Überlegungen über die Beurteilung der Qualität von Anwendungsarchitekturen

White Paper

Siegfried Nolte

2005

Inhaltsverzeichnis

1 Einleitung und Motivation

Wenn ein Handwerker „vom Gerüst fällt“, dann muss, so bedauerlich das auch ist, ein anderer mit vergleichbaren Kenntnissen die Arbeit fortsetzen können. Wenn Künstler wie Michelangelo vom Gerüst gefallen wäre, dann hätte es die Sixtinische Kapelle in dieser Form nicht gegeben.

Die Fragen, auf die ich in diesem Papier eingehen möchte, sind:

- Wie lässt sich die Qualität von Anwendungsarchitekturen beurteilen?
- Wie lässt sich auf der Basis von qualitativ hochwertigen Architekturen qualitativ bessere Anwendungsentwicklung betreiben ?

Eine ***Anwendungsarchitektur*** ist das Ergebnis der Phase „Entwurf des Anwendungssystems“ oder kurz Entwurf[1]. Das Dokument der Entwurfsphase sei das DV-Konzept, also eine Beschreibung, wie sich ein gefordertes und fachlich analysiertes System mit DV-Mitteln realisieren und in eine vorgegebene Systemumgebung (Systemarchitektur) integrieren lässt.

Das DV-Konzept geht eigentlich etwas weiter, indem es die Anwendungsarchitektur und die Systemarchitektur beschreibt. Das Vorhandensein einer Systemarchitektur wird hier jedoch vorausgesetzt. D.h. die Erarbeitung einer qualitativ guten Systemarchitektur ist nicht Gegenstand der Überlegungen dieses Papiers[2].

Das Vorhandensein eines Analysedokumentes (aus der Phase „Fachliche Analyse“) sei hier ebenfalls vorausgesetzt; auch die Form und die Qualität des Analysedokumentes steht hier nicht zur Diskussion.

Die Anwendungsarchitektur ist das Ergebnis der methodischen Arbeit bei dem Entwurf des Anwendungssystems, wobei hier nicht die Methoden an sich erörtert werden sollen, sondern die Güte des Ergebnisses und Aspekte zur Beurteilung des Ergebnisses.

Im Sinne einer traditionellen Sichtweise eines ingenieurmäßigen Vorgehens beschreibt eine Architektur das (statische) Erscheinungsbild eines Gebäudes resp. Systems. DV-Systeme sollen darüber hinaus nicht nur einen statischen Aufbau haben (Komponenten, Pakete, Frameworks, Datentypen, ...) sondern auch dynamisch etwas leisten, nämlich das, was gefordert ist, und das in zuverlässiger und performanter Weise. Demzufolge ist der Begriff der ***Tragfähigkeit*** aus anderen Ingenieursdisziplinen mit dem Begriff der ***Performability*** (Performance und Reliability) der Informatik vergleichbar.

Zudem gilt, dass DV-gestützte Anwendungssysteme auch auf lange Sicht pflegbar und änderbar sein sollen. D.h. neben der Performability als Qualitätsaspekt kommt auch im Wesentlichen die Verständlichkeit und Nachvollziehbarkeit ins Spiel. Dies resultiert aus einem Sachverhalt, der für DV-Systeme typisch ist: Die

---

1 Gemäß Vorgehensmodell von HHLA-I

2 Lt. Rumbaugh et al (OMT) ist Systementwurf die globale Problemlösungsstrategie für die Implementierung eines Systems; die Systemarchitektur ist die Gesamtorganisation des Systems, also die Organisation der fachlichen Dinge und Sachverhalte (Anwendungssicht) wie auch der Systemumgebung, in der die Implementierung stattfindet (Systemsicht). Mit dem Begriff Anwendungsarchitektur möchte ich mich hier auf die Anwendungssicht beschränken.

Anforderungen ändern sich sehr kurzfristig, so dass stets und unmittelbar auf die geänderten Forderungen reagiert werden muss. D.h. neben der Qualitätsforderung der Performability, die von den meisten SW-Systemen mehr oder weniger erfüllt wird, ist auch in starkem Maße die Dokumentation gefordert, und unter dem Aspekt der Verständlichkeit und Nachvollziehbarkeit bedeutet dies nicht notwendigerweise Prosatext[3].

Im Folgenden sollen einige Aspekte beschrieben werden, die zur Bewertung einer Anwendungsarchitektur herangezogen werden können. Dabei möchte ich mich auf eine überblicksartige Beschreibung beschränken und nicht im Einzelnen in die Tiefe gehen.

Selbstverständlich hat jedes Gebäude - so auch jedes (DV-)System - eine Beschaffenheit, ein Erscheinungsbild, eine Form, einen Aufbau, aber Architektur ist das nicht. So hat z.B. eine Strohhütte einen Aufbau und eine Struktur, von Architektur reden wir aber erst dann, wenn in modellierter und strukturierter Form auch die Tragfähigkeit beschrieben und nachvollziehbar ist. So würde z.B. niemand über eine Brücke fahren oder in ein Flugzeug steigen, die nicht nach in diesem Sinne konkreten architektonischen Maßstäben konstruiert und gebaut worden sind. Der Ansatz des ingenieurmäßigen Erstellens von Software (Software Engineering) dient dem Zweck, den Software-Entwicklungsprozess aus dem künstlerischen Bereich[4] in den handwerklichen Bereich zu übertragen. Das muss man akzeptieren – alle Beteiligte. Der damit verbundene Umdenkungsprozess fällt den meisten „künstlerisch geprägten“ Entwicklern heute noch schwer.

2 Qualitätsmerkmale von Anwendungsarchitekturen

2.1 Das Vorhandensein einer Anwendungsarchitektur

Eine wesentliche Qualitätsanforderung an Anwendungsarchitekturen ist: es gibt eine. Die Architektur eines Software-Systems existiert erst dann, wenn sie in dokumentierter Form vorliegt. „Selbstverständlich gibt es eine Architektur, sie ist nur nicht dokumentiert.“, wird in diesem Sinne als Qualitätsmerkmal nicht akzeptiert[5]. In anderen Worten, wenn diese Worte fallen, dann ist die Güte der Anwendungsarchitektur per se fragwürdig.

Die mindest geforderten Dokumente sind – in objekt- wie auch nicht-objektorientierten Systemen – die Spezifikationen des statischen Aufbaus (Struktur) und die Spezifikation des (funktionalen) Verhaltens. Wie gesagt, dies gilt gleichermaßen für objektorientierte wie auch für nicht-objektorientierte Anwendungssysteme.

---

[3] Mit dem Formulieren von flüssiger Prosa haben Ingenieure (Informatiker sehe ich hier mal als Software-Ingenieure) ohnehin ihre Probleme (vgl. DeMarcos „Der Termin“, literarisch nicht gerade herausragend. In Sachen Romane in dem Kontext Architekturen gibt es von anderen anderes, z.B. „Die Säulen der Erde“, Ken Follet). Also, gut, dass wir dafür Modelle haben.

[4] In den Anfängen der Informatik hatte Programmierung eher den Stellenwert eines künstlerisch kreativen Schaffensprozesses (vgl. Knuth, „The Art of Computer Programming“).

[5] Es würde auch niemand in ein Flugzeug einsteigen, von dem die Fluggesellschaft sagen würde: „Es gibt keine ingenieurmäßig dokumentierte, nachvollziehbare und prüfbare Konstruktion. Aber machen Sie sich mal keine Sorgen, wir haben die besten Techniker ...“.

Auch wenn man nicht in einer objektorientierten Programmierumgebung entwickelt, haben die Systeme doch einen Aufbau, eine Struktur und ein Verhalten.

Spezifikationen des Aufbaus sind (im Kontext „Architektur“[6])

- Komponentendiagramme
- Klassendiagramme
- Objektdiagramme
- Deployment-Diagramme

Spezifikationen des Verhaltens sind (im Kontext „Architektur“)

- Sequenzdiagramme (ggf. strukturierte algorithmische Spezifikation /Actiondiagramme)
- Zustandsdiagramme
- Interaktionsdiagramme (Dialoge)
- Kommunikationsdiagramme

2.2 Vollständigkeit, Einfachheit, Verständlichkeit

Bei den oben umrissenen Darstellungstypen der Architektur handelt es sich nicht um ein „kann sein“ sondern um ein „muss sein“. Die Architektur ist erst dann vollständig, wenn alle Diagramme vorliegen (oder begründet entfallen, so ist z.B. fraglich, ob man in Batch-Applikationen Interaktionsdiagramme benötigt). Wie bereits gesagt, dadurch dass wir uns in der Designphase befinden, die an sich schon einen Schritt weiter in Richtung Abstraktion ist, ist hier nicht Prosa gefragt, sondern Diagramme auf der Basis von standardisierten Modellen (und das ist UML).

Darüber hinaus gilt, die Anwendungsarchitektur ist eine Verfeinerung und Konkretisierung des Fachkonzeptes, d.h. alles was im Fachkonzept genannt ist, muss in der Anwendungsarchitektur aufgegriffen und weiter nach Design-Aspekten behandelt werden.

Auch wenn ein Aspekt nicht weiter mit der Zielsetzung der Implementierung durch eine DV-Lösung verfolgt werden soll, ist dies aufzugreifen und explizit zu verfolgen. Dies gilt gleichermaßen für die strukturellen wie auch für die funktionalen Sachverhalte[7].

Zudem gibt es unter dem Aspekt der DV-technischen Umsetzung weitere strukturelle Ergänzungen (Benutzer-Oberfläche, Datenhaltung, Client/Server-Schichten), die stets in einem direkten Bezug zu den Komponenten und Klassen stehen müssen, die aus den fachlichen Anforderungen resultieren. Hier gilt, soviel und so komplex „wie nötig“ (und nicht „wie möglich“).

---

[6] Die Modelle und Darstellungsmittel der Analysephase sollen hier nicht betrachtet werden.

[7] Hiermit ist implizit schon gesagt, dass eine Anwendungsarchitektur nichts mehr enthalten darf, als in dem Fachkonzept beschrieben ist. Wenn sich zeigt, dass es doch mehr Bedarf an fachlichen Dingen gibt, dann ist die Güte des Fachkonzeptes an der Stelle zu überprüfen – was hier allerdings nicht Gegenstand der Betrachtung sein soll.

Bei all dem muss eine Anwendungsarchitektur so verständlich sein, dass sie von einem anderen Experten, der nicht an der Entwicklung beteiligt ist, gelesen und verstanden werden kann. Wenn dies nicht der Fall ist, dann ist das Konzept eben schlechter als wenn es der Fall wäre.

2.3 Kopplung und Kohäsion

Dies sind Begriffe, die nicht ganz neu sind in der Geschichte der ingenieurmäßigen Softwareentwicklung. *Kopplung* bezeichnet das Maß der Beziehungen zwischen Komponenten wie auch der Klassen innerhalb von Komponenten; *Kohäsion* beschreibt die Bindung und den Zusammenhang der Komponenten resp. Klassen in sich. Die zentralen Gegenstände eines objektorientierten Designs sind die Komponenten und Klassen. D.h. *Kopplung* und *Kohäsion* müssen jeweils auf der Ebene der Komponenten und der Klassen betrachtet werden. Zudem sei darauf hingewiesen, dass hier nur die eigenen Klassen betrachtet werden; Klassen externer Standardbibliotheken werden in diese Überlegungen nicht einbezogen[8].

2.3.1 Kopplung

Komponenten und Klassen sind grundsätzlich miteinander gekoppelt. Wenn eine Klasse innerhalb eines Systems nicht in irgendeiner Beziehung zu einer anderen Klasse steht, dann kann man sich fragen, ob diese Klasse überhaupt Sinn macht. Aber die Kopplung zwischen Komponenten oder die Kopplung zwischen Klassen sollte so niedrig wir möglich sein.

