Gut aufgeräumt

Dependency-Management von technischen Standard-komponenten

Reik Oberrath

In jedem Projekt, in dem komponentenorientiert entwickelt wird, tritt immer wieder die gleiche Frage auf: Wohin mit all den allgemeinen Klassen wie Utilities, Exceptions, Interfaces und anderen Klassen, die mehrfach gebraucht werden. Dieser Artikel empfiehlt, einige technische Standardkomponenten zu bilden, die in ein spezielles Dependency-Management eingebunden sind. Außerdem erklärt er, welche Klassen in welche Komponenten gehören. Diese Vorgehensweise ergänzt sehr gut das fachlich offene Dependency-Management [Ober12], welches auf die nicht-technischen Komponenten fokussiert ist, und spiegelt moderne Muster modularer Architektur wider [PMA]. Zur Erläuterung greift der Artikel die Java-Anwendung OHO aus [Ober12] wieder auf. Dabei wird der Einsatz des Projekt-Management-Tools Maven erläutert.

Fallstudie: Online Horoskope

Die kleine Software-Schmiede „Gut & Günstig“ (G&G) erhält von der Astrologin „Starlet“ den Auftrag, eine Horoskop-Webanwendung zu entwickeln. Die Anwendung soll es ihren Kunden ermöglichen, ein Horoskop online sowohl zu beauftragen als auch das Horoskop nach erfolgter Zahlung abzurufen. Diese Anwendung wird *O*nline *Ho*roskop, kurz OHO, genannt. Die G&G-Entwickler diskutieren zurzeit das Buch „Growing Object-Oriented Software, Guided by Tests“ [GOOS]. Auch wenn die G&G-Entwickler (noch?) nicht wirklich Test-getrieben entwickeln, sind ihnen die Aspekte der Test- und Wartbarkeit sehr wichtig.

Zunächst möchten die G&G-Entwickler einen sogenannten „Walking Skeleton“ implementieren, das heißt, das kleinste mögliche Stückchen echter Funktionalität, das als Version 0.0.1 gebaut, installiert und als Komplettsystem getestet werden kann. Dazu wählen sie die Auftragsverwaltung aus. Auf einer Datenmaske wird ein neuer Auftrag nur mit dem Feld „Auftraggeber“ erfasst. Beim Speichern wird automatisch eine Auftrags-ID generiert und in einer Auftragsliste angezeigt. Für diese minimale Funktionalität setzen die G&G-Entwickler eine Datenbank mit einer Auftragstabelle auf, welche zwei Datenfelder beinhaltet: Auftraggebername und ID. Sie implementieren die Datenmaske, einen Auftragsservice und ein Auftrags-DAO-Objekt. Zu jeder wichtigen Klasse wird eine Testklasse mit JUnit-Tests implementiert.

Die Komponente „Builder“

Den Produktionscode der Anwendung packen die G&G-Entwickler in ein Artefakt „auftrag.jar“, das wiederum für ein Deployment auf einem Tomcat in ein Web Application Archive (WAR) gepackt wird. Für den Bau des WARs erzeugen die G&G-Entwickler die technische Komponente *Builder*. Als Build-Tool verwenden sie Maven. Der Build eines Release-Kandidaten besteht aus zwei separaten Schritten:

- Bau des Artefakts auftrag.jar und
- Bau des Webarchivs (WAR-Datei).

Nach einigen Tagen Arbeit läuft der Walking Skeleton und der manuelle Systemtest der OHO-Anwendung ist nach dem Deployment auf den Tomcat erfolgreich. Nach dieser ersten Iteration der OHO-Entwicklung besteht die Anwendung aus zwei Komponenten, der fachlichen *Auftrag* und der technischen *Builder*.

Die Komponente „Parent“

Bevor die G&G-Entwickler die Auftragsverwaltung weiterimplementieren, wollen sie erst die übrigen Komponenten aufsetzen. Als Nächstes soll die Horoskopverwaltung erzeugt werden. Allerdings fehlt noch ein Hauptmenü, in dem die Auftrags- und Horoskopverwaltung eingehängt werden können. Aus diesem Grund realisieren die G&G-Entwickler in der zweiten Iteration zunächst die fachliche Komponente *Main*. In dieser Komponente werden die übrigen fachlichen Komponenten eingebettet. Als minimale Implementierung beinhaltet *Main* ein Hauptmenü mit dem einzigen Eintrag „Aufträge“ für die Auftragsverwaltung. Anschließend wird die Horoskopverwaltung mit Minimalimplementierung aufgesetzt und mit dem Menüpunkt „Horoskope“ in das Hauptmenü integriert.

