INKL. DVD

Dreamteam RAP und Eclipse 4 > 85

S&S

5.14

Deutschland € 9,80
Österreich € 10,80, Schweiz sFr 19,20

# eclipse MAGAZIN

www.eclipse-magazin.de

W-JAX 14 Alle Infos im Heft > 24, 34

## DVD-INHALT


- **Awesome Eclipse Platform News**
  von Lars Vogel und Hendrik Still
- **Making the Eclipse IDE fun again**
  von Martin Lippert
- **Wider den Innovationsstau bei Eclipse**
  von Marcel Bruch

u.v.m.

## Buchauszug:


**Baukunst für Softwarearchitekten**

von Jan Peuker

4 196603 209806 05

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

# Eclipse 4.4 Luna

**Alles Wissenswerte zum neuen Release > 26**

**Internationalisierung:** Top-Feature in Luna > 42

**Oomph:** Neues Installer-Tool bringt Schwung ins Projekt > 60

**Testautomatisierung:** Jubula meets JavaFX > 80

## Diagnose, Therapie und Prophylaxe

# Paketzyklen

Sobald ein Softwaresystem mit Paketzyklen infiziert ist, besteht große Gefahr, dass sich diese Zyklen über die Zeit der Weiterentwicklung vergrößern und erweitern – sie wuchern. Die Begleiterscheinungen eines solchen Befalls sind gravierend: Komplizierte Modularisierung, erschwerte Verständlichkeit und damit auch teurere Weiterentwicklung. Wir zeigen anhand eines praktischen Beispiels, wie eine gezielte Diagnose und Therapie von Paketzyklen durchgeführt werden kann. Weiterhin stellen wir eine wirksame Methode zur Prophylaxe für Softwarepatienten vor, die eine äußerst geringe Ansteckungsgefahr mit Zyklen garantiert.

von David Burkhart und Marc Philipp

Ein Java-Paket A hängt von einem Java-Paket B genau dann ab, wenn es in A eine Klasse gibt, die eine Klasse im Paket B referenziert. Ein Paketzyklus entsteht durch eine zyklische Abhängigkeit zwischen zwei oder mehr Paketen. Wir bezeichnen eine maximal große Menge von Paketen, die zyklisch voneinander abhängen, als *Paketzyklus*. Dabei werden auch transitive Abhängigkeiten berücksichtigt. Die Anzahl der Pakete in einem Paketzyklus wird als dessen *Größe* bezeichnet. In **Abbildung 1** ist ein Paketzyklus der Größe 2 dargestellt.

Ein Paketzyklus wird durch Abhängigkeiten zwischen Klassen hervorgerufen. In **Abbildung 2** sind die Klassen abgebildet, die den Paketzyklus aus **Abbildung 1** verursachen. Die Farben der Klassen entsprechen ihrer Paketzugehörigkeit. In diesem Fall gibt es keinen Zyklus auf Klassenebene, sehr wohl aber auf Paketebene.

### Warum sollte man Paketzyklen vermeiden?

Wie bei vielen anderen Metrikverletzungen oder Code-Smells leidet auch bei Paketzyklen die Verständlichkeit des Codes. Beim Versuch, den Code zu verstehen und dem Ablauf zu folgen, entsteht durch die Zyklen eine verwirrend große Anzahl von Möglichkeiten, die Abhängigkeitsstrukturen zu verfolgen. Abläufe des Programmcodes sowie die Verantwortlichkeiten einzelner Pakete sind schwer zu erkennen.

Weiterhin wird es in einer Paketstruktur mit zyklischen Abhängigkeiten ungleich komplizierter, einzelne Pakete wiederzuverwenden. Genauso verhält es sich mit parallelem Arbeiten an der Codebasis: Durch die zyklischen Abhängigkeiten werden sich verschiedene Teams bzw. Entwickler immer wieder in die Quere kommen. Eine klare Trennung mittels einer klaren Schnittstelle zwischen den Paketen ist unmöglich.

