SerialPort, 4. Teil

Inhalt

- Wir beenden das Beispiel SerialPort, indem wir für die Klasse SerialPort ein Observer-Interface (siehe auch Observer-Pattern) definieren.

SerialPort, letzter Teil

Ein Observer für SerialPort

Das Interface SerialPortObserver

Damit sich verschiedene Clients, die sich für einen bestimmten seriellen Port interessieren von diesem Daten erhalten, definieren wir ein Observer-Interface. Dieses Interface soll dazu dienen, erhaltene Daten weiterzuleiten. Damit ein Client sich bei mehreren SerialPort-Objekten eintragen kann und diese Unterscheiden kann, gibt der SerialPort eine Referenz auf sich selber mit. Hier eine ungefähre Idee:

```
// Interface für einen Observer
// für den seriellen Port
class SerialPortObserver
{
    public:
        virtual void newSerialData(const SerialPort& port,
                                   char data) = 0;

};
```

Änderungen an der Klasse SerialPort

Diese Klasse muss nun noch eine Methode bieten mit der sich solche Observer eintragen können (z.B. *addObserver, removeObserver)*. Die Observer-Objekte können wir uns im SerialPort-Objekt in einem *std::set* merken.
Eine weitere dient dazu die Portnummer abzufragen, was nützlich sein kann wenn sich ein Observer bei mehreren SerialPort-Objekten einträgt.

```
typedef std::set<SerialPortObserver*> Observers;

class SerialPort
{
    public:
        ~SerialPort();

        void addObserver(SerialPortObserver* observer);
        void removeObserver(SerialPortObserver* observer);
        unsigned char getPortNumber() const;
//... etc.
    private:
        HANDLE        m_handle;
        unsigned char m_portNumber;
        Observers     m_observers;
};
```

Hier eine vorläufige Implementation:

```
////////////////////////////////////////////////

unsigned char SerialPort::getPortNumber() const
{
    return m_portNumber;
}

////////////////////////////////////////////////

void SerialPort::addObserver(SerialPortObserver* observer)
{
    m_observers.insert(observer);
}

////////////////////////////////////////////////

void SerialPort::removeObserver(SerialPortObserver* observer)
{
    m_observers.erase(observer);
}

////////////////////////////////////////////////
```

Der Lesende Thread

Die Idee ist nun im SerialPort-Objekt einen Thread zu erzeugen, sobald sich ein Observer einträgt um Daten zu erhalten. Sobald wir auf diesem Thread etwas vom seriellen Port empfangen, notifizieren wir einfach die Observer, die sich im set befinden.

Die Thread-Funktion

Die Thread Funktion muss eine statische Methode der Klasse SerialPort sein, dass heisst wir kennen in dieser Funktion nicht wirklich zu welchem SerialPort-Objekt der Thread gehört. Es gibt jedoch die Möglichkeit einem neuen Thread Daten mitzugeben. Wir geben also unserem Thread einen Zeiger auf das erzeugende SerialPort-Objekt mit.
Ergänze zuerst in der Klasse SerialPort das Datenelement für den Thread-Handle, und füge die statische Methode *ReadThread* in der Klassendeklaration ein.

Hier die Methode *addObserver*, in der wir den Thread erzeugen, falls es ihn noch nicht gibt:

```
void SerialPort::addObserver(SerialPortObserver* observer)
{
    m_observers.insert(observer);

    if(0 == m_readThread)
    {
        DWORD threadId = 0;
        m_readThread = CreateThread(0, 0,
                                    readThread,
                                    this,
                                    0,
                                    &threadId);
    }
}
```

Im lesenden Thread rufen wir auf das übergebene SerialPort-Objekt eine Methode *readAndDispatch* auf. Diese Funktion liest in einer Schleife Daten vom seriellen Port, solange es noch Observer im set hat. Hier zuerst unsere Thread-Funktion:

```
DWORD WINAPI SerialPort::readThread(LPVOID threadParam)
{
    SerialPort* thePort = (SerialPort*)threadParam;

    // thread nicht ohne Parameter erzeugen
    _ASSERT(thePort != 0);

    thePort->readAndDispatch();

    return 0;
}
```

Die readAndDispatch Methode

Und hier eine einfache Implementation der Methode *readAndDispatch*:

```
void SerialPort::readAndDispatch()
{
    while(!m_observers.empty())
    {
        unsigned char data = readByte();
        Observers::iterator it = m_observers.begin();
        Observers::iterator end = m_observers.end();

        for( ; it != end; ++it)
        {
            SerialPortObserver* observer = (*it);
            observer->newSerialData(*this, data);
        }
    }
}
```

Synchronisation zur Änderung des set

Beachte nun, dass die Methode *readAndDispatch* auf einem anderen Thread aufgerufen wird als die Methoden *addObserver*, *removeObserver*. Wir wissen aber, dass wir nicht gleichzeitig auf gemeinsame Daten zugreifen dürfen. Hier kommt unsere *CriticalSection* Klasse vom Abend 6 gerade recht:

```
#ifndef CRITICALSECTION_H
#define CRITICALSECTION_H

#include <windows.h>

class CriticalSection
{
public:
    CriticalSection()
    {
        InitializeCriticalSection(&_cs);
    }

    ~CriticalSection()
    {
        DeleteCriticalSection(&_cs);
    }

    void enter()
    {
        EnterCriticalSection(&_cs);
    }

    void leave()
    {
        LeaveCriticalSection(&_cs);
    }
private:
    CRITICAL_SECTION _cs;
};

class CSLock
{
public:
    CSLock(CriticalSection& cs) : _cs(cs)
    {
        _cs.enter();
    }
    ~CSLock()
    {
        _cs.leave();
    }

private:
    CriticalSection& _cs;
};

#endif
```

Ergänze nun die Klasse SerialPort um ein Objekt der Klasse CriticalSection:

```
class SerialPort
{
    public:
        ~SerialPort();

        void addObserver(SerialPortObserver* observer);
        void removeObserver(SerialPortObserver* observer);
        unsigned char getPortNumber() const;

        static SerialPort& getSerialPort(unsigned char portNumber);

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

        friend SerialPorts;

    private:
        SerialPort();
        void open(unsigned char port);
        void close();
        // diese Funktion wird aufgerufen
        // solange sich Observer im set
        // befinden
        void readAndDispatch();

        static DWORD WINAPI readThread(LPVOID threadParam);

    private:
        HANDLE        m_readThread;
        HANDLE        m_handle;
        unsigned char m_portNumber;
        Observers     m_observers;
        CriticalSection m_cs;
};
```

Und ändere die Methoden in der Klasse, die auf das *m_observers* set zugreifen so, dass vorher *CriticalSection::enter* und danach *CriticalSection::leave* aufgerufen wird. Am besten erzeugst du eine Variable vom Typ CSLock, denn diese macht beide Aufrufe automatisch.

Test und Verbesserungen

Versuche nun eine Klasse ObserverImpl zu schreiben, die sich auf ein SerialPort-Objekt einträgt (*addObserver*) und die erhaltenene Daten auf der Konsole ausgibt.
Durch Verwendung von asynchronem Lesen und Abbruch durch Events könnte die Klasse SerialPort noch sauberer gestaltet werden. Versuche es, denn so ist die Klasse noch nicht Profi-tauglich.