FH-OÖ Hagenberg/HSD
SDP3, WS 2019
Übung 4

Name(1): Viktoria Streibl	Abgabetermin: 26.11.2019
Name(2): Daniel Weyrer	Punkte:
Übungsgruppe: 1	korrigiert:
Geschätzter Aufwand in Ph: 8 | 3	Effektiver Aufwand in Ph: 8 | 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:

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

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

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

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

JavaTypes.sym:

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

JavaVars.sym:

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

IECTypes.sym:

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

IECVars.sym:

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

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

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

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

Eine mögliche Anwendung im Hauptprogramm könnte so aussehen:

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


int main()
```

```
{
   SymbolParser parser;

   parser.SetFactory(JavaSymbolFactory::GetInstance());
   parser.AddType("Button");
   parser.AddType("Hugo");
   parser.AddType("Window");
   parser.AddVariable("mButton","Button");
   parser.AddVariable("mWin","Window");

   parser.SetFactory(IECSymbolFactory::GetInstance());
   parser.AddType("SpeedController");
   parser.AddType("Hugo");
   parser.AddType("Nero");
   parser.AddVariable("mCont","SpeedController");
   parser.AddVariable("mHu","Hugo");

   parser.SetFactory(JavaSymbolFactory::GetInstance());
   parser.AddVariable("b", "Button");

   parser.SetFactory(IECSymbolFactory::GetInstance());
   parser.AddType("Hugo");
   parser.AddVariable("mCont","Hugo");

   return 0;
}
```

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

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

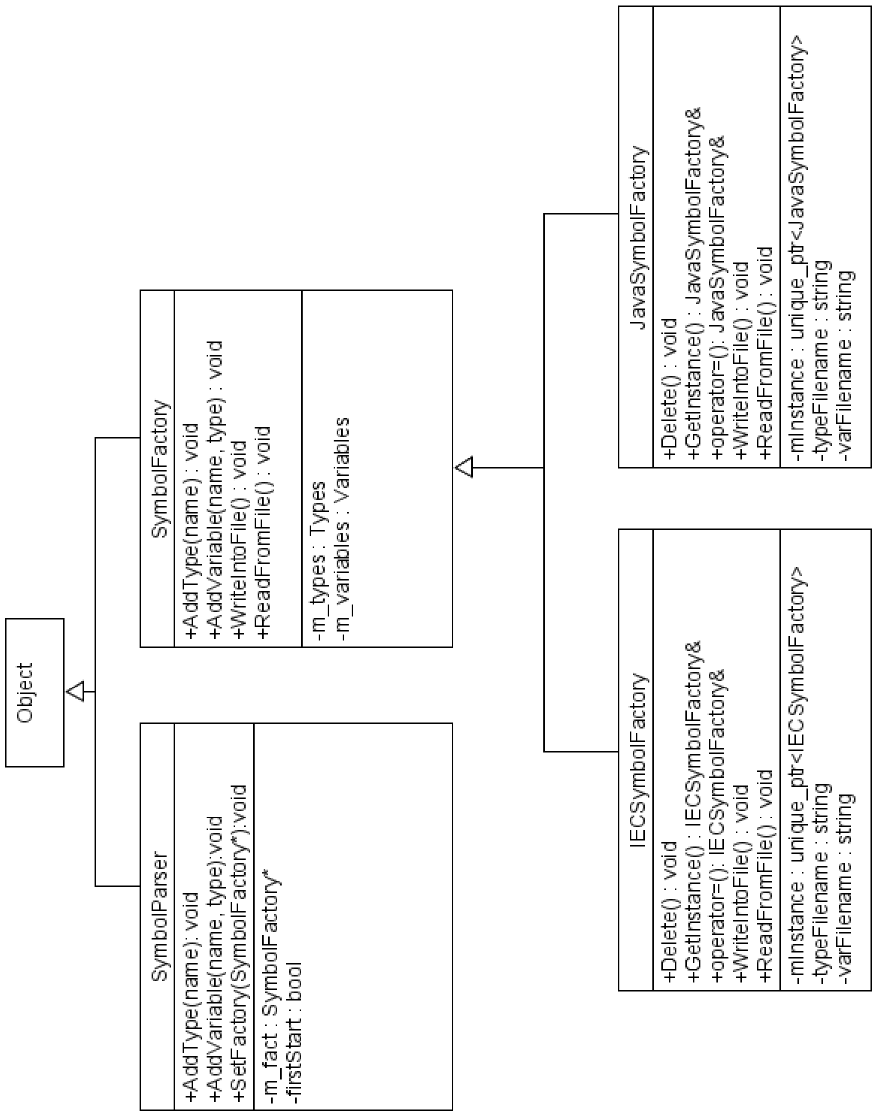
1.3.2 Daniel Weyrer

- estimated: 3 ph
- actually: 3 ph

2 Requirenment 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 ad 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

5.2 IECTypes.sym

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

5.3 IECVars.sym

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

5.4 JavaTypes.sym

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

5.5 JavaVars