Viele dieser Nachteile könnten dadurch abgeschwächt werden, dass Gruppen von Paketen in eigenen Subprojekten zusammengefasst werden. Das reduziert die Gefahr, einen riesigen Paketzyklus zu erhalten, deutlich, da die einzelnen Projekte nicht zyklisch voneinander abhängen können. Innerhalb eines jeden Projekts können immer noch Zyklen entstehen, die aber insgesamt wesentlich kleiner ausfallen. Ist diese Modularisierung nicht vorhanden, behindern Paketzyklen das nachträgliche Schneiden und Aufteilen ungemein. Dennoch: Auch

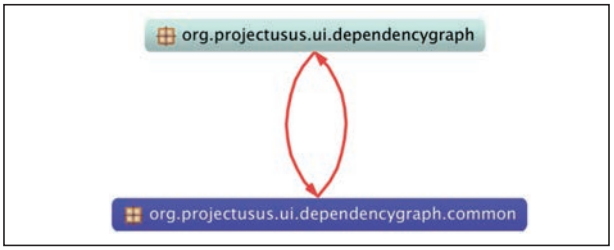



Abb. 1: Paketzyklus der Größe 2

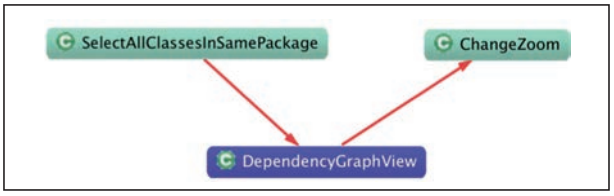



Abb. 2: Paketzyklus verursachende azyklische Klassenabhängigkeiten

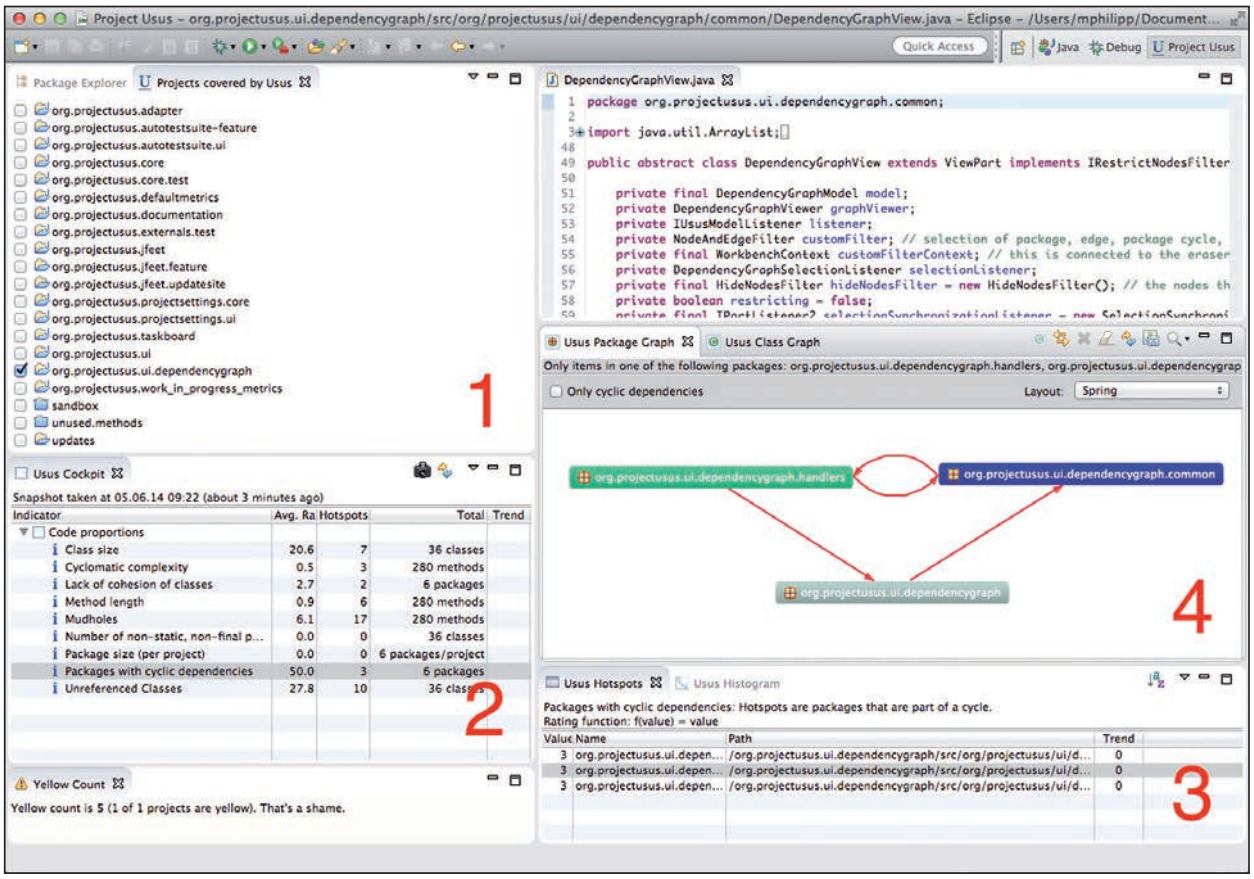



Abb. 3: Die „Project Usus"-Perspektive in Eclipse

wenn die Zyklen innerhalb eines Moduls nicht größer werden können als das Modul selbst, sollten sie trotzdem eingedämmt werden, da auch Module wachsen und dann die genannten Nachteile von Paketzyklen innerhalb des Moduls auftreten.

