Software-Metriken gezielt einsetzen

Daniel Fischer, daniel.fischer@fh-offenburg.de
Eugenia Drosdezki, eugenia.drosdezki@printus.de
Konstantin Schuraev, schuraev@verifysoft.com

Software-Metriken stellen Kennwerte dar, um Software hinsichtlich Komplexität, Wartbarkeit und Fehleranfälligkeit zu bewerten. So empfiehlt die DIN EN 61508 die Erhebung von Software-Metriken für die Safety Integrity Level (SIL) 1-4. Weitere Empfehlungen der DIN EN 61508, wie beispielsweise die der strukturierten Programmierung, können aus einzelnen Software-Metriken abgeleitet werden. Software-Metriken bewerten allerdings jeweils nur einen Teilaspekt der Komplexität und sind daher entsprechend zu interpretieren. Die einzelnen Metriken werden vorgestellt und einer kritischen Bewertung unterzogen. Ebenso wird dargestellt, welche Metriken für die jeweiligen Zielgruppen von Interesse sein könnten.

1. Software-Metriken

Tom DeMarco, der Erfinder der Strukturierten Analyse und Autor zahlreicher wegweisender Publikationen aus den Gebieten des Software Projektmanagements und der Softwaretechnik, stellte folgende Behauptung auf: *„You can't control what you can't measure"*. Sein Ansinnen war es, Software-Projekte durch eine Vielzahl von Kennzahlen besser überwachen und zum Erfolg führen zu können.

Software-Metriken können dabei grob in die folgenden Bereiche unterteilt werden:

- Produkt-Metriken
- Prozess-Metriken
- Projekt-Metriken

Im Rahmen dieser Veröffentlichung soll aufgezeigt werden, wie eine Ausprägung von Produkt-Metriken, die sogenannten Code-Metriken, im Bereich von Embedded Systems eingesetzt werden können.

2. Code-Metriken

In den 70er und 80er Jahren dominierten neben Assembler die prozeduralen Programmiersprachen. Mit den verschiedenen Sprachen wurden in dieser Zeit auch die ersten Code-Metriken eingeführt. Neuere prozedurale Code-Metriken wie beispielsweise der Maintainability-Index oder die maximale Schachtelungstiefe ergänzen die Code-Metriken der ersten Generation. Ebenso findet man in der neueren Literatur ([Mar09]) weitere interessante Ansätze für mögliche neue Metriken, wie beispielsweise die Begrenzung der Anzahl der Übergabeparameter sowie die Begrenzung der Anzahl unterschiedlicher Datentypen.

Prozedurale Code-Metriken können auf verschiedenen Ebenen angewendet werden:

- Prozedurebene (Funktion in C, bzw. Methode in C++)
- Modulebene (C-Datei(en) bzw. Klasse in C++)
- Programmebene (gesamtes Projekt)

Diese Metriken können auch auf objektorientierte Software mit wenigen Einschränkungen angewendet werden. In den folgenden Abschnitten soll nun im Detail auf die prozeduralen Code-Metriken eingegangen werden.

Lines of Code (LOC)

Die LOC-Metrik stellt eine recht einfache Maßzahl da, um die Komplexität der Software zu bestimmen. Bei der Untersuchung der Programmzeilen kann nochmals unterschieden werden:

- LOCphy: Gesamtanzahl der Zeilen (physical lines)
- LOCpro: Anzahl der Programmzeilen (program lines)
- LOCcom: Anzahl der Zeilen mit Kommentar (commented lines)
- LOCbl: Anzahl der Leerzeilen (blank lines)

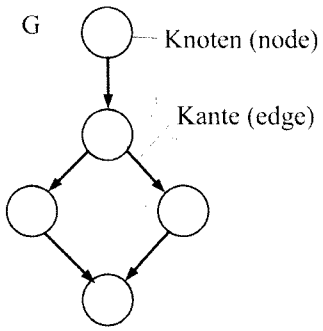
Wichtig ist in diesem Zusammenhang auch das Verhältnis LOCcom/LOCphy, welches in den Maintainability-Index einfließt. Ebenso kann dieses Verhältnis bei Code Reviews herangezogen werden, um ggf. die Einhaltung von Coding-Styleguides zu überprüfen.
In der Literatur ([CL07]) werden für Funktionen und Module (Dateien) Empfehlungen für die unterschiedlichen LOC-Metriken gegeben. So soll beispielsweise das Verhältnis LOCcom/LOCphy zwischen 0.35 und 0.75 liegen. LOCphy einer Funktion sollte kleiner als 40 sein, während LOCphy einer Datei den Wert 400 nicht übersteigen soll.

