FH-OÖ Hagenberg/HSD
SDP3, WS 2023
Übung 4

Name(1): Paul Engelhardt — Abgabetermin: 22.11.2023

Name(2): Harald Kiss — Punkte:

Übungsgruppe: 2 — korrigiert:

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

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

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

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

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

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

JavaTypes.sym:

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

JavaVars.sym:

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

IECTypes.sym:

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

IECVars.sym:

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

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

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

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

Eine mögliche Anwendung im Hauptprogramm könnte so aussehen:

```
1 #include "SymbolParser.h"
2 #include "JavaSymbolFactory.h"
```

```
3  #include "IECSymbolFactory.h"
4
5
6  int main()
7  {
8     SymbolParser parser;
9
10    parser.SetFactory(JavaSymbolFactory::GetInstance());
11    parser.AddType("Button");
12    parser.AddType("Hugo");
13    parser.AddType("Window");
14    parser.AddVariable("mButton","Button");
15    parser.AddVariable("mWin","Window");
16
17    parser.SetFactory(IECSymbolFactory::GetInstance());
18    parser.AddType("SpeedController");
19    parser.AddType("Hugo");
20    parser.AddType("Nero");
21    parser.AddVariable("mCont","SpeedController");
22    parser.AddVariable("mHu","Hugo");
23
24    parser.SetFactory(JavaSymbolFactory::GetInstance());
25    parser.AddVariable("b", "Button");
26
27    parser.SetFactory(IECSymbolFactory::GetInstance());
28    parser.AddType("Hugo");
29    parser.AddVariable("mCont","Hugo");
30
31    return 0;
32 }
```

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

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

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

System Dokumentation Symbolparser:

Organisatorisches

Teammitglieder:

Paul Engelhardt

Harald Kiss

Arbeitsteilung

Harald Kiss:

UML-Diagramm

Teil Implementierung

Systemdokumentation

Paul Engelhardt:

Code (Architektur)

Teil Implementation

Geschätzter Arbeitsaufwand:

12 Stunden Kiss:

10 Stunden Engelhart

3 Stunden für die Planung

16 Stunden für die Entwicklung

3 Stunden für die Dokumentation

Anforderungen

Ziel

Entwicklung einer Symbolparser-Anwendung, die es ermöglicht, verschiedene Arten von Programmiersprachen von einer Datei einzulesen und wieder zurück auszuschreiben und dies mithilfe eines Factorys. Durch die Factorys kann man durch eine einheitliche Schnittstelle die Programmiersprachen steuern und mit den Symbolparser diese dementsprechend verwalten. Dadurch, dass man eine Factory verwendet, muss bei einer neuer Programmiersprache nicht mehr viel neu implementieren, da die Factory als Schablone arbeitet.

Systementwurf:

