Mehr Software – Sicherheit, mehr Software - Verlässlichkeit durch generalisierende und komponierende Abstraktion

Von Gisbert Englmeier

In Zeiten des nahezu uneingeschränkten Computereinsatzes gewinnen die Software – Sicherheit und Software – Verlässlichkeit immer mehr an Bedeutung. Dies trifft sowohl für die Neuentwicklung als auch für die Wartung und Pflege von Software – Systemen zu. In sicherheitsrelevanten, lebenserhaltenden Systemen, wie beispielsweise im medizinischen Bereich, im Bereich der Steuerung von Sicherheitssystemen in Kraftfahrzeugen, in technischen Anwendungen mit Mikroprozessunterstützung und ähnlichen Anwendungsgebieten ist es notwendiger denn je, äußerst stabile, sichere und verlässliche Software - Systeme einzusetzen. Gerade in sicherheitssensitiven Systemen sind permanent Verbesserungen der Technik zu erwarten. Die „*Halbwertzeit*“ von technischen Problemlösungen fällt ständig. Solche Änderungen in der Technik bringen in der Regel auch Software – Änderungen mit sich. Änderungen in Software – Systemen verursachen häufig Software – Fehler, weil diese Änderungen sehr oft in bestehenden Programmen durchgeführt werden und Testverfahren oft weder den notwendigen Umfang annehmen noch die notwendige Qualität besitzen, um Software - Strukturfehler systematisch zu erkennen. Die „*Bananen – Politik, das Produkt reife beim Kunden*“, kann man sich eigentlich im Bereich der sicherheitssensitiven Software nicht leisten. Die vorgeschlagene Vorgehensweise bietet die Möglichkeit, Software – Neuentwicklungen durch die Zuhilfenahme logisch – mathematischer Verfahren schnell und sicher zu strukturieren. Änderungen in Software - Systemen werden nach dem gleichen Konzept systemimmanent eingebracht und haben damit die gleiche Qualität wie bei Neuentwicklungen. Die Sicherheit und Verlässlichkeit von Software - Systemen wird durch den kompromisslosen Einsatz von logisch – mathematischen Vorgehensweisen verbessert, die bereits Kraft der solchen Konzepten innewohnenden Ordnungseigenschaften Software – Strukturen entstehen lassen, die den Zustands- und Prozessraum von Software - Systemen strukturell so begrenzen und festlegen, damit unkontrollierte Strukturen möglichst per Definition erst gar nicht entstehen können. Solche Vorgehensweisen beeinflussen auch wesentlich die Entwicklungszeit und den Testaufwand und damit die Entwicklungs- und Wartungskosten von Software – Systemen günstig.

Nach wie vor ist die Objektorientierung eines der am Besten geeigneten Konzepte, mehrdeutige, redundante Zustandsräume und mehrdeutige, redundante funktionale Systemausprägungen systematisch zu verhindern. Innerhalb der Objektorientierung trägt das Konzept der Vererbung, der Generalisierung, mit dazu bei, eindeutige Systeme mit mehr Sicherheit und Verlässlichkeit zu erzeugen. Daneben leisten lineare, minimal ausgelegte Schnittstellen zu den Anwendungsprogrammierern und auch zu den Schnittstellenbenutzern (Mensch oder Maschine) hin ihren Beitrag zur sicheren Anwendung von Software - Systemen. Neuerliche Konzepte, die empfehlen, gerade die Vererbung, die Generalisierungs-/Spezialisierungskonzepte, aus dem Methodenbereich in der Anwendungsentwicklung wegen ihrer für Programmierer oft schwer zu durchschauenden Komplexität herauszunehmen, scheinen den Sicherheitsinteressen und den Sicherheitsbedürfnissen bei der Software - Entwicklung und Wartung gerade in sicherheitssensitiven Anwendungen zuwider zulaufen. Andere Konzepte, die denen der objektorientierten Systementwicklung, speziell der Generalisierung in Gestalt des Vererbungskonzeptes zur eindeutigen systemischen Gestaltung von Software, überlegen wären, stehen nicht zur Verfügung. Systemisch wirkende Konzepte zur Erzeugung von eindeutigen Generalisierungsstrukturen sind noch nicht ausgeschöpft und können zur systemischen Gestaltung von Software genutzt werden.

Wie am Ende dieses Beitrags gezeigt wird, führen u. U. kleinste Änderungen in Eigenschaften, aus denen sich Generalisierungsstrukturen konstituieren, zu größeren Veränderungen in der Topologie dieser Generalisierungsstrukturen. Generalisierungsstrukturen und die darauf bauenden Vererbungskonzepte von Eigenschaften (Daten, Methoden) sind eigentlich die tragenden Säulen, die Fundamente der Sicherheit und Verlässlichkeit von sicherheitsrelevanten Software – Systemen, denn die eindeutige Vererbung garantiert die eindeutige Verwendung von

Daten und Methoden, eine für sicherheitsrelevante Software - Systeme vorausgesetzte Eigenschaft. Kleinste konstitutive Änderungen von Eigenschaften in Generalisierungsstrukturen wirbeln in der Regel das seither gültige Modell der Vererbungsstrukturen völlig durcheinander. Über dieses Phänomen muss man sich als verantwortlicher Entwickler im Bereich von sicherheitsrelevanter Software im Klaren sein. Mit Hilfe des Distributionsgesetzes der Booleschen Algebra, angewendet auf linear definierte Schnittstellen (Kompositionen), die als Boolesche Gleichung in disjunktiver Normalform notiert werden, lassen sich während der Systementwicklung Vererbungsstrukturen (Generalisierungsstrukturen) sicher, nachvollziehbar und vor allen Dingen eindeutig erzeugen. Diese Konzepte unterstützten die Entwicklung und Wartung von sicherheitsrelevanten Software – Systemen höchst effizient. Bei Änderungen entsteht durch Rückführung auf die bei der Erstentwicklung benutzte Gleichung der Booleschen Algebra in disjunktiver Normalform, das Einbringen dieser Änderungen in diese Gleichung und die erneute Anwendung des Distributivgesetzes auf den Ausdruck in Boolescher Algebra eine neue, jetzt optimale Vererbungsstruktur, so dass diese Systeme per Definition nicht altern. Damit bleibt auch bei System - Änderungen die Verlässlichkeit und Sicherheit der Software – Systeme in der ursprünglichen Stabilität und ursprünglichen Qualität erhalten.

Stand der Entwicklungstendenzen. In der Literatur sind Tendenzen zu beobachten, die Generalisierung zu Gunsten der Komposition aufzugeben, weil die Handhabung und Abbildung generalisierter Strukturen in den objektorientierten Programmiersprachen, insbesondere auch die Mehrfachvererbung, durch ihre Komplexität, Anwendungsentwicklern Schwierigkeiten bereiten würde [DrH]. Ein so wichtiges Ordnungsinstrument wie das der Generalisierung sollte man nicht leichtfertig aufgeben, wenn keine Konzepte zur Verfügung stehen, die nicht mindestens eine gleiche Ordnungskraft besitzen. Wie hier gezeigt wird, kann die Ordnungskraft *beider* Prinzipien, der Generalisierung sowie auch der Komposition, gemeinsam gleichzeitig genutzt werden, um Software – Systeme in der Entwicklung, Anwendung und Wartung sicherer nutzen zu können. Software-Systeme leiden nicht selten gerade in sicherheitsrelevanten Bereichen, wie der Verkehrsicherheit von Fahrzeugen oder der Medizintechnik, unter einer funktionalen Unsicherheit, die sehr oft auf mangelhaft geeignete systemische Konzepte in der Software - Konstruktion zurückzuführen ist. Dies ist oft dadurch bedingt, dass die in den Systemen mit herkömmlichen Konzepten und Methoden intuitiv definierte Zustands- und Strukturräume Zustände annehmen können, die nicht gewollt sind. Ähnlich wie bei einem Schachspiel, hat man zu Beginn eines Software - Entwicklungsprozesses noch alle Freiheitsgrade in der Systementwicklung zur Verfügung.
Im Zuge der Fortentwicklung engen sich diese Freiheitsgrade durch Festlegungen während des Entwicklungsprozesses ständig ein, bis man gegen Ende des Entwicklungsprozesses gezwungen ist, noch notwendige „Züge“ unter Missachtung und „Opferung“ bekannter Software - Sicherheits- und Software – Verlässlichkeitskonzepte zu machen, um das „Werk“ noch sinnbringend zu vollenden. Dies ist z. B. auch dann der Fall, wenn während des Entwicklungsprozesses zu berücksichtigende Eigenschaften (Daten- / Prozesselemente) zu spät entdeckt werden und in ein bereits nahezu fertig gestelltes Modell möglicherweise mit Hilfe von Overloading Funktionen eingebracht werden. Hier methodisch vermeintliche Freiheitsgrade systemisch über geeignete Strukturierungskonzepte in die richtigen Bahnen zu lenken, verleiht der Software eine erhebliche Stabilität und gewährleistet eine sichere und verlässliche Software - Statik. Beispielsweise ist es durch eine logisch - mathematische Behandlung von Vererbungsstrukturen, die auch für Mehrfachvererbungen gelten, möglich, einen in sich geschlossenen, integeren Zustands- und Strukturraum, unter den von der Aufgabenstellung her geforderten Prämissen, formal festzulegen. Herkömmlich intuitiv konstruierte Zustands- und Strukturräume sind bei Veränderungen schwierig, wenn überhaupt, integer (widerspruchsfrei) zu halten. Sie werden durch häufige Wartung oft „*verschlechtbessert*“, weil gesamtsystemisch die gewollten Zustände nicht ideal in bestehende Strukturen eingebracht werden können und deshalb teilweise nur ungenügend zu kontrollieren sind. Die tatsächlich sich neu ausprägenden Zustandsräume können nicht im notwendigen Umfang vollständig getestet werden. Die Menge der möglichen zusätzlichen Zustände und/oder wegfallende Zustände (kontrolliert oder nicht) steigt/fällt in Verbindung mit den seitherigen möglichen Zuständen durch Zustandsmengen des sich neu ergebenden kartesischen Produkts der alten und neuen Zustandsmengen und bilden einen neuen Zustandsraum.

Die „*Entropie*“ (der Grad der Unordnung, [FuH]) nimmt durch Systemänderungen in der Regel nicht ab, wenn in bestehende Systeme neue zusätzliche Funktionen aufgenommen werden. Dies kann in sicherheitsrelevanten Systemen fatale Folgen nach sich ziehen.

Überlegungen bezüglich der Zustands- und Strukturräume. Wenn man bedenkt, in welchen Größenordnungen sich ein Zustands- und Strukturraum von Systemen bezüglich der Anzahl der strukturellen Zustände aufspannen kann und welche Möglichkeiten man hat, auch nur einen minimalen Teil des Zustands- und Strukturraums von Systemen über Tests zu erreichen und welcher Aufwand entstehen würde, diesen Zustandsraum einer vollständigen Test - Enumeration zu unterwerfen, kann man die ordnende Eigenschaft eines logisch - mathematisch konstruierten Zustands- und Strukturraums bei voller Funktionalität einschätzen. Eine vollständige Enumeration eines Zustands- und Strukturraums im Test würde bei den meisten größeren Systemen in absehbarer Zeit praktisch außerhalb jeder zeitlichen Realisierungsmöglichkeit liegen. Deshalb wäre es von Vorteil, für die Konstruktion von Zustands- und Strukturräumen Vorgehensweisen einzusetzen, die nach logisch - mathematischen Konzepten die Statik von Zuständen und Strukturen bestimmen. Die Neukonstruktion von Vererbungsstrukturen lässt sich mit Hilfe einer Booleschen Gleichung in disjunktiver Normalform und der Anwendung des Distributivgesetzes der Booleschen Algebra auf eine solche Gleichung äußerst schnell und sicher gestalten. Diese Gleichung wird ihrerseits aus einer sogenannten Tabelle mit Zweckorientierten Objektsichten (Phänomenen:= beobachtete Erscheinungsformen von Objekten bzw. Substanzen := Zusammenstellungen von Eigenschaften, um bestimmte, beabsichtigte Zwecke verfolgen zu können) abgeleitet. Die Wartung bzw. die nachträgliche Veränderung von Vererbungsstrukturen, die mit Hilfe des Distributivgesetzes der Booleschen Algebra erzeugt wurden, kann systemimmanent mit dem gleichen Konzept durchgeführt werden. Damit liegt immer die optimale Struktur des Zustands- und Strukturraums der Klassen vor, die Struktur altert somit wartungsbedingt nicht. Sie entsteht bei jeder Änderung immer wieder unter den neuen Eigenschaftskonfigurationen optimal.
Durch die vorgeschlagene Vorgehensweise, Generalisierungsstrukturen mit Hilfe des Distributivgesetzes der Booleschen Algebra, angewandt auf eine Booleschen Strukturgleichung in disjunktiver Normalform, zu generieren, bringt folgende Vorteile bzw. hat folgende Merkmale:

a) Die Klassen und deren Anzahl werden durch den Prozess bestimmt.
b) Die in den Klassen befindlichen Klasseneigenschaften werden durch den Prozess zugeordnet.
c) Vererbungsstrukturen werden durch den Prozess gebildet, die Richtung der Vererbung wird ebenfalls durch den Prozess bestimmt, d. h. die hierarchische Platzierung einer Klasse im Vererbungsnetz wird durch den Prozess übernommen.
d) Bei der Anwendung des Distributivgesetzes der Booleschen Algebra sind die am häufigsten in einer Klammernebene vorkommenden Variablen zuerst auszuklammern. Damit werden die Klassen nach ihrer Häufigkeit der vorkommenden Eigenschaften (Daten, Prozesse) in der hierarchischen Struktur systemisch, so weit wie nur möglich, oben, und soweit wie nötig, unten angesiedelt, um die eindeutige Vererbung der sich selbst prozessimmanent strukturell einmalig generierenden Eigenschaften zu gewährleisten.
e) Falls bei der Anwendung des Distributivgesetzes der Booleschen Algebra bei den auszuklammernden Booleschen Variablen die Anzahl von Booleschen Variablen gleich ist, gibt es mehrere Lösungen, die aber logisch - mathematisch alle gleichwertig sind. Die Anzahl der entstehenden Klassen bei unterschiedlichen Lösungen und die Zuordnungen der Eigenschaften zu den Klassen bleiben gleich, die Anzahl der Strukturverbindungen kann variieren. Das „Vererbungsnetz“ wird „breiter“ oder „tiefer“ und kann zweckbezogen ausgewählt werden.

Alle eben genannten Ergebnisse entstehen *simultan* durch die Anwendung des Distributivgesetzes der Booleschen Algebra. Kein Ergebnis ist deshalb von einem andern technisch abhängig, kein Ergebnis ist Voraussetzung für ein anderes Ergebnis. Die hier vorgestellte Vorgehensweise geht davon aus, dass *alle* Eigenschaften neben einer *komponierenden Abstraktion* orthogonal dazu gleichzeitig auch einer *generalisierenden Abstraktion* unterzogen werden. Alle Eigenschaften werden also komponiert und gleichzeitig dazu generalisiert / spezialisiert. Diese *komponierten*, *generalisierten* / *spezialisierten* Eigenschaften können gleichzeitig auch noch zu *Aggregationen* / *Assoziationen* gehören, wie nachfolgend gezeigt werden wird.

Komplexität und Variabilität in der Anordnung von Vererbungsklassen. Die tatsächliche Verbundenheit (ausgeprägte Vererbungsstrukturen) von Systemelementen (hier Klassen) untereinander wird als Konnektivität bezeichnet. Die Komplexität *K* eines Zustands- /Strukturraums von Klassen mit Vererbungsstrukturen berechnet sich nach der aus der Kombinatorik bekannten „Kombination ohne Wiederholung“ $K = n(n\text{-}1)/2$, wobei *n* die Anzahl der Klassen ist [FuH]. Die Komplexität ist die maximal mögliche Konnektivität (maximal mögliche Verbundenheit) der Klassen untereinander. Die hier gezeigte Berechnung der Ausprägungsmöglichkeiten als Kombination ohne Wiederholung stellt die Untergrenze der Möglichkeiten dar, denn es können noch zusätzlich nicht geeignete Vererbungsrichtungen benutzt werden, die bei dieser Betrachtung nicht berücksichtigt wurden, die die Anzahl der Ausprägungen noch weiter erhöhen. Bei der als Beispiel aus den Angaben der Tabelle 2 entwickelten Struktur - Abbildung 5 gezeigten Anwendung zur Bearbeitung der Zweckorientierten Objektsichten (Phänomene) *Quadrat*, *Rechteck* und *Dreieck* haben sich durch die Anwendung der hier vorgestellten Vorgehensweise fünf Klassen ($n = 5$) ergeben. Damit errechnet sich eine Komplexität dieser fünf Klassen von $K = n(n\text{-}1)/2 = 5(4)/2 = 10$. Die Anzahl der Vererbungsstrukturen zwischen diesen fünf Klassen könnte theoretisch von keiner Vererbungsstruktur (= leere Menge der Strukturen) bis hin zu maximal zehn Vererbungsstrukturen reichen. Die in Abbildung 5 gezeigten fünf Klassen und ihre vier Vererbungsstrukturen wurden mit der hier beschriebenen Vorgehensweise eindeutig ermittelt, sie könnten aber auch intuitiv festgelegt worden sein. Wie viel Klassen und welche Vererbungsverbindungen bei intuitiver Vorgehensweise gewählt werden, hängt vom jeweiligen Entwickler ab. Die Variabilität *VA* gibt Auskunft über die Strukturverbindungsmenge des gesamten Zustands- /Strukturraums, aus dem Entwickler ihre Modelle zur Bildung von Vererbungsstrukturen für eine festgelegte Anzahl von Klassen entnehmen [FuH]. Auch die mit dieser Vorgehensweise erstellten Vererbungsstrukturen der Klassen in der Abbildung 5 werden dieser Menge entnommen bzw. sind Untermenge der Potenzmenge dieser Menge. Die Variabilität ergibt sich aus $VA = 2^{n(n\text{-}1)/2}$, dies entspricht der Potenzmenge der Menge der *n* Klassen unter der Mengenbedingung der „*Kombination ohne Wiederholung*“ [$n(n\text{-}1)/2$] als Exponent zur Basis 2. Die fünf Klassen aus dem eben genannten Beispiel ergeben eine Variabilität von $VA = 2^{5*4/2} = 2^{10} = 1.024$. Es gibt folglich bei fünf Klassen 1.024 Möglichkeiten die Klassen untereinander durch Vererbungsstrukturen zu verbinden. Angefangen mit der Möglichkeit, keine Vererbungsstrukturen zu benutzen, was einer Konstruktion ohne Vererbung entsprechen würde, über die Möglichkeit Teilvererbungsstrukturen zu benutzen und diese sowohl vererbungstechnisch als auch nichtvererbungstechnisch prozessual zu verbinden, bis hin zur Möglichkeit alles mit allem zu vererben und die „überflüssigen“ Vererbungen prozessual wieder zu neutralisieren, ist alles vorstellbar. Man möge dabei bedenken, dass Vererbungen nicht ausschließlich mit Programmiersprachen realisiert werden können, die formal die Vererbung unterstützen. Beim unterschiedlichen „Ausbildungsstand“ der Entwickler muss mit vielen benutzten Varianten gerechnet werden. Die hier vorgeschlagene Vorgehensweise mit Hilfe des Distributivgesetzes der Booleschen Algebra erzeugt im Beispiel der Abbildung 5 genau *eine* Vererbungsstruktur. In Abbildung 1 sei beispielhaft zur Veranschaulichung die vollständige Variabilität *VA* der Strukturverbindungen zwischen drei Klassen dargestellt. Bei diesen drei Klassen würde sich eine Komplexität $K = n(n\text{-}1)/2 = 3*2/2 = 3$ ergeben. Damit wären die drei maximal möglichen Strukturbeziehungen in dieser Kombination ausgeprägt, wie dies in der Figur mit dem Index 8 in Abbildung 1 gezeigt wird. Die Variabilität *VA* der theoretisch möglichen Vererbungsstrukturen der drei Klassen würde sich aus $VA = 2^{n(n\text{-}1)/2} = 2^{3(2)/2} = 2^3 = 8$ ergeben.

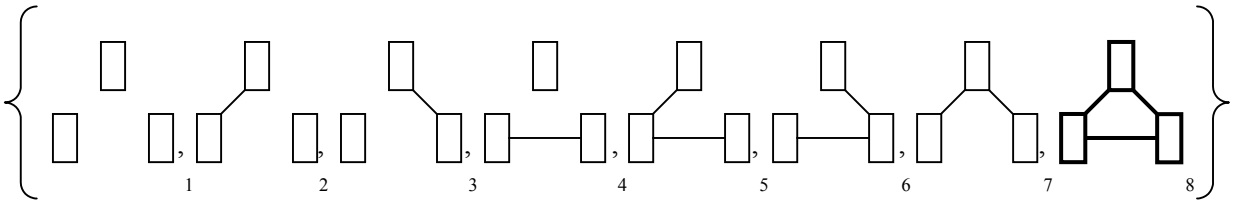


Abbildung 1 Zustandsstrukturmenge bei drei Klassen (n = 3) mit einer Variabilität $VA = 2^{n(n-1)/2} = 2^{3*2/2} = 8$ und einer Komplexität (maximalen Konnektivität) mit $K = n(n-1)/2 = 3*2/2 = 3$. Dies ergibt maximal drei Strukturverbindungen, wie sie in der letzten Struktur mit Index 8 gezeigt werden. Lösung für jede systemweite Generalisierungsabstraktion mit drei Klassen ist ein Element dieser Menge.

In Abbildung 1 sind die acht möglichen Vererbungsstrukturverbindungen, die aus drei Klassen gebildet werden können, beginnend bei der leeren Menge mit Index 1 (= keine Strukturverbindungen, keine Vererbung) bis zur Möglichkeit, dass alle drei Strukturen zwischen den drei Klassen gemäß Index 8 existieren und zur Vererbung benutzt werden, dargestellt. Welche Struktur (oder Strukturen) im Einzelfall geeignet ist, die Realität abzubilden (z. B. auch als Mehrfachvererbungsstruktur) wird durch den Entwicklungsalgorithmus bzw. den intuitiven Entwickler festgelegt. Abbildung 2 zeigt in einer Mengendarstellung beispielhaft den möglichen Strukturzustandsraum bei fünf Klassenelementen. Die Mächtigkeit dieser Menge wird durch die Berechnung der Variabilität ermittelt. Eine der 1.024 Strukturen der Menge der in Abbildung 2 gezeigten Strukturen wurde durch die hier vorgeschlagene Vorgehensweise als Lösung für die in Abbildung 5 gezeigte Struktur ermittelt: die Struktur mit dem symbolischen Index *i*.

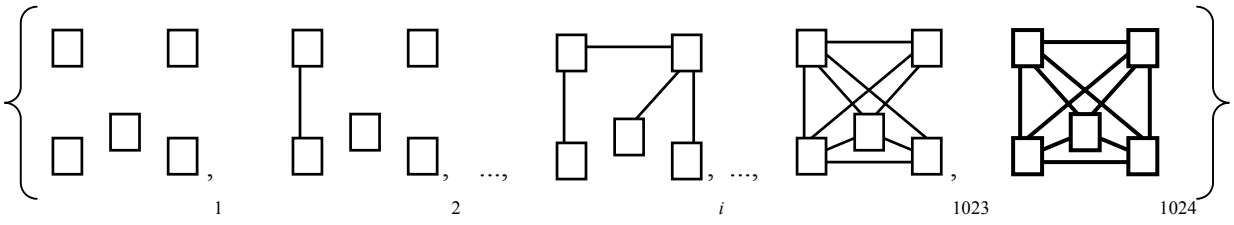


Abbildung 2 Zustandsstrukturmenge bei fünf Klassen (n = 5) mit einer Variabilität $VA = 2^{n(n-1)/2} = 2^{5*4/2} = 1024$ und einer Komplexität (maximalen Konnektivität) mit $K = n(n-1)/2 = 5*4/2 = 10$. Dies ergibt maximal zehn Strukturverbindungen, wie sie in der letzten Struktur mit Index 1024 gezeigt werden. Lösung für jede systemweite Generalisierungsabstraktion mit fünf Klassen ist ein Element dieser Menge. Die Lösung für die Vererbungsstruktur in Abbildung 5 ist das Element *i* dieser Menge mit vier Verbindungen.

Alle andern 1.023 Strukturen wurden durch den Algorithmus zur Generalisierung/Spezialisierung verworfen. Aus den theoretisch möglichen 1.024 Kombinationen ist diese Struktur nach Anwendung der hier beschriebenen Vorgehensweise durch Benutzung des Distributivgesetzes der Booleschen Algebra „berechnet“ bzw. ermittelt worden. Abbildung 3 zeigt diese Vererbungsstruktur mit Vererbungsrichtungen symbolisch.

Wie nachfolgend gezeigt werden wird, ergibt sich dieser Graph G der Vererbungsstruktur in Abbildung 3 aus dem Ausdruck $G = A(B \vee C(D \vee E))$, den man durch die Anwendung des Distributivgesetzes der Booleschen Algebra auf die Ausgangsgleichung in disjunktiver Normalform erhält, wobei „$\vee$“ dem „logisch inklusiv Oder“ entspricht.

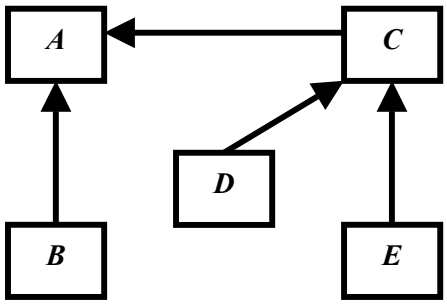


Abbildung 3 Aus theoretisch 1.024 möglichen Kombinationen ausgewählte Vererbungsstruktur**.** Die Anzahl der Kombinationen ergibt sich aus der Variabilität $VA = 2^{n(n-1)/2} = 2^{5*4/2} = 1.024$ bei fünf Klassenelementen (basierend auf dem Beispiel der Abbildung 5).

Zur Realisierung einer in Abbildung 5 geforderten Vererbungsstruktur könnten aus den 1.024 theoretisch möglichen Vererbungsstrukturkombinationen anstatt der durch den Algorithmus ermittelten Vererbungsstruktur gemäß Abbildung 3 bzw. 5, die sich über zwei Generalisierungsebenen erstreckt, intuitiv auch die beiden in Abbildung 4 gezeigten, getrennten Teilstrukturen, die gegenseitig durch *keine* formale Vererbungsstruktur verbunden sind, ausgewählt worden sein. Die Gesamtfunktionalität ist dann prozessual (gestrichelte Linie) herzustellen. Diese (intuitive) Lösung würde dann datenseitige und/oder funktionale Redundanzen beinhalten müssen. Sie hätte eine Klasse mehr, denn sie beinhaltet die Klassen A, B, C_1, C_2, D, E, wobei sich dann auch eine höhere Komplexität von $K = 6*5/2 = 15$ und eine Variabilität von $VA = 2^{6*5/2} = 2^{15} = 32.768$ ergeben würden, ein wesentlich komplexeres System. An diesem Beispiel ist gut zu erkennen, wie sich die Komplexität K und die Variabilität VA (die Anzahl der Zustände, die das zu entwickelnde System annehmen kann) rapide erhöhen, wenn die Anzahl der Klassen um eine *einzige* Klasse von fünf auf sechs erhöht wird. Der intuitive Entwickler könnte nicht mehr eine Strukturkombination aus 1.024 wählen, sondern er müsste sich jetzt für mindestens eine Strukturkombination aus 32.768 möglichen Kombinationen entscheiden. Dazu kämen die Nachteile der Redundanzen, denn die Daten und Methoden in den Klassen A und C_1 könnten nicht in den Klassen C_2, D und E benutzt bzw. vererbt werden und müssten deshalb nochmals in diesen Klassen redundant vorgegeben werden. In der gezeigten Vorgehensweise mit Hilfe des Distributivgesetzes der Booleschen Algebra ist davon auszugehen, dass bei voller Funktionalität der ermittelten Lösung immer ein Minimum der Menge der Klassen erreicht wird und gleichzeitig die notwendigen und hinreichenden Vererbungsbeziehungen gefunden werden. Tabelle 1 zeigt, wie schnell die Komplexität und die Variabilität von Systemen in Abhängigkeit der Anzahl der Klassen steigt. Hier immer die minimalen Auslegungen eines Systems bei gleicher Funktionalität zu finden, ist selbst für einen versierten intuitiven Entwickler eine schier unlösbare Aufgabe:

Anzahl Klassen n	Komplexität $K = n\,(n\text{-}1)/2$	Variablilität $VA = 2^{n\,(n\text{-}1)/2}$
1	0	1
2	1	2
3	3	8
4	6	64
5	10	1024
6	15	32768
7	21	2097152
8	28	268435456
9	36	68719476736
10	45	35184372088832
11	55	36028797018964000
12	66	73786976294838200000
…	…	…

Tabelle 1 Komplexität und Variabilität von Systemen in Abhängigkeit der Anzahl ihrer Klassen

Durch die Anwendung der hier gezeigten Vorgehensweise ist gleichzeitig sichergestellt, dass jede der Eigenschaften (Daten, Prozesse) systemweit nur einmal vorhanden ist. Wie noch gezeigt werden wird, sind für die Menge der entstehenden Klassen und für die Menge der Strukturbeziehungen und deren Richtung die gleichzeitige Anwesenheit von Eigenschaften in den zu generalisierenden Phänomenen maßgeblich bestimmend. Der Vererbungsprozess selbst bestimmt die genannten abhängigen Größen. Durch Einführung einer Eigenschaft, die in jedem Phänomen vorhanden ist, wie zum Beispiel das Erstellungsdatum der Instanziierung einer Klasse, wird erreicht, dass die gesamte Vererbungsstruktur zentral nur eine einzige Wurzel besitzt. Damit wird u. A. auch erreicht, dass die Lösung eine einzige zusammenhängende Strukturkombination darstellt, in der alle Klassen gemeinsam über ihre Strukturverbindungen verbunden sind.

