Von den Anfängen der objektorientierten Softwareentwicklung an der TU Ilmenau bis zu einem Ansatz für ein komponentenorientiertes Softwaresystem

Reinhold Schönefeld

Schulung & Beratung für Software
Naumannstr. 9a
98693 Ilmenau
schenski@t-online.de

Abstract: In der Arbeit wird kurz die Geschichte der objektorientierten Softwareentwicklung an der TH Ilmenau aufgezeigt, um dann auf die durch Kompositionsabstraktion erreichten Eigenschaften eines Objektes einzugehen. Der Hauptteil beschäftigt sich damit, wie durch systemtheoretische Konzepte ein Ansatz für die Entwicklung von komponentenorientierter Software aus den in der DDR begonnenen Untersuchungen zur Objektorientierung entstand.

1 Kurzer Abriss zur Geschichte der objektorientierten Softwareentwicklung an der TH Ilmenau in den 80-er Jahren

Anfang der 80-er Jahre wurde im Rechenzentrum der TH Ilmenau in Zusammenarbeit mit dem Kombinat Robotron Dresden am Betriebssystem MUTOS (Multi User Time Sharing Operating System) für die Klein- und Mikrorechner der DDR gearbeitet. Das MUTOS war kompatibel zum UNIX. Im Rahmen dieser Forschungs- und Entwicklungsarbeiten erschien eine erste Veröffentlichung „Objektorientierte Betriebssysteme – ihre Implementierung auf Kleinrechnern“ [SCHE84] aus dem Rechenzentrum, in der der Grundgedanke der Objektorientierung erstmalig eine Rolle spielte.

Kurze Zeit später konnten wir der internationalen Literatur entnehmen, dass die Wurzeln des objektorientierten Paradigmas bereits 1967 in der Programmiersprache SIMULA 67 zu finden waren und dass verschiedene Arbeiten zu Abstrakten Datentypen von Anfang der 70-er Jahre und auch die Module von D. L. Parnas aus dieser Zeit zum Themenkreis der Objektorientierung von Software gehörten. Die objektorientierte Programmiersprache Smalltalk 80 aus den USA war ab 1986 an der TH Ilmenau auf Kleinrechnern verfügbar und bis zum Herbst 1989 wurden an der TH Ilmenau bereits Vorlesungen zur Programmierung in C, C++ und Smalltalk gehalten. Eine „Studie zum objektorientierten Softwareentwurf“ als Forschungsauftrag für VEB Robotron-Projekt Dresden durch das Rechenzentrum der TH Ilmenau spiegelte den damaligen Wissensstand in Ilmenau im Jahre 1989 wider. Das Literaturverzeichnis enthält weitere Arbeiten aus dieser Zeit.

2 Das objektorientierte Paradigma aus systemtheoretischer Sicht

Ein Objekt bzw. ein Objekttyp (Klasse) ist bekanntlich die Zusammenfassung (Komposition) von Funktionen und Daten (Datenobjekte) zu einer neuen integrierten Softwareeinheit (Abb.1). In einem Objekt (Klasse) hängen die Funktionen und Daten von einander ab. Sie erlangen beide bestimmte Eigenschaften, die erst durch ihre Komposition zum Objekt erzeugt werden und das Objekt selbst hat bestimmte Eigenschaften, die nicht unmittelbar in den Funktionen und Daten zu finden sind. In der Systemtheorie wird von Emergenz neuer Eigenschaften gesprochen, die durch die Komposition verursacht werden.

Welche neuen Eigenschaften erlangen die Funktionen und Daten durch die Komposition zu einem Objekt? Die Funktionen dürfen direkt auf die Daten zugreifen und alle Funktionen zusammen bilden mit ihren Signaturen die Schnittstelle des Objekts zu seiner Umgebung. Andere Funktionen, außerhalb des Objekts, haben diese Eigenschaften nicht. Die Daten wiederum werden im Objekt zu Trägern des Objektzustandes. Sie spannen einen Zustandsraum des Objekts auf. Für sich betrachtet haben die Daten diese Eigenschaft nicht.

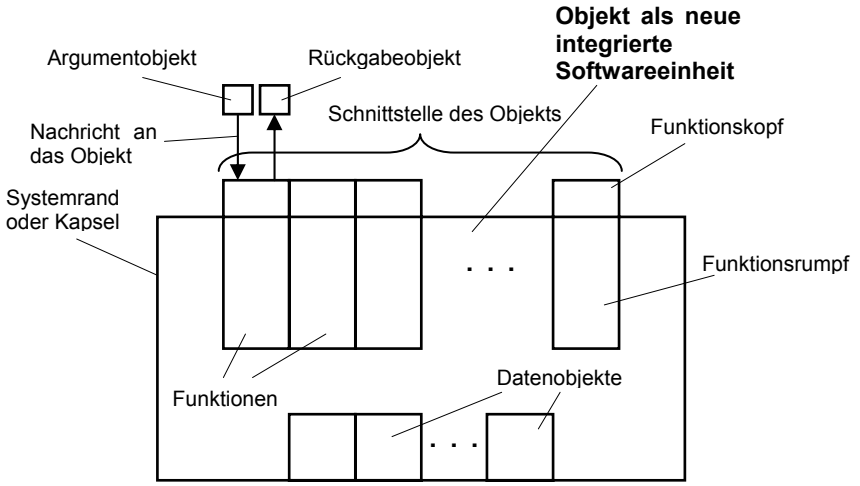

Abbildung 1: Sinnbild eines Objekts als Komposition von Funktionen und Datenobjekten

