Seminar „Moderne Softwareentwicklung“ SS 2005

Software-Metriken

Wolfgang Globke

Betreuer: Olaf Seng

15. April 2013

Zusammenfassung

Software-Metriken liefern Kennzahlen von Programmelementen oder Systemen, die bei der Beurteilung der inneren Qualität eines Software-Produktes helfen sollen, also derjenigen Qualitätsaspekte, die für den Software-Entwickler relevant sind. Da eine Metrik nur eine ganz bestimmte Eigenschaft des Produktes misst, muss man ihren Wert geeignet interpretieren, um Aussagen über die Qualität machen zu können. Für eine Metrik ist also experimentell nachzuweisen, dass sie in zuverlässiger Weise das misst, was sie messen soll. Wir werden hier einige der klassischen Metriken und einige Metriken für objektorientierte Systeme kennenlernen. Insbesondere bei letzteren werden wir sehen, dass sie dann besonders mächtig sein können, wenn man sie geschickt kombiniert und in einem geeigneten Qualitätsmodell auswertet.

Inhaltsverzeichnis

1 Einführung

Eines der Ziele jeglicher Software-Entwicklung muss es sein, möglichst fehlerfreie und qualitativ hochwertige Produkte zu entwickeln. Es ist die Aufgabe der Qualitätssicherung, die wünschenswerten Qualitätsziele zu ermitteln und geeignete Maßnahmen zu treffen, um diese Ziele zu erreichen. Dies umfasst insbesondere das Prüfen von Produkten und Zwischenergebnissen auf vorher festgelegte Qualitätsmerkmale. Software-Metriken stellen dabei ein wichtiges Hilfsmittel der Qualitätssicherung dar. Sie ermöglichen das Auffinden kritischer Teile des entwickelten Systems und können sogar konkrete Hinweise auf Verbesserungsansätze liefern, wenn sie nur in einem geeigneten Rahmen zur Anwendung kommen.

In Kapitel 2 werden wir zunächt in sehr allgemeiner Form erklären, was wir unter Metriken verstehen und welche Eigenschaften sie haben können. Anschließen betrachten wir in den Kapiteln 3 und 4 einige Beispiele von verbreiteten Metriken. In Kapitel 5 werden wir sehen, wie man zu einem konkreten System von Metriken kommt, das es ermöglicht, Schwächen im Entwurf eines objektorientierten Systems aufzuspüren.

2 Metriken in der Qualitätssicherung

In diesem Abschnitt werden wir Metriken einführen, ihre Rolle bei der Qualitätssicherung von Software erläutern und einige Eigenschaften von Metriken betrachten.

Wenn wir im Folgenden von **(Programm-)Elementen** sprechen, so meinen wir damit die grundlegenden Strukturen einer Programmiersprache, etwa Klassen bei objektorientierten Sprachen oder Funktionen und Module bei imperativen Sprachen. Wir identifizieren dabei ein Programmelement stets mit seinem Quellprogramm. Das **System** entspricht der Gesamtheit aller Programmelemente.

2.1 Qualität in der Softwaretechnik

In der Qualitätsicherung ist man naturgemäß daran interessiert, die **Qualität** eines Software-Produktes in irgendeiner Weise greifbar zu machen. Eine formale Definition scheint wenig sinnvoll, da dieser Begriff sehr abstrakt ist und es viele verschiedene Auffassungen darüber gibt, was Qualität ist. So ist es etwa denkbar, dass ein Programm technisch hervorragend umgesetzt ist, aber seine Funktionalität nicht den Ansprüchen des Kunden genügt. Die Entwickler und die Abnehmer des Programms würden seine Qualität dementsprechend unterschiedlich einschätzen.

In dieser Arbeit betrachten wir die **innere Qualität**, d.h. die Qualität vom Standpunkt des Entwicklers aus gesehen. Die zentrale Frage ist dabei, wie leicht sich die Entwicklungsarbeit an dem Software-Produkt heute und zukünftig gestaltet. Es geht also beispielsweise um Aspekte wie Verständlichkeit und Wiederverwendbarkeit der einzelnen Teile des Systems.

