Hochschule für angewandte Wissenschaften
Fachhochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Schwerpunktseminar

State und Chain of Responsibility

Erarbeitet im Seminar der Vertiefungsrichtung Technische Informatik des Studiengangs Informatik

Ilhan Diler

Eingereicht am: 26 November 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 State

1.1 Kurzbeschreibung

Das Entwurfsmuster State verfolgt das Ziel, mittels Delegation und Komposition Zustandsmaschinen zu realisieren, die zum einen ermöglicht, Zustände in eigenen Klassen zu kapseln, um die Erweiterbarkeit der Zustände zu erhöhen und den Code zu strukturieren, dass die Übersichtlichkeit gewährleistet wird.

1.1.1 Definition

„Ermöglicht es einen Objekt sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seinen internen Zustand geändert hat“. [EF04]

1.2 Motivation

Vor Zeiten der objektorientierten Programmierung war das Abbilden von Zustandsautomaten nur über IF-Strukturen möglich. Dabei wurde der Programmiercode sehr unübersichtlich und einzelne SZustände konnten nur über Hilfsvariablen definiert werden. Diese Form erschwert dem Programmierer die Strukturierung und Erweiterung seiner Anwendung. Mit diesem Pattern soll dem Anwender die Möglichkeit gegeben werden, mithilfe der Objektorientierung eine Instanz zu erzeugen, das sein Verhalten ändert, wenn sich sein interner Zustand ändert. [EF04]

1.3 Struktur

Für das State Pattern bestehen unterschiedliche Varianten, um einen Zustandsübergang darzustellen, wie im Folgenden detailliert beschrieben wird.

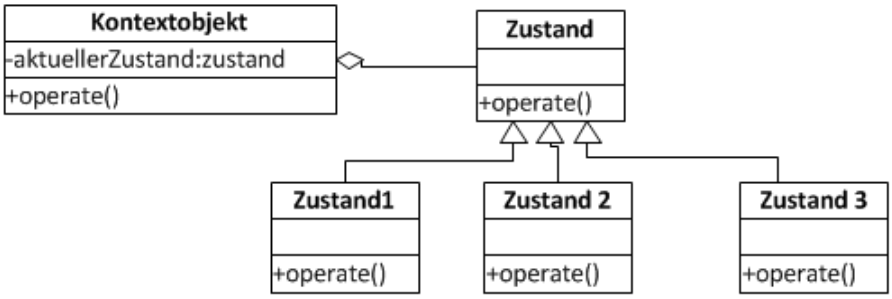

Abbildung 1.1: Darstellung State Pattern

1.4 Erstellen der Zustände

Zustandsübergänge können durch verschiedene Varianten ausgeführt werden. Jedoch soll in Betracht gezogen werden, dass abhängig von der Auswahl, Einschränkungen hinsichtlich Erweiterungen der Anwendung resultieren.

1.4.1 Zustand vom Kontext bestimmt

Diese Variante stellt eine einfache Implementierung dar. Es besteht keine Abhängigkeit zwischen dem Kontextobjekt und den konkreten Zuständen. Jedoch wird die Struktur des Kontexts viel komplexer, sodass diese Klasse gegen Veränderungen unflexibler wirkt. Daher sollte dieser Weg nur dann gewählt werden, falls der Automat überschaubar ist und kaum noch Änderungen am Kontextobjekt vorgesehen sind. [Hau09]

1.4.2 Zustandsübergänge vom Zustand bestimmt

Diese Variante wird sehr häufig benutzt und ist besonders dafür geeignet, wenn der Folgezustand dynamisch erzeugt werden soll. Bei dieser Ausführung wird der Zustandswechsel im jeweils konkretem Zustandsinstanz übernommen. Zusätzlich wird eine Referenz auf das Kontexobjekt benötigt.[Hau09]

1.4.3 Definition der Zustände im Kontext als Attribut

Beim Erzeugen des Kontext werden gleichzeitig auch alle konkreten Zustände erstellt und als Attribut im Kontext gehalten. Für sehr häufig auftretende Zustandsübergänge ist diese Variante vom großen Vorteil, da nicht bei jedem Übergang das neue Objekt erstellt und das Alte gelöscht werden muss.[Hau09]

