

# Einsicht und Handeln mit Usus

## Integrierte Codeanalyse: Einfache Handhabung und unmittelbares Feedback

In den letzten Jahren hat sich in der Entwicklergemeinde mehr und mehr ein Bewusstsein für Codequalität bzw. innere Softwarequalität entwickelt. Doch was bedeutet Codequalität eigentlich? Kann man sie messen? Und falls ja, welche Erkenntnisse gewinnt man daraus und wie kann man diese zur Verbesserung der Qualität nutzen? Usus versucht eine Antwort auf diese Fragen zu geben.

## Warum Codequalität?

Jedes Stück Code wird nur einmal geschrieben, aber möglicherweise Hunderte von Malen gelesen und nachvollzogen. Daher ist es sehr effektiv, den Code so zu gestalten, dass das Verstehen einfach wird. Denn im direkten Zusammenhang mit der Nachvollziehbarkeit steht die Änderbarkeit: Man sollte nur Code ändern, den man versteht! Nur so lassen sich Fehler vermeiden. Usus prüft verschiedene Aspekte des Codes, die in unseren Augen essentiell für eine gute Nachvollziehbarkeit und damit eine problemlose Änderbarkeit des Codes sind.

**Warum soll ich meinen Code überhaupt ändern?** Beim Entwickeln von Software ist die Versuchung groß, sich nicht allzu lange mit dem Aufräumen der Codebasis aufzuhalten. Unsere Erfahrungen aus Legacy-Projekten lehren uns jedoch, dass solche Versäumnisse früher oder später zu schwer wartbarem Code führen.

**Warum ein Analysewerkzeug?** Analysewerkzeuge unterstützen den Entwickler dabei, Fehler und negative Entwicklungen frühzeitig zu erkennen und zu beheben. So kann man daraus lernen und derartige Probleme im weiteren Verlauf des Projekts vermeiden. Des Weiteren geben Analysewerkzeuge Kriterien dafür vor, was lesbaren und verständlichen Code ausmacht. So gelten für alle Teammitglieder dieselben Regeln und erlauben es ihnen, langfristig ihr Handeln an diesen Kriterien auszurichten.

**Wo kann ich weiterlesen?** Weitere Informationen zu Refactoring und Clean Code findet man in den Büchern von Fowler [1] bzw. Martin [2]. Die Clean Code Developer-Initiative [3] bietet einen Ansatz, wie man auch im Entwicklungsalltag die Codequalität nicht aus den Augen verliert.

## Warum Usus?

Es gibt viele Code-Analysewerkzeuge für Java, die sich in Eclipse integrieren lassen, wie zum Beispiel Checkstyle [4], FindBugs [5] und PMD [6]. Warum bauen wir also noch ein weiteres derartiges Werkzeug? Bei der Entwicklung waren folgende Aspekte ausschlaggebend:

**Einfache Benutzbarkeit.** Es genügt, Usus zu installieren, in die *Project Usus*-Perspektive zu wechseln, die gewünschten Projekte auszuwählen und die initiale Berechnung anzustoßen. Alles weitere geschieht automatisch. Es ist nicht erforderlich, umfangreiche Konfigurationen vorzunehmen oder Detailwissen über die einzusetzenden Metriken zu haben.

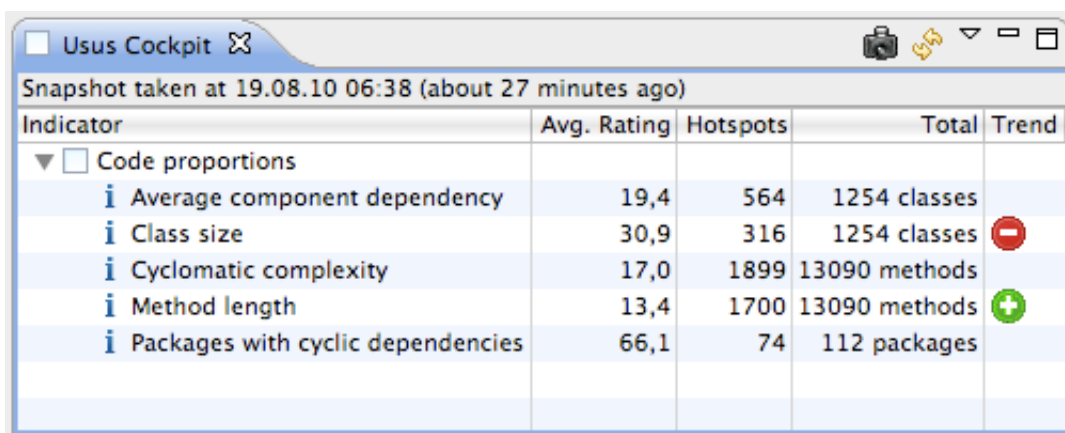
**Unmittelbares Feedback.** Abgesehen vom ersten Berechnungslauf, der bei großen Projekten ein paar Minuten in Anspruch nehmen kann, werden Änderungen im Code innerhalb von Sekunden analysiert und sofort in sämtliche Views übernommen: Das Cockpit ändert seine Werte und den Trend, die Hotspot-Listen werden aktualisiert, und die Graphen passen sich an die neuen Gegebenheiten an.

