Abhängigkeiten managen mit Degraph

Jens Schauder
T-Systems on site services GmbH
Wolfsburg

Schlüsselworte
Java, Packages, Abhängigkeiten,Visualisierung, Module, Testen, Degraph

Abstract
Wer mit offenen Augen Software entwickelt (und alles andere erschwert die Sache ungemein) merkt schnell, dass Abhängigkeiten zwischen verschiedenen Teilen einer Anwendung ein schier unerschöpflicher Quell von Problemen sind.

Die Antwort auf diese Probleme lautet, Abhängigkeiten von Anfang an unter Kontrolle zu behalten. Zwischen Artefakten (d.h. z.B. Jar-Files) erledigt dies in einem gewissen Umfang das Build Tool der Wahl. Aber wie sieht es mit Abhängigkeiten zwischen Packages aus? Hier geht oft alles wie Kraut und Rüben durcheinander. Viele Tools bieten die Möglichkeit Warnungen zu produzieren, wenn z.B. Zyklische Abhängigkeiten entstehen. Aber wenn es darum geht solche Zyklen aufzulösen, in einem Projekt welches Dutzende davon enthält, ist die Unterstützung minimal.

In diesem Vortrag wird demonstriert, wie Abhängigkeiten aussehen sollten um die langfristige Wartbarkeit von Anwendungen zu unterstützen. Es wird auch gezeigt, wie sie oft in der Realität aussehen. Dafür werden Antipattern identifiziert, und beschrieben, wie man von letzterem zu Ersterem kommt. Dabei wird für die Visualisierung das Open Source Werkzeug Degraph (https://github.com/schauder/degraph) genutzt, das auf ganz besondere Weise die Optionen zum aufbrechen von ungewünschten Abhängigkeiten aufzeigt.

Abhängigkeiten, Packages und Wartbarkeit

Software verbringt meist sehr viel länger in der Wartung und Weiterentwicklung als in der initialen Entwicklungsphase vor dem ersten produktiven Deployment. Damit wird die Wartbarkeit und Weiterentwickelbarkeit zu einer der wesentlichen Qualitätsmerkmalen von Software. Oft in wesentlich höherem Maße als Fehlerfreiheit.

Ein Faktor der diese Qualtitätseigenschaften beschränkt, ist sind Abhängigkeiten. Wenn ich alles von allem in meiner Anwendung abhängt, kann ich keine Teile an ein anderes Team geben, ich kann nur schwer Teile in Isolation testen. Änderungen an einem Teil verursachen Bugs in anderen Teilen. Es können keine Teile überarbeitet oder gar neu implementiert werden ohne unkalkulierbare Risiken einzugehen.

Praktisch jedes Projekt managed mehr oder weniger explizit Abhängigkeiten auf Artefakt Ebene, d.h. WAR, EAR oder JAR Dateien. Zumindest sind diese meist frei von zyklischen Abhängigkeiten.

Auch bei der ganz feinen Granularität wird auf die Struktur geachtet:

- Zeilen werden zu Methoden zusammengefasst. Typischerweise in der Größenordnung von 10 Zeilen

- Methoden werden in Klassen gruppiert. Typischerweise in der Größenordnung von 10 Methoden pro Klasse.

Aber Klassen werden dann zu Hunderten oder Tausenden in ein Jar gepackt. Hier fehlt eine Strukturierungsebene. Die Mittel dafür, die Java anbietet sind sehr schwach: Packages. Packages sind nicht viel mehr als ein Namensraum. Setzt man sie aber bewusst ein, so können sie zur Basis für zukünftige Jar Files werden, für Module, die von getrennten Teams entwickelt werden, wenn das Projekt wächst. Teile die separat deployed werden können, wenn ein anderes Projekt diese Funktion benötigt.

Dies geht aber nicht, wenn Packages wie Pech und Schwefel zusammenhängen. **Daher müssen Abhängigkeiten zwischen Packages zyklenfrei sein**.

Wie sollten Packages strukturiert sein?

Darüber hinaus sollten Packages eine klare Struktur haben. Für recht umfangreiche Projekte hat sich das folgende Muster bewährt:

```
<prefix>.<deployable>.<Fachliches Modul>.<Schicht>
```

Der Prefix ist Firmen- und/oder Projekt-spezifisch, z.B. `de.schauderhaft.degraph.*`

`<Deployable>` kennzeichnet das Jar File welches aus diesem Package und den Subpackages entsteht. In einer Rich Client Applikation könnte dies zum Beispiel die Werte `client`, `server` und `common` annehmen. Dieser Teil entfällt, wenn es nur ein Artefakt gibt.

Die fachlichen Module sind die vielleicht wichtigste Trennung von Programmteilen. Sie strukturiert die Anwendung nach fachlichen Aspekte und macht diese offensichtlich in der Architektur. Beim Öffnen des Projektes kann man nun direkt sehen, worum es in der Anwendung geht.

Ist die Anzahl der Klassen in den fachlich Klassen groß, so können diese noch weiter nach Schichten der Anwendung unterteilt werden: `ui, domain, persistence`.