McCabe Komplexität v(G)

Thomas J. McCabe legte bei seiner Maßzahl den gerichteten Kontrollflussgrafen G einer Funktion zugrunde ([McC76]). Ausgehend vom gerichteten Kontrollflussgrafen G kann seine Komplexitätsmetrik, auch zyklomatische Komplexität (Cyclomatic Complexity CC) genannt, wie folgt bestimmt werden:

$$v(G) = CC = e - n + 2 \cdot p \qquad (1)$$

Dabei handelt es sich bei e um die Anzahl der Kanten (*edges*) und bei n um die Anzahl der Knoten (*nodes*) des Kontrollflussgrafen G. Hingegen gibt p die Anzahl der nicht verbundenen Teilgrafen an, was der Anzahl der Prozeduren in C (Funktionen) entspricht. Wird v(G) einer Prozedur bestimmt, so ist p=1 zu setzen. Soll v(G) eines Moduls bestimmt werden, so repräsentiert p die Anzahl der darin enthaltenen Prozeduren. In Abbildung 1 ist ein einfacher Kontrollflussgraf einer Prozedur mit v(G)=2 aufgezeigt.




```
int foo (int a)
{
a = a%10;
if (a <3)
  a = 0;
else
  a = a – 2;
return a;
}
```

Abbildung 1: *Gerichteter Kontrollflussgraf einer Funktion sowie der entsprechende Sourcecode*

Die McCabe Komplexität bewertet den Kontrollfluss einer Prozedur. Jede weitere Verzweigung (if-Kontrollstruktur) oder Schleife (abweisende oder nicht abweisende Schleife) erhöht v(G) um jeweils eins. Zusätzliche einfache oder komplizierter erscheinende Anweisungen (Sequenzen) wie beispielsweise *a++* oder **pid = pmt->taskid + 1* ergeben einen neuen Knoten (node) als auch eine neue Kante (edge) im Kontrollflussgrafen, was sich aber durch den Teilausdruck $e - n$ wieder bei der Berechung von v(G) aufhebt.

Weitere verschiedene Kritikpunkte sind in der Literatur zu finden. So werden, ausgehend vom Kontrollflussgrafen G, Abfragen wie if (a<3) genauso behandelt wie Abfragen, die wesentlich komplexer sind wie z.B. *if (a<3 && b >(a+10))*. Diesen Kritikpunkt behebt die Extended Cyclomatic Complexity (ECC). Bei dieser Metrik fließen alle binären Entscheidungen ein. Sie stellt eine Verbesserung der ursprünglich von McCabe definierten Metrik dar und lässt sich wie folgt bestimmen:

$$ECC = Anzahl\ binärer\ Entscheidungen + 1 \qquad (2)$$

Softwarewerkzeuge bestimmen heutzutage meist schon ECC, weisen diese Variante der McCabe Komplexität aber oft nicht aus. Weitere Einschränkungen ergeben sich aber daher, dass der Datenfluss sowie die Schachtelungstiefe unberücksichtigt bleiben. Ebenso gehen Abfragen (if) mit gleicher Gewichtung ein wie Schleifen (for, while, do-while). Auch erhöht ein else-Zweig die Komplexität von v(G) oder ECC nicht.

Eine weitere auf v(G) basierende Komplexität wurde schon von McCabe eingeführt. Er reduzierte den Kontrollflussgrafen G schrittweise von innen nach außen, indem er strukturierte Basisstrukturen (if/else, if, for, while, do/while) mit jeweils einem Eingang und einem Ausgang aus dem Grafen entfernte. Vom verbleibenden Grafen bestimmte er mittels der Gleichung (1) die verbliebene Komplexität, welche er essenzielle zyklomatische Komplexität nannte (ev(G)). Nur Funktionen mit ev(G)=1 können als strukturiert bezeichnet werden, da sie keinerlei unstrukturierte Kontrollstrukturen enthalten. Um ev(G)=1 zu erreichen, sind im Sinne der strukturierten Programmierung die folgenden Grundregeln einzuhalten:

- Eine Funktion hat einen Eingang und einen Ausgang
- Es darf nicht in Abfragen gesprungen werden
- Es darf nicht aus Abfragen herausgesprungen werden
- Es darf nicht in Schleifen gesprungen werden
- Es darf nicht aus Schleifen herausgesprungen werden

In der Literatur findet man verschiedene Empfehlungen. McCabe selbst empfiehlt in [McC76] v(G) einer Funktion sollte maximal 10 betragen. Andere Autoren, z.B. [CL07], empfehlen die Begrenzung auf v(G)<=15. Interessanterweise empfiehlt die DIN EN 61508, Teil 3 ([DIN01]), in Tabelle A.4 für alle Safety Integrity Level (SIL) die Einhaltung der strukturierten Programmierung, was einer Forderung nach ev(G)=1 für alle Funktionen gleichzusetzen ist.

Halstead-Metriken

Maurice Halstead ([Hal77]) verfolgte hingegen einen anderen Ansatz. Er versuchte die Verständlichkeit eines Programms anhand der Anzahl der verwendeten Operanden und Operatoren zu bestimmen. Die Details, was nach Halstead zu den Operanden und Operatoren gezählt wird, ist in [CL07] im Detail beschrieben. Tabelle 1 gibt hierzu einen Überblick.

	Beispiele	Basismetriken
Operatoren	C-Operatoren wie %, /, * usw. Klammern, Kommata, logische Operatoren, Schlüsselwörter, ...	N1: Gesamtanzahl Operatoren
		n1: Anzahl unterschiedlicher Operatoren
Operanden	Konstanten, Variablen, Datentypen, Funktionsaufrufe, Funktionsname, ...	N2: Gesamtanzahl Operanden
		n2: Anzahl unterschiedlicher Operanden

Tabelle 1: *Halstead Basismetriken*

Aus diesen vier Basismetriken leitete Halstead verschiedene weitere Metriken ab. Die wichtigsten Metriken sind hierbei die Programmgröße (Volumen) V, der Schwierigkeitsgrad D, der Aufwand E sowie die geschätzte Zahl ausgelieferter Fehler B.

$$V = (N1 + N2) \cdot lb(n1 + n2) \quad (3)$$
$$D = (n1/2) \cdot (N2/n2) \quad (4)$$
$$E = V \cdot D \quad (5)$$
$$B = E^{2/3}/3000 \quad (6)$$

Maintainability-Index MI (Wartbarkeitsindex)

Die bisher vorgestellten Metriken bewerten dabei immer nur einen Teilaspekt des Programmcodes. Zu jeder der dieser Metriken lassen sich Gegenbeispiele konstruieren, bei denen einzelne Metriken fragwürdige Ergebnisse liefern.
Ausgehend von dieser Problematik wurde an der University of Idaho der Maintainability-Index MI entwickelt. MI stellt dabei eine Kostenfunktion dar, in welcher die prozeduralen Code-Metriken nach Halstead (V) und McCabe (v(G), bzw. ECC) sowie die Programmzeilen (LOCphy) als Strafterme Einfluss finden. Der Wartbarkeitsindex ohne Berücksichtigung kommentierter Zeilen (MIwoc – MI without comments) ist wie folgt definiert.

$$MIwoc = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot v(G) - 16.2 \cdot \ln(LOCphy) \quad (7)$$

Berücksichtigt man nur Kommentare, so ergibt sich ein Wartbarkeitsindex MIwc (MIwc – MI with comments) wie in der folgenden Gleichung dargestellt ist.

$$MIwc = 50 \cdot \sin(\sqrt{2.4 \cdot (LOCcom/LOCphy)}) \quad (8)$$

Der gesamte Wartbarkeitsindex MI ergibt sich durch Summation von MIwoc und MIwc. Werden Kommentare nicht berücksichtigt ist MI = MIwoc zu setzen.

$$MI = MIwoc + MIwc \quad (9)$$

Die zusätzliche Berücksichtigung von MIwc ist schon deshalb kritisch zu hinterfragen, da auskommentierter Programmcode sich noch positiv auf MI auswirken würde, was allerdings nicht der Fall sein kein.

