KaffeeKlatsch
Das Magazin rund um Software-Entwicklung
ISSN 1865-682X
01/2011

Das Auge (m)isst mit!

von der Darstellung innerer Werte

von Thomas Haug

Copyright © 2011 Bookware 1865-682x/11/01/004

Das Auge (m)isst mit

Von der Darstellung innerer Werte

von Thomas Haug

Die Code-Analyse von Software kann mitunter sehr trocken sein. Dass man hierbei nicht nur Code und nackte Zahlen gewonnen aus Software-Metriken interpretieren muss, soll in diesem Artikel gezeigt werden. Gerade die visuelle Darstellung von Metriken kann helfen, Klassen von besonderer Größe, Komplexität, Kopplungsgrad und Vererbungstiefe schnell zu identifizieren, ohne langweilige Zahlenkolonnen lesen zu müssen. In diesem Artikel werden verschiedene Ansätze vorgestellt um Metriken zu visualisieren, mit dem Ziel potentiell problematische Klassen schnell identifizieren zu können.

Polymetrische Views

Bereits in den Artikeln [1], [2] und [3] wurden Metriken vorgestellt und gezeigt, dass mittels dieser Aussagen über die Qualitätsaspekte der bewerteten Artefakte getätigt werden können. Sind die bewerteten Systeme groß, so kann die schiere Anzahl an Messwerten leicht unübersichtlich werden, sodass selbst geübte „Software-Analytiker“ Schwierigkeiten haben werden, die Spreu vom Weizen zu trennen. In [3] und [4] wurde gezeigt, wie durch geeignete Kombination von Metriken die Zahl der zu analysierenden Klassen reduziert werden kann. Was aber, wenn lediglich ein kurzer Blick auf die Software Auskunft über dessen Qualität geben soll? In diesem Fall muss eine visuelle Darstellung gefunden werden, die möglichst viele Aspekte im Sinne von Metriken der Software abdecken soll. Hierzu wurde von Lanza und Marinescu in [5] der sogenannte *polymetric view* definiert.

Die Polymetrische Darstellung

Im *polymetric view* können bis zu vier Metriken simultan aufgezeigt werden. Als dargestellte Artefakte können

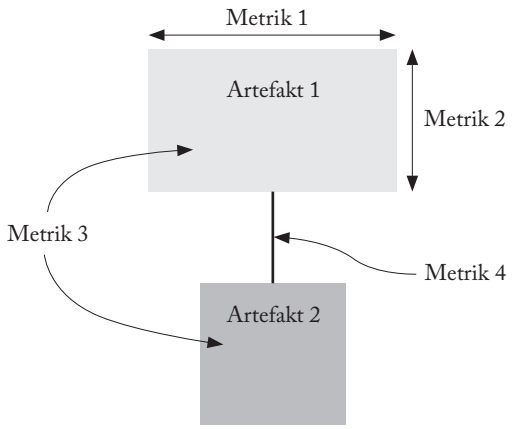

Abbildung 1: *Polymetric view*

Namensräume (Packages), Klassen oder Methoden dienen. Jedes einzelne Artefakt kann mit maximal drei Metriken gekennzeichnet werden: Breite, Höhe und Einfärbung. Stehen zwei Artefakte in Beziehung, so kann dies durch eine Verbindung visualisiert werden. Auch hier ist es möglich durch die Breite der Linie Aussagen über die Beziehung zu machen. Bei der Einfärbung werden bewusst Graustufen genutzt, da das menschliche Gehirn Grautöne im Gegensatz zu anderen Farben gut in eine Reihenfolge von Hell nach Dunkel bringen kann. Im Folgenden wird ein Beispiel dieser Darstellung erläutert.

System Complexity View

In [5] wird die polymetrische Darstellung zur Veranschaulichung der System-Komplexität genutzt. Hierzu werden alle Klassen des Systems visualisiert. Die Breite einer Klasse wird durch dessen Attribute (Number of Attributes (NOA)) bestimmt, während die Höhe durch die Anzahl der Methoden (Number of Methods (NOM)) festgelegt ist. Stehen zwei Klassen in einer Vererbungsbeziehung, so wird dies durch eine Verbindung symbolisiert, wobei die abgeleitete Klasse unterhalb der Superklasse angeordnet wird. Die Linienbreite hat in dieser Darstellung keinerlei Aussagekraft und wird konstant gehalten.