Das bedeutet auf der Ebene der Komponenten: Alle Komponenten sollten ausschließlich über Interfaces miteinander in Beziehung stehen (vgl. Komponentendiagramme in UML2.0). Die Interfaces kapseln dabei die *public* Methoden der Klassen, die Bestandteile der Komponenten sind. Die Signaturen der Methoden der Interfaces sollten so einfach und übersichtlich wie möglich sein.

Für Klassen bedeutet das: Klassen sollten nur so wenig wie möglich *public* Methoden anbieten. Die Signaturen dieser Methoden sollten so „klein“ wie möglich sein.

Diese Sicht der Dinge hat im Prinzip zur Folge, dass über Komponentengrenzen hinaus keine Vererbung erlaubt ist. Unter Design-Gesichtspunkten macht das auch Sinn, da Vererbung, so positiv sie auch ist im Sinne der Reduzierung von Code, grundsätzlich dazu führt, dass Anwendungssysteme unübersichtlicher und schwieriger zu warten sind; Vererbungsbeziehungen implizieren stets eine starke *Kopplung*[9]. Allerdings soll dies nicht heißen, dass Vererbungsbeziehungen grundsätzlich schädlich sind, denn auf der anderen Seite, da wo Vererbung fachlich zweckmäßig ist, z.B. in fachlichen Generalisierungs-/Spezialisierungsbeziehungen, da macht es meist auch Sinn, dies in der Anwendungsarchitektur beizubehalten, doch dann möglichst in derselben Komponente also nicht über Komponentengrenzen hinaus.

---

[8] Zu diesen soll auch keine Qualitätsaussage gemacht werden.

[9] Coad/Yourdon – „OO-Design“ – definieren die Begriffe *Interaction Coupling* und *Inheritance Coupling*, und sie fordern geringes *Interaction Coupling* und hohes *Inheritance Coupling*. Allerdings gab es in den Anfängen der OO SWE noch nicht die Sicht der Komponenten.

2.3.2 Kohäsion

Auch *Kohäsion* muss auf zwei Ebenen betrachtet werden. *Kohäsion* auf der Ebene der Komponenten bedeutet, dass die betrachtete Komponente aus Klassen besteht, die untereinander in einem recht hohen Wechselwirkungsgefüge stehen. Interaktion unter Verwendung von Interfaces ist hier nicht vonnöten und auch nicht immer möglich.

Kohäsion auf der Ebene der Klassen bedeutet, dass Methoden – zumindest die privaten Methoden - der Klassen einen starken Wechselbezug zueinander haben. In der Regel ist das natürlicherweise der Fall; wenn es private Methoden gibt, die in der Klassen nicht benutzt werden, dann sind sie überflüssig. In aller Regel bedeutet dies, dass es sich um eine Methode handelt, die vielleicht fachlich einer anderen Klasse zuzuordnen ist.

Aus den Prinzipien Kopplung und Kohäsion folgt, dass z.B. alleinstehende Klassen in Komponenten oder Klassengefüge, die keinen Bezug zu anderen Klassen der Komponente haben[10], im Rahmen eines Redesigns noch einmal überarbeitet und umstrukturiert werden sollten.

Allgemein gilt: Geringe Kohäsion impliziert eine hohe Kopplung und damit eine hohe Komplexität. Hohe Komplexität bedeutet relativ hohen Pflegeaufwand. Darüber hinaus kann gesagt werden, dass stark gekoppelte und schwach kohäsive Komponenten ein geringeres Potential an Wiederverwendbarkeit in sich bergen.

2.4 Designschema der Klassen

Im Folgenden werden einige weitere Designkriterien aufgelistet[11], auf die später im Kapitel Refactoring näher eingegangen werden soll:

- kein *tramp data* – „Durchreichen“ von Attributswerten ohne weitere Aktion
- keine *public* Attribute, besser *private* mit *gettern* und *settern*
- keine *super*-Aufrufe
- keine globalen Variablen, besser mit Parameterübergabe
- keine Design-Entscheide auf Grund der Schreib/Tipp-Ersparnis
- kleine „auf einem Blick“ überschaubare Methoden
- jede Methode sollte dabei genau eine Funktion realisieren (funktionale Kohäsion)
- eine Unterklasse sollte immer aller ererbten Attribute nutzen und nicht nur ein Teil
- keine Zusicherungen auf ererbte Attribute

[10] Dies kann man in der Praxis der Programmierung recht häufig beobachten.

[11] Dies ist im Wesentlichen eine Auflistung, die an Oesterreich „OO-Software-Entwicklung angelehnt ist.

- wenn ererbte Methoden redefiniert werden, sollten sie zum Verhalten der geerbten Methoden kompatibel sein
- Client/Server-Prinzip zwischen kooperierenden Klassen verwenden (hier können ggf. schon vordefinierte Design-Patterns Verwendung finden)
- Eine Klasse sollte wissen, was sie selbst tut; sie sollte nicht unbedingt wissen, was ihre Nachbarklasse tut (mit Ausnahme der public Methoden, die von ihr benötigt werden)
- Trennung von Fachklassen und Implementierungs-spezifischen Klassen in Schichten (GUI, Datenhaltung, andere Serviceklassen[12])

2.5 Weitere Techniken zur Verbesserung der Qualität

2.5.1 Design by Contract

Design by Contract ist ein interessantes Architekturprinzip, welches von Bertrand Meyer stammt. *Design by Contract* bedeutet, dass gewährleistet sein soll, dass OO-Klassen genau das leisten, was von ihnen gefordert ist. Prinzipiell sollen sie das auch ohne diese neue Forderung; doch *Design by Contract* geht etwas weiter: jede Klasse prüft mittels *Assertions* oder *Invarianten* (lt. Meyer) explizit die Vorbedingungen und Nachbedingungen für ein ordnungsgemäßes Erfüllen der Anforderungen.

Dafür gibt es 2 Techniken:

- Zum Einen kann jede Methode – mindestens die public Methoden, die die Leistungen einer Klasse nach außen anbieten – über derartige Prüfschritte verfügen, in denen zu Beginn die gültigen Ausprägungen ihrer Argumente geprüft werden, und in denen zum Ende die korrekte Ausprägung des return-Wertes geprüft wird. Diese Technik orientiert sich an der Signatur der Methoden.
- Zum anderen kann es in Klassen sogenannte *Klasseninvarianten* geben, in denen die korrekten möglichen Ausprägungen der Attribute der Klassen festgelegt sind. Die Methoden, die auf diesen Attribute operieren, können dann nicht anders als korrekte Werte erzeugen.

Die Technik der Invarianten wird nicht von jeder Programmiersprache unterstützt[13]. Im Rahmen der Anwendungsarchitektur soll uns das unter dem Aspekt eines guten Design-Stils aber nicht stören. Die UML als Sprachmittel für die Designphase bietet für diese Zwecke die *Object Constraint Language* (OCL) an. Jede Klasse sollte im Hinblick auf die korrekten Ausprägungen ihrer Attribute über derartig formale oder semi-formale OCL-Spezifikationen verfügen.

---

[12] Von Coad/Yourdon – „OO-Design“ – werden folgende Design-Schichten definiert, die sich bis heute in den verschiedenen Tier-Sichten wiederfinden: *Human Interaction*, *Problem Domain*, *Task Management*, *Data Management*.

[13] Eiffel tut dies zufällig. Das Prinzip der Invarianten gab es davor aber auch schon bei Ada.

2.5.2 Refactoring

Refactoring (oder Refaktorisierung) ist ein Verfahren aus dem Xtreme Programming-Ansatz, in dem Anwendungsarchitekturen nicht explizit über eine Design-Phase erarbeitet werden sondern im Rahmen eines iterativen Wechselspiels zwischen Design und Entwicklung. In anderen Worten (Martin Fowler[14]), „Refactoring ist ein Prozess, ein vorhandenes Software-System zu verändern, dass das externe verhalten nicht verändert wird, der Code aber eine bessere innere Struktur erhält"; daraus resultiert eine in diesem Sinne qualitativ bessere Architektur. Die Architektur des Systems wird verbessert, nachdem der Code dazu implementiert worden ist.

Fowler definiert den Begriff des „***stinkenden Codes***"[15] für fragwürdige Design-Ansätze, die zu untersuchen und geeignet zu behandeln sind. Hierzu zählen

- Duplizierter Code

  Diegleiche Code-Stuktur an mehreren Stellen im System, z.B. dieselbe Code-Sequenz in unterschiedlichen Methoden einer Klasse, oder dieselbe Codesequenz in unterschiedlichen Unterklassen einer Vererbungshierarchie.

- Lange Methoden

  Lange Methode sind Methoden, die mehr als einen funktionalen Aspekt implementieren (vgl. Begriff der **funktionalen Kohäsion**) in Form von komplexen Verschachtelungsstrukturen mit vielen *ifs* oder *switches*, gesteuert über Kontroll-Argumente. In aller Regel sollte gelten, keine Methode länger als eine Seite, die man auf einem Blick im Editor übersehen kann.

- Große Klassen

  Große Klassen sind Klassen, die zuviel an logischen Aufgaben erfüllen sollen. Erkennbar sind sie daran, dass sie über eine große Menge von Attributen und Methoden verfügen, die logisch nicht unbedingt zusammengehören (vgl. Das Prinzip der funktionalen Abhängigkeit und Normalisierung im relationalen DB-Design). Beispiel hierfür ist eine Klasse Person, die neben den persönlichen Angaben (Name, Alter etc.) auch Angaben zum Wohnort (Straße, Hausnummer, Postleitzahl, Wohnort etc.) umfasst.

- Lange Parameterlisten

  Lange Parameterlisten entstammen der Tradition, dass alles, was eine Prozedur benötigt, per Parameterübergabe zur Verfügung gestellt werden soll. In der strukturierten Software-Entwicklung resultierte aus diese Komplexitätsproblematik bereits die Forderung nach geringer Datenstrukturkopplung[16]. In der objektorientierten Software-Entwicklung benötigen Methoden eigentlich kaum Parameter, da sie auf den privaten Attributen arbeiten können und i.d.R. nur auf diesen arbeiten sollten.

---

[14] Refactoring, Martin Fowler; Addison-Wesley

[15] smells

Wenn sie das nicht tun, d.h. wenn ihnen in Form von Parameterlisten umfangreiche Fremddaten mitgegeben werden, dann ist eben etwas faul. Schädlich ist in diesem Zusammenhang auch, wenn in Form der Parameterlisten der Methoden Objektdeklarationen vorgenommen werden.

- Divergierende Änderungen

  Divergierende Änderungen entstehen dann, wenn eine Klasse häufig auf verschiedene Weise und aus verschiedenen Gründen geändert werden muss.

- „Schrotkugeln“

  Das Prinzip „Schrotkugeln“ zeigt sich dann, wenn Änderungen in einer Sache nicht nur an einer Stelle vollzogen werden müssen, sondern wenn zudem noch daraus kleine Änderungen und Anpassungen an vielen anderen Stellen erforderlich sind. Besonders schlimm ist es dann, wenn die Änderungen Klassen in unterschiedlichen Paketen betreffen.

- „Neid“

  Ein klassischer Abstraktionsansatz in der OO-SWE ist, dass eine Klasse genau für einen bestimmten fachlichen Aspekt zuständig ist. Bei der Erfüllung der Aufgaben kann eine Klasse z.B. per Nachrichten andere Klassen zu Hilfe nehmen. Wenn sich jedoch zeigt, dass eine Klasse über Gebühr an anderen Klassen interessiert ist und diese mehr in Anspruch nimmt als seine eigenen Aufgaben zu erfüllen, dann zeigt sich darin ein gewisser „Neid“. Besonders ungünstig ist es dann, wenn eine Klassen in besonderem Maße „neidisch“ auf die Daten einer anderen Klasse ist.

