Wissensorientierte Beschreibung großer Softwaresysteme - ein Ansatz jenseits softwareorientierter Konzepte

Dr.–Ing. Peter Tabeling

HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik
Prof. Dr. Helmert - Straße 2-3
14482 Potsdam

Peter.Tabeling@hpi.uni-potsdam.de

Wissensorientierte Beschreibung großer Softwaresysteme - ein Ansatz jenseits softwareorientierter Konzepte

Zusammenfassung:
Gängige Ansätze zur Beschreibung von Softwaresystemen zielen primär auf die Beschreibung von Software ab. Sie erleichtern damit das Verständnis von Codestrukturen und stellen Software im Wesentlichen als Ergebnis eines Entwurfsprozesses dar. Soll die Beschreibung von Softwaresystemen jedoch die Grundlage des Wissensmanagements bilden und einen effizienten Wissenstransfer zwischen den beteiligten Entwicklern fördern, so werden Beschreibungsansätze benötigt, die über softwareorientierte Konzepte hinausgehen.

1 Einführung

1.1 Wissen über Softwaresysteme

Bevor auf zentrale Fragen dieses Beitrags eingegangen wird, ist es erforderlich, den Wissensbereich abzugrenzen, auf den sich die weiteren Überlegungen beziehen. Den Ausgangspunkt bildet das Wissen derjenigen Personen, die direkt oder indirekt an der Entwicklung eines Systems beteiligt sind. Dabei sollen unter der Entwicklung alle Tätigkeiten verstanden werden, die ausgehend von der anfänglichen (mehr oder weniger detaillierten) Aufgabenstellung bis hin zum vollständigen Programmcode durchzuführen sind.

Ein Teil des Wissens betrifft den *Vorgang* der Entwicklung, also z.B. Fragen bezüglich der Entwicklungsphasen, ihrer Koordination und bezüglich der Verantwortlichkeiten der Beteiligten. Davon zu unterscheiden ist Wissen über den *Gegenstand* der Entwicklung, also das eigentlich gewünschte System bzw. dessen Software. Dieses Wissen steht im Zentrum der weiteren Betrachtungen.

1.2 System versus Software

Bei der Beschreibung von Softwaresystemen ist es dringend erforderlich, klar zwischen dem zu schaffenden System und der zugehörigen Software zu unterscheiden. Dass es sich dabei um grundsätzlich verschiedene Dinge handelt, lässt sich am besten durch einen Vergleich verdeutlichen: Man betrachte ein Flugzeug als ein *System* in dem Sinne, dass es sich um ein technisches Produkt handelt, welches einen komplexen Aufbau aus Komponenten aufweist, deren Zusammenspiel ein bestimmtes Systemverhalten ergibt [1]. Hier entspräche also das fertige, funktionstüchtige Flugzeug einem fertigen Softwaresystem wie z.B. einem laufenden Textverarbeitungssystem, mit dem der Benutzer arbeiten kann.

Bevor jedoch ein System hergestellt werden kann, muss es zunächst als Modell gegeben sein, d.h. als *gedachtes* System im Kopf der mit dem Entwurf befassten Personen. Dies wäre z.B. das gedachte Flugzeug vor den geistigen Augen des Flugzeugkonstrukteurs bzw. die abstrakte Vorstellung des Softwareentwicklers von dem zu erstellenden Textverarbeitungssystem.

Damit ein System auch als physikalisches Gebilde entstehen kann, muss es gefertigt werden. Dies erfordert eine Beschreibung der bis dahin nur gedachten Sachverhalte derart, dass eine Fertigung ausgehend von dieser Beschreibung das gewünschte System herstellen kann. Im Flugzeug-Beispiel sind dies die detaillierten Konstruktionspläne, die als Vorgabe für die Herstellung des Flugzeugs geeignet sind. Den detaillierten Plänen des Flugzeuges entspräche bei der Textverarbeitung die Software, denn der Programmcode beschreibt erstens das bisher nur gedachte System und ist zweitens als Ausgangspunkt für eine „Fertigung“ geeignet. Diese „Fertigung“ besteht im Falle des Softwaresystems aus der Übersetzung des Programms, dessen Installation auf einem Computer und dem Start der Programmabwicklung. Es mag zwar diskussionswürdig sein, ob nicht zumindest ein Teil der Softwarentwicklung ebenfalls als Teil der Fertigung anzusehen ist. Dies stellt jedoch nicht die Sichtweise in Frage, dass Software als Beschreibung anzusehen ist während das eigentlich gewünschte System Gegenstand dieser Beschreibung ist.