Zusätzlich wird mittels der Einfärbung der Klassenrechtecke die Gewichtung die im Sinne der McCabe-Komplexität bewertet.

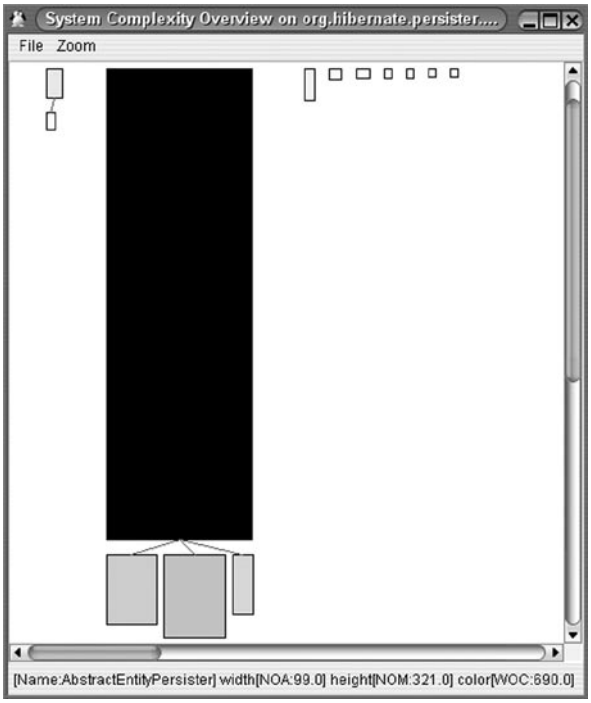

Abbildung 2: System Complexity View

In den Werkzeugen IPlasma [6] und dessen Nachfolger InFusion [7] kann man diese Ansicht nutzen, um die Komplexität eines Systems bzw. eines Packages innerhalb eines analysierten Systems zu visualisieren. Wählt man diese Ansicht auf ein komplettes System, so kann die Ansicht schnell unübersichtlich sein; aus diesem Grund wird im folgenden Screenshot (Abbildung 2) der System Complexity View auf ein Subsystem von Hibernate 3.5.6 angewendet, das die Persistierung von Java-Objekten in relationale Datenbanken ermöglicht.

Abgesehen von den recht unspektakulär „daherkommenden Klassen" auf der rechten Seite, fällt das tief schwarz eingefärbte Rechteck auf, das die Klasse *AbtractEntityPersister* charakterisiert. Diese Klasse besitzt 99 Attribute und 321 Methoden. Der Weigthed Method Count der Klasse wird auf 690 beziffert, mit diesem Wert ist die Klasse die mit Abstand komplexeste des Hibernate Frameworks im Sinne der zyklomatischen Komplexität. Zusätzlich leiten von dieser Klasse drei konkrete Klassen ab

- *JoinedSubclassEntityPersister,*
- *SingleTabelEntityPersister* und
- *UnionSubclassEntityPersister*,

die alle samt im Vergleich zu den übrigen Klassen des Packages groß im Bezug auf hinzugefügte Methoden und Attribute sind.

Diese Metrik-Ansicht eignet sich, um schnell komplexe Klassen in einem System aufzuspüren und kann Indizien für problematische Stellen im Bezug auf Erweiterung und Änderung eines Systems liefern.

Blaupause

Hat man mit dem System Complexity View potentiell problematische Klassen identifiziert, so sollten diese einer weiteren Analyse unterzogen werden. Dies könnte im Editor des eingesetzten Integrated Development Environments (IDE) geschehen, aber auch hierfür beschreiben Lanza und Marinescu in ihrem Buch eine weitere polymetrische Ansicht, den sogenannten Class Blueprint. Ziel dieser Ansicht ist es dem Entwickler ein Modell an die Hand zu geben, das den Ablauf von Methodenaufrufen bis hin zu Attributzugriffen zeigt. Hierzu werden in dieser Ansicht sowohl Methoden als auch Attribute einer Klasse dargestellt.

Die Ansicht der Klassen ist in fünf Spalten unterteilt. Die „Öffentlichkeit" der Klasse nimmt im Allgemeinen von links nach rechts ab. (siehe Abbildung 3)

Zur Einteilung der Klassenmethoden in dieses Schema werden in [5] die nachfolgenden Heuristiken verwendet.