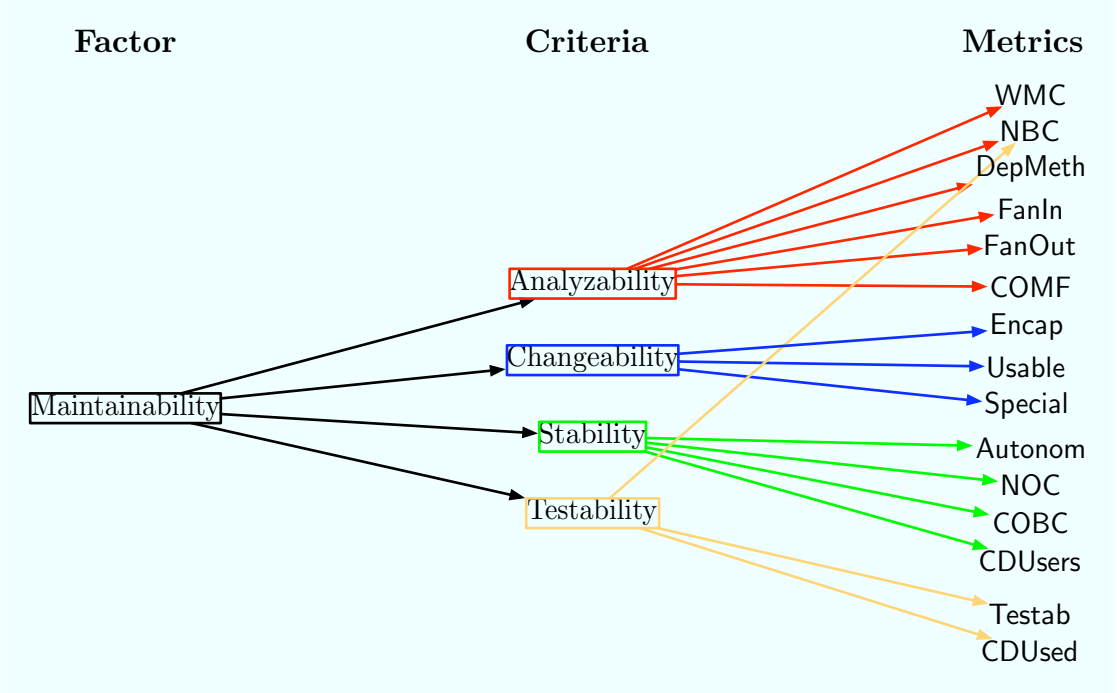
2.2 Qualitätsmodelle und Metriken

Um die Qualität eines Software-Produktes beurteilen zu können, benötigt man ein **Qualitätsmodell**, in dem die einzelnen Aspekte der Qualität greifbar gemacht werden.

Ein solches Qualitätsmodell ist das **Factors-Criteria-Metrics-Modell**, kurz **FCM-Modell**. Hierbei werden einzelne relevante **Qualitätsmerkmale** (engl. *factors*) herausgearbeitet, die widerspiegeln sollen, wie die Qualität vom Benutzer empfunden wird.

Die Qualitätsmerkmale sollen sich wiederum in leichter zu messenden **Teilmerkmale** (engl. *criteria*) aufteilen lassen. Die einzelnen Qualitätsmerkmale und ihre Teilmerkmale kann man sich dabei in einer baumähnlichen Struktur angeordnet vorstellen.

Die Teilmerkmale sollen dabei in Beziehung zu gewissen messbaren Eigenschaften eines Programmelementes stehen. **Metriken** sind Funktionen, die bestimmten Eigenschaften eines Programmelementes, das in Form seiner Quelldatei vorliegt, numerische Werte zuweisen. Die folgende Grafik aus [Mar02] zeigt dies exemplarisch für das Qualitätskriterium „Maintainability“.



Bei der Auswertung wird man zuerst für jedes Teilmerkmal einen Wert berechnen, der sich als Funktion aus den von den Metriken gemessenen Werten ergibt. Beim Aufstellen dieser Funktion muss man sich auf die Erfahrung der Entwickler verlassen. Diese müssen die gemessenen Werte interpretieren können, um Schlüsse daraus zu ziehen, in welcher Weise ein Teilmerkmal ausgeprägt ist. Daraus leitet man eine heuristische Bewertungsfunktion für das entsprechende Merkmal ab. Anhand der Werte dieser Funktion kann man dann ein Qualitätsurteil für das Teilmerkmal bestimmen. In analoger Weise errechnet man aus den Bewertungen der Teilmerkmale eine Bewertung des übergeordneten Qualitätsmerkmals.

Die Ergebnisse für die einzelnen Qualitätsmerkmale können in einem Bericht zusammengefasst werden, aus dem zu entnehmen ist, in welchen Bereichen die Qualität zu wünschen übrig lässt und welche Elemente dafür verantwortlich sind. In Abschnitt 5.2 werden wir ein modifiziertes FCM-Modell kennenlernen, das zusätzlich noch Informationen über Verbesserungsansätze liefert.

2.3 Eigenschaften von Metriken

Unsere Definition einer Metrik ist recht vage. Dies kommt daher, dass wir Metriken nicht durch ihre Eigenschaften definiert haben, sondern durch die Aufgabe, die sie bei der Qualitätsbeurteilung erfüllen sollen. Wir betrachten nun einige Eigenschaften, die Metriken idealerweise erfüllen sollten, um ihrer Aufgabe gerecht zu werden.

Will man eine bestimmte Eigenschaft E messen, so ist ein intuitives Verständnis dieser Eigenschaft notwendig, um eine **empirische Ordnung** $\prec$ zu erhalten, die das Verhältnis zweier (vergleichbarer) Programmelemente P_1 und P_2 bezüglich dieser Eigenschaft beschreibt. Ist die gemessene Eigenschaft beispielsweise „Länge des Programms in Codezeilen“ (LOC, siehe Abschnitt 3.1), so würde man als empirische Ordnung „ist länger als“ erhalten.

Ist m eine Metrik, die E misst, so ist es wünschenswert, dass die gemessenen numerischen Werte $\mathsf{m}(P_1)$ und $\mathsf{m}(P_2)$ die empirische Ordnung zwischen P_1 und P_2 wiedergeben, oder formal ausgedrückt

$$\mathsf{m}(P_1) < \mathsf{m}(P_2) \qquad \Leftrightarrow \qquad P_1 \prec P_2.$$

Dies ist die sogenannte **Darstellungsbedingung**, weitere Ausführungen dazu finden sich in [Fen94]. Die **Validierung** von m ist die empirische Überprüfung, ob m die Darstellungbedingung erfüllt. Das genaue Vorgehen bei der Validierung hängt nicht zuletzt davon ab, welchen Zweck die Metrik erfüllen soll. Eine angemessene Erläuterung wäre hier zu umfangreich, weswegen wir dafür auf [BBM95], [Fen90] und [KC85] verweisen.