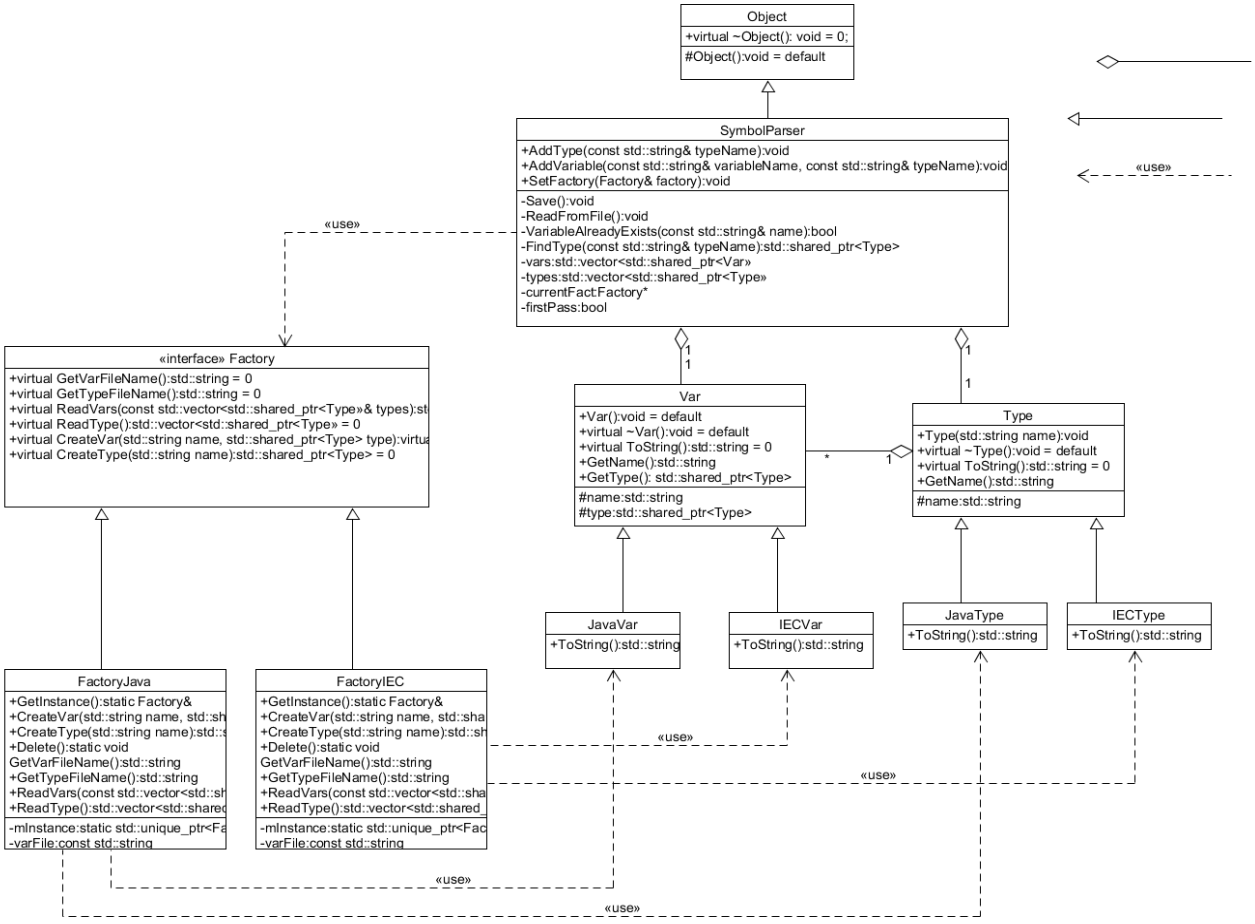
Allgemeines

In diesem Programm wurden Factorys verwendet, da wie oben schon erwähnt dies wie eine

einheitliche Schnittstelle wirkt und dadurch, dass man eine Factory verwendet, muss bei einer neuer Programmiersprache nicht mehr viel neu implementieren ,da die Factory als Schablone arbeitet um man deshalb einfach eine neue Unterfactory erstellen muss.

Klassendiagramm

Das Klassendiagramm zeigt die Struktur des Systems und die Beziehungen zwischen den verschiedenen Komponenten.



SymbolParser-Klasse:

Der SymbolParser verwaltet die Programmiersprachen indem man diese einlesen, adden von Typen und Variablen und wieder zurückschreiben in der originalen Datei.

Factory-Klasse/FactoryIEC/ FactoryJava/Schnittstelle:

Die Factory hilft bei der Verwaltung der Programmiersprachen für den Symbolparser. Außerdem definiert sie eine Einheitliche Schnittstelle für den Symbolparser und deren Unterfactorys und den existierenden Var/Type Klassen

Var/Type Klasse:

Verwaltet die Variablen und Typen der dazugehörigen Programmiersprachen und gibt an, wie die Variablen und Typen dementsprechen in die Datei eingeschrieben werden. Natürlich gibt es für jede Programmiersprache eine eigene Variablen /Typen Klasse die von den Variable/Typ – Schablonen vererben.

Komponentenentwurf

Siehe Doxygen-Dokumentation für detaillierte Informationen zum Komponentenentwurf.

Doxygen

Die Doxygen-Dokumentation enthält ausführliche Informationen zum Code, zur Klassenstruktur und zur

Verwendung der einzelnen Komponenten. Sie ist im Ordner "Doxygen/html" verfügbar.

Dateibeschreibung

In diesem Beispiel verwenden wir vier Dateien um die Progammiersprachen erhalten zu haben/zu speichern. Eine JavaVar.sym, JavaType.sym, IECVar.sym und IECType.sym. Diese werden im Programm erzeugt falls zum ersten mal eine Datei verwaltet wird und ansonsten wird diese mithilfe von fstream aufgerufen.

Testprotokollierung

Die Tests befinden sich in den Datenfiles jedoch kommt manchmal ein Lesezugriffsverletzungs error raus. Trotzdem sollten die meisten Funktionalitäten noch in Ordnung sein.

Sourcode:

