

Kapitel 10

Event Handling

10.1 MVC

Erinnerung:

Model - View - Controller (Smalltalk-80): Schema zum Implementieren graphischer Benutzeroberflächen

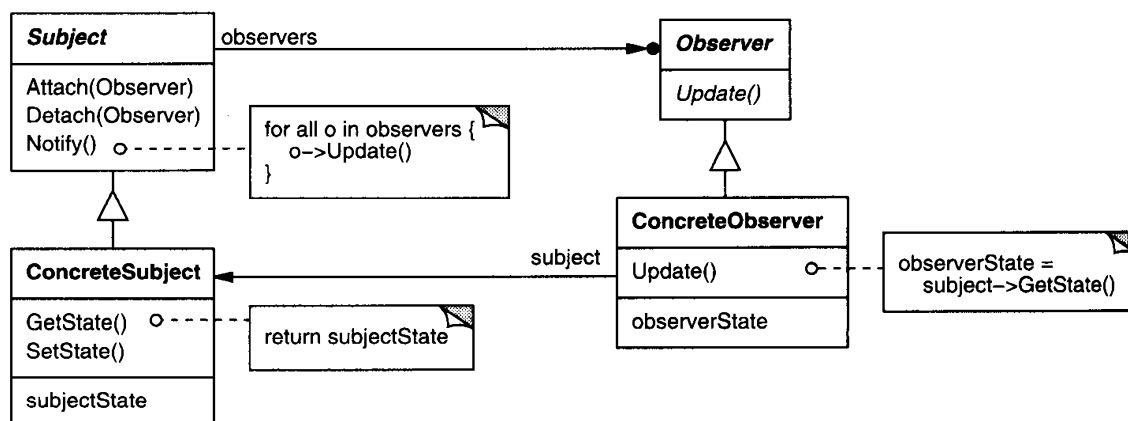
- Model: Klasse, die das funktionale Verhalten der Anwendung realisiert
- View: Klasse, die den aktuellen Zustand des Modells anzeigt (kann anwendungsspezifisch verfeinert werden)
- Controller: Klasse, die Reaktionen auf Benutzereingaben definiert

Vorteil: schwache Kopplung, mehrfache Views zum gleichen Model

Realisierung im wesentlichen durch drei Design Patterns:

- *Observer*: realisiert Kopplung internes Modell – externe Darstellung
- *Composite*: beschreibt Struktur zusammengesetzter, heterogener Objekte (zB Fenster)
- *Strategy*: realisiert Reaktion auf Events

Observer-Pattern:



10.2 Observer in C++

Beispiel Thermometer mit 2 Anzeigen (F° / C°)
und 2 GUI-Events

Standardimplementierung des Observer-Patterns mit Iteratoren (jedoch nur Textausgabe)

```
#include <list>
#include <iostream>
using namespace std;

class Observer {
public:
    virtual void update() = 0;
};

class Observed {
private:
    list<Observer *> observers;
    // all objects which observe this subject
protected:
    void notify() const { // tell all observers
        // that the subject has changed
        for(list<Observer*>::const_iterator o =
            observers.begin(); o!=observers.end(); ++o)
            (*o)->update();
    }
}
```

```
public:
    void add_observer(Observer *observer) {
        observers.push_back(observer);
    }
    void remove_observer(Observer *observer);
};

class Thermometer : public Observed {
private:
    float temperature; /* stored in kelvin */
public:
    Thermometer():temperature(0) {}
    void set_temperature(float new_temperature) {
        temperature=new_temperature;
        notify(); // call update() for all observers
    }
    inline float Thermometer::get_temperature() const {
        return temperature;
    }
};

class ThermometerView : public Observer {
protected:
    Thermometer const &thermometer;

public:
    ThermometerView(Thermometer const &t)
        :thermometer(t) {}
};
```