Viele gängige Software-Metriken erfüllen die Darstellungsbedingung jedoch nicht. Dies bedeutet keineswegs, dass diese Metriken unbrauchbar sind, jedoch sind die von ihnen gelieferten Messungen oft nur als grobe Anhaltspunkte zu sehen. Beispiele hierfür sind die Metriken aus Kapitel 3. Da diese Metriken aber oft sehr einfach umzusetzen und für den praktischen Bedarf „hinreichend genau“ sind, erfreuen sie sich dennoch großer Beliebtheit.

In [Bal98] werden weitere wünschenswerte Eigenschaften von Metriken aufgezählt.

- **Objektivität** Die Metrik sollten unabhängig von subjektiven Einflüssen des Messenden sein.
- **Zuverlässigkeit** Die Metrik liefert bei gleichen Voraussetzungen immer das gleiche Ergebnis.
- **Normierung** Die Metrik bildet auf eine Skala ab, die den Vergleich verschiedener Messergebnisse erlaubt.

- **Wirtschaftlichkeit** Die Messung sollte mit möglichst geringem Aufwand durchzuführen sein.
- **Nützlichkeit** Die gemessene Eigenschaft sollte in einem Zusammenhang mit einem Qualitätsmerkmal stehen und es sollte (heuristische) Regeln dafür geben, wie man aus dem Messwert auf die Qualität zurückschließt.

Wie stark eine dieser Eigenschaften für eine Metrik ausgeprägt sein muss, ist auch vom Verwendungszweck abhängig.

2.4 Axiomensysteme für Metriken

Mehrere Autoren haben versucht, die Ansprüche, die man an Metriken hat, durch ein Axiomensystem zu formalisieren. Das attraktive an diesem Ansatz ist, dass man logisch überprüfbare Kriterien erhält, eine Metrik zu validieren (und sich im bequemsten Fall die experimentelle Validierung ersparen kann). Am bekanntesten davon sind wohl die von Weyuker in [Wey88] vorgeschlagenen Axiome. Diese Axiomensysteme beschreiben aber bestenfalls *notwendige* und keine hinreichenden Bedingungen für Metriken, so dass sie in der Regel weder bei der Bestimmung neuer Metriken noch bei der Validierung bereits bekannter Metriken hilfreich sind. Die meisten nichttrivialen Axiomensysteme haben sich auch als in sich widersprüchlich herausgestellt, siehe [Fen94] und [KS97].

2.5 Klassifikation von Metriken

Es gibt verschiedene Dimensionen, nach denen sich Metriken klassifizieren lassen.

1. Eine **direkte** Metrik $\mathsf{m}^{\mathrm{dir}}$ berechnet einen Wert $\mathsf{m}^{\mathrm{dir}}(P)$ für eine zu untersuchende Eigenschaft eines Programmelementes P. Eine **indirekte** Metrik $\mathsf{m}^{\mathrm{ind}}$ ergibt sich als Funktion $\mathsf{m}^{\mathrm{ind}} = f(\mathsf{m}_1^{\mathrm{dir}}, ..., \mathsf{m}_k^{\mathrm{dir}})$ von direkten Metriken.

   So ist beispielsweise die Anzahl der Fehler in einem Programm ist eine direkte Metrik, aus der man die Anzahl der Fehler pro tausend Quellcodezeilen als indirekte Metrik berechnen kann.

2. Die Metriken unterscheiden sich durch ihren **Abstraktionsgrad**. Welche Stufen es dabei im einzelnen gibt, hängt von dem konkreten Software-Produkt ab bzw. davon, wie es entwickelt wird. So könnte man beispielsweise beim objektorientierten Entwurf folgende Stufen unterscheiden:

   - *Systemebene*: Diese Metriken berücksichtigen die Gesamtheit aller Klassen oder auch Module. Sie sind in der Regel indirekte Metriken, die sich aus Metriken von niederen Abstraktionsstufen berechnen lassen.
   - *Modulebene*: Lässt sich das ganze System in Module aufteilen, so kann man Teile davon betrachten, die logisch zusammenhängen.

- *Klassenebene*: Auf dieser Ebene werden Zusammenhänge, wie etwa die Vererbungsstrukturen, zwischen den Klassen innerhalb eines Moduls betrachtet.
- *Methodenebene*: Auf dieser Ebene werden Funktionalität, Umfang und Komplexität einer einzelnen Klasse untersucht.

2.6 Schwierigkeiten beim Einsatz von Metriken

Beim Einsatz von Metriken gibt es eine Reihe von Schwierigkeiten, die teilweise durch unsachgemäße Anwendung zustandekommen, teilweise aber auch fundamentale Probleme sind.

Das erste ist sicherlich der Fall, wenn man ungenau definierte Metriken verwendet oder nicht in der Lage ist, ein Messergebnis sinnvoll zu interpretieren. Wie in [Jon94] beschrieben ist es beispielsweise sinnlos, die Lines-Of-Code-Metrik (Abschnitt 3.1) zur Beurteilung zweier Software-Produkte heranzuziehen, die in sehr unterschiedlichen Programmiersprachen wie Assembler und Ada entwickelt wurden.

Zu den fundamentalen Problemen gehört beispielsweise, dass man für eine bestimmte Eigenschaft, die man messen will, keine sinnvolle (berechenbare) Metrik finden kann. So gibt es keine bekannte Metrik, die die „Komplexität“ eines Programms misst und dabei *sämtliche* Aspekte dieses Begriffs berücksichtigt. Vielmehr gibt es viele Metriken, die nur einen ganz bestimmten Aspekt der Komplexität messen, wie etwa die Metriken aus 3.

