

METRIKEN IM PRAKTISCHEN EINSATZ

Seitdem Probleme mit Hilfe von Softwaresystemen gelöst werden, ist man auf der Suche nach Verbesserungen. Die Verbesserungen betreffen das Projektmanagement, den Entwicklungsprozess und die eingesetzten Techniken. Um Verbesserungen feststellen zu können, müssen die Auswirkung von Veränderungen bestimmt werden. Diese Veränderungen können mit Hilfe verschiedener Kennzahlen, den Metriken, bestimmt, bzw. gemessen werden. Für die Beurteilung von Prozessen, z. B. nach CMMI (Capability Maturity Model), ist das Steuern durch Metriken unerlässlich. Doch wie sieht dies auf der technologischen Seite aus und wie kann ich Verbesserungen und die Qualität der Software messen? Dieser Artikel stellt eine Auswahl von Metriken vor, zeigt deren Praxisrelevanz und skizziert einige Tools und Vorgehensweisen, die in Softwareprojekten eingesetzt werden können.

Softwaresysteme, die mit Hilfe von Java- oder Java-Enterprise-Technologien implementiert werden, sind schon lange den Kinderschuhen entwachsen. Java hat sich als Programmiersprache für eine Vielzahl von Systemen durchgesetzt. Entwurfsmuster und *Best Practices* für Analyse, Design, Implementierung und Test wurden definiert und haben sich in der Praxis bewährt. Neue Ideen und Ansätze werden entwickelt, doch wie kann ihr Mehrwert bestimmt werden?

Je länger ein System verändert wird, je mehr Anpassungen und Erweiterungen hinzugefügt werden, um so höher das Risiko der Änderungen. Wie kann anhand der Implementierung bestimmt werden, ob die vorhandene Qualität ausreichend ist?

Dieser Artikel soll die Frage beantworten, wie ein komplexes Softwaresystem mit Hilfe von Metriken bewertet werden kann und welche Methoden und Werkzeugen hierfür zur Verfügung stehen. Die Werkzeuge liefern verschiedenen Metriken, mit deren Hilfe Rückschlüsse auf die Qualität und Eigenschaften des Systems möglich sind. Der Artikel beleuchtet außerdem eine Auswahl von Metriken und zeigt deren Bedeutung.

Was ist Qualität?

Häufig werden Anforderungen an die Qualität von Software nur über die funktionalen Anforderungen definiert. Die Qualität von Software definiert sich aber ebenfalls durch die Punkte Verständlichkeit, Vollständigkeit, Wartbarkeit, Testbarkeit und einige mehr.

Daher ist es notwendig, für die technische Qualität Maßstäbe und Anforderungen zu definieren. Diese Anforderungen lassen sich mit Hilfe von Metriken definieren und überprüfen. Die Metriken sind so zu definieren, dass diese sinnvolle Hinweise auf die Qualität bieten. Alexander v. Zitzewitz hat in [Zit07] unter anderem dargestellt, wie sich die logische Architektur definieren lässt, welche Faktoren die Qualität beeinflussen

und wie sich die korrekte Umsetzung der Architektur in der Implementierung anhand von technischen Metriken prüfen lässt.

Metriken – die Theorie

„You cannot control what you cannot measure“ ist Tom DeMarcos viel bemühtes Zitat, wenn es um Metriken geht (vgl. [DeM82]). Besteht keine Möglichkeit, für ein Projekt Messungen verschiedener Kennzahlen durchzuführen, so lassen sich keine Bewertungen durchführen und das Projekt lässt sich nicht kontrollieren. Ich kann ohne Messung keine Veränderungen bemerken und die Auswirkungen von Veränderungen nicht bewerten. „Die Metrik (griechisch μετρική, Zählung, Messung) bezeichnet im Allgemeinen ein System von Kennzahlen oder ein Verfahren zur Messung einer quantifizierbaren Größe“ (vgl. [Wik07]). Softwaremetriken sind Maßzahlen für bestimmte Eigenschaften einer Software oder deren Spezifikationen.

Metriken sollten immer so eingesetzt werden, dass ein bestimmtes Ziel erreicht wird. Zuerst muss eine Fragestellung formuliert werden, welche die Zielerreichung bestätigt. Dann muss identifiziert werden, welche Metriken diese Frage beantworten können, und anschließend muss ein Verfahren zur Bestimmung der Metrik festgelegt werden. In [Fen97] wird dieses Verfahren zum Auffinden notwendiger Metriken als „Goal-Question-Metric“ bezeichnet.

Metriken sind erst einmal nur Zahlenwerte. Sie erlangen ihre Aussagekraft durch ihre Wertentwicklung während der Projektlaufzeit oder im Vergleich zu Werten aus anderen Softwareprojekten. Da Metriken immer nur einen Messwert für den aktuellen Zustand des Systems liefern, müssen für alle verwendeten Metriken der zeitliche Verlauf und die Veränderung des Wertes beobachtet werden. Oft bekommt man besondere Einsichten, wenn die Metriken in Kombination bzw. deren Beziehungen betrachtet werden.



