Raus aus der Dependency-Hölle

Das fachlich offene Dependency-Management

Reik Oberrath

Unter den beiden Paradigmen komponentenorientierte Architektur und fachlich getriebene Komponentenbildung ist das Dependency-Management häufig schwierig. Bei wachsender Funktionalität und zunehmender Komplexität hat man es schnell mit einem dichten Netz von Abhängigkeiten zu tun, in dem es immer schwieriger wird, gegenseitige Abhängigkeiten und Abhängigkeitszyklen zu vermeiden: die Dependency-Hölle. Dieser Artikel zeigt, wie fachlich entworfene Komponenten voneinander entkoppelt werden können, sodass technische Beschränkungen die fachliche Komponentenbildung nicht mehr beeinträchtigen: das fachlich offene Dependency-Management.

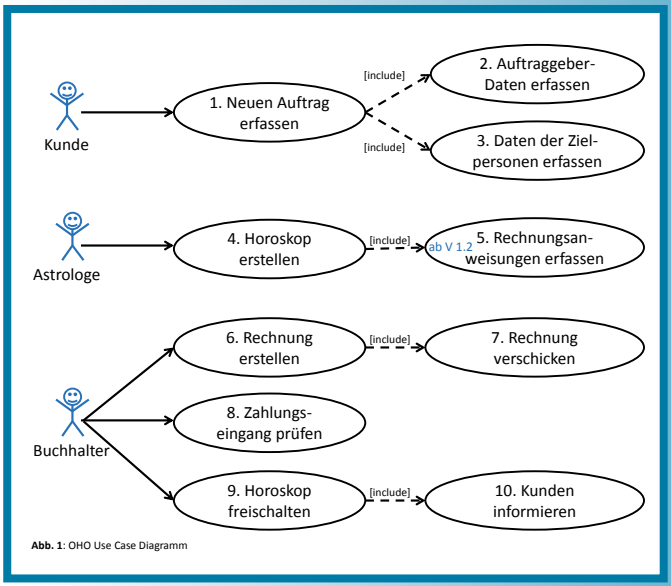

Abb. 1: OHO-Use-Case-Diagramm

Fallstudie: Online Horoskope

▶ Dieser Artikel basiert auf den beiden Paradigmen
▼ komponentenorientierte Architektur [KE] und
▼ fachlich getriebene Komponentenbildung [Sinz99].
Die Dependency-Hölle, die mit diesem Ansatz häufig verbunden ist, sowie der Ausweg aus dem Abhängigkeitswirrwarr wird anhand der fiktiven Webanwendung Online-HOroskop (OHO) dargestellt. Die grundlegenden Begriffe dieses Artikels werden im Kasten „Begriffserläuterungen" zusammengefasst.

Die Webanwendung OHO, Version 1.0

Die Astrologin „Starlet" macht sich selbstständig und möchte online einen Horoskop-Dienst anbieten. Sie beauftragt eine kleine Software-Schmiede namens „Gut&Günstig" (G&G), die Webanwendung OHO zu entwickeln. Diese Anwendung soll es Kunden (Auftraggebern) ermöglichen, die Erstellung eines Horoskops online sowohl zu beauftragen als auch das erstellte Horoskop nach erfolgter Zahlung abzurufen.

Im Auftrag muss ein Auftraggeber alle nötigen Angaben zum Horoskop angeben: Kontoverbindung des Auftraggebers und persönliche Daten der Zielperson, für die ein Horoskop erstellt werden soll. Der Auftragsteller kann zwischen vollständigem Horoskop und verschiedenen Teilhoroskopen (z. B. nur Ratschläge in Sachen Liebe, Finanzen, usw.) auswählen. Aus den persönlichen Daten der Zielperson wird ein Horoskop erstellt, das in der Datenbank der Anwendung gespeichert wird. Außerdem wird mit OHO die Rechnungsstellung abgewickelt und nach Zahlungseingang das Horoskop online zur Verfügung gestellt (Abb. 1).

Die G&G-Entwickler realisieren OHO mit Java, Spring und Maven. Um die Qualität der Software zu testen, schreiben sie sowohl Unit-Tests als auch Integrationstests. Letztere leiten von der Spring-Klasse `AbstractTransactionalSpringContextTests` ab. Diese Klasse löst die Abhängigkeiten zwischen den Spring-Beans verschiedener Komponenten automatisch auf. Die OHO-Anwendung besteht aus den vier Komponenten *Stammdaten*, *Auftrag*, *Horoskop* und *Abrechnung*.

Das klassische Dependency-Management

Zunächst ist das Abhängigkeitsgefüge klar: Sowohl für die Erstellung des Horoskops als auch für die der Abrechnung werden Auftragsdaten benötigt. Aus diesem Grund referenziert sowohl ein Horoskop als auch eine Abrechnung den dazugehörenden Auftrag. Die OHO-Version 1.0 beinhaltet die in Abbildung 2 gezeigten Abhängigkeiten.

