KaffeeKlatsch
Das Magazin rund um Software-Entwicklung

ISSN 1865-682X
05/2009

Messgenau

Was man nicht messen kann, kann man nicht kontrollieren

von Thomas Haug

Messgenau

Was man nicht messen kann, kann man nicht kontrollieren

von Thomas Haug

Das Begleiten einer Software-Entwicklung in der Rolle des Software-Architekten erfordert es, die entstehenden Software-Artefakte kontinuierlich zu bewerten. Wie nämlich allgemein bekannt ist, hat Software und die zugrunde liegende Software-Architektur oft die unangenehme Eigenschaft über die Iterationen und Inkremente hinweg zu degenerieren. Deshalb werden in diesem Artikel Metriken vorgestellt und deren Interpretationsmöglichkeiten beleuchtet, die dabei helfen, diesen Verfall aufzuhalten.

Ist das Kind schon in den Brunnen gefallen – die Software bereits in einem Zustand, in dem die Weiterentwicklung immer mühseliger wird –, so wird man kaum um eine eingehende Begutachtung und ggf. Refaktorierung des Systems kommen. Allerdings stellt sich einem die Frage, wie man an diese Aufgabe herantritt.

Eine Möglichkeit wäre es, per Zufallsprinzip Klassen aus dem System herauszupicken und diese nach eigenen Design-Vorstellungen bzw. Architekturvorgaben zu ändern. Dieses Vorgehen ist aber nicht sehr zielführend, denn die Wahrscheinlichkeit unangenehme *Design-* und *Code-Gerüche* [1] zu finden, dürfte einem großen Software-System verschwindend gering sein.

Metriken können in dieser Situationen helfen, eine Software zu bewerten. Erste Indizien können Basismetriken liefern, die einfach zu ermittelnde Kennzahlen über ein System liefern. Zur genaueren Betrachtung der Metriken wird das Werkzeug iPlasma [2] eingesetzt, das sowohl Java- als auch C++-Artefakte analysieren kann. An verschiedenen Stellen des Artikels werden zusätzlich Hinweise über die Tools NDepend [3], JavaNCSS [4] und PMD [5] gegeben.

Grundlegende Metriken

Mit den folgenden einfachen Metriken lassen sich erste Aussagen über ein System treffen.

- *Number of Packages* (NOP) berechnet die Anzahl der Packages (in Java) bzw. Namespaces (in C++ und C#) in dem zu bewertenden Software-System.

- *Number of Classes* (NOC)[1] zählen die Anzahl der Klassen in der Software. Hierbei werden bei iPlasma im Falle von Java sowohl Klassen, Interfaces als auch Enumerations gezählt.

- Mittels NOP und NOC lassen sich Aussagen über den Strukturierungsgrad des Softwaresystems treffen. Der Quotient

$$\frac{NOC}{NOP}$$

  liefert den Durchschnitt von Klassen pro Package bzw. Namespace des Systems. Man kann damit prüfen, ob dieser errechnete Durchschnitt der Norm entspricht, darunter oder darüber liegt. Was dabei eine mögliche Norm ist, wird im anschließenden Abschnitt erläutert.

Die nächste leicht zu ermittelnde Metrik ist die *Number of Methods* (NOM), die die Anzahl der Methoden im Gesamtsystem berechnet. Berechnet man nun das Verhältnis,

$$\frac{NOM}{NOC}$$

so erhält man die durchschnittliche Anzahl von Methoden pro Klasse. Hieraus lässt sich nun ableiten, ob das System eher wenige oder eher viele Methoden pro Klasse definiert. Zum Beispiel können sehr viele Methoden pro Klasse ein Indiz dafür sein, dass die Klassen zu viele Verantwortlichkeiten übernehmen, und man kann dies zum Anlass nehmen, darüber nachzudenken die Verantwortlichkeiten der Klassen zu prüfen.

