Architekturerkennung aus Quellcode?
- Möglichkeiten und Grenzen anhand einer Fallstudie -

Meik Teßmer, Thorsten Spitta

Technische Fakultät, Fakultät für Wirtschaftswissenschaften
Universität Bielefeld
33615 Bielefeld
mtessmer@users.sourceforge.net
thSpitta@wiwi.uni-bielefeld.de

Abstract. The source code is the only really true structure in a software system's life cycle, let it be excellent or poor. Its structure-in-the-large, the architecture, is its most important technical artefact. Programming languages have no syntactical elements to express architectural structures like components or connections. Therefore architectural recovery is one of the most interesting topics in the reengineering area. The source code is parsed to find different views of the architecture in order to judge its maintainability. That's why architectural recovery is a very important topic also for software management purposes. We investigated a large component of a 1 million lines C++ system, which is in operation since 1,5 years. We show what can be detected in the code and how it can be done under real life conditions.

Einleitung

Es ist heute nicht mehr umstritten, dass die Architektur für die langfristige Wartbarkeit von Softwaresystemen ein entscheidendes Qualitätsmerkmal ist, in vielen Fällen wichtiger als die benutzte Programmiersprache und -umgebung [SG95; BCK98; Sz99; So01].

Unter *Softwarearchitektur*, oft auch nur *Architektur*, versteht man die Strukturierung eines Softwaresystems in voneinander unabhängig benutzbare *Komponenten* und deren *Interaktionen* [BCK98, p.23]. Dabei muss man mindestens ein statisches Strukturmodell (*enthält-Beziehungen*), ein dynamisches Prozessmodell (*benutzt- oder reagiert-auf-Beziehungen*) und ein Schnittstellenmodell (*interfaces*; Leistungen mit Import/Export) erstellen [So01, ch.10]. Es kann, diesen Sichten entsprechend, nicht *die* einheitliche Architekturbeschreibung geben (vgl. auch [Wr02, 11ff]).

Bis in die 90er Jahre tauchte der Entwicklungsschritt "Entwurf einer Architektur" nicht einmal in international verbreiteten Lehrbüchern auf (vgl. z.B. [Som89]). Dem entsprechend wurde die sich schnell verbreitende Lehre von der Objektorientierung [Bo91] vor allem von Laien und Praktikern dankbar auch als "Lösung" des Architekturproblems aufgegriffen. Unter anderem wurde behauptet, man würde bei Benutzung einer objektorientierten Sprache wie Eiffel oder C++ eine pflegbare Architektur erhalten (vgl. z.B. [Me88, ch.4; Sto98, 746]). Diese Behauptungen wurden in Frage gestellt [SG95; BS02], u.a. mit dem Argument, die Klassenstruktur von C++, C# oder Java sei viel zu kleingranular, um einen strukturellen Überblick über ein Softwaresystem zu ermöglichen, und Werkzeuge wie JavaDoc würden diese Unübersichtlichkeit nur zeigen und keine relevanten Strukturen offen legen.

Bereits Shaw und Garlan als Pioniere einer Lehre expliziter, sprachgestützter Softwarearchitektur vermissten Konstrukte in den Programmiersprachen, die die für Komponenten notwendigen Abstraktionen ausdrücken können [SG95, ch.7]. Die in Folge dieses Mangels entstandenen Architektur-Beschreibungssprachen und -Werkzeuge werden offenbar nicht benutzt oder sind nicht praktisch einsetzbar. Dies ergab eine Untersuchung von Wrede [Wr02]. Man muss sich also mit dem Quellcode und mehr oder weniger informellen Architekturbeschreibungen behelfen. Grob zusammengefasst haben wir zum Thema Architektur folgende Situation (vgl. [SG95; BCK98]):

- Expliziter Architekturentwurf wird entweder kaum gemacht oder kann sich mangels Sprachunterstützung nur auf grobe, mehrdeutige Grafiken stützen
- Objektorientierte Sprachen lösen das Architektur-Dilemma nicht, da sie nur den Entwurf im Kleinen unterstützen
- Auch wenn keine explizite Architektur existiert, gibt es immer eine Implementierung mit einer faktischen, vielleicht mangelhaften Architektur, die man aber nicht sehen kann.

Den letzten Aspekt wollten wir an Hand einer realen, nicht trivialen Implementierung untersuchen, und zwar mit der Reengineering-Methode *architectural recovery* [Ar93, ch. 12], indem der Quellcode auf direkte oder indirekt interpretierbare Sprachmerkmale hin untersucht wurde. Daraus ergibt sich, dass wir uns auf statische Quellcodeanalyse beschränkt haben, also auf die Elemente des Systems, die unmittelbarer Manipulationsgegenstand von Softwareevolution und -wartung sind.

Im folgenden Beitrag skizzieren wir kurz den Diskurs- und Einsatzbereich der untersuchten Software, die von uns untersuchten Architekturhinweise im Quellcode von C++-Programmen, die Anforderungen an einen Parser, den man für eine solche Untersuchung braucht und die Ergebnisse für den untersuchten Fall. Der Aufsatz schließt mit Empfehlungen für Architekturentwurf und -recovery für die Ebene des IT-Managements.

Es ist *nicht* Ziel des Beitrages, konstruktive Ansätze für die Erstellung guter Softwarearchitekturen zu vermitteln. Dies wäre zweifellos ein wichtiges Thema, aber eben ein anderes als das hier gestellte. Uns kommt es hier darauf an, für das Management aufzuzeigen, wie man mit dem scheinbar undurchdringlichen Thema Quellcode umgehen kann.

Die Fallbeschreibung

Das von uns analysierte System ist ein völlig neu entwickeltes Release einer MS-DOS-basierten Handwerkersoftware. Der Benutzer kann am Bildschirm Konstruktionen erstellen, das System druckt dazu Stücklisten, Materialbedarf und -verwendung aus, die einem Angebot oder der Materialbeschaffung zugrunde gelegt werden können. Release 1 war in C geschrieben und verwendete ein zugekauftes grafisches Kernsystem. Als Ausgabesoftware für Papier diente MS-Word 97. Release 1 war rund 2.000 mal installiert.

Die Neuentwicklung wurde 1998 begonnen, war für MS-Windows konzipiert, benutzte das COM-Framework von Microsoft und ein ebenfalls gekauftes, gegenüber Release 1 neues grafisches Paket. Word als Ausgabemedium wurde beibehalten. Implementiert wurde mit der Entwicklungsumgebung Visual C++ von Microsoft. Das System hat ggw. (Anfang 2004) ca. 1 Million selbst erstellte lines of code, die gewartet werden müssen. Allein wegen der Schnittstellen zu gekauften Komponenten ist dies ein recht komplexes System. Es muss dem Benutzer neben der Funktionalität auch gute Antwortzeiten bieten, um am Markt Akzeptanz zu finden.

