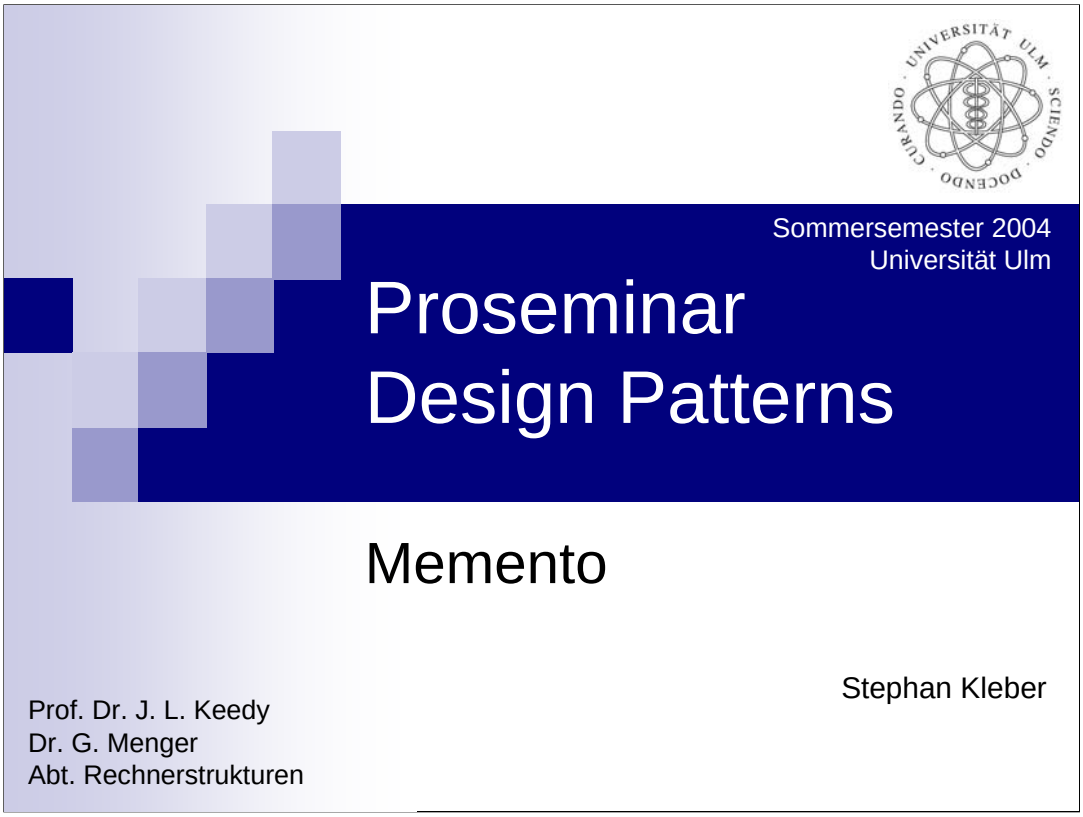


Vorstellung;
kurze Einleitung;
bis zum Punkt: „Zielsetzung des Patterns“

## Memento: Ziele

- Flexibilität der Programmstruktur
- Externalisieren der Daten eines Objektes
- Wiederherstellung des alten Zustandes
- Verwalten des „Snapshots“
- Kapselung des Datenobjekts aufrechterhalten

## Memento: Einsatz

- Undo-Funktion
- Zwischenspeichern von Objektzuständen

…

**Einsatz:**

Zwischenspeichern: bei DB-Daten, die im Speicher gehalten werden,

Verwendet in einer Form von Iterator (Gamma) um den aktuellen Iterationszustand zu sichern.

# Memento: Struktur

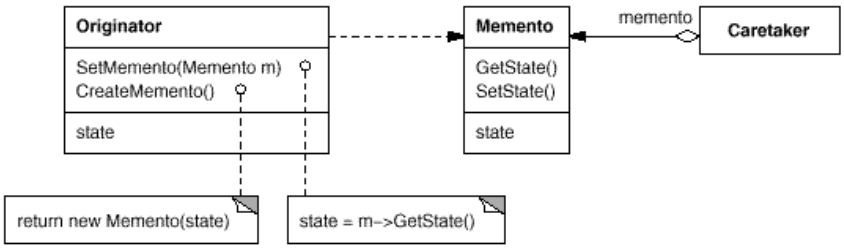


Abb.: Gamma et. al

Wie ist ein Memento-Pattern prinzipiell aufgebaut?