3 Einige klassische Metriken

Den sehr allgemein gehaltenen Begriffen aus Abschnitt 2 soll nun durch einige Beispiele Leben eingehaucht werden.

3.1 Lines-Of-Code

Die **Lines-Of-Code-Metrik**, kurz **LOC**, misst die Anzahl der Zeilen im Quellcode eines Programms. Dabei liegt die Annahme zugrunde, dass die Unübersichtlichkeit des Quellcodes mit seiner Länge ansteigt. Es liegt auf der Hand, dass diese Annahme nur eine sehr grobe Näherung der Realität darstellt. Zwei Zeilen Quellcode können sich beispielweise in ihrer Übersichtlichkeit stark unterscheiden, oder eine zusätzliche Kommentarzeile kann die Übersichtlichkeit fördern, zu viele nichtssagende Kommentarzeilen jedoch können kontraproduktiv sein.

Von LOC gibt es daher mehrere Varianten, in denen etwa die Kommentarzeilen nicht berücksichtigt werden oder anstelle der Zeilen die Anweisungen im Quellcode gezählt werden. Vergleicht man also Programmelemente, die mit LOC gemessen wurden, so muss man stets darauf achten, dass die selbe Variante von LOC gemeint ist.

Es ist offensichtlich, dass LOC nur in sehr eingeschränktem Maße dafür geeignet ist, Quellcode in verschiedenen Programmiersprachen zu vergleichen. Jones zeigt in [Jon94] an einen Beispiel, dass die erwarteten Kosten pro Codezeile beim Umstieg von Assembler

auf Ada83 ansteigen, wenn man LOC als Grundlage der Berechnung nimmt, obwohl die Produktivität in Wirklichkeit um ein Vielfaches erhöht wird.

3.2 Die Halstead-Metrik

Unter dem Namen **Halstead-Metrik** fasst man eine Reihe von Kennzahlen zusammen, die Größe und Komplexität des Codes widerspiegeln sollen. Diese Kennzahlen basieren auf der Anzahl der unterschiedlichen im Quellcode verwendeten Operanden und Operatoren und auf der Gesamtzahl der verwendeten Operanden und Operatoren.

Als **Operator** wird jedes Symbol oder Schlüsselwort bezeichnet, dass eine Aktion kennzeichnet, z.B. `+`, `if`, `(`.

Operanden sind alle übrigen Symbole, also Variable, Sprungmarken, Konstanten usw. Zur Verdeutlichung betrachten wir ein kleines Programm, das Fakultäten berechnet. Die Operatoren sind fett gedruckt, die Operanden unterstrichen.

```
int fak(int n) {
    if (n == 0) return 1;
    return n * fak(n - 1);
}
```

Zur Halstead-Metrik gehören folgende Kennzahlen:

$$\begin{aligned} n &= \text{Anzahl der unterschiedlichen Operatoren} \\ m &= \text{Anzahl der unterschiedlichen Operanden} \\ N &= \text{Gesamtzahl der verwendeten Operatoren} \\ M &= \text{Gesamtzahl der verwendeten Operanden} \end{aligned}$$

Daraus ergeben sich weitere Kennzahlen: $M+N$ ist die Länge der Implementierung, $m+n$ entspricht der Größe des verwendeten Vokabulars. Als *Schwierigkeit, ein Programm zu lesen*, ist

$$S = n \cdot \frac{1}{2} \cdot \frac{M}{m}$$

zu verstehen. Der Quotient $\frac{M}{m}$ beschreibt, wie oft ein Operand im Mittel verwendet wird.

Die Halstead-Metrik hat den Vorteil, dass sie sehr leicht zu ermitteln ist. Allerdings gibt es hier ähnliche Probleme wie bei LOC: Die Vergleichbarkeit von Software, die in verschiedenen Programmiersprachen geschrieben wurde, ist im Allgemeinen nicht gegeben, und die Interpretation der Größe S als Schwierigkeit setzt die grob vereinfachende Annahme voraus, dass diese nur von n, m und M abhängt. Dabei wird nur der lineare Aufbau des Quellcodes berücksichtigt, nicht aber Verschachtelungen oder Verzweigungen beim Programmablauf.

3.3 Die McCabe-Metrik

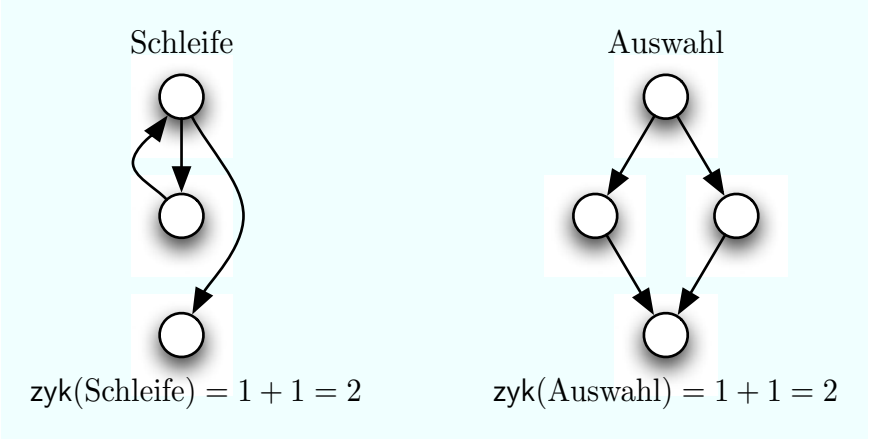
Mit der **McCabe-Metrik** will man die strukturelle Komplexität eines Programmes ermitteln, d.h. man will insbesondere auch Verzweigungen beim Programmablauf berücksichtigen. Dabei orientiert man sich an einem Kontrollflussgraphen $\mathcal{G}$ und drückt die Komplexität durch die sogenannte **zyklomatische Zahl**

$$\mathsf{zyk}(\mathcal{G}) = e - v + 2c,$$

wobei e die Anzahl der Kanten von $\mathcal{G}$ ist, v die Anzahl der Knoten und c die Anzahl der verbundenen Programmelemente ist.