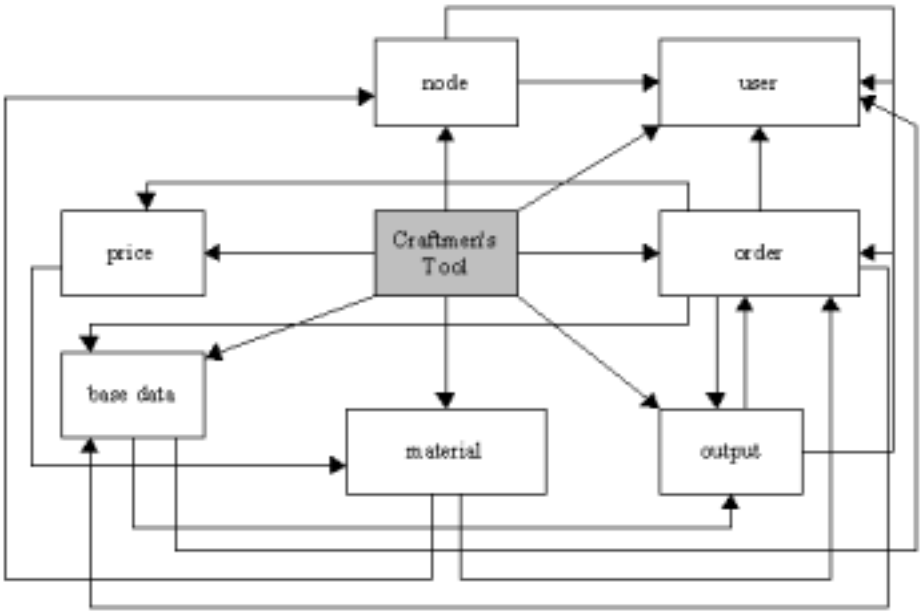

Abbildung 1: Grobarchitektur des Systems

Entwickelt wurde nach einer groben Architekturskizze und auf Basis von Style Guidelines, die jegliche Namengebung im Quellcode genau regelten. Die Entwicklungs- und Wartungsumgebung basiert auf einer die Grobarchitektur wiedergebenden Verzeichnisstruktur, den "Projekten", als statischem Strukturierungsmittel. Ein Repository, das z. B. benutzt-Beziehungen offenlegt, existiert nicht. Es war also für die Qualitätssicherung bereits während der Entwicklung nicht möglich zu überprüfen, ob Vorschriften oder Pläne (u. a. die Grobarchitektur) auch tatsächlich implementiert wurden. Abbildung 1 zeigt die Grobarchitektur, angelehnt an die von uns vorgefundene Form. Wie häufig bei informellen Architekturskizzen, sind auch hier vor allem die Kanten des Graphen, also die Beziehungen zwischen den Komponenten, semantisch vage.

Da das System in einem sehr dynamischen Markt eingesetzt wird, muss es ständig evolutionär weiter entwickelt werden. Dies führt für die Kunden zu jährlich ca. vier Versionen. Da die Stammdaten z. T. Geometriedaten sind, die mit der Software auszuliefern sind, ist dies keineswegs trivial. Die Unternehmensleitung muss an einer gut wartbaren Architektur und deren Erhaltung über viele Versionen interessiert sein.

Für unsere Analyse wurde eine als besonders zentral geltende Komponente mit 26.303 lines of code (Kommentare enthalten) ausgewählt. Sie wird in Abschnitt 4 genauer beschrieben, nachdem die syntaktischen Möglichkeiten der Sprache C++ für unser Vorhaben diskutiert wurden.

Architekturhinweise im Quellcode

Architektur*elemente* sind Komponenten und Konnektoren, die allerdings keine Entsprechungen auf der Ebene gängiger Programmiersprachen haben [SG95, 156ff]. So findet man auch bei C++ keine syntaktischen Analoga zu Komponenten in der Sprache. Namensräume sind zwar ein notwendiges aber keineswegs hinreichendes Mittel, um eine Menge von Klassen als Komponente auszuweisen. Also suchen wir hilfsweise nach indirekten *Hinweisen*. Die größten syntaktischen Einheiten objektorientierter Sprachen sind Klassen, bei C++ noch zu ergänzen um Includes, die überwiegend in Form von Header-Dateien vorkommen. In der Regel werden Includes (Makros) als Hilfmittel zur Textersetzung für mehrfach verwendete Deklarationen benutzt [St98, 213]. Häufig enthalten sie neben den Schnittstellen auch Forward-Deklarationen von Klassen, um eine separate Kompilation zu ermöglichen. Da eine Komponente aus sehr vielen Klassen bestehen kann, gilt es diejenigen Klassen zu identifizieren, deren Methoden von außerhalb der vermuteten Komponente benutzt werden oder auf deren Ereignisse reagiert wird. Das wäre die *Exportschnittstelle* einer Komponente, also die von ihr angebotenen Leistungen. Im Fall einer explizit entworfenen Architektur wäre dies genau eine Klasse, die alle von der Komponente nach außen angebotenen Methoden enthält. Diese Klasse verwaltet *keine* Datenstruktur, sondern ein Artefakt, nämlich genau ein virtuelles Exemplar der Komponente, das als solches nirgendwo im Speicher mit Zuständen residiert. Eine solche Schnittstellenklasse taucht auch in den Entwurfsmustern von Gamma et al. auf und wird dort *Fassade* genannt [Ga95, 185ff.]. Sie ist eines der wichtigsten Muster komponentenorientierter Entwicklung, das sich aus der Lehre von der Objektorientierung aber nicht ohne Weiteres ergibt. Eine Fassade zeigt die von der Komponente exportierten Operationen in Form aufrufbarer Signaturen.

Umgekehrt benötigt man eine entsprechend klare *Importschnittstelle*, und zwar in Form des Entwurfsmusters *Adapter*, auch *Wrapper* genannt [Ga95, 139ff; SD00]. Dies sind dann mehrere Klassen, deren Aufgabe es ist, die Komponente von der Syntax importierter Leistungen anderer Komponenten abzuschirmen. Das wäre bei Benutzung eines Frameworks der Benutzeroberfläche besonders wichtig und ist zumindest als Abschirmung zur Datenzugriffs-Schicht u.W. weit verbreitet. Bei einem Call-Mechanismus ist die Importschnittstelle explizit im Code erkennbar, evtl. sind Namenskonventionen hilfreich, wenn auch keinesfalls zuverlässig. Ereignisorientierte Schnittstellen kann man an dem Token-Paar `try/catch` für die selbe Variable erkennen.

