Die Hofzwerge — Ein kurzes Tutorium zur objektorientierten Modellierung

Ernst-Erich Doberkat

1. Oktober 2000

Inhaltsverzeichnis

1 Einleitung

Informatiker modellieren die Welt. Nur auf der Grundlage aussagekräftiger Modelle ist es möglich, von der Anforderung eines Anwenders zu einem arbeitsfähigen Programm zu kommen, das die Anforderungen des Anwenders erfüllt. Daher ist die Modellierung der Anwendung ein wesentlicher Schritt in der Konstruktion eines Programms. Sie bedient sich hierbei formaler oder weniger formaler Hilfsmittel. In aller Regel wird eine praktikable Mischung formaler und semiformaler Methoden verwendet, bei steigenden Anforderungen an die Korrektheit des Systems findet sich ein zunehmender Anteil an formalen Modellierungsmethoden. Üblicherweise zerfallen Programmen in Daten und Operationen auf diesen Daten, daraus folgt, daß Daten und funktionale Anforderungen modelliert werden müssen. Daten beschreiben den eher statischen Teil eines Systems, Funktionen den eher dynamischen Anteil. In der objektorientierten Systemkonstruktion wird mit Konstrukten wie *abstrakten Datentypen* eine zusammenfassende Darstellung von Daten und Operationen darauf realisiert. Da die Daten statisch sind, es ist meist leichter, zunächst die Daten mit ihren Beziehungen zu modellieren und dann zur Funktionsmodellierung zu kommen. Modellierungsmethoden wie etwa *UML* (vgl. etwa [5]) verknüpfen beide Arten der Modellierung.
Ich möchte in diesem kurzen Überblick aufzeigen, wie sich an einer recht einfachen Problemstellung die Datenmodellierung systematisch realisieren läßt, und wie dann diese Modellierung, die in eine Klassifikationshierarchie mündet, in eine Klassenhierarchie in einer objektorientierten Programmiersprache (hier: C++) transformiert werden kann. Dies tue ich aus drei Gründen:

- Um zu zeigen, wie sich Klassifikationshierarchien (also reine Entwurfsnotationen) in Klassenhierarchien (also Vehikel zur Implementierung mit einer objektorientierten Progarammiersprache) transformieren lassen, insbesondere welche Probleme sich ergeben.
- Um zu zeigen, daß wichtige Tätigkeiten beim Systementwurf ohne programmiersprachlichen Ballast durchgeführt werden können. Das bedeutet insbesondere, daß modellierende Tätigkeiten bereits im Informatik-Unterricht eingeführt werden können, wenn der Grundvorrat der zu lehrenden Programmiersprache noch nicht oder nur rudimentär verfügbar ist. Es zeigt sich auch, daß auch nicht-technische Problembereiche zur Modellierung herangezogen werden können.
- Um zu zeigen, daß der objektorientierte Ansatz nicht nur in der Modellierung, sondern auch in der Implementierung Vorteile bringt, die sich direkt bei Generalisierung und Wartung sichtbar machen lassen.

Die Modellierung bezieht sich auf die zugrunde liegenden Daten, Algorithmen (also der dynamische Anteil) werden dadurch noch nicht gefaßt. Das ist eine Vorgehensweise, die sich möglicherweise nicht direkt anbietet, wenn die Dynamik des Geschehens im Vordergrund steht. Da viele Anwendungen jedoch eher datenzentriert sind, erscheint diese Vorgehensweise als durchaus betrachtenswert.
Wir gehen so vor: Im nächsten Abschnitt wird eine Klassifikationshierarchie entwickelt. Hierbei haben wir uns an die Besoldungshierarchie der Hofzwerge in der Wiener Hofburg gehalten, eine Population, die aus dem Sagenschatz des goldumkränzten Wien um die Jahrhundertwende stammt. Diese Klassifikationshierarchie wird ausgearbeitet und im Abschnitt über die Implementierung in ein C++-Programm übertragen. Wir nehmen an, daß der Leser über

die Kenntnis der notwendigen Sprachkonstrukte verfügt, sonst sei ihm ein elementares Lehrbuch zu C++, etwa [3] empfohlen. Der Implementierungsabschnitt enthält auch ein kleines Gedankenexperiment, in dem gezeigt wird, daß der objektorientierte Zugang mit seinen Möglichkeiten des dynamischen Bindens eine sehr flexible und änderungsfreundliche Vorgehensweise gestattet.
Die Wichtigkeit für die Modellierung in einem objektorientierten Zugang ist seit langem bekannt, insbesondere Ideen aus der Programmiersprache Beta (vgl. [7, 4]). Diese Ideen wurden in der objektorientierten Gemeinschaft schnell aufgenommen, wie wirksam sie sind, zeigt sich am Siegeslauf der graphischen Entwurfsnotation UML, vgl. [5]. Dieser kleiner Überblicksaufsatz soll dabei helfen, die objektorientierte Modellierung fester in der Programmiererausbildung im Informatik-Unterricht der Sekundarstufe II zu verankern.
Das Beispiel selbst hat auch eine Geschichte. Eine Klassifikationshierarchie für Laufbahnen und Gehälter taucht in [7, 7.5.2] auf. Dort heißen die Positionen noch `NonPermanentJobs` etc. In [4, 6.1.2] wird das Beispiel übernommen und in [2, 6.7] an Hofzwerge angepaßt. In [3] schließlich hilft es dabei, einen weiten Bogen von der Modellierung in Kapitel 1 zur Implementierung im dortigen Kapitel 16 zu schlagen. *Habent suaque fata nani.*