## Diagnose und Therapie

Um einen Paketzyklus entfernen zu können, muss man ihn zunächst einmal erkennen – und verstehen, was ihn verursacht. Dazu ist unserer Erfahrung nach ein gutes Werkzeug zur Visualisierung und Analyse unerlässlich. Erst wenn man den Paketzyklus verstanden hat, kann man eine adäquate Lösung ableiten.

Häufig gibt es mehrere mögliche Lösungen. Speziell bei größeren Paketzyklen fällt es schwer, auf Anhieb eine Folge von Schritten aufzustellen, die den Zyklus auflösen werden. Stattdessen ist eine iterative, explorative Herangehensweise erforderlich. Dazu benötigt man direktes Feedback, ob der gerade durchgeführte Teilschritt in die richtige Richtung führt.

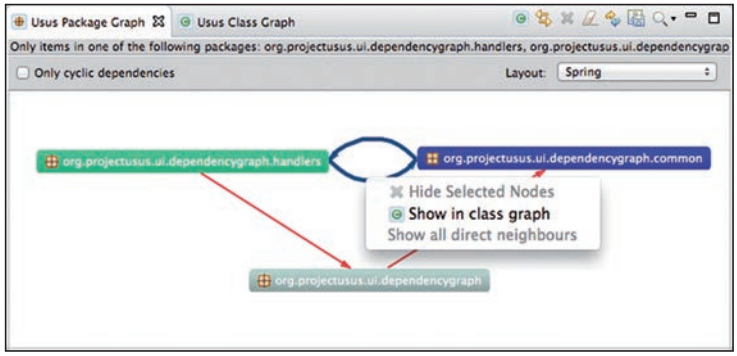



Abb. 4: Paketzyklus der Größe 3

Project Usus [1] ermöglicht genau das. Usus ist ein Eclipse-Plug-in, das sich über den Eclipse Marketplace sehr einfach installieren lässt. Nach der Installation kann Usus über eine eigene Perspektive (**Abb. 3**) erreicht werden. Dort müssen zunächst die zu untersuchenden Projekte in der View *Projects covered by Usus* (1) angewählt werden.