Alle Konstruktoren und Methoden mit einem Bestandteil *init* werden als Initialisierungsmethoden identifiziert. Dies bedeutet aber auch, dass in Java gebräuchliche Methoden wie *newInstance()* und *getInstance()* nicht als Initialisierungsmethode erkannt

Initialisierungs-methoden	Externe Schnittstellen Methoden	Internal Implementierungs-methoden	Zugriffsmethoden	Attribute

Abbildung 3: Class Blueprint

werden, sehr wohl aber eine Methode mit dem Namen *getFiniteElemente()*.[1]

Zur Menge der Externen Schnittstellenmethoden werden alle Methoden gezählt, die entweder als *protected* oder *public* markiert sind und ausschließlich von Initialisierungsmethoden der Klasse oder von anderen Klassen aufgerufen werden. Die Methoden dieser Kategorie sind der Eintrittspunkt von anderen Klassen zur bereitgestellten Funktionalität.

Implementierungsmethoden enthalten alle mit *private* markierten Methoden und alle Methoden, die sowohl von außen als auch direkt von der Klasse aufgerufen werden. Das heißt, wird eine *public*-Methode sowohl von einer anderen Klasse als auch von einer Methode dieser Klasse aufgerufen, so ist sie nicht Bestandteil der externen Schnittstelle, sondern wird als interne Implementierungsmethode gesehen. Über diese Heuristik lässt sich trefflich streiten, da argumentiert werden könnte, dass öffentliche Methoden grundsätzlich zur externen Schnittstelle gehören und die Nutzung dieser Methoden innerhalb der Klasse doch eine Art der Wiederverwendung ist.[2]

Die Kategorie Zugriffsmethoden stellt eine Ausnahme dar, da in dieser alle Methoden zusammengefasst werden, die ausschließlich dem Zugriff bzw. der Änderung von Attributen dienen. In der Java-Welt sind dies die *Getter*- und *Setter*-Methoden. Diese Methoden werden oftmals sowohl von der eigenen als auch von anderen Klassen genutzt.

In der letzten Kategorie werden alle Attribute zusammengefasst. Hierbei spielt Sichtbarkeit oder Geltungsbereich (class vs member scope) keine Rolle.

Werden alle Kategorien in Betracht gezogen, so ist festzustellen, dass im Allgemeinen Aufrufe an Objekten einer so dargestellten Klasse von links nach rechts ausgeführt werden, d. h. diese Ansicht „führt" den Betrachter beim Lesen des Diagramms und kann so helfen, die innere Struktur einer Klasse schneller zu erkennen.

Dies ist aber längst nicht alles. Nachdem die Methoden und Attribute den verschiedenen Spalten zugeteilt werden können, kommen bei der Darstellung dieser Artefakte als Rechtecke ähnlich zum *polymetric view* unterschiedliche Metriken zum Einsatz.

Bei Methoden wird über die Breite die Anzahl der Aufrufe durch die Methode dargestellt, während die Höhe die Lines of Code (LOC) charakterisieren. Bei Attributen wird mit der Breite die Anzahl externer Zugriffe angezeigt, d. h. Zugriffe auf diese Attribute über Getter, Setter und direkte Zugriffe. Die Höhe eines Attributs entspricht der Anzahl der internen Zugriffe. Des Weiteren wird im Class Blueprint der Aufruf von Methoden bzw. der Zugriff von Methoden auf interne Attribute durch Verbindungen der entsprechenden Artefakte symbolisiert: blaugrüne Linien stellen den Zugriff auf ein Attribut dar, während dunkelblaue Linien den Aufruf von Methoden zeigen.

Im folgenden Screenshot wird exemplarisch der *class blueprint* der Klasse *org.hibernate.engine.CollectionLoadContext* gezeigt.