Sind all diese "Slicings", d.h. Arten und Weisen, wie die Anwendung unterteilt werden kann zyklenfrei, erhält man eine Package Struktur, die es extrem flexibel erlaubt über Applikationsbestandteile nach zu denken und diese unabhängig voneinander weiter zu entwickeln.

Antipatterns

Big Ball of Mud

Abbildung 1 Ball of Mud Antipattern

Ein Big Ball of Mud liegt vor, wenn es keine erkennbare Struktur in den Packages gibt. Abhängigkeiten gehen wild durcheinander, selbst die Package Namen sind nicht klar strukturiert.

Big Ball of Mud ist in gewissem Maße ein Anti-Anti-Pattern, da es zu unspezifisch ist um sinnvoll Maßnahmen zu empfhelen. Erfahrungsgemäß setzt sich ein Big Ball of Mud aus diversen Antipattern zusammen, so dass er Stück für Stück aufgeräumt werden kann, wenn man die kleineren Antipatterns identifiziert.

Unechter Zyklus

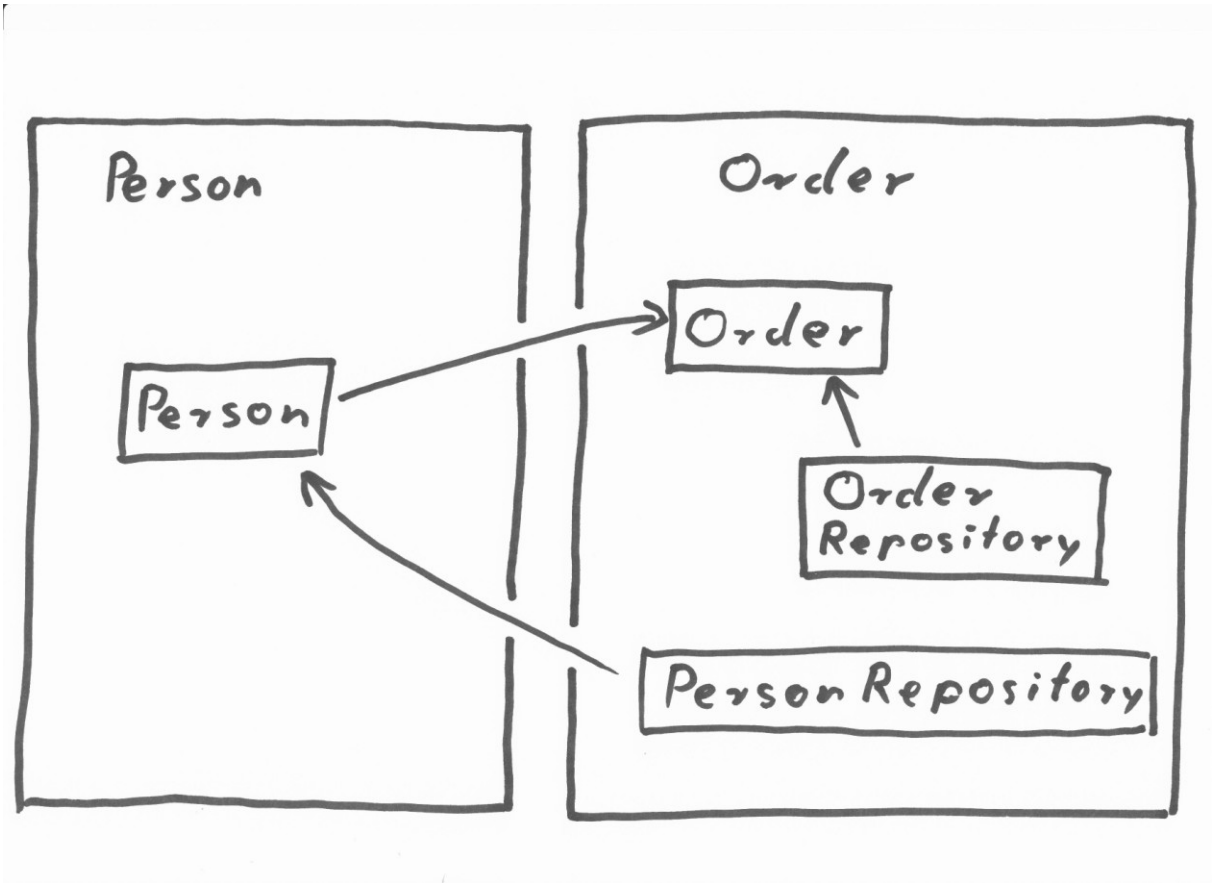

Abbildung 2 Unechter Zyklus Antipattern

Als unechten Zyklus bezeichne ich Zyklen zwischen Packages, die dadurch entstehen, das eine Klasse in einem falschen Package gelandet sind. Durch Korrektur dieses Lapsus lässt sich der Zyklus einfach bereinigen.