**Integration in die IDE.** Bei Verwendung eines externen Werkzeugs zur statischen Analyse wie z. B. Sonar [7] muss man die gewonnenen Erkenntnisse stets in die Entwicklungsumgebung übertragen. Zur Überprüfung, ob eine Änderung im Code den gewünschten Effekt auf die Codequalität gehabt hat, ist ein erneuter Aufruf des externen Werkzeugs erforderlich. Mit Usus entfällt dieser Wechsel zwischen zwei Umgebungen.

**Ganzheitliche Betrachtung der Codebasis.** Das Cockpit liefert einen komprimierten Überblick über alle Aspekte des gesamten Codes. Möchte man z. B. einen Paketzyklus entfernen, ändert man den Code in den beteiligten Klassen und sieht sofort die Auswirkungen dieser Änderung in den Usus-Views. So behält man immer das große Ganze im Blick und verliert sich nicht in Details.

## Die Bestandteile von Usus

**Cockpit.** Das Cockpit (Abb. 1) ist der zentrale View von Usus. Es gibt einen Überblick über die berechneten Statistiken und ihre aktuellen Werte. Neben der Gesamtzahl der zugrundeliegenden Elemente (*Total*) wird die Anzahl derjenigen Elemente mit einer Bewertung größer als 0 (*Hotspots*) sowie die durchschnittliche Bewertung der Statistik in Prozent (*Avg. Rating*) angezeigt. In der *Trend*-Spalte ist die Veränderung seit dem letzten *Snapshot* dargestellt. Snapshots lassen sich über den Toolbar-Button mit Kamerasymbol erstellen und ermöglichen es, Veränderungen im Code sichtbar zu machen. Beispielsweise ist es ratsam, einen Snapshot vor einem größeren Refactoring oder vor Beginn eines Implementierungstasks anzulegen.



The screenshot shows the 'Usus Cockpit' window with a toolbar at the top containing a camera icon, a magnifying glass, and window controls. Below the toolbar, a status bar indicates 'Snapshot taken at 19.08.10 06:38 (about 27 minutes ago)'. The main area contains a table with the following data:

Indicator	Avg. Rating	Hotspots	Total	Trend
▼ <input type="checkbox"/> Code proportions				
ⓘ Average component dependency	19,4	564	1254 classes	
ⓘ Class size	30,9	316	1254 classes	–
ⓘ Cyclomatic complexity	17,0	1899	13090 methods	
ⓘ Method length	13,4	1700	13090 methods	+
ⓘ Packages with cyclic dependencies	66,1	74	112 packages	

Abb. 1: Das Usus-Cockpit

**Hotspots.** Das Motto von Usus lautet “Einsicht und Handeln”. Das Cockpit erlaubt zwar Einsicht in den aktuellen Zustand des Codes, aber zum Handeln genügt dies nicht. Deshalb gelangt man von jeder Statistik durch einen Doppelklick zum *Hotspots View*. Dort öffnet sich durch Doppelklicken auf einen Hotspot entweder das entsprechende Element im *Java Editor* oder (bei Paketzyklen) der *Package Graph View*. Wie im Cockpit kann man auch hier den Trend seit dem letzten Snapshot im Auge behalten.