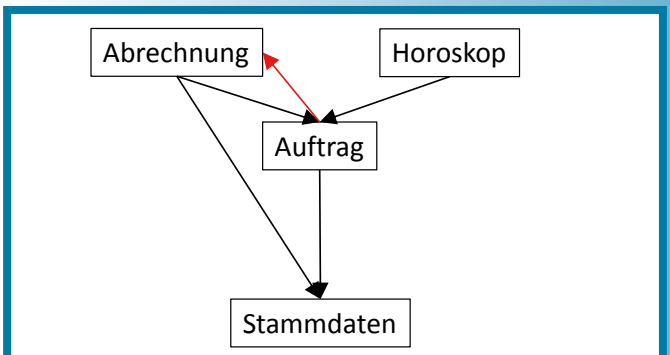

Abb. 2: OHO 1.0-Abhängigkeiten – Die beiden Komponenten *Abrechnung* und *Horoskop* stehen in der Dependency-Hierarchie ganz oben. Rot: diese Abhängigkeit wäre für OHO 1.1 nötig, ist aber technisch nicht möglich

Das Geschäft mit OHO 1.0 läuft gut und Starlet kann viele Horoskope online erstellen und abrechnen. Die Arbeit wird bald so viel, dass sie sich ganz auf die fachliche Arbeit konzentrieren möchte, d. h. auf die Erstellung der Horoskope. Sie stellt den Buchhalter „Billy" ein, der sich um die Abrechnungen kümmert. In der Zusammenarbeit mit Billy gibt es aber immer wieder Reibungsverluste, denn aus Starlets Sicht sind die Abrechnungen nicht immer korrekt. Außerdem beschweren sich die Kunden über mangelnde Transparenz der Kosten bei der Auftragsstellung.

Aus diesem Grund sollen die Rechnungspositionen im Auftrag angezeigt werden und damit sowohl für den Kunden als auch für Starlet leichter einzusehen sein. Technisch bedeu-

Begriffserläuterungen

- *Komponentenorientierte Architektur [KE]:* Vorgehensweise, den Quellcode in separate Bestandteile bzw. einzelne Bausteine zu unterteilen, mit dem Hauptziel, durch die Einhaltung des Prinzips „Separation of Concerns“ [CCDSoC] die strukturelle Qualität und damit die Wartbarkeit des Quellcodes zu verbessern. Weiteres Ziel ist außerdem, mehr Möglichkeiten zu schaffen, Quellcode wiederzuverwenden.
- *Komponente:* Komponenten sind die großen Bausteine einer deploybaren Softwareeinheit (z. B. jar-Dateien innerhalb einer ear-Datei). Dieser Artikel unterscheidet bewusst nicht zwischen Komponente und Modul [Modul], da beide Bausteinarten die gleichen Dependency-Probleme aufweisen. Komponenten bestehen aus kleineren Bausteinen wie kompilierten Klassen und anderen Ressourcen. Eine OHO-Komponente wird durch die pom-Datei ihres Maven-Projekts definiert.
- *Fachliche und technische Komponenten:* Fachliche Komponenten stellen separate Funktionsblöcke dar, die sich aus der Geschäftslogik ableiten und damit fachlich begründen lassen. Technische Komponenten sind Bausteine des Softwaresystems, die keine fachliche Bedeutung haben und für die Bewältigung von technischen Problemen entworfen wurden.
- *Fachliche Abhängigkeiten:* Logische, rein konzeptionelle Abhängigkeiten einer Komponente von einer anderen, z. B. der Komponente *Auftrag* von der Komponente *Stammdaten*, weil letztere Kundendaten zur Verfügung stellt, die in *Auftrag* benötigt werden. Fachliche Abhängigkeiten entstehen beim Design der Komponenten dort, wo Funktionsblöcke wie Auftrag und Stammdaten konzeptionell voneinander getrennt werden.
- *Technische Abhängigkeiten:* Auf verschiedenen Abstraktionsebenen gibt es unterschiedliche Arten von technischen Abhängigkeiten. Auf Klassenebene existieren Abhängigkeiten, die wechselseitig sein können (Klasse A ist abhängig von Klasse B und umgekehrt). Auf Komponentenebene existieren Abhängigkeiten, die nicht wechselseitig sein dürfen (Wenn Komponente A von B abhängt, darf B nicht von A abhängig sein). Abhängigkeiten zwischen Komponenten sind die Ursache für die im Artikel beschriebene Dependency-Hölle.
- *Dependency-Hierarchie:* Komponenten müssen aufgrund der technischen Abhängigkeiten in einer bestimmten Reihenfolge gebaut werden. Basis-Komponenten, die als erstes gebaut werden können, liegen in der Dependency-Hierarchie unten. Andere Komponenten, die auf den Basis-Komponenten aufbauen, liegen in der Dependency-Hierarchie darüber.
- *Fachliche Offenheit:* Nach dem Open-Closed-Prinzip [CCDOCP] soll ein System geschlossen sein für Veränderungen, aber offen für Erweiterungen. Muss ein System für eine Erweiterung umgebaut werden, ist dieses Prinzip verletzt. Können Erweiterungen ohne Umbau vorgenommen werden, ist das System „offen“. Dieser Artikel stellt ein Dependency-Management vor, in dem neue fachliche Abhängigkeiten ohne Umbau des bisherigen Beziehungsgeflechts realisiert werden können. Dieses Dependency-Management wird hier deshalb als „fachlich offen“ bezeichnet.