Echter Zyklus

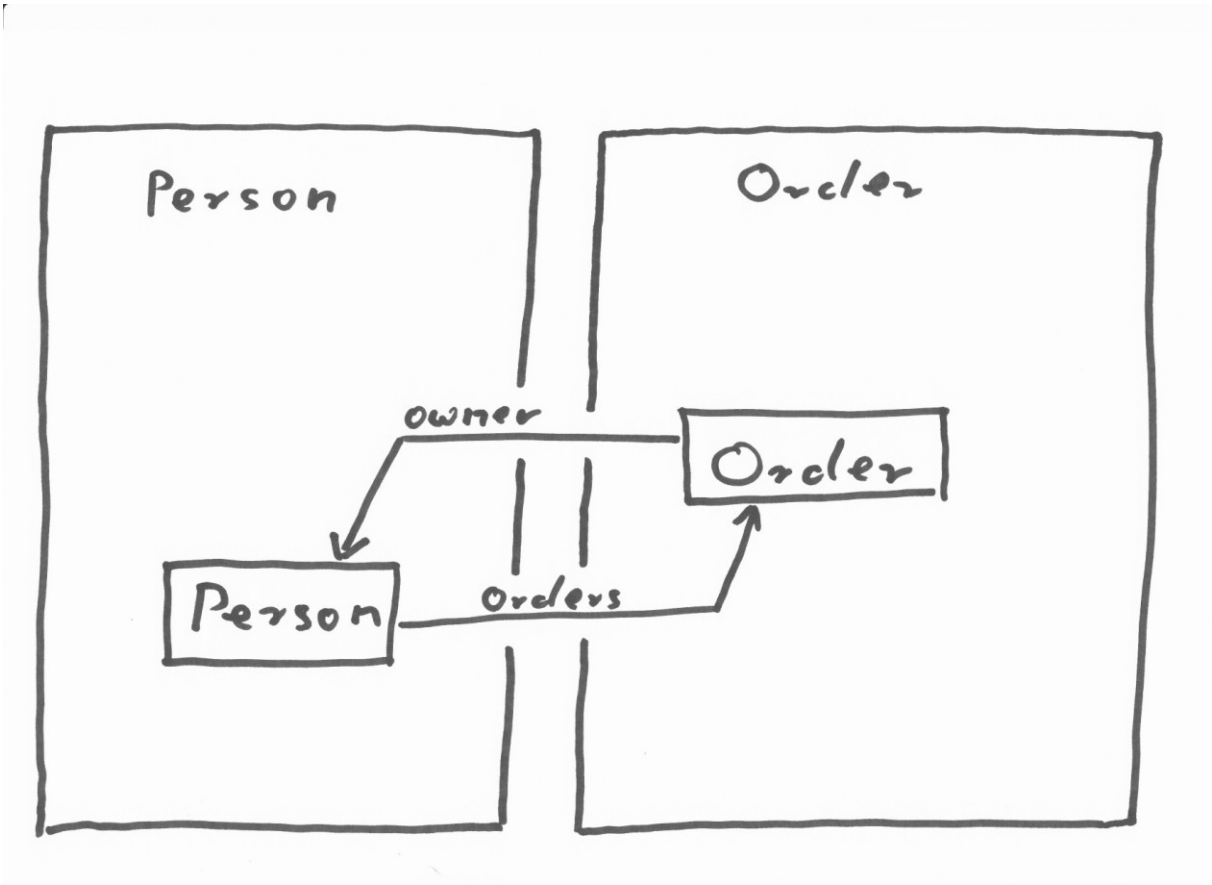

Abbildung 3 Echter Zyklus Antipattern

Als echten Zyklus bezeichne ich zyklische Abhängigkeiten, die nicht durch einfaches verschieben von Klassen bereinigt werden kann, da dann Klassen in den falschen Packages landen würden. Derartige

Zyklen können aufgelöst werden, in dem man Interfaces einführt.