**Trends.** Usus zeigt Veränderungen sowohl im Cockpit als auch in den Hotspots durch ein grünes Plus- bzw. ein rotes Minuszeichen an. Interessant hieran ist, dass sich bei der Verbesserung eines Hotspots an anderen Stellen Verschlechterungen ergeben können, z. B. beim Extrahieren von Methoden. Die extrahierten Methoden können zu lang sein, oder andere Statistiken können sich verschlechtern, etwa die Klassengröße. Da die Cockpit-Trends ein rotes Minus anzeigen, sobald sich auch nur ein Hotspot verschlechtert hat, fungiert dieses Feedback als Frühwarnsystem insbesondere dann, wenn sich eine beabsichtigte Verbesserung als Seiteneffekt an anderer Stelle nachteilig auswirkt.

**Metriken und Statistiken.** In Usus unterscheiden wir zwischen *Metriken* und *Statistiken*. Eine Metrik führt eine statische Analyse des vorliegenden Codes durch und extrahiert den Code beschreibende *Messergebnisse*. Statistiken fassen in einem separaten Schritt die Resultate der Metriken zusammen und ordnen jedem Messergebnis  $x$  eine *Bewertung*  $f(x)$  zu. Dabei gilt: Je höher das Ergebnis der Bewertungsfunktion  $f(x)$ , umso schlechter ist die Bewertung von  $x$ .

Die meisten Statistiken arbeiten linear, d. h. Messergebnisse unterhalb eines festgelegten *Limits*  $L$  werden mit 0 bewertet, die Bewertung darüberliegender Ergebnisse wird durch eine linear steigende Funktion berechnet. So bekommen Elemente mit sehr schlechten Messergebnissen eine schlechtere Bewertung als solche, die nur knapp über dem Limit liegen. Zur Ermittlung der *prozentualen Durchschnittsbewertung* einer Statistik wird die Summe aller Bewertungen durch die Gesamtzahl aller Elemente dividiert und als Prozentzahl dargestellt, d. h. mit 100 multipliziert.

Tabelle 1 gibt eine Übersicht über die Usus-Metriken und -Statistiken. Eine detaillierte Beschreibung kann man jederzeit über Tooltips und die dynamische Hilfe in Eclipse nachlesen.

*Tabelle 1: Metriken und Statistiken*

Statistik	Art	Ebene	Messergebnis $x$	Limit $L$	Bewertung $f(x)$ für $x > L$
Methodenlänge	dateilokal	Methoden	Anzahl Statements	9	$1/L \cdot x - 1$
Zyklomatische Komplexität	dateilokal	Methoden	Anzahl Verzweigungen	4	$1/L \cdot x - 1$
Klassengröße	dateilokal	Klassen	Anzahl Methoden	12	$1/L \cdot x - 1$
Average Component Dependency	dateiübergreifend	Klassen	Anzahl bekannter Klassen (reflexiv, transitiv)	$1,5 / 2^{\log_5(\#Klassen)}$	$x / \#Klassen$
Paketzyklen	dateiübergreifend	Pakete	Anzahl Pakete im gleichen Zyklus (reflexiv)	1	1

## Usus im praktischen Einsatz

Wir möchten die Arbeit mit Usus anhand eines Beispielprojekts demonstrieren. Um ein Projekt zur Verfügung zu haben, das eine interessante Größe und eine realistische Struktur besitzt, haben wir einen Teil der Eclipse-Sourcen gewählt, und zwar das Platform/Core-Projekt. Dieser Code liegt als Workspace auf der beiliegenden Heft-CD. Da es sich bei den Beispielprojekten selbst um Eclipse-Plugins handelt, benötigt man zum Öffnen des Workspaces eine geeignete Eclipse-Version, zum Beispiel das Eclipse Classic-Paket, und natürlich Usus [8].

Anhand der Methodenlänge und der Paketzyklen zeigen wir im Folgenden beispielhaft, wie Usus Schwachstellen im Code aufzeigt und wie man diese beheben kann.

**Methodenlänge.** Der Ermittlung der Methodenlänge liegt die Gesamtzahl von 13089 Methoden zugrunde. Von diesen haben 1700 Methoden mehr als 9 Statements, mit einer durchschnittlichen Bewertung von 13,4. Ein Doppelklick auf die Statistik öffnet den Hotspots View und zeigt die Methoden an. Die Methode `FrameworkCommandProvider._bundle()` ist mit 164 Statements mit Abstand am größten. Per Doppelklick auf diesen Hotspot gelangt man zur Methode im Java Editor. Hier lässt sich durch das Extrahieren von Methoden mehr Struktur in den Code hineinbringen, Redundanz vermeiden, die Übersichtlichkeit erhöhen und — quasi als Nebeneffekt — die Methode

kürzen.