```
Button mButton;
Window mWin;
Button b;
```

6 Source Code

6.1 SymbolParser

6.1.1 SymbolParser.h

```cpp
/* -----------------------------------------------------------------------
| Workfile : SymbolParser.h
| Description : [ HEADER ]
| Name : Daniel Weyrer      PKZ : S1820306044
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ---------------------------------------------------------------------- */
#ifndef SYMBOLPARSER_H
#define SYMBOLPARSER_H

#include <string>
#include <iostream>
#include <fstream>

#include "Object.h"
#include "SymbolFactory.h"
#include "JavaSymbolFactory.h"
#include "IECSymbolFactory.h"

class SymbolParser : public Object
{
public:
  //Konstruktor
  SymbolParser() = default;
  //Dekonstruktor
  ~SymbolParser() = default;

  //add a type to the list
  void AddType(std::string const& name);

  //add a new variable to the list
  void AddVariable(std::string const& name, std::string const& type);

  void ReadFile();

  //set the current programming language
  void SetFactory(SymbolFactory* fact);
private:
  SymbolFactory* m_fact = nullptr;
  bool firstStart = false;
};

#endif // !SYMBOLPARSER_H
```

6.1.2 SymbolParser.cpp

```cpp
/* -----------------------------------------------------------------------
| Workfile : SymbolParser.cpp
| Description : [ SOURCE ]
| Name : Daniel Weyrer      PKZ : S1820306044
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ---------------------------------------------------------------------- */
#include "SymbolParser.h"

//calls the addType function of the language
void SymbolParser::AddType(std::string const& name) {
  m_fact->AddType(name);
}

//calls the addVariable function of the language
void SymbolParser::AddVariable(std::string const& name, std::string const& type) {
  m_fact->AddVariable(name, type);
}

//set current factory
void SymbolParser::SetFactory(SymbolFactory* fact) {
  if (m_fact != nullptr) {
    m_fact->WriteIntoFile();
  }
  m_fact = fact;
  if(!firstStart) {
    m_fact->ReadFromFile();
  }

  if (!firstStart) {
    firstStart = true;
  }
}

void SymbolParser::ReadFile() {
  JavaSymbolFactory& factJava = JavaSymbolFactory::GetInstance();
  factJava.ReadFromFile();

  IECSymbolFactory& factIEC = IECSymbolFactory::GetInstance();
  factIEC.ReadFromFile();
}
```

6.2 SymbolFactory

6.2.1 SymbolFactory.h

```
/* ----------------------------------------------------------------------
| Workfile : SymbolParser.h
| Description : [ HEADER ]
| Name : Viktoria Streibl     PKZ : S1810306013
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ---------------------------------------------------------------------- */
#ifndef SYMBOLFACTORY_H
#define SYMBOLFACTORY_H

#include <stdio.h>
#include <iostream>
#include <exception>
#include <memory>
#include <string>
#include <vector>

#include "Object.h"

typedef std::vector<std::string> Types;
typedef std::pair<std::string, std::string> Variable;
typedef std::vector<Variable> Variables;

//template singleton base class
class SymbolFactory : public Object {

public:
  SymbolFactory() = default;
  ~SymbolFactory() = default;
  //create new type
  void AddType(std::string const& name);

  //create new variable
  void AddVariable(std::string const& name, std::string const& type);

  //write all types and variables into the file
  virtual void WriteIntoFile() = 0;

  //read from the file
  virtual void ReadFromFile() = 0;

protected:
  //vector types
  Types m_types;
  //vector variables(pair)
  Variables m_variables;
};

//
//Exceptions
//
struct TypeAlreadyDefinedException : public std::exception
{
  const std::string what(std::string typeName) const throw ()
  {
    return "Type " + typeName + " is already defined.";
  }
};

struct VariableAlreadyDefinedException : public std::exception
{
  const std::string what(std::string varName) const throw ()
  {
    return "Variable " + varName + " is already defined.";
  }
};
struct TypeNotDefinedException : public std::exception
```

```
{
  const std::string what(std::string typeName) const throw ()
  {
    return "Type " + typeName + " is not defined.";
  }
};


#endif // !SYMBOLFACTORY_H
```

6.2.2 SymbolFactory.cpp

```
/* _______________________________________________________________________
| Workfile : SymbolParser.h
| Description : [ HEADER ]
| Name : Viktoria Streibl     PKZ : S1810306013
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| _______________________________________________________________________ */
#ifndef SYMBOLFACTORY_H
#define SYMBOLFACTORY_H