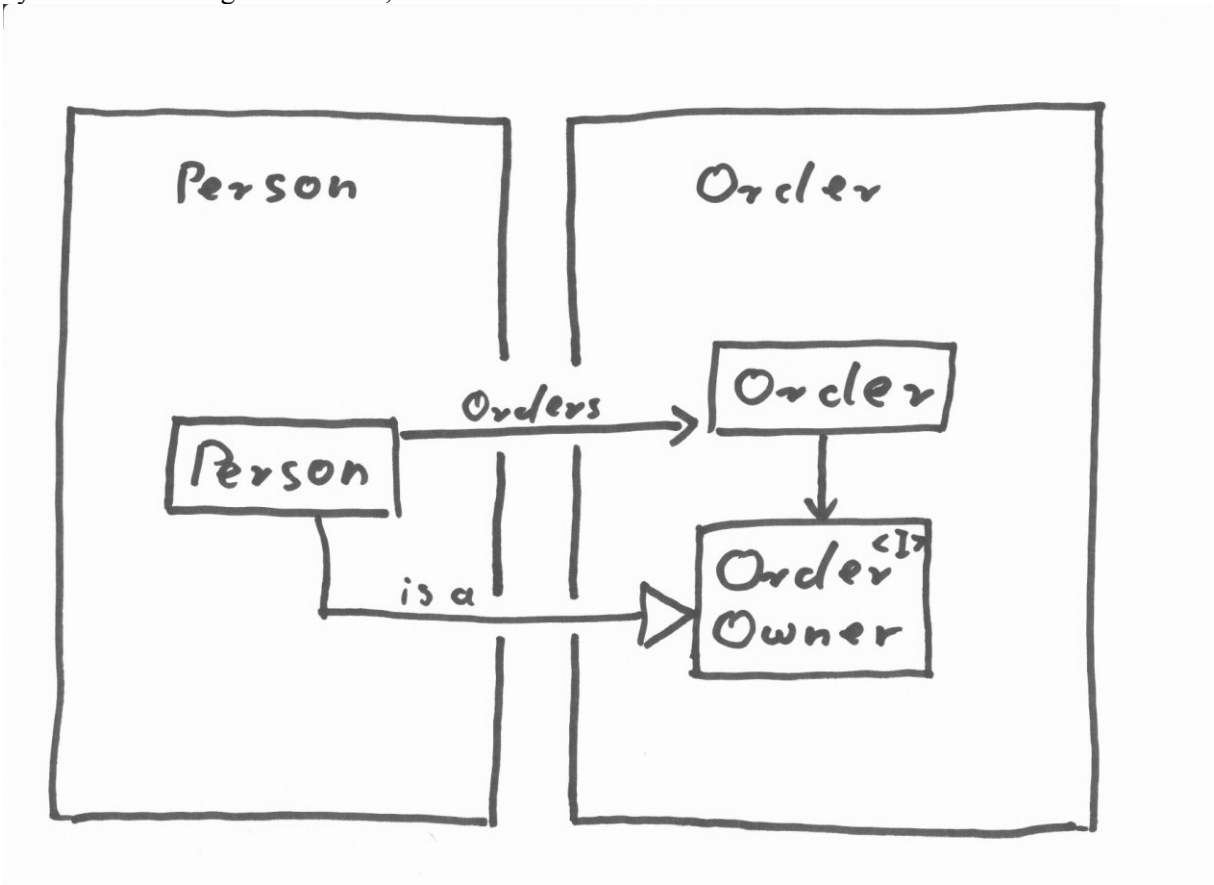

Abbildung 4 Echter Zyklus aufglöst durch Einführen eines Interfaces

Umleitung

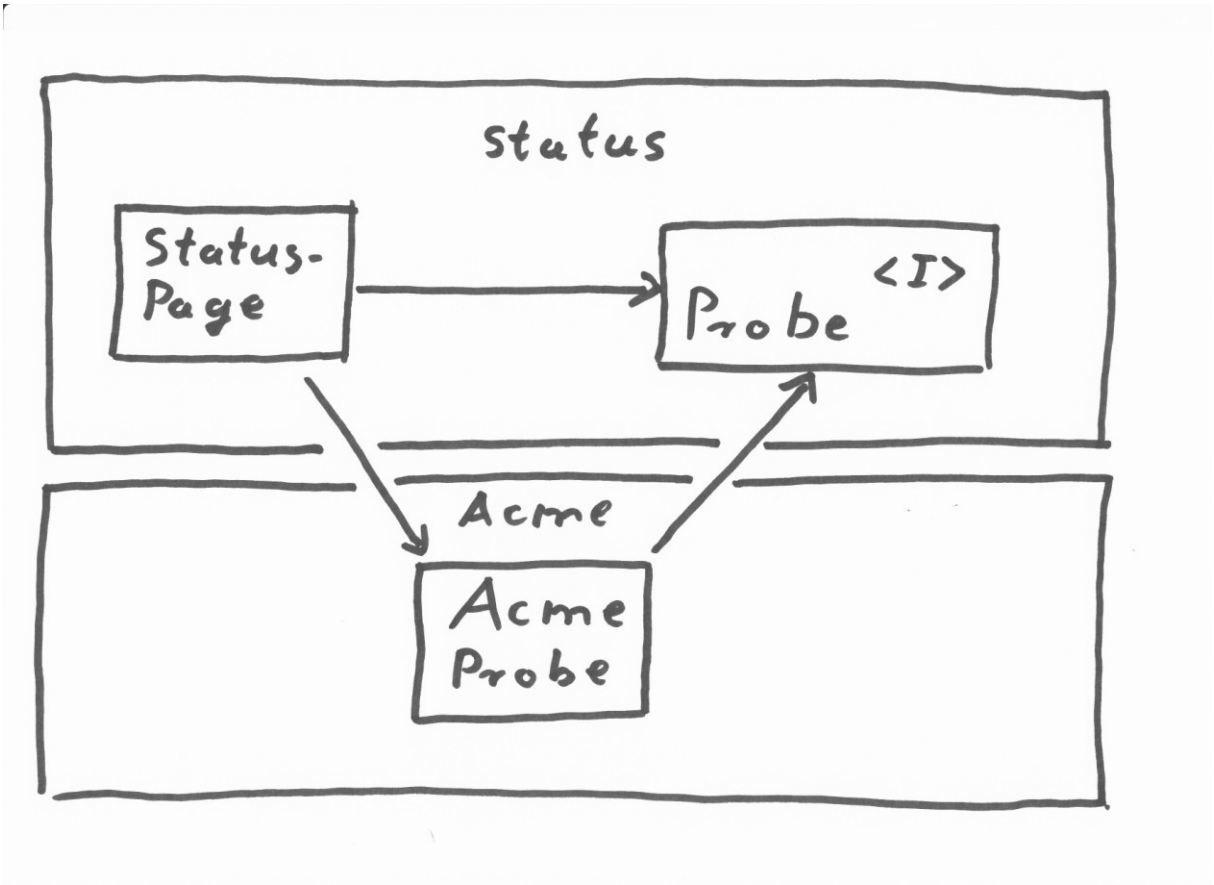

Abbildung 5 Umleitung Antipattern

Die Umleitung ist ein Antipattern welches oft entsteht, wenn versucht wird, einen Anwendungsteil sauber zu strukturieren, aber kurz vor dem Ziel aufgibt. In dem skizzierten Beispiel soll eine Anwendung über eine Statusseite Auskunft über seinen eigenen Gesundheitszustand und den von diversen Schnittstellen geben. Dafür wird ein StatusPage erzeugt, die die Information