KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

06/2009

Supergenau

Messen objektorientierter Eigenschaften

von Thomas Haug

Copyright © 2009 Bookware 1865-682x/09/06/001

Supergenau

Messen objektorientierter Eigenschaften

von Thomas Haug

In der letzten Ausgabe[1] wurden die Grundzüge des Werkzeugs iPlasma[2] gezeigt und hierbei die Metriken zur Bewertung der Größe und Komplexität eines objekt-orientieren Systems erläutert. In diesem Artikel werden Metriken zur Bestimmung des Vererbungs- und Kopplungsverhalten eines Systems vorgestellt und beispielhaft angewendet.

Bereits in [1] wurde der *Pyramidal View* eines Systems gezeigt und Metriken, die die Größe und die Komplexität eines Systems bewerten, vorgestellt (Abbildung 1):

- NOP Number of Packages
- NOC Number of Classes
- NOM Number of Method
- LOC Lines of Code
- CYCLO zyklomatische Komplexität

Ein objekt-orientiertes System alleine an seiner Größe und Komplexität zu bewerten ist nicht ausreichend, denn zwei wesentliche Aspekte der Objekt-Orientierung werden hierbei außer Acht gelassen: Vererbung und die Kopplung der Klassen.

Lanza und Marinescu (LM) beschreiben in ihrem Buch *Object-Oriented Metrics in Practice* [3] Metriken, die diese Aspekte für ein Gesamtsystem beurteilen.

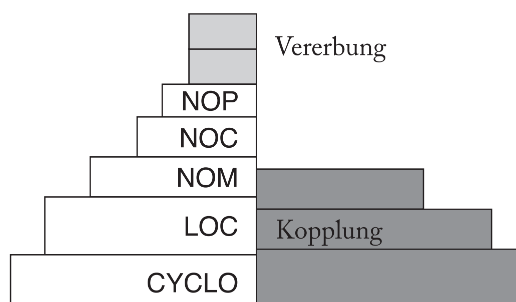

Abbildung 1: *Pyramidal View*

Kopplungsmetriken

Die Metrik Number of Operation Calls (CALLS) misst die Anzahl der „eindeutigen" Methodenaufrufe im System:

$$\text{CALLS} = \sum_{\text{alle Methoden } m} \text{eindeutigeAufrufe}(m)$$

Wobei die Funktion *eindeutigeAufrufe* wie folgt definiert ist:

> *Wird eine Methode von einer anderen mehrfach aufgerufen, so wird dieses Vorkommen lediglich einmal gezählt. Wird eine Methode von verschiedenen Methoden aufgerufen, so wird jeder Aufruf gezählt.*

Hat man nun diese Zahl für ein System ermittelt, kann man die Intensität der Kopplung im System bewerten, in dem man folgenden Quotienten bildet:

$$\frac{CALLS}{NOM}$$

Die Kopplungsintensität gibt Auskunft, wie viele Methodenaufrufe im Durchschnitt beim Ausführen einer Methode beteiligt sind. In Artikel [1] wurde zum Beispiel der Wert 3,45 für das Spring Framework in der Version 2.5.1 ermittelt. Dies bedeutet, dass beim Aufruf einer Methode durchschnittlich 3,45 andere Methoden beteiligt sind.

Sehr hohe Werte bei der Kopplungsintensität können auf eine zu hohe Verflechtung der Klassen des bewerteten Systems hinweisen. Dies wiederum kann ein Indiz für hohe Wartungsaufwände bzw. erschwerte Weiterentwicklung sein.

Als weitere Kennzahl nutzen Lanza und Marinecsu die Metrik FANOUT_{LK}[1], die von Lorenz und Kidd in [4] beschrieben wurde. Diese Metrik misst die Beteiligung von Klassen an einem Methodenaufruf, d. h. es wird für einen Methodenaufruf abgezählt, mit wie viel anderen Klassen innerhalb des Aufrufs zusammengearbeitet wird. Da im *Pyramidal View* ein Gesamtsystem bewertet wird, summiert man die berechneten FANOUT_{LK}-Werte aller Klassen und erhält damit den FANOUT-Wert nach Lanza und Marinescu:

$$\text{FANOUT} = \sum_{\text{alle Methoden } m} \text{FANOUT}_{LK}(m)$$

Dieses Maß soll Auskunft darüber geben, wie stark sich Methodenaufrufe auf das Gesamtsystem ausbreiten.

