

Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku

Mat.Nr: S2410306027 / S2410306014

Punkte:

Übungsgruppe: 1

korrigiert:

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

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

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

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

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

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

*JavaTypes.sym:*

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

*JavaVars.sym:*

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

*IECTypes.sym:*

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

*IECVars.sym:*

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

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

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

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

Eine mögliche Anwendung im Hauptprogramm könnte so aussehen:

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

```

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

```

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

Treffen Sie für alle unzureichenden Angaben sinnvolle Annahmen und begründen Sie diese. Verfassen Sie weiters eine Systemdokumentation (entsprechend den Vorgaben aus Übung 1)!

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



**HSD**

---

**FH-HAGENBERG**

# **Systemdokumentation Projekt Symbolparser**

**Version 1.0**

S. Offenberger, S. Vogelhuber

Hagenberg, 13. November 2025

# Inhaltsverzeichnis

<b>1</b>	<b>Organisatorisches</b>	<b>7</b>
1.1	Team . . . . .	7
1.2	Aufteilung der Verantwortlichkeitsbereiche . . . . .	7
1.3	Aufwand . . . . .	8
<b>2</b>	<b>Anforderungsdefinition (Systemspezifikation)</b>	<b>9</b>
<b>3</b>	<b>Systementwurf</b>	<b>12</b>
3.1	Klassendiagramm . . . . .	12
3.2	Designentscheidungen . . . . .	13
<b>4</b>	<b>Dokumentation der Komponenten (Klassen)</b>	<b>14</b>
<b>5</b>	<b>Testprotokollierung</b>	<b>15</b>
<b>6</b>	<b>Quellcode</b>	<b>24</b>
6.1	Object.hpp . . . . .	24
6.2	Symbolparser.hpp . . . . .	25
6.3	Symbolparser.cpp . . . . .	27
6.4	ISymbolFactory.hpp . . . . .	30
6.5	Identifier.hpp . . . . .	31
6.6	Identifier.cpp . . . . .	32
6.7	Variable.hpp . . . . .	33
6.8	Variable.cpp . . . . .	35
6.9	Type.hpp . . . . .	36
6.10	Type.cpp . . . . .	37
6.11	Test.hpp . . . . .	38
6.12	SingetonBase.hpp . . . . .	41
6.13	JavaVariable.hpp . . . . .	42
6.14	JavaVariable.cpp . . . . .	43
6.15	JavaSymbolFactory.hpp . . . . .	45
6.16	JavaSymbolFactory.cpp . . . . .	46
6.17	IECVariable.hpp . . . . .	47
6.18	IECVariable.cpp . . . . .	48
6.19	IECSymbolFactory.hpp . . . . .	50

---

6.20 IECSymbolFactory.cpp . . . . .	51
-------------------------------------	----

# 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 8 Ph
- Simon Vogelhuber: geschätzt 10 Ph / tatsächlich 10 Ph



## 2 Anforderungsdefinition (Systemspezifikation)

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

### **Funktionen des Symbolparsers:**

- Auswählen der Programmiersprachen (auswählen der SymbolFactory)
- Speichern der erzeugten Objekte in einem Container.
- Erzeugen von Variablen und Typen über die SymbolFactory
- Überprüfung ob Typen und Variablen gültig sind.
- Beim Wechsel der SymbolFactory, werden alle Objekte der alten Factory in ein File gespeichert. Und die Objekte der neuen Factory werden aus dem File geladen.

### **Funktionen der SymbolFactory:**

- Erzeugen von Variablen und Typen der jeweiligen Programmiersprache.