Source Lines of Code (SLOC) liefert die Anzahl von Codezeilen in einer Methoden, einer Klasse, einem Package bzw. Namespace eines Systems. Wie in [7] beschrieben ist die stichfeste Definition einer Source-Code-Zeile nicht trivial: Zählt man Kommentar-Zeilen? Werden Zeilen, die ausschließlich eine geschweifte Klammer enthält, ignoriert? Was ist mit Zeilen, die mehrere Anweisungen enthalten? Fragen über Fragen.

[1] Die in [6] und iPlasma gewählte Abkürzung NOC ist Vorsicht zu verwenden, denn unter der Bezeichnung NOC wird üblicherweise *Number of Children* verstanden. Diese Metrik berechnet wie viele direkte bzw. indirekte Kinder eine Klasse besitzt, also wie viele abgeleitete Klassen sie hat. NDepend verwendet im Zusammenhang mit der Anzahl von Klassen den Begriff *Number of Types*, den man mit NOT abkürzen könnte und somit eine mögliche Verwechselung vermeidet würde.

Die einfachste Variante ist das schlichte Abzählen der *Lines of Code* (LOC), der physikalischen Zeilen in einer Code-Datei ohne die Kommentarzeilen mitzuzählen. Diese Art des Zählens funktioniert aber nicht bei compilierten Code und widerspricht somit der Forderung nach einer umfassenden Einsetzbarkeit der Metrik (siehe [7]), falls der Source-Code aus welchen Gründen auch immer nicht verfügbar ist. In NDepend kann die LOC-Metrik nur dann berechnet werden, wenn die entsprechenden PDB-Dateien neben den DLLs zur Verfügung stehen. Alternativ könnte man auf die Anzahl der Anweisungen für die Virtuellen Maschinen (z. B. der Intermediate Language) zurückgreifen; diese könnte aber in Abhängigkeit vom Kompilat schwanken, je nach dem, ob dieses für ein Release oder fürs Debugging erzeugt wurde.

> *Was man nicht messen kann,*
> *kann man nicht kontrollieren.*
>
> Tom DeMarco

Die zweite und schwieriger umsetzbare Variante ist das Zählen von *logischen Code-Zeilen* (lLOC). Diese Variante versucht Anweisungen zu zählen, ohne dabei die Formatierungen des Source-Codes zu berücksichtigen.

Die folgenden Code-Fragmente haben beispielsweise die gleiche Anzahl von Anweisungen, haben aber bei gleichem lLOC-Wert unterschiedliche LOC-Werte:

```
for (int i = 0; i < 10; i++)
{
  System.out.println(i + ". Ausgabe");
}
```

und

```
for (int i = 0; i < 10; i++)
  System.out.println(i + ". Ausgabe");
```

Während der lLOC-Wert bei beiden 2 beträgt, ist der LOC-Wert des ersten Fragments 4 und der des zweiten 2.

iPlasma und das zugrunde liegende Buch [6] nutzen die LOC-Metrik, d. h. es werden alle Zeilen gezählt, die Funktionalität enthalten, aber auch Zeilen die ausschließlich Klammern enthalten. Für einen ersten Überblick wird dort das Verhältnis zwischen den Code-Zeilen (LOC) und Methoden (NOM) gebildet:

$$\frac{LOC}{NOM}$$

Dieses Verhältnis beschreibt die durchschnittliche Anzahl von physikalischen Codezeilen pro Methode. Hieraus kann man ableiten, ob das begutachtete System eher zu kleinen oder zu großen Methodenrümpfen neigt. Kleine Methoden haben den Vorteil, dass sie im Allgemeinen leichter zu durchschauen sind, während große Methoden eventuell Wartungs- und Erweiterungsprobleme mit sich ziehen können.

