

Abend 11

Das Observer Pattern, Beispiel

Ziel, Inhalt

- Wir finden hier Varianten für das Observer Pattern, ausgehend von einer einfachen Implementation

Abend 11 Das Observer Pattern, Beispiel	1
Ziel, Inhalt	1
Beispiele zum Observer Pattern	2
Variante 1, Pro Observer ein Subject	2
Subject	2
Observer	2
SubjectImpl	3
ObserverImpl	4
Variante 2, Mehrere Subjects pro Observer	6
Subject	6
SubjectImpl	6
Object	6
ObjectImpl	7
Test Variante 2	8
Variante 3, Gesteuerte Updates	9
Subject	9
SubjectImpl	10
Observer	11
ObserverImpl	11

Beispiele zum Observer Pattern

Variante 1, Pro Observer ein Subject

Diese erste Version lässt pro Observer nur ein Subject zu. Das heisst ein Observer kann nur ein Subject beobachten.

Das Subject benachrichtigt alle Observer sobald sich die Daten ändern.

Der Observer erhält eine Referenz auf das Subject als Konstruktors-Argument.

Du findest die Daten hier zum Download:

<http://www.devmentor.ch/teaching/additional/001/Semester5/Abend11/ObserverPattern1.zip>

Subject

```
#ifndef SUBJECT_H
#define SUBJECT_H

class Observer;

typedef int Data;

class Subject
{
public:
    virtual ~Subject() {}

    virtual void subscribe(Observer* observer) = 0;
    virtual void unsubscribe(Observer* observer) = 0;

    virtual const Data& getData() const = 0;
    virtual void setData(const Data& newData) = 0;
};

#endif
```

Observer

```
#ifndef OBSERVER_H
#define OBSERVER_H

class Observer
{
public:
    virtual ~Observer() {}

    virtual void update(Subject& subjectThatChanged) = 0;
};

#endif
```

SubjectImpl

HIER DIE KLASSENDEKLARATION AUS DER HEADERDATEI

```
#ifndef SUBJECTIMPL_H
#define SUBJECTIMPL_H

#pragma warning (disable : 4786)

#include "Subject.h"
#include <set>

typedef std::set<Observer*> Observers;

class SubjectImpl : public Subject
{
public:
    SubjectImpl();
    virtual ~SubjectImpl();

    // interface Subject
    virtual void subscribe(Observer* observer);
    virtual void unsubscribe(Observer* observer);

    virtual const Data& getData() const;
    virtual void setData(const Data& newData);

private:
    Data _data;
    Observers _observers;
};

#endif
```

HIER DIE KLASSENDEFINITION AUS DER CPP DATEI

```
#include "SubjectImpl.h"
#include "Observer.h"
#include <iostream>

SubjectImpl::SubjectImpl()
    :_data(0)
{
    std::cout << "A Subject has been created" << std::endl;
}

SubjectImpl::~SubjectImpl()
{
    std::cout << "A Subject has been destroyed" << std::endl;
}

void SubjectImpl::subscribe(Observer* observer)
{
    _observers.insert(observer);
}
```

```
void SubjectImpl::unsubscribe(Observer* observer)
{
    _observers.erase(observer);
}

const Data& SubjectImpl::getData() const
{
    return _data;
}

void SubjectImpl::setData(const Data& newData)
{
    _data = newData;
    Observers::iterator it = _observers.begin();
    Observers::iterator end = _observers.end();

    for( ; it != end; ++it)
    {
        Observer* p0 = *it;
        p0->update();
    }
}
```

ObserverImpl

HIER ZWEI MÖGLICHE BEISPIELE FÜR OBSERVER-IMPLEMENTATIONEN

```
#ifndef OBSERVERIMPL_H
#define OBSERVERIMPL_H

#include "Observer.h"

class Subject;

class Observer1 : public Observer
{
public:
    Observer1(Subject& theSubject);
    virtual ~Observer1();

    virtual void update();

private:
    Subject& _subject;
};

class Observer2 : public Observer
{
public:
    Observer2(Subject& theSubject);
    virtual ~Observer2();

    virtual void update();
}
```

```
private:
    Subject& _subject;
};

#endif

UND DIE CPP DATEI:

#include "ObserverImpl.h"
#include <iostream>
#include "Subject.h"

using namespace std;

Observer1::Observer1(Subject& theSubject)
    :_subject(theSubject)
{
    cout << "Observer1 created" << endl;
    _subject.subscribe(this);
}

Observer1::~~Observer1()
{
    _subject.unsubscribe(this);
    cout << "Observer1 destroyed" << endl;
}

void Observer1::update()
{
    Data changedData = _subject.getData();
    cout << "Observer1 Data is now " << changedData << endl;
}

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

Observer2::Observer2(Subject& theSubject)
    :_subject(theSubject)
{
    cout << "Observer2 created" << endl;
    _subject.subscribe(this);
}

Observer2::~~Observer2()
{
    _subject.unsubscribe(this);
    cout << "Observer2 destroyed" << endl;
}

void Observer2::update()
{
    Data changedData = _subject.getData();
    cout << "Observer2 : " << changedData << endl;
}
```