### **Funktionen der Variable:**

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

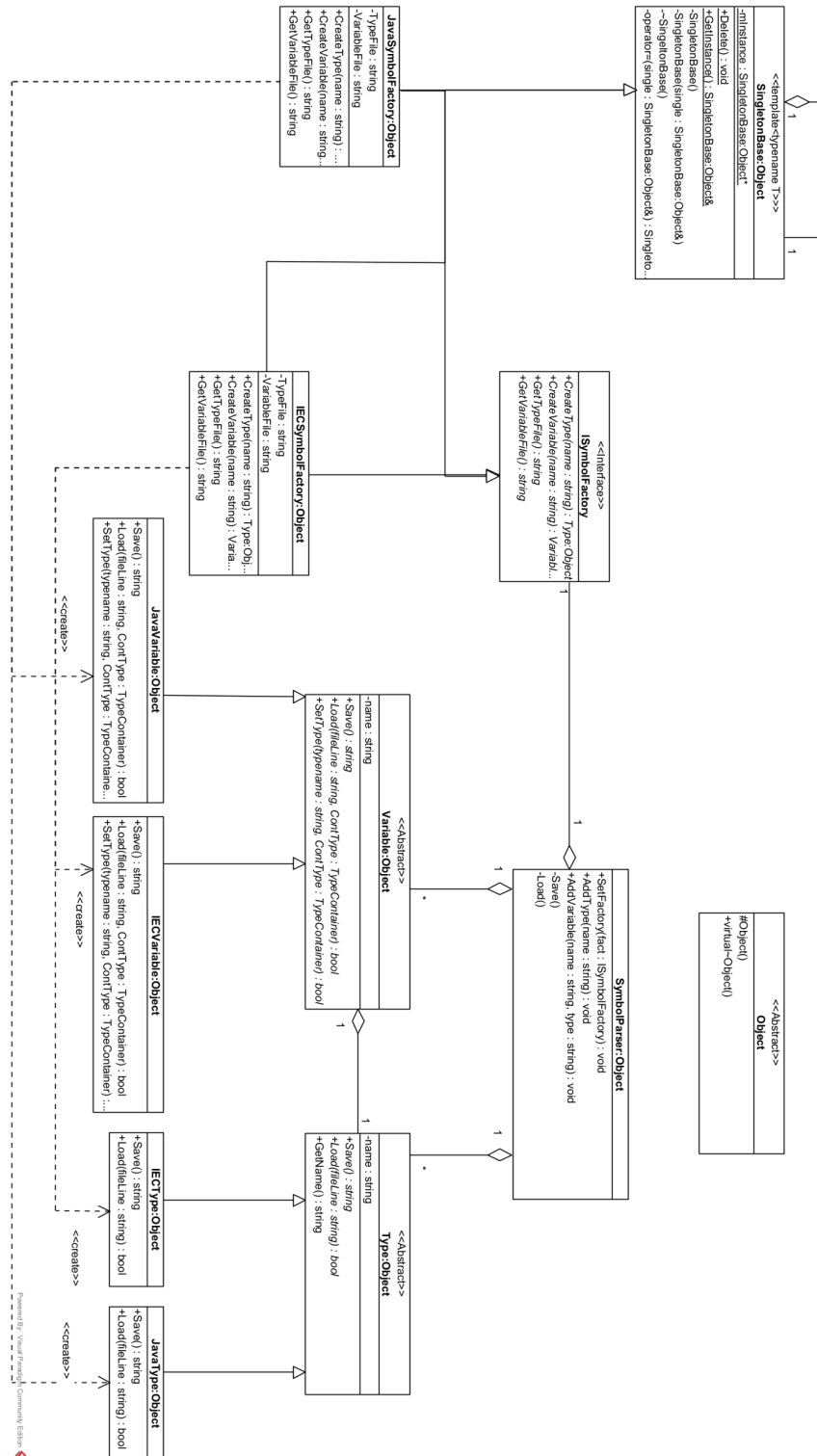
### **Funktionen des Type:**

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



## 3 Systementwurf

### 3.1 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 hierfür 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](http://../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_fast256_t == Result:
    ↪ uint_fast256_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;
175 == Result: VAR mCont : SpeedController;
176 )
177
178 Test Save LineFormat IEC Variable
179 [Test OK] Result: (Expected: == Result: )
180
181 Test for Exception in TestCase
182 [Test OK] Result: (Expected: true == Result: true)
183
184
185 *****
186
187
188 *****
189 TESTCASE START
190 *****
191
192 Test Load Type Name Java Var
193 [Test OK] Result: (Expected: mCont == Result: mCont)
194
195 Test Load Var Name Java Var
196 [Test OK] Result: (Expected: mBut == Result: mBut)
197
198 Test Load Type Name Java Var invalid Format
199 [Test OK] Result: (Expected: == Result: )
200
201 Test Load Var Name Java Var invalid Format
202 [Test OK] Result: (Expected: == Result: )
203
204 Test Load Type Name Java Var invalid Format
205 [Test OK] Result: (Expected: mCont == Result: mCont)
```