Wenn auch auf einer sehr kleingranularen Ebene, ist es mit Werkzeugen doch prinzipiell möglich, die wichtigsten Beziehungstypen zwischen konzeptionellen Komponenten syntaktisch zu erkennen. Man muss allerdings eine konzeptionelle Vorstellung davon haben, nach welchen Komponenten man sucht. Es ist sehr unwahrscheinlich, dass man durch Analyse einer Klassenstruktur Komponenten findet, von denen noch nicht einmal eine konzeptionelle Vorstellung existert[1]. Eine Codeanalyse kann jedoch das im Quellcode ausdrückbare statische Prozessmodell offen legen. Die wichtigsten Architekturhinweise, die nach der Codeanalyse interpretiert und aggregiert werden müssen[2], sind bei C++

- die Klassen, insbesondere deren Schnittstellen
- die Klassenmethoden
- die Header-Dateien.

Es werden also durch Quellcodeanalyse offen gelegt:

- die **Aufruf-Sicht** (benutzt-Beziehungen über `includes/included by`)
- die **enthält-Sicht** (Vererbungs-Beziehungen über `public, proctected und private`)
- die **Reaktions-Sicht** (reagieren auf / weitergeben von Ereignissen).

Damit kann man die Export- und die Importschnittstelle einer Komponente offen legen und die Frage beantworten, ob die vermutete Komponente wirklich eine solche ist, d.h. ob sie selbstständig bzw. in genau vorhersagbarer Weise benutzen lässt. Auch andere Fragen aus der Pflege von Software, etwa das inkrementelle Kompilieren, hängen stark von der tatsächlichen Komponentenstruktur ab. Eine solche Analyse liefert als "Beiprodukt" auch Erkenntnisse über die Beziehungen innerhalb der Komponenten, also zwischen *allen* Klassen und deren Methoden.

Hat man alle Komponenten analysiert und die Einzelanalysen aggregiert, ist die Architektur einer Software transparent und damit aus Management-Sicht beherrschbar. Für Reengineering-Maßnahmen, aber auch für eine evolutionäre Weiterentwicklung, die nicht zur Erosion der Struktur führen soll, ist das Wissen aus einer solchen Analyse unverzichtbar.

Auf weitere Sprachelemente von C++, die ebenfalls Hinweise auf eine Architektur liefern, wollen wir erst in Abschnitt 5 im Zusammenhang mit den Ergebnissen der Fallstudie eingehen (Beispiel: virtuelle Klassen als Schnittstellen). Zunächst wird jedoch das Vorgehen beim *Architectural Recovery* dargestellt.

---

[1] Dies käme einem „Code-Mining“ in Analogie zum sog. „Data Mining“ nahe, wozu wir uns nicht versteigen möchten.
[2] Dies muss bei den uns bekannten Werkzeugen noch manuell erfolgen.

Vorgehen bei einer Architektur-Recherche

Da die Architektur über eine statische Quellcodeanalyse erfolgt, die weit mehr offen legt als für das Erkennen einer Architektur notwendig ist, müssen zwei Phasen der Recherche unterschieden werden:

(1) Komponentensuche

(2) Klassenanalyse.

In Phase (1) geht es nur um Architekturerkennung, während in Phase (2) eine umfassende Codeanalyse notwendig ist, die auch für allgemeine Aufgaben des Reengineering genutzt werden sollte, vor allem für die Qualitätssicherung.

Phase (1) sucht nach einer expliziten Komponentenstruktur. Von dieser kann man bei C++ nur sprechen, wenn man Namensräume findet und zusätzlich klar erkennbare Klassen für die Exportschnittstelle und entsprechende Importschnittstellen, wie in Abschnitt 3 beschrieben. Bei positivem Ergebnis ist die Aufgabe mit Schritt (1) erledigt. Dieser Fall liegt jedoch nur sehr selten vor, wie in jüngster Zeit Lanza nach Analyse mehrerer industrieller Systeme festgestellt hat [La03].

Hat man keine klare Struktur gefunden, muss sich Phase (2) anschließen, die alle im Quellcode erkennbaren Strukturen aufzeigen muss. Die Klassenanalyse umfasst die benutzt-Beziehungen, die enthält-Sicht und die Reaktions-Sicht zwischen allen Klassen. Es werden dabei *Cluster* oder *Muster* gesucht, aus denen man auf eine mehr oder weniger ausgeprägte Architektur schließen kann. Von Mustern (Bilder) als Alternative zu Clustern (Zahlen) sprechen wir deshalb, weil ein visuelles Erkennen bei der Strukturanalyse eine nicht unwesentliche Rolle spielt (vgl. [KaCa98; La03; Di03]. Die Ergebnisse aus Phase (2) gruppieren sich in zwei Fälle:

1. es wird eine brauchbare Clusterung gefunden, aus der man durch Reengineering des Codes eine Komponentenstruktur erstellen kann

2. es wird eine "Spaghetti-Struktur" gefunden, die bei objektorietierten Sprachen inzwischen auch „Ravioli-Struktur“ genannt wird. Man kann durch Reparatur keine Struktur mehr herstellen, höchstens noch an besonders kritischem Code ein wenig "reparieren".

Auch das zweite Ergebnis ist für das Management sehr wichtig. Die Führung weiß nach einem solchen Analyseergebnis, dass sie sich mittelfristig um eine völlige Erneuerung der Software kümmern muss, weil eine Weiterentwicklung des bestehenden Codes bei ständig steigenden Änderungskosten immer schlechtere Leistungen und immer mehr Fehler hervorbringen wird.

Die folgenden Ergebnisse unserer Fallstudie demonstrieren die Vorgehensweise und machen deutlich, wie wichtig eine Quellcodeanalyse gerade auch für das Management ist, das häufig den Programmcode einer Software nicht einmal in Auszügen lesen kann. Aber auch fachlich versierte Softwaremanager sollten wissen, dass *niemand* eine Chance hat, einige hunderttausend Lines of Code noch zu überblicken. Dies wird durch einige unserer Beispiele wohl sehr deutlich.