1.3 Softwarestrukturen versus Systemstrukturen

Die in Abschnitt 1.2 erläuterte Unterscheidung von Software und dem durch Software beschriebenen System hat gerade im Bereich großer Softwaresysteme besondere Bedeutung. Dort ist nicht nur die Software sehr umfangreich und stark strukturiert, sondern das durch sie beschriebene System ist ebenfalls ein sehr komplexes Gebilde.

Diese beiden Strukturbereiche - Softwarestruktur und Systemstruktur - sind von grundsätzlich unterschiedlicher Art (siehe Bild 1).

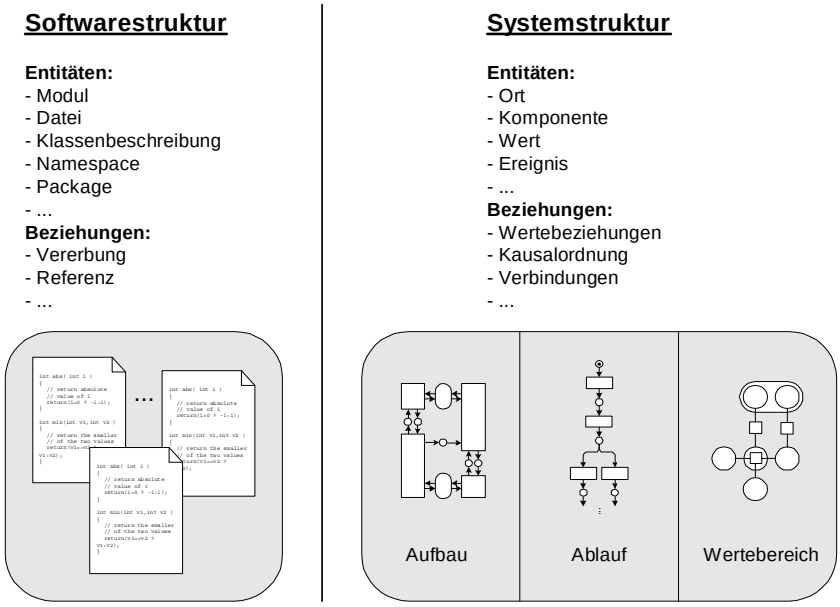

Bild 1

Bei der Systemstruktur sind primär Aufbau-, Ablauf- und Wertebereichsstrukturen zu unterscheiden [2]. Der Aufbau des Systems ist durch die Menge der Systemkomponenten sowie deren Verbindungen untereinander gegeben. Dabei übernehmen aktive Systemkomponenten bestimmte, klar abgrenzbare Aufgaben wie z.B. die Rechtschreibeprüfung in einem Textverarbeitungssystem, während an bestimmten Orten (Speichern) Information aufbewahrt wird bzw. über andere Orte

(Schnittstellen) Information von einer Komponente zur nächsten fließt. Im Falle von Softwaresystemen ist dieser Aufbau zwar oftmals fiktiv, da er sich nicht im physikalischen Aufbau des Systems widerspiegelt, aber er stellt eine wertvolle gedankliche Aufteilung des Systems in überschaubare Einheiten dar.

Ablaufstrukturen erfassen dagegen die Vorgänge, die im System stattfinden. Es handelt sich im Wesentlichen um Strukturen aus kausal bzw. zeitlich geordneten Ereignissen. Diese werden von zuständigen Komponenten erzeugt und sind an bestimmten Orten des Systems beobachtbar.

Hinzu kommen als dritter Strukturtyp die Wertebereichsstrukturen, d.h. die Strukturen der im System verarbeiteten informationellen Einheiten. Beispiele hierfür wären komplexe Datentypen wie Bäume, Listen oder Tabellen.

Systemstrukturen sind Strukturen, die der Mensch beim Nachdenken über Systeme vor seinem geistigen Auge sieht. Sie stellen eine gedachte Arbeitsteilung im System dar, die die Verständlichkeit des Gesamtsystems erhöht. Dass dieses Denken in interagierenden Akteuren tatsächlich naheliegend ist, erklärt, warum bei frühen Methoden der Softwareentwicklung Datenflussdiagramme verwendet wurden, die einen Systemaufbau in dem beschriebenen Sinne wiedergeben [3].

