Mementos basierend auf Java's Serialisierungsmechanismus

Markus Völter, voelter@acm.org, www.voelter.de

Das Mementomuster aus dem Buch der *Gang of Four* dient dazu, den Zustand eines Objektes zu externalisieren und einem anderen Objekt zur Aufbewahrung zu überlassen, ohne dass die Kapselung des Objektes aufgebrochen wird. Java bietet mittels der Serialisierung einen einfachen Mechanismus, um generisch verwendbare Mementos zu implementieren.

Ein Beispiel

In vielen Anwendungen müssen bestimmte Aktionen rückgängig gemacht werden können. Ein gängiger Ansatz hierfür ist es, alle Aktionen mittels Kommandoobjekten zu implementieren. Kommandoobjekte kapseln eine Operation als Objekt (siehe [1]). Die Ausführung der Operation dann geschieht dadurch, daß auf dem Objekt eine Operation *execute()* aufgerufen wird. Dabei sind alle Kommandos sind Unterklassen einer abstrakten Kommandoklasse:

```
public abstract class Command {
 public abstract void execute();
}
```

Da die Anwendung (ausser bei Erstellung eines Kommandos) nur die Schnittstelle von *Command* verwendet, können Kommandos unabhängig von ihrer konkreten Aufgabe ausgeführt, zwischengespeichert usw. werden.

Um Kommandos (bzw. deren Auswirkungen auf andere Objekte) rückgängig machen zu können, kann man den Kommandoklassen eine Operation *undo()* hinzufügen. Diese Operation muss die modifizierten Objekte dazu veranlassen, sich wieder in den Zustand zu versetzen, den sie vor der Ausführung des entsprechenden Kommandos inne hatten.

```
public abstract class Command {
 public abstract void execute();
 public void undo() {
    // Implementierung s.u.
  }
}
```

Der Ablauf sieht dann folgendermassen aus: Nach ihrer Ausführung werden die verwendeten Kommandoobjekte in einer Liste gespeichert. Das Rückgängigmachen einer Folge von Operationen geschieht später dadurch, dass bei den gespeicherten Kommandos in umgekehrter Reihenfolge die Operation *undo()* aufgerufen wird.

Dabei stellt sich die Frage, wie die *undo()*-Operation, also das Zurücksetzen des Zielobjektes in den alten Zustand, implementiert werden kann. Eine Möglichkeit ist, das Memento-Muster [1] einzusetzen. Das Muster wird in [1] folgendermassen beschrieben:

> *Erfasse und externalisiere den Zustand eines Objektes ohne seine Kapselung zu verletzen, sodaß das Objekt später in diesen Zustand zurückversetzt werden kann.*

Das obige Beispiel könnte also folgendermassen ablaufen (siehe Abbildung 1): Die Anwendung führt das Kommando durch Aufruf von *execute()* aus. Das Kommando bittet zunächst das Zielobjekt um ein Memento des aktuellen Zustands und merkt sich dieses. Erst dann modifiziert Das Kommandoobjekt das Zielobjekt.

Wenn dann das Kommando rückgängig gemacht werden soll, so wird bei Aufruf der *undo()*-Operation des Kommandos das dort gespeicherte Memento an das Zielobjekt zurückgegeben. Dieses setzt daraufhin seinen Zustand zurück. Diese Rückgabeoperation kann generisch implementiert werden – das Kommando muss sich nur das Zielobjekt merken und diesem das Memento zurückgeben. Das Kommando muss (und soll) keine weiteren Informationen über den Inhalt des Mementos haben.