- Datenklumpen

  Datenklumpen sind nicht disjunkte Attributmengen aus dem Attributbestand einer Klasse, die oft gemeinsam in den Methoden einer Klasse auftreten. Dies ist ggf. ein Indiz für eine weitere Zerlegung in eine eigene Klasse.

- Neigung zu elementaren Typen

  Hinter „Neigung zu elementaren Typen“ steckt eine gewisse Zurückhaltung, bei der Programmierung auf OO-Ansätze zurückzugreifen. So kann es z.B. zweckmäßiger sein, anstelle von Attributen in simplen Datentypen (Postleitzahl : INTEGER) einfache abstrakte Datentypen (Klassen) zu implementieren, die unter anderem Aufgaben der Plausibilitätsprüfung übernehmen (Ausprägung nur 5 stellig ganzzahlig, ggf. in einem gültigen Rahmen für Postleitzahlen der BRD).

- *Switch*-Befehle

  Switch-Anweisungsfolgen in Methoden sind oft ein Indiz für eine strukturierte SWE mit den Mitteln einer OO Sprache. Immer wenn man *switch*-Konstrukte in einer Methode findet, sollte man prüfen, ob eine Zerlegung in mehrere Methoden oder ggf. eine Vererbungsabstraktion möglich.

---

[16] Schwache Kopplung durch kleine überschaubare Datenstruktur-Schnittstellen (formale Parameterlisten).

- Parallele Vererbungshierarchien

  Parallele Vererbungshierarchien zeigen sich daran, dass man, immer wenn man eine Klasse in eine oder mehrere Unterklassen zerlegt, man auch eine andere Klasse analog spezialisieren muss. (Dies ist ein Spezialfall des Schrotkugel-Syndroms.) Oft sind die Präfixe der Klassennamen gleich oder ähnlich. Z.B. Fahrzeuge, Landfahrzeuge, Wasserfahrzeuge und Fahrgeräte, Landfahrgeräte, Wasserfahrgeräte.

- Faule Klassen

  Faule Klassen sind Klassen, die nicht genug leisten, die also kaum von anderen Klassen benötigt werden. Diese sollten eliminiert werden. Oft handelt es sich um Klassen, die im rahmen zuvor durchgeführter Refactoring-Maßnahmen hinfällig geworden sind, auch wenn sie früher einmal „ihre Dienste getan haben“.

- Spekulative Allgemeinheit

  Spekulative Allgemeinheit ergibt sich aus „spekulativen“ Anforderungen, „wir werden diese Leistungen irgendwann schon benötigen“. Mit der Motivation werden alle möglichen und unmöglichen Spezialfälle implementiert.

- Temporäre Felder

  Temporäre Felder sind Attribute von Klassen, die bei instanziierten Objekten nur manchmal, unter bestimmten Bedingungen, gesetzt werden. Temporäre Felder kommen dann häufig vor, wenn eine Klasse mehr als einem logischen Aspekt dienen soll, was dann über Kontrollattribute gesteuert wird. Das ist, wie oben schon gefordert, zu vermeiden.

- Nachrichtenketten

  Nachrichtenketten werden daran erkannt, dass ein Objekt ein anderes Objekt nach einem anderen Objekt fragt, ohne von diesem weitere Leistungen zu fordern (z.B. lange ***getthis***-Ketten). Nachrichtenketten sind somit vergleichbar zu dem Begriff „tramp data“ der strukturierten SW-Entwicklung.

- Vermittler

  Vermittlung bedeutet, dass eine Klasse bestimmte Leistungsanforderungen zur Erfüllung an eine andere Klasse delegiert. Wenn dies übermäßig oft geschieht, dann stimmt mit der Implementierung der Funktionen (Methoden) auf der Basis der logischen Anforderungen etwas nicht. In anderen Worten, wenn eine Klasse oft delegiert, dann ist besser gleich die andere Klasse zur Erfüllung bestimmter Anforderungen zu benutzen.

- Unangebrachte Intimität

  Unangebrachte Intimität bedeutet, dass sich Klassen über Gebühr mit den privaten Strukturen und Leistungen anderer Klassen beschäftigen, obwohl sie sich eigentlich „um ihre eigenen privaten Dinge“ (Attribute und Methoden) kümmern sollten.

Unangebrachte Intimität findet sich oft in Vererbungsbeziehungen, wo „Kinder mehr über ihre Eltern erfahren wollen“; dies zeigt sich z.B. in der Verwendung der *super*-Methode.

- Alternative Klassen mit verschiedenen Schnittstellen

  Hierbei geht es eigentlich um Methoden von Klassen, die gleiches oder ähnliches machen aber unterschiedliche Signaturen besitzen, also augenscheinlich andere Funktionalität implementieren.

- Unvollständige Bibliotheksklassen

  Hier ist der Entwickler von Standard-Klassenbibliotheken gefordert. Nach Fowler[17] ist ein Design von Anwendungssystemen erst nach vollständiger Fertigstellung des Systems letztendlich fertigt; vorher ist es eben entsprechend unvollständig. Entwickler von Standardbibliotheken (hier mehr die selbstentwickelten Basiskomponenten) müssen ihre Systeme einer Anwendungsentwicklung vorher zur Verfügung stellen; im Rahmen der Anwendungsentwicklung kann jedoch erst der entgültige funktionale Test der Standardbibliotheken stattfinden, so dass diese nach dem Verständnis eben unvollständig sind. Also im Rahmen des Refactorings von Anwendungssystemen findet sukzessive auch ein Refactoring von Standardbibliotheken statt.

- Datenklassen

  Datenklassen sind Klassen, die über Attribute verfügen und deren Verhaltensteil ausschließlich aus getter- und setter-Methoden besteht. Solche Klassen, auch wenn es sich dabei um „dumme Datencontainer[18]“ handelt, sind zunächst unkritisch. Problematisch ist es dann, wenn diese „Datencontainer“ ihre Attribute öffentlich anbieten (denn wozu braucht man die getter und setter eigentlich). Damit ist dann allerdings das grundlegendes OO-Prinzip der Kapselung verletzt[19]. Nach Fowler sind Datenklassen wie Kinder; „anfangs lassen wir sie gewähren, nach und nach sollten sie jedoch erwachsen werden und Verantwortung übernehmen“.

- Ausgeschlagenes Erbe

  Unterklassen erben Struktur und Funktion von ihren Oberklassen. Manchmal schlagen sie jedoch das Erbe aus und verwenden die Methoden nicht. Das ist nicht natürlich sondern eine künstliche Konstruktion im Rahmen der Programmierung. Man muss sich fragen, ob man diese Programmiertechniken zulassen will, dann aber hinreichend dokumentiert, oder ob man Umstrukturierung fordert, indem eine weitere abstraktere Oberklasse eingeführt wird[20].

---

[17] und gemäß der Extremen Programmierung

[18] Fowler

[19] Z.B. ist in der EJB-Spezifikation (1.x wie auch 2.x) gefordert, das alle EJB-Attribute *public* sein sollen. Dafür gibt es meines Erachtens nach eigentlich keine Notwendigkeit.

[20] Fowler selbst ist an dieser Stelle eher moderat. Erst wenn das Erbe auf Schnittstellenebene abgelehnt wird, dann reagiert Fowler etwas empfindlicher (sie Refactoring, p82).

- Kommentare

  … sind nicht notwendig (aber hilfreich[21]). Häufig ist es jedoch so, dass Kommentare verwendet werden, um Dinge zu beschreiben, die eigentlich machen wollte oder die man später besser machen sollte (die TODOs). Derartige Kommentare sind Indizien für spätere Refactoring-Maßnahmen und dann, wenn diese durchgeführt worden sind, hinfällig.

3 Software-Metriken und Audits

Software-Metriken und *Audits* sind Werkzeug-gestützte Verfahren, um Aussagen über die Güte von Architekturen machen zu können. Die untenstehende Liste von Metriken und Audits stammt aus dem Modellierungs- und Design-Werkzeug **Together ControlCenter** (CC). In Together erfolgt die Bewertung der Qualität auf der Basis des Codes, so dass nicht immer differenzierbar ist, ob hier die Güte des Codes oder die Güte der Architektur betrachtet wird. Ich möchte versuchen, die Charakteristiken der Metriken und Audits auf der Ebene der Architektur darzustellen.

Die Betrachtung von Kopplung und Kohäsion oben beschreibt mehr die Beziehungen zwischen Komponenten und Klassen in Komponenten; Kopplung und Kohäsion in dem hiesigen Kontext der Metriken umfasst mehr eine Betrachtung der Klassen in sich, also die Beziehungen der Methoden innerhalb einer Klasse; das ist gewissermaßen gleichzusetzen mit den traditionellen Begriffen der Kopplung und Kohäsion aus der strukturierten Software-Entwicklung[22].

Im Folgenden soll eine Übersicht über in **Together** angebotenen Metriken und Audits gegeben werden. Über die aufgeführten Metriken und Audits hinaus ist man in der Lage, auf der Basis der Together Open API eigene Metriken und Audits zu definieren. In der neueren Version von Together (Together 2006) erfolgt dies mit Hilfe der UML OCL[23].

3.1 Metriken

Im Allgemeinen kann man bereits recht viel qualitative Aspekte wie Komplexität zwischen Klassenbenutzungen und Strukturen innerhalb von Packages und Komponenten in den Modellen der Anwendungsarchitektur erkennen und verfolgen, vorausgesetzt, es gibt Modelle und Diagramme[24]. Metriken können ergänzend dazu hilfreich sein, Hinweise auf strukturelle Mängel und Sacherhalte zu erhalten. Eine ***Software-Metrik*** ist eine Funktion, die ein Software-Konstrukt in einen Zahlenwert abbildet.

---

[21] Sie haben nach Fowler einen „süßen Geruch“.

[22] Was ja an sich auch nichts schlechtes ist.

[23] Object Constraint Language

[24] Dies ist ja gerade eine der zentralen Qualitätsanforderungen.

Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft des Software-Konstruktes. Qualitätseigenschaften in diesem Sinne sind Fehlerfreiheit, Zuverlässigkeit, Effizienz, Benutzerfreundlichkeit, Wartbarkeit.

Hier wird Metrik als zahlenmäßiger Wert einer Berechnung verstanden – dies enspricht dem Verständnis der statischen Metrik; etwas weiter gefasst kann man auch die Zahlenwerte von Messungen einbeziehen, dies würde der Betrachtung und Bewertung des dynamischen Systems entsprechen, hierdurch wird der Begriff der dynamischen Metrik definiert. Im Folgenden soll ausschließliche statische Metriken behandelt werden.