Das *Usus Cockpit* (2) zeigt dabei verschiedene Metriken, die sofort nach dem Speichern einer Java-Datei aktualisiert werden. Durch die Integration von Usus in die IDE ist kein Kontextwechsel notwendig, um Metriken zu analysieren und Problemstellen zu beheben. Über die Anzeige von Trends werden die Auswirkungen der zuletzt durchgeführten Änderungen am Code direkt im *Usus Cockpit* sichtbar.

Um vom groben Überblick im *Usus Cockpit* mehr in die Detailbetrachtung überzugehen, kann man die einzelnen Verletzungen einer Metrik in der View *Usus Hotspots* (3) darstellen, indem man die entsprechende Metrik im *Usus Cockpit* doppelklickt. Werden Paketzyklen in der Hotspotsview dargestellt, kann man einen konkreten Hotspot durch Doppelklick im *Usus Package Graph* (4) visualisieren.

Bevor wir nun damit beginnen, diesen Zyklus zu beheben, möchten wir Usus signalisieren, dass wir einen

„Snapshot“ des aktuellen Stands der Metriken markieren wollen. Denn damit bekommen wir von Usus direktes Feedback über die Auswirkungen unserer Aktionen. Dieser Snapshot kann im *Usus Cockpit* mit dem Kamera-Icon markiert werden. Verbessern oder verschlechtern Refactorings die Metriken gegenüber dem markierten Stand, wird das sofort in der Trendspalte angezeigt.

Der in **Abbildung 4** dargestellte Paketzyklus kann mithilfe des *Usus Class Graph* genauer analysiert werden. In unserem Beispiel sieht der Zyklus zwischen *...common* (in blau) und *...handlers* (in grün) interessant aus, da sich direkte Zyklen zwischen zwei Paketen häufig relativ einfach auflösen lassen.

In Usus gibt es die Möglichkeit, im *Usus Class Graph* nur diejenigen Klassen anzuzeigen, die für diesen Zyklus relevant sind. Dazu muss das Kontextmenü auf dem Zyklus geöffnet werden. Die entsprechenden Kanten im *Usus Package Graph* erscheinen als markiert. Über die Aktion *Show in class graph* wird der Kontext dieser Paketabhängigkeiten in den *Usus Class Graph* übernommen, und dieser zeigt die Klassen aus beiden Paketen wie in **Abbildung 5** an. Die Farbe der Klassen im *Usus Class Graph* entspricht dabei der Farbe der Pakete aus dem *Usus Package Graph*. In unserem Anwendungsfall hilft dort die Auswahl eines der beiden Tree-Layouts, da so besonders diejenigen Klassen hervorstechen, die der allgemeinen Baumstruktur der Abhängigkeiten entgegenstreben. Ein Tree (oder Baum) ist eben ein zyklenfreier Graph.

