

Name(1): Viktoria Streibl

Abgabetermin: 26.11.2019

Name(2): Daniel Weyrer

Punkte:

Übungsgruppe: 1

korrigiert:

Geschätzter Aufwand in Ph: 6 | 3

Effektiver Aufwand in Ph: 6 | 3

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     SymbolParser();
5     ~SymbolParser();
6     void AddType(std::string const& name);
7     void AddVariable(std::string const& name, std::string const& type);
8     void SetFactory(SymbolFactory* fact);
9 protected:
10     ...
11 private:
12     ...
13 };
```

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 (Funktionalität, Klassendiagramm, Schnittstellen der beteiligten Klassen, etc.)!

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!

SDP - Exercise 04

winter semester 2019/20

Viktoria Streibl - S1810306013

Daniel Weyrer - S1820306044

November 26, 2019

Contents

1	Organizational	6
1.1	Team	6
1.2	Roles and responsibilities	6
1.2.1	Jointly	6
1.2.2	Viktoria Streibl	6
1.2.3	Daniel Weyrer	6
1.3	Effort	6
1.3.1	Viktoria Streibl	6
1.3.2	Daniel Weyrer	6
2	Requirement Definition(System Specification)	7
3	System Design	7
3.1	Classdiagram	7
4	Component Design	8
4.1	SymbolParser	8
4.2	SymbolFactory	8
4.3	IECSymbolFactory	8
4.4	JavaSymbolFactory	9
4.5	TestDriver	9
5	Test Protocol	10
5.1	Testfiles	10
5.1.1	IECTypes.sym	10
5.1.2	IECVars.sym	10
5.1.3	JavaTypes.sym	10
5.1.4	JavaVars	10
5.2	ConsoleOutput	10
6	Source Code	11
6.1	SymbolParser	11
6.1.1	SymbolParser.h	11
6.1.2	SymbolParser.cpp	11
6.2	SymbolFactory	13
6.2.1	SymbolFactory.h	13
6.2.2	SymbolFactory.cpp	15
6.3	IECSymbolFactory	17
6.3.1	IECSymbolFactory.h	17
6.3.2	SymbolFactory.cpp	17
6.4	JavaSymbolFactory	19
6.4.1	JavaSymbolFactory.h	19
6.4.2	JavaSymbolFactory.cpp	20
6.5	TestDriver	21
6.5.1	TestDriver.cpp	21

1 Organizational

1.1 Team

- Viktoria Streibl - S1810306013
- Daniel Weyrer - S1820306044

1.2 Roles and responsibilities

1.2.1 Jointly

- Planning
- Documentation
- Systemdocumentation
- Class Diagram

1.2.2 Viktoria Streibl

- Class SymbolFactory
- Class IECSymbolFactory
- Class JavaSymbolFactory
- Documentation

1.2.3 Daniel Weyrer

- Documentation
- TestDriver
- Class SymbolParser

1.3 Effort