Bei Metriken kann man grundsätzlich unterscheiden zwischen konventionellen Metriken und objekt-orientierten Metriken. Objektorientierte Metriken betrachten die Software als objekt-orientierte Struktur, also eine Klasse, bestehend aus Attributen und Methoden. Konventionelle Metriken sehen die Software-Einheit allgemeiner als Quelle in einer Datei. Konventionelle Metriken kann man weiter differenzieren in

- Umfangsmetriken, die die Größe der Programmquelle und die Anzahl der darin befindlichen Artefakte (Datentypen, Funktionen, Prozeduren) messen
- Datenstrukturmetriken, die den Umfang und die Verwendung von Daten im Programm messen
- Programmstrukturmetriken, mit denen die Anzahl und Verschachtelungstiefe von Anweisungen in Funktionen berechnet werden
- Stilmetriken, die den Stil der Programmierung in Zahlenwerten festhalten, z.B. Anzahl der Kommentarzeilen oder Verhältnis der Kommentierung zu dem Gesamt-Code

Die Klasse der OO-Metriken werden folgendermaßen gegliedert:

- Maße auf Methodenebene, mit denen analog zu den Programmstrukturmetriken die metrischen Eigenschaften von Methoden bewertet werden
- Maße auf Klassenebene, mit denen Strukturmerkmale von Klassen bestimmt werden
- Maße auf Vererbungshierarchien zur Betrachtung von hierarchischen Abstraktionsbeziehungen zwischen Klassen
- Maße auf Aggregationsbeziehungen (besser Assoziationen), mit denen alle nicht-hierarchieschen Beziehungen zwischen Klassen betrachtet werden

Der im weiteren aufgeführten Liste von Metriken liegt die Analyseeinheit des Modellierungswerkzeugs Together ControlCenter zugrunde. Together CC bewertet die Klassen eines Systems, wobei die berechneten Metrikwerte auf unterschiedliche Weise in den Packages und Komponenten aggregiert werden können:

- Summe – die Werte alle Klassen werden in den Komponenten summiert
- Maximium – der Maximalwert einer Klasse wir als Wert der Komponente angegeben
- Average – der Durchschnittswert über alle Klassen einer Komponente

Bei den Grenzwerten werden hier zunächst die voreingestellten Werte von Together angegeben, wobei die Grenzen in Together aufgrund eigener Erfahrungswerte für die Metrikgrenzen spezifiziert werden können.

So gibt es z.B. bei EJB-Projekten andere Komplexitätsvorgaben als bei simplen Java-Projekten. Es ist zweckmäßig, zu jedem Projekt ein Qualitätsprofil zu definieren, in dem die für das Projekt sinnvollen Metriken mit ihren spezifischen Grenzwerten festgelegt werden.

Am Ende einer Metrik-Beschreibung werde ich darauf eingehen, wie die jeweilige Metrik auch im Sinne der Fowler'schen Smells hilfreich sein kann.

3.1.1 Basismetriken / Umfangsmetriken

Die Basismetriken messen die Anzahl verschiedener Code-Artefakte wie Anzahl Zeilen, Anzahl Attribute und Operationen etc. Diese Metriken dienen im Wesentlichen zur quantitativen Bestimmung von Sachverhalten und Zuständen in dem Anwendungssystem, die man qualitativ auf der Basis von OO-Modellen bereits ebenso erkennen kann. Trotzdem sind die Basismetriken als Ergänzung zu den weiter unten aufgeführten Metriken hilfreich, worauf im einzelnen noch hingewiesen werden soll.

LOC *Lines of Codes* der Klasse resp. Komponente insgesamt

Art der Aggregation: Summe

Obere Grenze für eine Klasse: 1000

Die Anzahl der Zeilen einer Klasse oder Komponente (oder auch eines Anwendungssystems) ist architektonisch nicht so interessant. Allerdings gibt es in Form von „Faustregeln" gewisse Prinzipien wie „eine Komponente sollte ca. 5 bis 10 Klassen umfassen", „eine Methode sollte inhaltlich auf einer Seite (im Editor) vollständig angezeigt werden können", viele und mehrere unterschiedliche Programmstrukturen (if, case, switch, for, while) deuten i.A. darauf hin, dass in einer Methode mehr als eine Funktion implementiert ist (=> Verletzung der funktionalen Kohäsion). So kann die Metrik LOC herangezogen werden, um ggf. in den Klassen die Stellen mit der höheren Komplexität aufzufinden.

Smell: LOC kann herangezogen werden, um große Klassen zu finden.

NOA *Number of Attributes* der Klasse resp. Komponente insgesamt

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 30

Mit dieser Metrik wird die Anzahl der Attribute insgesamt gezählt; Konstanten gelten nicht als Attribute. Eine Differenzierung in der Datentypen und der Schutzeigenschaften (private, protected, public) wird nicht vorgenommen. Insofern ist dieser Metrik ein eher allgemeingültiger Wert für die Datenkomplexität einer Klasse, der mit Hilfe weiterer Metriken konkretisiert werden kann, z.B. AC, AHF, PPkgM, PPrivM, PProtM, PPubM.

Klassen mit einem NOA-Wert 0 sind ein Hinweis darauf, dass es sich um abstrakte Klassen oder Interfaces handelt, d.h. hiermit hat man ein Indiz für eine Abstraktion, auch wenn dies z.B. über eine entsprechende Benennung nicht erkennbar ist.

Alle Klassen mit NOA = 0 sollten untersucht werden; es gibt pathologische Fälle derart, dass Klassen nur Methoden und keine Attribute enthalten; dies ist nur für Interfaces, abstrakte Klassen und für die Klasse mit der „main“-Methode erlaubt.

Alle vermeintlichen Interfaces und abstrakte Klassen sollten untersucht werden; in diesen befinden sich zuweilen globale Konstanten (*public static final*).

Smell: Mit Hilfe von NOA kann man ebenfalls große Klassen finden.

NOC *Number of Classes* einer Komponente insgesamt

Art der Aggregation: Summe

Obere Grenze für eine Komponente: 10

Insgesamt muss man als Designer abwägen zwischen einer Architektur mit einer ausgewogenen Anzahl an Komponenten und einer ausgewogenen Anzahl an Klassen in den Komponenten.

Grundsätzlich ist die Frage, ob man Klassen innerhalb von Klassen zulassen möchte (NOC > 1 bezogen auf eine Klasse).

Smell: Die Anzahl von Klassen ist zwar nicht ein „unangenehmer Geruch“ im Sinne von Fowler, aber es ist trotzdem zweckmäßig, die Anzahl von Klassen einer Komponente oder eines Paketes zu begrenzen.

NOCON *Number of Constructors* der Klasse resp. Komponente

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 5

Klassen mit NOCON = 0 sind wieder vermeintliche abstrakte Klassen oder Interfaces. Wenn das nicht der Fall ist, dann kann es sein, dass diese Klasse über eine Factory-Konstruktion instantiiert wird. Das sollte ggf. über eine entsprechende Bennennung ersichtlich sein. Auch nach oben hin sollte die Anzahl von Konstruktoren sinnvoll begrenzt werden.

Smell: nicht definiert

NOO *Number of Operations* - Methoden - der Klasse resp. Komponente

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 70

Bei der Betrachtung von Operationen (im OO-Sprachgebrauch Methoden) ist eine Differenzierung nach den Schutzkriterien sinnvoll; siehe dazu MHF, PPkgM, PPrivM, PProtM, PPubM.

Smell: nicht definiert

NOM *Number of Members* – Attribute und Methoden - der Klasse resp. Komponente

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 50

I.A gilt: NOM = NOA + NOO, d.h. NOM ist nicht unbedingt eine sinnvolle Ergänzung.

Smell: nicht definiert

NOIS *Number of Import Statements* der Klasse resp. Komponente

Art der Aggregation: Maximum.

Mit NOIS hat man schon einen konkreteren Hinweis auf die Komplexität in der Weise, dass Klassen mit einem hohen NOIS-Wert „wahrscheinlich“ vergleichsweise stärker gekoppelt sind als andere. (Allerdings kann man das Bild z.B. mit import xxxx.*; verfälschen. Doch i.A. ist import xxxx.* ein eher schlechter Stil.) Hierbei sollen nur die imports berücksichtigt werden, die auf Komponenten des eigenen Systems verweisen; imports von Standard-Bibliotheken sind nicht so relevant. Dies kann natürlich mit dieser Metrik nicht unterschieden werden, in sofern sind alle kritischen NOIS-Werte unter diesem Gesichtspunkt noch einmal zu prüfen.

Interessant sind auch die Klassen mit einem NOIS = 0; d.h. diese Klassen importieren keine andere Klasse (sie assoziieren ggf. noch Klassen des Paketes, in dem sie sich befinden; diese müssen nicht importet werden). Klassen mit dem NOIS = 0 sollten untersucht werden, ob sie von anderen Klassen benutzt werden. Wenn nicht, dann sind sie isoliert und damit überflüssig. In Together CC gibt es für diese Assoziationsrichtung keine Metrik. Hierzu dienen die Analysetechniken „*Add Linked*“ oder „*Search for usages*“.

Smell: nicht definiert

3.1.2 Maximummetriken

Mit den Maximummetriken kann die Qualität des Designs bewertet werden, indem diverse Strukturen in ihrer Design-Tiefe betrachtet werden. Maximummetriken sind insofern Programmstrukturmetriken. In dieser Metrik können aus der Design-Erfahrung resultierende Grenzwerte vorgegeben werden.

MNOL *Maximum Number of Levels*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 7

Gezählt wird die maximale Tiefe von strukturierten Anweisungen in den Methodenrümpfen einer Klasse. MNOL liefert die maximale Schachtelungstiefe einer Klasse, d.h. es wird festgestellt, dass eine Methode der Klasse in Bezug auf MNOL einen maximalen Wert hat. (Mit dieser Metrik kann man nicht herausfinden, um welche Methode es sich handelt; hierzu muss man explizit die verdächtige Klasse untersuchen). Abweichend von der CC-Metrik wird im MNOL erst dann gezählt, wenn eine Methode auch über mindestens eine Kontrollstruktur verfügt. Methoden, die nur aus simplen Anweisungsfolgen bestehen, haben den MNOL-Wert 0.

Smell: lange Methoden; mit dieser Metrik kann festgestellt werden, wie tief eine Methode gegliedert ist. Tief unterstrukturierte Methoden „riechen unangenehm“.

MSOO *Maximum Size of Operation*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 10

Berechnet wird die maximale Größe über alle Operationen einer Klasse. Die Größe einer Operation ergibt sich aus dem Maß der Kontrollstrukturen (if, case, for, while). In sofern ist diese Metrik vergleichbar mit der CC-Metrik (cyclomatic complexity). Hier wird nicht die Verschachtelungstiefe zugrunde gelegt, sondern die Anzahl der Zweige. Das Ergebnis ist dar Maximalwert über alle Methoden einer Klasse; d.h. auch hier werden nur verdächtige Klassen gezeigt, die entsprechende Methode muss analytisch ermittelt werden.

Smell: lange Methoden; mit dieser Metrik kann die Anzahl der Kontrollstrukturen einer Methode gezählt werden. Methoden mit vielen Kontrollstrukturen, ob nun tief geschachtelt oder flach, „riechen unangenehm“

MNOP *Maximum Number of Parameters*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 4

Berechnet wird die maximale Anzahl von Parametern in den Signaturen der Methoden einer Klasse. Ermittelt wird der Maximalwert über alle Methoden der Klasse. Welche die tatsächlich komplexe Methode ist, muss analytisch untersucht werden. Die Komplexität der Parameter an sich, also der Umfang der Datentypen, wird hier nicht betrachtet. Insofern ist diese Metrik wenig aussagekräftig.

Smell: lange Parameterlisten; sowohl Signaturen mit einer großen Anzahl von Parametern wie auch Signaturen mit komplexen (stark strukturierten Parametern) „riechen unangenehm“. Mit dieser Metrik wird die Anzahl der Parameter in Signaturen gezählt, ohne die Strukturen zu differenzieren.

3.1.3 Allgemeine Komplexitätsmetriken

Mit den Komplexitätsmetriken wird der innere Aufbau der Klassen betrachtet. Anhand des Komplexitätswertes kann die Struktur einer Klasse weiter untersucht werden.