Es sollte immer Header und cpp sein, main ist ganz am Schluss.

```
/** @file
* @brief Declaration of the Object class
* @details Defines an abstract Object class. Created by Paul Engelhart.
*/

#ifndef OBJECT_H
#define OBJECT_H

/** @brief Abstract Object class. */
class Object {

public:

    /** @brief Virtual destructor for Object. */
    virtual ~Object() = 0;

protected:
```

```
    /** @brief Default constructor for Object. */
    Object() = default;

private:

};

#endif

/** @file
* @brief Main file for the SymbolParser application
* @details This file contains the main function that demonstrates the use of
Factory, FactoryIEC, FactoryJava, and SymbolParser classes.
* @creator Both
*/

#include <stdio.h>
#include "Factory.h"
#include "FactoryIEC.h"
#include "SymbolParser.h"
#include "FactoryJava.h"

/** @brief Main function to demonstrate the SymbolParser application.
* @return 0 on successful execution.
*/
int main() {

    SymbolParser parser;

    // Using FactoryJava
    parser.SetFactory(FactoryJava::GetInstance());
    parser.AddType("Test");
    parser.AddVariable("bb", "Test");
    parser.AddType("Bibo");
    parser.AddVariable("blob", "Bibo");

    // Using FactoryIEC
    parser.SetFactory(FactoryIEC::GetInstance());
    parser.AddType("Test2");
    parser.AddVariable("bbs", "Test2");

    // Using FactoryJava again
    parser.SetFactory(FactoryJava::GetInstance());
    parser.AddType("Test3");
    parser.AddVariable("mbbss", "Test3");

    // Using FactoryIEC again
    parser.SetFactory(FactoryIEC::GetInstance());
    parser.AddType("Test4");
```

```
    parser.AddVariable("mbbssbs", "Test4");

    return 0;
}
```

```
/** @file
* @brief Declaration of the Factory class
* @details Defines an abstract factory class to create variables and types.
Created by Paul Engelhart.
*/

#include "Object.h"

#include "Var.h"
#include <memory>
#include "Type.h"
#include <string>
#include <vector>
#include <fstream>
#include <iostream>

#ifndef FACTORY_H
#define FACTORY_H

/** @brief Abstract Factory class for creating variables and types. */
class Factory {

public:

    /** @brief Default constructor for Factory. */
    Factory() = default;

    /** @brief Default destructor for Factory. */
    ~Factory() = default;

    /** @brief Copy constructor for Factory.
    * @param other The Factory object to copy.
    */
    Factory(Factory const&) = default;

    /** @brief Copy assignment operator for Factory.
    * @param other The Factory object to assign from.
    * @return A reference to the assigned Factory object.
    */
    Factory& operator= (Factory const&) = default;

    /** @brief Test function (TODO: Remove) */
    virtual void Print() = 0;
```

```
    /** @brief Gets the file name for variables.
    * @return The file name for variables.
    */
    virtual std::string GetVarFileName() = 0;

    /** @brief Gets the file name for types.
    * @return The file name for types.
    */
    virtual std::string GetTypeFileName() = 0;

    /** @brief Reads variables from a file based on the provided types.
    * @param types The types used to create variables.
    * @return A vector of shared pointers to the created variables.
    */
    virtual std::vector<std::shared_ptr<Var>> ReadVars(const
std::vector<std::shared_ptr<Type>>& types) = 0;

    /** @brief Reads types from a file.
    * @return A vector of shared pointers to the created types.
    */
    virtual std::vector<std::shared_ptr<Type>> ReadTypes() = 0;

    /** @brief Creates a variable with the given name and type.
    * @param name The name of the variable.
    * @param type The type of the variable.
    * @return A shared pointer to the created variable.
    */
    virtual std::shared_ptr<Var> CreateVar(std::string name,
std::shared_ptr<Type> type) = 0;

    /** @brief Creates a type with the given name.
    * @param name The name of the type.
    * @return A shared pointer to the created type.
    */
    virtual std::shared_ptr<Type> CreateType(std::string name) = 0;

protected:
private:

};

#endif
```

```
/** @file
* @brief Declaration of the FactoryJava class
```

```
* @details Defines a FactoryJava class that extends the abstract Factory class
for creating variables and types in Java. Implements singleton pattern.
Created by Paul Engelhart.
*/

#include "Factory.h"

#ifndef FACTORYJAVA_H
#define FACTORYJAVA_H

/** @brief FactoryIEC class that extends the abstract Factory class for
creating variables and types in Java. */
class FactoryIEC : public Factory {

public:

    /** @brief Gets the singleton instance of FactoryIEC.
    * @return The singleton instance of FactoryIEC.
    */
    static Factory& GetInstance() {
        if (mInstance == nullptr) mInstance = std::unique_ptr<Factory>{ new
FactoryIEC };
        return *mInstance;
    }

    /** @brief For testing purposes, prints a message with a counter.
    */
    void Print() { std::cout << i++ << "\n"; };

    /** @brief Creates a variable with the given name and type.
    * @param name The name of the variable.
    * @param type The type of the variable.
    * @return A shared pointer to the created variable.
    */
    std::shared_ptr<Var> CreateVar(std::string name, std::shared_ptr<Type>
type);

    /** @brief Creates a type with the given name.
