ASPECT-ORIENTED

DESIGN PATTERNS

Seminar: Aspektorientierte Systemprogrammierung
Thema: Aspect-Oriented Design Patterns
Referent: Sand Rainer

Gliederung

I. Entwurfsmuster – Aufbau und Beschreibung

II. Probleme objektorientierter Entwurfsmuster

III. Entwicklung eines aspektorientierten Musters

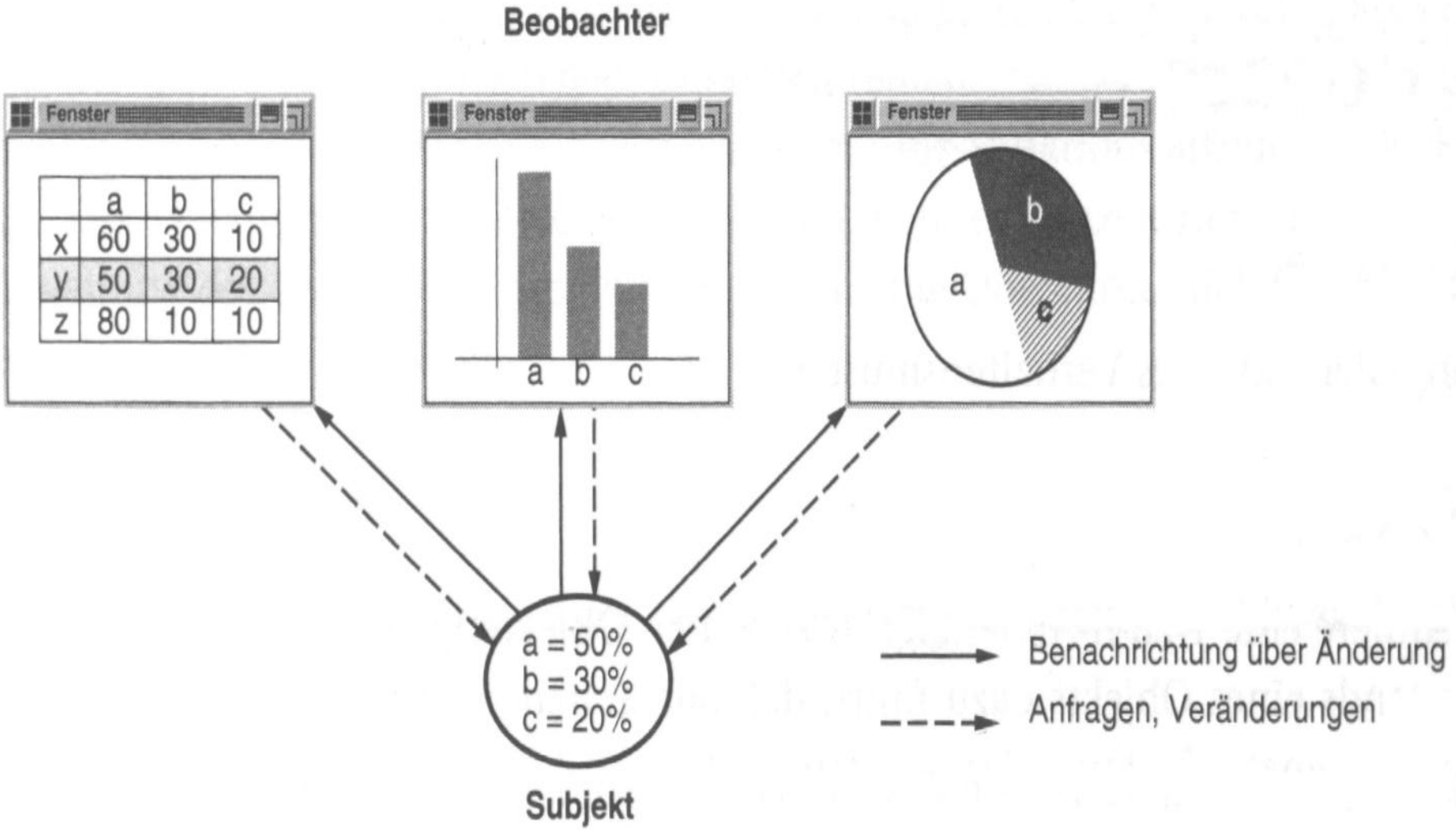
Abschnitt I:

Entwurfsmuster – Aufbau und Beschreibung

Was sind Entwurfsmuster ?