1.3.1 Viktoria Streibl

- estimated: 6 ph
- actually: 6 ph

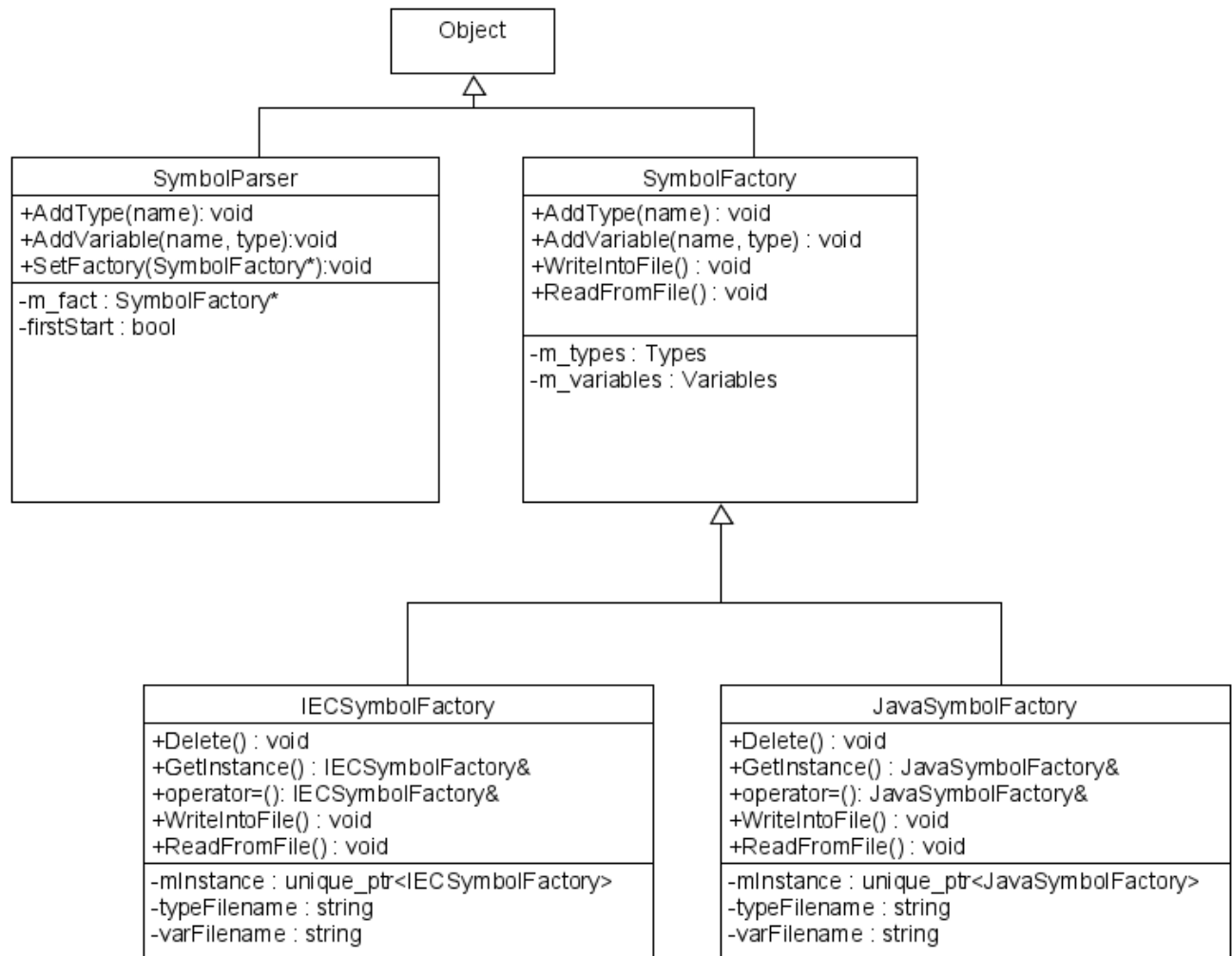
1.3.2 Daniel Weyrer

- estimated: 3 ph
- actually: 3 ph

2 Requirement Definition(System Specification)

3 System Design

3.1 Classdiagram



4 Component Design

4.1 SymbolParser

- AddType
Calls the AddType of SymbolFactory
- AddVariable
Calls the AddVariable of SymbolFactory
- ReadFile
Reads the files and adds all existing types and variables
- SetFactory
Gets the current language and sets it as current factory. Calls ReadFile at the first call

4.2 SymbolFactory

- AddType
Gets the type and adds to the others if it's not exists.
- AddVariable
Gets the variable name and adds to the others. It checks if it is the first variable-name of this type.
- WriteIntoFile
Virtual Method to write all types and variables back into the file
- ReadFromFile
Virtual Method to read all types and variables of a file

4.3 IECSymbolFactory

Is a Singleton Class.

- AddType
Gets the type and adds to the others if it's not exists.
- AddVariable
Gets the variable name and adds to the others. It checks if it is the first variable-name of this type.
- WriteIntoFile
Writes all types and variables into the iec files.
- ReadFromFile
Reads from the iec-files and save all types and variables into the programm.

4.4 JavaSymbolFactory

Is a Singleton Class.

- AddType
Gets the type and adds to the others if it's not exists.
- AddVariable
Gets the variable name and adds to the others. It checks if it is the first variable-name of this type.
- WriteIntoFile
Writes all types and variables into the java files.
- ReadFromFile
Reads from the java-files and save all types and variables into the programm.

4.5 TestDriver

The Testdriver test alle functions of the clients. It tests the interface for the Epos-Company as well as the NortelNetwork-Company. It encrypt and decrypt several files. It contains also some functions:

- Adds Types and Variables of Java and IEC
- Tests what happens to add a Variable but no existing Type
- Tests to read the Files and a new Variable
- Tests to add existing Type

5 Test Protocol

5.1 Testfiles

5.1.1 IECTypes.sym