André Fleischer
(E-Mail: andre.fleischer@lhsystems.com)
ist Senior Consultant bei der Lufthansa Systems AS in Norderstedt bei Hamburg. Er war in verschiedenen großen Projekten in der Transport&Logistik-Branche tätig. In den Projekten war er Softwarearchitekt und Coach für objektorientierte Vorgehensweisen und Methoden.

Nutzer von Metriken sind vor allen Projektmanager, Softwareentwickler und Qualitätssicherer:

- Projektmanager sind daran interessiert, Antworten auf Fragen zum Fortschritt zu bekommen.
- Softwareentwickler interessiert dagegen eher, aus wie vielen Teilen eine bestimmte Komponente besteht oder wie stark die Kopplung bestimmter Komponente ist.
- Die Qualitätssicherer möchten die Qualität der Software bestimmen, z. B. welche Komponenten viele Fehler aufweisen,

Metriken sollten immer zielgerichtet einsetzen, d. h. es ist eine situationsbedingte Auswahl notwendig. Entsprechend dem Nutzer sollte eine Auswahl getroffen werden. Nicht jeder Benutzer ist an allen Metriken interessiert. Der Projektmanager kann mit vielen technischen Metriken wenig anfangen und sie kaum interpretieren. Die Auswahl der Metriken sollte passend zur Projektphase sein und gezielt bestimmte Fragestellungen beantworten. Die Metriken sollten die Behebung akuter Probleme unterstützen.

Metriken – eine Auswahl

Metriken lassen sich grob in zwei Typen einteilen: technische und nicht-technische Metriken.

Technische Metriken beurteilen sowohl den Quellcode als auch das Objektmodell und damit die Implementierung des Systems als solches.

Im Bereich der Implementierung und der Objektorientierung sind zahlreiche Metriken definiert. Diese sind entstanden, um die positiven Auswirkungen von neuen Methoden und Technologien messen zu können. Die nicht-technischen Metriken entstammen dem Projekt-Controlling; sie können bei-

spielsweise das Projektmanagement unterstützen. Diese Metriken können auch für die Schätzung von Projekten eingesetzt werden.

Hier eine Auswahl technischer Metriken:

- Anzahl Klassen, Anzahl abstrakte Klasse, Anzahl Interfaces
- Anzahl Methoden
- Anzahl der Quellcodezeilen (*Lines of Code*, *LOC*), Anzahl der Anweisungszeilen (*Non-Commenting-Source-Statement* (*NCSS*), Anzahl der Kommentarzeilen
- Anzahl Klassen pro Entwickler
- strukturelle Komplexität (McCabe-Metrik, *Cyclomatic Complexity Number* (*CCN*))
- ausgehende Abhängigkeit (*Efferent Coupling*, *Ce*), eingehende Abhängigkeit (*Afferent Coupling*, *Ca*), Abstraktheit (*Abstractness*, *A*), Instabilität (*Instability*, *I*), Distanz (*Distance*, *D*)
- durchschnittliche Abhängigkeit (*Average Component Dependency*, *ACD*)

Und hier einige nicht-technische Metriken:

- Anzahl der Use-Cases,
- Aufwand, Aufwand pro Klasse, allgemeine Budgetkontrolle, Restaufwand
- Anzahl der Fehler,
- Anzahl der Änderungen,
- Meilenstein-Erreichung,
- Termineinhaltung
- Funktionspunkte (*Function Points*)
- Widget-Punkte (*Widget-Points*)

Es reicht nicht aus, nur das Gesamtsystem zu betrachten – die Metriken müssen ebenso für sinnvoll gewählte Subsysteme und Komponenten erstellt werden, um Problemfelder eingrenzen zu können. Weicht die Metrik für eine Komponente von den Werten anderer Komponenten ab, so muss dies nicht zwingend ein schlechtes Zeichen sein. Es ist lediglich ein Indikator für Probleme; eventuell ist eine Komponente aus einem bestimmten Grund sehr komplex, beispielsweise aufgrund implementierter Optimierungsalgorithmen. In diesem Fall wäre eine Abweichung tolerierbar. Kann aber keine Begründung für eine Abweichung gefunden werden, so ist dies ein Indikator für Probleme.

Technische Metriken im Detail

Im Folgenden sollen die wichtigsten technischen Metriken vorgestellt und ihr Hintergrund kurz erläutert werden.