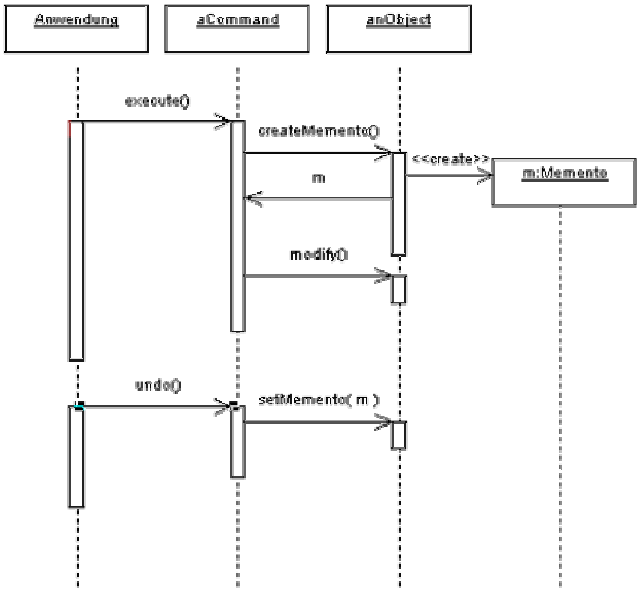

Abbildung 1: Sequenzdiagramm des Beispiels

Im Folgenden soll nun eine einfache und generische Implementation des Mementomusters für Java vorgestellt werden – die allerdings auch ihre Grenzen hat.

Was zu beachten ist

Mementos sollen den Zustand eines Objektes externalisieren, ohne die Kapselung des Objektes dessen Zustand externalisiert wird, aufzuheben. Dies bedeutet, dass dasjenige Objekt, welches ein Memento aufbewahrt, dieses auf keinen Fall modifizieren können darf. Im Idealfall kennt es nicht einmal den genauen Typ des Mementos bzw. der darin gespeicherten Daten.

Desweiteren muss sichergestellt sein, dass der Zustand im Memento wirklich als Kopie vorliegt. Andernfalls, also bei Speicherung als Referenz, würde der externe Zustand praktisch mit geändert wenn sich der Zustand des ursprünglichen Objektes ändert – sicherlich nicht der gewünschte Effekt.

Interessant ist auch die Frage, ob es erlaubt sein soll, dass ein Objekt *b* seinen Zustand aus einem Memento wiederherstellt, welches vorher von einem Objekt *a* erzeugt wurde. Damit würde dann nicht der Zustand eines Objektes zurückgesetzt, sondern Zustand kopiert.

Eine Lösung in Java

Kern des Mementomusters ist die Externalisierung des internen Zustands eines Objektes ohne dass derjenige, der den Zustand aufbewahrt, diesen verändern kann. Diese Externalisierung muss per Kopie und nicht per Referenz passieren. In Java lassen sich diese beiden Forderungen sehr leicht unter einen Hut bringen – unter Verwendung von Java's Serialisierungsmechanismus. Dabei wird der Zustand eines Objektes serialisiert und als Byte-Array nach aussen gegeben, bei der vorgeschlagenen Implementierung nach vorherigem Einpacken in eine Instanz der Klasse *Memento*.

In diesem einfachsten Fall gehen wir davon aus, dass ein Memento immer den gesamten Zustand eines Objektes enthält und nicht nur inkrementelle Wiederherstellungsinformationen. Um den Vorgang der Serialisierung weiter zu vereinfachen, kann der Zustand eines Objektes bereits von Anfang an als separates Objekt innerhalb des eigentlichen Objektes gespeichert werden. Dass heisst, statt

```
public class TestObject {

    protected int a;
    protected int b;
    protected int c;

    public TestObject() {
    }

    public int doSomething() {
        return a*b*c;
    }

    // weitere Operationen...
}
```

würde man folgendermassen vorgehen:

```
public class TestObject {

    class State implements Serializable {
        public int a;
        public int b;
        public int c;
    }

    private State state;

    public TestObject() {
        state = new State();
    }

    public int doSomething() {
        return state.a * state.b * state.c;
    }

    // weitere Operationen...

}
```

Übrigens: die direkte Serialisierung der Klasse *TestObject* im Rahmen eines Mementos ist kein gangbarer Weg. Bei Deserialisierung wird ja eine neues Objekt erstellt, d.h. alle Referenzen auf das bisherige werden ungültig. Man müsste also alle Refernzen innerhalb eines Programmes ändern – im Allgemeinen nicht praktikabel. Im obigen Beispiel mit der internen *State*-Klasse ist dies kein Problem, da es nur eine einzige Referenz auf das S*tate*-Objekt gibt: die Variable *state*.

