Prof. Dr. J. L. Keedy
Dr. G. Menger
Abt. Rechnerstrukturen

Stephan Kleber

Sommersemester 2004
Universität Ulm

Proseminar Design Patterns

Memento

Inhalt

Abstract

Der Zweck des Design-Patterns Memento besteht darin, das Zwischenspeichern von objektinternen Daten zu ermöglichen, ohne dabei die Kapselung des Datenobjekts zu verletzen. Zudem soll dafür gesorgt werden, dass die einzelnen Objekte möglichst einfach und unabhängig bleiben, um die Erweiterbarkeit sicherzustellen.

In Java kann dies auf verschiedene Arten erreicht werden. Hier soll ein Überblick über die Folgenden gegeben werden: Es sollen die Static und die Nonstatic Inner Class, sowie die Local und die Anonymous Inner Class betrachtet werden und gegen das Konzept der Paketzugriffsrechte der Programmiersprache Java gestellt werden. Das Verständnis dieser Möglichkeiten soll auch eine Erleichterung bei der konkreten Implementierung sein.

1 Einleitung

1.1 Thema

Das Design Pattern Memento, wie es von [Gamma] in Bezug auf die Programmiersprache C++ beschrieben wurde, hat den zentralen Vorteil, dass es keinen Bruch der Kapselung erfordert um Daten eines Objekts zu externalisieren und zwischenzuspeichern. Um gemäß dem Pattern vorgehen zu können, muss man in der jeweiligen Programmiersprache eine Möglichkeit finden, die Felder des Memento-Objekts für das korrespondierende Daten-Objekt zugänglich zu machen, während alle anderen Objekte keinerlei Kenntnis vom Inneren des Objekts erlangen sollen.

In Java bieten sich mehrere Vorgehensweisen an, um diese Anforderungen zu erfüllen, die hier diskutiert werden sollen.

1.2 Motivation

Die Implementierung eines Mementos wird in [Gamma] auf einen für C++ spezifischen Mechanismus gegründet: Die Deklaration einer Klasse als `friend`. In Java gibt es keine direkte Entsprechung hierfür, allerdings gibt es dort durchaus andere Möglichkeiten, ein vergleichbares oder gar identisches Verhalten zu erzielen. In [Cooper] wird die Option propagiert, die erforderlichen Klassen in ein Paket zusammenzufassen und das Standartverhalten von Java bezüglich der Zugriffsregeln auszunutzen.

In der Tat bietet sich in Java jedoch nicht nur diese Vorgehensweise an, sondern es gibt noch die Möglichkeit, die betroffenen Klassen als *Inner Classes* zu kaskadieren und den privilegierten Zugriff, den eine solche innere bezüglich ihrer umgebenden Klasse besitzt, auszunutzen.

Bei näherer Betrachtung stellt sich heraus, dass durch die komplexen und vielfältigen Möglichkeiten, die die Programmiersprache Java bietet, einige beachtenswerte Probleme und Fragen aufgeworfen werden.

Zum einen soll hier gezeigt werden, wie die jeweiligen Implementierungen bewältigt werden können und welche Besonderheiten hier zu beachten sind, um auf dieser Grundlage die entscheidendere Frage beantworten zu können, welche Vorteile die Implementierung unter Verwendung einer der vier zur Auswahl stehenden Arten von inneren Klassen bringt und welche Nachteile man sich bei deren Verwendung einhandelt. Es muss dann auch noch beleuchtet werden, in wieweit die in [Cooper] beschriebene Vorgehensweise hierbei ihre Berechtigung hat und wie alle diese Möglichkeiten im Vergleich zueinander stehen. Dadurch soll ein tieferes Verständnis für das Pattern sowie eine Entscheidungsgrundlage geschaffen werden, auf deren Basis eines der Verfahren für die Implementierung des Patterns in Java gewählt werden kann.

1.3 Begriffsklärung

Design Patterns beschreiben bewährte Vorgehensweisen zur Lösung häufiger auftretender Probleme bei der Programmierung von objektorientierten Softwarelösungen. Hauptaugenmerk liegt dabei auf der einfachen Wartbarkeit und Erweiterbarkeit der Software. [Gamma S. 2f]

Inner Classes (innere Klassen) wird in der Literatur nicht eindeutig verwendet [Bloch], [Horstmann S. 282f]. In vorliegender Arbeit sind in Java Klassen gemeint, die innerhalb einer anderen Klasse deklariert werden und privilegierte Rechte beim Zugriff auf die umgebende Klasse haben. Hier wird dieser Begriff als Oberbegriff über alle vier in Java möglichen Arten dieser Technik der objektorientierten Programmierung verwendet.