```
206
207 Test Load Var Name Java Var invalid Format
208 [Test OK] Result: (Expected: == Result: )
209
210 Test Load Type Name Java Var invalid Format
211 [Test OK] Result: (Expected: == Result: )
212
213 Test Load Var Name Java Var invalid Format
214 [Test OK] Result: (Expected: == Result: )
215
216 Test Load Type Name Java Var invalid Format
217 [Test OK] Result: (Expected: mCont == Result: mCont)
218
219 Test Load Var Name Java Var invalid Format
220 [Test OK] Result: (Expected: == Result: )
221
222 Test Load Type Name Java Var invalid Format
223 [Test OK] Result: (Expected: == Result: )
224
225 Test Load Var Name Java Var invalid Format
226 [Test OK] Result: (Expected: == Result: )
227
228 Test Load Type Name Java Var invalid Format
229 [Test OK] Result: (Expected: == Result: )
230
231 Test Load Var Name Java Var invalid Format
232 [Test OK] Result: (Expected: == Result: )
233
234 Test Save LineFormat IEC Variable
235 [Test OK] Result: (Expected: mCont mBut
236 == Result: mCont mBut
237 )
238
239 Test Save LineFormat IEC Variable
240 [Test OK] Result: (Expected: == Result: )
241
242 Test for Exception in TestCase
243 [Test OK] Result: (Expected: true == Result: true)
244
245
246 *****
247
248
249 *****
```

```
250             TESTCASE START
251 *****
252
253 Test Load Type Name IEC Type
254 [Test OK] Result: (Expected: SpeedController == Result:
    ↪ SpeedController)
255
256 Test Load Type Name IEC Type invalid Format
257 [Test OK] Result: (Expected:  == Result: )
258
259 Test Load Type Name IEC Type invalid Format
260 [Test OK] Result: (Expected:  == Result: )
261
262 Test Load Type Name IEC Type invalid Format
263 [Test OK] Result: (Expected: S2speedController == Result:
    ↪ S2speedController)
264
265 Test Load Type Name IEC Type invalid Format
266 [Test OK] Result: (Expected:  == Result: )
267
268 Test Load Type Name IEC Type invalid Format
269 [Test OK] Result: (Expected:  == Result: )
270
271 Test Save LineFormat IEC Type
272 [Test OK] Result: (Expected: TYPE SpeedController
273 == Result: TYPE SpeedController
274 )
275
276 Test for Exception in TestCase
277 [Test OK] Result: (Expected: true == Result: true)
278
279
280 *****
281
282
283 *****
284             TESTCASE START
285 *****
286
287 Test Load Type Name Java Type
288 [Test OK] Result: (Expected: SpeedController == Result:
    ↪ SpeedController)
289
290 Test Load Type Name Java Type invalid Format
```

```
291 [Test OK] Result: (Expected: == Result: )
292
293 Test Load Type Name Java Type invalid Format
294 [Test OK] Result: (Expected: == Result: )
295
296 Test Load Type Name Java Type invalid Format
297 [Test OK] Result: (Expected: S2speedController == Result:
    ↪ S2speedController)
298
299 Test Load Type Name Java Type invalid Format
300 [Test OK] Result: (Expected: == Result: )
301
302 Test Load Type Name Java Type invalid Format
303 [Test OK] Result: (Expected: == Result: )
304
305 Test Save LineFormat Java Type
306 [Test OK] Result: (Expected: class SpeedController
307 == Result: class SpeedController
308 )
309
310 Test for Exception in TestCase
311 [Test OK] Result: (Expected: true == Result: true)
312
313
314 *****
315
316
317 *****
318 TESTCASE START
319 *****
320
321 Normal Operating Parser
322 [Test OK] Result: (Expected: true == Result: true)
323
324 .AddType() - add empty type to parser
325 [Test OK] Result: (Expected: ERROR: Provided string is empty.
    ↪ == Result: ERROR: Provided string is empty.)
326
327 .AddVariable() - add empty type to factory
328 [Test OK] Result: (Expected: ERROR: Provided string is empty.
    ↪ == Result: ERROR: Provided string is empty.)
329
330 .AddVariable() - add empty var to factory
```

