mehr zum thema:
http://loose.upt.ro/iplasma/index.html

KENNZAHLBASIERTE BESEITIGUNG VON „CODE SMELLS”: REFAKTORISIERUNGSPOTENTIALE MIT OBJEKTORIENTIERTEN METRIKEN AUFDECKEN

Refaktorisierungspotentiale in großen bestehenden Systemen aufzudecken, sollte nicht ad hoc geschehen: Die Gefahr, die Klassen falsch zu bewerten und zu refaktorisieren, ist zu groß. Dieses Risiko vermeidet die systematische Suche nach Refaktorierungskandidaten. Softwaremetriken, die Kennzahlen über das implementierte System liefern, können helfen, Kandidaten aufzudecken. Im praktischen Einsatz zeigt sich allerdings, dass nur eine Kombination von Metriken den Architekten und den Entwickler auf Designschwächen stoßen lässt.

▶ der autor

Thomas Haug
(E-Mail: thomas.haug@mathema.de)
ist als Senior Consultant, Architekt und Trainer für die MATHEMA Software GmbH tätig. Seine Themenschwerpunkte sind Java EE sowie das .NET-Umfeld im Hinblick auf Enterprise-Anwendungen.

Refaktorisierungen durchzuführen, ist eine keineswegs triviale Tätigkeit, die man nicht unterschätzen sollte, wenn man einmal von einfachen Maßnahmen wie dem Umbenennen von Methoden, Klassen und Paketen absieht. Eine Schwierigkeit besteht darin, die richtigen Stellen für Refaktorisierungen zu finden. Kent Beck hat hierfür den Begriff der *Code Smells* geprägt (vgl.[Fow00]): Ein Programmfragment verströmt einen unangenehmen Geruch, wenn es allgemein anerkannten Qualitätsansprüchen nicht genügt und somit ein tiefer liegendes Designproblem offenbart. Designprobleme können lokal beschränkt sein und lassen sich dann oftmals auch lokal beheben, wie zum Beispiel durch das Extrahieren von Methoden.

Die Auslöser dieser Art der lokal beschränkten Gerüche sollten im Rahmen eines kontinuierlichen Refaktorisierens des Systems beseitigt werden. Dies kann durch eine geeignete Vorgehensweise, zum Beispiel eine testgetriebene Entwicklung, unterstützt werden. Sind die Designprobleme allerdings tiefgreifender, so finden sich im untersuchten System häufig Anti-Patterns, d. h. Problemlösungen, die negative Konsequenzen nach sich ziehen (vgl. [Bro04]). Diese Code-Gerüche sind im Allgemeinen nicht mehr auf eine Klasse beschränkt, sondern ufern auf einen Verbund von Klassen beziehungsweise auf das gesamte System aus. In [Fow00] werden gängige Code-Gerüche beschrieben und deren Korrektur mittels geeigneter Refaktorisierungen erläutert. Doch wie findet man diese?

Gängige Praxis in Projekten ist es, durch Erfahrung, Gespür und mehr oder minder konsequent durchgeführte Code-Reviews die unrühmlichen *Code Smells* zu entdecken und diese mittels angebrachter Refaktorisierungen zu beseitigen. Sobald ein Projekt eine gewisse Größe erreicht, wird es aber immer unwahrscheinlicher, diese Gerüche zu finden, da die schiere Menge an Code uns den Wald vor lauter Bäumen nicht mehr sehen lässt.

Aber wie könnte man nun Refaktorisierungspotentiale in großen Systemen automatisiert bzw. zumindest halb-automatisiert aufdecken? Dies möchte ich im Folgenden anhand des Open-Source-Projekts „Spring” (Version 2.5.6) demonstrieren (vgl. [Spr]). Mit über 2.500 Klassen ist Spring als Framework umfangreich genug, um als großes Projekt zu gelten, in dem Refaktorisierungen mittels Erfahrung, Gespür und Code-Reviews nicht zwangsläufig entdeckt werden. Nichtsdestotrotz bedarf jede Software einer kontinuierlichen Pflege, sonst droht die berüchtigte Entropie in Softwaresystemen.

Metriken als Indikatoren

Softwaremetriken sind eine Möglichkeit, um die Systemqualität zu beurteilen und daraus resultierende Refaktorisierungen abzuleiten. Zur Bewertung objektorientierter Software gibt es eine Flut unterschiedlicher Metriken. Spinellis spricht von ca. 375 objektorientierten Metriken, die folgende Aspekte bewerten (vgl. auch [Spi06]):

- Größe und Komplexität
- Einsatz von Vererbung
- Grad der Kopplung zwischen Klassen

Bekannte Vertreter dieser Messgrößen sind das Abzählen von Codezeilen, die Bewertung der möglichen Ausführungspfade in einem Softwareartefakt (z. B. durch die Berechnung der zyklomatischen Komplexität nach McCabe) und die Verwendung bekannter objektorientierter Metriken, wie die *Suite von Chidamber und Kemerer (CK-Suite)*, die sechs Metriken beinhaltet (vgl. [Chi94]).