Softwarestrukturen sind dagegen Beschreibungsstrukturen. Diese bestehen aus Komponenten, die einzelne Bereiche der Systemstruktur beschreiben, wie z.B. die Beschreibung eines bestimmten Datentyps oder die Beschreibung einer Operation. Beziehungen zwischen Softwarekomponenten sind z.B. import-Beziehungen zwischen Programmteilen oder die Vererbungsbeziehung zwischen Klassenbeschreibungen bei der objektorientierten Programmierung.

Eine klare Unterscheidung von Softwarestruktur und Systemstruktur wird leider nicht überall vollzogen, was zwangsläufig zu Verständnisproblemen führt. Dies wird z.B. bei bestimmten Begriffen ersichtlich, die oft benutzt werden, aber nicht eindeutig einer der beiden Strukturbereiche zuzuordnen sind. So ist z.B. nicht immer klar, was mit dem Begriff „Komponente“ gemeint ist. Bezeichnet der Begriff eine Systemkomponente oder ist damit eine Softwarekomponente gemeint? Betrachtet man wieder das Flugzeug-Beispiel aus Abschnitt 1.2, so wäre ein Triebwerk eine Systemkomponente, während der Softwarekomponente ein einzelner Konstruktionsplan aus der Menge aller Konstruktionspläne des Flugzeugs entspräche.

Wären Softwarestrukturen und Systemstrukturen stets leicht aufeinander abbildbar, so wäre deren Unterscheidung von geringerer Bedeutung. Dass dies gerade bei großen Systemen nicht der Fall ist, ergibt sich aus den unterschiedlichen Kriterien, nach denen diese Strukturen ausgedacht werden bzw. werden sollten. David L. Parnas hat früh erkannt, dass es nicht zweckmäßig ist, Software ausgehend von der Ablaufstruktur des Systems zu modularisieren [4]. Er schlägt vielmehr vor, Software derart zu strukturieren, dass jedes Modul eine Entwurfsentscheidung verbirgt. Auf diese Weise erfordert eine revidierte Entwurfsentscheidung nur eine lokale, leicht durchführbare Änderung der Software. Dieses „Geheimnisprinzip“ wird heute durch die Kapselung bei der objektorientierten Programmmierung unterstützt.

Verallgemeinert man die Erkenntnis von Parnas, so bedeutet dies, dass Software zwar Systemstrukturen beschreibt, aber keineswegs gemäß den Systemstrukturen aufgebaut sein sollte [5].

Diese Erkenntnis ist natürlich nicht so zu verstehen, dass Software losgelöst von dem durch sie beschriebenen System zu betrachten sei. Letztlich ist Software ja nur Mittel zum Zweck - nämlich der Realisierung des gedachten Systems. Software kann somit nicht ohne ein Verständnis des beschriebenen Systems erstellt oder verändert werden.

Die während der Systementwicklung getroffenen Entwurfsentscheidungen äußern sich auf der Seite der Systemstrukturen in der Weise, dass nicht eine einzige allgemeingültige Vorstellung vom System gegeben sein kann. Jede Entwurfsentscheidung beschreibt letztlich einen Wechsel der Betrachtungsebene, bei der Elemente eines Systemmodells durch implementierende Strukturen ersetzt werden.

Derartige Übergänge zwischen Betrachtungsebenen können grundsätzlich in jedem der drei Strukturbereiche Aufbau, Ablauf bzw. Wertebereiche gegeben sein. Bild 2 zeigt dazu einfache Beispiele. Im Falle der Dekomposition wird ein Speicher S bzw. eine Komponente K des Systemaufbaus durch jeweils implementierende Speicher (S_1,S_2) bzw. Komponenten (K_1,K_2) ersetzt. Die Implementierung einer Tauschoperation (swap) durch eine Operationsfolge (Dreieckstausch) stellt ein Beispiel für den Bereich Ablauf dar. Ein entsprechendes Beispiel für den Strukturtyp Wertebereich ist die Kodierung einer reellen Zahl durch zwei ganze Zahlen (Fließkomma-Darstellung).