Die *Anzahl aller Klassen* im System ist einfach zu ermitteln und bildet die Basis für wei-

tere Metriken. Die Metrik sollte auch für Pakete (physikalische Gruppierungen von Klassen, in Java für *Packages*), Komponenten und Subsysteme verfügbar sein. Ebenso sollten auch die Durchschnittswerte für die Anzahl von Klassen pro Paket oder Komponente berechnet werden, um die Teile des Systems miteinander vergleichen zu können. Aus der Anzahl kann man bereits ableiten, ob eine Komponente im Vergleich zu anderen groß und somit eventuell komplex ist. Die Komplexität wird auch durch andere Faktoren – etwa die Kopplung und die Abhängigkeit – beeinflusst. Eine feste Grenze für die Anzahl Klassen pro Paket zu definieren, ist nicht sinnvoll. Neben den Klassen sollte die Metrik eben so für abstrakte Klassen und Interfaces bestimmt werden. Diese Unterscheidung wird später für die Kopplungsmetriken benötigt.

Wenn man die *Anzahl Klassen pro Entwickler* bestimmt, kann man kontrollieren, ob einzelne Entwickler zu viele Klassen bearbeiten müssen. Dadurch kann gegebenenfalls eine zu hohe Belastung der Entwickler vermieden werden (vgl. [Lor93]). Dazu gehört aber eine klare Zuordnung der Zuständigkeiten pro Klasse. Die Metrik „Klassen pro Entwickler“ hat daher häufig nur eine geringe Aussagekraft, da die Zuordnung oft nicht eindeutig ist. Der Ansatz der agilen Vorgehensweise, eine gemeinsame Quellcode-Verantwortung zu haben, lässt sich mit dieser Metrik nicht in Einklang bringen.

Für jede Klasse sollte die Metrik *Anzahl der Methoden* bestimmt werden. Die Anzahl sollte pro Klasse ermittelt und auf Paket- und Komponentenebene zusammengefasst werden. Außerdem sollte die durchschnittliche Anzahl Methoden über alle Klassen eines Pakets, einer Komponente und des Systems ermittelt werden. Diese Metrik ist eine gute Vorgabe für Designer und Entwickler. Die Anzahl der Methoden pro Klasse sollte eingeschränkt werden, um die Übersichtlichkeit zu bewahren. Diese Grenze sollte allerdings weich gesehen werden – es wird immer gute Gründe geben, warum eine Klasse mehr Methoden hat, als die Grenze vorsieht.

Zu den Quellcodezeilen (LOC) gehören die Zeilen mit eigentlichen Programmieranweisungen, die Leerzeilen und die Zeilen mit Kommentaren. Die Zeilen mit Anweisungen der Programmiersprache (*Statements*) werden als *NCSS*-Metrik bestimmt. Die *NCSS*-Metrik zählt alle Zeilen mit einer Anweisung oder – einfach gesagt – in Java alle Zeilen mit einem Semikolon und alle öffnenden geschweiften Klammern.

Die *Kommentar-Metrik* zählt alle Zeilen innerhalb der Kommentar-Blöcke. Diese

beiden *LOC*-Metriken (*NCSS* und *Kommentar*) können als absoluter Wert für das gesamte System, pro Subsystem oder Paket gebildet werden und auch als Durchschnittswert auf der entsprechenden Ebene.

Das Verhältnis zwischen *NCSS* und *Kommentaren* sollte als Regel im Projekt festgelegt werden, aus Erfahrung empfiehlt sich ein Verhältnis von 1:4 als Durchschnitt über alle Klassen (mindestens eine Zeile *Kommentar* auf vier Zeilen *Statements*). Klassen mit Geschäftslogik benötigen ein geringeres Verhältnis (1:3), *Value*-Objekte vertragen dagegen einen höheren Wert (1:6). Diese Verhältnisse sind aber auch nicht als feste Grenze zu sehen. Die Grenze zwischen „gut“ und „schlecht“ ist schwimmend.

Die *NCSS*-Metrik ist am leichtesten zu erstellen, sie ist aber auch eine der Metriken mit der geringsten Aussagekraft. Wenn eine Software 5.000 Codezeilen umfasst, so lassen sich daraus keine Schlüsse ziehen. Der Aufwand für die Entwicklung ist mit Hilfe dieser Metrik nicht abschätzbar, auch nicht die Komplexität. Die Anzahl der Zeilen muss immer im Kontext gesehen werden, also zum Beispiel in Bezug auf die zeitliche Entwicklung der Anzahl der Zeilen oder in Bezug auf andere Metriken und deren Entwicklung. Auch sagt die *Kommentar-Metrik* nichts über die Qualität der Kommentare aus.

Es muss überlegt werden, welche Dateien mit in die Metrik aufgenommen werden müssen. In Web-Anwendungen wird man die Zeilen der *HTML*- oder *JSP*-Dateien mit in die Metrik aufnehmen, also zu den *LOCs* zählen, da diese Dateien die Präsentationsschicht der Anwendung bilden.

