Komponentenbasierte Client-Architektur

Martin Haft · Bernd Olleck

Die Komplexität von Clients wird meist unterschätzt. Ein Client besteht eben nicht nur aus ein paar Masken. Es steckt noch eine Menge Funktionalität unter der „Oberfläche". Eine geeignete Strukturierung hilft die Komplexität beherrschbar zu machen.

Einleitung

Die Aufgaben eines Anwendungsservers sind gut verstanden und dessen Architektur entsprechend dieser Aufgaben etabliert. Das wird unterstützt durch technische Komponenten, wie Persistenzmanager, Transaktionsmanager und Autorisierungskomponente etc.

Der Client wird dagegen meist als ein Ganzes ohne weitere innere Struktur verstanden. Hier drehen sich die Überlegungen mehr um Verteilungsfragen. Wichtigste Entscheidungen sind oft, ob der Client bereits Geschäftslogik enthalten und ob der Client lokal auf dem Computer des Nutzers oder im Browser laufen soll. Die damit diskutierte Verteilung der Funktionalität des Clients auf Schichten, die sich an den Rechnergrenzen orientieren, wird mit den Begriffen Fat- und Thin-Client bzw. Rich-/Smart-/Web-Clients sowie mit Kombinationen wie Smart-Web-Clients oder Rich Internet Application bezeichnet.

Die Frage, was aber die Aufgaben des Clients alles umfassen und wie diese am besten strukturiert werden, wird kaum diskutiert. Sie wird eventuell durch ein verwendetes Framework teilweise vorgegeben oder gänzlich ignoriert.

Diese Strukturierung des Client-Codes nach Aufgaben eines Clients und unabhängig von der Rechnerarchitektur wollen wir als Client-Architektur bezeichnen.

Als Teilaspekt der Architektur von Clients ist die Strukturierung von Dialogen nach dem Model-View-Controller-Pattern [1] oder Abwandlungen davon, wie dem Model-2-Pattern, bekannt. Da MVC nur Teilaspekte löst, wird MVC durch das Hierarchical-MVC [2] auf die Situation mit mehreren hierarchischen Dialogen mit Kommunikation erweitert, ähnlich wie auch beim Presentation-Abstraction-Control [4]. Diese Pattern lösen einen weiteren Teilaspekt, eine komplette Client-Architektur definieren sie damit noch nicht.

Allgemeine Architekturmodelle wie beispielsweise das Arch/Slinky-Metamodell [6] oder Quasar [12] haben Dialoge als Ganzes im Fokus. Diese sind auf einer sehr groben Abstraktionsstufe formuliert und müssen bei der Entwicklung eines Clients verfeinert und ergänzt werden.

Diverse Client-Frameworks geben dagegen konkretere Architekturen vor, wie die Eclipse–Rich-Client-Platform [5], der Composite-UI-Application-Block [11], JavaServer-Faces [9] oder das Spring-Rich-Client-Project [13]. Hier sind die Architekturkonzepte nicht ausformuliert, sondern implizit durch die Frameworks definiert und auf ihren Kontext zugeschnitten.

DOI 10.1007/s00287-007-0153-9

Dr. Martin Haft
sd&m AG,
Carl-Wery-Straße 42,
81739 München
E-Mail: Martin.Haft@sdm-research.de

Bernd Olleck
IT-Beratung Olleck,
Frundsbergstraße 46,
80634 München
E-Mail: Bernd.Olleck@it-beratung.olleck.de

Zusammenfassung
Die Komplexität von Clients in Client-Server-Systemen wird meist unterschätzt. Dies liegt oft daran, dass ein Client als Ganzes ohne weitere innere Struktur betrachtet wird. Komplexität wird aber dann beherrschbar, wenn man das Ganze im Sinne der Trennung nach Zuständigkeiten, nach seinen Aufgaben zerlegt: Teile und Herrsche! Die hier vorgestellte aufgabenorientierte Zerlegung der Funktionalitäten eines Clients in Komponenten und deren Interaktion wollen wir als komponentenbasierte Client-Architektur bezeichnen.

Wir wollen hier die verschiedenen Fäden zusammenführen und eine allgemeine Architektur für Clients betrieblicher Informationssysteme motivieren, indem wir die Aufgaben eines Clients sammeln und daraus Komponenten innerhalb eines Clients ableiten, welche die einzelnen Aufgaben übernehmen. Die Gesamtheit der Komponenten und ihre Interaktion ergeben dann eine komponentenbasierte Client-Architektur.

Die wesentlichen Aufgaben eines Clients sind von der Art der Umsetzung des Clients unabhängig. Sowohl ein nativer Smart-Client wie ein Web-Client muss die Daten des Servers formatiert darstellen oder Benutzereingaben entgegennehmen und verarbeiten. Die aus den Aufgaben abgeleitete Architektur ist daher frei von der gewählten Rechnerarchitektur und kann je nach Art des zu erstellenden Clients verschieden auf die Software-Schichten verteilt werden. Daher ist auch die vorgestellte Architektur unabhängig vom Kanal – Web oder Native – und dem genutzten UI-Framework – Swing, SWT, JSF, WinForms, ASP.NET, GWT etc.

Die hier vorgestellte Client-Architektur ist „Paper-Ware". Sie bleibt damit abstrakt. Sie ist jedoch nicht auf der grünen Wiese entstanden, sondern basiert auf den Erfahrungen vieler unterschiedlicher Projekte der sd&m AG und etlichen Client-Frameworks. Sie ist Konsens diverser Client-Chefarchitekten – die Autoren stehen hier nur stellvertretend für das ganze Team. Die Architektur kommt somit aus der Praxis. Ihre theoretischen Wurzeln hat sie in der in Quasar [12] vorgestellten Architektur grafischer Bedienoberflächen, welche sie konkretisiert und vertieft.

Um die Konzepte zu verifizieren gibt es bei sd&m auch Implementierungen der Architektur als Client-Framework und Client-Komponenten für Java-Swing- und JSF-Clients sowie prototypische Implementierungen für .NET/WinForms und das Google-Web-Toolkit [7]. Die Implementierungen sind aber nicht Gegenstand dieses Artikels und würden den Rahmen deutlich sprengen. Nur in einzelnen Fällen werden wir Schnittstellen oder Abläufe zur Verdeutlichung der Konzepte erklären, die der bestehenden Implementierung entlehnt sind. Eine Verallgemeinerung der Schnittstellen und Abläufe im Sinne einer Referenzarchitektur ist auch in [8] zu finden.

Was ist ein Client?

Als „Client" verstehen wir die oberste Schicht der Drei-Schichten-Architektur, die dem Benutzer die Bedienoberfläche zur Verfügung stellt (Abb. 1).

Oft werden hierfür auch Begriffe wie „(grafische) Bedienoberfläche", „Benutzerschnittstelle" (englisch: User Interface), Präsentationsschicht (Presentation Layer) oder „Dialoge" benutzt. Wie auch „Client" reduzieren diese Begriffe die Aufgaben auf einzelne Aspekte und werden daher dem Umfang der obersten Schicht eines Client-Server-Systems nicht gerecht. Wir bleiben daher bei dem Begriff „Client", der sich weitgehend durchgesetzt hat.

So einfach die drei Schichten (Abb. 1) zu verstehen sind, so vielfältig sind mittlerweile die unterschiedlichen Möglichkeiten, den Client auf

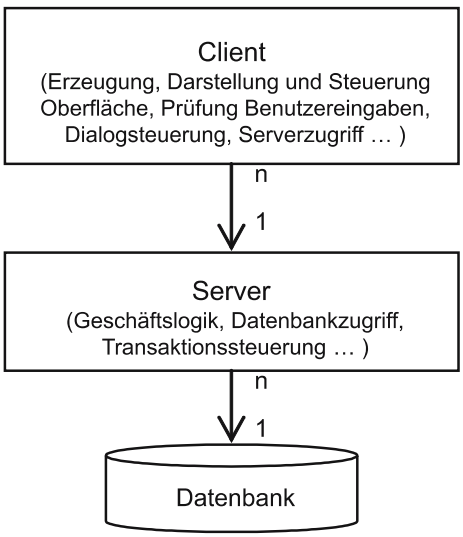

Abb. 1 3-Schichten-Architektur

Abstract
It is easy to underestimate the complexity of clients in client server systems. A reason for this is that the client is usually considered to be one piece of software without inner structure. However, divide and conquer is the best approach to complexity. For clients this would mean a division into responsibilities and tasks. We want to call this component-oriented client architecture. It's a partition of functionalities of a client into components and their interaction.

verschiedene Rechner zu verteilen. Die direkteste Umsetzung der drei Schichten ist beim nativen Client zu finden (Abb. 2).

Beim klassischen Web-Client läuft der wesentliche Teil der Client-Logik im Web-Server, der Browser am Benutzerrechner ist nur noch für die Darstellung der Oberfläche und das Entgegennehmen von Benutzeraktionen zuständig, wie in Abb. 3 dargestellt.