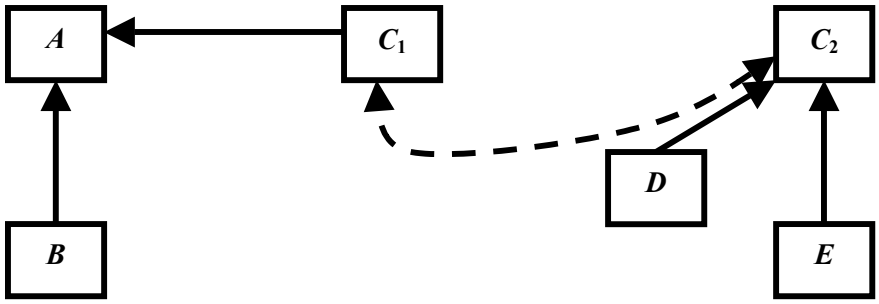


Abbildung 4 Alternativ zu Abbildung 3 sind hier zwei „intuitiv“ ausgewählte Strukturen, die getrennt als zwei Vererbungsklassen zu realisieren wären, um die Gesamtstruktur aus Abbildung 3 bzw. 5 zu ersetzen. Die Programmierung wäre im Gegensatz zur Struktur in Abbildung 3 bzw. 5 etwas aufwändiger, es wäre eine prozessuale Bindung zwischen C_1 und C_2 notwendig (unterbrochene Linie), redundante Daten- bzw. Prozesselemente in den Klassen C_1 und C_2 wären die Folge. Eine Vererbung von Daten und Methoden aus den Klassen A und C_1, wie dies in Abbildung 3 bzw. 5 möglich ist, entfällt. Außerdem hätte die Konstruktion noch eine Klasse mehr als in Abbildung 3 bzw. 5 zu sehen ist.

Beim zweiten Beispiel Tabelle 6 aus diesem Beitrag wird die Bearbeitung der drei Phänomene *Quadrat*, *Rechteck* und *Dreieck* aus Tabelle 2 um das Phänomen *unregelmäßiges Vieleck* und um weitere zusätzliche Eigenschaften erweitert. Durch die hier vorgeschlagene Vorgehensweise werden aus den Phänomenen der Tabelle 6 neun Klassen ermittelt, die untereinander zu generalisieren / spezialisieren sind. Insgesamt ergeben neun Klassen eine Komplexität von $K = n(n\text{-}1)/2 = 9(8)/2 = 36$ und eine Variabilität $VA = 2^{n(n\text{-}1)/2} = 2^{36} = 68.719.476.736$, eine enorme Anzahl von Möglichkeiten, wie sich Vererbungsstrukturen bei neun Klassen ausprägen können. Es ist leicht zu erkennen, aus welchem Potential dieses Lösungsraums intuitive Entwickler mit welchen Lösungskombinationen hier schöpfen können. Die meisten der 68.719.476.736 möglichen Strukturkombinationen bei neun Klassen werden von erfahrenen, intuitiv generalisierenden Experten sicherlich ausgeschlossen, weil sie keine zielführenden Strukturlösungen darstellen. In der hier vorgeschlagenen Vorgehensweise gibt es jedoch für das System rechnerisch genau *zwei* Lösungen (Abbildung 10 und 15) von denen eine ausgewählt werden kann. Das heißt, von den 68.719.476.736 möglichen Lösungen bei neun untereinander zu vererbenden Klassen stehen 68.719.476.734 mögliche Strukturen erst gar nicht zur Diskussion, weil sie vom Algorithmus nicht als minimale Lösungen erzeugt und damit auch nicht akzeptiert werden. Zwei Lösungen ergeben sich deshalb beim Ausklammerungsprozess, weil sich innerhalb einer Klammernebene im behandelten Beispiel mehrere verschiedene Boolesche Variablen befinden, die gleich oft zum Ausklammern vorkommen, die aber nicht zusammen in jedem auszuklammernden Term vorkommen. Jede der beiden zur Diskussion stehenden Booleschen Variablen - Terme a_8, a_9 (Gleichung 10) oder a_1, a_4 (Gleichung 17) könnte logisch - mathematisch gleichwertig zum Ausklammern benutzt werden (Ausklammerungsvarianten). In so einem Fall könnten andere Kriterien herangezogen werden, um die Reihenfolge der Booleschen Variablen des Ausklammerns zu bestimmen. Es könnte z B. die Struktur mit der geringsten Tiefe oder mit der geringsten Breite bevorzugt werden. Wie die beiden Lösungen zeigen, unterscheiden sie sich nur in der Anzahl der Vererbungsstrukturverbindungen: Die erste Lösung hat zehn (Abbildung 10), die zweite elf Verbindungen (Abbildung 15), die Anzahl der Klassen und die Zuweisung der Eigenschaften auf die Klassen ist jedoch bei beiden Lösungen völlig identisch. Die Lösungen sind vom logisch - mathematischen Gehalt her völlig gleichwertig. Nach dem ökonomischen Prinzip ist jedoch davon auszugehen, dass, bei logisch – mathematisch gleichwertigen Lösungen bei gleicher Funktionalität, diejenigen mit dem geringsten strukturellen Aufwand am wirtschaftlichsten arbeiten, da die sie weniger komplex sind. Kommen bei einem Ausklammerungsprozess zu viele gleiche Anzahlen von Booleschen Variablen vor, so dass der Suchprozess zu lange dauern würde, kann eine suboptimale Lösung auch experimentell (über eine Simulation) betrieben werden.

Dieser soeben gezeigte Vergleich der Zustands- und Strukturräume macht insgesamt deutlich, in welchen ungeheueren Freiheitsgraden sich Entwickler bei der Gestaltung der Zustands- und Strukturräume für das zu entwickelnde System in der Auswahl der Vererbungsstrukturen befinden und wie sie sich gleich zu Beginn der Entwicklung durch einen „falschen Zug“ in der Gestaltung von Vererbungsstrukturen Möglichkeiten, die gegen Ende der Entwicklung noch interessant wären, „verbauen“ und einengen können. Andererseits zeigt der Vergleich die ordnende und zielgestaltende Kraft von logisch - mathematischen Vorgehensweisen, wie sie nachfolgend beschrieben werden, und den damit verbundenen Gewinn an Software - Sicherheit und Software – Verlässlichkeit und an „Berechenbarkeit solcher Strukturen“ gerade in sicherheitsrelevanten Einsatzgebieten von Software, eben durch eine logisch - mathematisch kontrollierte Strukturierung. Es bilden sich nur Lösungsmöglichkeiten im Rahmen der logisch - mathematischen Struktur nach den Regeln des Distributionsgesetzes der Booleschen Algebra aus, wobei häufiger vorkommenden Eigenschaften tendenziell im Vererbungsnetz weiter „oben“ angesiedelt werden, denn die am häufigsten vorkommenden Booleschen Variablen sind zuerst auszuklammern. Die so ermittelten Lösungen haben gegenüber allen andern Lösungen den Vorteil, dass es sich bei diesen Lösungen um Ausdrücke nach formalen Regeln des Distributivgesetzes der Booleschen Algebra handelt, die dann auch die ihnen zugeschriebenen Eigenschaften besitzen: Jede Klammernebene ist bei geschachtelten Klammern mit ihren übergeordneten Klammernebenen logisch - multiplikativ verknüpft. Jede Boolesche Variable innerhalb eines Klammernpaares ist mit den andern sich darin befindlichen Booleschen Variablen untereinander logisch additiv verknüpft.

Ausdrücke von mehreren Klammernpaaren in der gleichen Ebene stehen zueinander orthogonal angeordnet. In der UML – Methode werden solche Strukturen Diskriminatoren genannt. Minimale, „eineindeutige“ und „berechenbare“ Systeme besitzen allemal weniger Komplexität und damit weniger „Gelegenheit“ Zustandsfreiheitsgrade zu bieten, um unkontrollierte bzw. schlecht zu kontrollierende Zustände anzunehmen. Außerdem ist davon auszugehen, dass so ermittelte Vererbungsstrukturen vom Entwicklerteam akzeptiert werden, denn diese Strukturen lassen sich jederzeit für alle am Entwicklungsprozess Beteiligten „algorithmisch“ wiederholen. Das bedeutet, unterschiedliche Entwickler kommen bei der gleichen Aufgabenstellung zu gleichen Lösungen. Es lassen sich lange und endlose „Expertendiskussionen“ vermeiden, was neben dem Gewinn an Sicherheit und Verlässlichkeit auch erhebliche Zeit- und damit Kostenersparnisse mit sich bringt.

Softwaresicherheit und Verlässlichkeit. Software – Sicherheit, Software - Verlässlichkeit und Kostenminimierung während der Entwicklung und während des Betriebs von Software – Systemen, insbesondere kurze Entwicklungs- und kurze Wartungszeiten, sind heute mehr denn je Forderungen an Software produzierende Prozesse. Insbesondere kommt der Betriebssicherheit von Software eine immer größere, nicht zu unterschätzende Bedeutung zu. Immer mehr Software wird in sicherheitsrelevanten Bereichen, wie z. B. in Kraftfahrzeugen oder auch im Bereich der Gerätemedizin eingesetzt. Hier kann der Einsatz logisch - mathematisch formaler Methoden die Sicherheit und Verlässlichkeit fördern. Die Eigenschaften logisch - mathematischer Prinzipien und Konzepte wirken in entsprechend konstruierten Software - Systemen sozusagen gemäß Definition. Diese logisch - mathematischen Verfahren lassen bei gleichen Aufgabenstellungen gleiche Softwarekonstruktionen mit gleichen Ergebnissen auch beim Einsatz unterschiedlicher Software – Konstrukteure erwarten. Nach wie vor erfordert sichere Software unausweichlich die Minimierung von zustandsseitiger- und funktionaler Redundanz. Ein abzubildender zu organisierender Phänomenbereich der Realität lässt sich durch eine adäquate Systemgestaltung mit Hilfe von geeigneten Gestaltungsprinzipien nach gewünschten, erwarteten Leistungsmerkmalen mit erwarteten Wirkungen organisieren.

Gestaltungsprinzipien. Zwei Gestaltungsprinzipien der *modellbildenden Abstraktion* sind dazu geeignet, eine eindeutige Ordnung bei minimaler Auslegung und minimalem Kommunikationsbedarf im Entwickler - System zu etablieren: die *generalisierende Abstraktion*, die aus der Sicht des Systemgestalters *alle* Eigenschaften im Gesamtsystem *einmalig* und *eindeutig* ordnet und die *komponierende Abstraktion*, die die Eigenschaften eines Systems eindeutig aus der Sicht des Anwendungsentwicklers und auch aus der Sicht des Benutzers ordnet [JuH]. Beide Gestaltungsprinzipien sind dabei gleichzeitig, das heißt parallel nebeneinander, nicht alternativ, auf ein und denselben Phänomenbereich, anzuwenden, denn dies eröffnet die Vorteile beider Prinzipien für eine effiziente Systemgestaltung. Die *generalisierende Abstraktion* erbringt aus der Sicht der Software - Konstruktion *systemweit eindeutige*, also redundanzfreie Systemelemente, die *komponierende Abstraktion* ordnet aus der Sicht der Benutzer genau dieselben Systemelemente in ihrem Benutzungszusammenhang in einer zweckgerichteten Benutzerabstraktion. *Alle Eigenschaften*, sind einer systemweiten *generalisierenden* Abstraktion zu unterziehen. Gleichzeitig sind sie benutzerorientiert als *komponierende* Abstraktion auszubilden. Dies sind, sozusagen, zwei Seiten ein und der selben „*Medaille*“.
Die zu organisierenden Eigenschaften zerfallen also *nicht* in zwei Gruppen, wobei die einen sich in *generalisierenden Klassen* und die andern sich in den *Klassen* von *Assoziationen* oder *Aggregationen* wieder finden. Getrennte Organisationsformen fördern zustandsseitige- und funktionale Redundanzen, die ihrerseits ganz erhebliche Risiken der Zuverlässigkeit von Programmsystemen begünstigen. Deshalb ist die Forderung evident, jede Eigenschaft uneingeschränkt der generalisierenden *und* der komponierenden *Abstraktion* ohne Ausnahme zu unterwerfen. Beide Abstraktionsarten sind zueinander als orthogonal anzusehen. Folgendes Beispiel zeigt eine solche, zweifache, zueinander orthogonale Ordnung in einer *generalisierenden* und gleichzeitig *komponierenden Abstraktion*:

Generalisierende Abstraktion: 1) „*Jacke, Weste, Hose*“ **sind** „*Kleidungsstücke*“.

Komponierende Abstraktion: 2) „*Jacke, Weste, Hose*“ **bilden** einen „*Anzug*“.

Der Phänomenbereich wird hier durch die drei Begriffe „*Jacke, Weste, Hose*“ repräsentiert. Nach dem Generalisierungsprinzip werden diese Phänomene im Begriff „*Kleidungsstücke*“ aus der *Sicht* der *Konstrukteure verallgemeinert* (generalisiert), einer Menge mit dem Namen „*Kleidungsstücke*“ zugeordnet. Nach dem Prinzip der *Komposition* werden die gleichen Phänomene aus der Sicht derer, die sie anwenden, gemeinsam nutzen, zu einem „*Anzug*“ *komponiert* (aggregiert, zusammengefasst). Die *komponierende Abstraktion* ist die Sicht derjenigen, die Objekte aus dem Phänomenbereich zweckorientiert benutzen, zweckorientiert anwenden wollen. Der Benutzer kauft sich ein „*Aggregat Anzug*“, bestehend aus den Komponenten „*Jacke, Weste, Hose*“ und wendet dieses „*am Zweck orientiert an*“, nämlich um sich zu bekleiden. Der Benutzer möchte eine „*Ausprägung*“, ein „*Exemplar Anzug*“ benutzen, um sich zu kleiden. Dafür schafft der Konstrukteur eine lineare Benutzerschnittstelle. Der Konstrukteur selbst orientiert sich primär an den Kleidungsstücken und versucht bei seiner Konstruktion in einer Analyse Gemeinsamkeiten durch die *generalisierende Abstraktion* festzustellen, die er, wenn möglich, nur einmal beschreiben und in der gleichen Art konstruieren möchte: Was haben Hosen oder Jacken oder Westen jeweils gemeinsam, um diese gemeinsamen Erkenntnisse nur einmal zu bearbeiten, zu installieren, wie beispielsweise die Verfahrensweise zur Einarbeitung von Ärmeln in Jacken (z. B. bei Damenjacken, Herrenjacken, Kinderjacken) aller Kleidungsstücke der gleichen Art.

Generalisierende und komponierende Abstraktion. Beide Prinzipien der *generalisierenden* und *komponierenden Abstraktion* werden auf den identischen Phänomenbereich („*Jacke, Weste, Hose*“) *gleichzeitig*, zueinander *orthogonal*, angewendet. Sie werden ***nicht*** im Sinne, „***entweder** gehören die Merkmale, Eigenschaften des Phänomenbereichs zu einer Klasse der generalisierenden **oder** zu einer Klasse der komponierenden Abstraktion*“. Beide Gestaltungsprinzipien entfalten ihre Ordnungskraft und Ordnungseigenschaft *gleichzeitig* auf den Phänomenbereich, sich gegenseitig *ergänzend* und *nicht* sich gegenseitig *ausschließend*. Es zeigt sich, dass in den linearen Schnittstellen der *komponierenden Abstraktion*, wie sie der Benutzer für seine Sicht benötigt und wie sie auch bevorzugt vom Anwendungsprogrammierer benutzt wird, bereits die Informationen zur Strukturierung der *generalisierenden Abstraktion* enthalten sind. Durch eine geeignete Aufbereitung ist es damit möglich, aus den Strukturdaten der *komponierenden Abstraktion* unmittelbar direkt die Vererbungsstrukturen durch einen einfachen logisch - mathematischen Prozess abzuleiten.
Die Strukturdaten der *komponierenden Abstraktion* werden dazu in disjunktiver Normalform der Booleschen Algebra notiert. Anschließend werden die in disjunktiver Normalform befindlichen Booleschen Ausdrücke mit Hilfe des Distributivgesetzes der Booleschen Algebra bearbeitet. Als Ergebnis erhält man direkt eindeutige, jederzeit logisch - mathematisch nachvollziehbare Generalisierungsstrukturen, die unmittelbar als Vererbungsstrukturen Verwendung finden. In den Generalisierungsprozess werden sowohl Daten- bzw. Zustandsvariable als auch Variablen verändernde Prozesse (Methoden) einbezogen. Allein durch die in den *komponierenden Abstraktionen* implizit befindlichen Strukturinformationen können die in disjunktiver Normalform notierten Strukturen durch die Anwendung des Distributivgesetzes der Booleschen Algebra direkt in die Vererbungsstruktur überführt werden. Damit wirken die diesem logisch - mathematischen Konzept des Distributivgesetzes der Booleschen Algebra innewohnenden Strukturierungseigenschaften unmittelbar auf die Form der Vererbungsstrukturen und damit auf die Stabilität des Gesamtsystems. Diese so erzeugten Strukturen sind den intuitiv erzeugten Strukturen gegenüber, vor allem bei einer größeren Menge von Eigenschaften, systematisch von Vorteil, denn die formalen Eigenschaften der Ordnungskraft des Distributivgesetzes der Booleschen Algebra wirken sich sozusagen per Definition auf die zu erzeugende Struktur, ohne dass es zu unterschiedlichen Ausprägungen des Gestaltungsraums durch intuitiv arbeitende Individuen kommt. Die ordnenden Eigenschaften des logisch - mathematischen Konzepts des Distributivgesetzes der Booleschen Algebra werden zur eindeutigen Bestimmung der Vererbungsstrukturen genutzt.

Gleiche Aufgabenstellungen führen damit durch unterschiedliche Bearbeiter zu gleichen Ergebnisstrukturen. Es werden damit die Eigenschaften in gleicher Qualität strukturiert und in den Vererbungsstrukturen im Sinne der Objektorientierung abgebildet. Dies geschieht in Form einer sicheren, eindeutigen, verlässlichen und für alle am Entwicklungsprozess Beteiligten nachvollziehbaren Art und Weise.

Zweckorientierte Objektsicht als Ausgangsbasis. Die folgenden Ausführungen sollen zeigen, wie beide Konzepte gleichzeitig einzusetzen sind und deren Ordnungsfähigkeit zu nutzen ist. Als Ausgangstableau, sowohl für die *komponierende* als auch für die *generalisierende Abstraktion*, kann eine Tabelle der *Zweckorientierten Objektsichten* dienen. Diese Tabelle enthält bereits die Strukturdaten sowohl für die *komponierende* als auch für die *generalisierende Abstraktion*. In der jeweiligen Tabellen - Vorspalte befinden sich die Eigenschaften des Phänomenbereichs in Form von Daten- und/oder Prozessattributen. In den Hauptspalten befinden sich die Sichten auf die Objekte für die Zwecke des Benutzers. Im Sinne der Ontologie können diese Sichten auch als Phänomene (sich den Sinnen darbietende Erscheinungsformen, sich dem Erkenntnisprozess darbietende Bewusstseinsinhalte [DUD]) bezeichnet werden. Die Sichten des Benutzers sollen im Beispiel der Tabelle 2 auf die drei Objekte (Phänomene) *Quadrat*, *Rechteck* und *Dreieck* gerichtet sein. Das ist die „Welt" des Benutzers, die „Welt" seiner Phänomene. Der Zweck für den Benutzer in diesem Beispiel ist die Berechnung der Flächen der drei Sichten auf die Objekte (Phänomene). Die Eigenschaften der Vorspalte sind den Zweckorientierten Objektsichten (den Phänomenen) über ein (+) - Zeichen zuzuordnen.
Bildet man über der Menge aller Eigenschaften (Phänomenbereich) der Vorspalte einer Tabelle der Zweckorientierten Objektsichten die Potenzmenge, dann ist jede Zweckorientierte Objektsicht (Hauptspalten) in dieser Tabelle Teilmenge dieser Potenzmenge. Für die Anwendung der vorgeschlagenen Vorgehensweise sollten mindestens zwei Teilmengen dieser Potenzmenge (ohne die leere Menge) benutzt werden. Die in die Vorgehensweise einbezogenen Teilmengen untereinander sollten bezüglich ihrer Elemente (Eigenschaften) Schnittmengen besitzen.
Die Spalte zwei (*BV*) stellt die Eigenschaften als Boolesche Variable in abgekürzter Form „a_i, p_i" dar, um sie später im Booleschen Ausdruck einfacher handhaben zu können. Mit „a_i" werden bevorzugt Datenattribute, mit „p_i" bevorzugt Prozessattribute bezeichnet. Als Boolesche Variable für den Bearbeitungsprozess könnten ebenso die Langnamen des Phänomenbereichs in Spalte eins der Tabelle benutzt werden.

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften, Methoden)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen, Phänomene) *Quadrat* S_1	 *Rechteck* S_2	 *Dreieck* S_3	Anzahl
	1	2	3	4	5	6
1	*Seite a*	a_1	+	+	+	3
2	*Seite b*	a_2		+	+	2
3	*Seite c*	a_3			+	1
4	*Fläche F*	a_4	+	+	+	3
5	*Halbe Seiten s*	a_5			+	1
6	$(a > 0) = 1$	p_1	+			1
7	$(a <> b)(a > 0)(b > 0) = 1$	p_2		+		1
8	$(a+b>c)(a+c>b)(b+c>a) = 1$	p_3			+	1
9	$F = a * a$	p_4	+			1
10	$F = a * b$	p_5		+		1
11	$s = (a + b + c) / 2$	p_6			+	1
12	$F = [s (s - a)(s - b)(s - c)]^{0,5}$	p_7			+	1

Tabelle 2: Zweckorientierte Objektsichten (Phänomene) – komponierende Abstraktionen

Der Bereich der Eigenschaften (Phänomenbereich) ist der Spalte eins der Tabelle 2 zu entnehmen. In den Spalten drei, vier und fünf findet man drei *komponierende Abstraktionen* (Kompositionen, Phänomene), wie sie der Benutzer für seine Zwecke sehen möchte. Er möchte aus der Menge der Eigenschaften des Phänomenbereichs die Kompositionen *Quadrat*, *Rechteck* oder *Dreieck* bilden. Es sind dies am Zweck orientierte Sichten auf Objekte, *Zweckorientierte Objektsichten*. In diesen drei Kompositionen (Zusammensetzungen) sollen im Beispiel der Tabelle 2 die Menge der Eigenschaften des Phänomenbereichs als Kompositionselemente benutzt werden. Die jeweiligen Eigenschaften der drei Zweckorientierten Objektsichten (Phänomene) *Quadrat*, *Rechteck* und *Dreieck* sind drei Mengen der Potenzmenge aller Eigenschaften des Phänomenbereichs.
Der Benutzer möchte jeweils für seine Zwecke die Fläche eines Exemplars (*Quadrat*, *Rechteck*, *Dreieck*) berechnen. Die einzelnen Objekte (Phänomene) zeichnen sich durch den Besitz ihrer Eigenschaften aus, sie sind aus diesen Eigenschaften *komponiert*. Der Konstrukteur analysiert gleiche Eigenschaften, gleiche Elemente verschiedener Figuren (verschiedener Sichten, Phänomene) und vereinheitlicht die Verfahrensweisen für gleiche Eigenschaften, gleiche Elemente, wie z. B. die „*Seite a*“, die alle Figuren (Phänomene) besitzen. Die „*Seite b*“ besitzen nur das *Rechteck* und das *Dreieck*. Über eine besondere Verfahrensweise wird die *generalisierende Abstraktion* vollzogen. In der Tabelle der Zweckorientierten Objektsichten sind die Strukturinformationen für beide Abstraktionsarten, der *komponierenden* und der *generalisierenden Abstraktion*, bereits implizit enthalten. Die Tabelle der Zweckorientierten Objektsichten selbst enthält sozusagen die Kompositionen per Definition: Die Hauptspalten enthalten die Sichten der Benutzer, indem dort die Eigenschaften, die in der Vorspalte referenzt sind, zu jeder Benutzersicht markiert sind. Notiert man die markierten Eigenschaften jeder der Hauptspalten als Boolesche Konjunktion und notiert man alle Konjunktionen als disjunktive Normalform, dann kann man diese Disjunktion mit Hilfe des Distributionsgesetzes der Booleschen Algebra einem Ausklammerungsprozess unterwerfen, der genau die eindeutige Vererbungsstruktur erzeugt. Die Ausklammerung ist damit eine Strukturtransformation der in der Tabelle enthaltenen Strukturinformationen auf eine alternative Darstellungsform, in der jede Eigenschaft nur ein einziges Mal formal im Gesamtsystem erscheint. Die Verbindung dieser nur einmal im Gesamtsystem erscheinenden Eigenschaften untereinander wird über Strukturverbindungen (die Klammernebenen) dargestellt. Die Generalisierung ist eine andere Form der inhaltlich absolut identischen Strukturdaten, die sich aus der komponierenden Darstellung der Zweckorientierten Objektsicht ergibt. Es sind zwei verschiedene Seiten ein und derselben Medaille. Es ist die Überführung eines Koordinatensystems in ein anderes, in dem die Basis gewechselt wird. Im ersten System bilden die Zweckorientierten Objektsichten die Basis, in dem die Eigenschaften des Phänomenbereichs die Koeffizienten darstellen. Jede Zweckorientierte Sicht ist nur einmal vorhanden, während eine Eigenschaft mehreren Zweckorientierten Sichten zugeordnet sein kann. Durch den Wirkungsmechanismus des Distributivgesetzes der Booleschen Algebra wird jede mehrfach in Zweckorientierten Objektsichten vorkommende Eigenschaft auf ein im Gesamtausdruck einmaliges Vorkommen im sich ergebenden Klammernausdruck reduziert. Dieser sich ergebende Klammernausdruck stellt gleichzeitig die sich ergebende Generalisierungs-/Spezialisierungsstruktur dar, die bei Mehrfachvererbungen bezüglich der vorkommenden Mehrfacheigenschaften noch zur Einmaligkeit konsolidiert werden muss. Die Transformation kann jederzeit aus dem generalisierten Zustand durch Logisches Ausmultiplizieren der entstandenen Klammernausdrücke wieder auf den *komponierten Zustand* des Ursystems zurückgeführt werden. Es entstehen wieder die ursprünglichen Konjunktionen, die spaltenweise aus den Zweckorientierten Objektsichten zur Booleschen Gleichung in disjunktiver Normalform entwickelt wurden.

Komponierende Abstraktion. Die *komponierende Abstraktion* befasst sich mit dem Zusammenbau von Merkmalen zu jeweils einer zweckbezogenen Objekt – Komposition mit Phänomencharakter. Tabelle 2 enthält drei komponierende Abstraktionen (drei Phänomene): *Quadrat*, *Rechteck* und *Dreieck*. Sie sind in den Spalten dieser Tabelle mit den hier interessierenden Eigenschaften (+) abgebildet. So wird beispielsweise die Komposition *Quadrat* aus den Eigenschaften „*Seite a*“ und „*Fläche F*“ und den Rechen- / Prozess - Vorschriften „$(a > 0) = 1$“ und „$F = a * a$“ gebildet. Die Komposition *Rechteck* wird aus den Eigenschaften „*Seite a*“, „*Seite b*“ und „*Fläche F*“ und den Vorschriften „$(a <> b)(a > 0)(b > 0) = 1$“ und „$F = a * b$“ gebildet.

Die Komposition *Dreieck* wird gebildet aus den Eigenschaften „*Seite a*“, „*Seite b*“, „*Seite c*“, „*Fläche F*“ und den „*Halbe Seiten s*“ und von den Vorschriften „$(a + b > c)(a + c > b)(b + c > a) = 1$“ (:= Vorschriften der Dreiecksungleichungen), „$s = (a + b + c) / 2$“ (:= um nach Heron aus den drei Seiten eines Dreiecks die Fläche ermitteln zu können) und „$F = [s\ (s - a)(s - b)(s - c)]^{0,5}$“ als Flächenermittlung. In der *Komposition* ist eines der beiden Konzepte zu sehen, um den Phänomenbereich benutzerorientiert zu beschreiben. Eine Komposition *bildet*, *baut*, *gruppiert* gleichsam die Konstrukte durch Bausteine, Eigenschaften, mit denen sich Benutzer am „*Zweck orientiert*“ befassen wollen. Dies sind die Sichten, mit denen sich der Benutzer in seiner Oberfläche, in seiner Sicht der Dinge, beschäftigen möchte, wenn er sich diesen Themen nähert. Aus Gründen der Übersichtlichkeit sind nur die sachbezogenen strukturbildenden Merkmale und Prozesse in das Beispiel einbezogen. Programmtechnische Konstrukte und Details, wie z. B. „*ObjektErzeugen( )*;“, Datentyp „*integer*“ u. Ä, wurden aus Gründen der Übersichtlichkeit weggelassen, sie werden von Programmierern später im „program listing“ sicherlich als solche erkannt werden, sie können aber jederzeit mit in das Verfahren einbezogen werden.

Generalisierende Abstraktion. Im Gegensatz zur *komponierenden Abstraktion* befasst sich die *generalisierende Abstraktion* damit, identisch gleiche vorkommende Eigenschaften in verschiedenen Phänomenen, wie sie in der Vorspalte der Tabelle der „*Zweckorientierten Objektsichten*“ zu sehen sind, zu erkennen, um sie nur ein einziges Mal zu be- und umschreiben. Um die *generalisierende Abstraktion* auf die zur disjunktiven Normalform der Booleschen Algebra formalisierten Phänomene anzuwenden, kann die vom Autor dieses Beitrags in seinem Buch „Objektstrukturen – praxisbezogenes Konzept der Attribut-/Methodenvererbung“ [EnG] oder in der Zeitschrift „Objekt Fokus“ vom 11/12 1998 unter dem Titel „Exakte Klassenzuordnung der Attribute – Reproduzierbare Vererbungsstrukturen“ beschriebene Vorgehensweise benutzt werden. Dieses Vorgehen sei hier kurz demonstriert. Man bildet aus den Spalten drei bis fünf der Kompositionen in Tabelle 2 für jede Spalte einen Ausdruck in Form einer *Konjunktion* (= logische Multiplikation) von Booleschen Variablen, ordnet die so entstandenen Konjunktionen in einer Disjunktion (= logische Addition) in disjunktiver Normalform an, und wendet darauf das Distributivgesetz der Booleschen Algebra an. Um den Ausdruck etwas kürzer zu gestalten, werden nicht die langen Bezeichnungen im Phänomenbereich benutzt, sondern die Bezeichnungen in Spalte zwei der Tabelle 2 (*BV* = Boolesche Variable), wobei a_i Variablen - Namen für Datenattribute und p_i Variablen - Namen für Prozessattribute (Methoden, Vorschriften) benutzt werden. Jede Eigenschaft wird als Boolesche Variable aufgefasst, die prinzipiell den Wert „0“ oder „1“ annehmen kann. In den Booleschen Ausdruck einer „*Konjunktion*“ der Vorgehensweise werden nur Eigenschaften aufgenommen, die in den Hauptspalten (in den Zweckorientierten Objektsichten) mit „+“ markiert sind, sie repräsentieren den Wert „1“. Dadurch ergibt jede Konjunktion den Wert 1. Nicht benutzte Eigenschaften in den Hauptspalten wird der Wert „0“ zugeordnet. Sie werden *nicht* in die Konjunktionen mit aufgenommen, sie würden das gesamte logische Produkt zu „0“ werden lassen.
Es folgen die zu *Booleschen Konjunktionen* zusammengefassten drei Kompositionen („*Zweckorientierte Objektsichten*“) *Quadrat*, *Rechteck* und *Dreieck*, wie sie in den Hauptspalten drei, vier und fünf der Tabelle der „*Zweckorientierten Objektsichten*“, Tabelle 2, vorkommen:

Quadrat	$= S_1$	$= a_1\ a_4\ p_1\ p_4$	= 1*1*1*1	= 1.
Rechteck	$= S_2$	$= a_1\ a_2\ a_4\ p_2\ p_5$	= 1*1*1*1*1	= 1.
Dreieck	$= S_3$	$= a_1\ a_2\ a_3\ a_4\ a_5\ p_3\ p_6\ p_7$	= 1*1*1*1*1*1*1*1	= 1.