Die von Thomas McCabe definierte *CCN-Metrik* misst die strukturelle Komplexität einer Methode oder einer Funktion. Die Metrik kann für alle Programmiersprachen erstellt werden. Jede Methode und Funktion hat per Definition der Metrik eine Komplexität mit dem Wert 1. Zusätzlich erhöht jeder Entscheidungspunkt in der Methode oder Funktion die Komplexität um den Wert 1. Entscheidungspunkte sind alle bedingten Anweisungen, wie z. B. *if then...else* oder *while*, aber auch *catch*-Blöcke zählen in die *CCN*-Metrik.

```
public boolean isValid(Object obj){
    if(obj==null) {
        return false;
    }else{
        return true;
    }
}
```

Außerdem werden alle logischen Verknüpfungen in den bedingten Anweisungen hinzugezählt, z. B. erhöhen alle *or*- oder *and*- ▶

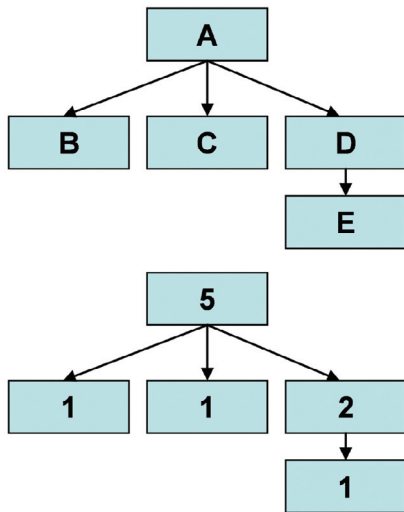


Abb. 1: Durchschnittliche Abhängigkeit der Komponenten (ACD)

Anweisungen die Komplexität und somit die CCN. In Java erhöhen die finally-Blöcke nicht die CCN-Metrik, weil diese auf jeden Fall ausgeführt werden.

Dieser Quellcode-Ausschnitt hat eine CCN von 3; dieser Wert ergibt sich wie folgt: ein if plus ein else plus 1 = 3. Die CCN-Metrik sollte mit der Anzahl der Testfälle auf Methodenebene (Unit-Test) übereinstimmen. Im Beispiel müssen für den if-Teil und für den else-Teil jeweils ein Test implementiert werden und ein Test ist notwendig für den Methodenaufruf selbst. In der Praxis hat sich 20 als zulässiger Grenzwert für die CCN pro Methode bewährt. Ein hoher CCN deutet auf eine komplexe Methode hin, die schwerer zu ändern ist und einen hohen Testaufwand erzeugt.

Mit Hilfe eines Werkzeugs, z.B. „Cobertura“ (vgl. [Cob]), und Unit-Tests kann die prozentuale *Testabdeckung* ermittelt werden. Sie gibt an wie viel Prozent des Quellcodes durch Unit-Tests abgedeckt werden. Ist die Abdeckung nicht bei 100%, so müssen gegebenenfalls weitere Testfälle in die Unit-Tests aufgenommen werden. Auch wenn eine komplette Abdeckung aufgezeigt wird, kann sich zusammen mit der CCN ein anderes Bild zeigen. D.h. der Entwickler muss nicht nur seine Abdeckung überprüfen, sondern auch die CCN mit der Anzahl der implementierten Testfälle. Im Idealfall sollten die Anzahl der Testfälle mit der CCN übereinstimmen.

Der Grad der *Abhängigkeit* einer Komponente von einer anderen wird mit Hilfe der ACD- und der rACD-Metrik gemessen. Die ACD ist die mittlere Anzahl von Abhängigkeiten einer Komponente zu anderen Komponenten. Eine Komponente kann direkt oder indirekt von anderen Komponenten

abhängig sein. Die Anzahl der Abhängigkeiten einer Komponente ist die Summe über alle Abhängigkeiten der direkt verwendeten Komponenten. Jede Komponente ist auch von sich selbst abhängig und erhöht daher die Abhängigkeit um eins.

Abbildung 1 zeigt ein Beispiel: Die Komponente A ist direkt abhängig von B, C und D. A ist indirekt abhängig von E. B, C und E haben eine Abhängigkeitswert von 1. D hat einen Wert von 2 (1 für sich selbst plus 1 aus der Abhängigkeit von E). A hat somit eine Abhängigkeit von 5. Die ACD-Metrik wird gebildet, indem die Summe der Abhängigkeiten aller Komponenten durch die Anzahl der Komponenten geteilt wird. Im Beispiel ergibt sich $ACD=10/5=2$. An der ACD-Metrik lässt sich direkt der Kopplungsgrad ablesen. Diese Metrik lässt sich daher nutzen, um die Qualität der Architektur zu bewerten.

Die rACD ist die relative Abweichung zwischen der ACD-Metrik vor und der ACD-Metrik nach einer Veränderung der Abhängigkeiten, die sich z.B. durch Anpassungen der Architektur ergeben. Hierfür wird der alte ACD-Wert zu dem neuen ACD-Wert in Relation gesetzt. Je größer die Abweichung und je höher der Wert der rACD-Metrik ist, um so stärker hat sich die Veränderung der Abhängigkeiten auf die Gesamtzahl der Abhängigkeiten im System ausgewirkt.