Welche neuen Eigenschaften bekommt ein Objekt durch die Komposition? Das Objekt als neues Ganzes besitzt eine Identität und zeigt ein äußeres Eingabe-Ausgabe-Verhalten an seiner Schnittstelle zur Umgebung. Dieses Verhalten ist nicht allein durch das Verhalten der einzelnen Funktionen zu erklären, sondern hängt auch von der Lage des Zustandsvektors im Zustandsraum des Objekts ab. Eine Funktion für sich hat für denselben Eingabewert den gleichen Ausgabewert. Das Verhalten des Objekts ist eine neue Qualität, es wird durch die Menge aller Eingabefolgen (Nachrichtenfolgen) an das Objekt und die dazugehörigen Ausgaben (Nachrichten an die Umgebung) während des Lebenszyklus eines Objekts gebildet. Dabei müssen auch wiederholt auftretende

Nachrichten in den Folgen berücksichtigt werden. Für dieselbe Nachricht hat ein Objekt nicht zwangsläufig das gleiche Rückgabeobjekt. Sein Verhalten zeigt ein Objekt an seiner Schnittstelle, die lediglich eine Passfähigkeit für die Nachrichten darstellt, die das Objekt empfangen kann, aber nicht das Verhalten selbst beschreibt. Die Ansicht, dass die statische Schnittstelle das dynamische Verhalten des Objekts ausdrückt, ist in der Literatur leider öfter zu beobachten.

Das Verhalten eines Objekts ist nicht einfach zu ermitteln. Für ein bereits existierendes Objekt entsteht es durch Überlagerung in den Funktionen des Objekts und wird durch die fundamentalen Steuerkonstrukte (Sequenz, Alternative, Iteration) und die aktuellen Werte der Zustandsvariablen verursacht. Das Verhalten ist durch die so genannte Programmfunktion mathematisch beschreibbar. Sie kann aus dem Programmcode oder dem graphisch dargestellten Programmablaufplan extrahiert werden (s. Kap. 4.2). Existiert ein Objekt noch nicht, kann sein gefordertes Verhalten durch die so genannte Sequenzbasierte Spezifikation ermittelt werden, indem die Nachrichtenfolgen oder Sequenzen systematisch durch Kombination erzeugt und den Ausgabewerten zugeordnet werden [Sch07, Kap. 6].

Systemtheoretisch betrachtet ist ein Objekt Ausdruck des Holismus, der Ganzheitslehre (Konzept erstmals bei J. Smuts 1926), indem ein System in seiner Wechselwirkung mit der Umgebung als Ganzes betrachtet wird und dabei die Teile, aus denen das System besteht, aus Sicht der Umgebung keine Rolle spielen. Der Holismus resultiert aus der Kompositionsabstraktion. In der ganzheitlichen Betrachtungsweise gehen die Eigenschaften des Systems nicht unmittelbar aus den Eigenschaften der Teile hervor, sondern „das Ganze ist mehr als die Summe seiner Teile“ (Aristoteles 384-322 v. Chr.). Durch Komposition von Teilen treten, wie schon erwähnt, vor allem bei komplexen Systemen emergente Eigenschaften auf, die nicht in den Teilen für sich betrachtet zu finden sind. Das Konzept der Abstrakten Datentypen, auch das der Softwaremodule und das der Endlichen Automaten entsprechen dem Konzept des Holismus der Systemtheorie.

3 Die Bedeutung der Objekte für die Softwareentwicklung der 90-er Jahre

Das Interesse der praktischen Softwareentwicklung an Objekten und Klassen war Ende der 80-er Jahre deshalb geweckt, weil viele reale Dinge oder Objekte der Wirtschaft und anderer Problemdomänen genau solche Objekte mit Identität, Verhalten und Zustand zu verwalten hatten. Denken wir dabei an die bekannten Businessobjekte ein Konto, einVertrag, einePerson usw. Der Entwurf eines Softwaresystems wurde durch das Objektkonzept vereinfacht, da ein Softwareentwickler die realen Objekte möglichst in entsprechende Softwareobjekte übertrug. Ein objektorientiertes Programm war durch die angewandte Kompositionsabstraktion bei der Objekt- bzw. Klassenbildung weitaus weniger komplex als ein vergleichbares strukturiertes oder gar unstrukturiertes Programm. Die Objekte (Objekttypen) wurden, ähnlich wie die Zellen durch die Natur, als elementare Bausteine in der Softwareentwicklung eingesetzt.