Mit JavaScript, AJAX und anderen Techniken (aktuell mit dem Schlagwort Web 2.0 zusammengefasst) kann der Schnitt zwischen Aufgaben, die im

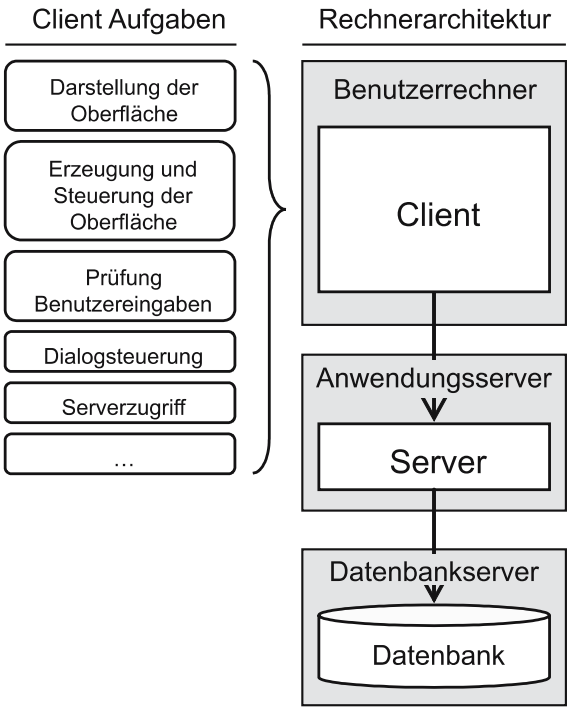

Abb. 2 Rechnerarchitektur des nativen Clients

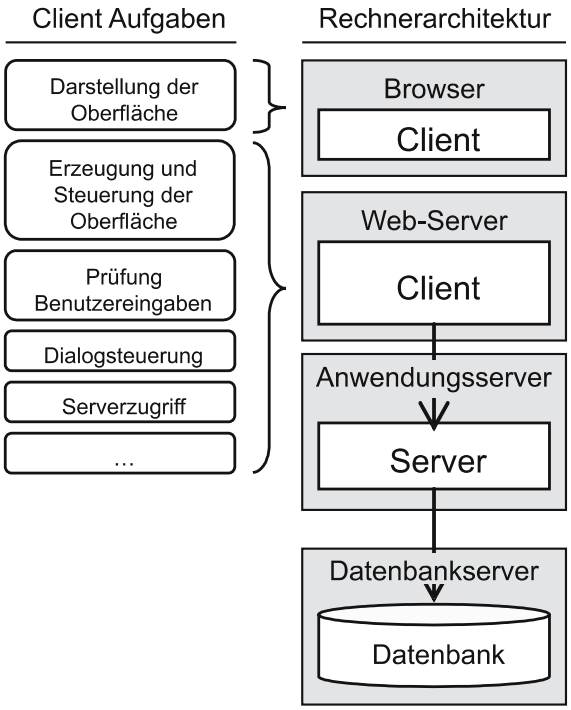

Abb. 3 Rechnerarchitektur des Web-Clients

Browser bearbeitet werden, und Aufgaben, die im Web-Server erfolgen, verschoben werden.

Unabhängig von der Verteilung des „Clients" auf Benutzerrechner und Server bleiben die Aufgaben, die der Client erledigen muss, dieselben. Daher kann eine allgemeine Client-Architektur unabhängig von der Rechnerarchitektur formuliert werden: Sie muss lediglich die oben benötigten Rechnergrenzen ermöglichen.

Ziele der Architektur

Wir wollen eine möglichst einfache und trotzdem flexible Architektur entwickeln, die auf Clients unterschiedlichster Natur angewendet werden kann. Die Grundlage dafür bildet das Verständnis der Aufgaben, welche ein Client übernehmen muss. Wir basieren die Architektur auf den Softwareprinzipien

- Trennung von Zuständigkeiten,
- Bildung von Komponenten und
- Reduzierung der Abhängigkeiten.

Diese komponentenorientierte Client-Architektur macht dadurch die Komplexität der Software beherrschbar. Im Folgenden gehen wir auf drei uns besonders wichtige Aspekte näher ein.

Kein invasives Framework

Client-Basis-Software kennt eine zusätzliche, ihr eigene Herausforderung: Der Kontrollfluss der Anwendung, d.h. welche Aktion ausgeführt oder welcher Dialog geöffnet wird, wird vom Benutzer, also von außen, gesteuert. Dies führt leicht zu einer Vermischung von technischem Code (wie werden Dialoge geöffnet und Aktionen angesteuert?) und fachlichem Code (welche Dialoge werden geöffnet und wie ist eine Aktion implementiert?). Client-Basis-Software tendiert hier schnell dazu, ein invasives Framework zu werden, in welchem sich der fachliche Code dem technischen Code unterordnen muss.

Im Gegensatz zu typischen Client-Frameworks hat z.B. ein Persistenzmanager ein API in Form von Schnittstellen, über welches er die Bewerkstelligung seiner Aufgaben als Dienste bereitstellt. Er ist im Sinne der Definition von Szyperski [14] eine Komponente. Im Idealfall muss sich ein Entwickler nicht um dessen Innenleben kümmern. Ein Persistenzmanager wird genutzt, indem gegen seine Schnittstellen programmiert wird.

Client-Frameworks stellen dagegen einen Rahmen für die Entwicklung von Dialogen bereit: Die Dialoge müssen in das Framework „eingepasst" werden. Ein Framework kann nur genutzt werden, indem man sich in das Framework integriert. Passt das Framework nicht, so muss es verbogen werden oder es wird am Framework vorbeigearbeitet, d.h. man arbeitet nicht mehr mit dem Framework, sondern dagegen. Und man kann sich sicher sein: Innerhalb der Dialoge, die erstellt werden sollen, gibt es immer einen, der nicht ins Framework passt – außer es handelt sich um das eigene, maßgeschneiderte Framework, was dann aber nicht mehr allgemein nutzbar ist.

Eine gute Client-Architektur minimiert den invasiven Anteil auf das Unabdingbare und definiert ansonsten Komponenten, die ihre Dienste als Nutz-Schnittstellen ohne Rückrufmethoden anbieten.

Komponentenarchitektur

Mit dieser Client-Architektur stellen wir deshalb eine Architektur von Komponenten vor, die nicht nur dedizierte Aufgaben übernehmen, sondern diese auch – wo möglich – über APIs als Dienste zur Verfügung stellen. Die frameworkartigen Anteile sind reduziert auf die Stellen, wo sie unumgänglich sind, nämlich auf die Steuerung (d.h. das Starten und Beenden) von Dialogen. Alle weiteren Elemente sind Komponenten mit Nutzschnittstellen oder Adapter.

Nicht nur die Clients im Ganzen können unterschiedlich komplex sein. Clients bestehen meist selbst aus Dialogen unterschiedlicher Komplexität. Es gibt immer ein paar sehr einfache Hilfs- oder Nachfragedialoge. Die meisten Dialoge haben mittlere Komplexität, z.B. zur Pflege fachlicher Objekte. Oft kommt eine gewisse Kommunikation zwischen den Dialogen vor, z.B. bei der Parameterübergabe. In größeren Dialogsystemen ist dann immer mindestens ein extremer Dialog dabei, der hohe Anforderungen an ein eventuell genutztes Framework stellt – und auch an den Entwickler. Frameworks sind oft auf eine bestimmte Art von Dialogen optimiert: entweder zur schnellen Erstellung vieler einfacher Dialoge oder abgestimmt auf komplexe Dialoge. Ersteres führt dazu, dass das Framework keine Hilfestellung für schwierige Dialoge liefert, Zweiteres dazu, dass einfache Dialoge mit viel Overhead erstellt werden müssen.

Die Client-Architektur ist so angelegt, dass mit ihr parallel Dialoge unterschiedlicher Komplexität gebaut werden können. Auch hier hilft der komponentenbasierte Ansatz und im Besonderen die möglichst weitgehende Entkopplung der Komponenten. Dabei sind zwei Arten zu unterscheiden:

1. Die Architektur beschreibt einzelne Komponenten, die auch unabhängig voneinander genutzt werden können. Dies bedeutet auch, dass nicht immer alle Komponenten benötigt werden (kein „one size fits all"). Beispiel: Eine Komponente wie eine Datenhaltung kann in allen Dialogen verwendet werden. Eine Komponente wie eine Zustandsmaschine wird nur in komplexeren Dialogen mit internen Zuständen benötigt. Die Entkopplung garantiert: Ob ein Dialog mit oder ohne Zustandsmaschine umgesetzt wird, ist für die weiteren Teile des Dialogs sowie für andere Dialoge nicht ersichtlich.
2. Die Nutzung einer Komponente definiert nicht die Architektur der Dialoge oder des gesamten Clients. Beispielsweise bestimmt die Steuerung der Dialoge nicht auch die Architektur der Dialoge. Erst die gewählte Zusammenstellung der Komponenten ergibt die Architektur.

Flexibilität

Wie oben beschrieben sind die Aufgaben, die ein Client erledigen muss, von Kanal und Technik unabhängig. Trotzdem schlägt sich beides oft in der Architektur nieder. Lediglich die Umsetzung bestimmter Komponenten muss Kanal oder Technik berücksichtigen. An der abstrahierten und komponentenbasierten Architektur kann man ablesen, welche Komponenten technikspezifisch umgesetzt werden müssen und welche unabhängig von Kanal und Technik gestaltet werden können.