Die ACD-Metrik kann ebenfalls für Klassen gebildet werden, doch ist diese Granulierung häufig zu fein. In der Praxis hat sich gezeigt, dass eine ACD auf Basis der Komponenten ausreicht. Mehr Informationen zu der ACD-Metrik sind in [Zit07] zu finden.

Auf Basis der Abhängigkeiten sind noch die Metriken *Instabilität (I)* und *Abstraktheit (A)* definiert. Zur Bestimmung von I und A werden zunächst die eingehenden und die ausgehenden Abhängigkeit eines Pakets gebildet. Die ausgehende Abhängigkeit (*Efferent Coupling, Ce*) gibt die Anzahl verwendeten Pakete an. Die eingehende Abhängigkeit (*Ca*) gibt die Anzahl Verwendungen in anderen Paketen an. Diese Metriken wurden von Robert C. Martin entwickelt.

Die Instabilität ist definiert als $I= Ce/(Ce+Ca)$ und liegt zwischen 0 und 1. Liegt der Wert von I nahe bei 0, ist das Paket stabil. Ist der Wert in der Nähe von 1, ist das Paket instabil, d.h. es wird sehr häufig verwendet. Die Instabilität ist ein Indikator für den Grad der Anfälligkeit eines Pakets gegenüber Veränderungen (hohe Instabilität), bzw. zeigt sie an, inwieweit die

Änderungen eines Pakets andere beeinflusst (niedrige Instabilität).

Die Abstraktheit ist festgelegt als das Verhältnis zwischen der Anzahl abstrakter Klassen (und Interfaces) (Na) und der Anzahl aller Klassen (Nc). Dabei gilt: $A=Na/Nc$.

Aus Instabilität und Abstraktheit ergibt sich noch die Entfernung (*Distance, D*). Die Distanz ist die Entfernung eines Pakets von der Linie $A+I=1$. Die Distanz kann einen Wert zwischen 0 und 1 annehmen. Sie ist ein Gradmesser dafür, ob A und I ausgewogen sind (siehe Abb. 2).

Liegt der Distanzwert bei 1, ist das Paket entweder nutzlos, weil es komplett abstrakt und instabil ist und somit fast nur andere Pakete verwendet, oder es verursacht viel Mühe, weil es nur konkrete Implementierungen enthält, aber stabil ist und somit von vielen anderen Paketen verwendet wird. Wird das „mühevoll“ Paket geändert, betrifft dies viele andere Pakete direkt. Daher sollte ein Paket um so abstrakter sein, je stabiler es ist.

Die Metriken Ca , Ce , A , I und D lassen sich auch auf Komponentenebene ermitteln und bilden dann die Abhängigkeiten zwischen den Paketen ab.

Eine sehr gute Zusammenstellung verschiedener technischer Metriken und eine Darstellung, wie mit Hilfe von Metriken die Testbarkeit von Anwendung bewertet und verbessert werden kann, ist in [Jun03] zu finden.

Nicht-technische Metriken im Detail

Zur Ergänzung noch die Vorstellung von einigen wichtigen nicht-technischen Metriken. Anforderungen werden häufig in Form von Use-Cases formuliert. Die *Anzahl der Use-Cases* ist somit ein geeignete Metrik, um die Größe und den *Fertigstellungsgrad* zu messen. Die Größe ist die Anzahl aller formulierten Use-Cases, der *Fertigstellungsgrad* ist die Anzahl aller im System implementierten Use-Cases. Use-Cases lassen sich häufig einer fachlichen Komponente zuordnen. Über die durchschnittliche Zahl Use-Cases pro Komponente lassen sich große, komplexe Komponenten identifizieren.

Die Komplexität der beschriebenen Funktion ist von Use-Case zu Use-Case sehr unterschiedlich. Das System umfasst auch Implementierung von Anforderungen, die aus nicht-funktionalen Anforderungen entstehen. Das heißt die reine Anzahl der Use-Cases ist nicht ausreichend.

Funktionspunkte (*Function Points*) und Widget-Punkte (*Widget Points*) sind daher

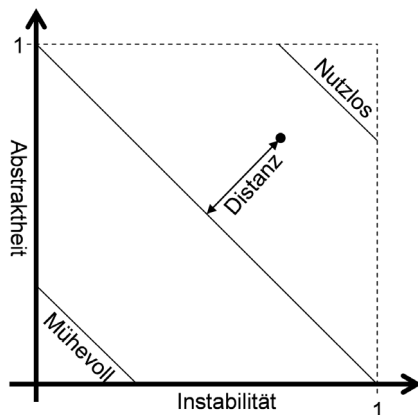


Abb. 2: Zusammenhang zwischen Abstraktheit, Instabilität und Distanz

zwei Möglichkeiten, um Use-Cases weiter herunter zu brechen und die Messungen zu verfeinern. Beide Möglichkeiten werden unter anderem in [Oes01] beschrieben. Sie können verwendet werden, um Schätzungen vorzunehmen und die Fertigstellung zu beobachten. Für die Funktionspunkte wird jede dem Benutzer zugängliche Funktion gezählt. Besser geeignet sind oft die *Widget*-Punkte, die anhand der Benutzungsoberfläche (GUI) bestimmt werden. Jedes Element der GUI (*Widget*) wird gezählt und gewichtet. Die *Widget*-Punkte sind eine feinere Metrik als die Funktionspunkte.

