Einsicht und Handeln mit Usus

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 wenn man Erkenntnisse gewonnen hat, wie reagiert man darauf und was kann man zur Verbesserung unternehmen? 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 Lesen und Verstehen einfach wird. Die Änderbarkeit steht als weitere wichtige Eigenschaft des Codes im direkten Zusammenhang mit der Nachvollziehbarkeit: Man sollte nur Code ändern, den man versteht! Usus prüft verschiedene Aspekte des Codes, die in unseren Augen essentiell für eine gute Nachvollziehbarkeit und damit eine leichte 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: Aufräumen braucht Zeit, die man besser zum Codieren verwenden kann; das Projekt ist doch bald zu Ende; alle Entwickler kennen ohnehin den ganzen Code; neue Teammitglieder wird es nicht geben; und außerdem wird die Software sowieso nicht lange leben, keine neuen Features bekommen und nicht gewartet werden müssen, denn sie hat ja keine Bugs. Unsere Erfahrungen in Legacy-Projekten lehren uns, dass diese Annahmen ins Reich der Entwicklermärchen gehören. Software-"Zwischenlösungen", die seit vielen Jahren weiterentwickelt werden und inzwischen ihren Funktionsumfang vervielfacht haben, scheinen eher an der Tagesordnung zu sein.

Warum ein Analysewerkzeug? Analysewerkzeuge unterstützen den Entwickler dabei, Fehler und negative Entwicklungen frühzeitig zu erkennen. So kann man daraus lernen und derartige Probleme im weiteren Verlauf des Projekts vermeiden. Des Weiteren helfen Analysewerkzeuge dabei, die vielfältigen Meinungen dazu, was lesbaren und verständlichen Code ausmacht, zu vereinheitlichen. So legt jedes Mitglied eines Teams dieselben Regeln für seinen Code zugrunde, genauso wie dies bei Formatierungsregeln geschieht.

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 noch ein Analysewerkzeug?

Es gibt viele Code-Analysewerkzeuge, 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? In erster Linie liegt dies daran, dass die existierenden Tools unsere Bedürfnisse in der einen oder anderen Hinsicht nicht erfüllen.

Ungeeignete Standardeinstellungen. Vor allem die Grundeinstellungen von Checkstyle sind in unseren Augen nicht hilfreich, um gut lesbaren Code zu erzeugen. Beispielsweise ist die Maximallänge von Methoden auf 150 Zeilen voreingestellt; uns schwebt dagegen eine Methodenlänge von 15 Zeilen vor. Das gleiche gilt für Einstellungen, die das Fehlen von Javadoc bemängeln; wir würden im Normalfall eher das Vorhandensein von Javadoc bemängeln.

Warnings, so weit das Auge reicht. Einige Tools sind in ihren Standardeinstellungen so scharf gestellt

und berücksichtigen so viele relativ unwichtige Probleme, dass man förmlich in Warnings ertrinkt. Mit Ausnahme von FindBugs ist auf den ersten Blick nicht klar, welche Priorität einzelne Warnings haben und wie man ohne große Mühe alle Warnings einer Art im gesamten Workspace identifizieren und bearbeiten kann.

Warnings ohne Hilfestellung. Warnings sind nur dann hilfreich, wenn sie behoben werden können. Checkstyle und FindBugs bieten in einigen Fällen zwar die Möglichkeit eine Warning per *Quick Fix* zu beheben, allerdings fällt es bei vielen der standardmäßig eingestellten Überprüfungen auf den ersten Blick schwer, ihren Hintergrund zu erfassen und ihre Relevanz im Verhältnis zu den anderen Warnungen zu beurteilen.

Ausschließlich dateilokale Checks. Die Überprüfungen aller drei Tools beschränken sich auf die Auswertung von Regeln auf Dateiebene. Sie machen keine Aussage über die Gesamtstruktur bzw. Architektur des Systems.

Veränderungen sind Geheimsache. Keines der oben genannten Werkzeuge bietet einen unmittelbaren Überblick darüber, ob mein aktuelles Tun gerade die Qualität der Codebasis verbessert oder verschlechtert.

Metriken und Statistiken

In Usus unterscheiden wir zwischen *Metriken* und *Statistiken*. Metriken führen eine statische Analyse des vorliegenden Codes durch und extrahieren verschiedene den Code beschreibende Werte, während Statistiken die Resultate der Metriken zusammenfassen und bewerten. Diese beiden Schritte sind voneinander getrennt. Insbesondere ist es möglich, mehrere Statistiken zu implementieren, die auf denselben Metrikwerten operieren. So kann Usus an verschiedene Einsatzsituationen angepasst werden, z. B. durch das Zuschneiden auf unterschiedliche Qualitätsniveaus des zu untersuchenden Codes oder indem eine Statistik die Werte mehrerer Metriken gemeinsam betrachtet.