    * @param name The name of the type.
    * @return A shared pointer to the created type.
    */
    std::shared_ptr<Type> CreateType(std::string name);

    /** @brief Deletes the singleton instance of FactoryIEC. */
    static void Delete() {
        mInstance.reset();
    }

    /** @brief Gets the file name for variables.
    * @return The file name for variables.
```

```
    */
    std::string GetVarFileName();

    /** @brief Gets the file name for types.
    * @return The file name for types.
    */
    std::string GetTypeFileName();

    /** @brief Reads variables from a file based on the provided types.
    * @param types The types used to create variables.
    * @return A vector of shared pointers to the created variables.
    */
    std::vector<std::shared_ptr<Var>> ReadVars(const
std::vector<std::shared_ptr<Type>>& types);

    /** @brief Reads types from a file.
    * @return A vector of shared pointers to the created types.
    */
    std::vector<std::shared_ptr<Type>> ReadTypes();

protected:

    /** @brief Default constructor for FactoryIEC. */
    FactoryIEC() = default;

private:
    int i = 0;
    static std::unique_ptr<Factory> mInstance; /**< The singleton instance of
FactoryIEC. */
    ~FactoryIEC() = default;
    FactoryIEC(const FactoryIEC&) = delete;
    FactoryIEC& operator=(const FactoryIEC&) = delete;

    std::string varFile = "IECVars.sym"; /**< The file name for variables. */
    std::string typeFile = "IECTypes.sym"; /**< The file name for types. */
};

#endif

/** @file
* @brief Implementation of the FactoryIEC class
* @details Provides methods to create variables and types based on IEC
standards.
* @author Harald Kiss
*/

#include "FactoryIEC.h"
#include <memory>

#include "IECType.h"
```

```cpp
#include "IECVar.h"
#include <sstream>

/** @brief The singleton instance of the FactoryIEC class. */
std::unique_ptr<Factory> FactoryIEC::mInstance{ nullptr };


std::shared_ptr<Var> FactoryIEC::CreateVar(std::string name,
std::shared_ptr<Type> type)
{
    auto var = std::shared_ptr<Var>(std::make_shared<IECVar>(name, type));
    return var;
}

std::shared_ptr<Type> FactoryIEC::CreateType(std::string name)
{
    return std::shared_ptr<Type>(std::make_shared<IECType>(name));
}
std::string FactoryIEC::GetVarFileName()
{
    return varFile;
}

std::string FactoryIEC::GetTypeFileName()
{
    return typeFile;
}
std::vector<std::shared_ptr<Var>> FactoryIEC::ReadVars(const
std::vector<std::shared_ptr<Type>>& types)
{

    std::fstream new_file;

    std::vector<std::shared_ptr<Var>> VVec;

    std::vector<std::shared_ptr<Var>> FalseVec;

    new_file.open(this->GetVarFileName(), std::ios::in);

    bool Good = true;

    // Checking whether the file is open.
    if (new_file.is_open()) {
        std::string inStr;

        // Read data from the file object and put it into a string.
        while (getline(new_file, inStr)) {

            std::vector<std::string> v;
```

```
            std::stringstream ss(inStr);

            //Splitting big string
            while (ss.good()) {
                std::string substr;
                getline(ss, substr, ' ');
                v.push_back(substr);
            }

            if (v.size() == 4) {

                if (v.at(0) != "VAR") {

                    std::cout << "Wrong syntax!" << std::endl;
                    Good = false;
                    break;
                }

                if (Good == false) break;

                //searching if var already exists
                for (size_t j = 0; j < VVec.size(); j++) {

                    if (VVec.at(j)->GetName() == v.at(1))
                    {
                        std::cout << "Variable already existing!" <<
std::endl;
                        Good = false;
                        break;
                    }


                }

                if (Good == false) break;

                if (v.at(2) != ":") {

                    std::cout << "Wrong syntax!" << std::endl;
                    Good = false;
                    break;
                }

                if (Good == false) break;

                //Deleting the Semikolon
                v.at(3).erase(v.at(3).size() - 1);

                for (size_t i = 0; i < VVec.size(); i++) {
```

```
                    //Looking if the type exists
                    if (types.at(i)->GetName() == v.at(3))
                    {

                        VVec.push_back(CreateVar(v.at(1), types.at(i)));
                        Good = true;
                        break;
                    }
                    else {

                        Good = false;

                    }

                }

                if (Good == false) break;

            }

        }


    }
    else {

        std::cout << "File couldn't be opened!" << std::endl;

    }
    // Close the file object.
    new_file.close();

    if (Good == false) {

        return FalseVec;
    }

    return VVec;

}

std::vector<std::shared_ptr<Type>> FactoryIEC::ReadTypes()
{

    std::fstream new_file;

    std::vector<std::shared_ptr<Type>> TVec;
    std::vector<std::shared_ptr<Type>> FalseVec;

    new_file.open(this->GetTypeFileName(), std::ios::in);
```

```
bool Good = true;

// Checking whether the file is open.
if (new_file.is_open()) {
