1 SUV

In dieser Aufgabe wird eine Klassen-Hierarchie bestehend aus den Klassen `Vehicle` und `Suv` aufgebaut. Hierbei soll das Überschreiben von Methoden und der Aufruf von Basis-Konstruktoren ausgenutzt werden.

1.1 Aufgabe

Implementieren Sie beide Klassen, damit beim Ausführen des Test-Programmes folgender Output ausgegeben wird:

```
Vehicle weighs 1200
its speed is 145
```

main.cpp

```cpp
  std::shared_ptr<Vehicle> vehicle = std::make_shared<Suv>(1200, 145);
  std::cout << "Vehicle weighs " << vehicle->GetMass() << std::endl;
  std::cout << "its speed is " << vehicle->GetSpeed() << std::endl;
```

vehicle.h

```cpp
#pragma once

#include <cstddef>

class Vehicle {
 public:
  Vehicle(size_t mass);

  size_t GetMass() const;
  int GetSpeed() const { return 2; };

 private:
  size_t mass_;
};
```

suv.h

```cpp
#pragma once

#include <cstddef>

#include "vehicle.h"

class Suv : public Vehicle {
```

```
8   public:
9    Suv(size_t mass, double speed);
10
11   double GetSpeed() const;
12
13  private:
14   double speed_;
15 };
```

2 Virtueller Destruktor

In C++ gibt es kein separates Interface-Konzept wie in Java. Man kann Interfaces aber mit virtuellen Methoden erstellen. In dieser Aufgabe sollen Sie die Frage beantworten, weshalb ein virtueller Destruktor in Basis-Klassen eine wichtige Rolle spielt.

2.1 Aufgabe

Kompilieren Sie das gegebene Programm mit und ohne virtuellen Destruktor der Interface-Klasse `BarInterface`, analysieren Sie den Output des Compilers und des Test-Programms und beantworten Sie folgende Fragen:

a) Was ist problematisch an der Zeile a)?
b) Was ist problematisch an der Zeile b)?

main.cpp

```
1   // lambda
2   auto descriptor = [](BarInterface* obj) {
3     obj->Describe();
4   };
5
6   std::cout << "BarTester Testing..." << std::endl;
7   BarTester* obj1 = new BarTester("Declared with BarTester");
8   descriptor(obj1);
9   delete obj1;
10  obj1 = nullptr;
11
12  std::cout << "BarInterface Testing..." << std::endl;
13  BarInterface* obj2 = new BarTester("Declared with BarInterface");
14  descriptor(obj2);
15  delete obj2;
16  obj2 = nullptr;                                    // a)
17
```

```
18    std::cout << std::endl << "BarTester not defined..." << std::endl;
19    descriptor(new BarTester("Not defined"));             // b)
```

bar_interface.h

```
1  #pragma once
2
3  class BarInterface {
4   public:
5    virtual ~BarInterface() = default;
6    virtual void Describe() = 0;
7  };
```

bar_tester.h

```
1  #pragma once
2
3  #include <string>
4
5  #include "bar_interface.h"
6
7  class BarTester : public BarInterface {
8   public:
9    explicit BarTester(const std::string& name);
10   ~BarTester();
11
12   void Describe();
13
14  private:
15   std::string private_name_;
16 };
```

bar_tester.cpp

```
1  #include "bar_tester.h"
2
3  #include <iostream>
4
5  BarTester::BarTester(const std::string& name) : private_name_(name) {
6    std::cout << "BarTester constructor" << std::endl;
7  }
8
9  BarTester::~BarTester() {
10   std::cout << "BarTester destructor" << std::endl;
11 }
12
13 void BarTester::Describe() {
14   std::cout << "I'm BarTester [" << private_name_ << "]" << std::endl;
15 }
```

3 Interfaces und Smart-Pointer

Vererbung und im Speziellen auch Interfaces funktionieren nur mit dynamischem Binding wie gewünscht. Das heisst, es müssen immer Pointer- oder Referenz-Typen für die Funktionsaufrufe verwendet werden. Die Smart-Pointer-Klassen ab C++11 sind ebenfalls darauf ausgelegt und unterstützen dementsprechend implizite Up-Casts und dynamisches Binding.

3.1 Aufgabe

a) Schreiben Sie ein Interface `SocketInterface`, welches folgende Methoden definiert:

```
1 bool Send(const std::string& message)
2 bool Receive(std::string* message)
```

b) Erstellen Sie zwei Implementationen dieses Interface: `DummySocket` und `LoopbackSocket`. Der Dummy soll immer senden und auch empfangen können. Er empfängt immer denselben Payload. Der Loopback soll das gesendete in einer `std::deque` speichern und über `Receive` zurückgeben, solange die Queue nicht leer ist.

c) Schreiben Sie einen Unit-Test, welcher einen `std::unique_ptr` vom Typ `SocketInterface` erstellt und diesen mit einer `DummySocket` Instanz füllt. Testen Sie nun die Funktionalität. Danach weisen Sie dem Pointer eine neue Instanz vom Typ `LoopbackSocket` zu und testen dessen Funktionalität.

4 Mehrfachvererbung

Generell sollte Mehrfachvererbung vermieden werden, um dadurch erzeugte Probleme zu verhindern. In gewissen Situationen kann sie aber trotzdem hilfreich sein. Z.B. wollen Sie eine `BaseImplementation` erstellen, welche gewisse Basis-Funktionen zur Verfügung stellt. Diese soll auch über ein Interface ansprechbar sein, weshalb Sie ein Interface `BaseInterface` definieren und davon ableiten. Dadurch können Sie nun unterschiedliche Implementationen von `BaseInterface` definieren.

Im weiteren Verlauf, wollen Sie eine erweitere Variante dieses Interface definieren, welche aber auch alle Basis-Funktionen anbietet. Sie leiten also `ExtendedInterface` von `BaseInterface` ab. Natürlich wollen Sie Implementationen zu diesem Interface erstellen, welche sinnvollerweise die Basis-Funktionalität nicht nochmals implementieren, sondern von der Basis erben. Dementsprechend implementieren Sie `ExtendedImplementation` welche von `BaseImplementation` ableitet. Nun haben Sie folgende Klassen-Hierarchie:

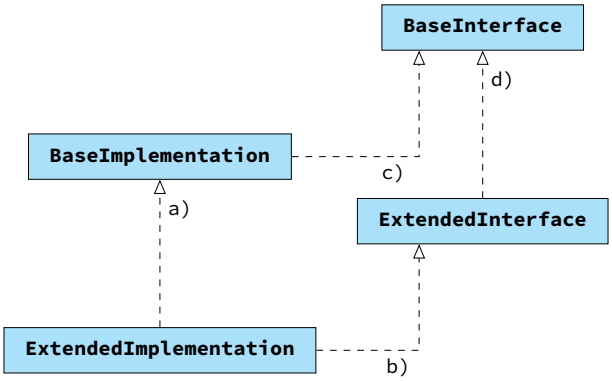

Abbildung 1: Klassen-Hierarchie

4.1 Aufgabe

a) Erstellen Sie alle beschriebenen Klassen mit den entsprechenden Vererbungs-Beziehungen. Als Interface Methode verwenden Sie z.B. `int BaseMethode()` bzw. `int ExtendedMethode()`.

b) Verwenden Sie die Klasse `ExtendedImplementation` über das Interface `ExtendedInterface`. Welche Vererbungs-Beziehungen müssen Sie nun durch *Virtual Inheritance* ersetzten, damit das Program kompiliert?