Die Usus-Metriken analysieren entweder den Inhalt eines einzelnen Source-Files oder die Relationen zwischen mehreren Source-Files. Daher unterscheiden wir zwischen dateilokalen (D) und relationenzentrierten (R) Metriken. Für jede Metrik geben wir an, zu welcher Kategorie sie gehört.

Methodenlänge (D): Diese Metrik bestimmt die Anzahl der Statements in jedem Methodenrumpf. Dieser Wert ist sehr ähnlich zur Anzahl der Codezeilen, ignoriert aber beispielsweise Leerzeilen, Kommentare und einzelne Klammern.

Die zugehörige Statistik arbeitet linear. Methoden mit einer Länge von 9 oder weniger werden mit 0 bewertet. Längere Methoden werden gemäß der Funktion $f(x) = 1/9\ x - 1$ bewertet. So bekommen sehr lange Methoden eine höhere Bewertung als solche, die nur knapp über dem Limit liegen. Wir unterscheiden zwischen dem Wert, der einer Methode durch die Metrik zugeordnet wird, und der Bewertung, die die Statistik aus diesem Wert aufgrund bestimmter Kriterien ermittelt. In den Hotspots werden die Werte angezeigt. Die Summe aller Bewertungen wird durch die Gesamtzahl aller Methoden geteilt, um das Statistik-Level zu ermitteln.

Zyklomatische Komplexität (D): Diese Metrik bestimmt die Anzahl der möglichen Verzweigungen im Ausführungspfad durch einen Methodenrumpf. Ein leerer Methodenrumpf hat hierbei eine zyklomatische Komplexität von 1. Jedes Vorkommen eines verzweigenden Sprachelements, z. B. *if*, *while*, *catch* oder die Operatoren && und ||, erhöht diesen Wert um 1. Die zugehörige Statistik wertet Methoden mit einer zyklomatischen Komplexität von 5 oder mehr mit $f(x) = 1/4\ x - 1$.

Klassengröße (D): Diese Metrik bestimmt die Anzahl von statischen und nichtstatischen Methoden und Initializern in einer Klasse. Die Methodensichtbarkeit wird hierbei nicht berücksichtigt. Die zugehörige Statistik wertet Klassen mit einer Methodenanzahl von 13 oder mehr mit $f(x) = 1/12\ x - 1$.

Average Component Dependency (R): Die hier zugrundeliegende Metrik heißt *Cumulative Component Dependency* (CCD). Sie ermittelt für jede Klasse, wie viele andere Klassen sie kennt. Die zugehörige Statistik (ACD) gibt den durchschnittlichen CCD aller Klassen in Prozent an. Ihr Grenzwert hängt von der Projektgröße ab: Für kleine Projekte ist ein ACD von 15 % noch ok, während große Projekte einen Wert von 5 % nicht überschreiten sollten. Als Statistiklevel wird 100 - ACD verwendet, damit alle Levelangaben dieselbe Tendenz haben. In den Hotspots wird für jede Klasse ihr CCD angegeben.

Paketzyklen (R): Die zugrundeliegende Metrik analysiert die Beziehungen zwischen Klassen. In der Paketzyklen-Statistik werden diese Klassenbeziehungen auf die beteiligten Pakete reduziert und auf Zyklen untersucht. In Abb. 1 sind beispielhaft vier Klassen dargestellt, wobei die gleichfarbigen jeweils im selben Paket liegen. Die Beziehungen zwischen den Klassen bilden keinen Zyklus, auf Paketebene gibt es jedoch eine zyklische Abhängigkeit. Eine solche Abhängigkeit entsteht, wenn Klassen nicht in den richtigen Paketen liegen, und deutet auf Probleme im Design und in der Strukturierung des Projekts hin.

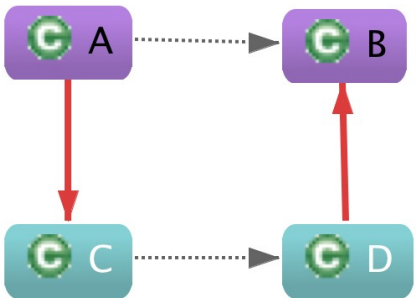

Abb. 1: Paketzyklus

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, mit installiertem Usus [7].