Beim Schreiben der ersten Unit-Tests für *Main* und *Horoskop* stellen die G&G-Entwickler fest, dass diesen neuen Komponenten die Abhängigkeit zur JUnit-Bibliothek fehlt. Der Komponente *Auftrag* wurde diese Abhängigkeit manuell zugefügt. Gingen die Entwickler für *Main* und *Horoskop* genauso vor, würden sie das DRY-Prinzip [DRY] verletzen. Aus diesem

MU-Klassen

MU steht für *M*ultiple *U*sage. MU-Klassen werden also für mehrfache Benutzung entworfen. MU-Charakter haben zwar die meisten Klassen, aber bei folgenden Klassentypen (s. auch Tabelle 1) ist dieser besonders ausgeprägt:

Utility-Klassen im weitesten Sinn
Diese Klassen stellen Funktionalität an zentralen Stellen bereit, die werkzeugartig für verschiedene Zwecke von unterschiedlichen Codeblöcken aus genutzt werden. Dazu zählen:

- Utilities im engeren Sinn: nicht instanziierte Klassen mit statischen Methoden
- Helferklassen, die instanziiert genutzt werden
- Stammklassen (häufig abstrakt), die durch Vererbung Funktionalität zur Verfügung stellen

Informationstransferklassen
Klassen für den Informationsaustausch zwischen unterschiedlichen Teilen der Software (z. B. zwischen Komponenten). Dazu zählen:

- Datenklassen, deren Inhalte nicht wie Domain-Objekte persistiert werden und die typischerweise als Typen in Signaturen von Interface-Methoden verwendet werden
- Exception-Klassen, um Ausnahmebedingungen anzuzeigen

Grund führen sie die neue technische Komponente *Parent* ein. Das *Parent*-Maven-Projekt wird mit dem packaging-Typ „pom“ erzeugt. In den pom-Dateien der drei fachlichen Komponenten *Auftrag*, *Horoskop* und *Main* wird *Parent* als „parent“ konfiguriert. Anschließend wird die Definition der JUnit-Abhängigkeit von der pom-Datei der Auftragskomponente in die *Parent*-pom-Datei verschoben (und der „scope“ von „test“ auf „provided“ gesetzt). Jetzt erben die drei fachlichen Komponenten diese Abhängigkeit und können die JUnit-Sourcen benutzen, ohne das DRY-Prinzip zu verletzen.

Mit der Komponente *Parent* existiert jetzt eine zentrale Stelle zur Definition aller allgemeinen Abhängigkeiten. Das nutzen die G&G-Entwickler, um für OHO die Logging-Funktionalität einzurichten. Sie definieren in der *Parent*-pom-Datei eine Abhängigkeit zur log4j-Bibliothek und können damit in allen Komponenten die log4j-Logger benutzen. Außerdem definieren sie in der *Parent*-pom-Datei die Komponenten *Main*, *Auftrag*, *Horoskop* und *Builder* als Module. Damit sind sie in der Lage, den Build eines Release-Kandidaten in nur einem Schritt („mvn install“ auf dem Parent-Projekt) durchzuführen. Nach dieser zweiten Iteration besteht die OHO-Anwendung aus fünf Komponenten (s. Abb. 1).

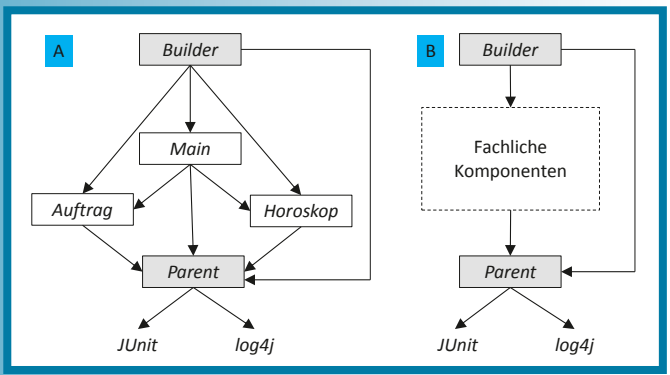

Abb. 1: Dependency-Management der ersten fünf OHO-Komponenten, technische Komponenten sind grau dargestellt. Aus Gründen der Übersichtlichkeit sind in B und den folgenden Abbildungen die Abhängigkeiten zwischen den fachlichen Komponenten (weiß) ausgeblendet. Oben dargestellte Komponenten hängen immer von weiter unten stehenden ab (s. „Dependency-Hierarchie“ in [Ober12] und „Levelize Modules“ in [PMA])