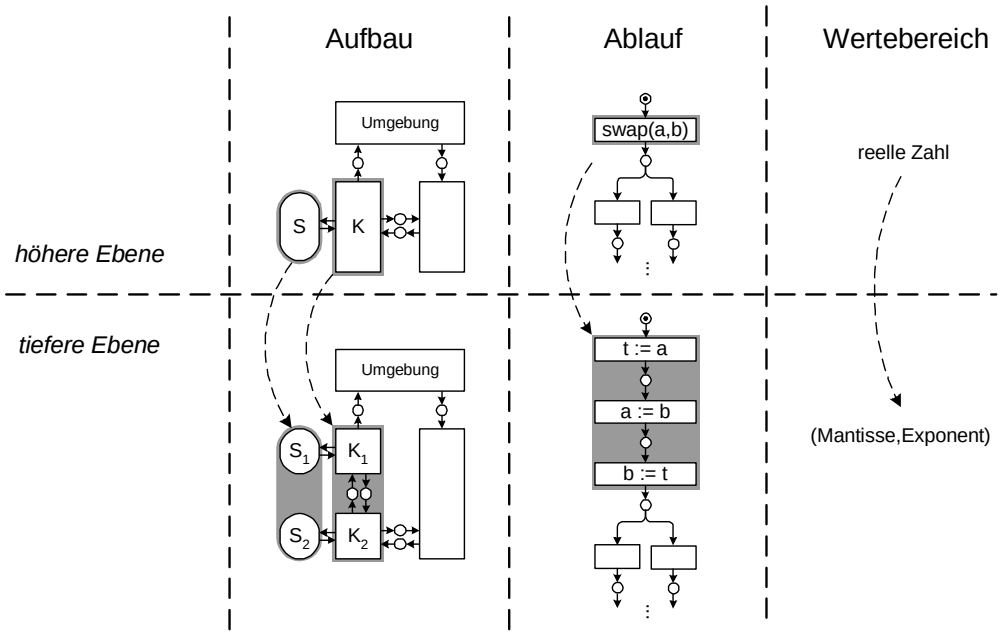

Bild 2

Speziell bei komplexen Systemen, bei denen ein große Zahl von Entwurfsentscheidungen getroffen wurde, sind daher viele Systemmodelle zu unterscheiden. Welches dieser Modelle angemessen ist, hängt davon ab, wie weit sich der Betrachter für bestimmte Entwurfsentscheidungen interessiert. In jedem Falle stellt das Festlegen unterschiedlicher Systemmodelle und ihrer (Implementierungs-) Beziehungen untereinander ein wesentliches Merkmal des Entwurfs komplexer Systeme dar.

1.4 Kennen versus Verstehen von Systementwürfen

Angenommen, ein vor kurzem bei einem Softwarehersteller eingestellter Mitarbeiter stünde vor der Aufgabe, bei der Weiterentwicklung eines Lagerverwaltungssystems mitzuarbeiten. In diesem Falle müsste er sich zunächst auf den Wissensstand der Kollegen bringen. Möglicherweise würde er sich dazu alle Quelltexte des bisherigen Systems sowie die zugehörigen Begleitdokumente

besorgen und diese studieren. Daraus wäre zum Beispiel ersichtlich, wie bestimmte Datentypen codiert wurden oder dass bestimmte Informationen in der Datenbank eines bestimmten Herstellers abgelegt werden usw.

Nach dieser Einarbeitung würde der Mitarbeiter den bestehenden Entwurf *kennen*, denn ihm wären alle Entwurfsentscheidungen bekannt. Möglicherweise würde ihm dennoch wichtiges Wissen, welches er für seine Mitarbeit bei der Weiterentwicklung benötigt, fehlen. Um nämlich den Entwurf in zweckmäßiger Weise abändern zu können, benötigt er noch das Wissen, warum der bisherige Entwurf so ist wie er ist. Warum wurde z.B. bei der Datenbank nicht auf ein billigeres Konkurrenzprodukt zurückgegriffen? Wäre es von Nachteil, dies zu tun?

Verallgemeinert bedeutet dies, dass man ein komplexes Softwaresystem nur dann *verstehen* kann, wenn man nicht nur die Entwurfsentscheidungen kennt, sondern auch weiß, welche Alternativen es gab und warum diese verworfen wurden. Nur wenn dieses *Hintergrundwissen* präsent ist, kann z.B. aus bereits gemachten Fehlern gelernt werden.

