Für und Wider von Geschäftslogik in der Präsentationsschicht

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

Version vom 18.03.2007

Zusammenfassung: Eine wichtige Regel bei der Entwicklung von Informationssystemen lautet, in der Präsentationsschicht keine Geschäftslogik zu implementieren – Geschäftslogik gehört vielmehr in eine speziell darauf ausgelegte Domänenschicht. Auch wenn diese Regel grundsätzlich richtig und sinnvoll ist, schränkt sie die Möglichkeiten im Entwurf eines Systems ein. Insbesondere zur Vorbeugung von Performance-Problemen lässt es sich selten vermeiden, Geschäftslogik in die Präsentationsschicht aufzunehmen. Dieser Artikel geht den Ursachen dafür auf den Grund und zeigt Gestaltungsmöglichkeiten auf.

Einführung

Die Implementierung von Geschäftslogik in einem Informationssystem ist alles andere als trivial. Geschäftslogik muss nicht nur fachlich korrekt implementiert, sondern auch geeignet in das System integriert werden. Dabei sollte sie von der technischen Infrastruktur (z.B. für Persistenz und Kommunikation) und von der Steuerung der Benutzerinteraktionen (also der Präsentationsschicht) getrennt sein.

Ohne eine solche Trennung entstehen Abhängigkeiten zwischen der fachlichen und technischen Seite des Systems, die die eigentliche Geschäftslogik verschleiern können. Als Folge kann sich jede technische Änderung auf die Implementierung der Geschäftslogik auswirken. In einem mehrschichtig aufgebauten Informationssystem gehört sie daher in eine spezielle Domänenschicht.

Abbildung 1 zeigt als Beispiel die Architektur einer typischen mehrschichtigen Web-Anwendung. Ein Web-Client kommuniziert über HTTP-Aufrufe mit der Präsentationsschicht der Web-Anwendung. Die Präsentationsschicht teilt sich auf in die Steuerung der Interaktionen und die Aufbereitung der Ansichten, die an den Web-Client zurückgeliefert werden.

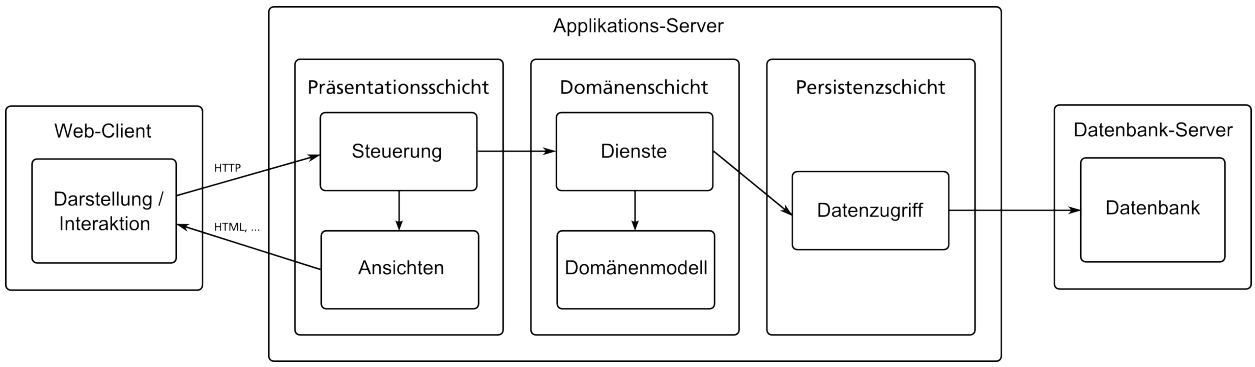

Abb. 1: Architektur einer Web-Anwendung

Die Präsentationsschicht greift für alle fachlichen Belange auf die Domänenschicht zu. Die Domänenschicht beinhaltet die Geschäftslogik. Sie besteht aus Diensten und einem objektorientierten Domänenmodell mit den Geschäftsobjekten. Die Domänenschicht verändert das Modell und liefert der Präsentationsschicht die gewünschten Informationen zurück. Das eigentliche Laden und Speichern der Daten übernimmt die Persistenzschicht, die mit der Datenbank kommuniziert.

Bei einer Rich-Client-Anwendung (s. Abb. 2) läuft die Präsentationsschicht lokal auf dem Rechner eines Anwenders. Sie kommuniziert dabei über Protokolle wie z.B. RMI oder SOAP mit der Domänenschicht im Applikations-Server und bekommt von ihr Daten übermittelt.