1.4.4 Zustände bei Bedarf erzeugen

Bei dieser Ausführung wird die konkrete Zustandsinstanz erst beim Zustandsübergang erzeugt. Um dabei einen Speicherüberlauf zu vermeiden, muss die Instanz beim Verlassen des Zustands gelöscht werden. Diese Variante ist relativ einfach zu implementieren und sollte eingesetzt werden, falls keine häufigen Zustandsübergänge auftreten.[Hau09]

1.4.5 Bestimmung des Zustandsübergang über Rückgabewert

Bei dieser Ausführung gibt der aktuelle konkrete Zustand eines Objekts den Folgezustand als Rückgabewert zurück. Der eigentliche Zustandswechsel wird zwar im Kontext vorgenommen, aber im Vorgängerzustand erstellt. Als Konsequenz kann bei dieser Ausführung die Funktionen der konkreten Zustände keine weiteren Ergebnisse zurückliefern. [Hau09]

1.5 Beispiel

Mit diesem Beispiel soll die Struktur des State Pattern verdeutlich werden. In einem Speditionsunternehmen werden Lebensmittel auf Paletten kommissioniert, siehe Abb.1.2. Der Lagerarbeiter verfolgt seine Aufträge über ein am Hubwagen integriertem Bediengerät. Dazu muss er zunächst über seine Eingabedaten identifiziert werden. Ist er bereits angemeldet, kann er mit der Bearbeitung der am Monitor angezeigten Aufträge beginnen. Sobald der Kommisionierer seinen Auftrag beendet hat, muss er dies am System bestätigen bzw. den Auftrag abschließen. Befindet sich das Bediengerät im Zustand *AuftragErledigt*, wird überprüft ob der Lagerarbeiter mehr als 30 Aufträge ausgeführt hat. Falls dies nicht der Fall ist, so soll er einen weiteren Auftrag bearbeiten, ansonsten meldet sich das Bediengerät vom System automatisch ab.

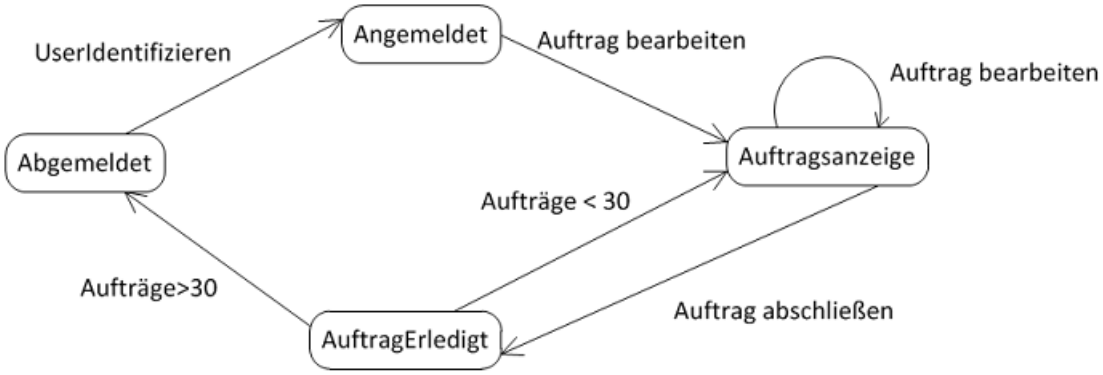

Abbildung 1.2: Zustandsdiagramm für eine Auftragsabwicklung

1.6 Strukturierte Implementierung des Entwurfs

Im Vergleich zum objektorientierten Entwurf, wie in Abb.1.3, besteht diese Implementierung lediglich nur aus einer umfangreichen Klasse, in der ein Aufzählungstyp mit seinen definierten Zuständen dargestellt wird.

