HANSER

Leseprobe

zu

Grundlagen des modularen Softwareentwurfs

von Herbert Dowalil

ISBN (Buch): 978-3-446-45509-2
ISBN (E-Book): 978-3-446-45600-6

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/
sowie im Buchhandel

9 Metriken

„Grossmann, du musst mir helfen, sonst werde ich verrückt!“
Albert Einsteins berühmter Hilferuf an einen befreundeten Mathematiker
im Zuge der Entwicklung der Relativitätstheorie

Metriken, also Kennzahlen, gibt es im Softwareengineering für diverse Kategorien. Man kann damit Dinge wie die Wartungsproduktivität oder auch die Testeffizienz messen [Sne10]. In diesem Buch möchte ich ganz konkret auf Metriken eingehen, welche dazu dienen können, die strukturelle Qualität und den Grad der Modularisierung eines Softwaresystems in Zahlen auszudrücken.

Zunächst sei noch erwähnt, wie diese Kennzahlen am besten einzusetzen sind. Ich möchte nämlich davon abraten, Teams und Führungskräfte einfach durch Setzen von Grenzwerten dieser Metriken in ihrem Wirken einzuschränken, in der wohlmeinenden Hoffnung, sie dadurch positiv zu beeinflussen. Dies ist eher fragwürdig, weil es meist nur dazu führen wird, dass die Mitarbeiter durch wenig sinnvolle Tricksereien diese eng gesetzten Ziele erreichen werden. Eine vernünftige Modulstruktur, welche auch frei von Abhängigkeitszyklen ist, mag ein hehres Ziel sein, aber ein Big Ball Of Mud, bei dem man mit aller Gewalt Strukturzyklen aufzulösen versucht, ist dies sicherlich nicht. Statt strikter Obergrenzen für Metriken empfehle ich eher, die Kennzahlen möglichst häufig und öffentlich zu kommunizieren, um Probleme transparent zu machen und in eine subjektive Führungsarbeit einfließen zu lassen. Als „smarte“ Ziele taugen sie meiner Erfahrung nach eher nicht. Eine gute Idee kann es auch sein, hierbei Ansätze der Gameification ins „Spiel“ zu bringen [WKO16].

9.1 Unit-Test-Abdeckung und das Legacy-Code-Dilemma

Anwendung der Kennzahl:

Niedrige Abdeckung durch automatisierte Unit-Tests kann (!) ein Indiz für schlecht strukturierten Code sein.

Bereits mehrmals wurde ich um Unterstützung gebeten, weil Teams mit Qualitätsproblemen zu kämpfen hatten, wo man der Meinung war, dass dies an der geringen Absicherung durch automatisierte Unit-Tests läge. Aus irgendwelchen Gründen tat sich das Team allerdings schwer, solche Testfälle zu implementieren. Dabei lag die eigentliche Ursache meist darin, dass der Code so schlecht strukturiert war, dass es diese Units, also Komponenten, gar nicht erst gab. In Kapitel 1 wurde bereits definiert, dass einer der Gründe für eine Modularisierung des Codes die hierdurch gewährleistete gute Testbarkeit der einzelnen Komponenten ist. Wenn man diese nun nicht hat, so ist dies besonders problematisch, weil man einerseits für ein Refactoring die Absicherung durch automatisierte Tests benötigt, aber andererseits nicht dazu in der Lage ist, weil es an diesen Unit-Tests fehlt, ein typisches Beispiel für ein sogenanntes Legacy-Code-Dilemma [Fea10].

Der Ausweg aus diesem Dilemma ist denkbar einfach. Unit-Testing ist bekanntlich nicht die einzige Art und Weise, um für eine Automatisierung von Testfällen zu sorgen. Genauso gut ist es möglich, das gesamte System an seinen Schnittstellen bzw. dem User-Interface zu testen. Wenn man also damit beginnt, diese Art von Tests zu automatisieren, ist man in weiterer Folge dazu in der Lage, Schritt für Schritt eine Modularisierung des Systems herbeizuführen. Unit-Tests können Sie dann nach erfolgter Strukturierung bei Bedarf immer noch erstellen.