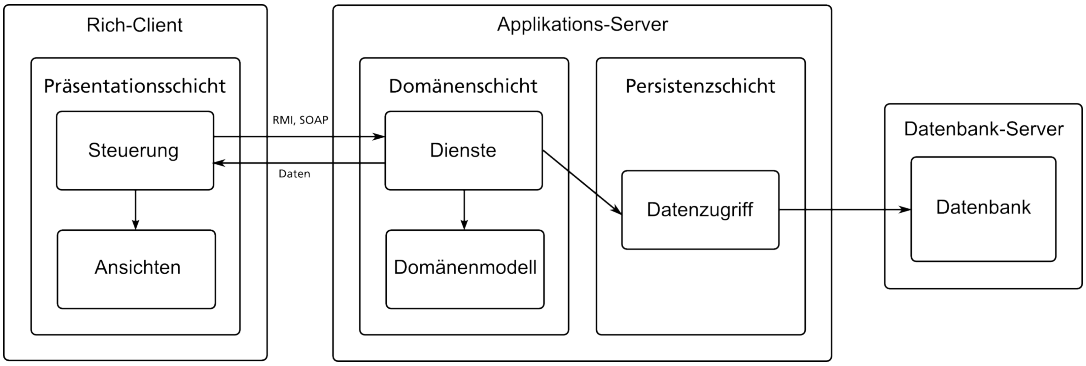

Abb. 2: Architektur einer Rich-Client-Anwendung

Für die Domänen- und Persistenzschicht bieten leichtgewichtige Frameworks wie *Spring* ([Spring]) Hilfe, eine der beschriebenen Architekturen umzusetzen. Wertvolle Erkenntnisse finden sich auch in der Literatur (z.B. [Evans], [Starke] und [Fowler]). Damit die Präsentationsschicht Geschäftsdaten anzeigen und Benutzerinteraktionen steuern kann, benötigt sie ebenfalls Logik. Wie dort Geschäfts- bzw. Präsentationslogik integriert werden kann, wurde bisher aber weniger ausführlich behandelt.

Geschäftslogik und Präsentationslogik lassen sich dabei nur schwer voneinander abgrenzen. Zur Präsentationslogik zählt vor allem die Aufbereitung der Geschäftsdaten zur Darstellung. Der Grat zwischen der reinen Formatierung eines Wertes (z.B. für die Ausgabe einer Zahl als Prozentwert) und Berechnungen auf Grundlage von Geschäftsdaten (z.B. für die Ausgabe des Verhältnisses zweier Werte) ist jedoch schmal. Im Weiteren wird daher nicht explizit zwischen beiden unterschieden.

Es gibt zunächst gute Gründen dafür, Geschäftslogik von der Präsentationsschicht fernzuhalten:

- *Konsistenz der Daten*. Die Präsentationsschicht ist im Normalfall nicht in Transaktionen eingebunden. Somit kann sie nicht die Datenkonsistenz sicherstellen.
- *Sicherheit*. Die Präsentationsschicht hat im Normalfall keinen Zugriff auf alle Sicherheitsmechanismen. Somit kann sie nicht die Rechtmäßigkeit von Datenzugriffen und -änderungen prüfen.
- *Konsistenz der Implementierung.* Wenn Geschäftslogik in der Präsentationsschicht implementiert wird, besteht die Gefahr, dass sich die Logik dort schleichend an viele Stellen verbreitet. Redundanzen und Inkonsistenzen sind die Folge. Client-Anwendungen leiden häufig darunter, dass Geschäftslogik innerhalb der Steuerungslogik verteilt ist, beispielsweise in der Ereignissteuerung der GUI-Komponenten.
- *Wartung*. Geschäftslogik in der Präsentationsschicht zu halten, kann bei jeder Änderung zu großen Schwierigkeiten führen: Wenn die GUI geändert werden soll, muss die Ge-

schäftslogik mit angepasst werden; wenn sich Geschäftslogik ändert, kann es schwierig werden, alle betroffenen Stellen in der GUI zu finden.

