

# Ausgewählte Entwurfsmuster

## Vergleich der Implementierung in Smalltalk, Java und ObjectPascal

# Bekannte Entwurfsmuster

- Singleton
- Beobachter
- Zustand
- Fassade
- Abstrakte Fabrik

# Muster (Allgemein)

„Vorlage, Modell; Vorbild, Vollkommenes in seiner Art; [..] einzelnes Stück zur Ansicht, zur Auswahl [..]“  
Wahrig Deutsches Wörterbuch

„Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.“  
Christopher Alexander

# Struktur

- Entwurfsmuster
- Singleton
  - Allgemeine Darstellung
  - Implementierungsaspekte in Java, ObjectPascal und Smalltalk
- Vermittler
  - Allgemeine Darstellung
  - Implementierungsaspekte in Java, Smalltalk und ObjectPascal

# Quellen

- E. Gamma, R. Helm, R. Johnson, J. Vlissides.  
Design Patterns, Addison-Wesley, 1994
- M. Fowler. Analysis Patterns, Reusable Object  
Models, Addison-Wesley, 2000
- A. Bien, J2EE Patterns, Entwurfsmuster für die  
J2EE, Addison-Wesley, 2002
- Hillside Group, <http://hillside.net>
- Google, [search?q=design+patterns](http://www.google.com/search?q=design+patterns)

# Entwurfsmuster (nach GoF)

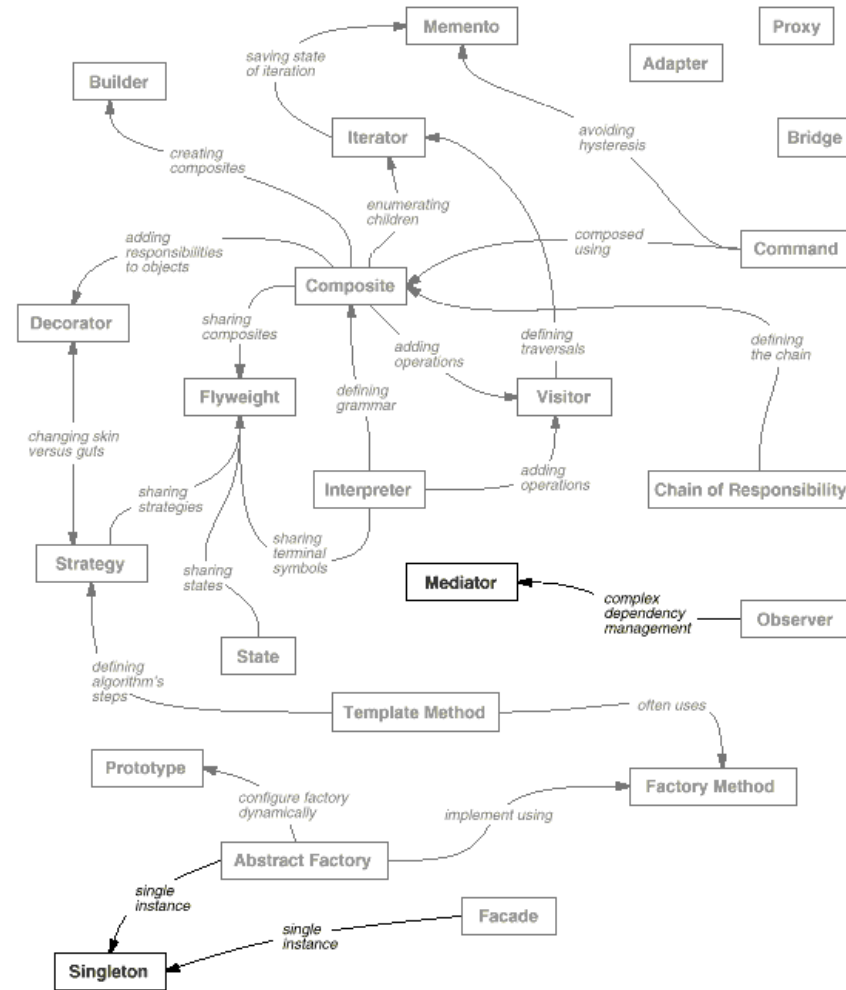
- Mustername
- Problemabschnitt
- Lösungsabschnitt
- Konsequenzenabschnitt