Test Driven Development (TDD)

Um ein solches Dilemma präventiv zu verhindern, setzen einige Unternehmen inzwischen auf Test Driven Development. Ich muss gestehen, dass ich kein allzu großer Fan davon bin. Für mich rückt es nämlich ein einziges der Ziele der Architekturarbeit zu sehr in den Vordergrund und bringt die Gefahr mit sich, dass es alle anderen ersetzt. Im Endeffekt ist es natürlich wichtig, eine Abdeckung durch automatisierte Testfälle zu haben, um eine reibungslose Wartungstätigkeit zu gewährleisten. Welcher Weg dorthin genommen wurde, ist aber nicht so wichtig, und Architekturarbeit sollte man auch unabhängig von der Erstellung der Tests machen. Außerdem kann man heute fast genauso gut das gesamte System bzw. Service am Stück testen und muss dann nicht unbedingt jede einzelne Komponente auch noch explizit testen. Konkrete Designfehler, welche im Zuge des TDD manchmal begangen werden, sind:

- Unnötige Abstraktionen wie zusätzliche Interfaces, die nur dazu dienen, die Implementierung für die Durchführung eines Unit-Tests durch einen Stub austauschen zu können.
- Schnittstellen, welche eigentlich verborgen werden können, werden öffentlich gemacht, um sie im Testfall auch ansprechen zu können. Beispiel für eine solche Verletzung des Information Hidings wäre es, wenn in Java eine Methode public anstatt private definiert wird, um einzeln testbar zu sein.

Legacy-System-Dilemma

Während es beim Legacy-Code-Dilemma und seinem hier vorgestellten Ausweg in erster Linie um Mikro-Architekturen geht, so sind ähnliche Situationen auch auf Makro-Architekturebene vorstellbar. Meist dann, wenn es gleich zwischen mehreren Systemen oder Services zur Integration über dieselben Shared Kernels kommt. Also beispielsweise, wenn gleich mehrere Datenbanken, und im Extremfall auch noch der Code zum Zugriff darauf, geteilt werden. Hier tut man sich dann nicht selten schon schwer, manuell die Systeme zu testen, und das einfach wegen der Abhängigkeiten, die diese untereinander haben, von Testautomatisierung ganz zu schweigen. Hier ist ein Ausweg wesentlich schwieriger zu finden. Man könnte versuchen, die Releases und die notwendigen Tests dafür zu koordinieren, bis für Abhilfe gesorgt wurde. Dies ist aber aufwendig, bremst die Time-To-Market und kann daher nur temporär als ein Workaround um das eigentliche Problem gesehen werden. Eine Zeitlang könnte man für jedes System auch komplexe Stubs entwickeln, die dann jedes andere System für die Durchführung der eigenen Tests benutzen kann, bis bessere Schnittstellen gebaut wurden. Beneidenswert sind Unternehmen, die darunter leiden, jedenfalls nicht. Üblicherweise ist dies eine Folge degenerierter, weil meist ignorierter, Makro-Architekturen.

9.2 Technische Schuld

Anwendung der Kennzahl:

- *Plakative Art und Weise, um Probleme der Software auf den Punkt zu bringen.*
- *Geringe Nachvollziehbarkeit und dadurch nicht so einfach durch ansonsten unnütze Interventionen im Code zu manipulieren.*

Empfehlenswert ist es, Probleme einer Softwarelösung in einer möglichst plakativen Zahl auf den Punkt zu bringen. Manager und Aufsichtsräte werden mit einer Aussage wie „Die Anzahl der Strukturzyklen wurde im Laufe des Jahres von 28 auf 22 reduziert“ nicht viel anfangen können. Stattdessen kann man aber jedes gemessene Problem in der Software (wie Code Smells, Komponente mit zu viel Komplexität, unerwünschte Abhängigkeiten oder eben Strukturzyklen) in die Zeit umrechnen, die man zu seiner Behebung benötigen würde. Dazu kommen dann auch Aufwände für konkrete Issues, welche von den Mitarbeitern entdeckt wurden und die einer automatisierten statischen Code-Analyse verborgen bleiben. Da diese Stunden natürlich Arbeitszeit entsprechen, kann man danach noch, je nachdem, was ein Mitarbeiter im Unternehmen durchschnittlich verdient, die kumulierte Zeitdauer in einen Geldbetrag umrechnen. Dem Management wird danach kommuniziert, dass die gesamte Technische Schuld der Softwarelösung z. B. 150 000 EUR entspricht. Dabei handelt es sich natürlich um Softwareprobleme, bei denen eine Behebung jeweils billiger ist, als auf Dauer damit zu leben. Jede Wette, dass dies der Unternehmensführung nicht egal sein wird.

