



## Alles im Blick

# STAN – Strukturanalyse für Java

Christoph Beck, Oliver Stuhr

Neben der üblicherweise durch Tests gewährleisteten äußeren Qualität einer Software ist auch die innere Qualität entscheidend für den Projekterfolg. Ein geeignetes Mittel zur Sicherstellung der inneren Qualität bietet die Strukturanalyse, basierend auf der Visualisierung des Designs durch Abhängigkeitsgraphen und der Validierung gegen bewährte Design-Prinzipien durch Qualitätsmetriken. Voraussetzung ist dabei eine nahtlose Integration in den Entwicklungsprozess. Dies leistet das in Eclipse integrierte Tool STAN.

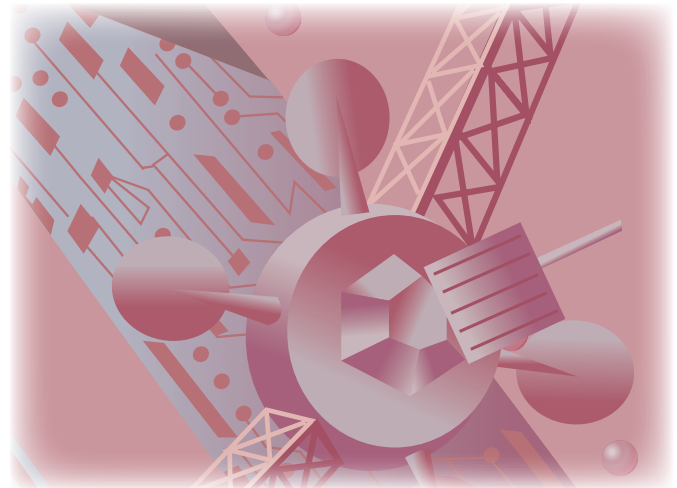
## Einführung

Im Laufe der letzten Jahre hat das Thema Softwarequalität mehr und mehr an Bedeutung gewonnen. Dieser Trend manifestiert sich unter anderem in der allgemeinen Verbreitung von Unit Testing, Ansätzen wie Test-Driven Development sowie Akzeptanztests zum Zeitpunkt der Abnahme. Mit Hilfe von Tests wird die äußere Qualität (Funktionalität, Korrektheit, Zuverlässigkeit, ...) der Software sichergestellt. Neben der äußeren Qualität spielt jedoch auch die innere Qualität einer Software eine entscheidende Rolle für den Erfolg eines Projekts.

Die innere Qualität resultiert aus der Struktur und dem Design der Software und wirkt sich auf die Wartbarkeit, Flexibilität, Erweiterbarkeit, Wiederverwendbarkeit und nicht zuletzt auch die Testbarkeit aus. Mit der Sicherung der inneren Qualität werden also überwiegend mittel- bis langfristige Ziele verfolgt, was vielleicht dazu beigetragen hat, dass ihr oft nicht die nötige Aufmerksamkeit zu Teil wird.

Dieser Artikel zeigt auf, wie die innere Qualität mit Hilfe von *Strukturanalyse* während des gesamten Entwicklungsprozesses auf einem hohen Niveau gehalten werden kann. Hierzu ist es einerseits erforderlich, den Entwicklern ein Werkzeug an die Hand zu geben, das sich nahtlos in die Entwicklungsumgebung einfügt und somit ein begleitendes Qualitäts-Monitoring ermöglicht. Analog zu Unit-Tests wird die Strukturanalyse so zu einem festen Bestandteil des Entwicklungsprozesses. Dadurch können Designschwächen frühzeitig erkannt und durch geeignete Refaktorisierungen behoben werden. Zusätzlich – aber nicht stattdessen – kann ein Architekt oder Qualitätsmanager sich bei Bedarf einen Überblick über den aktuellen Stand der inneren Qualität verschaffen. Auf der anderen Seite ist der Einsatz der Strukturanalyse auch sinnvoll im Rahmen einer Abnahme, um die Einhaltung vereinbarter Qualitätskriterien sicher zu stellen, z. B. bei Outsourcing-Projekten.

Die hier vorgestellten Ideen zur Strukturanalyse werden durch das Werkzeug „STAN – Structure Analysis for Java“ [STAN] in die Praxis umgesetzt. STAN ermöglicht Entwicklern, ihr Software-Design zu visualisieren, den Code besser zu verstehen, sowie verschiedene Aspekte der inneren Qualität zu messen und Qualitätsreports zu erstellen. STAN ist sowohl als Eclipse-Plug-In als auch als Standalone-Applikation verfügbar. Die in diesem Artikel vorgestellten Abhängigkeitsgraphen und Metriken werden unterstützt und die verwendeten Abbildungen wurden mit STAN erstellt.



## Strukturanalyse

Was versteht man unter Softwarestruktur, woraus setzt sich Software zusammen? Die Bausteine einer Code-Basis heißen *Artefakte*: Methoden und Felder bilden eine Klasse, Klassen werden in Pakete geschnürt, Pakete werden in Bibliotheken zusammengefasst. Eine Reihe von Bibliotheken ergibt schließlich eine Anwendung. Methoden, Felder und Klassen bilden die Code-Ebene, die darüber liegenden Artefakte bilden die Design-Ebene. Unter *Softwarestruktur* verstehen wir:

- ▼ wie sich die Artefakte zu Artefakten auf höherer Ebene zusammensetzen,
- ▼ wie die Artefakte voneinander abhängen.

