

.NET: Delegates und Events

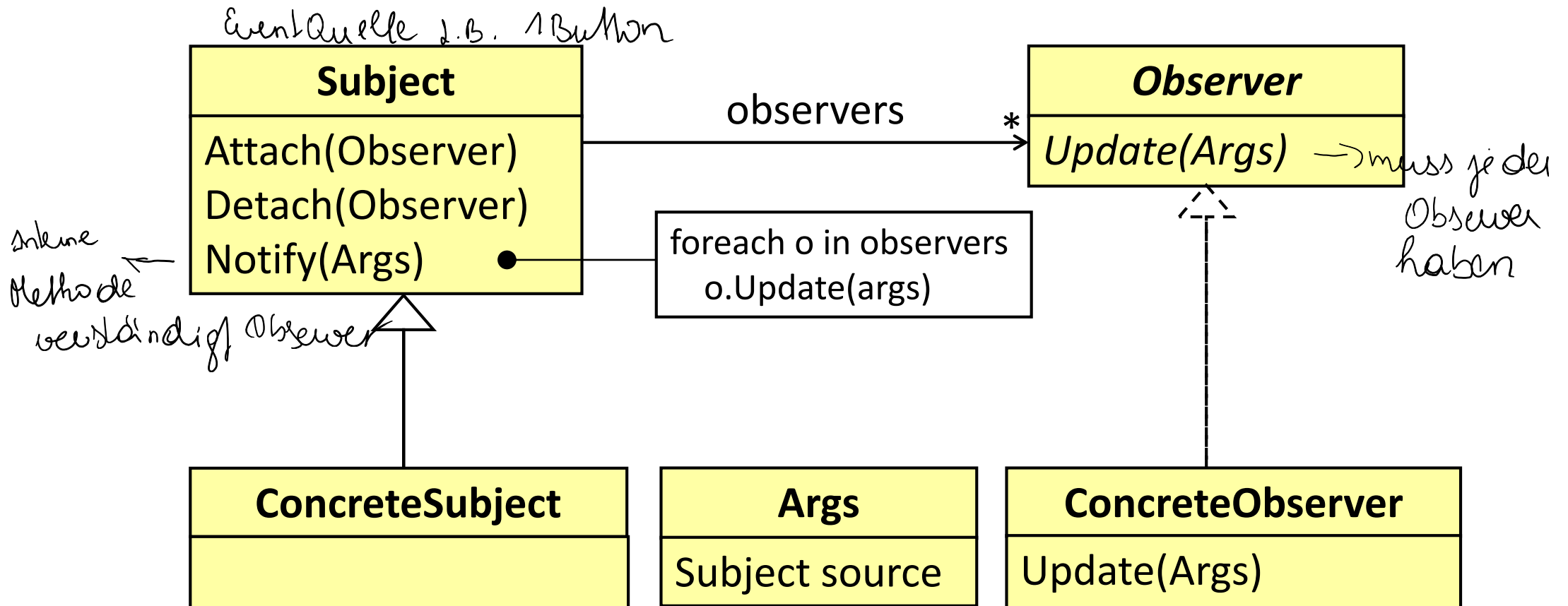
© J. Heinzlreiter
Version 5.2

Motivation

- GUI-Frameworks **sind ereignisorientiert.**
 - Bei Eintreten bestimmter Ereignisse werden **Callback-Funktionen** aufgerufen.
 - Callback-Funktionen werden **erst zur Laufzeit an Ereignisse gebunden.**
 - Man möchte **mehrere Callback-Funktionen** an ein Ereignis binden.
- Funktionale Programmierung
 - Funktionen höherer Ordnung: Funktionen, an **denen Funktionen als Parameter übergeben werden können.**
- Implementierung in **anderen Programmiersprachen**
 - **C++: Funktionszeiger** (sind aber nicht typsicher).
 - **Java: Beobachterobjekte (Listener), Lambda-Ausdrücke** (ab Java 8).

↗
Observer pattern

Das Beobachter-Muster / Publish-Subscribe Pattern



Observer Pattern: Realisierung in Java

functional IF
Basis für
Lambda-Expr.