```
1 TYPE SpeedController
2 TYPE Hugo
3 TYPE Nero
```

5.1.2 IECVars.sym

```
1 VAR mHu : Hugo;
```

5.1.3 JavaTypes.sym

```
1 class Button
2 class Hugo
3 class Window
```

5.1.4 JavaVars

```
1 Button mButton
2 Window mWin
3 Button b
```

5.2 ConsoleOutput

```
1 Test 1...
2 Type Button is already defined.
3 Type Hugo is already defined.
4 Type Window is already defined.
5 Test 2...
6 Type SpeedController is not defined.
7 Test 3...
8 Test 4...
9 Type Hugo is already defined.
```

6 Source Code

6.1 SymbolParser

6.1.1 SymbolParser.h

```
1  /* -----
2  | Workfile : SymbolParser.h
3  | Description : [ HEADER ]
4  | Name : Daniel Weyrer      PKZ : S1820306044
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #ifndef SYMBOLPARSER_H
10 #define SYMBOLPARSER_H
11
12 #include <string>
13 #include <iostream>
14 #include <fstream>
15
16 #include "Object.h"
17 #include "SymbolFactory.h"
18 #include "JavaSymbolFactory.h"
19 #include "IECSymbolFactory.h"
20
21 class SymbolParser : public Object
22 {
23 public:
24     //Konstruktor
25     SymbolParser() = default;
26     //Dekonstruktor
27     ~SymbolParser() = default;
28
29     //add a type to the list
30     void AddType(std::string const& name);
31
32     //add a new variable to the list
33     void AddVariable(std::string const& name, std::string const& type);
34
35     void ReadFile();
36
37     //set the current programming language
38     void SetFactory(SymbolFactory* fact);
39 private:
40     SymbolFactory* m_fact = nullptr;
41     bool firstStart = false;
42 };
43
44 #endif // !SYMBOLPARSER_H
```

6.1.2 SymbolParser.cpp

```
1  /* -----
2  | Workfile : SymbolParser.cpp
3  | Description : [ SOURCE ]
4  | Name : Daniel Weyrer      PKZ : S1820306044
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #include "SymbolParser.h"
10
11 //calls the addType function of the language
12 void SymbolParser::AddType(std::string const& name) {
13     m_fact->AddType(name);
14 }
15
16 //calls the addVariable function of the language
17 void SymbolParser::AddVariable(std::string const& name, std::string const& type) {
```

```
18     m_fact->AddVariable(name, type);
19 }
20
21 //set current factory
22 void SymbolParser::SetFactory(SymbolFactory* fact) {
23     if (m_fact != nullptr) {
24         m_fact->WriteIntoFile();
25     }
26     m_fact = fact;
27     if (!firstStart) {
28         m_fact->ReadFromFile();
29     }
30
31     if (!firstStart) {
32         firstStart = true;
33     }
34 }
35
36 void SymbolParser::ReadFile() {
37     JavaSymbolFactory& factJava = JavaSymbolFactory::GetInstance();
38     factJava.ReadFromFile();
39
40     IECSymbolFactory& factIEC = IECSymbolFactory::GetInstance();
41     factIEC.ReadFromFile();
42 }
```

6.2 SymbolFactory

6.2.1 SymbolFactory.h