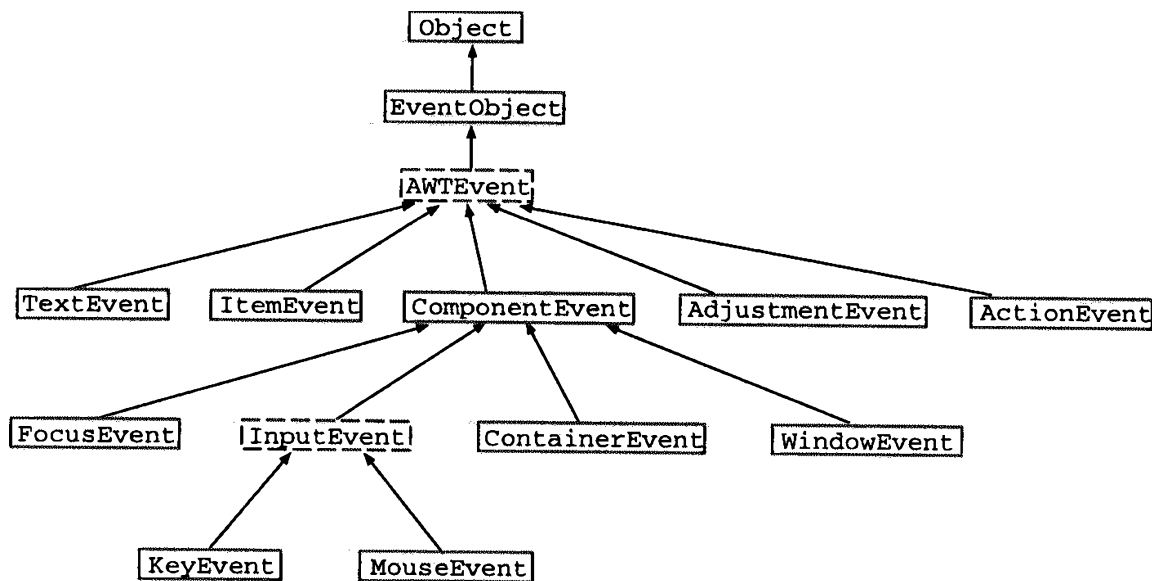
```
class CelsiusView : public ThermometerView {  
public:  
    CelsiusView(Thermometer const &t)  
        :ThermometerView(t) {}  
    void update() {  
        cout << thermometer.get_temperature()-273.3  
            << "C" << endl;  
    }  
};  
  
class FahrenheitView : public ThermometerView {  
public:  
    FahrenheitView(Thermometer const &t)  
        :ThermometerView(t) {}  
    void update() {  
        cout  
        << (thermometer.get_temperature()-273.3)*9/5 +32  
        << "F" << endl;  
    }  
};  
  
class TemperatureSlider { /* Event-Handler 1 */  
private:  
    Thermometer &thermometer;  
public:  
    TemperatureSlider(Thermometer &t):thermometer(t) {}  
    float get_value(); // read value from gui element  
    void value_changed() {  
        // called if slider in gui is moved  
        thermometer.set_temperature(get_value());  
    }  
};
```

```
class ResetButton { /* Event-Handler 2 */
private:
    Thermometer &thermometer;
public:
    ResetButton(Thermometer &t):thermometer(t) {}
    void button_pressed() {
        // called if gui button is pressed
        thermometer.set_temperature(0);
    }
};

main() {
    // create one thermometer, two views, and two controls
    Thermometer t;
    CelsiusView cv(t);
    t.add_observer(&cv);
    FahrenheitView fv(t);
    t.add_observer(&fv);
    TemperatureSlider s(t);
    ResetButton r(t);
    // initialise temperature, create giu events
    t.set_temperature(25);
    r.button_pressed();
}
```

10.3 Events in Java

Event: Ereignis (zB Mausklick auf Button, Keyboardeingabe).
Eventhierarchie:



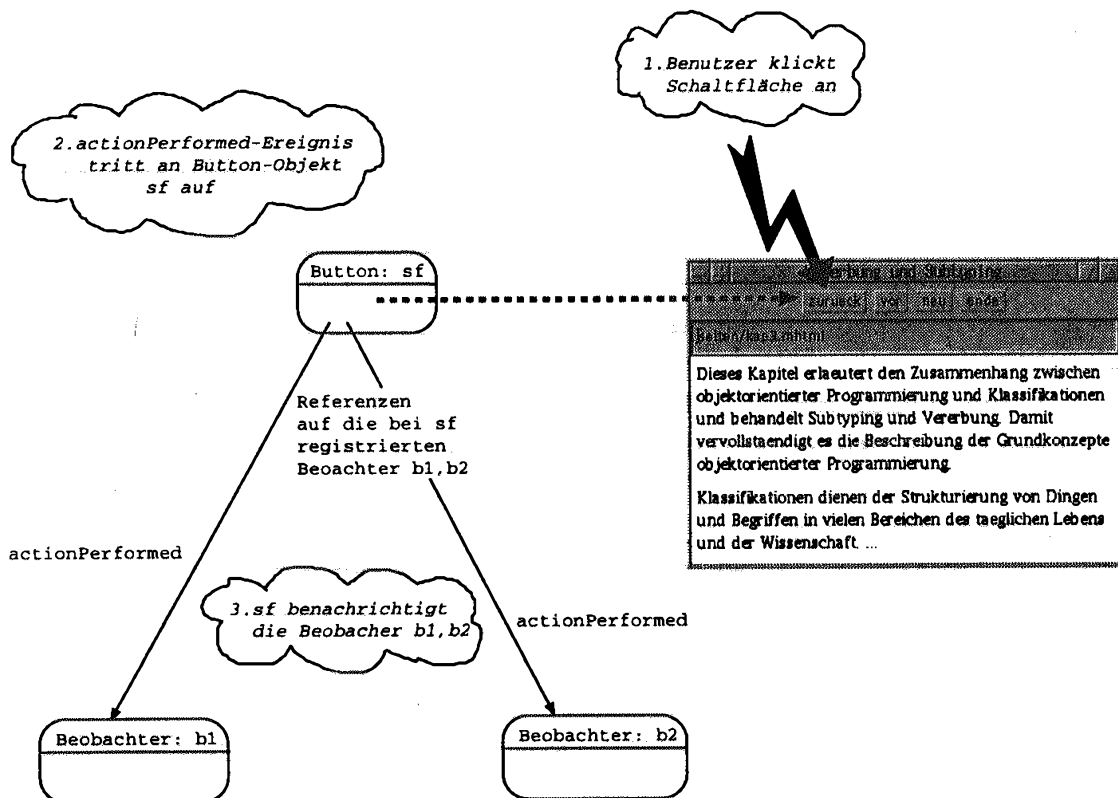
Events:

- **ComponentEvents:** componentMoved, componentHidden, componentResized, componentShown
- **FocusEvents:** focusGained, focusLost
- **KeyEvents:** keyPressed, keyReleased, keyTyped
- **MouseEvent:** mousePressed, mouseReleased, mouseClicked, mouseEntered, mouseExited, mouseDragged, mouseMoved
- **ContainedEvents:** componentAdded, componentRemoved
- **WindowEvents:** windowOpened, windowClosing, windowClosed, windowIconified, windowDeiconified, windowActivated, windowDeactivated
- **ActionEvents:** actionPerformed (Kombination von pressed/released; sehr beliebt)
- **AdjustmentEvents:** adjustmentValueChanged für Scrollbars
- **ItemEvents:** itemStateChanged für Auswahlkomponenten

falls Ereignis stattfindet, wird *Event-Objekt* der entsprechenden Klasse erzeugt, das Eventart (s.o.) sowie Verweis auf Komponente etc enthält

Java bietet direkte Implementierung des Observer-Patterns:
Listener

(selbsterzeugte) Objekte können sich bei Widgets als Listener registrieren lassen. Tritt ein Event am Widget auf, werden alle registrierten Listener benachrichtigt:



Programmcode:

```
class MeinBeobachter implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Knopf gedrueckt");  
    }  
}
```

```
Button sf = new Button();  
MeinBeobachter b = new MeinBeobachter();  
sf.addActionListener(b);
```

bei Ereignis `actionPerformed` wird selbstdefinierte Methode `actionPerformed` aufgerufen (Callback), die Event übergeben bekommt

zu jeder Eventklasse gibt es entsprechendes Listener-Interface. Dieses muss vom Benutzer implementiert werden (Callback-Realisierung). Registrierung von Observern mit `add...Listener`

Korrespondenz zwischen Observern und Widgets kann $n : m$ sein; Widgets können selbst Observer sein

⇒ Interaktion diverser, schwach gekoppelter Objekte, die Aufgaben delegieren indem sie Methoden aufrufen („Nachrichten verschicken“)

Nachteil dieses Konzeptes:

Eventhandler-Methode (`actionPerformed`) ist statisch festgelegt!

10.4 Callbacks in C++

- Verwendung von Function Pointern ermöglicht dynamische Konfiguration der Callback-Funktionen
- jedes Widget hat Callback-Objekt; dieses verwaltet Liste der aufzurufenden Callback-Funktionen nebst Bezugsobjekten
- völlige Unabhängigkeit vom Klienten (i.e. Model + View) durch *Function Pointer* (schwache Kopplung); Flexibilität durch generische Klassen
- generische Klassen und Function Pointer sind in C++ typischer!

```

template<class Param1, class Param2> class callback {
private:
    class instance { /* generic instance, private inner class */
        public:
            virtual void activate(Param1,Param2) = 0;
    };
    /* instance for method call */
    template<class O> class instance_obj : public instance {
    private:
        O *obj;
        void (O::*func)(Param1,Param2);
    public:
        instance_obj(O *xobj,void (O::*xfunc)(Param1,Param2))
            :obj(xobj),func(xfunc) {}
        void activate(Param1 p1, Param2 p2) {
            (obj->*func)(p1,p2);
        }
    }; /* instance_obj */

