Streckenzeichner als Beispiel für Polymorphienutzung

Diese Übung dient dazu, Ihnen die Möglichkeit zu geben, ein erstes objektorientiertes Programm zu sehen und zu verändern. Es wurde hier eine Aufgabenstellung gewählt, die besonders geeignet ist, den Vorteil der objektorientierten Programmierung plausibel zu machen. Es geht um die Definition und Darstellung graphischer Objekte. Wir beschränken uns auf Zeichnungen, die ausschließlich aus geraden Stecken bestehen. Bild 69 zeigt den Systemaufbau, der uns vor Augen stehen sollte, wenn wir mit der Konstruktion des Programmes beginnen.

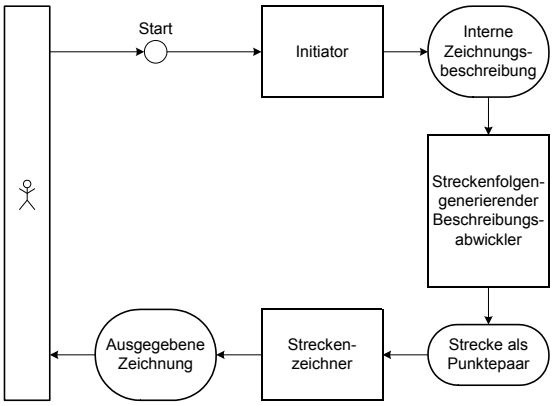

Bild 69 Systemaufbau im betrachteten Beispiel

Zuerst wird der dafür vorgesehene Behälter mit einer Zeichnungsbeschreibung belegt. Danach beginnt der Generator im Wechselspiel mit dem Streckenzeichner die Zeichnung auszugeben (siehe Bild 70). Der Streckenzeichner kennt ein karthesisches Koordinatensystem, in das er die einzelnen Strecken einträgt. Vom Generator erhält er jeweils die Information über ein Punktepaar, zwischen dem als nächstes eine gerade Strecke zu zeichnen ist. Dieses Punktepaar muss der Generator in Koordinaten angeben, die zum Koordinatensystem des Zeichners gehören.

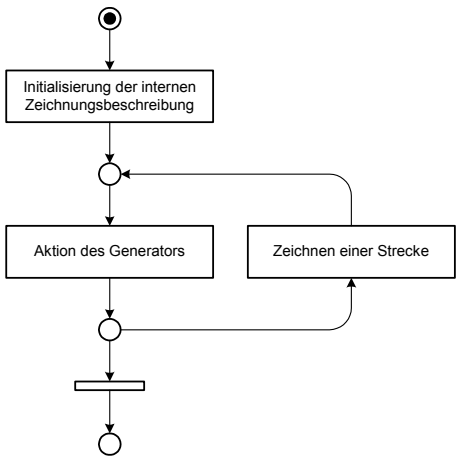

Bild 70 Ablauf der Aktionen in Bild 69

Domänenwissen

Jedesmal wenn wir mit einer neuen Aufgabe befasst werden, ist es zweckmäßig, die nun anzustellenden Überlegungen streng in zwei Themenbereiche zu trennen. Zuerst muss der Themenbereich behandelt werden, den wir als Anwendungsbereich (engl. Application Domain) bezeichnen. In diesem Themenbereich geht es überhaupt nicht um den Sachverhalt, dass irgendein programmiertes System realisiert werden soll. Es geht nur um das Verständnis der Themen, die den Aufgabenbereich kennzeichnen, in welchem später das programmierte System eingesetzt werden soll. Wenn wir also beispielsweise ein programmiertes System realisieren sollen, welches die Vorgänge in einem Krankenhaus unterstützen soll, ist der Anwendungsbereich bestimmt durch alle Vorgänge, die in einem Krankenhaus vorkommen können.

Die Anwendungsbereiche unserer bisherigen Übungsaufgaben waren immer sehr einfach. Zum einen war es der Bereich der numerischen Funktionen einer reellen Variablen x. In diesem Bereich musste man also den Begriff der Arithmetik verstehen, man musste die Vorstellung einer kontinuierlichen Kurve im zweidimensionalen Koordinatensystem haben usw. Ein anderer Anwendungsbereich war das Spiel „Türme von Hanoi“, wo es um Scheiben, Positionen und Bewegungsregeln ging.

Nun geht es um einen neuen Anwendungsbereich, der durch die Begriffe Punkt, Strecke, Koordinatensystem und Koordinatensystemtransformation gekennzeichnet ist. Wir betrachten diesen Anwendungsbereich zuerst einmal völlig unabhängig von der Frage, was denn später programmiert werden soll.

In Bild 71 ist gezeigt, dass die Lage zweier Koordinatensysteme in der Ebene zueinander durch 6 Parameter bestimmt wird, wenn zwischen ihnen die Beziehung „affine Abbildung“ besteht. Das eine Koordinatensystem KS_{Bild} sieht so aus, wie wir uns ein karthesisches Koordinatensystem normalerweise vorstellen: Die beiden Koordinatenachsen stehen senkrecht aufeinander und die Einheitslängen auf der x-und der y-Achse sind gleich.

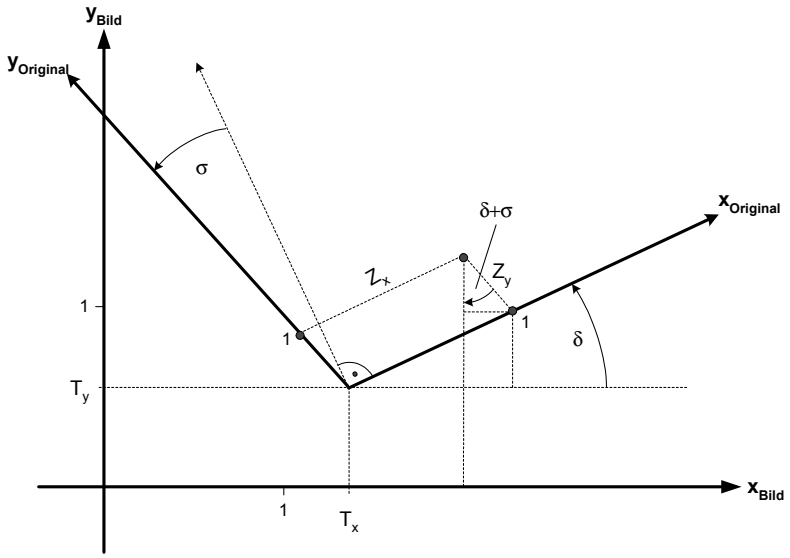

Bild 71 Koordinatensystemtransformation

Wir stellen uns nun vor, das Koordinatensystem $KS_{Original}$ wäre anfangs völlig deckungsgleich mit dem Koordinatensystem KS_{Bild} gewesen, d.h. die Ursprungspunkte der beiden Koordinatensysteme lagen aufeinander, die Achsen lagen aufeinander und die Einheitslängen waren identisch. Anschließend wurde das Koordinatensystem $KS_{Original}$ wie folgt manipuliert: Es wurde aus seiner bisherigen Lage wegbewegt, so dass sich sein Ursprung nun in KS_{Bild} am Punkt (T_x, T_y) befindet. Der Buchstabe T ist die Abkürzung für Translation. Anschließend wurde $KS_{Original}$ um den Winkel δ gedreht unter Beibehaltung seiner neuen Ursprungslage. Dann wurde die Achse $y_{Original}$ um den Winkel σ weitergedreht, ohne dass $x_{Original}$ mitgedreht wurde. Die beiden Achsen $x_{Original}$ und $y_{Original}$ stehen nun nicht mehr senkrecht aufeinander. Die Bezeichnung der beiden Winkel δ und σ wurde so gewählt, dass man mit δ das Wort Drehung und mit σ das Wort Scherung verbinden kann. Zum Schluss wurden die Längenmaßstäbe auf den beiden Achsen $x_{Original}$ und $y_{Original}$ verändert, so dass nun die Einheitslängen in KS_{Bild} und $KS_{Original}$ nicht mehr identisch sind. Die neue Einheitslänge auf der Achse $x_{Original}$ entstand durch Multiplikation mit dem Zoom-Faktor Z_x aus der bisherigen Einheitslänge, und die neue Einheitslänge auf der Achse $y_{Original}$ entstand durch Multiplikation mit dem Zoom-Faktor Z_y aus der bisherigen Einheitslänge.

Wenn man die 6 Transformationsparameter (Z_x, Z_y, δ, σ, T_x, T_y) kennt, kann man aus den Koordinaten eines Punktes in $KS_{Original}$ die Koordinaten des gleichen Punktes in KS_{Bild} berechnen. Diese Berechnung geschieht durch Multiplikation eines Vektors mit einer Matrix, wie dies in Bild 72 dargestellt ist. Damit man die Koordinatentransformation mit einer einzigen Multiplikation erledigen kann, mussten die ursprünglich zweidimensionalen Koordinatenvektoren durch Ergänzung mit der Konstanten 1 zu dreidimensionalen Vektoren gemacht werden, und die ursprüngliche 2 x 3 Transformationsmatrix musste durch eine zusätzliche Konstantenzeile zu einer quadratischen 3 x 3 Matrix ergänzt werden.

$$\begin{pmatrix} 1 \\ x_{Bild} \\ y_{Bild} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ T_x & Z_x * \cos(\delta) & -Z_y * \sin(\delta+\sigma) \\ T_y & Z_x * \sin(\delta) & Z_y * \cos(\delta+\sigma) \end{pmatrix} * \begin{pmatrix} 1 \\ x_{Original} \\ y_{Original} \end{pmatrix}$$

Bild 72 Koordinatentransformation mittels einer Matrix

Wir gehen nun der Frage nach, wie im Falle einer Verkettung von Koordinatensystemtransformationen verfahren werden muss. Hierzu betrachten wir das Beispiel in Bild 73. Wir gehen aus vom Koordinatensystem KS_1 und legen dies in das Koordinatensystem KS_2, indem wir es um den Ursprungspunkt um 45 Grad drehen. Der Punkt, der in KS_1 durch das Koordinatenpaar (1,1) identifiziert wird, hat in KS_2 die Koordinaten (0, `SQRT(2)`). Die Transformationsmatrix, die beschreibt, wie KS_1 in KS_2 liegt, ist ebenfalls in Bild 73 dargestellt.

Nun wird KS_2 in KS_3 eingebracht, indem der Ursprung aus seiner ursprünglichen Deckungslage mit dem Ursprung von KS_3 herausbewegt wird um 3 Einheiten auf der KS_3-Achse nach rechts. Der Punkt, der in KS_1 durch das Koordinatenpaar (1,1) identifiziert wird, hat nun in KS_3 die Koordinaten (3, `SQRT(2)`). Die Matrix, die beschreibt, wie KS_2 in KS_3 liegt, ist ebenfalls in Bild 73 dargestellt.