Ein verbundenes Programmelement ist hierbei ein einzelner (Teil-)Kontrollflussgraph, der eine Prozedur oder Funktion darstellen kann. Ein Programm, das ohne Aufruf von Prozeduren oder Funktionen abläuft, würde demnach aus einem einzigen verbundenen Element bestehen (also $c = 1$). In diesem Fall kann man die zyklomatische Zahl allein durch die Anzahl b der Verzweigungen im Graphen bestimmen:

$$\mathsf{zyk}(\mathcal{G}) = b + 1.$$



Auch die McCabe-Metrik zeichnet sich durch ihre leichte Berechenbarkeit aus. Aber auch hier stellt die Reduktion der Komplexität auf eine einzige Zahl eine grobe Vereinfachung dar. Jede Verzweigung im Quellcode trägt im gleichen Maße zur gemessenen Komplexität bei, obwohl sicher nicht jede Verzweigung im gleichen Maße zur intuitiv empfundenen Komplexität des Programms beiträgt.

4 Objektorientierte Metriken

Die klassischen Metriken aus Abschnitt 3 entspringen einer funktionsorientierten Sichtweise der Programmierung.

Bei der objektorientierten Entwicklung kommen aber einige neue Freiheitsgrade hinzu, die die Komplexität eines Systems beeinflussen, wie etwa die Abstraktion in Klassen,

Vererbung, Kapselung und Polymorphismus. Diese Aspekte werden von den klassischen Metriken nur unzulänglich berücksichtigt. Wendet man etwa die McCabe-Metrik auf die Methoden einer Klasse an, so wird man häufig den minimalen Wert $\mathsf{zyk}(\mathcal{G}) = 1$ erhalten, da die meisten Methoden nur wenige und einfach strukturierte Anweisungen enthalten. Die Komplexität ergibt sich vielmehr aus der Anzahl und dem Zusammenspiel der Klassen und Methoden.

Um der veränderten Lage Herr zu werden, wurde eine Vielzahl von auf objektorientierte Systeme spezialisierten Metriken vorgeschlagen. Diese Metriken sind in der Regel einfache Zählmetriken, die bei geeigneter Kombination für verschiedene Messziele von Nutzen sein können. Solche Ziele könnten etwa sein, gegebenen Code besser zu verstehen, wiederverwendbare Elemente zu identifizieren oder Schwächen im objektorientierten Entwurf aufzudecken.

Bei der Auswahl der in den folgenden Abschnitten vorgestellten Metriken orientieren wir uns an [Mar02] und den dort zitierten Quellen. Wir ordnen die Metriken denjenigen Aspekten der Objektorientierung zu, die die Komplexität maßgeblich beeinflussen: *Kopplung*, *Kohäsion*, *Vererbung* und *strukturelle Komplexität*.

4.1 Kopplung

Die **Kopplung** beschreibt die Verbundenheit der Elemente (etwa Klassen oder Pakete) eines Systems. Eine starke Kopplung entspricht starken Abhängigkeiten zwischen den Elementen, was die Wiederverwendung erschwert. Auch die Verständlichkeit wird beeinträchtigt, da für das Verständnis einer Klasse das Verständnis vieler anderer Klassen vorausgesetzt wird. Daher ist eine *niedrige Kopplung* wünschenswert.

- **Coupling Between Objects (CBO)** zählt die Anzahl der Klassen, auf deren Dienst eine bestimmte Klasse zugreift. Dies ist unabhängig davon, wie oft auf eine andere Klasse zugegriffen wird.
- **Number Of Services (NOS)** zählt die Anzahl der Dienste (d.h. der verschiedenen aufgerufenen Methoden) die eine bestimmte Klasse von anderen Klassen in Anspruch nimmt.
- **Number Of Accesses (NOA)** zählt alle Zugriffe einer bestimmten Klasse auf andere Klassen.

4.2 Kohäsion

Unter **Kohäsion** oder **Bindung** verstehen wie den Grad an Verknüpfung zwischen den einzelnen Teilen eines Programmelements (also in der Regel einer Klasse). Eine niedrige Kohäsion lässt erfahrungsgemäß darauf schließen, dass in einer Klasse mehrere Funktionalitäten implementiert wurden. Solche Klassen enthalten in der Regel viele Methoden, die semantisch nicht zusammenhängen. Dies erschwert einerseits das Verständnis der

Klasse, andererseits aber auch die Wiederverwendbarkeit. Daher ist eine *hohe Kohäsion* wünschenswert.

- **Lack Of Cohesion in Methods (LCOM)** soll ein Maß für den Mangel an Kohäsion sein. Es ist definiert durch die Differenz zwischen der Anzahl der Paare von Methoden einer Klasse, die eine gemeinsame Instanzvariable verwenden, und der Anzahl der Paare, die das nicht tun. Wie sich herausstellt, ist diese Definition nicht ganz zufriedenstellend, weshalb inzwischen mehrere Verbesserungen für diese Metrik vorgeschlagen wurden. Eine dieser Verbesserung legt den Begriff einer *perfekten Kohäsion* zugrunde und misst LCOM als prozentualen Anteil an der perfekten Kohäsion.
- **Tight Class Cohesion (TCC)** ist die Anzahl derjenigen Methoden einer Klasse, die mindestens eine gemeinsame Instanzvariable verwenden.
- **Loose Class Cohesion (LCC)** ist die Anzahl derjenigen Methoden einer Klasse, die mindestens eine gemeinsame Instanzvariable verwenden *oder* die durch Aufruf anderer Methoden indirekt eine gemeinsame Instanzvariable verwenden.