Die Komponenten „Common“, „Test“ und „Parent.BLoC“

Die G&G-Entwickler betrachten den bisher erreichten Stand und stellen fest, dass sowohl im Produktions- als auch im Testcode das DRY-Prinzip verletzt wurde: Jede fachliche Komponente enthält eine Reihe von Codeblöcken, die ursprünglich für die Komponente *Auftrag* geschrieben und dann fast genauso in die Komponente *Horoskop* übernommen wurden. Grund dafür ist die „parallele Funktionalität“ beim Zugriff auf die Datenbank durch ein DAO-Objekt, bei der Darstellung der Daten in der GUI oder beim Validieren von Testergebnissen im Testcode. Deshalb wollen die G&G-Entwickler mit einem Refactoring die strukturelle Qualität und damit die Wartbarkeit der gesamten Anwendung verbessern, bevor sie die nächste fachliche Komponente *Rechnung* aufsetzen.

In der dritten Iteration erzeugen die G&G-Entwickler zwei neue technische Komponenten. Die Komponente *Common* beinhaltet MU-Klassen (s. Kasten „MU-Klassen“) wie `AbstractDao.java` und `GuiValidationHelper.java`, die Komponente *Test* die Klasse `AbstractOhoUnitTest.java`. Beide Komponenten bekommen *Parent* als Maven-Parent, damit sie die dort konfigurierten Abhängigkeiten erben. Für den automatischen Bau werden die neuen Komponenten in der *Parent*-pom-Datei als Module konfiguriert. Zuletzt werden sie in der *Parent*-pom-Datei als neue Abhängigkeit hinzugefügt, damit alle fachlichen Komponenten darauf zugreifen können.

Diese Vorgehensweise verursacht aber ein Problem. Maven kann die Anwendung jetzt nicht mehr bauen. Die Komponenten *Common* und *Test* waren von der *Parent*-Komponente abhängig (wegen der Parent-Konfiguration), und die *Parent*-Komponente ist jetzt wegen der Maven-Dependencies von diesen beiden Komponenten abhängig (s. Abb. 2). Der Maven-Reaktor weiß also nicht, welche Komponente (welches Projekt) er zuerst bauen soll, und liefert deshalb die Fehlermeldung *„The projects in the reactor contain a cyclic reference“*.

Die G&G-Entwickler finden folgende Lösung: Sie führen eine zweite Maven-Parent-Komponente ein, welche nur für die fachlichen Komponenten zuständig ist. Diese Komponente nennen sie *Parent.BLoC* (*B*usiness *Lo*gic Components). Alle drei fachlichen Komponenten *Auftrag*, *Horoskop* und *Main* bekommen *Parent.BLoC* als Maven-Parent. *Parent.BLoC* werden die technischen Komponenten *Common* und *Test* als Abhängigkeiten zugefügt. Damit erben die fachlichen Komponenten alle Abhängigkeiten von dieser zentralen Stelle und verletzen so nicht das DRY-Prinzip. *Parent.BLoC* bekommt wiederum *Parent* als Maven-Parent. Damit erben die fachlichen Komponenten auch die Abhängigkeiten zu den allgemeinen Third-Party-Bibliotheken JUnit und log4j (s. Abb. 2).