Ein Vorteil der Verwendung von inneren Klassen in diesem Zusammenhang ist der, dass die Mementodaten selbst dann nicht sinnvoll verwendet werden kann wenn es von einem anderen Objekt deserialisiert wird: die Klasse ist nicht verfügbar und daher ist ein Downcast nicht möglich.

Nachdem nun die Mechanik geklärt ist, ist jetzt die Kosmetik an der Reihe. Die Verwendung der Mementos sollte möglichst einfach sein. Sowohl Klassen, die ihren Zustand per Memento externalisieren können, als auch deren Benutzer, sollten nichts vom Serialisierungsprozess mitbekommen – dieser Vorgang sollte automatisiert und versteckt werden. Genau da liegt auch einer der Vorteile des auf Serialisierung beruhenden Ansatzes: Er ist insofern generisch, als die Serialisierung prinzipiell für alle Klassen funktioniert.

Zunächst soll nun also eine Schnittstelle eingeführt werden, die von allen Klassen, die Mementos erzeugen können implementiert werden muss:

```
public interface MementoCapable {
    public Memento createMemento();
```

```
        public void setMemento(Memento o) throws
                    WrongTypeException,WrongInstanceException;
    }
```

Die Operation *createMemento()* liefert – naheliegenderweise – ein Memento des aktuellen Zustands des Objekts zurück. Die Operation *setMemento(Memento)* setzt den Zustand des Zielobjekts auf den im Memento mitgelieferten Zustand. Dabei kann es zu einigen Problemen kommen, wenn der Erzeuger des Mementos (nennen wir in *a*) nicht gleich dem ist, der seinen Zustand vom Memento wiederherstellen soll (*b*):

- Die Klasse von *b* kann eine andere sein als die von *a*.
- Die Klassen von *b* kann eine Unterklasse der Klasse von *a* sein.
- *a* kann ein anderes Objekt sein als *b*.

Je nachdem welcher Fall eintritt – und ob dies in einer bestimmten Anwendung eine Fehlersituation darstellt – kann dann in *setMemento()* eine *WrongTypeException* oder eine *WrongInstanceException* geworfen werden.

Die Klasse *Memento* selbst dient dazu, den serialisierten Zustand zu speichern, zusammen mit einigen Zusatzinformationen wie z.B. eine Referenz auf den Erzeuger und einigen (später beschriebenen) Hilfsoperationen. Abb. 2 zeigt die entsprechenden Klassen. *Memento* besitzt einen Konstruktor, der eine Referenz auf den Erzeuger sowie auf das zu serialisierende Objekt als Argumente erwartet.

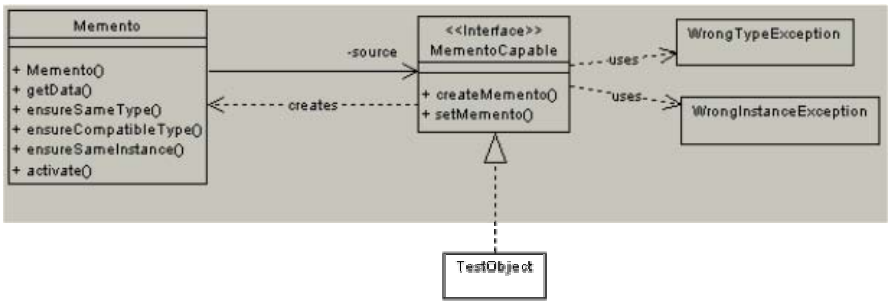

Abbildung 2: Klassendiagramm der vorgeschlagenen Lösung

Aus Sicht des Clients (also z.B. eines Kommandoobjekts) ist die Verwendung eines Mementos extrem einfach:

```
TestObject o = new TestObject();
Memento mem = o.createMemento();
  // tue irgend etwas anderes...
try {
    o.setMemento( mem );
} catch (Exception ex ) {
 // behandeln der Exception
```

```
}
```