Diese Konjunktionen werden zu einer Disjunktion in *disjunktiver Normalform* zusammengefügt, in dem sie untereinander *inklusiv - logisch - Oder* („$\vee$“) verknüpft werden:

Gleichung 1

$$S_1 \vee S_2 \vee S_3 = \boldsymbol{a_1}\ \boldsymbol{a_4}\ p_1\ p_4 \vee \boldsymbol{a_1}\ a_2\ \boldsymbol{a_4}\ p_2\ p_5 \vee \boldsymbol{a_1}\ a_2\ a_3\ \boldsymbol{a_4}\ a_5\ p_3\ p_6\ p_7 = 1$$

Klammert man in diesen Ausdruck die Booleschen Variablen $\boldsymbol{a_1}$ $\boldsymbol{a_4}$, die in allen drei Teiltermen (Konjunktionen) vorkommen, aus, erhält man:

Gleichung 2

$$\boldsymbol{a_1}\ \boldsymbol{a_4}\ (p_1\ p_4 \vee\ \boldsymbol{a_2}\ p_2\ p_5 \vee\ \boldsymbol{a_2}\ a_3\ a_5\ p_3\ p_6\ p_7) = 1$$

Weiter ergibt sich durch Ausklammern der Booleschen Variablen $\boldsymbol{a_2}$ innerhalb der Klammer der folgende Ausdruck:

Gleichung 3

$$\boldsymbol{a_1}\ \boldsymbol{a_4}\ (\boldsymbol{p_1}\ \boldsymbol{p_4} \vee\ \boldsymbol{a_2}\ (\boldsymbol{p_2}\ \boldsymbol{p_5} \vee\ \boldsymbol{a_3}\ \boldsymbol{a_5}\ \boldsymbol{p_3}\ \boldsymbol{p_6}\ \boldsymbol{p_7})) = 1$$

Damit ist die *generalisierende Abstraktion* durchgeführt, jedes Attribut (jedes Datum, jeder Prozess) kommt nur einmal im Gesamtsystem vor. In der objektorientierten Systementwicklung kann die soeben erhaltene Struktur die Vererbung von Klassen abbilden. Eine Klassennamenzuordnung der Booleschen Variablen in Gleichung 3 möge folgende Notation ergeben:

GenGeometrie	=	$a_1\ a_4$	= 1 (Generalisierte Klasse Geometrieobjekt).
GenQuadrat	=	$p_1\ p_4$	= 1 (Spezialisierte Klasse Quadrat).
GenNichtQuadrat	=	a_2	= 1 (Generalisierte/spezialisierte Klasse Nicht-Quadrat).
GenRechteck	=	$p_2\ p_5$	= 1 (Spezialisierte Klasse Rechteck).
GenDreieck	=	$a_3\ a_5\ p_3\ p_6\ p_7$	= 1 (Spezialisierte Klasse Dreieck).

Aus der obigen Klammernstruktur der Gleichung 3 lässt sich der Vererbungsstruktur direkt ableiten. Jede „*Klammer – auf*“ eröffnet eine Generalisierungsebene, jede „*Klammer – zu*“ schließt eine Generalisierungs-/Spezialisierungsebene. Unter Beachtung der soeben zugeordneten Klassennamen zu den ausgeklammerten Termen der Booleschen Variablen ergibt sich folgende Ausklammerungs- bzw. Vererbungsstruktur:

Gleichung 4

$$a_1\ a_4\ (p_1\ p_4 \vee\ a_2\ (p_2\ p_5 \vee\ a_3\ a_5\ p_3\ p_6\ p_7)) =$$

GenGeometrie **(***GenQuadrat* $\vee$ *GenNichtQuadrat* **(***GenRechteck* $\vee$ *GenDreieck***)**) = 1

Daraus lässt sich die Vererbungsstruktur Abbildung 5 als grafische Darstellung direkt ableiten. Jede „***Klammer auf***“ eröffnet eine Spezialisierungsebene, jede „***Klammer zu***“ schließt eine solche Ebene:

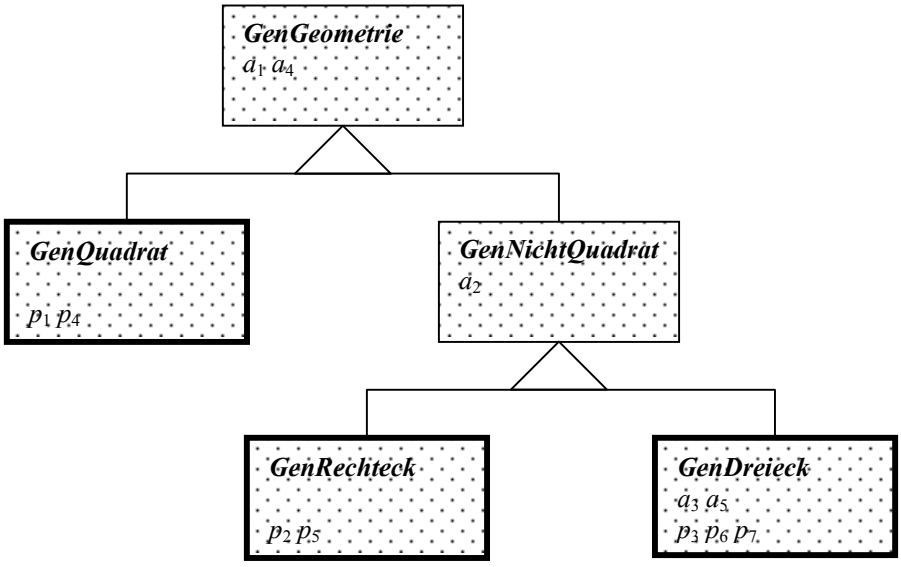


Abbildung 5: Vererbungsstruktur der *generalisierenden Abstraktion* der Gleichung 4 in Form einer graphischen Darstellung.

Damit ist die *generalisierende Abstraktion* abgeschlossen, jedes Element kommt nur einmal im Gesamtsystem vor. Hier seien noch einmal die beiden Abstraktionsprinzipien am konkreten Beispiel dargestellt:

1) Komponierende Abstraktionen:
   a) „*Quadrat*“, „*Rechteck*“, „*Dreieck*“ **bilden** „*Flächen*“.
   b) „*Seite a*“, „*Fläche F*“, „(a>0)“ = 1“, „$F = a * a$“ **bilden die** „*Fläche eines Quadrats*“.
   c) „*Seite a*“, „*Seite b*“, „*Fläche F*“, „($a <> b$)($a > 0$)($b > 0$) = 1“, „$F = a * b$“ **bilden die** „*Fläche eines Rechtecks*“.
   d) „*Seite a*“, „*Seite b*“, „*Seite c*“, „*Fläche F*“, „*Halbe Seiten s*“, „($a+b>c$)($a+c>b$)($b+c>a$) = 1“, „$s = (a+b+c) / 2$“, „$F = [s (s - a)(s - b)(s - c)]^{0,5}$“ **bilden die** „*Fläche eines Dreiecks*”.

2) Generalisierende Abstraktion:
   „*Quadrat, Rechteck, Dreieck*“ **sind** „*geometrische Objekte*“.
   Oder, als formal „*berechnete*“, generalisierte Abstraktion aus Abbildung 5:
   (((„*GenRechteck*“ **oder** „*GenDreieck*“) **sind** „*GenNichtQuadrat*“ **oder** „*GenQuadratClass*“) **sind** „*GenGeometrie*“).

Die Abbildungen 6 bis 8 zeigen die *komponierenden Abstraktionen* der Benutzer und die betroffenen Teile (Klassen) der *generalisierenden Abstraktionen*, aus denen sich die *komponierenden Abstraktionen* zusammensetzen. Die Klassen, die zur jeweiligen Komposition einer Zweckorientierten Objektsicht gehören, sind durch die im Hintergrund hinterlegten Flächen zusammengefasst und mit „***Komp....***“ benannt. Abbildung 6 zeigt die Zweckorientierte Objektsicht *Quadrat* mit Namen „***Komp****Quadrat*“, wie sie im folgenden C++ Programm verwendet wird. Die Objektsicht „***Komp****Quadrat*“ beinhaltet die generalisierten/spezialisierten Klassen „***Gen****Geometrie*“ mit den Eigenschaften a_1 „*Seite a*“ und a_4 „*Fläche F*“ und „***Gen****Quadrat*“ mit den Eigenschaften p_1 „($a > 0$) = 1“ und p_4 „$F = a * a$“.

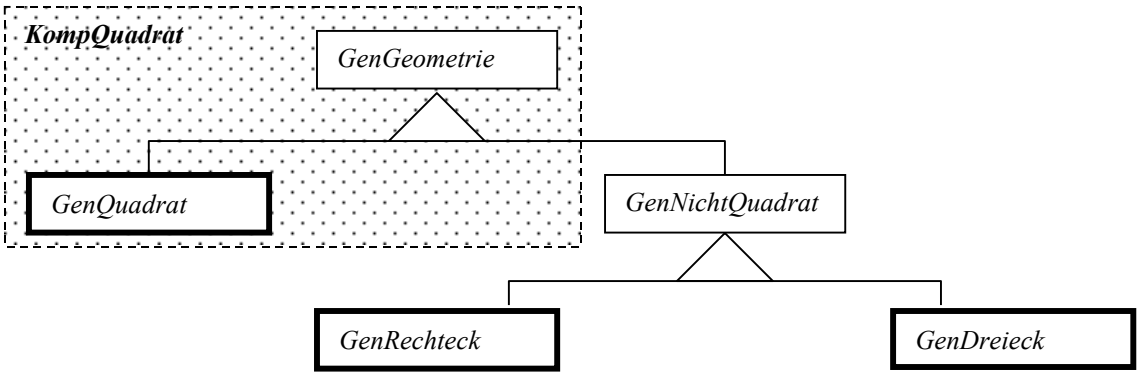


Abbildung 6 *Komponierende Abstraktion „Quadrat“* aus **Tabelle 2** , im C++ -Programm als „*KompQuadrat*“ über der *generalisierenden Abstraktion* realisiert, welche im C++ -Programm ihrerseits die generalisierten/spezialisierten Klassen „*GenGeometrie*“ und „*GenQuadrat*“ umfasst.

Abbildung 7 zeigt die Zweckorientierte Objektsicht *Rechteck* mit Namen „***Komp****Rechteck*“, wie sie im folgenden C++ Programm verwendet wird. Die Objektsicht „***Komp****Rechteck*“ beinhaltet die generalisierte Klasse „***Gen****Geometrie*“ mit den Eigenschaften a_1 „*Seite a*“ und a_4 „*Fläche F*“, die generalisierte/spezialisierte Klasse „***Gen****NichtQuadrat*“ mit der Eigenschaft a_2 „*Seite b*“ und der spezialisierten Klasse „***Gen****Rechteck*“ mit den Eigenschaften p_2 „$(a <> b)(a > 0)(b > 0) = 1$“ und p_5 „$F = a * b$“.

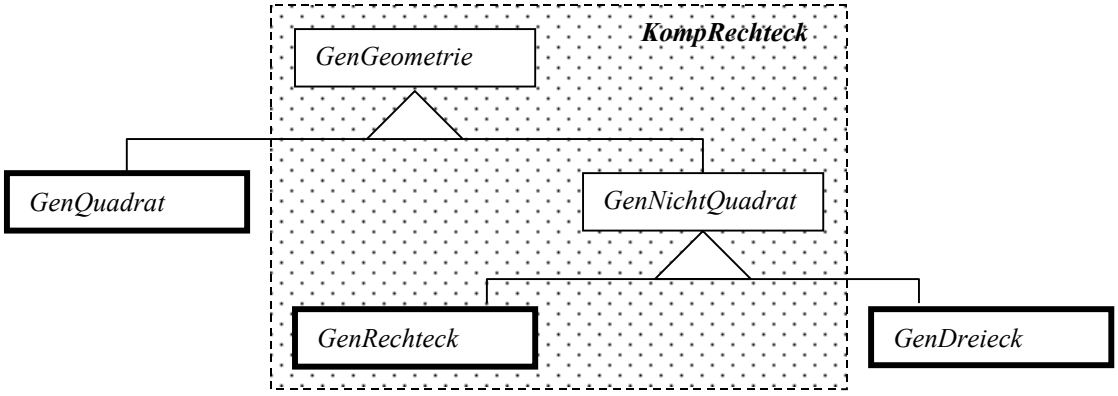


Abbildung 7: *Komponierende Abstraktion* „*Rechteck*“ aus **Tabelle 2**, im C++ - Programm als „*KompRechteck*“ über der *generalisierenden Abstraktion* realisiert, welche ihrerseits im C++ - Programm die generalisierten/spezialisierten Klassen „*GenGeometrie*“, „*GenNichtQuadrat*“ und „*GenRechteck*“ umfasst

Abbildung 8 zeigt die Zweckorientierte Objektsicht *Dreieck* mit Namen „***Komp****Dreieck*“, wie sie im folgenden C++ Programm verwendet wird. Die Objektsicht „***Komp****Dreieck*“ beinhaltet die generalisierte Klasse „***Gen****Geometrie*“ mit den Eigenschaften a_1 „*Seite a*“ und a_4 „*Fläche F*“, die generalisierte/spezialisierte Klasse „***Gen****NichtQuadrat*“ mit der Eigenschaft a_2 „(*Seite b*)“ und die spezialisierte Klasse „***Gen****Dreieck*“ mit den Eigenschaften a_3 „*Seite c*“, a_5 „*Halbe Seiten s*“, p_3 „$(a+b>c)(a+c>b)(b+c>a) = 1$“, p_6 „$s = (a+b+c) / 2$“ und p_7 „$F = [s\ (s - a)(s - b)(s - c)]^{0,5}$“.

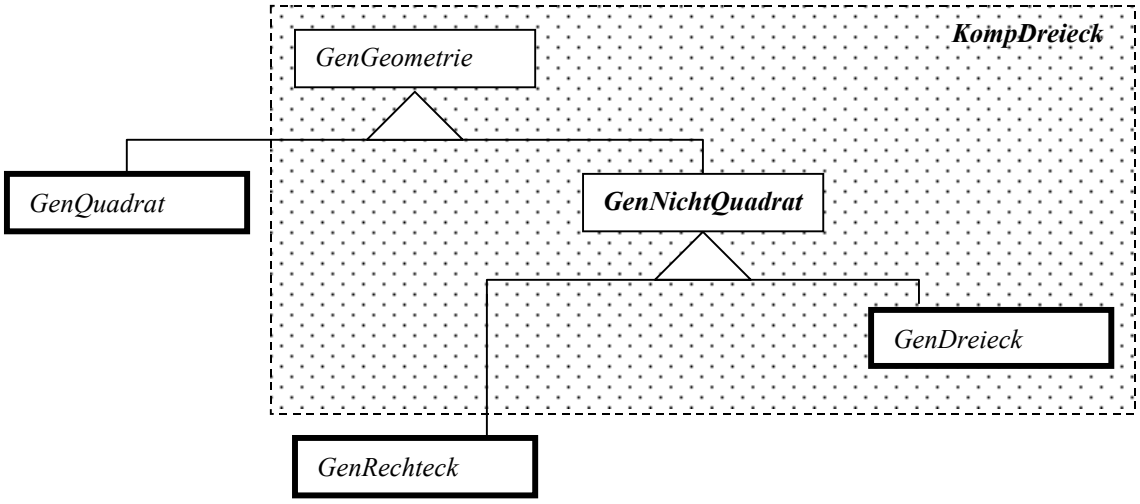


Abbildung 8 *Komponierende Abstraktion* „*Dreieck*“ aus **Tabelle 2**, im C++ - Programm als „*KompDreieck*“ über der *generalisierenden Abstraktion* realisiert, welche im C++ - Programm ihrerseits die generalisierten/spezialisierten Klassen „*GenGeometrie*“, „*GenNichtQuadrat*“ und „*GenDreieck*“ umfasst

Das nachfolgende Programm, in der Programmiersprache C++ realisiert, zeigt einmal die Anwendung des Konzeptes der *generalisierenden Abstraktion* aus der Sicht des Konstrukteurs und zum Andern die Anwendung des Konzeptes der *komponierenden Abstraktion* aus der Sicht des Nutzers bzw. des Anwendungsprogrammierers unter Benutzung von linearen Schnittstellen, abgeleitet aus den Elementen der jeweiligen Zweckorientierten Objektsichten. Die Konstruktion der Vererbungsstrukturen bleibt in dieser Vorgehensweise damit nicht mehr wenigen, „begnadeten“ Konstrukteuren vorbehalten, sondern wird durch den Vorgang der Anwendung des Distributivgesetzes der Booleschen Algebra formal bewerkstelligt. Die Vererbung als Ausdruck der Generalisierung / Spezialisierung ist damit eindeutig wiederholbar und für jeden Entwickler nachvollziehbar. Bei Systemerweiterungen /-veränderungen werden die Änderungen in die Tabelle der Zweckorientierten Objektsichten aufgenommen, eine neue, jetzt gültige disjunktive Normalform gebildet und neu strukturiert. Wenn die Granularität der Prozesse (Methoden) fein genug gewählt wird, lassen sich die Prozesse leicht in den Prozess der Bildung der Vererbungsstruktur einbeziehen. Die folgende Darstellung zeigt den orthogonalen Charakter der *generalisierenden* und *komponierenden Abstraktion*.

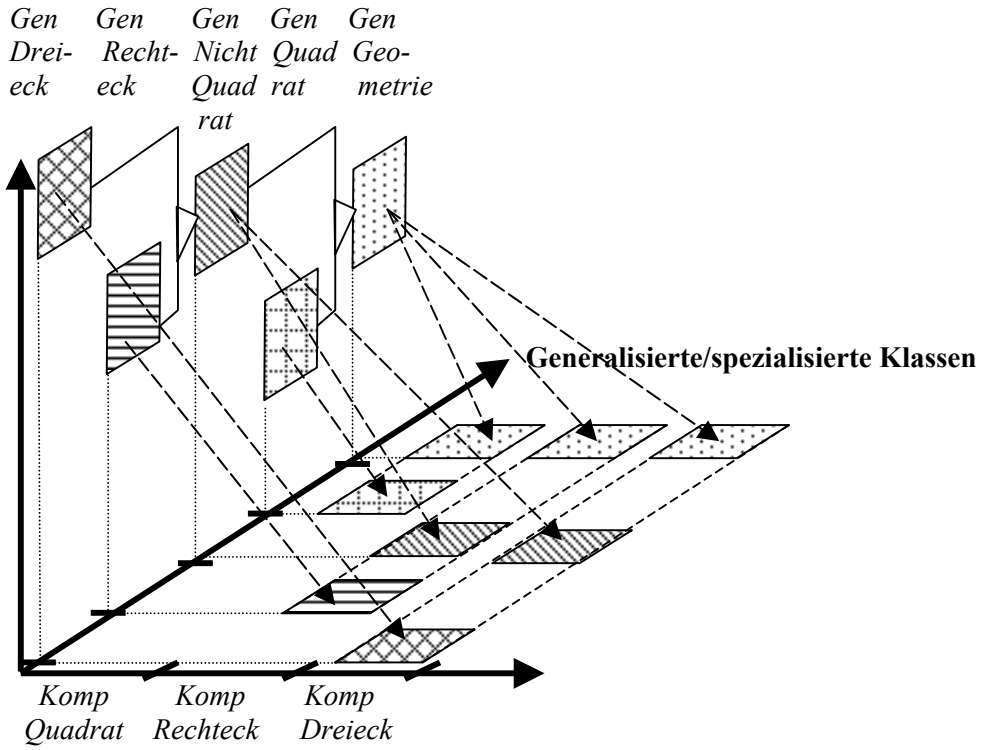


Abbildung 9: Orthogonale Darstellung der *generalisierenden* und *komponierenden Abstraktion*

Das C++ - Programm zeigt die wesentlichen Merkmale der Umsetzung dieser Vorgehensweise in einem Programm. Aus Gründen der Übersichtlichkeit und der Durchschaubarkeit wurden ausschließlich *fachliche Eigenschaften* in die *generalisierenden* / *komponierenden Kompositionen* in Tabelle 2 einbezogen. Komponenten der *Programmansteuerung* / der *Programmsteuerung* (void main( ) – folgende) wurden *nicht* in die Betrachtungen einbezogen, um die Überschaubarkeit zu gewährleisten, sondern „*herkömmlich*“ dem Programm hinzugefügt. Es wäre jedoch ohne weiteres möglich, in einer Wurzelklasse (oder auch in einer zweiten Wurzelklasse) programmsteuernde Eigenschaften zentral zu generalisieren, und auch über eigene Zweckorientierte Sichten (eigene Phänomene) alle programmsteuerungsrelevanten Eigenschaften in das Gesamtsystem nach gleichen Kriterien einzubringen, so dass auch die programmsteuernden Elemente in die Vorgehensweise voll und ausnahmslos integriert sind. Ebenso können Ein-/Ausgabeoperationen in den Phänomenbereich und damit in die Zweckorientierten Objektsichten mit einbezogen werden. Diese Möglichkeit der Berücksichtigung von „*Steuerungssichten*“ ist in Tabelle 3 angedeutet:

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen) *Quadrat* S_1	*Rechteck* S_2	*Dreieck* S_3	*Steuerung* S_4	Anzahl
	1	2	3	4	5	6	7
1	*Seite a*	a_1	+	+	+		3
2	*Seite b*	a_2		+	+		2
...	...	...	...	...	...	...	...
m	void main( ) [*Funktionen n-mal aufrufen];* *Wiederhole bis Ende-Signal:* *Wenn Eingabe „Q“:* *Quadrat bearbeiten;* *Wenn Eingabe „R“:* *Rechteck bearbeiten;* *Wenn Eingabe „D“:* *Dreieck bearbeiten;* *Wiederhole - Ende;*		+ [1,*n*] [1,*n*] [1,*n*]	+ [1,*n*]	+ [1,*n*]	+ [1,1] [1,1] [1,1]	4

Tabelle 3 Zweckorientierte Objektsicht mit Integration des Phänomens „*Steuerung*“

Die Zweckorientierte Sicht „*Steuerung*“ (Spalte 6 in Tabelle 3) ist aus Gründen der Übersichtlichkeit im folgenden Programmcode nicht enthalten. Der Programmcode zeigt in C++ - Notation die Umsetzung des Fallbeispiels, entwickelt aus Tabelle 2 und der aus dieser Tabelle abgeleiteten Abbildungen und Gleichungen:

```
#include <iostream.h>                          // Standardbibliothek zu C++
#include <math.h>
// Globale Variable definieren
static int   EingW;

class GenNichtQuadrat;
class GenGeometrie;
class GenQuadrat;
class GenRechteck;
class GenDreieck;

// Komposition Quadrat definieren
class KompQuadrat   /* Zweckorientierte Objektsicht (Komposition) */
{ public:    void zeigen (GenQuadrat a); };

// Komposition Rechteck definieren
class KompRechteck  /* Zweckorientierte Objektsicht (Komposition) */
{ public:    void zeigen (GenRechteck a); };

// Komposition Dreieck definieren
class KompDreieck   /* Zweckorientierte Objektsicht (Komposition) */
{ public:    void zeigen (GenDreieck a); };

// Generalisierungen / Spezialisierungen
/* :=Generalisierung      Wurzel                                    */
class GenGeometrie
{
protected:   void   Seite_a_belegen ();
                    double Flaeche;                                /* a4 */
                    float  Seite_a;                                /* a1 */
};

/* Spezialisierung    : Generalisierung (Einfachvererbung)  */
class GenNichtQuadrat : protected GenGeometrie
{
protected:   void   Seite_b_belegen ();
```

```
                    float  Seite_b;                                  /* a2 */
};

/* Spezialisierung : Generalisierung (Einfachvererbung)    */
class GenDreieck   : protected GenNichtQuadrat
{
public:             void   D_belegen ();
protected:   void   Seite_c_belegen_1 ();
                    void   Seite_c_belegen_2 ();
                    void   Dreieck_erzeugen ();
                    float  Halbe_Seiten;                            /* a5 */
                    float  Seite_c;                                 /* a3 */
                    friend KompDreieck;
};

/* Spezialisierung : Generalisierung           */
class GenQuadrat   : protected GenGeometrie
{
public:             void   Q_belegen ();
protected:   void   Quadrat_belegen ();
             void   Quadrat_erzeugen ();
                    friend KompQuadrat;
};

/* Spezialisierung : Generalisierung           */
class GenRechteck  : protected GenNichtQuadrat
{
public:             void   R_belegen ();
protected:   void   Rechteck_belegen ();
                    void   Rechteck_erzeugen ();
                    friend KompRechteck;
};

// Definitionen zu den Klassen der linearen Schnittstellen
// (Kompositionen der Zweckorientierten Objektsichten)
void KompQuadrat :: zeigen (GenQuadrat a)
{
    cout << "----- Schnittstelle Quadrat ---------" <<                "\n";
    cout << "Flaeche                              " << a.Flaeche <<    "\n";
    cout << "Seite a                              " << a.Seite_a <<    "\n";
    cout << "----- Ende Schnittstelle Quadrat ----" <<                "\n";
}

void KompRechteck :: zeigen (GenRechteck a)
{
    cout << "----- Schnittstelle Rechteck --------" <<         "\n";
    cout << "Flaeche                              " << a.Flaeche <<    "\n";
    cout << "Seite a                              " << a.Seite_a <<    "\n";
    cout << "Seite b                              " << a.Seite_b <<    "\n";
    cout << "----- Ende Schnittstelle Rechteck ---" <<         "\n";
}

void KompDreieck :: zeigen (GenDreieck a)
{
    cout << "----- Schnittstelle Dreieck_autonom ----------" <<        "\n";
    cout << "Flaeche                                  " << a.Flaeche <<    "\n";
    cout << "Seite a                                  " << a.Seite_a <<    "\n";
    cout << "Seite b                                  " << a.Seite_b <<    "\n";
    cout << "Seite c                                  " << a.Seite_c <<    "\n";
    cout << "1/2(a+b+c) (Halbe Seiten nach Heron) " << a.Halbe_Seiten <<    "\n";
    cout << "----- Ende Schnittstelle Dreieck_autonom -----" <<       "\n";
}

// Definitionen zu Klasse GenDreieck
void GenDreieck :: Seite_c_belegen_1 ()
{
      cout << "Seite c eingeben:"<<"\n";
      cin >> Seite_c;
}

void GenDreieck :: Seite_c_belegen_2 ()
{
      if ((Seite_a + Seite_b > Seite_c)*
```

```
              (Seite_b + Seite_c > Seite_a)*

              (Seite_a + Seite_c > Seite_b))                        /* p3  */
       {      Halbe_Seiten = (Seite_a + Seite_b + Seite_c) / 2;    /* p6  */
              Flaeche      = sqrt (Halbe_Seiten *
                                   (Halbe_Seiten - Seite_a)*
                                   (Halbe_Seiten - Seite_b)*
                                   (Halbe_Seiten - Seite_c));      /* p7  */
       }
}

void GenDreieck :: Dreieck_erzeugen ()
{
       cout << "Auspraegungen des Dreiecks eingeben. " << "\n";
       Seite_a_belegen ();
       Seite_b_belegen ();
       Seite_c_belegen_1 ();
       Seite_c_belegen_2 ();
}

void GenDreieck :: D_belegen ()
{    Dreieck_erzeugen (); }

// Definitionen zu Klasse GenRechteck
void GenRechteck :: R_belegen()
{    Rechteck_erzeugen ();}

void GenRechteck :: Rechteck_belegen ()
{
       if ((Seite_a > 0) * (Seite_b > 0) * (Seite_a != Seite_b))   /* p2  */
       {
              Flaeche = (Seite_a * Seite_b);                       /* p5  */
       }
}

void GenRechteck :: Rechteck_erzeugen ()
{
       cout << "Auspraegungen des Rechtsecks eingeben. " << "\n";
       Seite_a_belegen ();
       Seite_b_belegen ();
       Rechteck_belegen ();
}

// Definitionen zu Klasse GenNichtQuadrat
void GenNichtQuadrat :: Seite_b_belegen ()
{
       cout << "Seite b eingeben:"<<"\n";
       cin >> Seite_b;
}

// Definitionen zu Klasse GenQuadrat
void GenQuadrat :: Q_belegen()
{    Quadrat_erzeugen (); }

void GenQuadrat :: Quadrat_belegen ()
{
       if (Seite_a > 0)                    /* p1  */
       {
              Flaeche = (Seite_a * Seite_a);  /* p4  */
       }
}

void GenQuadrat :: Quadrat_erzeugen ()
{
       cout << "Auspraegungen des Quadrats eingeben. " << "\n";
       Seite_a_belegen ();
       Quadrat_belegen ();
}

// Definitionen zu Klasse GenGeometrie
void GenGeometrie :: Seite_a_belegen ()
{
       cout << "Seite a eingeben:"<<"\n";
```

```
      cin >> Seite_a;
}

// Ansteuerung der Klassen
void main()
{
      cout << "\n" << "Flaeche berechnen: 1 Quadrat, 2 Rechteck, 3 Dreieck" << "\n"
      << "\n";
      cout << "Geben Sie bitte die Zahl der gewuenschten Aktion ein"<< "\n";
      cin>> EingW;
      if (EingW == 1)
      {
            GenQuadrat UFQO;
            UFQO.Q_belegen ();
            KompQuadrat Q_Komposition;
            Q_Komposition.zeigen(UFQO);
      }
      if (EingW == 2)
      {
            GenRechteck UFRO;
            UFRO.R_belegen();
            KompRechteck R_Komposition;
            R_Komposition.zeigen(UFRO);
      }
      if (EingW == 3)
      {
            GenDreieck UFDO;
            UFDO.D_belegen();
            KompDreieck D_Komposition;
            D_Komposition.zeigen(UFDO);
      }
      cout << "Ende" << "\n";    /* ~ */
}
```