[1] Laut [3] wird von Lorenz und Kidd die Abkürzung FANOUT verwendet. Da aber diese Abkürzung in [1] und [3] für die Metrik-Variante von Lanza und Marinescu verwendet wird, wird in diesem Artikel die Abkürzung FANOUT_{LK} eingeführt.

Der Quotient

$$\frac{FANOUT}{CALLS}$$

beschreibt die Kopplungsausbreitung, also wie viele andere Klassen durchschnittlich an einem Methodenaufruf einer Klasse beteiligt sind. Der Wert 0,5 besagt beispielsweise, dass jeder zweite Methodenaufruf eine andere Klasse aufruft.

Wie bereits in [1] beschrieben, wurden in [3] 45 Java-Projekte und 37 C++-Projekte analysiert und für die oben genannten Quotienten Schwellwerte ermittelt. Die folgende Tabelle enthält die Schwellwerte, die aufgrund der Analyse der Java-Projekte erhoben wurden:

Metrik	Niedrig	Durchschnitt	Hoch
$\frac{CALLS}{NOM}$	2,01	2,62	3,2
$\frac{FANOUT}{CALLS}$	0,56	0,62	0,68

Tabelle 1: Schwellwerte für Quotienten in Java Projekten

Vererbungsmetriken

Zur Beurteilung des Einsatz von Vererbung in dem zu bewertenden System werden in [3] zwei Metriken vorgeschlagen. Diese Metriken messen sowohl die Vererbungsbreite (d. h. wie viele Klassen werden von einer Klasse direkt abgeleitet) als auch die Vererbungstiefe (d. h. wie lang ist der Pfad von einer Basisklasse zu der am tiefsten gelegenen Klasse).

Die Metrik *Average Number of Derived Classes* (ANDC)[2] misst die Anzahl direkt abgeleiteter Klassen im Gesamtsystem und soll laut Lanza und Marinscu Auskunft geben, wie oft Abstraktionen von Oberklassen durch abgeleitete Klassen verfeinert werden. Hierzu wird die Chidamber-Kemerer-Metrik *Number of Children* (NOC_{CK})[3] verwendet (siehe [5]). Die Metrik ANDC berechnet die Summe der NOC_{CK}-Werte für alle Klassen des Systems und teilt diesen Wert durch die Anzahl der Klassen. Interfaces werden hierbei nicht gezählt:

$$ANDC = \frac{\sum_{\text{alle Klassen } k} NOC_{CK}(k)}{\text{Anzahl der Klassen}}$$

[2] iPlasma verwendet abweichend den Begriff *Number of Direct Descendants* (NDD).

[3] Diese Metrik wird eigentlichen mit NOC abgekürzt, da dieses Kürzel aber mit der von Lanza und Marinescu verwendeten Metrik *Number of Classes* kollidiert, wird im Folgenden die Abkürzung NOC_{CK} verwendet. Die Benennung in [3] verletzt leider deutlich die Forderung nach Eindeutigkeit von Metriken (siehe [5]), denn die Namen sollten unbedingt eindeutig bleiben, um Verwirrungen und Missinterpretationen zu vermeiden.

Sehr breite Vererbungshierarchien können ein Indiz für das Anti-Pattern Copy-Paste(-And-Adapt) sein, wenn Subklassen durch Kopieren und Anpassen bestehender Subklassen entstehen.

In der folgenden Abbildung ist beispielhaft eine Vererbungshierarchie mit zugehörigen NOC_{CK}-Werten unterhalb der Klassennamen (C1 bis C8) gezeigt.