Wir haben die Client-Architektur komponentenbasiert angelegt, um die Komplexität am Client beherrschbar zu machen. Große Dialoge haben oft auch eine fachliche Komplexität, die eine Zerlegung in Teile erfordert, um beherrschbar zu bleiben. Eine gute Client-Architektur berücksichtigt daher auch die Möglichkeit von fachlichen (Dialog-) Komponenten: Dialoge sollen aus Unterdialogen kombinierbar sein. Die Interaktion der Dialoge muss hier gewährleistet sein.

Der Zweck der hier vorgestellten Client-Architektur ist auch, die in der Client-Schicht anfallenden Aufgaben so zu strukturieren und zu verteilen, dass eine möglichst flexible Aufteilung von Aufgaben an Entwickler sowie eine gezielte Bibliotheksunterstützung ermöglicht wird.

Architektur

Unser wesentliches Ziel bei der Entwicklung dieser Architektur ist, die Aufgaben zu sammeln und zu strukturieren, die in einem Client erledigt werden müssen. Dabei liegt unser Fokus sowohl auf einzelnen Dialogen (Wie strukturiere ich einen Dialog?) wie auch deren Integration zu einem Client (Wie arbeiten mehrere Dialoge so zusammen, dass sich ein Client ergibt? Was macht einen Client neben den Dialogen noch aus?). Das sollte unabhängig von der Verteilung der Software auf Schichten sein, sodass diese Prinzipien sowohl für Thin- und Fat-Clients, für Rich-/Smart- und Web-Clients einen Mehrwert bei der Entwicklung bieten. Die Unterschiede ergeben sich erst in der Umsetzung der Architektur für die benötigte Art von Client.

Die vorgestellte Client-Architektur ergibt sich daher aus der Charakterisierung und Verteilung der Aufgaben, die in einem Client anfallen. Die Tabelle 1 führt etliche dieser Aufgaben auf. Diese Aufgaben sind nicht in allen Clients gleich wichtig und manche mögen in einer konkreten Situation auch irrelevant sein. Wir haben jedoch festgestellt, dass sie sehr häufig anzutreffen sind und meistens die gleichen Lösungsansätze zum Ziel führen. Die rechte Spalte der Tabelle erwähnt als Vorgriff bereits die später detaillierten Architekturbegriffe.

Die Aufgaben innerhalb eines Dialogs und die Aufgaben der Verwaltung und Steuerung von Dialogen wollen wir strikt trennen. Wir führen deshalb als erste grobe Unterteilung in der Architektur die Trennung in eine Architektur der Dialoge und eine Architektur des Dialograhmens ein (Abb. 4).

Viele Dialoge werden nach dem MVC-Pattern [1] oder im Web nach dem Model-2-Pattern gebaut. Auch das Arch/Slinky-Metamodell [6] beschreibt die Architektur eines Dialoges. Ein Client besteht aber in der Regel nicht nur aus einem Dialog. Vielmehr werden dem Benutzer viele Dialoge präsentiert, die nicht unabhängig voneinander sind, sondern miteinander interagieren. HMVC [2] erweitert daher das MVC-Pattern um eine Hierarchie von MVC-Triaden/Dialogen, in welchen die Controller über Events innerhalb der Hierarchie miteinander kommunizieren. Aber auch HMVC lässt offen, wie die Controller und damit die MVC-Triaden verwaltet werden und wer einen neuen Controller mit seinem Model und seiner View instanziiert. Auch muss bei hierarchisch aufgebauten Dialogsystemen geklärt werden, wer dafür sorgt, dass abhängige Dialoge zum richtigen Zeitpunkt geschlossen werden. Die Fragen der Verwaltung und Steuerung seiner „Dialoge“ wird beispielsweise durch Eclipse RCP [5] beantwortet: Hier sind die Dialoge optisch eingebettete Plug-ins, die durch RCP instanziiert, verwaltet und wieder freigegeben werden.

Diese Verwaltungs- und Steuerungsaufgaben übernimmt der Dialograhmen, der damit die grund-

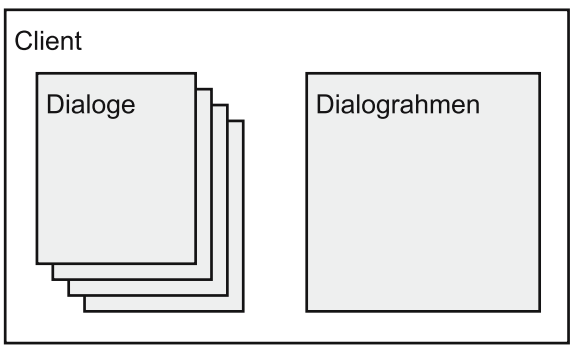

Abb. 4 Dialoge und Dialograhmen

Tabelle 1

Typische Aufgaben im Client und ihre Abbildung in die Architektur

Aufgabe	Charakterisierung	Lösung in der Architektur
An- und Abmelden	Ein Benutzer kann sich an- und abmelden.	Trennung von Sitzung und Anwendung
Globale Konfiguration	Es gibt Konfigurationseinstellungen, die für den gesamten Client gelten.	Konfiguration in der Anwendung
Benutzereinstellung	Es gibt benutzerspezifische Konfigurationseinstellungen.	Konfiguration in der Sitzung
Sitzungsrelevante Daten	Es gibt Daten, die übergreifend allen Dialogen innerhalb einer Sitzung zur Verfügung stehen, z.B. der Name des Benutzers oder seine persönlichen Einstellungen.	Datenhaltung in der Sitzung
Wechsel von Dialogen	Beim Wechsel von Dialogen müssen sowohl fachliche als auch organisatorische Aspekte berücksichtigt werden.	Trennung von Dialograhmen und Dialogen
Gleichen Dialog mehrfach gleichzeitig darstellen	Es gibt Dialoge, die dem Benutzer gleichzeitig mehrfach angezeigt werden sollen, z.B. durch die Möglichkeit, zwei Fenster offen zu haben, oder durch ein Bestellformular mit zwei Subdialogen zur Adresseingabe für Rechnungs- und Lieferadresse.	Unterscheidung zwischen Dialog und Dialoginstanz
Oberfläche darstellen	Die optische Darstellung eines Dialoges mit seinen Widgets, seiner Organisation und ggf. einem eigenen Fenster.	Visuelle Darstellung eines Dialoges, definiert in der Präsentation
Daten verwalten	Daten müssen lokal im Client gehalten und ggf. verarbeitet werden. Sie haben andere Formate als in der Oberfläche (z.B. Datum als Objekt statt als String).	Fachliche Datenhaltung im Dialogkern
Daten darstellen	Daten müssen in den Elementen der Oberfläche dargestellt werden. Dies beinhaltet oft auch eine Konvertierung von einem internen Format in ein Benutzerformat.	Datenanbindung in Präsentation
Eingaben prüfen	Benutzereingaben müssen geprüft und ggf. in ein internes Format umgewandelt werden (z.B. Datumseingabe).	Validierung in Kombination mit der Datenanbindung in der Präsentation
Aktionen ausführen	Datenverarbeitung oder andere fachliche Aktionen können vom Benutzer über Widgets ausgelöst werden.	Aktionsanbindung und -verarbeitung
Zustände in Dialogen	Ein Dialog kann mehrere Zustände haben, z.B. durch einen expliziten Wechsel zwischen einem Anzeige- und Editiermodus.	Zustände im Dialogkern
Zustände in der Präsentation	Bestimmte Widgets können verschiedene Zustände in Abhängigkeit von anderen Widgets oder von Werten haben, z.B. ein „Speichern"-Button ist auswählbar, wenn sich Werte geändert haben.	Präsentationszustände
Dialogbausteine	Dialoge können andere Dialoge zu einem ganzen Dialog kombinieren oder andere Dialoge aufrufen, z.B. für modale Eingabe.	Dialoge bilden eine Hierarchie
Server aufrufen	Ein Client muss Daten mit einem Server austauschen.	Serveraufrufe in Aktionen

legende Infrastruktur bildet, die notwendig ist, um einen lauffähigen Client zu erhalten. Er ist die Ablaufumgebung, welche die Dialoge in einer Hierarchie verwaltet, ihr „Öffnen" und „Schließen" steuert und die Kommunikation zwischen den Dialogen ermöglicht. Man könnte ihn auch als Dialog-Container bezeichnen. Dialoge selbst sind die fachlichen Bausteine, mit denen der Benutzer interagiert.

Dialoge und Dialograhmen

Dialoge und Dialograhmen sollen möglichst stark entkoppelt sein, da alles, was der Dialograhmen von seinen Dialogen fordert, diese in ihrer Architektur

und in ihrem Ablauf einschränken: Der Dialograhmen hat dabei einen minimalen, frameworkartigen, invasiven Anteil. Die starke Entkopplung gibt dabei die Freiheit, Dialoge mit unterschiedlicher Komplexität auch mit unterschiedlichen, dem jeweiligen Problem angemessenen Architekturen zu bauen.

Die Abhängigkeiten zwischen Dialograhmen und Dialogen reduzieren sich auf wenige Aspekte:

- Ein Dialog muss für den Dialograhmen identifizierbar sein, damit er ihn verwalten kann.
- Das „Öffnen" und „Schließen" von Dialogen muss über den Dialograhmen konsistent in der Hierarchie ausgeführt werden, d.h. ihr Lebenszyklus muss vom Dialograhmen gesteuert werden.
- Der Dialograhmen muss auch eine Kommunikation zwischen den Dialogen ermöglichen.