1.4 Abgrenzung

Obwohl einige konkrete Beispiele angeführt werden sollen, ist das vorliegende Papier nicht in erster Linie als Anleitung zur Programmierung des Memento-Patterns unter Java zu verstehen, sondern vielmehr eine Entscheidungshilfe für die Wahl der Strategie einer eigenen

Implementierung. Hier sollen vor allem die Möglichkeiten von Java diskutiert und kein Vergleich mit C++ oder anderen Programmiersprachen geleistet werden.

1.5 Vorgehensweise

Zunächst wurden einige Besonderheiten der Sprache Java näher ergründet, um Informationen darüber zu gewinnen, welche Möglichkeiten existieren, das Design Pattern Memento gemäß seiner Definition zu implementieren. Ausgehend von der Literatur wurden die verschiedenen Möglichkeiten an Beispielen getestet und Vereichfachungen davon in dieser Arbeit auch als Anschauungsmaterial verwendet. Dabei auffallende Eigenschaften der Implementierungen dienten als Grundlage für die Bewertung der verschiedenen Methoden.

2 Grundlegendes

Dieser Abschnitt trägt Grundlegendes über das Design Pattern Memento und auch die Programmiersprache Java zusammen, so dass dieses Basiswissen komplett zur Verfügung steht.

2.1 Memento für Java

Der Zweck des Design-Patterns Memento besteht darin, das Zwischenspeichern objektinterner Daten zu ermöglichen, ohne dabei die Kapselung des Datenobjekts zu verletzen. Zudem soll dafür gesorgt werden, dass die einzelnen Objekte möglichst einfach und unabhängig bleiben, um die Erweiterbarkeit sicherzustellen. [Gamma S. 283]

In diesem Pattern spielen drei Klassen eine Rolle: Das *Originator*-Objekt ist das Datenobjekt, dessen Inhalt externalisiert und zwischengespeichert werden soll. Das *Memento*-Objekt soll eben diese Daten aufnehmen können. Das *Caretaker*-Objekt kümmert sich schließlich um die Aufbewahrung und Organisation der Mementos.

Um die Kapselung nicht zu verletzen, ist auch der Inhalt des Memento-Objekts vor der Außenwelt zu schützen und nur dem Originator Zugriff auf den Inhalt des Mementos zu erlauben. Dazu müssen zwei verschiedene Typen für das Memento-Objekt geschaffen werden: Gegenüber seinem Originator eine weite Schnittstelle (Typ, *Interface*) um die Speicherung im und Wiederherstellung der Daten aus dem Memento zu ermöglichen. Alle restlichen Objekte, zu denen auch der Caretaker zählt, dürfen keinerlei Kenntnis über den Inhalt der Mementos bekommen können. Die von [Gamma] in den Beispielquellen verwendete Methode dies zu erreichen ist der `friend`-Mechanismus unter C++. Dieser semantische Trick ermöglicht einer Klasse von Objekten Zugriff auf die privaten Felder und Methoden einer ganz bestimmten anderen Klasse. Die Kontrolle darüber welchen Klassen dieses Privileg eingeräumt wird, verbleibt bei der Klasse, auf deren Inhalte zugegriffen werden soll, womit die Auflagen der Kapselung erfüllt bleiben. Ein direkt vergleichbares Konzept hierfür ist in Java nicht gegeben.

Nun gibt es dort zwar keine Möglichkeit, eine gegenüber den eigenen privaten Feldern und Methoden privilegierte Klasse zu bestimmen, aber Javas Zugriffsregelsystem erlaubt eine ganze Reihe von anderen Optionen.

2.2 Javas Zugriffsregelsystem

Die vier Zugriffsebenen in Java lauten:

`private`
`protected`
`public`
[*package-protected*]

`Private` deklarierte Methoden und Felder erlauben Zugriff nur den Methoden der eigenen Klasse, allerdings auch den so genannten „inneren Klassen“ (*Inner Classes*). Dabei handelt es sich um Klassen, die innerhalb einer anderen Klasse deklariert sind und somit als „Mitgliedsklasse“ (*Member Class*) Zugriffsrechte auf private Felder und Methoden der umgebenden Klasse haben.