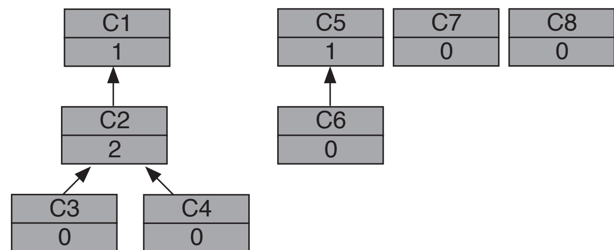

Abbildung 2: NOC_{CK} Werte

Somit ergibt sich für die ANDC-Metrik der folgende Wert:

$$ANDC = \frac{(5 * 0 + 1 + 2 + 1)}{8} = 0{,}5$$

Dies bedeutet, dass durchschnittliche jede zweite Klasse eine direkt abgeleitete Klasse besitzt.

Als weitere Metrik kann man die durchschnittliche Tiefe des Vererbungsbaums des Systems berechnen. Diese Metrik wird in [3] als *Average Hierachy Height* (AHH) berechnet. Hierzu wird zuerst die Menge der sogenannten *Root Classes* bestimmt. Hierbei handelt es sich um Klassen, die sich entweder von einer allen Klassen gemeinsamen Oberklasse (z. B. java.lang.Object), einem Interface oder einer Bibliotheksklasse (also einer Klasse außerhalb des zu bewertenden Systems) ableiten.

Hat man diese Menge ermittelt, wird für jede enthaltene Klasse die maximale Vererbungstiefe bestimmt. Diese Tiefe wird als *Height of Inheritance Tree* (HIT) bezeichnet. Die bestimmten Tiefen werden summiert und durch die Anzahl der Basisklassen geteilt:

$$AHH = \frac{\sum_{\text{alle Basisklassen } k} HIT(k)}{\text{Anzahl der Basisklassen}}$$

Der so ermittelte Wert gibt Aufschluss darüber, ob das analysierte System zu flachen oder tiefen Vererbungshierarchien neigt. Eine tiefe Vererbungshierarchie bedeutet zwar einen höheren Anteil an Wiederverwendung, aber die Verständlichkeit eines Systems kann darunter leiden, wenn Vererbung allzu leichtfertig eingesetzt wird.

Wie bei der Bewertung der Größe bzw. Komplexität und Kopplung des Systems wurden in [3] Schwellwerte für die beiden Vererbungsmetriken erhoben. Diese sind in Tabelle 2 dargestellt.

Metrik	Niedrig	Durchschnitt	Hoch
ANDC	0,25	0,41	0,57
AHH	0,09	0,21	0,32

Tabelle 2: Schwellwerte der Vererbungsmetriken in Java-Projekten

Überblick

Nachdem nun alle drei Dimensionen – Größe/Komplexität, Kopplung und Vererbung – durch die Metriken von Lanza und Marinescu beschrieben sind, kann man den *Pyramidal Overview* vollständig interpretieren. In Abbildung 3 ist das Open-Source-Projekt Tomcat in der Version 6.0.20 mittels dem Werkzeug inCode [7], einem Eclipse Plug-in, analysiert worden. Neben der grafischen Darstellung des *Pyramidal Views* werden die gemessenen Werte interpretiert. So wird z. B. die Vererbungsbreite und -tiefe als durchschnittlich erachtet, während die Methoden als überdurchschnittlich lang eingestuft und die Anzahl der Methoden pro Klasse als zu hoch betrachtet werden.