Der Versuch, auch nur die wichtigsten dieser Metriken zu erläutern, würde den Rahmen des Artikels sprengen. Aus diesem Grund sind in **Kasten 1** lediglich die in diesem Artikel exemplarisch verwendeten Metriken beschrieben. Zum Auffinden von potentiellen Refaktorisierungskandidaten könnte man die Metriken NOM, WMC und CBO heranziehen, denn mit ihnen kann man die Größe und Komplexität sowie den Kopplungsgrad der Software in Abhängigkeit zueinander bewerten. Zur Bestimmung der Metriken in Messgrößen verwenden wir das Werkzeug IPlasma (vgl. [LOO]). Dieses bietet zum einen die Möglichkeit, die gewünschten Metriken zu erheben, und zum anderen kann man damit Metriken kombinieren, was im Folgenden genutzt wird.

```
WMC > 100
CBO >   5
NOM>  40
```

Die gewählten Schwellenwerte entsprechen hierbei den Werten, die in einer Studie der NASA gefunden wurden (vgl. [Spi06]). Wenn eine Klasse über einem dieser Grenzwerte liegt, ist sie ein Refaktorisierungs-

Logical Lines Of Code (LLOC)
LLOC ist ein Größen- und Komplexitätsmaß, das die Anzahl ausführbarer Codeinstruktionen in einem Softwareartefakt berechnet. Diese Größe kann ein Indiz für Fehleranfälligkeit sein, denn je mehr Code vorliegt, desto höher ist die Wahrscheinlichkeit für eingeschlichene Fehler. Interessanterweise berichtet Kan in [Kan03], dass die Analyse von großen Softwareprojekten in C und Ada aus den 90er Jahren einen kurvenförmigen Zusammenhang zwischen LLOC und Fehlern gezeigt hat. Dieser Zusammenhang besagt, dass sehr kleine Module ähnlich fehleranfällig sind wie große und dass es ein Optimum gibt, das auch von der eingesetzten Programmiersprache abhängt. Leider sind mir keine vergleichbaren Analysen für die Programmiersprachen Java und C# bekannt. Hierbei wäre insbesondere ein Abgleich mit der *Clean-Code*-Bewegung lohnend, die kurze prägnante Methoden und Klassen propagiert (vgl. [Mar09]).

Number Of Methods (NOM)
NOM ein Größenmaß, das die Anzahl der Methoden innerhalb einer Klasse berechnet. Wie viele Methoden sollte eine wohldefinierte Klasse besitzen? Hierbei verwenden Metrikwerkzeuge unterschiedlichste Schwellwerte. Zum Beispiel haben Lanza und Marinescu einen Durchschnitt von sieben Methoden pro Klasse in 45 analysierten Java-Projekten ermittelt. Klassen mit mehr als 10 bzw. 15 Methoden sind laut ihrer Analyse bereits groß bzw. sehr groß und sind somit auffällige Klassen im Sinne ihrer Einteilung (vgl. [Lan06]).

Zyklomatische Komplexität (CC)
CC ist ein Komplexitätsmaß nach McCabe (vgl. [McC76]. Diese Metrik berechnet die Anzahl möglicher Ausführungspfade in einem System, z. B. in einer Methode, anhand der folgenden Vorschrift:

$$CC = e - n + 2p$$

Dabei steht *e* für die Anzahl der Kanten (*edges*), *n* für die Anzahl der Knoten (*nodes*) und *p* für die der Komponenten. Für Methoden wird *p* mit dem Wert *1* belegt. Für die in Abbildung 1 skizzierten Methode „Größter-Gemeinsamer-Teiler" (*ggt*) wird die McCabesche Komplexitätszahl *2* errechnet. *n* hat den Wert *4*, da die Methode 4 Knoten besitzt: den Eintrittspunkt in die Methode, die *while*-Kontrollstruktur, den Rumpf der *while*-Schleife und die abschließende *return*-Anweisung. Zusätzlich finden sich vier Kanten, die diese Knoten miteinander verbinden. Somit errechnet sich die Komplexität wie folgt:

$$CC = 4 - 4 + 2 = 2$$

Als Faustregel gilt, dass eine Methode grundsätzlich die Komplexität *1* besitzt und Kontrollstrukturen (wie *if, while, for, case, catch*) die Komplexität um einen Zähler erhöhen, da durch diese Strukturen weitere Ausführungspfade innerhalb der Methode entstehen. Welche Kontrollstrukturen zur Berechnung der Komplexität tatsächlich verwendet werden, hängt vom Werkzeug ab und sollte somit immer bei einer Analyse von Klassen beachtet werden.