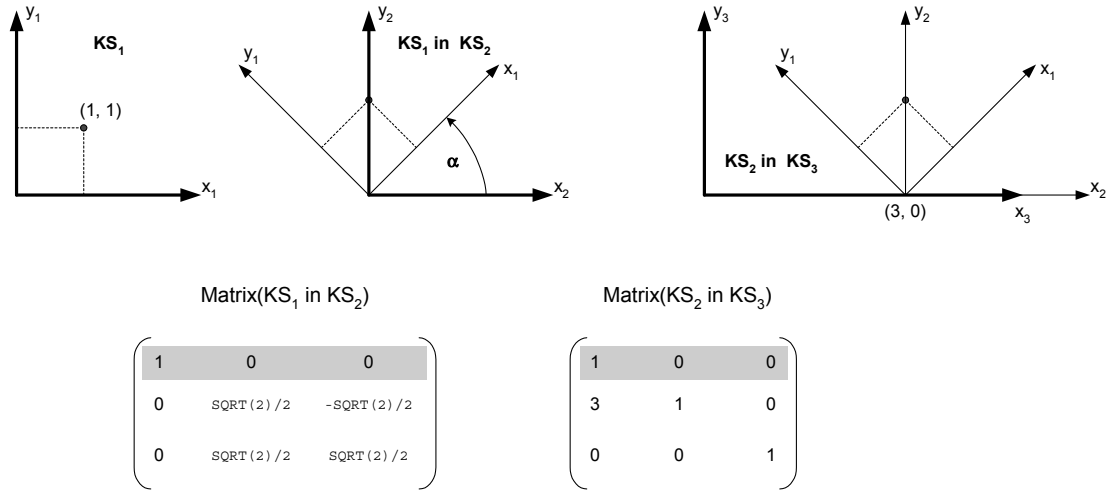

Bild 73 Beispiel zweier verketteter Koordinatensystemtransformationen

Aus den beiden in Bild 73 angegebenen Matrizen können wir durch Multiplikation eine dritte Matrix erhalten, die beschreibt, wie das Koordinatensystem KS_1 in KS_3 liegt. Diese Multiplikation ist in Bild 74 dargestellt. Für die Multiplikation sind die Matrizen in einem bestimmten Layout angeordnet, und es sind durch eingetragene Bögen die Spalten des 1. Faktors mit den Zeilen des 2. Faktors verbunden worden. Für mich stellt dieses Layout eine Hilfe bei der Berechnung der Matrizenmultiplikation dar, denn es hilft mir, die jeweils miteinander zu multiplizierenden Koeffizienten leichter zu finden. Jeder Koeffizient in der Ergebnismatrix ergibt sich als Skalarprodukt des links stehenden Zeilenvektors mit dem oben stehenden Spaltenvektor.

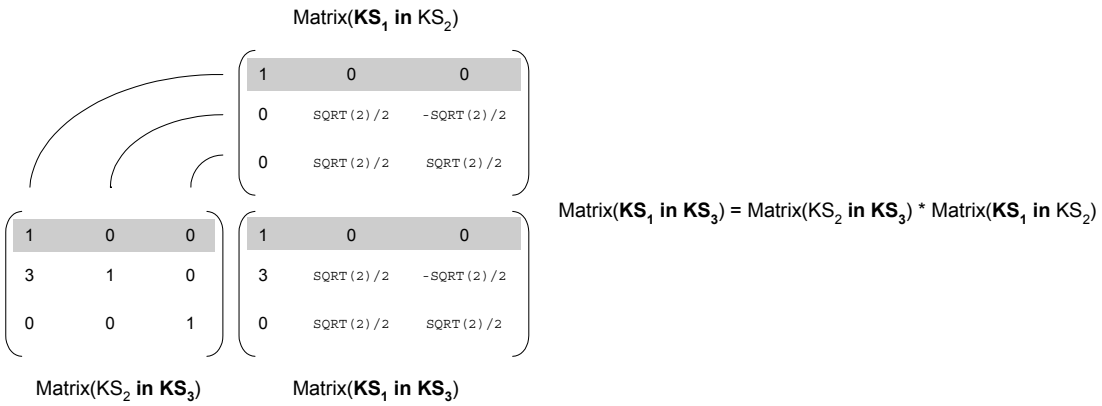

Bild 74 Absolute Transformation als Produkt zweier relativer Transformationen

Nachdem man die Matrix gewonnen hat, die beschreibt, wie KS_1 in KS_3 liegt, kann man berechnen, wie der Punkt, der in KS_1 die Koordinaten (1, 1) hat, in KS_3 liegt. Dies ist in Bild 75 dargestellt.

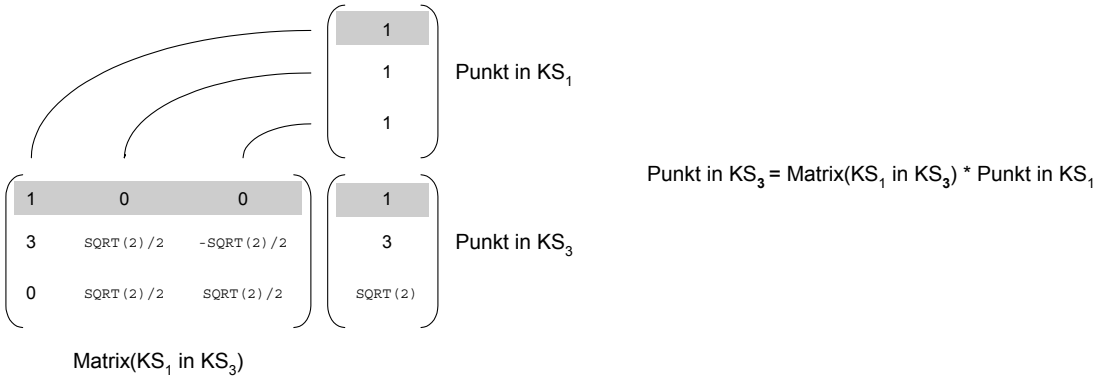

Bild 75 Transformation eines Punktes

Es kann nun durchaus vorkommen, dass man eine längere Kette von Koordinatensystemtransformationen durchführen muss. In diesem Fall muss man sich überlegen, wie man zweckmäßigerweise verfahren soll, wenn man die Koordinaten eines Punktes, die im am Anfang der Kette liegenden Koordinatensystem gegeben sind, umrechnen soll in die Koordinaten des am Ende der Kette liegenden Koordinatensystems. Bild 76 gibt für diese Überlegung eine Hilfe. Die Matrizenmultiplikation ist zwar keine kommutative, aber eine assoziative Operation, d.h. man darf die Faktoren nicht vertauschen, weil sich sonst das Ergebnis ändert, aber man darf sie beliebig klammern, weil die Klammerung keinen Einfluss auf das Ergebnis hat.

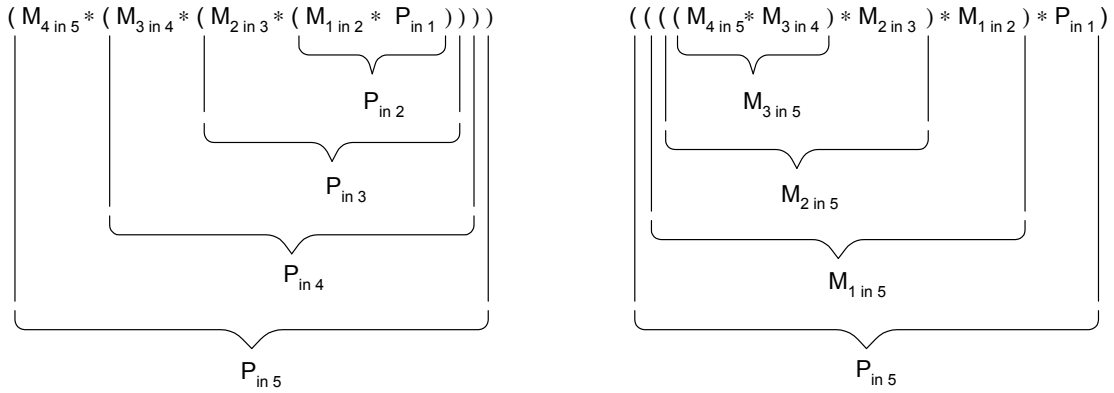

Bild 76 Alternativen für die Klammerung bei verketteten Transformationen

In Bild 76 sind zwei unterschiedliche Arten der Klammerung gezeigt. Im linken Fall werden die Koordinaten des Punktes nacheinander in die Koordinatensysteme der Kette transformiert, bis man schließlich den Punkt im äußersten Koordinatensystem liegend berechnet hat. Im rechten Fall berechnet man zuerst die Matrizen, die angeben, wie ein jeweiliges Koordinatensystem im äußersten Koordinatensystem liegt, und nur ganz zum Schluss führt man eine Koordinatensystemtransformation für den Punkt durch.

Da eine Matrizenmultiplikation etwas aufwändiger ist als die Multiplikation einer Matrix mit einem Vektor, ist die linke Klammerung aufwandsgünstiger als die rechte. Dies gilt aber nur für den Fall, dass man einen einzigen Punkt mit den gegebenen Matrizen transformieren muss. Wenn nun aber in KS_1 eine aus vielen Punkten bestehende Figur liegt, die ins äußere Koordinatensystem transformiert werden muss, dann ist die rechte Form der Klammerung

deutlich aufwandsgünstiger. Denn dann muss man die Matrizenmultiplikationen nur einmal durchführen und kann anschließend jeden Punkt mit der gewonnenen Gesamtmatrix multiplizieren. Die linke Form der Klammerung würde in diesem Fall die Tatsache nicht berücksichtigen, dass für alle Punkte die gleichen Matrizen gelten.

Überlegungen zur Systemgestaltung anhand eines Beispiels

Damit ist die allgemeine Betrachtung des Anwendungsbereichs abgeschlossen und wir kehren zur konkreten Aufgabenstellung zurück. In Bild 69 wurde das System vorgestellt, welches programmiert realisiert werden soll. In dem dort dargestellten Aufbau gibt es einen Speicher für die interne Zeichnungsbeschreibung. Wir müssen nun der Frage nachgehen, wie diese interne Zeichnungsbeschreibung konkret aussehen soll. Hierzu betrachten wir das Beispiel in Bild 77.

Ich habe entschieden, dass nicht bei jeder Koordinatensystemtransformation jeweils alle 6 Parameter benutzt werden sollen. Nur die Zoomfaktoren Z_x und Z_y sollen in jedem Falle angewendet werden dürfen. Eine Scherung soll nur vorkommen bei der Benutzung von Streckenzügen in Figuren, und Drehung und Translation sollen nur vorkommen bei der Benutzung von Figuren oder Kompositionen in Kompositionen. Streckenzüge sollen in Kompositionen nicht direkt benutzt werden dürfen, sondern nur über den Umweg von Figuren.

Die auszugebende Zeichnung sei die links oben dargestellte Komposition mit der Bezeichnung „Zweimal-Dreifiguren mit Rahmen“. Diese Komposition wurde durch zweimalige Verwendung der darunter liegenden Komposition „Dreifiguren“ gewonnen und anschließend mit einem rechteckigen Rahmen versehen. Das Koordinatensystem der Komposition Dreifiguren wurde also auf zwei unterschiedliche Arten in das Koordinatensystem der Komposition Zweimal-Dreifiguren hineingelegt, und das Koordinatesystem der Figur Rechteck wurde ebenfalls transformiert, damit der Rahmen die richtige Größe und Position bekam. Es muss also drei ZZDTT-Matrizen geben, welche diese Koordinatentransformation beschreiben

Die Komposition Dreifiguren wurde dadurch gewonnen, dass die darunter liegenden Figuren Parallelogramm, Rechteck und Dreieck jeweils mit einer bestimmten Koordinatensystemtransformation benutzt wurden. Die Figuren Parallelogramm, Rechteck und Dreieck wurden dadurch gewonnen, dass die Streckenzüge Refquadrat und Refdreieck über bestimmte ZZS-Koordinatensystemtransformationen benutzt wurden.

Da die Streckenzüge nicht selbst wieder durch Benutzung von darunter liegenden geometrischen Gebilden definiert werden, müssen sie unmittelbar durch Angabe von Punktefolgen definiert werden. Das Koordinatensystem der Streckenzüge bildet also den Anfang der Transformationskette. Die Koordinaten der in diesem Koordinatensystem definierten Punkte müssen in das Koordinatensystem der Komposition Zweimal-Dreifiguren mit Rahmen transformiert werden.