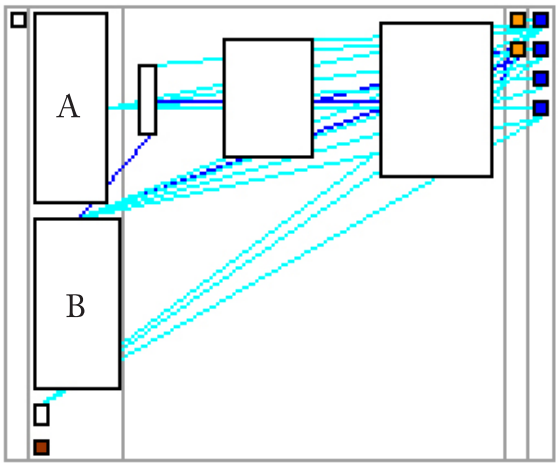

Abbildung 4: Beispiel Blaupause

Diese Klasse besitzt einen Konstruktor (auf der linken Seite) und vier Methoden, die der Kategorie Externe Schnittstellen-Methoden zugeordnet werden. Im Screenshot sind die oberen beiden Methoden mit *A* und *B* gekennzeichnet, dies ist nicht Bestandteil des Werkzeugs bzw. des Class Blueprints, sondern dient der Erläuterung. Drei Methoden sind interne Methoden und werden im Zuge des Aufrufs der externen Methode *B* (der eigentliche Methodenname ist *endLoadingCollections*) gerufen. Die vierte Methode der externen Methoden mit dem Namen *toString* ist braun eingefärbt, da sie eine überschreibende Methode ist; Sie überschreibt die Methode *toString* der Klasse *java.lang.Object*. Wie zu sehen ist, greifen die Methoden der Klasse teilweise direkt auf die Attribute der Klasse zu, es wird aber auch über *getter*-Methoden auf die Attribute zugegriffen, was durch die blauen Verbindungen zu den ockerfarbenen Rechtecken in der Zugriffsmethoden-Kategorie symbolisiert wird. Hätte die Klasse *setter*-Methoden, so würden diese als rote Rechtecke dargestellt werden.

[1] Ein Werkzeug, das diese Heuristik verwendet, sollte die Möglichkeit einer Anpassung bieten. Über diesen könnte Sprach- und Projekt-abhängig definiert werden, welche Methoden der Initialisierungsmethoden-Kategorie zuzuordnen sind. Leider ist dies bei IPlasma und InFushion nicht gegeben.

[2] Der Autor würde aus diesem Grund öffentliche Methoden grundsätzlich der Kategorie Externe Schnittstellenmethoden zuordnen.

Anwendung der polymetrischen Darstellung und der Blaupause

Die Anwendung der beiden vorgestellten Darstellungen wird im Folgenden exemplarisch an einem Package des Frameworks demonstriert. Hierbei wird sowohl InFushion als auch die MOOSE Plattform [8] zur Visualisierung des Designs genutzt. Ziel der Code-Analyse ist das Package *org.hibernate.ejb*. Wird dieses in InFushion geöffnet, so wird der folgende *polymetric view* angezeigt, der eine Abart des System Complexity Views darstellt.

Abbildung 5: Package overview

Die Breite der Klassen zeigt wiederum die Anzahl der Attribute, aber die Höhe gibt in diesem Fall Auskunft über die Komplexität (WMC). Die Einfärbung gibt Auskunft, ob die Klasse keine Design- (grau), Methodendesign- (weiss) oder Klassendesignprobleme (gestrichelt) besitzt. Offensichtlich werden vier Klassen als problematisch gekennzeichnet. Wird die Klasse *Ejb3Configuration* gewählt, so zeigt sich, dass diese ein Klassendesignproblem hat, das sich durch eine Vielzahl von Methodendesignproblemen ergibt (siehe Abbildung 6).

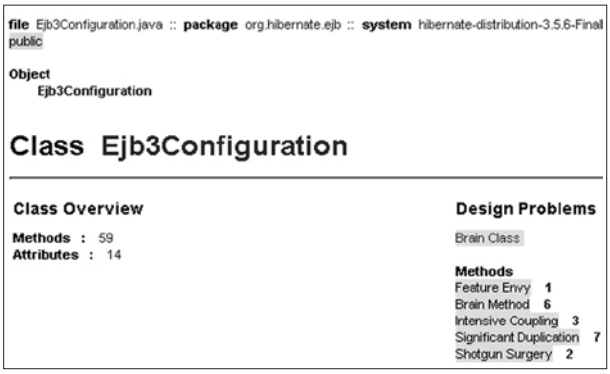

Abbildung 6: Klassendetails *Ejb3Configuration*

Laut der Analyse von InFushion besitzt diese Klasse sechs Methoden, denen der Code Smell *brain method* anhaftet. Dieser Geruch besagt, dass eine solche Methode einen Großteil der Funktionalität der gesamten Klasse bündelt und oft sehr komplex ist. Hierzu werden in [5] folgende Metriken zum Aufdecken dieser Schwäche kombiniert:

- Line of Code (LOC) ist größer als die Hälfte des empirisch ermittelten Werts für sehr lange Klassen, d. h. länger als 65 Zeilen (siehe hierzu [2] und [5])
- Die zyklomatische Komplexität der Methode ist größer als 3
- die maximale Verschachtelungstiefe ist größer 3
- Es werden mehr als 7 – 8 Attribute innerhalb der Methode verwendet.