Neben der Reduktion der Komplexität objektorientierter Programme interessierte die Praxis vor allem die Wiederverwendung einmal entworfener Objekttypen (Klassen) in weiteren Programmen. Dazu dienten Klassenbibliotheken und das Konzept der Vererbung, das auf der Basis der bekannten Generalisierungsabstraktion zum Aufbau von Klassenhierarchien führte. Der damit einhergehende Polymorphismus war ein weiteres Konzept, was Flexibilität und mehr Möglichkeiten zur Wiederverwendung bot. Die 90-er Jahre zeigten jedoch, dass die Wiederverwendung von Objekten nicht den erhofften Erfolg bei der Steigerung der Arbeitsproduktivität und der besseren Beherrschbarkeit bei der Entwicklung komplexer objektorientierter Programme brachte. Woran lag das? Was kann uns die Geschichte hierzu lehren?

4 Ansatz für eine komponentenorientierte Softwareentwicklung

Aus heutiger Sicht hätte das systemtheoretische Konzept der Kompositionsabstraktion damals wiederholt angewandt werden müssen, um nicht nur Daten und Funktionen, sondern unterschiedliche Objekte zu einer weiteren neuen integrierten Softwareeinheit zusammen zu fassen. Diese Einheit wollen wir als Verbundkomponente bezeichnen (Abb. 2).

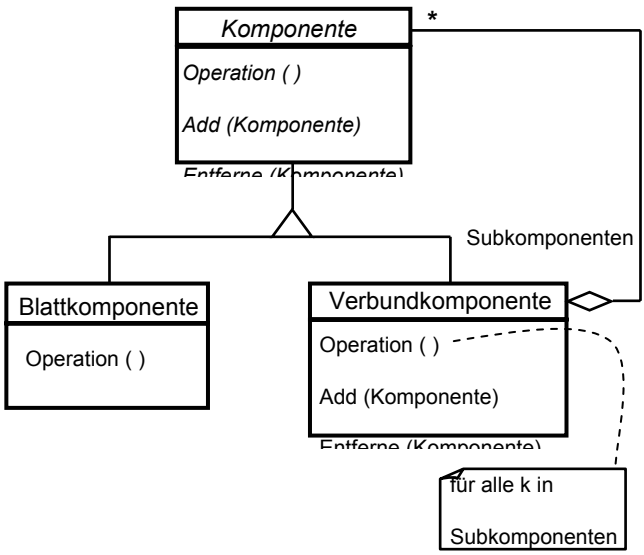

Abbildung 2: Kompositionsmuster zur Darstellung von Komponentenhierarchien

Wiederholt angewandt heißt, es können auch Verbundkomponenten und Objekte oder nur Verbundkomponenten zu einer neuen Verbundkomponente zusammengefasst werden. Eine Verbundkomponente setzt sich somit aus weiteren Verbundkomponenten und Objekten in einer hierarchischen Struktur zusammen, einer Komponentenhierarchie (Abb. 3). An der Spitze (Wurzel) befindet sich eine i. d. R. komplexe Verbundkomponente, die das gesamte zu bauende Softwaresystem darstellt und auf den jeweils niedrigsten Ebenen befinden sich ausschließlich Blattkomponenten, die einfache Objekte sind. Dazwischen können weitere Verbundkomponenten angeordnet sein. Wichtig ist, dass erkannt wird, dass die Kompositionsabstraktion nicht nur auf Funktionen und Daten im Objekt angewandt werden kann, sondern jetzt auf alle Teilsysteme in der Komponentenhierarchie in einheitlicher Weise anwendbar ist. Dabei wirkt die Kompositionsabstraktion, mit ihren durch Emergenz verursachten zusätzlichen Eigenschaften, bei jeder Verbundkomponente und ihren Teilkomponenten in gleicher Weise, wie bei den Objekten (s. Kap. 2) beschrieben. Jede so gebildete Verbundkomponente hat eine eigene Identität, zeigt ein äußeres Verhalten an ihrer Schnittstelle und besitzt einen inneren Zustand in ihrem Zustandsraum.

Zur Darstellung der hierarchischen Verhältnisse von Komponenten eignet sich das Kompositionsmuster aus der Sammlung der Entwurfsmuster für Softwaresysteme [GAM95] in bester Weise. Abbildung 2 zeigt den rekursiven Zusammenhang zwischen den Klassen der Komponenten. Die abstrakte Klasse *Komponente* vererbt ihre Funktionen (Methoden) und Daten (Attribute) an die konkreten Klassen Blattkomponente und Verbundkomponente. Eine Verbundkomponente setzt sich aus beliebig vielen (*) weiteren Subkomponenten zusammen, die entweder wieder Verbund- oder Blattkomponenten sind. Die Rekursion bricht ab, wenn nur noch Blattkomponenten (Objekte) auftreten. Eine aus dem Kompositionsmuster abgeleitete beispielhafte Komponentenhierarchie ist in Abbildung 3 dargestellt.