Während der Entwicklung unterliegt die Softwarestruktur einer Code-Basis ständiger Veränderung. Zum Beispiel wird einem Paket eine neue Klasse hinzugefügt oder durch eine neue Methode entstehen zusätzliche Abhängigkeiten zu anderen Klassen und Paketen. Die Struktur unserer Software ist somit nicht nur ein beliebiger Aspekt des Designs. Vielmehr spiegelt sie unser Design direkt wider: Struktur ist Design!

So weit, so gut. Aber mit welchen Mitteln können wir die Struktur unserer Software greifbar machen? Hier bieten sich zwei Ansätze an:

- ▼ Die Visualisierung der Softwarestruktur durch Abhängigkeitsgraphen: *ein Bild sagt mehr als tausend Worte!* Abhängigkeitsgraphen sind ein unentbehrliches Mittel zum Code-Verständnis, sei es eigener oder auch fremder Code. Die Graphen stellen Abhängigkeiten zwischen Artefakten einer Ebene (Methoden und Felder einer Klasse, Klassen eines Pakets, Pakete einer Bibliothek oder der gesamten Anwendung, ...) oder auch zwischen Artefakten verschiedener Ebenen (z. B. eine Klasse mit den Artefakten auf der Design-Ebene, die mit ihr eine Abhängigkeitsbeziehung haben) dar.
- ▼ Die Messung von Struktureigenschaften durch Software-Metriken: *Vertrauen ist gut, Kontrolle ist besser!* Geeignete Metriken ermöglichen es, Schwachstellen im Design aufzudecken. Hierbei dienen Bewertungen der Messwerte dazu, die Struktur unserer Software gegen bewährte Design-Prinzipien (z. B. Zyklen-Freiheit auf der Design-Ebene oder die Vermeidung übermäßiger Komplexität) zu validieren. Die Priorisierung der Metrik-Verletzungen hilft bei der Unterscheidung von relevanten und weniger relevanten Problemen.

Abhängigkeitsanalyse und Qualitätsmetriken bilden die Basis der Strukturanalyse.

## Abhängigkeitsanalyse

Wie bereits erwähnt, bilden Abhängigkeiten zwischen Artefakten einen zentralen Bestandteil der Struktur einer Applikation. Für nicht-triviale Anwendungen ist es praktisch unmöglich, das Zusammenspiel zwischen Klassen, Paketen und Package-Trees anhand des Quellcodes zu verstehen oder gar zu steuern. Genau dies ist aber die Voraussetzung dafür, die richtigen Design-Entscheidungen während des Entwicklungsprozesses zu treffen. Wir können nicht erwarten, ein gutes Design zu erhalten, wenn wir es nicht einmal kennen!

Auf der Code-Ebene entstehen Abhängigkeiten durch Vererbung, den Aufruf einer Methode, den Zugriff auf ein Feld, Deklaration von Feldern und Methoden, lokale Variablen, usw. Abhängigkeiten zwischen Artefakten höherer Ebene entstehen durch Aggregation der Abhängigkeiten zwischen den jeweils enthaltenen Artefakten: Eine Methode einer Klasse aus Paket  $p_1$  ruft eine Methode einer Klasse aus Paket  $p_2$  auf und verursacht so eine Abhängigkeit zwischen  $p_1$  und  $p_2$ . Oft sind auch Abhängigkeiten zwischen Artefakten unterschiedlicher Ebenen von Interesse: Welche Pakete oder Bibliotheken verwenden eine bestimmte Klasse?

Für die Darstellung von Abhängigkeiten gibt es zwei gängige Ansätze: die Visualisierung durch Abhängigkeitsgraphen oder die Aufbereitung in Form von Abhängigkeits-Matrizen. Die für das Verständnis von Abhängigkeitsstrukturen besser geeignete Darstellung sind sicherlich Grafen. Im Folgenden wollen wir uns daher mit diesem Ansatz näher beschäftigen. Ein *Abhängigkeitsgraf* ist ein gerichteter Graf. Dabei bilden Artefakte die Knoten und Abhängigkeiten die Kanten des Grafen. Zusätzlich sind die Kanten mit der Anzahl der zugrundeliegenden Code-Abhängigkeiten gewichtet (s. Abb. 1).

Die Nützlichkeit von Grafen hat jedoch ihre Grenzen: Wird die Anzahl der Knoten und Kanten zu groß, wird der Graf unübersichtlich und Zusammenhänge sind nicht mehr erkennbar. Es ist daher wichtig, die darzustellenden Artefakte sorgfältig auszuwählen. Konzentriert man sich auf ein Artefakt, so lassen sich seine Abhängigkeiten aus zwei Perspektiven betrachten:

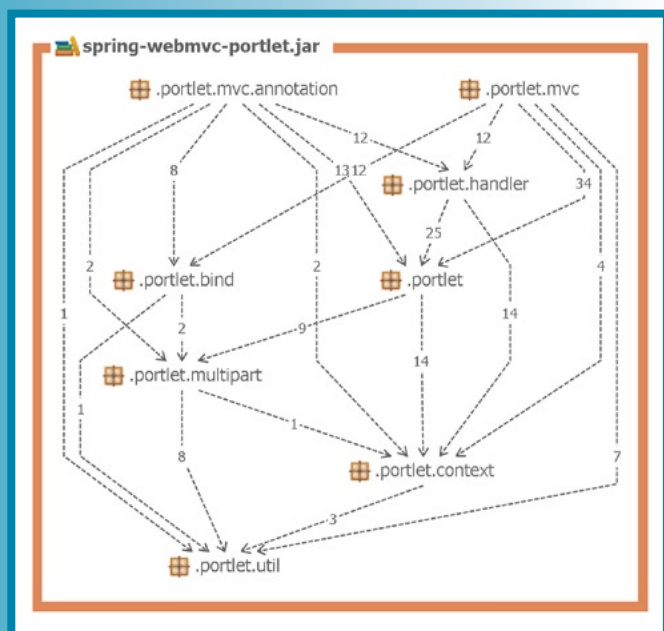


Abb. 1: Kompositionsgraf einer Spring-Bibliothek

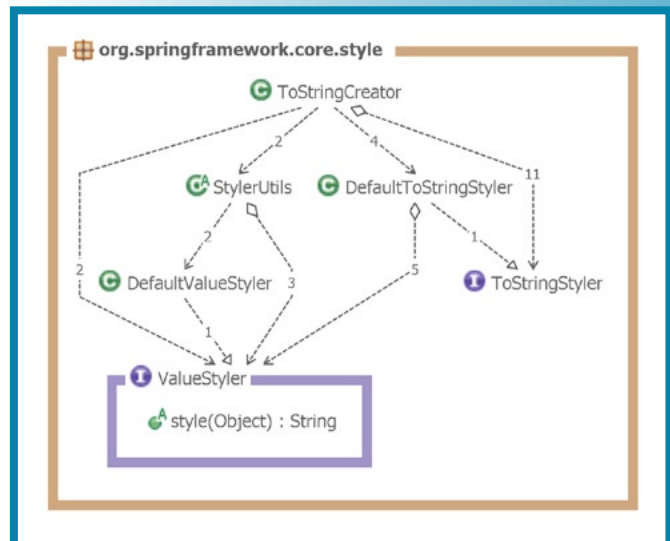


Abb. 2: Kompositionsgraf eines Spring-Pakets

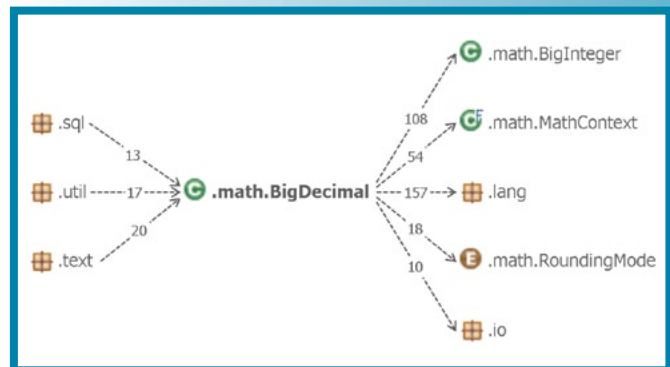


Abb. 3: Kopplungsgraf von java.math.BigDecimal

- ▼ Die *Komposition* eines Artefakts stellt die Abhängigkeiten zwischen den enthaltenen Artefakten und somit seine innere Struktur dar. Der Kompositionsgraf einer Klasse stellt die Beziehungen zwischen ihren Methoden, Feldern und inneren Klassen dar. Der Kompositionsgraf eines Pakets stellt die Beziehungen zwischen den enthaltenen Klassen dar (s. Abb. 2). Bei einem ausgewogenen Design ist die Komplexität gleichmäßig über die Anwendung verteilt: Eine Klasse beinhaltet nicht zu viele Methoden und Felder, ein Paket enthält nicht zu viele Klassen, usw. Dadurch ist gewährleistet, dass der Kompositionsgraf eines Artefakts übersichtlich bleibt.

- ▼ Die *Kopplung* eines Artefakts stellt die Abhängigkeiten von und zu anderen Artefakten und somit seine äußere Struktur dar. Diese Artefakte müssen nicht auf der gleichen Ebene liegen. Der Kopplungsgraf einer Klasse kann andere Klassen, aber auch Pakete und Bibliotheken enthalten, die mit der Klasse in Beziehung stehen (s. Abb. 3). Durch die Verschmelzung mehrerer Artefakte zu einem übergeordneten Artefakt wird auch hier eine übersichtliche Darstellung erreicht.

Mit Hilfe der vorgestellten Grafen – Komposition und Kopplung – haben wir nun die Möglichkeit, für beliebige Artefakte die Abhängigkeitsstruktur zu verstehen. Deren Visualisierung unterstützt uns dabei, die richtigen Design-Entscheidungen zu treffen, was letztlich zu einem verbesserten Software-Design führt.



## Zyklen

Ein *Zyklus* ist eine Menge von Kanten (Abhängigkeiten), die einen Kreis bilden. In einem Zyklus hängt jeder Knoten (Artefakt) von jedem anderen Knoten ab. Zyklen treiben daher die Anzahl der indirekten Abhängigkeiten im System in die Höhe. Zyklische Artefakte sind untrennbar. Sie können nur gemeinsam getestet, wiederverwendet, ausgeliefert und verstanden werden.

Glücklicherweise lassen sich Zyklen auf der Design-Ebene vermeiden. Das „Acyclic Dependencies Principle“ (ADP) fordert daher [Mar03]: „Allow no cycles in the package dependency graph.“

In der Praxis zeigt sich jedoch, dass sich bei wachsenden Anwendungen fast zwangsläufig Design-Zyklen einschleichen, wenn die Zyklenfreiheit nicht überwacht wird. Oftmals finden sich sogar Zyklen zwischen Bibliotheken, ohne dass dies den Entwicklern bewusst ist. Im Gegensatz zur Design-Ebene sind zyklische Abhängigkeiten auf der Code-Ebene zum Teil sogar erwünscht (Beispiel: Besuchermuster), aber auch hier gilt natürlich, dass unnötige Zyklen zu vermeiden sind. Des Weiteren dürfen Klassenzyklen gemäß ADP nicht durch mehrere Pakete laufen.