Der Class Blueprint dieser Klasse ist in Abbildung 7 gezeigt. Die Methoden, die als *brain methods* erkannt wurden, sind zur besseren Nachvollziehbarkeit mit den Ziffern 1 – 6 markiert. Die Methode mit der Ziffer 6 ist *addClassesToSessionFactory*. Diese Methode ist 77 Zeilen lang, hat eine zyklomatische Komplexität von 12, ihre maximale Verschachtelungstiefe ist 4 und sie greift auf insgesamt 19 Attribute der Klasse bzw. Attribute anderer Klassen zu. Somit treffen alle oben genannten Indizien zu.

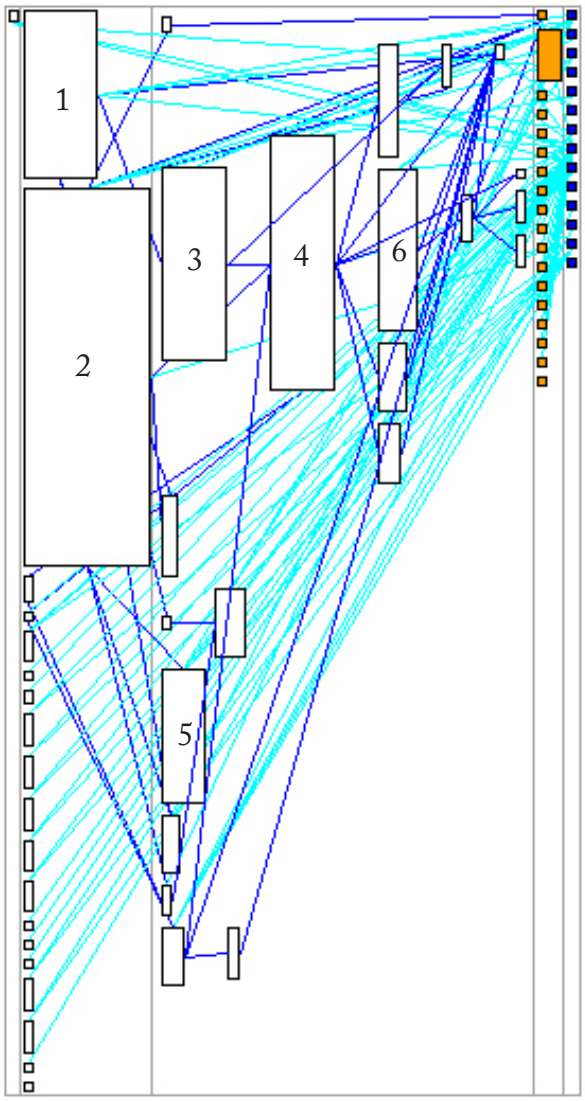

Abbildung 7: Class blueprint von *Ejb3Configuration*

Wie zu erkennen ist, werden Methoden leicht identifiziert, die im Vergleich zu den übrigen Methoden groß sind. Die Abhängigkeiten der Methoden sind gut durch diese Darstellung erkennbar. Der Zugriff von Methoden auf Attribute ist zwar dargestellt, aber die Zuordnung von Methoden zu Attributen fällt bereits jetzt schon schwer. Bei der gewählten Designschwäche spielt die Länge einer Methode und der Zugriff auf Attribute eine wesentliche Rolle, somit kann dieser Class Blueprint hierbei gute Dienste leisten. Werden aber Designschwächen gesucht, die auf anderen Metriken basieren, so kann dieser kaum Hilfestellung geben. Hier bedarf es weiterer Ansichten.

Fazit

In diesem Artikel wurden drei Darstellungen von Metriken vorgestellt, die das Auffinden von Designproblemen erleichtern sollen.

Mit dem System Complexity View bzw. dem Package Overview können potentiell problematische Klassen identifiziert werden. Die interne Struktur dieser Klassen kann mit dem Class Blueprint visualisiert werden und dieser kann helfen die gefundenen Designschwächen besser zu verstehen.

Vorsicht ist bei sehr großen Klassen bei der Verwendung des Class Blueprints geboten, denn dieser kann sehr schnell unübersichtlich werden, wie die folgende Abbildung 8 einer Klasse aus dem Hibernate Framework zeigt. In dieser Blaupause lassen sich Methoden- und Attributabhängigkeiten nicht mehr ermitteln.