- Bieten Lösungen für Standardprobleme
- Entwicklung wiederverwendbarer Software
- Muster sind informell beschrieben
- Abstraktionsebene: System mit interagierenden Objekten bzw. Klassen
- Beispiel für „guten objektorientierten Entwurf“

Beispiel: Der „Observer“



I.

Beschreibung eines Entwurfsmusters

- Mustername: beschreibt knapp den Gehalt
- Zweck / Motivation: Beispiele verschiedener Bereiche
- Problemabschnitt: Kontext / Anwendungsgebiet
- Lösungsabschnitt: Struktur und Teilnehmer
- Interaktionen: Interaktionsdiagramm
- Konsequenzabschnitt: entstehende Vor-/ Nachteile
- Implementierung: Diskussion und Beispielcode
- Bekannte Verwendungen & Verwandte Muster

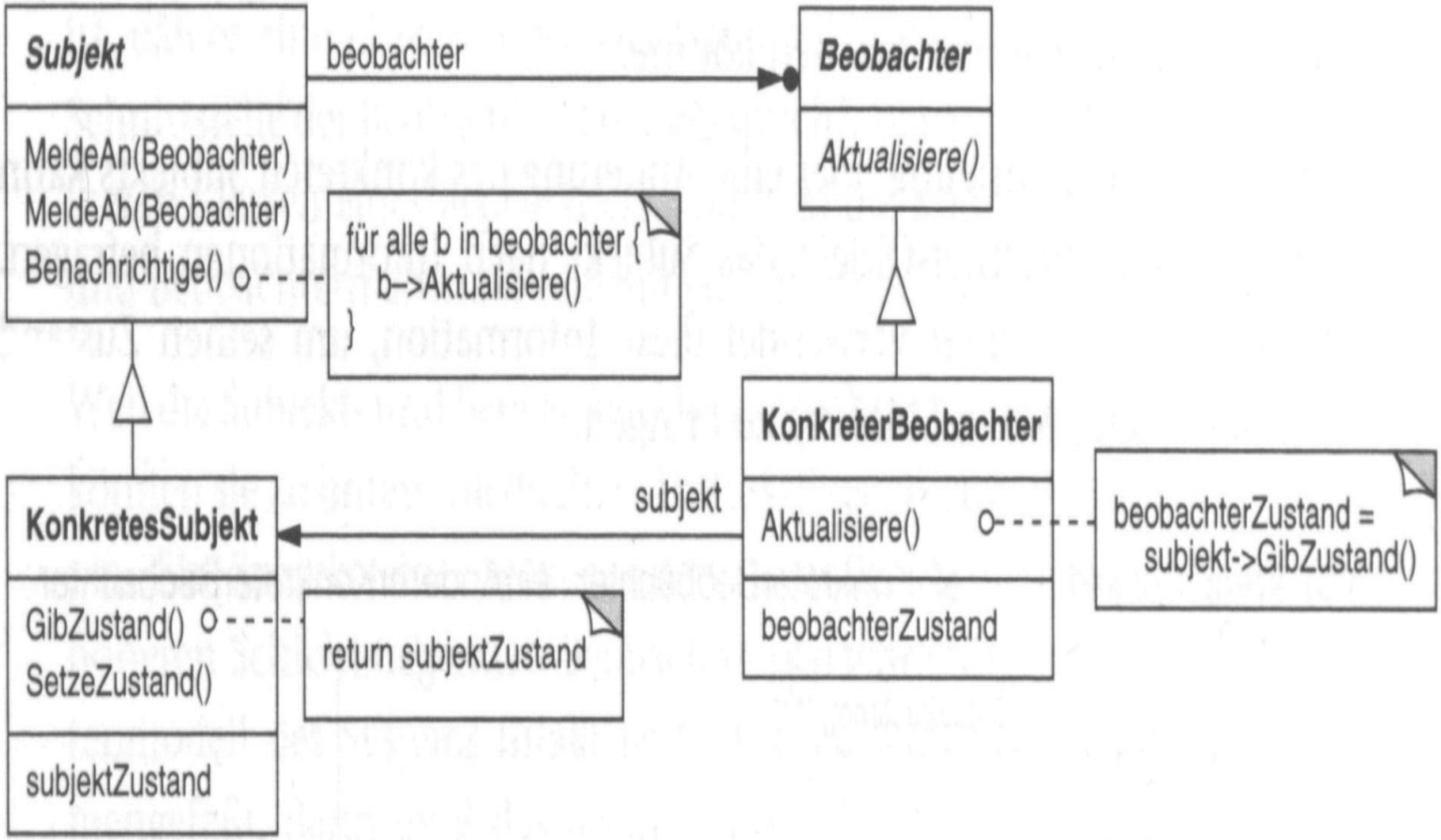
I.