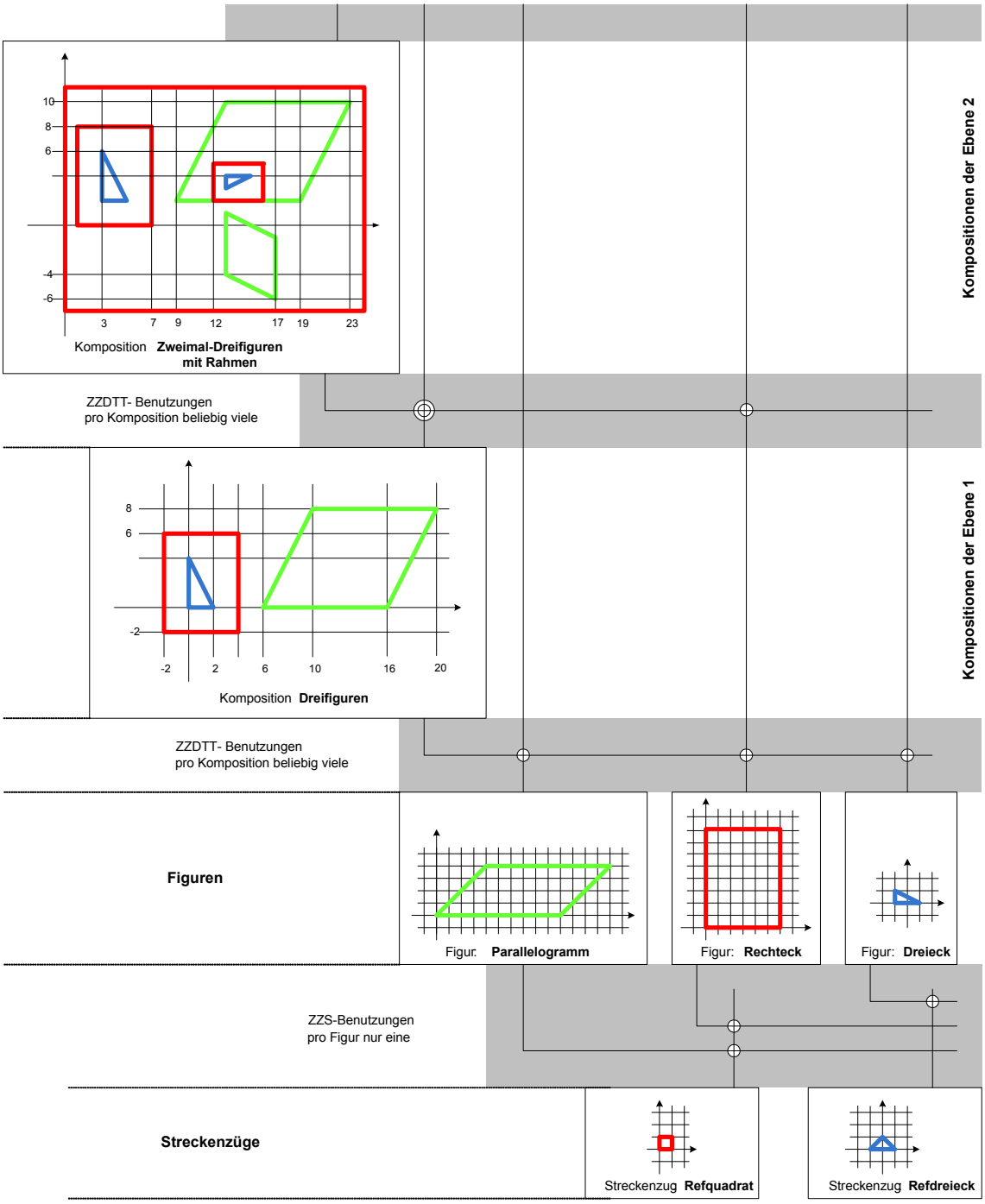

Bild 77 Beispiel einer zu beschreibenden Zeichnung

Zu der Struktur in Bild 77 soll die Speicherstruktur gelten, die in Bild 78 dargestellt ist.

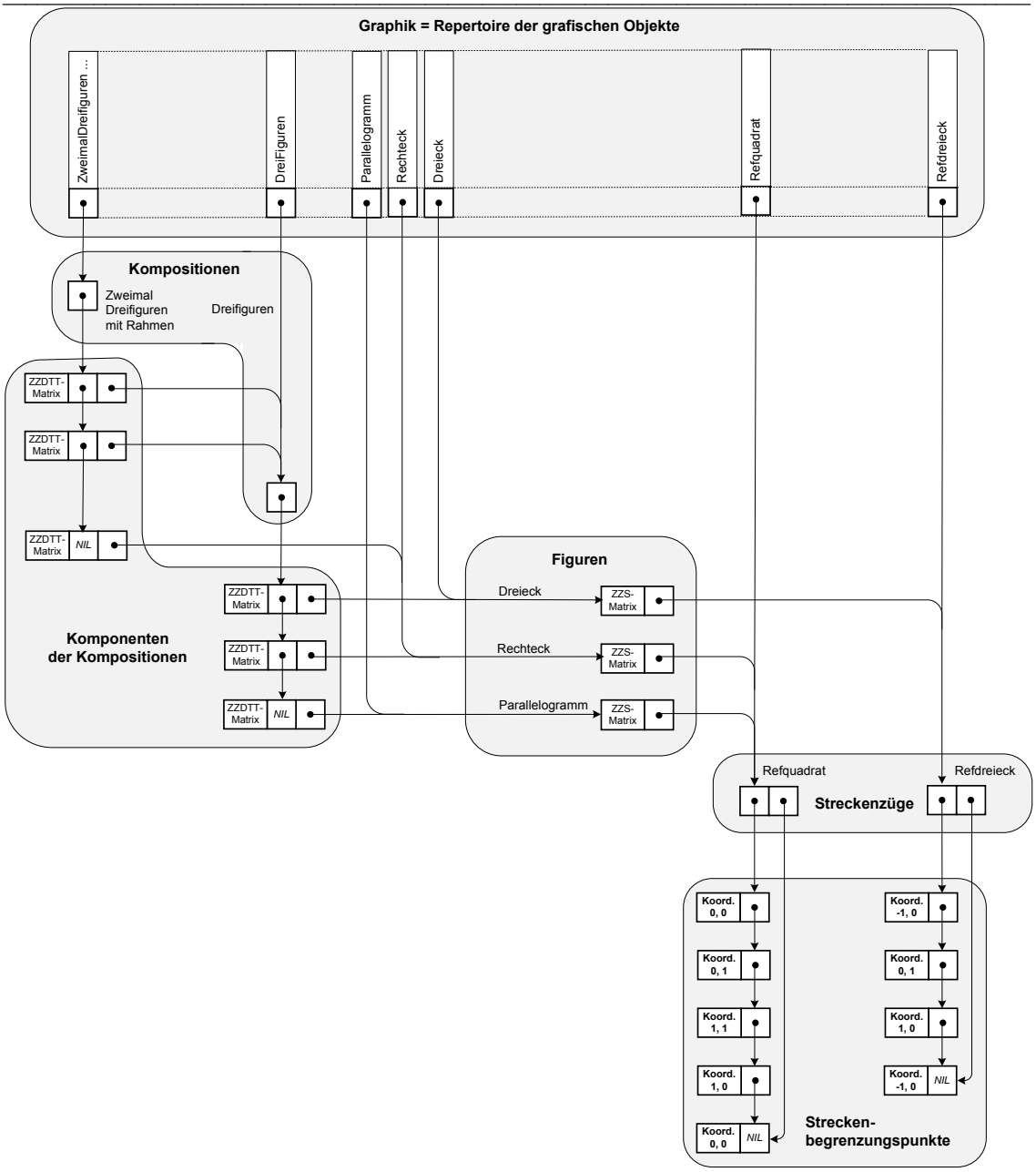

Bild 78 Datenstruktur zur Speicherung der Information aus Bild 77

Es fällt Ihnen vielleicht auf, dass auf die Liste der Komponenten, aus denen eine Komposition besteht, nur ein Pointer zeigt, nämlich auf den ersten Eintrag in dieser Liste, während auf die Liste der Punkte, aus denen ein Streckenzug besteht, zwei Pointer zeigen, nämlich auf den ersten und den letzten Punkt in dieser Liste. Darin äußert sich der Sachverhalt, dass eine Komposition schon mit einer Komponente bestehen kann und dass die Ordnung der Komponenten keinen anderen Grund hat, als die Aufzählbarkeit zu garantieren. Eine Änderung dieser Ordnung würde keine Änderung der Komposition bewirken. Ein Streckenzug braucht dagegen mindestens zwei Punkte, und die Ordnung der Punkte bestimmt das Aussehen des Streckenzugs.

Abstraktion von der Beispielsgraphik

Als nächstes wollen wir von dem in den Bildern 77 und 78 beschriebenen Beispiel abstrahieren und die Struktur der internen Zeichnungsbeschreibung allgemein erfassen. Dies gelingt uns mit Hilfe eines sogenannten Entity-Relationship-Diagramms. Zum gegebenen Problem ist dieses sogenannte ER-Diagramm in Bild 79 dargestellt. Die Rundknoten symbolisieren die sogenannten Entitäten. Das sind die Dinge, über die wir im gegebenen Zusammenhang reden müssen. Im konkreten Fall sind dies Punkte, Strecken, Streckenzüge, Figuren und Kompositionen. In Bild 79 gibt es zwar noch einen weiteren Rundknoten mit der Beschriftung „Komponente“. Aber dieser Rundknoten symbolisiert keine elementare Entität, sondern ist erst durch Abstraktion einer Beziehung gewonnen worden. „Beziehung“ und „Relation“ sind synonyme Bezeichnungen. In unserer Form der Darstellung von ER-Diagrammen werden Relationen durch rechteckige Knoten symbolisiert.

Zu jedem Streckenzug muss es einen Anfangs- und einen Endpunkt geben. Deshalb gibt es eine entsprechende Relation zwischen der Entität Streckenzug und der Entität Punkt. Jede Strecke entspricht der Ordnungsbeziehung zwischen ihrem Anfangs- und ihrem Endpunkt. Jede Figur ist definiert als transformierter Streckenzug. Deshalb gibt es die entsprechende Relation. In dieser Relation steht nicht nur die relationserklärende Beschriftung, sondern auch der Hinweis „Attribut ZZS“. Dies bedeutet, dass diese Beziehung mit der Information über eine Koordinatensystemtransformation der Form ZZS angereichert sein muss. Entsprechendes gilt für die Enthaltenseinsrelation, die die Beziehungen zwischen einer Komposition und ihren Bestandteilen erfasst. Jeder Bestandteil einer Komposition ist über eine Koordinatensystemtransformation der Form ZZDTT in die Komposition einzubringen. Damit die Komponenten einer Komposition nacheinander abgearbeitet werden können, müssen sie in einer Ordnung liegen. Damit man diese Ordnung als Relation darstellen kann, braucht man die zu ordnenden Elemente als Entität. Deshalb ist um die Enthaltenseinsrelation ein Entitätsknoten gezeichnet. Dadurch wird symbolisiert, dass die Elemente der Relation nun selbst wieder als Entitäten betrachtet werden, die selbst wieder an Relationen beteiligt sein können, wie in diesem Fall an der Ordnungsrelation.