```
1  /* -----
2  | Workfile : SymbolParser.h
3  | Description : [ HEADER ]
4  | Name : Viktoria Streibl      PKZ : S1810306013
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #ifndef SYMBOLFACTORY_H
10 #define SYMBOLFACTORY_H
11
12 #include <stdio.h>
13 #include <iostream>
14 #include <exception>
15 #include <memory>
16 #include <string>
17 #include <vector>
18
19 #include "Object.h"
20
21 typedef std::vector<std::string> Types;
22 typedef std::pair<std::string, std::string> Variable;
23 typedef std::vector<Variable> Variables;
24
25 //template singleton base class
26 class SymbolFactory : public Object {
27 public:
28     SymbolFactory() = default;
29     ~SymbolFactory() = default;
30     //create new type
31     void AddType(std::string const& name);
32
33     //create new variable
34     void AddVariable(std::string const& name, std::string const& type);
35
36     //write all types and variables into the file
37     virtual void WriteIntoFile() = 0;
38
39     //read from the file
40     virtual void ReadFromFile() = 0;
41
42 protected:
43     //vector types
44     Types m_types;
45     //vector variables(pair)
46     Variables m_variables;
47 };
48
49 //
50 //Exceptions
51 //
52
53 struct TypeAlreadyDefinedException : public std::exception
54 {
55     const std::string what(std::string typeName) const throw ()
56     {
57         return "Type " + typeName + " is already defined.";
58     }
59 };
60
61 struct VariableAlreadyDefinedException : public std::exception
62 {
63     const std::string what(std::string varName) const throw ()
64     {
65         return "Variable " + varName + " is already defined.";
66     }
67 };
68 struct TypeNotDefinedException : public std::exception
```

```
69 {
70     const std::string what(std::string typeName) const throw ()
71     {
72         return "Type " + typeName + " is not defined.";
73     }
74 };
75
76
77 #endif // !SYMBOLFACTORY_H
```

6.2.2 SymbolFactory.cpp

```
1  /* -----  
2  | Workfile : SymbolParser.h  
3  | Description : [ HEADER ]  
4  | Name : Viktoria Streibl      PKZ : S1810306013  
5  | Date : 24.11.2019  
6  | Remarks : -  
7  | Revision : 0  
8  | ----- */  
9  #ifndef SYMBOLFACTORY_H  
10 #define SYMBOLFACTORY_H  
11  
12 #include <stdio.h>  
13 #include <iostream>  
14 #include <exception>  
15 #include <memory>  
16 #include <string>  
17 #include <vector>  
18  
19 #include "Object.h"  
20  
21 typedef std::vector<std::string> Types;  
22 typedef std::pair<std::string, std::string> Variable;  
23 typedef std::vector<Variable> Variables;  
24  
25 //template singleton base class  
26 class SymbolFactory : public Object {  
27  
28 public:  
29     SymbolFactory() = default;  
30     ~SymbolFactory() = default;  
31     //create new type  
32     void AddType(std::string const& name);  
33  
34     //create new variable  
35     void AddVariable(std::string const& name, std::string const& type);  
36  
37     //write all types and variables into the file  
38     virtual void WriteIntoFile() = 0;  
39  
40     //read from the file  
41     virtual void ReadFromFile() = 0;  
42  
43 protected:  
44     //vector types  
45     Types m_types;  
46     //vector variables(pair)  
47     Variables m_variables;  
48 };  
49  
50 //  
51 //Exceptions  
52 //  
53 struct TypeAlreadyDefinedException : public std::exception  
54 {  
55     const std::string what(std::string typeName) const throw ()  
56     {  
57         return "Type " + typeName + " is already defined.";  
58     }  
59 };  
60  
61 struct VariableAlreadyDefinedException : public std::exception  
62 {  
63     const std::string what(std::string varName) const throw ()  
64     {  
65         return "Variable " + varName + " is already defined.";  
66     }  
67 };  
68 struct TypeNotDefinedException : public std::exception  
69 {  
70     const std::string what(std::string typeName) const throw ()
```

```
71 {  
72     return "Type " + typeName + " is not defined.";   
73 }  
74 };  
75  
76  
77 #endif // !SYMBOLFACTORY_H
```


6.3 IECSymbolFactory

6.3.1 IECSymbolFactory.h