Ein System, das aus diesen Gründen vollständig auf Geschäftslogik in der Präsentationsschicht verzichtet, ist jedoch anfällig für Performance-Probleme. Dafür gibt es vor allem zwei Gründe:

- *Antwortzeiten*. Jeder Aufruf der Domänenschicht verlängert die Antwortzeiten der Benutzeraktionen, vor allem, wenn die Kommunikation nicht nur lokal in einem System stattfindet, sondern auch zwischen Systemen. Bei aufeinanderfolgenden, feinkörnigen Aufrufen der Domänenschicht addieren sich die Antwortzeiten.
- *Last*. Die Präsentationsschicht ist meistens zustandsbehaftet (z.B. `SessionContext` bei Servlets, Rich-Client-Anwendungen sowieso), die Domänenschicht ist dies üblicherweise nicht. Daher kann eine Präsentationsschicht Ergebnisse vorangegangener Berechnungen leichter wiederverwenden.

Lösungsansätze

Welche Möglichkeiten gibt es, mit den beschriebenen Problemen umzugehen? In der Präsentationsschicht völlig auf Geschäftslogik zu verzichten, ist aufgrund der drohenden Performance-Probleme kaum möglich. Somit stellt sich die Frage, wie die in der Präsentationsschicht benötigte Logik dort lokal implementiert werden kann, ohne die beschriebenen Nachteile in Kauf zu nehmen.

Hier ein kurzer Überblick über die im Folgenden beschriebenen Lösungsansätze:

- *Sie benötigen eine einfache Lösung und das Informationssystem als Ganzes ist eher übersichtlich?* Dann erweitern Sie die Geschäftsobjekte mit der in der Präsentationsschicht benötigten Funktionalität und reichen die Objekte an die Präsentationsschicht durch.
- *Die Präsentationsschicht erhält von der Domänenschicht Datentransferobjekte und benötigt wenig eigene Logik ?* Dann programmieren sie die Logik der Präsentationsschicht prozedural, z.B. in statischen Methoden.
- *Die Präsentationsschicht erhält Datentransferobjekte und benötigt viel eigene Logik?* Dann erstellen Sie in der Präsentationsschicht eine Hierarchie von Wrapper-Objekten.
- *Sie möchten dieselbe Logik in der Präsentationsschicht und in der Domänenschicht verwenden?* Dann erstellen Sie die Logik deklarativ, sodass sie in beiden Schichten verteilt und lokal aufgerufen werden kann.

Diese Lösungsansätze teilen sich eine Gemeinsamkeit: In der Präsentationsschicht werden Geschäftsdaten nur ausgewertet und zur Anzeige aufbereitet; die Veränderung der Daten bleibt der Domänenschicht vorbehalten.

Erweiterte Geschäftsobjekte

Eine Domänenschicht enthält die Geschäftslogik teilweise prozedural in Diensten, teilweise objektorientiert in einem Domänenmodell. Logik, die die Beziehungen zwischen mehreren Objekten verwaltet, ist eher prozedural organisiert; Logik, die sich auf die Daten eines einzelnen Geschäftsobjekts bezieht, ist eher in der betroffenen Klasse implementiert (siehe [Evans]).

Wenn sich die Komplexität eines Informationssystem in Grenzen hält, dann kann es sinnvoll sein, die Geschäftsobjektklassen um diejenige Logik zu ergänzen, die von der Präsentationsschicht benötigt wird. Die Geschäftsobjekte werden dann zur Laufzeit an die Präsentationsschicht weitergeleitet und deren Logik dort ausgeführt (s. Abb. 3).

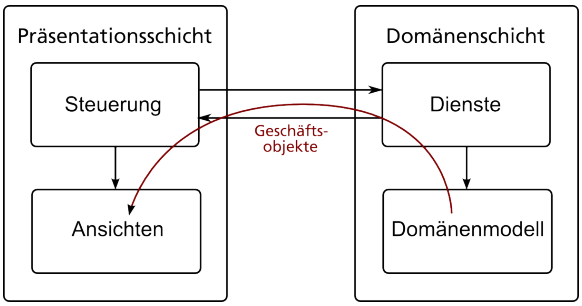

Abb. 3: Erweiterte Geschäftsobjekte

Ein Beispiel dazu: Angenommen, ein Statistikprogramm benötigt eine Web-Oberfläche. Diese Oberfläche soll Berechnungen anstoßen und die Ergebnisse anzeigen. Weiter angenommen, die Ergebnisdaten müssen für die Präsentation besonders formatiert werden, die Formatierung muss jedoch nicht flexibel sein. Dann bietet es sich an, die Klassen der Ergebnisobjekte um Formatierungsmethoden zu erweitern und die Objekte an die Präsentationsschicht durchzureichen.

Die Logik bleibt dadurch in sich konsistent und übersichtlich, da sie nur an einer Stelle implementiert wird. Es ist daher nicht zu befürchten, dass sich die Logik innerhalb der Präsentationsschicht an unterschiedlichen Stellen ausbreitet.