tet das, dass die Komponente *Auftrag* eine Abhängigkeit zu *Abrechnung* benötigt, um die zum Auftrag gehörenden Rechnungsdaten einzulesen. Das führt allerdings zu einer wechselseitigen fachlichen Abhängigkeit zwischen den Komponenten *Auftrag* und *Abrechnung* (s. Abb. 2, roter Pfeil), die technisch nicht realisiert werden kann (s. Kasten „Begriffserläuterungen“, Technische Abhängigkeiten). Die G&G-Entwickler diskutieren mehrere Lösungsmöglichkeiten:

- Die Gemeinsamkeiten der Komponenten *Auftrag* und *Abrechnung* werden aus den beiden Einzelkomponenten herausgezogen und daraus eine neue Komponente erzeugt. Zwei Nachteile dieser Lösung führen zur Ablehnung. Erstens würde aus technischen Gründen eine neue Komponente gebaut, die fachlich nicht gut zu begründen ist. Zweitens würden die Komponenten *Auftrag* und *Abrechnung* dadurch so klein, dass sie ihre Existenzberechtigung verlieren.
- Die Komponenten *Auftrag* und *Abrechnung* werden zu einer verschmolzen. Auch diese Lösung wird verworfen, um dem Paradigma der komponentenorientierten Architektur treu zu bleiben und eine beginnende Monolithen-Architektur zu vermeiden.
- Die bisherige Vorgehensweise wird geändert: Statt eine leere Abrechnung zu erstellen und von dort einen Auftrag zu referenzieren, wird für einen selektierten Auftrag eine neue Abrechnung erzeugt und von dort werden alle für die Abrechnung nötigen Auftragsdaten in die neue Abrechnung kopiert.

Der Nachteil der letzten Alternative ist redundante Informationen im Auftrag und in der dazugehörenden Abrechnung. Der Vorteil aber für das Dependency-Management besteht darin, dass die Komponente *Abrechnung* technisch nicht mehr von der Komponente *Auftrag* abhängig ist. Jetzt kann die benötigte umgekehrte Abhängigkeit der Komponente *Auftrag* von der Komponente *Abrechnung* eingebaut werden (s. Abb. 3). Deshalb wählen die G&G-Entwickler diese Alternative.

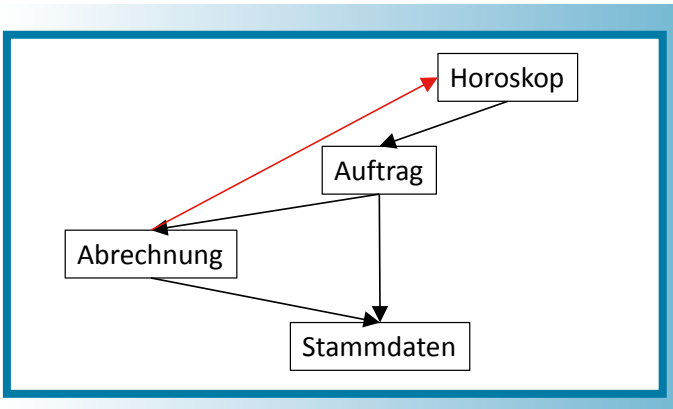

Abb. 3: OHO 1.1-Abhängigkeiten – Die Komponente *Abrechnung* wurde in der Dependency-Hierarchie unter die Komponente *Auftrag* geschoben. Rot: diese Abhängigkeit wäre für OHO 1.2 nötig, ist aber technisch nicht möglich

Fortdauernder Umbau

Das Geschäft mit O HO 1.1 läuft weiterhin gut und Hunderte Horoskope werden erstellt und abgerechnet. Starlet kann jetzt einfacher die Rechnungspositionen eines Auftrages einsehen, ist aber in vielen Einzelfällen mit einigen Buchungsdetails nicht einverstanden. Mit der Zeit wird klar, dass die Kommunikation zwischen den beiden Fachkräften Starlet und Billy noch effektiver sein würde, wenn Starlet bei Bedarf Abrechnungsanweisungen bei der Erstellung des Horoskops im Horoskop

speichern und diese Information von Billy bei der Abrechnungserstellung ausgewertet werden könnte.