Problemabschnitt: Anwendbarkeit

- Zwei Aspekte, von denen der eine vom anderen abhängt
- Eine Zustandsänderung verlangt Veränderungen bei mehreren (?) anderen Objekten
- Benachrichtigung, ohne den anderen zu kennen

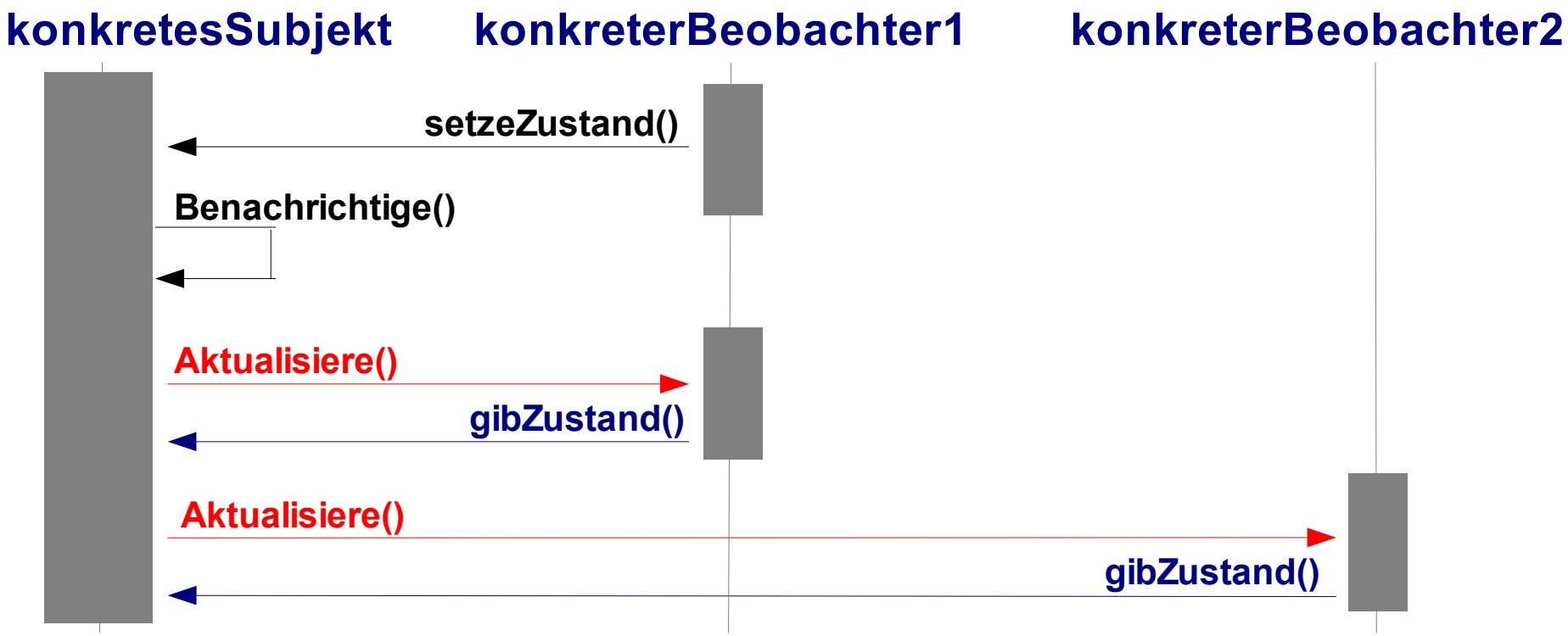
I.