Statt *o.setMemento(mem)* kann auch *mem.activate()* verwendet werden – dies hat denselben Effekt. Auch die Implementierung der Schnittstelle *MementoCapable* ist sehr einfach, wenn der Zustand (wie oben beschrieben) als separates Objekt gehalten wird:

```
public class TestObject implements MementoCapable {

    class State implements Serializable {
        // ...
    }

    private State state;

    public Memento createMemento() {
        return new Memento( this, state );
    }

    public void setMemento( Memento m )
          throws WrongTypeException, WrongInstanceException {
          state = (State)m.getData();
    }

}
```

Wenn die oben beschriebenen Probleme mit Instanzen und Klassen in der spezifischen Anwendung einen Fehlerfall darstellen, so können die vorhandenen Hilfsmethoden verwendet werden, wie das folgende Beispiel verdeutlicht:

```
public class TestObject implements MementoCapable {

    // ...

    public void setMemento( Memento m )
        throws WrongTypeException,  WrongInstanceException {
           // ueberprueft, ob das Memento von dem gleichen
           // Objekt erzeugt wurde. Wenn nicht, dann wird eine
           // WrongInstanceException geworfen. checkSameType()
           // und checkCompatibleType() funktionieren
           //prinzipiell genauso.
        m.ensureSameInstance( this );
        state = (State)m.getData();
    }

}
```

Wir können nun das Eingangsbeispiel – die Kommandoobjekte – zu Ende bringen. Die hier vorgestellte Implementierung der Klasse *Command* setzt voraus, dass nur ein Zielobjekt von dem Kommando betroffen ist (andernfalls: siehe nächste Ausgabe des JavaMagazins).

```
public abstract class Command {
   MementoCapable target;
```

```
  Memento memento;
 public Command( MementoCapable theTarget ) {
      target = theTarget;
 }
 public void execute() {
    memento = target.createMemento();
    doExecute();
  }
 public void undo() {
     memento.activate();
  }
 public abstract void doExecute();
}
```

Diskussion der Lösung

Natürlich ist die Lösung nicht immer so einfach wie hier im Beispiel. Java's Serialisierung hat ihre eigenen Tücken. Wie wird z.B. vorgegangen, wenn als Teil des Objektzustandes ein anderes Objekt referenziert wird, dessen Inhalt nicht serialisiert werden soll? Oder was passiert bei Zirkelbezügen zurück zum Ursprungsobjekt? Hier muss sicherlich von Anwendung zu Anwendung überprüft werden, wie die Externalisierung des Zustandes erfolgt. Gegebenenfalls kann statt der Serialisierung auch Java's *clone()*-Mechanismus verwenden – aber auch dieser hat seine eigenen Tücken.

Generell ist festzustellen, dass man die Implementierung der Externalisierung jederzeit ändern kann, die restliche Anwendung bleibt davon unbeeinflusst. Die öffentliche Schnittstelle von *Memento* muss selbstverständlich erhalten bleiben, aber die Informationen im Memento koennen von der Mementoklasse selbst bestimmt werden. Man kann sich z.B. durch Unterklassenbildung eigenen Mementotypen erstellen, die andere Externalisierungsmechanismen verwenden.

Das schöne an der hier vorgestellten Lösung ist jedoch die Tatsache, dass sie prinzipiell mit beliebigen (serialisierbaren) Objekten arbeiten kann, ohne dass zusätzlicher Programmieraufwand getrieben werden muss um das Memento zu erstellen. Die Lösung eignet sich also sehr gut als Anfang. Später im Laufe eines Projektes können dann – transparent für die Anwendung – neue, spezialisierte Mementoklassen erstellt werden.

Der Code für diesen Artikel findet sich wie immer auf der Begleit-CD oder auf www.voelter.de.

Referenzen

[1] Gamma, Helm, Johnson, Vlissides, *Entwurfsmuster*, Addison-Wesley 1995