**Paketzyklen.** Von den 112 analysierten Paketen befinden sich 74 in Paketzyklen, was zu einer sehr hohen Durchschnittsbewertung von 66,1 führt. Durch Doppelklicken auf die Statistik kann man die betroffenen Pakete in einer Liste sehen, wobei die Größe des zugehörigen Zyklus angegeben ist. Der größte Zyklus enthält 29 Pakete, es gibt weitere Zyklen mit 15, 13 und 5 sowie zweimal 3 und dreimal 2 Paketen. Klickt man doppelt auf ein solches Paket, öffnet sich der *Package Graph View* mit dem zugehörigen Zyklus.

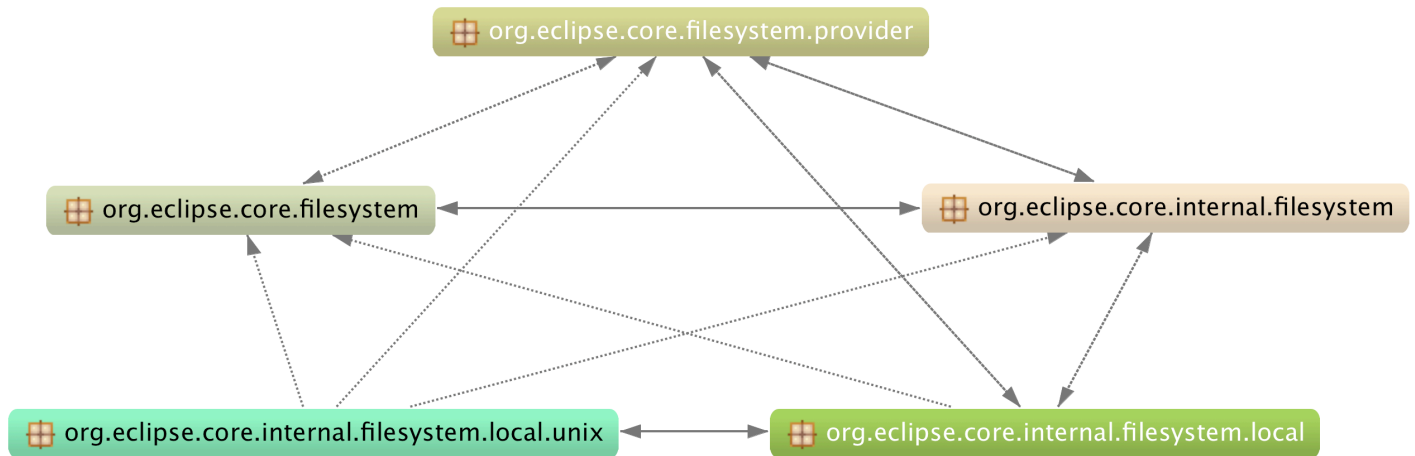


Abb. 2: Paketzyklus mit 5 Paketen

In dem Zyklus mit 5 Paketen (Abb. 2) sind fast alle der Kanten zwischen den Paketen bidirektional. Um herauszufinden, welche Klassen für diese Beziehungen verantwortlich sind, kann man Pakete und/oder Beziehungen auswählen und alle damit zusammenhängenden Klassen durch Klicken auf das Klassensymbol im Toolbar im *Class Graph* anzeigen lassen.