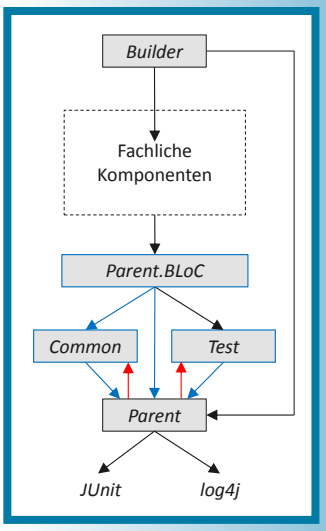

Abb. 2: Dependency-Management mit den neuen technischen Komponenten *Common* und *Test*. Blau: Unterschiede zu Abb. 1b. Wegen der nicht möglichen, rot dargestellten Abhängigkeiten wurde die Komponente *Parent.BLoC* erzeugt

Die Beziehung zwischen „Common“ und „Test“

In der vierten Iteration implementieren die G&G-Entwickler nun die fachlichen Komponenten *Rechnung* und *Stammdaten* mit minimaler Funktionalität. Sie stellen fest, dass die neuen fachlichen Komponenten sehr einfach aufgesetzt werden können. Lediglich durch Definition der Komponente *Parent.BLoC* als Parent stehen alle benötigten Abhängigkeiten zur Verfügung. Außerdem kann durch die Verwendung der bereits implementierten Funktionalität in den technischen Komponenten *Common* und *Test* sowohl der Produktions- als auch der Testcode der neuen Komponenten schnell und sauber (ohne Verletzung des DRY-Prinzips) implementiert werden. Die G&G-Entwickler sind deshalb mit ihrem Refactoring aus der dritten Iteration sehr zufrieden.

Allerdings diskutieren sie die Frage, ob die Komponente *Common* von *Test* abhängig sein sollte oder umgekehrt (beides gleichzeitig ist wegen einer dann zyklischen Referenz nicht möglich):

- *Common* abhängig von *Test:* Die allgemeine OHO-Test-Funktionalität steht im Testcode von *Common* zur Verfügung. Diese Wahl würde den Testcode von *Common* optimieren.
- *Test* abhängig von *Common:* Die *Common*-Funktionalität steht im Code von *Test* zur Verfügung. Diese Wahl würde den Code von *Test* und damit den Testcode aller fachlichen Komponenten optimieren.

Die G&G-Entwickler entscheiden sich für die zweite Alternative, weil sich fachliche Komponenten wegen Kundenwünschen häufiger ändern als technische (siehe unten). Um beim Aufsetzen neuer und beim Ändern existierender fachlicher Komponenten möglichst flexibel zu sein, soll der Testcode der fachlichen Komponenten optimiert werden.

Das Problem der Wiederverwendbarkeit

In den weiteren Iterationen implementieren die G&G-Entwickler die Details der fachlichen Funktionalität in allen fünf fachlichen Komponenten. Dabei gehen sie nach folgender Regel vor: Wird ein Block von Programmzeilen an mehreren Stellen benötigt, wird er in einer MU-Klasse zur Verfügung gestellt – für den Produktionscode in der Komponente *Common*, für den Testcode in der Komponente *Test*.

Dabei müssen die G&G-Entwickler leider feststellen, dass viele Codeblöcke nicht zentral zur Verfügung gestellt werden können, weil diese Codezeilen fachspezifische Java-Typen (z. B. `Auftrag.java`) beinhalten, die in den fachlichen Komponenten definiert sind und deshalb in den technischen Komponenten *Common* und *Test* nicht referenziert werden können. Die Entwickler diskutieren dieses Problem, haben aber noch keine saubere Lösung dafür. Daher markieren sie diese Codestellen mit Kommentaren, die das Problem schildern, und tolerieren bis auf Weiteres diesen Schmutz im Code.

Die Komponente „API“

Schließlich ist alle benötigte Fachlichkeit implementiert und die OHO-Version 1.0.0 wird produktiv. Durch den produktiven Einsatz ergeben sich Kundenwünsche, die Änderungen und damit weitere OHO-Versionen nach sich ziehen (s. [Ober12]). Dabei stellen die G&G-Entwickler zahlreiche Probleme fest, die das Dependency-Management zwischen den fachlichen Komponenten betreffen.