darstellt, und dafür eine Anzahl Probes abfragt. Für jede Schnittstelle, oder Komponente die auf der Statusseite dargestellt werden soll, kann nun eine solche Probe implementiert werden. Diese gehören natürlich zu den jeweiligen Schnittstellen und sind in den entsprechenden Packages implementiert.

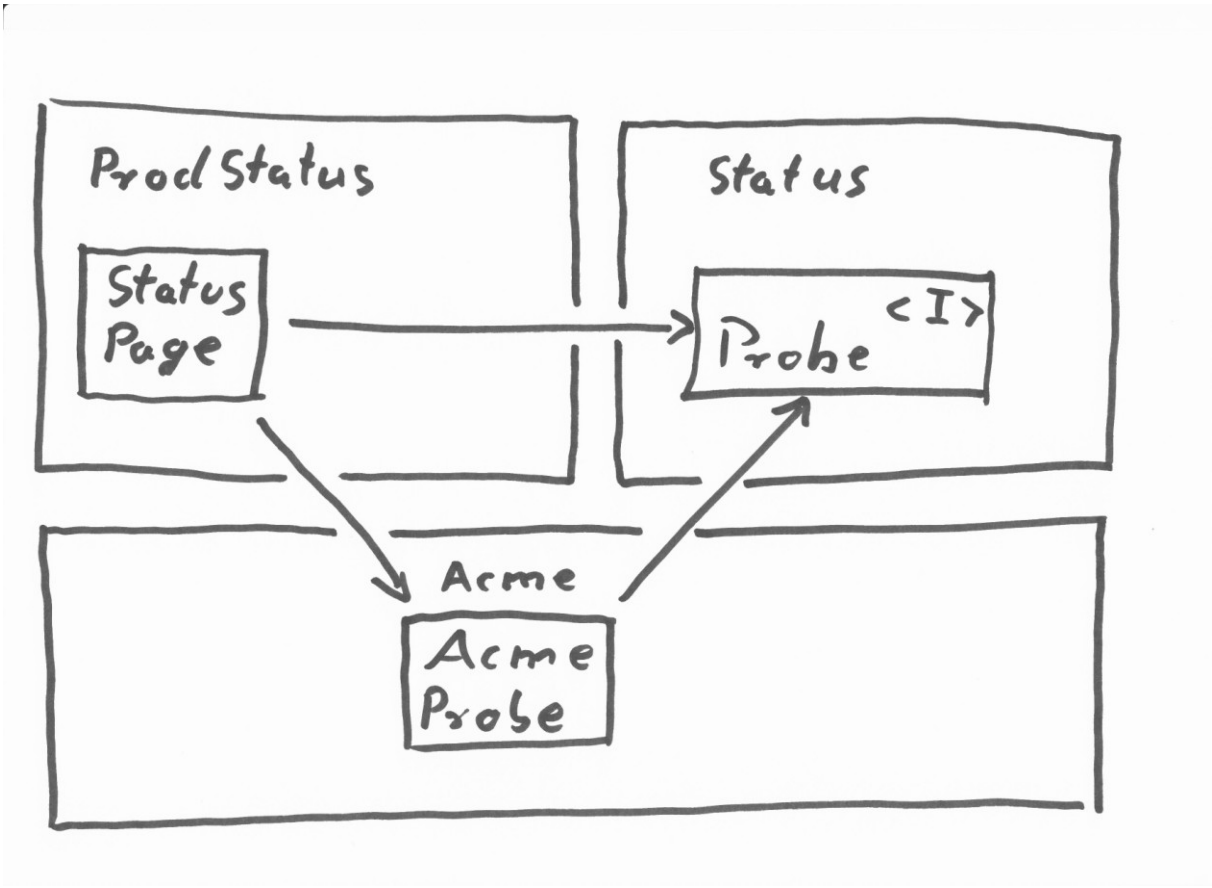

Abbildung 6 Umleitung aufgelöst durch Teilen eines Packages

Das Problem ist, dass der Teil, der alle relevanten Probe Implementierungen kennt und erzeugt, nicht Teil des Statuspackages ist, sondern Teil eines separaten Packages.