2 Beschreibung großer Softwaresysteme

Es wurde verdeutlicht, dass Software als Beschreibung eines gedachten Systems zu verstehen ist, die für die Fertigung des eigentlich gewünschten Systems benötigt wird (siehe Abschnitt 1.2). Eine Beschreibung des Systems wird natürlich nicht erst in diesem Zusammenhang erforderlich. Die Erstellung von großen Softwaresystemen erzwingt ausgeprägte Arbeitsteilung, sodass viele verschiedene Personen an der Entwicklung beteiligt sind. Eine funktionierende Arbeitsteilung wiederum erfordert effiziente Kommunikation, da die Beteiligten ihr Wissen über das zu erstellende System austauschen müssen. Zu diesem Zweck werden spezielle Beschreibungen benötigt, die im Hinblick auf die Mensch-Mensch-Kommunikation besonderen Anforderungen genügen müssen.

2.1 Softwareorientierte Ansätze und ihre Grenzen

Betrachtet man die Tätigkeit eines Softwareentwicklers, so ist Software der unmittelbare Gegenstand seiner Arbeit. Erst wenn diese Software - und sei es nur zu Testzwecken - zur Ausführung gebracht wird, entsteht das eigentlich gewollte System. Somit steht meist die Erstellung und Erweiterung von Software im Zentrum der Arbeit. Vor diesem Hintergrund ist es nicht verwunderlich, dass bei vielen Ansätzen zur Beschreibung großer Softwaresysteme ebenfalls Software im Zentrum des Interesses steht.

Bei diesen softwareorientierten Ansätzen wird vor allem versucht, Software systematisch zu erstellen und ihre Strukturen kompakt darzustellen. Die Darstellung von Systemstrukturen spielt dabei oft nur eine untergeordnete Rolle. Beispielsweise wird bei der objektorientierten Vorgehensweise teilweise empfohlen, bereits während der ersten Analysephase zu einem Klassendiagramm zu gelangen, welches im Wesentlichen auch die Softwarestruktur bestimmt [6]. Auf diese Weise ermöglicht man zwar einen einfacheren Übergang zu gut modularisierter Software. Eine angemessene Darstellung des Systemaufbaus entsteht auf diese Weise jedoch nicht. Die beim objektorientierten Entwurf gern verwendete Unified Modeling Language [7] bietet dazu auch keinen geeigneten Diagrammtyp an.[1] Dies ist gerade bei verteilten Systemen von Nachteil, da bei diesen besonders interessante Konstruktionsmerkmale im Bereich der Aufbaustrukturen gegeben sind.

1. allein die „collaboration diagrams“ sind dazu geeignet - allerdings nur sehr eingeschränkt, da nur primitive Aufbaustrukturen auf Objektebene darstellbar sind.

Die in der Praxis verwendeten CASE-Tools bieten oft die Möglichkeit an, ausgehend von Diagrammen Programmcode zu erzeugen. Dies hat jedoch zur Folge, dass die Diagramme in enger, formal definierter Beziehung zum Programmcode stehen müssen, da sonst eine automatische Codeerzeugung nicht möglich wäre. Es ist dann nicht mehr möglich, diese Diagramme im Hinblick auf Verständlichkeit zu optimieren, da z.B. das bewusste Weglassen unwichtiger Details die zur Codeerzeugung erforderliche Vollständigkeit aufhebt.

Gerade bei der Beschreibung großer Softwaresysteme ist es besonders wichtig, zunächst ein Verständnis des eigentlich gewollten Systems zu vermitteln. Das notwendige Wissen ist softwareorientierten Beschreibungen oft nicht zu entnehmen. Dies liegt zum einen daran, dass zuviel Information über Softwarestrukturen enthalten ist, die bei der Erlangung eines Systemverständnises nur stört. Zum anderen sind codenahe Beschreibungen oft nicht ausdrucksstark genug, um die eigentlich interessanten Systemstrukturen wiederzugeben.

Eine von Softwarestrukturen losgelöste Beschreibung des Systems ist spätestens dann erforderlich, wenn zu bestimmten Systemteilen gar keine Software gegeben ist. Dies ist z.B. dann der Fall, wenn Fremdprodukte integriert worden sind, zu denen kein Quelltext verfügbar ist oder ein System teilweise Hardwarelösungen enthält.