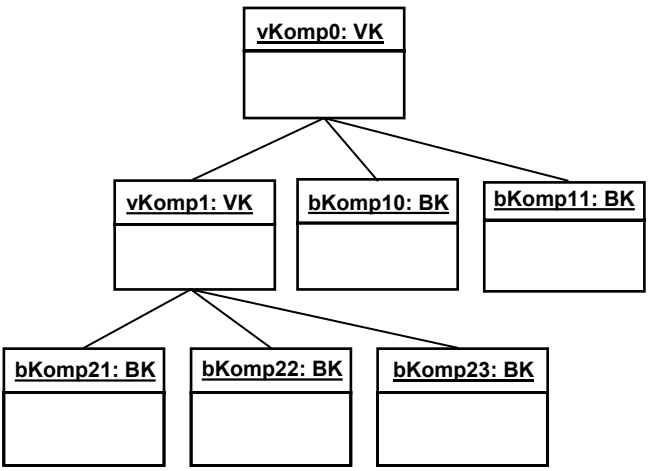

Abbildung 3: Komponentenhierarchie aus Verbund- und Blattkomponenten

4.1 Die Vorgehensweise beim Aufbau komponentenorientierter Software

Für den Aufbau eines komponentenorientierten Softwaresystems können zwei unterschiedliche Ausgangsituationen vorliegen:

1. Für ein solches Softwaresystem sind nur die informalen Anforderungen bekannt, jedoch keine vorbereiteten Komponenten vorhanden.

2. Das Softwaresystem existiert ebenfalls nicht, soll aber aus vorhandenen Komponenten zusammengebaut werden.

In beiden Fällen kann für die Entwicklung die so genannte Box-Struktur-Methode (BSM) zum Einsatz kommen [SCH07, Kap.5]. Die BSM ist die Anwendung der Systemtheorie auf ein Softwaresystem, indem es durch drei Sichten beschrieben wird. Die Black-Box-Sicht (BB-Sicht) oder äußere Verhaltenssicht, die State-Box-Sicht (SB-Sicht) oder Datensicht und die Clear-Box-Sicht (SB-Sicht) oder Architektursicht. Durch Anwendung der BSM wird ein Softwaresystem zunächst als Ganzes betrachtet und dann durch eine systematisch betriebene Dekomposition in Komponenten zerlegt. Die BSM geht auf Harlan D. Mills (1919-1996) zurück und ist Teil des Cleanroom Software Engineering (s. dazu [SCH07]).

Die BB-Sicht beschreibt in formaler Weise das geforderte funktionale äußere Verhalten des gesamten Softwaresystems in Form einer tabellarisch repräsentierten Funktion und ist die mathematisch formulierte Spezifikation des Softwaresystems. Die BB-Funktion beschreibt in abstrakter Weise das Eingabe-Ausgabe-Verhalten (die Dienste), die das Softwaresystem als zusammengesetztes Ganzes ausführen soll und nach seiner korrekten Implementierung auch ausführen kann.

Die SB-Sicht beschreibt ebenfalls in formaler Weise das äußere Verhalten (die Dienste) des Softwaresystems, jedoch unter Einführung von inneren Zustandsgrößen. Die SB-Sicht ist die bekannte Darstellung eines endlichen Automaten (auch Zustandsmaschine) in Form einer tabellarisch notierten Funktion. Die BB- als auch die SB-Sicht werden durch die Methode der Sequenzbasierten Spezifikation [SCH07, Kap. 6] aus den auf dem Rand des Softwaresystems eingehenden Stimuli (Eingaben) und den dort verlangten Reaktionen (Ausgaben) durch eine systematisch betriebene Auflistung von Stimulussequenzen (Stimulusgeschichten) und durch ihre Zuordnung zu den geforderten Reaktionen gefunden.

Aus der Auflistung gehen für das Softwaresystem charakteristische Sequenzen hervor, die so genannten kanonischen Sequenzen. Aus diesen lassen sich die Zustandsgrößen des Softwaresystems herleiten, die die Daten für das Softwaresystem, also der Verbundkomponente auf der obersten Ebene der Komponentenhierarchie, sind. Die Zustandsgrößen akkumulieren die auf dem Rand des Softwaresystems eintreffenden Stimuli und spannen je nach Anzahl der Stimuli einen mehrdimensionalen Zustandsraum auf. Die Menge der kanonischen Sequenzen und damit der unterscheidbaren Zustandsgrößen und ihrer Werte ist ein Maß für die Komplexität des Softwaresystems.

Sind die Zustände bekannt, kann die SB-Funktion des zu bauenden Softwaresystems als Zustandsmaschine ermittelt werden, die alle geforderten Transitionen enthält, die das zu bauende Softwaresystem ausführen können muss. Eine Transition ist ein Abbildungspaar in der SB-Funktion, das einem aktuellen Stimulus und dem existierenden Vorzustand eine Reaktion und einen Nachzustand zuordnet. Die SB-Funktion kann auch als geforderte Übertragungsfunktion des Softwaresystems angesehen werden, die angibt, wie unter Beachtung des Zustandes alle eintreffenden Stimuli in Reaktionen zu übertragen sind. Die SB-Funktion gibt somit vor, wie die Architektur und die daraus resultierende Implementierung aufgebaut sein müssen, um das gleiche Übertragungsverhalten wie die SB-Funktion zu haben. Die SB-Funktion wird deshalb auch als Vorgabefunktion bezeichnet.