In der Microsoft Visual Studio Team System 2008 Entwicklungsumgebung sind Code-Metriken integriert. Neben zwei objektorientierten Metriken werden die McCabe Komplexität sowie LOC explizit als Metrik ausgewiesen. Ebenso wird MIwoc ermittelt, ohne dabei das notwendige Halstead-Volumen auszugeben. Während MIwoc auch negative Werte annehmen und im besten Fall einen Wert nahe 171 haben kann, wird dort der Maintainability-Index auf den Bereich zwischen 0 und 100 abgebildet ([MS09]).

$$MIwoc_{MS} = \max(0, (171 - 5.2 \cdot \ln(V) - 0.23 \cdot v(G) - 16.2 \cdot \ln(LOCphy)/1.71) \quad (10)$$

In der Literatur ([CL07]) wird ein Wert für MIwoc größer oder gleich 85 als gut wartbar interpretiert. Liegt MIwoc im Bereich von 65 bis 85, so wird von einer mäßigen Wartbarkeit ausgegangen. Funktionen mit Werten kleiner als 65, werden als schlecht wartbar eingestuft.

Microsoft unterteilt seinen auf 100 normierten Wartbarkeitsindex ebenso in drei Bereiche, welche farblich mit den Farben Rot (0-9), Gelb (10-19) und Grün (20-100) gekennzeichnet

sind. Rechnet man die normierten Werte auf die ursprüngliche Skalierung zurück, so ergibt sich ein differenziertes Bild, wie Wartbarkeit zu interpretieren ist (siehe Abbildung 2).

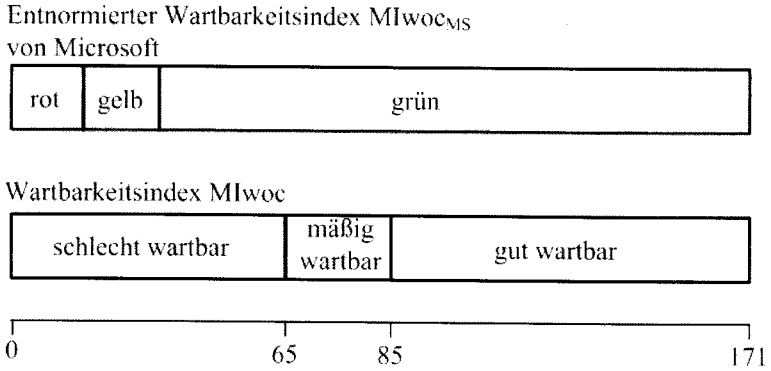


Abbildung 2: *Vergleich MIwoc und MIwoc$_{MS}$*

Maximale Schachtelungstiefe MaxND

Ein Kritikpunkt an v(G) und ECC ist, dass fehleranfällige ineinander geschachtelte Kontrollstrukturen genauso bewertet werden, wie nicht geschachtelte Kontrollstrukturen. Die Metrik MaxND gibt dabei die maximale Schachtelungstiefe einer Funktion an.

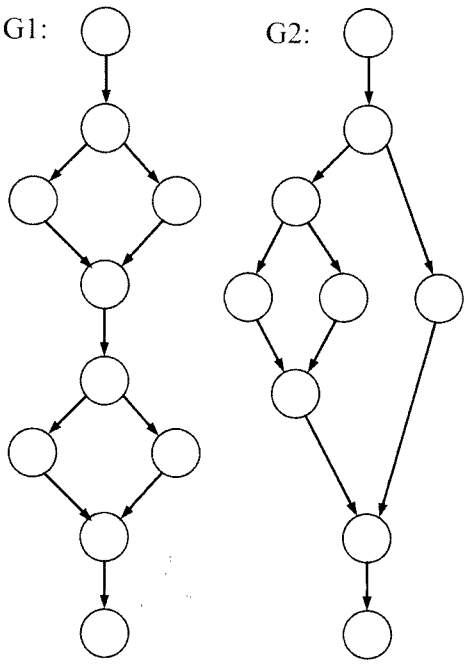


Abbildung 3: *Kontrollflussgraf zweier Funktionen mit gleichem v(G) aber unterschiedlichem MaxND*

Abbildung 3 zeigt dies exemplarisch auf. Beide dargestellten Kontrollgrafen G1 und G2 weisen ein v(G) von drei auf. Allerdings ergibt sich für G1 ein MaxND von zwei, während sich für G2 ein MaxND von drei ergibt.

Dabei gilt immer:

$$v(G) \geq MaxND \qquad (11)$$

MaxND kann demnach höchstens den Wert von v(G) annehmen.