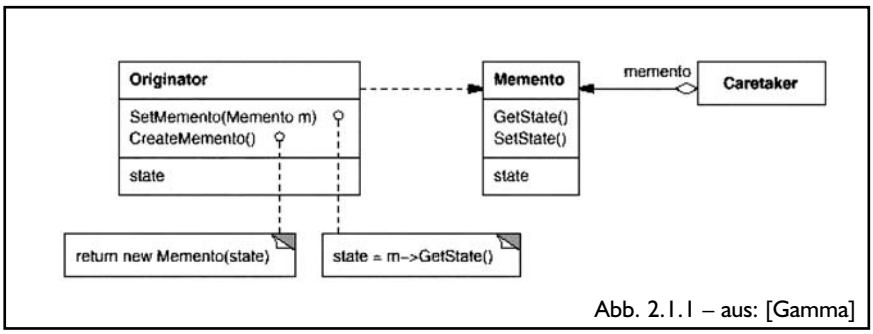

Abb. 2.1.1 – aus: [Gamma]

Der Modifikator (*Modifier*) `protected` erlaubt allen Klassen eines Paketes (*Packages*) den Zugriff auf die damit bezeichneten Inhalte einer Klasse aus ebendiesem Paket. Aber auch durch Vererbung erweiterte Klassen, erhalten Zugriff auf Felder und Methoden der Klasse auf der sie basieren, wenn diese Inhalte dieses Attribut tragen. Dem ganzen Programm zugänglich sind `public` deklarierte Methoden und Felder.

Die vierte Zugriffsebene hat kein Schlüsselwort mit dem sie erzwungen werden könnte, sondern ist die Standardebene, die bei Fehlen eines Modifikators verwandt wird. Sie wird in der Literatur relativ inkonsequent mit „*package-protected*“, „*private-protected*“, „*package-private*“ oder „*default*“ bezeichnet [Cooper S. 180], [Bloch S. 48]. Es wird zwar den Klassen des eigenen Paketes der Zugriff erlaubt (vgl. `protected`), allerdings haben die abgeleiteten Klassen in anderen Paketen keinerlei Zugriff auf die entsprechenden Klasseninhalte.

Wir werden feststellen, dass unter anderem die Rechte von inneren Klassen bezüglich ihrer umgebenden sowie die Umkehrung, dass nämlich die umgebende Klasse gleichermaßen auf die innere Zugriff hat, uns genau die Möglichkeiten eröffnen, die wir brauchen, um damit den `friend`-Mechanismus aus C++ nachbilden und auch übertreffen zu können [Cooper S. 180ff].

2.3 Innere Klassen in Java

Nun gibt es in Java allerdings nicht nur eine Art innerer Klasse, sondern es gibt vier, wobei sich auch deren Eigenschaften und Verhalten zum Teil beträchtlich unterscheiden.

Hier seien diese zunächst aufgezählt:

Static Inner Class
Nonstatic Inner Class
Anonymous Inner Class
Local Inner Class

Im Folgenden wird ein Überblick über die wichtigsten Eigenschaften dieser inneren Klassen gegeben.

Sowohl *Nonstatic* als auch *Static Inner Classes* werden direkt innerhalb einer anderen Klassendeklaration vereinbart. Die erstere enthält eine Referenz auf die erzeugende Objektinstanz und kann nur erzeugt werden, wenn es eine Instanz der umgebenden Klasse gibt [Bloch S. 72]. Daraus folgt auch, dass beliebig viele Instanzen einer *Nonstatic Inner Class* kreiert werden können, von denen jede das Objekt der umgebenden Klasse vor dem *Garbage Collector* bewahrt.

Static Inner Classes besitzen noch weitere und eventuell missverständliche Bezeichnungen: Da diese sich exakt so verhalten wie die *Nested Classes* in C++ [Horstmann S. 283], werden beide Bezeichnungen Synonym verwendet, obwohl in der Literatur auch alle vier inneren Klassen unter diesem Begriff erfasst werden [Bloch S. 71]. Wir wollen beim Begriff *Static Inner Class* bleiben um Verwirrungen zu vermeiden.

Auch eine *Static Inner Class* muss zur Verwendung instanziert werden, wie jede andere innere Klasse auch. Daher sind trotz Schlüsselwort `static`, mehrere Instanzen der Klasse möglich, ja gar die Regel. Sie enthält jedoch anders als die *Nonstatic Inner Class* keine Referenz auf das erzeugende Objekt und kann daher unabhängig von diesem existieren.

Anders als bei den beiden bisher beschriebenen Klassenarten, werden *Local Inner Classes* (*Local Classes*) und *Anonymous Inner Classes* (*Anonymous Classes*) innerhalb einer Methodendeklaration definiert. Dadurch sind sie komplett unsichtbar für die Außenwelt. Eine *Local Inner Class* ist nur innerhalb der erzeugenden Methode sichtbar und hat Zugriff auf Felder der Klasse, sowie auf die lokalen Variablen der Methode, die `final` deklariert sind. *Anonymous Classes* verhalten sich ähnlich der *Local Classes*, mit dem Unterschied, keinen Namen zu tragen. Es kann daher auch kein Konstruktor deklariert werden, so dass eine *Anonymous Class* nur eine Schnittstelle implementieren oder eine bereits bestehende Klasse erweitern, aber keine völlig neue Klasse definieren kann.