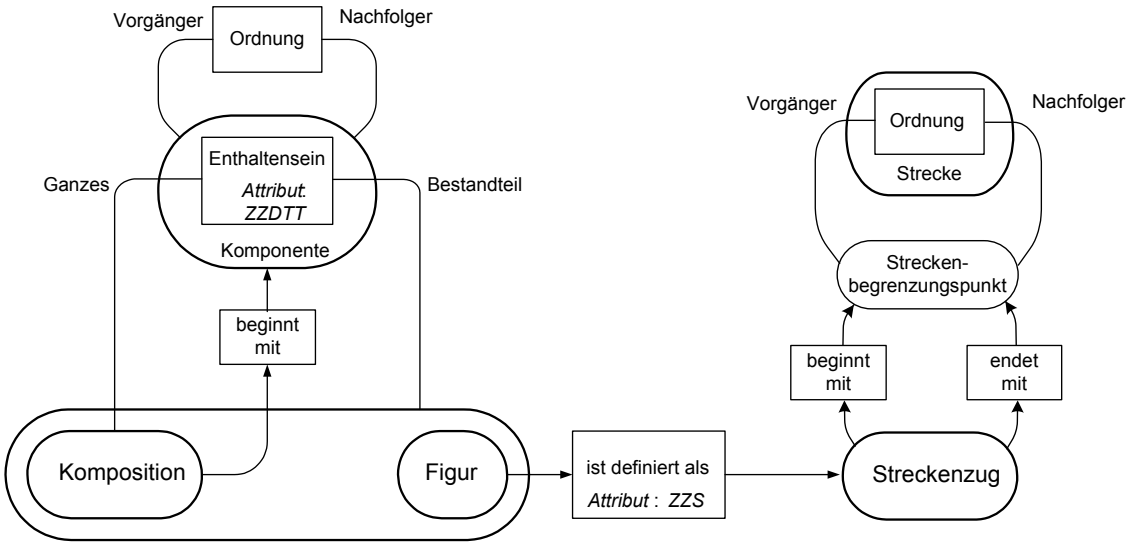

Bild 79 Entity-Relationship-Diagramm als Abstraktion von Bild 78

Systemaufbau als „Bühne“ für den Ablauf

Ich habe schon angekündigt, dass wir diese Aufgabe durch ein objektorientiertes Programm lösen wollen. Dennoch will ich nun zuerst einmal der Frage nachgehen, wie wir die Aufgabe denn gelöst hätten, wenn uns das Konzept der Objektorientierung nicht bekannt gewesen wäre. Ich gehe sogar so weit, dass ich annehme, der Computer wäre noch gar nicht erfunden und wir befänden uns beispielsweise im Jahre 1912. Ich stelle mir vor, wie ich am Schreibtisch sitzend und nur unter Verwendung von Papier, Bleistift und Radiergummi die Aufgabe erledigen würde. Im Laufe meiner Überlegungen bin ich zu dem Aufbauplan in Bild 80 gelangt. Sie sollten nun keinen Schreck bekommen und befürchten, dass ich von Ihnen das Gestalten eines derart komplizierten Aufbauplans erwarten würde. Bedenken Sie, dass ich solche Pläne schon seit mehr als 20 Jahren mache und deshalb eine gewisse Routine habe.

Der Plan ist nicht allzu schwer zu verstehen, wenn wir ihn in der richtigen Reihenfolge betrachten. Die rechteckigen Knoten stellen Akteure dar. Da nie zwei Akteure gleichzeitig agieren müssen, kann ich selbst nacheinander die Rollen dieser Akteure übernehmen. Die runden oder abgerundeten Knoten stellen Speicherplätze dar, die ich mir als beschreibbare Papierblätter vorstelle.

Der Initiator in Bild 80 ist der gleiche, den wir auch schon aus Bild 69 kennen. Auch den oben rechts im Bild 80 liegenden Streckenausgeber kennen wir schon aus Bild 69. Die Aufgabe des Initiators besteht in unserer Vorstellung darin, die Datenstruktur aus Bild 78 auf ein Blatt Papier oder eine Tafel zu schreiben und zu zeichnen, die als Speicher für die geschichtete Zeichnungsbeschreibung dienen. Außerdem gehört es auch noch zur Aufgabe des Initiators, ein Formular auszufüllen, das in Bild 80 links neben dem Initiator liegt und drei Felder enthält: In das Feld „ID eines Bestandteils“ muss die Bezeichnung *zweimalDreifiguren mit Rahmen* eingetragen werden, weil dies die auszugebende Komposition ist. In das Feld „$KS_{Bestandteil}$ in $KS_{Ausgabe}$“ muss die sog. neutrale Matrix eingetragen werden, weil das Koordinatensystem der Ausgabefläche gleich dem Koordinatensystem der auszugebenden Komposition *zweimalDreifiguren mit Rahmen* sein soll. Und in das Zahlenfeld muss eine Eins eingetragen werden, weil hier die Nummer der als nächstes zu lesenden Komponente der Komposition stehen muss.

Formulare mit drei Feldern, wie der Initiator eines ausfüllen muss, kommen in Bild 80, d.h. also auf meinem Schreibtisch noch mehrfach vor. Der Stack besteht ausschließlich aus einem Stapel solcher Formulare, die jeweils in bestimmter Weise ausgefüllt wurden. Ganz oben in Bild 80 finden wir ein weiteres Formular dieser Art; es liegt neben dem Streckenzugabwickler, der dem Streckenausgeber nacheinander die Koordinaten der auszugebenden Strecken liefert. Hierzu braucht er Zugang zur geschichteten Zeichnungsbeschreibung, von wo er die Koordinaten der geordneten Punkte entnimmt, die er einer Koordinatensystemtransformation unterwerfen muss, bevor er sie als Endpunkte von zu zeichnenden Strecken weitergeben kann. Damit er jeweils weiß, welche Punktkoordinaten er als nächstes lesen und transformieren muss, kann er einerseits lesend auf das Feld „Identifikator eines Streckenzugs“ zugreifen, wo beispielsweise die Identifikation *refquadrat* stehen könnte. Und andererseits braucht er die links daneben liegende Ordnungsnummer des Punktes in der Punktefolge. Die Matrix, welche die Koordinatensystemtransformation beschreibt, die der Streckenzugabwickler auf die einzelnen Punkte des identifizierten Streckenzugs anwenden soll, steht ganz links außen im dritten Feld des Formulars.

Diese Matrix findet man allerdings nicht in der geschichteten Zeichnungsbeschreibung, denn dort sind ja nur die Matrizen enthalten, die die relativen Koordinatensystemtransformationen darstellen, aber nicht die absoluten. Die absolute Koordinatensystemtransformation ergibt sich erst durch Multiplikation mehrerer Matrizen aus der Zeichnungsbeschreibung (siehe Bild 76). Diese Multiplikationen muss irgendwer durchgeführt haben, bevor der Streckenzugabwickler in Aktion treten kann.

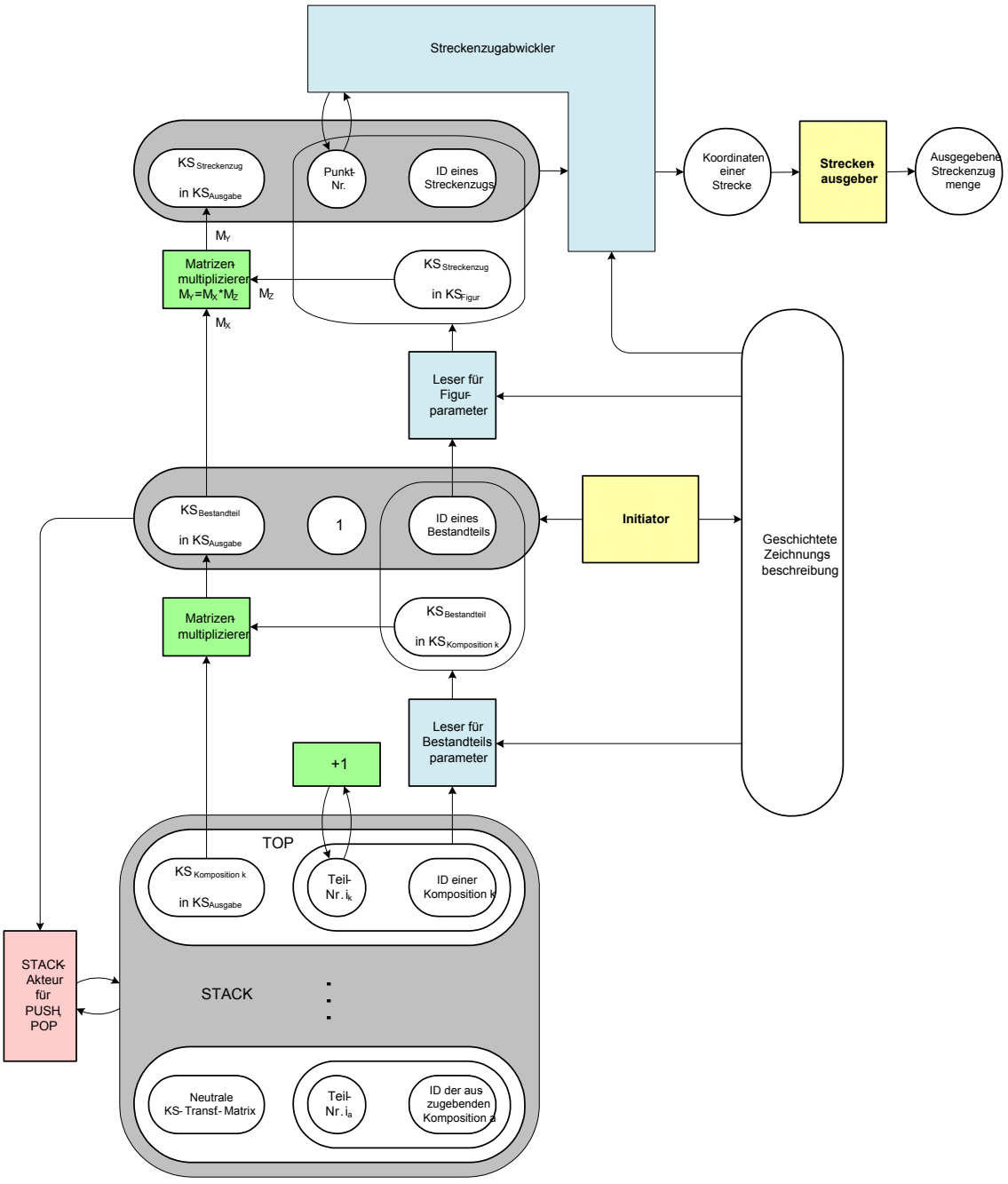

Bild 80 Aufbau des Systems zur Streckenfolgengenerierung aus der Zeichnungsbeschreibung

Dass wir einen Stack benötigen, sollte niemanden überraschen, denn wir haben ja hier eine Rekursion, weil in Kompositionen selbst wieder Kompositionen als Komponenten vorkommen dürfen.

Weil einer Komposition kein Streckenzug zugeordnet werden kann, muss die vom Initiator initiierte Informationsstruktur ge-PUSH-t werden. Sie wird dadurch zum TOP des Stack. Nun kann der Leser für Bestandteilsparameter in der Zeichnungsbeschreibung nachschauen, wie die erste Komponente dieser Komposition heißt und wie sie durch eine Koordinatensystemtransformation in die Komposition eingebracht wurde. In unserem Beispiel aus Bild 78 ist die erste Komponente von *zweimalDreiFiguren* die Komposition *dreiFiguren*. Als Ergebnis des Lesens liegt nun also im Top des Stacks der Identifikator *dreiFiguren* und eine entsprechende Koordinatensystemtransformation, wie *dreiFiguren* das erste Mal in *zweimalDreiFiguren* liegen soll. Nun kann der Matrizenmultiplizierer aktiv werden, der die beiden an seinem Eingang liegenden Matrizen miteinander multipliziert und als Ergebnis die Matrix liefert, die beschreibt, wie *dreiFiguren* in der Ausgabe liegt.