Der *Aufwand* in Stunden oder Tagen kann relativ leicht bestimmt werden, er wird in jedem Projekt für das Projekt-Controlling benötigt. Die aufgewendeten Zeiten für eine projektrelevante Tätigkeit werden gemeldet. Eventuell ist es notwendig, Zeiten und Aufwände doppelt zu erfassen: einmal für die Tätigkeiten im Projekt und dann für die kaufmännische Abrechnung. Der Aufwand dient zusammen mit dem Fertigstellungsgrad dazu, den Restaufwand abzuschätzen und zu kontrollieren, ob eine termingerechte Fertigstellung möglich ist.

Wird der benötigte Aufwand pro Paket oder Komponente erfasst, so kann man daraus den durchschnittlichen Aufwand pro Klasse ableiten und ermitteln, welche Systemteile besonders viel Aufwand verursachen. Es ist zu klären, warum diese außerhalb des Rahmens liegen. Dies kann ein Indikator für schlechte Qualität sein. Der Aufwand pro Klasse kann auch für eine Aufwandsschätzung verwendet werden (vgl. [Lor93]). Diese Art, den Aufwand abzuschätzen, ist sehr ungenau, da der Aufwand durch viele weitere Faktoren getrieben wird, wie beispielsweise die Erfahrung des Entwicklers und die Komplexität.

Die Qualität der Implementierung kann sehr gut mit Hilfe von Metriken bestimmt werden, die in den verschiedenen Testläufen (Integrationstest, Systemtest) erzeugt werden. Dies sind die *Anzahl der erfolgreichen Testfälle*, die *Anzahl der fehlgeschlagenen Testfälle*, die *Anzahl der entdeckten Fehler*. Die *Fehlerdichte*, also die Anzahl der Fehler pro tausend LOC, ist ein guter Indikator dafür, ob die Intensität der Tests erhöht werden muss oder reduziert werden kann. Sinkt die Fehlerdichte im zeitlichen Verlauf deutlich ab, so kann die Testintensität reduziert werden. Aus der Anzahl erfolgreicher Testfälle kann abgeleitet werden, ob sich Änderungen stark auf die Qualität auswirken haben. Fällt die Anzahl stark ab, sind aufwändige Fehlerbehebungen notwendig.

Nutzt man ein Tool zur systematischen Erfassung von Anforderungen und Änderungen (im Notfall reicht auch eine Tabelle in Excel), so kann mit Hilfe der *Anzahl der Änderungen* und deren Häufigkeit festgestellt werden, wie stabil die Anforderungen sind. Bricht man die Änderungshäufigkeit auf Subsysteme herunter, so können kritische Komponenten identifiziert werden, die z.B. sorgfältig getestet werden müssen. Die Anzahl der Änderungen sagt etwas darüber aus, wie gut die Anforderungen spezifiziert sind und ob die Anforderungen und das Design von den Programmierern verstanden worden sind.

Zusammen mit der Fehlerdichte lassen sich mit der Änderungshäufigkeit kritische Komponenten identifizieren. Mehr als die Hälfte aller Fehler treten in weniger als 10% des Quellcodes aus [Jon96]. Erfahrungen aus verschiedenen Projekten bestätigen diesen Zusammenhang. In der Praxis hat sich ferner gezeigt, dass eine hohe Fehlerdichte mit einer hohen Änderungshäufigkeit korreliert.

Werkzeuge zur Bestimmung von Metriken

Ein wichtiges Kriterium für den Einsatz von Werkzeugen ist die Automatisierung der Bestimmung der Metriken. Speziell Metriken, die sich auf den Quellcode beziehen, sollten mit Werkzeugen erstellt werden, die sich in den Entwicklungsprozess einbetten lassen, ohne diesen zu behindern.

Das Open-Source Projekt „XRadar“ (vgl. [XRa]) hat sich als Plattform zur Bestimmung von Metriken bewährt. Es lässt sich gut in einen automatischen Kompilierungsprozess in die sogenannten Build- oder Make-Skripte integrieren. Die Ergebnisse werden als Bericht in HTML-Form erstellt, der sich dann wiederum gut in eine Projekt-Portal-

Umgebung integrieren lässt. Ein praktischer Einsatz von XRadar ist in [Kva05] beschrieben. XRadar stellt sowohl den aktuellen Zustand der Metriken als auch deren zeitlichen Verlauf dar.