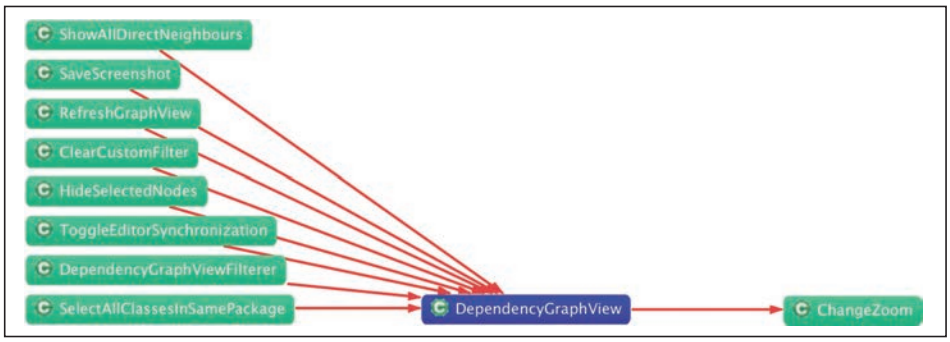



Abb. 5: Klassenabhängigkeiten, die Paketzyklus aus Abbildung 3 verursachen

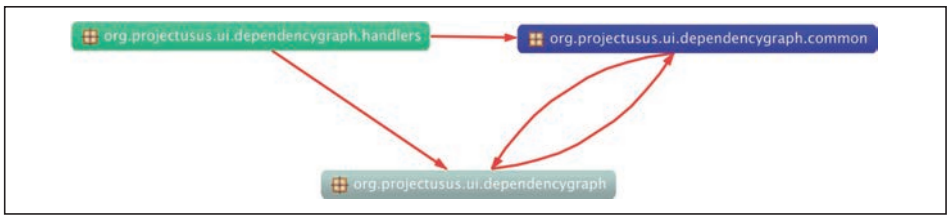



Abb. 6: Veränderter Paketzyklus nach erstem Refactoring-Versuch

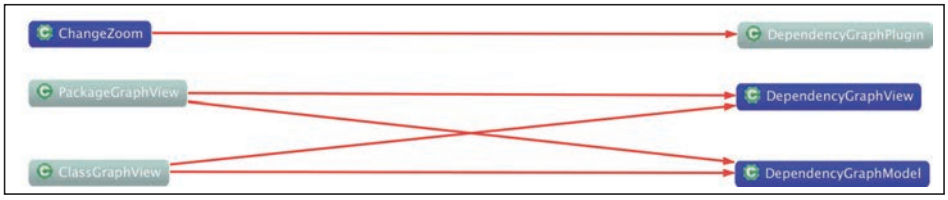



Abb. 7: Klassenabhängigkeiten, die Paketzyklus aus Abbildung 6 verursachen

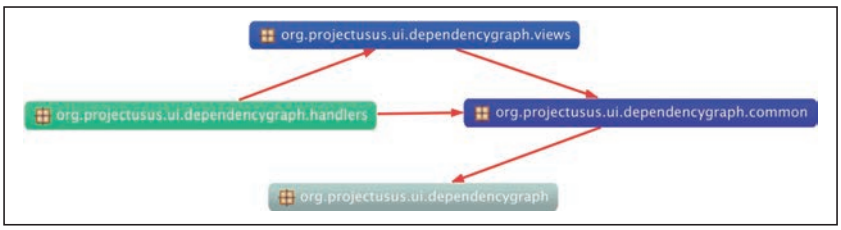



Abb. 8: Zyklenfreie Paketabhängigkeiten nach erfolgreichem Refactoring

Uns fällt so schnell die Klasse *ChangeZoom* ins Auge, die als einzige grüne Klasse aus dem blauen Paket referenziert wird. Über einen Doppelklick öffnen wir die Klasse und verschieben sie mit Eclipse-Bordmitteln in das blaue Paket (*...common*).

Leider hat die Verschiebung der Klasse *ChangeZoom* in das *...common*-Paket zwar den direkten Zyklus aus **Abbildung 4** behoben, dafür aber, wie man in **Abbildung 6** sieht, einen neuen direkten Zyklus zwischen den Paketen *...common* (in blau) und *...dependencygraph* (in grau) verursacht.

Öffnet man per Kontextmenü den *Usus Class Graph* mit den Klassenabhängigkeiten, die den neuen direkten Paketzyklus verursachen, ergibt sich das Bild in **Abbildung 7**.

Von diesem Zustand aus erscheint es uns als sinnvoll, ein eigenes Paket für die Views einzuführen. Nach der Erstellung des Pakets und der Verschiebung der entsprechenden Klassen mithilfe der Eclipse-Refactoring-Möglichkeiten in das neue Paket ergibt sich die neue Abhängigkeitssituation der Pakete in **Abbildung 8**.