Da *dreiFiguren* selbst wieder eine Komposition ist, wird auch dieser Informationssatz nun ge-PUSH-t und wird zum neuen TOP. Als nächstes wird nun der erste Bestandteil von *dreiFiguren* geholt, welcher nach Bild 78 das *dreieck* ist. Am Ausgang des Lesers für Bestandteilsparameter liegt nun also der Identifikator *dreiFiguren* und die Koordinatensystemtransformation, die beschreibt, wie *dreieck* in *dreiFiguren* liegt. Es findet wieder eine Matrizenmultiplikation statt, so dass nun wieder ein vollständiger Informationssatz links vom Initiator liegt. Dieser Informationssatz wird nun aber nicht mehr ge-PUSH-t, denn es wird ja keine Komposition identifziert, sondern eine Figur. Dieser Figur kann ein Streckenzug zugeordnet werden. Der Leser für Figurparameter schaut in der Zeichnungsbeschreibung nach, welcher Streckenzug mit welcher Transformation der Figur *dreieck* zugeordnet ist. Diese Information liefert er ganz oben an seinen Ausgang. Als Streckenzugidentifikator wird in diesem Fall *refdreieck* gelesen. Neben dem Eintragen dieses Identifikators in das rechte Feld des Formulars kann der Leser auch gleich die Zelle für die laufende Nummer des aktuellen Punktes in der zugehörigen Punktefolge mit dem Anfangswert 1 belegen. Nun kann wieder eine Matrizenmultiplikation stattfinden, an deren Ende die Information vollständig vorliegt, die der Streckenzugabwickler benötigt, um die Ausgabe aller Strecken von *dreieck* in seiner ersten Erscheinungsform in *zweimalDreiFiguren* zu veranlassen. Nach dieser Ausgabe wird im TOP des Stacks die Bestandteilsnummer um 1 erhöht und der Leser schaut in der Zeichnungsbeschreibung nach, ob es einen Bestandteil mit dieser höheren Ordnungsnummer gibt. Dies ist hier der Fall, denn als nächstes kann *rechteck* ausgegeben werden. Irgendwann aber wird die Bestandteilsnummer einen Wert erreicht haben, unter dem kein Bestandteil mehr vorhanden ist. Dann ist die gesamte Komposition, die durch den TOP definiert wird, dargestellt und es wird ein POP ausgeführt. Wenn nun der Stack nach diesem POP nicht leer ist, kann wieder die Bestandteilsnummer im TOP erhöht werden und das Spiel geht weiter. Irgendwann ist der Stack leer, und damit ist die gesamte Ausgabe erledigt.

Gewinnung der Ablaufdarstellung

Ich habe den Ablauf, den ich hier verbal beschrieben habe, in einen Ablaufplan gefasst, der in Bild 81 gezeigt ist. Man sieht sofort, dass dieser Ablauf nicht baumstrukturiert ist. Ich habe deshalb anschließend versucht, den Ablauf in eine baumstrukturierte Form zu transformieren. Das Ergebnis meiner Bemühungen ist in Bild 82 gezeigt. Auch dieser Ablauf ist nicht baumstrukturiert, aber er hat schon eine Form, die darauf hinweist, was man tun muss, um zu einer

baumstrukturierten Form zu kommen. Man erkennt nämlich die Notwendigkeit der Rekursion. In Bild 83 ist deshalb das mit Stackplätzen angereicherte Petrinetz gezeigt.