Ein Dialog ist aus Sicht des Dialograhmens daher eine identifizierbare fachliche Komponente mit Lebenszyklus. Im Gegenzug bietet der Dialograhmen jedem Dialog den Zugriff auf seine Infrastruktur an sowie die Möglichkeit der Kommunikation mit anderen Dialoginstanzen.

Dialogarchitektur

Der Dialograhmen gibt keine Dialogarchitektur vor, damit Dialoge unterschiedlicher Komplexität auch mit der geeigneten Architektur umgesetzt werden können. Wir setzen also nicht die *eine* Dialogarchitektur voraus, sondern ermöglichen es, eine der Komplexität des jeweiligen Dialoges angemessene zu wählen. Ein einfacher Dialog zur Anzeige einer Meldung sollte nicht durch Framework oder Architekturvorgaben gezwungen werden, eine innere Struktur zu besitzen, welche die Implementierung von mehreren jeweils fast inhaltsleeren Klassen erfordert oder zu unnötiger Kommunikation zwischen den inneren Strukturen führt. Umgekehrt muss ein komplexer Dialog gut strukturiert entwickelt werden können.

Die wesentlichen Aufgaben eines Dialoges sind:

- den statischen Teil der Oberfläche erstellen (Maske),
- Daten verwalten, in der Oberfläche darstellen, Eingaben entgegennehmen und prüfen,
- auf Benutzereingaben reagieren und Aktionen ausführen,
- dynamische Teile der Oberfläche steuern,
- eigene Unterdialoge steuern und sich mit anderen Dialogen koordinieren (Dialoglogik) und
- die Kommunikation mit einem Server, Dateisystem etc..

Um eine passende Dialogarchitektur zu finden, hilft es, die daten-, aktions- und steuerungsbezogenen Aufgaben in zwei Schichten zu gliedern, in die Präsentationsschicht und den Dialogkern (nach Quasar [11]). Dadurch werden die Abhängigkeiten zur genutzten Visualisierungstechnik und zum Server getrennt. Abbildung 5 verdeutlicht diese Unterteilung.

Die Präsentation hat dabei die Aufgabe, mit der GUI-Bibliothek (z.B. Swing, JSF, WinForms ...) zu interagieren, Daten des Dialogkerns darzustellen und Präsentationsereignisse auf Ereignisse im Dialogkern abzubilden. Die Präsentation ist von der gewählten GUI-Bibliothek abhängig, aber nicht vom Serverzugriff. Die Präsentation „kennt" ihren Dialogkern, die umgekehrten Abhängigkeiten werden wie bei MVC [1] durch das Observer-Pattern aufgelöst.

Der Dialogkern hat die Aufgabe, die fachlichen Daten und Zustände zu verwalten, fachliche Logik und Dialoglogik auszuführen und mit dem Server zu kommunizieren. Der Dialogkern ist unabhängig von der gewählten GUI-Bibliothek.

Je nach Komplexität eines Dialoges kann eine angemessene Detailtiefe für diese Strukturierung gewählt werden. In extremen Dialogen kann jede dieser Aufgaben in jeder Schicht durch jeweils eigene Sub-Komponenten erfüllt werden. Für einfache Dialoge kann jegliche Trennung entfallen und ein „Dialog aus einem Guss" entwickelt werden. Nicht jede Aufgabe ist in jedem Dialog gleich schwierig

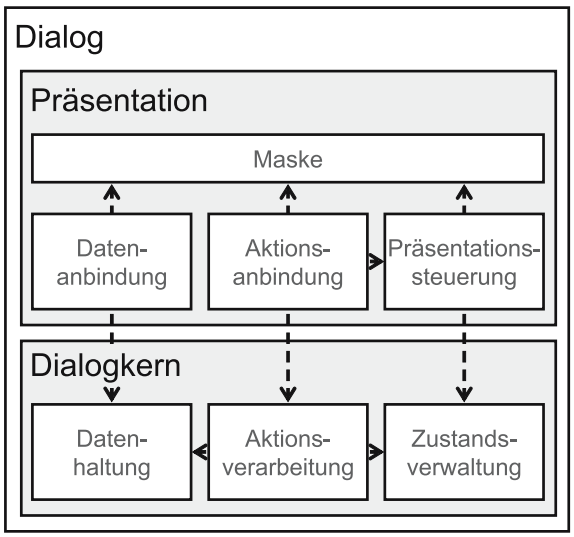

Abb. 5 Aufgabenverteilung in Dialogen

zu erfüllen, sodass auch Zwischenstufen angebracht sein können. Selbst bei einfachen Dialogen hilft das Verständnis der Trennung von Dialogkern und Präsentation und deren Aufgaben zur Strukturierung des Codes.

Ein weiterer Vorteil der aufgabenorientierten Dialogarchitektur ist, dass etliche der Aufgaben durch Bibliotheken unterstützt werden können. Dadurch ist es möglich, verschiedene wiederverwendbare Lösungen für viele Probleme einzusetzen, ohne in die strengen Strukturen eines Frameworks gezwungen zu werden.

Im Folgenden charakterisieren wir die einzelnen Aufgaben bezogen auf ihre Schichten.

Dialogkern

Im Dialogkern werden die Daten des Dialoges verwaltet. Diese Aufgabe bezeichnen wir als **Datenhaltung**. Die Datenhaltung kann im einfachsten Fall als Klassen mit Methoden zum Zugriff auf die Datenobjekte realisiert werden. Diese Form wird meistens im Web gewählt, z.B. bei JSP und JSF als „Backing Bean". In nativen Dialogen ist es oftmals hilfreich, vergleichbar dem Model des MVC-Pattern, eine observierbare Datenhaltung zu verwenden, sodass die Oberfläche bei feingranularen Änderungen sofort aktualisiert werden kann. Hier kann eine komplexe Datenhaltung mit Zugriff per Schlüssel und Wissen über die fachliche Struktur der Daten den Entwickler unterstützen. Eine auf das Wesentliche reduzierte Schnittstelle einer Datenhaltung ist in Abb. 6 zu sehen. Hat man einmal eine Datenhaltung, so kann diese je nach Kontext, in welchem sie eingesetzt wird, auch weitere Aufgaben übernehmen, wie für thread-sicheren Zugriff auf die Daten sorgen, ein Transaktionskonzept anbieten, Änderungsflags von Daten innerhalb einer Transaktion verwalten und Metainformationen über ihre Daten halten, etc.

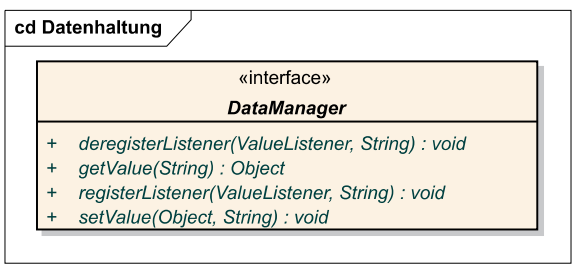

Abb. 6 Exemplarische Schnittstelle einer Datenhaltung

Es gibt im Wesentlichen zwei Nutzer der Datenhaltung: die Präsentation, welche die Daten darstellt und Eingaben zurückschreibt, und die fachliche Logik bzw. Dialoglogik im Dialogkern, welche Daten manipuliert und sie mit anderen Dialogen oder einem Server austauscht. Wichtig wird die Datenhaltung im Dialogkern dadurch, dass in ihr die Daten nicht in der Benutzerrepräsentation abgelegt sind, sondern in einem logisch verarbeitbaren Format: Ein Datum ist kein Text, sondern eine dedizierte Klasse (in Java z.B. ein Date-Objekt), die logische Operationen ermöglicht.

Die Umsetzung der fachlichen Logik und der Dialoglogik bezeichnen wir als „fachliche Aktionen", die entsprechende Sub-Komponente als **Aktionsverarbeitung**. Fachliche Aktionen können mit Daten arbeiten, Serveraufrufe durchführen oder Steuerungsfunktionen übernehmen, wie z.B. einen anderen Dialog aufblenden. Sie werden der Präsentation oder anderen Dialogkomponenten auf einer relativ grobgranularen fachlichen Ebene angeboten. Dies kann technisch durch einfache Methoden einer Schnittstelle gelöst werden, wie bei JSF ebenfalls durch ein Backing Bean. Etwas mehr Entkoppelung bietet ein Command- oder Action-Pattern. Commands/Actions können dabei in Abhängigkeit von ihrer Aufgabe ausführbar oder gesperrt sein. Die Änderung dieses Ausführbarkeitszustandes kann auch observierbar sein. Abbildung 7 zeigt eine einfache Version der Schnittstellen einer Aktionsverarbeitung auf Basis des Command-Pattern.

Eine weitere Form der Umsetzung der fachlichen Logik bzw. Dialoglogik ist eine Zustandsmaschine, welche durch Events getriggert je nach Zustand unterschiedliche Aktionen ausführt. In diesem Fall sind die fachlichen Aktionen als Zustandsübergänge integriert.