Projekte zur Beschreibung großer Systeme (u.a. R/3 von SAP) brachten die Erfahrung, dass wesentliches Wissen über das System überhaupt nicht dem Programmcode zu entnehmen ist, sondern nur den Köpfen der Entwickler. Dieses Wissen kann nicht erfasst werden, wenn man primär Softwarestrukturen beschreibt. Es handelt sich zum einen um übergeordnete Vorstellungen des Systems, die die Entwickler vor Augen haben. Zum anderen ist es gerade das in Abschnitt 1.4 behandelte Hintergrundwissen, ohne das kein wirkliches Systemverständnis möglich ist.

2.2 Ein wissensorientierter Ansatz

Hat man nur die Absicht, den Entwicklern den Zugang zu Softwarestrukturen zu erleichtern, so leisten softwareorientierte Ansätze gute Dienste. Möchte man jedoch möglichst alles Wissen der Entwickler über das Softwaresystem verfügbar machen, um z.B. die Einarbeitungszeiten für neue Mitarbeiter zu verkürzen, so treten deutliche Defizite zu Tage. Die vorangegangenen Betrachtungen haben gezeigt, dass softwareorientierte Ansätze zwei wichtige Wissensbereiche nicht abdekken: Wissen um Systemstrukturen und Hintergrundwissen zum Entwurf.

> *Ein wissensorientierter Ansatz zur Beschreibung von Softwaresystemen erfasst neben dem Wissen um Softwarestrukturen auch das Wissen um Systemstrukturen und das Hintergrundwissen zum Entwurf.*

Ein derartiger Ansatz wird am Hasso-Plattner-Institut verfolgt. Dabei haben sich zur Beschreibung von System- bzw. Softwarestrukturen grafische Darstellungen als besonders geeignet erwiesen. Während zur Darstellung von Softwarestrukturen inzwischen gute Diagrammtypen (z.B. UML-Klassendiagramme) allgemein verfügbar sind, werden zur Darstellung von Aufbau-, Ablauf- und Wertebereichsstrukturen eigene Notationen verwendet. Dabei wurden bestimmte Prinzipien übernommen, die sich im Ingenieursbereich bei der Gestaltung von Konstruktionsplänen bewährt haben [8]. Beispielsweise wird zwecks schnellerer Erfassung durch den Menschen auf ein möglichst einfaches Grundrepertoire grafischer Elemente zurückgegriffen. Da Systemstrukturen dabei losgelöst von Softwarestrukturen beschrieben werden, können die Darstellungen im Hinblick auf Verständlichkeit optimiert werden [9].

Die grafischen Darstellungen erfassen die wesentlichen Merkmale des Entwurfs und sollen prägend für die Systemvorstellung sein. Unwichtige Details, die leicht durch Studieren des Programmcodes in Erfahrung zu bringen sind, werden dabei bewusst weggelassen. Um auch das

Hintergrundwissen zu erfassen, werden die Pläne durch kommentierende Texte und weitere Diagramme erläutert. Die in dieser Weise gestalteten Beschreibungen haben sich nicht nur bei der Dokumentation kleinerer Projekte, sondern auch bei der Nachdokumentation großer Systeme bewährt.

Gerade bei großen Systemen hat es sich als vorteilhaft erwiesen, auch auf höheren Betrachtungsebenen Aufbaustrukturen (siehe Abschnitt 1.3) zu beschreiben. Zum einen liefert ein fiktiver Aufbau eine bessere Grundlage für ein anschauliches Verständnis des Systems, da dabei eine klare Trennung zwischen den agierenden Systemkomponenten und den von Aktivitäten betroffenen Schnittstellen bzw. Speichern vollzogen wird. Diese Unterscheidung wird z.B. bei der objektorientierten Modellierung vernachlässigt, sodass es oft unklar ist, ob Objekte als aktive Systemkomponenten oder als passive Daten zu deuten sind. Auf diese Weise ensteht keine anschauliche Systemvorstellung.

Unanschaulichkeit des Systemmodells ist nur eines der Probleme, welche durch die aufbauorientierte Modellierung angegangen werden. Große Systeme sind typischerweise verteilt. Die aufbauorientierte Sichtweise erlaubt es, diese Systeme angemessener zu modellieren, da nicht nur eine Verteilung im physikalischen Sinne, sondern auch ein abstraktes Verteiltsein des Systems beschrieben werden kann. Grundlage dieses erweiterten Verteilungsbegriffes ist ein abstrakter Ortsbegriff, der auf der Anschauung basiert, dass ein Ort ein eng abgegrenzter räumlicher Bereich ist, in dem mittels einer einzelnen Operation Daten verändert bzw. beobachtet werden können. Daraus leitet sich die Atomarität von Zugriffen auf Orte sowie bestimmte Konsistenzeigenschaften bei der Beobachtung von Orten durch Systemkomponenten ab. Die tatsächliche Abbildung abstrakter Orte auf physikalische Orte wird jedoch erst durch die Realisierung des Systems bestimmt [10] [11].