    std::string inStr;

    // Read data from the file object and put it into a string.
    while (getline(new_file, inStr)) {

        std::vector<std::string> v;

        std::stringstream ss(inStr);

        while (ss.good()) {
            std::string substr;
            getline(ss, substr, ' ');
            v.push_back(substr);
        }

        if (v.size() == 2) {

            if (v.at(0) != "TYPE") {

                std::cout << "Not the right syntax!" << std::endl;
                Good = false;
                break;
            }

            if (Good == false) break;

            //looking if type exists already
            for (size_t i = 0; i < TVec.size(); i++) {

                if (TVec.at(i)->GetName() == v.at(1))
                {
                    std::cout << "Type already existing!" << std::endl;
                    Good = false;
                    break;
                }

            }

            if (Good == false) break;

            TVec.push_back(CreateType(v.at(1)));

        }

    }
```

```
    }
    else {

        std::cout << "File couldn't be opened!" << std::endl;

    }


    // Close the file object.
    new_file.close();

    if (Good == false) {

        std::cout << "Incorrect File structur" << std::endl;
        return FalseVec;
    }

    return TVec;

}
```

```
/** @file
* @brief Declaration of the FactoryJava class
* @details Defines a FactoryJava class that extends the abstract Factory class
for creating variables and types based on Java standards. Implements singleton
pattern. Created by Paul Engelhart.
*/

#include "Factory.h"
#include <iostream>

#ifndef FACTORYIEC_H
#define FACTORYIEC_H

/** @brief FactoryJava class that extends the abstract Factory class for
creating variables and types based on Java standards. */
class FactoryJava : public Factory {
public:

    /** @brief Gets the singleton instance of FactoryIEC.
    * @return The singleton instance of FactoryIEC.
    */
    static Factory& GetInstance() {
        if (mInstance == nullptr) mInstance = std::unique_ptr<Factory>{ new
FactoryJava };
        return *mInstance;
```

```
    }

    void Print() { std::cout << i++ << "\n"; };

    /** @brief Creates a variable with the given name and type.
    * @param name The name of the variable.
    * @param type The type of the variable.
    * @return A shared pointer to the created variable.
    */
    std::shared_ptr<Var> CreateVar(std::string name, std::shared_ptr<Type>
type);

    /** @brief Creates a type with the given name.
    * @param name The name of the type.
    * @return A shared pointer to the created type.
    */
    std::shared_ptr<Type> CreateType(std::string name);

    /** @brief Deletes the singleton instance of FactoryJava. */
    static void Delete() {
        mInstance.reset();
    }

    /** @brief Gets the file name for variables.
    * @return The file name for variables.
    */
    std::string GetVarFileName();

    /** @brief Gets the file name for types.
    * @return The file name for types.
    */
    std::string GetTypeFileName();

    /** @brief Reads variables from a file based on the provided types.
    * @param types The types used to create variables.
    * @return A vector of shared pointers to the created variables.
    */
    std::vector<std::shared_ptr<Var>> ReadVars(const
std::vector<std::shared_ptr<Type>>& types);

    /** @brief Reads types from a file.
    * @return A vector of shared pointers to the created types.
    */
    std::vector<std::shared_ptr<Type>> ReadTypes();

protected:

    /** @brief Default constructor for FactoryJava. */
    FactoryJava() = default;
```

```
private:
    int i = 0;
    static std::unique_ptr<Factory> mInstance; /**< The singleton instance of
FactoryJava. */
    ~FactoryJava() = default;
    FactoryJava(const FactoryJava&) = delete;
    FactoryJava& operator=(const FactoryJava&) = delete;

    const std::string varFile = "JavaVars.sym"; /**< The file name for
variables. */
    const std::string typeFile = "JavaTypes.sym"; /**< The file name for
types. */
};

#endif

/** @file
* @brief Implementation of the FactoryJava class
* @details Provides methods to create variables and types for Java. Created by
Kiss Harald.
*/

#include "FactoryJava.h"
#include "JavaVar.h"
#include "JavaType.h"
#include <sstream>

/** @brief The singleton instance of the FactoryJava class. */
std::unique_ptr<Factory> FactoryJava::mInstance{ nullptr };

std::shared_ptr<Var> FactoryJava::CreateVar(std::string name,
std::shared_ptr<Type> type)
{
    auto var = std::shared_ptr<Var>(std::make_shared<JavaVar>(name, type));
    return var;
}

std::shared_ptr<Type> FactoryJava::CreateType(std::string name)
{
    return std::shared_ptr<Type>(std::make_shared<JavaType>(name));
}

std::string FactoryJava::GetVarFileName()
{
    return varFile;
}

std::string FactoryJava::GetTypeFileName()
{
    return typeFile;
```

```
}

std::vector<std::shared_ptr<Var>> FactoryJava::ReadVars(const
std::vector<std::shared_ptr<Type>>& types)
{
    std::fstream new_file;

    std::vector<std::shared_ptr<Var>> VVec;

    std::vector<std::shared_ptr<Var>> FalseVec;

    std::shared_ptr<Type> Helper;

    new_file.open(this->GetVarFileName(), std::ios::in);

    bool Good = true;

    // Checking whether the file is open.