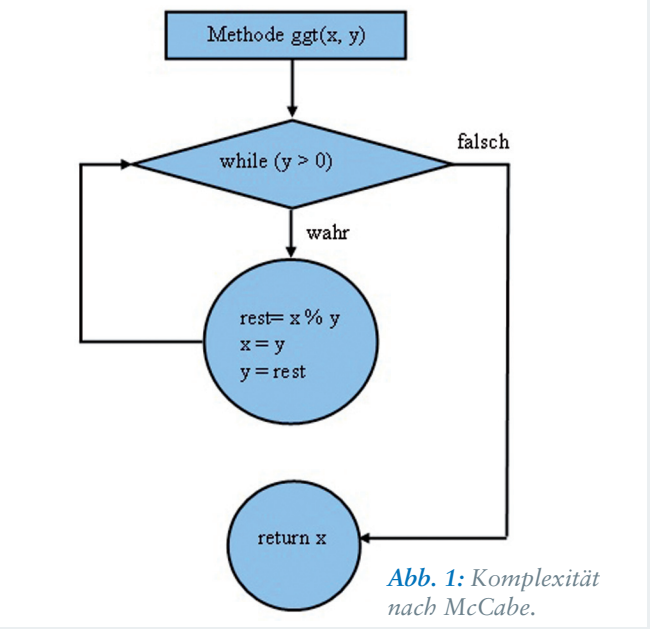

Abb. 1: *Komplexität nach McCabe.*

Weigthed Method Count (WMC)
WMC ist eine Metrik der CK-Suite und berechnet die Summe der Komplexitäten einer Klasse, indem die Komplexität jeder Methode berechnet wird. Hierbei kann zum Beispiel das CC-Maß verwendet werden:

$$WMC(Klasse) = \sum_{\text{über alle Methoden } m} CC(m)$$

Die CK-Suite-Implementierung von Spinellis berechnet den WMC-Wert, indem jede Methode grundsätzlich den Komplexitätswert 1 erhält (vgl. [Spi09]). Faktisch wird also die NOM-Metrik errechnet.

Coupling Between Objects (CBO)
CBO berechnet den Grad der Kopplung, indem die Anzahl der Klassen, die von der bewerteten Klasse verwendet werden, abgezählt werden. CBO ist ein Bestandteil der CK-Suite. Hierbei werden sowohl Methodenaufrufe, Variablenzugriffe, Argumente in Methoden, Typen in *Return*-Anweisungen als auch *Exceptions* gezählt. Hohe Werte bei dieser Metrik können ein Indiz für eine zu hohe Kopplung sein und somit Wiederverwendungsprobleme verursachen, denn alle verwenden Klassen müssen im Falle einer Wiederverwendung der taxierten Klasse auch in den neuen Kontext übertragen werden.

Response for a Class (RFC)
RFC ermittelt sowohl die Anzahl aller Methoden der Klasse als auch die Anzahl aller Methoden der Klassen, die von den Methoden der bewerteten Klasse aufgerufen werden:

$$RFC(Klasse) = NLM + NRM$$

In dieser Formel stellt *NLM (Number of Local Methods)* die Anzahl der lokalen Methoden und *NRM (Number of Remote Methods)* die Anzahl der durch die Klassenmethoden gerufen Methoden dar. Auch RFC gehört zur CK-Suite.

Kasten 1: *Die in diesem Artikel verwendeten Metriken im Überblick.*

Name	WMC	CBO	NOM	LOCC
BeanDefinitionParserDelegate	215	42	47	1288
AbstractBeanFactory	211	26	73	1315
AbstractAutowireCapableBeanFactory	204	34	50	1334
StringUtils	175	2	56	1060
JdbcTemplate	161	36	87	1279
ObjectUtils	158	0	35	799
HibernateTemplate	156	6	96	1205
DefaultListableBeanFactory	142	17	33	667
MockHttpServletRequest	139	6	103	798
AspectJAdviceParameterNameDiscoverer	133	3	25	708
AbstractPlatformTransactionManager	130	15	49	1198
DispatcherServlet	130	29	36	1139
AbstractBeanDefinition	129	13	77	946
AbstractApplicationContext	126	24	76	1087
MBeanExporter	119	28	42	986
DispatcherPortlet	115	25	30	1035
BeanWrapperImpl	114	19	35	814
ClassUtils	112	2	48	934
JtaTransactionManager	111	19	53	1066
JmsTemplate	108	10	73	951

Tabelle 1: Spring 2.5.6 auffällige Klassen.

kandidat. Leider wird man feststellen, dass im Spring-Framework das gewählte Kriterium 267 Klassen aufdeckt und somit 10 % des gesamten Frameworks zu analysieren wären. In Tabelle 1 sind 20 der 267 gefunden Klassen dargestellt, sortiert nach der WMC-Metrik.

Wie gut zu erkennen ist, haben alle eine sehr hohe Komplexität, viele Methoden und sind umfangreiche Klassen im Sinne der Codezeilen (LOCC). Der Kopplungsgrad (CBO) variiert allerdings stark – von sehr hohen bis hin zu niedrigen Werten. Wie lassen sich diese Zahlen interpretieren? Welche Klassen benötigen am dringendsten eine Umgestaltung?

Man wird sehr schnell feststellen, dass die singuläre Auswertung einer einzelnen Softwaremetrik nicht aussagekräftig genug ist. Zwar kann die Messung eines sehr hohen WMC-Werts in einer Klasse darauf hindeuten, dass diese zu viel Verantwortung wahrnimmt und dass man mittels geeigneter Refaktorisierungen den WMC-Wert der Klasse verbessern könnte. Aber wurde der gemessene Wert der Metrik im richtigen Kontext gesehen? Und welche Klassen müssten in diesem Zusammenhang noch bewertet werden?