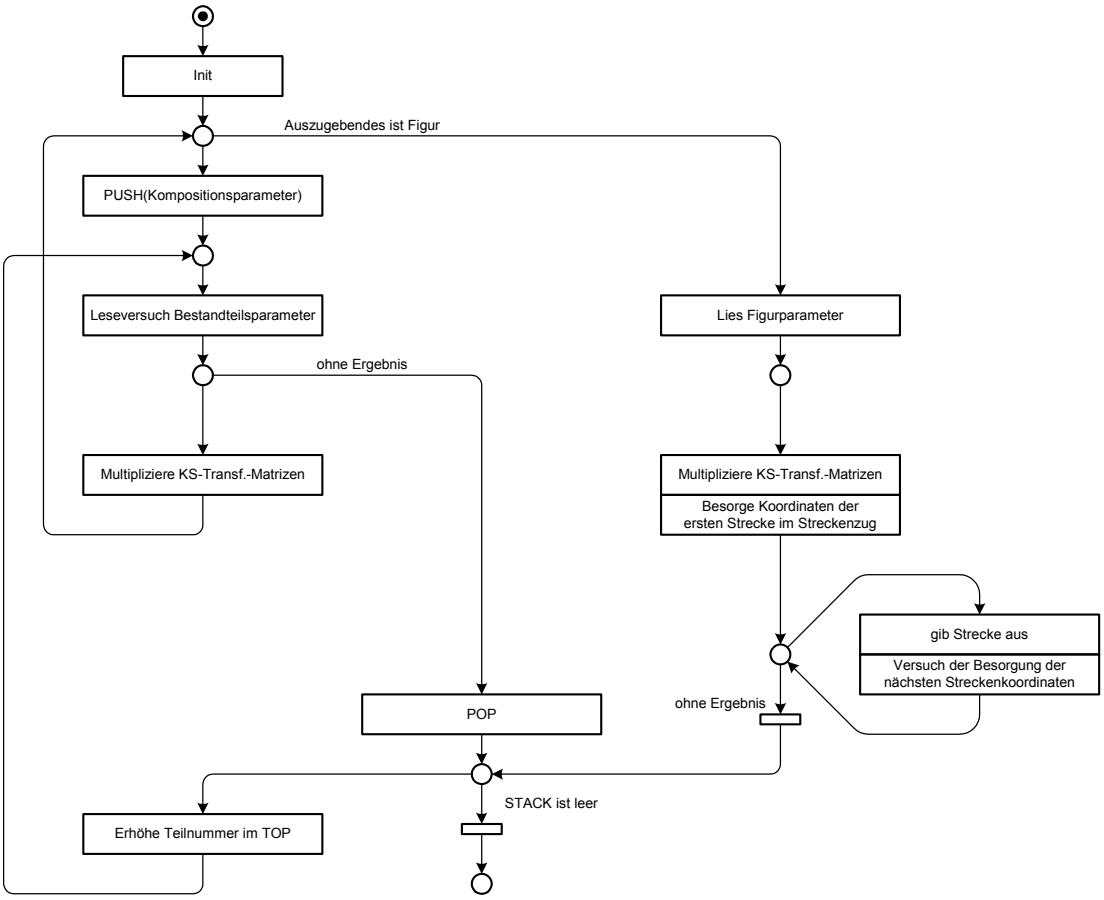

Bild 81 „Wildstrukturierter“ Ablauf zum Aufbau in Bild 80

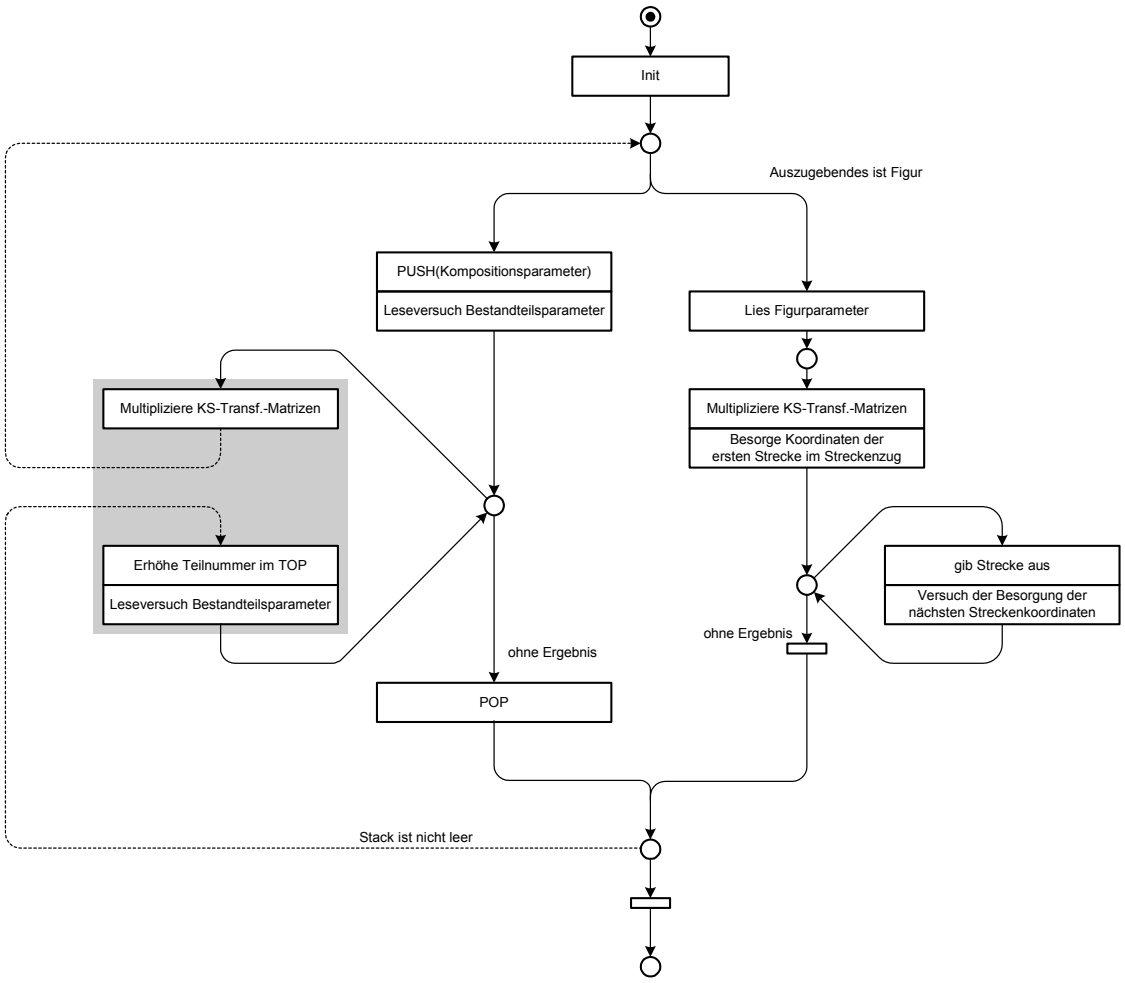

Bild 82 Strukturverbesserte Version des Ablaufs aus Bild 81

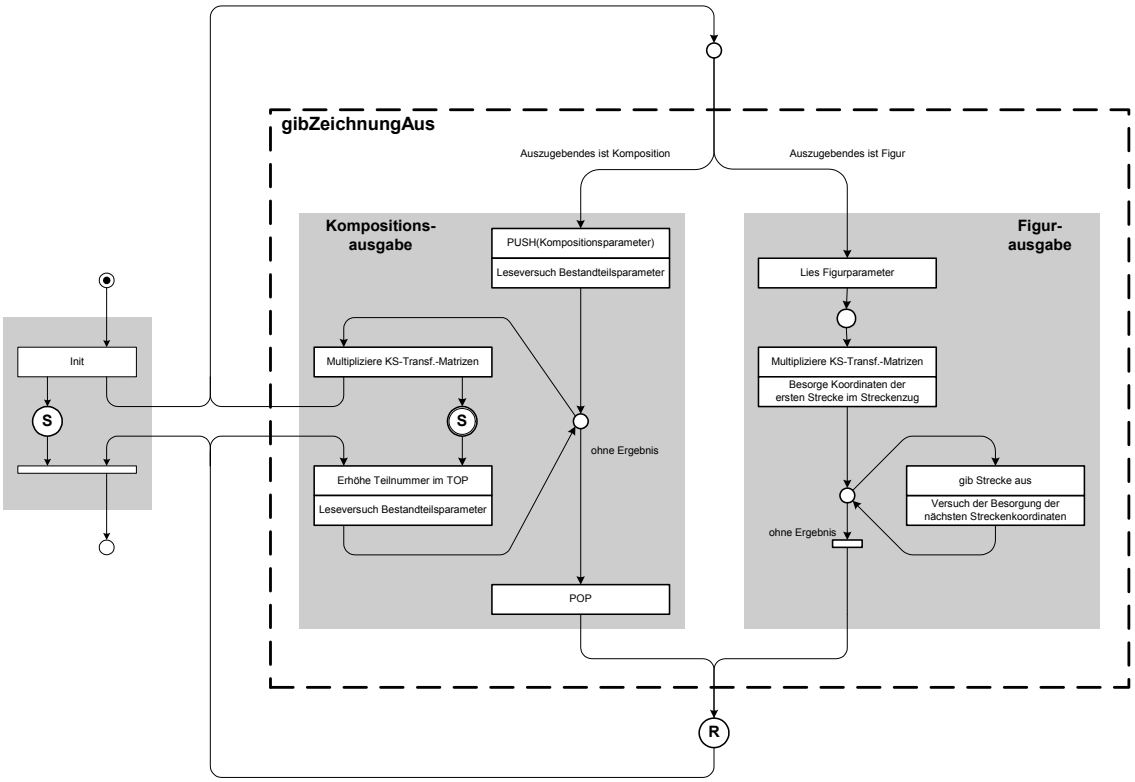

Bild 83 Endgültige Version des Ablaufs aus Bild 81 mit expliziter Rekursion

Es ist uns also gelungen, die Aufgabenstellung völlig ohne Objektorientierung zu realisieren. Es fragt sich deshalb, worin denn die Vorteile der Objektorientierung bestehen. Dieser Vorteil kann nicht darin bestehen, dass wir einen besseren Algorithmus finden, denn der in der jetzt dargestellten Lösung benutzte Algorithmus ist nicht mehr grundsätzlich verbesserbar. **Somit kann der Vorteil der Objektorientierung nur darin liegen, dass der Programmtext übersichtlicher strukturiert wird als in unserem ersten Lösungsansatz.** Diese bessere Strukturierung kommt zum einen von der Kapselung durch die abstrakten Datentypen, und zum anderen durch die Nutzung der Polymorphie, die es uns erlaubt, die Abfrage im Ablauf, ob das Auszugebende eine Figur oder eine Komposition sei, aus dem Programm zu verbannen und in die Vererbungsstruktur zu transportieren.

Wo wir nur kapseln wollen, aber keine Vererbung brauchen, könnten wir uns auf die Benutzung von abstrakten Datentypen beschränken. Wenn man aber schon objektorientiert programmiert, verzichtet man ganz auf abstrakte Datentypen und nimmt auch dort Klassen, wo man eigentlich keine Vererbung braucht. Dabei braucht man Vererbung zumindest immer dann, wenn man Polymorphie haben will.

Um in unserem Falle Polymorphie zu nutzen, habe ich über die beiden Klassen Figuren und Kompositionen ein Interface gesetzt, das ich „Verwendbares“ genannt habe. Zusammen mit den Streckenzügen nutzen die „verwendbaren“ Objekte das Interface „Darstellbares“.

Gewinnung des Programms aus dem Systemverständnis: Steuerkreisstrukturierung versus Klassenstrukturierung

Anhand der Bilder 80 bis 83 wurde versucht, ein Verständnis des Systems und des in diesem System ablaufenden Geschehens zu vermitteln. Der Weg von diesem Systemverständnis zum Programm ist nicht allzu schwer, wenn man weiß, in welcher Richtung man suchen muss. Es gibt mindestens zwei grundsätzlich unterschiedliche Vorgehensweisen – die Steuerkreisstrukturierung und die Klassenstrukturierung. Die Steuerkreisstrukturierung wird hier nur kurz skizziert, wogegen die Klassenstrukturierung im Beispiel konkret ausgeführt wird.

Die Steuerkreisstrukturierung hat ihren Namen von dem „Kreis“, der im Systemaufbau sichtbar ist (siehe Bild 84): Das System besteht aus einem sog. „Operationsakteur“ und einem „Steuerakteur“, die zusammen mit den verbindenden Kanälen einen Kreis bilden. Der innere Aufbau des Operationsakteurs würde im betrachteten Beispiel genau die Struktur haben, die in Bild 80 dargestellt ist. Die darin vorkommenden Speicher würden sich programmtechnisch als Datenstrukturen äußern, und jeder Akteur in diesem Aufbau entspräche einer Prozedur. Der Steuerakteur würde durch das in Bild 81 dargestellte Hauptprogramm bestimmt, welches festlegt, wann welche Operationsprozedur aufgerufen werden muss.

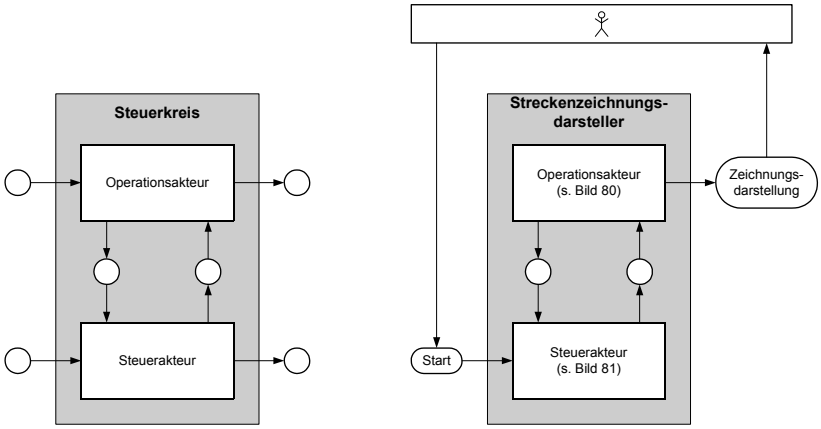

Bild 84 Das Steuerkreismodell, Konzept und Anwendungsbeispiel

Bei der Steuerkreisstrukturierung eines Programms entstehen immer drei streng getrennte Programmabschnitte:
- Datenstrukturen
- Operationsprozeduren
- Hauptprogramm

Die Steuerkreisstrukturierung ist eine gut verständliche Strukturierung eines Programms, weil die Programmabschnitte eins-zu-eins den Elementen des Systemaufbaus zugeordnet werden können. Aber im Hinblick auf eine gewünschte Änderungsfreundlichkeit ist die Steuerkreisstrukturierung eines Programms nicht zweckmäßig. Denn wenn man die Codierung irgendwelcher Datenstrukturelemente ändern will, bringt dies i.a. die Notwendigkeit mit sich, viele Operationsprozeduren ändern zu müssen, die man sich zusammensuchen muss. Deshalb wurde die Klassenstrukturierung erfunden, die dadurch gekennzeichnet ist, dass die Implementierung jedes Datenstrukturelementtyps zusammen mit seinen zugehörigen benutzenden Prozeduren in einem zusammenhängenden Programmabschnitt gekapselt wird. Bild 85 veranschaulicht das Prinzip.

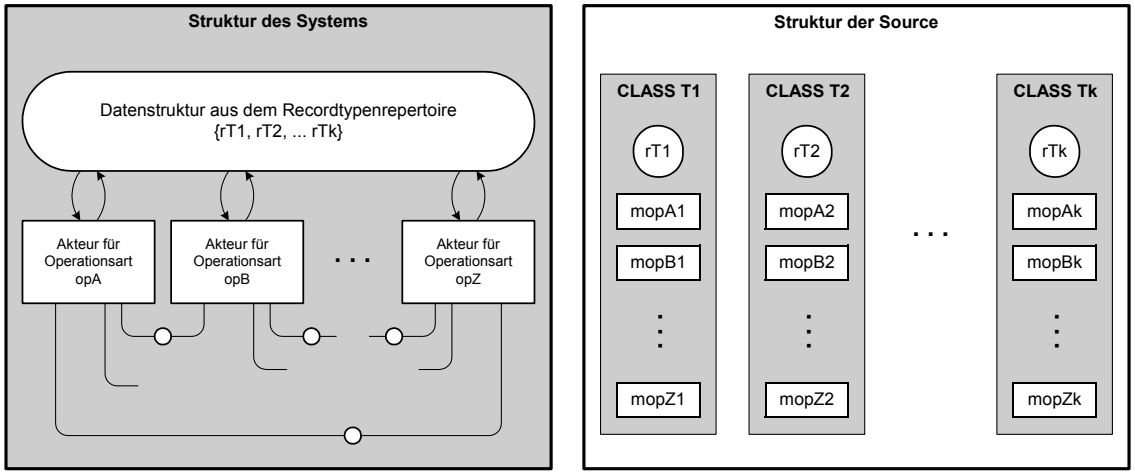

Bild 85 Das Prinzip der Klassenstrukturierung

Zum Verständnis des Begriffs „Recordtyp“ rTi betrachte man noch einmal die in Bild 78 dargestellte Datenstruktur. Jedes in den Bereichen Graphik, Kompositionen, Komponenten, Figuren, Streckenzüge und Punkte vorkommende, aus nebeneinanderliegenden Teilen bestehende Rechteck ist datentechnisch gesehen ein sog. Record. Jeder solche Record ist eine geordnete Folge von Speicherzellen, deren Größe durch die Art der Informationen bestimmt wird, die in die Zellen eingebracht werden sollen. Der Typ eines Records wird also durch die Typen der einzelnen Speicherzellen, aus denen er besteht, und ihre Ordnung bestimmt. Im Falle der Datenstruktur in Bild 78 gibt es sechs verschiedene Recordtypen:

- *Graphik* enthält ein Array gleichartiger Records, die jeweils zwei Felder haben. In dem einen liegt der Pointer auf ein bestimmtes Exemplar, welches von der Klasse Graphik erzeugt wird. In dem anderen steht die Bezeichnung dieses Objekts.
- Die Records vom Typ *Komposition* haben nur ein Feld. Dieses ist für einen Pointer auf einen Record vom Typ Komponente vorgesehen.
- Die Records vom Typ *Komponente* haben drei Felder: eines für eine Transformationsmatrix vom Typ ZZDTT, ein zweites für einen Pointer auf den Record der nächsten Komponente, und ein drittes für einen Pointer auf den Record des verwendeten grafischen Objekts, welches eine Komposition oder eine Figur sein kann.
- Die Records vom Typ *Figur* haben zwei Felder: eines für eine Transformationsmatrix vom Typ ZZS, und ein zweites für einen Pointer auf den Record des Streckenzuges, aus dem die Figur gewonnen wird.
- Die Records vom Typ *Streckenzug* haben zwei Felder: eines für den Pointer auf den Record des Anfangspunkt des Streckenzugs, und ein zweites Feld für den Pointer auf den Record des Endpunkts.
- Die Records vom Typ *Punkt* haben zwei Felder: eines für die Koordinaten des Punktes, und ein zweites für den Pointer auf den Record des nächsten Punktes im Streckenzug.

Suche nach den Klassen

Unser Programm zur Ausgabe von Zeichnungen aus Streckenzügen wurde in der Sprache JAVA formuliert. In JAVA sind alle per Bezeichnung eingeführten Speicherzellen Behälter für Pointer, ohne dass dies explizit durch eine besondere Symbolik zum Ausdruck gebracht werden muss. Wenn also deklariert wird

```
Klassenname  var1, var2, var3;
```

dann bewirkt dies die Reservierung von drei Speicherzellen für die spätere Aufnahme von Pointern auf Attributrecords der angegebenen Klasse. Der Speicherplatz für einen Attributrecord wird erst bei der Ausführung einer NEW-Anweisung reserviert:

```
var2 = new Klassenname( Liste der Konstruktorparameter );
```

In JAVA kann es deshalb keinen *call by value* geben, d.h. alle Parameterübergaben erfolgen zwangsläufig als *call by reference*.

