Christian Silberbauer

Framework für Zustandsorientierte Programmierung

Diplomarbeit

Examicus

FACHHOCHSCHULE REGENSBURG
UNIVERSITY OF APPLIED SCIENCES
HOCHSCHULE FÜR
TECHNIK
WIRTSCHAFT
SOZIALWESEN

Diplomarbeit

Framework für Zustandsorientierte Programmierung

- Einsatz bei der Implementierung eines Service-Containers -

Christian Silberbauer

Fachhochschule Regensburg

Fakultät Informatik/Mathematik

Sommersemester 2007

Vorwort

Object-oriented software development requires that individual developers have unscheduled critical masses of time in which they can think, innovate, and develop, and meet informally with other team members as necessary to discuss detailed technical issues. The management team must plan for this unstructured time.

Grady Booch, Object-Oriented Analysis and Design

Inhaltsverzeichnis

1. Einleitung

Die Zustandsorientierung und das Zustandsframework

Diese Ausarbeitung beschreibt ein Framework zur Unterstützung der *Zustandsorientierten Programmierung*. Mit Zustandsorientierter Programmierung ist hier ein Programmierparadigma gemeint, das Variablen und deren Zustände in den Mittelpunkt stellt. Die Variablen werden funktional in Beziehung gesetzt. Ändert sich der Zustand einer Variablen, so werden die davon abhängenden Variablen angepasst.

Die Zustandsorientierung kann auf der Objektorientierung basieren. Genau dies ist beim hier beschriebenen Framework (im Folgenden auch *Zustandsframework* genannt) der Fall. Das Zustandsframework ist in der objektorientierten Programmiersprache C# entwickelt. Es umfasst hauptsächlich sog. *Zustandsklassen* und erlaubt eine Zustandsorientierte Programmierung, indem Attribute von Klassen durch jene Zustandsklassen gekapselt sind. Die Zustandsklassen fungieren gewissermaßen als Container für Attribute und kontrollieren deren Zugriff, insbesondere deren Manipulationen. Hierzu ein einfaches Beispiel:

```
class Boss
{
    AggregateState<double>
        staffPerformance;

    void PerformanceChanged()
    {
        if (staffPerformance < 0.98)
            FireSomeEmployees();
    }
}
```

```
class Employee
{
    State<double> performance;

    void Work() { … }
}
```

Mitarbeiter (`Employee`) haben eine Arbeitsleistung (`performance`). Deren Chef (`Boss`) wacht über ihre durchschnittliche Arbeitsleistung (`staffPerformance`). Sinkt diese unter 98%, so werden ein paar Mitarbeiter entlassen[1]. Die beiden Performance-Attribute sind eigentlich vom Typ `double`, werden aber durch eine sog. *Einfache Zustandsklasse* (`State`) bzw. eine *Aggregierte Zustandsklasse* (`AggregateState`) gekapselt. Im Code-Listing nicht abgebildet ist die Konfiguration

[1] Dieser Chef ist zwar einfach gestrickt, dafür aber berechenbar.

der entsprechenden Zustandsobjekte. Insbesondere die Aggregierten Zustandsobjekte müssen durch eine Aggregatsfunktion zur Durchschnittbildung parametrisiert werden und die zugehörigen Einfachen Zustandsobjekte müssen als Kind-Zustände definiert werden. Darüber hinaus ist die Methode `PerformanceChanged()` als sog. *Transitions-Listener* für `staffPerformance` zu definieren. Ändert sich also die durchschnittliche Arbeitsleistung der Belegschaft bzw. findet diesbezüglich ein *Zustandsübergang* bzw. eine *Transition* statt, so wird `PerformanceChanged()` aufgerufen.

Unter Verwendung des Zustandsframeworks lassen sich Attribute durch deklarative Programmierung funktional miteinander in Verbindung bringen, Zustandsübergänge lassen sich verfolgen und, wie später noch ersichtlich wird, lassen sich vom Zustandsobjekt vorgesehene Zustandsübergänge sogar blockieren. Letzteres bedeutet, dass der Client bei Transitionen die Möglichkeit hat, entsprechende Aktivitäten durchzuführen. Sind diese nicht erfolgreich, so kann der Client bestimmen, dass das Zustandsobjekt in seinem alten Zustand verharrt.

Alles in Allem ist das Zustandsframework ein hilfreiches Werkzeug, um Zusammenhänge zwischen Attributen einfach herzustellen und damit eine zustandorientierte Sichtweise in der Programmierung zu fördern.

Der Kontext

Entstanden ist das Framework in Folge einer Beschäftigung des Autors bei Siemens Corporate Research in Princeton in den USA. Dort wird ein Verteiltes System entwickelt. Der Service-Container des Systems hat ein komplexes Lifecycle-Management für seine Komponenten zu realisieren. Entsprechend ist die Ausrichtung des Zustandsframeworks. Seine Zustandsklassen sind nach den Anforderungen des Service-Containers von Siemens entworfen und sind daher keine allgemeine Sammlung von Zustandsklassen für Zustandsorientierte Programmierung. Dennoch sind vor allem die primitiveren Zustandsklassen recht universell einsetzbar. Außerdem ist eine Erweiterung des Frameworks um weitere Zustandsklassen sehr einfach möglich.