Diese Probleme und deren Lösung wurden in [Ober12] ausführlich erläutert. Die Lösung besteht in der Einführung einer neuen technischen Komponente *API* (Application Interface). Diese beinhaltet sämtliche Interface-Typen für alle konzeptionellen Schnittstellen zwischen den Komponenten (z. B. `IAuftrag.java`, `IHoroskop.java`, `IAuftragDao.java`, `IHoroskopDao.java`, `IAuftragService.java`, `IHoroskopService.java`). Die G&G-Entwickler bauen *API* in die Version OHO 2.0 ein. Dabei stellen sie einen großen Vorteil fest: Die als Code-Duplikate markierten Code-Blöcke (siehe oben Kapitel „Das Problem der Wiederverwendbarkeit“), welche fachliche Typen beinhalten und deshalb vorher nicht in *Common* oder *Test* ausgelagert werden konnten, können jetzt aus den fachlichen Komponenten herausgezogen werden, wenn an diesen Stellen die Interface-Typen verwendet werden. Dieses gilt für den Produktions- und den Testcode. Aus diesem Grund bekommen die Komponenten *Common* und *Test* eine Abhängigkeit zu *API*. Jetzt werden alle Code-Duplikate entfernt und der gemeinschaftlich genutzte Code wird sauber an einer einzigen, zentralen Stelle implementiert. Abbildung 3 zeigt das neue Beziehungsgefüge.

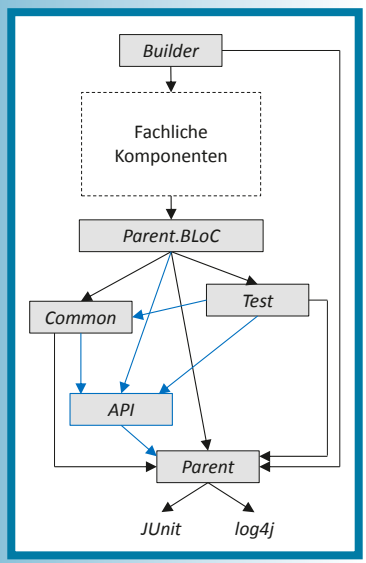

Abb. 3: Dependency-Management mit der neuen technischen Komponenten *API*. Blau: Unterschiede zu Abb. 2

Die Komponente „Global“

Beim Einbau von *API* stoßen die G&G-Entwickler auf ein weiteres Problem, das ebenfalls die Wiederverwendbarkeit betrifft. Die bisherige Sammlung von MU-Klassen in der *Common*-Komponente wurde zum Teil für OHO neu entwickelt. Ein anderer Teil stammt aber aus anderen Anwendungen, in denen diese Funktionalität ursprünglich entwickelt wurde. Diesen Code wiederzuverwenden, war nicht immer leicht, denn der ursprüngliche Code hatte einige Abhängigkeiten zu Klassen, die spezifisch für diese anderen Anwendungen waren. Das machte manchmal ein Refactoring nötig, damit der wiederverwertbare Code-Block herausgelöst und problemlos in den OHO-Code integriert werden konnte.

Mit der Einführung der *API*-Komponente bekommen die G&G-Entwickler das gleiche Problem in der OHO-Anwendung: Wenn später *Common*-Funktionalität für andere Anwendungen wiederverwendet werden soll, darf sie keine Abhängigkeiten zu den fachspezifischen Interface-Typen der *API*-Komponente haben.

Um die Effizienz ihrer Softwareentwicklung zu steigern, beschließen die G&G-Entwickler eine technische Komponente mit Namen *Global* zu erzeugen (s. Abb. 4). Diese Komponente soll diejenigen MU-Klassen beinhalten, die fachunspezifisch sind und in verschiedenen Anwendungen wiederverwendet werden können. Um das deutlich zu machen, enthält der Namespace der globalen Klassen keine Anwendungsbezeichnung und lautet nur „de.gg.global“. Dagegen heißt der Namespace in der Komponente *Common* „de.gg.oho.common“. Das zeigt an, dass sich hier Klassen befinden, die für die OHO-Anwendung spezifisch, aber für alle ihre fachlichen Komponenten allgemein sind. Tabelle 1 nennt einige Beispiele für globale Klassen und solche, die nur für OHO-Code allgemein sind.