In [Spi06] berichtet der Autor, dass bereits 1999 in einer Veröffentlichung der NASA die Kombination von Metriken zur Identifikation von Problemstellen in Systemen beschrieben und eingesetzt wurde. Treffen mindestens zwei der folgenden Bedingungen bei einer Klasse zu, sollte man diese einer genaueren Untersuchung unterziehen:

```
WMC > 100
CBO >   5
RFC  > 100
NOM >  40
RFC > 5 * NOM
```

Zur Identifikation von problematischen Klassen werden also die Komplexität (WMC), die Kopplung (CBO, RFC) und die Größe (NOM) herangezogen. Die Vererbung wird dabei außer Acht gelassen.

Da man mit IPlasma die Metrik *Response For a Class (RFC)* nicht berechnen kann, könnte man die oben genannten Metriken folgendermaßen kombinieren: Liegen mindestens zwei der gemessenen Werte für eine Klasse über den oben gezeigten Grenzwerten, so ist diese Klasse genauer zu untersuchen. Hiermit erhält man immerhin noch 26 potentielle Kandidaten (siehe Tabelle 2). Das entspricht ca. 1 % des Frameworks – ein Wert, der bei der Größe des Spring-Frameworks in Anbetracht des daraus resultierenden Analyseaufwands vertretbar erscheint.

Leider liefert die von der NASA gefundene Bewertungsrichtlinie keine Indizien dafür, welche *Code Smells* aufgedeckt wurden und mit welcher Refaktorisierungsmaßnahme eine Bereinigung durchgeführt werden kann. Das verlangt dann wieder Gespür und Erfahrung des Entwicklers und Architekten. Insbesondere muss bedacht werden, dass Klassen mit einem hohen CBO-Wert einen sehr hohen Kopplungsgrad aufweisen – eine Refaktorisierung wird in diesem Fall sicherlich weitere Klassen betreffen, die noch nicht in der Liste enthalten sind.

Metrikbasierte Code-Smell-Analyse

Weitreichende Unterstützung zur Auffindung von Refaktorisierungpotentialen erhält man durch den Ansatz von Lanza und Marinescu. In ihrem Buch „Software Metrics in Practice" (vgl. [Lan06]) beschreiben sie unter anderem, wie man durch eine geeignete Kombination von bekannten und eigenen Metriken Code-Gerüche aufdecken kann. Mittels boolescher Ausdrücke kombinieren Lanza und Marinescu die Metriken und formulieren Strategien zur Identifikation von Code-Gerüchen. In Abbildung 2 ist eine Kombination von Metriken schematisch dargestellt.

Dieses Beispiel deutet darauf hin, dass ein potentieller Defekt in der entsprechenden Klasse vorliegt, wenn entweder Metrik 1 über und Metrik 2 unter dem spezifischen Schwellwert liegen oder Metrik 3 darüber liegt.

Lanza und Marinescu unterscheiden drei Kategorien von so genannten *Disharmonies* (Designschwächen):

- *Identity:* Identitätsschwächen
- *Collaboration:* Kollaborationsprobleme
- *Classification:* Vererbungsschwächen

Identitätsschwächen zeigen Klassen und Methoden, deren Designschwächen sich zumeist isoliert betrachten lassen. Beispiele sind (vgl. auch [Fow00]):

- *Feature Envy:* Methoden der Klasse sind mehr an Daten anderer Klassen interessiert als an den eigenen.

Name	WMC	CBO	NOM	LOCC
BeanDefinitionParserDelegate	215	42	47	1288
AbstractBeanFactory	211	26	73	1315
AbstractAutowireCapableBeanFactory	204	34	50	1334
StringUtils	175	2	56	1060
JdbcTemplate	161	36	87	1279
HibernateTemplate	156	6	96	1205
DefaultListableBeanFactory	142	17	33	667
MockHttpServletRequest	139	6	103	798
AbstractPlatformTransactionManager	130	15	49	1198
DispatcherServlet	130	29	36	1139
AbstractBeanDefinition	129	13	77	946
AbstractApplicationContext	126	24	76	1087
MBeanExporter	119	28	42	986
DispatcherPortlet	115	25	30	1035
BeanWrapperImpl	114	19	35	814
ClassUtils	112	2	48	934
JtaTransactionManager	111	19	53	1066
JmsTemplate	108	10	73	951
LocalSessionFactoryBean	104	10	39	915
MimeMessageHelper	101	4	67	985
SessionFactoryUtils	101	20	23	705
CallMetaDataContext	101	11	27	540
AdvisedSupport	77	15	43	531
RequestContext	71	16	44	668
TopLinkTemplate	60	8	51	431
JdoTemplate	51	7	44	530

Tabelle 2: Kombination von Metriken analog zu Spinellis.

- *Data Class:* Die Klasse bündelt Daten, besitzt aber kein Verhalten.
- *God Class:* Die Klasse interagiert mit vielen anderen Klassen und besitzt eine übermäßige Komplexität.
- *Brain Method:* Eine Methode der betroffenen Klasse ist sehr komplex und bündelt die wesentliche Funktionalität der Klasse.