Lösungsabschnitt: Struktur und Teilnehmer



Interaktionen beim Observer

- KonkretesSubjekt benachrichtigt Beobachter bei einer Zustands-änderung
- Danach kann der Beobachter Informationen vom konkretenSubjekt abfragen



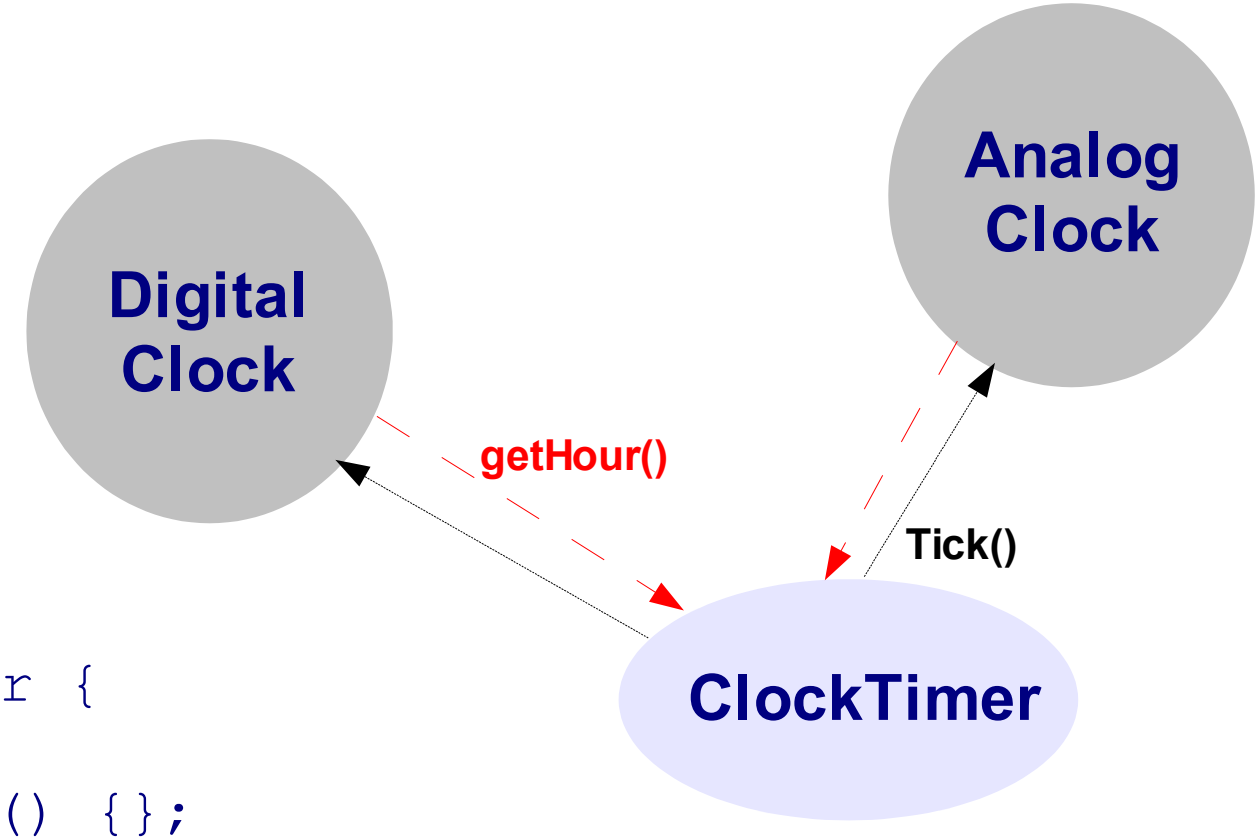
Konsequenzabschnitt

- Unterstützung von Broadcast-Kommunikation
- Unerwartete Aktualisierung
  (einfaches vs. erweitertes Protokoll)
- Abstrakte Kopplung zwischen Subjekt und Beobachter
  - Schichtungskonzept
  - Schichtenabstraktionsbildung

Diskussion der Implementierung

- Abbildung von Subjekten auf ihre Beobachter
- Beobachten von mehr als einem Subjekt
- Fehlerhafte Referenzen auf gelöschte Subjekte
- Kapseln komplexer Aktualisierungssemantik
  - Änderungsmanager

Szenario mit Beispielcode




```
class Observer {
   public:
      Observer() {};
      ~Observer() {};
      virtual void Update(Subject* ChangedSubject) = 0;
};
```

Beispielcode: Subject-Klasse

```
class Subject {
   public:
      Subject() {};
      ~Subject() {};
      void addObserver(Observer*);
      void removeObserver(Observer*);
      void Notify();
   private:
      vector<Observer*> _observers;
};

void Subject::Notify () {
       int count = _observers.size();

       for (int i = 0; i < count; i++)
         (_observers[i])->Update(this);
}
```

Beispielcode: ClockTimer-Klasse

```
class ClockTimer : public Subject {
   public:
      ClockTimer() { /* initialisiere Zeit */ };
      int GetHour();
      int GetMinute();
      int GetSecond();

      void Tick() {
          /* ...aktualisiere Zeit ... */
         Notify();
       };

   private:
      char timebuf[128]; // speichert die Zeit
};
```