#include <stdio.h>
#include <iostream>
#include <exception>
#include <memory>
#include <string>
#include <vector>

#include "Object.h"

typedef std::vector<std::string> Types;
typedef std::pair<std::string, std::string> Variable;
typedef std::vector<Variable> Variables;

//template singleton base class
class SymbolFactory : public Object {

public:
  SymbolFactory() = default;
  ~SymbolFactory() = default;
  //create new type
  void AddType(std::string const& name);

  //create new variable
  void AddVariable(std::string const& name, std::string const& type);

  //write all types and variables into the file
  virtual void WriteIntoFile() = 0;

  //read from the file
  virtual void ReadFromFile() = 0;

protected:
  //vector types
  Types m_types;
  //vector variables(pair)
  Variables m_variables;
};

//
//Exceptions
//
struct TypeAlreadyDefinedException : public std::exception
{
  const std::string what(std::string typeName) const throw ()
  {
    return "Type " + typeName + " is already defined.";
  }
};

```

```
struct VariableAlreadyDefinedException : public std::exception
{
  const std::string what(std::string varName) const throw ()
  {
    return "Variable " + varName + " is already defined.";
  }
};
struct TypeNotDefinedException : public std::exception
{
  const std::string what(std::string typeName) const throw ()
  {
    return "Type " + typeName + " is not defined.";
  }
};


#endif // !SYMBOLFACTORY_H
```

6.3 IECSymbolFactory

6.3.1 IECSymbolFactory.h

```
/* -----------------------------------------------------------------------
| Workfile : IECSymbolFactory.cpp
| Description : [ HEADER ]
| Name : Viktoria Streibl     PKZ : S1810306013
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ----------------------------------------------------------------------- */
#ifndef IECSYMBOLFACTORY_H
#define IECSYMBOLFACTORY_H

#include <iostream>
#include <fstream>

#include "SymbolFactory.h"

class IECSymbolFactory : public SymbolFactory
{
public:
  //create singleton instance
  static IECSymbolFactory& GetInstance() {
    if (mInstance == nullptr) mInstance = std::unique_ptr<IECSymbolFactory>(new IECSymbolFactory);
    return *mInstance;
  }
  //free singleton before end of program
  static void Delete() {
    mInstance.reset();
    mInstance.get()->WriteIntoFile();
  }

  //write all types and variables into the file
  virtual void WriteIntoFile() override;

  //read all types and variables from the files
  virtual void ReadFromFile() override;

private:
  //hide default ctor
  IECSymbolFactory() = default;
  IECSymbolFactory(IECSymbolFactory const&) = delete;
  IECSymbolFactory& operator= (IECSymbolFactory const&) = delete;
  static std::unique_ptr<IECSymbolFactory> mInstance;

  //IECTypes.sym und IECVars.sym
  std::string typeFilename = "IECTypes.sym";
  std::string varFilename = "IECVars.sym";
};

#endif // !IECSYMBOLFACTORY_H
```

6.3.2 SymbolFactory.cpp

```
/* -----------------------------------------------------------------------
| Workfile : IECSymbolFactory.cpp
| Description : [ HEADER ]
| Name : Viktoria Streibl     PKZ : S1810306013
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ----------------------------------------------------------------------- */
#ifndef IECSYMBOLFACTORY_H
#define IECSYMBOLFACTORY_H

#include <iostream>
#include <fstream>

#include "SymbolFactory.h"

```

```cpp
class IECSymbolFactory : public SymbolFactory
{
public:
  //create singleton instance
  static IECSymbolFactory& GetInstance() {
    if (mInstance == nullptr) mInstance = std::unique_ptr<IECSymbolFactory>(new IECSymbolFactory);
    return *mInstance;
  }
  //free singleton before end of program
  static void Delete() {
    mInstance.reset();
    mInstance.get()->WriteIntoFile();
  }

  //write all types and variables into the file
  virtual void WriteIntoFile() override;

  //read all types and variables from the files
  virtual void ReadFromFile() override;

private:
  //hide default ctor
  IECSymbolFactory() = default;
  IECSymbolFactory(IECSymbolFactory const&) = delete;
  IECSymbolFactory& operator= (IECSymbolFactory const&) = delete;
  static std::unique_ptr<IECSymbolFactory> mInstance;

  //IECTypes.sym und IECVars.sym
  std::string typeFilename = "IECTypes.sym";
  std::string varFilename = "IECVars.sym";
};

#endif // !IECSYMBOLFACTORY_H
```

6.4 JavaSymbolFactory

6.4.1 JavaSymbolFactory.h