God Class/Package/Slice

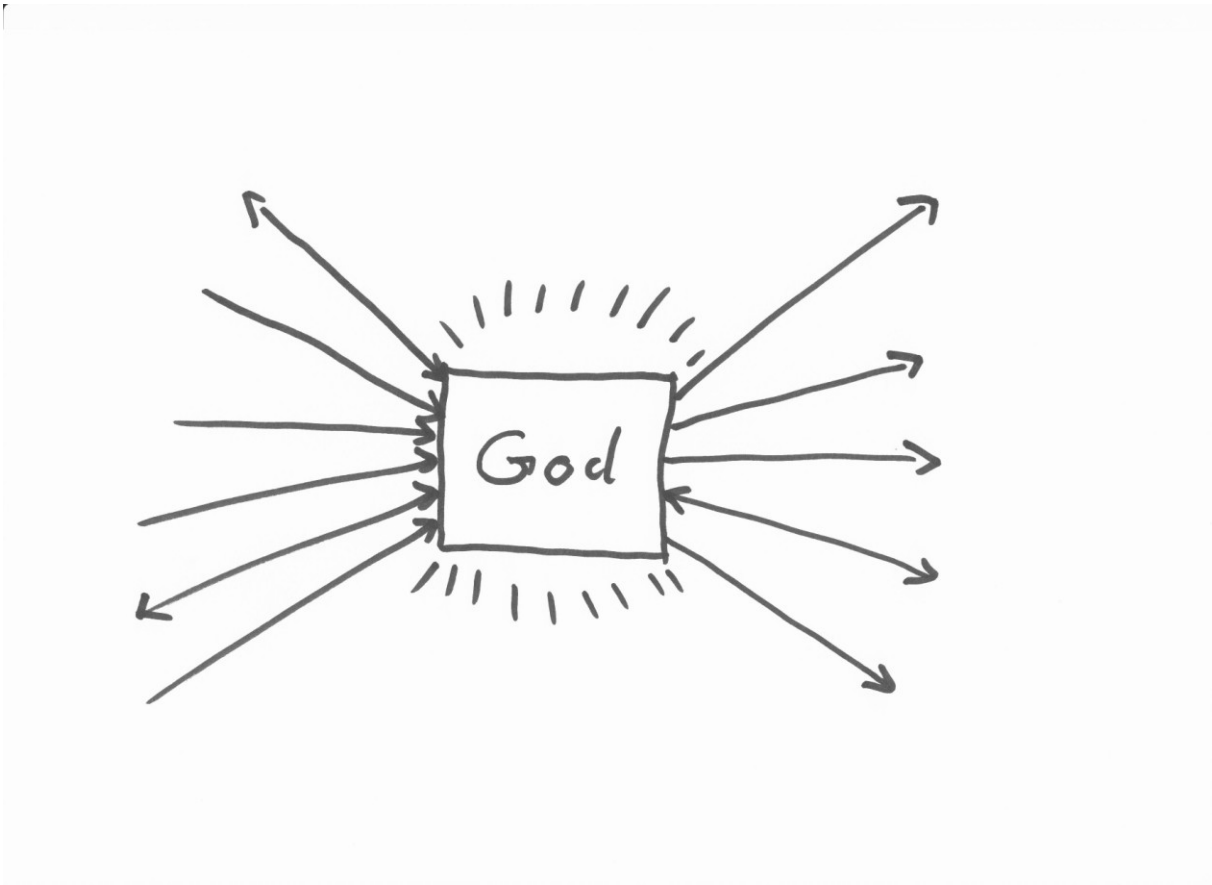

Abbildung 7 Gott Klasse Antipattern

Gott Klassen (oder Packages, oder Slices) zeichnen sich dadurch aus, dass sie sehr viele Abhängigkeiten haben, und zwar sowohl eingehende als auch ausgehende. Existiert nur die eine Art von Abhängigkeit gäbe es kaum ein Problem. Zum Beispiel hat die Klasse `java.lang.String` in den meisten Anwendungen extrem viele Klassen, die von ihr abhängen, was aber nicht wirklich ein Problem ist. Andersherum hängt eine Spring Konfiguration von extrem vielen Klassen ab, aber da nichts von der konkreten Spring Konfiguration abhängt ist auch dies nicht sonderlich problematisch. Sind aber beide Arten von Abhängigkeiten in großer Zahl vorhanden, bedeutet dies, das die Klasse (oder das Package) sich oft ändert (wenn sich eine der Knoten ändert, von dem die Gottklasse abhängt) und dass durch diese Änderung wiederum viele andere Knoten beeinflusst werden.

Auflösen lässt sich dieses Antipattern in dem man Implementierung und Interface separiert. Dies trennt die eingehenden und die ausgehenden Abhängigkeiten.