AC *Attribute Complexity*

Art der Aggregation: Average

Die Attributkomplexität fällt in die Klasse der Datenstrukturmetriken. Die AC betrachtet die Anzahl der Attribute einer Klasse und deren Datentypen. Attribute unterschiedlichen Typs haben unterschiedliche Komplexität. D.h. simple Datentypen sind i.A. weniger komplex als abstrakte Datentypen (resp. Instanzen von Klassen).

Die Attributkomplexität ist eine gute Ergänzung zu der Basismetrik NOA. Die Verwendung von simplen Attributen impliziert eine geringere Komplexität als die Verwendungen von komplexen Attributen. Grenzwerte werden von Together nicht vorgegeben, können aber eingestellt werden. Allgemein kann gesagt werden, eine hohe AC impliziert eine höhere Komplexität. Klassen mit der AC (analog NOA) = 0 sind interessant; hierbei handelt es sich ggf. im Interfaces oder abstrakte Klassen.

Smell: die Komplexität der Attribute von Klassen ist im Fowler'schen Sinne nicht relevant. Allerdings kann man diese Metrik heranziehen, um die „Neigung zu elementaren Typen" zu prüfen.

CC *Cyclomatic Complexity*

Art der Aggregation: Maximum

Die *Cyclomatic Complexity*-Metrik fällt in die Klasse der Programmstrukturmetriken. Mit *CC* wird das Maß der Verzweigungen und Pfade (Anzahl und Tiefe der Kontrollstrukturen) in den Methoden einer Klasse berechnet und aggregiert. Die CC-Metrik ist vergleichbar mit der MNOL-Metrik; CC wird allerdings nicht als Maximum sondern als Summe der zyklomatischen Werte über alle Methoden einer Klasse ermittelt. Aus dem Grund ist der CC-Wert stets größer als der MNOL-Wert.

Im Gegensatz zu der MNOL-Metrik wird bei CC bereits das Vorhandensein einer Methode als „minimale" Komplexität mit 1 bewertet. Grenzwerte werden in Together nicht vorgegeben, können aber eingestellt werden; der kritische Wert, verglichen mit MNOL, liegt irgendwo bei 100. Allgemein kann gesagt werden, eine hohe CC impliziert eine höhere Komplexität. Mit dem CC-Wert ist keine direkte Bezugnahme auf eine Methode möglich, insofern ist die MNOL-Metrik aussagekräftiger.

Smell: lange Methoden

NORM *Number of Remote Methods*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 30

Remote Methods sind die Methoden, die über eine Benutzungsbeziehung in die betrachtete Klasse „hinein kommen“, also Methoden, die nicht in dieser Klasse oder in deren Oberklassen deklariert sind. Bei der NORM-Metrik geht es um den Grad der Benutzung von *Remote Methods*. Auffällig ist auch wieder der Wert NORM = 0; dies ist per se bei Interfaces und abstrakten Klassen der Fall, aber auch bei Klassen, die ausschließlich eigene oder ererbte Methoden verwenden.

Smell: Es ist auffällig, wenn eine Klasse kaum Methoden besitzt oder kaum eigene Methoden benutzt aber in starkem Maße die Methoden von instantiierten Objekten. Dies „riecht nach“ Neid. Außerdem ist zu erwarten, dass diese Klasse oft dann zu ändern ist, wenn die benutzten Klassen geändert werden (divergierende Änderungen).

RFC *Response for Class*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 50

Eine Klasse hat eine gewisse Verantwortung gegenüber seiner Umwelt in dem Sinne, dass sie für alles verantwortlich ist, was sie der Umwelt öffentlich zur Verfügung stellt. Dies sind alle public Methoden, die sie selbst enthält (oder erbt) zuzüglich der Remote-Methoden. Mit RFC wird eine Metrik für diese Verantwortung definiert (*Number of Local Methods* + *Number of Remote Methods*). (Diese Metrik ergibt leicht ein negatives Bild, wenn z.B. im Kontext J2EE von EntityBeans oder SessionBeans geerbt wird.) I.A. ist die RFC-Metrik vergleichbar mit NORM, allerdings umfassender.

Smell:nicht definiert

WMPC1 *Weighted Methods per Class 1*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 30

Weighted Methods per Class beschreibt die Gesamtheit der Komplexität der Methoden einer Klasse, wobei jede Methode gewichtet wird nach ihrer „cyclomatic complexity“ (siehe oben CC, in den Beispielen sind die Werte auch oft identisch). In diesem Kontext werden nur Methoden betrachtet, die Bestandteil der Klasse selbst sind, also nicht etwa ererbte Methoden.

Die WMPC1 liefert im Vergleich zu CC oder MNOL keine neuen Erkenntnisse.

Smell: nicht definiert

WMPC2 *Weighted Methods per Class 2*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 100

Bei der WMPC2-Metrik wird die Anzahl der Methoden einer Klasse und der Umfang der Signaturen (Anzahl der Parameter) berechnet. Der WMPC2-Metrik liegt die Annahme zugrunde, dass Klassen mit mehr Methoden oder Klassen mit Methoden mit umfangreichen Signaturen komplexer sind. Auch die WMPC2–Metrik ist relativ unverbindlich und nicht so ergiebig.

Smell: nicht definiert

3.1.4 Kohäsionsmetriken

Bei Kohäsionsmetriken handelt es sich um Messgrößen aus der Klasse der OO-Metriken. Mit Kohäsionsmetriken wird der innere Aufbau einer Klasse bewertet in dem Sinne, dass die „Ähnlichkeit" der Methoden betrachtet wird. Methoden einer Klasse sind dann ähnlich, wenn sie auf dengleichen Attributen arbeiten. In dem Fall ist die Kopplung zwischen den Modulen verhältnismäßig hoch und die Kohäsion demzufolge gering. LOCOM heißt *Lack of Cohesion of Methods* (Mangel an innerer Bindung); bezogen auf eine Methode ist eine möglichst hohe Kohäsion gewünscht, d.h. *Lack of Cohesion* sollte eher klein sein, also die Kohäsion ist dann hoch, wenn der Mangel an Kohäsion gering ist.

LOCOM1 *Lack of Cohesion of Methods 1*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: keine Angabe

Bei LOCOM1 wird bewertet, wie die Methoden einer Klasse die Attribute der Klasse benutzen. Dazu werden die Methoden paarweise verglichen. Wenn verschiedene Methoden dieselben Attribute benutzen, dann ist die Kohäsion gering (und die Kopplung der Methoden in der Klasse somit hoch). Wenn die Kombinationen der Attribute, die von den Methoden benutzt werden, weitgehend disjunkt ist, dann ist die Kohäsion hoch (dies entspricht dem Begriff der funktionalen Kohäsion) und die Kopplung eher gering.

LOCOM1 = (Anzahl der Paare von Methoden der betrachteten Klasse ohne gemeinsame Instanzvariablen) - (Anzahl der Paare von Methoden dieser Klasse mit gemeinsamen Instanzvariablen)

Ziel ist es hier, dass die Methoden der Klasse möglichst disjunkte Mengen der Attribute benutzen. Methoden, die auf nicht disjunkten Attributmengen arbeiten, sind stärker miteinander gekoppelt (und damit schwächer kohäsiv).

Ein hoher Wert zeigt ein hohes Maß an Kohäsion (und nicht etwa einen hohen Mangel an Kohäsion, was man mit LOCOM vermuten könnte). Der Wert dieser Metrik ist schwierig auf die betrachtete Klasse umzusetzen, zumal da Klassen mit einer hohen Anzahl von Methoden auch einen verhältnismäßig hohen LOCOM1-Wert ergeben können. D.h. wenn überhaupt, dann macht eine Betrachtung der LOCOM1-Metrik Sinn im Zusammenhang mit der NOO-Metrik.

Smell: Wenn der LOCOM1-Wert hoch ist, dann gibt es ein hohes Maß an Datenklumpen in der betrachteten Klasse.

LOCOM2 *Lack of Cohesion of Methods* 2

Art der Aggregation: Maximum.

Grenze für eine Klasse: 30 - 100

Allgemein gilt, LOCOM2 ist ein Prozentwert, der aussagt, wie gut die Attribute einer Klasse von den Methoden der Klasse genutzt werden.

$M(a_i)$ ist die Anzahl aller Methoden, die das Attribut a_i benutzen.

LOCOM2 = (1 – (Summe aller $M(a_i)$ / (Anzahl der Attribute) * (Anzahl der Methoden)) * 100

LOCOM2 ergibt einen prozentualen Wert; ein hoher Wert bedeutet eine hohe Kohäsion der Klasse. Dadurch, dass LOCOM2 relativ über alle Methoden und Attribute der Klasse bestimmt wird, ist er aussagekräftiger als LOCOM1.

Smell: nicht definiert

LOCOM3 *Lack of Cohesion of Methods* 3

Art der Aggregation: Maximum.

Grenze für eine Klasse: 30 – 100

LOCOM3 = ((Anzahl aller Methoden) - (Summe aller $M(a_i)$) / (Anzahl aller Attribute))
/ ((Anzahl aller Methoden) – 1) * 100

Ein geringer LOCOM3-Wert zeigt eine geringe Kohäsion an. LOCOM2 und LOCOM3 sind in ihrer Aussage in etwa gleichwertig.

Smell: nicht definiert

Anmerkung:

Die Implementierung von Beans mit den *getter*- und *setter*-Methoden würde das Kohäsionsbild auf der Basis dieser Metriken verfälschen, da *getter* und *setter* per se auf den gleichen Attributen operieren.

3.1.5 Kopplungsmetriken

Kohäsionsmetriken beleuchten den inneren Aufbau der Klassen und die Beziehungen der Methoden untereinander. Mit den Kopplungsmetriken wird das Maß der Kopplung zwischen Klassen ermittelt, d.h. es wird die Frage aufgeworfen, wie und in welchem Maße die Klassen untereinander in Beziehung stehen. Kopplungsmetriken gehören zu der Klasse der OO-Metriken.

Bezogen auf die Kohäsion war ein hohes Maß gefordert; die Kopplung zwischen den Klassen sollte eher gering sein, allerdings muss eine Mindestkopplung vorliegen, andernfalls ist die betrachtete Klasse nicht brauchbar.

Klassen sind dadurch miteinander gekoppelt, dass sie in irgendeiner Assoziation zueinander stehen (dies ist bereits in den jeweiligen Modellen ersichtlich). Mit Hilfe der Kopplungsmetriken erhält man einen metrischen Wert über das Maß der Kopplung.

CBO *Coupling between Objects*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 30

CBO repräsentiert die Anzahl der Klassen, die mit der betrachteten Klasse gekoppelt sind. Hierfür werden alle Objektreferenzen (nicht aus java.lang.*) innerhalb der Klasse gezählt, Attributdeklarationen, formale Parameter, Returnwerte etc. Je höher dieser Wert ist, desto größer ist das Maß der Kopplung.

Zudem ist es auffällig, wenn dieser Wert wesentlich höher ist als die Anzahl der Attribute (NOA); dies ist ein Indiz dafür, dass relativ viel Objektinstantiierungen z.B. in formalen Parametern „versteckt“ sind, was allemal überprüft und überdacht werden sollte.

CBO = 0 würde bedeuten, dass diese Klasse (aus ihrer Sicht) mit keiner anderen gekoppelt ist. Dann ist interessant, ob eine Kopplung aus der Sicht anderer Objekte vorliegt; wenn nicht, dann ist die Frage, ob die Klasse überhaupt benötigt wird.

Smell: Wenn dieser Wert hoch ist, dann „riecht es“ verdächtig danach, dass bezogen auf die gekoppelten Objekte divergierende Änderungen zu erwarten sind.

CDBC Change Dependency Between Classes

Art der Aggregation: Maximum.