Subject muss alle Observer verwalten können

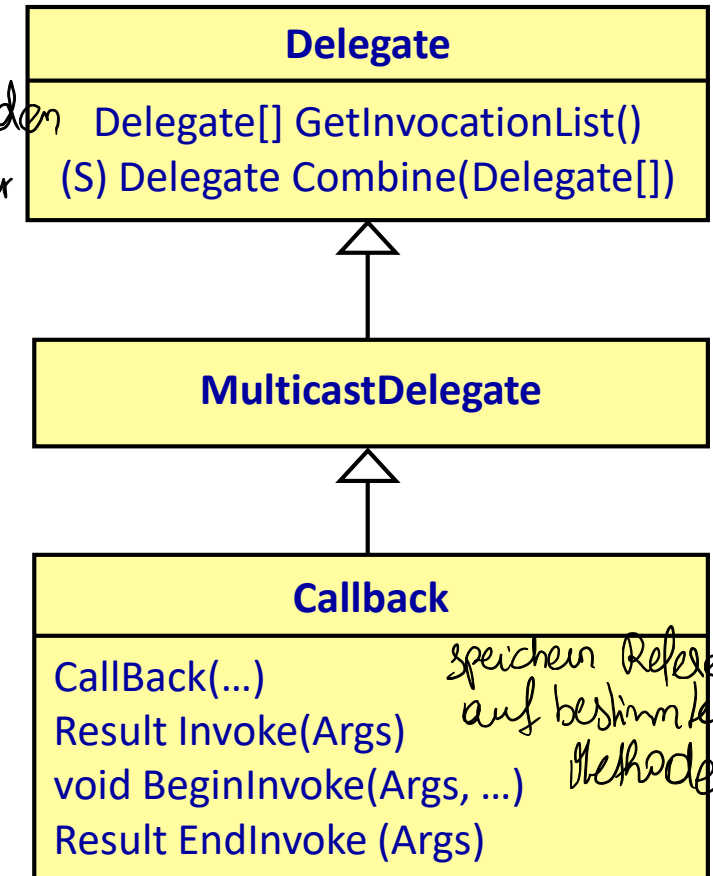
```
class Subject {  
    protected Collection<Observer>  
        observers =  
        new ArrayList<Observer>();  
    public void Attach(Observer o) {  
        observers.add(o);  
    }  
    public void Detach(...) { ... }  
    protected void Notify(Args a) {  
        for (Observer o : observers)  
            o.Update(a);  
    }  
}
```

```
interface Observer {  
    public void Update(Args a);  
}  
class ConcreteObserver  
    implements Observer {  
    Subject subj = new Subject();  
    public ConcreteObserver() {  
        subj.Attach(this);  
    }  
    public void Update(Args a) {  
        ...  
    }  
}
```

muss Observer IF implementieren

Definition von Delegates = nichts anderes als spezieller Datentyp

- Definition eines **neuen Delegate-Typs** (C#-Syntax): *legt Wertebereich fest = alle Methoden mit dieser Signatur*
delegate Result Callback(Args args);
- Definiert eine **neue Klasse Callback**.
- Ist von MulticastDelegate abgeleitet:
Mehrere Methoden können Delegate-Objekt zugewiesen werden.
- Mit Invoke werden **alle mit Delegate assoziierten Methoden aufgerufen.**
- Invoke hat **dieselbe Signatur wie Delegate-Definition.**
- Mit **BeginInvoke, EndInvoke** können assoziierte Methoden **asynchron** aufgerufen werden.



kann auch delegate Variablen / Objekte erzeugen

Verwenden von Delegates

- Deklaration einer Delegate-Referenz

```
Callback callback1 = null;
```

= Referenz auf Typ, also kann es null sein

- Erzeugen eines **Delegate-Objekts**

```
class Observer {  
    public Result Method(Args arg) { ... }  
    public static Result StaticMethod(Args arg) { ... }  
}
```

müssen die Signatur des Typs haben
- Objektmethode
- Klassenmethode

```
Observer o = new Observer();  
callback1 = new Callback(o.Method);  
callback2 = new Callback(Observer.StaticMethod);  
Callback3 = o.Method; // ab C# 2.0
```

← Referenz auf Methode und Objekt gespeichert
brauche hier kein Objekt

- Aufruf der **registrierten Methoden mit ()-Operator**

```
callback1(new Args()); // callback1 muss ungleich null sein.  
callback2(new Args());
```

Multicast-Delegates

- Einem Delegate-Objekt können mehrere Methoden-Referenzen zugewiesen werden.

```
Observer o = new Observer();  
Callback callback = new Callback(o.Method);  
callback += new Callback(Observer.StaticMethod);
```

ein Objekt zeigt nun gleichzeitig
auf zwei Methoden → Zeigerkonzept
nicht mehr vorhanden

- `+=` wird vom Compiler in `Delegate.Combine` übersetzt.