Diese neue Anforderung bedeutet eine fachliche Abhängigkeit der Komponente *Abrechnung* von der Komponente *Horoskop*. Technisch würde das zu einem Abhängigkeitszyklus zwischen den Komponenten *Horoskop*, *Auftrag* und *Abrechnung* führen (s. Abb. 3). Das neue Abhängigkeitsdilemma wird von den G&G-Entwicklern diskutiert und sie einigen sich auf folgende Lösung:

Für einen selektierten Auftrag wird ein neues Horoskop erzeugt und alle für das Horoskop nötigen Auftragsdaten werden aus dem Auftrag in das Horoskop kopiert. Der Nachteil dieser Vorgehensweise sind weitere redundante Informationen. Der Vorteil aber für das Dependency-Management besteht darin, dass die Komponente *Horoskop* technisch nicht mehr von der Komponente *Auftrag* abhängig ist. Jetzt können die G&G-Entwickler in OHO 1.2 die neue benötigte Abhängigkeit von der Komponente *Abrechnung* zur Komponente *Horoskop* einbauen, ohne einen Abhängigkeitszyklus zu verursachen (s. Abb. 4a).

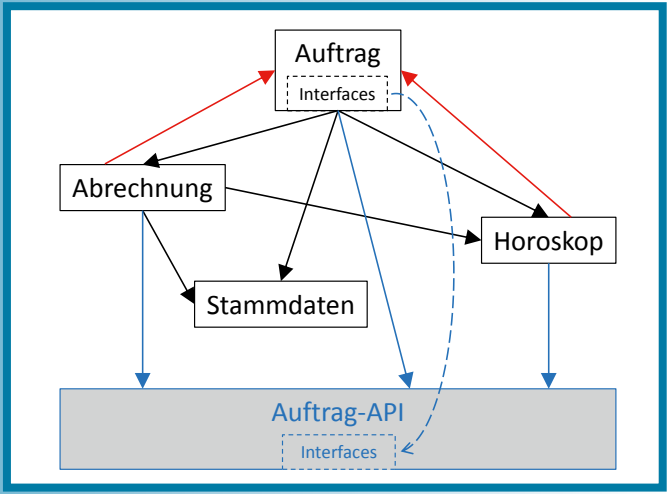

Abb. 4: Unterschiede zwischen OHO 1.2 und 1.3
a) Schwarz: OHO 1.2-Abhängigkeiten: die Komponente *Auftrag* steht jetzt in der Dependency-Hierarchie ganz oben. Rot: diese Abhängigkeiten wären für OHO 1.3 nötig, sind technisch aber nicht möglich
b) Blau und Schwarz: OHO 1.3-Abhängigkeiten, durch die Verschiebung der Auftrag-Interfaces in die neue technische Komponente *Auftrag-API* (grau) werden die rot markierten Abhängigkeiten unnötig

Die Dependency-Hölle

Das Geschäft mit OHO 1.2 läuft immer besser und Tausende Horoskope müssen erstellt und abgerechnet werden. Dank der verbesserten Kommunikation zwischen Starlet und Billy kann die gute Auftragslage einigermaßen bewältigt werden. Allerdings kommt es immer häufiger vor, dass sich die beiden Fachkräfte wünschen, aus einer Abrechnung bzw. aus einem Horoskop nach früheren Aufträgen des gleichen Kunden zu suchen, um daraus Daten zu übernehmen oder mit aktuellen Daten abzugleichen.

Diese neue Anforderung stellt eine fachliche Abhängigkeit der beiden Komponenten *Abrechnung* und *Horoskop* von der Komponente *Auftrag* dar. Jetzt sind die G&G-Entwickler endgültig in der Dependency-Hölle angekommen, weil mit dem aktuellen Dependency-Management diese Abhängigkeiten technisch nicht realisiert werden können. Die G&G-Entwickler erwägen wieder die oben genannten ersten beiden Alternativen. Die hitzige Diskussion darüber schürt das Feuer der Dependency-Hölle.

Ausweg aus dem Abhängigkeitswirrwarr

In dieser heißen Phase kommt ein neues Mitglied ins G&G-Entwicklerteam. Es schlägt vor, die Schnittstellenbeschreibung (API) aus der Komponente *Auftrag* herauszuziehen und von der Implementierung zu trennen. Zu diesem Zweck soll eine neue technische Komponente eingeführt werden, welche nur die Schnittstelle der Komponente *Auftrag* definiert, d. h. nur deren Interfaces beinhaltet. Diese Komponente wird *Auftrag-API* genannt.