Ein kurzer Überblick hierüber, findet sich auch in [Bloch S. 71ff], eine sehr ausführliche Diskussion der Mechanismen von inneren Klassen findet sich in [Horstmann S. 282ff].

3 Die Implementierungen

Damit sind alle Grundlagen geschaffen. Das Design Pattern Memento ist genauso bekannt, wie auch die Eigenschaften der Programmiersprache Java, die wir im Folgenden benötigen werden.

3.1 Static Inner Class

Die Klasse für das Memento-Objekt soll zunächst als *Static Inner Class* innerhalb der Originator-Klasse deklariert werden. Dadurch kann der Originator ungehindert die privaten Felder des Mementos schreiben und lesen, was auch die Verwendung von Akzessoren und Mutatoren überflüssig macht, da nichts den Originator dazu zwingen kann diese zu verwenden. Die Last sicherzustellen, dass Daten sinnvoll im Memento abgelegt werden, liegt hier allein beim Originator. Die Memento-Klasse als solche ist dagegen für alle anderen Klassen des Paketes sichtbar, da sie keinen zugriffsbezogenen Modifikator in der Signatur trägt. Ein allgemeines Beispiel für eine solche Implementierung enthält Codefragment 3.1.1.

```
public class Originator {

   // Die Datenfelder des Originators
   public String state;

   // Erzeugung des Mementos
   Memento createMemento() {
      Memento newMemento = new Memento();
      newMemento.state = this.state;
      return newMemento;
   }

   // Wiedereinlesen des Mementos
   public void setMemento(Memento memento) {
      this.state = memento.state;
   }

   // ...

   // Das Memento mit den relevanten Feldern
   static class Memento {
      // versteckter Memento-Konstruktor
      private Memento() { };
      private String state;
   }
}
```

Codefragment 3.1.1

Im Prinzip könnte die Klasse Memento innerhalb des Originators auch von außen komplett unerreichbar gemacht werden (vgl. Codefragment 3.1.2). Die Kapselung wird dadurch nicht tangiert, da im einen Fall die Felder, im andern die ganze Klasse `private` ist. Es gibt aber dennoch einen entscheidenden Unterschied zwischen beiden Ansätzen:

```
public class Originator {
   // ...
   private static class Memento {
      String state;
   }
   // ...
}
```

Codefragment 3.1.2

Obwohl bei Ersterem die nach außen hin geschützten Felder nur von der umgebenden Klasse aus sichtbar sind, hat die vermeintlich leere und – so scheint es – sinnlose Klasse eine wichtige Funktion, die der zweite Ansatz nicht berücksichtigt. Dadurch, dass der Caretaker explizit nach dieser Klasse verlangen kann, können Typenfehler schon durch den Compiler vermieden werden, denn die Klasse – nicht aber deren Inhalt – ist der Öffentlichkeit bekannt und kann als Objekttyp bei der Parameterübergabe an Methoden verwendet werden. Dadurch kann garantiert werden, dass das Objekt ein Memento ist, obwohl der Caretaker keine Möglichkeit hat, den Inhalt des Objekts zu inspizieren.

Da ein sinnvoll mit Daten gefülltes Memento offensichtlich nur von einem Originator selbst erzeugt werden kann, muss durch einen privaten Konstruktor die Instanzierung eines solchen Objekts außerhalb seiner übergeordneten Klasse verhindert werden, damit die Klasse selbst sichtbar bleiben kann. Andernfalls könnte ein Memento von einem Punkt im Programm kreiert werden, der leicht manipulierbar oder häufig Veränderungen unterworfen ist. So können bewusst oder versehentlich Daten als Memen-

tos an den Originator übergeben werden, die für die Programmausführung möglicherweise gefährlich sind. Ein solcher versteckter Konstruktor ist natürlich dennoch immer von der umschließenden Klasse – hier also dem Originator – aufrufbar.

Es bleibt darauf zu achten, dass bei Änderungen am Memento die Zugriffsebene richtig gesetzt wird: Sollte eine weitere Methode oder ein Feld hinzukommen, das nicht `private` ist, das sich aber auf irgend eine Art und Weise auf ein geschütztes Feld insbesondere der äußeren Klasse bezieht, eröffnet man – meist versehentlich und ungewollt – einen Weg die Kapselung des umgebenden Objekts zu brechen [Bloch S. 48].