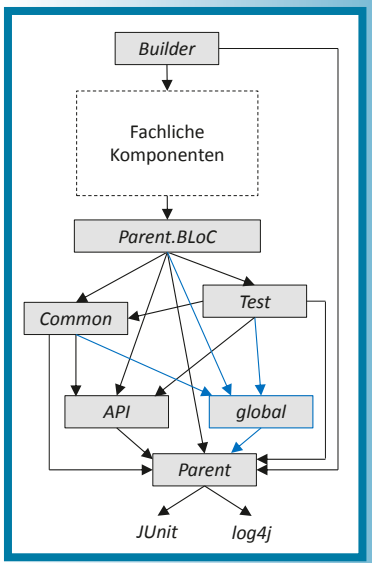

Abb. 4: Dependency-Management mit der neuen technischen Komponente *Global*. Blau: Unterschiede zu Abb. 3

Klassentyp	Common	Global	Test
statische Utilities	`AuftragsUtil.java`	`GGValidationUtils.java`	`GlobalTestUtils.java,OHOTestUtils.java`
nicht statische Helferklasse	`AuftragSorter.java`	`GGTextFieldValidator.java`	
Stammklasse	`OHOTextfield.java`	`GGTextField.java`	`OhoUnitTest.java`
nicht persistierte Datenklasse	`RawAuftrag.java`	`GGValidationResult.java`	
Exception-Klasse	`OhoDaoException.java`	`GGValidationException.java`	

Tabelle 1: Beispiele für MU-Klassen

Globaler Testcode

Die G&G-Entwickler überlegen, die Komponente *Test* aufzuteilen in *Test.Global* und *Test.Common*. Für die Wiederverwendung von Testcode wäre das sinnvoll. Doch sind sie „vorsichtig vor dieser Optimierung“ [VvO] und beherzigen das Clean-Code-Prinzip *„Keep It Simple, Stupid“* [KISS]. Sie entscheiden sich stattdessen, in der Komponente *Test* die Klasse `GlobalTestUtils.java` zu erzeugen und dort den wiederverwendbaren Testcode zu pflegen. Für den aktuellen Umfang des Testcodes reicht das derzeit vollständig aus. Die Entscheidung, für globalen Testcode eine eigene Komponente anzulegen, wird solange verschoben, bis die jetzige Entscheidung Probleme macht.

Wohin mit welcher Klasse?

Für die weitere Entwicklung erstellen die G&G-Entwickler ein Schema, nach dem sie entscheiden, in welche Komponente eine neue MU-Klasse gehört. Dabei wird ihnen bewusst, dass *API* nicht nur Interface-Code beinhaltet, sondern auch Implementierung und zwar in Form von MU-Klassen in Methoden-Signaturen (Exceptions und nicht persistierte Datenklassen). Dieser Teil des Produktionscodes ist ungetestet, denn für die Komponente *API* sind keine Unit-Tests vorgesehen. Dieser Code kann aus *API* nicht herausgelöst werden, weil er von den Interfaces genutzt wird und deshalb direkt zur Schnittstelle (*API*) der Anwendung gehört.

Entscheidungsbaum für die Komponentenfindung neuer MU-Klassen

A: Ist die Klasse spezifisch für eine spezielle fachliche Komponente?
Ja: Klasse gehört in diese Komponente.
Nein: → B

B: Handelt es sich bei dieser Klasse um Testcode?
Ja: Klasse gehört in die Komponente *Test*.
Nein: → C

C: Wird die Klasse in einem *API*-Interface benutzt?
Ja: Klasse gehört in die Komponente *API*.
Nein: → D

D: Enthält die Klasse OHO-spezifische Java-Typen?
Ja: Klasse gehört in die Komponente *Common*.
Nein: Klasse gehört in die Komponente *Global*.

Die Entwickler einigen sich darauf, diese MU-Klassen nur mit einfachen Getter- und Setter-Methoden auszustatten, und erlauben sich ganz pragmatisch, für diese einfachen Methoden auf Unit-Tests zu verzichten. Schließlich einigen sie sich auf die im Kasten „Entscheidungsbaum für die Komponentenfindung neuer MU-Klassen“ dargestellte Vorgehensweise.

Das Problem der „API“-Komponente

Im Laufe der weiteren Entwicklung der OHO-Anwendung stellen die G&G-Entwickler fest, dass ihr Vorsatz, in den MU-Klassen der API-Komponente nur Getter- und Setter-Methoden zu implementieren, im Projektalltag verloren ging. Einige nicht persistierte Datenklassen (s. Kasten „MU-Klassen“) beinhalten mittlerweile mehr oder weniger komplexe Code-Blöcke, die jetzt ungetestet sind und deshalb ein stark erhöhtes Potenzial haben, einen (schwerwiegenden) Fehler in der Produktion zu verursachen.