Die *komponierende Abstraktion* ist realisiert durch die Abbildung jeder Spalte der Zweckorientierten Objektsichten *Quadrat*, *Rechteck* und *Dreieck* aus Tabelle 2 als jeweils eine eigene Klasse mit dem Präfix „**Komp**...“. Damit diese Klassen auf die originären Eigenschaften der Klassen aus der *generalisierenden Abstraktion* mit dem Präfix „**Gen**...“ und deren Inhalte zugreifen können, wurde bei jeder dieser Klassen der *generalisierenden Abstraktion* vom Typ „**Gen**...“ eine sogenannte „*friend* “-Eintragung vorgenommen.

```
// Komposition Quadrat definieren
class KompQuadrat   /* Zweckorientierte Objektsicht (Komposition) */
{ public:    void zeigen (GenQuadrat a); };

// Komposition Rechteck definieren
class KompRechteck  /* Zweckorientierte Objektsicht (Komposition) */
{ public:    void zeigen (GenRechteck a); };

// Komposition Dreieck definieren
class KompDreieck   /* Zweckorientierte Objektsicht (Komposition) */
{ public:    void zeigen (GenDreieck a); };
```

Die einzelnen Eigenschaften „`Flaeche`“, „`Seite_a`“, „`Seite_b`“, „`Seite_c`“ und „`Halbe_Seiten`“ der *komponierenden Abstraktion* des Typs „**Komp**...“ der Zweckorientierten Objektsichten *Quadrat*, *Rechteck* und *Dreieck* aus Tabelle 2, stellen *originär* identisch die Inhalte der einmalig definierten Klassen „**Gen**...“ aus der *generalisierenden Abstraktion* dar. Diese stehen somit als lineare Schnittstellendaten für Anwendungsprogrammierer/Nutzer zur Verfügung, die im vorliegenden Beispielprogramm zu „**cout**“ – Ausgabezwecken genutzt werden:

```
// Definitionen zu den Klassen der linearen Schnittstellen
// (Kompositionen der Zweckorientierten Objektsichten)
void KompQuadrat :: zeigen (GenQuadrat a)
{
    cout << "----- Schnittstelle Quadrat ---------" <<                "\n";
    cout << "Flaeche                              " << a.Flaeche <<   "\n";
    cout << "Seite a                              " << a.Seite_a <<   "\n";
    cout << "----- Ende Schnittstelle Quadrat ----" <<                "\n";
```

```
}
void KompRechteck :: zeigen (GenRechteck a)
{

    cout << "----- Schnittstelle Rechteck --------" <<         "\n";
    cout << "Flaeche                              " << a.Flaeche <<    "\n";
    cout << "Seite a                              " << a.Seite_a <<    "\n";
    cout << "Seite b                              " << a.Seite_b <<    "\n";
    cout << "----- Ende Schnittstelle Rechteck ---" <<         "\n";
}
void KompDreieck :: zeigen (GenDreieck a)
{
    cout << "----- Schnittstelle Dreieck_autonom ----------" <<        "\n";
    cout << "Flaeche                              " << a.Flaeche <<    "\n";
    cout << "Seite a                              " << a.Seite_a <<    "\n";
    cout << "Seite b                              " << a.Seite_b <<    "\n";
    cout << "Seite c                              " << a.Seite_c <<    "\n";
    cout << "1/2(a+b+c) (Halbe Seiten nach Heron) " << a.Halbe_Seiten <<    "\n";
    cout << "----- Ende Schnittstelle Dreieck_autonom -----" <<       "\n";
}
```

Tabelle 4 zeigt die Zuordnung der Eigenschaften der *komponierenden Abstraktion* zu den Zweckorientierten Objektsichten und der im obigen Programm verwendeten Namen in den Kompositionen „***KompQuadrat***“, „***KompRechteck***“ und „***KompDreieck***“:

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen)		
			KompQuadrat S_1	*KompRechteck* S_2	*KompDreieck* S_3
	1	2	3	4	5
1	*Seite a*	a_1	+ **Seite_a**	+ **Seite_a**	+ **Seite_a**
2	*Seite b*	a_2		+ **Seite_b**	+ **Seite_b**
3	*Seite c*	a_3			+ **Seite_c**
4	*Fläche F*	a_4	+ **Flaeche**	+ **Flaeche**	+ **Flaeche**
5	*Halbe Seiten s*	a_5			+ **Halbe_Seiten**

Tabelle 4 Eigenschaftenschnittstellendaten der *komponierenden Abstraktion*

Exklusiv Oder Ausdrücke. Die soeben dargestellte Vorgehensweise reicht für die Ableitung der Struktur des oben gezeigten Programms aus. Sie lässt sich jedoch auch präziser mit Hilfe von negierten Booleschen – Variablen als „*exklusiv Oder*“ Verknüpfung der Zweckorientierten Objekt-sichten darstellen. Tabelle 5 , die originär aus Tabelle 2 abgeleitet wurde, zeigt diese Möglichkeit:

Lfd. Nr.	Phänomenbereich Merkmale (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen)			Anzahl
			Quadrat S_1	*Rechteck* S_2	*Dreieck* S_3	
	1	2	3	4	5	6
1	*Seite a*	a_1	+	+	+	3
2	*Seite b*	a_2	-	+	+	2
3	*Seite c*	a_3	-	-	+	1
4	*Fläche F*	a_4	+	+	+	3
5	*Halbe Seiten s*	a_5	-	-	+	1
6	$(a > 0) = 1$	p_1	+	-	-	1
7	$(a <> b)(a > 0)(b > 0) = 1$	p_2	-	+	-	1
8	$(a+b>c)(a+c>b)(b+c>a) = 1$	p_3	-	-	+	1
9	$F = a * a$	p_4	+	-	-	1
10	$F = a * b$	p_5	-	+	-	1
11	$s = (a + b + c) / 2$	p_6	-	-	+	1
12	$F = [s\ (s - a)(s - b)(s - c)]^{0,5}$	p_7	-	-	+	1

Tabelle 5: Tabelle der Zweckorientierten Objektsichten mit expliziten negierten Booleschen – Variablen

Die mit „-“ markierten Tabellenelemente können mit negierten Booleschen – Variablen belegt werden. Damit lassen sich „*exklusiv Oder – Beziehungen*“ explizit zwischen Zweckorientierten Objektsichten ausdrücken. Die Werteausprägung für negierte Boolesche – Variable ist: Wenn $a_i, p_i = 1$, dann ist $\underline{a_i}, \underline{p_i} = \neg a_i, \neg p_i = 0$ und, wenn $a_i, p_i = 0$, dann ist $\underline{a_i}, \underline{p_i} = \neg a_i, \neg p_i = 1$. Nur wenn verschiedene Phänomene gleichzeitig ausgeprägt werden sollen, kann hier auf die sich gegenseitig ausschließenden Negationen verzichtet werden, wie z. B. bei einem für ein Unternehmen gleichzeitig als „Kunde“ und als „Lieferant“ auftretenden externen Partner. Zuviel gesetzte Negationen neutralisieren sich durch die Anwendung des Distributivgesetzes der Booleschen Algebra. Die Hauptspalten drei, vier und fünf in Tabelle 5 in disjunktiver Normalform interpretiert, ergeben mit den explizit negierten Booleschen – Variablen als Disjunktion in Gleichung 5:

Gleichung 5

$$S_1 \vee S_2 \vee S_3 = \boldsymbol{a_1}\ \underline{a_2}\ \underline{a_3}\ \boldsymbol{a_4}\ \underline{a_5}\ p_1\ \underline{p_2}\ \underline{p_3}\ p_4\ \underline{p_5}\ \underline{p_6}\ \underline{p_7} \vee \boldsymbol{a_1}\ a_2\ \underline{a_3}\ \boldsymbol{a_4}\ \underline{a_5}\ \underline{p_1}\ p_2\ \underline{p_3}\ \underline{p_4}\ p_5\ \underline{p_6}\ \underline{p_7} \vee \boldsymbol{a_1}\ a_2\ a_3\ \boldsymbol{a_4}\ a_5\ \underline{p_1}\ \underline{p_2}\ p_3\ \underline{p_4}\ \underline{p_5}\ p_6\ p_7 = 1.$$

Klammert man diesen Ausdruck den Term $\boldsymbol{a_1}\ \boldsymbol{a_4}$ aus, erhält man:

Gleichung 6

$$a_1\ a_4\ (\underline{a_2}\ \underline{a_3}\ \underline{a_5}\ p_1\ \underline{p_2}\ \underline{p_3}\ p_4\ \underline{p_5}\ \underline{p_6}\ \underline{p_7} \vee \boldsymbol{a_2}\ \underline{a_3}\ \underline{a_5}\ \boldsymbol{\underline{p_1}}\ p_2\ \underline{p_3}\ \boldsymbol{\underline{p_4}}\ p_5\ \underline{p_6}\ \underline{p_7} \vee \boldsymbol{a_2}\ a_3\ a_5\ \boldsymbol{\underline{p_1}}\ \underline{p_2}\ p_3\ \boldsymbol{\underline{p_4}}\ \underline{p_5}\ p_6\ p_7) = 1.$$

Klammert man innerhalb der Klammer den Term $\boldsymbol{a_2}\ \boldsymbol{\underline{p_1}}\ \boldsymbol{\underline{p_4}}$ weiter aus, erhält man:

Gleichung 7

$$a_1\ a_4\ (\underline{a_2}\ \underline{a_3}\ \underline{a_5}\ p_1\ \underline{p_2}\ \underline{p_3}\ p_4\ \underline{p_5}\ \underline{p_6}\ \underline{p_7} \vee a_2\ p_1\ p_4\ (\underline{a_3}\ \underline{a_5}\ p_2\ \underline{p_3}\ p_5\ \underline{p_6}\ \underline{p_7} \vee a_3\ a_5\ \underline{p_2}\ p_3\ \underline{p_5}\ p_6\ p_7)) = 1$$

In dieser Form werden explizit formal die Klassen

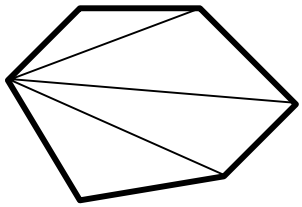
„**für alle**“ „*GenGeometrie*“ := ***a*₁ *a*₄ (** „**entweder**“ „*GenQuadrat*“ := „***a*₂** *a*₃ *a*₅ ***p*₁** *p*₂ *p*₃ ***p*₄** *p*₅ *p*₆ *p*₇“ „**oder**“ „*GenNichtQuadrat*“ := „***a*₂ *p*₁ *p*₄ (** „**entweder**“ „*GenRechteck*“ := „*a*₃ *a*₅ *p*₂ *p*₃ *p*₅ *p*₆ *p*₇“ „**oder**“ „*GenDreieck*“ := „*a*₃ *a*₅ *p*₂ *p*₃ *p*₅ *p*₆ *p*₇“ **))**

als sich gegenseitig ausschließend notiert. Dies ist die korrekte, sich gegenseitig ausschließende Ausdrucksform, die letztlich im Programm prozessual realisiert ist. Es kann sinnvoll sein, neben der in Tabelle 2 gezeigten „**inklusive Oder**“ und der in Tabelle 5 gezeigten „**exklusiv Oder**“ – Darstellung beide Darstellungsarten in einer Tabelle zu benutzen. Näheres zu diesen Möglichkeiten der negierten Booleschen – Variablen – Benutzung ist in [EnG] beschrieben. *Alle* Merkmale im Phänomenbereich sind grundsätzlich nach *beiden* Abstraktionskonzepten organisiert: Nach der *komponierenden Abstraktion* und nach der *generalisierenden Abstraktion*, denn beide Bereiche sind immer evident.

Systemwartung und Systemerweiterungen. Eine Erweiterung des soeben in Tabelle 2 beschriebenen Systems soll die Flexibilität dieser Vorgehensweise aufzeigen. Das Gesamtsystem soll um eine Zweckorientierte Sicht (Phänomen) „*Vieleck unregelmäßig*“ erweitert werden. Dazu ist die obige Tabelle 2 um diese Sicht zu erweitern. Die Fläche eines unregelmäßigen Vielecks kann auf verschiedene Arten ermittelt werden. Hier sei die Ermittlung der Fläche eines *unregelmäßigen* Vielecks mit Hilfe von Dreiecken, die einen Eckpunkt eines Vielecks gemeinsam haben, demonstriert. Um die Fläche für ein Exemplar „*Vieleck unregelmäßig*“ zu ermitteln, benötigt man *n* Dreiecke. Das folgende Beispiel zeigt ein unregelmäßiges, konvexes *m*-Eck am Beispiel eines 6-Ecks, darstellbar mit vier Dreiecken.

Anzahl Ecken:	m; $(m > 3)$;
Anzahl Dreiecke im unregelmäßigen Vieleck	n: $n = m - 2$
Fläche eines unregelmäßigen Vielecks:	$F = A_1 + A_2 + A_3 + ... + A_i + ... + A_n$
Halbe Seiten eines Dreiecks *i* nach Heron:	$s_i = 0{,}5 * (a + b + c)_i$
Fläche eines Dreiecks *i* nach Heron:	$A_i = [(s(s - a)(s - b)(s - c))^{0,5}]_i$.

Beispiel eines unregelmäßigen 6-eckigen Vielecks:



Anzahl *m* Ecken im unregelmäßigen Sechseck:	6
Anzahl *n* Dreiecke im unregelmäßigen Sechseck:	4 = 6 - 2
Fläche eines unregelmäßigen Sechsecks:	$F = A_1 + A_2 + A_3 + A_4$.

In Tabelle 2 wurde in Spalte fünf bereits eine Zweckorientierte Sicht, eine *komponierende Abstraktion* „*Dreieck*“ beschrieben. Um möglichst wenig zusätzlich Aufwand zu haben und um gleichzeitig eine Lösung zu haben, in der die Anzahl der Ecken des *unregelmäßigen* Vielecks nicht im Programm bereits festgelegt ist, sind für eine Zweckorientierte Objektsicht „*Vieleck unregelmäßig*“ bis zu *n* Zweckorientierte Objektsichten „*Dreieck*“ auszuprägen. Folglich stehen die beiden Zweckorientierten Objektsichten „*Vieleck unregelmäßig*“ und „*Dreieck*“ in einer 1 : *n* Beziehung. Dies bedeutet, jedes auszuprägende „*Vieleck unregelmäßig*“ hat bis zu *n* individuell ausgeprägte „*Dreiecke*“. Bei der Analyse eines „*Dreiecks*“, das Teil eines *unregelmäßigen* Vielecks ist, und einem „*Dreieck*“, das unabhängig, eigenständig zu berechnen ist, ergeben sich bei einigen, nicht allen, Eigenschaften existenzielle Unterschiede. Dem wird insofern Rechnung getragen, als das seither in Spalte fünf in Tabelle 2 in der *komponierenden Abstraktion* beschriebene „*Dreieck*“ als „*Dreieck autonom*“ unverändert in das neue Modell eingeht, wie in Tabelle 6, Spalte fünf, gezeigt.

In einer weiteren *komponierenden Abstraktion* wird nochmals ein „*Dreieck abhängig*“ in Tabelle 6, Spalte sechs, beschrieben, das in einer *n* : 1 Beziehung zur ebenfalls zu beschreibenden *komponierenden Abstraktion* „*Vieleck unregelmäßig*“ in Tabelle 6, Spalte sieben, abgebildet ist.
Eigenschaften der Spalte eins, die in beiden Dreiecken gleich sind, wie z. B. die „*Seite c*“, werden mit (+) einfach von der Spalte „*Dreieck autonom*“ in die Spalte „*Dreieck abhängig*“ übernommen. Hinzu kommen Eigenschaften, die das „*Dreieck abhängig*“ nur alleine besitzt, wie z. B. die laufende Nummer *i* innerhalb eines *unregelmäßigen* Vielecks. Neben dieser Erweiterung des Beispiels aus Tabelle 2 kommen noch weitere Eigenschaften des Gesamtsystems hinzu. Insgesamt sollen in die Modellerweiterung der Tabelle 2 folgende Eigenschaften in Tabelle 6 zusätzlich aufgenommen werden:

1. Objekt - Identifikation als Primary-Key-Teil l (in allen Zweckorientierten Objektsichten);
2. Laufende Nr. eines abhängigen Dreiecks *i* als Primary-Key-Teil 2 (nur „*Vieleck unregelmäßig*“);
3. *X*-Koordinate (Position zur Anzeige aller Sichten außer „*Dreieck abhängig*“);
4. *Y*-Koordinate (Position zur Anzeige aller Sichten außer „*Dreieck abhängig*“);
5. Summe Fläche „*Vieleck unregelmäßig*“ aus *n* „*Dreiecken abhängig*“;
6. Anzahl Ecken „*Vieleck unregelmäßig*“;
7. Summe „*Vieleck unregelmäßig*“ auf 0 setzen;
8. Fläche „*Dreieck abhängig*“ nach Summe Fläche „*Vieleck unregelmäßig*“ addieren;
9. Steuerung der 1 : *n* - Beziehung: Ein „*Vieleck unregelmäßig*“ hat *n* „*Dreiecke abhängig*“;
10. Anzahl „*Dreiecke abhängig*“ aus Anzahl *m* der Ecken des „*Vielecks unregelmäßig*“ ermitteln.

Alle andern Eigenschaften des neuen Systems, wie in Tabelle 6 beschrieben, können aus Tabelle 2 übernommen werden (Zeilen eins bis zwölf der Tabelle 2 nach Zeilen eins bis sechs bzw. zwölf bis achtzehn der Tabelle 6). In die neue Organisationsform sollen bestimmte Erweiterungen mit aufgenommen werden. Beispielsweise sollen die einzelnen Ausprägungen (alle Instanzen) aller Zweckorientierten Objektsichten eine Identifizierungsnummer „*Objekt – Identifikation*“ erhalten. Ebenso soll ein *X*, *Y* – Wert zur Positionierung der Figur (Ausprägung) auf dem Bildschirm aufgenommen werden.
Die Aufnahme einer Zweckorientierten Objektsicht „*Vieleck unregelmäßig*“ erfordert zur Flächenberechnung nach der hier gewählten Berechnungsart die Aufteilung eines „*Vielecks unregelmäßig*“ in *n* „*Dreiecke abhängig*“. Die Summe der *n* „*Dreiecke abhängig*“ ergibt die Gesamtfläche des „*Vielecks unregelmäßig*“. Da jedes „*Dreieck abhängig*“ in seinen Abmessungen anders sein kann, sollen die Abmessungen jedes einzelnen der *n* „*Dreiecke abhängig*“ vorgehalten werden. Deshalb ist für jedes „*Vieleck unregelmäßig*“ eine eigene Relation zu *n* „*Dreiecken abhängig*“ auszuprägen: 1 „*Vieleck unregelmäßig*“ : *n* „*Dreiecke abhängig*“. Für jedes existierende „*Vieleck unregelmäßig*“ sind die „*Vieleck unregelmäßig*“ - spezifischen Eigenschaften *einmal* vorzuhalten. Da sich alle „*Dreiecke abhängig*“, die notwendig sind, um ein unregelmäßiges „*Vieleck unregelmäßig*“ abzubilden, voneinander (bis auf zwei paarweise benachbart liegende, gleiche Strecken) unterscheiden können, sind genau *n* „*Dreiecke abhängig*“ notwendig, um diese unterschiedlichen Eigenschaftsinhalte jedes einzelnen „*Dreiecks abhängig*“ auch ausprägen zu können. Gleichwohl ist jedes „*Dreieck abhängig*“ nur *einmal* in einer Zweckorientierten Objektsicht zu beschreiben. Es muss nur prozessual gewährleistet werden, dass je auszuprägendem „*Vieleck unregelmäßig*“ mindestens zwei, maximal *n* „*Dreiecke abhängig*“ mit ihren Eigenschaften ausgeprägt (instanziiert) werden können. Bei einem „*Vieleck unregelmäßig*“, das sich mit weniger als zwei Dreiecken darstellen lässt, liegt kein „*Vieleck*“ vor. In diesem Fall ist ein normales „*Dreieck autonom*“ auszuprägen. In der Eigenschafts- (Vorspalte) – Zweckorientierte Objektsicht (Hauptspalten) - Beziehung sind zwei verschiedene Zweckorientierte Objektsichten für „*Dreiecke*“ auszubilden, denn die Eigenschaften eines Typs „*Dreieck autonom*“ unterscheiden sich von einem Typ „*Dreieck abhängig*“, das zur Darstellung eines „*Vielecks unregelmäßig*“ dient, wie in Tabelle 6 im Vergleich zu Tabelle 2 gezeigt wird.

Ein Vergleich der beiden Hauptspalten fünf („*Dreieck autonom*“) und sechs („*Dreieck abhängig*“) in Tabelle 6 zeigt in den Eigenschaftsausprägungen die Unterschiede in Form der gesetzten „+“ - Zeichen. Die Hauptspalten drei („*Quadrat*“), vier („*Rechteck*“) und fünf („*Dreieck*“ umbenannt in „*Dreieck autonom*“) wurden mit ihren Eigenschaften originär aus Tabelle 2 in Tabelle 6 übernommen.

Komponierende Abstraktion als Aggregation. Die beiden Zweckorientierten Objektsichten „*Vieleck unregelmäßig*“ und „*Dreieck abhängig*“ sind bezüglich ihrer Ausprägungen zueinander als 1:*n* - Aggregation auszubilden: Jedes „*Vieleck unregelmäßig*“ ist aus mindestens zwei „*Dreiecken abhängig*“ zusammengesetzt, aggregiert. Einige der mit „+“ gekennzeichneten Eigenschaften werden ausschließlich in der Zweckorientierten Objektsicht „*Dreieck abhängig*“ im Zusammenhang mit der Zweckorientierten Objektsicht „*Vieleck, unregelmäßig*“ benötigt, da sie dem „*Vieleck unregelmäßig*“ zugeordnet sind. Deshalb empfiehlt sich eine eigene Zweckorientierte Objektsicht auf ein vom „*Vieleck unregelmäßig*“ abhängiges „*Dreieck abhängig*“ mit den vollständigen Eigenschaften eines „*Dreiecks abhängig*“ zu definieren, wie in der Tabelle 6, Spalte sieben, dargestellt. Eigenschaften, die sowohl in der Zweckorientierten Objektsicht „*Dreieck autonom*“ als auch in der Zweckorientierten Objektsicht „*Dreieck abhängig*“ gemeinsam vorkommen, werden durch die *generalisierende Abstraktion* verallgemeinert, d. h. als einmalig erkannt und damit auch nur einmal im künftigen Programm realisiert. Der Grundsatz lautet, eigenständige Eigenschaften bedingen eine eigene am „*Zweck orientierte Sicht auf ein Objekt*“. Eine *Aggregation* wird auch als *mereologische Abstraktion* (Teil - Ganze - Beziehung) bezeichnet [DrH]. Im Beispiel der Tabelle 6 wird ein *n* - stelliges Prädikat als Relation von einer Zweckorientierten Objektsicht „*Vieleck unregelmäßig*“ auf eine andere Zweckorientierte Objektsicht „*Dreieck abhängig*“ abgebildet. Diese Funktionalität der *mereologischen Abstraktion* findet ihren statischen Ausdruck in den Zeilen sechs und sieben in Form der beiden Variablen „*Objekt-ID*“ und „*Lfd-Nr.*“, die in einer *n*-fach ausgeprägten Zweckorientierten Objektsicht „*Dreiecks abhängig*“ mit *n* „*Lfd-Nr.*“ und einer einfach ausgeprägten, Zweckorientierten Objektsicht „*Vieleck unregelmäßig*“ mit einer „*Objekt-ID*“ zusammen vorkommt. Ihre dynamische Realisierung findet sie in Zeile 21 in Form des Prozesses p_{10}, der im realisierten Programm die aggregative Eigenschaft erzeugt bzw. kontrolliert.

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen)					Anzahl
			Quadrat S_1	*Recht-eck* S_2	*Dreieck autonom* S_3	*Dreieck abhäng.* S_4	*Vieleck unreg.* S_5	
	1	2	3	4	5	6	7	8
1	*Seite a*	a_1	+	+	+	+		4
2	*Seite b*	a_2		+	+	+		3
3	*Seite c*	a_3			+	+		2
4	*Fläche F*	a_4	+	+	+	+		4
5	*Halbe Seiten s*	a_5			+	+		2
6	***Objekt–Id–Primary-Key*-1**	a_6	+	+	+	+	+	5
7	***Lfd. Nr. – Primary-Key*-2** (*n* mal je Objekt-ID)	a_7				+*)	+	2
8	*x-Koordinate*	a_8	+	+	+		+	4
9	*y-Koordinate*	a_9	+	+	+		+	4
10	*Summe Fläche Vieleck SFV*	a_{10}				+*)	+	2
11	*Anzahl Ecken Vieleck*	a_{11}					+	1
12	$(a > 0) = 1$;	p_1	+					1
13	$(a <> b)(a > 0)(b > 0) = 1$;	p_2		+				1
14	$(a+b>c)(a+c>b)(b+c>a)= 1$;	p_3			+	+		2
15	$F = a * a$;	p_4	+					1
16	$F = a * b$;	p_5		+				1
17	$s = (a + b + c) / 2$;	p_6			+	+		2
18	$F = [s (s - a)(s - b)(s - c)]^{0,5}$;	p_7			+	+		2
19	$SF = 0$; *Lfd. Nr.* $= 0$;	p_8				+	+	2
20	$SFV = SFV + F$;	p_9				+*)	+	2
21	*Aggregation* (*UML*) S_4, S_5 *Lfd. Nr. = Lfd. Nr.* + 1;	p_{10}				+ [2, *n*]**)	+*) [1, 1]**)	2
22	*Anzahl Dreiecke n* = $[a_{11}]$ -2;	p_{11}					+	1

Tabelle 6: Erweitertes Beispiel aus **Tabelle 2**, neu aufgenommene Eigenschaften sind in den Zeilennummern der Spalte „*Lfd. Nr.*“ fett gedruckt.

*):= Beziehung zwischen den beiden Zweckorientierten Objektsichten „*Vieleck unregelmäßig*“ (Tabelle 6, Spalte sieben) und „*Dreieck abhängig*“ (Tabelle 6, Spalte sechs) bewirkt, dass beide Objektsichten betreffende Eigenschaften zur Steuerung der 1:*n* - Aggregation sich nach der Generalisierung in einer gemeinsamen Klasse befinden werden.

):= 1:*n* – Multiplizität (Kardinalität) [OeB] der Zweckorientierten Objektsichten ist konkret eine [1, 1]:[2, *n*] – Beziehung, wobei [1, 1]:[2, *n*] in Spalte sechs bzw. Spalte sieben der **Tabelle 6 bedeutet, [minimal 1, maximal 1] Exemplar eines „*Vielecks unregelmäßig*“ (also **jedes** „*Vieleck unregelmäßig*“) ist verknüpft mit [minimal 2, maximal *n*] „*Dreiecken abhängig*“. Die Übernahme der Booleschen Variablen der *komponierenden Abstraktion* aus **Tabelle 6** ergibt folgende Konjunktionen S_1 bis S_5:

Quadrat	$= S_1$	$= a_1\, a_4\, a_6\, a_8\, a_9\, p_1\, p_4$	$= 1.$
Rechteck	$= S_2$	$= a_1\, a_2\, a_4\, a_6\, a_8\, a_9\, p_2\, p_5$	$= 1.$
Dreieck autonom	$= S_3$	$= a_1\, a_2\, a_3\, a_4\, a_5\, a_6\, a_8\, a_9\, p_3\, p_6\, p_7$	$= 1.$
Dreieck abhängig	$= S_4$	$= a_1\, a_2\, a_3\, a_4\, a_5\, a_6\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10}$	$= 1.$
Vieleck unregelmäßig	$= S_5$	$= a_6\, a_7\, a_8\, a_9\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11}$	$= 1.$

Überführt man die Konjunktionen $\boldsymbol{S_1}$ **bis** $\boldsymbol{S_5}$ der Zweckorientierten Objektsichten aus Tabelle 6 in die disjunktive Normalform, ergibt sich Gleichung 8:

Hauptgeneralisierung/Spezialisierung:

Gleichung 8

$S_1 \vee S_2 \vee S_3 \vee S_4 \vee S_5 =$
$a_1\, a_4\, \boldsymbol{a_6}\, a_8\, a_9\, p_1\, p_4 \vee a_1\, a_2\, a_4\, \boldsymbol{a_6}\, a_8\, a_9\, p_2\, p_5 \vee a_1\, a_2\, a_3\, a_4\, a_5\, \boldsymbol{a_6}\, a_8\, a_9\, p_3\, p_6\, p_7 \vee a_1\, a_2\, a_3\, a_4\, a_5\, \boldsymbol{a_6}\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10} \vee \boldsymbol{a_6}\, a_7\, a_8\, a_9\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Klammert man die Boolesche Variable $\boldsymbol{a_6}$ aus, die in jeder Konjunktion vorkommt, ergibt sich Gleichung 9:

Gleichung 9

$\boldsymbol{a_6}\,(a_1\, a_4\, \boldsymbol{a_8}\, \boldsymbol{a_9}\, p_1\, p_4 \vee a_1\, a_2\, a_4\, \boldsymbol{a_8}\, \boldsymbol{a_9}\, p_2\, p_5 \vee a_1\, a_2\, a_3\, a_4\, a_5\, \boldsymbol{a_8}\, \boldsymbol{a_9}\, p_3\, p_6\, p_7 \vee a_1\, a_2\, a_3\, a_4\, a_5\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10} \vee a_7\, \boldsymbol{a_8}\, \boldsymbol{a_9}\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11}) = 1.$

Innerhalb der Klammer ergeben sich nochmals folgende Möglichkeiten, die beiden Booleschen Variablen $\boldsymbol{a_8}\, \boldsymbol{a_9}$ auszuklammern:

Gleichung 10

$\boldsymbol{a_6}\,(\boldsymbol{a_8}\, \boldsymbol{a_9}\,(a_1\, a_4\, p_1\, p_4 \vee a_1\, a_2\, a_4\, p_2\, p_5 \vee a_1\, a_2\, a_3\, a_4\, a_5\, p_3\, p_6\, p_7 \vee a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11}) \vee a_1\, a_2\, a_3\, a_4\, a_5\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10}) = 1.$

Bei gleichen Booleschen Variablen, die in unterschiedlichen Klammernebenen vorkommen, sind sogenannte Nebengeneralisierungen-/Spezialisierzungen zu bilden (siehe auch [EnG]):

Nebengeneralisierung/-spezialisierung 1:

Gleichung 11

$a_1\, a_4\, p_1\, p_4 \vee a_1\, a_2\, a_4\, p_2\, p_5 \vee a_1\, a_2\, a_3\, a_4\, a_5\, p_3\, p_6\, p_7 \vee a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} \vee a_1\, a_2\, a_3\, a_4\, a_5\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10} = 1.$

Gleichung 12

$a_1\, a_4\, [p_1\, p_4 \vee a_2\, p_2\, p_5 \vee a_2\, a_3\, a_5\, p_3\, p_6\, p_7 \vee a_2\, a_3\, a_5\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10}] \vee a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Gleichung 13

$a_1\, a_4\, [p_1\, p_4 \vee a_2\, [p_2\, p_5 \vee a_3\, a_5\, p_3\, p_6\, p_7 \vee a_3\, a_5\, a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10}]] \vee a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Gleichung 14

$\boldsymbol{a_1\, a_4}\, [\boldsymbol{p_1\, p_4} \vee \boldsymbol{a_2}\, [\boldsymbol{p_2\, p_5} \vee \boldsymbol{a_3\, a_5\, p_3\, p_6\, p_7}\, [1 \vee a_7\, a_{10}\, p_8\, p_9 p_{10}]]] \vee a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Nebengeneralisierung/-spezialisierung 2:

Gleichung 15

$a_7\, a_{10}\, p_8\, p_9 p_{10} \vee a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Gleichung 16

$\boldsymbol{a_7\, a_{10}\, p_8\, p_9 p_{10}}\, [1 \vee \boldsymbol{a_{11}\, p_{11}}] = 1.$

Eine Zuordnung von Bezeichnern zu den generalisierten/spezialisierten Klassen, deren Repräsentanten als Boolesche Variablen in den obigen Gleichungen fett markiert wurden, kann wie folgt vorgenommen werden:

GenGeoFigur	$= a_6$	= 1 (Generalisierte Klasse *Geometrie Figur*).
GenKoordin	$= a_8\, a_9$	= 1 (Generalisierte/spezialisierte Klasse *Koordinaten*).
GenGeometrie	$= a_1\, a_4$	= 1 (Generalisierte/spezialisierte Klasse *Geometrie*).
GenQuadrat	$= p_1\, p_4$	= 1 (Spezialisierte Klasse *Quadrat*).
GenNichtQuadrat	$= a_2$	= 1 (Generalisierte/spezialisierte Klasse *NichtQuadrat*).
GenRechteck	$= p_2\, p_5$	= 1 (Spezialisierte Klasse *Rechteck*).
GenDreieck	$= a_3\, a_5\ p_3\, p_6\, p_7$	= 1 (Generalisierte/spezialisierte Klasse *Dreieck*).
GenVieleck	$= a_7\, a_{10}\, p_8\, p_9 p_{10}$	= 1 (Generalisierte/spezialisierte Klasse *Vieleck*).
GenEckenanz	$= a_{11}\, p_{11}$	= 1 (Spezialisierte Klasse *Eckenanzahl*).