2 Die Hofzwerge in der Wiener Hofburg

Im alten Wien (nicht dem Wien des harten Aufklärers Josef II. oder des biedermeierlich reaktionären Meisters Metternich, nein, in dem Wien, das Gold umglänzt ist und über das uns Fritz von Herzmanowski-Orlando [8] solche bittersüßen und abstrusen Geschichten erzählt), im alten Wien also, war die Hofburg der Mittelpunkt des politischen und gesellschaftlichen Geschehens. Ganze Heerscharen von Bediensteten arbeiteten in der Hofburg, sichtbare und vielleicht auch unsichtbare, und unser Gewährsmann könnte von einer Klasse von Bediensteten berichten, den Hofzwergen, die zwar im Untergrund ihre Arbeit verrichteten, gleichwohl aber entlohnt werden mußten. Vielleicht ist die folgende Geschichte apokryph, sie kann uns aber doch als Einführung in die Möglichkeiten der objektorientierten Softwarekonstruktion dienen.
Die Hofzwerge, von denen hier die Rede sein wird, sind ein quirliges Volk, für viele Arbeiten zu gebrauchen, die offiziellen oder — am k.u.k.Hof — weniger offiziellen Charakter haben. Sie bilden keine homogene Schicht von Bediensteten. Vielmehr gibt es innerhalb der Hofzwerge vielfältige und feine Rangabstufungen, die von der Tätigkeit, aber auch von der Art der Position abhängen. Die Bezahlung, die wir vielleicht lieber *Remuneration* nennen sollten, da wir hier in Kakanien sind, hängt von der Stellung eines Hofzwergs in dieser Hierarchie ab. Bei einigen Hofzwergen ist das Salär fixiert, bei anderen Hofzwergen wiederum abhängig von den Stunden, die tatsächlich gearbeitet worden sind. Wir wollen uns die Hofzwerge unter diesen Gesichtspunkten einmal genauer ansehen. Das Ziel dieser Betrachtung besteht darin, mit ordnendem Sinn eine übersichtliche Klassifikation der Hofzwerge zu erarbeiten. Sie kann dann z. B. dazu benutzt werden, die Steuereinkünfte oder die Sozialabgaben für diese Klasse von Bediensteten auf, wie sich herausstellen wird,recht einfache Weise zu bestimmen.

2.1 Die Hierarchie der Hofzwerge

Da läuft uns Cyriakus von Pizziculli über den Weg. Er wird in den Akten als Hofzwerg erster Klasse geführt, denn schließlich ist er dafür zuständig, die adriatischen Winde in der Gegend von Venedig einzufangen und in das Lüfterl-Zimmer der Hofburg zu bringen, wo sich das