3.2 Nonstatic Inner Class

Nonstatic Inner Classes haben die gleichen Eigenschaften wie *Static Inner Classes*, wodurch auch die Optionen für die Implementierung die gleichen sind. Allerdings erweitert Java jede Instanz einer *Nonstatic Inner Class*, gegenüber einer ansonsten völlig gleich deklarierten *Static Inner Class*, implizit um eine Referenz auf die erzeugende Instanz der umschließenden Klasse [Horstmann S. 285ff], [Bloch S. 71f].

Im Rahmen des Design Patterns Memento ist eine sinnvolle Nutzung dieses Mechanismus unwahrscheinlich, wenn nicht gänzlich ausgeschlossen. Wollte man die Möglichkeiten dieser Einrichtung der Java-Programmiersprache ausloten, müsste man sich deutlich vom vorgegebenden Pattern entfernen, weshalb eine mögliche Nutzung hier nur kurz angedacht wird. An dieser Stelle wird nur ein ungetesteter Ansatz in den Raum gestellt, der eine *Nonstatic Inner Class* sinnvoll in Verbindung mit dem Design Pattern Memento bringen kann.

Um abhängig von der Art des Mementos das Verhalten einer Routine anpassen zu können, wird es für den Caretaker oder andere Objekte nötig, auf Inhalte des Mementos – beispielsweise in Form seines Herkunftsobjektes – zugreifen zu können. Die dazu nötige öffentliche Zugänglichkeit der Referenz erzwingt, dass eine Zugriffsmöglichkeit darauf öffentlich zur Verfügung gestellt wird.

Wenn eine Methode der Memento-Klasse ohnehin öffentlich sichtbar ist, um den Referenzzugriff zu ermöglichen, kann man, anstatt die Referenz über eine Methode frei zugänglich zu machen und diese in einer anderen Klasse auslesen zu lassen, das Zurückschreiben der Memento-Daten in den Originator von einer Methode des Mementos selbst erledigen lassen. Wenn schon nicht patternkonform, so erscheint der Ansatz aber recht elegant, da hier keine zusätzlichen und unnötigen, da impliziten Parameter mehr übergeben werden müssen, wie im Codefragment 3.2.1 zu sehen ist. Es genügt ein Methodenaufruf im Memento und der Originator hat seinen alten Zustand. Die Methode `recall()` der Klasse Memento ersetzt dabei die Methode `setMemento()` der Klasse Originator.

Der Hauptkritikpunkt an dieser Vorgehensweise ist aber gerade diese Verlagerung der Objektkompetenz, denn laut Pattern soll das Memento einfach aufgebaut sein, weshalb gerade der Originator auch `setMemento()` enthält. Hinzu kommt noch, dass der Originator natürlich somit die Kontrolle darüber abgibt, wann er ein Memento akzeptiert.

Geeignet wäre dieses Vorgehen für *Undo*-Listen, vergleichbar der, die in [Cooper S. 185] implementiert ist.

```
public class Originator {
   // ...
   class Memento {
      private String state;
      /*
         Wiederherstellung der Daten durch
         eine Methode des Mementos selbst
      */
      public void recall() {
         /*
            Zugriff auf die Referenz zum
            Erzeugerobjekt erfolgt über
            folgende Schreibweise:
         */
         Originator.this.state = this.state;
         // ...
      }
   }
   // ...
}
```

Codefragment 3.2.1

3.3 Local Inner Class

Diese Art von inneren Klassen wird innerhalb einer Methode deklariert und ist für die Außenwelt völlig unsichtbar. Selbst andere Methoden derselben Klasse wissen nichts von der Existenz dieser inneren Klasse. Da die Instanz des erzeugten *Local Inner Class*-Objekts aber dem Caretaker und damit nach außen übergeben wird und später wieder von einer anderen Methode zurück in den Originator geschrieben werden soll, muss zumindest eine Schnittstelle der Klasse bekannt sein, um auf die nötigen Felder zugreifen zu können. Es reicht aber aus, wenn der Originator Kenntnis über das `interface` des Mementos hat, alle anderen Klassen dürfen die Felder des Mementos nicht sehen.

Methoden die in einem `interface` deklariert werden sind immer `public`, was aber kein Hindernis für die Verwendung ist, schließlich kann zum Beispiel auch ein `interface` in eine umgebende Klasse so eingebettet werden, dass die gewünschten Zugriffsbeschränkungen erreicht werden. Wie im Codefragment 3.3.1 zu sehen ist, benötigen wir hier einen Akzessor (*accessor*) – also eine `Get`-Methode –, die im entsprechenden `interface` deklariert ist.