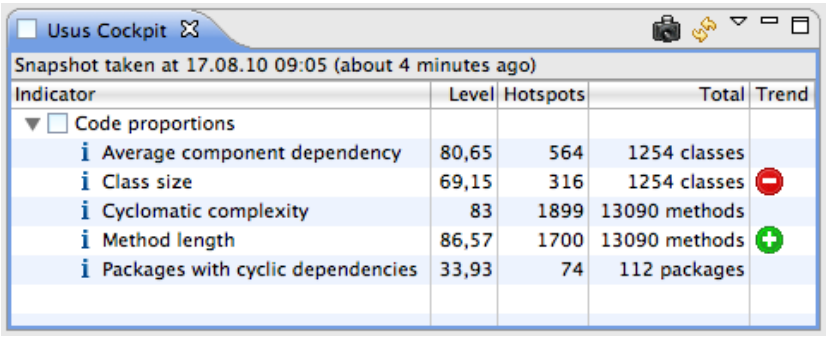

Abb. 2: Das Usus-Cockpit

Cockpit. Das Cockpit (Abb. 2) ist der zentrale View von Usus. Hier erhält man 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 Gesamtbewertung der Statistik (*Level*) angezeigt. In der *Trend*-Spalte wird die Veränderung seit dem letzten *Snapshot* angezeigt. Snapshots (über den entsprechenden Toolbar-Button) ermöglichen es, Veränderungen im Code sichtbar zu machen. Beispielsweise ist es ratsam einen Snapshot vor einem größeren Refactoring oder vor Beginn eines Implementierungstasks anzulegen.

Hotspots. Das Motto von Usus lautet "Einsicht und Handeln". Das Cockpit erlaubt zwar Einsicht in den aktuellen Zustand des Codes, aber zum Handeln genügt es nicht. Deshalb gelangt man von jeder Statistik

durch einen Doppelklick zum *Hotspots View*, der die Elemente mit einer Bewertung größer als 0 anzeigt. Bei dateilokalen Statistiken gelangt man von dort per Doppelklick zum entsprechenden Element im *Java Editor*, bei Paketzyklen öffnet sich der *Package Graph View*. Wie im Cockpit kann man auch hier den Trend seit dem letzten Snapshot im Auge behalten.

Anhand der Methodenlänge und der Paketzyklen zeigen wir im Folgenden, wie Usus Schwachstellen im Code aufzeigt und wie man diese beheben kann. Um die anderen Metriken bzw. Statistiken zu verbessern, geht man analog vor.

Methodenlänge

Der Ermittlung der Methodenlänge liegt die Gesamtzahl von 13089 Methoden zugrunde. Von diesen haben 1700 Methoden mehr als 9 Statements. Dies ergibt ein Level von 86,56. Ein Doppelklick auf die Cockpitzeile öffnet den Hotspots View und zeigt diese Methoden an: Die Methode *FrameworkCommandProvider._bundle()* ist mit 164 Statements mit Abstand am größten. Hier können wir hoffentlich einiges zur Verbesserung der Codequalität beitragen -- also los!

Per Doppelklick auf diesen Hotspot gelangen wir zur Methode im Java Editor. Die Methode erzeugt detaillierte Informationen zu einem Bundle und übergibt diese an einen *CommandInterpreter*. Nach kurzem Studium des Codes erkennt man mehrere duplizierte Codeblöcke, die jeweils dieselbe Information an den *CommandInterpreter* weitergeben. Hier lässt sich durch das Extrahieren von Methoden mehr Struktur hineinbringen, die Lesbarkeit und Übersichtlichkeit erhöhen und -- quasi als Nebeneffekt -- die Methode kürzen.

Nach diesem Editiervorgang sollte der betreffende Hotspot mit einem grünen Pluszeichen markiert worden sein, um eine Verbesserung anzuzeigen. Trotzdem kann es sein, dass sich der Gesamttrend der Methodenlängen-Statistik verschlechtert hat, zu erkennen an einem roten Minuszeichen im Cockpit. Dies passiert, wenn neue Hotspots entstanden sind, z. B. beim Extrahieren zu langer Methoden, denn der Gesamttrend wirkt als Frühwarnsystem und verschlechtert sich, sobald sich auch nur ein Hotspot verschlechtert hat.

Paketzyklen

Von den 112 analysierten Paketen befinden sich 74 in einem Paketzyklus, was zu einem sehr niedrigen Level von 33,93 führt. Durch Doppelklicken auf die Statistik kann man die betroffenen Pakete in einer Liste sehen, wobei die Größe des Zyklus, der das Paket enthält, 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.