Kein Muster jedoch

ADTs (einfach) oder Architekturen  
(komplex)

Zusammensetzung anderer Muster (zB MVC)

# Dargestellte Entwurfsmuster



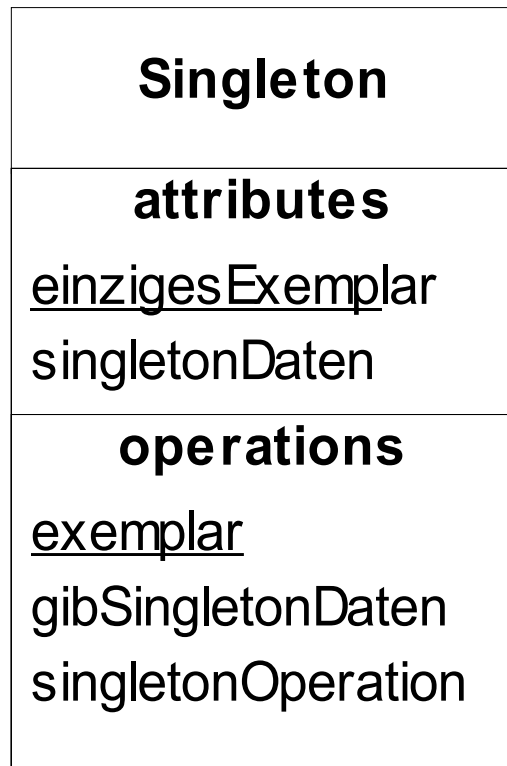
# Singleton

- Sichert ab, dass eine Klasse genau ein Exemplar besitzt
- Stellt globalen Zugriffspunkt darauf bereit

## **Verwendung**

- Druckerspooler, Dateisystem, Logging, Datenbank, etc.

# Singleton (Struktur)



initialisiere ggf einziges Exemplar  
return einziges Exemplar

# Singleton (Konsequenzen)

- Zugriffskontrolle auf das Exemplar
- Verfeinerung von Funktionalität
- Variable Anzahl von Exemplaren
- Flexibler als Klassenmethoden

# Singleton (Java)

- Garantie des einzigen Exemplars durch Klassenvariable
- Privater Konstruktor verhindert Exemplarbildung durch Klienten
- Lazy-Initialization ermöglicht Konfiguration

# Singleton (Java Implementierung)

```
public class Singleton {  
    private static Singleton einzigesExemplar;  
    private Singleton(){  
        // nicht sichtbar für Klienten  
    }  
    public static Singleton exemplar(){  
        // ggf einziges Exemplar erzeugen  
        if(einzigesExemplar==null)  
            einzigesExemplar = new Singleton();  
  
        return einzigesExemplar;  
    }  
}
```

# Singleton (Java Klient)

```
public class SingletonTest {  
  
    public static void main(String[] args) {  
        // s1 = new Singleton() nicht möglich  
  
        // erzeugung des einzigen Objekts  
        Singleton s1 = Singleton.exemplar();  
  
        // bereits erzeugtes Objekt zurückgeben  
        Singleton s2 = Singleton.exemplar();  
  
        // es gilt: s1.equals(s2) && (s1==s2)  
    }  
}
```

# Exkurs: Objekte in ObjectPascal

- Typenstrenge Überprüfung zur Übersetzung

```
var
```

```
    meinExemplar: TMeineKlasse;
```

```
begin
```

```
    meinExemplar := TMeinNachfahre.Create;
```

- Speicherverantwortlichkeit beim Entwickler

```
Objekterzeugung;
```

```
try
```

```
    Objektverwendung;
```

```
finally
```

```
    Objektfreigabe;
```

```
end;
```

# Singleton (ObjectPascal)

- Verwaltung des einzigen Exemplars durch globale Variable
- Referenzzählung notwendig

## **Achtung:**

- Privater Konstruktor kann Exemplarbildung vom Klienten nicht verhindern
- Freigabe durch Klienten möglich

# Singleton (ObjectPascal Impl.)

**interface**

**type**

TSingleton = **class**

**private**

**constructor** Create;

**public**

**class function** Exemplar: TSingleton;

**procedure** GibFrei;

**end;**

**implementation**

**constructor** TSingleton.Create;

**begin**

*// nicht sichtbar für Klienten*

*// (verwendet TObject.Create)*

**end;**

**var**

FSingleton: TSingleton;