Durch günstige Deklaration der Mutatoren im entsprechenden `interface` werden folgende Ziele erreicht: Das `public interface BasicMemento` ermöglicht dem Caretaker festzustellen, ob er ein Objekt des korrekten Typs übergeben bekommt. Das `private interface Memento extends BasicMemento` öffnet die Kapselung des Mementos dann gerade so weit, dass der Originator – und nur er – über die Methode `getState()` auf die Daten im Memento Zugriff erlangt. Die `class ConcreteMemento implements Memento` wird in der Methode `createMemento()` implementiert, die dann ohne Mutator direkt auf die Felder der Memento-Klasse zugreift.

In der erzeugenden Methode `createMemento()` können beliebige Änderungen an nichtöffentlichen Feldern vorgenommen werden. Nach Beendigung der Methode ist bei Fehlen von Mutatoren im `interface` keine Möglichkeit mehr gegeben, das einmal erzeugte Objekt zu ändern (*immutable object*). Dies ist generelles Wunschverhalten der objektorientierten Programmierung im Allgemeinen und im Speziellen von Mementos, schließlich soll durch die externalisierten Daten ein Objektzustand dauerhaft gesichert werden.

```
public class Originator {

  // leeres interface als Übergabetyp für den Caretaker
  public interface BasicMemento { }

  // enthält den Akzessor für die Methode setMemento
  private interface Memento extends BasicMemento
    { String getState(); }

  // ...

  private String state;

  public void setMemento(BasicMemento memento) {
    this.state = memento.getState();
  }

  public Memento createMemento() {
    class ConcreteMemento implements Memento {
      // in anderen Methoden unsichtbar
      private String state;

      // Implementationen des Akzessors des Interfaces
      public String getState() {
        return this.state;
      }
    }

    ConcreteMemento newMemento = new ConcreteMemento();
    newMemento.state = this.state;
    return newMemento;
  }
}
```

Codefragment 3.3.1

3.4 Anonymous Inner Class

Die Implementierung als *Anonymous Inner Class* unterscheidet sich zunächst nicht wesentlich von der als *Local Inner Class* [Bloch S. 73f]. Die Klasse bekommt aber keinen eigenen Namen mehr, sondern wird direkt bei der Instanzierung deklariert, so dass auch diese Klasse von außen nicht sichtbar ist. Dadurch erfordert auch dieses Vorgehen die Erweiterung einer abstrakten Klasse oder die Implementierung eines `interface`s. Allerdings zeigt sich hier schon der entscheidende Unterschied zur *Local Inner Class*, da außer den Akzessoren hier auch noch Mutatoren in einem `interface` vereinbart werden müssen, damit die `createMemento()`-Methode die Felder im Memento setzen kann. Durch den anonymen Status der Klasse ist es nicht einmal der Methode, in der die Klasse implementiert wurde, möglich deren Methoden anders als über ein vorher an anderer Stelle vereinbartes `interface` zu erreichen.

Das Codefragment 3.4.1 zeigt, wie eine solche *Anonymous Inner Class* aussehen würde.

```
public class Originator {

   // leeres interface als Übergabetyp für den Caretaker
   public interface BasicMemento { }

   /*
      neben Akzessoren für die Methode setMemento()
      enthält das interface nun auch noch
      Mutatoren für die Methode createMemento()
   */
   private interface Memento extends BasicMemento {
      String getState();
      void setState(String m);
   }

   private String state;

   public void setMemento(BasicMemento memento) {
      this.state = memento.getState();
   }

   // ...

   public Memento createMemento() {
      Memento newMemento = new Memento() {
         private String state;
         // Implementationen des Interfaces
         public String getState() {
            return this.state;
         }
         public void setState(String state) {
            this.state = state;
         }

      }

      newMemento.setState(this.state);
      return newMemento;
   }
}
```

Codefragment 3.4.1

3.5 Packages

Durch die beiden letzten Arten von inneren Klassen waren wir bereits gezwungen nicht nur eine Klasse, sondern zusätzlich noch einige `interface`s als innere in eine umgebende Klasse zusammenzustellen, um damit die Klassen und `interface`s vom restlichen Javaprogramm zu trennen und somit die gewünschte Kapselung zu erzielen. Aber auch durch den Einsatz von Paketen kann eine ganz ähnliche Gruppierung erreicht werden. Die Grundlage ist wie bei den inneren Klassen das ausgeklügelte System von Zugriffsregeln, das auch in Verbindung mit Paketen prinzipiell ein mächtiges Werkzeug bei der Implementierung des Memento Design Patterns sein kann.