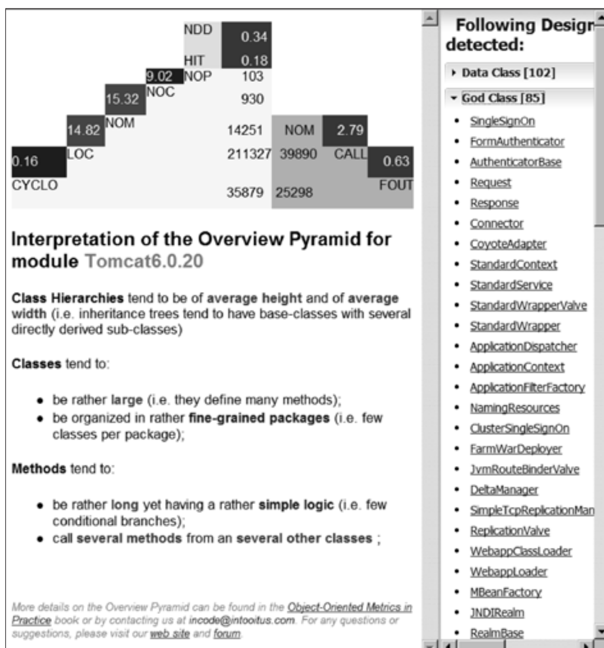

Abbildung 3: Tomcat 6.0.20 mittels inCode analysiert

Der *Pyramidal Overview* kann, wie sein Name bereits anklingen lässt, lediglich einen Überblick über das zu bewertende System geben: ob dieses im Durchschnitt liegt oder ob es offensichtliche „Ausreißer“ enthält. Dieser Überblick kann ggf. helfen zu bewerten, welche Aufwände wohl für eine Weiterentwicklung bzw. Refaktorierung des Systems anfallen werden. So ist es offensichtlich, dass bei Systemen, die in einigen Metriken über dem Durchschnitt liegen, wohl höhere Aufwände anfallen werden, als bei Systemen, die im Durchschnitt liegen, denn es ist mit höherer Komplexität zu rechnen. Wichtig ist aber festzuhalten, dass dieser Überblick nur ein grober Anhaltspunkt über die Strukturierung des Systems darstellt.

Tiefenbohrung

Um eine genauere Bewertung des Systems durchzuführen, muss zwangsläufig die Oberfläche verlassen werden und in die Tiefe des Systems abgestiegen werden. Sicherlich helfen hierbei die oben genannten Metriken potentielle Kandidaten für eine genauere Betrachtung herauszufinden, doch sie werden nicht ausreichen. In [3] wird aus diesem Grund auf bestehende objektorientierte Metriken[4] zurückgegriffen. Beispielsweise können die bereits erwähnten Metriken von Chidamber und Kemerer hierbei wertvolle Dienste leisten. Des Weiteren definieren Lanza und Marinescu noch eigene Metriken, um spezielle Eigenschaften an dem zu untersuchenden System herauszustellen. So misst die Metrik *Access To Foreign Data* (ATFD) an einer Klasse den direkten Zugriff auf Attribute fremder Klassen.

Die Autoren von [3] gehen in der Verwendung von Metriken noch einen Schritt weiter. Sie definieren eine Methodik, um Metriken mit Boole'schen Ausdrücken zu kombinieren und formulieren anhand dieser zusammengesetzten Metriken Strategien, gerade die Klassen zu identifizieren, die potentielle Design-Schwächen aufweisen. [3] unterscheidet dabei drei Kategorien von Schwächen („*disharmonies*“):

- Identitätsschwächen (identity),
- Kollaborationsprobleme (collaboration),
- Vererbungsschwächen (classification).

Identitätsschwächen charakterisieren Probleme die sich zumeist nur auf diese Entität beziehen und isoliert betrachtet werden können. Beispielsweise wird das Identitätsproblem *Feature Envy* (Neid) beschrieben, das nach Fowler ein typischer „Code-Smell“ ist, wenn Klassen mehr an Daten anderer Klassen interessiert sind als an den eigenen [8].

[4] Laut [5] gab es eine Erhebung, wie viele objektorientierte Metriken existieren. Zur Zeit dieser Erhebung existierten ca. 375 verschiedene Metriken!

Kollaborationsprobleme betreffen mehrere Entitäten und können oftmals nur im Verbund gelöst werden, z.B. der in [8] beschriebene Gestank *Shotgun Surgery* (Schrottkugeln herausoperieren) wird von [3] als ein typisches Kollaborationsproblem beschrieben, da in diesem Fall Änderungen an der einen Methode oder Klasse viele Änderungen an anderen Klassen zur Folge haben.