    if (new_file.is_open()) {
        std::string inStr;

        // Read data from the file object and put it into a string.
        while (getline(new_file, inStr)) {

            std::vector<std::string> v;

            std::stringstream ss(inStr);

            // Splitting the big string into strings
            while (ss.good()) {
                std::string substr;
                getline(ss, substr, ' ');
                v.push_back(substr);
            }

            if (v.size() == 2) {

                for (size_t i = 0; i < VVec.size(); i++) {

                    // Looking if the type exists
                    if (types.at(i)->GetName() == v.at(0))
                    {

                        Helper = types.at(i);
                        Good = true;
                        break;
                    }
                    else {

                        Good = false;
```

```
                    }

                }

                if (Good == false) break;

                // Deleting the Semikolon at the end
                v.at(1).erase(v.at(1).size() - 1);

                for (size_t j = 0; j < VVec.size(); j++) {

                    if (VVec.at(j)->GetName() == v.at(1))
                    {
                        std::cout << "Variable already existing!" <<
std::endl;
                        Good = false;
                        break;
                    }

                }

                if (Good == false) break;

                VVec.push_back(CreateVar(v.at(1), Helper));

            }

        }


    }

    // Close the file object.
    new_file.close();

    if (Good == false) {

        return FalseVec;
    }

    return VVec;
}

/** @brief Reads types from a file for Java.
* @return A vector of shared pointers to the created types.
*/
std::vector<std::shared_ptr<Type>> FactoryJava::ReadTypes()
{
```

```
    std::fstream new_file;

    std::vector<std::shared_ptr<Type>> TVec;
    std::vector<std::shared_ptr<Type>> FalseVec;

    new_file.open(this->GetTypeFileName(), std::ios::in);

    bool Good = true;

    // Checking whether the file is open.
    if (new_file.is_open()) {
        std::string inStr;

        // Read data from the file object and put it into a string.
        while (getline(new_file, inStr)) {

            std::vector<std::string> v;

            std::stringstream ss(inStr);

            // Splitting big string
            while (ss.good()) {
                std::string substr;
                getline(ss, substr, ' ');
                v.push_back(substr);
            }

            if (v.size() == 2) {

                if (v.at(0) != "class") {

                    std::cout << "Wrong syntax!" << std::endl;
                    Good = false;
                    break;
                }

                if (Good == false) break;

                // Searching if type exists
                for (size_t i = 0; i < TVec.size(); i++) {

                    if (TVec.at(i)->GetName() == v.at(1))
                    {
                        std::cout << "Type already existing!" << std::endl;
                        Good = false;
                        break;
                    }

                }
```

```
                if (Good == false) break;

                TVec.push_back(CreateType(v.at(1)));

            }

        }


    }
    else {

        std::cout << "File couldn't be opened!" << std::endl;

    }
    // Close the file object.
    new_file.close();

    if (Good == false) {

        return FalseVec;
    }

    return TVec;
}
```

```
/** @file
* @brief Declaration of the IECType class
* @details Defines an IECType class that inherits from the abstract Type
class. Created by Paul Engelhart.
*/

#ifndef IECTYPE_H
#define IECTYPE_H

#include "Type.h"

/** @brief IECType class that inherits from the abstract Type class. */
class IECType : public Type {

public:

    /** @brief Constructor for IECType.
    * @param name The name of the IECType.
    */
    IECType(std::string name);

    /** @brief Converts the IECType to a string.
```

```
    * @return A string representation of the IECType.
    */
    std::string ToString();

protected:

private:

};

#endif


/** @file
* @brief Implementation of the IECType class
* @details Defines the methods for creating and manipulating IEC types.
Created by Kiss Harald.
*/

#include "IECType.h"
#include <sstream>


IECType::IECType(std::string name) : Type(name)
{
}
std::string IECType::ToString()
{
    std::ostringstream ost;
    ost << "TYPE " << GetName() << std::endl;

    return ost.str();
}
```

```
/** @file
* @brief Declaration of the IECVar class
* @details Defines an IECVar class that inherits from the abstract Var class.
Created by Paul Engelhart.
*/

#ifndef IECVAR_H
#define IECVAR_H

#include "Var.h"

/** @brief IECVar class that inherits from the abstract Var class. */
class IECVar : public Var {
```

```
public:

    /** @brief Constructor for IECVar.
    * @param name The name of the IECVar.
    * @param type The type of the IECVar.
    */
    IECVar(std::string name, std::shared_ptr<Type> type);

    /** @brief Converts the IECVar to a string.
    * @return A string representation of the IECVar.
    */
    std::string ToString();

protected:

private:

};

#endif

/** @file