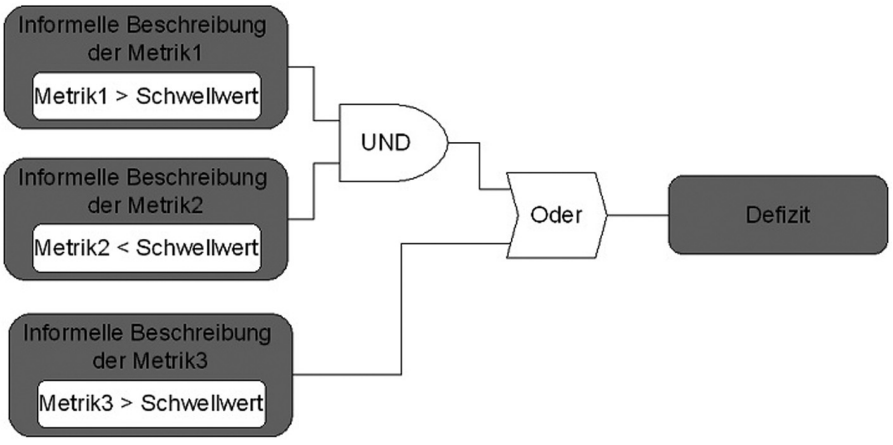

Abb. 2: Kombination von Metriken.

- *Brain Class:* Diese Klassen bündeln sehr viel Funktionalität in Form von mehreren *Brain Methods*. Im Gegensatz zu den *God Classes* interagieren sie aber nicht mit vielen anderen Klassen.
- *Significant Duplication (Duplizierter Code):* Diese Klassen weisen einen hohen Grad an dupliziertem Code auf.

Kollaborationsprobleme charakterisieren im Allgemeinen Probleme in der Zusammenarbeit von verschiedenen Klassen. Diese Art der Probleme lassen sich nur durch eine Bewertung und anschließende Refaktorisierung des Klassenverbunds beseitigen. Beispiele für diese Kategorie sind (vgl. [Fow00]):

- *Shotgun Surgery:* Diese Klasse wird von sehr vielen anderen Klassen genutzt.
- *Intensive Coupling:* Die Klasse interagiert intensiv mit wenigen Klassen.
- *Dispersed Coupling:* Die Klasse interagiert mit vielen unterschiedlichen Klassen.

Vererbungsschwächen versuchen Designprobleme in Vererbungshierarchien aufzudecken. Hierbei kombinieren Lanza und Marinescu Metriken, um die folgenden Code-Gerüche aufzudecken:

- *Refused Parent Bequest:* Die abgeleitete Klasse nutzt die geerbten Methoden nur unzureichend oder gar nicht.
- *Tradition Breaker:* Die abgeleitete Klasse nutzt bzw. überschreibt kaum die Methoden der Superklassen; stattdessen wird die Schnittstelle der Superklasse wesentlich erweitert, obwohl diese bereits viel Funktionalität mitbringt.

Die in den drei Kategorien beschriebenen Designprobleme stehen üblicherweise nicht „frei im Raum", sondern bedingen sich häufig gegenseitig. Abbildung 3 veranschaulicht die von Lanza und Marinescu aufgezeigten Zusammenhänge. Im Folgenden möchte ich anhand der Designschwäche *God Class* zeigen, an welchen Stellen sich dieser Geruch am Spring-Framework identifizieren lässt.

God Class

Eine *God Class* bündelt die wesentliche Funktionalität des Systems bzw. eines Subsystems. Dabei bedient sie sich üblicherweise der Daten anderer Klassen. Das