FSingletonReferenzen: Integer;

**class function** TSingleton.Exemplar:  
TSingleton;

**begin**

**if not** Assigned(FSingleton) **then**  
FSingleton := TSingleton.Create;

Inc(FSingletonReferenzen);

Result := FSingleton;

**end;**

**procedure** TSingleton.GibFrei;

**begin**

Dec(FSingletonReferenzen);

**if** FSingletonReferenzen<=0 **then**  
FreeAndNil(FSingleton);

**end;**

# Singleton (ObjectPascal Klient)

```
var
  s1, s2: TSingleton;
begin
  // s1 := TSingleton.Create möglich aber fehlerhaft

  // erzeugung des einzigen Objekts
  s1 := TSingleton.Exemplar;
  try
    // bereits erzeugtes Objekt zurückgeben
    s2 := TSingleton.Exemplar;
    try
      // es gilt: s1=s2
    finally
      // s2.Free möglich aber fehlerhaft
      s2.GibFrei;
    end;
  finally
    s1.GibFrei;
  end;
end.
```

# Singleton (ObjectPascal Erweitert)

## **Besseres Design möglich**

- Transparente Referenzzählung über `NewInstance` und `FreeInstance`
- Ressourcenverwaltung durch implizite Referenzzählung
- Zuweisbare lokale Konstante statt globaler Variablen

# Exkurs: Objekte in Smalltalk

- **Erzeugung durch Nachricht an Klasse**

```
meinObjekt := MeineKlasse new.  
meineMenge := ArrayedCollection with: 5 with: 6.  
meinString := String readFrom: aStream.
```

- **Objekt durch speziellen Konstruktor**

```
meineMenge := #(1, 2, 3).  
meinString := 'multiply 6 by 9'.
```

- **Erzeugung durch Nachricht an Objekt**

```
meinPunkt := 23 @ 137.  
meineSumme := 6 * 9.
```

# Singleton (Smalltalk)

- Garantie des einzigen Exemplars durch Klassenvariable
- Erzeugung über `new` verursacht Fehler und verhindert Erzeugung weitere Exemplare
- Vorkonfiguration des Exemplars im Image möglich

# Singleton (Smalltalk Implementierung)

```
Smalltalk.Examples defineClass: #Singleton
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: ''
  classInstanceVariableNames: 'einzigesExemplar '
  imports: ''
  category: 'Examples'
```

## Klassenmethoden für „instance creation“

### **new**

```
self error: 'Zur Erzeugung Nachricht exemplar an Klasse schicken,
```

### **exemplar**

```
einzigesExemplar isNil ifTrue:[einzigesExemplar := super new].
^einzigesExemplar
```

# Singleton (Smalltalk Erweiterung)

## Sogar transparente Erzeugung denkbar:

```
new  
    einzigesExemplar isNil ifTrue:[einzigesExemplar := super new].  
    ^einzigesExemplar
```