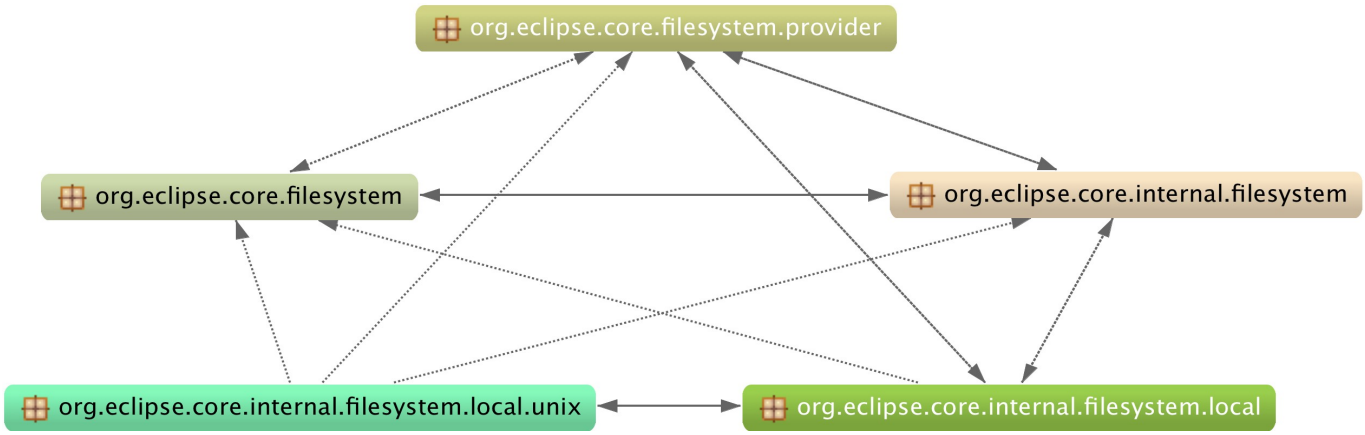

Abb. 3: Paketzyklus mit 5 Paketen

Betrachten wir zum Beispiel den Zyklus mit 5 Paketen (Abb. 3): Fast alle der Kanten zwischen den Paketen sind 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.

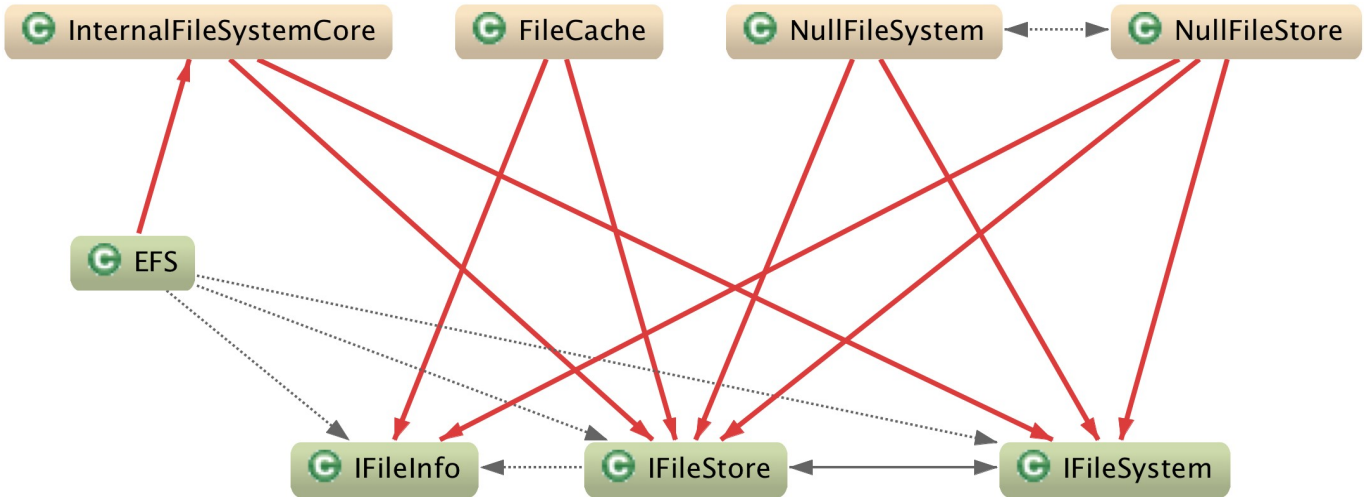

Abb. 4: Klassen aus org.eclipse.core.filesystem und org.eclipse.core.internal.filesystem

Wählen wir beispielsweise die Kante zwischen *org.eclipse.core.filesystem* und *org.eclipse.core.internal.filesystem* aus, ergibt sich das Bild in Abb. 4. 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.

Betrachtet man größere Paketzyklen, beispielsweise den größten mit 29 Paketen, 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 Aufheben des Filters klickt man auf den Radiergummi.

Ausblick

Bislang haben wir uns darauf konzentriert, den Rechenkern von Usus rund zu machen. Diese Arbeiten stehen nun kurz vor dem Abschluss. Es ist jetzt möglich, Usus mit eigenen (dateilokalen) Statistiken zu erweitern; dazu muss lediglich ein Extension Point benutzt werden. Als nächster Schritt steht die Erweiterbarkeit durch eigene Metriken an.

Darüberhinaus sind weitere Entwicklungen geplant. Interessant wären 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 schief gehen.

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. Usus