Als letzte grundlegende Metrik wird die *zyklomatische Komplexität* (*Cyclomatic Complexity Number,* die üblicherweise mit CC oder CCN abgekürzt wird) nach McCabe berechnet [8]. Übertragen auf eine objektorientiere Sprachen berechnet diese Metrik die möglichen Ausführungspfade in einer Methode basierend auf graphentheoretischen Aspekten. Methoden, die einen hohen Komplexitäts-Wert besitzen, sind schwer zu testen und zu warten. McCabe empfiehlt Methoden, die einen Wert von 10 überschreiten zu refaktorieren. PMD definiert folgende Schwellwerte:

- 1 – 4: niedrige Komplexität
- 5 – 7: mittlere Komplexität
- 8 – 10: hohe Komplexität
- über 11 sehr hohe Komplexität

NDepend berechnet zum Beispiel die CC-Metrik auf die folgende Art: jede Methode hat grundsätzlich die Komplexität 1. Für jeden der folgenden Ausdrücke wird die Komplexität um 1 erhöht:

if
while
for
foreach
case
default
continue
goto
&&
||
catch
? : (Tenärer Operator)

Die nachfolgenden Ausdrücke erhöhen aber die Komplexität nicht:

else
do
switch
try,
using
throw
finally
return

Anweisungen wie Methodenaufrufe, Zugriffe auf Instanzvariablen und Objekt-Erzeugung haben auch keinen Einfluss auf die Komplexität.

Zum Beispiel hat die folgende C#-Methode demnach die CC-Zahl 2:

```
public String WieHochIstMeineCC(String wert) {
  if (wert.Equals("Bar")) {
    Console.WriteLine("Bar");
    return "Bar gesetzt";
  }
  return "das war nix";
}
```

Das Java-Pendant dieser Methode produziert allerdings bei dem Werkzeug JavaNCSS die CC-Zahl 3, da bei diesem Werkzeug die *return*-Anweisung im *if*-Zweig die Komplexität um einen Punkt nach oben setzt. D. h. man sollte genau prüfen, auf welche Art und Weise das eingesetzte Werkzeug die CC berechnet.

Zurück zu iPlasma, diese nutzt die CC auf eine andere Art. Anstatt wie üblich die CC Zahl für jede Methode zu berechnen, wird die durchschnittliche Komplexität einer Programmierzeile berechnet. Hierzu wird zuerst die Metrik CYCLO errechnet:

$$\text{CYCLO} = \sum_{\text{alle Methoden } m} \text{CC}(m)$$

Hat man diese Metrik, so ergibt sich aus dieser die durchschnittliche Komplexität pro Zeile Source-Code:

$$\frac{CYCLO}{LOC}$$

Diese Kennzahl ist ein Indiz, wie komplex Programmzeilen sind. Ein Wert von 0,1 beispielsweise bedeutet, dass jede zehnte Programmzeile einen neuen Ausführungspfad enthält.

Schwellwerte

Wie will man nun die Zahlen, die durch die oben genannten Metriken erhoben werden können, interpretieren? Was ist eine gesunde Größe für eine Metrik? Ab wie viel Zeilen sollte man eine Methode oder eine Klasse genauer betrachten? Welche Komplexität ist noch akzeptabel?

Nun ja, da finden sich in den Werkzeugen bzw. in der Literatur verschiedenste Aussagen. PMD kennt Regeln für exzessive, lange Methoden und Klassen, deren Schwellwerte bei 100 Zeilen pro Methode und 1000 Zeilen pro Klasse liegen. Diese Grenzen lassen sich allerdings per Konfiguration verschieben. Robert C. Martin nimmt in seinem Buch *Clean Code* [9] schon eine extremere Haltung gegenüber Methodenlängen ein: Im Idealfall er empfiehlt eine Länge von bis zu vier Zeilen.