```

1  /* -----
2  | Workfile : IECSymbolFactory.cpp
3  | Description : [ HEADER ]
4  | Name : Viktoria Streibl      PKZ : S1810306013
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #ifndef IECSYMBOLFACTORY_H
10 #define IECSYMBOLFACTORY_H
11
12 #include <iostream>
13 #include <fstream>
14
15 #include "SymbolFactory.h"
16
17 class IECSymbolFactory : public SymbolFactory
18 {
19 public:
20     //create singleton instance
21     static IECSymbolFactory& GetInstance() {
22         if (mInstance == nullptr) mInstance = std::unique_ptr<IECSymbolFactory>(new IECSymbolFactory);
23         return *mInstance;
24     }
25     //free singleton before end of program
26     static void Delete() {
27         mInstance.reset();
28         mInstance.get()->WriteIntoFile();
29     }
30
31     //write all types and variables into the file
32     virtual void WriteIntoFile() override;
33
34     //read all types and variables from the files
35     virtual void ReadFromFile() override;
36
37 private:
38     //hide default ctor
39     IECSymbolFactory() = default;
40     IECSymbolFactory(IECSymbolFactory const&) = delete;
41     IECSymbolFactory& operator= (IECSymbolFactory const&) = delete;
42     static std::unique_ptr<IECSymbolFactory> mInstance;
43
44     //IECTypes.sym und IECVars.sym
45     std::string typeFilename = "IECTypes.sym";
46     std::string varFilename = "IECVars.sym";
47 };
48
49 #endif // !IECSYMBOLFACTORY_H

```

6.3.2 SymbolFactory.cpp

```

1  /* -----
2  | Workfile : IECSymbolFactory.cpp
3  | Description : [ HEADER ]
4  | Name : Viktoria Streibl      PKZ : S1810306013
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #ifndef IECSYMBOLFACTORY_H
10 #define IECSYMBOLFACTORY_H
11
12 #include <iostream>
13 #include <fstream>
14
15 #include "SymbolFactory.h"
16

```

```
17 class IECSymbolFactory : public SymbolFactory
18 {
19 public:
20     //create singleton instance
21     static IECSymbolFactory& GetInstance() {
22         if (mInstance == nullptr) mInstance = std::unique_ptr<IECSymbolFactory>(new IECSymbolFactory);
23         return *mInstance;
24     }
25     //free singleton before end of program
26     static void Delete() {
27         mInstance.reset();
28         mInstance.get()->WriteIntoFile();
29     }
30
31     //write all types and variables into the file
32     virtual void WriteIntoFile() override;
33
34     //read all types and variables from the files
35     virtual void ReadFromFile() override;
36
37 private:
38     //hide default ctor
39     IECSymbolFactory() = default;
40     IECSymbolFactory(const IECSymbolFactory&) = delete;
41     IECSymbolFactory& operator= (const IECSymbolFactory&) = delete;
42     static std::unique_ptr<IECSymbolFactory> mInstance;
43
44     //IECTypes.sym und IECVars.sym
45     std::string typeFilename = "IECTypes.sym";
46     std::string varFilename = "IECVars.sym";
47 };
48
49 #endif // !IECSYMBOLFACTORY_H
```

6.4 JavaSymbolFactory

6.4.1 JavaSymbolFactory.h

```
1  /* -----
2  | Workfile : JavaSymbolFactory.h
3  | Description : [ HEADER ]
4  | Name : Viktoria Streibl      PKZ : S1810306013
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #ifndef JAVASYMBOLFACTORY_H
10 #define JAVASYMBOLFACTORY_H
11
12 #include "SymbolFactory.h"
13
14 #include <iostream>
15 #include <fstream>
16
17 class JavaSymbolFactory : public SymbolFactory
18 {
19 public:
20     static JavaSymbolFactory& GetInstance() {
21         if (mInstance == nullptr) mInstance = std::unique_ptr<JavaSymbolFactory>(new JavaSymbolFactory);
22         return *mInstance;
23     }
24
25     //free singleton before end of program
26     static void Delete() { mInstance.reset(); }
27
28     //write all types and variables into the files
29     virtual void WriteIntoFile() override;
30
31     //read all types and variables from the files
32     virtual void ReadFromFile() override;
33
34 private:
35     //hide default ctor
36     JavaSymbolFactory() = default;
37     JavaSymbolFactory(JavaSymbolFactory const&) = delete;
38     JavaSymbolFactory& operator= (JavaSymbolFactory const&) = delete;
39     static std::unique_ptr<JavaSymbolFactory> mInstance;
40
41     //JavaTypes.sym und JavaVars.sym
42     std::string typeFilename = "JavaTypes.sym";
43     std::string varFilename = "JavaVars.sym";
44 };
45
46 #endif // !JAVASYMBOLFACTORY_H
```

6.4.2 JavaSymbolFactory.cpp