In JAVA muss der gesamte Programmtext in Klassen formuliert sein. Wenn man einen solchen absoluten Klassenzwang haben will, muss man etwas zulassen, was man ursprünglich nicht mit dem Begriff einer Klasse verbindet. Man muss nämlich zulassen, dass nicht nur die Exemplare von Klassen Objekte sind, sondern dass die Klassen selbst auch Objekte sind. Was unter einem Objekt zu verstehen ist, wurde schon bei der Einführung der abstrakten Datentypen erklärt. Wenn man nun will, dass sowohl die Exemplare einer Klasse als auch die Klasse selbst Objekte sind, muss es im Definitionstext einer Klasse zwei Abschnitte geben, von denen der eine die Eigenschaften beschreibt, welche diese Klasse als Objekt hat, während der andere beschreibt, welche Eigenschaften die Objekte haben sollen, die später als Exemplare der definierten Klasse geschaffen werden.
Es gibt zwei unterschiedliche Arten von Eigenschaften, die ein Objekt haben kann:
Die Attribute sind die Felder des Records, also die Speicherzellen für die aktuellen „Gedächtnisinhalte“ des Objekts, und die Methoden sind die Prozeduren, deren Ausführung man von dem Objekt verlangen kann.

```
CLASS xyz
{ {ac1, ac2, ... ack}      /*Attribute der Klasse als Objekt*/
  {mc1, mc2, ... mcp}      /*Methoden  der Klasse als Objekt*/

  {ae1, ae2, ... aem}      /*Attribute der Exemplare der Klasse*/
  {me1, me2, ... meq}      /*Methoden  der Exemplare der Klasse*/
}
```

Damit man in ein- und derselben Klassendefinition die beiden Abschnitte unterscheiden kann, wurde das Unterscheidungsprädikat „static“ eingeführt. Ein Attribut oder eine Methode der Kategorie *static* gehören zur Klasse, wenn sie als Objekt betrachtet wird, während sich die anderen Attribute und Methoden in der gleichen Klassendefinition erst äußern, wenn Exemplare zu dieser Klasse geschaffen werden.

Die static-Attribute sind insbesondere von Bedeutung, wenn von einer Klasse gar keine Exemplare erzeugt werden. Mindestens eine solche Klasse muss in jedem Programm vorkommen, nämlich die Klasse, welche die Methode `main` enthält. Diese Methode wird zu Beginn der Programmausführung automatisch als erste ausgeführt, obwohl es im Programm keinen Aufruf dieser Methode gibt.

Die Überlegungen, welche Klassen zusätzlich zu `Aufgabe8Main` für unsere Aufgabe noch gebraucht werden, habe ich gemeinsam mit dem Assistenten angestellt, der anschließend den Java-Code erstellte. Das Ergebnis unserer Planungsüberlegungen habe ich mit zwei Diagrammen (siehe Bilder 85a und 85b) veranschaulicht. Diese Diagramme sind vom Typ ERD (Entity-Relationship-Diagram), zu dem auch das Diagramm in Bild 79 gehört.

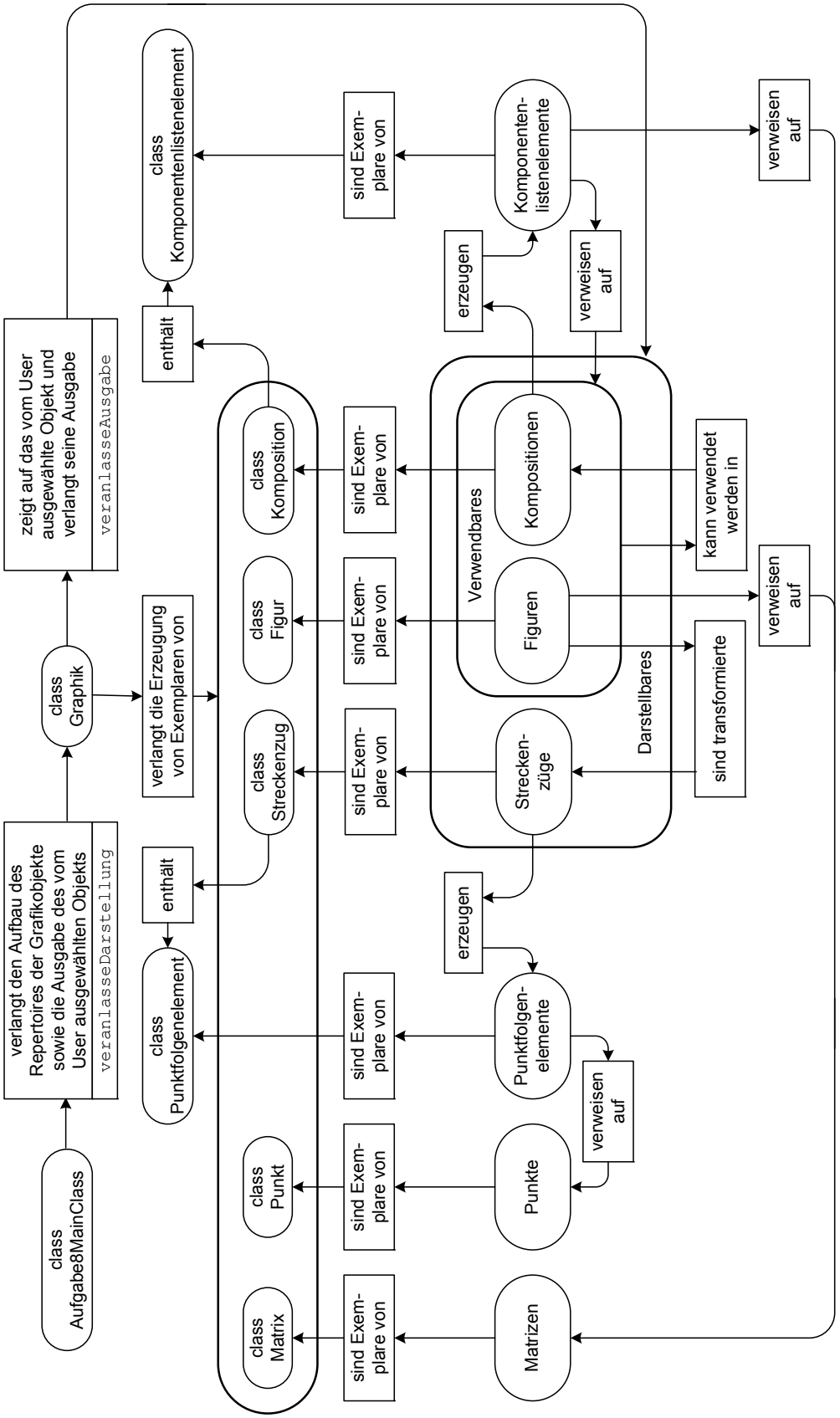

Bild 85a Planung der Klassenstruktur für Aufgabe 8 (Teil 1)

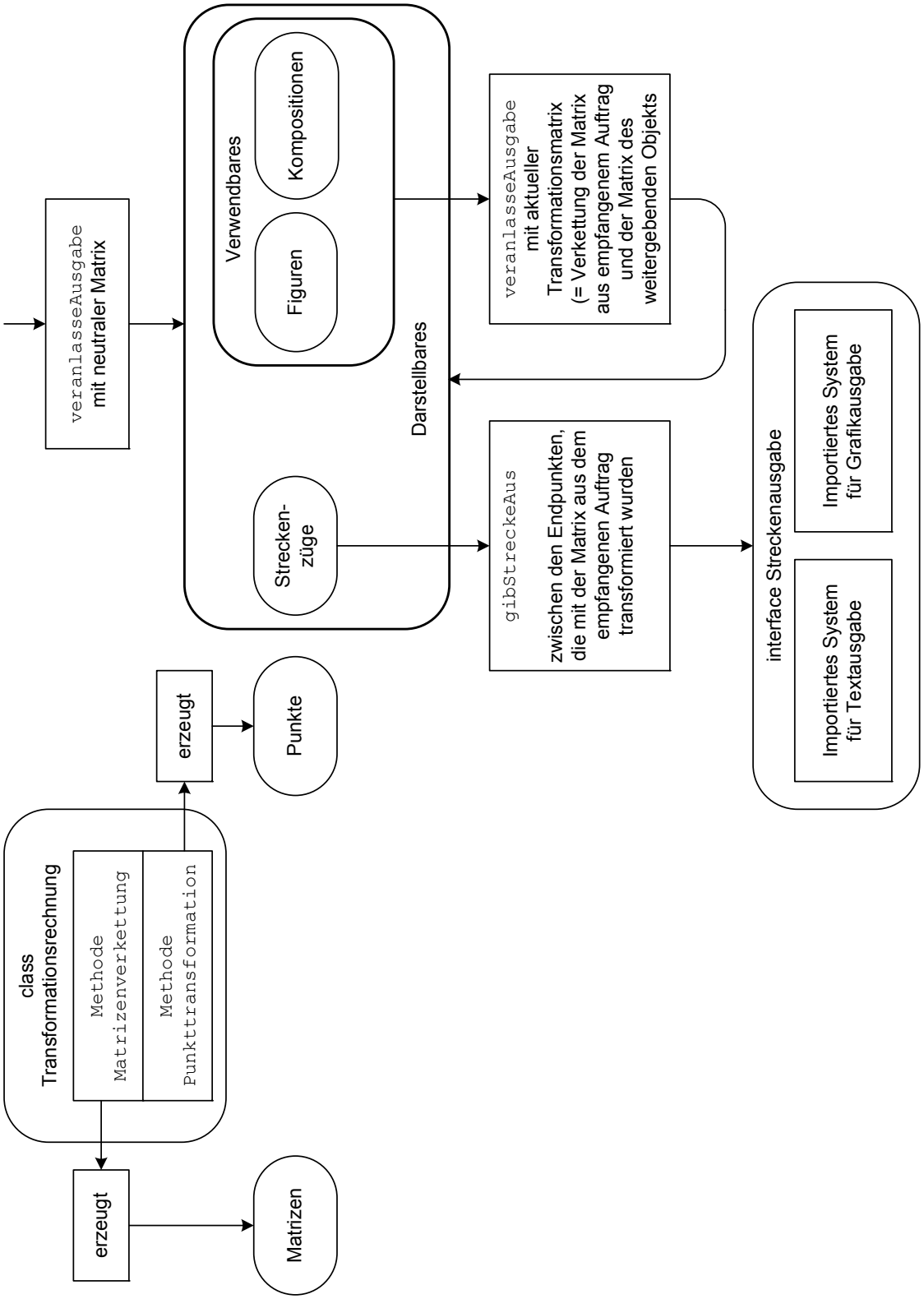

Bild 85b Planung der Klassenstruktur für Aufgabe 8 (Teil 2)

Ich möchte an dieser Stelle betonen, dass die beiden Diagramme ausschließlich für die Betrachtung durch Menschen erstellt wurden, so dass ausschließlich ihr didaktischer Nutzen zu beachten war und keinerlei Zwang bestand, formale Regeln zu beachten. Diese Diagramme sind nicht als Eingabe für irgendwelche Tools erstellt worden, die daraus irgendetwas Formales erzeugen. Es steht für mich außer Frage, dass man die Teilnehmerin nicht ernst zu nehmen braucht, die einmal auf einer Software-Engineering-Konferenz behauptete: „Wenn man aus Ihren Diagrammen keinen Code generieren kann, sind sie nutzlos!“ Die Diagramme stellen auch keine „Nachdokumentation“ zum fertigen Programmcode dar, denn sie sind zum Zeitpunkt der Planung entstanden und haben geholfen, ein wohlstrukturiertes Programm zu schreiben.

Man sieht, dass es neben der Klasse `Aufgabe8Main` noch zwei weitere Klassen gibt, zu denen keine Exemplare erzeugt werden. Das eine ist die Klasse `Graphik`, die zwei Methoden enthält, welche von `main` aufgerufen werden. Die eine Methode ist `initialisiere`; sie erzeugt alle Objekte, welche die Datenstruktur in Bild 78 bilden. Die zweite Methode ist `veranlasseDarstellung`; ihre Ausführung führt zu Aufrufen von Methoden, die vom `interface Darstellbares` die Signatur `veranlasseAusgabe` geerbt haben. Hier kommt also Polymorphie vor.