Beispielcode: DigitalClock-Klasse

```
class DigitalClock: public Observer {

   public:
      DigitalClock(ClockTimer *); // addObserver(this)
      ~DigitalClock();   // removeObserver(this)

      void Update(Subject *);   // Zeit aktualisieren
      void Draw();      // neue Zeit ausgeben

   private:
      ClockTimer *_subject;   // Referenz zum Subjekt
};

 void DigitalClock::Update (Subject *theChangedSubject)
      {
        if(theChangedSubject == _subject)
          Draw();
      }
```

Abschnitt II:

Probleme objektorientierter Entwurfsmuster

Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

➔ Code von Muster und Applikation sind vermischt

➔ Semantik des Codes (*Update()*) ist unklar

Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

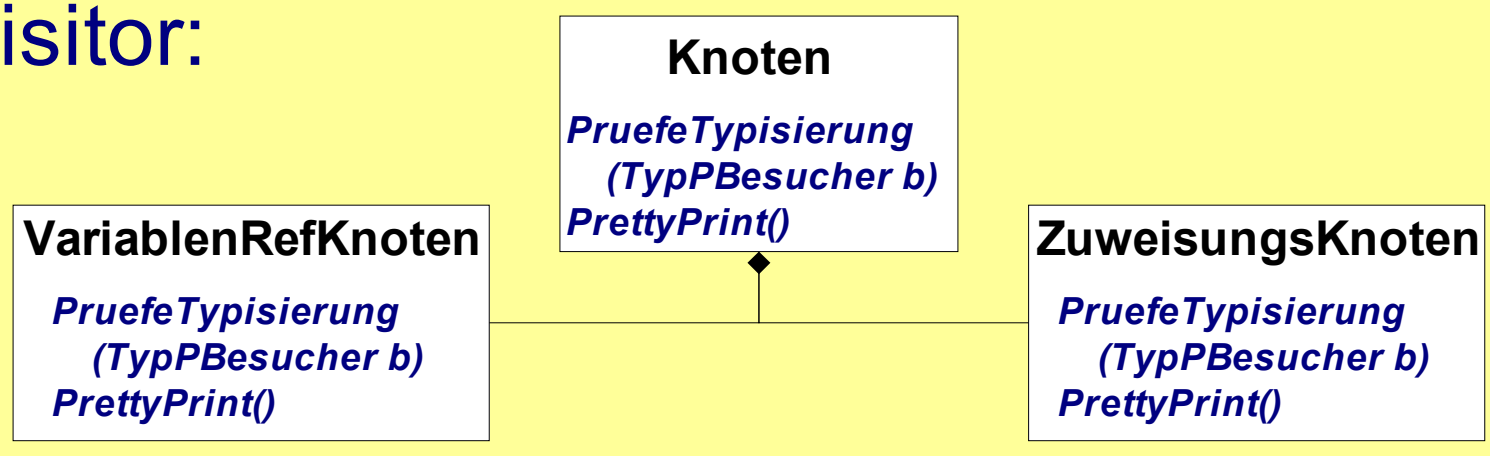
➔ Explizite Delegation macht Code schwer zu lesen: *Notify() -> Update() -> Draw()*

➔ Abhängigkeiten für spätere Veränderungen

Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

➔ Visitor:



Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

➔ Mehrfachvererbung vorausgesetzt

➔ Klassen lassen sich nicht mehr in anderen Hierarchien (ohne Pattern) wiederverwenden

Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

Diese haben 2 Gründe als Ursache:

➔ Code von Muster und Applikation ist nicht getrennt

➔ Datenkapselung muss wegen Delegation aufgegeben werden

Vier Arten von Problemen

- Confusion Problem
- Indirection Problem
- Encapsulation Breaking Problem
- Inheritance Related Problems

Diese haben 2 G rsache:

➔ Code v likation ist nicht getrennt

➔ Datenkapselung Delegation aufgegeben werden

AOP

Abschnitt III:

Entwicklung eines aspektorientierten Musters

Klassifizierung der Observer-Belange

- Generell gültige Belange
  - Existenz von Subjekt und Observer
  - Abbildung von Subjekten zu Observer
  - Update Logik (Trigger)