Die CB-Sicht des Softwaresystems ist seine Architektur in Form von Komponenten (Blatt- und Verbundkomponenten), die über die fundamentalen Steuerstrukturen Sequenz, Alternative und Iteration verknüpft sind. Die Architektur ist die Verbundkomponente auf der obersten Ebene („das neue Ganze“), die sich aus den Komponenten auf der nächst niederen Ebene zusammensetzt, und wie sie in der obersten Ebene über die Steuerstrukturen verknüpft sind. Das Entwurfsproblem dabei ist bekanntlich, welche Komponenten mit welchem Verhalten benutze ich und mit welchen Steuerstrukturen verknüpfe ich sie, um mit der daraus resultierenden Architektur genau das geforderte Übertragungsverhalten zu erzeugen, das durch die Vorgabefunktion gefordert wird. Es geht also darum, eine Funktion für das Übertragungsverhalten der Architektur aus einem ersten Ansatz für die Architektur zu finden. Die so gewählte Definition der Architektur eines Softwaresystems besitzt den Vorteil der Verifizierbarkeit und Zerlegbarkeit. Verifizierbar, indem sie auf Korrektheit gegenüber der Vorgabefunktion überprüft werden kann. Zerlegbar, da die Komponenten auch über die gleichen fundamentalen Steuerkonstrukte verknüpft sind wie die elementaren Anweisungen innerhalb des Codes der Komponenten, die letztendlich durch fortgesetzte Zerlegung (schrittweise Verfeinerung) aus den Komponenten entstehen.

In der o. g. ersten Ausgangsituation, wo keine vorbereiteten Komponenten verfügbar sind, müssen diese erst entworfen werden. Dabei bieten die Transitionen der SB-Funktion einschließlich der gefundenen Zustandsgrößen wichtige Informationen für den ersten Ansatz solcher Komponenten. Das hauptsächlich eingesetzte Entwurfsprinzip ist das der „Kapselung der Zustandsgrößen“. Das bedeutet, bilde für eine oder für mehrere Zustandsgrößen zunächst eine Blattkomponente, in der als Daten die Zustandsgrößen eingehen. Als Funktionen werden solche eingeführt, die einen Teil der geforderten Transitionen der Vorgabefunktion ausführen können. Dabei ist die Abgeschlossenheit der Transitionen für die entworfene Komponente ein hilfreiches Kriterium. Die mit den Funktionen ausführbaren Transitionen müssen notwendig und hinreichend sein für die Bildung und Modifikation der gekapselten Zustandsgrößen und die Erzeugung der Reaktionen. Die Zustandsgrößen ihrerseits müssen notwendig und hinreichend sein für die Ausführung aller Transitionen der Blattkomponente.

Sind alle Blattkomponenten der Architektur entworfen worden, müssen sie geeignet zur Architektur über Steuerkonstrukte verknüpft werden. Die so erzeugte Ausgangsarchitektur ist entweder in einer Entwurfssprache (z. B. Pseudocode) oder als Box-Struktur-Diagramm notiert, das einem Ablaufplan für ein komponentenorientiertes Programm entspricht. Aus beiden lässt sich die Übertragungsfunktion der Architektur extrahieren, die auch Architekturfunktion genannt werden soll, und die in Form einer mit der Vorgabefunktion vergleichbaren Tabelle notiert ist. Die Architekturfunktion ist für die Ausgangsarchitektur erwartungsgemäß nicht in Übereinstimmung mit der Vorgabefunktion. Wie es möglich ist, die Architekturfunktion zu extrahieren und wie sie an die Vorgabefunktion angepasst werden kann, wird in Kap. 4.2 gezeigt.

Zunächst soll noch ein anderer Weg aufgezeigt werden, wie die Komponenten für den ersten Ansatz der Architektur gewonnen werden können. Das ist der Weg über die Spezifikation des geforderten Verhaltens für jede Komponente aus den informalen Anforderungen heraus. Es werden informale Anforderungen für die beabsichtigten Komponenten aufgestellt und für diese die BB- und SB-Funktion über eine Auflistung für jede Komponente ermittelt. Damit ist das Verhalten jeder Komponente zunächst nur in abstrakter Weise als BB- oder SB-Funktion gegeben. Das genügt aber, um die Architekturfunktion aus der Verknüpfung der Komponenten extrahieren zu können.

In der oben genannten zweiten Ausgangssituation, wo schon fertige Komponenten vorhanden sind, müssen diese mit einer exakten Verhaltensbeschreibung als BB- oder SB-Funktion geliefert werden. Denn nur so können sie zu einem ersten Ansatz für eine Architektur verknüpft werden. Ist eine explizite Verhaltensbeschreibung nicht vorhanden, muss sie zunächst aus der Architektur der jeweiligen Komponente oder aus ihrem Code extrahiert werden.