Parsen von Programmen und Maße zur Softwarequalität

Quellcodeanalyse in einem kommerziellen Umfeld heißt, dass die Programmbasis mit einem auf die Zielsprache ausgerichteten Parser zunächst automatisch untersucht wird. Der Parser arbeitet im ersten Schritt nicht interaktiv, denn er muss alle gefundenen Referenzen in einer Datenbasis ablegen. Diese müssen für die interaktive Arbeit zunächst atomar gespeichert werden, sind aber danach zu Kennzahlen zu verdichten. In der Informatik werden Maße und Kennzahlen gerne *Metriken* genannt. Wenn verdichtete Daten gespeichert werden sollen, muss man den Parser mit dem Compiler verknüpfen und betroffene Kennzahlen bei jeder Programmänderung neu berechnen.

Während es für C und auch für Java offenbar recht viele professionelle Werkzeuge gibt [Ho97], konnten wir für C++ nur wenige finden[3]. In die engere Wahl gezogen wurden *Parasoft* [Pa03], *Analyzer* [CC03] und *Understand C++* [Un03]. Parasoft wurde wegen unakzeptabler Testbedingungen, verbunden mit einem extrem hohen Preis, im Vorfeld ausgeschlossen. Nach einem Vergleich zwischen den beiden verbliebenen Produkten wurde bei in etwa gleichen Leistungen des Parsers das letztgenannte Produkt ausgewählt. Bild 2 zeigt den Screenshot einer interaktiven Beispielanalyse, die eine Klasse, den Quellcode der Header-Datei und die von dieser Basisklasse abgleiteten Klassen offen legt.

Ein Quellcodeparser muss alle in Abschnitt 3 genannten Sichten je Objekttyp ermitteln, die sich automatisch aus dem ausführbarem Code erkennen lassen. Dazu liefern beide Werkzeuge eine Fülle von Kennzahlen, die aus den gefunden atomaren Sprachelementen berechnet werden. Als wichtige Beispiele für die Architekturerkennung oder -beurteilung seien hier erwähnt:

- Anzahl Export- und Importschnittstellen aus Sicht einer Komponente
- Anzahl Methoden je Klasse
- Anzahl nicht geerbter Methoden in abgleiteten Klassen
- Anzahl privater und öffentlicher Methoden
- Kohäsion von Klassen (*wie viele Methoden benutzen die Instanzvariablen?*)
- Anzahl Module, die geklonten (= kopierten) Code enthalten.

---

[3] Die in vielen CASE-Werkzeugen enthaltenen Parser schieden alle in einer Voruntersuchung aus, die meisten wegen mangelnder Leistung.

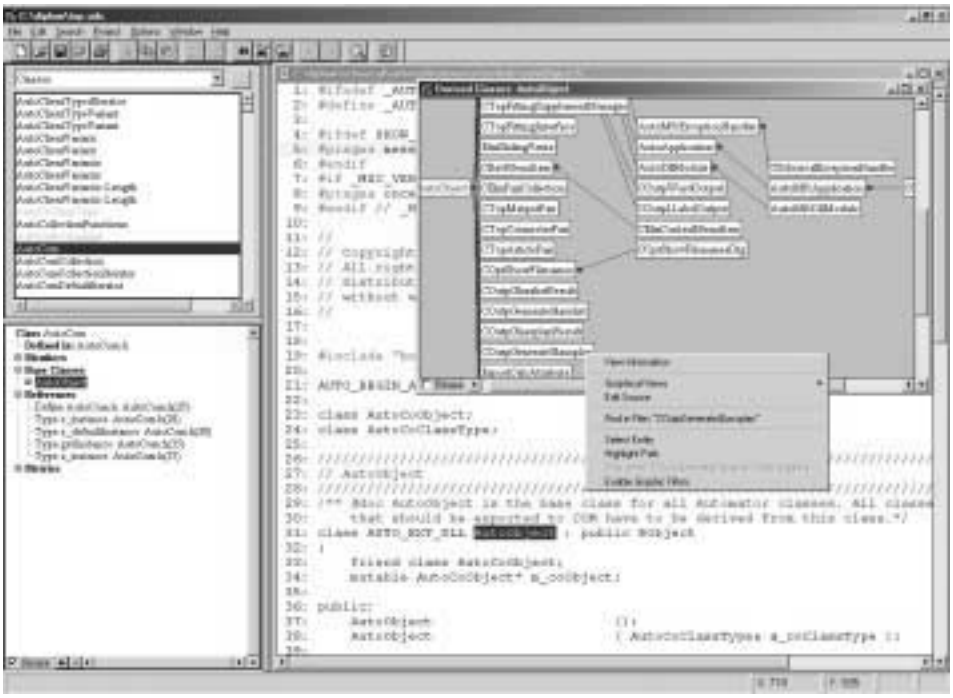

Abbildung 2: Screenshot einer interaktiven Analyse mit Understand C++

Zu den messtheoretischen Grundlagen sowie der Darstellung und Ermittlung aller allgemeinen Maße hat Zuse ein umfassendes Standardwerk veröffentlicht [Zu98], zu den objektorientierten Maßen sei auf Sneed/Winter verwiesen [SW02]. Dieser Blick auf den Stand der Wissenschaft erscheint wichtig, weil in den Werkzeugen zum Teil falsche oder irreführende Größen implementiert sind, wichtige Kennzahlen aber fehlen. Ganz besonders die populären (arithmetischen) Mittelwerte führen leicht zur Desinformation des Managements. Bei der Analyse größerer Quellcodebestände entstehen fast immer stark schiefe und nicht etwa symmetrische Verteilungen. In diesen Fällen wären Konzentrations- und Streuungsmaße aussagekräftiger. Diese sind leider in keinem uns bekannten Werkzeug implementiert. Wir haben sogar Mittelwerte qualitativer, völlig unterschiedlich definierter Maße gefunden, was ganz einfach falsch ist [SS01, Kap. 6][4].

Für die reine Architekturermittlung benötigt man viele Maße, die die Parser liefern, zunächst nicht. Wenn man allerdings in Phase (2) einer Analyse eintreten muss, ist deren Nutzung zur Beurteilung der Codequalität unbedingt erforderlich. Im folgenden Abschnitt zeigen wir Ergebnisse aus der von uns durchgeführten Analyse.

[4] Getestet wurde dort eine Freeware zur Analyse von Java-Programmen, die aus einem C++ Analyzer entstanden war.