allerhöchstselbige Oberhaupt der Familie gern zum Mittagsschlaf ausruht. Der Hofzwerg hat, wie Sie sehen, eine überaus verantwortungsvolle Aufgabe, als Besoldung erhält er monatlich einen Festbetrag von fl 35, also 35 Gulden. Unglücklicherweise sind die Steuern in Kakanien recht hoch (Märchenglanz will schließlich aufrechterhalten werden: Das hat seinen Preis), die Steuer beträgt einheitlich an der Hofburg 45 v.H. von der Differenz zwischen dem Gehalt und dem Freibetrag. Als Hofzwerg erster Klasse hat von Pizziculli einen Grundfreibetrag von zehn Gulden, der Zusatzfreibetrag beträgt für ihn drei Gulden. Damit ist das steuerpflichtige Einkommen dieses Hofzwergs erster Klasse fixiert auf 22 Gulden.
In der Stille seiner Dienstkammer begegnen wir einem Exemplar der Klasse der Hofzwerge zweiter Klasse, Herrn Dr. Johann Nepomuk Hofzinser. Er ist, wie gesagt, ein Hofzwerg zweiter Klasse, promoviert gar, wir wissen aus den Archiven der Hofburg, daß er sich in seiner Freizeit gerne mit Zauberkunststücken beschäftigt. Dienstlich ist er hingegen dafür zuständig, die Schreibfedern des Sektionschefs der Hofburg stets schreibfähig zu halten, damit Erlasse und andere wichtige Schriftstücke jederzeit aufs Pünktlichste unterzeichnet werden können. Diese Tätigkeit ist nervenaufreibend, wie man sich leicht vorstellen kann, daher wird die Liebhaberei von Dr. Hofzinser auch verständlich. Die Besoldung ist der Verantwortung der Tätigkeit durchaus angemessen: das Grundgehalt für unseren guten Doktor beträgt fl 45, der Basisfreibetrag ist derselbe wie bei Herrn von Pizziculli, nämlich zehn Gulden. Als Hofzwerg zweiter Klasse hat er freilich einen erhöhten Zusatzfreibetrag, denn eine verantwortliche Tätigkeit bringt schließlich auch die Verpflichtung mit sich, verantwortungsbewußt auszusehen, und das kostet Geld. Mit dem Zusatzfreibetrag von vier Gulden berechnet sich das steuerpflichtige Einkommen von Dr. Hofzinser also als Differenz zwischen fl 45 und fl 14, mithin 31 Gulden.
Unsere beiden Exemplare, von Pizziculli und Dr. Hofzinser, gehören zu den festangestellten Hofzwergen, deren Gehalt unabhängig von den konkret laufenden Arbeiten fest ist. Etwas anders sieht es bei denjenigen Hofzwergen aus, die als Zeitarbeiter angestellt sind. Diese Zeitarbeiter werden stundenweise bezahlt, da sie Aufgaben erfüllen, die nur von Zeit zu Zeit anfallen. Betrachten wir als Beispiel den Kammerkalligraphen Johann Ptaschnik. Sein Sohn wird später Lehrer am Gymnasium der k.u.k. Theresianischen Akademie sein und einen Leitfaden zum Lesen geographischer Karten verfassen, er selbst ist als Kammerkalligraph dafür zuständig, kaiserliche Erlasse in Carrara-Marmor zu meißeln, um es auch noch den entferntesten Völkerschaften (z. B. auf der Kaiser-Franz-Josefs Insel) mitzuteilen. Herr Ptaschnik hat sich schon in frühester Jugend durch eine besonders hübsche Runenschrift ausgezeichnet, so daß sich eine Karriere als Kammerkalligraph fast zwingend ergeben hat. Nun ist es so, daß in Kakanien nicht wie bei uns ein steter Fluß von Gesetzen, Verordnungen und Erlassen auf die schicksalsergeben wartende Bevölkerung niedergegangen ist, vielmehr sind Erlasse und andere kaiserliche Anordnungen recht sparsam und eher selten veröffentlicht worden. Daher erweist es sich als vernünftig, einen Kammerkalligraphen lediglich stundenweise zu beschäftigen. Herr Ptaschnik verdient mit seiner wunderschönen Runenschrift zwei Gulden die Stunde, was auf die einzelne Rune umgerechnet eine fast fürstliche Besoldung ist und die Großzügigkeit der Hofverwaltung zeigt. Auch seine Steuer berechnet sich aus der Differenz von Gehalt und Freibetrag, wobei der Freibetrag sich wieder zusammensetzt aus einem Grundfreibetrag von zehn Gulden und einem Zusatzfreibetrag, der für alle Zeitarbeiter bei Hofe gleichmäßig auf drei Gulden festgesetzt ist. Natürlich wird festgehalten, daß bei Zeitarbeitern nur dann Steuern gezahlt werden müssen, wenn das steuerpflichtige Gehalt positiv ist. Wenn also Herr Ptaschnik weniger als 13 Gulden pro Monat verdient, so bleibt sein Einkommen steuerfrei.
Wenden wir uns schließlich der unabänderlichen Tatsache zu, daß an der Hofburg auch Be-

gräbnisse stattfinden. Sie werden dem Rang des oder der Verblichenen angemessen mehr oder minder pompös gefeiert. Da derartige Feierlichkeiten einen herausragenden und sehr spezialisierten Sachverstand erfordern, ist eine spezielle Berufsgruppe dafür zuständig, nämlich die Hofpompfünebristen. Ihre Aufgabe bestehtz darin, Begräbnisfeierlichkeiten auszurichten, den Staatsalmanach zu informieren und schließlich drei Wochen nach der Beerdigung in den Dienstzimmern des Verblichenen Weihrauch und Myrrhe zu verbrennen — eine Tätigkeit, die, wie Sie leicht erkennen können, von fast unerläßlicher Wichtigkeit (um nicht zu sagen: Relevanz) für das Funktionieren eines geordneten Hofwesens ist. Als typischen Hofpompfünebristen möchte ich Jeremias Käfermacher anführen, mit dessen Namen das tragische Schicksal einer unrechtmäßig für tot erklärten Gattin verknüpft ist, aber davon an anderem Ort! Uns geht es zunächst um die Einkünfte dieses Hofzwergs, der ebenfalls stundenweise bezahlt wird (denn Trauerfeierlichkeiten mußten zum Glück nicht so häufig ausgerichtet werden, daß ein festangestellter und sozusagen berufsmäßig Trauernder sich rentiert hätte). Jeremias Käfermacher also bezieht einen Stundenlohn von acht Gulden, die Freibeträge werden wie für Kammerkalligraphen berechnet.