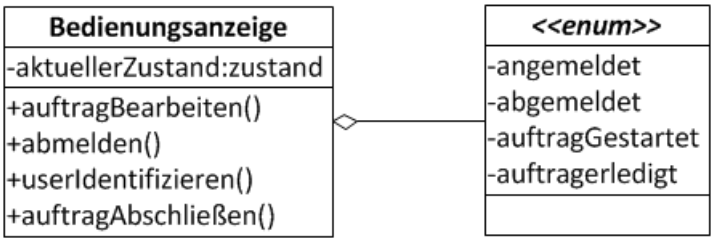

Abbildung 1.3: Klassendiagramm der strukturierten Programmierung

1.7 Objektorientierte Implementierung des Entwurfs

Wie im objektorientiertem Entwurf dargestellt, siehe Abb. 1.4 wurden alle konkreten Zustände mit ihren vererbten Funktionen aus der Zustandsklasse ausgelagert. So können alle Zustände eindeutig voneinander getrennt dargestellt werden. Hier wurde die Variante ausgewählt, den Folgezustand über den aktuellen Zustand des Objekt, entsprechend über das Bediengerät zu bestimmen. Da Zustandsübergänge dynamisch ermittelt werden müssen, erweist sich diese Variante als sehr geeignet. Es muss bei dieser Implementierung lediglich bei der Ausführung eines Auftrages deren Anzahl ermittelt werden, um entweder in den Zustand *Abgemeldet* oder *AuftragErledigt* zu wechseln.

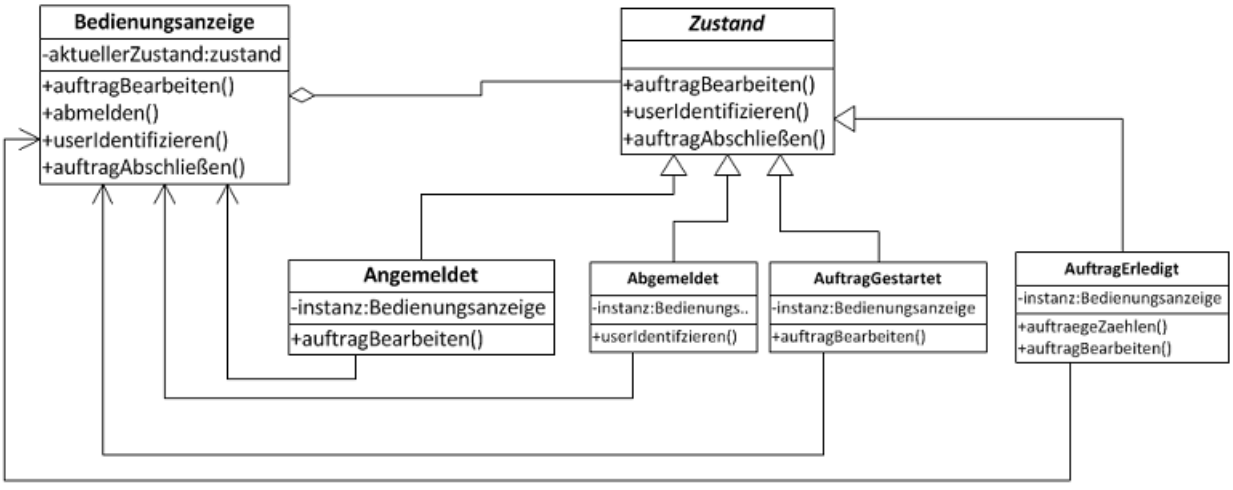

Abbildung 1.4: Klassendiagramm des objektorientiertem Entwurfs

1.8 Bewertung

1.8.1 Vorteile

Anders als bei der strukturierten Programmierung der Zuständen kann mit diesem Design Pattern Klassen gekapselt und somit alle relevanten Informationen konkret zusammengefasst werden. Dadurch wird auch die Übersichtlichkeit des Codes erhöht. Da eine Umsetzung auch ständig Änderungen ausgesetzt werden kann bzw. erwünscht wird, können anhand dieses Entwurfsmusters weitere Zustandsklassen hinzugefügt werden, ohne Änderungen in der Programmlogik zu übernehmen. [EF04]

1.8.2 Nachteile des State Patterns

Durch Kapselung wird der Code einerseits strukturiert und übersichtlich dargestellt und mehr Flexibilität hinsichtlich Erweiterbarkeit geboten. Auf der anderen Seite kann die erhöhte Anzahl der Klassen auf diese Weise die Übersichtlichkeit des Gesamtentwurfs beeinträchtigen. Für kleinere Anwendungen wird der Aufwand des State Patterns als aufwändig eingeschätzt. Daher sollte in diesem Fall die Variante mit der strukturierten Programmierung bevorzugt werden.