Dialoge besitzen **Zustände**. Diese können fachliche Zustände der geladenen Daten sein (z.B. ein Artikel eines Webshops ist ein Buch oder eine DVD) oder technische Zustände der Daten, wie „Daten sind geändert, aber nicht gespeichert". Selbst einfache Dialoge können Zustände haben. Beispielsweise kann ein Pflegedialog für Bücher unterschiedliche Modi haben: einen für Aufnahme eines neuen Buches und einen zur Änderung der Daten eines Buches, wobei in letzterem die fachlichen Schlüssel (z.B. ISBN) nicht mehr geändert werden dürfen.

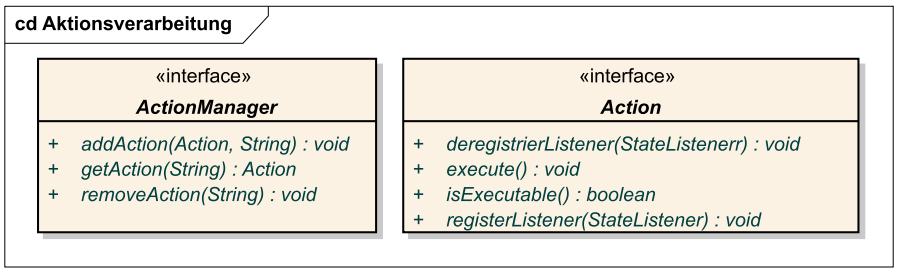

Abb. 7 Exemplarische Schnittstelle einer Aktionsverarbeitung auf Basis von Commands/Actions

Die Zustände sind oft aus Daten ablesbar und damit implizit vorhanden oder durch Flags realisierbar. Für einfache Dialoge reicht diese Form der „Zustandsverwaltung". Werden die Zustände komplexer, so führt die implizite Verwaltung leicht zu Wenn-dann-Kaskaden in den fachlichen Aktionen und ihre Visualisierung an der Oberfläche benötigt Wissen über Details der Verarbeitung im Dialogkern. Spätestens hier empfiehlt es sich, die Zustände in Form einer Zustandsverwaltung explizit zu machen. Für den Einsatz einer Zustandsmaschine im Dialogkern ist dies sogar Voraussetzung. In nativen Dialogen ist es hilfreich, observierbare Zustandsänderungen zu haben.

Damit sehen wir im Dialogkern bis zu drei Subkomponenten: Datenhaltung, Aktionsverarbeitung und Zustandsverwaltung.

Präsentation

Die Präsentation ist verantwortlich für die Bereitstellung und Steuerung der Anzeige des Dialogs. Die Anzeige besteht meistens aus einem statischen Teil (der **Maske** oder visuellen Darstellung, d.h. Anzeige- und Eingabefeldern, Buttons und Organisationselementen wie Fenstern und Tabs) und dynamischen Teilen (Daten in unterschiedlichen Darstellungsformen sowie Zustandsinformationen, vorwiegend in Form von Sichtbarkeit bzw. Bedienbarkeit von Bildschirmelementen). Sehr viel Code, der zur Präsentationsschicht gehört, ist abhängig von der GUI-Bibliothek.

Die Daten kommen in der Regel vom Dialogkern. Zur Anzeige der Daten müssen diese normalerweise in eine für den Benutzer lesbare Form konvertiert werden. Die Formatierung ist dabei vom Locale des Benutzers und eventuell von anderen Benutzereinstellungen abhängig. Umgekehrt müssen vom Benutzer eingegebene Daten auch wieder geparst und dabei auf syntaktische Korrektheit geprüft werden, bevor sie weiterverarbeitet werden. Den Vorgang des Verbindens der Oberflächendaten mit Dialogkerndaten, also des Konvertierens von Datenobjekten mit lokalisierter Formatierung, Parsen und syntaktischer Validierung, bezeichnen wir als **Datenanbindung** oder Data-Binding.

Die Datenanbindung ist dabei reiner Konfigurationscode: Bei der Initialisierung wird definiert welches Datum der Datenhaltung an welches Oberflächenelement gebunden wird und welche Konvertier- und Prüfroutinen dabei benutzt werden. Die wesentlichen Aufgaben für einen einfachen Adapter ohne Konvertierung und Prüfroutinen für eine native GUI-Bibliothek auf eine wie in Abb. 6 dargestellte Datenhaltung haben wir für Abb. 8 beschrieben.

Eine Bibliotheksunterstützung zur Datenanbindung ist relativ einfach verwendbar und besteht in der Regel aus kombiniert einsetzbaren Konvertier- und Prüfmodulen sowie einer spezifischen Befüllung je Widget. Softwareunterstützung zum Data-Binding gibt es bereits in verschiedenster Form, z.B. in

```
public TextFieldAdapter(DataManager dataManager, String key, TextField textField)
{
        1. Registriere diesen Adapter als ValueListener am DataManager
           => Bei Datenänderung wird das Textfeld aktualisiert

        2. Registriere diesen Adapter als Listener am TextFeld
           => Bei Benutzereingaben in das Textfeld wird die Datenhaltung aktualisiert
}
```

Abb. 8 Textuelle Beschreibung eines einfachen Datenadapters

.NET oder bei JSF. Auch Client-Frameworks wie JGoodies [10] oder CUF (Client Utilities & Framework) [3] bieten hierfür Implementierungen an.

Vom Benutzer ausgelöste Aktionen, z.B. durch einen Button oder die Auswahl in einer Klappliste, werden in der Präsentation entgegengenommen. Aktionen, die mehr als eine primitive Manipulation der Oberfläche sind, werden an den Dialogkern weitergegeben. Dabei erfolgt eine Umdefinition von einem technischen Präsentationsereignis (Klick des Buttons „Speichern“) in eine fachliche Aktion („Prüfen und Speichern der Daten“). Diesen Vorgang nennen wir **Aktionsanbindung** oder Action-Binding. Die Aktionsanbindung entkoppelt die von der GUI-Bibliothek abhängigen Mechanismen, z.B. Registrieren an einem GUI-Event, von der fachlichen Verarbeitung. Eine einfache Aktionsanbindung ist der Aufruf einer Methode im Dialogkern oder, wie beim Method Bindung von JSF, des Backing Beans.

Auch die Aktionsanbindung ist reiner Konfigurationscode. Für native GUI-Bibliotheken ist sie durch die Nutzung von Adaptern, wie in Abb. 9 exemplarisch und vereinfacht beschrieben, möglich.

Zustände in der Präsentation sind – im Gegensatz zu Zuständen im Dialogkern – relativ feingranular. Es gibt zwei verschiedene Situationen für Zustände:

- vom Benutzer manipulierbare, beispielsweise ob eine Checkbox selektiert ist oder in einer Tabelle eine Zeile ausgewählt ist, und
- auf die Anzeige wirkende, z.B. ob ein Button verfügbar ist.

Diese Boole'schen Zustände der Widgets der Oberfläche bezeichnen wir als Präsentationszustände. Die Präsentationszustände sind oft voneinander und von Zuständen im Dialogkern abhängig, wobei häufig logische Verknüpfungen eingesetzt werden. Zum Beispiel kann ein Bestellen-Button verfügbar sein, wenn mindestens ein Artikel im Einkaufskorb liegt, die Zahlungsmethode ausgewählt ist und die Checkbox zur Akzeptanz der AGB selektiert wurde. Die Implementierung dieser Abhängigkeiten nennen wir **Präsentationssteuerung**.

In Abb. 10 ist grob ein Ablauf der Verarbeitung einer Benutzeraktion beschrieben. Dieser stellt exemplarisch die wichtigsten Aspekte des Zusammenspiels der einzelnen Komponenten eines nativen Clients dar.

Dialoge sind Komponenten

Wir haben hier eine Strukturierung eines Dialogs vorgestellt, welche die Komplexität dadurch beherrschbarer macht, dass die Aufgaben in verschiedenen Komponenten innerhalb eines Dialogs getrennt bearbeitet werden. Oft sind Dialoge jedoch so umfangreich, dass sie in einzelne fachliche Teile zerlegt werden müssen, um überschaubar zu blieben. Wir sprechen daher von Dialogen und Unterdialogen oder ganz allgemein von Dialogkomponenten. Dialogkomponenten können optisch in einen übergeordneten Dialog integriert sein oder optisch eigenständig sein, z.B. in einem eigenen Fenster. Der Unterschied zwischen optisch integriert und eigenständig verschwindet sogar bei „abreißbaren“ Fensterteilen.

Eine hierarchische Abhängigkeit zwischen Dialogen ergibt sich immer dann, wenn ein Unterdialog, z.B. eine separate Detailsicht, wieder zusammen mit dem übergeordneten Dialog geschlossen wird. Dies ist bei optisch integrierten Dialogkomponenten zwingend, aber auch bei eigenständigen Dialogen oft sinnvoll. Dadurch ergibt sich eine baumartige Dialoghierarchie.