2.2 Die Analyse

Wir haben damit einen kleinen Ausschnitt aus der recht komplexen Hierarchie der Hofzwerge geschildert. Man kann sich nun denken, daß der Zahlmeister, der für die Gehaltszahlungen zuständig ist, leicht graue Haare bekommt, wenn er für die ganze, vielfältige Schar der Hofzwerge die Gehälter berechnen soll. Daher wollen wir uns nun, wie angekündigt, an eine systematische Exploration dieser kleinen Welt machen.
Zunächst stellen wir fest, daß es offenbar zwei Klassen von Bediensteten gibt, nämlich die festangestellten Hofzwerge, die ein festes Gehalt haben, und die nicht permanent beschäftigten Zeitarbeiter bei Hofe, die einen Stundenlohn beziehen. Unabhängig von der Zugehörigkeit zu der einen oder anderen Klasse berechnet sich jedoch das steuerpflichtige Gehalt eines jeden dieser Bediensteten aus der Differenz zwischen dem individuell zu berechnenden Gehalt und dem ebenfalls individuell zu berechnenden Freibetrag. Der Freibetrag mag in seinen Einzelheiten durchaus auf verschiedene Arten zustande kommen, er ist jedoch gleichförmig zusammengesetzt als Summe aus einem Freibetrag, der für alle Bediensteten zehn Gulden beträgt, und einem Zusatzfreibetrag, der von der Art der Stelle abhängig ist.
Für das weitere Vorgehen erweist es sich als sinnvoll, Klassen von Bediensteten zu bilden (das haben wir implizit eigentlich schon getan), und diese Klassen zueinander in Beziehung zu setzen. Die Beziehung zwischen einzelnen Klassen wird sich als hierarchisch herausstellen, eine Eigenschaft, die bei der Umsetzung dieser Überlegungen im konkreten Code gleich sehr hilfreich sein werden. Aber eins nach dem anderen.
Als allgemeinste Klasse halten wir die Klasse aller Hofarbeiter fest. In dieser Klasse notieren wir alle Eigenschaften, die für alle Hofarbeiter gelten. Eine Eigenschaft wird also für die Klasse aller Hofarbeiter notiert, wenn sie sowohl für festangestellte Hofzwerge als auch für die zeitlich befristeten Angestellten in der Hofburg gilt. In der obigen Diskussion haben wir ja bereits notiert, wie die Berechnung des Gehalts und der Freibeträge vorgenommen wird, so daß wir also jetzt festhalten können:

Klasse aller Hofarbeiter: Die Steuer berechnet sich als 45 v. H. von der Differenz zwischen Gehalt und Freibetrag, der Basisfreibetrag beträgt zehn Gulden, der Freibetrag selbst ist die Summe aus dem Basisfreibetrag und dem Zusatzfreibetrag.

Wir haben gesehen, daß die Klasse der Hofarbeiter auf natürliche Weise in die Klasse der eigentlichen Hofzwerge und die Klasse der Zeitarbeiter bei Hofe zerfällt. Hierbei bedeutet *zerfällt*, daß jeder Hofarbeiter entweder ein Hofzwerg oder ein Zeitarbeiter bei Hofe ist, umgekehrt jeder Hofzwerg ein Hofarbeiter und jeder Zeitarbeiter bei Hofe ebenfalls ein Hofarbeiter ist. Also ist die Klasse der Hofarbeiter disjunkt in die beiden genannten Klassen zerlegt. Da jeder Hofzwerg nun ein Hofarbeiter ist, hat jeder Hofzwerg alle Eigenschaften eines Hofarbeiters, die wir oben gerade festgelegt haben. Dies bezieht sich auf die Steuerzahlung und auf die Art, wie Freibeträge sich als Summe aus Basis- und Zusatzfreibetrag zusammensetzen. In analoger Weise werden die Eigenschaften eines Hofarbeiters an Zeitarbeiter bei Hofe weitergegeben und möglicherweise weiter spezialisiert.
Halten wir aus der obigen Diskussion fest:

- Die Klasse der Hofzwerge ist eine Unterklasse der Klasse der Hofarbeiter, jeder Hofzwerg bezieht ein festes Gehalt.
- Die Klasse der Zeitarbeiter bei Hofe ist eine Unterklasse der Klasse der Hofarbeiter, jeder Zeitarbeiter bei Hofe bezieht einen Stundenlohn, der Zusatzfreibetrag beträgt drei Gulden.

Mit diesen Informationen können wir schon einen kleinen Klassifikationsbaum malen, der die Eigenschaften, die wir gerade herausgearbeitet haben, festhält. Wir zeichnen für jede Klasse ein Rechteck und schreiben die Eigenschaften der entsprechenden Klasse in dieses Rechteck. Dabei beachten wir die hierarchische Beziehung, die wir gerade herausgearbeitet haben, so daß wir zum Beispiel die Methode der Steuerberechnung nicht in jedes Rechteck schreiben, sondern nur in das oberste, das die alle umfassende Klasse angibt. Dies finden Sie in Bild 1.