4.3 Vererbung

Es ist wichtig, die Vererbungsstruktur zu untersuchen, wobei man die Tiefe der Vererbungshirarchie und die Anzahl der Knoten misst. Eine Idee dahinter ist, dass eine gut strukturierte Klassenhierarchie eher ein Wald von Vererbungsbäumen ist, als eine lange Kette von erbenden Klassen. Anders ausgedrückt hat eine Klasse, die in der Vererbungshierarchie sehr tief steht, eine größere Chance, fehlerhaft zu sein, weil sie viel Funktionalität von ihren Vorgängern erbt. Die Messung der Vererbungsstruktur kann sowohl auf der Klassen- als auch auf der Systemebene sinnvoll sein.

- **Depth Of Inheritance (DIT)** ist die Anzahl der Oberklassen einer bestimmten Klasse. Auf Systemebene kann man eine gemittelte DIT bestimmen.
- **Number Of Descendants (NOD)** ist die Anzahl aller Klassen, die eine bestimmte Klasse als Oberklasse haben.
- **Reuse Ratio** ist auf der Systemebene das Verhältnis der Anzahl der Oberklassen zur Gesamtzahl aller Klassen. Ein Wert nahe bei 1 deutet auf einen lineare Vererbungskette und folglich auf schlechtes Design hin. Ein Wert nahe bei 0 lässt auf eine sehr flache Vererbungshierarchie schließen.
- **Specialization Ratio** ist das Verhältnis der Anzahl aller Unterklassen zur Anzahl aller Oberklassen.

4.4 Größe und strukturelle Komplexität

Anhand dieser Metriken lassen sich einerseits Klassen ausfindig machen, die eine wichtige Rolle im Entwurf spielen, aber auch Klassen, die überdimensioniert oder zu komplex sind.

- **Weighted Methods per Class (WMC)** misst die Komplexität einer Klasse K mit n Methoden, indem jeder Methode ein Gewicht κ_i zugeordnet und über die Methoden summiert wird:
$$\mathsf{WMC}(K) = \sum_{i=1}^{n} \kappa_i.$$
Dabei soll für die Bestimmung von κ_i ein je nach Anwendung sinnvolles Komplexitätsmaß verwendet werden.

- **Number Of ...** Metriken, die Anzahl und Umfang von bestimmten Elementen des Systems messen, sind immer nützlich. Man kann auf der Systemebene etwa die Anzahl der Klassen in Relation zur Anzahl der abstrakten und nicht-abstrakten Klassen bestimmen, auf der Methodenebene kann man die Länge von Methoden messen um herauszufinden, ob eine Klasse zu große Methoden hat, um noch hinreichend objektorientiert zu sein. Eine zu große oder zu niedrige Zahl an Methoden in einer Klasse kann ein Hinweis darauf sein, dass gewissen Klassen aufgeteilt bzw. mit anderen Klassen zusammengelegt werden sollten.

5 Anwendung: Erkennen von Problemen im objektorientierten Entwurf

In diesem Abschnitt wollen wir eine Vorgehensweise für den praktischen, zielorientierten Einsatz von Metriken betrachten. Wir wählen als instruktives Beispiel das Vorgehen, das Marinescu in seiner Dissertation [Mar02] beschreibt. Sein Vorgehen hat zum Ziel, in einem objektorientierten System durch geschickte Kombination von Metriken zu sogenannten *Erkennungsstrategien* (Abschnitt 5.1) die Schwächen im Entwurf ausfindig zu machen.

Dabei liefert eine Strategie eine Menge von Elementen des Systems, die ein bestimmtes Defizit aufweisen. Das zugrundeliegende Qualitätsmodell soll darüberhinaus noch Ansätze für Verbesserungen liefern.

Unter einem **Entwurfsproblem** verstehen wir im Folgenden eine strukturelle Eigenschaft eines Programmelementes, die abweicht von gewissen Kriterien, die eine Regel für den guten Entwurf charakterisieren. Eine Fülle von konkreten Beispielen findet sich in [Mar02].

5.1 Strategien

Eine der wesentlichen Schwierigkeiten bei der Anwendung von Metriken ist die Interpretation der Messergebnisse im Hinblick auf ein bestimmtes Messziel, in unserem konkreten Fall das Erkennen von Entwurfsproblemen. Da eine einzelne Metrik für sich genommen in der Regel nur eine geringe Aussagekraft hat, stellt sich die Frage, wie Metriken geeignet zu kombinieren sind, um relevante Informationen zu liefern.

Unter einer **Erkennungsstrategie s** zur Aufdeckung eines bestimmten Entwurfsproblems verstehen wir hier eine Liste $\mathsf{m}_1, ..., \mathsf{m}_n$ von Metriken mit jeweils einem zugehörigen *Datenfilter* F_i und eine *Verknüpfungsvorschrift* für die Metriken.