4.2 Extraktion der Übertragungsfunktion eines Softwaresystems

Jedes Programm (Softwaresystem) in Form seines Programmablaufs ist mathematisch betrachtet eine graphische oder prozedurale Regel, die einer Funktion oder Relation äquivalent ist. Das Ziel der Extraktion ist es diese Übertragungsfunktion, die das Eingabe-Ausgabeverhalten beschreibt, frei von graphischen oder prozeduralen Ausdrucksmitteln, rein als mathematische Funktion darzustellen. Diese Funktion wird als Programmfunktion bezeichnet. Die Architekturfunktion im speziellen ist die Programmfunktion für eine Verbundkomponente auf der obersten Hierarchieebene eines komponentenorientierten Softwaresystems. Die entscheidende Idee zur erfolgreichen Extraktion der Programmfunktion ist die Tatsache, dass die fundamentalen Steuerstrukturen selbst mathematischen Funktionen entsprechen, d. h. eine Abbildung ihrer Eingabewerte in ihre Ausgabewerte vornehmen. Die Architekturfunktion ist somit eine zusammengesetzte Funktion aus den Übertragungsfunktionen der Steuerstrukturen und den Übertragungsfunktionen der Komponenten. Sie hängt davon ab, welche Komponenten mit welchen Steuerstrukturen in verschachtelter und sequenzierter Weise verknüpft sind.

Zunächst ist es sinnvoll, die Programmfunktionen der bekannten Steuerstrukturen Sequenz, Alternative und Iteration mit ihren minimal notwendigen Komponenten in SB-Sicht aufzuzeigen. Die Programmfunktion ist die Abbildung, die zwischen den Daten am Eintrittspunkt (E) und den Daten am Austrittspunkt (A) jeder Steuerstruktur definiert ist.

Für die Sequenz ist in (Abb. 4) die Programmfunktion [P] durch die Funktionskomposition der beiden Übertragungsfunktionen $\boldsymbol{g}$ und $\boldsymbol{h}$ der einzelnen Komponenten gegeben:

$$[P] = [\boldsymbol{g} ; \boldsymbol{h}] = [\boldsymbol{h}] \circ [\boldsymbol{g}] \qquad (1)$$

Eckige Klammern bezeichnen Programmfunktionen und „o“ ist der Kompositionsoperator, der auch als „;“ (fettes Semikolon), mit Vertauschung der Operanden, geschrieben werden kann.

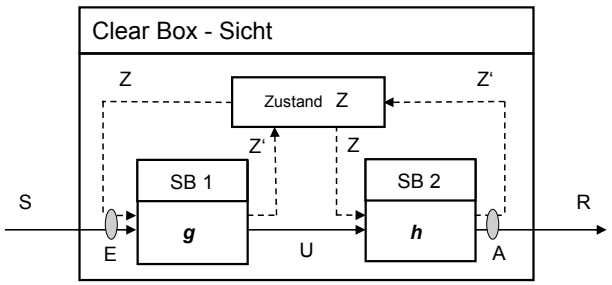

Abbildung 4: Zwei durch Sequenz verknüpfte Komponenten

Die Programmfunktion einer Alternative (Abb. 5) ist wie folgt definiert:

$$[P] = [\text{if } \boldsymbol{p} \text{ then } \boldsymbol{g} \text{ else } \boldsymbol{h} \text{ fi}] = ([\boldsymbol{p}] = \text{true} =: [\boldsymbol{g}] \vee [\boldsymbol{p}] = \text{false} =: [\boldsymbol{h}]) \quad (2)$$

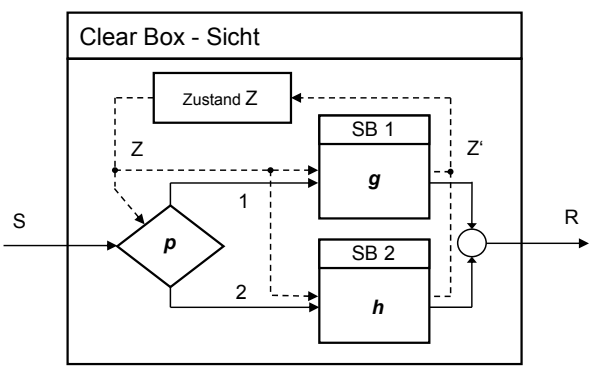

Abbildung 5: Zwei durch Alternative verknüpfte Komponenten

Für die Semi-Alternative gilt:

$$[P] = [\text{if } \boldsymbol{p} \text{ then } \boldsymbol{g} \text{ fi}] = ([\boldsymbol{p}] = \text{true} =: [\boldsymbol{g}] \vee [\boldsymbol{p}] = \text{false} =: I) \quad (3)$$

“I” ist die Identitätsfunktion. Die Programmfunktion der Alternative ist je nach Wahrheitswert des Prädikats ***p*** entweder die Programmfunktion der Komponente ***g*** oder die der Komponente ***h***. Das sind aber die tabellarisch gegebenen Übertragungsfunktionen der Komponenten. Entsprechendes gilt für die Semi-Alternative.

Die Programmfunktion der Iteration (Abb. 6) ist nur definierbar, wenn der Abbruch der Iteration gesichert ist. Unter dieser Voraussetzung lautet sie:

[P] = [while ***p*** do ***g*** od] = [if ***p*** then ***g*** **;** while ***p*** do ***g*** od fi] (4)

Wird die Schleife einmal durchlaufen, wird zunächst der Prädikatknoten mit true verlassen und ***g*** einmal ausgeführt und danach wird der Programmablauf wieder in die Schleife eintreten. Das ist aber funktionsäquivalent einer Semi-Alternative mit ***p*** im Prädikatknoten und ***g*** in Sequenz mit der Iterationsschleife im then-Zweig.