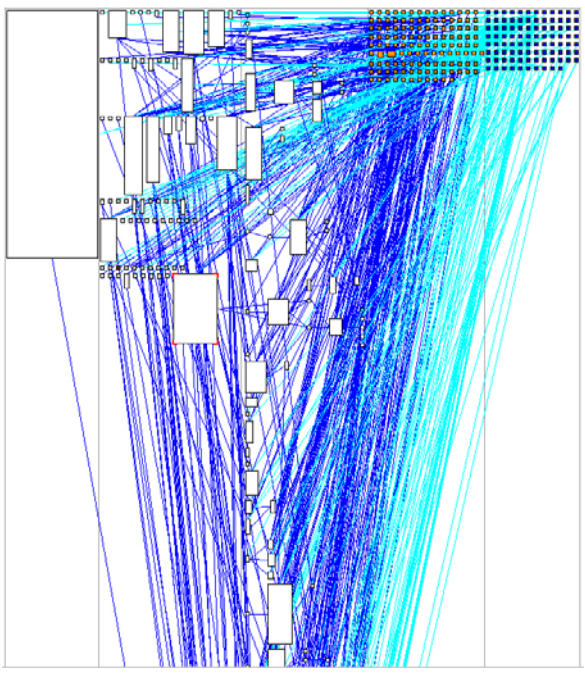

Abbildung 8: überladene Blaupause

Eine weitere Schwäche des „normalen“ Class Blueprints ist das Fehlen einer Darstellung von Abhängigkeiten zwischen Klassen. Besitzt eine Klasse Designschwächen, die sich mit den Heuristiken von [5] identifizieren lassen, die sich aus der Zusammenarbeit mit anderen Klassen ergeben (z. B. Shotgun Surgery siehe [5]), so kann dies nicht dargestellt werden. Allerdings bietet die MOOSE-Plattform an, mittels sogenannter *Easels* (englisch für Staffelei) eigene Ansichten auf Klassen zu erzeugen. Mit diesen *Easels* ist es möglich, Abhängigkeiten in abgewandelten Class Blueprints zu visualisieren.

Referenzen

[1] Wiedeking, Michael *Messlattenzaun*, KaffeeKlatsch, 03/2009

[2] Haug, Thomas *Messgenau*, KaffeeKlatsch, 05/2009

[3] Haug, Thomas *Supergenau*, KaffeeKlatsch, 06/2009

[4] Haug, Thomas *Kennzahlenbasierte Beseitigung von Code Smells*, Objektspektrum, Ausgabe 4, 2010

[5] Lanza, Michele; Marinescu, Radu *Object-Oriented Metrics in Practice* , Springer Verlag, 2006

[6] LOOSE Research Group iPlasma *An Integrated Platform for Quality Assessment of Object-Oriented Design*, http://loose.upt.ro/iplasma/index.html

[7] intooitus *InFushion*, http://www.intooitus.com/inFusion.html

[8] moosetechnology *Moose*, http://www.moosetechnology.org

Kurzbiographie

Thomas Haug (thomas.haug@mathema.de) ist als Senior-Consultant und Trainer für die MATHEMA Software GmbH tätig. Seine Themenschwerpunkte umfassen die Java Standard und Enterprise Edition (Java SE und Java EE) sowie das .NET-Umfeld, insbesondere im Hinblick auf Enterprise-Anwendungen. Neben seiner Projekttätigkeit hält er Technologietrainings für MATHEMA und berät verschiedene Projekte hinsichtlich des optimalen Einsatzes von .NET- oder Java-Technologien.

Copyright © 2011 Bookware 1865-682x/11/01/004 Von diesem KaffeeKlatsch-Artikel dürfen nur dann gedruckte oder digitale Kopien im Ganzen oder in Teilen gemacht werden, wenn deren Nutzung ausschließlich privaten oder schulischen Zwecken dient. Des Weiteren dürfen jene nur dann für nicht-kommerzielle Zwecke kopiert, verteilt oder vertrieben werden, wenn diese Notiz und die vollständigen Artikelangaben der ersten Seite (Ausgabe, Autor, Titel, Untertitel) erhalten bleiben. Jede andere Art der Vervielfältigung – insbesondere die Publikation auf Servern und die Verteilung über Listen – erfordert eine spezielle Genehmigung und ist möglicherweise mit Gebühren verbunden.