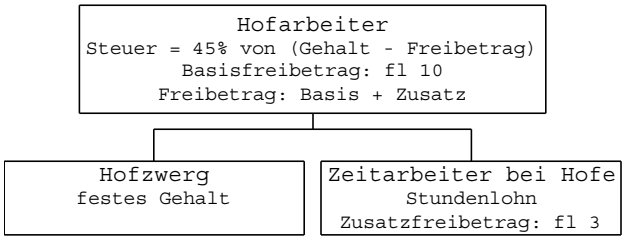

Abbildung 1: Eine erste Klassifikation

Ähnlich wie die Hofarbeiter in Hofzwerge und Zeitarbeiter bei Hofe zerfallen, bemerken wir eine Einteilung der Hofzwerge in solche erster und solche zweiter Klasse. Völlig analog zu den oben festgestellten Eigenschaften ist ein Hofzwerg entweder ein Hofzwerg erster Klasse oder ein Hofzwerg zweiter Klasse, so daß wieder eine disjunkte Zerlegung vorliegt. Hofzwerge erster und zweiter Klasse haben alle Eigenschaften von Hofzwergen (und damit auch alle Eigenschaften von Hofarbeitern), sie haben einige spezifische Eigenschaften, die wir in der folgenden Aufstellung noch einmal zusammenfassen:

Klasse der Hofzwerge erster Klasse: Dies ist eine Unterklasse der Klasse aller Hofzwerge, das Gehalt beträgt 35 Gulden, der Zusatzfreibetrag beträgt drei Gulden.

Klasse der Hofzwerge zweiter Klasse: Dies ist eine Unterklasse der Klasse aller Hofzwerge, Gehalt 45 Gulden, Zusatzfreibetrag vier Gulden.

Damit können wir unsere partielle Klassifikationshierarchie fortsetzen, wie Sie in Bild 2 sehen können.

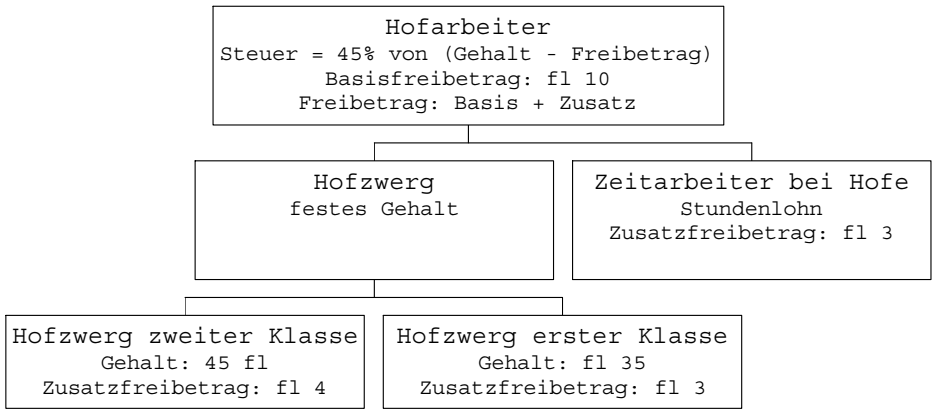

Abbildung 2: Erweiterung der Klassifikationshierarchie

Kommen wir zu den Zeitarbeitern bei Hofe, so finden wir *cum grano salis* ebenfalls eine disjunkte Zerlegung in Kammerkalligraphen und Hofpompfünebristen. Daher können wir für die Klasseneinteilung festhalten:

Klasse der Kammerkalligraphen: Dies ist eine Unterklasse der Zeitarbeiter bei Hofe mit einem Stundenlohn von zwei Gulden.

Klasse der Hofpompfünebristen: Dies ist eine Unterklasse der Zeitarbeiter bei Hofe mit einem Stundenlohn von acht Gulden.

Mit diesen Angaben können wir unsere Hierarchie fertigzeichnen, es ergibt sich die baumförmige Struktur aus 3.

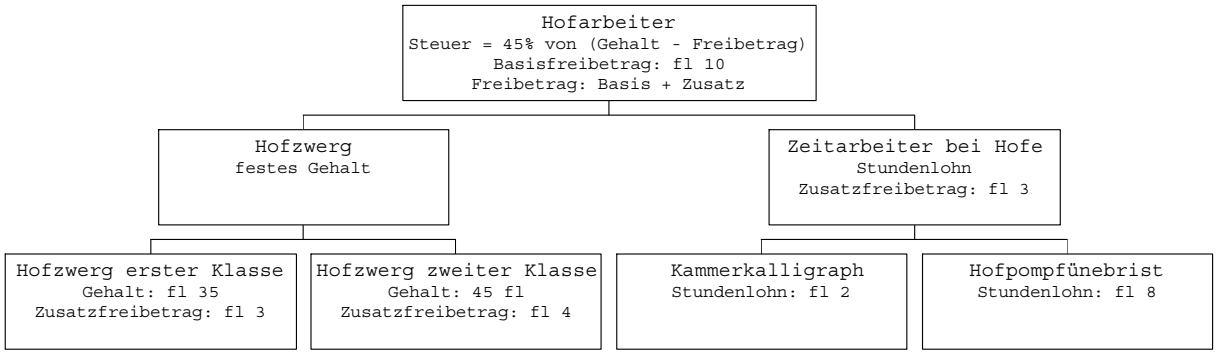

Abbildung 3: Endgültigere Form der Klassifikationshierarchie

Wenn wir diesen Baum noch einmal diskutieren, so stellen wir fest, daß wir vom Allgemeinen zum Speziellen gegangen sind, wobei wir durchaus auch Zwischenschritte eingelegt haben. Die Einteilung ist jeweils von Stufe zu Stufe so angelegt, daß sich disjunkte Zerlegungen ergeben.