Bei *Dependency Between Classes* werden Client/Server-Beziehungen zwischen Klassen definiert: eine Klasse ist Client einer Serverklasse, wenn sie

a) im Sinne einer Vererbungsbeziehung die untergeordnete Klasse ist,
b) sie ein Attribut einer anderen Klasse besitzt (dies ist im Sinne der Kapselungsforderung eher suspekt),

c) sie Instanzen von einer anderen Klasse (als Attribute oder Methodenparameter) besitzt.

CDBC ist ein Maß für den Aufwand der Änderung einer Client-Klasse, wenn die dazugehörige Server-Klasse geändert wird. CDBC sollte möglichst gering sein.

Smell: nicht definiert

CF *Coupling Factor*

Der *Coupling Factor* wird ermittelt aus dem Verhältnis zwischen der Anzahl der "nicht-Vererbungs-Beziehungen zwischen Klassen" und der „möglichen Anzahl der maximalen Kopplungen eines Systems". Diese Metrik, die nicht auf Klassen sondern auf Pakete/Komponenten angewendet werden kann, ist wenig aussagekräftig.

Smell: nicht definiert

DAC *Data Abstraction Coupling*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 5

Metrik für die Anzahl von referenzierten Klassen; die referenzierten Klassen werden hier als Attribute mit komplexen Datentypen betrachtet. Attribute von simplen Datentypen werden in dieser Metrik ignoriert. DAC ist vergleichbar mit CBO.

CBO ist umfassender, da hier nicht nur Objektreferenzen in den Attributen gezählt werden, sondern auch die in den Parameterlisten.

Smell: Wenn zwischen Klassen eine hohe DAC-Komplexität erkannt wird, ist der Vedacht groß, dass divergierende Änderungen zu erwarten sind.

FO *FanOut*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 15

Analog zu CBO und DAC; hier werden alle komplexen Attribute, formalen Parameter und Rückgabewerte etc. gezählt. FO ist wie DAC eine Untermenge von CBO, insofern ergibt sich hiermit auch keine zusätzliche Erkenntnis.

Smell: analog zu DAC und CBO ein Indiz für divergierende Änderungen

MIC *Method Invocation Coupling*

Art der Aggregation: Maximum.

Diese Metrik misst die (relative) Anzahl von anderen Klassen, die von dieser Klasse per Methodenaufruf benutzt werden. Dieser Wert ist relativ zu der Anzahl aller Klassen des Systems.

Smell: ein Maß für divergierende Änderungen

VOD *Violation of Demeter's Law*

Art der Aggregation: Maximum

Demeter's Law: „*Don't talk to strangers*" oder „*Do use only one dot*" (Quelle: *Radu Marinescu: An Object Oriented Metrics Suite on Coupling,* siehe auch Erläuterungen dazu im Together CC) In den Methoden einer Klasse dürfen nur andere Methoden der Klasse oder Methoden anderer Klassen über deren Objektinstanzen oder über die Argumente der Methode aufgerufen werden.

Umgangssprachlich erläutert sagt Demeter's Law aus, man soll bei der Benutzung von Methoden (objektinstanz.methode()) immer nur über eine Ebene referenzieren; objektinstanz.objektinstanz.methode() wäre eine Verletzung dieser Regel.

Ein geringer VOD-Wert (möglichst = 0) ist allemal anzustreben.

Smell: nicht definiert

3.1.6 Kapselungsmetriken

Kapselung ist ein wesentlicher Designaspekt, der aus der strukturierten SW-Entwicklung in die OO-Methodik eingeflossen ist. Kapselung – oder auch *Information Hiding* - wird in OO-Sprachen per se dadurch erreicht, dass alles, was nicht nach außen bekannt gemacht werden muss, als privat definiert wird. Ziel ist, eine möglichst hohe Kapselungsrate (~100%) zu erzielen. In den folgenden Kapselungsmetriken werden nun sowohl die Attribute als auch die Methoden betrachtet.

Bezogen auf die Attribute ist 100 % durchaus zweckmäßig, bezogen auf die Methoden macht eine 100% Kapselung natürlich keinen Sinn, obwohl auch hier ein möglichst hohes Maß angestrebt werden sollte.

AHF *Attribute Hiding Factor*

AHF wird über das gesamte Projekt betrachtet. AHF ergibt sich aus dem Faktor (Anzahl aller geschützten Attribute) / (Anzahl aller Attribute)

Ziel ist hier ein hoher Wert an AHF (=100%), was einem hohen Maß an *information hiding* entspricht.

Smell: nicht definiert

MHF *Method Hiding Factor*

MHF wird analog zu AHF über das gesamte Projekt betrachtet. In der Regel ist der MHF-Wert eines Projektes stets kleiner als 100, da ja die Methoden als Services einer Klasse gerade zur Benutzung angeboten werden.

Wenn der MHF-Wert von einer Methode = 100% ist, dann ist diese Methode „absolut“ gekapselt und damit nicht verwendbar.

Smell: nicht definiert

Bei Beans ergibt sich mit diesen Metriken abermals eine Schieflage, da die Bean-Attribute laut EJB-Spezifikation „public“ sein müssen und die Methoden in der Regel ebenfalls alle „public“ sind.

3.1.7 Inheritance-Metriken

Diese Metriken entstammen im Wesentlichen – wie auch die Metrik CF – der Quelle *Metrics for Object-Oriented Development* (MOOD).

AIF *Attribute Inheritance Factor*

AIF ist vergleichbar zu der Metrik AHF (*Attribute Hiding Factor*) – es entstammt auch derselben Quelle; berechnet wird hier (Summe aller ererbten Atttribute in allen Klassen) / (Summe alle Attribute aller Klassen). AIF wird stets über alle Klassen eines Paketes ermittelt, d.h. diese Metrik macht wie auch AHF eine Aussage über die Komponente.

Smell: nicht definiert

MIF *Method Inheritance Factor*

MIF ist vergleichbar zu der Metrik MHF (*Method Hiding Factor*); berechnet wird hier der Faktor aus (Summe aller ererbten Methoden aller Klassen) / (Summe alle Methoden aller Klassen). MIF wird stets über alle Klassen eines Paketes ermittelt, d.h. diese Metrik macht wie auch MHF eine Aussage über die Komponente.

Smell: ausgeschlagenes Erbe; wenn AIF und MIF hoch sind, dann ist zudem der Verdacht auf unangebrachte Intimität groß.

DOIH *Depth Of Inheritance Hierarchy*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 5

Ermittlung der Tiefe der Vererbung. Es wird hier zugrunde gelegt, dass eine große Vererbungstiefe im Sinne der Pflege und Wartung eine entsprechend große Komplexität bedeutet.

Smell: ein weiteres Indiz für das Schrotkugel-Syndrom

NOCC *Number of Child Classes*

Art der Aggregation: Maximum.

Anzahl der Klassen, die Spezialisierungen von der betrachteten Klasse sind. Dies bedeutet eine Betrachtung der Vererbungstiefe „nach unten“, wobei eine große Tiefe eben auch als komplex angesehen wird. Hierzu ist allgemein zu sagen, dass Vererbung stets eine große Kopplung zwischen Klassen bedeutet und demnach entsprechend vorsichtig anzuwenden ist.

Smell: ein weiteres Indiz für das Schrokugel-Syndrom

Die TR-Metriken (Total Reuse ...) entstammen – wie auch die Metriken MID und VOD – der Quelle *Radu Marinescu: An Object Oriented Metrics Suite on Coupling*. Die Definitionen dieser Metriken sind relativ komplex, eine allgemein formulierte Zusammenfassung ist ein wenig schwierig, so dass hierzu auf die Quelle verwiesen werden soll (siehe auch Erläuterung der Metriken in Together ControlCenter). Total Reuse-Metriken geben in Vererbungsbeziehungen das Maß der der Wiederverwendbarkeit wieder jeweils aus der Sicht der Superklasse (Ancestors) und der Nachfolger (Descendants); dieser Wert kann jeweils absolut oder prozentual ermittelt werden. Die Werte aus dem Blickwinkel Ancestor oder Descendant korrelieren in dem Sinne, dass wenn der eine Wert hoch ist, dann ist der jeweils andere niedrig.

TRDU *Total Reuse in Descendants Unitary*

Absolute Ermittlung der TR-Metrik bezogen auf die erbenden Klassen.

Smell: nicht definiert

TRDP *Total Reuse in Descendants Percentage*

Prozentuale Ermittlung der TR-Metrik bezogen auf die erbenden Klassen.

Smell: nicht definiert

TRAU *Total Reuse from Ancestors Unitary*

Absolute Ermittlung der TR-Metrik aus Sicht der Vorgängerklassen.

Smell: nicht definiert

TRAP *Total Reuse from AncestorsPercentage*

Prozentuale Ermittlung der TR-Metrik aus Sicht der Vorgängerklassen.

Smell: nicht definiert

3.1.8 Polymorphiemetriken

Bei den diesen Metriken wird das Polymorphie-Prinzip ***inclusion*** betrachtet (daneben werden noch differenziert ***coercion***, ***overloading***, ***parametric***); vorausgesetzt ist eine Vererbungsbeziehung zwischen Klassen, wobei die erbenden Klassen zu ihren ererbten Artefakten (hier: Attribute und Methoden) weitere

Attribute und Methoden ergänzen bzw. Methoden redefinieren können, wodurch das in diesem Sinne polymorphe Verhalten begründet wird. Gleich welches Polymorphie-Prinzip vorliegt und so elegant Polymorphie in OO-Sprachen auch ist, es ist stets ein Sachverhalt von höherer Komplexität.

NOAM *Number of Added Methods*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 50

Mit dieser Metrik werden die Anzahl der Methoden gezählt, die in einer erbenden Klasse zu den ererbten Methoden hinzugefügt werden. Die aus Together abgeleitete Grenze von 50 ist aus meiner Sicht recht hoch.

Smell: nicht definiert

NOOM *Number of Overridden Methods*

Art der Aggregation: Maximum.

Obere Grenze für eine Klasse: 10

Mit dieser Metrik werden die Anzahl der ererbten Methoden gezählt, die von der erbenden Klasse redefiniert werden.

Smell: nicht definiert

PF *Polymorphism Factor*

Der PF ist eine Metrik, die über das gesamte Projekt ermittelt wird:

Summe der redefinierten Methoden aller Klassen / maximale Anzahl der „polymorphen Situationen“ (Summe über alle Produkte von „alle neuen Methoden einer Klasse multipliziert mit der Anzahl der Nachfolger dieser Klasse“) (Quelle: MOOD).

Smell: nicht definiert

3.1.9 Halstead-Metriken

Die Halstead-Metriken (*Halstead Software Science Metrics*) fallen in der Klassifizierung aus dem Rahmen, da sie eine Anzahl von Umfangsmetriken umfassen, die wir so ähnlich bereits kennen gelernt haben; diese werden allerdings benötigt, um weitere Größen zu bestimmen, die eine Beurteilung der Entwicklungsschwierigkeit (*Difficulty*) und des Testaufwands (*Effort*) beschreiben.