Durch ein Refactoring wird dieses Problem aus der Welt geräumt: Für alle nicht persistierten Datenklassen wird grundsätzlich ein Interface in der API-Komponente angelegt, das in Methoden-Signaturen anderer Interfaces genutzt wird. Die Implementierung dieser Datenklassen und der dazugehörende Testcode kommen in die Komponente *Common*. Außerdem wählen die G&G-Entwickler einen Code-Watch, der dafür verantwortlich ist, den Code zu überprüfen und dafür zu sorgen, dass dies nicht wieder vorkommt.

Während dieser Entscheidung trifft von ihrer Auftraggeberin „Starlet“ eine Anfrage ein: Die Anwender würden sich immer häufiger über fehlende oder falsche Validierungsfehler beschweren, welche die Dateneingabe erschweren. Deshalb soll ermittelt werden, wie die Validierung optimiert werden kann und wie viel Aufwand das bedeuten würde. Die G&G-Entwickler können die Beschwerden nachvollziehen, denn auch aus Entwicklersicht sind sie mit dem Validierungs-Framework von OHO nicht zufrieden.

Als Verbesserung diskutieren sie folgende Idee: An die Interface-Getter-Methoden der Domain-Objekte (`IAuftrag.java`, `IHoroskop.java` ...) werden Validierungsregeln annotiert. Zu jeder Regel gibt es eine Klasse, die diese Regel implementiert. Ein zentraler OHO-Validator analysiert die Instanzen der Domain-Objekte, durchsucht ihre Annotationen nach Validierungsregeln, ruft diese für die vorliegenden Instanzen auf, sammelt die Fehlermeldungen und liefert die Fehlerliste als Resultat. Auf diese Weise könnte sowohl in der GUI als auch im Backend die Validierung durchgeführt werden. Diese Vorgehensweise würde es nötig machen, die Validierungsregeln in *API* zu implementieren. Dadurch könnte die *API*-Komponente, die ursprünglich testfrei konzipiert wurde, auf Testcode nicht mehr verzichten. Damit wäre aber auch der Code-Watch überflüssig.

Nach zähen Verhandlungen und Zugeständnissen aller Stakeholder können die G&G-Entwickler die neue Validierung umsetzen. Damit ist unvermeidbar, dass die API-Komponente zu einer Mischung aus Interface-, Implementierungs- und Testcode wird. Interfaces und Implementierung gehören technisch so eng zusammen, dass die fachliche Implementierung nicht davon getrennt werden kann.

Fazit

Der Kasten „Entscheidungsbaum für die Komponentenfindung neuer MU-Klassen“ erklärt, welche Klasse in welche technische Komponente gehört. Abbildung 4 zeigt die Abhängigkeiten zwischen diesen Komponenten. Software, die nach dieser Vorgehensweise entworfen wurde, besitzt ein robustes und gleichzeitig flexibles Dependency-Management. Es ist robust, weil es für jedes Stück Implementierung einen Platz vorsieht, und flexibel, weil dieser Platz bei Bedarf wechseln kann, ohne eine Dependency-Hölle zu verursachen. Allerdings neigt die API-Komponente dazu, Implementierungscode anzureichern und zu einer monolithischen Struktur zu werden. Wer das zu verhindern weiß, oder es wenigstens einschränkt, dem bieten die technischen Komponenten *Global*, *Common*, *Test*, *API*, *Parent* und *Parent.BLoC* in dem Abhängigkeitsgefüge von Abbildung 4 einen stabilen technischen Rahmen, um flexibel auf Änderungen zu reagieren und die Wiederverwendbarkeit von Code zu verbessern.

Literatur und Links

[DRY] http://www.clean-code-developer.de/Don-t-Repeat-Yourself-DRY.ashx?HL=dry

[GOOS] St. Freeman, N. Pryce, Growing Object-Oriented Software, Guided by Tests, Pearson Education, Inc, 2010

[KE] http://de.wikipedia.org/wiki/Komponentenbasierte_Entwicklung

[KISS] http://www.clean-code-developer.de/Keep-it-simple-stupid-KISS.ashx?From=KISS

[Ober12] R. Oberrath, Das fachlich offene Dependency-Management, in: JavaSPEKTRUM, 6/2012

[PMA] http://refcardz.dzone.com/refcardz/patterns-modular-architecture

[VvO] http://www.clean-code-developer.de/Vorsicht-vor-Optimierungen.ashx?HL=vorsicht

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