Zyklen sind wie Unkraut, das unser Design durchwuchert. Die Anzahl der Zyklen wächst und Zyklen vereinigen sich zu Knäueln (Tangles). Ab einem gewissen Grad ist es praktisch unmöglich, die unerwünschten Abhängigkeiten vollständig zu beseitigen. Das beste Mittel, Zyklenfreiheit zu garantieren, ist daher die frühe Entdeckung und Beseitigung. Letztere kann etwa im Rahmen eines iterativen Entwicklungsprozesses in einer Refaktorisierungsphase stattfinden.

Wie rücken wir nun den Tangles zu Leibe? Nehmen wir an, wir haben zyklische Abhängigkeiten zwischen Paketen und wollen diese aufbrechen. Wir wollen also eine Refaktorisierung durchführen, die unseren Abhängigkeitsgraphen so modifiziert, dass er azyklisch wird.

Hierzu wollen wir zunächst den Begriff Tangle etwas genauer fassen: Ein *Tangle* ist ein Subgraf mit mindestens zwei Knoten, in dem jeder Knoten von jedem anderen Knoten erreichbar ist. In der Grafentheorie sind Tangles auch als starke Zusammenhangskomponenten bekannt. Man kann sich relativ leicht überlegen, dass Tangles aus sich überlagernden Zyklen bestehen. Ein einfacher Zyklus ist somit eine Art minimalistischer Tangle. Wichtig ist aber vor allem die Erkenntnis, dass ein Graf genau dann azyklisch ist, wenn er keine Tangles enthält.

Anstatt einzelne Zyklen zu betrachten, können wir uns also lieber gleich die Tangles zur Brust nehmen. Das ist auch klug,

da in einem Tangle verblüffend viele Zyklen liegen können. Beispielsweise kann ein Graf mit nur 10 Knoten bereits über 1 Million Zyklen enthalten!

Die Aufgabe ist also, eine geeignete Menge von Kanten des Abhängigkeitsgraphen zu eliminieren, sodass dieser keine Tangles mehr enthält. Eine solche Kantenmenge heißt Feedback (Arc) Set. Die Kunst liegt nun also darin, ein Feedback Set zu bestimmen, welches den Refaktorisierungsaufwand möglichst klein hält. Dazu wird ein Feedback Set mit minimalem Kantengewicht (Anzahl der zugrunde liegenden Code-Abhängigkeiten) berechnet, das sogenannte *Minimum Feedback Set* (MFS).

Das MFS hat die folgenden sympathischen Eigenschaften:

- ▼ Durch die Entfernung der Kanten im MFS wird der Graf azyklisch.
- ▼ Die Anzahl der zu eliminierenden Code-Abhängigkeiten ist minimal.

Das MFS bildet somit die Sollbruchstellen der Tangles. Oder – mit anderen Worten – das MFS enthält die Abhängigkeiten, die in die „falsche“ Richtung weisen (s. Abb. 4). Das MFS liefert daher den Schlüssel zur Eliminierung zyklischer Abhängigkeiten.

Die letztendliche Refaktorisierung zur Auflösung der Tangles kann sich dann verschiedener, bekannter Methoden bedienen, etwa

- ▼ die Vereinigung mehrerer Pakete zu einem Paket,
- ▼ das Verschieben einzelner Klassen in andere Pakete,
- ▼ die Anwendung des Dependency-Inversion-Musters (Umkehrung einer Abhängigkeit durch Einführung eines Interface).

Insbesondere die erste Möglichkeit – das Vereinigen von Paketen – könnte auf den ersten Blick als „Killer-Methode“ zur Eliminierung von zyklischen Abhängigkeiten erscheinen. Aber Vorsicht, dadurch wird leicht ein anderes Prinzip verletzt: Der Aufbau eines Artefakts darf nicht zu komplex sein. Für ein Paket heißt das, dass die Anzahl der Abhängigkeiten zwischen den enthaltenen Klassen nicht zu hoch sein darf (Fat-Metrik, mehr dazu später). Dem simplen Zusammenführen von Paketen sind also Grenzen gesetzt.

## Metriken

Neben dem eher intuitiven Zugang über Abhängigkeitsgraphen bilden Qualitätsmetriken das zweite Standbein der Strukturanalyse. Diese erlauben die automatisierte Erhebung aller gewünschten Kenngrößen und ihren Abgleich gegen zuvor festgelegte Grenzwerte. So werden Designschwächen aufgedeckt, um sie anschließend durch geeignete Refaktorisierungen beseitigen zu können. Sehen wir uns nun den Begriff einer Metrik etwas näher an.

Das Wort *Metrik* kommt aus dem Griechischen und bedeutet zunächst einmal einfach „Zählung“ oder „Messung“. Für uns interessant ist die Softwaremetrik, wie sie im IEEE-Standard 1061 (von 1992) definiert ist [Wik08]: „Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.“

Neben dieser eher engen Definition gibt es auch weiter gefasste Ansätze, die auch den Prozess, den Aufwand, das Team, die Dokumentation und andere Aspekte eines Softwareprojektes umfassen. Für den hier verfolgten Zweck ist die Beschränkung auf die eher technischen Parameter der Software selbst jedoch adäquat.

Im Laufe der Jahre hat sich eine Vielzahl von Softwaremetriken angesammelt, die uns heute zur Verfügung stehen.