Um eine statistische Basis für die oben genannten Metriken bereitzustellen, wurde in [6] für 45 Java Projekte die genannten Metriken berechnet und daraus drei Schwellwerte pro Metrik definiert:

- *Niedriger Wert*:
  Statistisches Mittel – Standardabweichung
- *Durchschnittlicher Wert*:
  Statistisches Mittel
- *Hoher Wert*:
  Statistisches Mittel + Standardabweichung

In der folgenden Tabelle sind die aus [3] ermittelten Schwellwerte für Java-Projekte gezeigt.

Metrik	Niedrig	Durchschnitt	Hoch
$\frac{NOC}{NOP}$	6	17	26
$\frac{NOM}{NOC}$	4	7	10
$\frac{LOC}{NOM}$	7	10	13
$\frac{CYCLO}{LOC}$	0,16	0,20	0,24

Tabelle 1: Schwellwerte für Quotienten in Java Projekten

Zum Vergleich sind in der folgenden Tabelle die Schwellwerte von 37 untersuchten C++-Projekten gezeigt.[2]

Metrik	Niedrig	Durchschnitt	Hoch
$\frac{NOC}{NOP}$	3	19	35
$\frac{NOM}{NOC}$	4	9	15
$\frac{LOC}{NOM}$	5	10	16
$\frac{CYCLO}{LOC}$	0,20	0,25	0,30

Tabelle 2: Schwellwerte für C++ Projekte

[2] Interessant ist hierbei, dass die Komplexität, also die zyklomatische Komplexität der Code-Zeilen, zwischen C++ und Java um einen Schwellwert „verschoben" ist: Was für Java Durchschnitt ist, ist für C++ noch niedrig, etc.

Akzeptiert man nun diese Zahlen als repräsentativ, so kann man die erwähnten Metriken auf sein eigenes Projekt anwenden und hieraus Schlüsse ziehen, ob an der einen oder anderen Stelle Handlungsbedarf ist.

Darstellung der Metriken

Nackte Zahlen zu betrachten und zu bewerten ist oftmals schwierig. Eine geeignete Visualisierung kann hier ausgesprochen hilfreich sein. Hierdurch ergibt sich nämlich für den Code-Analysten die Möglichkeit, auf einen Blick die gewählte Metrik am Projekt auszuwerten.

Abbildung 1 zeigt die Anwendung der LOC-Metrik auf das Open-Source-Projekt Spring.Net unter Zuhilfenahme von NDepend. Dabei werden die LOC pro Methode als Rechtecke dargestellt.[3] Je größer ein Rechteck ausfällt, desto mehr Zeilen hat die Methode. Wählt man auf der linken Seite im Navigator einen Namespace oder eine Klasse aus, so werden die entsprechenden Rechtecke eingefärbt. Somit erhält man einen guten Überblick über die Größe der Methode, der Klasse oder des Namespaces im Vergleich zum übrigen System. Im dargestellten Screenshot wurden per Filter die zehn größten Methoden eingefärbt. Damit erhält man schnell eine Aussage darüber, welche Methoden man zuerst bewerten sollte.

In [6] wird eine weitere Art zur Darstellung der oben genannten Metriken vorgestellt, der sogenannte *Pyramidal View*, der neben den oben genannten Metriken noch weitere nutzt, um einen ersten Überblick über das Gesamtsystem zu vermitteln. So bewerten sie die Kopplung zwischen den Klassen und den Einsatz von Vererbung. Diese Metriken werden an dieser Stelle nicht weiter betrachtet, sondern in einem der folgenden Artikel aufgegriffen.

[3] Die Darstellung erinnert stark an das Werkzeug SequoiaView [10] , das sogenannte „squarified treemaps“ zur Darstellung von Dateigrößen nutzt. Diese Werkzeug erleichtert durch seine visuelle Darstellung das Auffinden von großen Dateien im Dateisystem.

