::string ScanTypeName(scanner& scan)
{
   string typeName = scan.get_identifier();
   scan.next_symbol();
   return typeName;
}

static std::string ScanVarName(scanner& scan)
{
   string varName;
   varName = scan.get_identifier();

   return varName;
}

std::string JavaVariable::GetSaveLine() const
{
   stringstream line;

   if (m_type != nullptr) {
      line << m_type->GetType() << "␣" << m_name;
   }

   return line.str();
}

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 = "";

   return varName;
}
```

6.13 JavaSymbolFactory.hpp

```
/*****************************************************************//**
 * \file   JavaSymbolFactory.hpp
 * \brief  A factory for parsing java variables and types
 * \author Simon
 * \date   Dezember 2025
 *********************************************************************/

#ifndef JAVA_SYMBOL_FACTORY_HPP
#define JAVA_SYMBOL_FACTORY_HPP
#include "Object.h"
#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 uniqie pointer to variable
    */
   virtual Variable::Uptr CreateVariable(const std::string& name) override;

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

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

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

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

6.14 JavaSymbolFactory.cpp

```cpp
/*****************************************************************//**
 * \file   JavaSymbolFactory.cpp
 * \brief  A factory for parsing 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)
{
    return std::make_unique<JavaVariable>(JavaVariable{ name });
}

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

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

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

6.15 IECVariable.hpp

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

6.16 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
{
   stringstream line;

   if (m_type != nullptr) {
      line << "VAR " << m_type->GetType() << " : " << m_name << ";";
   }

   return line.str();
}

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 "";
}

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;

   Scan.set_istream(converter);

   return ScanTypeName(Scan);
}

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

```
73
74      Scan.set_istream(converter);
75
76      string Typename = ScanTypeName(Scan);
77      string VarName = ScanVarName(Scan);
78
79      if (Typename.empty()) VarName = "";
80
81      return VarName;
82  }
```

6.17 IECSymbolFactory.hpp

```cpp
/*****************************************************************//**
 * \file   IECSymbolFactory.hpp
 * \brief  A factory for parsing 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:

   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) override;

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

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

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

protected:
private:
   IECSymbolFactory() = default;
   const std::string m_TypeFileName = "IECTypes.sym";
   const std::string m_VariableFileName  =  "IECVars.sym";
};

#endif
```

6.18 IECSymbolFactory.cpp

```cpp
/*****************************************************************//**
 * \file   IECSymbolFactory.cpp
 * \brief  A factory for parsing 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)
{
    return std::make_unique<IECVariable>(IECVariable{name});
}

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

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

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