Der andere große Vorteil einer Berechnung der Technischen Schuld ist, dass sie durch ebenso gezielte wie sinnlose Maßnahmen im Code nicht direkt beeinflussbar ist. Sobald die Bewertung von Mitarbeitern von diesen Kennzahlen abhängt, wird es früher oder später zu solchen Manipulationen kommen. Durch den Abstraktionslevel, welchen eine Technische

Schuld zwangsläufig hat, ist es auch denkbar, einen solchen Wert für eine Managed Evolution (Kapitel 1.5.1) zu verwenden. Von den Führungskräften und Projektleitern wird dann verlangt, dass diese Technische Schuld keinesfalls weiter steigen darf. Dadurch sollte es automatisch zu einem Verbesserungsprozess kommen.

9.3 Komplexität und Modulgröße

Anwendung der Kennzahlen:

Steuerung der Obergrenzen, ab denen eine weitere Strukturierung jeweils sinnvoll wird.

Strukturen machen natürlich erst ab einem gewissen Umfang Sinn. In Mikro-Architekturen sollten die Obergrenzen für einzelne Module dort gesetzt werden, wo es schwierig wird, diese am Stück zu verstehen. In Makro-Architekturen wiederum, wenn ein Team alleine nicht mehr in der Lage ist, das Modul laufend zu betreuen und die anfallenden Arbeiten daran zu erledigen. Letzteres möchte ich zunächst noch etwas näher erläutern, weil es unweigerlich die Frage aufwirft, bis zu welcher Größe man noch von einem Team sprechen kann. Bei Amazon ist dies so definiert, dass es sich so lange um ein Team handelt, solange die informelle Kommunikation ausreicht, um noch von selbst für den nötigen Informationsfluss zu sorgen [Cho14]. Die Obergrenze wird dabei von den existierenden Verbindungen zwischen den Mitgliedern der Gruppe festgelegt. Es kommt bei jedem weiteren Teammitglied zu jeweils einer neuen Verbindung (und somit einem Kommunikationsweg) zu einem der bereits bestehenden Teammitglieder. Sie können bei einer gegebenen Anzahl von n Teammitgliedern mit der folgenden Formel die Anzahl möglicher Verbindungen berechnen:

Verbindungen = (n*(n-1))/2

Das bedeutet, dass ein Team, welches aus zwölf Personen besteht, bereits über 66 mögliche Verbindungen und somit notwendige Kommunikationskanäle verfügt. Man nähert sich bei einer solchen Teamgröße also bereits einem gewissen gesunden Maximum.

In der Mikro-Architektur geht es bei der Festlegung einer Obergrenze wiederum eher um die Leistungsfähigkeit eines menschlichen Gehirns. Code von wenigen Zeilen Länge (Lines of Code oder kurz LOC) muss natürlich nicht weiter strukturiert werden und findet beispielsweise gut in einer einzelnen Methode Platz. Während Codeblöcke von über 1000 LOC nur mehr schwierig zu verstehen sind und bereits erste Programmstrukturen erfordern. Modularisierung ist also immer erst ab einer gewissen Grenze notwendig. Wie legt man allerdings eine solche Grenze fest? Die hier verwendeten LOC taugen dafür nur bedingt, einfach weil es manche Code-Konstrukte gibt, die es schaffen, in einer Zeile die Komplexität unterzubringen, für die ein anderer Developer eine ganze Klasse geschrieben hätte. Wie es möglich ist, eine solche Komplexität in einer Zahl abzubilden, werde ich anhand eines kleinen Java-Code-Snippets (Listing 9.1) darstellen.

Listing 9.1 Beispielcode, anhand dessen die verschiedenen Möglichkeiten zur Berechnung der Komplexität dargestellt werden

```
if (toggle == true) {
  if (input < 10 || input > 99) {
    log("1 or more than 2 digits");
  } else {
    log("2 digits");
  }
} else {
  log("Lorem ipsum dolor sit amet");
}
```

9.3.1 Semantische Komplexität

Die Berechnung einer semantischen Komplexität geht davon aus, dass es umso schwieriger ist, ein Stück Code zu verstehen, je mehr verschiedene Schlüsselwörter und Anweisungen verwendet werden. Meist geht es dabei um die Relation von unterschiedlichen Schlüsselworten zur Gesamtanzahl der Schlüsselwörter. In unserem Beispiel (Listing 9.1) gibt es neun verschiedene Schüsselwörter, nämlich „if“, „toggle“, „==“, „input“, „<“, „||“, „>“, „log“ und „else“. Diese kommen in Summe 14-mal zur Anwendung. Eine semantische Komplexität könnte dann wie folgt berechnet werden: 9 / 14 = 0,64.

