Objektorientierte Programmierung für Dummies

Marcus Bäckmann

mitp-Verlag, ISBN 3-8266-2984-1

Nachtrag zu Kapitel 23

Die folgenden Teile erscheinen nicht im Buch, da sie aus Platzgründen gekürzt wurden – der Text würde eigentlich ab Seite 424 im Buch erscheinen... leider wurde die Sache ein wenig dick.

23Von virtuellen Konstruktoren, Objektfabriken und noch mehr Virtualität

In diesem Kapitel

- bekommen Sie ein Beispiel, wie man das Innenleben von Objekten ändert, ohne dass man es von außen sieht

<Seiten 411 – 424>

23.2 Des Kaisers neue Kleider

Menschen ändern sich, Objekte auch. Nehmen Sie eine Spielfigur, die in zwei Modi unterwegs sein kann: laufend und schwimmend. Oder kämpfend und bauend. Wer es bodenständiger mag, nimmt einen Server, der an einem Netzwerk hängt. Mal wartet der Server auf Verbindungen, mal besteht eine aktive Verbindung, mal geht der Server in einen Fehlerzustand.

Alle diese Dinge haben eines gemeinsam, in Abhängigkeit seines Zustands ändert ein Objekt sein Verhalten. Nehmen Sie an, dass sich zwei Zustände durch zwei Klassen beschreiben lassen, so könnte man das ja ungefähr so aussehen lassen:

```
class State1 : public BasisState;

class State2 : public BasisState;

BasisState* pState = new State1();

pState->doSomething();

delete pState;

pState = new State2();

pState->doSomething();
```

Über eine virtuelle Methode kann das Objekt `pState` sich dann je nach Zustand richtig verhalten. Eine gute Lösung? Leider nein. Bedenken Sie, dass andere Objekte bereits Zeiger und Referenzen auf dieses Objekt besitzen können. Und nun wollen Sie alle bestehenden Zeiger und Referenzen durch neue ersetzen? Das geht nicht. Das Objekt selbst darf sich für einen Außenstehenden nicht verändern.

Ein Entwurfsmuster, das dieses Problem löst, nennt sich *State-Pattern*. Um ein Objekt seinen inneren Zustand ändern zu lassen, benötigt man insgesamt vier Klassen (für zwei Zustände). Wollte man drei Zustände modellieren, bräuchte man fünf Klassen (für n Zustände immer 2 + n Klassen).

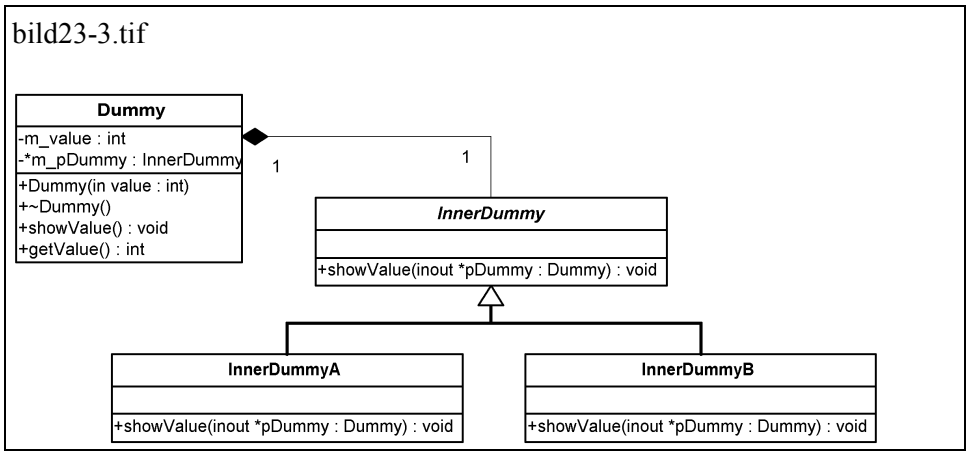

Abbildung 23.3: Realisierung des State-Pattern

Abbildung 23.3 zeigt den Zusammenhang zwischen den vier Klassen. Von außen wird nur die Klasse `Dummy` sichtbar. Intern gibt es für jeden Zustand eine eigene Klasse, diese sind von einer gemeinsamen Basisklasse abgeleitet. Jede Methode von `Dummy` hat eine entsprechende virtuelle Methode in `InnerDummy`.

Betrachten Sie ein Programmbeispiel zunächst aus Sicht des Aufrufers.

```
#include <iostream>

#include "dummy.h"

using namespace std;

int main()

{

   Dummy dummy(5);

   for (int i = 0; i < 1000; i++)

   {

      dummy.doSomething();

      dummy.showValue();

   }

   return 0;

}
```

Listing 1: *KAP23/STATE.CPP*

Wie Sie sehen können, erfährt man als Nutzer der Klasse `Dummy` nichts von deren Eigen- und Innenleben. Das ist schon mal ein guter Beginn. Nun geht es weiter zu den inneren Klassen.

```
#ifndef _DUMMY_H

#define _DUMMY_H

class Dummy;

class InnerDummy

{

public:

   virtual void showValue(Dummy* pDummy) = 0;

};

class InnerDummyA : public InnerDummy

{

public:

   virtual void showValue(Dummy* pDummy);

};

class InnerDummyB : public InnerDummy

{

public:

   virtual void showValue(Dummy* pDummy);

};

class Dummy

{

public:

   Dummy(int value);

   ~Dummy();

   void showValue()

   {

      m_pDummy->showValue(this);

   }
```

```
   void doSomething();
   int getValue() const
   {
      return m_value;
   }
private:
   class InnerDummy* m_pDummy;
   int m_value;
};
#endif
```