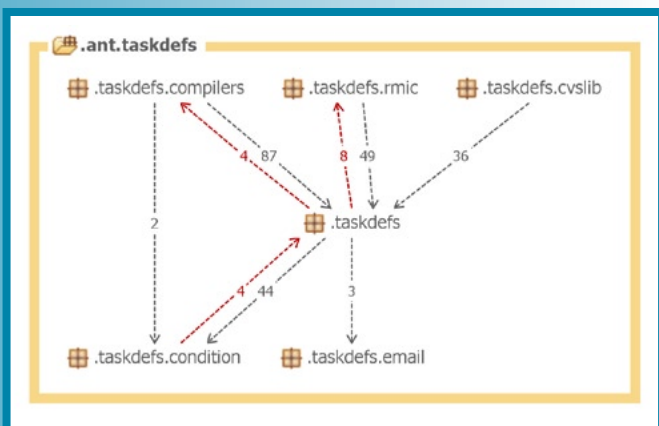


Abb. 4: Design-Tangle mit vier Paketen aus Ant (MFS rot)

Category/Metric	Value
Count	
Units	35
Classes / Class	0.23
Methods / Class	5.23
Fields / Class	1.29
ELOC	2168
ELOC / Unit	61.94
Complexity	
CC	2.33
Fat	25
ACD - Unit	2.94%
Robert C. Martin	
D	-0.3
A	0.63
I	0.08
Ca	158
Ce	13
Chidamber & Kemerer	
WMC	12.19
DIT	1.02
NOC	0.73
CBO	3.97
RFC	7.19
LCOM	3.83

Abb. 5: Metriken für ein Paket

Der sinnvolle Einsatz von Metriken zur Qualitätssicherung setzt deshalb zunächst einmal voraus, eine geeignete Auswahl zu treffen. Der Weg dorthin führt typischerweise ausgehend von der Klärung der zu erreichenden Ziele über die hierzu passenden konkreten Fragestellungen zu der Bestimmung von Metriken, die die gewünschten Antworten liefern. Ein solcher Ansatz wird auch als „Goal-Question-Metric“-Verfahren bezeichnet. Wir werden uns im Folgenden also gezielt denjenigen Metriken zuwenden, die Parameter der Softwarestruktur messen und uns somit darin unterstützen, dem Ziel eines gelungenen Designs näher zu kommen.

Im Rahmen dieses Artikels liegt der Fokus auf der Betrachtung der moderneren Metriken der zweiten und dritten Gruppe, während die Metriken der ersten und vierten Gruppe hier nicht weiter vertieft werden sollen.

Innerhalb aller Gruppen gibt es einerseits Basis-Metriken, die sich direkt aus der Code-Basis gewinnen lassen, und andererseits abgeleitete Metriken, die aus Werten von Basis-Metriken oder anderen abgeleiteten Metriken berechnet werden. Zu den abgeleiteten Metriken zählen vor allem Durchschnittswerte, wie z. B. die durchschnittliche zyklomatische Komplexität einer Methode, aber auch Quotienten, Summen oder Differenzen anderer Metriken, wie später im Abschnitt über die Distanz-Metrik zu sehen ist.

Nach der Auswahl der geeigneten Metriken gilt es im zweiten Schritt, geeignete Grenzwerte festzulegen. Es gibt zwar Ausnahmefälle, bei denen die Grenzwerte mehr oder weniger als inhärente Eigenschaften der Metrik erscheinen, so z. B. bei der Tangled-Metrik: Diese soll den Wert 0 haben, da Tangles unerwünscht sind, auch ganz kleine! Bei den meisten Metriken gibt es hingegen einen gewissen Bereich, in dem „vernünftige“ Grenzwerte liegen. Hier stößt das einfache Ja/Nein-Prinzip eines einzelnen Grenzwerts auch an gewisse Grenzen: Ist beispielsweise eine zyklomatische Komplexität von bis zu zehn immer akzeptabel und oberhalb von zehn immer schlecht? Einen differenzierteren Ansatz bieten Ampel-Bewertungen: durch Festlegung von zwei Grenzwerten definiert man drei Bereiche: einen unkritischen (grün), einen fraglichen (gelb) und einen fehlerhaften (rot).

Die im Rahmen der Strukturanalyse interessanteren Softwaremetriken lassen sich in vier Gruppen (s. Abb. 5) aufteilen:

## 1. Zählmetriken

Sie bestimmen die Anzahl von Einheiten eines Typs in einer höheren Einheit, beispielsweise

- ▼ Lines of Code eines beliebigen Artefakts,
- ▼ Anzahl der Methoden bzw. Felder einer Klasse,
- ▼ Anzahl der Klassen in einem Artefakt,
- ▼ Anzahl der Pakete einer Bibliothek bzw. Applikation.

## 2. Komplexitätsmetriken

Sie messen die innere Komplexität eines Artefaktes:

- ▼ Für Methoden bietet die zyklomatische Komplexität nach McCabe ein geeignetes Maß.
- ▼ Für Artefakte höherer Ebenen können aussagekräftige Metriken direkt aus ihren Abhängigkeitsgraphen abgeleitet werden, hierzu zählen die Metriken Fat, Tangled und Average Component Dependency.

## 3. Package-Metriken von Robert C. Martin

Sie dienen der Umsetzung des „Stable Abstractions“-Prinzips, hierzu zählen Abstraktheit, Instabilität und Distanz [Mar03].