- `callback = Delegate.Combine(callback, Observer.StaticMethod);`

- `()`-Operator ruft alle zugewiesenen Methoden auf.

```
Args a = new Args();  
callback(a) // → o.Method(a);  
            // → Observer.StaticMethod(a);
```

→ könnte theoretisch auch null sein,
Compiler/Combine können damit
umgehen

- Rückgabewerte bei Multicast-Delegates:

- Letzte Delegate-Methode definiert den Rückgabewert.
- Bei Übergangsparametern wird der Wert von Methode zu Methode weitergereicht.

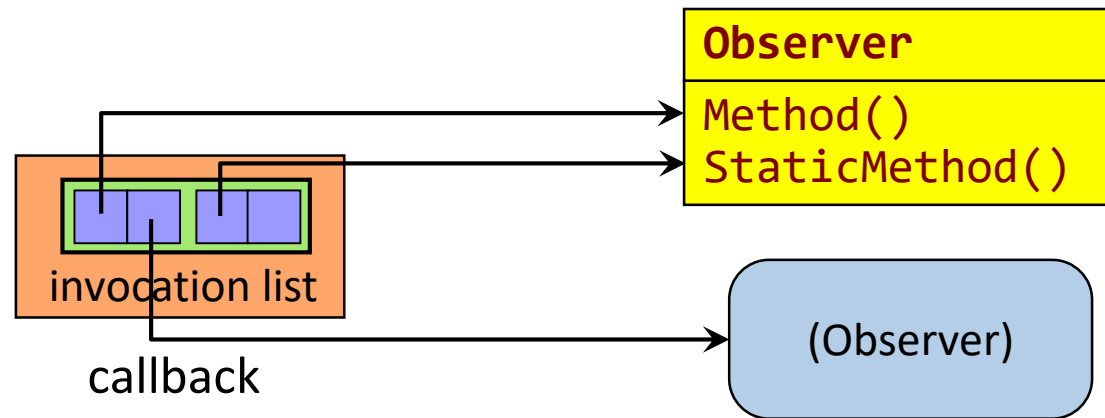
Result r = callback(a); → ginge
auch da
in Signatur
ein Result
ist.

Repräsentation von Delegate-Objekten

```
class Observer {  
    public Result Method(Args arg) { ... }  
    public static Result StaticMethod(Args arg) { ... }  
}
```

```
Observer o = new Observer();  
Callback callback = new Callback(o.Method);  
callback += new Callback(Observer.StaticMethod);
```

callback Objekt zeigt nun
auf 2 Methoden mit der
selben Signatur



Methoden sind in
der invocation list
gespeichert

= zeigt auch auf Objekt, da
das erst eine Objekt
Methode ist, bei der
beiden nicht

Events

- Delegate-Felder können als event deklariert werden:

```
class EventSource {  
    public event Callback OnCallback;  
    void FireEvent() {  
        if (OnCallback != null)  
            OnCallback(new Args());  
    }  
}
```

da öffentlich kann ich mich dafür registrieren
event bedeutet, dass Klasse nur noch
eingeschrieben & verwendet werden kann
= - Operator ist verborgen
Aufruf des Callbacks ebenso nicht mehr
notwendig

- Auswirkungen:

- Event kann nur von einer Methode jener Klasse, in der Event deklariert wurde, ausgelöst werden.
- Von außen können dem Delegate-Feld keine Werte zugewiesen werden.

```
EventSource source = new EventSource();  
source.OnCallback += new Callback(o.Method);
```

```
source.OnCallback = new Callback(...);  
source.OnCallback(new Args());
```

// Syntaxfehler!
// Syntaxfehler!

dadurch geht aller Callback
verloren / überschrieben

Abfrage auf null

- Wird eine Delegate-Variablen keine Methoden-Referenz zugewiesen, ist diese null →
- Bei Ausführung des Delegates ist dies zu berücksichtigen:

```
public event Callback OnCallback;  
void FireEvent() {  
    if (OnCallback != null)  
        OnCallback(new Args());  
}
```

- Mit dem Null-Conditional-Operator lässt sich die Delegate-Ausführung vereinfachen:

```
public event Callback OnCallback;  
void FireEvent() {  
    OnCallback?.Invoke(new Args());  
}
```