Darüber hinaus kann in den meisten Fällen feststellen, dass auf der Interface Seite das Interface Segregation Principle verletzt ist. D.h. das Interface kann und sollte in mehrere Interfaces getrennt werden, da die meisten Clients jeweils nur einen Teil des Interfaces benutzen.

Umgekehrt ist auf der Implementierungsseite meist das Single Responsibility Principle verletzt. D.h. es werden mehrere verschiedene Dinge von einer Entität erledigt. Trennt man diese Aufgaben in separate Klassen auf und delegiert an diese, reduziert sich die Anzahl der Ausgehenden Abhängigkeiten.

Wäscheleinen

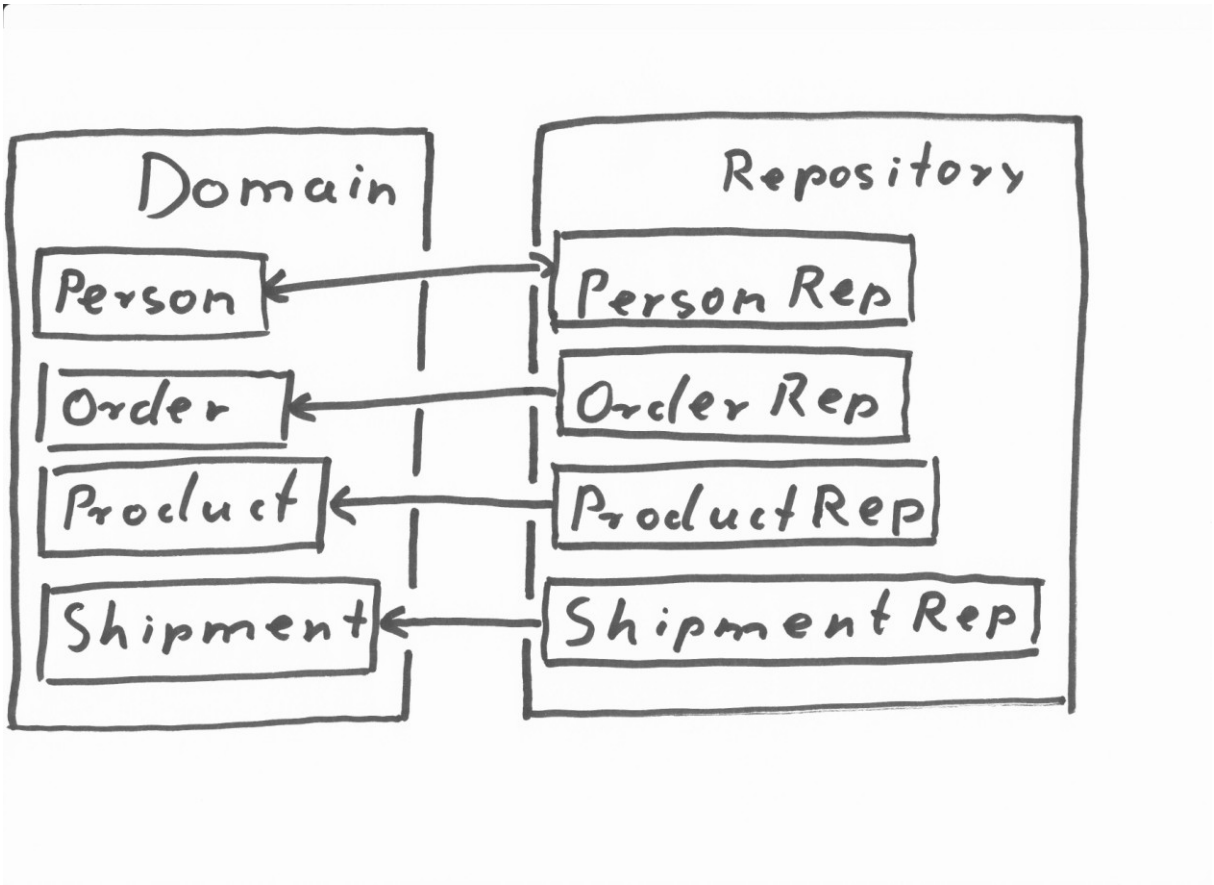

Abbildung 8 Wäscheleinen Antipattern

Das Wäscheleinen Antipattern entsteht, wenn Klassen nicht danach unterteilt werden, welche Klassen eng zusammen arbeiten, sondern nach Art der Klasse. Beliebt ist es Interfaces in ein Package zu packen, und Implementierungen in ein anderes, Hibernate Entities in ein Package, Value-Klassen in ein anderes. Das Ergebnis ist eine Verletzung des High Cohesion, Low Coupling Prinzips und sieht in Degraph aus, als hätte jemand Wäscheleinen zwischen den Packages gespannt.

In diesen Fällen ist meist eine völlige Umstrukturierung der Packages angesagt, bei der dann die Teile in ein Package kommen, die intensiv zusammenarbeiten, und Cluster, die nur wenige Abhängigkeiten haben in getrennte Packages kommen.