Analyse einer vermuteten Komponente

Da nicht mit Namensräumen gearbeitet worden war, schied Phase (1) für das System von vornherein aus. Um den erwarteten Arbeitsaufwand in Grenzen zu halten, aber auch um einen vertieften Fokus für die Untersuchung zu bekommen, wurde ein als besonders schwierig geltender Baustein ausgewählt. Er ist im System sehr zentral verankert, wurde intern als "Modul" mit dem Namen `top` bezeichnet, hatte aber ganz offensichtlich allein aufgrund der Größe von 26.303 lines of code in 185 Dateien und 207 Klassen zumindest quantitative Komponenteneigenschaften. Analysiert wurden:

- die **Aufruf-Sicht**, um Export- und Importschnittstellen zu suchen, also zu prüfen, ob es sich um eine Komponente handelt
- die **Enthält-Sicht**, um die Klassenstruktur, insbesondere die Vererbungsbeziehungen sichtbar zu machen.

Die **Reaktions-Sicht** konnte zum Zeitpunkt der Untersuchung (Okt. 2003) mit *Understand C++* noch nicht analysiert werden. Dies ist aber seit Anfang 2004 möglich.

Nach diesen der Architekturermittlung dienenden Schritten wurde eine vertiefte Analyse der Innenstruktur des "Moduls" durchgeführt. Es folgen die wichtigsten Ergebnisse der Detailanalysen.

Export- und Importschnittstellen

Durch die Analyse *includeby* / *includes* wurde eine zwar nicht ideale, aber noch akzeptable Exportschnittstelle gefunden. Von den insgesamt 94 Header-Dateien mit zusammen 42 Importen aus anderen Komponenten tragen 16 zum Export von `top` bei, davon 12 Header-Dateien nur einmal. Die Exportschnittstelle konzentriert sich also im Wesentlichen auf 4 Header-Dateien. Bild 3 zeigt die Verteilung. Es erscheint möglich, die Exportschnittstelle auf eine Fassadenklasse zu konzentrieren. Das "Modul" ist also aus Sicht der übrigen Komponenten durchaus eine Komponente, die mit überschaubarem Aufwand verbessert werden könnte.

Die Importschnittstellen zeigten ein ganz anderes Bild. Es wurden insgesamt 413 Importe von Dateien aus anderen Modulen gefunden. Die beiden umfangreichsten Importe zweier Programmdateien umfassen 31 und 27 Includes. Eine Detailanalyse deckte auf, dass nicht nur auf das verwendete Framework (Microsoft COM), sondern auch auf die Datenbasis direkt zugegriffen wird, dass also ein Schichtenmodell zwar Konzept aber keinesfalls Realität ist. Dies wäre eine umfangreiche Restrukturierung, die aber dringend geboten erscheint, soll die Software nicht vollständig von der Releasepolitik des Lieferanten abhängig bleiben.

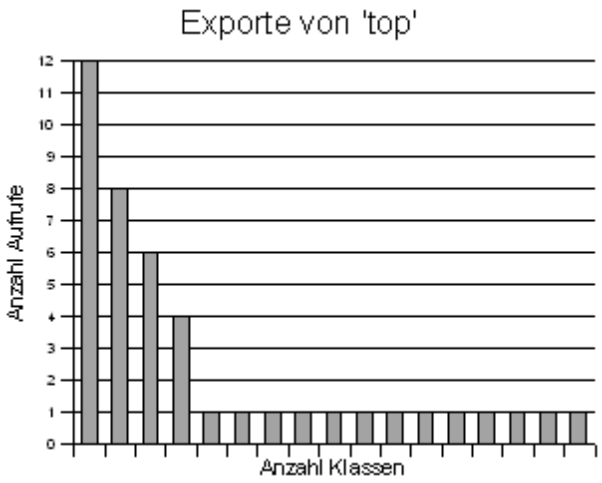

Abbildung 3: Verteilung der Exportschnittstelle auf Klassen

Die Enthält-Sicht (Klassenanalyse)

Aus Sicht der Architektur kann es sinnvoll sein, dass Klassen von Klassen außerhalb einer Komponente abgeleitet sind. Dies gilt allerdings nur genau dann, wenn die Vater-Klassen sog. *Basisklassen* sind, die Standardoperationen vorgeben oder globale Daten verwalten. Diese können abstrakte oder reale Klassen sein. Hier lässt sich das `top`-Modul recht gut abgrenzen. 207 von 219 Klassen werden innerhalb des Moduls definiert. Ob die 12 externen Vater-Klassen wirklich Basisklassen sind, kann leicht überprüft werden. Die Abgrenzung der Klassen nach außen ist also überschaubar.

Mit der Klassenanalyse lässt sich auch der interne Aufbau eines Bausteins zeigen. Fragen daran wären etwa: *Wie gut ist die Vererbungsstruktur?* oder: *Welche internen Basisklassen der Komponente gibt es?* Bild 4 zeigt die interne Vererbungshierarchie grafisch. Man sieht deutlich die drei internen Basisklassen und deren Benutzung als Vater-Klassen.

Die Klassenanalyse kann aber auch Beziehungen zwischen Klassen und ihren Methoden aufdecken. Damit ist eine Beurteilung der Qualität der Vererbungsbeziehungen möglich. Unter Anderem lässt sich eine Kennzahl summarisch und je Klasse ausweisen:

Anzahl geerbter Methoden = alle Methoden - lokale Methoden.

Je größer diese Zahl im Rahmen der Gesamtzahl der Methoden ist, desto besser ist die Modellierung mit Hilfe der Vererbung. Hier ist allerdings weniger der Mittelwert von Belang, sondern vielmehr das Aufspüren derjenigen Klassen, für die dieser Wert = 0 ist. Dies war bei 12,5% der Klassen der Fall.

Auch die Anzahl privater im Verhältnis zu öffentlichen Methoden gibt Qualitätshinweise. Ein geringer Anteil an privaten Methoden lässt auf Verletzungen des Geheimnisprinzips schließen. Dieses war im vorliegenden Fall nicht gerade dominierend. Nur 1,5% der Methoden waren privat. In der diesem Aufsatz zu Grunde liegenden Arbeit ist von "*Spaghetti-Code öffentlicher Methoden*" die Rede [Te04, 91].

Auch die Kohäsion, ein altes Prinzip der Softwaretechnik [YC76], kann gemessen werden. *Understand C++* weist deutlich mehr als die Hälfte der Klassen (n = 25) mit einem *lack of cohesion* von über 50% aus, 93 (also 45%) der Klassen haben sogar einen Wert von über 80%, acht Klassen gar keine Kohäsion, also 100%. Die Methoden dieser Klassen könnten irgendwo deklariert sein. Der Sinn solcher Klassen ist unklar, denn die Klassenmethoden benutzen die Instanzvariablen der Klassen überhaupt nicht. Auch hierzu ein Zitat aus der Diplomarbeit: "*Spätestens die Ergebnisse der Kohäsion erwecken den Eindruck, dass zwar in C++ programmiert, aber in C 'gedacht' wurde*."