* @brief Implementation of the IECVar class
* @details Defines the methods for creating and manipulating IEC variables.
Created by Kiss Harald.
*/

#include "IECVar.h"
#include <sstream>

IECVar::IECVar(std::string name, std::shared_ptr<Type> type) : Var(name, type)
{
}

std::string IECVar::ToString()
{
    std::ostringstream ost;
    ost << "VAR" << " " << GetName() << " : " << type->GetName() << ";" <<
std::endl;

    return ost.str();
}
```

```
/** @file
* @brief Declaration of the JavaType class
* @details Defines a JavaType class that inherits from the abstract Type
class. Created by Paul Engelhart.
*/
```

```
#ifndef JAVATYPE_H
#define JAVATYPE_H

#include "Type.h"

/** @brief JavaType class that inherits from the abstract Type class. */
class JavaType : public Type {

public:

    /** @brief Constructor for JavaType.
    * @param name The name of the JavaType.
    */
    JavaType(std::string name);

    /** @brief Converts the JavaType to a string.
    * @return A string representation of the JavaType.
    */
    std::string ToString();

protected:

private:

};

#endif

/** @file
* @brief Implementation of the JavaType class
* @details Defines the methods for creating and manipulating Java types.
Created by Harald Kiss.
*/

#include "JavaType.h"
#include <sstream>

/** @brief Constructor for JavaType.
* @param name The name of the Java type.
*/
JavaType::JavaType(std::string name) : Type(name)
{
}

std::string JavaType::ToString()
{
    std::ostringstream ost;
    ost << "class" << " " << GetName();
```

```
    return ost.str();
}
```

```
/** @file
* @brief Declaration of the JavaVar class
* @details Defines a JavaVar class that inherits from the abstract Var class
and uses JavaType. Created by Paul Engelhart.
*/

#ifndef JAVAVAR_H
#define JAVAVAR_H

#include "Var.h"
#include "JavaType.h"

/** @brief JavaVar class that inherits from the abstract Var class and uses
JavaType. */
class JavaVar : public Var {

public:

    /** @brief Constructor for JavaVar.
    * @param name The name of the JavaVar.
    * @param type The type of the JavaVar, expected to be a JavaType.
    */
    JavaVar(std::string name, std::shared_ptr<Type> type);

    /** @brief Converts the JavaVar to a string.
    * @return A string representation of the JavaVar.
    */
    std::string ToString();

protected:

private:

};

#endif

/** @file
* @brief Implementation of the JavaVar class
* @details Defines the methods for creating and manipulating Java variables.
Created by Kiss Harald.
*/

#include "JavaVar.h"
#include <sstream>
```

```
JavaVar::JavaVar(std::string name, std::shared_ptr<Type> type) : Var(name,
type)
{
}
std::string JavaVar::ToString()
{
    //std::string Help = type->GetName();
    std::ostringstream ost;
    ost << type->GetName() << " " << GetName() << ";" << std::endl;

    return ost.str();
}


/** @file
* @brief Declaration of the SymbolParser class
* @details Defines a SymbolParser class for parsing symbols and managing types
and variables. Created by Paul Engelhart.
*/

#ifndef SYMBOLPARSER_H
#define SYMBOLPARSER_H

#include "Factory.h"
#include <string>
#include <vector>
#include "Type.h"
#include "Var.h"

/** @brief SymbolParser class for parsing symbols and managing types and
variables. */
class SymbolParser {

public:

    /** @brief Default constructor for SymbolParser. */
    SymbolParser() = default;

    /** @brief Default destructor for SymbolParser. */
    ~SymbolParser() = default;

    /** @brief Adds a type to the SymbolParser.
    * @param typeName The name of the type to be added.
    */
    void AddType(const std::string& typeName);

    /** @brief Adds a variable to the SymbolParser.
    * @param variableName The name of the variable to be added.
```

```
    * @param typeName The name of the type of the variable.
    */
    void AddVariable(const std::string& variableName, const std::string&
typeName);

    /** @brief Sets the factory for the SymbolParser.
    * @param factory The factory to be set.
    */
    void SetFactory(Factory& factory);

private:

    /** @brief Saves the types and variables to a file. */
    void Save();

    /** @brief Reads types and variables from files. */
    void ReadFromFile();

    /** @brief Checks if a variable with the given name already exists.
    * @param name The name of the variable to check.
    * @return True if the variable already exists, false otherwise.
    */
    bool VariableAlreadyExists(const std::string& name);

    /** @brief Finds a type with the given name.
    * @param typeName The name of the type to find.
    * @return A shared pointer to the found type, or nullptr if not found.
    */
    std::shared_ptr<Type> FindType(const std::string& typeName);

    std::vector<std::shared_ptr<Var>> vars; /**< Vector to store variables. */
    std::vector<std::shared_ptr<Type>> types; /**< Vector to store types. */
    Factory* currentFact; /**< Pointer to the current factory. */
    Var* currentVar; /**< Pointer to the current variable. */
    Type* currentType; /**< Pointer to the current type. */
    bool firstPass = true; /**< Flag to indicate the first pass. */
};

#endif

/** @file
* @brief Implementation of the SymbolParser class
* @details This file contains the implementation of the SymbolParser class,