```
331 [Test OK] Result: (Expected: ERROR: Provided string is empty.  
    ↪ == Result: ERROR: Provided string is empty.)  
332  
333 .AddVariable() - add variable with nonexisting type  
334 [Test OK] Result: (Expected: ERROR: Provided type does not  
    ↪ exist. == Result: ERROR: Provided type does not exist.)  
335  
336 .AddType() - add duplicate type  
337 [Test OK] Result: (Expected: ERROR: Provided type already  
    ↪ exists. == Result: ERROR: Provided type already exists.)  
338  
339 .AddVar() - add duplicate Var  
340 [Test OK] Result: (Expected: ERROR: Provided Variable already  
    ↪ exists. == Result: ERROR: Provided Variable already  
    ↪ exists.)  
341  
342  
343 *****  
344  
345 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 protected:  
25     Object() = default;  
26 };  
27  
28 #endif // !OBJECT_HPP  
29
```



## 6.2 Symbolparser.hpp

```

1  /*****
2  * \file SymbolParser.hpp
3  * \brief A multi language parser for types and variables
4  * \author Simon
5  * \date Dezember 2025
6  *****/
7
8  #ifndef SYMBOL_PARSER_HPP
9  #define SYMBOL_PARSER_HPP
10
11 #include <vector>
12 #include <map>
13
14 #include "Object.h"
15 #include "Variable.hpp"
16 #include "Type.hpp"
17 #include "ISymbolFactory.hpp"
18
19 class SymbolParser : public Object
20 {
21 public:
22     inline static const std::string ERROR_EMPTY_STRING = "ERROR:_Provided_string_is_empty.";
23     inline static const std::string ERROR_NONEXISTING_TYPE = "ERROR:_Provided_type_does_not_exist.";
24     inline static const std::string ERROR_DUPLICATE_TYPE = "ERROR:_Provided_type_already_exists.";
25     inline static const std::string ERROR_DUPLICATE_VAR = "ERROR:_Provided_Variable_already_exists.";
26
27     /**
28      * \brief Polymorphic container for saving variables
29      */
30     using TVariableCont = std::vector<Variable::Uptr>;
31
32     /**
33      * \brief Polymorphic container for saving types
34      */
35     using TTypeCont = std::vector<Type::Sptr>;
36
37     /**
38      * \brief Sets Factory for parsing a language
39      * \brief Previous variables and types of prior factory get saved,
40      * \brief then the subsequent factories variables and types get loaded.
41      * \param Reference to a SymbolFactory
42      * \return void
43      */
44     void SetFactory(ISymbolFactory& Factory);
45
46     /**
47      * \brief Adds a new type to the language
48      * \param string of typename
49      * \return void
50      */
51     void AddType(std::string const& name);
52
53     /**
54      * \brief Adds a new variable if type exists
55      * \param string of variable, string of type
56      * \return void
57      */
58     void AddVariable(std::string const& name, std::string const& type);
59
60     /**
61      * \brief CTOR of a Symbol Parser Object.
62      * \param fact
63      */
64     SymbolParser(ISymbolFactory & fact) : m_Factory{fact} {}
65
66     // Delete CopyCtor and Assign Op to prevent untested behaviour.
67     SymbolParser(SymbolParser& s) = delete;
68     void operator=(SymbolParser s) = delete;
69
70 protected:

```

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

## 6.3 Symbolparser.cpp

```
1  /*****  
2  * \file    SymbolParser.cpp  
3  * \brief   A multi language parser for types and variables  
4  * \author  Simon  
5  * \date    Dezember 2025  
6  *****/  
7  
8  
9  #include <algorithm>  
10 #include <fstream>  
11 #include <iostream>  
12 #include "SymbolParser.hpp"  
13 #include "ISymbolFactory.hpp"  
14  
15 using namespace std;  
16  
17 void SymbolParser::SaveState(const std::string & type_file, const std::string & var_file)  
18 {  
19     ofstream type_File;  
20     ofstream var_File;  
21  
22     type_File.open(m_Factory.GetTypeFileName());  
23  
24     // check if file is good  
25     if (!type_File.good()) {  
26         type_File.close();  
27         return;  
28     }  
29  
30     for_each(m_typeCont.cbegin(), m_typeCont.cend(), [&](const auto& type) { type_File << type->  
31         GetSaveLine(); });  
32  
33     type_File.close();  
34  
35     var_File.open(m_Factory.GetVariableFileName());  
36  
37     // check if file is good  
38     if (!var_File.good()) {  
39         var_File.close();  
40         return;  
41     }  
42  
43     for_each(m_variableCont.cbegin(), m_variableCont.cend(), [&](const auto& var) { var_File << var->  
44         GetSaveLine(); });  
45  
46     var_File.close();  
47 }  
48  
49 void SymbolParser::LoadNewState(const std::string& type_file, const std::string& var_file)  
50 {  
51     ifstream type_File;  
52     ifstream var_File;  
53  
54     m_typeCont.clear();  
55     m_variableCont.clear();  
56  
57     type_File.open(m_Factory.GetTypeFileName());  
58  
59     // check if file is good  
60     if (!type_File.good()) {  
61         type_File.close();  
62         return;  
63     }  
64  
65     string line;  
66  
67     while (getline(type_File, line)) {  
68         Type::Uptr pType = m_Factory.CreateType("");  
69         pType->SetName(pType->LoadTypeName(line));  
70     }
```

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