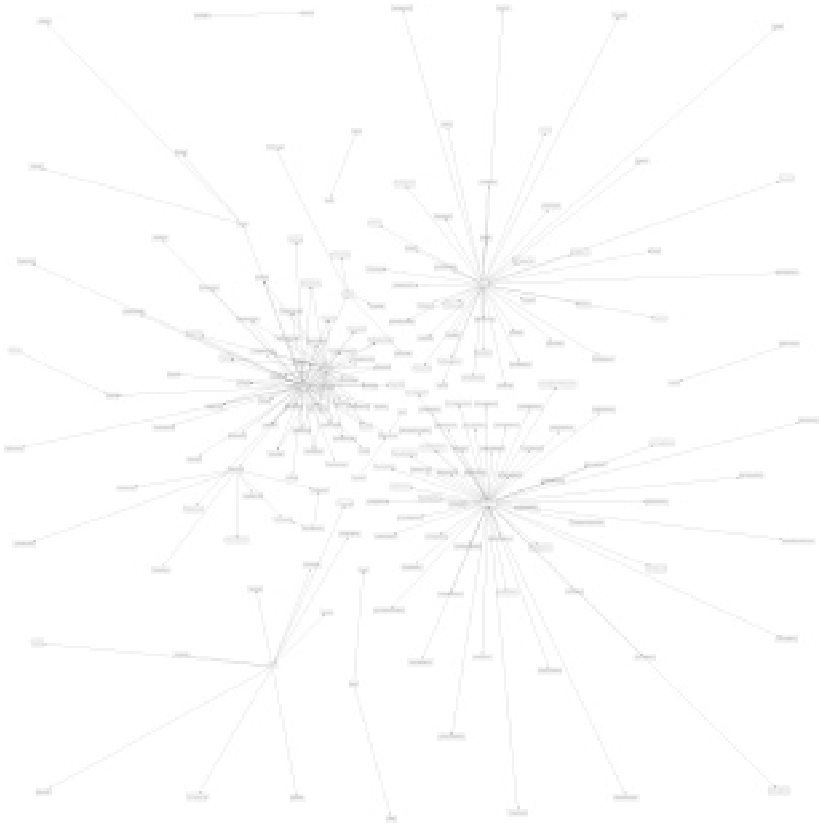

Abbildung 4: Struktur der Vererbungsbeziehungen (erzeugt mit Graphviz)[5]

Die Innen-Sicht des Moduls

Nach der Analyse der Import-Schnittstelle wurden auch die benutzt-Beziehungen innerhalb des Bausteins untersucht. Dies war wiederum nur möglich durch Visualisierung "per Hand" mit einem Spezialwerkzeug. Bild 5 zeigt die internen Include-Beziehungen.

[5] **Hinweis** für den zunächst nur „blätternden“ Leser: Es ist Konzept – siehe Text – dass man zunächst nur *Muster* und keine Details erkennen kann.

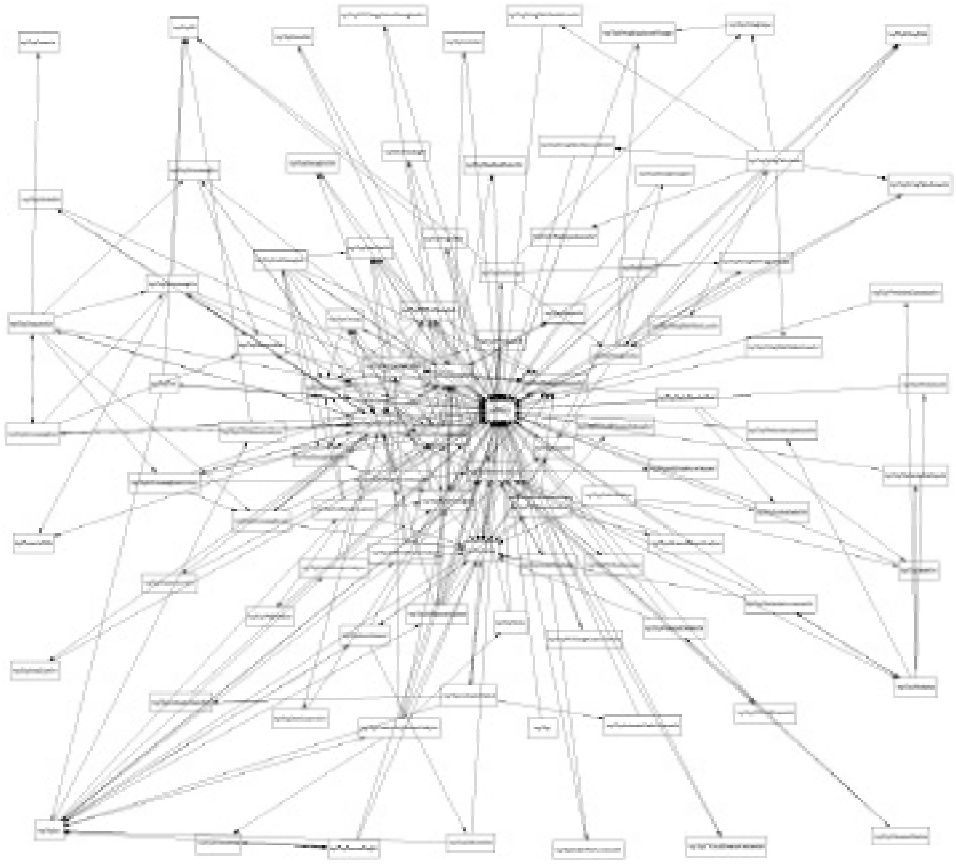

Abbildung 5: Interne Include-Beziehungen (erzeugt mit Graphviz)

Man sieht auf den *ersten* Blick, dass genau eine Klasse zentral für das Modul ist und von fast allen anderen benutzt wird. Auf den *zweiten* Blick lassen sich noch einige weitere, recht zentrale Klassen ausmachen. Der *dritte* Blick wird sicher erst durch eine Detailanalyse schärfer und bestätigt das anfänglich nur vermutete Urteil: Die interne Struktur des Moduls zeigt als negatives Qualitätsmerkmal eine geringe Kohäsion, m.a.W. eine *„Ravioli-Struktur“*. Hätte das Modul eine „saubere“ Struktur, würde man deutliche Cluster auch rein optisch erkennen.