```
/* ----------------------------------------------------------------------
| Workfile : JavaSymbolFactory.h
| Description : [ HEADER ]
| Name : Viktoria Streibl     PKZ : S1810306013
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ---------------------------------------------------------------------- */
#ifndef JAVASYMBOLFACTORY_H
#define JAVASYMBOLFACTORY_H

#include "SymbolFactory.h"

#include <iostream>
#include <fstream>

class JavaSymbolFactory : public SymbolFactory
{
public:
  static JavaSymbolFactory& GetInstance() {
    if (mInstance == nullptr) mInstance = std::unique_ptr<JavaSymbolFactory>(new JavaSymbolFactory);
    return *mInstance;
  }

  //free singleton before end of program
  static void Delete() { mInstance.reset(); }

  //write all types and variables into the files
  virtual void WriteIntoFile() override;

  //read all types and variables from the files
  virtual void ReadFromFile() override;

private:
  //hide default ctor
  JavaSymbolFactory() = default;
  JavaSymbolFactory(JavaSymbolFactory const&) = delete;
  JavaSymbolFactory& operator= (JavaSymbolFactory const&) = delete;
  static std::unique_ptr<JavaSymbolFactory> mInstance;

  //JavaTypes.sym und JavaVars.sym
  std::string typeFilename = "JavaTypes.sym";
  std::string varFilename = "JavaVars.sym";
};

#endif // !JAVASYMBOLFACTORY_H
```

6.4.2 JavaSymbolFactory.cpp

```
/* ----------------------------------------------------------------------
| Workfile : JavaSymbolFactory.h
| Description : [ HEADER ]
| Name : Viktoria Streibl     PKZ : S1810306013
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ---------------------------------------------------------------------- */
#ifndef JAVASYMBOLFACTORY_H
#define JAVASYMBOLFACTORY_H

#include "SymbolFactory.h"

#include <iostream>
#include <fstream>

class JavaSymbolFactory : public SymbolFactory
{
public:
```

```cpp
  static JavaSymbolFactory& GetInstance() {
    if (mInstance == nullptr) mInstance = std::unique_ptr<JavaSymbolFactory>(new JavaSymbolFactory);
    return *mInstance;
  }

  //free singleton before end of program
  static void Delete() { mInstance.reset(); }

  //write all types and variables into the files
  virtual void WriteIntoFile() override;

  //read all types and variables from the files
  virtual void ReadFromFile() override;

private:
  //hide default ctor
  JavaSymbolFactory() = default;
  JavaSymbolFactory(JavaSymbolFactory const&) = delete;
  JavaSymbolFactory& operator= (JavaSymbolFactory const&) = delete;
  static std::unique_ptr<JavaSymbolFactory> mInstance;

  //JavaTypes.sym und JavaVars.sym
  std::string typeFilename = "JavaTypes.sym";
  std::string varFilename = "JavaVars.sym";
};

#endif // !JAVASYMBOLFACTORY_H
```

6.5 TestDriver

6.5.1 TestDriver.cpp

```cpp
/* ----------------------------------------------------------------------
| Workfile : Testdriver.cpp
| Description : [ SOURCE ] Class for testing all functions
| Name : Daniel Weyrer      PKZ : S1820306044
| Date : 24.11.2019
| Remarks : -
| Revision : 0
| ---------------------------------------------------------------------- */
#include "SymbolParser.h"
#include "JavaSymbolFactory.h"
#include "IECSymbolFactory.h"

int main() {
  SymbolParser parser;

  //
  // Test Case 1
  //
  std::cout << "Test 1..." << std::endl;
  JavaSymbolFactory& fact = JavaSymbolFactory::GetInstance();
  parser.SetFactory(&fact);
  parser.AddType("Button");
  parser.AddType("Hugo");
  parser.AddType("Window");
  parser.AddVariable("mButton", "Button");
  parser.AddVariable("mWin", "Window");

  //
  // Test Case 2
  //
  std::cout << "Test 2..." << std::endl;
  parser.SetFactory(&IECSymbolFactory::GetInstance());
  parser.AddType("SpeedController");
  parser.AddType("Hugo");
  parser.AddType("Nero");
  parser.AddVariable("mCont", "SpeedControlleer");
  parser.AddVariable("mHu", "Hugo");

  //
  // Test Case 3
  //
  std::cout << "Test 3..." << std::endl;
  parser.SetFactory(&JavaSymbolFactory::GetInstance());
  parser.AddVariable("b", "Button");

  //
  // Test Case 4
  //
  std::cout << "Test 4..." << std::endl;
  parser.SetFactory(&IECSymbolFactory::GetInstance());
  parser.AddType("Hugo");
  parser.AddVariable("mCont", "Hugo");

  return 0;
}
```