## 4. Klassen-Metriken von Chidamber & Kemerer

Sie erfassen Kenngrößen bezüglich Vererbung, Komplexität, Kopplung und Kohäsion [ChK94]:

- ▼ WMC (Weighted Methods per Class): Summe einer Methodenmetrik (typischerweise die zyklomatische Komplexität) über alle Methoden einer Klasse,
- ▼ DIT (Depth of Inheritance Tree): Vererbungstiefe einer Klasse,
- ▼ NOC (Number of Children): Anzahl der direkten Subklassen einer Klasse,
- ▼ CBO (Coupling between Object Classes): Kopplungsgrad (Anzahl anderer Klassen mit einer Abhängigkeitsbeziehung),
- ▼ RFC (Response for a Class): Anzahl der Methoden der Klasse zuzüglich direkt aufgerufener Methoden anderer Klassen,
- ▼ LCOM (Lack of Cohesion in Methods): Mangel an Kohäsion (Zusammenhalt) einer Klasse (Berechnung über Felder-Zugriffe der einzelnen Methoden).

## Komplexitätsmetriken

Die oben aufgeführten Komplexitätsmetriken dienen der Erfassung unterschiedlicher Aspekte der Komplexität von Artefakten. Allen ist gemeinsam, dass sie aus Grafen abgeleitet werden. Für Methoden als elementare Artefakte wird der zugehörige Kontrollflussgraph zur Bestimmung der *zyklomatischen Komplexität* (CC) nach McCabe herangezogen: sie ergibt sich aus der Anzahl der binären Verzweigungspunkte im Kontrollflussgraphen plus 1. Die zyklomatische Komplexität ist unter anderem ein wichtiges Maß für die Abschätzung des zu erwartenden Testaufwands für eine Methode, da in den Tests sicher gestellt werden sollte, dass alle Kanten des Kontrollflussgraphen auch durchlaufen werden (Testabdeckung).

Mit Blick auf die Strukturanalyse stehen zwar eher die im Folgenden vorgestellten Metriken Fat und Tangled im Mittelpunkt, die sich aus den Kompositionsgraphen von Artefakten ergeben. Die zyklomatische Komplexität muss jedoch immer auch mit berücksichtigt werden, da sie verhindert, dass überschüssige Komplexität von höheren Ebenen einfach in einzelne immens komplizierte Methoden verlagert wird.

Betrachten wir nun noch einmal die oben eingeführten Kompositionsgraphen im Hinblick auf die aus ihnen abgeleiteten Metriken. Für alle Artefakte von Klassen an aufwärts gibt es Grafen, die die jeweils enthaltenen Artefakte samt der zwischen ihnen bestehenden Abhängigkeiten darstellen. Die *Fat*-Metrik eines Artefakts ist nun einfach die Anzahl der Kanten in seinem Kompositionsgraphen (ohne Berücksichtigung der Gewichte). Dies ist also gerade die Anzahl der Abhängigkeiten zwischen den enthaltenen Artefakten der nächst tieferen Stufe. Das in Abbildung 2 dargestellte Paket hat somit einen Fat-Wert von 9, die in Abbildung 1 dargestellte Bibliothek hat einen Fat-Wert von 20. Die *Fat*-Metrik verhindert so auf allen Artefakt-Ebenen oberhalb von Methoden, dass die inneren Abhängigkeitsstrukturen einzelner Artefakte zu komplex werden. Insbesondere ist sie es, die dem beliebigen Zusammenlegen von Paketen zur Vermeidung von Design-Tangles einen Riegel vorschiebt.



Bleibt noch zu klären, wie es denn nun die Tangled-Metrik schafft, genau dann anzuschlagen, wenn irgendwo ein Design-Tangle vorhanden ist. Der Schlüssel hierzu liegt im bereits erwähnten MFS (Minimum Feedback Set). Das MFS enthält die (in Hinblick auf die Kantengewichte, also die Anzahl der Code-Abhängigkeiten) minimale Kantenmenge eines Abhängigkeitsgraphen, die entfernt werden muss, damit der Graf zyklensfrei wird. Somit ist das MFS genau dann leer, wenn der Graf keine Zyklen (und damit auch keine Tangles) enthält. Definiert man nun die *Tangled*-Metrik als Quotient aus der Summe der Kantengewichte des MFS und der Summe der Gewichte aller Kanten im Abhängigkeitsgraphen, so hat sie genau dann einen Wert größer als 0, wenn der Graf Zyklen enthält. Um alle Design-Zyklen als fehlerhaft einzustufen, wird also einfach der Grenzwert der Tangled-Metrik für Package-Trees auf 0 gesetzt (für Pakete wird die Metrik nicht erhoben, da Klassen-Zyklen nicht verboten sind). Im Beispiel aus Abbildung 4 ergibt sich ein Tangled-Wert von ca. 7 Prozent.

Es ist nun in erster Linie dieses Dreigespann aus zyklomatischer Komplexität, Fat und Tangled, das eine ausgewogene und zyklensfreie Struktur der Anwendung sicherstellt. Die Fat-Metrik für Pakete verhindert, dass diese im Lauf des Entwicklungsprozesses zu komplex werden. Einen scheinbaren Ausweg aus dieser Situation bietet das wiederholte Zusammenlegen von enthaltenen Klassen. Dies senkt die Anzahl der Abhängigkeiten zwischen Klassen im Paket und damit den Fat-Wert des Pakets. Es führt jedoch zu einer Steigerung des Fat-Werts der einzelnen Klassen und ist somit kein Patentrezept. Versucht man wiederum dies zu umgehen, indem man innerhalb der Klassen Methoden zusammenlegt, steigt deren zyklomatische Komplexität. Auch dies ist also nur in Maßen möglich. Es bleibt daher nur die Möglichkeit, Pakete mit zu hohen Fat-Werten in kleinere Pakete zu zerlegen, also Änderungen auf der Design-Ebene vorzunehmen. Auf dieser greift nun die Tangled-Metrik und sorgt dafür, dass die so neu entstehenden Paket-Abhängigkeiten dem Acyclic Dependencies Principle genügen.