Das erfolgreiche Refactoring äußert sich nicht nur in den Graphen, sondern auch in den Usus-Hotspots und im *Usus Cockpit*. In den Hotspots ist die Verletzung als gelöscht dargestellt und im *Usus Cockpit* ist, wie in **Abbildung 9**, in der Trendspalte zu sehen, dass die Metriken „Lack of cohesion of classes“ und „Packages with cyclic dependencies“ verbessert wurden und keine weitere Metrik verschlechtert wurde. Dies ist nun ein günstiger Zeitpunkt, um Tests durchzuführen und den Zwischenstand zu sichern. Jetzt sollte auch ein neuer Snapshot in Usus markiert werden, um für weitere Arbeiten wieder den Trend, ausgehend vom aktuellen Stand, zu erhalten.

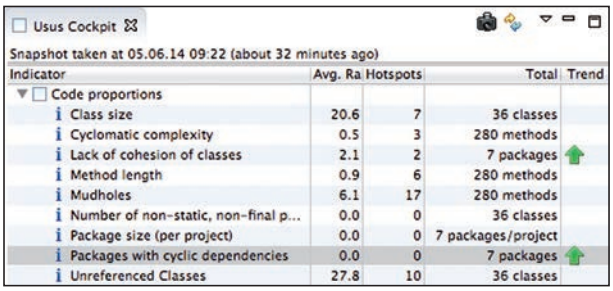



Abb. 9: Das „Usus Cockpit“ mit positivem Trend

### Weitere Therapieformen

Für das eigentliche Refactoring, mit dem der Paketzyklus entfernt wird, gibt es verschiedene Möglichkeiten. Durch die Integration von Usus in Eclipse stehen ohne Kontextwechsel alle von Eclipse mitgebrachten, automatisierten Refactorings zur Verfügung. Damit ist es extrem günstig, einen Lösungsansatz auszuprobieren.

- Klasse liegt im falschen Paket: Bei dieser einfachen Form muss die Klasse nur in das richtige Paket verschoben werden. Bei komplizierteren Fällen, bei denen mehrere Klassen im falschen Paket liegen, ist es eventuell notwendig, zusätzliche Pakete anzulegen und die Aufteilung der Klassen in die Pakete zu überarbeiten.
- Klassen referenzieren sich gegenseitig: Bei dieser Form sind mehrere Lösungen möglich. Eventuell ist es sinnvoll, die Klassen zu zerlegen und neu aufzuteilen. Eine noch häufiger angewandte Methode ist die Einführung eines Interface.

Nehmen wir beispielsweise an, eine *View* kennt eine *Action*, da sie diese auf ihrer *Toolbar* hinzufügt. Die *Action* bekommt weiterhin die Instanz der *View* übergeben, da sie diese *View* refreshen soll, nachdem die eigentliche Operation durchgeführt wurde. Liegen die *Action* und die *View* in verschiedenen Paketen, liegt ein Paketzyklus vor. Hier ist ein *Interface* sinnvoll, das die *Action* zur Verfügung stellt und die Refresh-Funktionalität als Methode deklariert. Die *View* implementiert dieses *Interface*. Vorteile: Die *Action*-Klasse ist leichter verständlich. Es ist klar, was sie mit der View vorhat. Weiterhin ist die *Action* nun durch einen Unit Test testbar, da das *Interface* mit einem *Dummy* oder einem *Mock* erfüllt werden kann. Zuvor konnte die *View* nicht einfach instanziiert werden, da sie Abhängigkeiten auf SWT mitbringt.

Bei größeren Zyklen, an denen mehrere Klassen beteiligt sind, sind oft beide Arten von Refactorings notwendig. Es empfiehlt sich, einzelne wenige Pakete und Klassen zu identifizieren, bei denen klar ist, wie die gewünschte Abhängigkeitsrichtung auszusehen hat. Sind diese bereinigt, wird der Zyklus überschaubarer, und man kann sich Stück für Stück voranarbeiten. Wichtig dabei ist, sinnvolle Zwischenstände zu sichern. Bei dieser inkrementellen Vorgehensweise unterstützt Usus den Entwickler durch direkte Anzeige von Trends und sofortige Aktualisierung der Charts.