Etwas Kurioses dazu noch am Rande, was gleichzeitig als Warnung davor dienen soll, diese Metriken allzu ernst zu nehmen. Nachdem etwas angestaubte Sprachen wie COBOL aus wesentlich weniger möglichen Schlüsselwörtern bestehen als modernere Pendants, könnte man naiverweise dazu tendieren, wieder vermehrt auf diese zu setzen, um die Komplexität zu verringern. Nun, es ist sicherlich ein Körnchen Wahrheit darin zu finden und COBOL-Code vermutlich leichter zu lesen als Java-Code. Eine solche Strategie wäre aber selbstverständlich trotzdem Unsinn.

Beispiel zur Berechnung:

Halstead-Metrik [Hal77]

9.3.2 Strukturelle Komplexität

Mögliche Abläufe im Code kann man auch als einen gerichteten Graphen betrachten. An diversen Weichen wie if- oder switch-Anweisungen tun sich dabei jeweils neue Zweige auf. In unserem Beispiel (Listing 9.1) finden sich zwei if-Anweisungen in den ersten beiden Zeilen. Jede kann als Knoten gesehen werden, welche jeweils zu zwei unterschiedlichen Kanten führen. Je mehr dieser Knoten und Kanten ein Code hat, desto größer seine Komplexität.

Beispiel zur Berechnung:

Die Zyklomatische Komplexität nach McCabe [McC76] ist bestimmt die empfehlenswerteste und die am weitesten verbreitete Möglichkeit zur Messung von Komplexität.

9.3.3 Verschachtelungskomplexität

Eine andere Möglichkeit, die Komplexität von Code zu messen, ist es, die Tiefe der Verschachtelung der Anweisungen zu messen. In unserem Beispiel (Listing 9.1) wird durch jedes if eine jeweils weitere Verschachtelungstiefe geöffnet, durch das erste if in Zeile 1 also eine zweite Ebene und durch das if in Zeile 2 gleich im Anschluss eine dritte. Wenn man es in Relation setzen möchte, kann man dies dann noch durch die Anzahl der Anweisungen dividieren, so wie das von R. Prather vorgeschlagen wurde.

Beispiel zur Berechnung:

Prather-Metrik [Pra844]

9.4 Kohäsion

9.4.1 Relational Cohesion

Anwendung der Kennzahl:

Identifikation von Komponenten, deren Subbausteine von mangelhafter Zusammengehörigkeit sind.

Ein Anzeichen für gute Kohäsion ist ein durchaus hohes Niveau an Abhängigkeiten zwischen den Subbausteinen einer Komponente. Daher gibt die Kennzahl Relational Cohesion das Verhältnis dieser Subbausteine zu den Verbindungen zwischen diesen wieder [JAR17]. Laut der Dokumentation von JArchitect sollten gesunde Werte >= 1,5 liegen. Das würde bedeuten, dass die Anzahl interner Verbindungen bzw. Abhängigkeiten immer etwas höher sein sollte als die Anzahl interner Bausteine selbst.

Relational Cohesion = (Anzahl Verbindungen + 1) / Anzahl Subbausteine

9.4.2 Lack of Cohesion in Methods IV (LCOM4)

Anwendung der Kennzahl:

Identifikation von Klassen, in denen es mehr als eine Submenge von Variablen und Methoden gibt, welche nichts miteinander zu tun haben.

Einige Anläufe waren nötig, um eine Kennzahl zu definieren, welche die mangelhafte Zusammengehörigkeit von Membern einer Klasse aufzeigen kann. Auch wenn inzwischen bereits ein LCOM5 definiert wurde, so empfehle ich nach wie vor die Berechnung des LCOM4. Es geht darum, Situationen aufzuzeigen, wie sie in Listing 9.2 dargestellt werden. Der dafür berechnete LCOM4-Wert wäre 2, weil es hier keine Verbindung zwischen den beiden komplett voneinander getrennten Teilmengen 1 (var1, setVar1, methodA und methodB) und 2 (var2, setVar2, methodC) gibt. Man sollte also in der Regel einen LCOM4 von 1 anstreben, was allerdings nicht immer praktikabel ist. In den folgenden Fällen ist die Berechnung des

LCOM4 wenig sinnvoll, wobei dies wiederum von den einzelnen Tools berücksichtigt wird, wenn auch natürlich nicht auf einheitliche Weise:

- Getter und Setter auf unterschiedliche Properties in Java stehen meist in keiner Beziehung zueinander. Die häufig ohnehin sinnvolle Erstellung von hashcode- und equals-Methoden erweist sich als eine einfache Möglichkeit, hier wieder für Kohäsion zu sorgen.
- In manchen Klassen besteht die einzige Gemeinsamkeit der Methoden darin, dass sie sich einfache Dinge, wie eine Instanz einer Klasse zum Schreiben von Logstatements, teilen. Dies mag dann für eine statische Codeanalyse nach Kohäsion aussehen, hat aber genau genommen nichts damit zu tun.
- Es gibt immer wieder Klassen, welche von einer Parent-Klasse einwandfreier Kohäsion erben, welche einzelne Methoden dieser überschreiben, die in der erbenden Klasse wiederum nichts miteinander zu tun haben. Für diese wird dann ein hoher LCOM4-Wert errechnet, auch wenn dies so nicht ganz gerechtfertigt ist.
- Typische Utility-Klassen bestehen oft aus vielen Support-Methoden. Hier ist es ganz normal, wenn es zu einem hohen Wert des LCOM4 kommt.

Am Beispiel des LCOM4 sehen Sie, dass es tatsächlich wenig sinnvoll ist, strikte Grenzwerte für Kennzahlen festzulegen.

Listing 9.2 Beispiel für eine Java-Klasse schlechter Kohäsion, weil sie aus zwei disjunkten Teilmengen besteht

```
public class Demo {

   private int var1;
   private int var2;

   public void setVar1(int var1) {
      this.var1 = var1;
}

   public void setVar2(int var2) {
      this.var2 = var2;
   }

   public int methodA() {
      return var1*2;
      }

   public int methodB() {
      return methodA()*3;
   }

   public int methodC() {
      return var2*4;
   }
}
```

9.5 Component Rank

Anwendung:

- *Einzelne Komponenten mit sehr hohen Werten könnten Indikatoren für schlechte Modularität sein.*
- *Bedarf an Unit-Tests bei Komponenten mit hohem Wert besonders groß.*

Wenn Sie die Komponenten identifizieren möchten, die besonders häufig benutzt werden, so bietet sich dafür die Berechnung des Component Rank an. Angelehnt ist die Berechnung dabei an den Page-Rank-Algorithmus von Google, welcher Webseiten, auf die andere Webseiten besonders häufig verweisen, in der Anzeige der Suchergebnisse bevorzugt. In Tabelle 9.1 sehen Sie, für welche Klassen des JDK hier besonders hohe Werte errechnet werden [Ino17]. Es dürfte für recht wenig Verwunderung sorgen, dass die Klasse java.lang.Object hier an der Spitze steht. Der Wert von 0,16126 bedeutet, dass diese Klasse etwas über 16 % aller eingehenden Abhängigkeiten des gesamten JDK auf sich vereint.

Tabelle 9.1 Tabelle der Component-Rank-Werte der Klassen des JDK 1.3

Klasse	Component Rank
`java.lang.Object`	0,16126
`java.lang.Class`	0,08712
`java.lang.Throwable`	0,05510
`java.lang.Exception`	0,03103
`java.io.IOException`	0,01343
`java.lang.StringBuffer`	0,01214
`java.lang.SecurityManager`	0,01169
`java.io.InputStream`	0,01027
`java.lang.reflect.Field`	0,00948
`java.lang.reflect.Constructor`	0,00936

9.6 Software-Package-Metriken nach Robert C. Martin

Die sogenannten Software-Package-Metriken wurden von Robert C. Martin im Jahr 2002 definiert [Mar02]. Zur näheren Erläuterung möchte ich das Beispiel verwenden, welches in Bild 9.1 dargestellt ist. In der Abbildung sind auch gleich die Kennzahlen der Software-Package-Metriken berechnet, welche hier in weiterer Folge noch vorgestellt werden. Das Architekturbeispiel selbst wird uns auch danach noch weiter durch Kapitel 9 begleiten.

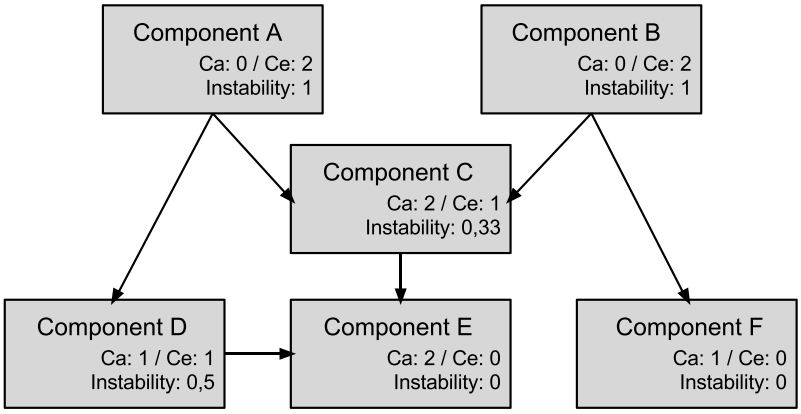

Bild 9.1 Unser Beispiel für Kapitel 9

9.6.1 Afferent Coupling (Ca)

Auch bekannt als:

- *Number of Incoming Dependencies*
- *Fan-in*

Anwendung:

- *Identifikation von Komponenten, für die automatisierte Komponententests besonders wichtig sind (hoher Ca-Wert).*
- *Identifikation von Komponenten, bei denen man idealerweise mit einem Refactoring beginnen würde (niedriger Ca-Wert).*

Diese Kennzahl gibt an, wie vielen anderen Komponenten diese Komponente bekannt ist. Es geht dabei also darum, auf wie viele andere Komponenten es potenziell Auswirkungen gibt, wenn diese Komponente geändert wird.

9.6.2 Efferent Coupling (Ce)

Auch bekannt als:

- *Number of Outgoing Dependencies*
- *Fan-out*

Diese Kennzahl gibt die Anzahl der Komponenten an, die dieser Komponente bekannt sind bzw. von der diese abhängig ist. Es beantwortet also die Frage: „In wie vielen anderen Komponenten oder Bausteinen können Änderungen potenziell Auswirkungen auf diese Komponente haben?“

9.6.3 Instability

Anwendung:

- *Identifikation potenzieller Schwachstellen im System (Instability nahe bei 1).*
- *Gezielter Entwurf von Komponenten , welche einfach zu ändern sind (Instability nahe bei 0).*

Die Instability gibt das Verhältnis der ausgehenden Abhängigkeiten (Ce) zu allen Abhängigkeiten (also ein- und ausgehend Ce+Ca) an. Sie wird wie folgt berechnet:

Instability = Ce / (Ce + Ca).

Ein Wert nahe 1 bedeutet demnach, dass eine Komponente besonders anfällig für Änderungen anderer Komponenten ist, während sie selbst aber wiederum relativ einfach geändert werden kann, ohne dass man sich um potenziell negative Auswirkungen auf andere Komponenten sorgen muss. Gegenteiliges gilt jeweils für einen Wert nahe bei 0. Beim Entwurf einer Systemstruktur kann man dies gezielt berücksichtigen und Komponenten entsprechend flexibel oder bewusst stabil gestalten.

9.7 Metriken nach John Lakos

Um die strukturelle Qualität und somit die Güte der Architektur eines gesamten Systems zu messen, bieten sich die Kennzahlen von John Lakos [Lak96] an. Diese bauen teilweise aufeinander auf, weshalb sie hier der Reihe nach vorgestellt werden.

9.7.1 Depends Upon und Used From

In Bild 9.2. sehen Sie das Beispiel aus 9.6. Für jede der sechs Komponenten wurden beispielhaft die Kennzahlen Depends Upon und Used From berechnet. Bei Depends Upon handelt es sich um alle Komponenten, von denen diese Komponente abhängig ist, inklusive sich selbst. Used From wiederum ermittelt die Anzahl der Komponenten, welche von dieser abhängig sind, wieder inklusive der Komponente selbst. Dabei werden Abhängigkeiten kumuliert, was bedeutet, dass auch indirekte Abhängigkeiten dazugezählt werden. Depends Upon betrifft also alle Komponenten, die von einer Komponente benutzt werden, sowie alle Komponenten, welche diese wiederum benutzen, und so weiter. Diese Kumulierung ist auch der große Unterschied zu Ca und Ce der Software-Package-Metriken (Abschnitt 9.6).

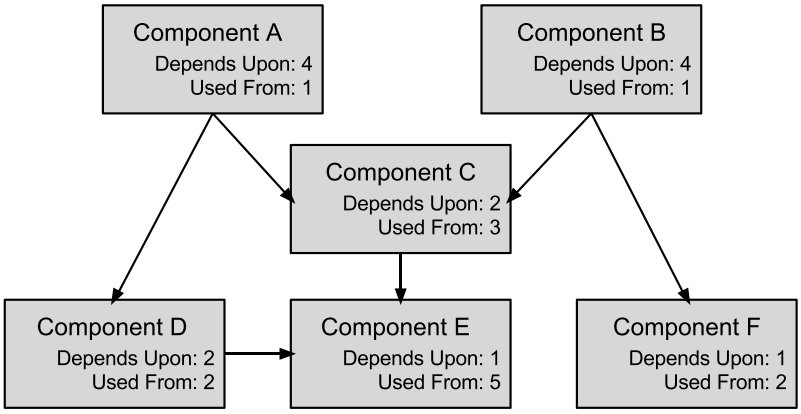

Bild 9.2 Sechs Komponenten, für die jeweils Depends Upon und Used From berechnet wurde.