Dieser Ansatz geht jedoch mit einer engeren Kopplung der Domänenschicht an die Bedürfnisse der Präsentationsschicht einher. Das bedeutet, der Abstimmungsbedarf zwischen den Schichten, und dadurch zwischen den Verantwortlichen, wird größer und somit die Gefahr, dass die von der Präsentationsschicht benötigte Logik doch in die Präsentationsschicht selbst implementiert wird.

Zudem können die Geschäftsobjekte eventuell nicht vollständig in der Präsentationsschicht verwendet werden. Da sich die Präsentationsschicht meistens außerhalb eines Transaktionskontextes befindet, funktionieren Mechanismen wie das automatische Nachladen von Objekten durch einen O/R-Mapper nicht mehr. Zudem dürfen Funktionen zum Manipulieren der Geschäftsdaten nicht der Präsentationsschicht zugänglich gemacht werden. Schließlich muss geklärt werden, wie die Präsentationsschicht auf die kompilierten Klassen der Geschäftsobjekte zugreifen kann.

Dieser Ansatz eignet sich daher vor allem für Web-Anwendungen, in denen die Präsentationsschicht eng mit der Domänenschicht verbunden ist und der Fokus auf einer sauberen Trennung von Interaktion und Geschäftslogik liegt.

Prozedurale Logik

Seit dem Aufkommen der Objektorientierung genießt die prozedurale Entwicklungsmethode einen eher schlechten Ruf. Schließlich trennt sie Daten und Verhalten voneinander und erschwert dadurch das Verständnis für die Zusammenhänge ([Riel]). In vielen Systemen beruht der Datenaustausch zwischen der Präsentationsschicht und der Domänenschicht jedoch auf Datentransferobjekten. Das bedeutet, die Präsentationsschicht erhält nur noch Datenobjekte, die kein eigenes Verhalten aufweisen und nicht erweitert werden können.

Wenn in einem solchen Fall die Präsentationsschicht hauptsächlich für die Darstellung und Bearbeitung von Daten zuständig ist, also nur wenig Geschäftslogik benötigt, dann bietet sich ein prozeduraler Ansatz an. Das bedeutet, die Geschäftslogik wird in der Präsentationsschicht prozedural implementiert (z.B. in statischen Methoden) und zentral gesammelt (s. Abb. 4).

Ein Beispiel: Angenommen, ein Produktkatalog wird als Rich-Client-Anwendung entwickelt. In einer Seitenleiste soll der Gesamtpreis aller ausgewählten Produkte (ohne Rabatte, usw.) angezeigt werden. Ebenfalls angenommen, die Client-Anwendung erhält für jedes Produkt ein Daten-

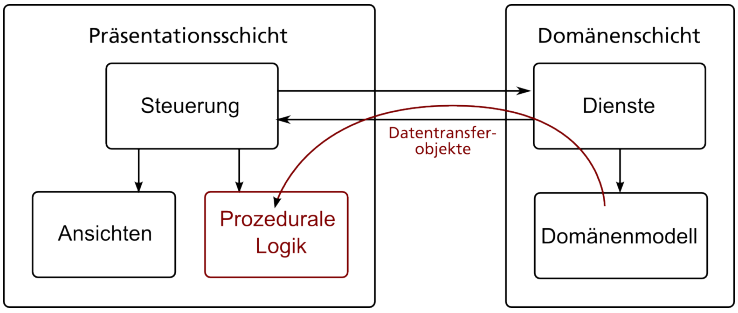

Abb. 4: Prozedurale Logik

transferobjekt der Klasse `Produkt`. Dann bietet es sich an, in einer Klasse `ProduktLogik` eine Methode mit folgender Signatur einzuführen:

```
public static Preis berechneGesamtpreis(List<Produkt> produkte)
```

Die in der Präsentationsschicht benötigte Geschäftslogik lässt sich mit diesem Vorgehen relativ einfach an wenigen Stellen sammeln und von der Steuerungslogik trennen. Wartungsprobleme durch gemischte Steuerungs- und Geschäftslogik werden vermieden. Ebenso bleiben die Präsentationsschicht und die Domänenschicht nur durch die Datentransferobjekte miteinander verbunden.

Der Nachteil dieser Lösung liegt darin, dass sie leicht zu Redundanzen führt. Statische Methoden gelten zudem eher als Anzeichen für ein schlecht modelliertes objekt-orientiertes System; die Vorteile der Objektorientierung (Vererbung, Komposition) gehen verloren.