In Abbildung 5 kann man anhand der Funktionen, welche sich im Linux Kernelverzeichnis befinden, erkennen, dass MaxND meist begrenzt ist. Die maximale Schachtelungstiefe MaxND ergab bei dieser Untersuchung einen Wert von MaxND=6.

Anzahl der Übergabeparameter

Robert C. Martin weist in [Mar09] auf eine weitere mögliche Metrik hin. Er empfiehlt die Anzahl der Überparameter auf kleiner als drei zu begrenzen: *„Three arguments [..] should be avoided where possible"*. Dabei wird auch argumentiert, dass auch das Testen aufwendiger wird, da alle möglichen Kombinationen zu testen sind.
Dazu konform findet man in der DIN EN 61508 ([DIN01]) in Tabelle B.9 auch die Empfehlung, die Anzahl der Übergabeparameter zu begrenzen.

3. Zielgruppen

Die dargestellten Metriken sollten gezielt eingesetzt werden. Für das **Management** sind die Metriken in verdichteter Form darzustellen. So eignen sich hier insbesondere Kiviat- und Häufigkeitsdiagramme. Abbildung 4 zeigt exemplarisch das Häufigkeitsdiagramm von MIwoc der Funktionen des Linux Kernelverzeichnisses (Version 2.6.16.60).

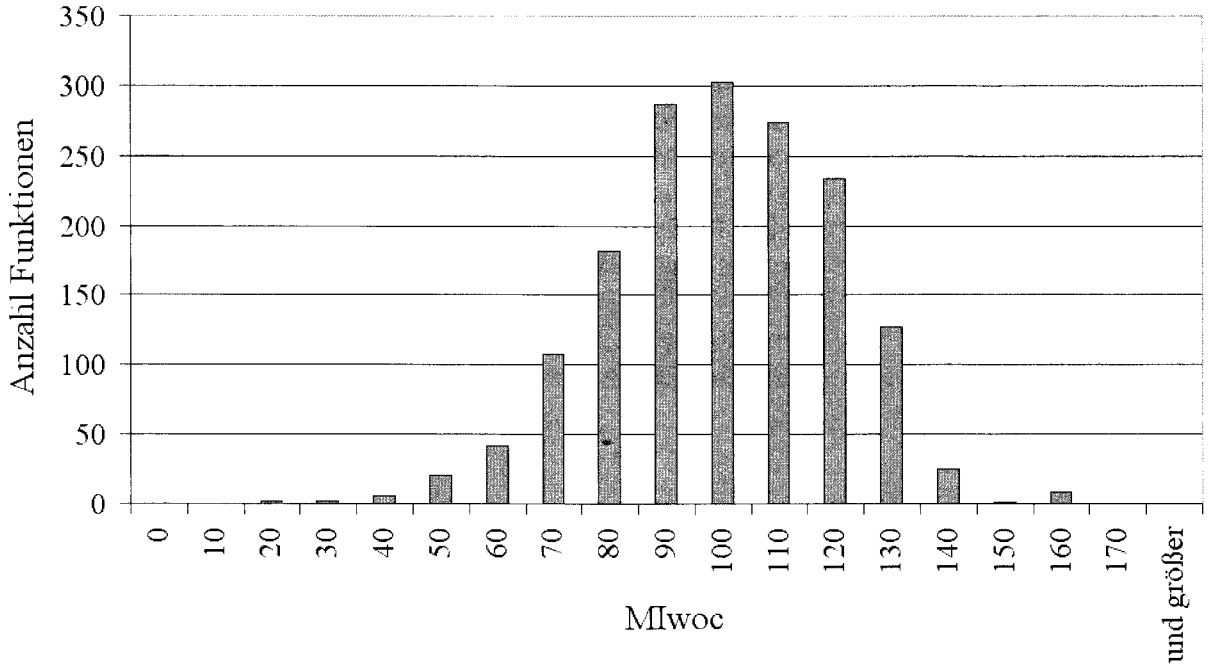


Abbildung 4: *Häufigkeitsverteilung von MIwoc (Linux Kernelverzeichnis 2.6.16.60)*

Oft findet man heutzutage zur Aufwandsabschätzung von Wartungs- und Entwicklungsprojekten so genannte Function Points (FP) als Maßeinheit. Die vorgestellte Zeilenmetrik LOC lässt sich in Function Points umrechnen. Jones gibt in [Jon99] an, wie viel LOC im Mittel in der jeweiligen Sprache nötig sind, um einen Function Point zu realisieren.