9.7.2 Cumulative Component Dependency (CCD)

Bei der CCD handelt es sich um die Anzahl der kumulierten Abhängigkeiten der Bausteine eines Systems. Sie wird berechnet, indem man einfach sämtliche Depends-Upon-Werte summiert. Dasselbe Ergebnis erhält man übrigens immer, wenn man die Used-From-Werte zusammenzählt. In unserem Beispiel (Bild 9.2 6 Komponenten, wo jeweils Depends Upon und Used From berechnet wurden.) würde das, basierend auf den Depends-Upon-Werten, wie folgt aussehen:

CCD = 4 (A) + 4 (B) + 2 (C) + 2 (D) + 1 (E) + 1 (F) = 14

Das bedeutet demnach, dass es in Summe 14 direkte und indirekte Abhängigkeiten in diesem Beispiel gibt, egal in welche Richtung (ein- oder ausgehend). Dabei gilt natürlich: je geringer, desto besser, weil einfacher und mit geringerer Gefahr von Seiteneffekten zu ändern.

9.7.3 Average Component Dependency (ACD)

Um eine Zahl wie die CCD in Relation zu setzen, berechnen wir als Nächstes die Average Component Dependency oder ACD. Dafür dividieren wir die CCD durch die Anzahl der Komponenten:

ACD = 14 / 6 = 2,33

Die ACD gibt an, von wie vielen anderen Komponenten eine Komponente im Durchschnitt abhängig ist. Im konkreten Fall bedeutet eine ACD von 2,33, dass eine Komponente im Schnitt von 2,33 Komponenten abhängig ist, inklusive sich selbst.

9.7.4 Relative Average Component Dependency (RACD)

Anwendung:

Abbildung des Grads der allgemeinen Kopplungen der Bausteine eines Systems.

Die ACD lässt sich noch nicht dazu benutzen, verschiedene Architekturen miteinander zu vergleichen. Die ACD wird bei einem System, welches aus vielen Komponenten besteht, größer sein als bei kleineren Systemen. Daher setzen wir diese Zahl noch in Relation zur Gesamtanzahl der Komponenten und berechnen somit die Relative ACD:

RACD = 2,33 / 6 * 100 = 39 %

Das bedeutet nun, dass eine Komponente dieser Architektur im Schnitt zu 39 % aller Komponenten kumulierte Abhängigkeiten hat. Auf eine Komponente können sich also Änderungen an durchschnittlich 39 % der anderen Komponenten direkt oder indirekt als Seiteneffekt negativ auswirken. Je geringer dieser Wert, desto geringer der Grad an Kopplung zwischen den Komponenten des Systems. Für Systeme mit größerer Komponentenanzahl ergeben sich dabei aber üblicherweise etwas kleinere Werte. Empfehlenswert ist es, eine RACD kleiner 25 % anzustreben.

9.7.5 Normalized Cumulative Component Dependency (NCCD)

Anwendung:

Wie RACD, allerdings ist die NCCD etwas unabhängiger von der konkreten Anzahl der Komponenten, was diesen Wert vergleichbarer macht.

Als Alternative zur RACD sei noch die NCCD erwähnt, welche die Relation der CCD einer Architektur zur CCD eines ausbalancierten Binärbaums mit derselben Anzahl an Komponenten angibt. Da wir in unserem Beispiel in Summe sechs Komponenten haben, sähe ein solcher als Pendant dazu aus, wie Bild 9.3 zeigt. Die CCD dieser Struktur berechnet sich so:

CCD = 6 + 3 + 2 + 1 + 1 + 1 = 14

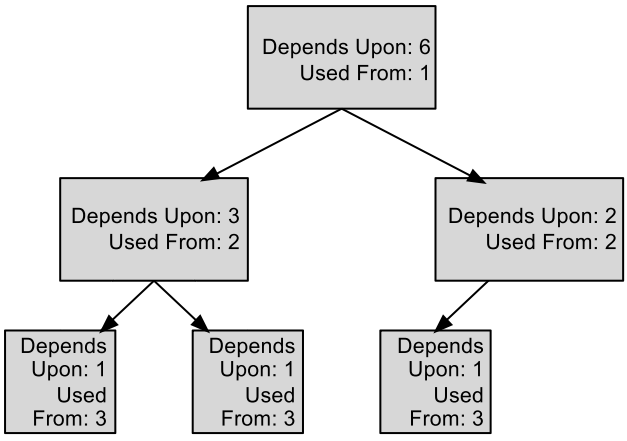

Bild 9.3 Dieselbe Anzahl an Komponenten in einem Binärbaum

Da wir vorher ebenfalls eine CCD von 14 hatten, ergibt sich für diesen Fall eine NCCD von exakt 1.0, was in der Praxis wohl selten vorkommen wird. Empfehlenswert ist eine Zielsetzung von einer NCCD kleiner als 6.