Erst bei Nutzung von Paketen entfalten die Zugriffsregeln ihr gesamtes Potential, da nun auch der *default*-Zustand und der Modifikator `protectet` sinnvoll eingesetzt werden können. Eine Möglichkeit der Nutzung dieser Mechanismen besteht darin, eine vollwertige Klasse in das Paket des Originators zu stellen. Dadurch wird die Deklaration einer inneren Klasse ersetzt und es kann direkt mit der regulären Klasse gearbeitet, das heißt diese instanziert werden [Cooper S. 180ff]. Genau wie bei den inneren Klassen ist die Memento-Klasse für die Außenwelt unsichtbar, da sie *private-protected* deklariert wird. Weitere Besonderheiten, auf die im Rahmen dieser Arbeit nicht eingegangen werden soll, sowie einige konkrete Vorschläge und Anleitungen zur Umsetzung enthält ebenfalls [Cooper S. 180ff], wobei der Autor dort nicht konsequent genug auf die Problematik der Kapselung eingeht.

Die Codefragmente 3.5.1 und 3.5.2 bieten in zwei getrennten Dateien einen kurzen Einblick in diese Art der Implementierung.

```
package Pattern;

public class Originator {

   public String state;

   public Memento createMemento() {
      Memento newMemento = new Memento();
      newMemento.setState(this.state);
      return newMemento;
   }

   public void setMemento(Memento memento) {
        this.state = memento.getState();
   }

   // ...
}
```

Codefragment 3.5.1

```
package Pattern;

public class Memento {
   // Felder sind package-protected
   private String state;
   String getState() {
      return this.state;
   }
   void setState(String state) {
      this.state = state;
   }
}
```

Codefragment 3.5.2

4 Bewertung

Aufgrund des nun folgenden Überblicks, wird dann eine begründete Empfehlung für die Herangehensweise an eine Implementieriung des Patterns mit der Programmiersprache Java abgegeben.

4.1 Stärken und Schwächen

Die für den Zweck der Implementierung des Memento-Patterns einfachste Methode – nämlich die *Static Inner Class* – ist auch die eleganteste: Ohne umständliche Handhabung von Paketen und ohne zunächst all zu viele Gedanken an Zugriffsregeln verschwenden zu müssen, kann ein übersichtliches System von Klassen gebildet werden, das zudem durch seine gemeinsame Deklaration in einer Klassendatei, zusammengehörigen Programmcode aneinander bindet. Dies bietet sich vor allem für einfache Problemstellungen an.

Die *Nonstatic Inner Class* ist auf die gleichen Aufgabenstellungen anwendbar, wobei die hier zur Verfügung gestellte Referenz auf das erzeugende Objekt weitere Möglichkeiten eröffnet, ohne dass hierfür explizit Quellcode geschrieben werden müsste. Allerdings sollte die Referenz auch notwendig sein und Verwendung finden, um den erhöhten Speicher- und Organisationsaufwand zu rechtfertigen, der durch das Halten der Referenz erzeugt wird und sich bei Anwendungen des Patterns mit einer großen Zahl an Memento-Instanzen durchaus bemerkbar machen wird.

Sobald komplexere Anforderungen an das Pattern gestellt werden, ist weder die *Static* noch die *Nonstatic Inner Class* die erste Wahl. Durch *Local* oder *Anonymous Inner Classes*, die durch geeignete Vererbung einer `interface`-Struktur auf verschiedenen Ebenen bekannte, unterschiedliche Typen haben, erzielt man weitaus höhere Flexibilität. Beispielsweise kann es nötig sein die Felder des Mementos von Akzessoren oder Mutatoren schützen zu lassen um bestimmte, die Speicherung vorbereitende Aktionen mit den Daten vornehmen zu können, damit etwa Aggregationen bei der Wiederherstellung berücksichtigt werden können, indem nicht nur die betreffenden Referenzen, sondern auch das aggregierte Objekt selbst kopiert wird bzw. seinerseits ein Memento generieren kann. Oder es kann gar eine Auslagerung der Mementos ganz aus dem Arbeitsspeicher erforderlich sein, wenn ein Memento auch nach einem Neustart des Programms oder einem Systemausfall noch zur Verfügung stehen soll.

Da selbst private Methoden und Felder der meisten inneren Klassen vor der umgebenden nicht geschützt sind, kann nur eine *Anonymous Inner Class* in Verbindung mit einer `interface`-Struktur den Originator dazu zwingen, die Mutatoren des Mementos nicht zu umgehen. Allerdings können auch durch zwei Klassen im selben Paket mit *package-protected* gesetzten Mutatoren und privaten Feldern diesen Effekt erzielen. Allerdings ist es die *Local Inner Class*, die die Voraussetzungen schafft, sehr elegant, unkompliziert und schnell unveränderliche (*immutable*) Klassen zu erzeugen, was grundsätzlich erstrebenswert ist [Bloch S. 50ff].