Aufbauend auf diesem abstrakten Ortsbegriff kann einerseits Verteilung in Systemmodellen höherer Betrachtungsebenen beschrieben werden, wobei zwischen einer Verteilung im Sinne einer Aufteilung in Systemkomponenten und einer abstrakten räumlichen Verteilung von Daten zu unterscheiden ist. Andererseits ergibt sich nun ein klarer Zusammenhang zwischen Aufbau- und Ablaufstrukturen, da sich aus der abstrakten räumlichen Verteilung der (maximale) Nebenläufigkeitsgrad des Systems ableitet.

Erst die Unterscheidung von Betrachtungsebenen in Verbindung mit einer aufbauorientierten Modellierung erlaubt es, typische Konstruktionsmerkmale verteilter Systeme wie z.B. Replikation oder Caching auf elegante Weise zu beschreiben. Diese stellen nämlich im wesentlichen Konstruktionsentscheidungen dar, die Aufbaustrukturen unterschiedlicher Systemmodelle in Beziehung setzen. Auch Transaktionen sowie der Bedarf nach Snapshot- und Broadcast-Algorithmen ergeben sich geradezu zwingend aus der Abbildung abstrakter Aufbaustrukturen auf implementierende Aufbaustrukturen [10]. Umgekehrt bedeutet dies, dass diese technischen Details bei der Modellierung eines Systems auf höherer Betrachtungsebene “ausgeblendet” werden können, was eine wertvolle Komplexitätsreduktion mit sich bringt. So stellen z.B. Transaktionen auf höherer Betrachtungsebene einen Zugriff auf einen abstrakten Ort dar, der im Rahmen einer einfachen Operation stattfindet. Mit dem vorgestellten Ansatz können daher nicht nur realisierungsnahe detaillierte Modelle, sondern auch aufgabennahe übergeordnete Modelle entwickelt werden, wobei stets die gleiche Begriffswelt und Notation zugrunde liegt.

Literatur

[1] Siegfried Wendt. *Nichtphysikalische Grundlagen der Informationstechnik*. Springer Verlag, Heidelberg 1989, S. 134

[2] Siegfried Wendt. *Der Kommunikationsansatz in der Software-Technik*. Data Report, Nr. 17, 1982, S. 4ff

[3] Tom DeMarco. *Structured Systems Analysis and System Specification.* Englewood Cliffs: Yourdon Press 1978

[4] David Lorge Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules*. Communications of the ACM, Vol. 15, Nr. 12, December 1972

[5] Wolfram Kleis. *Konzepte zur verständlichen Beschreibung objektorientierter Frameworks*. Shaker Verlag, Aachen 1999 (zugl. Dissertation, Universität Kaiserslautern), S. 19f

[6] Martin Hitz, Gerti Kappel. *UML@work*. dpunkt-Verlag, Heidelberg 1999, S. 174f

[7] Object Management Group. *OMG Unified Modeling Language Specification.* Version 1.3, März 2000, Object Management Group, www.omg.org/uml

[8] Peter Tabeling. *Mittel zur effizienten Kommunikation über große Softwaresysteme*. KnowTech 2000, www.knowtech.net/2000

[9] Andreas Bungert. *Beschreibung programmierter Systeme mittels Hierarchien intuitiv verständlicher Modelle.* Shaker Verlag, Aachen 1998 (zugl. Dissertation, Universität Kaiserslautern)

[10] Peter Tabeling. *Der Modellhierarchieansatz zur Beschreibung nebenläufiger, verteilter und transaktionsverarbeitender Systeme.* Shaker Verlag, Aachen 2000 (zugl. Dissertation, Universität Kaiserslautern)

[11] Peter Tabeling. *Der Modellhierarchieansatz zur Beschreibung nebenläufiger und verteilter Systeme*. Beitrag zum GI-Workshop „Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme“, Münster, November 2000