Es ergibt sich eine durch den Algorithmus des Distributivgesetzes der Booleschen Algebra erzeugte und damit von diesem Algorithmus kontrollierte Mehrfachvererbung, ein Zustands- und Prozessraum, der eindeutig nachzuvollziehen ist. Die Realität ist komplex und Mehrfachvererbungen von einmalig vorkommenden Datenzustands- und Prozesselementen müssen eindeutig abbildbar sein. Wenn die Datenzustands- und Prozesslage es erfordert, ist die Mehrfachvererbung einzusetzen, damit alle Eigenschaften eindeutig bleiben. Somit kann mehrdeutige Zustands- und Prozessredundanz vermieden werden. Wenn die eben beschriebene Vorgehensweise hilft, solche Zustands- und Prozessräume redundanzfrei eindeutig zu gestalten, um die Sicherheit und Verlässlichkeit von Software Systemen zu steigern, kann dies nur der Sache dienen. Die Darstellung in Abbildung 10 zeigt ein konsolidiertes Vererbungsdiagramm, entwickelt nach den Endgleichungen der Hauptgeneralisierung/Spezialisierung Gleichung 10 und den beiden Nebenspezialisierungen Gleichung 14 und Gleichung 16 (siehe auch [EnG]).

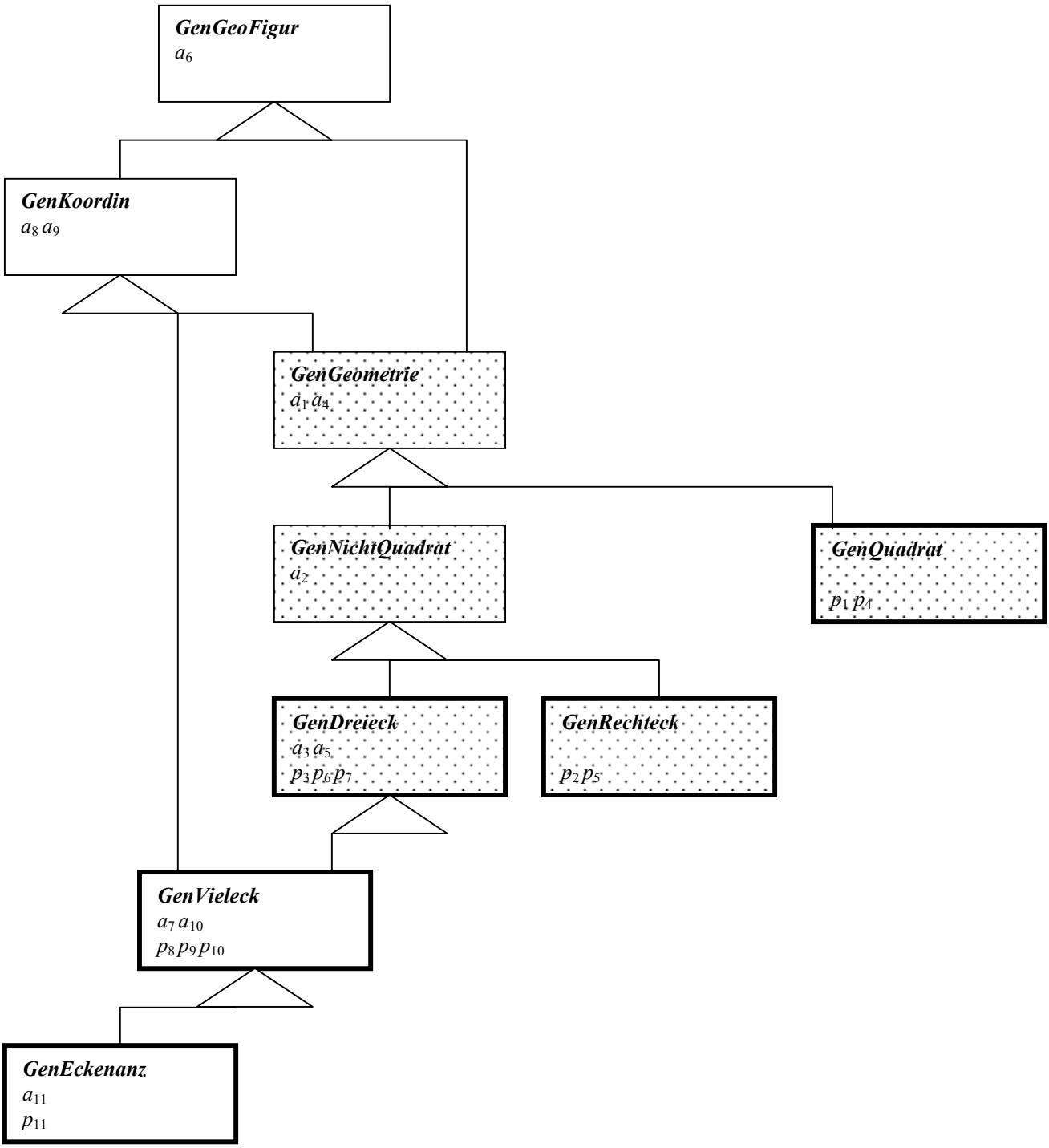


Abbildung 10 *Generalisierende Abstraktion*, Mehrfachvererbung, abgeleitet aus Tabelle 6, Klassendiagramm mit zehn Strukturverbindungen zwischen den Klassen

Vergleicht man die beiden Ergebnisse der *generalisierenden Abstraktion* des ersten und zweiten Beispiels der Abbildung 5 und Abbildung 10, kann man erkennen, dass durch die Vorgehensweise die dem ersten System innewohnende Vererbungsstruktur im zweiten System erhalten wurde. Die Grundstruktur aus dem ersten System bezüglich der Verarbeitung des Dreiecks blieb im zweiten System, was die generalisierten Klassen betrifft, erhalten.

Ebenso erhalten blieben in diesem Beispiel die Verbindungsstrukturen zwischen den Klassen aus dem ersten Beispiel, was jedoch nicht immer sein muss. In Abbildung 10 kann man dies an den mit Punkten gerasterten Klassen erkennen. Sie entsprechen den Klassen des Beispiels Abbildung 5. Da sich alternative Ausklammerungsmöglichkeiten ergeben können, könnten sich die Verbindungsstrukturen ebenfalls ändern. Dies hat jedoch auf die „*logisch - mathematische*“ Qualität der gefundenen Lösung keinen Einfluss, denn jede nach den Regeln des Distributivgesetzes der Booleschen Algebra gefundene Lösung ist eine korrekte Lösung. Im zweiten System in Abbildung 10 sind gegenüber dem System in Abbildung 5 durch die Einführung des Phänomens „*Dreiecks abhängig*“ und des Phänomens „*Vieleck*“ keine zusätzlichen Klassenelemente entstanden. Alles, was zur Bearbeitung von Dreiecken aus dem ersten System vorhanden war, ist vom Algorithmus im zweiten System voll benutzt worden. Es entstanden keine mehrdeutigen Zustand- oder Prozesselemente. Dies liegt in der ordnenden Strukturierungseigenschaft dieser Vorgehensweise, abgeleitet aus der systematisch ordnenden Strukturierungseigenschaft des Distributivgesetzes der Booleschen Algebra. Man kann erkennen, dass diese ordnende Eigenschaft auch bei zu erweiternden Software - Systemen vorteilhaft ist, in dem der Algorithmus gewährleistet, dass die Eigenschaften des alten und neuen Systems vollständig *konsolidiert* sind und damit *keine* mehrdeutigen Eigenschaften aufweisen.

Inkrementalismus. Das zweite System in Abbildung 10 ist gegenüber dem ersten System in Abbildung 5 wesentlich komplexer, es hat sich eine Mehrfachvererbung ergeben. In der intuitiven Vorgehensweise würde man möglicherweise der „*menschlichen*“ Eigenschaft folgen und versuchen die seitherige Struktur in Abbildung 5 dahingehend zu erweitern, dass sie die neuen Funktionen mit erfüllen kann. Unter anderem würde möglicherweise versucht werden, mit Hilfe von Overloading - Funktionen der Programmiersprachen zu arbeiten, was natürlich die Mehrdeutigkeiten fördern könnte. Die Sicherheit und die Verlässlichkeit der Software würde darunter sicherlich leiden. Durch entsprechende Testreihen würde man versuchen, dem Problem Herr zu werden und dem neuen System eine möglicherweise vermeintliche Stabilität geben, die aber mit Overloading – Funktionen nicht erreicht werden kann. Denn Overloading – Funktionen, so gut und praktikabel sie auch sein mögen, haben eher den Charakter eines „*Schaltersystems*“, das in seiner Dynamik nur sehr schwer zu kontrollieren ist. Sie sollten deshalb in sicherheitssensitiven Softwareanwendungen eher gemieden werden.
Der Mensch neigt in seinem Problemlösungsverhalten dazu, den Inkrementalismus (das Hindurchwursteln, Muddling Through) zu benutzen, um die nächst beste Lösung zu erreichen. Erweist sich einmal ein eingeschlagener Weg als unzweckmäßig oder gar als vermeintliche Sackgasse, so kehrt man nicht zum Eingang des Labyrinths zurück, sondern lediglich zur letzten Abzweigung, wo man versucht, einen anderen der möglichen Pfade zu gehen. Man versucht bei einem Misserfolg in der Lösung eines Subproblems das nächst höhere Subproblem auf eine andere Weise zu lösen. Erst wenn daraus immer neue, weitere Subprobleme entstehen, die nicht befriedigend gelöst werden könnten, würde man versuchen ein übergeordnetes Problem neu zu lösen. Die Gründe für den Inkrementalismus sind darin zu sehen, dass man nur fragmentarische Informationen über die mutmaßlichen Konsequenzen „*großer*“ Änderungen zur Verfügung hat und Entscheidungen unter Unsicherheit aus dem Weg geht. Man realisiert Maßnahmen, die „in der Nähe“ des Status quo liegen, weil man sich nur bei kleinen, minimalen (optischen!) Änderungen vorzustellen vermag, welche Auswirkungen insgesamt zu erwarten sind. Darüber hinaus sind kleine Änderungen bei entdeckten Fehlern oder sonstigen Änderungen in nachfolgend notwendigen Schritten leichter zu korrigieren als große [KiW]. Wie später gezeigt werden wird, können minimale Änderungen im seitherigen System u. U. zu „*chaotischen Änderungen*“ in der Topologie der Vererbungsstrukturen führen.
Ein Zurückgehen auf den Punkt im Entscheidungsverhalten, der wieder alle optimalen Lösungen neu eröffnet, würde man bei einem intuitiven Vorgehen versuchen zu vermeiden. Wie das gesamte System nach der zweiten, dritten oder noch weitergehenden Änderung sich verändert hat, wenn man intuitiv vorgegangen ist und iterativ geändert hat, mag sich jeder vorstellen, der solche Änderungen in Programme schon einmal eingebracht hat. Selbst wenn ein Programmierer seine eigenen Programme nach einiger Zeit ändern muss, tut er sich oft schon schwer, durch sein seitheriges Konstrukt sicher hindurch zu navigieren.

Wie viel mühevoller ist es, ein fremdes Programm zu ändern und welche Gefahren für die Sicherheit und Verlässlichkeit des Programmsystems sind hier bei kritischen Anwendungen von lebenserhaltenden Systemen gegeben. Die beschriebene Vorgehensweise hingegen ermittelt immer die neuen, jetzt optimalen Strukturen von Anfang an.

Komponierende Abstraktionen des erweiterten Beispiels. Die folgenden Strukturen zeigen die Klassen nach Abbildung 10 in ihrer Form als *komponierende Abstraktionen*, wie sie von Anwendungsprogrammierern respektive von Benutzern (Mensch, Maschine) als lineare Schnittstellen bevorzugt gesehen werden. In der *komponierten Abstraktion* „***KompQuadrat***“ sind die generalisierten / spezialisierten Klassen „***GenGeoFigur***“, „***GenKoordin***“, „***GenGeometrie***“ und „***GenQuadrat***“ mit den durch den Prozess zugeordneten Eigenschaften in der ermittelten, aufgezeigten Vererbungsstruktur, sowohl für den Anwendungsprogrammierer als auch für den Benutzer (Mensch, Maschine), erreichbar, was in Abbildung 11 durch die markierte Fläche signalisiert wird:

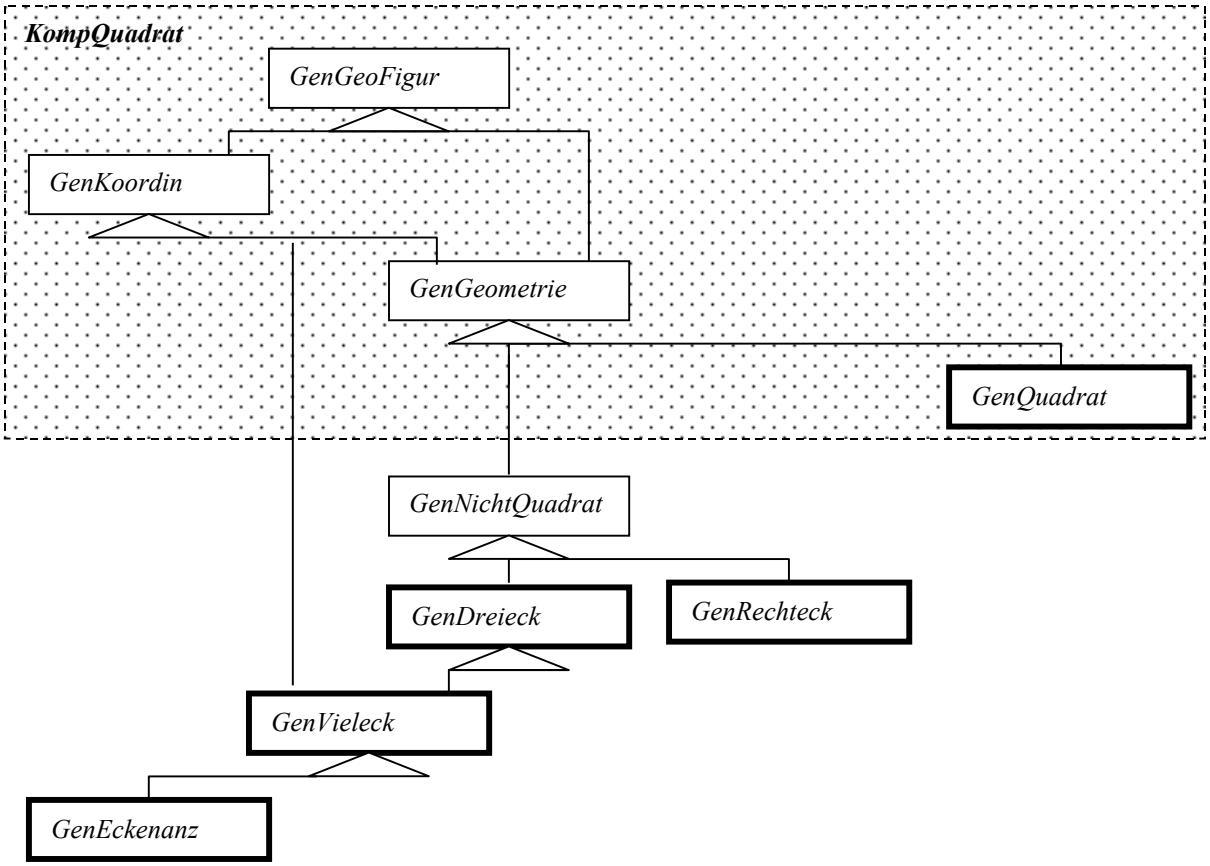


Abbildung 11: *Komponierende Abstraktion* „*KompQuadrat*“ über der *generalisierenden Abstraktion*, welche ihrerseits die generalisierten/spezialisierten Klassen „*GenGeoFigur*“, „*GenKoordin*“, „*GenGeometrie*“, und „*GenQuadrat*“ umfasst

In der *komponierten Abstraktion* „***KompRechteck***“ sind die generalisierten/spezialisierten Klassen „***GenGeoFigur***“, „***GenKoordin***“, „***GenGeometrie***“, „***GenNichtQuadrat***“ und „***GenRechteck***“ mit den durch den Prozess zugeordneten Eigenschaften in der ermittelten, aufgezeigten Vererbungsstruktur, sowohl für den Anwendungsprogrammierer als auch für den Benutzer (Mensch, Maschine), erreichbar, was in Abbildung 12 durch die markierte Fläche signalisiert wird:

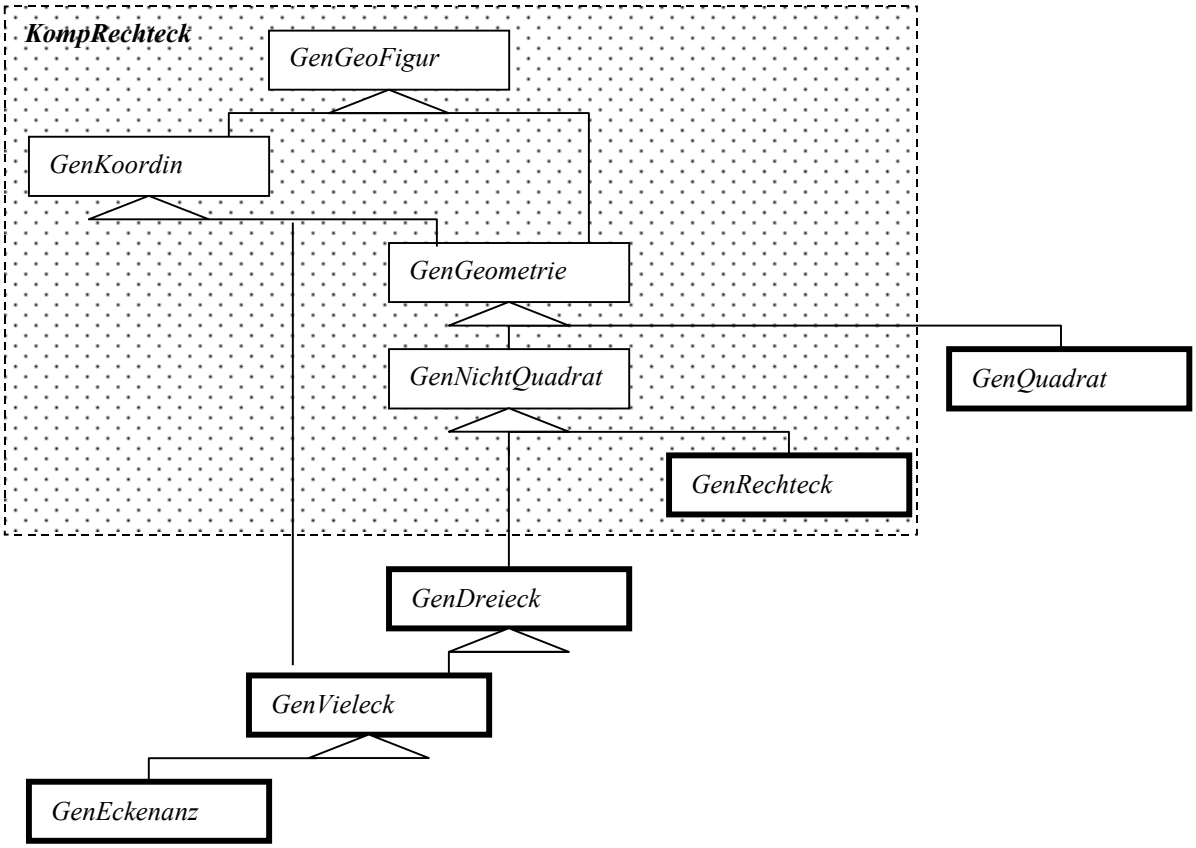


Abbildung 12: *Komponierende Abstraktion* „*KompRechteck*“ über der *generalisierenden Abstraktion*, welche ihrerseits die generalisierten/spezialisierten Klassen „*GenGeoFigur*“, „*GenKoordin*“, „*GenGeometrie*“, „*GenNichtQuadrat*“ und „*GenRechteck*“ umfasst

In der *komponierten Abstraktion* „***KompDreieck***“ sind die generalisierten/spezialisierten Klassen „***GenGeoFigur***“, „***GenKoordin***“, „***GenGeometrie***“, „***GenNichtQuadrat***“ und „***GenDreieck***“ mit den durch den Prozess zugeordneten Eigenschaften in der ermittelten, aufgezeigten Vererbungsstruktur, sowohl für den Anwendungsprogrammierer als auch für den Benutzer (Mensch, Maschine), erreichbar, was in Abbildung 13 durch die markierte Fläche signalisiert wird:

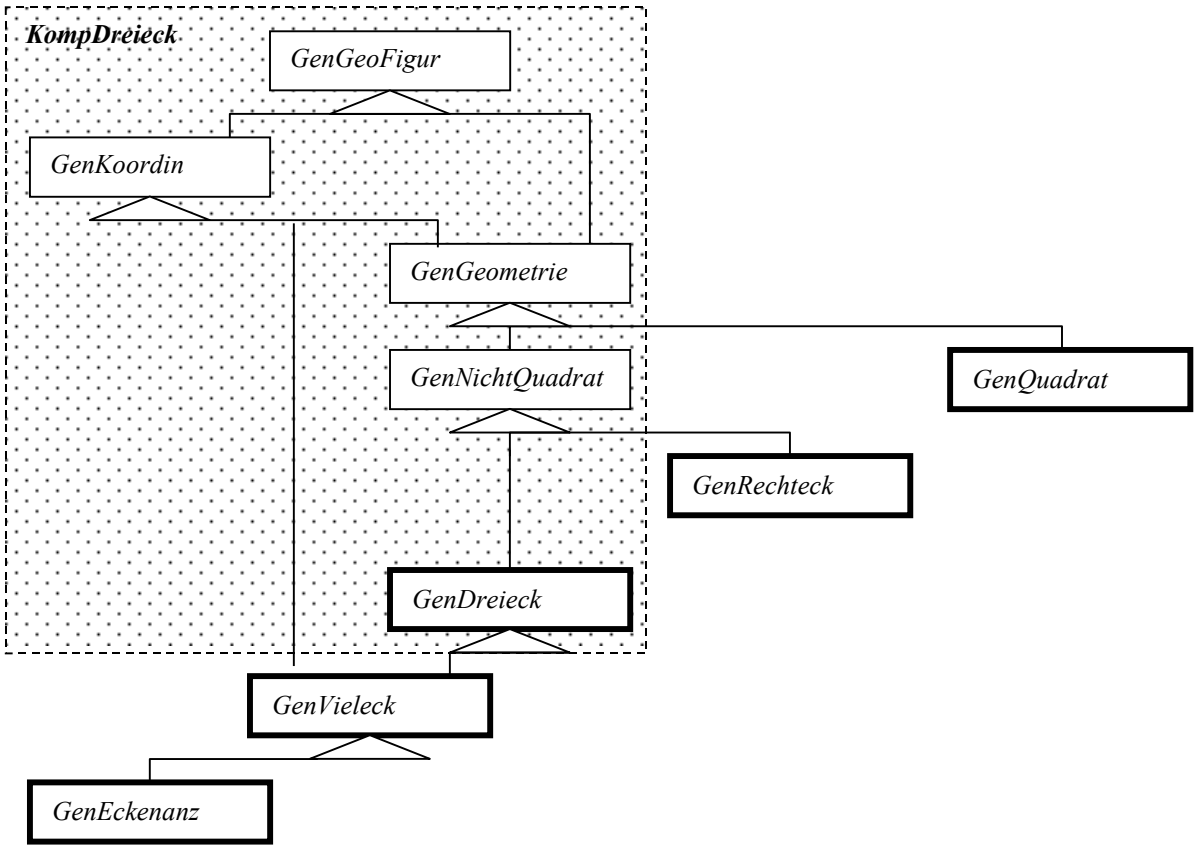


Abbildung 13: *Komponierende Abstraktion* „*KompDreieck*“ über der *generalisierenden Abstraktion*, welche ihrerseits die generalisierten/spezialisierten Klassen „*GenGeoFigur*“, „*GenKoordin*“, „*GenGeometrie*“, „*GenNichtQuadrat*“ und „*GenDreieck*“ umfasst

In der *komponierenden Abstraktion* „***KompVieleck***“ sind die generalisierten/spezialisierten Klassen „***GenGeoFigur***“, „***GenKoordin***“, „***GenVieleck***“ und „***GenAnzahl***“ mit den durch den Entwicklungsprozess zugeordneten Eigenschaften in der ermittelten, aufgezeigten Vererbungsstruktur, sowohl für den Anwendungsprogrammierer als auch für den Benutzer (Mensch, Maschine), erreichbar, was in Abbildung 14 durch die markierte Fläche signalisiert wird. Gleichzeitig sind aus der *komponierenden Abstraktion* „***KompVieleck***“ heraus die *komponierende Abstraktion* „***KompDreieck_abh***“ *n* – Mal über das Konstrukt der programmtechnischen *Aggregation* mit ihren generalisierten/spezialisierten Klassen „***GenGeoFigur***“, „***GenGeometrie***“, „***GenNichtQuadrat***“, „***GenDreieck***“ und „***GenVieleck***“ mit den durch den Entwicklungsprozess zugeordneten Eigenschaften in der ermittelten, aufgezeigten Vererbungsstruktur, sowohl für den Anwendungsprogrammierer als auch für den Benutzer (Mensch, Maschine), erreichbar, was ebenfalls in Abbildung 14 durch die markierte Fläche signalisiert wird:

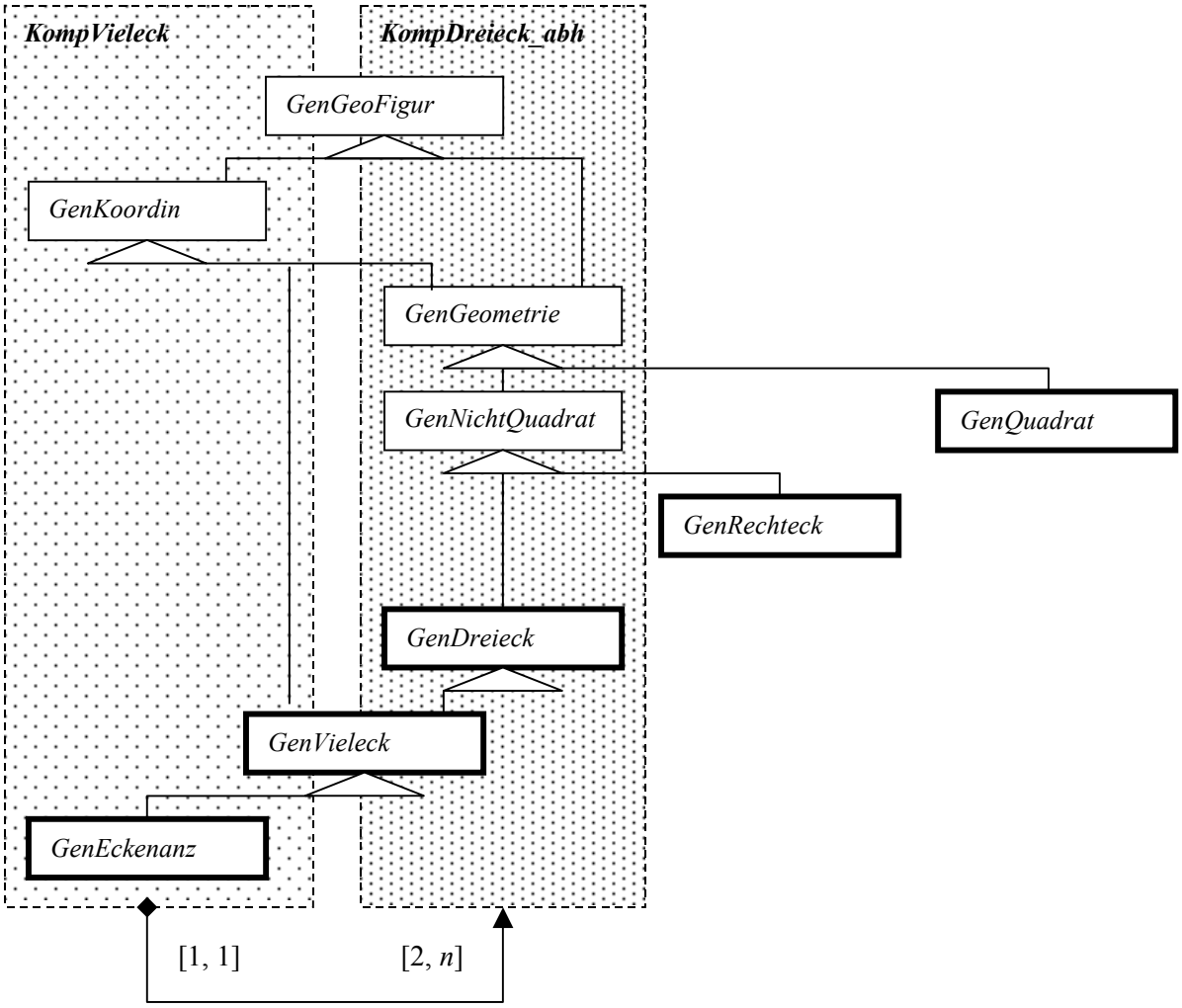


Abbildung 14: Die beiden k*omponierenden Abstraktionen* „*KompVieleck*“ und „*KompDreieck_abh*“, die ihrerseits zueinander eine [1, 1], [2, *n*] – Aggregations - Beziehung besitzen, stehen gemeinsam über der *generalisierenden* Abstraktion. Die *komponierende Abstraktion* „*KompVieleck*“ umfasst die *generalisierenden*/*spezialisierenden* Klassen „*GenKoordin*“ und „*GenEckenanz*“, die *komponierende Abstraktion* „*KompDreieck_abh*“ umfasst die *generalisierenden*/*spezialisierenden* Klassen „*GenGeometrie*“, „*GenNichtQuadrat*“ und „*GenDreieck*“. Die Funktionalität der generalisierten / spezialisierten Klassen „*GenGeoFigur*“ und „GenVieleck“ teilen sich die beiden komponierenden Abstraktionen „*KompVieleck*“ und „*KompDreick_abh*“.