Welche Akteure (Objekte) haben welche Aufgaben?

# Memento: Hauptproblematik

**Kapselung des Datenobjekts aufrechterhalten:**

Memento-Inhalt …

- … geschützt vor dem *Caretaker* (und allen anderen Objekten).
- … frei verfügbar für den *Originator*.

Lösung? (= Motivation des weiteren Vortrags)

# C++ und Memento

```
class Originator {
public:
  Memento* CreateMemento() {
      Memento* myMemento = new Memento();
  }
}

class Memento {
  private:
      friend class Originator
      Memento();
}
```

kurz "friend" erläutern:

Der Konstruktor Memento() ist private,
Originator kann ihn dennoch aufrufen.

# Java und Memento

Kein „`friend`“ wie in C++

Alternativen in Java:

- Pakete (Packages)
- Geschachtelte Klassen (Inner/Nested Classes)

Grober Überblick über die Alternativen;

Nicht nur eine Alternative;

Daher:

Wer kennt diese Java-Besonderheiten?

Welche verwenden?

# Java und Memento

Allgemeine Charakteristika Innerer Klassen:

- Innerhalb der Deklaration einer anderen, „äußeren“ Klasse
- Innere und äußere Klasse haben Zugriff auf private Inhalte der jeweils anderen

# Java und Memento

Arten Innerer Klassen:

- Static Inner Class („[Static] Nested Class“)
- Nonstatic Inner Class („Inner Class“)
- Local Inner Class („Local Class“)
- Anonymous Inner Class („Anonymous C.“)

Innere Klassen aufzählen;
Namensgebung kurz diskutieren:
Nested Class = C++
(Oder: Static/Nonstatic Member Class)
sollen im Folgenden beschrieben werden

# Static Inner Class

```
public class Originator {
      private String state;
      Memento createMemento() {
            Memento myMemento = new Memento();
            myMemento.state = this.state;
      }
      void setMemento(Memento aMemento) {
            this.state = aMemento.state;
      }
      // …
      static class Memento {
            private Memento() { };
            private String state;
      }
}
```

erstes Code-Beispiel einer Inner Class;
Detaillierte Erläuterung des Codes:

Dieses Memento besitzt keine Get/SetState()-Methoden:

O. kann nicht gezwungen werden diese zu nutzen,
damit sind diese sinnlos.

Konstruktor und Felder im Memento private;
Alternative: die ganze innere Klasse private;

aber: Typ? Caretaker kann jetzt nur noch „objects“ verwalten

# Nonstatic Inner Class

- Vergleichbar der Static Inner Class
- Referenz auf das Herkunftsobjekt

Nutzbarkeit der Referenz in Memento ist bestenfalls zweifelhaft

Grundbetrachtung wie Static Inner Class

Neu: Referenz

Umständliche Implementierung, Speicher- und Rechenaufwändig;

In jedem Fall handelt es sich streng genommen nicht mehr um das Memento-Pattern.

# Local Inner Class

- Deklaration innerhalb einer Methode
- Außerhalb der Methode völlig unsichtbar
- Implizite Referenz (vgl. Nonstatic Inner C.)
  - nicht in static Kontext (static-Methode)
  - verfügbar in nicht-statischen Methoden

Referenz für das Memento-Pattern uninteressant (vgl. Nonstatic IC)

# Local Inner Class

```
public class Originator {
   public interface BasicMemento { }
   private interface Memento extends BasicMemento
      { String getState(); }
   // …
   private String state;
   public void setMemento(BasicMemento memento)
      { this.state = ((Memento)memento).getState(); }
   public Memento createMemento() {
      class ConcreteMemento implements Memento
            { private String state; }
      ConcreteMemento newM = new ConcreteMemento();
      newM.state = this.state;
      return newM;
   }
}
```

Problematik:

• LIC für die Außenwelt unsichtbar
aber: andere Methode liest Memento wieder ein
> interface/Vererbung!