Vererbungsschwächen versuchen Design-Probleme in Vererbungshierarchien zu charakterisieren. So soll die Ermittlungsstrategie *Refused Parent Bequest* das *Code-Smell Ausgeschlagene Erbe* nach Fowler aufdecken, wenn Subklassen die geerbten Methoden und Attribute nicht nutzen.

Im Folgenden wird exemplarisch gezeigt, wie man die Identitätsschwäche *God Class* durch Kombination von Metriken definieren und in einem System entdecken kann. Eine *God Class* versucht viel, wenn nicht gar alle Arbeit in einem (Sub-)System zu erledigen. Hierbei nutzt sie üblicherweise Daten anderer Klassen und delegiert lediglich kleine Aufgabe an andere Klassen. D.h. die Komplexität einer solchen Klasse ist oft sehr hoch, doch die Kohäsion einer solchen Klasse ist auch oftmals gering, da sie zu viele unterschiedliche Verantwortlichkeiten wahrnimmt. Um solche Klassen identifizieren zu können, kann man die in Abbildung 4 skizzierten Metrik-Kombination nutzen.

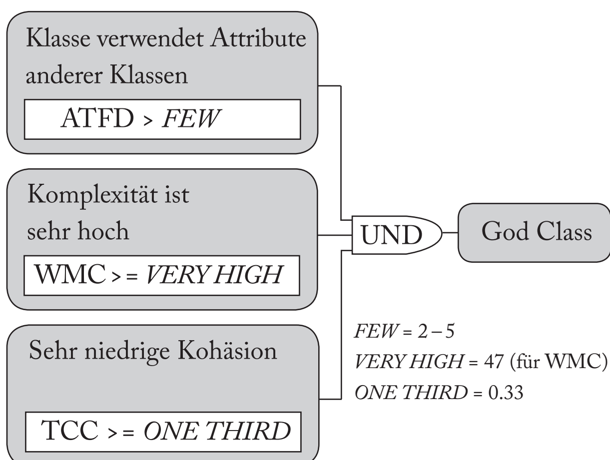

Abbildung 4: *God Class*

Zuerst errechnet man für die potentielle *God Class* die ATFD Metrik (siehe oben). Werden mehr als eine handvoll, – d.h. zwei bis fünf – Attribute fremder Klassen[5] gefunden, so ist dieses Filterkriterium gegeben.

Als nächstes kann man die Metrik *Weighted Method Count* (WMC) nach Chidamber und Kemerer als Filterkriterium nutzen. Die WMC-Metrik berechnet anhand einer Komplexitätsmetrik, zum Beispiel nach McCabe (siehe [1]) die Summe der Komplexitäten einer Klasse.

$$\text{WMC}\ (\textit{Klasse}) = \sum_{\textit{alle Methoden } m} \text{CC}\ (m)$$

Ist der WMC-Wert der Klasse *sehr hoch* (laut [3] in Java Projekten bei einem Wert über 47), so ist auch dieses Filterkriterium erfüllt.

Die dritte Metrik, die zur Beurteilung einer *God Class* herangezogen wird, ist *Tight Class Cohesion* (TCC), die von Bieman und Kang definiert wurde:

$$\textit{TCC (Klasse)} = \frac{\text{Anzahl verbundener Methoden}}{\text{Gesamtanzahl Methoden}}$$

Zwei Methoden sind miteinander verbunden, wenn sie auf das selbe Attribut zugreifen. Ein empirisch gewonnener Wert besagt, dass ein Wert von 0,5 für eine Klasse nicht unterschritten werden sollte, denn in diesem Fall würde jede zweite Methode auf einem anderen Attribut arbeiten. [3] ist an dieser Stelle weniger restriktiv und legt den TCC Wert auf ein Drittel fest.

Die Anwendung der oben gezeigten Metrikkombination auf das Tomcat Projekt kann mittels iPlasma durchgeführt werden. Dieses Werkzeug kennt bereits diese Erkennungsstrategie und kann somit eigentlich ohne Kenntnis der Filterkriterien gewählt werden.[6] Für Tomcat werden 109 Klassen gefunden, die diese Filter erfüllen. Im dargestellten Screenshot (Abbildung 5) wurden die verwendeten Metriken ATFD, WMC, TCC und zusätzlich die Metrik LOCC (*Lines of Code per Class*) in der rechten Tabelle angezeigt, um die wesentlichen Merkmale der gefundenen Klassen aufzuzeigen. Hierbei wurde die Sortierung der Klassen an der WMC-Spalte vorgenommen, da die WMC-Metrik am Besten Aufschluss über die Komplexität der Klasse gibt und somit die Klasse mit der höchsten Komplexität in den Vordergrund tritt.