Observer Pattern: Realisierung in C#

```
delegate void  
    UpdateCallback(Args a);
```

```
class Subject {  
    public event UpdateCallback  
        Observers;  
    public void Notify(Args a) {  
        Observers?.Invoke(a);  
    }  
}
```

collection
mit Referenzen
(Invocation List)

```
class Observer {  
    public static void  
        StaticUpdate(Args a) {  
        ...  
    }  
    public void Update(Args a) {  
        ...  
    }  
}
```

```
Subject s = new Subject();  
Observer o = new Observer();  
s.Observers += o.Update; // Attach()  
s.Observers += Observer.StaticUpdate;  
s.Notify(new Args());
```

new UpdateCallback(o.Update);
new UpdateCallback(Observer.StaticUpdate);

Anonyme Methoden und Lambda-Asdrücke

- Explizite Implementierung der Methode:

```
public class Timer {  
    private void Run() { Implementierung von Run() }  
    private Thread thread =  
        new Thread(new ThreadStart(this.Run));  
}
```

delegate void ThreadStart();

- Anonyme Methode:

```
public class Timer {  
    private Thread thread =  
        new Thread(  
            delegate() { Implementierung von Run() }));  
}
```

- Lambda-Ausdruck:

```
public class Timer {  
    private Thread thread =  
        new Thread( () => { Implementierung von Run() } );  
}
```

*Datentypen muss ich nicht
angeben in Lambda, macht
Compiler*

Lambda-Ausdrücke

- Methoden mit Delegate-Parametern sind sehr flexibel einsetzbar:

```
public delegate bool Predicate<T>(T obj);  
private static IEnumerable<T> FilterWhere(  
    this IEnumerable<T> numbers, Predicate<T> filter) {  
    foreach (T n in numbers)  
        if (filter(n))  
            yield return n;  
}
```

invoke passiert hier

=> weil Enumeration zurückkommt, also wird rekursiv aufgebaut, erst wenn drüber iteriert wird werden

- An diese Methoden können anonyme Methoden übergeben werden: *Werte ermittelt*

```
var oddNumbers = numbers.FilterWhere(  
    delegate(int n) { return n % 2 != 0; });
```

- Lambda-Ausdrücke ermöglichen eine einfachere Schreibweise für anonyme Methoden:

```
var oddNumbers = numbers.FilterWhere(n => n % 2 != 0);
```

- In LINQ-Ausdrücken werden sehr häufig anonyme Methoden benötigt.

Asynchrone Ausführung

```
delegate bool CalcCallback(double d, out double r);
```



```
class CalcCallback : MultiCastDelegate {  
    // Synchroner Aufruf  
    public bool Invoke(double d, out double r);  
    // Asynchroner Aufruf  
    public IAsyncResult BeginInvoke(double d, out double r,  
        AsyncCallback cb, Object asyncState);  
    // Liefert Ergebnisse der Methode (synchron)  
    public bool EndInvoke(out double r, IAsyncResult ar);  
}
```

```
public Interface IAsyncResult {  
    bool IsCompleted { get; } // Asynchroner Aufruf fertig?  
    object AsyncState { get; } // Letztes Argument von BeginInvoke()  
}  
public delegate void AsyncCallback(IAsyncResult ar);
```

Asynchroner Aufruf mit Polling

```
delegate bool CalcCallback(double d, out double r);
static bool Sqrt(double d, out double r) {
    r = d >= 0 ? Math.Sqrt(d) : 0;
    return d >= 0;
}

void DoIt() {
    // Generierung eines Delegate-Objekts.
    CalcCallback calc = new CalcCallback(Sqrt);
    // Asynchroner Aufruf von Sqrt.
    IAsyncResult ar = calc.BeginInvoke(2, out r, null, null);
    // Warte bis Sqrt terminiert.
    while (!ar.IsCompleted) {
        // Dieser Code wird parallel zu Sqrt ausgeführt.
        Thread.Sleep(0);
    }
    // Abholen der Ergebnisse des anynchronen Methodenaufrufs.
    bool ok = calc.EndInvoke(out r, ar);
}
```

Asynchroner Aufruf mit Callback

```
delegate bool CalcCallback(double d, out double r);
CalcCallback calc;
// Callback-Methode, die aufgerufen wird, wenn Sqrt terminiert.
void CalcCompleted(IAsyncResult ar) {
    double r = 0;
    // Abholen der Ergebnisse des anynchronen Methodenaufrufs.
    bool ok = calc.EndInvoke(out r, ar);
}
void DoIt() {
    double r;
    // Generierung eines Callbacks, das bei Termination von Sqrt
    // aufgerufen wird.
    AsyncCallback cb = new AsyncCallback(CalcCompleted);
    // Asynchroner Aufruf von Sqrt.
    calc = new CalcCallback(Sqrt);
    calc.BeginInvoke(5, out r, cb, null);
}
```