Die Bilder 4 und 5 sollen neben den Ergebnissen des untersuchten Falles auch folgende Tendenz im Forschungsbebiet *Architekturerkennung* demonstrieren: Das visuelle Erkennen von Mustern spielt eine zunehmende Rolle [La03; Di03]. Anhand von Mustern kann man zielgerichteter in eine Detailanalyse "einsteigen" als mittels Seiten langer Zahlenkolonnen. Man kann auch dem Management ein Problem "zeigen", ohne sich der Wolken aus Powerpoint zu bedienen, die eben nur das zeigen, was das Bild "Wolke" meint: Nebel. Das wissen Statistiker schon sehr lange.

Folgerungen für das Management

Selbst nur beim Blättern in diesem Beitrag werden Manager sich fragen:

Was haben wir mit so etwas Kryptischem wie C++ Programmcode zu tun?

Spätestens die Bilder 4 und 5 senden vor allem einem des Programmierens unkundigen Manager zunächst nur das Signal: *Kapituliere!*

Deshalb sollen zum Abschluss Folgerungen gezogen und Ratschläge gegeben werden, die auch für Manager deutlich machen, was sie denn tun oder fordern sollten und welche Rolle in diesem Kontext die genannten Bilder spielen.

Mit „Management" meinen wir sowohl in den Techniken bewanderte als auch allgemeine Manager, die Details der benutzten Technik nicht oder nicht mehr nachvollziehen können. Auch fachlich ausgebildete Manager entwickeln sich schnell von den Basistechniken weg oder können zumindest die Zeit nicht mehr aufbringen, sich auf die Detailebene zu begeben. Bei Quellcode gibt es aufgrund des Volumens der erzeugten Ergebnisse schnell die Situation, dass die Entwickler noch *glauben*, alle ihre „Schöpfungen" zu durchblicken, dies aber in Wirklichkeit längst nicht mehr können.

Dies wissen wir seit dem Weg weisenden Aufsatz von Dijkstra [Di72], der eindringlich darauf hingewiesen hat, dass ab einer bestimmten Größe von Code die menschliche Perzeptionsfähigkeit völlig überfordert ist. Es gibt zwar eine gewisse Bandbreite zwischen den Fähigkeiten guter und schlechter Programmierer, die generelle Grenze aber wird nie verschwinden. Die Grenze ist auch davon abhängig, wie „grob-" oder „feinkörnig" die verwendete Programmiersprache ist. Als „grobkörnig" wären etwa COBOL, SQL und die sog. 4GL[6] zu bezeichnen, als „feinkörnig" die C-Sprachfamilie, zu der C++, Java oder C# gehören. Die Freiheitsgrade grobkörniger Sprachen sind geringer, natürlich auch ihre Möglichkeiten, maschinennahe Probleme überhaupt zu behandeln. Dafür sind aber auch die Risiken geringer, Fehler zu machen [BP86]. Wir kommen zu einem ersten Ergebnis für Manager:

Fazit 1: Sorgen Sie für qualifizierte Entscheidungen, mit welcher Sprache programmiert wird, falls überhaupt selbst entwickelt werden muss.

[6] 4GL = fourth generation languages. C++ und Java sind typische 3GL.

Wenn die menschlichen Fähigkeiten, komplexe formale Gebilde zu durchschauen und zu überblicken, so begrenzt sind, dann muss der Computer selbst der Intelligenzverstärker sein. Wenn ein Manager es wirklich will, lassen sich für *jede* Umgebung Werkzeuge finden oder behelfsmäßig herstellen, die maschinell Strukturen transparent machen und deren Veränderung gerade in einem dynamischen Umfeld (Markt, Kunden, Technologie) jederzeit offen legen (vgl. [Sp96]). Die Vorteile des Computers gegenüber unserem Gehirn liegen noch nicht einmal in der Schnelligkeit, sondern in der Monotoniefestigkeit. Die Ergebnisse des Computers können nach gründlichen Tests als richtig gelten, die Ergebnisse menschlicher Recherche sind höchst unsicher, bei Programmierern auch gerne einmal subjektiv. Die in unserer Fallstudie verwendeten Werkzeuge (Parser, Auswertungsprogramme) sind solche Intelligenzverstärker.

Fazit 2: Sorgen Sie für Werkzeuge, die den Quellcode Ihrer Mitarbeiter nach vorher gut kommunizierten Regeln durchsuchen und die Programmsstrukturen offen- und in einer Datenbank ablegen. Bei der Einführung sollte man mit wenigen, von allen Beteiligten verstandenen Regeln anfangen, diese generell durchsetzen und erst dann stufenweise erweitern. Die Werkzeuge müssen den Entwicklern in die Hand gegeben, aber auch bei der Abnahme für die Qualitätssicherung benutzt werden.

Die Sicht der von den Parsern erzeugten Ergebnisse in Form langer Zahlenkolonnen ist auf der Ebene des einzelnen Entwicklers noch überschaubar, auf übergeordneter allerdings nicht mehr. Der Parser schützt zwar den Entwickler davor, sich im „Firmament" der Programm-Tokens zu „verirren", zeigt aber von selbst keine klaren (oder eben auch unklaren!) „Sternbilder". Hier wird die gezeigte Visualisierung wichtig. Sie verdichtet ja Fakten und unterscheidet sich damit grundsätzlich vom Nebel der „Powerpoint-Wolke". Wir wissen aus einer anderen Untersuchung, dass die Ergebnisse des hier gezeigten Falles kein Einzelergebnis sind, sondern ein verbreitetes Phänomen. In dieser Untersuchung zum IT-Controlling mit rund 100 beteiligten Mittelständlern zeigte sich unter Anderem, dass die Mehrzahl der Firmen keinerlei Möglichkeiten hatten, ihren Quellcode zu durchschauen und sachgemäß weiter zu entwickeln [Sp98]. Deshalb sind schrittweise verfeinerbare Grafiken gerade für das Management besonders wichtig. Diese kann man interpretieren, ohne eine Zeile C++ lesen zu können. Grafiken wie die Bilder 4 und 5 sind Führungsinstrumente.

Fazit 3: Lassen Sie sich die Grafiken der Architektur regelmäßig vorlegen, in jedem Fall vor der Freigabe jeder neuen Version. Wenn Sie die Grafik des ersten Prototyps einer Neuentwicklung gesehen haben, werden Sie wissen, wie eine klare Architektur aussehen muss[7]. Bekommen Sie weiterhin gut erkennbare „Sternbilder", befindet sich die Entwicklung strukturell auf einem guten Weg. Driften die Bilder in Richtung „Milchstraße", müssen Sie frühzeitig nachhaken.

---

[7] Es wird als selbstverständlich angenommen, dass dieser erste Prototyp auf Basis einer klaren Architektur erstellt wurde, Näheres s. [Sp96] oder [Sp89, Kap. 11].