Dementsprechend eignet sich dieser Ansatz besonders gut für Systeme, in denen die Präsentationsschicht in erster Linie für die Darstellung und Pflege von Daten benötigt wird und die Systemschichten klar voneinander getrennt werden sollen.

Wrapper-Objekte

Bei komplexen Systemen mit einem ebenso komplexen Geschäftsobjektmodell kann es sinnvoll sein, nicht die volle Komplexität an die Präsentationsschicht weiterzureichen, sondern nur einen Ausschnitt davon. Datentransferobjekte sind hierfür eine Lösung.

Ein komplexes System hat häufig eine komplexe Benutzungsoberfläche. Dementsprechend steigt der Anspruch an die Geschäftslogik in der Präsentationsschicht und somit der Vorteil einer objektorientiert entwickelten Schnittstelle.

Diese beiden Konzepte, also Datentransferobjekte und objektorientierte Geschäftslogik, lassen sich durch den Einsatz von Wrapper-Objekten zusammenbringen: Für jedes aus der Domänenschicht gelieferte Transferobjekt wird in der Präsentationsschicht ein Wrapper-Objekt erzeugt, in dem das Transferobjekt gekapselt und die Geschäftslogik dafür bereitstellt wird (s. Abb. 5).

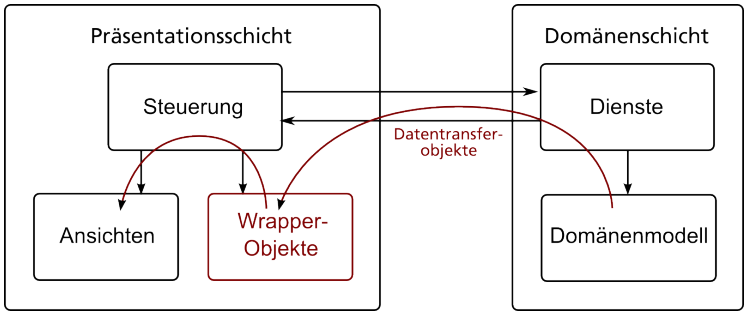

Abb. 5: Wrapper-Objekte

Ein Beispiel für dieses Vorgehen: Ein Planungssystem soll die Abhängigkeitsstruktur eines Projektplans anzeigen. Die Domänenschicht liefert dazu auf Anfrage Datentransferobjekte für alle Planelemente. Damit die Präsentationsschicht den Projektplan darstellen kann, muss sie eine Reihe von Abfragen darauf ausführen: Welche Knoten sind die Ausgangspunkte, welche sind die Endpunkte? Wo liegt der kritische Pfad? Usw.

Die für diese Abfragen benötigten Berechnungen lassen sich so in Wrapper-Objekte integrieren, dass die Präsentationsschicht nur gegen die Wrapper-Objekte programmiert wird. Das bedeutet, aus Sicht der Präsentationsschicht sind Datenobjekte und Verhalten vereint.

Wrapper-Objekte bieten somit die Vorteile der objektorientierten Programmierung von Geschäftslogik in der Präsentationsschicht. Wenn die Wrapper-Objekte in der Präsentationsschicht zentral gebündelt werden, verringern sie die Gefahr, dass sich die Geschäftslogik unkontrolliert verbreitet. Zudem müssen Wrapper-Objekte nicht zwangsläufig dieselbe Klassenhierarchie aufweisen wie die Datentransferobjekte, wodurch sich Unzulänglichkeiten in der Domänenschicht bzw. in Fremdsystemen ausgleichen lassen.

Wrapper-Objekte erhöhen jedoch die Komplexität eines Systems, da sie eine weitere Struktur einführen. Jede Änderung am Domänenmodell kann eine Änderung der Wrapper-Objekte nach sich ziehen. Insbesondere bei einem Domänenmodell mit tiefen Klassenhierarchien und vielen Assoziationen zwischen den Entitäten entsteht für die Pflege der Wrapper-Objekte viel Aufwand.

Somit eignet sich dieser Ansatz vor allem dann, wenn in der Präsentationsschicht viel Logik für die Darstellung und Interaktion benötigt wird und der Einsatz objektorientierter Strukturen für die Organisation der Logik größere Vorteile mit sich bringt.