Dies soll im Folgenden genauer erläutert werden. Dazu sei $\mathcal{K}$ die Menge der Programmelemente auf einer bestimmten Abstraktionsebene des zu untersuchenden Systems (wir können uns $\mathcal{K}$ als die Menge der Klassen des Systems vorstellen, aber das Vorgehen ist auch auf jeder anderen Abstraktionsebene denkbar). Der zur Metrik m_i gehörige **Filter** F_i liefert diejenigen Elemente $F_i(\mathcal{K}) \subseteq \mathcal{K}$, die außergewöhnliche Werte hinsichtlich m_i aufweisen. Ein Filter kann beispielsweise die Elemente mit den größten bzw. kleinsten Werten für m_i liefern („Higher Than“, „Bottom Values“ usw.), oder Elemente, deren Werte in einem vorgegebenen Intervall liegen („Between(x,y)“).

Damit die Metriken in ihrer Kombination die Eigenschaften einer Entwurfsregel (bzw. ihrer mangelhaften Anwendung) wiedergeben können, benötigen wir zusätzlich eine **Verknüpfungsregel** für die gefilterten Mengen $F_1(\mathcal{K}), ..., F_n(\mathcal{K})$, nach deren Anwendung wir die Menge $\mathbf{s}(\mathcal{K})$ derjenigen Elemente erhalten, die gegen die Entwurfsregel verstoßen. Als grundlegende Verknüpfungen verwendet man *and* (d.h. „$\cap$“), *or* (d.h. „$\cup$“) und *butnot* (d.h. „$\setminus$“).

Wir wollen diese recht abstrakte Beschreibung durch ein Beispiel verdeutlichen. Dazu betrachten wir eine Erkennungsstrategie, die sogenannte *Gott-Klassen* aufspüren soll. Dabei handelt es sich um Klassen, die einen großen Teil der Funktionalität des Systems in sich vereinigen, und anderen Klassen nur kleinere Details überlassen. Dies wirkt sich nachteilhaft auf Verständlichkeit und Wiederverwendbarkeit aus. Wir machen die folgenden Eigenschaften von Gott-Klassen aus: Sie greifen auf viele „leichtgewichtige“ Klassen zu, sie sind groß und haben eine niedrige Kohäsion. Für die Strategie verwenden wir die Metriken *Access To Foreign Data* (ATFD), *Weighted Method Count* (WMC) und *Tight Class Cohesion* (TCC). Kleine und stark gebundene Klassen sollen dabei ignoriert werden. Das ergibt folgende Verknüpfungsregel:

$$\begin{aligned}\mathbf{s}_{\text{Gott-Klassen}} := &(\langle \mathsf{ATFD}, \text{TopValues}(20\%)\rangle \cap \langle \mathsf{ATFD}, \text{HigherThan}(4)\rangle) \\ &\cap (\langle \mathsf{WMC}, \text{HigherThan}(20)\rangle \cup \langle \mathsf{TCC}, \text{LowerThan}(0.33)\rangle)\end{aligned}$$

Ein umfangreicher Katalog weiterer Entwurfsprobleme und der zugehörigen Erkennungsstrategien findet sich in [Mar02].

Beim Anwenden einer Strategie **s** geht man wie folgt vor:

1. Man wendet die Strategie auf alle Elemente $K \in \mathcal{K}$ an und erhält eine Menge $\mathbf{s}(\mathcal{K})$ von „verdächtigen“ Elementen.

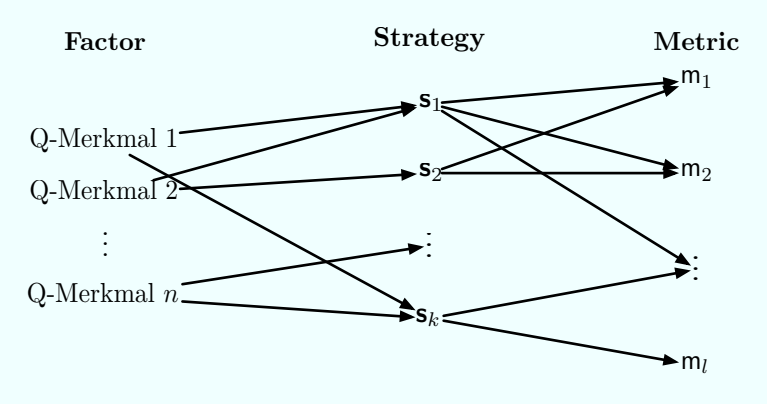
2. Man untersucht die Elemente aus $\mathbf{s}(\mathcal{K})$, ob sie wirklich an dem entsprechenden Entwurfsmangel leiden, oder die Strategie **s** womöglich zu ungenau definiert wurde, um den Mangel aufzuspüren.

Bei der Definition von Strategien sind natürlich auch subjektive Einflüsse enthalten, da für das genaue Festlegen der Parameter von **s** Erfahrung und Augenmaß der Tester bzw. Entwickler vonnöten sind.

5.2 Das Factor-Strategy-Modell

Das in Abschnitt 2.2 vorgestellte FCM-Modell ist in der Lage, Qualitätsmängel zu erkennen und die verantwortlichen Programmelemente zu identifizieren. Es ermöglicht aber in der Regel nicht, *Ursachen* für die Probleme zu finden. Dies kommt daher, dass der Zusammenhang von Qualitäts(teil)merkmalen zu Entwurfsregeln (und somit auch zu konkreten Entwurfsproblemen) nicht explizit im Modell vorhanden ist, d.h. es ist nicht oder nur schwer möglich, Schlüsse darüber zu ziehen, gegen welche Entwurfsregel verstoßen wurde. Als Folge davon kann das FCM-Modell auch keine Hinweise darauf liefern, wie der Entwurf verbessert werden kann.