Dialoge und Unterdialoge sind also fachliche Komponenten, die in der Regel fachliche

```
public Button2ActionAdapter(ActionManager actionManager, String key, Button button)
{
        1. Registriere diesen Adapter als ActionListener am Button
           => Bei Auslösen des Buttons wird die Aktion ausgeführt

        2. Registriere diesen Adapter als StateListener an der Aktion
           => Die Verfügbarkeit des Buttons wird entsprechend
              der Ausführbarkeit der Aktion gesetzt
}
```

Abb. 9 Textuelle Beschreibung eines einfachen Aktionsadapters

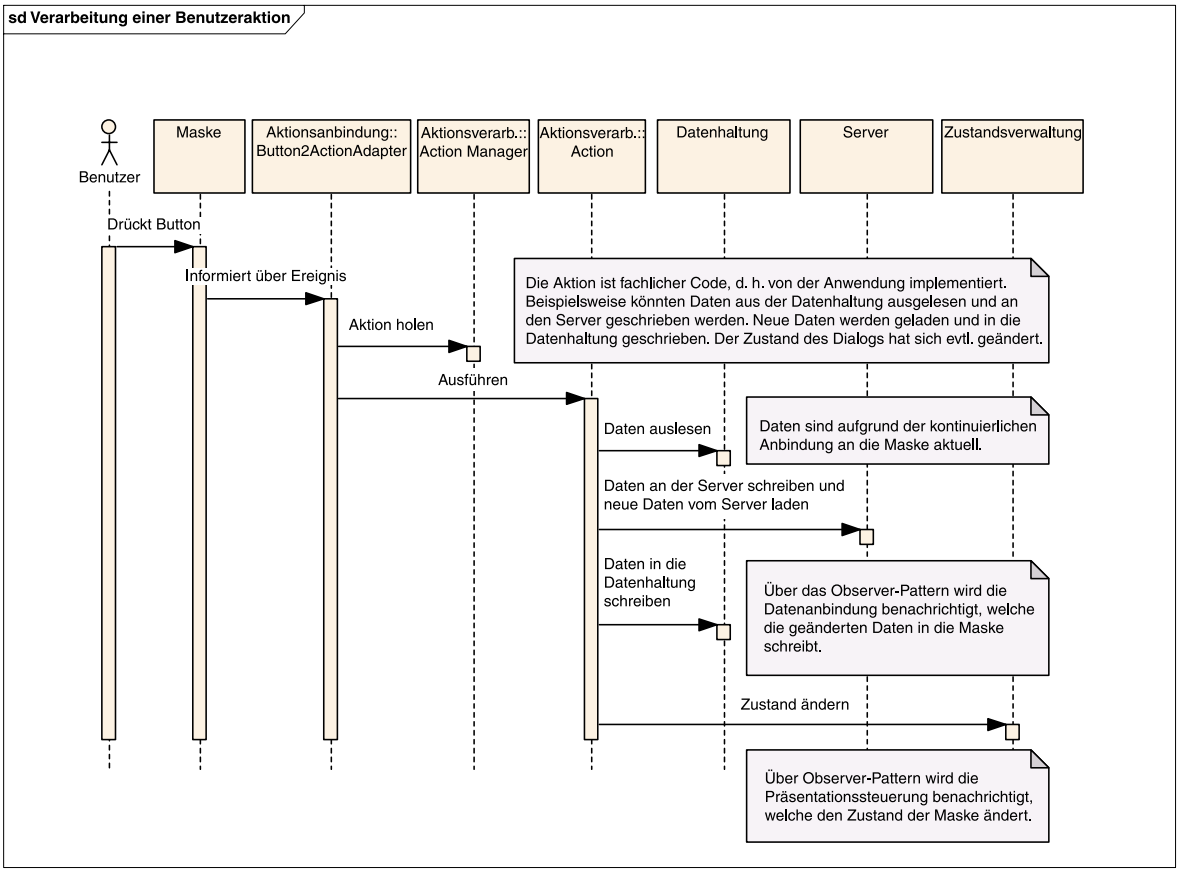

Abb. 10 Exemplarische Beschreibung der Verarbeitung einer Benutzeraktion

oder optische Einheiten bilden und von einem übergeordneten Dialog koordiniert werden. Dialogkomponenten werden durch ihre angebotenen und benötigten Schnittstellen definiert, kommunizieren ausschließlich über diese Schnittstellen miteinander und haben keine weiteren Referenzen aufeinander.

Wir unterscheiden weiterhin zwischen Dialogkomponenten und ihren Instanzen. Die Unterscheidung ist analog zu der von Klasse und Objekt. Das bedeutet, dass es Dialogkomponenten geben mag, die dem Benutzer in mehreren Instanzen zur gleichen Zeit an verschiedenen Stellen angeboten werden. Ein typisches Beispiel dafür ist ein Bibliothekssystem, in dem die Detailansicht zu verschiedenen Büchern gleichzeitig geöffnet sein kann. Ein anderes Beispiel ist eine als Baustein verwendbare Dialogkomponente zur Eingabe einer Adresse, die auf einem Bestellformular sowohl für die Lieferadresse als auch für die Rechnungsadresse verwendet wird.

Dialogkomponenten werden über eine ID identifiziert.

Einbettung von Dialogen

Nach der Aufteilung in abhängige Dialogkomponenten müssen diese auch wieder zu einem Dialog mit gemeinsamer und konsistenter Oberfläche integriert werden. Diese Integration bezeichnen wir als Einbettung von Dialogen oder Embedding.

Embedding beinhaltet viele verschiedene Aspekte. Grob gesehen fallen diese Aspekte in die bereits bekannten Aufgabenkategorien: Darstellung, Daten, Aktionen und Zustände.

Der offensichtlichste Aspekt ist, dass die visuelle Darstellung integriert werden muss. Wenn z.B. ein Dialog eine Tab-Leiste enthält und jedes Tab von einem untergeordneten Dialog bereitgestellt wird, dann müssen die einzelnen Tab-Seiten in das Tab-Widget integriert werden. Da die visuelle Darstellung in der Verantwortung der Präsentation des untergeordneten Dialogs liegt, meldet dieser

seine visuelle Darstellung unter Verwendung der passenden Schnittstelle des übergeordneten Dialogs, der sie dann in seine eigene visuelle Darstellung integriert.

Weitere Beispiele für Embedding sind die Bereitstellung von Aktionen für Menüpunkte, die Ausgabe von Meldungen in einer Statuszeile, der Austausch oder die gemeinsame Verwendung von Daten sowie die Koordination bei der Ausführung von Aktionen. Dies gilt auch für optisch separate Dialoge in einem eigenen Fenster, die Aktionen im übergeordneten Dialog auslösen können. Zum Beispiel kann man in einem Fenster mit Detaildarstellung eines Listeneintrags auf den vorherigen oder nachfolgenden Eintrag der Liste springen, auch wenn diese Liste in einem anderen Fenster dargestellt wird.

Zu beachten ist, dass auch hier eine Dialogkomponente nicht direkt auf übergeordnete Komponenten zugreift. Zum Beispiel darf kein Unterdialog direkt den Titel eines Fensters manipulieren, da dieser in der Verantwortung eines anderen Dialogs liegt. Ein Unterdialog kann aber über eine von der übergeordneten Komponente angebotene Schnittstelle deren Dienste – z.B. zum Setzen des Fenstertitels – nutzen. Eine hierarchisch organisierte Dienstverwaltung (siehe „Kommunikation und Dienste“) sorgt dabei für eine Einbettung von Unterdialogen, ohne viel Wissen über die Dialogumgebung vorauszusetzen, in welche der Dialog eingebettet wird.

Dialograhmen

Der Dialograhmen stellt die Ablaufumgebung für Dialoge dar. Er bietet sowohl die grundlegende Infrastruktur zur Steuerung von Dialogen und zur Kommunikation zwischen Dialogen als auch übergreifende Funktionalität.

Logisch gesehen zerfällt der Dialograhmen in zwei Teile: die Sitzung und die Anwendung. Die Sitzung ist dialogübergreifend, aber benutzerspezifisch, die Anwendung ist sitzungsübergreifend und global. Die Trennung in Anwendung und Sitzung ist für einen Web-Client essenziell. Sie ist aber ebenso für eine Client-Architektur eines nativen Clients geeignet, da sich hinter der Trennung im Web wie im nativen Fall letztlich nur eine Trennung von Aufgaben verbirgt. Abbildung 11 illustriert die Komponenten des Dialograhmens mit ihren Abhängigkeiten.

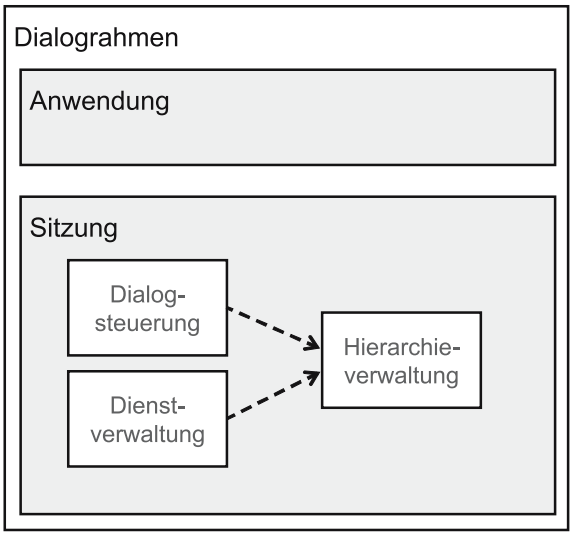

Abb. 11 Komponenten des Dialograhmens

Anwendung

Der Teil „Anwendung“ entspricht in seinem Lebenszyklus der gesamten Laufzeit des Clients vom Start bis zum Ende und bildet den übergreifenden Rahmen. In diesem Zusammenhang ist er für die globale (d.h. benutzerunabhängige) Konfiguration des Clients verantwortlich und sorgt dafür, dass globale Ressourcen oder Dienste zur Verfügung stehen. Die Anwendung steuert weiterhin ihre Sitzungen, d.h. sie erzeugt und beendet sie. Der einzige Unterschied zwischen einem „Single-User“-Client (wie meist beim nativen Client) und einem „Multi-User“-Client (wie im Web) ist, dass sich im ersteren Fall die Anwendung beenden kann, wenn keine Sitzung mehr vorhanden ist.