**Zwecke der drei Typen-Ebenen:**

• public interface *BasicMemento*
Typsicherheit für den Caretaker

• private interface *Memento* extends BasicMemento
Zugriff auf Accessor für setMemento()

• class *ConcreteMemento* implements Memento
kein Mutator: („immutable object“)

# Anonymous Inner Class

```
public class Originator {
   public interface BasicMemento { }
   private interface Memento extends BasicMemento
      { String getState(); void setState(String m); }
   // …
   public Memento createMemento() {
      Memento newM = new Memento() {
            private String state;
            public String getState() {/* … */}
            public void setState(String m); {/* … */}
      };
      newM.setState(this.state);
      return newM;
   }
}
```

Vergleichbar mit der Local Inner Class,

Semantik: new Memento() heißt implements/extends Memento

aber: komplexer, da selbst für setState() ein interface bemüht werden muss.

# Packages

```
package Pattern;
public class Originator {
      private String state;
      public Memento createMemento()
            { m = new Memento();
            m.setState(this.state); }
      public void setMemento(Memento m)
            { this.state = m.getState(); }
}
```

---

```
package Pattern;
public class Memento {
      private String state;
      String getState() { /* … */ }
      void setState(String state) { /* … */ }
}
```

[Gamma]s Pattern-Beschreibung tritt hier am deutlichsten hervor.

# Nachtrag: Der Caretaker

```
import java.util.LinkedList;
public class Caretaker {
   private LinkedList mementos;
   public Caretaker() { mementos = new LinkedList(); }
   public void pushMemento(Memento memento)
      { mementos.addLast(memento); }
   public Memento popMemento() {
      Memento value;
      if (mementos.isEmpty()) { value = null;
      } else {    value = mementos.getLast();
                  mementos.removeLast();  }
      return value;
   }
   // …
}
```

Bisher unterschlagen,
aber hier nicht das zentrale Problem,
der Vollständigkeit halber noch als Nachtrag.

Diese Implementierung sollte mit minimalen Änderungen in allen Beispielen funktionieren.

Stack, denkbar wären auch: Random Access, ein einziger Zustand (…)

# Bewertung und Schlussfolgerung

Static/Nonstatic Inner Class:

- Einfach, Übersichtlich
- Keine Kapselung zw. verschachtelten Klassen
- Accessors/Mutators nicht sinnvoll

Nonstatic Inner Class:

- Referenz rechtfertigen

Zusammenfassung der Stärken und Schwächen der Implementierungen:

• Einfach: kann schnell und kompakt geschrieben werden
Übersichtlich: beisammen was zusammen gehört,
dennoch eigenständiger Code (Ggs. Local Inner Class)

• Kapselung / keine Mutators/Accessors erzwingbar
z.B. für Aggregation anderer Klassen
aber: sehr effizient

• Referenz: Aufwand (Speicher, Operationen) für Referenz

## Bewertung und Schlussfolgerung

Local/Anonymous Inner Class:

- Vererbung
- Schwer verständliches Verhalten
- Unübersichtliche Verschachtelung

Local Inner Class

- immutable-Objekte

• Vererbung: notwendig, erzwingt komplexen Code
• erschwertes Verständnis: Klassendeklaration in Methode
• kompakter aber evtl. unübersichtlicher Code

• Immutable hier einfach, elegant, schnell erreichbar

# Bewertung und Schlussfolgerung

Packages

- Übersichtlich:
  komplexe Strukturen durch die Aufteilung in separate Dateien leichter lesbar
- Einfache Kapselung auf mehreren Ebenen allein durch Modifikatoren

Übersichtlich: Zwar in getrennten Dateien, aber einem Paket

Kapselung: differenziert zw. Welt, Paket, Selbst
dadurch erzwungene Mutatoren möglich

Bsp: Festspeicherung in File durch eine Methode

## Bewertung und Schlussfolgerung

- Nonstatic Inner Class
- Anonymous Inner Class
- Package
- Local Inner Class
- Static Inner Class

Übersicht über die behandelten Fälle:
(in Klammern die Reihenfolge meiner Bevorzugung)

NIC (5):

- zu Resourcenintensiv für die meisten Fälle

AIC (4):

- Relativ kurz und einfach
- kompliziert

Package (3):

- übersichtlich
- einfach aber etwas schreibaufwändig
- Gamma-konform

LIC (2):

- kompakt

SIC (1):

- sehr kurz und einfach