Die Merkmale der beiden generalisierten/spezialisierten Klassen „*GenGeoFigur*“ und „*GenVieleck*“ werden für beide *komponierenden Abstraktionen* „*KompVieleck*“ und „*KompDreieck_abh*“ benötigt. So werden z. B. beide *komponierenden Abstraktionen* betreffende Daten oder Methoden der Summenbildung in der Klasse „*GenVieleck*“ benutzt. Eine redundante Realisierung ist auszuschließen, denn die in beiden Klassen enthaltenen Eigenschaften werden je nur einmal programmseitig ausgeprägt. Um die beiden Klassen in den beiden *komponierenden Abstraktionen* „*KompVieleck*“ und „*KompDreieck_abh*“ nutzen zu können, die beide als lineare Schnittstellen fungieren, wird beispielsweise in einem C++ Programm jede der beiden Klassen mit dem Eintrag „*friend KompVieleck*“ bzw. „*friend KompDreieck_abh*“ versehen. Die in Abbildung 14 gezeigte Darstellung reflektiert ein Modell einer [1, 1], [2, *n*] - Aggregations - Beziehung. Für jede Ausprägung einer in einer „*KompVieleck*“ - Komposition ausgeprägten Klasse [min = 1, max = 1] werden die in mindestens zwei, maximal *n* „*KompDreick_abh*“ - Kompositionen angesprochenen Klassen [min = 2, max = *n*] für *n* Dreiecke, aus denen ein darzustellendes Vieleck besteht, ausgeprägt, wobei die beiden Klassen „*GenGeoFigur*“ und „*GenVieleck*“ von beiden Kompositionen gleichzeitig angesprochen werden. Realisierte Eigenschaften aus diesen Klassen werden von beiden Kompositionen benutzt. Das folgende in der Programmiersprache C++ realisierte Programm repräsentiert die Umsetzung des in Tabelle 6 dargestellten Programmsystems und der daraus abgeleiteten Gleichungen, sowie das Generalisierungsmodell der Abbildung 10:

```
#include <iostream.h>                          // Standardbibliothek zu C++
#include <math.h>
// Globale Variable definieren
static int   n;
static int   EingW;

class GenGeoFigur;
class GenKoordin;
class GenEckenanz;
class GenNichtQuadrat;
class GenGeometrie;
class GenQuadrat;
class GenRechteck;
class GenDreieck;
class GenVieleck;

// Komposition Quadrat definieren
class KompQuadrat       /* Zweckorientierte Objektsicht (Komposition) */
{ public:   void zeigen (GenQuadrat a); };

// Komposition Rechteck definieren
class KompRechteck           /* Zweckorientierte Objektsicht (Komposition)
*/
{ public:   void zeigen (GenRechteck a); };

// Komposition Dreieck definieren
class KompDreieck       /* Zweckorientierte Objektsicht (Komposition) */
{ public:   void zeigen (GenDreieck a); };

// Komposition Dreieck abhängig definieren
class KompDreieck_abh   /* Zweckorientierte Objektsicht (Komposition) */
{ public:   void zeigen_Dreieck_abh (GenVieleck a); };

// Komposition Vieleck definieren
class KompVieleck       /* Zweckorientierte Objektsicht (Komposition) */
{ public:   void zeigen_Beginn      (GenEckenanz a);
                 void zeigen_Schluss     (GenEckenanz a, GenVieleck b); };
// Generalisierungen / Spezialisierungen
/* :=Generalisierung    Wurzel                                        */
class GenGeoFigur
{
protected:  double      Objekt_ID;                          /* a6 */
                 void  Objekt_ID_belegen ();
};
```

```
/* Spezialisierung : Generalisierung (Einfachvererbung)  */
class GenKoordin   : virtual protected GenGeoFigur
{
protected:  void  x_y_belegen ();
                  float x;                                        /* a8 */
                  float y;                                        /* a9 */
};

/* Spezialisierung : Generalisierungen (Mehrfachvererbung)  */
class GenGeometrie : virtual protected GenKoordin, virtual protected
GenGeoFigur
{
protected:  void  Seite_a_belegen ();
                  double      Flaeche;                            /* a4 */
                  float Seite_a;                            /* a1 */
};

/* Spezialisierung    : Generalisierung  (Einfachvererbung)  */
class GenNichtQuadrat : virtual protected GenGeometrie
{
protected:  void  Seite_b_belegen ();
                  float Seite_b;                            /* a2 */
};

/* Spezialisierung : Generalisierung (Einfachvererbung)  */
class GenDreieck   : virtual protected GenNichtQuadrat
{
public:           void  D_belegen ();
protected:  void  Seite_c_belegen_1 ();
                  void  Seite_c_belegen_2 ();
                  void  Dreieck_erzeugen ();
                  float Halbe_Seiten;                       /* a5 */
                  float Seite_c;                            /* a3 */
                  friend      KompDreieck;
};
/* Spezialisierung : Generalisierungen (Mehrfachvererbung)  */
class GenVieleck   : virtual protected GenKoordin, protected GenDreieck
{
public:           void  S_belegen ();
                  void  V_belegen ();
protected:  void  Summe_Vieleck_belegen ();     /* p12 */
                  void  Summe_Vieleck_erzeugen ();
                  void  Summe_Vieleck_init ();
                  int         Lfd_Nr;                                 /*
a7  */
                  double      Summe_Vieleck;                      /* a10 */
                  friend      KompDreieck_abh;
                  friend      KompVieleck;
};

/* Spezialisierung : Generalisierung (Einfachvererbung)  */
class GenQuadrat   : protected GenGeometrie
{
public:           void  Q_belegen ();
protected:  void  Quadrat_belegen ();
                  void  Quadrat_erzeugen ();
                  friend      KompQuadrat;
};

/* Spezialisierung : Generalisierung (Einfachvererbung)  */
class GenRechteck  : protected GenNichtQuadrat
{
public:           void  R_belegen ();
protected:  void  Rechteck_belegen ();
```

```
                  void  Rechteck_erzeugen ();
                  friend      KompRechteck;
};

/* Spezialisierung : Generalisierung (Einfachvererbung)  */
class GenEckenanz  : protected GenVieleck
{
public:           void  E_belegen ();
protected:  void  Eckenanzahl_belegen ();
                  void  Eckenanzahl_erzeugen ();
                  int         Eckenanzahl;                        /* a11 */
                  friend      KompVieleck;
};

// Definitionen zu den Klassen der linearen Schnittstellen
// (Kompositionen der Zweckorientierten Objektsichten)
void KompQuadrat :: zeigen (GenQuadrat a)
{
   cout << "----- Schnittstelle Quadrat ---------" <<                  "\n";
   cout << "Objekt-ID                          " << a.Objekt_ID <<  "\n";
   cout << "x-Koordinate                       " << a.x       <<    "\n";
   cout << "y-Koordinate                       " << a.y       <<    "\n";
   cout << "Flaeche                            " << a.Flaeche <<    "\n";
   cout << "Seite a                            " << a.Seite_a <<    "\n";
   cout << "----- Ende Schnittstelle Quadrat ----" <<                 "\n";
}

void KompRechteck :: zeigen (GenRechteck a)
{
   cout << "----- Schnittstelle Rechteck --------" <<         "\n";
   cout << "Objekt-ID                          " << a.Objekt_ID <<  "\n";
   cout << "x-Koordinate                       " << a.x       <<    "\n";
   cout << "y-Koordinate                       " << a.y       <<    "\n";
   cout << "Flaeche                            " << a.Flaeche <<    "\n";
   cout << "Seite a                            " << a.Seite_a <<    "\n";
   cout << "Seite b                            " << a.Seite_b <<    "\n";
   cout << "----- Ende Schnittstelle Rechteck ---" <<         "\n";
}

void KompDreieck :: zeigen (GenDreieck a)
{
   cout << "----- Schnittstelle Dreieck_autonom ----------" <<        "\n";
   cout << "Objekt-ID                          " << a.Objekt_ID <<  "\n";
   cout << "x-Koordinate                       " << a.x       <<    "\n";
   cout << "y-Koordinate                       " << a.y       <<    "\n";
   cout << "Flaeche                            " << a.Flaeche <<    "\n";
   cout << "Seite a                            " << a.Seite_a <<    "\n";
   cout << "Seite b                            " << a.Seite_b <<    "\n";
   cout << "Seite c                            " << a.Seite_c <<    "\n";
cout << "1/2(a+b+c)(Halbe Seiten nach Heron)" << a.Halbe_Seiten <<    "\n";
   cout << "----- Ende Schnittstelle Dreieck_autonom -----" <<       "\n";
}

void KompDreieck_abh :: zeigen_Dreieck_abh (GenVieleck a)

{
   cout << "----- Schnittstelle Vieleck einzeln --" <<         "\n";
   cout << "Objekt-ID                          " << a.Objekt_ID <<      "\n";
   cout << "Lfd Nr. Dreieck im Vieleck         " << a.Lfd_Nr    <<      "\n";
   cout << "Seite a                            " << a.Seite_a   <<      "\n";
   cout << "Seite b                            " << a.Seite_b   <<      "\n";
   cout << "Seite c                            " << a.Seite_c   <<      "\n";
cout << "1/2(a+b+c)(Halbe Seiten nach Heron) " << a.Halbe_Seiten  <<  "\n";
   cout << "----- Ende Schnittstelle Vieleck einzeln --" <<         "\n";
}
```

```
void KompVieleck :: zeigen_Beginn (GenEckenanz a)
{
   cout << "----- Schnittstelle Vieleck Beginn --" <<         "\n";
   cout << "Objekt-ID                          " << a.Objekt_ID <<  "\n";
   cout << "x-Koordinate                       " << a.x       <<    "\n";
   cout << "y-Koordinate                       " << a.y       <<    "\n";
   cout << "Vieleck Eckenanzahl             " << a.Eckenanzahl <<    "\n";
   cout << "----- Ende Schnittstelle Vieleck Beginn -----" <<         "\n";
}

void KompVieleck :: zeigen_Schluss (GenEckenanz a, GenVieleck b)
{
   cout << "----- Schnittstelle Vieleck Schluss ----------" <<        "\n";
   cout << "Objekt-ID                          " << a.Objekt_ID <<  "\n";
   cout << "x-Koordinate                       " << a.x       <<    "\n";
   cout << "y-Koordinate                       " << a.y       <<    "\n";
   cout << "Flaeche Vieleck                " << b.Summe_Vieleck <<    "\n";
   cout << "Vieleck Eckenanzahl                " << a.Eckenanzahl << "\n";
   cout << "---- Ende Schnittstelle Vieleck Schluss -----" <<         "\n";
}

void GenGeoFigur :: Objekt_ID_belegen ()
{
      Objekt_ID = 123;
}

// Definitionen zu Klasse GenKoordin
void GenKoordin :: x_y_belegen ()
{
      x = 15;
      y = 27;
}

// Definitionen zu Klasse GenEckenanz
void GenEckenanz :: E_belegen ()
{
      Eckenanzahl_erzeugen ();
}

void GenEckenanz :: Eckenanzahl_erzeugen ()
{
      Objekt_ID_belegen ();
      x_y_belegen ();
      Eckenanzahl_belegen ();
      n = Eckenanzahl - 2;                          /* p11 */
}

void GenEckenanz :: Eckenanzahl_belegen ()
{
      cout << "Anzahl Ecken Vieleck > 3 eingeben:"<<"\n";
      cin >> Eckenanzahl;
}

// Definitionen zu Klasse GenDreieck
void GenDreieck :: Seite_c_belegen_1 ()
{
      cout << "Seite c eingeben:"<<"\n";
      cin >> Seite_c;
}

void GenDreieck :: Seite_c_belegen_2 ()
{
      if ((Seite_a + Seite_b > Seite_c)*
            (Seite_b + Seite_c > Seite_a)*
            (Seite_a + Seite_c > Seite_b))                    /* p3  */
```

```
      {     Halbe_Seiten = (Seite_a + Seite_b + Seite_c) / 2;    /* p6  */
            Flaeche  = sqrt (Halbe_Seiten *
                                (Halbe_Seiten - Seite_a) *
                                (Halbe_Seiten - Seite_b) *
                               (Halbe_Seiten - Seite_c));     /* p7  */
      }
}

void GenDreieck :: Dreieck_erzeugen ()
{
      Objekt_ID_belegen ();
      x_y_belegen ();
      cout << "Auspraegungen des Dreiecks eingeben. " << "\n";
      Seite_a_belegen ();
      Seite_b_belegen ();
      Seite_c_belegen_1 ();
      Seite_c_belegen_2 ();
}

void GenDreieck :: D_belegen ()
{    Dreieck_erzeugen (); }

// Definitionen zu Klasse GenVieleck
void GenVieleck :: V_belegen ()
{     Summe_Vieleck_init (); }

void GenVieleck :: S_belegen ()
{     Summe_Vieleck_erzeugen ();
      Summe_Vieleck_belegen ();
}

void GenVieleck :: Summe_Vieleck_erzeugen ()
{
      cout << "Auspraegungen des Vielecks eingeben: " << "\n";
}

void GenVieleck :: Summe_Vieleck_init ()
{
      Objekt_ID_belegen ();
      x_y_belegen ();
      Summe_Vieleck = 0;                                        /* p8  */
      Lfd_Nr = 0;                                               /* p8  */
}

void GenVieleck :: Summe_Vieleck_belegen ()
{
      Lfd_Nr = Lfd_Nr + 1;                                  /* p10 */
      Dreieck_erzeugen ();
      Summe_Vieleck = Summe_Vieleck + Flaeche;  /* p9  */
      n = n - 1;
}

// Definitionen zu Klasse GenRechteck
void GenRechteck :: R_belegen()
{    Rechteck_erzeugen ();    }

void GenRechteck :: Rechteck_belegen ()
{
      if ((Seite_a > 0) * (Seite_b > 0) * (Seite_a != Seite_b))   /* p2  */
      {
            Flaeche = (Seite_a * Seite_b);                        /* p5  */
      }
}

void GenRechteck :: Rechteck_erzeugen ()
```

```
{
      Objekt_ID_belegen ();
      x_y_belegen ();
      cout << "Auspraegungen des Rechtsecks eingeben. " << "\n";
      Seite_a_belegen ();
      Seite_b_belegen ();
      Rechteck_belegen ();
}

// Definitionen zu Klasse GenNichtQuadrat
void GenNichtQuadrat :: Seite_b_belegen ()
{
      cout << "Seite b eingeben:"<<"\n";
      cin >> Seite_b;
}

// Definitionen zu Klasse GenQuadrat
void GenQuadrat :: Q_belegen()
{    Quadrat_erzeugen ();     }

void GenQuadrat :: Quadrat_belegen ()
{
      if (Seite_a > 0)                    /* p1  */
      {
            Flaeche = (Seite_a * Seite_a);  /* p4  */
      }
}

void GenQuadrat :: Quadrat_erzeugen ()
{
      Objekt_ID_belegen ();
      x_y_belegen ();
      cout << "Auspraegungen des Quadrats eingeben. " << "\n";
      Seite_a_belegen ();
      Quadrat_belegen ();
}

// Definitionen zu Klasse GenGeometrie
void GenGeometrie :: Seite_a_belegen ()
{
      cout << "Seite a eingeben:"<<"\n";
      cin >> Seite_a;
}

// Ansteuerung der Klassen
void main()
{
      cout << "\n" << "Flaeche berechnen: 1 Quadrat, 2 Rechteck, 3 Dreieck,
4 Vieleck" << "\n" << "\n";
      cout << "Geben Sie bitte die Zahl der gewuenschten Aktion ein"<<
"\n";
      cin>> EingW;
      if (EingW == 1)
      {
            GenQuadrat UFQO;
            UFQO.Q_belegen ();
            KompQuadrat Q_Komposition;
            Q_Komposition.zeigen(UFQO);
      }
      if (EingW == 2)
      {
            GenRechteck UFRO;
            UFRO.R_belegen();
            KompRechteck R_Komposition;
            R_Komposition.zeigen(UFRO);
      }
```

```
      if (EingW == 3)
      {
            GenDreieck UFDO;
            UFDO.D_belegen();
            KompDreieck D_Komposition;
            D_Komposition.zeigen(UFDO);
      }
      if (EingW == 4)
      {
            GenEckenanz UFEA;
            UFEA.E_belegen ();
            GenVieleck UFVE;
            UFVE.V_belegen ();
            KompVieleck VB_Komposition;
            VB_Komposition.zeigen_Beginn (UFEA);
            do
            {
                  UFVE.S_belegen ();
                  KompDreieck_abh Da_Komposition;
                  Da_Komposition.zeigen_Dreieck_abh (UFVE);
            }
            while (n > 0);
            KompVieleck VS_Komposition;
            VS_Komposition.zeigen_Schluss (UFEA, UFVE);
      }
      cout << "Ende" << "\n"; /* ~ */
}
```

Fasst man persistent instanziierte Ausprägungen als instanziierte Tupel der durch Anwendung des Distributivgesetzes der Booleschen Algebra entstandenen Generalisierungs-/Spezialisierungsstruktur auf, lässt sich dieser Tupel - Bestand durch logisches Ausmultiplizieren in einen Tupel - Bestand überführen, der dem Schema eines Ausdrucks in disjunktiver Normalform entspricht. Von diesem Zustand aus lassen sich die Tupel leicht in eine Ausprägungsstruktur überführen, die dem Schema des neu gebildeten, jetzt optimalen Vererbungsstrukturausdrucks entspricht.

Alternative Ausklammerung zur Struktur der Abbildung 10. Wie sich in **Tabelle 6** in der Spalte acht (Anzahl) zeigt, hätte man in **Gleichung 9** auch die beiden Booleschen Variablen $\boldsymbol{a_1}\ \boldsymbol{a_4}$ anstatt der beiden Booleschen Variablen $\boldsymbol{a_8}\ \boldsymbol{a_9}$ zum Ausklammern benutzen können, denn diese Booleschen Variablen kommen ebenfalls viermal vor. Bei einer Ausklammerung von $\boldsymbol{a_1}\ \boldsymbol{a_4}$ in **Gleichung 9** ergibt sich **Gleichung 17**:

Gleichung 17

$$\boldsymbol{a_6}\ (\boldsymbol{a_1}\ \boldsymbol{a_4}\ (a_8\ a_9\ p_1\ p_4 \vee\ a_2\ a_8\ a_9\ p_2\ p_5 \vee\ a_2\ a_3\ a_5\ \ a_8\ a_9\ p_3\ p_6\ p_7 \vee a_2\ a_3\ a_5\ \ a_7\ a_{10}\ p_3\ p_6\ p_7\ p_8\ p_9\ p_{10}) \vee\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ p_8\ p_9\ p_{10}\ p_{11}) = 1.$$

Daraus ergibt sich eine notwendige Nebengeneralisierung/-spezialisierung (siehe auch [EnG]):

<u>Nebengeneralisierung/-spezialisierung 1:</u>

Gleichung 18

$$\boldsymbol{a_8}\ \boldsymbol{a_9}\ p_1\ p_4 \vee\ a_2\ \boldsymbol{a_8}\ \boldsymbol{a_9}\ p_2\ p_5 \vee\ a_2\ a_3\ a_5\ \ \boldsymbol{a_8}\ \boldsymbol{a_9}\ p_3\ p_6\ p_7 \vee a_2\ a_3\ a_5\ \ a_7\ a_{10}\ p_3\ p_6\ p_7\ p_8\ p_9\ p_{10} \vee\ a_7\ \boldsymbol{a_8}\ \boldsymbol{a_9}\ a_{10}\ a_{11}\ p_8\ p_9\ p_{10}\ p_{11} = 1.$$

Gleichung 19

$$\boldsymbol{a_8}\ \boldsymbol{a_9}[p_1\ p_4 \vee a_2\ p_2\ p_5 \vee a_2\ a_3\ a_5\ \ p_3\ p_6\ p_7 \vee a_7\ a_{10}\ a_{11}\ p_8\ p_9\ p_{10}\ p_{11}] \vee a_2\ a_3\ a_5\ a_7\ a_{10}\ p_3\ p_6\ p_7\ p_8\ p_9\ p_{10} = 1.$$

Nebengeneralisierung/-spezialisierung 2:

Gleichung 20

$\boldsymbol{a_2}\, p_2\, p_5 \vee\ \boldsymbol{a_2}\, a_3\, a_5\ p_3\, p_6\, p_7 \vee\ a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} \vee \boldsymbol{a_2}\, a_3\, a_5\ a_7\, a_{10}\, p_3\, p_6\, p_7\, p_8\, p_9 p_{10} = 1.$

Gleichung 21

$\boldsymbol{a_2}\, [p_2\, p_5 \vee\ \boldsymbol{a_3}\, \boldsymbol{a_5}\ \boldsymbol{p_3}\, \boldsymbol{p_6}\, \boldsymbol{p_7} \vee \boldsymbol{a_3}\, \boldsymbol{a_5}\ a_7\, a_{10}\, \boldsymbol{p_3}\, \boldsymbol{p_6}\, \boldsymbol{p_7}\, p_8\, p_9 p_{10}] \vee\ a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Gleichung 22

$\boldsymbol{a_2}\, [p_2\, p_5 \vee\ \boldsymbol{a_3}\, \boldsymbol{a_5}\ \boldsymbol{p_3}\, \boldsymbol{p_6}\, \boldsymbol{p_7}\, [1 \vee a_7\, a_{10}\, p_8\, p_9 p_{10}]] \vee\ a_7\, a_{10}\, a_{11}\, p_8\, p_9\, p_{10}\, p_{11} = 1.$

Nebengeneralisierung/-spezialisierung 3:

Gleichung 23

$\boldsymbol{a_7}\, \boldsymbol{a_{10}}\, \boldsymbol{p_8}\, \boldsymbol{p_9 p_{10}} \vee\ \boldsymbol{a_7}\, \boldsymbol{a_{10}}\, a_{11}\, \boldsymbol{p_8}\, \boldsymbol{p_9}\, \boldsymbol{p_{10}}\, p_{11} = 1.$

Gleichung 24

$\boldsymbol{a_7}\, \boldsymbol{a_{10}}\, \boldsymbol{p_8}\, \boldsymbol{p_9 p_{10}}\, [1 \vee\ a_{11}\, p_{11}] = 1.$

Die graphische Interpretation der Konsolidierung der Gleichung 17, Gleichung 19, Gleichung 22 und Gleichung 24 ergibt Abbildung 15. Diese Generalisierungs-/Spezialisierungsmodell besitzt gegenüber dem Modell in Abbildung 10 insgesamt elf Strukturverbindungen. Abbildung 10 besitzt nur zehn Strukturverbindungen. Die beiden Modelle der Abbildungen 10 und 15 sind funktional völlig gleichwertig. Abbildung 15 ist die zweite Möglichkeit, den Vererbungsgraphen in dieser Vorgehensweise aus Tabelle 6 abzuleiten. Da das Strukturmodell nach Abbildung 10 weniger Komplexität aufweist als das Strukturmodell nach Abbildung 15, wäre nach ökonomischen Entscheidungskriterien das Strukturmodell nach Abbildung 10 dem Strukturmodell nach Abbildung 15 vorzuziehen. Die beiden Lösungen sind die einzigen Lösungen, die nach dieser Vorgehensweise durch Benutzung des Distributivgesetzes der Booleschen Algebra angesagt sind. Bei neun Klassen, die sich aus dem Strukturierungsprozess ergeben haben, gäbe es theoretisch insgesamt eine Komplexität von $K = n(n\text{-}1)/2 = 9(8)/2 = 36$ und somit eine Variabilität von $VA = 2^{n(n\text{-}1)/2} = 2^{36} = 68.719.476.736$, also 68.719.476.736 Möglichkeiten, diese Klassen anzuordnen, wie nach Tabelle 1 zu ersehen ist. Auch intuitive Entwickler würden die meisten der 68.719.476.736 Strukturierungsmöglichkeiten wegen mangelnder zielführender Sinnhaftigkeit ausschließen. Dennoch müsste man bei unterschiedlichen intuitiven Entwicklern bei 68.719.476.736 Möglichkeiten, neun Klassen intuitiv strukturell anzuordnen, sicherlich mit mehr als zwei unterschiedlichen Lösungsstrukturen rechnen.

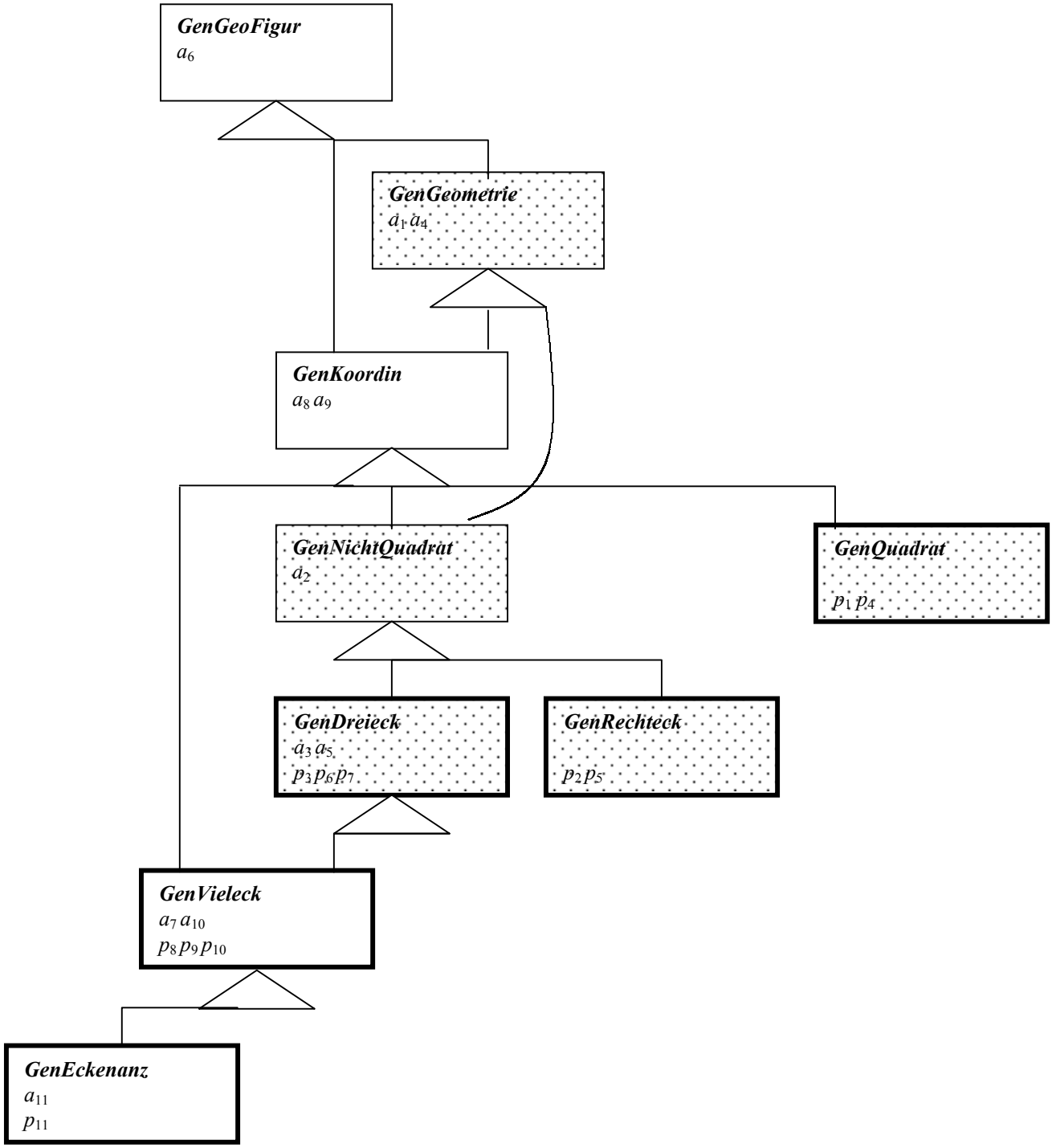


Abbildung 15 *Generalisierende Abstraktion*, Mehrfachvererbung, abgeleitet aus Tabelle 6, Klassendiagramm (alternative Ausklammerung) mit elf Strukturverbindungen zwischen den Klassen

Chaotisches Strukturverhalten bei Strukturänderungen. Der hier verfolgte Ansatz geht von Anfang an davon aus, dass bei allen, wenn auch noch so kleinen Änderung von Eigenschaften und deren Zuordnung zu Phänomenen, der Optimierungsprozess, ohne wenn und aber, ganz vorn begonnen wird, d h. dass, ausgehend von der disjunktiven Normalform der Booleschen Ausgangsgleichung nach der neuen/geänderten Tabelle der Zweckorientierten Objektsichten, das Distributivgesetz der Booleschen Algebra wieder angewendet wird, um die jetzt optimale Vererbungsstruktur zu erhalten. Nur dann wird gewährleistet, dass bei den gerade vorliegenden, zu optimierenden Bedingungen auch die kleinste Änderung in der Gestaltung des neuen Systems, die u. U. enorme Auswirkungen gegenüber der seither gültigen Vererbungsstruktur haben kann, in der neu sich ergebenden, jetzt optimalen Struktur berücksichtigt werden kann. Ähnlich wie dies in der Chaostheorie zu beobachten ist, dass kleinste Änderungen der Eingangsbedingungen das chaotische System in eine ganz andere als die seither wahrgenommene Richtung drängen können, können kleinste Änderungen die seither optimale Struktur völlig verändern. Ausgenommen von jeglichen Topologieänderungen sind triviale Änderungen der Gestalt, dass zum Beispiel ein neues, hinzuzufügendes Merkmal genau einem bereits vollzogenen Ausklammerungsgang entspricht, dann ändert sich die vorliegende Struktur nicht, sondern die neue Eigenschaft wird der Klasse hinzugefügt, die bereits die anderen Eigenschaften mit dem gleichen Ausklammerungsgang enthält. Der Fall einer trivialen Änderung liegt auch dann vor, wenn eine zu entfernende Eigenschaft nicht die einzige oder letzte Eigenschaft eines Ausklammerungsgangs darstellt. Die beiden Fälle werden jedoch immer beim neu Anwenden dieser Vorgehensweise systemimmanent berücksichtigt. Die folgenden Beispiele sollen dies verdeutlichen. Gegeben seien aus dem Bereich der Farbdarstellungen die beobachteten Phänomene „*TÜRKIS*“, „*GELB*“, „*WEISS*“ und „*ROT*“ und deren Repräsentanz aus den Grundfarben (Eigenschaften) „*blau*“, „*grün*“ und „*rot*“ und der zusätzlichen Eigenschaft „*Ortsvektor*(*x*,*y*)“. Die Eigenschaft „*Ortsvektor(x, y)*“ ist eine Eigenschaft, die hier jedes Phänomen besitzen soll und dient z. B. der Positionierung eines Pixels auf dem Bildschirm:

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen, beobachtbare Phänomene) *TÜRKIS*	*GELB*	*WEISS*	*ROT*	Anzahl
	1	2	3	4	5	6	8
1	*Ortsvektor(x,y)*	*a*	+	+	+	+	4
2	*blau*	*b*	+		+		2
3	*grün*	*g*	+	+	+		3
4	*rot*	*r*		+	+	+	3

Tabelle 7 Zweckorientierte Objektsichten (Phänomene) „*TÜRKIS*“, „*GELB*“, „*WEISS*“ und „*ROT*“ mit ihren Eigenschaften „*Ortsvektor(x, y)*“, „*blau*“, „*grün*“ und „*rot*“.