In XRadar selbst lassen sich andere Werkzeuge integrieren, die weitere Metriken an XRadar liefern. Hier ein Überblick über die wichtigsten Werkzeuge:

- **PMD:** PMD (vgl. [PMD]) analysiert den Quellcode und sucht nach häufig auftauchenden Fehlerquellen, wie etwa leere *Try-catch*-Blöcke, die nicht optimale Verwendung von Strings oder nach nicht verwendeten Variablen. PMD sucht ferner nach duplizierten Quellcode-Blöcken. Die Regeln, deren Einhaltung PMD prüfen soll, sind umfangreich und einfach konfigurierbar.
- **CheckStyle:** Das Werkzeug unterstützt den Entwickler beim Schreiben von Quellcode, das den gesetzten Programmierrichtlinien (Namenskonventionen, Formatierungen etc.) unterstützt. CheckStyle (vgl. [Che]) überprüft automatisch die Einhaltung der Richtlinien. Diese können an jede Richtlinie angepasst werden oder es kann in der Standardkonfiguration die Einhaltung der „Sun Code Conventions“ (vgl. [Sun]) geprüft werden.
- **JavaNCSS:** Hierbei handelt es sich um ein einfaches Werkzeug, mit dessen Hilfe die Anzahl Codezeilen, die NCSS- und die CCN-Metrik ermittelt werden. Die Werte werden absolut und als Durchschnitt berechnet und auf Gesamt-, Paket-, Klassen- und Methodenebene ermittelt (vgl. [Jav]).
- **JJDepend:** JJDepend (vgl. [JDe]) ermittelt anhand der kompilierten Klassendateien verschiedene Design-Metriken. Berechnet werden die Metriken Ca, Ce, Abstraktheit, Instabilität und Distanz. Außerdem werden die Klassen auf eventuell vorhandene zyklische Abhängigkeiten untersucht.
- **Cobertura:** Mit diesem Tool lassen sich die Teile des Quellcodes ermitteln, die durch Tests abgedeckt sind. Im Gegensatz zu anderen Open-Source-Werkzeugen, ist die Ausgabe sehr leserlich und übersichtlich. Es lässt sich auch in Serverumgebung integrieren, um dort die Abdeckung zu ermitteln (vgl. [Cob]).

Alle genannten Werkzeuge bilden zusammen mit XRadar einen umfangreichen Werkzeugkasten, der sich auf verschiedene Belange eines Systems und eines Projekts ▶

anpassen lässt. Mit den vorgestellten Werkzeugen lassen sich Metriken für Java-Systeme erstellen. Ähnliche Werkzeuge existieren auch für andere Sprachen, wie etwa C#. Einmal aufgesetzt helfen die Werkzeuge, die wichtigsten Metriken und Zustandsberichte über ein System zu erstellen. Das Aufsetzen eines solchen Werkzeugkastens lässt sich innerhalb weniger Tage realisieren. Mit dem Aufsetzen ist die Arbeit aber nicht getan – man muss die Metriken verstehen und ihre Zusammenhänge erkennen. Wichtig ist es auch, sich vorher im darüber im Klaren zu sein, was man messen möchte und welche Metriken mit welchem Werkzeug erstellt werden können.

Speziell die nicht-technischen Metriken lassen sich häufig nicht automatisiert erstellen. Im besten Fall sind für das Fehlermanagement oder das Anforderungsmanagement Werkzeuge im Einsatz, die eine konfigurierbare Berichtsfunktion anbieten. Oft ist manuelle Nacharbeit erforderlich. Einen guten Ansatz bietet das kommerzielle Produkt „Polarion“ (vgl. [Pol]) für Subversion, das aufgrund der einheitlichen Datenbasis schon eine Vielzahl an Metriken anbietet.

Einsatz in der Praxis

Die Lufthansa System AS sollte im Auftrag eines Kunden ein Softwareprodukt einer indischen Firma begutachten. Das Softwareprodukt sollte analysiert und bewertet werden, um die Kundenanforderungen an die Qualität der Software zu prüfen. Der Schwerpunkt der Analyse wurde auf die Bewertung der Technologie gelegt. Die fachliche Bewertung (Abdeckung der Anforderungen, Identifikation notwendige fachlicher/funktionaler Änderungen) wurde bereits in einer separaten Analyse durchgeführt. Dieses Softwareprodukt ist in Java implementiert, setzt Java-EE-Technologien ein und verwendet verschiedene Open-Source-Werkzeuge.

Konnten die Punkte „Dokumentation“ und „Methoden & Tools“ anhand von Vorgehensmodellen und Erfahrungswerten noch durch eine manuelle Prüfung gut auf Vollständigkeit und Qualität geprüft werden, war dies bei der „Software-Architektur, Objekt-Modell und Implementierung“ schwieriger.

Aufgrund des knapp bemessenen Zeitrahmens von nur vier Monaten, in dem die Begutachtung abgeschlossen werden musste, war eine komplette manuelle Überprüfung (Review) der vorhandenen Quellcodes und das gesamten Objektmodells nicht möglich. Im Anbetracht des Umfangs des

Quellcodes war eine umfassende Einarbeitung nicht möglich. Die LOC-Anzahl überstieg die 6-Millionen-Marke, die Anzahl der Klassen lag bei weiter über 7.000. Um die technische Qualität der Anwendung beurteilen zu können, wurden Methoden, Metriken und Werkzeuge verwendet, die sich in anderen Projekten der Lufthansa Systems AS bewährt hatten. Anhand der wichtigsten Metriken, wie LOC, Anzahl Klassen, CCN, Ca, Ce, Code-Redundanz, konnte aufgezeigt werden, dass die Software an verschiedenen Stellen Schwächen aufweist.