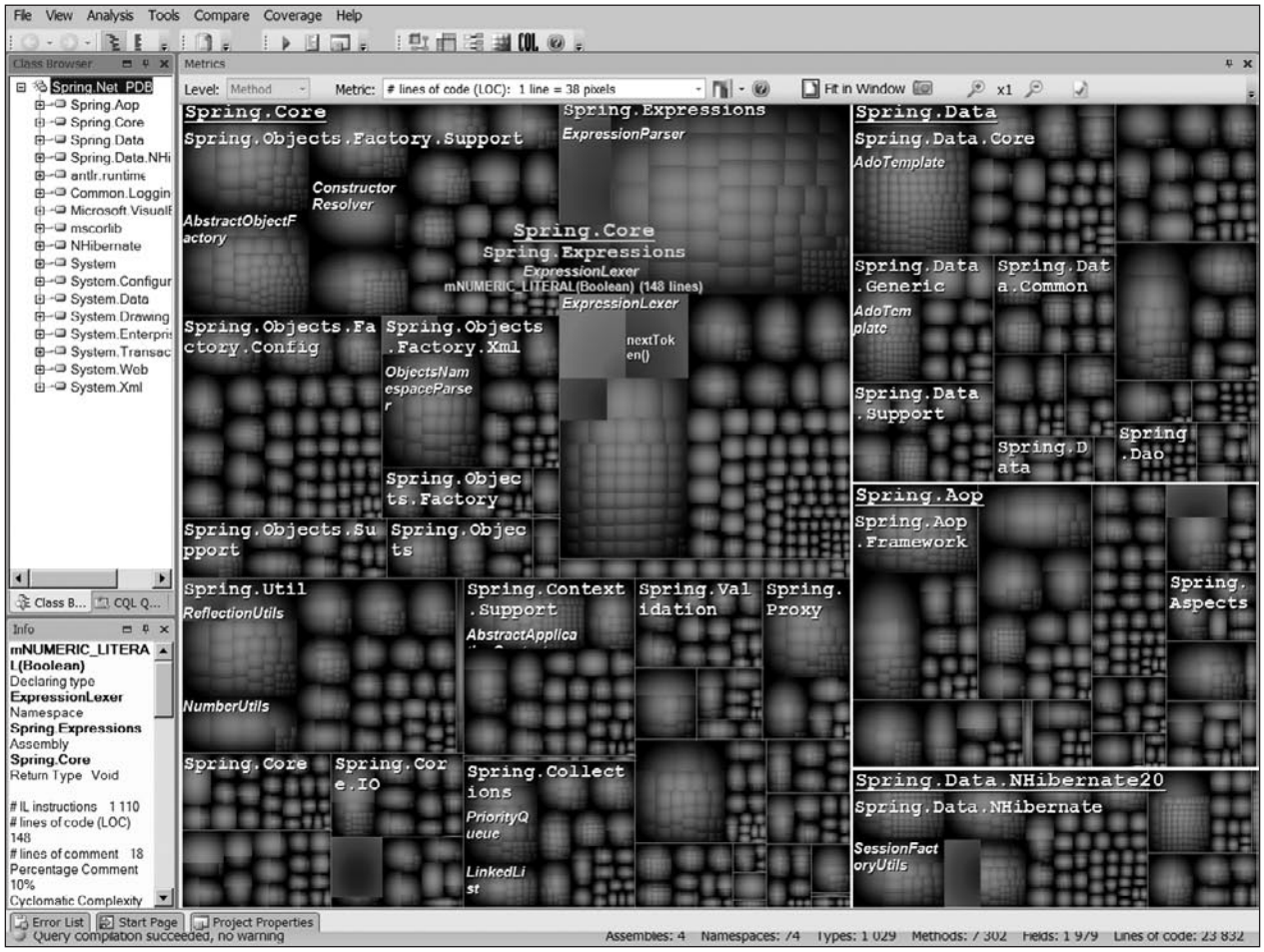

Abbildung 1: *LOC* Metrik auf Spring.Net in NDepend

Ein Beispiel für die Pyramiden-Darstellung mittels iPlasma ist in der Abbildung 2 für das Open Source Projekt Spring (Java) in der Version 2.5.1 gezeigt.