Auch die Klasse `Transformationsrechnung` erzeugt keine Exemplare. Sie enthält die Methode zur Verkettung zweier Transformationsmatrizen und die Methode zur Transformation eines Punktes gemäß einer Transformationsmatrix. Die erstgenannte Methode bewirkt die Erzeugung eines Exemplars der Klasse `Transformationsmatrix` und gibt einen Pointer auf dieses Exemplar an das aufrufende Objekt. Die zweite Methode bewirkt die Erzeugung eines Exemplars der Klasse `Punkt` und gibt einen Pointer auf dieses Exemplar an das aufrufende Objekt. Es wäre auch möglich gewesen, diese beiden Methoden als Exemplarmethoden der Klasse `Transfoprmationsmatrix` zuzuordnen, denn das jeweilige Matrixexemplar tritt in beiden Methoden als Faktor der Multiplikation auf.

Beispiele für konstante static-Attribute in unserem Programm zur Zeichnungsausgabe sind
- Pointer auf alle darstellbaren Gebilde in der Klasse `Graphik`, also refdreieck, refquadrat, dreieck, rechteck, parallelogramm, dreiFiguren und zweimalDreiFiguren;
- die neutrale Matrix in der Klasse `Transformationsmatrix`.

Beispiele für static-Methoden sind:
- `main` in der Klasse `Aufgabe8MainClass`;
- `initialize` und `veranlasseDarstellung`
- `newZZSMatrix` und `newZZDTTMatrix` in der Klasse `Transformationsmatrix`;
- `verketteTransformationen` und `transformierePunkt` in der Klasse `Transformationsrechnung`.

Der in Bild 80 vorkommende Stack für die Komponenten der Kompositionen muss im Programmtext nicht explizit auftauchen, denn jeder Prozeduraufruf führt implizit zu einem PUSH der lokalen Variablen und der Signaturvariablen, und jede Rückkehr aus einem Prozedurablauf ist mit dem zugehörigen POP dieser Variablen verbunden. Deshalb äußert sich der Stack aus Bild 80 nur implizit im Schichtungsplan in Bild 86.

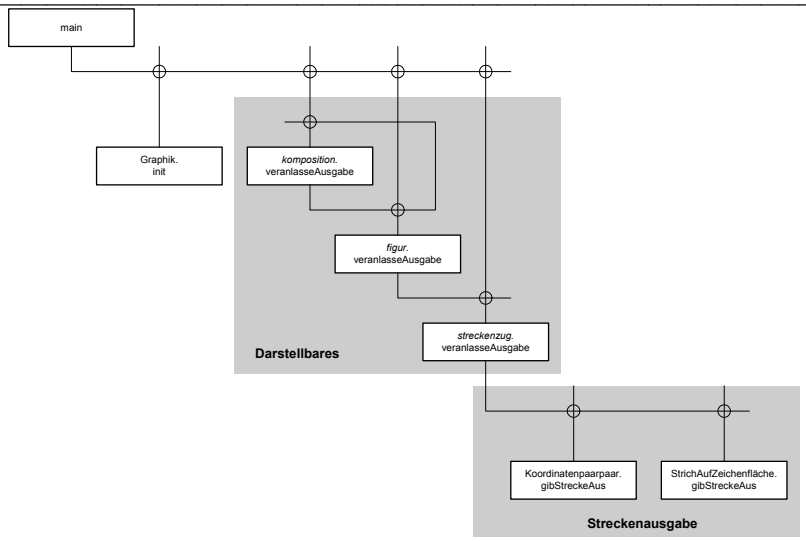

Bild 86 Schichtungsplan zur Ausgabe von Streckenzeichnungen

Vorhersehbare Änderungswünsche

Bei der Einführung der Steuerkreisstrukturierung und der Klassenstrukturierung von Programmsystemen habe ich gesagt, dass die Klassenstrukturierung den Vorteil der leichteren Änderbarkeit habe. Bei der Gestaltung eines Programms sollte man immer auch darüber nachdenken, welche Änderungsanforderungen in der Zukunft möglicherweise entstehen können. Wenn wir uns die Aufgabenstellung für unser Beispielsprogramm zur Darstellung von Zeichnungen aus Streckenzügen in Bild 69 anschauen, erkennen wir, dass dieses System in der gegebenen Form sicher kein langes Leben haben wird. Entweder wird das ganze System sehr bald weggeworfen oder es wird zu einem nützlicheren System erweitert. Wenn man über mögliche Erweiterungen nachdenkt, sollte man zweckmäßigerweise immer vom Aufbauplan ausgehen, der die Gesamtfunktionalität des aktuellen Systems möglichst anschaulich zeigt. In unserem Falle gehen wir also von Bild 69 aus. In Bild 88 sind nun Ergänzungen dargestellt, die zu der Struktur in Bild 69 hinzukommen können.

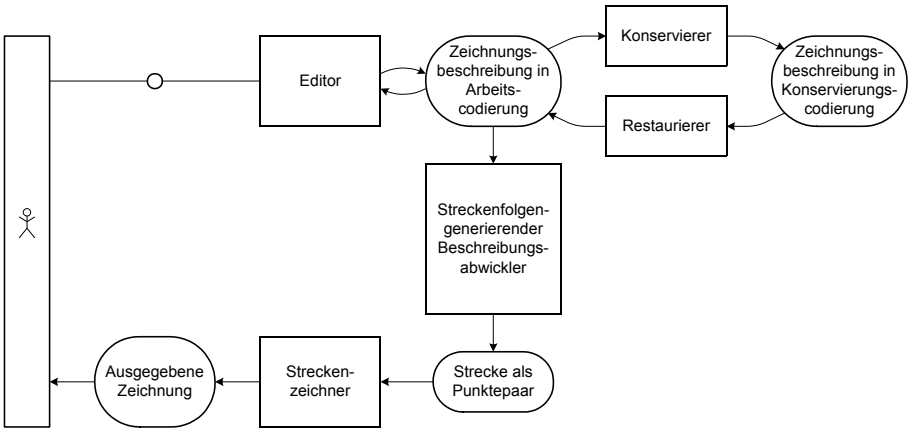

Bild 88 Erweiterungen des Aufbaus aus Bild 69

Von den in Bild 69 vorkommenden Systembestandteilen ist nur der Initiator ausgetauscht worden. An seiner Stelle steht nun ein Editor, der es dem Menschen erlaubt, interaktiv eine Zeichnungsbeschreibung zu erstellen. Zuvor wurde ja diese Zeichnungsbeschreibung als Konstante von der Klasse `Graphik` erzeugt. Außerdem gibt es in Bild 88 neben der Zeichnungsbeschreibung in Arbeitscodierung nun auch eine Zeichnungsbeschreibung in Konservierungscodierung. Unter einer Arbeitscodierung stellen wir uns eine Datenstruktur vor, wie sie in Bild 78 gezeigt ist. Auf einer solchen Struktur, die aus einer Menge durch Pointer verbundener Records besteht, kann sowohl der Editor als auch der Beschreibungsabwickler gut arbeiten. Eine solche Struktur belegt im Arbeitsspeicher unterschiedliche Bereiche. Anschaulich können wir uns in diesem Fall die Arbeitsspeicherbelegung vorstellen wie die Belegung einer großen Liegewiese im Strandbad. Es gibt viele freie Stellen und die Verteilung der belegten Stellen ist eher zufällig.
Ganz anders muss die Speicherbelegung aussehen, wenn wir die Zeichnungsbeschreibung auf einen Hintergrundspeicher, also beispielsweise auf eine Diskette oder ein Magnetband oder auf ein Dateisystem auf der Festplatte legen wollen. In diesem Fall muss es sich um eine ununterbrochene Folge von Bytes handeln. Wenn wir also konservieren wollen, brauchen wir einen Konservierer, der die Arbeitskodierung in die Konservierungscodierung überführt. Umgekehrt brauchen wir einen Restaurierer, wenn wir eine vorgegebene Konservierungscodierung wieder zu einer Arbeitscodierung machen wollen, damit wir an der Zeichnung weiter editieren können.

Im nächsten Abschnitt wird eine andere Änderung unseres ursprünglichen Systems betrachtet: Nun wird angenommen, dass die ursprüngliche Zeichnungsbeschreibung ein Text sei, der erst durch einen Übersetzer in die für die Zeichnungsdarstellung benötigte Arbeitscordierung transformiert wird.

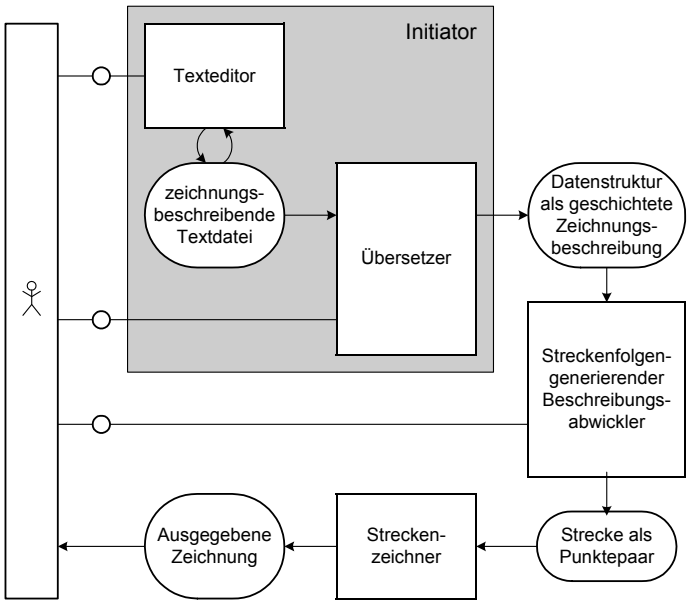

Bild 89 Systemaufbau

Der Übersetzer entspricht dem Restaurierer in Bild 88. Die Aufgabe der Übersetzung ergibt sich also nicht nur, wenn man den Text als Ergebnis einer Konservierung gewonnen hat.

Bei der Konservierung wird der Text aus der Datenstruktur erzeugt. Dies ist algorithmisch sehr viel einfacher als die Übersetzung und bedarf keiner besonderen Konzepte. Man erkennt dies leicht, indem man sich fragt, wie per Programm aus der Datenstruktur in Bild 78 der im folgenden Bild gezeigte Text gewonnenen werden kann.

```
< Graphik aus der 8. Aufgabe in Konservierungscodierung >

(   <  Streckenzuege (x Koord / y Koord)>

refdreieck  =( -1 / 0 )+( 1 / 0 )+( 0 / 1 )+( -1 / 0 )
refquadrat  =( 0 / 0 )+( 1 / 0 )+( 1 / 1 )+( 0 / 1 )+( 0 / 0 )

)(  <  Figuren  (  Zx  /  Zy  / sigma in Grad) >

dreieck       = refdreieck  ( 1  / 1.4142136 / 45 )
rechteck      = refquadrat  ( 6  / 8         / 0  )
parallelogramm= refquadrat  ( 10 / 5.6568542 / -45 )

)(  <  Kompositionen  (  Zx  /  Zy  /  delta in Grad/  Tx  /  Ty  ) >

dreiFiguren = dreieck      ( 1 / 4   / 0   / 1  / 0  )+
              rechteck     ( 1 / 1   / 0   / -2 / -2 )+
              parallelogramm ( 1 / 2 / 0   / 6  / 0  )

zweimalDreiFiguren mitRahmen = dreiFiguren ( 1 / 1   / 0   / 3  / 2  )+
                               dreiFiguren ( 0.5 / 0.5 / -90 / 13 / 4 )+
                               rechteck    ( 4 / 2.25 / 0   / 0  / -7 ))
```

Bild 90 Beschreibender Text für die geschichtete Graphik in Bild 77

Hier endet das Semester und damit die Vorlesung *Konzepte der Programmierung I*. Die Vorlesung *Konzepte der Programmierung II* setzt die Betrachtungen ohne Bruch fort.