Als letzte im Bunde der Komplexitätsmetriken gibt es noch die *Average Component Dependency* (ACD, s. z. B. [Fle07]). Sie misst ebenso wie die Fat-Metrik das Ausmaß der Abhängigkeiten in einem Kompositionsgraphen. Im Unterschied zu dieser berücksichtigt sie jedoch zusätzlich indirekte Abhängigkeiten und liefert ein Maß für die durchschnittliche Anzahl der Knoten im Graf, von denen ein gegebener Knoten abhängig ist.

### Die Distanz-Metrik

Im Laufe der 90er Jahren stellte Robert C. Martin einige Prinzipien für gelungenes Software-Design vor. Eines dieser Prinzipien ist das „Stable Abstractions Principle“. Es setzt zwei Package-Metriken zueinander in Beziehung: die Abstraktheit, die den Anteil der abstrakten Typen im Paket wiedergibt, und die Instabilität, die ein Maß dafür ist, ob das Paket vorwiegend von anderen Artefakten benutzt wird („stabil“) oder ob es vorwiegend andere Artefakte benutzt („instabil“). Das „Stable Abstractions Principle“ (SAP) lautet [Mar03]: „A package should be as abstract as it is stable.“

Das Prinzip dient vor allem der Vermeidung von Paketen, die vom Rest der Applikation vielfach benutzt werden und die gleichzeitig nur wenig abstrahiert sind. Diese Pakete verursachen in der Praxis immer wieder Probleme, da sie nur schwer modifizierbar oder erweiterbar sind. Auf der anderen Seite soll auch verhindert werden, dass Pakete, die kaum oder gar nicht von anderen benutzt werden, übermäßig abstrahiert werden, da dies als Verschwendung von Arbeitskraft angesehen wird.

Die Basis für die Formalisierung des SAP bilden die Package-Metriken Abstraktheit und Instabilität:

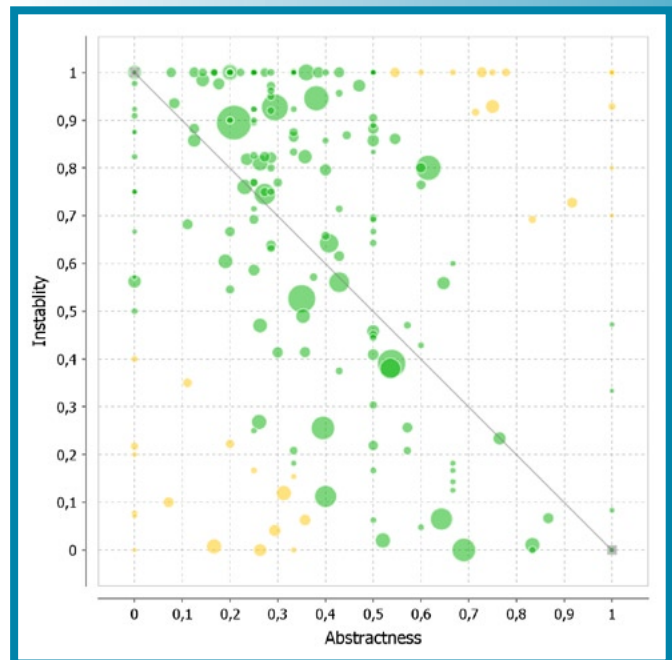


Abb. 6: Distanz-Diagramm für Spring

- ▼ Die *Abstraktheit*  $A$  eines Pakets  $p$  ist das Verhältnis zwischen der Anzahl der abstrakten Typen in  $p$  und der Anzahl aller Typen in  $p$ . Somit liegen die möglichen Werte zwischen 0 (ausschließlich konkrete Klassen) und 1 (ausschließlich Interfaces und abstrakte Klassen).
- ▼ Die *Instabilität*  $I$  eines Pakets  $p$  ist das Verhältnis zwischen der Anzahl der Klassen außerhalb von  $p$ , die von  $p$  benutzt werden, und der Anzahl aller Klassen außerhalb von  $p$ , die in einer Abhängigkeitsbeziehung zu  $p$  stehen. Wie zuvor liegen die möglichen Werte zwischen 0 (ausschließlich eingehende Abhängigkeiten) und 1 (ausschließlich ausgehende Abhängigkeiten).

Im nächsten Schritt werden alle Pakete in einem Diagramm platziert, das das Einheitsquadrat mit der Abstraktheit auf der horizontalen Achse und der Instabilität auf der vertikalen Achse darstellt (s. Abb. 6).

Übertragen auf das so entstehende Bild lässt sich das Stable Abstractions Principle wie folgt umformulieren: „Die Pakete sollten nicht zu weit entfernt von der abfallenden Diagonalen des Diagramms liegen, die auch als Hauptreihe (Main Sequence) bezeichnet wird.“

Auf der Hauptreihe liegen eben gerade die Pakete mit hoher Instabilität und geringer Abstraktheit (links oben), genauso wie diejenigen mit niedriger Instabilität und hoher Abstraktheit (rechts unten) oder diejenigen mit jeweils mittleren Werten (im Zentrum des Diagramms). Je weiter wir uns von der Hauptreihe entfernen, desto problematischer sind die dort anzutreffenden Pakete. In den beiden Ecken des Diagramms, die nicht zur Hauptreihe gehören, liegen die speziellen Problemzonen mit den als vermeidenswert eingestuften Eigenschaftskombinationen:

- ▼ Links unten liegt die „Zone of Pain“, deren Pakete gleichzeitig konkret und stabil sind, also von größeren Teilen der Anwendung benötigt werden.
- ▼ Rechts oben liegt die „Zone of Uselessness“ mit Paketen, die abstrakt und instabil sind, also kaum von anderen Anwendungsteilen verwendet werden.