Man könnte nun diese gesamte Einteilung auch flächig als Venn-Diagramm darstellen, die gewählte Darstellung hat aber den Vorteil, daß sie recht übersichtlich ist.
Eine wichtige Eigenschaft, die hier festgehalten werden soll, ist die Propagierung von Eigenschaften von oben nach unten: eine Klasse hat zusätzlich zu den explizit aufgeschriebenen Eigenschaften noch alle Eigenschaften der direkt übergeordneten Klasse. Für die weitere Diskussion wird sich als nützlich erweisen, daß die Blätter, also diejenigen Knoten in dieser Hierarchie, die keine weiteren Unterknoten haben, real existierende Hofzwerge beschreiben, während alle Knoten, die keine Blätter sind, lediglich dazu dienen, die Beschreibung der Blätter vorzubereiten. Wir sprechen von *abstrakten Klassen*, als solchen Klassen, denen keine Objekte im täglichen (oder fiktiven) Leben entsprechen.

3 Die Implementierung

Stellen Sie sich vor, der Zahlmeister in der Hofburg möchte gerne wissen, welche Gehaltszahlungen fällig werden, welche Steuern zu zahlen sind, welche Freibeträge er ausbringen muß und vielleicht auch noch ein wenig mehr.
Wir sollten also, um dem Zahlmeister zu helfen, die Klasse der Hofzwerge in einem Programm realisieren. Hierbei hilft uns die Klassifikationshierarchie auf sytematische Art und Weise. Uns wird auch helfen, daß wir über die methodischen Instrumente der objektorirntierten Programmierung wie dynamisches Binden und rein virtuelle Methoden, Vererbung und Ähnliches verfügen.Die *Klassenhierarchie*, die wir gerade entwickelt haben, soll in eine *Klassenhierarchie* umgewandelt werden. Einige Aspekte dieser Transformation gelingen, wie Sie sehen werden, recht mechanisch. Es zeigt sich jedoch, daß wir diesen Prozeß der Transformation nicht vollständig mechanisch lösen können, weil wir einige Probleme doch auf der Grundlage unseres Verständnisses lösen müssen.
Wir werden uns also jetzt mit der Implementierung der Klassenhierarchie befassen und den Zahlmeister der Hofburg mit Hilfe eines Programms unterstützen. Wir werden dabei sehen, daß ein strikt prozeduraler Zugang, also ein solcher Zugang, der keine Vererbung kennt, wesentlich umständlicher ist. Um das zu verdeutlichen, möchte ich Ihnen zeigen, wie wir bei einer Erweiterung der Problemstellung auch die Klassenhierarchie auf einfache Weise erweitern können. Das läßt sich wesentlich unkomplizierter bewerkstelligen, als dies mit prozeduralen Hilfsmitteln möglich gewesen wäre.

3.1 Zum Programmentwurf

Wir haben die ohne Zweifel faszinierende Welt der Hofzwerge an der Stelle verlassen, an der es uns gelungen ist, die Klassifikationshierarchie zu formulieren. Hierbei haben wir alle Informationen verwendet, die uns zur Verfügung gestanden haben. Beim Programmentwurf sollte diese Klassifikationshierarchie berücksichtigt werden. Wenn wir sie in ein Programm übersetzen wollen, so halten wir zunächst fest, daß die Vererbung dieser Hierarchie folgt. Aus der *Vater/Sohn*-Beziehung in der Klassifikationshierarchie wird eine *vererbt/erbt*-Relation in der Klassenhierarchie.
Technisch ist zu vermerken, daß die in der Hierarchie angegebenen Attribute der einzelnen Knoten in Attribute für die zugehörigen Klassen übersetzt werden. Weiterhin ist auch klar, daß Attribute und Methoden so hoch wie möglich im Baum angebracht werden sollten, um einen möglichst hohen Wirkungsgrad zu erzielen.

Was heißt das? Je höher eine Methode im Baum angebracht wird, je näher also der entsprechende Knoten an der Wurzel ist, desto mehr Knoten (also Klassen) gibt es, in denen diese Methode sichtbar ist. In der Tat: Sie ist in allen Knoten verfügbar, die im Unterbaum zu diesem Knoten hängen. Je höher also eine Methode angebracht ist, desto mehr Klassen deckt sie ab. Dies gilt genauso für Attribute, ist also durchaus im Sinne eines möglichst allgemeinen Entwurfs.