9.8 Relative Cyclicity

Anwendung:

Vermeidung von Strukturzyklen , dadurch Eindämmung der allgemeinen Kopplung.

Die Kennzahl Relative Cyclicity gibt an, wie groß der Prozentsatz an Komponenten ist, welche an einem Zyklus beteiligt sind. Wenn wir bei unserem Beispiel bleiben und nur eine zyklische Abhängigkeit von Komponente E zu B hinzufügen (Bild 9.4), so sind im Endeffekt über die Abhängigkeit von B zu C, welche ja wiederum von E abhängt, drei der sechs Komponenten an einem Zyklus beteiligt. Die Relative Cyclicity berechnet sich dann wenig überraschend wie folgt:

Relative Cyclicity = Anzahl zyklischer Komponenten / Anzahl Komponenten * 100

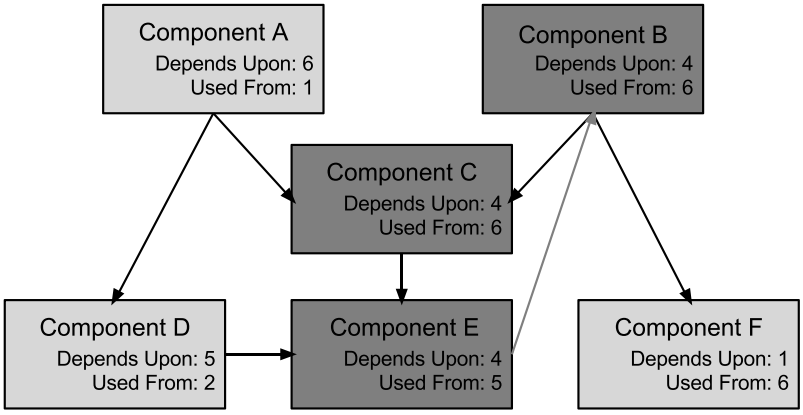

Bild 9.4 Dieselbe Architektur, allerdings mit einem Strukturzyklus

In unserem Fall ergibt sich demnach also eine Relative Cyclicity von 50 %. Genau die Hälfte aller Komponenten ist hier an einem Zyklus beteiligt. Um zu zeigen, warum man dies so gut es geht vermeiden sollte, möchte ich für unser Beispiel mit der zyklischen Abhängigkeit noch einmal die Kennzahlen nach John Lakos berechnen:

CCD = 6 + 4 + 4 + 5 + 4 + 1 = 26

ACD = 24 / 6 = 4

Relative ACD = 4 / 6 * 100 = 67 %

NCCD = 26 / 14 = 1,86

Sie sehen, dass es durch das Hinzufügen nur einer einzelnen zyklischen Abhängigkeit fast zu einer Verdopplung der allgemeinen Kopplung des Systems gekommen ist. Versuchen Sie

daher immer so gut es geht, Strukturzyklen zu vermeiden. Meiner Erfahrung nach ist es übrigens so, dass eine sinnvolle Architekturdefinition oft sowieso frei von Strukturzyklen ist. So kann eine Versicherungspolice ohne einen Schadenfall existieren und braucht bestimmt keinerlei Abhängigkeiten zu diesem. Police und Schadenfall ist es wiederum egal, ob und wer Provision für die Betreuung bekommt. Das Inkasso wird es wiederum nicht kümmern, wer Ein- und Auszahlungen vornimmt, während es selber aber eingehende Abhängigkeiten von Police, Schaden und Provision haben wird.

Ein angenehmer Nebeneffekt einer solchen Vermeidung zyklischer Abhängigkeiten ist außerdem, dass ein Team sich sehr konkrete Gedanken wird machen müssen, um ein solches Ziel zu erreichen. Und zwar darüber, welche Strukturen es baut, wo genau es die Bausteingrenzen zieht und welche Abhängigkeiten dazwischen dann erlaubt sein werden. Eine Architektur, welche zufällig nebenbei entsteht, wird es mit einem solchen Ziel also nicht geben können.