2 Chain of Responsibility

2.1 Kurzbeschreibung

Chain of Responsibility ermöglicht eine Anfrage weiterzugeben, bis ein geignetes Objekt für die Verarbeitung einer Anfrage gefunden wird. Mit diesem Design Pattern wird der Empfanger und Sender bei der Erstellung einer Anfrage voneinander entkoppelt.

2.2 Definition

„Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an einer Kette entlang, bis ein Objekt sie erledigt“. [EG94]

2.3 Beschreibung

Sobald der Client eine Anfrage erstellt oder ein Event auslöst, wird die Instanz der Verantwortungskette angesprochen. Diese delegiert ihre Nachricht an die nachfolgende Instanz, falls diese nicht von ihr bearbeitet werden kann. Die Schnittstelle Handler stellt eine Methode zur Verfügung, für die Instanz, um eine Anfrage zur nachfolgenden Instanz weiterzuleiten.

2.4 Absicht

Mit diesem Pattern soll dem Programmierer die Möglichkeiten geöffnet werden, seinen Code so zu strukturieren, damit die Kopplungen zwischen Objektinstanzen reduziert wird.

2.5 Struktur

Chain of Responsibility verfolgt das Ziel, die Programmstruktur wesentlich zu vereinfachen. Damit eine Nachricht durch die potentiellen Empfängerinstanzen bearbeitet werden können, wird dazu eine abstrakte Klasse Handler, wie in 2.1 dargestellt, implementiert. Zur Erstellung der Verantwortungskette muss die hierarchische Anordnung der Objekte für den Empfänger bekannt sein. Dies kann sowohl linear als auch hierarchisch erfolgen. Bei der Erstellung der hierarchischen Struktur kann auch die Baumstruktur vom Blatt bis zur Wurzel linear abgebildet werden. Jedoch können dabei die Regeln komplexer werden, je umfangreicher die Struktur bestimmt wird. Anhand der abstrakten Klasse Handler, wie in Abb. 2.1 dargestellt, hat jedes Kettenmiglied die Möglichkeit, seinen Nachfolger zu referenzieren. Soll die aktuelle Instanz in der Verantwortungskette, die Anfrage nicht behandeln kann, wird diese über die Funktion des abstrakten Handlers weitergeleitet. Erreicht dabei die Nachricht die letzte Instanz in der Verantwortungskette, so wird keine weitere Aktion ausgeführt.[EG94]

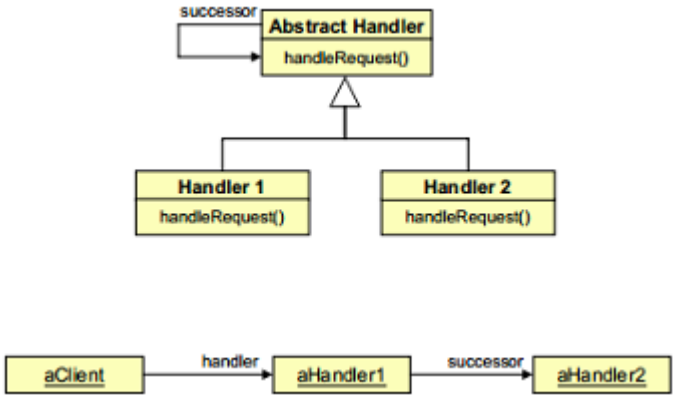

Abbildung 2.1: Darstellung der Struktur des Patterns Chain of Responsibility

2.6 Beispiel

Um die Struktur weiter zu verdeutlichen, soll dies anhand des folgenden Beispiels dargestellt werden. Eine Verantwortungskette in einem Unternehmen soll dazu dienen, eigenständig Entscheidungen über die Ausgabe von unternehmensinternen Ressourcen für Firmeninvestitionen zu fällen. Die Verantwortungskette ist hierarchisch angeordnet und besteht aus Manager, Director, VicePresident, President , wie in Abb. 2.2 dargestellt wird. Der Manager befindet sich in der untersten Hierachie und muss Anfragesummen entsprechend an die nächsthöhere Instanz weiterleiten, falls dieser keine Berechtigung für die angefragte Geldsumme besitzt. Dieser Vorgang setzt sich fort, bis eine berechtigte Stelle diese Anfrage erfüllen kann. Die Klasse *CheckAuthority* in Abb. 2.2 stellte eine abstrakte Klasse dar, die enstprechend dafür sorgt, dass ein geeigneter Handler für die Erfüllung der Anfrage ermittelt wird.