Mit dieser Konstruktion brauchen die beiden Komponenten *Abrechnung* und *Horoskop* keine technische Abhängigkeit zur Komponente *Auftrag*, sondern nur noch zu *Auftrag-API* (s. Abb. 4b). Die Abhängigkeiten, die die Dependency-Hölle verursachen würden, sind so nicht nötig. Die G&G-Entwickler implementieren in der Komponente *Auftrag* den Service „Auftragsuche“ und stellen das dazugehörende Interface `IAuftragssuche` in *Auftrag-API* zur Verfügung. Dieses Interface kann aus den Komponenten *Abrechnung* und *Horoskop* referenziert werden. Die Dependency-Injection von Spring übernimmt das Instanzieren des Auftragsucheservice.

Die G&G-Entwickler stellen fest, dass ihre Entwicklungsumgebungen keine Kompilierungsfehler melden, die Anwendung mit Maven gebaut, deployt und erfolgreich angewendet werden kann. Das Feuer der Dependency-Hölle ist mal wieder gelöscht.

Integrationstests im Maven-Build

Die G&G-Entwickler schreiben in der Komponente *Abrechnung* die Testklasse `AuftragsucheTest.java`, in der eine Auftragsuche durchgeführt wird. Dieser Test läuft in der Entwicklungsumgebung erst nach manueller Erweiterung des Klassenpfades, weil die Abhängigkeiten zur Komponente *Auftrag* nicht aufgelöst werden konnten. Später beim Maven-Build stellt sich heraus, dass dieser aus dem gleichen Grunde nicht läuft. Ursache ist, dass die G&G-Entwickler einen Integrationstest geschrieben haben, in dem eine in der Dependency-Hierarchie unten gelegene Komponente (*Abrechnung*) eine andere Komponente benutzen möchte, die in der Dependency-Hierarchie weiter oben liegt (*Auftrag*, s. Abb. 4a).

Beim Auflösen von nach oben gerichteten Abhängigkeiten müsste Maven eine alte, zuvor gebaute Version der Auftrag-Komponente benutzen, um die Integrationstests in der Komponente *Abrechnung* durchführen zu können. Diese Vorgehensweise wäre im Maven-Build falsch, weil hier sichergestellt werden soll, dass nur die aktuellen Ressourcen verwendet werden.

Mit dem Integrationstest `AuftragsucheTest.java` in der Komponente *Abrechnung* haben die G&G-Entwickler eine technische Abhängigkeit implementiert, die im Maven-Build unüberwindlich ist. Die Dependency-Hölle brennt wieder. Als schnelle Lösung, um den Maven-Build wieder zum Laufen zu bekommen, wird der Auftragsucheservice in der Komponente *Abrechnung* für die Testausführung gemockt [MockO]. Dieser Auftragsucheservice-Mock beinhaltet keine Funktionalität und dient nur dazu, die fehlende Abhängigkeit für die Dependency-Injection von Spring aufzulösen. Damit kann zwar der Maven-Build erfolgreich ausgeführt werden, aber der Integrationstest bleibt im Maven-Build unausgeführt.

Durch den Kunstgriff der API-Komponente konnte also die Dependency-Hölle zur Compile-Zeit überwunden werden. Über die in der Dependency-Hierarchie aufwärts gerichteten Integrationstests bekommt aber die Dependency-Hölle wieder eine offene Tür in Form der Maven-Build-Test-Runtime.

Grenzen des klassischen Dependency-Managements

Das Geschäft mit OHO 1.3 läuft so gut, dass Starlet sich entschließt, noch eine Fachkraft aus dem Bereich Sachbearbeitung einzustellen, die sich speziell um die Auftragsbearbeitung und um sonstige administrative Tätigkeiten kümmert. Zu diesem Zweck soll OHO eine neue Funktion bekommen, die es ermöglichen soll, aus den Stammdaten heraus nach Aufträgen und Abrechnungen zu suchen. Die Komponente *Stammdaten* wird damit abhängig von den Komponenten *Auftrag* und *Abrechnung*. Außerdem planen die G&G-Entwickler für die administrativen Tätigkeiten eine völlig neue Komponente *Administration*.

Damit lodert die Dependency-Hölle wieder auf. Die G&G-Entwickler überlegen jetzt, wie eine vollständige Überwindung der Dependency-Hölle aussehen könnte, die sowohl zur Compile-Zeit als auch zur Maven-Build-Test-Runtime alle Dependency-Probleme endgültig löst.