Variante 2, Mehrere Subjects pro Observer

Hier kann ein Observer mehrere Subjects beobachten. Der Observer merkt sich nun die Subjects in einem Array. Er hält sich Zeiger auf die Objekte die er beobachtet. Beachte die Verwendung des `std::find` Algorithmus.

Das Subject gibt jetzt jeweils einen `this`-Zeiger mit, damit der Observer herausfinden kann welches Subject sich da meldet. Hier der Download:

<http://www.devmentor.ch/teaching/additional/001/Semester5/Abend11/ObserverPattern2.zip>

Subject

Dieses Interface bleibt gegenüber oben unverändert.

SubjectImpl

Hier ändert sich nur die Methode `setData`.

```
void SubjectImpl::setData(const Data& newData)
{
    _data = newData;
    Observers::iterator it = _observers.begin();
    Observers::iterator end = _observers.end();

    for( ; it != end; ++it)
    {
        Observer* p0 = *it;
        // wir geben mit, dass wir uns
        // ändern
        p0->update(this); // HIER AENDERUNG
    }
}
```

Object

Dieses Interface ändert sich leicht. Da ein Observer mehrere Subjects beobachten kann, gibt es jetzt die Methode `observe`.

```
#ifndef OBSERVER_H
#define OBSERVER_H

class Subject;

class Observer
{
public:
    virtual ~Observer() {}

    virtual void update(Subject* subjectThatChanged) = 0;
    virtual void observe(Subject* subjectToObserve) = 0;
};

#endif
```

ObjectImpl

Diese Klasse merkt sich nun wie gesagt mehrere Subjects in einem vector.

```
#ifndef OBSERVERIMPL_H
#define OBSERVERIMPL_H

#include "Observer.h"
#include <vector>

// Vorwärtsdeklaration
class Subject;

typedef std::vector<Subject*> Subjects;

class ObserverImpl : public Observer
{
public:
    ObserverImpl();
    virtual ~ObserverImpl();

    virtual void update(Subject* theSubjectThatChanged);
    virtual void observe(Subject* theSubjectToObserve);

private:
    Subjects _subjects;
};

#endif
```

Mit dem entsprechenden CPP

```
#include "ObserverImpl.h"
#include <iostream>
#include <algorithm>
#include "Subject.h"

ObserverImpl::ObserverImpl()
{
    std::cout << "ObserverImpl created" << std::endl;
}

ObserverImpl::~~ObserverImpl()
{
    // von allen Subjects "abmelden"
    Subjects::iterator it = _subjects.begin();
    Subjects::iterator end = _subjects.end();

    for( ; it != end; ++it)
    {
        Subject* theSubject = (*it);
        theSubject->unsubscribe(this);
    }
    std::cout << "ObserverImpl destroyed" << std::endl;
}
```

```
void ObserverImpl::update(Subject* theSubjectThatChanged)
{
    // finden wir heraus welches Subject uns benachrichtigt
    Subjects::iterator it = std::find(_subjects.begin(),
                                     _subjects.end(),
                                     theSubjectThatChanged);
    int index = it - _subjects.begin();

    const Data& changedData = theSubjectThatChanged->getData();
    std::cout << "Data of Subject " << index;
    std::cout << " is now " << changedData << std::endl;
}

void ObserverImpl::observe(Subject* theSubjectToObserve)
{
    theSubjectToObserve->subscribe(this);
    _subjects.push_back(theSubjectToObserve);
}
```