Zusammenhang Der Zusammenhang zwischen der Klassifikationshierarchie und der Klassenhierarchie ist hier wichtig. Die Klassenhierarchie arbeitet auf der Ebene der Implementierung, die Klassifikationshierarchie hingegen auf der Ebene des Entwurfs. Die Klassifikationshierarchie bereitet die Klassenhierarchie vor. Idealerweise ist die Klassenhierarchie nicht nur eindeutig durch die Klassifikationshierarchie bestimmt (dies ist im wesentlichen bei uns der Fall), man kann vielmehr auch aus der Klassifikationshierarchie den Code ableiten. Ich habe das oben schon angedeutet, auch, daß das nicht in jedem Fall mechanisch möglich sein wird. Festzuhalten ist, daß ein solider Entwurf durch eine Klassifikationshierarchie unterstützt werden kann und das Argumentieren über das entstandene Programm wesentlich unterstüzt.

3.1.1 Zur Rolle rein virtueller Methoden

Bei rein virtuellen, also abstrakten Methoden wird sichtbar, daß Klassifikations- und Klassenhierarchie einander nicht eindeutig entsprechen, also nicht isomorph sind. Das liegt daran, daß wir über die Abstraktheit einer Methode nicht unbedingt in der Klassifikationshierarchie Auskunft geben. Es soll kurz rekapituliert werden, wo derartige abstrakte Methoden sinnvollerweise eingeführt werden sollten.
Wir können drei einfache Kriterien dafür angeben: rein virtuelle Methoden werden da eingeführt, wo

- der Name bereits bekannt sein sollte;
- der Name schon verwendet werden kann;
- der Code, also die Implementierung, noch nicht angegeben werden kann oder soll.

Gelegentlich möchte man bekanntlich einen Algorithmus an einen Methodennamen binden, um ihn (oder ihren Namen) bereits verwenden zu können, gibt aber die Implementierung für die Methode selbst noch nicht an. Das kann daran liegen, daß man die Realisierung an dieser Stelle verbergen möchte, oder daran, daß der Code sehr spezifisch von einer der abgeleiteten Klassen abhängt. Wir werden im folgenden einige Beispiele dafür studieren können.
Die Frage stellt sich, warum man den Namen an der Stelle überhaupt schon verwendet, warum man also nicht Implementierung und Vereinbarung zusammen in der entsprechenden Klasse realisiert. Diese Vorgehensweise hätte den — gelegentlich gravierenden — Nachteil, daß dann der Name der Methode noch nicht verwendet werden könnte. Die Verwendung des Namens wird hier jedoch möglich, wenn man diese Methode als rein virtuell definiert.

3.1.2 Zugriffsspezifikationen

Ein weiterer Gesichtspunkt, der in der Klassifikationshierarchie nur ungenügend berücksichtigt werden kann, ist durch die Verwendung von Zugriffsspezifikationen angedeutet. Sie werden

in der Hierarchie nicht oder nicht recht angemessen dargestellt, so daß der Code hierfür zusätzlich Auskunft geben muß. Führt man einen Entwurf vollständig durch, so wird man in aller Regel die Klassifikationshierarchie durch schriftliche Dokumente begleiten. Sie beschreiben den Entwurf näher. An dieser Stelle sollte natürlich klargestellt werden, welche Zugriffsspezifikationen für Attribute und Methoden intendiert sind.
Gegebenenfalls hilft auch der sogenannte *gesunde Menschenverstand*.

3.2 Die Klasse `Hofarbeiter` als Wurzel

Nach diesen Überlegungen können wir die Klasse `Hofarbeiter` als Wurzel der Klassenhierarchie angeben, vgl. Implementierung 3.1. Sie finden dort als privates Attribut den Namen des entsprechenden Hofarbeiters, als geschütztes und statisches Attribut die Konstante, mit deren Hilfe der Basisfreibetrag angegeben wird. Der Zusatzfreibetrag wird auch als `protected` vereinbart. Sie sehen, daß die Methode `Gehalt` ebenso wie die Methode `DerZusatzfreibetrag` als rein virtuelle Funktionen angegeben sind. Daraus folgt zum einen, daß die Klasse `Hofarbeiter` eine abstrakte Klasse ist, also nicht dazu herangezogen werden kann, Instanzen zu bilden. Weiterhin folgt daraus, unserer obigen Diskussion folgend, daß wir die entsprechenden Funktionen bereits benutzen können.

