Die Objektorientierte Hülle – Erweiterbarkeit imperativ-prozeduraler Altsysteme durch Verschalung

Henning Schwentner, Jens Barthel

C1 WPS GmbH, Vogt-Kölln-Straße 30, 22527 Hamburg
{henning.schwentner, jens.barthel}@c1-wps.de

Für einen Kunden aus dem Leasinggeschäft war das Vertragsverwaltungssystem ERIKA gebaut worden. Die technische Basis ist SAP mit der Programmiersprache ABAP. Der ursprüngliche Entwurf war imperativ-prozedural. Als wir in das Projekt kamen, entstand der Wunsch, auch „moderne“ Techniken wie objektorientierte Programmierung zu verwenden. Damit sollte insbesondere die Testbarkeit verbessert werden.

Das System ist zwar imperativ entworfen, aber nicht ohne Architekturvorstellung. Insbesondere gilt die Architekturregel: „kein direkter Zugriff auf die Datenbank“. Stattdessen dürfen nur sogenannte Read- und Write-Funktionsbausteine aufgerufen werden, die den eigentlichen Datenbankzugriff kapseln. Auf diese Weise ruft die Geschäftslogik zwar die Datenbank nicht direkt, sie ist aber immer noch indirekt von der Datenbank abhängig.

Dies verursacht verschiedene Probleme. So ist die Geschäftslogik schwierig zu testen. Angenommen der Funktionsbaustein `BERECHNE_MARGE` ruft den Read-Funktionsbaustein `LIES_VERTRAG` auf. Ein Modultest, der `BERECHNE_MARGE` aufruft, ist dann automatisch auch von `LIES_VERTRAG` abhängig.

Wie löst man diese Abhängigkeit? Die von uns verwendete Variante ist die Einführung von „Objektorientierten Hüllen“. Dazu wird zu einem Read-Funktionsbaustein ein Interface erzeugt. Dieses Interface erhält nur eine Methode. Die Parameterliste dieser Methode entspricht der des verhüllten Funktionsbausteins. Im Beispiel zum Legacy-Funktionsbaustein `LIES_VERTRAG(importing VERTRAGSNUMMER VNR returning VERTRAG VT)` also ein Interface `IF_VERTRAGS_LESER` mit der Methode `LIES(importing VERTRAGSNUMMER VNR returning VERTRAG VT).`

Das Interface bekommt eine Standardimplementierung. In dieser wird die Methode so implementiert, dass sie den verhüllten Funktionsbaustein aufruft und alle Parameter eins-zu-eins weitergibt. Im Beispiel also Klasse `CL_VERTRAGS_LESER`, bei der die Methode LIES so implementiert wird, dass sie alle Parameter an `LIES_VERTRAG` weiterleitet.

Nun wird neue Geschäftslogik so geschrieben, dass sie statt dem Funktionsbaustein `LIES_VERTRAG` direkt nun die Methode `IF_VERTRAGS_LESER->LIES` des neuen Interface aufruft. Im Produktivbetrieb wird als Implementierung von `IF_VERTRAGS_LESER` die Klasse `CL_VERTRAGS_LESER` verwendet. Modultests schieben dem *object under test* als Implementierung von `IF_VERTRAGS_LESER` eine eigene Klasse unter, mit der die für den Test gewünschten Rückgabewerte injiziert werden können.

In dem Legacy-System wurden fachlich zusammenhängende Datensätze ausschließlich durch Fremdschlüsselbeziehungen in der Datenbank repräsentiert. Um diese Beziehungen zu verdeutlichen und einen einfacheren Zugriff auf einzelne Komponenten fachlicher Entitäten zu gewährleisten, bot es sich an, diese als Klassen nachzubilden.

So gab es z. B. eine Tabelle VTKOPF für die Vertragskopfdaten und eine Tabelle VTPOSI für Vertragspositionsdaten. Es bot sich an, für die fachliche Entität „Vertrag“ eine Klasse CL_VERTRAG einzuführen. Die Implementierung dieser Klasse arbeitet dann auf einer Struktur vom Typ VTKOPF. Wenn die Struktur Felder wir VTNR (für Vertragsnummer) oder GBEREI (für Geschäftsbereich) anbot, so führten wir Getter- und Setter-Methoden wie GIB_VERTRAGSNUMMER oder SETZE_GESCHAEFTSBEREICH ein. So erhielten wir etwas ähnliches wie eine JavaBean.

Im nächsten Schritt führten wir in die Klasse CL_VERTRAG Vor- und Nachbedingungen ein. Zum Beispiel zur Überwachung von Statusübergängen. Dann fügten wir „richtige“ Geschäftslogik hinzu. CL_VERTRAG erhielt dann z.B. eine Methode ABRECHNEN.

In ABAP gibt es sogenannte klassische Ausnahmen und objektorientierte Ausnahmen. Die klassischen Ausnahmen sind einfach benannte Returncodes, die nicht weitergeworfen werden. Dies bedeutet, dass die Fehlerbehandlung immer direkt nach dem Aufruf einer Funktion mit klassischen Ausnahmen gemacht werden muss.

In ERIKA sind die Read- und Write-Funktionsbausteine mit klassischen Ausnahmen versehen. Im Rahmen der Einführung von objektorientierten Hüllen haben wir die verhüllenden Methoden gleich mit objektorientierten Ausnahmen versehen. Die verhüllenden Methoden prüfen den Returncode der verhüllten Funktion und werfen bei einem Fehler eine passende objektorientierte Ausnahme.

Neue Entwicklungen konnten nun so entworfen werden, dass sie einerseits die objektorientierten Hüllen von bestehendem Coding statt des Codings selbst aufrufen konnten. Das hatte wieder den Vorteil, dass dieses neue Modul einfach mit einem Modultest versehen werden konnte. Es war sogar möglich, hier testgetrieben vorzugehen, d. h. dass wir durch die neue Infrastruktur erst ein Stück Test, dann ein Stück Code usw. schreiben konnten. Außerdem musste das neue Modul nicht auf Strukturen und Datenbanktabellen wie VTKOPF aufsetzen. Stattdessen konnten wir hier die höhere Abstraktion CL_VERTRAG verwenden.

Durch Einführung von objektorientierten Hüllen und Fachlichen Metaphern (Materialien) als Klassen ermöglichen wir, ein Altsystem auch mit modernen Mitteln weiterzuentwickeln. Dies ist genau dann gut möglich, wenn das Altsystem über einen zentralisierten Datenbankzugriff verfügt. Dieser kann dann relativ einfach verkapselt werden. Ohne zentralisierten Zugriff müsste jeder einzelne Datenbankzugriff ersetzt werden.

Für neue Module ist ein objektorientierter Entwurf und eine objektorientierte Implementierung möglich. Die neuen Module können leicht mit Modultests versehen werden, was u. a. Test-First-Programming ermöglicht. Dabei ist auch die Verwendung von Techniken wie Dependency Injection und Mocking möglich.

Das hier beschriebene Vorgehen wurde mit ABAP umgesetzt, es ist allerdings nicht auf diese Programmiersprache begrenzt und sollte genauso in anderen Umgebungen funktionieren, in denen imperative Systemteile nun auch objektorientiert verwendet werden können sollen.