```
145         throw SymbolParser::ERROR_EMPTY_STRING;
146
147     if (type.empty())
148         throw SymbolParser::ERROR_EMPTY_STRING;
149
150     // check if variable already exists
151     auto it = find_if(m_variableCont.cbegin(), m_variableCont.cend(),
152         [&](const auto& t) { return t->GetType() == type && t->GetName() == name;});
153
154     if (it != m_variableCont.cend()) throw ERROR_DUPLICATE_VAR;
155
156     // look up if type even exists if yes add to type container
157     for (const auto& m_type : m_typeCont)
158     {
159         if (type == m_type->GetName())
160         {
161             auto pVar = m_Factory.CreateVariable(name);
162             pVar->SetType(m_type);
163
164             // Move ownership into container
165             m_variableCont.push_back(std::move(pVar));
166
167             // If each variable should only match one type, return early
168             return;
169         }
170     }
171
172     throw ERROR_NONEXISTING_TYPE;
173 }
```

## 6.4 ISymbolFactory.hpp

```
1  /*****  
2  * \file    ISymbolFactory.hpp  
3  * \brief   A Interface for creating SymbolFactories  
4  * \author  Simon  
5  * \date    Dezember 2025  
6  *****/  
7  #ifndef ISYMBOL_FACTORY_HPP  
8  #define ISYMBOL_FACTORY_HPP  
9  
10 #include "Variable.hpp"  
11 #include "Type.hpp"  
12  
13 class ISymbolFactory  
14 {  
15 public:  
16     /**  
17     * \brief Creates a variable  
18     *  
19     * \param string of variables name  
20     * \return unique pointer to variable  
21     */  
22     virtual Variable::Uptr CreateVariable(const std::string& name)=0;  
23  
24     /**  
25     * \brief Creates a type  
26     *  
27     * \param string of typename  
28     * \return unique pointer to type  
29     */  
30     virtual Type::Uptr CreateType(const std::string& name)=0;  
31  
32     /**  
33     * \brief Getter for file path of type file  
34     *  
35     * \return string of filePath  
36     */  
37     virtual const std::string& GetTypeFileName()=0;  
38  
39     /**  
40     * \brief Getter for file path of variable file  
41     *  
42     * \return string of filePath  
43     */  
44     virtual const std::string& GetVariableFileName()=0;  
45  
46  
47     virtual ~ISymbolFactory() = default;  
48  
49 protected:  
50 private:  
51 };  
52 #endif
```

## 6.5 Identifier.hpp

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

## 6.6 Identifier.cpp

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



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

```

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

## 6.8 Variable.cpp

```
1  /*****  
2  * \file   Variable.cpp  
3  * \brief  Abstract class for parsing types  
4  * \author  Simon Vogelhuber  
5  * \date   Dezember 2025  
6  *****/  
7  
8  #include "Variable.hpp"  
9  #include <cassert>  
10  
11  using namespace std;  
12  
13  
14  void Variable::SetType(Type::Sptr type)  
15  {  
16      if (type == nullptr) throw Type::ERROR_NULLPTR;  
17      m_type = std::move(type);  
18  }  
19  
20  
21  std::string Variable::GetType() const  
22  {  
23      return m_type->GetName();  
24  }  
25
```