Test Variante 2

```
#include "ObserverImpl.h"
#include "SubjectImpl.h"
#include <vector>

int main()
{
    SubjectImpl subject1;
    SubjectImpl subject2;

    ObserverImpl observer1;

    observer1.observe(&subject1);
    observer1.observe(&subject2);

    for(int i = 0; i < 4; ++i)
    {
        Data newData = i;
        subject1.setData(newData);
        subject2.setData(newData + 10);
    }

    return 0;
}
```

Variante 3, Gesteuerte Updates

Hier geht es darum, dass die Observer nicht automatisch über Änderungen benachrichtigt werden. Stattdessen bietet das Subject eine spezielle Methode an, damit jemand, der eine oder mehrere Änderungen macht, die anderen Observer benachrichtigen lässt. Da solche Änderungen häufig von anderen Observers erzeugt werden, kann der Observer einen Zeiger auf sich selber mitgeben, damit er merkt wenn ein Änderungsbenachrichtigung von ihm selbst stammt.

Nehmen wir wir hätten zwei Observer : eine Zelle in einer Tabellenkalkulation und ein Balken in einer Grafik, die beide den gleichen Wert anzeigen, einmal numerisch und ein andermal grafisch. Ändert nun der Benutzer die Tabellenzelle, zeigt diese bereits den richtigen Wert an. Die Tabellenzelle ruft trotzdem die Methode *updateObservers* auf, damit auch die Grafik neu gezeichnet wird, sie kann aber den *update*-Aufruf ignorieren. Die MFC bietet diese Möglichkeit an. Hier der Download:

<http://www.devmentor.ch/teaching/additional/001/Semester5/Abend11/ObserverPattern3.zip>

Subject

Hier gibt es die neue Methode *updateObservers*.

```
#ifndef SUBJECT_H
#define SUBJECT_H

class Observer;

typedef int Data;

class Subject
{
public:
    virtual ~Subject() {}

    virtual void subscribe(Observer* observer) = 0;
    virtual void unsubscribe(Observer* observer) = 0;

    virtual void updateObservers(Observer* observer) = 0;

    virtual const Data& getData() const = 0;
    virtual void setData(const Data& newData) = 0;
};

#endif
```

SubjectImpl

```
#ifndef SUBJECTIMPL_H
#define SUBJECTIMPL_H

#pragma warning (disable : 4786)

#include "Subject.h"
#include <set>

typedef std::set<Observer*> Observers;

class SubjectImpl : public Subject
{
public:
    SubjectImpl();
    virtual ~SubjectImpl();

    // interface Subject
    virtual void subscribe(Observer* observer);
    virtual void unsubscribe(Observer* observer);

    virtual void updateObservers(Observer* observer);

    virtual const Data& getData() const;
    virtual void setData(const Data& newData);

private:
    Data _data;
    Observers _observers;
};

#endif
```

Hier der Ausschnitt aus der cpp-Datei mir den geänderten Methoden

```
void SubjectImpl::setData(const Data& newData)
{
    _data = newData;
}
void SubjectImpl::updateObservers(Observer* observer)
{
    Observers::iterator it = _observers.begin();
    Observers::iterator end = _observers.end();

    for( ; it != end; ++it)
    {
        Observer* p0 = *it;
        // wir geben mit, dass wir uns
        // ändern und wer die Aenderung
        // veranlasst
        p0->update(this, observer);
    }
}
```

Observer

Hier das Observer-Interface mit der geänderten update-Methode

```
#ifndef OBSERVER_H
#define OBSERVER_H

class Subject;

class Observer
{
public:
    virtual ~Observer() {}

    virtual void update(Subject* subjectThatChanged,
                        Observer* observerThatUpdates) = 0;
    virtual void observe(Subject* subjectToObserve) = 0;
};

#endif
```

ObserverImpl

Es genügt hier wohl der Ausschnitt aus der .cpp-Datei.

```
void ObserverImpl::update(Subject* theSubjectThatChanged,
                          Observer* observerThatUpdates)
{
    if(observerThatUpdates == this)
    {
        // Falls wir selber die
        // Aenderung verursachen interessiert
        // uns die update-Meldung nicht
        return;
    }

    // finden wir heraus welches Subject uns benachrichtigt
    Subjects::iterator it = std::find(_subjects.begin(),
                                      _subjects.end(),
                                      theSubjectThatChanged);
    int index = it - _subjects.begin();

    const Data& changedData = theSubjectThatChanged->getData();
    std::cout << "Data of Subject " << index;
    std::cout << " is now " << changedData << std::endl;
}
```