- Spezifische Belange
  - Wer kann Subjekt, wer Observer sein
  - konkrete Ereignisse, die den Trigger auslösen
  - die Art und Weise einen bestimmten Observer upzudaten

Klassifizierung der Observer-Belange

abstrakter Aspekt

- Generell gültige Belange
  - Existenz von Subjekt und Observer
  - Abbildung von Subjekten zu Observer
  - Update Logik (Trigger)

konkrete Aspekt-Implementierung

- Spezifische Belange
  - Wer kann Subjekt, wer Observer sein
  - konkrete Ereignisse, die den Trigger auslösen
  - die Art und Weise einen bestimmten Observer upzudaten

Abstrakter Aspekt für allg. Belange

```
aspect ObserverPattern {

  SubjectMap _perSubjectObservers;

  pointcut virtual subjectChange
                        (Subject &subject) = 0;

  virtual void updateObserver
   (Subject *subject, Observer *observer) = 0;

public:

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

Abstrakter Aspekt für allg. Belange

```
aspect ObserverPattern {

  SubjectMap _perSubjectObservers;

  pointcut virtual subjectChange
                        (Subject &subject) = 0;

  virtual void updateObserver
   (Subject *subject, Observer *observer) = 0;

public: /* ... */

  void addObserver (Subject *s, Observer *o);
  void removeObserver (Subject *s, Observer *o);

  advice subjectChange (subject) :
              after (Subject &subject) { /*...*/ }
};
```

Abstrakter Aspekt für allg. Belange

```
aspect ObserverPattern {

  SubjectMap _perSubjectObservers;

  pointcut virtual subjectChange
                        (Subject &subject) = 0;

  virtual void updateObserver
   (Subject *subject, Observer *observer) = 0;

public: /* ... */

  void addObserver (Subject, Observer);
  void removeObserver (Subject, Observer);

  advice subjectChange (subject) :
              after (Subject &subject) { /*...*/ }
};
```

Abstrakter Aspekt: notify-Advice

```
advice subjectChange (subject) : after (Subject &s) {

      /* finde zu dem Subjekt die BeobachterMenge */

      for (ObserverSet::iterator iter = oset.begin();
           iter! = oset.end ();
           ++iter)

        updateObserver (&subject, *iter);
}
```

III. Konkreter Aspekt für spezif. Belange

```
aspect ClockObserver : public ObserverPattern {

  pointcut subjectChange (Subject &subject) =
    execution ("void ClockTimer::Tick()")
                && that(subject);
  pointcut observers() = "DigitalClock"||"AnalogClock";

public:
  advice "ClockTimer" : baseclass (Subject);
  advice observers () : baseclass (Observer);
  advice observers () : void update (Subject *subject){
    Draw ((const ClockTimer &)*subject);
  }

  virtual void updateObserver(Subject *s, Observer *o){
    observer->update (subject);
  }
};
```

III. Konkreter Aspekt für spezif. Belange

```
aspect ClockObserver : public ObserverPattern {

  pointcut subjectChange (Subject &subject) =
    execution ("void ClockTimer::Tick()")
                && that(subject);
  pointcut observers() = "DigitalClock"||"AnalogClock";

public:
  advice "ClockTimer" : baseclass (Subject);
  advice observers () : baseclass (Observer);
  advice observers () : void update (Subject *subject){
    Draw ((const ClockTimer &)*subject);
  }

  virtual void updateObserver(Subject *s, Observer *o){
    observer->update (subject);
  }
};
```

III. Konkreter Aspekt für spezif. Belange

```
aspect ClockObserver : public ObserverPattern {

  pointcut subjectChange (Subject &subject) =
    execution ("void ClockTimer::Tick()")
                && that(subject);
  pointcut observers() = "DigitalClock"||"AnalogClock";

public:
  advice "ClockTimer" : baseclass (Subject);
  advice observers () : baseclass (Observer);
  advice observers () : void update (Subject *subject){
    Draw ((const ClockTimer &)*subject);
  }

  virtual void updateObserver(Subject *s, Observer *o){
    observer->update (subject);
  }
};
```

Zusammenfassung

- Durch Aspekte gelöste Probleme:
  - Confusion Problem
  - Indirection Problem
  - Encapsulation Breaking Problem
  - Inheritance Related Problems
- Wiederverwendbarkeit (s. ObserverPattern)
- Teilnehmer enthalten keinen Pattern-Code mehr
- Modularisierung erreicht in 17 von 23 Mustern
- (Un)Pluggability der Entwurfsmuster

Einige Entwurfsmuster