Degraph

Wir haben gesehen, wie Packages aussehen sollten und wie sie in der Realität aussehen. Durch intensives lesen von Source Code sind diese Strukturen aber nur schwer, bis gar nicht zu erkennen. Was wir benötigen ist ein Tool welches

- Abhängigkeiten zwischen Klassen übersichtlich darstellt, d.h. wohl als Graph

- Gleichzeitig die Zugehörigkeit von Klassen zu Packages darstellt, z.B. in dem Klassen in Packages gruppiert.

- Und auch Gruppen von Packages, die zu einem "Slice" gehören darstellt, z.B. in dem diese Packages wiederum gruppiert werden.

Degraph tut genau dies. Als Kommandozeilen Tool analysiert es class-Files und erzeugt eine GraphML Datei, die z.B. mit yed angezeigt gelayoutet und manipuliert werden kann. Details dazu findet man in der Dokumentation (http://schauder.github.io/degraph/documentation.html#visualization_of_dependencies) oder in der Live-Demo.

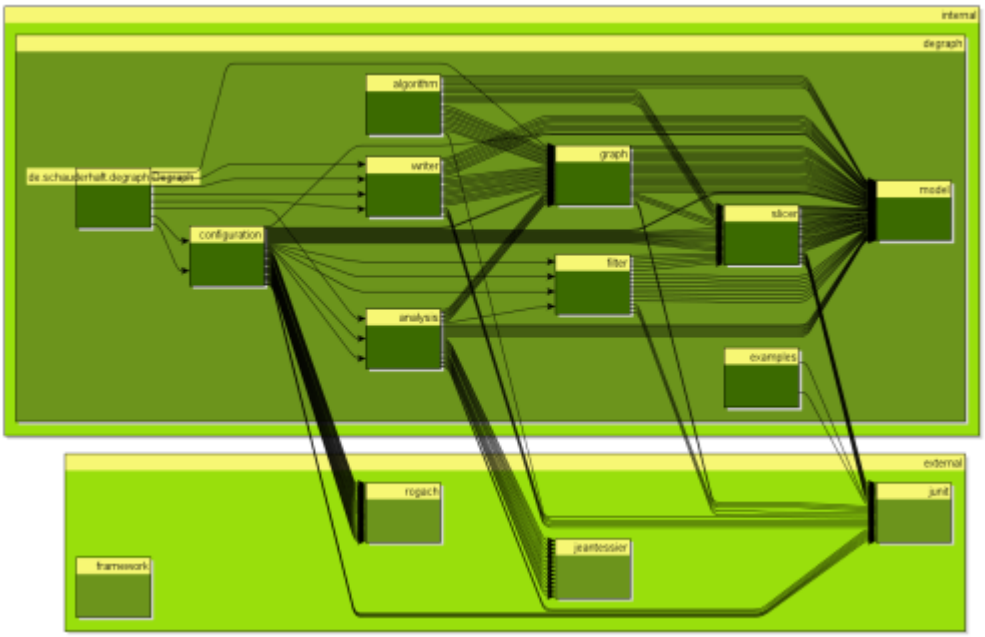

Abbildung 9 Beispiel Visualisierung von Degraph durch Degraph

Diese Art der Visualisierung ist extrem hilfreich, wenn man die Package-Struktur einer Anwendung verstehen oder verändern möchte.

Viel besser als später aufräumen ist es natürlich, gar nicht erst Unordnung aufkommen zu lassen. Dabei kann Degraph ebenfalls unterstützen. Eingebunden in ein Projekt bietet es eine DSL zum definieren von erlaubten und nicht erlaubten Abhängigkeiten.

Zur Zeit steht eine Scala DSL zur Verfügung, die Live demonstriert wird und deren Details in der Dokumentation nachgelesen (http://schauder.github.io/degraph/documentation.html#testing_of_dependencies) werden können.

Der folgende Code Block zeigt wie ein Test aussehen kann, der die aktuell Code Basis in Slices trennt und jeweils auf Zyklenfreiheit testet und auf zusätzliche Kriterien, die durch die "allow" Methoden definiert sind.

```
package de.schauderhaft.degraph.meta

import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers._
import de.schauderhaft.degraph.check.Check._
import de.schauderhaft.degraph.check.Layer._
import de.schauderhaft.degraph.check.StrictLayer

@RunWith(classOf[JUnitRunner])
class DependencyTest extends FunSuite {

    test("dependency test") {
        classpath //
            .including("de.schauderhaft.**") //
            .withSlicing("part", "de.schauderhaft.degraph.(*).**").allow("meta",
"demo")
            .withSlicing("module",
"de.schauderhaft.degraph.*.(*).**").allow("order", "person")
            .withSlicing("layer",
"de.schauderhaft.degraph.*.*.(*).**").allow(oneOf("persistence", "ui"), "domain")
should be(violationFree)
    }

}
```

Kontaktadresse:
Jens Schauder
T-Systems on site services GmbH
Alessandro-Volta-Straße 11
D-38440 Wolfsburg

E-Mail	jens.schauder@t-systems.com;	jens@schauderhaft.de
Internet:	https://www.t-systems-onsite.de/;	http://blog.schauderhaft.de