Die Farbenlehre ist gut dazu geeignet, den Vererbungsmechanismus adäquat darzustellen. Die zu vererbenden Eigenschaften werden hier durch die physikalischen Größen unterschiedlicher Wellenlängen elektro-magnetischer Schwingungen repräsentiert. Die zum Beispiel durch das menschliche Auge wahrgenommenen Farbempfindungen, die Phänomene (Erscheinungsbilder, von den Sinnen wahrgenommen, die zur Erkenntnis gelangenden Bewusstseinsinhalte), repräsentieren nichts Anderes als die Reiz-Eigenschaften der Wellenlängen dieser elektro-magnetischen Schwingungen. Um beispielsweise das Phänomen der Farbe „*GELB*“ zu repräsentieren, bildet sich die Vereinigungsmenge der Reizungen der beiden Farben *r* (*rot*) und *g* (*grün*). Das Wissen, das Bewusstwerden einer Farbempfindung „*GELB*“ erbt von den gleichzeitig dargebotenen elektro-magnetischen Schwingungen *rot* und *grün* eine neue Qualität an Information, die als Phänomen „*GELB*“ wahrgenommen wird. Das Farbphänomen „*GELB*“ muss die Semantik als Bewussteinsinhalt der beiden physikalischen Eigenschaften der Farben *rot* und *grün* gewissermaßen „semantisch vereinigen“, damit es bewusst als „*GELB*“-es Phänomen wahrgenommen wird. Das Phänomen „*GELB*“ erbt diese Semantik von den beiden Eigenschaften *rot* und *grün* entsprechender Licht-/Objektquellen.

Das stufenweise Erben der Eigenschaften der Grundfarben „*grün*“, „*rot*“ und „*blau*“, um entsprechende Farbphänomene wahrnehmen zu können, lässt sich sehr gut in Vererbungsmodellen, wie in den Abbildungen 16 bis 19 zu sehen ist, demonstrieren. Physikalisch lässt sich das in Rezeptoren einfallende Lichtphänomen „*GELB*“ über eine Spektralanalyse in die beiden ursprünglichen Eigenschaften „*rot*“ und „*grün*“ zerlegen und damit als getrennte elektro-magnetische Schwingungen mit unterschiedlicher Wellenlänge feststellen.
Aus den Zweckorientierten Objektsichten der Tabelle 7 können spaltenweise, Spalte drei bis sechs, folgende Konjunktionen abgeleitet werden. Für jedes „+“-Zeichen wird die entsprechende Boolesche Variable auf „1“ gesetzt und in die jeweilige Konjunktion übernommen:

TÜRKIS	= „*Ortsvektor(x, y)*“ * „*blau*“ * „*grün*“	= *abg*	= 1.
GELB	= „*Ortsvektor(x, y)*“ * „*blau*“ * „*grün*“	= *agr*	= 1.
WEISS	= „*Ortsvektor(x, y)*“ * „*blau*“ * „*grün*“ * „*rot*“	= *abgr*	= 1.
ROT	= „*Ortsvektor(x, y)*“ * „*rot*“	= *ar*	= 1.

Dabei wird beispielsweise nach dem Phänomen (Erscheinungsbild) „*ROT*“ und der Eigenschaft „*rot*“ unterschieden, denn die Eigenschaft „*rot*“ (physikalische elektro-magnetische Welle einer bestimmten Wellenlänge) kann in verschiedenen Phänomenen (Erscheinungsbildern, Interpretation einer physikalischen elektro-magnetischen Welle durch ein menschliches Auge), wie zum Beispiel in den Phänomenen „*WEISS*“ oder „*GELB*“ enthalten sein, kann aber auch eigenständig als Phänomen „*ROT*“ wahrgenommen werden. Bildet man aus diesen Strukturdaten in Form der spaltenweisen Konjunktionen einen Booleschen Ausdruck in disjunktiver Normalform, erhält man Gleichung 25, auf die das Distributivgesetz der Booleschen Algebra anzuwenden ist:

Gleichung 25

$TÜRKIS \vee GELB \vee WEISS \vee ROT =$
$\boldsymbol{a}bg \vee \boldsymbol{a}gr \vee \boldsymbol{a}bgr \vee \boldsymbol{a}r = 1$ | „*a*“ wird vor die Klammer gezogen

Gleichung 26

$\boldsymbol{a}(bg \vee g\boldsymbol{r} \vee bg\boldsymbol{r} \vee \boldsymbol{r}) = 1$ | „*r*“ wird vor die Klammer gezogen

Gleichung 27

$a(bg \vee r(g \vee bg \vee 1)) = 1.$

Im Teilausdruck $(g \vee bg \vee 1)$ des Ausdrucks $r(g \vee bg \vee 1)$ bedeutet die „1“ lediglich, dass die Eigenschaft *r* auch allein ausgeprägt sein kann, im Gegensatz zu abstrakten Klassen (siehe auch [EnG]). Ausgeklammerte Eigenschaften bilden immer abstrakte Klassen, die im Fall von einer ausgeklammerten Booleschen Variablen „1“ zur konkreten Klasse ausgeprägt sein kann. Die Booleschen Variablen *g* und *bg* haben eine Platzhalterfunktion. Der Teilausdruck $(g \vee bg \vee 1)$ ist deshalb nicht durch eine „1“ zu ersetzen (siehe auch [EnG]). Eine Zuordnung der generalisierten/spezialisierten Eigenschaften, wie sie in Gleichung 27 zu sehen sind, zu Klassennamen sei wie folgt gegeben, wobei der Einfachheit halber hier die Bezeichnungen (*a = Ortvektor(x, y)*, *bg = “blau grün“*, *g = “grün“*, *r=“rot“*) der Eigenschaften als Klassennamen übernommen werden: $\boldsymbol{A} = a$; $\boldsymbol{BG} = bg$; $\boldsymbol{G} = g$; $\boldsymbol{R} = r$.

Setzt man diese Klassenbezeichnungen in die Struktur der Gleichung 27 ein, erhält man die Vererbungsstrukturgleichung zur graphischen Interpretation gemäß Gleichung 28:

Gleichung 28

$\boldsymbol{A}(\boldsymbol{BG} \vee \boldsymbol{R}(\boldsymbol{G} \vee \boldsymbol{BG} \vee 1)) = 1.$

Die graphische Interpretation der Gleichung 28 ergibt Darstellung Abbildung 16, wobei logisch additiv verbundene Klassen immer innerhalb einer Vererbungsebene zu setzen sind, während logisch multiplikativ verbundene Klassen immer durch verschiedene Ebenen repräsentiert werden (siehe auch [EnG]) :

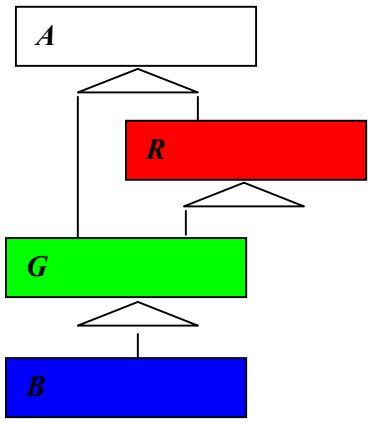


Abbildung 16 Vererbungsstruktur aus Tabelle 7 bzw. Gleichung 28 entwickelt, zur Darstellung der Phänomene „*TÜRKIS*“, „*GELB*“, „*WEISS*“ und „*ROT*“.

Ändert man in den Zweckorientierten Objektsichten der Tabelle 7, Spalte fünf, das Phänomen „*WEISS*“ in „*PURPUR*“ erhält man Tabelle 8. Faktisch wird dabei nur die Eigenschaft *g* (*grün*) im Phänomen „*WEISS*“ herausgenommen wird. Es wird nun das Phänomen (Erscheinungsbild) „*PURPUR*“ (Tabelle 8, Spalte fünf, Zeile drei) dargestellt:

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen, beobachtete Phänomene)				Anzahl
			TÜRKIS	*GELB*	***PURPUR***	*ROT*	
	1	2	3	4	**5**	6	8
1	*Ortsvektor(x,y) Pixelpositionierung*	*a*	+	+	+	+	4
2	*blau*	*b*	+		+		2
3	*grün*	*g*	+	+			2
4	*rot*	*r*		+	+	+	3

Tabelle 8 Zweckorientierte Objektsichten (Phänomene) „*TÜRKIS*“, „*GELB*“, „*PURPUR*“ und „*ROT*“ mit ihren Eigenschaften „*Ortsvektor(x, y)*“, „*blau*“, „*grün*“ und „*rot*“, wobei gegenüber **Tabelle 7**, Spalte fünf, das Phänomen „*WEISS*“ gegen das Phänomen „*PURPUR*“ ausgetauscht und die Eigenschaften angepasst wurden.

Aus den Zweckorientierten Objektsichten der Tabelle 8 können folgende Konjunktionen spaltenweise abgeleitet werden:

TÜRKIS = *abg* = 1.
GELB = *agr* = 1.
PURPUR = *abr* = 1.
ROT = *ar* = 1.

Bildet man aus diesen Strukturdaten in Form der Konjunktionen einen Booleschen Ausdruck in disjunktiver Normalform, erhält man den Ausdruck Gleichung 29, auf den ebenfalls das Distributivgesetz der Booleschen Algebra angewendet wird:

Gleichung 29

$TÜRKIS \vee GELB \vee PURPUR \vee ROT =$
$\boldsymbol{a}bg \vee \boldsymbol{a}gr \vee \boldsymbol{a}br \vee \boldsymbol{a}r = 1$ | „*a*“ wird vor die Klammer gezogen

Gleichung 30

$\boldsymbol{a}(bg \vee gr \vee br \vee r) = 1$ | „*r*“ wird vor die Klammer gezogen

Gleichung 31

$a(bg \vee r(g \vee b \vee 1)) = 1.$

Eine Zuordnung der generalisierten/spezialisierten Eigenschaften, wie sie in Gleichung 31 zu sehen sind, zu Klassennamen sei wie folgt gegeben: $\boldsymbol{A} = a$; $\boldsymbol{BG} = bg$; $\boldsymbol{G} = g$; $\boldsymbol{R} = r$.

Setzt man die Klassenbezeichnungen in die Struktur der Gleichung 31 ein, erhält man die Vererbungsstrukturgleichung zur graphischen Interpretation:

Gleichung 32

$\boldsymbol{A}(\boldsymbol{BG} \vee \boldsymbol{R}(\boldsymbol{G} \vee \boldsymbol{B} \vee 1)) = 1.$

Die graphische Interpretation der Gleichung 32 ergibt die Darstellung der Abbildung 17. Um die neue Realität im Modell abzubilden, hat sich das Struktursystem gegenüber der Abbildung 16 geändert, die beiden Klassen ***B*** und ***G*** wurden auf der Basis der Ebenen getauscht, die Anzahl der Strukturverbindungen hat sich von vier auf fünf erhöht:

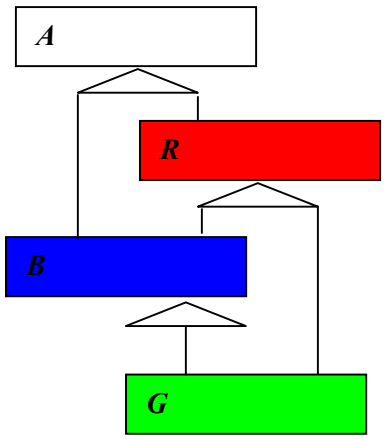


Abbildung 17 Vererbungsstruktur aus Tabelle 8 bzw. aus Gleichung 32 entwickelt, zur Darstellung der Phänomene „*TÜRKIS*“, „*GELB*“, „*PURPUR*“ und „*ROT*“.

Die Änderungen der aus Tabelle 7 sich ergebenden Vererbungsstruktur gegenüber der aus Tabelle 8 abgeleiteten Vererbungsstruktur sind jedoch erheblich. Außer dem Tausch der beiden Klassenebenen für die Klassen ***B*** und ***G*** mit der Eigenschaften *b* (*blau*) und *g* (*grün*) ergeben sich auch unterschiedliche Anzahlen von Vererbungsstrukturen:

In Abbildung 16 ergeben sich vier Strukturverbindungen, während sich in Abbildung 17 fünf Strukturverbindungen ergeben. Diese Veränderungen haben sich nur durch die Wegnahme einer einzigen Eigenschaft *g* (*grün*) in Spalte 5, Zeile 3, aus Tabelle 7 (kleine Ursache, große Wirkung) eingestellt. Die Komplexität der beiden Strukturen $K = n(n\text{-}1)/2 = 4*3/2 = 6$ ist relativ gering. Die dazugehörige Variabilität ist $VA = 2^{n(n\text{-}1)/2} = 2^{4*3/2} = 2^6 = 64$, aus dieser Potenzmenge rekrutieren sich die Lösungsstrukturen. Wie schnell sind Entwickler intuitiv geneigt, bei solch minimalen Änderungen (eine Eigenschaft!) dies schnell in der ersten Struktur in Abbildung 16 möglicherweise durch dem Einsatz von Overloading - Funktionen zu ändern, um den Umfang der notwendigen Programmänderungen zu minimieren, um den Preis einer unsauberen Vererbungsstruktur willen, die die neue Realität nicht mehr formal exakt abbildet. Man realisiert Maßnahmen, die „in der Nähe" des „Status quo" liegen, weil man sich möglicherweise nur bei kleinen Änderungen vorzustellen vermag, welche Auswirkungen im Verhalten des neuen Konstrukts zu erwarten sind.
Wenn die alte Vererbungsstruktur anstatt der vier Klassen beispielsweise 30 oder 40 Klassen umfassen würde, wie viele Vererbungsstrukturänderungen in Form des Austauschs von Vererbungsebenen, zusätzlichen Vererbungsebenen, wegfallenden Vererbungsebenen, Veränderungen in der Anzahl der Strukturverbindungen, möge dann die Veränderung einer einzigen Eigenschaft ergeben, um eine optimale integere neue, in sich logisch schlüssige Vererbungsstruktur zu erzeugen, in der jede Eigenschaft nur ein einziges Mal vorkommt? Wenn sich dagegen mehrere Eigenschaften einer seitherigen Struktur ändern und/oder neue Phänomene (weitere Zweckorientierte Objektsichten) mit neuen Eigenschaften hinzukommen oder zu entfernen sind und dabei auch seitherige, im Gesamtsystem verbleibende Eigenschaften den neu hinzu zufügenden, Zweckorientierten Objektsichten angehören oder aus ihnen entfernt werden müssen, welcher Entwickler vermag sich dann noch intuitiv vorstellen können, wie eine neue, jetzt eindeutige (eineindeutige) Vererbungsstruktur gestaltet sein muss, um jede Eigenschaft im neuen System einmalig zu erhalten? Über die Anzahl der Strukturzustände, über die sich ein Entwickler nur bei zwölf Klassen Gedanken machen muss, kann aus Tabelle 1 entnommen werden: Bei nur zwölf Klassen sind theoretisch 73.786.976.294.838.200.000 verschiedene Strukturzustände der Klassen untereinander möglich, von denen bei versierten Entwicklern sicherlich die meisten Strukturzustände ausscheiden. Aber immerhin kann noch eine erhebliche Anzahl von Strukturzuständen übrig bleiben, so dass die Chance, eine integere, eindeutige, widerspruchsfreie Vererbungsstruktur zu erhalten, gering ist, wenn man versucht, die seither benutzte Vererbungsstruktur zu ändern. Wie wird man sich helfen müssen, um die geänderte Vererbungsstruktur auch nur halbwegs sicher zu erhalten: Testen ohne Ende!

Ändert man in den Zweckorientierten Objektsichten der Tabelle 8 das Phänomen „*PURPUR*" in „*GRÜN*" erhält man Tabelle 9:

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen, Phänomene) *TÜRKIS*	*GELB*	***GRÜN***	*ROT*	Anzahl
	1	2	3	4	**5**	6	8
1	*Ortsvektor(x,y) Pixelpositionierung*	*a*	+	+	+	+	4
2	*blau*	*b*	+				2
3	*grün*	*g*	+	+	+		2
4	*rot*	*r*		+		+	3

Tabelle 9 Zweckorientierte Objektsichten (Phänomene) „*TÜRKIS*", „*GELB*", „*GRÜN*" und „*ROT*" mit ihren Eigenschaften „*Ortsvektor(x, y)*", „*blau*", „*grün*" und „*rot*", wobei gegenüber **Tabelle 8**, Spalte fünf, das Phänomen „*PURPUR*" gegen das Phänomen „*GRÜN*" ausgetauscht und die Eigenschaften angepasst wurden.

Aus den Zweckorientierten Objektsichten der Tabelle 9 können folgende Konjunktionen spaltenweise abgeleitet werden:

TÜRKIS = *abg* = 1.
GELB = *agr* = 1.
GRÜN = *ag* = 1.
ROT = *ar* = 1.

Bildet man aus diesen Strukturdaten der Tabelle 9 in Form der Konjunktionen einen Booleschen Ausdruck in disjunktiver Normalform, erhält man Gleichung 33, auf die das Distributivgesetz der Booleschen Algebra anzuwenden ist:

Gleichung 33

$TÜRKIS \vee GELB \vee GRÜN \vee ROT =$

$\boldsymbol{a}bg \vee \boldsymbol{a}gr \vee \boldsymbol{a}g \vee \boldsymbol{a}r = 1$ | „*a*“ wird vor die Klammer gezogen

Gleichung 34

$\boldsymbol{a}(bg \vee gr \vee g \vee r) = 1$ | „*g*“ wird vor die Klammer gezogen

Gleichung 35

$\boldsymbol{a}(g(b \vee r \vee 1) \vee r) = 1.$

Eine Zuordnung der generalisierten/spezialisierten Eigenschaften, wie sie in Gleichung 35 zu sehen sind, zu Klassennamen sei wie folgt gegeben: $\boldsymbol{A} = a$; $\boldsymbol{B} = b$; $\boldsymbol{G} = g$; $\boldsymbol{R} = r$.

Setzt man die Klassenbezeichnungen in die Struktur der Gleichung 35 ein, erhält man die Klassenvererbungsstrukturgleichung zur graphischen Interpretation in Gleichung 36:

Gleichung 36

$\boldsymbol{A}(\boldsymbol{G}(\boldsymbol{B} \vee \boldsymbol{R} \vee 1) \vee \boldsymbol{R}) = 1.$

Die graphische Interpretation der Gleichung 36 ergibt die Darstellung der Abbildung 18. Das Strukturensystem hat sich gegenüber Abbildung 17 um eine Ebene vermindert, eine Generalisierung/Spezialisierung ist entfallen:

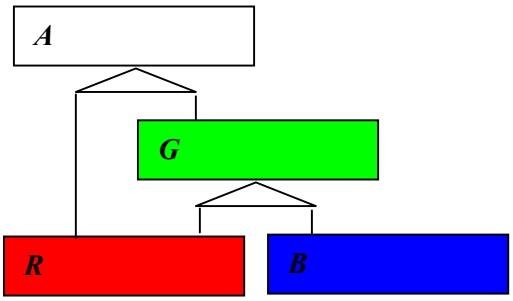


Abbildung 18 Vererbungsstruktur aus Tabelle 9 bzw. aus Gleichung 36 entwickelt, zur Darstellung der Phänomene „*TÜRKIS*“, „*GELB*“, „*GRÜN*“ und „*ROT*“.

Vergleicht man die Repräsentationsmöglichkeiten der Vererbungsstrukturmodelle in den Abbildungen 16 bis 18 untereinander, ist festzustellen, dass z. B. Abbildung 16 nicht geeignet ist, ein Phänomen „*BLAU*“ darzustellen, da die Repräsentationen in diesen Beispielen immer die Wurzelklasse ***A*** und die Eigenschaften aller Klassen miteinbeziehen müssen, die zwischen der untersten, darzustellenden Klasse und der Wurzelklasse liegen. Der Ausschluss der Darstellbarkeit aller nicht in einem Vererbungsstrukturmodell vorgesehenen Phänomene ist aus *vererbungstechnischer Sicherheit* ausdrücklich so gewollt. Die Vererbungsstrukturmodelle der Abbildungen 16 bis 18 sind deshalb so gewählt, weil sie bewusst die nur die gewollten Phänomene repräsentieren dürfen. Sie weisen nicht etwa deshalb einen Mangel auf, weil sie nur bestimmte Phänomendarstellungen zuließen, sondern es ist die Stärke dieser Modelle, dass sie nur die Ausprägungen dieser Phänomene zulassen, für die sie konzipiert wurden. Die Sicherheit liegt darin begründet, dass nur gewollte Vererbungsstrukturen entstehen und andere, nicht gewollte Vererbungsstrukturen, per Definition durch das Prinzip der Nicht - Vererbbarkeit ausgeschlossen werden. Dies bedeutet, dass nicht gewollte Vererbungsstrukturen erst gar nicht entstehen können, nicht einmal zufällig durch einen nicht vollzogenen, expliziten Ausschluss. Der Entwickler und der Benutzer kann sich mit Sicherheit darauf verlassen, dass solche nicht gewollten Vererbungsstrukturen schon per Definition nicht entstehen können, wenn die Ausformulierung der Tabelle der Zweckorientierten Objektsichten sorgfältig durchgeführt wird und die sich ergebende Struktur strukturgetreu in das Programm umgesetzt wird. Im Vererbungsstrukturmodell der Abbildung 17 ist im Gegensatz zum Vererbungsstrukturmodell der Abbildung 16 das Phänomen „*BLAU*“ darstellbar. Im Vererbungsstrukturmodell der Abbildung 18 wäre es ebenfalls nicht möglich, das Phänomen „*BLAU*“ zu repräsentieren, denn es würde immer die Klasse ***G*** mit der Eigenschaft *g* (*grün*) als übergeordnete Klasse mitvererbt werden. Es wäre jedoch möglich, im Vererbungsstrukturmodell der Abbildung 17 auch das Phänomen „*WEISS*“ zu repräsentieren, denn die Eigenschaften „*grün*“, „*rot*“ und „*blau*“ sind alle von der Wurzelklasse ***A*** aus ohne formale Beschränkung über den Teilterm $\boldsymbol{R}(\boldsymbol{G} \vee \boldsymbol{B} \vee 1)$ erreichbar. Wenn alle drei Klassen ***G***, ***R*** und ***B*** zusammen mit der Klasse ***A*** aktiviert werden würden, wäre das Phänomen „*WEISS*“ repräsentiert. Gewollt wäre dies aus der Anforderung der Tabelle 8 und der dort genannten Zweckorientierten Objektsichten nicht. In diesem Fall wäre die Verhinderung hier jedoch prozessual abzusichern, denn per Definition aus der Statik der Vererbungsstruktur der Abbildung 17 heraus wäre die Repräsentation des Phänomens „*WEISS*“ nicht zu verhindern. Um diese ungewollten Ausprägungen dennoch zu verhindern, kann die sogenannte *Negation* einer *Booleschen Variablen* benutzt werden, wie sie u. A. unten im Abschnitt „**Orthogonale Vererbung**“ angedeutet beschrieben ist (siehe dazu auch [EnG]).

Orthogonale Vererbung (nach UML: Diskrimonator - Bildung). Ein nicht zu unterschätzendes Thema bei einer intuitiven Vorgehensweise ist die Erkennung vom Diskriminator - Strukturen. Hier weicht man mangels der Erkennung solcher Strukturen häufig auf die Mehrfachvererbung aus, um die Daten-/Prozesslage abbilden zu können. Mit Hilfe dieser Vorgehensweise erkennt man durch die Art der sich ergebenden Ausklammerungsstrukturen orthogonale Vererbungsstrukturen (= Diskriminator - Strukturen). In einem solchen Fall wären innerhalb einer Spezialisierungsebene nicht nur eine Spezialisierung sondern mindestens deren zwei als Struktur in das Vererbungsmodell einzusetzen. **Tabelle 10** soll dazu benutzt werden, um einerseits orthogonale Vererbungen und andererseits sehr präzis Ausformulierungen mit negativen Booleschen Variablen zu demonstrieren, die eine sehr filigrane Modellierung im Bereich der Strukturvererbungen erlauben:

Lfd. Nr.	Phänomenbereich Eigenschaften (Daten / Vorschriften)	*BV*	Zweckorientierte Objektsichten (komponierende Abstraktionen, beobachtbare Phänomene) *BLAU*	*PURPUR*	*GRÜN*	*GELB*	Anzahl
	1	2	3	4	5	6	8
1	*Ortsvektor(x,y) Pixelpositionierung*	*a*	+	+	+	+	4
2	*blau*	*b*	+	+	-	-	2
3	*grün*	*g*	-	-	+	+	2
4	*rot*	*r*	-	+	-	+	2

Tabelle 10 Zweckorientierte Objektsichten (Phänomene) „*BLAU*“, „*PURPUR*“, „*GRÜN*“ und „*GELB*“ mit den entsprechend mit („+“) zugeordneten Eigenschaften „*Ortsvektor(x, y)*“, „*blau*“, „*grün*“ und „*rot*“, wobei nicht zu treffende Eigenschaften bei den Phänomenen mit („-“) markiert wurden, die ihrerseits als negative Boolesche Variablen interpretiert werden.

Zur präziseren Darstellung der Realität können auch, wie oben bereits erwähnt, *negierte Boolesche Variable* eingesetzt werden. Eine negierte Boolesche Variable drückt explizit aus, dass in einer bestimmten Zweckorientierten Objektsicht eine bestimmte Eigenschaft nicht gleichzeitig zusammen in verschiedenen Zweckorientierten Objektsichten vorkommen darf, dass also beispielsweise zwei Phänomene nicht gleichzeitig (parallel) mit der gleichen Eigenschaft instanziiert werden dürfen (siehe auch [EnG]). In den Tabellenelementen einer Tabelle der Zweckorientierten Objektsichten wird diese Forderung mit einem „-“ - Eintrag signalisiert, wobei die dort benutzte Boolesche Variable negativ belegt wird. Dies wird hier durch ein Unterstreichungszeichen unter der entsprechenden Booleschen Variablen zum Ausdruck gebracht: z. B. wenn eine Boolesche Variable a_i = „1“ dann ist die Boolesche Variable $\underline{a_i} = \neg a_i$ = „0“ und wenn eine Boolesche Variable a_i = „0“, dann ist eine Boolesche Variable $\underline{a_i} = \neg a_i$ = „1“. Negative Boolesche Variable mit „-“ - Eintrag treten also immer innerhalb einer Zeile (bezogen auf eine Eigenschaft) zusammen mit mindestens einer anderen Zweckorientierten Objektsicht (Phänomen) auf, die bei der gleichen Eigenschaft durch einen „+“-Eintrag gekennzeichnet ist. Solche Eigenschaften schließen sich, auf verschiedene Phänomene bezogen, gegenseitig in Form einer exklusiv – Oder Beziehung, aus. Aus den Zweckorientierten Objektsichten in Tabelle 10 können folgende Konjunktionen abgeleitet werden. Boolesche Variable einer Konjunktion sind alle untereinander Logisch – Und verknüpft: Logische Multiplikation „$\wedge$“, „*“ (siehe auch [EnG]):

BLAU = $a\ b\ \underline{g}\ \underline{r}$ = 1.
PURPUR = $a\ b\ \underline{g}\ r$ = 1.
GRÜN = $a\ \underline{b}\ g\ \underline{r}$ = 1.
GELB = $a\ \underline{b}\ g\ r$ = 1.