```
1  /* -----  
2  | Workfile : JavaSymbolFactory.h  
3  | Description : [ HEADER ]  
4  | Name : Viktoria Streibl      PKZ : S1810306013  
5  | Date : 24.11.2019  
6  | Remarks : -  
7  | Revision : 0  
8  | ----- */  
9  #ifndef JAVASYMBOLFACTORY_H  
10 #define JAVASYMBOLFACTORY_H  
11  
12 #include "SymbolFactory.h"  
13  
14 #include <iostream>  
15 #include <fstream>  
16  
17 class JavaSymbolFactory : public SymbolFactory  
18 {  
19 public:  
20     static JavaSymbolFactory& GetInstance() {  
21         if (mInstance == nullptr) mInstance = std::unique_ptr<JavaSymbolFactory>(new JavaSymbolFactory);  
22         return *mInstance;  
23     }  
24  
25     //free singleton before end of program  
26     static void Delete() { mInstance.reset(); }  
27  
28     //write all types and variables into the files  
29     virtual void WriteIntoFile() override;  
30  
31     //read all types and variables from the files  
32     virtual void ReadFromFile() override;  
33  
34 private:  
35     //hide default ctor  
36     JavaSymbolFactory() = default;  
37     JavaSymbolFactory(JavaSymbolFactory const&) = delete;  
38     JavaSymbolFactory& operator= (JavaSymbolFactory const&) = delete;  
39     static std::unique_ptr<JavaSymbolFactory> mInstance;  
40  
41     //JavaTypes.sym und JavaVars.sym  
42     std::string typeFilename = "JavaTypes.sym";  
43     std::string varFilename = "JavaVars.sym";  
44 };  
45  
46 #endif // !JAVASYMBOLFACTORY_H
```

6.5 TestDriver

6.5.1 TestDriver.cpp

```
1  /* -----
2  | Workfile : Testdriver.cpp
3  | Description : [ SOURCE ] Class for testing all functions
4  | Name : Daniel Weyrer      PKZ : S1820306044
5  | Date : 24.11.2019
6  | Remarks : -
7  | Revision : 0
8  | ----- */
9  #include "SymbolParser.h"
10 #include "JavaSymbolFactory.h"
11 #include "IECSymbolFactory.h"
12
13 int main() {
14     SymbolParser parser;
15
16     //
17     // Test Case 1
18     //
19     std::cout << "Test 1..." << std::endl;
20     JavaSymbolFactory& fact = JavaSymbolFactory::GetInstance();
21     parser.SetFactory(&fact);
22     parser.AddType("Button");
23     parser.AddType("Hugo");
24     parser.AddType("Window");
25     parser.AddVariable("mButton", "Button");
26     parser.AddVariable("mWin", "Window");
27
28     //
29     // Test Case 2
30     //
31     std::cout << "Test 2..." << std::endl;
32     parser.SetFactory(&IECSymbolFactory::GetInstance());
33     parser.AddType("SpeedController");
34     parser.AddType("Hugo");
35     parser.AddType("Nero");
36     parser.AddVariable("mCont", "SpeedController");
37     parser.AddVariable("mHu", "Hugo");
38
39     //
40     // Test Case 3
41     //
42     std::cout << "Test 3..." << std::endl;
43     parser.SetFactory(&JavaSymbolFactory::GetInstance());
44     parser.AddVariable("b", "Button");
45
46     //
47     // Test Case 4
48     //
49     std::cout << "Test 4..." << std::endl;
50     parser.SetFactory(&IECSymbolFactory::GetInstance());
51     parser.AddType("Hugo");
52     parser.AddVariable("mCont", "Hugo");
53
54     return 0;
55 }
```