In der unteren Hälfte des iPlasma-Screenshots sind die Details der ausgewählte Klasse *StandardContext* dargestellt. Man kann erkennen, dass diese Klasse laut den Erkennungsstrategien von [3] mit einer Vielzahl von Design-Problemen zu kämpfen hat. Bei einer Länge von über 5000 Zeilen mit 264 Methoden und 106 Attributen ist dies kaum verwunderlich. Ob diese Indizien ausreichen eine Refaktorierung der Klasse durchzuführen, darf der Leser selbst entscheiden.

[5] Fremde Klassen bezeichnen Klassen, die nicht in der Vererbungshierarchie der zu beurteilten Klasse liegen.

[6] Eine Metrik bzw. eine Kombination von Metriken sollten nie ohne Wissen, was sie messen, eingesetzt werden.

Abbildung 5: God Class Filter in iPlasma

Fazit

In diesem Artikel wurde der *Pyramidal View* komplettiert, d. h. die Metriken vorgestellt, die Kopplung und Vererbung bewerten. Basierend auf den Metriken dieser Überblicksansicht leiten die Autoren von [3] Interpretationen über die Strukturierung eines Gesamtsystems ab. Diese reichen aber im Allgemeinen nicht aus, um Refaktorierungspotentiale aufzudecken. Um diese zu erkennen, definieren Lanza und Marinescu Ermittlungsstrategien die Design-Schwächen aufdecken sollen, indem Metriken kombiniert werden.

Referenzen

[1] Haug, T. *Messgenau*,
KaffeeKlatsch, Mai 2009

[2] LOOSE Research Group *iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design*,
http://loose.upt.ro/iplasma/index.html

[3] Lanza, M.; Marinescu, R. *Object Oriented Metrics in Practice*,
Springer, 2006

[4] Lorenz *Kidd - Object-Oriented Software Metrics,*
Prentice-Hall, 1994

[5] Spinellis, D. *Code Quality The Open Source Perspective*,
Addison-Wesley, 2006

[6] Wiedeking, M. *Messlattenzaun*,
KaffeeKlatsch, März 2009

[7] inCode,
http://www.intooitus.com/inCode.html

[8] Fowler, M. *Refactoring*,
Addison-Wesley, 2000 Kurzbiographie

Thomas Haug (thomas.haug@mathema.de) ist als Senior-Consultant und Trainer für die MATHEMA Software GmbH tätig. Seine Themenschwerpunkte umfassen die Java Standard und Enterprise Edition (Java SE und Java EE) sowie das .NET-Umfeld, insbesondere im Hinblick auf Enterprise-Anwendungen. Neben seiner Projekttätigkeit hält er Technologietrainings für MATHEMA und berät verschiedene Projekte hinsichtlich des optimalen Einsatzes von .NET- oder Java-Technologien.

Copyright © 2009 Bookware 1865-682x/09/06/001 Von diesem KaffeeKlatsch-Artikel dürfen nur dann gedruckte oder digitale Kopien im Ganzen oder in Teilen gemacht werden, wenn deren Nutzung ausschließlich privaten oder schulischen Zwecken dient. Des Weiteren dürfen jene nur dann für nicht-kommerzielle Zwecke kopiert, verteilt oder vertrieben werden, wenn diese Notiz und die vollständigen Artikelangaben der ersten Seite (Ausgabe, Autor, Titel, Untertitel) erhalten bleiben. Jede andere Art der Vervielfältigung – insbesondere die Publikation auf Servern und die Verteilung über Listen – erfordert eine spezielle Genehmigung und ist möglicherweise mit Gebühren verbunden.