## 6.9 Type.hpp

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

## 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 Test.hpp

```

1  /*****
2  * \file   Test.hpp
3  * \brief  File that provides a Test Function with a formatted output
4  *
5  * \author Simon
6  * \date   April 2025
7  *****/
8  #ifndef TEST_HPP
9  #define TEST_HPP
10
11 #include <string>
12 #include <iostream>
13 #include <vector>
14 #include <list>
15 #include <queue>
16 #include <forward_list>
17
18 #define ON 1
19 #define OFF 0
20 #define COLOR_OUTPUT OFF
21
22 // Definitions of colors in order to change the color of the output stream.
23 const std::string colorRed = "\x1B[31m";
24 const std::string colorGreen = "\x1B[32m";
25 const std::string colorWhite = "\x1B[37m";
26
27 inline std::ostream& RED(std::ostream& ost) {
28     if (ost.good()) {
29         ost << colorRed;
30     }
31     return ost;
32 }
33 inline std::ostream& GREEN(std::ostream& ost) {
34     if (ost.good()) {
35         ost << colorGreen;
36     }
37     return ost;
38 }
39 inline std::ostream& WHITE(std::ostream& ost) {
40     if (ost.good()) {
41         ost << colorWhite;
42     }
43     return ost;
44 }
45
46 inline std::ostream& TestStart(std::ostream& ost) {
47     if (ost.good()) {
48         ost << std::endl;
49         ost << "*****" << std::endl;
50         ost << "      TESTCASE_START      " << std::endl;
51         ost << "*****" << std::endl;
52         ost << std::endl;
53     }
54     return ost;
55 }
56
57 inline std::ostream& TestEnd(std::ostream& ost) {
58     if (ost.good()) {
59         ost << std::endl;
60         ost << "*****" << std::endl;
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67     #if COLOR_OUTPUT
68         if (ost.good()) {
69             ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;
70         }
71     #else
72

```

```

73         if (ost.good()) {
74             ost << "TEST_OK!!" << std::endl;
75         }
76 #endif // COLOR_OUTPUT
77
78         return ost;
79     }
80
81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83 #if COLOR_OUTPUT
84     if (ost.good()) {
85         ost << colorRed << "TEST_FAILED_!!" << colorWhite << std::endl;
86     }
87 #else
88     if (ost.good()) {
89         ost << "TEST_FAILED_!!" << std::endl;
90     }
91 #endif // COLOR_OUTPUT
92
93     return ost;
94 }
95
96 /**
97  * \brief function that reports if the testcase was successful.
98  *
99  * \param testcase      String that indicates the testcase
100  * \param successful true -> reports to cout test OK
101  * \param successful false -> reports test failed
102  */
103 template <typename T>
104 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
105     if (ostr.good()) {
106 #if COLOR_OUTPUT
107         if (expected == result) {
108             ostr << testcase << std::endl << colorGreen << "[Test_OK]_" << colorWhite <<
109                 "Result:_(Expected:_" << std::boolalpha << expected << "==" << "_Result:_"
110                 << result << ")" << std::noboolalpha << std::endl << std::endl;
111         }
112         else {
113             ostr << testcase << std::endl << colorRed << "[Test_FAILED]_" << colorWhite <<
114                 "Result:_(Expected:_" << std::boolalpha << expected << "!=" << "_Result:_"
115                 << result << ")" << std::noboolalpha << std::endl << std::endl;
116         }
117 #else
118         if (expected == result) {
119             ostr << testcase << std::endl << "[Test_OK]_" << "Result:_(Expected:_" << std::
120                 boolalpha << expected << "==" << "_Result:_" << result << ")" << std::
121                 noboolalpha << std::endl << std::endl;
122         }
123         else {
124             ostr << testcase << std::endl << "[Test_FAILED]_" << "Result:_(Expected:_" <<
125                 std::boolalpha << expected << "!=" << "_Result:_" << result << ")" <<
126                 std::noboolalpha << std::endl << std::endl;
127         }
128 #endif
129
130         if (ostr.fail()) {
131             std::cerr << "Error:_Write_Ostream" << std::endl;
132         }
133     }
134     else {
135         std::cerr << "Error:_Bad_Ostream" << std::endl;
136     }
137     return expected == result;
138 }
139
140 template <typename T1, typename T2>
141 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
142     if (!ost.good()) throw std::exception("Error:_bad_Ostream!");
143     ost << "(" << p.first << ", " << p.second << ")";
144     return ost;
145 }