## Aber: potenzielle Fehlerquelle:

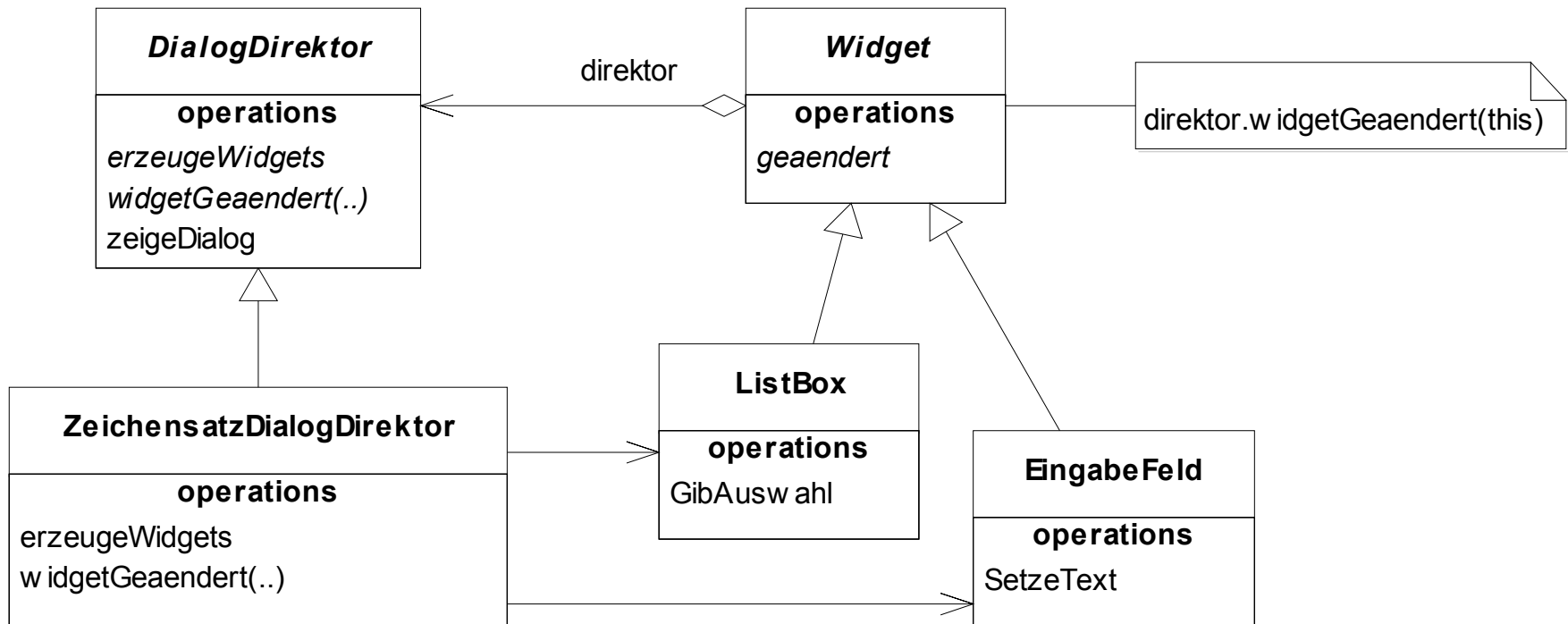
```
| left right |  
left := SingleWindow new.  
Right := SingleWindow new.  
left position: (100 @ 100).  
right position: (500 @ 100).
```

# Vermittler

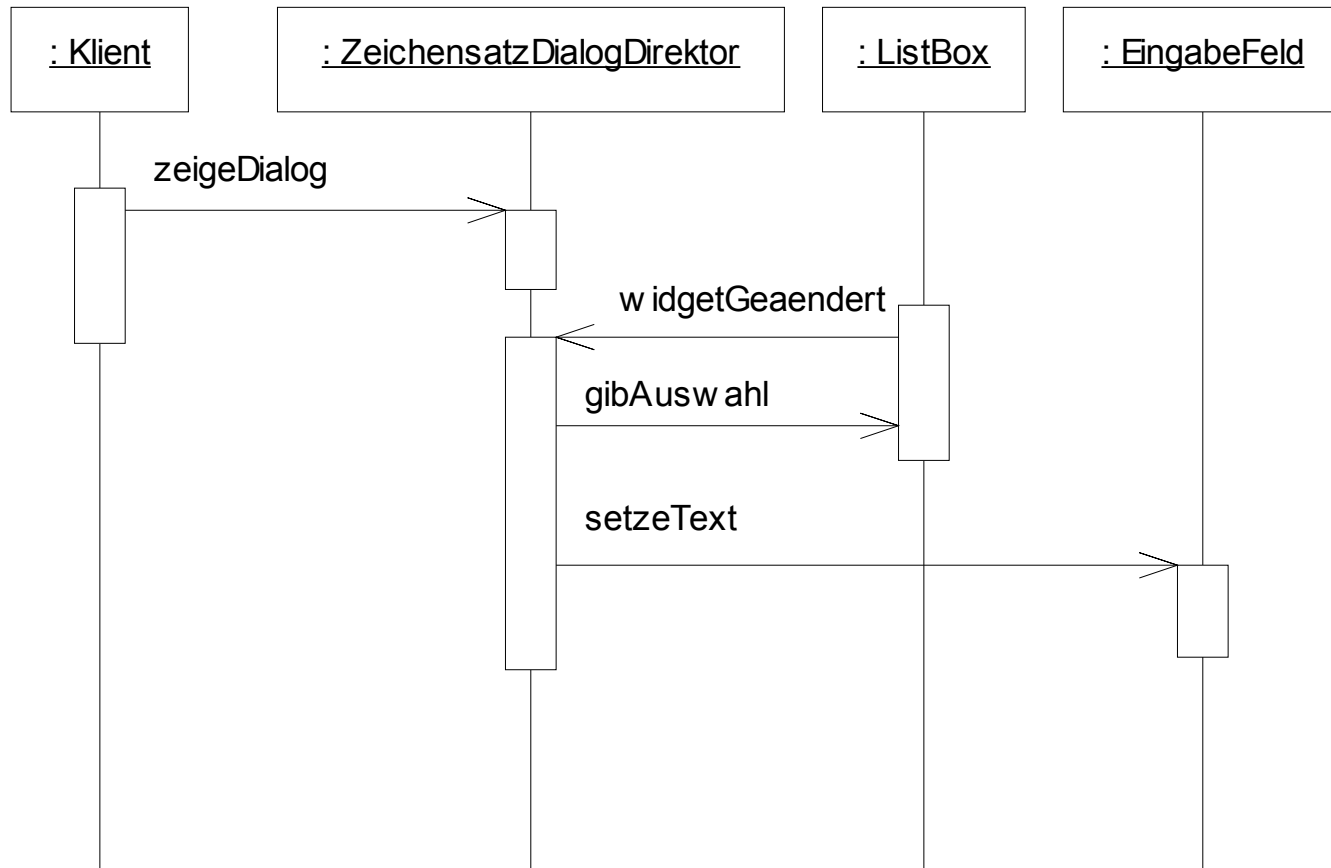
- Definiert Objekt, das Zusammenspiel von Objekten kapselt
- Fördert lose Kopplung, durch Vermeidung gegenseitiger Bezugnahme von Objekten
- Ermöglicht Zusammenspiel von Objekten zu variieren, ohne Objekte selbst zu verändern