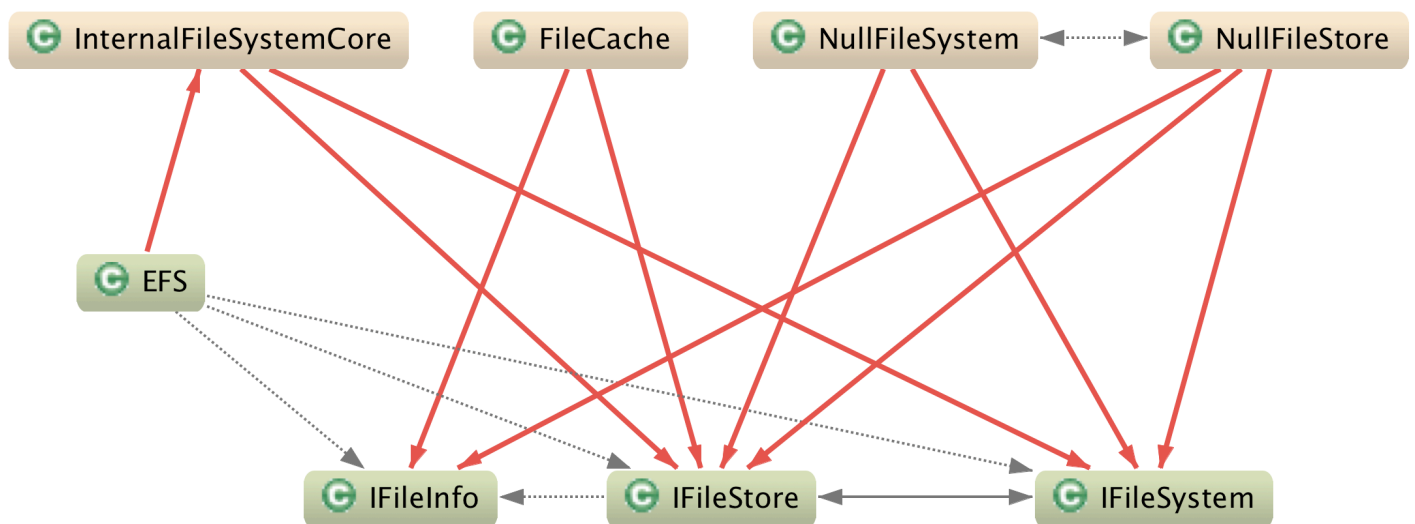


Abb. 3: Klassen aus `org.eclipse.core.filesystem` und `org.eclipse.core.internal.filesystem`

Wählen wir die Kante zwischen `org.eclipse.core.filesystem` und `org.eclipse.core.internal.filesystem` aus, ergibt sich das Bild in Abb. 3. Hierbei stellen die roten Kanten paketübergreifende Klassenbeziehungen dar, während Beziehungen innerhalb eines Pakets schwarz dargestellt sind. In unserem Beispiel referenziert die öffentliche Klasse `EFS` interne Klassen, die wiederum Bezug auf öffentliche Interfaces im Paket der Klasse `EFS` nehmen. Dieser Zyklus lässt sich durch Verschieben der Klasse `EFS` in ein anderes Paket auflösen. Der Graph wird sofort automatisch aktualisiert.

Betrachtet man große Paketzyklen, ist es schwer, auf einen Blick die relevanten Zusammenhänge zu erfassen. Daher kann man einzelne Pakete aus dem View herausfiltern, indem man sie markiert und

durch Klick auf das graue Kreuz im Toolbar ausblendet. So lassen sich beliebig viele Pakete in einem oder mehreren Schritten aus der Ansicht entfernen. Zum Entfernen des Filters klickt man auf den Radiergummi.

## Ausblick

Der Rechenkern von Usus ist fertiggestellt und arbeitet stabil, die Graphen bieten viel Komfort beim Bearbeiten von Code. Usus ist durch eigene Metriken und Statistiken flexibel erweiterbar, wofür Extension Points zur Benutzung bereitstehen.

Für die Zukunft sind weitere Entwicklungen geplant. Interessant sind beispielsweise “Schlammloch”-Hotspots, also solche Stellen im Code, die nach Aggregation mehrerer Einzelmetriken einen besonders großen Bedarf für ein Refactoring aufweisen. Ein Schlammloch ist also ein Ort, an dem es sich besonders lohnt aufzuräumen! Außerdem möchten wir mit Usus gezielt nach Code Smells wie z. B. Feature Envy oder Data Classes suchen und an den zugehörigen Hotspots am besten gleich das passende Refactoring anbieten.

Es bleibt spannend! Doch wir können beim Entwickeln von Usus auf Usus vertrauen, viel kann also nicht schiefgehen.

---

## Literatur

1. Martin Fowler - Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman, 1999
2. Robert C. Martin - Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008
3. [CCD](#)
4. [Checkstyle](#)
5. [FindBugs](#)
6. [PMD](#)
7. [Sonar](#)
8. [UsusUpdateSite](#)