Marinescu schlägt daher folgende Modifikation des FCM-Modells vor: Die übergeordneten Qualitätsmerkmale werden nicht in Teilmerkmale zerlegt, sondern ihnen werden Erkennungsstrategien zugeordnet, die jeweils zur Aufdeckung eines bestimmten Entwurfsproblems geeignet sind, das dem entsprechenden Qualitätsmerkmal abträglich ist:



Da jede Strategie einem konkreten, klar umrissenen Entwurfsproblem entspricht, kann man die mindere Qualität unmittelbar an diesem festmachen und weiß somit auch, wo man bei der Verbesserung des Entwurfs ansetzen muss. Man beachte, dass das FS-Modell, obwohl es im objektorientierten Kontext definiert wurde, sich im Prinzip auf jedes Entwicklungsmodell übertragen ließe.

Wir skizzieren nun, wie die Auswertung der Messergebnisse im FS-Modell erfolgt.

1. *Bewertung der Qualitätsmerkmale:* Einem bestimmten Qualitätsmerkmal $\mathfrak{Q}$ seien im FS-Modell die Strategien $\mathbf{s}_1, ..., \mathbf{s}_m$ zugeordnet. Zuerst wird eine Teilbewertung für jede einzelne Strategie $\mathbf{s}_j$ ermittelt. Man könnte beispielsweise die Anzahl $|\mathbf{s}_j(\mathcal{K})|$ der mängelbehafteten Programmelemente im Verhältnis zu Gesamtzahl $|\mathcal{K}|$ aller Programmelemente als Bewertung zu wählen. Denkbar ist auch eine unterschiedliche Gewichtung der einzelnen Metriken von $\mathbf{s}_j$. Sind alle Strategien bewertet, so kann man aus den einzelnen Bewertungen eine Gesamtbewertung für $\mathfrak{Q}$ bestimmen. Dabei kann man die Teilbewertungen nach der Wichtigkeit der jeweiligen Strategie (also der Bedeutung des zugehörigen Entwurfsproblems) gewichten. Hieraus erhält man durch eine vorgegebene Tabelle eine qualitative Bewertung von $\mathfrak{Q}$.

2. *Ursachen für Entwurfsprobleme identifizieren:* Wie bereits erwähnt entspricht jede Erkennungsstrategie $\mathbf{s}_j$ einem Entwurfsproblem und ihre Anwendung liefert eine Menge $\mathbf{s}_j(\mathcal{K})$ von Elementen, bei denen dieses Problem auftritt. Ist also $\mathbf{s}_j(\mathcal{K}) \neq \emptyset$ für ein j, so tritt dieses Problem im System $\mathcal{K}$ auf, und die Elemente von $\mathbf{s}_j(\mathcal{K})$ sind gerade die Elemente, die man verändern muss, um das Problem zu beseitigen.

6 Ausblick und Zusammenfassung

Metriken stellen ein nützliches Hilfsmittel der Qualitätssicherung dar. Dabei ist es wichtig, die von den Metriken gelieferten Daten sinnvoll zu interpretieren. Wie man sich anhand des FS-Modells klarmachen kann, sind Metriken dann besonders mächtig, wenn sie in geeigneter Weise kombiniert werden, da sie dann sehr konkrete Hinweise auf Schwächen im Software-Entwurf liefern können.

Es gibt eine Vielzahl von Metriken, insbesondere für objektorientierte Systeme. Wichtig ist die Konstruktion und Implementierung von Systemen, die diese Metriken kombinieren können, um zu leistungsfähigen Hilfsmitteln beim Aufdecken von Entwurfsproblemen zu werden. In Kapitel 5 haben wir ein solches System kurz kennengelernt. Von Interesse sind dabei Möglichkeiten, das System noch flexibler zu gestalten, und die Integration in kommerzielle Entwicklungsumgebungen.

In dieser Arbeit haben wir unsere Betrachtungen auf Metriken beschränkt, die die statische Struktur eines Software-Produktes im Hinblick auf die innere Qualität untersuchen. Ebenso gibt es aber auch Metriken, die verwendet werden, um die Prozesse bei der Software-Entwicklung zu beurteilen. Solche Metriken sind vorrangig von betriebswirtschaftlichem Interesse, man findet einiges dazu in [Bal98] und [Pre99].

Literatur

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik (Bd. 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, 1998.

[BBM95] Victor Basili, Lionel Briand, and Walcelio Melo. *A Validation of Object-Oriented Desing Metrics as Quality Indicators.* Technical Report, University of Maryland, Department of Computer Science, 1995.

[Fen90] Norman Fenton. *Software metrics: theory, tools and validation.* Software Engineering Journal, 1990.

[Fen94] Norman Fenton. *Software Measurement: A Neccessary Scientific Basis.* IEEE Transactions of Software Engineering vol 20 no 3, 1994.

[Jon94] Capers Jones. *Software metrics: Good, bad and missing.* 1994.

[KC85] Dennis Kafura and James Canning. *A Validation of Software Metrics Using Many Metrics and Two Resources.* 1985.

[KS97] Barbara Kitchenham and J.G. Stell. *The danger of using axioms in software metrics.* IEE Proceedings no 19771723, 1997.

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design.* http://www.cs.utt.ro/∼radum/papers.html, 2002.

[Pre99] Lutz Prechelt. *Ausgewählte Kapitel der Softwaretechnik.* http://page.mi.fu-berlin.de/∼prechelt/swt2/skript.html, 1999.

[Wey88] Elaine Weyuker. *Evaluating software complexity measures.* IEEE Transactions on Software Engineering vol 14 no 9, 1988.

Index