Deklarative Geschäftslogik

Manche Formen von Geschäftslogik werden in mehreren Schichten in identischer Form benötigt. Ein typischer Fall dafür ist die Validierung von Benutzereingaben. In einer Web-Anwendung sollten die eingegebenen Daten beispielsweise schon in der Präsentationsschicht überprüft werden. Aber auch die Domänenschicht muss alle zu verarbeitenden Daten prüfen, damit keine korrupten Daten an der Präsentationsschicht vorbei in das System eingeschleust werden können.

Manchmal ist zu Beginn der Entwicklung noch nicht ersichtlich, innerhalb welcher Schicht Logik am besten ausgeführt wird. Aus Performance-Gründen soll diese Entscheidung beispielsweise erst zu einem späteren Zeitpunkt getroffen werden.

Ein sinnvoller Ansatz kann dann darin bestehen, Geschäftslogik deklarativ zu entwickeln. Das bedeutet, die Logik wird in einer geeigneten, eventuell proprietären und auf das fachliche Problem abgestimmten Domänensprache beschrieben. Dateien, z.B. in einem XML-Format, lassen sich dann relativ problemlos an verschiedenen Stellen des Systems auslesen und verarbeiten (s. Abb. 6).

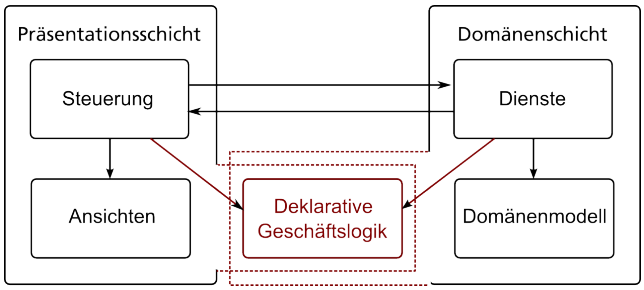

Abb. 6: Deklarative Geschäftslogik

Um bei dem Beispiel der Eingabe-Validierung zu bleiben: Eine Bibliothek wie *Jakarta Commons Validator* ([Validator]) kann in unterschiedlichen System-Schichten verwendet werden. Die eigentliche Geschäftslogik, also die fachlichen Regeln zur Prüfung der Datensätze, wird in Form von XML-Dateien erstellt.

Eine deklarative Beschreibung der Geschäftslogik ermöglicht es, die Logik dynamisch im System zu verteilen. Die Stärke dieses Konzepts tritt vor allem dann hervor, wenn sich die Anforderungen an die Geschäftslogik vereinheitlichen lassen. Eine eigene, spezialisierte Domänensprache kann den Gesamtaufwand für die Erstellung der Fachlogik reduzieren.

Diese Stärke ist zugleich die Schwäche: Eine eigene Domänensprache zu entwerfen ist aufwändig und in vielen Fällen kaum möglich. Der Aufwand steht daher eventuell nicht in Relation zum Nutzen. Somit eignet sich dieser Ansatz vor allem dann, wenn geeignete Domänensprachen und die technische Unterstützung dafür schon existieren und verwendet werden können.

Fazit

In vielen Systemen lässt es sich nicht vermeiden, Geschäftslogik in die Präsentationsschicht zu integrieren. Geschäftslogik sollte dort zwar nur für die Auswertung und Darstellung von Daten verwendet werden; manche Systeme haben daran jedoch große Ansprüche.

Die Herausforderung besteht darin, eine passende Lösung zwischen zwei Extremen zu finden: Wenn die Präsentationsschicht lokal auf zu wenig Geschäftslogik zugreifen kann, beeinträchtigt dies die Performance des Systems; wenn die Präsentationsschicht lokal zu viel Logik enthält, besteht die Gefahr von Inkonsistenzen, Redundanzen und Wartungsschwierigkeiten.

Ausgehend von diesem grundsätzlichen Problem wurden vier Ansätze vorgestellt, die für jeweils spezifische Gegebenheiten Hilfe bieten. Eine Lösung, die in jedem Fall hilft, ist leider nicht dabei. Somit bleibt die Entwicklung von Geschäftslogik in der Präsentationsschicht auch weiterhin ein schwieriges Feld.

Literatur und Links

[Evans] E. Evans, *Domain-Driven Design,* Addison Wesley, 2004

[Fowler] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003

[Riel] A. Riel, *Object-Oriented Design Heuristics,* Addison Wesley, 1996

[Spring] *Spring Framework*, siehe http://www.springframework.org

[Starke] G. Starke, *Effektive Software-Architekturen*, Hanser, 2005

[Validator] *Jakarta Commons Validator Project*, siehe http://jakarta.apache.org/commons/validator