Fachliche Offenheit zur Compile-Zeit

Der Ansatz im Kapitel „Ausweg aus dem Abhängigkeitswirrwarr" ist gut, aber nicht konsequent zu Ende gedacht und zwar aus zwei Gründen:

- Wenn die API-Komponente sämtliche Interfaces der Komponenten *Abrechnung*, *Auftrag* und *Horoskop* beinhalten würde, bräuchte man zur Compile-Zeit keine Abhängigkeiten mehr zwischen diesen fachlichen Komponenten. Die G&G-Entwickler führen ein Refactoring durch und bauen die Version OHO 1.4 (s. Abb. 5a).
- Es gibt immer noch technische Abhängigkeiten zu der Komponente *Stammdaten*. Hier hat das Feuer der Dependency-Hölle noch Brennmaterial. Die vollständige Entkoppelung zur Compile-Zeit erreichen die G&G-Entwickler in der Version 2.0, in der alle fachlichen Komponenten nur noch zur API-Komponente eine technische Abhängigkeit aufweisen (s. Abb. 5b und Abb. 6).

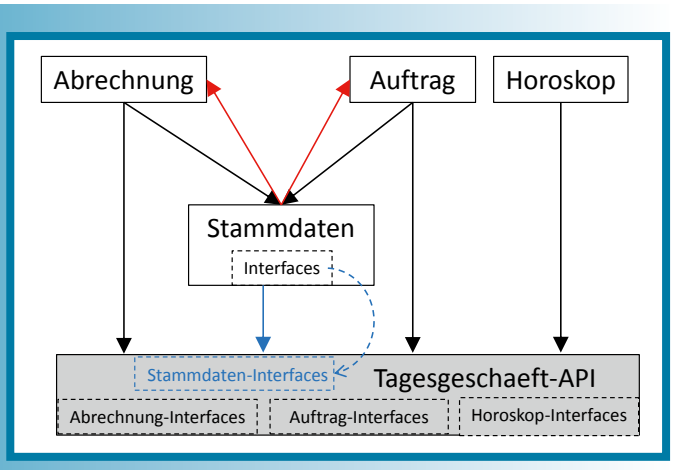

Abb. 5: Unterschiede zwischen OHO 1.4 und 2.0
a) Schwarz: OHO 1.4-Abhängigkeiten: zwischen den Komponenten *Abrechnung*, *Auftrag* und *Horoskop* gibt es keine Abhängigkeiten mehr. Die 1.3-Komponente *Auftrag-API* wurde in *Tagesgeschaeft-API* umbenannt. Rot: diese Abhängigkeiten wären für OHO 1.5 nötig, sind technisch aber nicht möglich
b) Blau: durch die Verschiebung der Stammdaten-Interfaces in die technische API-Komponente (grau), werden die letzten Abhängigkeiten zwischen den fachlichen Komponenten (weiß) unnötig. Statt 1.5 wird diese Version 2.0 genannt

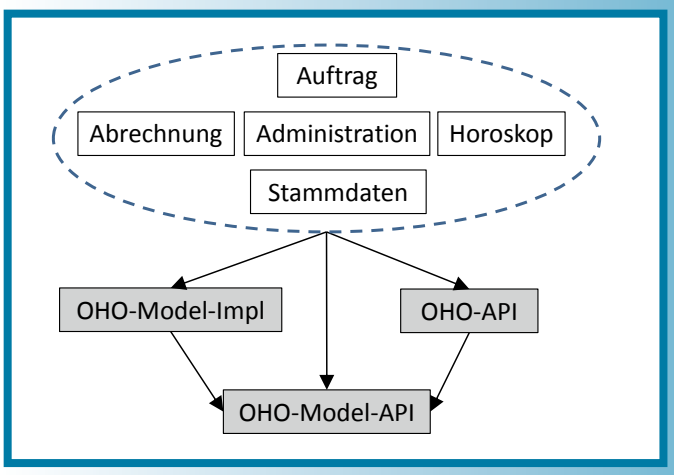

Abb. 6: OHO 2.1-Abhängigkeiten – Aus verschiedenen Gründen wurden mehrere technische Komponenten (grau) nötig. Die fachlichen Komponenten (weiß) haben untereinander keine (technischen) Abhängigkeiten mehr. Jede fachliche Komponente hat Abhängigkeiten zu allen drei technischen Komponenten. Die 1.4-Komponente *Tagesgeschaeft-API* wurde in *OHO-API* umbenannt

Fachliche Offenheit zur Maven-Build-Test-Runtime

Nach wie vor können für einige Integrationstests nicht alle benötigten Abhängigkeiten im Maven-Build aufgelöst werden (z. B. die Auftragssuche in der Komponente *Abrechnung*). Dieses Problem ließe sich lösen, indem G&G-Entwickler die nötigen Abhängigkeiten mit dem Maven-Scope „test" konfigurieren würden. Damit wäre aber nur die Dependency-Hölle von der Compile-Zeit in die Maven-Build-Test-Runtime verschoben.

Endgültig lösen die G&G-Entwickler das Problem, indem sie in der Phase „test" des Maven-Lebenszyklus nur echte Unit-Tests ausführen lassen. Das bedeutet, dass in den Tests des Maven-Builds nur Funktionalität aus der eigenen Komponente ausgeführt wird. Wird eine Funktionalität aus einer anderen Komponente zum Test gebraucht, muss diese Funktionalität gemockt werden [MockO].