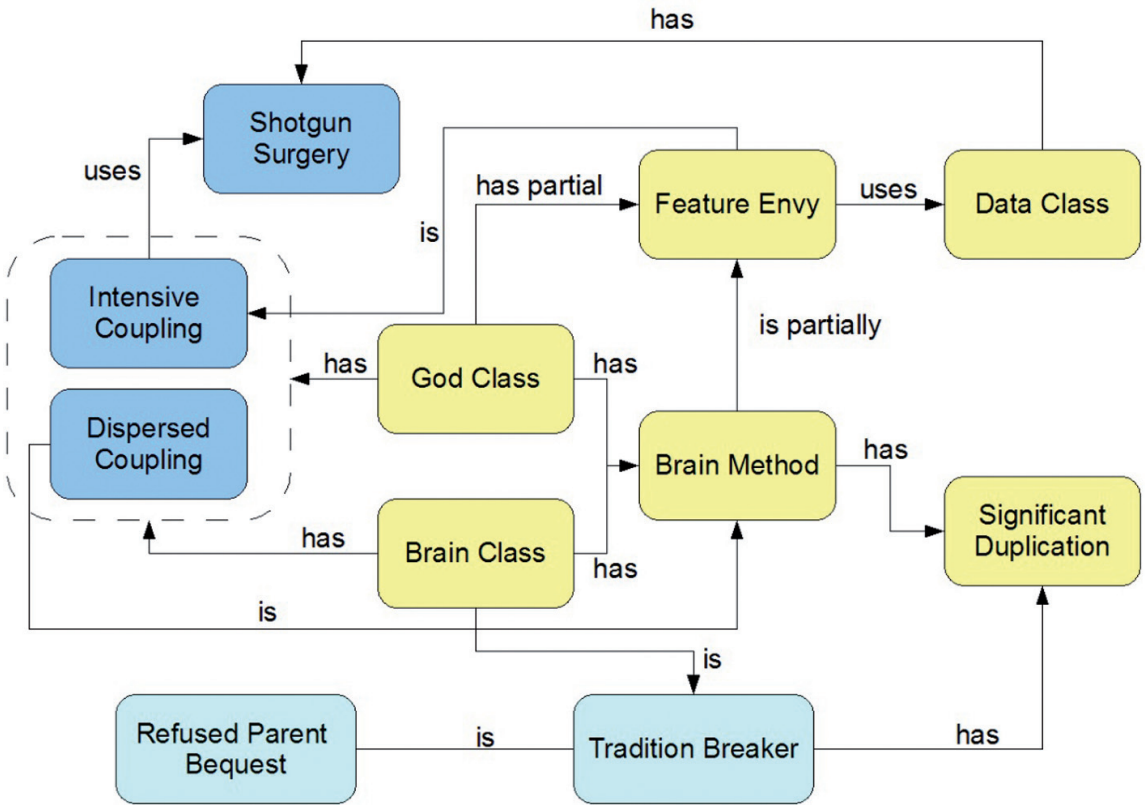

Abb. 3: Disharmonies nach Lanza und Marinescu.

bedeutet, dass die Komplexität einer solchen Klasse oft sehr hoch, doch die Kohäsion oftmals gering ist. Die Kohäsion einer Klasse ist ein Maß für den Zusammenhalt dieses Artefakts. Klassen, die sehr viel Verantwortung tragen, haben üblicherweise einen schlechteren Zusammenhalt, denn sie könnten eigentlich in mehrere kleinere, in sich abgeschlossene Klassen zerlegt werden. Somit ist also die Kohäsion bei Klassen mit zu vielen Verantwortlichkeiten niedrig. Lanza und Marinescu schlagen zur Identifikation einer solchen Klasse die in Abbildung 4 gezeigte Strategie vor. Zur Identifikation benötigt man drei Metriken:

- *Access To Foreign Data (ATFD):* Mit Hilfe dieser Metrik wird der Zugriff auf Attribute fremder Klassen ermittelt, indem der Zugriff auf Felder anderer Klassen gezählt wird. Liegt der Wert oberhalb von 5, so muss im vorgestellten Beispiel das erste Indiz erfüllt sein.
- *Weighted Method Count (WMC):* Diese Metrik wurde bereits im oberen Abschnitt erläutert. Als zu Grunde liegendes Komplexitätsmaß wird die zyklomatische Komplexität verwendet. Lanza und Marinescu zeigen in ihrem Buch, dass Klassen mit einem Wert von über 47 als sehr komplex eingestuft werden sollten. Das ist ein empirisch ermittelter Wert: Die Autoren haben 45 Java-Projekte analysiert, darunter Eclipse und ArgoUML, und Werte für niedrige, durchschnittliche, hohe und sehr hohe Komplexität bestimmt.
- *Tight Class Cohersion (TCC):* TCC ist ein Maß für die Kohäsion einer Klasse.

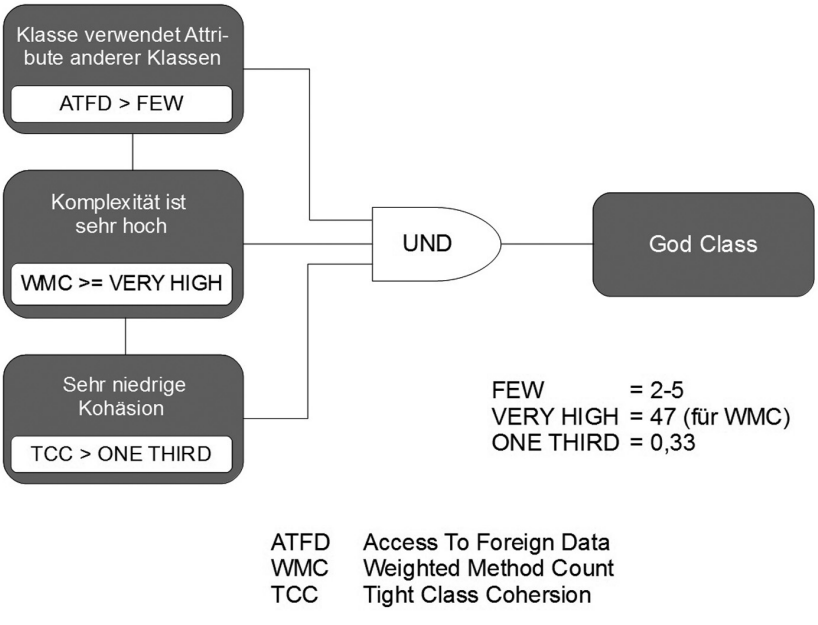

Abb. 4: God Class Disharmony (nach [Lan06]).