Architekturerkennung als notwendiges Mittel der Qualitätssicherung ist natürlich nur ein Mittel und kein Ziel. Managementziel muss hochwertiger, erweiterbarer Code sein, der im Rahmen einer von vornherein systematisch entworfenen Architektur entsteht und gepflegt wird. Nur Systeme mit guter Architektur lassen sich zu vertretbaren Kosten evolutionär weiter entwickeln. *Deshalb* ist Software-Architektur ein Management-Thema. Wenn eine Architektur nicht sichtbar ist, haben Manager wirklich keine Chance, für Qualität zu sorgen, dann können sie nur kapitulieren (s.o.) und an Symptomen herumkurieren. Dass eine objektorientierte Sprache allein eine durchschaubare Architektur erzeugt, wurde nicht nur hier empirisch widerlegt (s. auch [La03; BS02]).

Fazit 4: Machen Sie sich den grundsätzlichen Unterschied zwischen gezeichneten und generierten Bildern klar und handeln Sie danach. Jedes gezeichnete Bild kann lügen oder auf Fehleinschätzungen beruhen. Ein aus der Originalinformation generiertes Bild ist immer wahr, auch wenn es unangenehme Wahrheiten zeigt. Die Originalinformation ist der Programmcode. Den haben Sie *immer* zu verantworten, im Guten wie im Schlechten, selbst wenn keine einzige Zeile Text (Anforderungen, Spezifikation, Entwurf ect.) existieren sollte.

In der Diktion der Zeitschrift *Wirtschaftsinformatik* lassen sich daraus ableiten:

Empfehlungen für das Management

- Installieren Sie eine Qualitätssicherung, der Werkzeuge zur Quellcodeanalyse zur Verfügung stehen.
- Bewerten Sie die Qualität aller im Einsatz befindlichen Software.
- Leiten Sie ggf. Maßnahmen zum Reengineering oder zur Reinvestition ein.
- Sorgen Sie dafür und überprüfen Sie, dass Neuentwicklungen nur mit ständig auf Quellcode-Ebene überprüfter Architektur betrieben werden.

Dann werden Sie in Zukunft Sternbilder und keine Milchstraße mehr sehen.

Literaturverzeichnis

[Ar93] Arnold, R.S.: A Roadmap Guide to Software Reengineering Technology. 3rd printing, IEEE Comp. Society Press, Los Alamitos 1993.

[BCK98] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Addison-Wesley, Boston et al. 1998.

[BP86] Boehm, B.W.; Papaccio, P.N.: Understanding and Controlling Software Costs, in: IEEE Transactions on Software Engineering 14 (1988) 10, 1462-1477.

[Bo91] Booch, G.: Object Oriented Analysis and Design with Applications. Benjamin/ Cummings, Redwood City/CA 1991.

BS02] Broy, M.; Siedersleben, J.: Objektorientierte Programmierung u. Softwareentwicklung – Eine kritische Einschätzung. Informatik Spektrum 25 (2002)1, 3-11.

[CC03] CC GmbH: Analyzer. http://www.caseconsult.com/products/analyzer/en/ on May-15-2004.

[Di03] Diehl, S.: Softwarevisualisierung. Informatik Spektrum, 26 (2003) 4, 257-260.
[Di72] Dijkstra, E.W.: Notes On Structured Programming. In: Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, London - New York 1972.
[Ga95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison-Wesley, Boston et al. 1995.
[Ho97] Holt, R.: Software Bookshelf: Overview and construction. http://swag.uwaterloo.ca/pbs/papers/bsbuild.html on March 1997.
[La03] Lanza, M.: Object-Oriented Reverse Engineering -Coarse-grained, Fine-grained, and Evolutionary Software Visualization. Dissertation Universität Bern, Institut für Informatik und Angewandte Mathematik 2003.
[Me88] Meyer, B.: Object-oriented Software Construction. Prentice Hall, Hempstead 1988.
[Pa03] Parasoft Inc.: http://www.parasoft.com/jsp/home.jsp on May-15-2004.
[SD00] Siedersleben, J.; Denert, E.: Wie baut man Informationssysteme? – Überlegungen zur Standardarchitektur. Informatik Spektrum 23 (2000) 4, 247-257.
[SG96] Shaw, M.; Garlan, D.: Software Achitecture – Perspectives on an Emerging Diszipline. Prentice Hall, Upper Saddle River / NJ 1996.
[SW02] Sneed, H.M.; Winter, M.: Testen objektorientierter Software. Hanser, München 2002.
[So01] Sommerville, I.: Software Engineering. 6th ed., Addison-Wesley, Boston et al. 2001.
[Sp89] Spitta, T.: Software Engineering und Prototyping. Springer, Berlin et al. 1989.
[Sp96] Spitta, T.: CASE findet im Kopf statt. Informatik/Informatique 3 (1996) 3, 17-25.
[Sp98] Spitta, Th.: IV-Controlling in mittelständischen Industrieunternehmen - Ergebnisse einer empirischen Studie. Wirtschaftsinformatik 40 (1998) 5, 424-433.
[SS01] Spitta, T.; Sigge, A.: TQM II - Projektdokumentation. Fakuktät f. Wirtschaftswissenschaften, Universität Bielefeld 2001.*
[SSK02] Schätzle, R.; Seifert, T.; Kleine-Gung, J.: Enterprise JavaBeans – Kritische Betrachtungen zu einer modernen Software-Architektur. Wirtschaftsinformatik 44 (2002) 3, 217-224.
[St98] Stroustrup, B.: Die C++ Programmiersprache. 3.Aufl., Addison-Wesley, Bonn et al. 1998.
[Sz99] Szyperski, C.: Component Software – Beyond Object Oriented Programming. Addison-Wesley, Boston et al. 1999.
[Te04] Teßmer, M.: Architekturermittlung aus Quellcode. Diplomarbeit, Universität Bielefeld, Technische Fakultät, Jan. 2004.
[Un03] Understand C++: http://www.scitools.com/ on May-15-2004.
[Wr02] Wrede, S.: Software-Architektur und Kommunikation von Design. Diplomarbeit, Universität Bielefeld, Technische Fakultät, Juli 2002.*
[YC76] Yourdon, E., Constantine, L. L.: Structured Design. Prentice Hall, Englewood Cliffs/ N.J. 1976
[Zu98] Zuse, H.: A Framework of Software Measurement. DeGruyter, Berlin - New York et al. 1998.

* Unter www.wiwi.uni-bielefeld.de/~Spitta/download.html zugänglich.