Die Integrationstests führen die G&G-Entwickler also jetzt erst aus, nachdem der Maven-Build erfolgreich durchgelaufen ist und alle Komponenten frisch gebaut vorliegen. Dann spielt die Dependency-Hierarchie keine Rolle mehr. Zu diesem Zweck implementieren die G&G-Entwickler die Integrationstests in einem separaten Maven-Projekt, welches die Abhängigkeiten zu allen fünf fachlichen Komponenten auflösen kann [MavenIT].

Ein Continuous Integration (CI)-Server führt regelmäßig das OHO-Maven-Build durch. Ist dieses erfolgreich, führt der CI-Server im direkten Anschluss automatisch das Maven-Goal „test" im Integrationstest-Projekt durch. Damit werden die Integrationstests ganz ohne Dependency-Hölle regelmäßig immer direkt nach dem Maven-Build durchgeführt.

Das fachlich offene Dependency-Management

Mit der fachlichen Offenheit sowohl zur Compile-Zeit als auch zur Maven-Build-Test-Runtime ist es möglich, fachliche und

technische Abhängigkeiten vollständig voneinander zu entkoppeln. Neue fachliche Abhängigkeiten durch neue Anforderungen, die häufig in der Weiterentwicklung von Software auftreten, können so schmerzfrei und schnell implementiert werden.

Die G&G-Entwickler führen ein Refactoring durch und bauen die OHO-Version 2.0 nach dem fachlich offenen Dependency-Management (s. Abb. 5b). Jetzt können alle ausstehenden Änderungen problemlos durchgeführt und auch die neue Komponente *Administration* kann ohne Seiteneffekte eingebaut werden. Alle in *Administration* benötigten Funktionen aus fremden Komponenten stehen dank der API-Komponente nur durch eine Abhängigkeit zu der Komponente *OHO-API* zur Verfügung. Und auch umgekehrt können so alle anderen Komponenten bei Bedarf die Funktionalität der Komponente *Administration* nutzen. Das Dependency-Management ist fachlich völlig offen für alle Arten von Veränderungen (s. Kasten „Begriffserläuterungen", Fachliche Offenheit).

Nachteile des fachlich offenen Dependency-Managements

Als Nachteile sind folgende drei Punkte zu nennen:

- Das Paradigma der fachlich motivierten Komponentenbildung ist durch die Einführung der rein technischen API-Komponente aufgeweicht. Zum einen, weil eine zusätzliche technische API-Komponente nötig ist, zum anderen, weil diese API-Komponente einen Interface-Monolithen darstellt. Die fachliche Trennung gilt also nur für die Implementierung, nicht für die Deklaration der Schnittstellen.
- Die fachlichen Abhängigkeiten sind im Quellcode weniger transparent, weil die fachlichen Abhängigkeiten nicht mehr durch die Analyse der technischen Abhängigkeiten ermittelt werden können. Um die fachlichen Abhängigkeiten aus dem Quellcode zu ermitteln, müssen die Referenzen der einzelnen Interface-Klassen in der API-Komponente gesucht werden oder die Spring-Konfiguration muss analysiert werden. Mit einer modernen IDE ist das jedoch kein großer Aufwand.
- Für die Wiederverwendung einer Komponente reicht es nicht aus, die Komponente selbst in einen neuen Kontext einzufügen, sondern es muss zusätzlich auch ihr API-Teil aus der API-Komponente in den neuen Kontext eingebracht werden. Aber zum einen gehört die Wiederverwendung von Komponenten nicht zum Alltagsgeschäft und wird eher selten durchgeführt, zum anderen dürfte eine gute Package-Struktur in der API-Komponente diesen Zusatzaufwand minimieren.

Vorteile des fachlich offenen Dependency-Management

Durch die Anwendung des fachlich offenen Dependeny-Managements ist es möglich, fachliche und technische Abhängigkeiten vollständig und konsequent voneinander zu trennen. Damit wird dem Open-Closed-Prinzip [CCDOCP] im Dependency-Management Folge geleistet. Neue fachliche Abhängigkeiten können beliebig zwischen den Komponenten gezogen werden, ohne technische Probleme und Umbaumaßnahmen damit nötig zu machen. Das System ist offen für Erweiterungen.

Ein zweiter großer Vorteil besteht darin, dass die Entwickler die Schnittstelle einer Komponente nach außen (das heißt zu anderen Komponenten) bewusst formulieren müssen. Werden fachliche Abhängigkeiten durch technische Abhängigkeiten definiert, liegen alle als `public` deklarierten Klassen einer Komponente A im Klassenpfad der anderen Komponenten, die von Komponente A abhängig sind. Das verleitet Entwickler dazu, Klassen aus fremden Komponenten zu referenzieren, die eigentlich gar nicht zur Schnittstelle dieser Komponente gehören.