which is responsible for parsing symbols, adding types and variables, and
managing factories.
* @creator Harald Kiss
*/

#include "SymbolParser.h"
#include <iostream>
```

```cpp
void SymbolParser::AddType(const std::string& typeName)
{
    if (FindType(typeName) != nullptr)
    {
        std::cout << "This type does not exist!" << std::endl;
        return;
    }

    types.push_back(this->currentFact->CreateType(typeName));

    return;
}

bool SymbolParser::VariableAlreadyExists(const std::string& name)
{
    for (auto v : vars) {
        if (v->GetName() == name) return true;
    }
    return false;
}

std::shared_ptr<Type> SymbolParser::FindType(const std::string& typeName)
{
    for (auto t : types)
    {
        if (t->GetName() == typeName) return t;
    }

    return nullptr;
}

#include <iostream>

void SymbolParser::Save()
{
    std::ofstream outFileType{ this->currentFact->GetTypeFileName() };
    std::ofstream outFileVar{ this->currentFact->GetVarFileName() };

    if (outFileType.fail() || outFileVar.fail()) {

        std::cout << "Error Fail!" << std::endl;
        outFileType.close();
        outFileVar.close();
        return;
    }

    outFileType.seekp(0);
    outFileVar.seekp(0);
```

```
    //Writing the Vars and Types with the correct structure in file
    for (auto type : types) {
        outFileType << type->ToString() << std::endl;
    }

    for (auto var : vars) {
        outFileVar << var->ToString() << std::endl;
    }

    // Check for write errors
    if (outFileType.fail() || outFileVar.fail()) {
        std::cout << "Error Write!" << std::endl;
    }

    // Close the files
    outFileType.close();
    outFileVar.close();
}

void SymbolParser::ReadFromFile() {

    types = this->currentFact->ReadTypes();

    vars = this->currentFact->ReadVars(types);

}
void SymbolParser::AddVariable(const std::string& variableName, const
std::string& typeName)
{
    if (VariableAlreadyExists(variableName))
    {
        std::cout << "Variable already existing!" << std::endl;
        return;
    }
    if (FindType(typeName) == nullptr)
    {
        std::cout << "No searched Type found!" << std::endl;
        return;
    }

    vars.push_back(this->currentFact->CreateVar(variableName, this-
>FindType(typeName)));

}
void SymbolParser::SetFactory(Factory& factory)
{
    // save old Factory (only if not the first pass);
    // Clear old data

    if (firstPass != true) {
```

```
        Save();
        vars.clear();
    }
    this->currentFact = &factory;

    types.clear();
    ReadFromFile();

    currentFact->Print();

    firstPass = false;
    // read new factory
}
```

```
/** @file
* @brief Declaration of the Type class
* @details Defines an abstract Type class. Created by Paul Engelhart.
*/

#ifndef TYPE_H
#define TYPE_H

#include <string>

/** @brief Abstract Type class. */
class Type {

public:

    /** @brief Constructor for Type.
    * @param name The name of the Type.
    */
    Type(std::string name);

    /** @brief Virtual destructor for Type. */
    virtual ~Type() = default;

    /** @brief Abstract method to convert the Type to a string.
    * @return A string representation of the Type.
    */
    virtual std::string ToString() = 0;

    /** @brief Gets the name of the Type.
    * @return The name of the Type.
    */
    std::string GetName();
```

```
protected:

    std::string name; /**< The name of the Type. */

};

#endif


/** @file
* @brief Implementation of the Type class
* @details This file contains the implementation of the Type class, which
represents a data type.
* @creator Paul Engelhart
*/

#include "Type.h"

Type::Type(std::string name) : name(name)
{
}
std::string Type::GetName()
{
    return name;
}
```

```
/** @file
* @brief Declaration of the Var class
* @details Defines an abstract Var class. Created by Paul Engelhart.
*/

#ifndef VAR_H
#define VAR_H

#include <memory>
#include "Type.h"

/** @brief Abstract Var class. */
class Var {

public:

    /** @brief Constructor for Var.
    * @param name The name of the Var.
    * @param type A shared pointer to the Type of the Var.
    */
    Var(std::string name, std::shared_ptr<Type> type);
```

```
    /** @brief Default constructor for Var. */
    Var() = default;

    /** @brief Virtual destructor for Var. */
    virtual ~Var() = default;

    /** @brief Abstract method to convert the Var to a string.
    * @return A string representation of the Var.
    */
    virtual std::string ToString() = 0;

    /** @brief Gets the name of the Var.
    * @return The name of the Var.
    */
    std::string GetName();

    /** @brief Gets the type of the Var.
    * @return A shared pointer to the Type of the Var.
    */
    std::shared_ptr<Type> GetType();

protected:

    std::string name; /**< The name of the Var. */
    std::shared_ptr<Type> type; /**< A shared pointer to the Type of the Var.
*/

};

#endif

/** @file
* @brief Implementation of the Var class
* @details This file contains the implementation of the Var class, which
represents a variable.
* @creator Paul Engelhart
*/

#include "Var.h"

Var::Var(std::string name, std::shared_ptr<Type> type) : name(name),
type(type)
{
}

std::string Var::GetName()
{
    return name;
}
```

```
std::shared_ptr<Type> Var::GetType()
{
    return type;
}
```