Listing 1: *KAP23/DUMMY.H*

Beachten Sie, dass es für die Methode `Dummy::showValue` auch noch virtuelle Methoden in `InnerDummy` (und natürlich auch den abgeleiteten Klassen) gibt. Zudem besitzt `Dummy` einen Zeiger auf `InnerDummy` – wozu sehen Sie, sobald Sie sich die Implementation betrachten.

```
#include <iostream>
#include "dummy.h"
using namespace std;
void InnerDummyA::showValue(Dummy* pDummy)
{
   cout << "Jetzt bin ich A! "
        << pDummy->getValue() <<endl;
}
void InnerDummyB::showValue(Dummy* pDummy)
{
   cout << "Jetzt bin ich B! "
        << pDummy->getValue() << endl;
}
Dummy::Dummy(int value)
      : m_value(value)
```

```
{
   m_pDummy = new InnerDummyA();
}
Dummy::~Dummy()
{
   delete m_pDummy;
}
void Dummy::doSomething()
{
   delete m_pDummy;
   if (rand() % 2)
   {
      m_pDummy = new InnerDummyA();
   }
   else
   {
      m_pDummy = new InnerDummyB();
   }
}
```

Listing 1: *KAP23/DUMMY.CPP*

Jede der `show`-Methoden wird in `InnerDummyA` und `InnerDummyB` anders implementiert, das ist auch logisch, schließlich ändert sich mit dem Zustand das Verhalten. Im Konstruktor von `Dummy` wird zunächst ein `InnerDummyA`-Objekt erzeugt, das Objekt befindet sich zu Beginn im Zustand »A«. Ruft man von außen `Dummy::showValue` auf, so wird in Wirklichkeit die virtuelle Methode des zurzeit eingehängten `InnerDummy`-Objekts aufgerufen. Je nach Zustand des Objekts `Dummy` also eine andere Methode.

In der Funktion `doSomething` wird ein Zustandswechsel simuliert, hier einfach durch die Auswertung einer Zufallszahl. Je nach Ergebnis wechselt das Dummy-Objekt mal in den Zustand »A«, mal in den Zustand »B«. Sie sehen an der Bildschirmausgabe, dass der Aufrufer immer `showValue` aufruft, die Ausgabe aber dennoch unterschiedlich sein kann.

DEFINE

- Das State-Pattern ermöglicht es einem Objekt, je nach innerem Zustand ein anderes Verhalten zu zeigen.
- Implementiert wird das State-Pattern dadurch, dass man für jeden Zustand eine eigene Klasse realisiert. Diese besitzen eine gemeinsame Basisklasse. Für jede Funktion, deren Verhalten sich ändern soll, wird in der Basisklasse eine virtuelle Methode angelegt, die in den abgeleiteten Klassen überschrieben wird. Ruft man von außen eine Methode des Objekts auf, wird in Wirklichkeit eine Methode des gerade aktiven inneren Zustand-Objekts aufgerufen.
- Interessant ist das State-Pattern auch, falls jeder Zustand des Objekts noch eigene Variablen benötigt. Packt man alles in eine einzige Klasse, so landen letztlich viele Variablen in einer einzigen Klasse, die aber je nach Zustand immer nur einen Teil benötigt. Bei der State-Lösung ist jede Zustand-Klasse so schlank wie möglich.

TECHNICAL

- Rein technisch sieht das State-Pattern einem anderen Trick sehr ähnlich, dem so genannten *Pimpl-Idiom*. Dort besitzt eine äußere Klasse nur einen Zeiger auf eine innere Klasse und sonst nichts, ganz ähnlich zum State-Pattern:

```
class Inner;

class Outer

{

public:

   Outer()

   {

      m_pImpl = new Inner();

   }

   void doSomething()

   {

      m_pImpl->doSomething();

   }

private:

   Inner* m_pImpl;
```

```
};
```

Auch hier ist es so, dass die Klasse `Inner` genau die gleichen Methoden besitzt wie die äußere Klasse `Outer`, aber nicht, um Zustände zu modellieren. Der Gag an Pimpl ist, dass man die komplette Klasse `Inner` in einer anderen Datei implementieren kann. Hat Sie nicht auch schon oft gestört, dass bei der Ergänzung einer privaten Variablen in einer Klasse plötzlich alles neu kompiliert werden muss? Bei Pimpl wird dies vermieden, da sämtliche privaten Variablen und Methoden nur in `Inner` realisiert werden. Die Klasse `Outer` ändert sich dadurch nicht und man kann `Inner` verändern, ohne das komplette Projekt neu kompilieren zu müssen, es reicht aus, `Inner` neu zu übersetzen. Pimpl nennt man deshalb auch *Compiler-Firewall*.

- Der Begriff Pimpl kommt von *pointer to implementation* und erinnert daran, dass ein Zeiger auf das tatsächliche Implementations-Objekt benutzt wird, um die Klasse zu realisieren. Das Pimpl-Idiom wird oft in größeren Open-Source-Projekten verwendet, in denen viele Leute viele verschiedene Dateien bearbeiten.