Sprache	LOC	Sprache	LOC
Assembler	320	Ada83	71
Macro-Assembler	213	C++	53
C	128	Ada95	49
FORTRAN	107	Visual Basic	32
COBOL	107	Smalltalk	21
Pascal	91	SQL	12
PL/I	80		

Tabelle 2: *Anzahl von Programmzeilen pro Function Point*

Beim **Projektmanagement** sind Projekte über mehrere Releases oder Versionen zu überwachen. Daher ist es naheliegend, die Metriken von Funktionen oder Modulen sowie von dem gesamten Projekt über einen Zeitraum zu verfolgen. Je nach Software-Werkzeug können Metriken erhoben und diese über einen bestimmten Zeitraum verglichen werden. Diese Funktionalität wird als „Snapshot" oder „Report-Session" ([Dro09]) bezeichnet. Auch lassen sich damit erfolgreich durchgeführte Refactoring-Maßnahmen gegenüber dem Management belegen.

Reviews können nachhaltig zur Steigerung der Softwarequalität beitragen. Bei Code-Reviews ergibt sich die Herausforderung, dass man innerhalb einer begrenzten Zeit möglichst die kritischen Funktionen oder Module inspizieren sollte. Hierzu könnten Metriken herangezogen werden. Software-Werkzeuge sollten die wenigen Funktionen/Module für den Reviewer und Autor anzeigen, welche besonders komplex erscheinen. Hier eignet sich insbesondere der Wartbarkeitsindex MIwoc. Aber auch Funktionen mit einem hohen Wert für MaxND sollten berücksichtigt werden.

Für **Tester** ist insbesondere v(G) von Interesse. v(G) repräsentiert dabei auch die Anzahl linear unabhängiger Pfade einer Funktion. Dies lässt den zu erwartenden Testaufwand abschätzen. Funktionen, welche eine hohe Anzahl von Übergabeparametern haben, sollten ebenso intensiv getestet werden (Schnittstellenproblematik). Funktionen mit einem hohen MaxND lassen ebenso eine höhere Fehleranfälligkeit vermuten. So können beispielsweise Funktionen mit mittleren v(G) einfach erscheinen, während diese aber einen recht hohen Wert für MaxND haben können. Abbildung 5 zeigt hierbei das Punktdiagramm von MaxND in Abhängigkeit von v(G) des Linux Kernelverzeichnisses auf. Funktionen mit einem hohen MaxND sollten, unabhängig von deren v(G), intensiver getestet werden.