```

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



## 6.12 SingletonBase.hpp

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

## 6.13 JavaVariable.hpp

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

## 6.14 JavaVariable.cpp

```
1  /*****
2  * \file   JavaVariable.cpp
3  * \brief  A Class for parsing java variables
4  * \author Simon
5  * \date   Dezember 2025
6  *****/
7
8  #include "JavaVariable.hpp"
9  #include <sstream>
10 #include <string>
11 #include "scanner.h"
12
13 using namespace pfc;
14 using namespace std;
15
16 /**
17 * \brief Scans an input string for the Type name of the Var.
18 *
19 * \param scan Reference to scanner object
20 * \return empty string if no valid type name is found
21 * \return name of type
22 */
23 static std::string ScanTypeName(scanner& scan)
24 {
25     string typeName = scan.get_identifier();
26     scan.next_symbol();
27     return typeName;
28 }
29
30 /**
31 * \brief Scans an input string for the Variable name of the Var.
32 *
33 * \param scan Reference to scanner object
34 * \return empty string if no valid Variable name is found
35 * \return name of Variable
36 */
37 static std::string ScanVarName(scanner& scan)
38 {
39     string varName;
40     varName = scan.get_identifier();
41     scan.next_symbol();
42
43     // The line should be empty after the var Name!
44     if (!scan.has_symbol()) return varName;
45     else return "";
46 }
47
48 std::string JavaVariable::GetSaveLine() const
49 {
50     if (m_type == nullptr) return "";
51
52     return m_type->GetName() + " " + m_name + "\n";
53 }
54
55 std::string JavaVariable::LoadTypeName(std::string const& fileLine) const
56 {
57     stringstream lineStream;
58     lineStream << fileLine;
59     scanner scan(lineStream);
60
61     return ScanTypeName(scan);
62 }
63
64 std::string JavaVariable::LoadVarName(std::string const& fileLine) const
65 {
66     stringstream lineStream;
67     lineStream << fileLine;
68     scanner scan(lineStream);
69
70     string typeName = ScanTypeName(scan);
71     string varName = ScanVarName(scan);
72     if (typeName.empty()) varName = "";
```

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

## 6.15 JavaSymbolFactory.hpp

```
1  /*****  
2  * \file   JavaSymbolFactory.hpp  
3  * \brief  A factory for creating java variables and types  
4  * \author  Simon  
5  * \date   Dezember 2025  
6  *****/  
7  #ifndef JAVA_SYMBOL_FACTORY_HPP  
8  #define JAVA_SYMBOL_FACTORY_HPP  
9  
10 #include "ISymbolFactory.hpp"  
11 #include "SingletonBase.hpp"  
12  
13 class JavaSymbolFactory :public ISymbolFactory, public SingletonBase<JavaSymbolFactory>  
14 {  
15 public:  
16  
17     friend class SingletonBase<JavaSymbolFactory>;  
18  
19     /**  
20     * \brief Creates a java variable  
21     *  
22     * \param string of variables name  
23     * \return unique pointer to variable  
24     */  
25     virtual Variable::Uptr CreateVariable(const std::string& name) override;  
26  
27     /**  
28     * \brief Creates a java type  
29     *  
30     * \param string of typename  
31     * \return unique pointer to type  
32     */  
33     virtual Type::Uptr CreateType(const std::string& name) override;  
34  
35     /**  
36     * \brief Getter for file path of type file  
37     *  
38     * \return string of filePath  
39     */  
40     virtual const std::string& GetTypeFileName() override;  
41  
42     /**  
43     * \brief Getter for file path of variable file  
44     *  
45     * \return string of filePath  
46     */  
47     virtual const std::string& GetVariableFileName() override;  
48  
49     // delete CopyCtor and Assign operator to prevent untested behaviour  
50     JavaSymbolFactory(JavaSymbolFactory& fact) = delete;  
51     void operator=(JavaSymbolFactory fact) = delete;  
52  
53 protected:  
54 private:  
55     JavaSymbolFactory() = default;  
56     const std::string m_TypeFileName = "IECTypes.sym";  
57     const std::string m_VariableFileName = "IECVars.sym";  
58 };  
59 #endif
```