Die bei *Local* und *Anonymous Inner Classes* notwendige, unübersichtliche und mehrstufige Vererbung erzwingt recht komplexen Code, was für einen Neuling in der objektorientierten Programmierung problematisch sein kann. Dies sollte wohlbeachtet sein, da Design Patterns in erster Linie weniger erfahrenen Programmierern eine Hilfestellung sein sollen. Schließlich erleichtert schwer lesbarer Code auch nicht das Verständnis und ist zunächst auch schwer in eigenen Programmen zu realisieren. Positiv ist daran allerdings, dass auch diese beiden inneren Klassen sehr kompakten Code ermöglichen.

Der Einsatz von Paketen erzeugt dagegen sehr übersichlichen Code, der durch die Aufteilung in separate Dateien auch komplexe Strukturen anschaulich hält. Auch die Bedingungen für das Intakthalten der Kapselung sind hierbei leichter zu durchschauen. Der Aufwand der Entwicklung auf Basis dieses Ansatzes ist gemessen an der Zahl der Codezeilen etwas größer als bei den *Inner Classes* mit Ausnahme der *Anonymous Inner Class*, die noch umfangreicher ist.

4.2 Schlussfolgerungen

Die Nutzung der *Nonstatic Inner Class* zur Implementierung des Design Patterns Memento ist zunächst nicht von der der *Static Inner Class* zu unterscheiden, was sich aber bei Effizienzüberlegungen sofort ändert, wird die implizite Referenz nicht sinnvoll genutzt. Da im Fall dieses Patterns keine naheliegende Nutzung zu erwarten ist, muss die *Nonstatic Inner Class* als eher ungeeignet gewertet werden.

Während die *Anonymous Inner Class* durchaus ihre Vorteile hat, wiegt die Komplexität der Implementierung im Lichte der Zielgruppe aller Design Patterns zu stark um den Einsatz empfehlen zu können.

Die Möglichkeit zwei separate Klassen zu verwenden, ist ein völlig anderer Weg und darf als Fallback oder als der sprichwörtliche Mittelweg bezeichnet werden. Bei diesen Ansatz erhält man maximale Flexibilität bei vertretbarer Komplexität, die unter anderem durch die gebotene Vorsicht beim Setzen der Modifikatoren und deren eingehende Kenntnis gegeben ist. Die praktische Anwendbarkeit, das heißt den Programmieraufwand, beeinträchtigt dies aber nicht weiter. Zudem ist dies alles leicht nachzuschlagen und zu erlernen.

Die Universalität und der vergleichsweise hohe Bekanntheitsgrad der Zugriffsmöglichkeiten innerhalb von Paketen scheint eine Erklärung dafür zu sein, dass gerade diese Art der Implementierung in [Cooper] beschrieben wird, dessen kurz gefasste Programmierbeschreibungen der Patterns eher auf unerfahrene Applikationsdesigner abzielt [Cooper S. IXf (lat. 9f)]. Zudem lässt sich hiermit die Struktur des Patterns, wie es von [Gamma] beschrieben wird, am exaktesten Nachbilden.

Die *Local Inner Class* ist in der Implementierung weitaus kompakter als die soeben diskutierte Methode und dabei deutlich einfacher zu verstehen als die *Anonymous Inner Class*, wodurch sie auch für Anfänger interessant und relativ schnell durchschaubar wird. *Local Inner Classes* sind trotz alldem zwar immer noch relativ komplex, bieten dafür aber den Vorteil, einigermaßen kompakt und praktisch, von Haus aus unveränderliche Objekte zu erzeugen, wodurch die Objektbeziehungen deutlich einfacher zu verstehen sind.

Der erste Gedanke bei der Auswahl einer der genannten Vorgehensweisen bei der Implementierung des Mementos sollte in Anbetracht der leichten Handhabung und der weiteren Vorteile allerdings auf die *Static Inner Class* gerichtet sein, da mit wenig Aufwand der Problemstellung angepasste Implementierungen möglich sind. Durch eine *Static Inner Class* kann mit wenig Implementationsaufwand ein übersichtliches System von Klassen gebildet werden, das zudem durch seine gemeinsame Deklaration in einer Klassendatei logisch zusammengehörigen Programmcode aneinander bindet.

Literaturverzeichnis

[Bloch] Joshua Bloch. *Effective Java: Programming Language Guide*. Addison Wesley, 2001.

[Cooper] James W. Cooper. *Java™ Design Patterns: A Tutorial*. Addison Wesley, 2000.

[Gamma] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Horstmann] Cay S. Horstmann, Gary Cornell. *Core Java™ 2: Volume I – Fundamentals*, Fifth Edition. Prentice Hall, 2000.