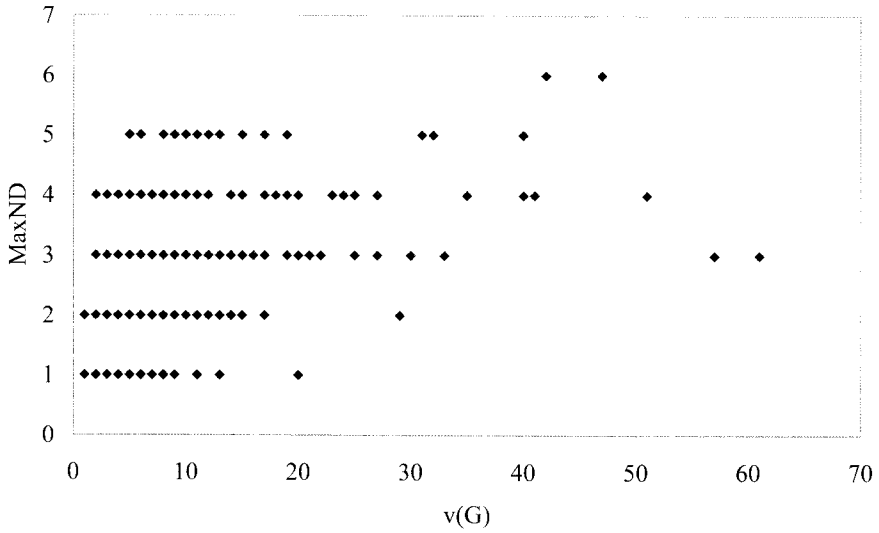


Abbildung 5: *Punktdiagramm MaxND und v(G) (Linux Kernelverzeichnis 2.6.16.60)*

4. Zusammenfassung

Code-Metriken können heutzutage effizient durch den Einsatz von SW-Werkzeugen ermittelt werden. Sie beschreiben quantitativ und reproduzierbar die Komplexität der Software und können verwendet werden, um Aufwände, Fehleranfälligkeit sowie Wartbarkeit abzuschätzen. Unterschiedliche Studien wurden hierzu mit teils unterschiedlichen Ergebnissen durchgeführt ([FO00], [AR07]). Eine Vergleichbarkeit dieser Studien ist dabei nur eingeschränkt möglich, da sich die einzelnen Projekte unterscheiden. Auch in [Kan03] wird die Vorgehensweise bei der Ermittlung von Korrelationskoeffizienten kritisiert. Hinsichtlich neuerer Studien sei auf [Cov08] und [Cov09] hingewiesen.
Grundsätzlich führt die Erhebung von Software- und Code-Metriken meist zu einer Qualitätssteigerung der Software. Allerdings sollte man auch bei der Weitergabe der Daten an das Management eine gewisse Sensibilität walten lassen. Hier empfiehlt es sich, verdichtete Daten zu verwenden, welche auch einen Vergleich mit anderen Projekten erlauben.

Literatur

[AR07] Andersson, Carina; Runeson, Per: *A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems*, IEEE Transactions on Software Engineering, Vol. 33, No. 5, 2007

[Cov08] Coverity™: *Open Source Report*, 2008, http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2008.pdf

[Cov09] Coverity™: *Open Source Report*, 2009, http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2009.pdf

[CL07] Cullmann, Xavier-Noel; Lambertz, Klaus: *Komplexität und Qualität von Software*, MSCoder, Ausgabe 1/2007, pp. 36-43, 2007

[DIN01] DIN EN 61508 – 3: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme, Teil 3, deutsche Fassung, DIN, 2001

[Dro09] Drosdezki, Eugenia: *Realisierung eines Java-Tools zum Qualitätsmonitoring basierend auf Codemetriken*, Bachelor-Thesis, Hochschule Offenburg, 2009

[FO00] Fenton, Norman E.; Ohlsson, Niclas: *Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Transaction on Software Engineering*, Vol. 26, No. 8, 2000

[Hal77] Halstead, Maurice H.: *Elements of Software Science*, Operating, and Programming Systems Series, Volume 7, New York, Elsevier, 1977

[Jon99] Jones, Capers: *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley, 1999

[Kan03] Kan, Stephan H.: *Metrics and Models in Software Quality Engineering*, 2nd Edition, Addison-Wesley, 2003

[Mar09] Martin, Robert C: *Clean Code*, A Handbook of Agile Software Craftsmanship, Pearson Eduction, 2009

[McC76] McCabe, Thomas J.: *A Complexity Measure*, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, 1976

[MS09] http://blogs.msdn.com/fxcop/archive/2007/11/20/maintainability-index-range-and-meaning.aspx, Stand: 04.10.2009

Autoren:

Daniel Fischer studierte Elektrotechnik an der FernUniversität in Hagen und promovierte auf dem Gebiet der dreidimensionalen Bildverarbeitung. Im Rahmen seiner beruflichen Tätigkeit bei Hewlett Packard und der Heidelberger Druckmaschinen AG war er im Bereich der Software-Qualitätssicherung und der Testautomatisierung für Embedded Systems tätig. Seit 2001 ist er Professor für Informationstechnologie an der Hochschule Offenburg. Er leitet dort die Studiengänge Angewandte Informatik und Informatik/Wirtschaft plus.

Eugenia Drosdezki hat an der Hochschule Offenburg Angewandte Informatik studiert. Im Rahmen ihrer Bachelor-Thesis hat sie bei Verifysoft Technology ein grafisches Frontend zur Verdichtung und Visualisierung von zielgruppenorientierten Code-Metriken konzipiert und realisiert. Seit Anfang des Jahres arbeitet sie als Software-Entwicklerin bei der Printus GmbH in Offenburg und ist dort im Bereich der Entwicklung webbasierter Frontend-Systeme tätig.

Konstantin Schuraev studiert derzeit an der Hochschule in Offenburg Angewandte Informatik mit den beiden Vertiefungsrichtungen Verteilte Systeme und Kommunikation sowie Anwendungsentwicklung. Momentan erstellt er im Rahmen seiner Bachelor-Thesis bei Verifysoft Technology ein plattformunabhängiges Analysewerkzeug zur Visualisierung von Testüberdeckungsmetriken.