Name	ATFD	WMC	TCC	CBO	NOM	LOCC
BeanDefinitionParserDelegate	53	215	0,13	42	47	1288
AbstractAutowireCapableBeanFactory	18	204	0,03	34	50	1334
JdbcTemplate	7	161	0,03	36	87	1279
AbstractPlatformTransactionManager	9	130	0,09	15	49	1198
AbstractBeanDefinition	9	129	0,06	13	77	946
MBeanExporter	8	119	0,08	28	42	986
BeanWrapperImpl	11	114	0,17	19	35	814
LocalSessionFactoryBean	14	104	0,06	10	39	915

Tabelle 3: „God-Class"-Strategie, kombiniert mit der angepassten NASA-Empfehlung.

Ein hoher Kohäsionsgrad ist wünschenswert und wird folgendermaßen ermittelt:

$$TCC\ \ Klasse = \frac{Anzahl\ verbundener\ Methoden}{Gesamtanzahl\ Methoden}$$

- Zwei Methoden gelten als verbunden, wenn sie auf dasselbe Attribut zugreifen. Ein empirisch gewonnener Wert besagt, dass ein Wert von 0,5 für eine Klasse nicht unterschritten werden sollte, denn in diesem Fall würde jede zweite Methode auf einem anderen Attribut arbeiten. Lanza und Marinescu sind weniger kritisch und begnügen sich mit einem Wert von 0,33 – d. h. jede dritte Methode arbeitet auf einem anderen Attribut.

Lässt man diese *Code-Smell*-Strategie auf Spring los, so werden 28 Kandidaten gefunden, d. h. wiederum rund 1 % des gesamten Frameworks.

Weitere Einschränkungen

Wenn man nun die Suche nach Refaktorisierungskandidaten noch weiter einschränken will, könnte man die an die NASA angelehnte Empfehlung mit der *God-Class*-Strategie kombinieren. Dies lässt sich leicht mit IPlasma erreichen, indem beide Filterkriterien mit einem logischen „Und" verknüpft werden: So erhält man im Ergebnis acht Klassen (siehe Tabelle 3). Diese acht Klassen stellen einen guten Ausgangspunkt für eine tiefer gehende Analyse dar. Pickt man sich zum Beispiel die Klasse BeanDefinitionParserDelegate heraus, so wird IPlasma mitteilen, dass diese Klasse neben dem *God-Class*-Geruch noch die folgenden Gerüche mitbringt:

- *Feature Envy* bei den Methoden initDefaults(), getBeanDefinitionDefaults(), parseBeanDefinitionAttributes(), parseMapElement(), parsePropsElement()
- *Brain Methods* bei den Methoden: parseBeanDefinitionElement(), parseBeanDefinitionAttributes(), parseMapElement(), parsePropertySubElement()
- *Intensive Coupling* bei der Methode: parseBeanDefinitionAttributes()

Wie leicht zu erkennen ist, bietet diese Klasse alle in Abbildung 2 charakterisierten Designschwächen, die eine Klasse aufweisen kann, die das *God-Class*-Symptom hat. Wenn man noch weiter bohrt, sieht man, dass die ausgewählte Klasse Zugriffe auf Klassen tätigt, denen der *Data-Class*-Geruch (wie bei der Klasse BeanDefinitionDefaults) anhängt.

Wenn man mit der Refaktorisierung der Klasse beginnen will, muss man festlegen, mit welcher Methode zu starten ist. Aufschluss können hierbei weitere Metriken liefern, die man mittels IPlasma berechnen kann (siehe Tabelle 4).

Neben den *Code Smells Brain Method* und *Intensive Coupling* lässt man sich die bereits bekannten Metriken ATFD, LOC und die zyklisch Komplexität nach McCabe (CYCLO) anzeigen. Zusätzlich werden mit FANOUT und FANOUTCLASS Indizien geliefert, mit wie vielen Klassen die Methode interagiert. FANOUTCLASS berechnet, mit wie vielen unterschiedlichen Klassen die Methode interagiert, während FANOUT den Grad misst, wie oft sie mit anderen Klassen interagiert. Hier sticht insbesondere die Methode parseBeanDefinitionAttributes hervor, der man sich zuerst widmen sollte. Schaut man sich den Quellcode der entsprechenden Methode an, ist zu erkennen, dass die Methode die XML-Konfiguration einer Spring-Komponente (Bean) einliest und mittels der eingelesenen Information ein Objekt der Klasse AbstractBeanDefinition parametrisiert.

Der erste Ansatzpunkt für eine Refaktorisierung wäre es, den *Feature-Envy*-Geruch so zu behandeln, dass der Zugriff auf *Data Classes* reduziert wird. Allerdings wird man feststellen, dass die genannte Methode im Wesentlichen auf Attribute der

Name	ATFD	Brain Method	CYCLO	FANOUTCLASS	FANOUT	LOC	Intensive Coupling
parseBeanDefinitionAttributes	22	true	22	8	29	92	true
initDefaults	8	false	4	3	13	20	false
getBeanDefinitionDefaults	6	false	2	4	12	11	false
parseBeanDefinitionElement	5	false	3	9	22	47	false
parseMapElement	4	true	25	8	24	105	false
parseConstructorArgElement	4	false	5	9	16	45	false
parsePropsElement	3	false	2	6	15	22	false

Tabelle 4: Details zur Klasse „BeanDefinitionParserDelegate".