Häufig eingesetzt in Dialogsystemen

# Vermittler (Struktur)



# Vermittler (Ablauf)



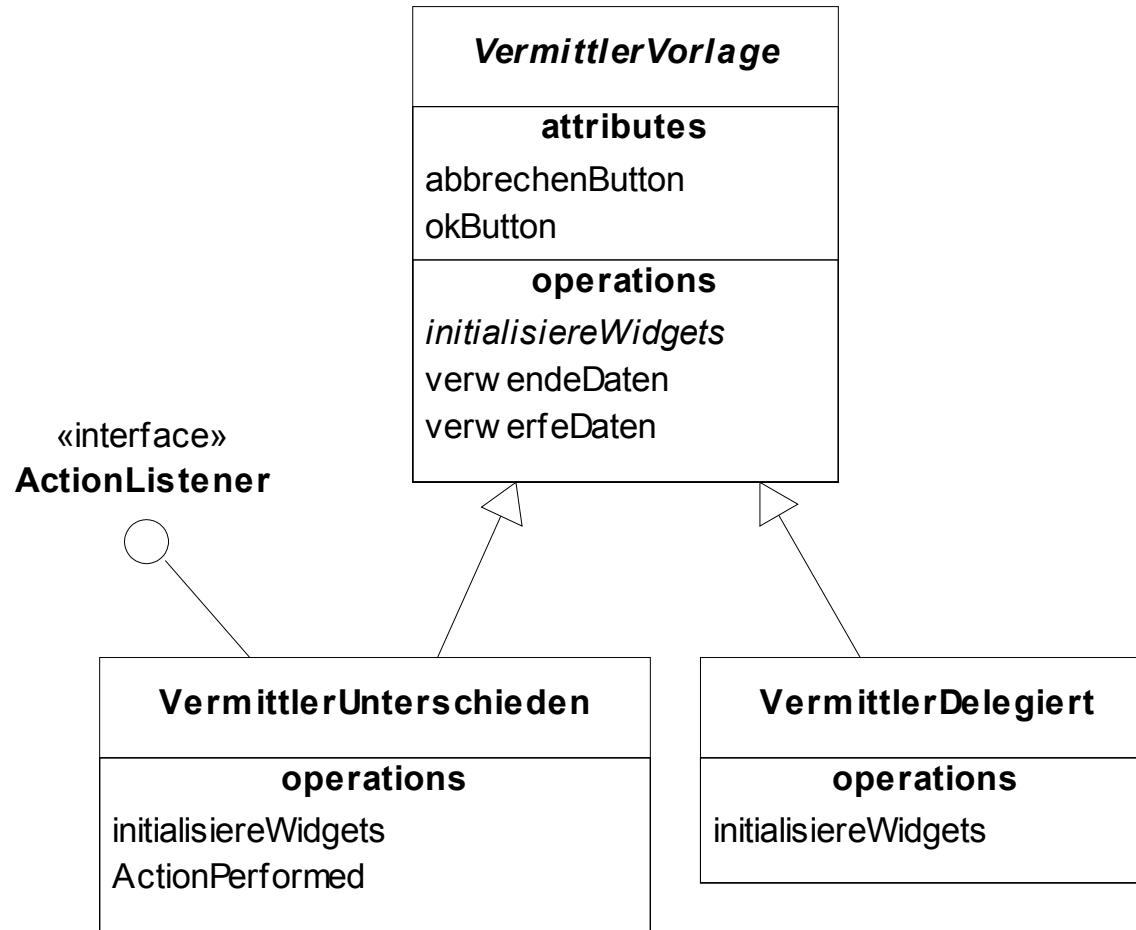
# Vermittler (Konsequenzen)