Sitzung

Der Teil „Sitzung“ umfasst die Interaktion mit einem spezifischen Benutzer. Sie beginnt mit dem An- und endet mit dem Abmelden des Benutzers. Die Sitzung ist benutzerspezifisch und wird von der Anwendung erzeugt. Sie ist daher auch für die benutzerspezifische Konfiguration zuständig, z.B. das Laden von Benutzereinstellungen.

So wie die Anwendung ihre Sitzungen steuert, steuert die Sitzung den Lebenszyklus ihrer Dialoge.

Die Sitzung ist der Teil des Dialograhmens, der den oben erwähnten technischen Dialogbegriff definiert: Sie ist für die Instanziierung, Verwaltung und Steuerung der Dialoge innerhalb der Dialoghierarchie zuständig und ermöglicht die Kommunikation

zwischen den Dialogen. Wir unterteilen daher die Sitzung in eine Dialogsteuerung, eine Hierarchieverwaltung und eine hierarchische Dienstverwaltung. Dialogsteuerung und Dienstverwaltung nutzen beide dieselbe Hierarchieverwaltung.

Semantisch gesehen liegen die Komponenten der Anwendung an der Wurzel der Dialoghierarchie, ihre Kinder sind die Sitzungskomponenten. Darunter liegen die Hauptdialoge, die Hauptfenstern oder Portalen entsprechen. Hauptdialoge können beliebig weiter in Unterdialoge strukturiert sein.

Dialogsteuerung und Lebenszyklus

Die Dialogsteuerung erzeugt und zerstört die Dialoginstanzen und organisiert sie in der Hierarchie. Sie definiert damit den Lebenszyklus der Dialoge. Zur Verdeutlichung der Aufgaben der Komponenten „Dialogsteuerung“ und „Hierarchieverwaltung“ sind in Abb. 12 und 13 einfache aber mögliche Schnittstellen der beiden Komponenten angegeben.

Für die Steuerung einfacher Komponenten wie in einem allgemeinen Komponentenframework würde ein Erzeugen und Zerstören reichen. Dialoge sind aber mehr als fachliche Komponenten: Sie besitzen in der Regel auch eine visuelle Darstellung. Mehr noch, wird ein Dialog aus mehreren Dialogen zusammengesetzt, so müssen die Darstellungen auch integriert werden (siehe Einbettung von Dialogen).

Bei dieser Integration wird der Ablauf bei der Einbettung der einzelnen Dialogkomponenten wichtig, da oft wechselseitige Abhängigkeiten zwischen über- und untergeordnetem Dialog auftreten, z.B. bei Größenberechnungen. Diese Abhängigkeiten müssen durch den Lebenszyklus aufgelöst werden.

Der Lebenszyklus definiert daher die Übergänge zwischen folgenden Zuständen:

- Erzeugt: Die Komponente des Dialogs (ohne Maske) ist bereitgestellt.
  Der Dialog wurde initialisiert und ist in der Hierarchie integriert. Er ist bereit für die Kommunikation von Dialog zu Dialog.
- Vorbereitet: Die Maske wird bereitgestellt.
  Die visuelle Darstellung des Dialogs ist erzeugt und verfügbar (z.B. zur Einbettung). Der Dialog stellt nun auch Dienste bereit, die sich auf seine Oberfläche beziehen, z.B. für das Einbetten der Darstellungen von Unterdialogen. Der Dialog erlaubt aber noch keine Interaktion mit dem Benutzer. Ressourcen, die zur Anzeige benötigt werden, können daher (noch oder wieder) freigegeben werden.
- Aktiv: Die Maske wird angezeigt.
  Der Dialog wird angezeigt und er akzeptiert Benutzereingaben. Wenn er selbst agierende Elemente enthält, z.B. Animationen, Polling etc., dann sind diese aktiv.

Ein Beispiel für Methoden eines Lebenszyklus und darauf aufbauend für das Interface, das einen Dialog implementieren muss, wenn er von einem Dialograhmen gesteuert werden will, sind in Abb. 14 zu sehen.

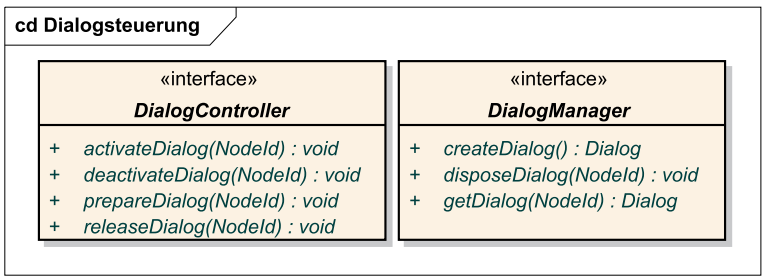

Abb. 12 Beispiel für die Schnittstelle einer Dialogsteuerung

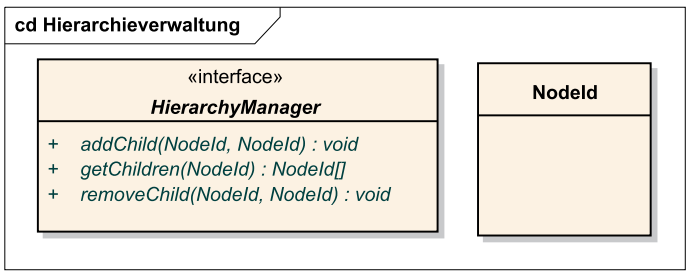

Abb. 13 Beispiele für die Schnittstelle einer Hierarchieverwaltung

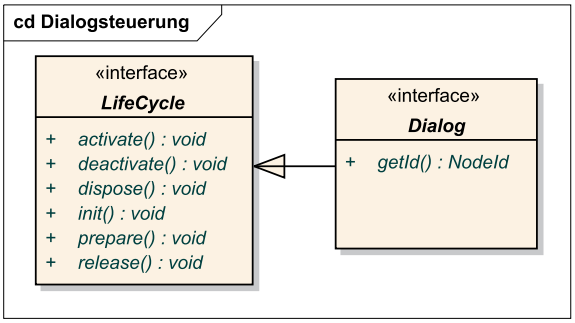

Abb. 14 Lebenszyklus- und Dialog-Schnittstelle

Bei einfachen Dialogen ist diese Untergliederung nicht notwendig – dort können die Übergänge weitgehend ignoriert werden. Sobald Dialogkomponenten unabhängig und wiederverwendbar zusammengesetzt werden, müssen sie sich an dieses Protokoll halten.

Die Dialogsteuerung stellt die Konsistenz des Lebenszyklus in der Hierarchie sicher: Ein untergeordneter Dialog kann nie in einem aktiveren Zustand sein als sein übergeordneter Dialog. Ein Übergang, der einen Dialog aktiver macht, wird zuerst auf dem übergeordneten Dialog durchgeführt, dann erst auf dem untergeordneten. Umgekehrt wird sichergestellt, dass ein untergeordneter Dialog zuerst deaktiviert wird, bevor sein übergeordneter Dialog deaktiviert wird. Abbildung 15 stellt eine vereinfachte Sicht des Zusammenspiels der Komponenten des Dialograhmens beim Deaktivieren eines Dialogs dar.

Kommunikation und Dienste

Die Kommunikation zwischen Dialogen untereinander und mit dem Dialograhmen (Sitzung und Anwendung) wird durch hierarchisch organisierte Dienste ermöglicht. Im Wesentlichen ist dies eine Service-Locator-Infrastruktur, die entlang der Dialoghierarchie einschließlich Sitzung und Anwendung organisiert ist.

Ein Dienst ist durch seine Schnittstelle definiert. Die Implementierung eines Dienstes kann von Komponenten innerhalb der Dialoghierarchie, der Sitzung oder der Anwendung angeboten werden und von anderen Komponenten angefragt werden. Beispiele für mögliche Dienste innerhalb des Clients sind Zugreifen auf Benutzerdaten, Anzeigen einer Meldung in der Statuszeile eines übergeordneten Dialogs oder Einbetten eines Menüeintrags in ein Menü eines Portals.