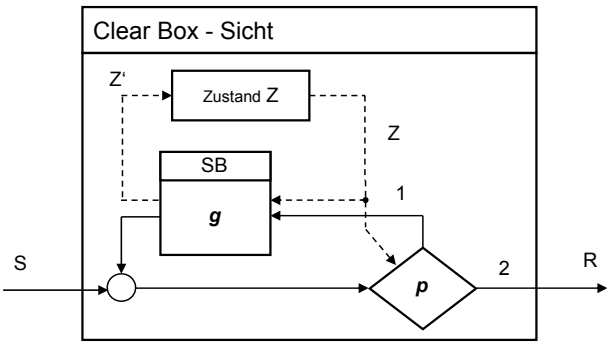

Abbildung 6: Eine Komponente in einer Iteration

Das bedeutet, die Programmfunktion einer Iteration ist nur rekursiv zu bestimmen. Sie ist, solange ***p*** true ist, durch wiederholte Funktionskomposition der Übertragungsfunktion ***g*** mit sich selbst zu ermitteln. Für ***p*** = false ist sie gleich der Identitätsfunktion I.

[P] = [if ***p*** then ***g*** **;** [P] fi] = ([***p***] = true =: [P] o [***g***] ∨ [***p***] = false =: I) (5)

Eine Architektur ist aus einer endlichen Anzahl von ineinander geschachtelten und sequenzierten Komponenten aufgebaut. Ihre Architekturfunktion kann extrahiert werden, indem mit den innen liegenden Steuerstrukturen begonnen wird und sukzessive nach außen hin die einzelnen Programmfunktionen der Steuerstrukturen ermittelt und zusammengesetzt werden. Diese Komposition der Steuerstrukturen ist, wie jede Komposition, hierarchisch geordnet. Da die Regeln dieser hierarchischen Ordnung definiert sind, ist der Extraktionsprozess prinzipiell algorithmisierbar unter den Vorrausetzungen, die bei der Iteration erfüllt sein müssen. Gleiches gilt für ein Programm in einer höheren Programmiersprache, auch hier ist die Programmfunktion extrahierbar. Statt der Übertragungsfunktionen der Komponenten gehen hier Anweisungen in die Steuerstrukturen ein.

Die bisherigen Betrachtungen zeigen, es besteht eine Äquivalenz zwischen einer mathematischen Funktion, sagen wir ***f*** und graphischen bzw. prozeduralen Regeln. Das kann in den folgenden Funktionsgleichungen formuliert werden:

f = ***g*** **;** ***h*** (6)
f = if ***p*** then ***g*** else ***h*** fi (7)
f = while ***p*** do ***g*** od (8)

Wenn ***p***, ***g***, ***h*** als gegebene Funktionen in der jeweiligen Regel betrachtet werden und ***f*** als gesuchte Funktion, drücken diese Gleichungen die Extraktions- bzw. Verifikationsaufgabe aus. Sie besteht darin, aus einer zusammengesetzten graphischen oder prozeduralen Regel (Architektur oder Programm) schrittweise für die jeweils innerste Regel die gesuchte Funktion ***f*** (Architektur- oder Programmfunktion) zu ermitteln und im Fall der Verifikation mit der Vorgabefunktion zu vergleichen.

Die entgegengesetzte Aufgabe ist die Entwurfs- bzw. Programmieraufgabe. Hier ist eine Spezifikationsfunktion ***f*** (als BB- oder SB-Spezifikation) gegeben und die Regel, wie diese Funktion graphisch oder prozedural durch ***p***, ***g***, ***h*** ausgedrückt werden kann, ist gesucht. Bekannterweise ist der erste Architekturentwurf nicht sofort geeignet, dass seine Architekturfunktion die Vorgabefunktion identisch erfüllt. Die Architektur muss schrittweise verbessert werden, bis Übereinstimmung mit der Vorgabefunktion erreicht ist. Dieses Vorgehen entspricht der iterativen Lösung der Entwurfsaufgabe, indem die Regel für den Entwurf in ihrer Zusammensetzung (Steuerkonstrukte) und in ihren dazugehörigen Konstituenten ***p***, ***g*** und ***h*** modifiziert werden. Die in iterativer Weise gefundene Architekturfunktion beschreibt vollständig und eindeutig das äußere Verhalten des Softwaresystems und erfüllt korrekt seine funktionale Spezifikation. Das Softwaresystem kann daher im Ganzen als Komponente in einem weiteren Softwaresystem genutzt werden.

Die theoretische Untersuchung des Lösungsraumes für die Gleichungen (6)-(8) wäre nützlich, um Anhaltspunkte für den Ausgangsentwurf und für den Einfluss der Konstituenten, d. h. ihre Empfindlichkeit auf die Architekturfunktion, zu erhalten.