```

```

// instance for method call + additional data
template<class O, class D>
  class instance_obj_data : public instance {
private:
  O *obj; D data;
  void (O::*func)(Param1,Param2,D);
public:
  instance_obj_data(O *xobj,
    void (O::*xfunc)(Param1,Param2,D), D xdata)
    :obj(xobj),func(xfunc),data(xdata) {}
  void activate(Param1 p1, Param2 p2) {
    (obj->*func)(p1,p2,data);
  }
}; /* instance_obj_data */
/* list of instances registered for this callback */
vector<instance *> instances;
public:
  /* connect method */
  template<class O>
  void connect(O *obj, void (O::*func)(Param1,Param2)) {
    instances.push_back(new instance_obj<O>(obj,func));
  }
  /* connect method which takes additionally data */
  template<class O, class D>
  void connect(O *obj, void (O::*func)(Param1,Param2,D),
    D data) {
    instances.push_back(new instance_obj_data<O,D>
      (obj,func,data));
  }
  /* activate callback */
  void operator ()(Param1 p1, Param2 p2) {
    for(vector<instance *>::iterator c=instances.begin();
      c!=instances.end();++c)
      (*c)->activate(p1,p2);
  }
};

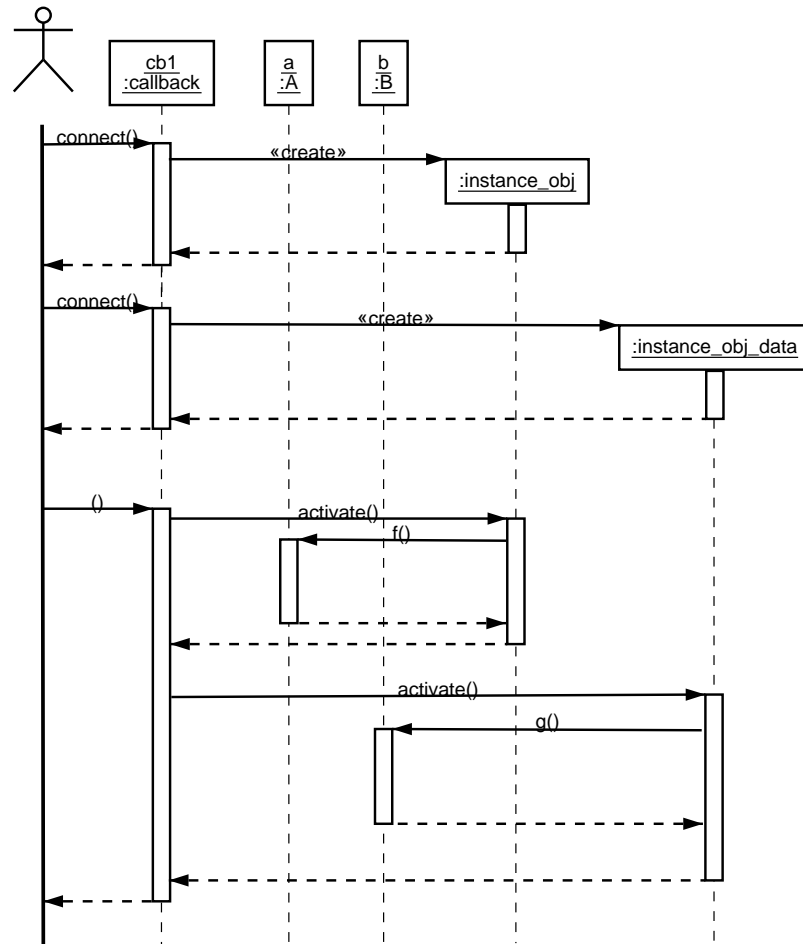
```

```
class A { /*Klient 1, "Model"*/
public:
    void f(int i, string s) {
        cout<<"A::f()"<<i<<" "<<s<<endl;
    }
};

class B { /*Klient 2, "Model"*/
public:
    void g(int i, string s, char const *a) {
        cout<<"B::g() "<<i<<" "<<s<<" "<<a<<endl;
    }
};

void widgetfragment(void) {
    A a; B b;
    callback<int,string> cb;
    cb.connect(&a,&A::f);
    cb.connect(&b,&B::g,"foo");
    cb.connect(&b,&B::g,"bar");
    cb(4,"hello");/* activate callbacks*/
}
```

Sequenzdiagramm: (UML)



Vorteile: 1. Callback-Funktionen können nicht wg Parameter-Typfehler abstürzen (Typsicherheit der Templates und Function Pointer)

2. Ohne Function Pointer müsste Reaktions-Methode des Eventhandlers fest codiert sein (wie in Java); Function Pointer erlauben flexible Controller-Konfiguration und flexible Handler-Reaktionen (totale Entkopplung!)