Mit einem fachlich offenen Dependency-Management ist das nicht möglich. Hier werden die Entwickler gezwungen, die Funktionalität einer Komponente, die nach außen zu Verfügung gestellt werden soll, bewusst in der API-Komponente in Form einer Interface-Klasse bzw. in einer ihrer Methoden zu definieren. Damit wird zusätzlich dem Dependency-Inversion-Prinzip [CCDDI] Folge geleistet – die Komponenten hängen nicht direkt voneinander ab, sondern nur von ihrer Abstraktion (API).

Notwendigkeit weiterer technischer Komponenten

Häufig kann es sinnvoll sein, die große API-Komponente in zwei oder mehrere technische Komponenten aufzuteilen. Beispielsweise lösen die G&G-Entwickler die Interface-Klassen ihrer Domänen-Objekte (z. B. `IAuftrag.java` aus der Komponente *Auftrag* oder `IHoroskop.java` aus der Komponente *Horoskop*) aus der großen API-Komponente heraus, weil ein Datenimport-Tool auf das OHO-Datenmodell zugreifen muss. Durch Erzeugung der technischen Komponente *OHO-Model-API* ist das gezielt möglich. Außerdem ist es sinnvoll, die Implementierung der Domänen-Objekte aus den fachlichen Komponenten in eine weitere technische Komponente *OHO-Model-Impl* herauszuziehen. Abbildung 6 zeigt das Resultat dieses Umbaus. Mit der Version OHO 2.1 können die Domänen-Objekte aller fachlichen Komponenten überall instanziiert werden. Das hat den Vorteil, dass beim Schreiben von Tests Domänen-Objekte aus fremden Komponenten (z. B. `AuftragImpl.java` in der Komponente *Horoskop*) nicht gemockt werden müssen [MockO], sondern einfach instanziiert werden können. Die Testbarkeit wird dadurch deutlich verbessert. Für die fachliche Offenheit spielt die Zahl der technischen Komponenten keine Rolle. Entscheidend ist nur, dass die fachlichen Komponenten keine technischen Abhängigkeiten untereinander, sondern nur Abhängigkeiten zu den technischen Komponenten haben.

Fazit

Das fachlich offene Dependency-Management steht auf zwei Säulen:

- Fachliche Komponenten haben untereinander keine technischen Abhängigkeiten, sondern sind technisch nur von nicht-fachlichen Komponenten abhängig.
- Im Maven-Build wird auf Integrationstests verzichtet, das heißt, beim Bau einer Komponente werden nur echte Unit-Tests durchgeführt und Integrationstests werden erst ausgeführt, wenn alle Komponenten gebaut wurden.

Die erste Säule entkoppelt die fachlichen und technischen Abhängigkeiten zur Compile-Zeit, die zweite Säule zur Maven-Build-Test-Runtime. Damit können fachliche Abhängigkeiten völlig frei von technischen Einschränkungen implementiert werden. Dieses fachlich offene Dependency-Management überwindet vollständig die Dependency-Hölle des klassischen Dependency-Managements und schenkt den Entwicklern und Architekten ein kleines Stückchen Himmel in ihrem irdischen Alltag.

Links

[CCDDI] http://www.clean-code-developer.de/Gelber-Grad.ashx#Dependency_Inversion_Principle_1
[CCDOCP] http://www.clean-code-developer.de/Gr%C3%BCner-Grad.ashx#Open_Closed_Principle_0
[CCDSoC] http://www.clean-code-developer.de/Oranger-Grad.ashx#Separation_of_Concerns_SoC_2
[KE] http://de.wikipedia.org/wiki/Komponentenbasierte_Entwicklung
[MavenIT] http://docs.codehaus.org/display/MAVENUSER/Maven+and+Integration+Testing
[MockO] http://de.wikipedia.org/wiki/Mock-Objekt
[Modul] http://de.wikipedia.org/wiki/Modul_%28Software%29
[Sinz99] E. J. Sinz, Anwendungssysteme aus fachlichen Komponenten, Wirtschaftsinformatik, Ausgabe 1999-01, http://www.wirtschaftsinformatik.de/index.php;do=show/site=wi/sid=4516462854f98ee4a52c49550952029/alloc=12/id=427

Dr. Reik Oberrath ist als IT-Berater bei der iks Gesellschaft für Informations- und Kommunikationssysteme mbH tätig. Er beschäftigt sich seit vielen Jahren mit der Entwicklung von individuellen Geschäftsanwendungen, speziell unter Swing und RCP. Einen besonderen Fokus legt er auf Automatisierung in der Qualitätssicherung und im Release Build sowie auf Clean Code.
E-Mail: R.Oberrath@iks-gmbh.com