NOprnd *Halstead Number of Operands*

number of operands used in a class

Smell: nicht definiert

NOprtr *Halstead Number of Operators*

number of operators used in a class

Smell: nicht definiert

NUOprnd *Halstead Number of Unique Operands*

number of unique operands used in a class

Smell: nicht definiert

NUOprtr *Halstead Number of Unique Operators*

number of unique operators used in a class

Smell: nicht definiert

HPLen *Halstead Program Length*

'Number of Operators' + 'Number of Operands'

Smell: nicht definiert

HPVoc *Halstead Program Vocabulary*

'Number of Unique Operators' + 'Number of Unique Operands'

Smell: nicht definiert

HPVol *Halstead Program Volume*

'Halstead Program Length' * Log2('Halstead Program Vocabulary')

Smell: nicht definiert

HDiff *Halstead Difficulty*

('Number of Unique Operators' / 2) * ('Number of Operands' / 'Number of Unique Operands')

Smell: nicht definiert

HEff *Halstead Effort*

'Halstead Difficulty' * 'Halstead Program Volume'

Smell: nicht definiert

In Together CC sind für die Halstead-Metriken keine Grenzwerte angegeben. Die berechneten Zahlen sind schwierig zu interpretieren. 0-Werte deuten wieder auf Interfaces oder abstrakte Klassen hin.

3.1.10 Ratio-Metriken

Mit den Ratio-Metriken werden bestimmte Verhältnisse in prozentualer Form dargestellt, die wir oben schon als Basismetriken kennen gelernt haben, unter anderem auch das Maß an Kommentierung (und das ist allemal ein interessanter Qualitätsaspekt).

PPkgM *Percentage of Package Members*

Art der Aggregation: Average.

Obere Grenzwerte für eine Klasse: 10

Verhältnis der Klassen in einer Komponente. Diese Metrik, die in Together CC eigentlich stets 0 ist, ist nicht aussagekräftig.

Smell: nicht definiert

PPrivM *Percentage of Private Members*

Art der Aggregation: Average.

Verhältnis der privaten Elemente einer Klasse im Vergleich zu der Gesamtanzahl.

Smell: nicht definiert

PProtM *Percentage of Protected Members*

Art der Aggregation: Average.

Verhältnis der geschützten Elemente einer Klasse im Vergleich zu der Gesamtanzahl.

Smell: nicht definiert

PPubM *Percentage of Public Members*

Art der Aggregation: Average.

Verhältnis der öffentlichen Elemente einer Klasse im Vergleich zu der Gesamtanzahl.

Smell: nicht definiert

Bei den *private, protected, public Membern* wird (bedauerlicherweise) nicht zwischen Methoden und Attributen unterschieden. Attribute sollten stets *private* sein, einige *public* Methoden sollte es geben.

CR *Comment Ratio*

Art der Aggregation: Minimum.

Grenzwerte für eine Klasse: 5 – 100

Ermittelt das Verhältnis der „Inline“-Dokumentation im Vergleich zu dem Code-Umfang der Klasse.

CR = (Anzahl der Kommentarzeilen) / ((Anzahl der Netto-Code-Zeilen) + (Anzahl der Kommentarzeilen)) * 100

Ein möglichst hoher Wert – also nahe bei 100 – ist natürlich anzustreben.

Smell: nicht definiert

TCR *True Comment Ratio*

Art der Aggregation: Minimum.

Grenzwerte für eine Klasse: 5 – 400

Ermittelt das Verhältnis der „Inline“-Dokumentation im Vergleich zum gesamten Code-Umfang der Klasse.

CR = (Anzahl der Kommentarzeilen) / (Anzahl der Netto-Code-Zeilen) * 100

Ein möglichst hoher Wert – also nahe bei 100 – ist natürlich anzustreben.

Smell: nicht definiert

Die *Comment Ratios* sind im Fowler’schen Sinne nicht so relevant. Kommentierung ist gut und wichtig und mit den *Comment Ratios* ist die Quantität der Kommentierung messbar. Fowler betrachtet mehr die Qualität in der Art, dass er nach nicht sachbezogenen Kommentierungen sucht, die mehr dem Zweck der ToDos dienen. Derartige Kommentierungen sind metrisch nicht messbar; hier können Audits helfen, auf die ich unten noch eingehen werde.

3.1.11 Userinterfacemetrik

NOCF *Number of Controls in Form*

Art der Aggregation: Summe.

Grenzwerte für eine Klasse: 50

Anzahl der GUI-Elemente in einer Maske resp. Formular. Diesbezüglich gibt es ergonomische Qualitätsanforderungen, die hier nicht weiter betrachtet werden sollen.

Smell: nicht definiert

3.2 Audits

Audits dienen der automatischen Source-Code Prüfung. Mit ihnen wird der Programmierstil nach vorgegebenen Regeln analysiert und bewertet. Audits sind demnach mehr ein qualitativer Maßstab für die Implementierung und nicht so sehr für das Design und die Anwendungsarchitektur[25].

Die Kategorisierung der Einsetzbarkeit ist bereits ein Vorschlag für die Anwendung der einzelnen Audits unter Berücksichtigungen von Projekterfahrungen. Eingeschränkt einsetzbar bedeutet, dass die Regel zwar relevant ist, aber in bestimmten Fällen die Fehlermeldungen ignoriert werden kann, weil es sich um

1. einen Audit handelt, der gegen die Spezifikation im Entwicklerhandbuch verstößt
2. z.B. von Together erzeugten Code handelt
3. eine nicht vollständige Implementation handelt:

3.1 einzelne Variablen oder Methoden werden in Zukunft gebraucht (p0)

3.2 unvollständige Code-Abschnitte (zu erkennen an einem Implementierungskommentar mit dem Schlüsselwort TODO)

Im Folgenden werden einige Audits des Modellierungswerkzeuges **Together** (Version 6) exemplarisch vorgestellt:

APAPIV: Variablen, die persistent sein müssen, werden public deklariert (Entwickler Handbuch).

BTIJDC: Selbstdefinierte Texte werden nach dem Entwicklerhandbuch definiert (sind nicht unbedingt JAVA Standard)

HON: wird von Together selbst generiert

LAPAPMF: Verstoß gegen das Entwickler Handbuch

NC: Verstoß gegen das Entwickler Handbuch

ULVAFP: wird von TogetherJ selbst generiert. Im technischen Durchstich sind einige Variablen unbenutzt.

Die Klassifizierung der Bedeutung der Audits in high (H), normal (N) und low (L) wird zunächst von Together übernommen. Dem Architekten und Qualitätssicherer ist es möglich, die Bedeutung der einzelnen Audits für ein Projekt selbst festzulegen.

3.2.1 Vollständige Liste der in Together vordefinierten Audits

ADVIL Avoid Declaring Variables inside Loops

AHIIV	Avoid Hiding Inherited Instance Variables
AHISM	Avoid Hiding Inherited Static Methods
AMBDIOSE	Attributes Must Be Declared in One Statement Each
AOSMTO	Access of Static Members through Objects
APAPIV	Avoid Public and Package Instance Variables
AFSWEB	Avoid Statements with Empty Body
ATFLV	Assignment to for-loop Variables
ATFP	Assignment to Formal Parameters
ATSWL	Append to String within a Loop
BLAD	Badly Located Array Declarators
BTIJDC	Bad Tag in JavaDoc Comments
CA	Complex Assignment
CLE	Complex Loop Expressions
CNMMIFN	Class Name Must Match Its File Name
CPAMBF	Constant Private Attributes Must Be Final
CQS	Command Query Separation
CVMBF	Constant Variables Must Be Final
DBJAOC	Distinguish between JavaDoc and Ordinary Comments
DCFPT	Don't Compare Floating Point Types
DID	Duplicate Import Declarations
DIPSFBT	Don't Import the Package the Source File Belongs to
DUNOF	Don't Use the Negation Operator Frequently
EBWB	Enclosing Body within a Block
EIAV	Explicitly Initialize All Variables
EIOJLC	Explicit Import of the java.lang Classes
EOOBA	Equality Operations on Boolean Arguments
GOWSNT	Group Operations with Same Name Together
HON	Hiding of Names

[25] was in Tools wie Together nicht immer leicht unterschieden werden kann

ICM	Inaccessible Constructor Matches
ICSBF	Instantiated Classes Should Be Final
IIMBU	Imported Items Must Be Used
IMM	Inaccessible Method Matches
LAPAPMF	List All Public and Package Members First
MFDTCSF	Method finalize() Doesn't Call super.finalize()
MLOWP	Mixing Logical Operators without Parentheses
MVDWSN	Multiple Visible Declarations with Same Name
NAICE	No Assignments in Conditional Expressions
NC	Naming Conventions
NOEC	Names of Exception Classes
OMNBU	Operator '?:' May Not Be Used
ONAMWAM	Overriding a Non-Abstract Method with an Abstract Method
OOAOM	Order of Appearance of Modifiers
OPM	Overriding a Private Method
OWS	Overloading within a Subclass
PIIFS	Provide Incremental in for-statement or Use while-statement
PMFL	Put the Main Function Last
RFDI	Replacement for Demand Imports
UAAO	Use Abbreviated Assignment Operator
UC	Unnecessary Casts
UCVN	Use Conventional Variable Names
UEIOE	Use 'equals' instead of '=='
UIOE	Unnecessary 'instanceof' Evaluations
ULIOL	Use 'L' instead of 'l' at the End of Integer Constant
ULVAFP	Unused Local Variables and Formal Parameters
UOOIM	Use of Obsolete Interface Modifier
UOSM	Use of the 'synchronized' Modifier
UOUIFM	Use of Unnecessary Interface Field Modifiers
UOUIMM	Use of Unnecessary Interface Method Modifiers

UPCM	Unused Private Class Method
UPCV	Unused Private Class Variable
USFI	Use of Static Field for Initialization
UTETACM	Use 'this' Explicitly to Access Class Members
VMBDIOSE	Variables Must Be Declared in One Statement Each

4 Metrikprofile – Vorschläge

Auf der Basis der oben beschriebenen Metriken zur quantitativen Bewertung von SW-Systemen sollen im Folgenden einige Metrikprofile festgelegt werden, um auf der einen Seite Hinweise auf die Komplexität des Systems zu geben, hier können neben den Komplexitätsmetriken auch die Metriken zur Bewertung der Kopplung und der Kohäsion heran gezogen werden, auf der anderen Seite sollen Metriken angegebene werden, mit denen man die im Fowler'schen Sinne auffälligen[26] Code-Bestandteile auffinden kann. Eine grundsätzliche Metrik zu Bewertung von Systemen sind die Halstead-Metriken, mit denen Schwierigkeit und Pflegeaufwand quantitativ bestimmt werden.

4.1 Komplexitäts-, Kopplungs- und Kohäsions-Metriken

Komplexität erkennt man auf oberen Ebenen bereits recht gut qualitativ anhand der UML-Darstellung der Architektur. Man kann darin schon sehen, ob es in Subsystemen und Paketen verhältnismäßig flache und stark komplexe Strukturen gibt – mit vielen Klassen und Subpaketen, die stark untereinander verknüpft sind – oder ob die Architektur mehr in die Tiefe geht und dort Subpakete hat mit geringerer Komplexität. Eine weitere Erkenntnis ist, ob die Komponenten einer inneren Schicht – innerhalb von Subsystemen oder Paketen – in gewisser Hinsicht im Sinne einer Kohäsion logisch sinnvoll zusammenhängen oder ob es isolierte alleinstehende Klassen oder Klassensysteme gibt. Eine weitergehende quantitative Betrachtung mit Metriken ist dann angezeigt, wenn man tiefer in die Klassen hinein blicken will, um dort z.B. Kohäsion zu untersuchen; quantitative Ansätze machen auch da Sinn, wo man eine Vielzahl von Komponenten und Klassen betrachten möchte ohne in jede explizit hineinzublicken. D.h. eine quantitative Bewertung dient dem Zweck, die Kandidaten herauszufinden, die einer weiteren qualitativen Untersuchung unterzogen werden sollen.