- Schränkt Unterklassenbildung ein
- Entkoppelt Kollegenobjekte
- Vereinfacht Protokoll der Objekte
- Abstrahiert von Art der Zusammenarbeit
  
- Zentralisiert Steuerung

# Vermittler (Java)

- Ereignisorientiert
- Widgets unterstützen Beobachtermuster
- Interfaces für Verbund verantwortlich  
→ genau eine Methode in Klasse pro Rolle
- Zusätzliche Informationen beim Widget hinterlegbar, um Sender zu identifizieren

# Vermittler (Java Beispielstruktur)



# Vermittler (Java Fallunterscheidung)

```
public class VermittlerUnterschieden extends VermittlerVorlage
    implements ActionListener{

    protected void initialisiereWidgets() {
        okButton.addActionListener(this);
        abbrechenButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent arg0) {
        if(arg0.getSource()==okButton)
            verwendeDaten();
        else if (arg0.getSource()==abbrechenButton)
            verwerfeDaten();
    }
}
```

# Vermittler (Java Delegation)

```
public class VermittlerDelegiert extends VermittlerVorlage{

    protected void initialisiereWidgets() {
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                verwendeDaten();
            }
        });

        abbrechenButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                verwerfeDaten();
            }
        });
    }
}
```

# Vermittler (Smalltalk)

- Nachrichten(-Sendungen) sind Objekte
- Selektion über Symbol bei Nachricht  
`perform: aSelector with: anArgument`
- Registratur in `ActionList` über Tripel  
`when: anEvent send: aSelector to: anObject`
- Versand von `MessageSend` im Ereignisfall  
`triggerEvent: anEvent`

## Alternative: Funktions Closures

# Vermittler (Smalltalk Beispiel)

## Smalltalk/V

```
self addSubPane: (ListPane new
  paneName: 'meineListPane';
  owner: self;
  when: #select send: #listSelect: to: self).
```

## Smalltalk/X

```
| meinView |
meinView := StandardSystemView new.
```

```
Button label: 'OK'
```

```
  action: [self verwendeDaten] in: meinView.
```

```
Button label: 'Abbrechen'
```

```
  action: [self verwerfeDaten] in: meinView.
```

# Vermittler (ObjectPascal)

- Vermittler als Standard zur Dialoggestaltung

- Referenz auf Methode eigener Datentyp

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

- Verwendung als normale Variable

```
var
```

```
    meinEvent, anderesEvent: TNotifyEvent;
```

```
begin
```

```
    // notwendig: TDieseKlasse.ButtonClick (Sender: TObject) ;
```

```
    meinEvent := Self.ButtonClick;
```

```
    anderesEvent := meinEvent;
```

```
    // entspricht Self.ButtonClick (AButton) ;
```

```
    anderesEvent (AButton) ;
```

# Vermittler (ObjectPascal Beispiel)

**begin**

**with** TButton.Create(Self) **do**

**begin**

    Caption := 'OK';

    Parent := Self;

    OnClick := VerwendeDaten;

**end;**

**with** TButton.Create(Self) **do**

**begin**

    Caption := 'Abbrechen';

    Parent := Self;

    OnClick := VerwerfeDaten;

**end;**