Beim Zugriff auf Dienste gibt es im Wesentlichen zwei Mechanismen: hierarchischer Zugriff

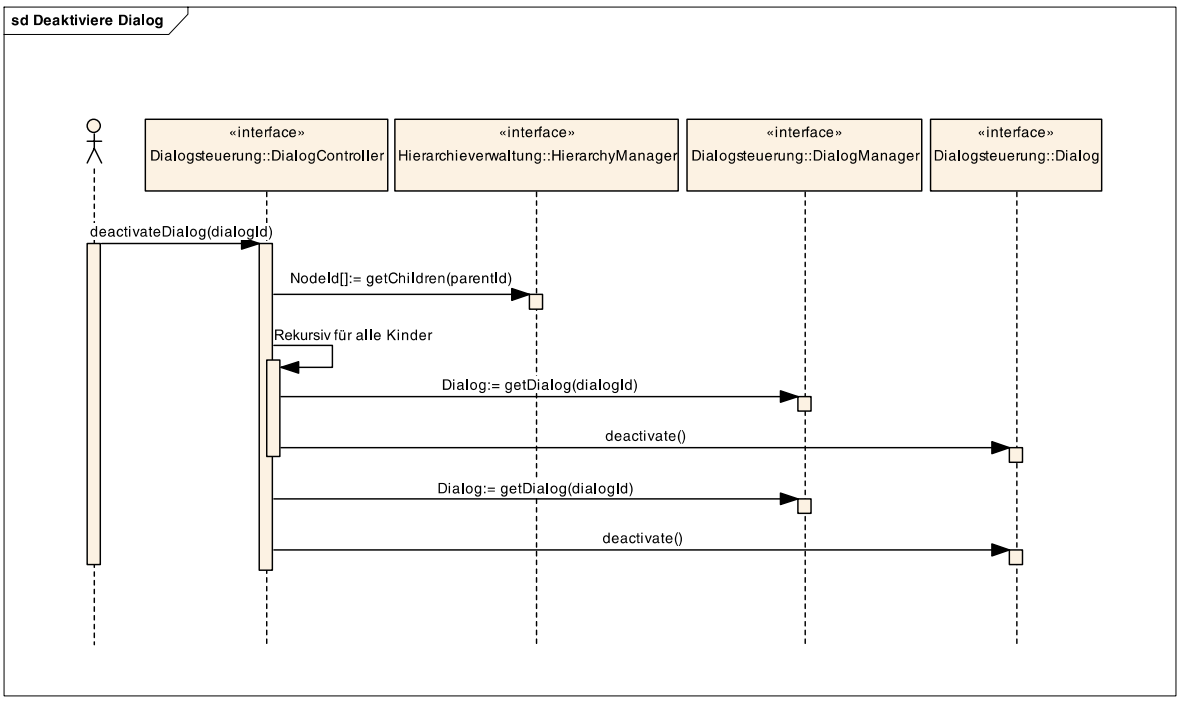

Abb. 15 Exemplarische Darstellung des Deaktivierens eines Dialogs

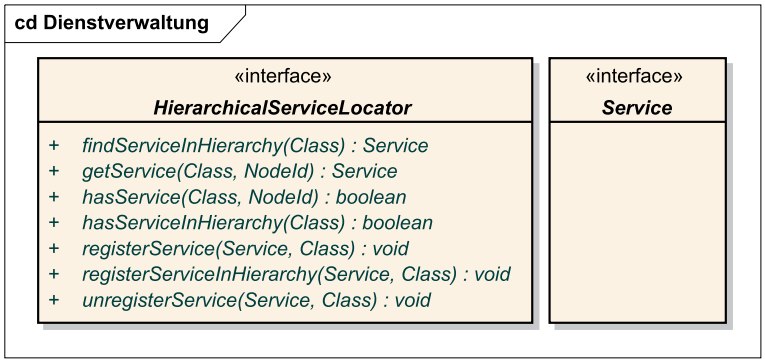

Abb. 16 Exemplarische Schnittstelle eines hierarchischen ServiceLocators der Komponente „Dienstverwaltung“

und direkter Zugriff. Beim hierarchischen Zugriff einer Komponente wird die erste Implementierung des Dienstes in der Hierarchie auf dem Weg zur Wurzel zurückgeliefert, beim direkten Zugriff wird die Implementierung des Dienstes des gewünschten Knotens zurückgeliefert (siehe Abb. 16).

Hierarchische Zugriffe dienen primär der Kommunikation von einer Komponente zu ihrer Umgebung, d.h. ihren übergeordneten Komponenten. Dadurch werden die Annahmen, die eine Komponente über ihre Umgebung trifft, kanalisiert und ohne starre Bindung behandelt. Die Komponenten der Anwendung und der Sitzung können ihre eigenen Dienste über die Dialoghierarchie den Dialogen anbieten. So können z.B. Dialoge auf die Konfiguration der Anwendung über einen hierarchisch erfragten Dienst zugreifen.

Direkte Zugriffe werden primär zur Kommunikation mit (direkten) Kindern einer Komponente verwendet. Die öffentliche Schnittstelle einer Kind-Komponente wird als Dienst angeboten, und die Vater-Komponente kann diesen Dienst per direkten Zugriff verwenden. Es ist auch möglich, auf Dienste beliebiger anderer Komponenten innerhalb der Hierarchie direkt zuzugreifen, sofern deren ID bekannt ist.

Der wesentliche Vorteil dieser Dienstarchitektur ist die starke Entkopplung der Komponenten. Das macht es einfach, gut strukturiert und komponentenbasiert zu entwickeln und klare Schnittstellen zur Kapselung zu verwenden. Direkte Referenzen auf die Implementierung von anderen Komponenten sind nicht vorgesehen und in der Architektur nur schwer realisierbar, sodass „Spaghettikomponenten“ zwar nicht ausgeschlossen, aber zumindest erschwert werden.

Wir bevorzugen diese Kommunikation mit Diensten gegenüber den in anderen Client-Architekturen verbreiteten Event-Mechanismen, z.B. bei HMVC [2] oder CAB [11], da sie eine typisierte Schnittstelle nahelegt. Ebenso ist die Existenz eines Dienstes abfragbar, während Events typischerweise ein „fire and forget“ bedeuten.

Ausblick

Wir wollen mit der hier vorgestellten Client-Architektur es ermöglichen, die Entwicklung von Clients beherrschbarer zu machen und verständlichere und wartbarere Software zu erhalten, die trotzdem flexibel an die schnell wandelnden Anforderungen der Client-Welt angepasst werden kann. Eine bessere Strukturierung führt auch dazu, dass Arbeit besser aufgeteilt und effizienter erledigt werden kann. Dazu tragen klare Trennungen wie z.B. die von Dialogen und Dialograhmen oder die Zerlegung von Dialogen in Dialogkomponenten bei. Sie dienen der Flexibilisierung der Dialogarchitekturen, der Entkopplung der Abläufe und damit der einfachen Nutzbarkeit. Gute Konzepte für Dialoge und deren Kommunikation unterstützen eine fachliche Entkopplung der Dialoge und damit die Wartbarkeit des Clients. Dies alles bleibt ohne eine durchdachte Architektur mehr oder weniger dem Zufall überlassen.

Die Architektur kann in verschiedener Form in Software gegossen werden. Wir haben gute Erfahrungen mit der auf Basis der Überlegungen selbst entwickelten Variante gemacht. In vielen der in den letzten Jahren erschienenen Frameworks existieren ähnliche Ansätze für etliche der genannten Aufgaben.

Selbst wenn die Client-Architektur nicht in Form von Sourcecode verwendet wird, hilft das Verständ-

nis der Prinzipien und Ideen, die Aufgaben von Clients in verschiedenen Umgebungen und Situationen besser zu analysieren und zu identifizieren und somit besser strukturierte Software zu entwickeln.

Literatur

1. Buschmann, F. et al.: Patternorientierte Softwarearchitektur, Kapitel 7 „Model-View-Controller". Addison-Wesley (1998)
2. Cai, J., Kapila, R., Pal, G.: HMVC: The layered pattern for developing strong client tiers. JavaWorld Magazine, http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html (2000)
3. Client Utilities and Framework (CUF), http://sourceforge.net/projects/cuf/
4. Coutaz, J.: PAC, an Object Oriented Model for Dialog Design. In: Proceedings Interact'87. North Holland (1987), S. 431–436
5. Eclipse Rich Client Platform (Eclipse RCP), http://wiki.eclipse.org/index.php/Rich_Client_Platform, http://www.eclipse.org/home/categories/rcp.php
6. Gram, C., Cockton, G. (Hrsg.) Design Principles for Interactive Software. Kapitel 4: Software Architecture Models. Chapman & Hall (1996)
7. Google Web Toolkit (GWT), http://code.google.com/webtoolkit/
8. Haft, M., Humm, B., Siedersleben, J.: The Architects' Dilemma – Will Reference Architectures Help? In: Reussner, R. et al. (Hrsg.) Proceedings of the First International Conference on the Quality of Software Architectures (QoSA 2005). Lecture Notes in Computer Science, vol. 3712, S. 106–122. Berlin, Heidelberg: Springer Verlag (2005)
9. JavaServer Faces (JSF), http://java.sun.com/javaee/javaserverfaces/
10. JGoodies, http://www.jgoodies.com
11. Microsoft patterns & practices: Smart Client – Composite UI Application Block, http://msdn.microsoft.com/practices/guidetype/AppBlocks/default.aspx?pull=/library/en-us/dnpag2/html/CAB.asp
12. Siedersleben, J.: Moderne Softwarearchitektur, Umsichtig planen, robust bauen mit Quasar, Kapitel 10: Architektur grafischer Bedienoberflächen. dpunkt.verlag (2004)
13. Spring Rich Client Project, http://www.springframework.org/spring-rcp
14. Szyperski, C.: Component Software – Beyond Object-Oriented Programming. Addison-Wesley (2002)