- Komplexitätsmetriken

  Zu den sinnvollen Komplexitätsmetriken gehören die allgemeinen Umfangsmetriken

  LOC *Lines of Code*

  NOC *Number of Classes*

  NOA *Number of Attributes*

NOO *Number of Operations*
NOIS *Number of Import Statements*

wie auch die allgemeinen Komplexitätsmetriken
AC *Attribut Complexity*

- Kohäsion

  Kohäsionsmetriken sind
  MSOO *Maximum Size of Operation*
  MNOP *Maximum Number of Parameters*

  wie auch die Kohäsionsmetriken
  LOCOM *Lack of Cohesion of Methods*
  CC *Cyclomatic Complexity*

- Kopplung

  Metriken zur Ermittlung von stark gekoppelten Komponenten sind
  CBO *Coupling Between Objects*
  CDBC *Change Dependency Between Classes*
  DAC *Data Abstraction Coupling*
  MIC *Method Invocation Coupling*

  wie auch Metriken zur Prüfung der Kapselung
  AHF *Attribut Hiding Factor*
  MHF *Method Hiding Factor*
  PPrivM *Percentage of Private Members*
  PProtM *Percentage of Protected Members*
  PPubM *Percentage of Public Members*

4.2 Metriken zur Unterstützung eines Refactorings

- Duplizierter Code

  Aus dem Spektrum der oben beschriebenen Metriken gibt es keine, die in dieser Sache weiterhilft. Analysewerkzeuge wie Sotograph oder PMD sind in der Lage, durch direkten Codevergleich betroffene Stellen aufzufinden.

- Lange Methoden

  Lange Methoden lassen sich finden mit
  MSOO *Maximum Size of Operation*)

[26] stinkenden

MNOL *Maximum Number of Levels*
CC *Cyclomatic Complexity*

- Große Klassen

  Große Klassen ergeben sich aus
  LOC *Lines of Code*
  NOA *Number of Attributes*
  NOO *Number of Operations*
  AC *Attribut Complexit*

- Lange Parameterlisten

  MNOP *Maximum Number of Parameters*

- Divergierende Änderungen

  Divergierende Änderungen bezogen auf eine Klasse sind dann zu erwarten, wenn diese Klasse viele andere per Objektinstanzen benötigt. D.h. immer dann, wenn eine der referenzierten Klassen geändert wird, dann muss die betrachtete Klasse auch geändert werden. Dieser Sachverhalt lässt sich erkennen mit den Kopplungs-Metriken

  CBO *Coupling Between Objects*
  DAC *Data Abstraction Coupling*
  FO *Fan Out*

- „Schrotkugeln“

  Der Schrotkugeleffekt ergibt sich dann, wenn eine Klasse von vielen anderen Klassen benutzt wird. Dies ist mit Toghether-Metriken dadurch zu fassen, dass geprüft wird, wie eine Klasse von aussen benutzt werden kann, in anderen Worten, wie die Verantwortung der Klasse gegenüber der Außenwelt ist,

  RFC *Response for Class*

  Auch umfangreiche Vererbungsstrukturen können schädlich in diesem Sinne sein,
  DOIH *Depth of Inheritance Hierarchy*
  NOCC *Number of Child Classes*

  Weitere Metriken sind
  PPubM *Percentage of Public Members*

- „Neid“

  Hierfür ist mir keine Metrik bekannt.

- Datenklumpen

  Man kann Datenklumpen nicht direkt mit Metriken erkennen, mit den Kohäsionsmetriken

LCOM *Lack of Cohesion of Methods*

kann man jedoch Klassen finden, die Datenklumpen in ihren Methoden verwenden.

- Neigung zu elementaren Typen

  Mit der Metrik

  AC *Attribut Complexity*

  werden Klassen dahingehend untersucht, wie groß die Strukturen der Attribute der Klassen sind. Grundsätzlich ist ein hoher wert ein Indiz für eine große Komplexität. Ein geringer Wert kann jedoch ein Hinweis sein auf diese Neigung.

- *Switch*-Befehle

  Man kann nicht mit Metriken nach *switch*-Konstrukten suchen – dafür gibt es andere einfachere Techniken -; Metriken wie

  CC *Cyclomatic Complexity*
  MSOO *Maximum Size of Operation*

  geben jedoch grundsätzlich Hinweise auf die Schachtelungs-Komplexität und -Tiefe von Methoden.

- Parallele Vererbungshierarchien

  Parallele Vererbungshierarchien sind mit den oben beschriebene Metriken kaum zu fassen; Sotograph kann da weiter helfen.

- Faule Klassen

  Faule Klassen sind Klassen mit einem geringen „Verantwortungsbewusstsein“,

  RFC *Response for Class*

- Spekulative Allgemeinheit

  Hieraus resultieren Methoden oder Klassen, die (noch) nicht benötigt werden. In Together kann man das auf der Ebene von Klassen im Diagramm qualitativ erkennen; zudem ist eine weitere Analyse mittels „*Search for Usages“* erforderlich (denn es kann Benutzungsreferenzen geben, die nicht UML-konform sind und somit von Together nicht unmittelbar erkannt werden). Sotograph kann in diesem Kontext mit den Metriken zur Suche nach Benutzung von *public* Methoden schneller zum Erfolg kommen (*class method not used*)[27]; eine Klasse wird dann nicht benutzt, wenn all ihre *public* Methoden nirgends benutzt werden.

- Temporäre Felder

  Hierfür ist mir keine Metrik bekannt.

[27] Die Sotograph-Metriken sind in diesem Dokument nicht erläutert; hier sei auf die Sotograph-Dokumentation verwiesen (KnowledgeBase>Werkzeuge>Sotograph>Metriken>Query and Metrics).

- Nachrichtenketten

  Hierfür ist mir keine Metrik bekannt.

- Vermittler

  Hierfür ist mir keine Metrik bekannt.

- Unangebrachte Intimität

  Hierfür ist mir keine Metrik bekannt. Suche z.B. nach *super*-Verwendungen.

- Alternative Klassen mit verschiedenen Schnittstellen

  Hierfür ist mir keine Metrik bekannt.

- Unvollständige Bibliotheksklassen

  Hierfür ist mir keine Metrik bekannt.

- Datenklassen

  In Together erkennt man diese Klassen als Beans (JavaBeans, EJB) in der UML-Darstellung. Diese sind qualitativ weiter zu untersuchen. Eine Metrik ist mir nicht bekannt.

- Ausgeschlagenes Erbe

  Metriken zur Unterstützung dieser Analyse liegen im Bereich der Vererbungskonstrukte, also überall dort, wo Vererbung vorliegt, kann nach diesen Auffälligkeiten gesucht werden,

  MIF *Method Inheritance Factoyr*
  DOIH *Depth of Inheritance Hierarch*

  Diese Metriken ergeben keinen direkten Hinweis auf ausgeschlagenes Erbe, sondern indirekt eine Position, wo weiter analysiert werden kann.

- Kommentare

  Kommentare, die auf ToDos hinweisen und keine Aussagen zur Sache machen, können nicht mit Metriken ermittelt werden, sondern mit Audits (dazu später).

5 Schlussfolgerung

Die Bewertung einer Architektur mit Metriken und Audits ergibt einen recht guten Eindruck von der Pflegbarkeit und Änderbarkeit der Applikation. Mit Metriken auf der Basis des Programmcodes erhält man Hinweise, die man auf der Design-Ebene mit den entsprechenden Modellen weiter verfolgen und untersuchen kann, z.B. bei Hinweisen auf große Komplexität und hohe Kopplung sollte man den Sachverhalt in den Klassendiagrammen und Komponentendiagrammen überprüfen, Hinweise auf eine fragwürdige (geringe) Kohäsion lässt sich z.B. in Sequenzdiagrammen nachvollziehen und prüfen.

Obige Merkmale und Kriterien stellen eine Auflistung von Aspekten vor, die nicht als Dogma angesehen werden sollten, sondern mehr als Richtlinien. Insgesamt dienen sie dem Zweck, nicht nur eine stabiles und funktionierendes System – das natürlich auch - sondern darüber hinaus auch ein besser pflegbares, änderbares und ausbaubares System zu entwickeln.

Man muss nicht krampfhaft versuchen, alle Design-Regeln zu erfüllen. Doch dort, wo man es nicht getan hat – oder nicht tun konnte –, sollte zumindest mit einer umfassenden Begründung dokumentiert werden, warum man die Richtlinien an der Stelle verlassen hat. Gerade dies ist wichtig für eine bessere Pflegbarkeit und Änderbarkeit des Systems.

Noch ein Wort zu den Design-Werkzeugen; Design-Werkzeuge unterstützen i.A. das Malen von UML. Das allein kann aber nicht alles sein. Design-Werkzeuge sollten auch bei der Modellierung und Entwicklung von Anwendungsarchitekturen unterstützen in der Art, dass beim Entwicklungsprozess ständig eine Überprüfung obiger Kriterien ermöglicht wird, entweder laufend interaktiv oder durch explizite Anwendung spezieller Funktionen. Together bietet hierfür z.B. ein recht umfassendes Spektrum an Metriken und Audits[28].

Werkzeuge: Together, Sotograph, Headway, Eclipse/PMD

[28] Diese Leistungen erkauft man sich i.A. zu einem recht hohen Preis. Aber „Malwerkzeuge“ aus dem Umfeld der Office-Produkte (VISIO, Powerpoint) sind in diesem Kontext völlig indiskutabel; und auch OpenSource-UML-Tools hatten gerade unter diesem Aspekt Schwächen.

6 Literatur

Basili; Caldiera; et al.. The Goal Question Metric Approach. Encyclopedia of Software Engineering, John Wiley & Sons. 1994

Beck; Brandt; Fowler; Opdyke; Roberts: Refactoring: Improving the Design of Existing Code; Addison-Wesley, 1999

Boehm, B.; Software Engineering Ergonomics, Prentice Hall, 1981

Boehm; Brown; et al.: Characteristics of Software Quality. TRW Series of Software Technology, Volume 1, North-Holland Publishing Company, 1978.

Booch, Grady: Object-Oriented Analysis and Design; Addison-Wesley, 1993

Coad; Yourdon:Object Oriented Analysis and Design; Prentice Hall, 1991

Dahl; Dijkstra; Hoare: Structured Programming; Academic Press, 1972

De Champeaux, D.: Object-Oriented Development Process and Metrics, Prentice Hall, 1997

Fenton, N. E.: Software metrics: A Rigorous Approach, Chapman and Hall, 1991

Gilb, T.: Principles of Software Enineering Management; Addison-Wesley, 1999

Gilb, T.: Software metrics. Cambridge, Massachusetts, Winthrop Publishers. 1977

Henderson-Sellers, B.: Object-oriented metrics: Measures of complexity, Prentice Hall, 1996

Lippert; Roock: Refactorings in großen Software-Projekten; dPunkt Verlag, 2003

Lorenz, M. und Kidd, J.: Object-oriented software metrics; Prentice Hall, 1994

DeMarco, T.: Structured Analysis And System Specification; Yourdon Press, 1978

Follet, Ken; Die Säulen der Erde

Hubert, Richard: Convergent Architecture – Building Model-Driven J2EE Systems with UML; Wiley&Sons, 2001

Knuth, Donald; The Art of Computer Programming, Vol 1 - 4; Addison-Wesley, 2006

Meyer, Bertrand; Objektorientierte Software-Entwicklung; Hanser-Verlag, 1990

Reißing, R.: Bewertung der Qualität Objekt-orientierter Entwürfe; Dissertation, Universität Stuttgart, 2002

Rumbaugh; Blaha; Premerlani; Eddy; Lorensen: Objektorientiertes Modellieren und Entwerfen; Hanser-Verlag, 1991

Warmer; Kleppe: Object Constraint Language 2.0; Addison-Wesley, 2003

Sotograph; www.software-tomography.de

Together; www.borland.com/together