## 6.16 JavaSymbolFactory.cpp

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

## 6.17 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
```

## 6.18 IECVariable.cpp

```
1  /*****
2  * \file IECVariable.cpp
3  * \brief A Class for parsing IEC variables
4  * \author Simon Vogelhuber
5  * \date Dezember 2025
6  *****/
7
8  #include "IECVariable.hpp"
9  #include <sstream>
10 #include <string>
11 #include <iostream>
12 #include "scanner.h"
13
14 using namespace pfc;
15 using namespace std;
16
17 std::string IECVariable::GetSaveLine() const
18 {
19     if (m_type == nullptr) return "";
20
21     return "VAR_" + m_type->GetName() + ":" + m_name + "\n";
22 }
23
24 /**
25 * \brief Scans an input string for the Type name of the Var.
26 *
27 * \param scan Reference to scanner object
28 * \return empty string if no valid type name is found
29 * \return name of type
30 */
31 static std::string ScanTypeName(scanner & scan) {
32     string TypeName;
33
34     if (scan.get_identifier() == "VAR") {
35         scan.next_symbol();
36         TypeName = scan.get_identifier();
37         scan.next_symbol();
38         return TypeName;
39     }
40
41     return "";
42 }
43
44 /**
45 * \brief Scans an input string for the Variable name of the Var.
46 *
47 * \param scan Reference to scanner object
48 * \return empty string if no valid Variable name is found
49 * \return name of Variable
50 */
51 static std::string ScanVarName(scanner & scan) {
52     string VarName;
53
54     if (scan.is(':')) {
55         scan.next_symbol();
56         VarName = scan.get_identifier();
57         scan.next_symbol();
58         if (!scan.is(';')) {
59             VarName = "";
60         }
61     }
62
63     return VarName;
64 }
65
66
67 std::string IECVariable::LoadTypeName(std::string const& fileLine) const
68 {
69     stringstream converter;
70     converter << fileLine;
71     scanner Scan;
72 }
```



```
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     }
```

## 6.19 IECSymbolFactory.hpp

```
1  /*****  
2  * \file IECSymbolFactory.hpp  
3  * \brief A factory for creating IEC variables and types  
4  * \author Simon  
5  * \date Dezember 2025  
6  *****/  
7  #ifndef IEC_SYMBOL_FACTORY_HPP  
8  #define IEC_SYMBOL_FACTORY_HPP  
9  
10 #include "Object.h"  
11 #include "ISymbolFactory.hpp"  
12 #include "SingletonBase.hpp"  
13  
14 class IECSymbolFactory : public ISymbolFactory , public SingletonBase<IECSymbolFactory>  
15 {  
16 public:  
17  
18     // This class is a Singleton  
19     friend class SingletonBase<IECSymbolFactory>;  
20  
21     /**  
22     * \brief Creates a IEC variable  
23     *  
24     * \param string of variables name  
25     * \return unique pointer to variable  
26     */  
27     virtual Variable::Uptr CreateVariable(const std::string& name) override;  
28  
29     /**  
30     * \brief Creates a IEC type  
31     *  
32     * \param string of typename  
33     * \return unique pointer to type  
34     */  
35     virtual Type::Uptr CreateType(const std::string& name) override;  
36  
37     /**  
38     * \brief Getter for file path of type file  
39     *  
40     * \return string of filePath  
41     */  
42     virtual const std::string& GetTypeFileName() override;  
43  
44     /**  
45     * \brief Getter for file path of variable file  
46     *  
47     * \return string of filePath  
48     */  
49     virtual const std::string& GetVariableFileName() override;  
50  
51     IECSymbolFactory(IECSymbolFactory& fact) = delete;  
52     void operator=(IECSymbolFactory fact) = delete;  
53  
54 protected:  
55 private:  
56     IECSymbolFactory() = default;  
57  
58     const std::string m_TypeFileName = "IECTypes.sym";  
59     const std::string m_VariableFileName = "IECVars.sym";  
60 };  
61  
62 #endif
```

## 6.20 IECSymbolFactory.cpp

```
1  /*****  
2  * \file   IECSymbolFactory.cpp  
3  * \brief  A factory for creating IEC variables and types  
4  * \author Simon  
5  * \date   Dezember 2025  
6  *****/  
7  
8  #include "IECSymbolFactory.hpp"  
9  #include "IECType.hpp"  
10 #include "IECVariable.hpp"  
11  
12  
13 Variable::Uptr IECSymbolFactory::CreateVariable(const std::string& name)  
14 {  
15     return std::make_unique<IECVariable>(name);  
16 }  
17  
18 Type::Uptr IECSymbolFactory::CreateType(const std::string& name)  
19 {  
20     return std::make_unique<IECType>(name);  
21 }  
22  
23 const std::string& IECSymbolFactory::GetTypeFileName()  
24 {  
25     return m_TypeFileName;  
26 }  
27  
28 const std::string& IECSymbolFactory::GetVariableFileName()  
29 {  
30     return m_VariableFileName;  
31 }
```