Bildet man aus diesen Strukturdaten in Form der Konjunktionen einen Booleschen Ausdruck in disjunktiver Normalform, erhält man Gleichung 37, auf die das Distributivgesetz der Booleschen Algebra anzuwenden ist:

Gleichung 37

$$BLAU \vee PURPUR \vee GRÜN \vee GELB =$$
$$\boldsymbol{a}\ b\ \underline{g}\ \underline{r} \ \vee\ \boldsymbol{a}\ b\ \underline{g}\ r \ \vee\ \boldsymbol{a}\ \underline{b}\ g\ \underline{r} \ \vee\ \boldsymbol{a}\ \underline{b}\ g\ r \ = 1.$$

Klammert man die Boolesche Variable *a* (*Ortsvektor*(*x, y*)), die viermal in Gleichung 37 enthalten ist, aus, erhält man Gleichung 38:

Gleichung 38

$\boldsymbol{a}\,(\boldsymbol{b}\,\underline{\boldsymbol{g}}\,\underline{r} \;\vee\; \boldsymbol{b}\,\underline{\boldsymbol{g}}\,r \;\vee\; \underline{\boldsymbol{b}}\,\boldsymbol{g}\,\underline{r} \;\vee\; \underline{\boldsymbol{b}}\,\boldsymbol{g}\,r) = 1.$

Klammert man die Booleschen Variablen $\boldsymbol{b}\,\underline{\boldsymbol{g}}$ und $\underline{\boldsymbol{b}}\,\boldsymbol{g}$ die je zweimal in Gleichung 38 enthalten sind, aus, ergibt sich Gleichung 39:

Gleichung 39

$a\,(b\,\underline{g}\,(\,\underline{r} \vee r) \vee \underline{b}\,g\,(\underline{r} \vee r)) = 1.$

Der sich ergebende Ausdruck $(\underline{r} \vee r)$ wird nicht durch $(\underline{r} \vee r) = 1$ (ausgeschlossenes Drittes, auch Tautologie) ersetzt und entfernt, wie dies in der Behandlung von Ausdrücken der Booleschen Algebra sonst sinnvoll wäre, da die beiden Booleschen Variablen nur Platzhalter für mögliche Zustände von Eigenschaften in der Booleschen Gleichung sind und als solche für das Verfahren erhalten bleiben sollen (siehe auch [EnG]). Eine Zuordnung der generalisierten / spezialisierten Eigenschaften zu Klassennamen, wie sie in Gleichung 39 zu sehen sind, sei wie folgt gegeben: $\boldsymbol{A} = a$; $\boldsymbol{B}\underline{\boldsymbol{G}} = \underline{bg}$; $\underline{\boldsymbol{B}}\boldsymbol{G} = \underline{bg}$; $\boldsymbol{R} = r$; $\underline{\boldsymbol{R}} = \underline{r}$.

Setzt man die Klassenbezeichnungen in die Struktur der Gleichung 39 ein, erhält man die Vererbungsstrukturgleichung zur graphischen Interpretation der Vererbungsstruktur der Klassen gemäß Gleichung 40:

Gleichung 40

$\boldsymbol{A}\,(\boldsymbol{B}\underline{\boldsymbol{G}}\,(\underline{\boldsymbol{R}} \vee \boldsymbol{R}) \vee \underline{\boldsymbol{B}}\boldsymbol{G}(\underline{\boldsymbol{R}} \vee \boldsymbol{R})) = 1.$

Beim Vorliegen einer Diskriminator - Strukturlage enden schon viele intuitiven Entwinklungen an dieser Stelle, in dem eine Mehrfachvererbung benutzt wird, weil diese Diskriminator - Strukturlage intuitiv von Entwicklern nicht ohne Weiteres erkannt wird. Eine Mehrfachvererbungslösung an dieser Stelle würde sich dann, wie nachfolgend gezeigt, ergeben, wobei die Klasse $\underline{\boldsymbol{R}}$ (*nicht rot*) nicht explizit instanziiert werden müsste, sondern einfach durch Nicht - Instanziierung der Klasse $\boldsymbol{R}$ (*rot*), z. B. beim Phänomen „*BLAU*“, zum Ausdruck gebracht werden könnte. Die graphische Interpretation der Gleichung 40 ergibt die Vererbungsstruktur in Abbildung 19:

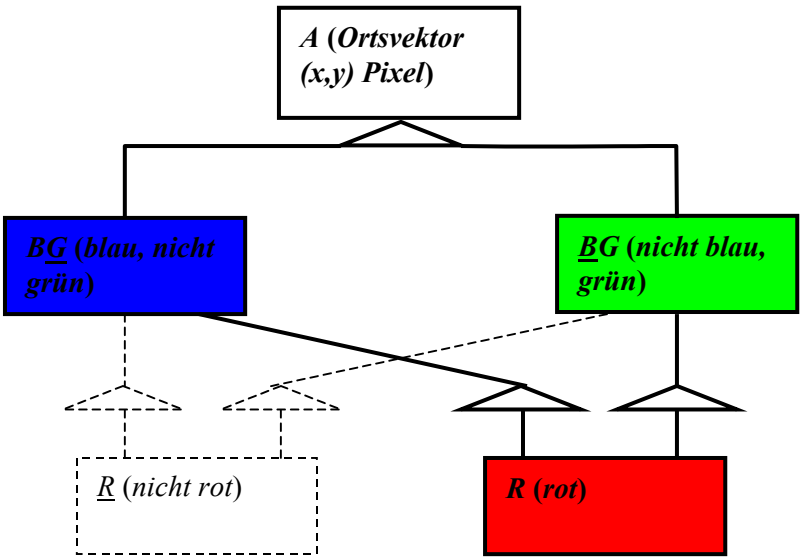


Abbildung 19 Strukturmodell einer Mehrfachvererbung aus Gleichung 40 entwickelt, ohne eine Diskriminator - Berücksichtigung nach UML, notgedrungen als Mehrfachvererbung

Arbeitet der intuitive Entwickler nicht mit den Ausdrücken der Booleschen Algebra, ist eine Diskriminator - Situation u. U. nicht ohne weiteres zu erkennen. Mehrfachvererbungen sollten (auch in der hier gezeigten Vorgehensweise) wegen ihrer höheren Komplexität möglichst vermieden werden, in dem ein strenger, hierarchischer Baum oder Teilbaum zur Repräsentation einer *generalisierenden Abstraktion* vorzuziehen ist. Mehrfachvererbungen dürfen aber in der hier vorgeschlagenen Vorgehensweise nicht ausgeschlossen werden, wenn eine minimale eindeutige Systemauslegung sie erfordert. Klammert man den Booleschen Term ($\underline{r} \vee r$) in Gleichung 39 weiter aus, erhält man den Ausdruck in Gleichung 41:

Gleichung 41

$$a\,((\underline{r} \vee r)\,(b\,\underline{g} * 1 \vee \underline{b}\,g * 1)) = a\,((\underline{r} \vee r)\,(b\,\underline{g} \vee \underline{b}\,g)) = 1.$$

Die graphische Interpretation dieses Ausdrucks ergibt einen streng hierarchischen Baum als eine orthogonale Spezialisierung, denn Klasse *a* (*Ortsvektor*(*x*, *y*)) ist an dieser Stelle zweidimensional ausgeprägt. Diese Zweidimensionalität kommt auch durch die beiden Klammernterme ($\underline{r} \vee r$) ($b\,\underline{g} \vee \underline{b}\,g$) als Faktoren zum Ausdruck, die in der Art eines lokalen, kartesischen Produktes „×“ zu sehen sind, die ihrerseits zusammen mit der Booleschen Variablen *a* (*Ortsvektor*(*x*, *y*)) multiplikativ auftreten: $a\,[(\underline{r} \vee r) \times (b\,\underline{g} \vee \underline{b}\,g)]$. Jede erlaubte Ausprägung des einen Faktors ($\underline{r} \vee r$) kann mit jeder erlaubten Ausprägung des andern Faktors ($b\,\underline{g} \vee \underline{b}\,g$) kombiniert vorkommen. Im Sinne von UML liegt eine Diskriminator - Ausprägung vor. Die zweifache „Klammer auf“ in einer Klammernebene begründet zwei Spezialisierungen in dieser Ebene. Die Klasse, die die Eigenschaft $\underline{r}$ (*nicht rot*) beinhaltet, muss nicht notwendigerweise instanziiert werden, in dem beim Fehlen der Eigenschaft *r* (*rot*) in einem darzustellenden Phänomen, z. B. „*BLAU*“ oder „*GRÜN*“, die Klasse mit der Eigenschaft *r* (*rot*) einfach nicht instanziiert wird. Benutzt man in Gleichung 41 die Klassenbezeichnungen $\boldsymbol{A} = a$, $\boldsymbol{B\underline{G}} = b\underline{g}$, $\boldsymbol{\underline{B}G} = \underline{b}g$, $\boldsymbol{R} = r$ *und* $\boldsymbol{\underline{R}} = \underline{r}$, erhält man Gleichung 42, die graphisch interpretiert wird:

Gleichung 42

$A\ ((\underline{R}\ \vee\ R)\ (B\ \underline{G}\ \vee\ \underline{B}\ G)) = 1.$

Abbildung 20 zeigt die graphische Interpretation der in Gleichung 42 als Vererbungsstruktur der Klassen:

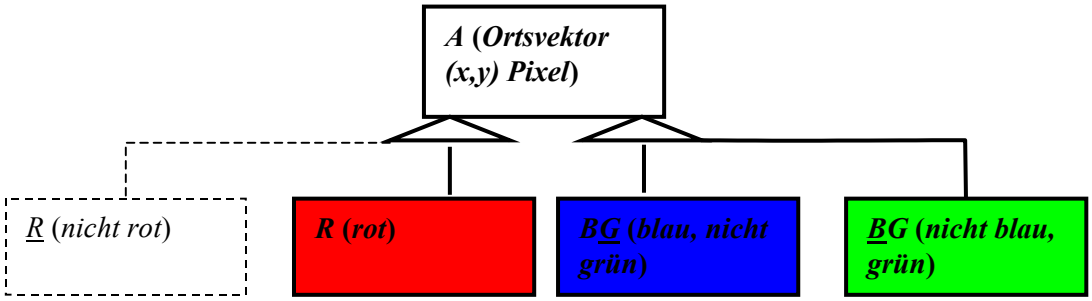


Abbildung 20 Vererbungsstrukturmodell einer orthogonalen Vererbung, entwickelt aus Tabelle 10 bzw. aus Gleichung 42 (Diskrimator - Bildung nach UML)

Nebenbei bemerkt treten die beiden Faktoren in der Vererbungsstruktur aus Abbildung 20, Eigenschaft *bg* bzw. Klasse ***BG*** und Eigenschaft *bg* bzw. Klasse ***BG***, zueinander immer in einer exklusiv Oder – Beziehung auf. Dies bedeutet, dass beide Klassen nie gleichzeitig zu instanziieren sind, wie dies auch in den unten gezeigten Abbildungen 21 bis 24 zu erkennen ist. Entweder ist die Klasse mit der Eigenschaft *b* (*blau*) und **nicht** gleichzeitig die Klasse mit der Eigenschaft *g* (*grün*) instanziiert, oder die Klasse mit der Eigenschaft *g* (*grün*) ist **nicht** gleichzeitig mit der Klasse der Eigenschaft *b* (*blau*) instanziiert: $(b\ \underline{g} \vee\ \underline{b}\ g) = 1$ bzw. $(\boldsymbol{B\underline{G}} \vee \boldsymbol{\underline{B}G}) = 1$, exklusiv - Oder Bedingung. Die Klasse ***R*** mit der Eigenschaft *r* (*rot*) kann vorhanden sein oder auch nicht, je nach dem Erfordernis des aktuell zu repräsentierenden Phänomens. Die Instanziierung der Klasse ***R*** mit der Eigenschaft *r* (*rot*) wird durch die jeweilige Faktenlage bestimmt, die in den geforderten Phänomenen (Zweckorientierten Objektsichten) in Tabelle 10 festgelegt ist: Gemäß Aufgabenstellung in Tabelle 10 sind nur die Phänomene „*BLAU*“, „*PURPUR*“, „*GRÜN*“ und „*GELB*“ darzustellen. Es gibt kein Phänomen, das *gleichzeitig* die beiden Eigenschaften „*blau*“ und „*grün*“ beinhaltet. Aus diesem Grund schließen sich die beiden Klassen mit den Eigenschaften „*blau*“ und „*grün*“ innerhalb einer Zweckorientierten Objektsicht (innerhalb eines betrachteten Phänomens) gegenseitig aus: $(b\ \underline{g} \vee\ \underline{b}\ g) = 1$ entspricht $(\boldsymbol{B\underline{G}} \vee \boldsymbol{\underline{B}G}) = 1$, denn („*blau*“ und „*nicht grün*“ oder „*nicht blau*“ und „*grün*“) = 1. Die Phänomene „*BLAU*“ oder „*GRÜN*“ enthalten die Eigenschaften entweder „*blau*“ oder „*grün*“ als essentielle Eigenschaften. Gleichzeitig (parallel dazu) können die Eigenschaften „*rot*“ oder „*nicht rot*“ in Abhängigkeit der jeweils darzustellenden Phänomene mit den Klassen ***R*** oder ***R*** kombiniert werden, wie es die zweite Spezialisierung im Modell der Abbildung 20 erlaubt.
Nach der Vererbungsstruktur Abbildung 20 sind demnach folgende Ausprägungen dieser Vererbungsstruktur als Phänomene nach Tabelle 10 zugelassen. Zur Darstellung des Phänomens „*BLAU*“ sind die Klassen ***A*** und ***BG*** (= *blau* und *nicht grün*), wie in Abbildung 21 gezeigt, zugelassen:

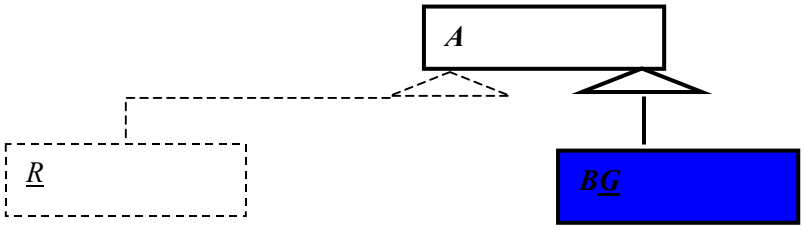


Abbildung 21 Instanziierung des Phänomens „*BLAU*“ aus Tabelle 10 bzw. Abbildung 20 mit den Eigenschaften „*a Ortsvektor (x, y)*“ und „*blau*“.

Zur Darstellung des Phänomens „*PURPUR*“ aus Tabelle 10 ist die Strukturausprägung der Klassen ***R*** und ***BG*** zugelassen bzw. erforderlich, wie in Abbildung 22 gezeigt:

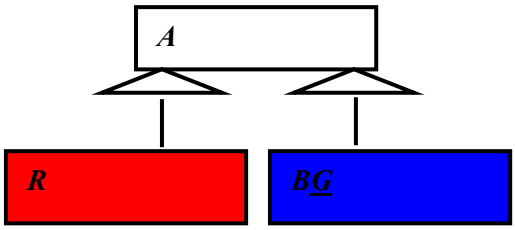


Abbildung 22 Instanziierung des Phänomens „*PURPUR*“ aus Tabelle 10 bzw. Abbildung 20 mit den Eigenschaften „*a Ortsvektor (x, y)*“, „*rot*“ und „*blau*“.

Zur Darstellung des Phänomens „*GRÜN*“ aus Tabelle 10 ist ist die Strukturausprägung der Klassen (***R*** und) ***BG*** zugelassen bzw. erforderlich, wie dies Abbildung 23 zeigt:

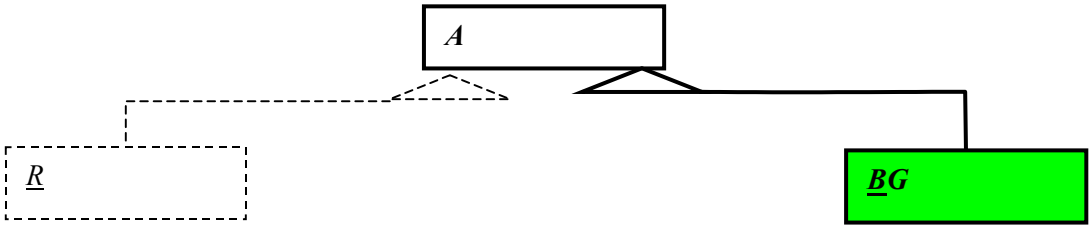


Abbildung 23 Instanziierung des Phänomens „*GRÜN*“ aus Tabelle 10 bzw. Abbildung 20 mit den Eigenschaften „*a Ortsvektor (x, y)*“ und „*grün*“.

Zur Darstellung des Phänomens „*GELB*“ aus Tabelle 10 ist die Strukturausprägung der Klassen ***R*** und ***BG*** zugelassen bzw. erforderlich, wie in Abbildung 24 gezeigt:

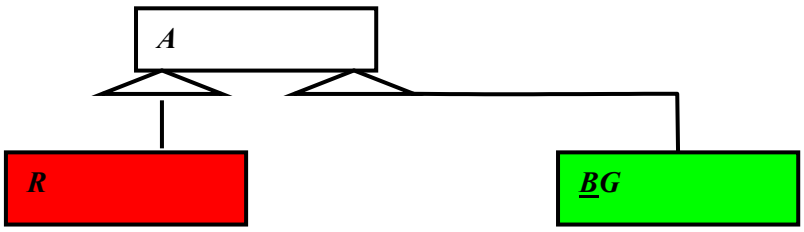


Abbildung 24 Instanziierung des Phänomens „*GELB*“ aus Tabelle 10 bzw. Abbildung 20 mit den Eigenschaften „*a Ortsvektor (x, y)*“, „*rot*“ und „*grün*“.

Alle anderen Kombinationen der Vererbungsstruktur aus Abbildung 20 sind durch die gegenseitigen Ausschlüsse per Definition in Tabelle 10 nicht zugelassen. Wäre beispielsweise ein Phänomen „*WEISS*“ in Tabelle 10 zugelassen, dann müssten die drei Klassen ***R***, ***B*** und ***G*** gleichzeitig instanziiert werden, was aber nach den Erfordernissen der Zweckorientierten Objektsichten (der beabsichtigten Phänomene) in Tabelle 10 nicht gewollt ist und nach Gleichung 42 $[\boldsymbol{A}\ ((\underline{\boldsymbol{R}} \vee \boldsymbol{R})\ (\boldsymbol{B}\ \underline{\boldsymbol{G}} \vee \underline{\boldsymbol{B}}\ \boldsymbol{G})) = 1]$ auch nicht zugelassen ist.

Strenge Kontrolle der Programmvariabilität. In sicherheitssensitiven Programmsystemen hat die folgende Forderung absoluten Vorrang: Bei jeder Änderung eines Programmsystems, und sei es nur das Entfernen oder Hinzufügen einer einzigen Eigenschaft in einer Zweckorientierten Objektsicht (Phänomensicht) wie oben beschrieben, kann es nicht die Frage sein, ob das Programmsystem so flexibel programmiert ist, dass es diese Veränderung möglicherweise auch noch ohne Programmänderung „verkraftet“. Ein Programmsystem, das höchsten Sicherheitsanforderungen genügen soll, muss so gestaltet sein, dass die kleinste Änderung in seiner statik-bildenden Vererbungsstruktur eine neue Vererbungsstruktur bedingt, es sei denn bei der veränderten Zweckorientierten Objektsicht würde es sich um eine Objektsicht (Phänomen) handeln, deren Menge der Eigenschaften es bereits gäbe (unechte Teilmenge). Selbst eine veränderte Zweckorientierte Objektsicht mit einer echten Teilmenge von Eigenschaften einer bereits vorhandenen Zweckorientierten Objektsicht würde eine veränderte Vererbungsstruktur ergeben.
Die jetzt nach einer Änderung neu geltenden Gegebenheiten und Abhängigkeiten müssen explizit in einer jetzt gültigen Vererbungsstruktur streng und ausnahmslos eineindeutig abgebildet sein. Es müsste gewährleistet sein, dass jede im Gesamtsystem verwendete Eigenschaft einmalig bleibt und ist und genau an der Stelle im Vererbungsstrukturnetz positioniert ist, an diese Eigenschaft den restlichen Systemelementen (Klassen) bedarfsgesteuert „vererbt“ werden kann. Jedes andere großzügigere Verhalten und Tolerieren von Änderungen als die einer „maßgeschneiderten“ individuell einmaligen Vererbungsstruktur ohne den geringsten Freiheitsgrad ist im Bereich von sicherheitsrelevanten Software - Systemen nicht hinnehmbar. Wie an der Veränderung einer einzigen Eigenschaft zwischen Tabelle 7 und Tabelle 8 oder zwischen Tabelle 8 und Tabelle 9 bei nur vier Klassen gezeigt werden konnte, ändert sich die Topologie von Vererbungsstrukturen vollständig, wenn sich Eigenschaften ändern.
Wenn ein Programmsystem eine solche Veränderung von Phänomeneigenschaften ohne Änderung der Vererbungsstruktur „schluckt“, muss man diesem Programmsystem seine seitherige „*angebliche*“ Sicherheit und Verlässlichkeit kategorisch absprechen. Es hätte Freiheitsgrade von Zuständen im Sinne einer „Vererbung von Eigenschaften“ ermöglicht, die eigentlich aus der Semantik der „Vererbung“ heraus ein Widerspruch in sich wären und die mit Sicherheit nicht unter Kontrolle einer in sich schlüssigen und integeren (widerspruchsfreien) Systemverhaltens - Logik gestanden hätten.

In sicherheitssensitiven Bereichen sind nicht die „Alleskönner“ – Programmsysteme gefragt, sondern die eng begrenzten „Spezialisten“ – Programmsysteme, die Freiheitsgrade in den Zustandsräumen ihrer Vererbungsstruktur und damit in den Phänomenausprägungen bezüglich der Einmaligkeit der Systemkomponenten und der Positionierung dieser Komponenten im Vererbungsstrukturnetz systemisch – kategorisch verhindern, respektive ausschließen.

Systemimmanente Redundanzkontrolle. An den gezeigten Beispielen von Tabelle 7 bis Tabelle 10 lässt sich deutlich zeigen, dass man mit dem Instrument der Nutzung des Distributivgesetzes der Booleschen Algebra zur Gestaltung der *generalisierenden Abstraktion* äußerst präzise und filigran modellieren kann, was der Software – Sicherheit und der Software – Verlässlichkeit per Definition sehr entgegen kommt. Diese beiden Kriterien werden durch eine solche Vorgehensweise geradezu unausweichlich, quasi „*ohne es verhindern zu können*“, durch den zu Grunde liegenden Algorithmus der Booleschen Algebra in Verbindung mit der ordnungsstrukturstiftenden Fähigkeit des Distributivgesetzes der Booleschen Algebra gestützt.
Essentiell wichtig dabei ist, dass sich Entwickler bei der Definition der Tabelle der Zweckorientierten Objektsichten **lokal** im Detail auf das Problem der darzustellenden Phänomene und deren Zuordnung zu den Eigenschaften fokussieren können, ohne sich **global** gesamtsystembezogen simultan um die Einmaligkeit, d. h. die Verhinderung von daten- und prozessseitiger Redundanzen, bemühen zu müssen, denn diese Eigenschaft wird durch die Ordnungswirkung des Distributionsgesetzes der Booleschen Algebra, sozusagen per Definition, formal gewährleistet.
Deshalb wird durch die konsequente Anwendung dieser Vorgehensweise die Harmonie der Software – Eigenschaften Sicherheit und Verlässlichkeit in Verbindung mit der Software – Vererbungsstrukturkorrektheit durch Eindeutigkeit bezüglich einer *generalisierenden* und *komponierenden Abstraktion* hervorgehoben. Die *komponierende Abstraktion* wird immer durch die Phänomenkompositionen (Zweckorientierten Objektsichten) repräsentiert. Um zur *generalisierenden Abstraktion* zu gelangen werden die die *komponierenden Abstraktion* darstellenden Zweckorientierten Objektsichten als *Konjunktionen* gedeutet. Sie werden ihrerseits in disjunktiver Normalform einer Booleschen Gleichung arrangiert. Diese ist die Ausgangsbasis für die Anwendung des Distributivgesetzes der Booleschen Algebra, das seinerseits die Vererbungsstrukturen eindeutig erzeugt. Sowohl die *komponierende* als auch die *generalisierende Abstraktion* haben damit die gleiche Abstraktionsbasis: die *Komposition* der *Phänomene*.
Die Harmonieeigenschaften hinsichtlich der Software – Sicherheit und Verlässlichkeit der beiden Abstraktionsausprägungen *Komposition* und *Generalisierung* werden in Form einer minimalen Auslegung der Systemvererbungsstruktur- und Kompositionskomponenten bei hinreichender Funktionalität erreicht. Bedingt durch eine zum Teil immens hohe Variabilität (Potenzmengeneigenschaft) solcher Struktursysteme, die bei einer großen Anzahl von Ausklammerungsvarianten aus der generell Kartesischen-Produkt-Eigenschaft variierender Mengen erreicht werden kann, empfiehlt sich hier über eine Simulation experimentell eine suboptimale Generalisierungslösung anzustreben. Dies ist immer dann der Fall, wenn sich bei gleicher Anzahl verschiedener Boolescher Variablen in der gleichen Klammernebene mehrere Ausklammerungsmöglichkeiten für Lösungen anbieten würden.
Die Anzahl der entstehenden Klassen und die Zuweisung der Eigenschaften zu den Klassen bleibt jedoch in allen Ausklammerungsvarianten immer gleich. Diese Eigenschaft ist auf die Anwendung des Distributionsgesetzes der Booleschen Algebra zurückzuführen, wie es in dieser Vorgehensweise vorgeschlagen wird. Denn mehrfach vorkommende Kompositionen aus Booleschen Variablen, Teiltermen oder Termen behalten ihre konjunktive Konsistenz über den gesamten Ausklammerungsprozess hinweg, so wie sie Eingangs aus den Spalten der Zweckorientierten Objektsichten (Phänomenen) als Konjunktionen in die für den Prozess benutzte Ausgangsgleichung in disjunktiver Normalform überführt worden sind.

Vorteile der Vorgehensweise. Es konnte gezeigt werden, dass durch die beiden Konzepte der *generalisierenden* und *komponierenden Abstraktion* und mit den Mitteln des Distributivgesetzes der Booleschen Algebra in Verbindung mit den Zweckorientierten Objektsichten, sich sowohl die Strukturen der Komposition in Form von linearen Schnittstellen als auch die orthogonal dazu strukturierten Generalisierungen / Spezialisierungen einfach abzuleiten sind. Die linearen Schnittstellen lassen sich sehr leicht durch benutzende Anwendungsprogrammierer und damit auch für den Endbenutzer (Mensch, Maschine) bedienen. Die dazu orthogonal angelegten Generalisierungsstrukturen sichern eine redundanzfreie Nutzung des Gesamtsystems.
Bei einer Systemerweiterung entstehen durch die Anwendung der Vorgehensweise keinerlei Probleme in Form von „*Verschlechtbesserungen*“, da durch die Aufnahme von weiteren Systemeigenschaften im Phänomenbereich und/oder in den Zweckorientierten Objektsichten durch neuerliche Anwendung des Distributivgesetzes der Booleschen Algebra neue, jetzt optimale Strukturen entstehen. Damit ist eine redundanzfreie Systemhaltung gegeben, die notwendig ist, um sichere und verlässliche Systeme auch bei einer Fortentwicklung betreiben zu können. Im oben genannten Beispiel sind beispielsweise die Generalisierungs-/Spezialisierungsstrukturen aus Tabelle 2 auch in den Generalisierungs-/ Spezialisierungsstrukturen aus der um zwei Spalten erweiterten Tabelle 6 wieder zu finden, was für die Qualität und Ebenmäßigkeit bei der Fortentwicklung spricht.

Zukunftsentwicklung. Es ist vorgesehen, Generalisierungs-/Spezialisierungsstrukturen aus SQL – Tabellen zu generieren, die die oben genannten Strukturen der Zweckorientierten Objektsichten beinhalten. Zur Zeit ist ein Strukturenerzeugendensystem im Test, das es ermöglicht, Generalisierungen/Spezialisierungen automatisch zu erzeugen. Damit ist es möglich, den Entwicklungsprozess erheblich zeitlich zu beschleunigen und gleichzeitig einen sehr hohen Standard in Sicherheit und Verlässlichkeit in Software – Systemen zu gewährleisten, die in dieser Weise entwickelt wurden. Die systemisch entwickelten Strukturen der linearen und der dazu orthogonalen Generalisierungsstrukturen „vererben“ ihre Konsistenz, ihren logisch - mathematisch ableitbaren, strukturellen Aufbau und damit auch ihre logisch - mathematisch Stabilität auf die sich generierende Funktionalität. Dies wird erreicht durch den klassischen Einsatz des Distributivgesetzes der Booleschen Algebra. Die so entwickelten Systeme nutzen die den Tabellen der Zweckorientierten Sichten (Phänomene) innewohnenden, logisch – mathematisch syntaktischen Strukturen zu semantisch wirkenden Strukturinformationen der *generalisierenden* und *komponierenden Abstraktion*. Sie lassen somit dem individuell intuitiv entwickelnden Gestalter nicht mehr die individuellen Freiheitsgrade wie bei einem Schachspiel, bei dem zwar zu Beginn alle Freiheitsgrade vorherrschen, bei dem dann aber die letzten Züge notgedrungen unter dem Zwang der noch verbliebenen wenigen Züge sehr stark stringent sind. Systeme, in der vorgeschlagenen Vorgehensweise entwickelt, dürften intuitiv entwickelten Systemen gegenüber leichter und wirkungsvoller nachvollziehbar sein. Die Ordnungskraft des Distributivgesetzes der Booleschen Algebra bezüglich der erzeugten Vererbungsstrukturen dürfte sich äußerst günstig auf die Programmsystemsicherheit, -verlässlichkeit und -stabilität auswirken.

12.06.2006

Gisbert Englmeier

G-Englmeier@t-online.de

Literaturverzeichnis:

[DrH]: von Drachenfels, H., Komponentenorientierte Programmierung im Kleinen, in: Informatik Spektrum, Band 28, Heft 2, April 2005.
[DUD]: Duden, Das Fremdwörterbuch, Augsburg, 1999.
[EnG]: Englmeier, G., Objektstrukturen – praxisbezogenes Konzept der Attribut-/Methodenvererbung, VDE-Verlag, Berlin – Offenbach, 1997.
[FuH]: Fuchs, Dr. H., Systemtheorie und Organisation, Wiesbaden, 1973.
[JuH]: Jungclaussen, H., Kausale Informatik, Deutscher Universitäts-Verlag, Wiesbaden, 2001.
[KiW]: Kirsch, W., Einführung in die Theorie der Entscheidungsprozesse, Wiesbaden, 1977.
[OeB]: Oestereich, B., Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language -, München, 1999.