5. Schlussbemerkungen

Die Architektur eines Softwaresystems besteht aus einer einzigen Hierarchieebene. Das ist die oberste oder nullte Ebene der Clear-Box-Sicht, in der die Struktur der Verbundkomponente durch die benutzten Steuerstrukturen festgelegt wird. Die Einbindung der Komponenten erfolgt durch Nachrichten an bestimmte Dienste der Komponenten, die selbst mit Rückgabeobjekten die Architektur bedienen. Die bekannte Ausnahme, wo keine Reaktion durch die Komponenten erfolgt, ist wenn bestimmte Datenobjekte in ihr direkt gesetzt werden. Die Komponenten einer Architektur sind ausschließlich Blattkomponenten, die aber empfehlenswerter Weise untereinander nicht durch Nachrichten verknüpft werden sollten, sondern nur mit der Architekturebene in vertikaler Richtung über Nachrichten kommunizieren. Andernfalls gehen die Vorzüge einer strengen Hierarchie verloren. Die Blattkomponenten selbst bilden somit keine eigene Ebene. Sie sind zunächst die elementaren Teilsysteme, die noch nicht weiter zerlegt wurden. Der Vorzug einer strengen Hierarchie besteht hier darin, dass jede Verfeinerung der Blattkomponenten arbeitsteilig durch unterschiedliche Entwickler simultan erfolgen kann, sobald eine korrekte Architektur vorliegt.

Das Bedürfnis nach Wiederverwendung und damit in Zusammenhang stehend, der Unempfindlichkeit gegen Änderungsauswirkungen fordert eine feiner granulierte Zerlegung der Blattkomponenten in weitere Komponenten, u. U. über mehrere

Hierarchieebenen verteilt. Dabei sollte nie, wie schon gesagt, das Prinzip der referenziellen Transparenz der Verfeinerung in streng hierarchischen Strukturen verletzt werden, da jede horizontale Verknüpfung von Komponenten die Komplexität des Softwaresystems erheblich steigert. Außerdem wird auch unerwartetes Verhalten bei der Komposition solcher horizontal und vertikal verknüpfter Komponenten verursacht. Damit eine Komponente zur Wiederverwendung geeignet ist, muss sie in den drei Sichten der Box-Struktur-Methode beschrieben sein. Diese Methode ist als Ansatz für eine komponentenorientierte Softwareentwicklung geeignet. Es ist für den Einsatz der Methode in der Praxis noch etliches zu erforschen, vor allem müssen die Sequenzbasierte Spezifikation, wie auch die Funktionsextraktion algorithmisiert werden. Nur der Einsatz des Rechners selbst kann nach diesem Ansatz zu einer Verbesserung der Qualität von komponentenorientierten Softwaresystemen führen.

Literaturverzeichnis

[SCHE84] Scheffel, H.; Schönefeld, R.: Objektorientierte Betriebssysteme – ihre Implementierung auf Kleinrechnern. Vortrag, Tagungsbericht INFO84, Dresden, 6.2.-10.2.1984.

[SCH86] Schönefeld, R.; Schrewe, I.: Objektorientierte Programmierung in C. edv-Aspekte, 5 (1986) 4, S. 34 – 37.

[SCH87] Schönefeld, R.: Datenabstraktion – ein wichtiges Prinzip der Softwaretechnik. Plenarvortrag, Kolloquium z. 25. Jahrestag des ORZ der TH Ilmenau, 11.9.1986 u. Wiss. Zeitschrift der TH Ilmenau 33 (1987) 5.

[SCH88] Schönefeld, R.: Objektorientierte Softwareentwicklung. Tagungsunterlagen, XXXIII. Intern. Wiss. Kolloquium der TH Ilmenau, Reihe B, 1988.

[SCH89] Schönefeld, R.: Warum ist C++ als Sprache für die Softwareentwicklung so bedeutsam? Wiss. Zeitschrift der TH Ilmenau 35 (1989) 2, S. 43 – 56.

[SCH90a] Schönefeld, R.: Eine objektorientierte Architektur für eine C++-Programmierumgebung (Teil 1 und 2). Wiss. Zeitschrift der TH Ilmenau, 1990, H1 und H2.

[SCH90b] Schönefeld, R.: Einfluß der objektorientierten Programmierung auf Softwaresysteme. edv-Aspekte, 9 (1990) 3, S. 57 – 60.

[KLO91] Klotz, St.; Schmidt, S.; Schönefeld, R.: Ein ECD-Dienst für die objektorientierte Programmentwicklung. Workshop “2. Dresdner Informatik-Tage“, Teil 1: Informatik und Umwelt, TU Dresden, Fak. Informatik, 4.2. – 6.2.1991.

[SCH91] Schönefeld, R.: Erfahrungen mit dem objektorientierten Paradigma,. Tagungsbericht XXXVI. Intern. Wiss. Kolloquium der TH Ilmenau, 21.10. – 24.10.1991.

[GAM95] Gamma, E. et.al: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

[SCH07] Schönefeld, R.: Theoretische Aspekte der Softwaretechnik – Softwaresysteme und Softwarekomponenten. Gastvorlesungen an der TU Ilmenau, SS 2007, http://www.tu-ilmenau.de/fakia/Theoretische-Aspekte.6107.0.html