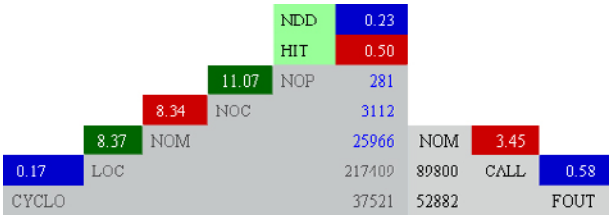

Abbildung 1: LOC-Metrik auf Spring.Net in NDepend

Auf der linken Seite des Dreiecks finden sich die Werte für die Metriken CYCLO, LOC, NOM, NOC und NOP innerhalb des Dreieckschenkels. In der Zeile LOC kann man sehen, dass insgesamt 217409 Code Zeilen gezählt wurden. Die Zahl die oberhalb der Abkürzung CYCLO steht, ist der Quotient aus LOC durch CYCLO, die Zahl oberhalb von LOC der Quotient von LOC durch NOM, usw.

Diese berechneten Quotienten werden von iPlasma entsprechend ihrer Position zu den im vorherigen Abschnitt genannten Schwellwerten farblich gekennzeichnet, wobei ein „sehr hoher Wert" nicht mehr betrachtet wird. Werte die über dem Schwellwert „hoch" liegen, werden rot eingefärbt, Werte die nahe dem Durchschnitt liegen, werden grün dargestellt und Werte nahe und unter dem Wert „niedrig" liegen, werden blau gefärbt.

Dies bedeutet für das Beispiel Spring 2.5.1, dass alle Werte bis auf die durchschnittliche Anzahl der Methoden pro Klasse in der Norm liegen. Spring tendiert also zu Klassen, die möglicherweise zu viele Methoden besitzen, dafür aber nur eine durchschnittliche Länge von 8,37 Zeilen besitzen.

iPlasma gibt zusätzlich eine Bewertung des analysierten Systems ab:[4]

Classes

- *tend to be rather large (i.e. they define many methods).*
- *tend to be organized in average-sized packages.*[5]

Methods

- *tend to be average in length and having a rather simple logic (i.e. few conditional branches).*[6]

Fazit

Im vor liegenden Artikel wurden einige grundlegende Metriken vorgestellt. Die Anwendung und Darstellung dieser Metriken wurde exemplarisch in zwei Werkzeugen gezeigt. Selbstverständlich gehen die Möglichkeiten der beiden Werkzeuge weit über das gezeigte hinaus. Insbesondere kann iPlasma durch Verwendung von weiteren Metriken und die Kombination von Metriken neue *Code-Gerüche* auffinden.

Referenzen

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

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

[3] NDepend http://www.ndepend.com

[4] JavaNCSS *A Source Measurement Suite for Java,* http://www.kclee.de/clemens/java/javancss

[5] PMD http://pmd.sourceforge.net/

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

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

[8] DeMarco, T. *Was man nicht messen kann...,* mitp-Verlag, 2004

[9] Martin, R. C. *Clean Code,* Prentice Hall, 2009

[10] Technische Universiteit Eindhoven *SequoiaView,* http://w3.win.tue.nl/nl/onderzoek/onderzoek_informatica/visualization/sequoiaview

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.

[4] Dies ist nur ein Ausschnitt der Bewertung von iPlasma, die hier nicht gezeigten Bewertungen beziehen sich auf noch nicht besprochene Metriken.

[5] *Klassen*
- *neigen dazu, zu groß zu sein (z. B. definieren sie zu viele Methoden).*
- *sind in Paketen durchschnittlicher Größen organisiert.*

[6] *Methoden*
- *neigen dazu, eine Durchschnittslänge und eine eher einfache Logik zu haben (z. B. wenige bedingte Anweisungen)*