Um die Qualität eines Pakets in Hinblick auf das Stable Abstractions Principle beurteilen zu können, fehlt nun nur noch

eine Metrik, die wiedergibt, ob das Paket eher auf der Hauptreihe oder eher in einer der gefürchteten Ecken liegt. Genau dies leistet die *Distanz*-Metrik. Sie berechnet sich als  $D = A + I - 1$ .

Auf diese Weise liefert sie Werte zwischen -1 und 1, da *A* und *I* ja jeweils zwischen 0 und 1 liegen. Pakete mit einer Distanz von 0 liegen genau auf der Hauptreihe, Pakete mit einer positiven Distanz liegen oberhalb, mit einer negativen unterhalb der Hauptreihe. In Verbindung mit einer geeigneten Bewertung der Distanz-Metrik ergibt sich ein mehr oder weniger breiter Korridor entlang der Hauptreihe, der die SAP-gemäßen Pakete enthält und zwei dreieckige Eckbereiche mit kritischen Paketen. Trägt man nun alle Pakete einer Anwendung in das Diagramm ein, so ergibt sich ein hilfreicher Überblick in Hinblick auf das Stable Abstractions Principle (s. Abb. 6). Jeder Kreis steht für ein Paket, die Größe hängt von der Anzahl der Klassen ab. Die Farbe eines Kreises spiegelt die Bewertung wider, diese wurde in der Abbildung für Distanz-Werte von -0,5 bis +0,5 auf Grün und außerhalb dieses Intervalls auf Gelb gesetzt.

## Fazit

Strukturanalyse ist ein geeignetes Mittel zur Sicherung der inneren Qualität. Einerseits trägt die Visualisierung von Abhängigkeiten zum Verständnis der vorhandenen Design-Strukturen bei und unterstützt dadurch den Architekten und Entwickler aktiv bei Entscheidungen bezüglich Design und Refaktorisierung. Andererseits gewährleisten geeignete Qualitätsmetriken die Einhaltung von bewährten Design-Prinzipien. Voraussetzung für eine erfolgreiche Anwendung ist dabei der kontinuierliche Einsatz eines Werkzeugs, das die nahtlose Integration von Strukturanalyse in den Entwicklungsprozess ermöglicht.

Hier bietet STAN eine schlanke, flexible und intuitive Lösung. Die Abhängigkeitsgraphen bieten vielfältige Möglichkeiten zur Interaktion wie Expansion von Knoten, Navigation, usw. Die den Kanten der Graphen zugrunde liegenden Code-Abhängigkeiten werden durch Selektion aufgelistet. Das Eclipse-Plug-

In ermöglicht darüber hinaus die Anzeige des dazugehörigen Quellcodes im Java-Editor. Metrikbewertungen sind individuell einstellbar, Verletzungen werden priorisiert und bei Bedarf gefiltert. Abfragen ermöglichen die gezielte Suche nach Artefakten über Metriken. Konfigurierbare Reports liefern einen Überblick über die relevanten Aspekte der inneren Qualität. Schließlich erlaubt eine Ant-Task die Integration der Report-Generierung in den Build-Prozess.

## Literatur und Links

- [ChK94] Sh. R. Chidamber, Ch. F. Kemerer, A Metrics Suite for Object Oriented Design, in: IEEE Transactions on Software Engineering, Vol. 20, No. 6, Juni 1994
- [Fle07] A. Fleischer, Metriken im praktischen Einsatz, in: OBJEKTSpektrum, 03/2007
- [Mar03] R. C. Martin, Agile Software Development – Principles, Patterns, and Practices, Prentice Hall, 2003
- [STAN] STAN – Structure Analysis for Java: <http://stan4j.com>
- [Wik08] Wikipedia, <http://de.wikipedia.org/wiki/Softwaremetrik> (Stand Mai 2008)



**Christoph Beck** ([beck@odysseus.de](mailto:beck@odysseus.de)) und **Oliver Stuhr** ([stuhr@odysseus.de](mailto:stuhr@odysseus.de)) sind Geschäftsführer der Odysseus Software GmbH in Frankfurt am Main. Ihre Schwerpunkte liegen in Analyse, Design und Implementierung im Java/JEE-Umfeld. Aktuelle Themen sind Softwarequalität sowie die Eclipse Rich Client Platform. Die Autoren engagieren sich in verschiedenen Open-Source-Projekten ([calyx.org](http://calyx.org), [juel.sf.net](http://juel.sf.net)) und sind an der Entwicklung des Analyse-Werkzeugs STAN beteiligt.

## SONDERDRUCK

aus JavaSPEKTRUM 5/08 für  
Odysseus Software GmbH



JavaSPEKTRUM ist eine Fachpublikation des Verlags:  
SIGS DATACOM GmbH · Lindlaustraße 2c · 53842 Troisdorf  
Tel.: 0 22 41/23 41-100 · Fax: 0 22 41/23 41-199  
E-mail: [info@sigs-datacom.de](mailto:info@sigs-datacom.de) · [www.javaspektrum.de](http://www.javaspektrum.de)  
[www.sigs.de/publications/aboservice.htm](http://www.sigs.de/publications/aboservice.htm)