Mit Hilfe der genannten Werkzeuge PMD, CheckStyle, JDepend, JavaNCSS und XRadar war es möglich, in kürzester Zeit die benötigten Metriken zu erstellen. Die ermittelten Metriken der Anwendung wurden verglichen mit Metriken aus anderen Projekten der Lufthansa Systems AS mit ähnlichem technologischem Kontext und ähnlichem Umfang.

In einem Maßnahmenkatalog wurden Verbesserungsvorschläge zusammengetragen. Die Umsetzung der Maßnahmen, sofern sie Auswirkungen auf die Implementierung haben, werden fortlaufend mit Hilfe der Metriken bewertet und beobachtet.

Fazit

Die vorgestellten Metriken können einen Gesamteindruck über den Zustand des Systems vermitteln oder gezielte Untersuchungen untermauern. Metriken können helfen, Problemstellen im System zu identifizieren. Die korrekte Umsetzung der logischen Architektur lässt sich anhand des Quellcodes mit Hilfe von Metriken prüfen. Schwächen in der Implementierung können identifiziert werden. Metriken können im laufenden Projekt die Überwachung der technischen Qualität unterstützen. Und sie sind ein Werkzeug, um Indikatoren für Qualität zu finden.

Metriken messen die Veränderungen im System und man kann Ableitungen machen, ob sich Veränderungen positiv oder negativ auf das System auswirken.

Eine Metrik allein hat wenig Aussagekraft, Metriken müssen immer in Bezug zueinander gesehen werden.

Mit Hilfe der erwähnten Werkzeuge lässt sich die Erstellung der Metriken leicht automatisieren. Wie dargestellt, lassen sich bestimmte Metriken nur manuell erstellen, hier sind gegebenenfalls projektspezifische Erweiterungen der Werkzeuge notwendig.

Ein beobachtetes Verhalten im Projekt ist die Anpassung der Projektmitarbeiter an die Metriken. Es wurde im Team so modelliert

und programmiert, dass die Metriken gut aussahen. Im manuellen Review wurden aber Probleme entdeckt, die sich anhand der Metriken nicht nachweisen ließen.

Eine wichtige Empfehlung von mir lautet, Personen nicht anhand der nackten Zahlenwerte einer Metrik zu bewerten und ihre Fehler mit Hilfe von Metriken bloßzustellen. Wichtiger ist es, anhand der Metriken zu identifizieren, wo die Probleme liegen, welche Ursachen die Probleme haben, um dann gezielt die Ursachen zu beheben.

Andersherum sollte man sich immer zuerst überlegen, was man messen möchte, bevor man eine konkrete Metrik bestimmt. Die Auswahl der Metriken sollte sich immer der Projektphase anpassen.

Metriken können helfen, die Komplexität zu zerlegen und sie kontrollierbar und beherrschbar zu machen. ■

Literatur & Links

[Che] Checkstyle 4.3, siehe: checkstyle.sourceforge.net

[Cob] Cobertura, siehe: cobertura.sourceforge.net/

[DeM] T. DeMarco, Controlling Software Projects: Management, Measurement, and Estimates, Prentice Hall 1982

[Fen97] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, Thomson Learning, 1997

[Jav] JavaNCSS, siehe: www.kclee.com/clemens/java/javancss/

[JDe] JDepend, siehe: clarkware.com/software/JDepend.html

[Jon96] T.C. Jones, Applied Software Measurement, McGraw-Hill, 1996

[Jun03] S. Jungmayr, Improving testability of object-oriented systems, dissertation.de, Verlag im Internet GmbH, 2003

[Kva05] K. Kvam, R. Lie, D. Bakkelund, Legacy System Exorcism by Pareto's Principle, in.: Proc. of: OOPSLA 2005, siehe: xradar.sourceforge.net/resources/legacy_system_excorsism_by_paretos_principle.pdf

[Lor93] M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice Hall, 1993

[Oes01] B. Oesterreich, Erfolgreich mit Objektorientierung, Oldenbourg Verlag, 2001

[PMD] PMD, siehe: sourceforge.net/projects/pmd

[Pol] Polarion for Subversion, siehe: www.polarion.org

[Sun] Sun Code Conventions, siehe: java.sun.com/docs/codeconv/

[Wik07] Wikipedia, Metrik, siehe: de.wikipedia.org/wiki/Metrik

[Xra] XRadar, siehe: xradar.sourceforge.net

[Zit07] A. v. Zitzewitz, Architektur Management, in: Java Magazin 2/07