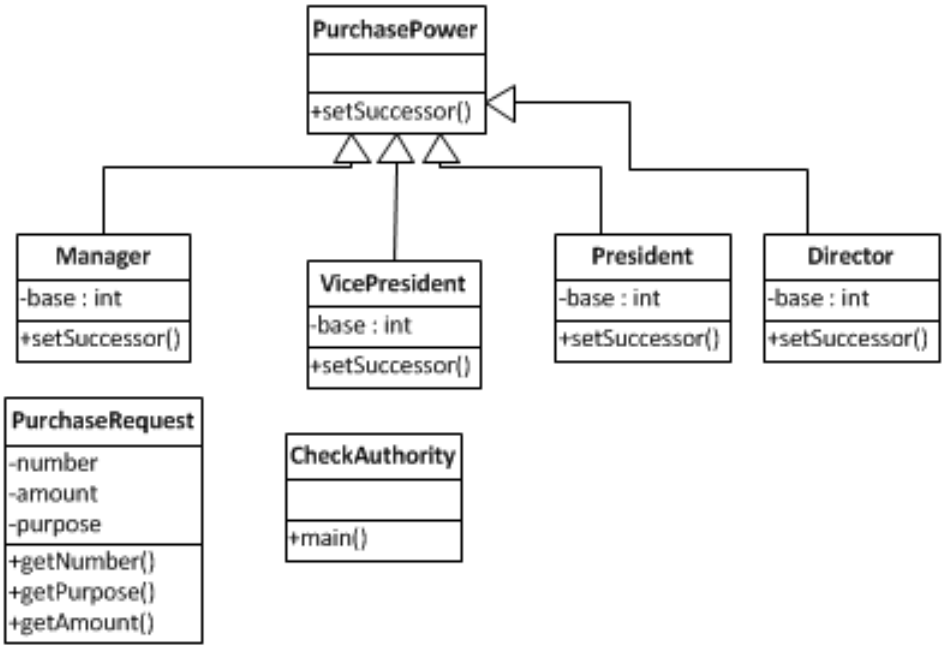

Abbildung 2.2: Beispiel eines Entwurfs mit Chain of Responsibility

2.7 Bewertung

Die Tatsache, dass keine Garantie für die Behandlung der Anfragen besteht, ist bei der Verrwendung dieses Patterns eher kritisch anzusehen. Jedoch ist Chain of Responsibility sehr gut einsetzbar, falls die Struktur der Nachrichtenkette schon vorher bekannt ist und jedes Element seinen Nachfolger kennt.

2.7.1 Vorteil

Reduktion der Kopplung

Der Sender übergibt seine Anfrage und ab diesem Zeitpunkt ist keine weitere Aktion vom Client mehr auszuführen. Die Verantwortungskette ist zu diesem Zeitpunkt dafür zuständig, die vom Client gesendete Nachricht zu bearbeiten. Zur Laufzeit der Anwendung besteht die Möglichkeit, das Verhalten der Verantwortungskette zu modifzieren bzw. Änderungen an dieser vorzunehmen.

2.7.2 Nachteile

keine Garantie auf Anfragebearbeitung

Es besteht keine Gewährleistung, dass jede Anfrage von der Verantwortungskette verarbeitet wird. Mit diesem Problemen verbunden ist auch das Debuggen zur Laufzeit der Anwendung. Die Struktur der Darstellung ist zu unübersichtlich, um genauere Information über das Laufzeitverhaltens der Anwendung zu bekommen.

Performanceprobleme

Es muss berücksichtigt werden, dass bei zunehmender Größe der Verarbeitungskette auch Performanceprobleme entstehen können. In diesem Zusammenhang ist sollte eine gute Entscheidung getroffen werden, dieses Pattern für komplexere Strukturen anzuwenden, weil diese auch komplexere Regeln mit sich bringt.

Literaturverzeichnis

[EF04] Eric Freeman, K. S. B. B., Elisabeth Freeman: *Head First Design Patterns*, O'Reilly, 2004.

[EG94] Erich Gamma, R. J. J. V., Richard Helm: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[Hau09] Hauer, P.: *Das Design Pattern Fassade*, 2009, http://www.philipphauer.de/study/se/design-pattern/state.php.