Klasse org.w3c.dom.Element zugreift – in diesem Fall wird man kaum die Element-Klasse erweitern können. Allerdings fällt weiterhin die hohe Komplexität der Methode auf. Abhilfe schafft an dieser Stelle die Refaktorisierung *Methode extrahieren* nach Fowler. Zum Beispiel könnte man folgendes Codefragment aus der Methode in eine kleine private Methode isLazyInitialized extrahieren:

```
String lazyInit = ele.getAttribute
          (LAZY_INIT_ATTRIBUTE);
if (DEFAULT_VALUE.equals(lazyInit) &&
          bd.isSingleton()) {
  // Just apply default to singletons, as lazy-init has no
  //  meaning for prototypes.
  lazyInit = this.defaults.getLazyInit();
}
bd.setLazyInit(TRUE_VALUE.equals(lazyInit));
```

Wenn man diese Refaktorisierung konsequent auf die Methode parseBeanDefinitionAttributes anwendet, reduziert sich die Komplexität drastisch, denn sie wird in die neu entstandenen Methoden verschoben. Diese kleinen Methoden lassen sich wesentlich besser testen bzw. einem Review unterziehen als eine Methode mit der Komplexität 22, denn für diese kleinen Methoden muss man im Verhältnis zur Komplexität 22 der ursprünglichen Methode weniger Tests schreiben. Andererseits wird aber auch der FANOUT der ursprünglichen Methode stark reduziert, da sich die neu erzeugten Methoden um den Zugriff auf das Element-Objekt kümmern. Als positiver Nebeneffekt wird die Metrik *Number of Accessed Variables (NOAV)* reduziert, die den Zugriff auf Parameter, aber auch Member-Variablen und Konstanten misst und Bestandteil der *Brain-Method*-Erkennungsstrategie ist. Durch einen einfachen Eingriff kann man also verschiedenste *Code Smells* reduzieren und bisweilen sogar eliminieren.

Fazit

Metriken geben uns die Möglichkeit, potentielle Refaktorisierungskandidaten aufzudecken. Hierbei erscheint es sinnvoll, nicht nur Metriken singulär heranzuziehen, sondern durch die Kombination von Metriken die Menge der Kandidaten einzuschränken. Durch eine geschickte Wahl der Metriken lassen sich insbesondere mit den Strategien von Lanza und Marinescu bestimmte Code-Gerüche aufdecken. Zwar lassen sich die Metriken und deren Kombinationen mittels IPlasma leicht berechnen – die richtigen Schlüsse zu ziehen, verlangt aber immer Fingerspitzengefühl und Erfahrung des Architekten und Entwicklers, wenn nicht-triviale Refaktorisierungen erfolgreich durchgeführt werden sollen.

Meine Erfahrungen aus Softwareprojekten zeigen, dass sich die hier gezeigten Metriken und Vorgehensweisen auch auf Projekte mit Geschäftslogik übertragen lassen, und nicht nur auf Framework-Code anwendbar sind.

Die in diesem Artikel aufgezeigten Möglichkeiten, mittels Metriken Refaktorisierungspotientale zu identifizieren, stellen lediglich eine Ergänzung zu den übrigen Analysemethoden für bestehenden Code dar. Sie ersetzen meiner Meinung nach nicht den Einsatz von Werkzeugen, wie z. B. „JDepend" (vgl. [Cla]), mit denen man die Paketstrukturierung analysieren kann, um zyklische Abhängigkeiten oder Zugriffe auf Pakete aufzudecken, die den Architekturvorgaben des Projekts widersprechen. ■

Literatur & Links

[Bro04] W. Brown, Anti Patterns – Entwurfsfehler erkennen und vermeiden, Mitp-Verlag, 2004

[Chi94] S. Chidamber, C. Kemerer, A Metrics Suite for Object Oriented Design, in: IEEE Transactions on Software Engineering, Vol. 20, 1994

[Cla] Clarkware Consulting, Inc., JDepend, siehe: http://clarkware.com/software/JDepend.html

[Fow00] M. Fowler, Refactoring – Wie Sie das Design vorhandener Software verbessern, Addison-Wesley, 2000

[Kan03] S.H. Kan, Metrics and Models in Software Quality Engineering, 2nd Ed., Addison-Wesley, 2003

[Lan06] M. Lanza, R. Marinescu, Object Oriented Metrics in Practice, Springer, 2006

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

[Mar09] R.C. Martin, Clean Code – A Handbook of Agile Software Craftsmanship, Prentice Hall, 2009

[McC76] T.J. McCabe, A Complexity Measure, in: IEEE Transactions on Software Engineering, Vol. 2. 1976

[Spi06] D. Spinellis, Code Quality – The Open Source Perspective, Addison-Wesley, 2006

[Spi09] D. Spinellis, ckjm – Chidamber and Kemerer Java Metrics, 2009, siehe: http://www. spinellis.gr/sw/ckjm/

[Spr] SpringSource.org, siehe: www.springsource. org