Die Klasse Serialport, Teil 2

Ziel, Inhalt

- Wir führen die Arbeit an unserer SerialPort Klasse fort. Wir lernen heute Objekte, die nur einmal vorhanden sein dürfen kennen und lernen wie man diese programmiert. Das Initialisieren des seriellen Port benötigt weitere API's, die wir ab heute kennen und anwenden können.

Die Klasse Serialport

Anforderungen/Analyse Klasse *SerialPort,* 2. Schritt

Design Patterns als Elemente von objektorientierter Software

Anhand der Klasse *SerialPort* lernen wir ein erstes Design Pattern kennen. Patterns sind Muster, nach denen immer wieder auftretende Probleme in Objektorientierten Systemen gelöst werden. Die bekanntesten Patterns findet ihr im sehr bekannten Buch „Design Patterns“ von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides.

Motivation zum Einsatz des Singleton-Pattern

Das Singleton-Pattern, das wir heute einsetzen werden, wird verwendet wenn von gewissen Objekten nur ein einziges im System vorhanden sein darf. Die Klasse *SerialPort*, die wir verwirklichen dient als Abstraktion des seriellen Ports an unserem PC. An einem Notebook ist häufig nur ein serieller Port vorhanden, der „COM1“. Auch an einem Desktop-PC sind im allgemeinen nicht mehr als zwei serielle Ports vorhanden. Vor allem ist von jeder Portnummer immer nur ein Anschluss vorhanden. Unsere Klasse *SerialPort* berücksichtigt dies nicht. Es ist möglich beliebig viele *SerialPort*-Objekte zu erzeugen und der Benutzer hat die Möglichkeit den gleichen Port mehrmals zu öffnen, wobei der zweite *SerialPort::open* Befehl zu einer Ausnahme führt, da der Aufruf von CreateFile fehlschlägt. Hier ein Beispiel:

```
int main()
{
    try
    {
        SerialPort meinSerialPort(1);
        SerialPort test(1); // schlägt fehl!
        test.close();
        meinSerialPort.close();
    }
    catch(const string& error)
    {
        cout << error << endl;
    }

    return 0;
}
```

Es kann nur ein *SerialPort*-Objekt für den COM-Port „COM1“ geben!
Hier tritt das Singleton-Pattern auf die Bühne und hilft uns Code wie oben zu verhindern.

Das Singleton-Pattern

Beispiel zum Singleton-Pattern

Das Singleton Pattern verhindert, dass man beliebig viele Objekte von einem Datentyp erstellen kann. Anstelle des Konstruktors, der *private* wird kommt eine statische Methode mit der man auf das gewünschte Objekt zugreifen kann.

Singleton.h

```
#ifndef SINGLETON_H
#define SINGLETON_H


class Singleton
{
    public:

        void doIt() const;

        // Methode um an das Objekt
        // zu gelangen
        static Singleton& getSingleton();

    private:
        // verhindere dass beliebige
        // viele Objekte/Instanzen
        // erzeugt werden mit privatem
        // Konstruktor
        Singleton();
};

#endif
```

Singleton.cpp

```
#include "singleton.h"

Singleton::Singleton()
{
}

void Singleton::doIt() const
{
}

Singleton& Singleton::getSingleton()
{
    // diese statische Variable wird
    // nur einmal erzeugt
    static Singleton dasEinzigeSingleton;

    return dasEinzigeSingleton;
}
```

Singletonklassen verwenden statische Methoden und manchmal auch statische Datenelemente.

Das Singleton-Pattern für die Klasse *SerialPort*

Bei der Klasse *SerialPort* ist die Lage noch ein wenig anders, als bei der Implementation der einfachen Klasse *Singleton*. Wir wollen eigentlich nur verhindern, dass für einen Port mehrere Objekte erzeugt werden. Wir werden in unserer Klasse *SerialPort* die Konstruktoren auch private machen. Dafür stellen wir eine statische *get*-Methode zur Verfügung bei der man angeben mss, welches *SerialPort*-Objekt man will. Es genügt dabei die Portnummer als Argument mitzugeben. Die Klasse *SerialPort* würde danach etwa so aussehen:

```
class SerialPort
{
    public:
        // statische get Methode
        SerialPort& getSerialPort(unsigned char portNumber);

        ~SerialPort();

        unsigned char readByte();
        void writeByte(unsigned char data);

    private:
        SerialPort();
        SerialPort(unsigned char portNumber);
        void open(unsigned char portNumber);
        void close();

    private:
        HANDLE m_handle;
};
```

Übung *SerialPort* als Singleton

Versuche die Klasse jetzt also so zu ergänzen, dass jedesmal wenn man ein Objekt *SerialPort* benötigt, die Methode *getSerialPort* aufgerufen werden kann. Dabei soll jedesmal das gleiche Objekt zurückgegeben werden, wenn diese statische Methode mit dem gleichen Argument aufgerufen wird.
Tipp: Mein erster Gedanke ist eine *std::map* einzusetzen. In dieser map wird als index die Portnummer verwendet um auf ein Objekt der Klasse *SerialPort* zuzugreifen. Wird die Methode *getSerialPort* aufgerufen wird in dieser map nachgesehen ob bereits ein Objekt existiert. Falls es nicht existiert wird es erzeugt, die *open* Methode wird aufgerufen und es wird in die map eingetragen. Die map wird dabei als statisches Objekt in der Klasse *SerialPort* definiert.