### Prophylaxe

Um auch in Zukunft zuverlässig über neue Paketzyklen informiert zu werden und diese schnell entfernen zu können, empfiehlt es sich, Toolunterstützung für den automatisierten Build einzusetzen. Zwar sind einzelne neue Zyklen in einem sonst zyklenfreien Projekt nicht ganz so schlimm, allerdings sind sie in diesem Stadium auch noch sehr einfach behebbar. Das wird später anders: Wenn sich bereits ein großer Knoten etabliert hat, ist die Therapie nahezu unmöglich. Daher ist es nach unserer Erfahrung insgesamt weniger aufwändig, an dieser Stelle lieber etwas zu strikt zu sein.

Die Toolunterstützung für den Build kann z. B. mit JDepend [2] umgesetzt werden. JDepend ist ein Werkzeug, das auf Class-Files verschiedene Metriken bezüglich Paketen und Architektur messen kann, unter anderem auch, welche Pakete in Zyklen beteiligt sind. Einbinden kann man das Tool in den kontinuierlichen Build auf verschiedene Arten. Zum einen ist ein einfacher JUnit-Test denkbar, der fehlschlägt, sobald neue Zyklen gefunden wurden. Ein Beispiel dafür ist auf der JDepend-Webseite selbst zu finden [2]. Zum anderen kann man JDepend auch mithilfe von Plug-ins in die entsprechenden Build-Systeme einbinden. Für Maven gibt es für das Maven-Enforcer-Plug-in eine so genannte *Rule* [3], die einfach in den Maven Build integrierbar ist. Ist diese Rule aktiviert, schlägt der Build fehl, sobald ein Paketzyklus entsteht.

### Fazit

Kombiniert man diese Maßnahmen der Diagnose, Therapie und Prophylaxe, so erhält man einen effektiven Schutz gegen Paketzyklen und somit eine aufgeräumte Codebasis, in der viele Voraussetzungen für gute Modularisierung und Verständlichkeit bereits sichergestellt sind. Zudem ist die inkrementelle Herangehensweise zur Verbesserung der Codequalität mit Usus auch für andere wichtige Metriken anwendbar, etwa für zyklomatische Komplexität oder Klassengröße. Es kann sich also in vielerlei Hinsicht lohnen, Usus einzusetzen.

**Marc Philipp** (andrena objects ag) beschäftigt sich neben seiner Tätigkeit als Softwareentwickler und Trainer mit der Arbeit an Entwicklungswerkzeugen, insbesondere JUnit und Project Usus.

**David Burkhart** ist seit 2004 professioneller Softwareentwickler. Sein besonderes Interesse gilt Clean Code Development, XP und SCRUM. Heute ist er für andrena objects sowohl als Entwickler als auch als Trainer und Coach für agile Methoden tätig.

### Links & Literatur

[1] Project Usus: http://www.projectusus.org

[2] JDepend: http://clarkware.com/software/JDepend.html

[3] NoPackageCyclesRule für das Maven Enforcer Plugin: https://github.com/andrena/no-package-cycles-enforcer-rule

# ECLIPSE³

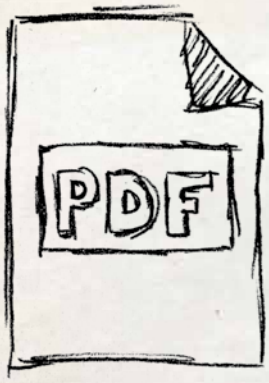
**Jetzt abonnieren!**
**www.eclipse-magazin.de**

## Jetzt 3 TOP-VORTEILE sichern!


**1**

- Alle Printausgaben frei Haus erhalten
- Intellibook-ID kostenlos anfordern (www.intellibook.de)


**2** Abo-Nr. (aus Rechnung oder Auftragsbestätigung) eingeben




**3** Zugriff auf das komplette PDF-Archiv mit der Intellibook-ID

S&S Media Group

**www.eclipse-magazin.de**