Das Flaggschiff: Integrierte Strebsame Zustandsobjekte

Der Schwerpunkt dieser Ausarbeitung bei der Definition von Zustandsklassen liegt in der Herleitung von *Strebsamen Zustandsobjekten* und den darauf basierenden *Integrierten Strebsamen Zustandsobjekten*. Letztere werden verwendet, um die Zustände der Komponenten in ihrem Lebenszyklus zu steuern. Die Bezeichnung *strebsam* rührt daher, dass durch einen sog. *angestrebten Zustand* festgelegt werden kann, welcher Zustand für das Zustandsobjekt wünschenswert wäre. Strebsame Zustandsobjekte haben aber Abhängigkeiten zu anderen Zustandsobjekten – sog. *Restriktive Zustandsobjekte* –, die entsprechende Zustandsübergänge verhindern können. Ob der angestrebte Zustand zum tatsächlichen Zustand wird, hängt also davon ab, ob die restriktiven Zustandsobjekte dies zulassen. Die Bezeichnung *integriert* meint, dass es sich um ein Netz von gleichartigen Zustandsobjekten handelt, die sich gegenseitig beeinflussen.

Zum Beispiel beim Service-Container von Siemens haben Komponenten nacheinander die Zustände *Deployed*, *Created*, *Initialized* und *Active*, dann ist die Komponente *hochgefahren* und bereit für ihre Verwendung. Im Übrigen wird aus technischer Sicht das entsprechende Attribut als Enumeration abgebildet. Mit jedem Übergang zwischen zweier dieser Zustände sind notwendige Aktivitäten verbunden, die jeweils erfolglos durchgeführt werden können und daraufhin das Hochfahren der Komponente ggf. abgebrochen werden muss. Zudem gilt, dass Komponenten von anderen Komponenten abhängig sein können und beim Hochfahren sichergestellt sein muss, dass eine Komponente erst *Active* wird, bevor ihre abhängige Komponente *Active* werden kann. Indes können andere Transitionen unabhängig von den Komponenten-Abhängigkeiten passieren. Und dieser Service-Container hat eine weitere Anforderung: Zwar muss eine Komponente beim Hochfahren u.U. darauf warten *Active* zu werden, bis ihre Abhängigkeiten *Active* sind, hingegen darf eine Komponente beim Herunterfahren *nicht* blockiert werden. Das bedeutet, dass Komponenten ihre abhängigen Komponenten ggf. zwingen müssen auf *Initialized* herunterzufahren, sodass sie selbst zum Herunterfahren im Stande sind.

Insgesamt also eine komplexe Anforderung an eine Zustandsklasse. Hier ist besonders ersichtlich, dass eine Auslagerung solcher Funktionalitäten in Zustandsklassen sinnvoll ist, statt diese direkt in jene Klassen zu implementieren, in

denen sie gebraucht werden. Denn diese haben i.d.R. andere Dinge zu erledigen und würden sonst unnötig überladen werden. Die Integrierte Strebsame Zustandsklasse ist modular implementiert. Des Weiteren kann sie durch eine überschaubare Schnittstelle einfach verwendet werden.

Der Aufbau der Diplomarbeit

In der vorliegenden Ausarbeitung werden zunächst in Kapitel 2 Grundbegriffe wie *Zustand* und *Framework* diskutiert, bevor im Kapitel 3 das Verteilte System von Siemens vorgestellt wird. Insbesondere wird dabei auf den Service-Container und dessen Lifecycle-Management eingegangen, denn dies soll durch das Zustandsframework letztlich realisiert werden. Kapitel 4 schafft dann aus Sicht des Service-Containers die Motivation für die Entwicklung des Zustandsframeworks. Im folgenden Kapitel wird dieses auf konzeptioneller Ebene vorgestellt. Erst wird auf das grundlegende Modell eines Zustandsobjektes eingegangen, dann werden die einzelnen Arten von Zustandsobjekten vorgestellt. Kapitel 6 behandelt die Implementierung des Frameworks. Für ein gutes Verständnis wird Bezug zu angewendeten Design Patterns genommen, sowie zu Programmierprinzipien, vor allem objektorientierter Natur. Anschließend wird die Programmierung des Service-Containers unter Anwendung der Zustandsklassen gezeigt. Und zum Abschluss beschreibt Kapitel 8 mögliche Erweiterungen des Frameworks, die insbesondere auf ein allgemeineres Einsatzgebiet abzielen, bis dann Kapitel 9 mit der Einordnung des beschriebenen Frameworks in das Paradigma der Zustandsorientierung die Arbeit abrundet.