Implementierung 3.1 Vereinbarung der Klasse `Hofarbeiter`

```
class Hofarbeiter {
  private:
     char * Name;
  protected:
     static const int BasisFreibetrag;
     virtual int DerZusatzfreibetrag() = 0;
  public:
     Hofarbeiter(char *);
     virtual int Gehalt() = 0;
     int Freibetrag();
     int Steuer();
     char * DerName();
};
```

Kommen wir zur Diskussion einzelner Vereinbarungen: Die Konstante `BasisFreibetrag` ist für alle Objekte notwendig, wie ein Blick auf die Klassifikationshierarchie lehrt. Die Wertzuweisung wird wie folgt vorgenommen:

```
const int Hofarbeiter::BasisFreibetrag = 10;
```

Die Konstante zum Basisfreibetrag außerhalb der Klasse definiert. Ein Blick auf die Vereinbarung der Klasse lehrt zudem, daß diese Konstante als `static` vereinbart wird.
Der Konstruktor für die Klasse `Hofarbeiter` ist ziemlich einfach (Implementierung 3.2), ist also nicht besonders aufregend. Wir übergeben eine Zeichenkette, die dann als Name des Hofarbeiters dienen soll. Die üblichen Techniken (Feststellen der Länge der Zeichenkette mit `strlen`, Allokation eines geeigneten Felds mit `new` und schließlich Kopie der Zeichenkette mit `strcpy`) sind in der Tat kanonisch und sollen nicht weiter kommentiert werden.

Implementierung 3.2 Konstruktor für die Klasse `Hofarbeiter`

```
Hofarbeiter::Hofarbeiter(char * t) {
             void strcpy(char *, char *);
             int strlen(char *);
             Name = new char[strlen(t)];
             strcpy(Name, t);
}
```

Über die Konsequenzen der reinen Virtualität für die Methode `Gehalt` haben wir ja bereits gesprochen. Wir haben noch nachzutragen, warum diese Methode als rein virtuell definiert worden ist: Dies liegt schlicht daran, daß es in unserer Welt der Hofzwerge keine einheitliche Möglichkeit gibt, die Berechnung der Gehaltszahlung festzulegen. Die Berechnung des Gehalts kann an dieser Stelle noch nicht formuliert werden, sie muß *klassenspezifisch* nachgetragen werden.
Als weitere Methode ist die Ausgabe des Namens zu nennen. Denken Sie daran, daß das Attribut Name als `private` vereinbart ist:

```
char * Hofarbeiter::DerName() {return Name;}
```

Die Berechnung des Freibetrags sollte nach unserer Aufgabenstellung so durchgeführt werden, daß die Summe aus dem Basisfreibetrag und dem Zusatzfreibetrag berechnet wird. Gegenwärtig ist der Code für die Methode zur Berechnung des Zusatzfreibetrags noch nicht bekannt, der Name für die Berechnung steht aber bereits zur Verfügung. Daher können wir die Berechnung des Freibetrags so durchführen:

```
int Hofarbeiter::Freibetrag() {
    return BasisFreibetrag + DerZusatzfreibetrag();
}
```

Wenn Sie die bisherige Entwicklung rekapitulieren, so sind virtuelle Methoden eingeführt worden, um dynamisches Binden des Methodennamens an die entsprechende Implementierung zu ermöglichen. Diese Vorgehensweise wird durch Verwendung rein virtueller Methoden noch einmal deutlich betont.

3.3 Die nächste Ebene

Nachdem wir also eine Entsprechung zwischen der Wurzel des Klassifikationsbaums und des Klassenbaums hergestellt haben, sollten wir uns um die nächste Ebene in beiden Bäumen kümmern. Die Verhältnisse bei den Hofzwergen sind nicht besonders kompliziert, es wird freilich ein wenig umständlicher, wenn wir über die Klasse der Zeitarbeiter reden.
Wir beginnen mit der Realisierung der Klasse der Hofzwerge.

3.3.1 Die Klasse der Hofzwerge

Diese Klassendefinition ist ziemlich einfach: Wir sagen, daß jeder Hofzwerg ein Hofarbeiter ist (dies entspricht haargenau der Klassifikationshierarchie). Allerdings müssen wir den Regeln

der Sprache zufolge einen Konstruktor definieren, und diesen Konstruktor wollen wir ein wenig näher unter die Lupe nehmen (auch wenn ähnliche Konstruktionen bereits durchgeführt worden sind). Der Konstruktor wird bereits in der Vereinbarung der Klasse implementiert, er ruft nämlich den Konstruktor für die Vaterklasse auf und tut — wie wir am leeren Block für den Rumpf sehen — sonst nichts.
Die Hofzwerge erster Klasse stellen keine abstrakte Klasse dar, weil wir diese Klasse ja instanziieren, also Objekte für diese Klasse herstellen können. Technisch liegen alle Angaben zur Berechnung des Gehalts und der Steuer vor, so daß eigentlich kein Grund mehr dafür vorhanden ist, weiter mit abstrakten Methoden zu arbeiten. Daher müssen wir all die virtuellen Methoden implementieren, die in den Klassen, von denen wir erben, noch nicht realisiert worden sind. Insgesamt führt dies zu der in Implementierung 3.3 abgegebenen Definition für Hofzwerge erster Klasse.

Implementierung 3.3 Vereinbarung der Hofzwerge Erster Klasse

```
class HofzwergErsterKlasse : public Hofzwerg {
  private:
       static const int FestGehalt;
       static const int Zusatzfreibetrag;
  public:
        HofzwergErsterKlasse(char * t): Hofzwerg(t) { }
        int Gehalt() { return FestGehalt; }
        int DerZusatzfreibetrag() {
               return Zusatzfreibetrag;
        }
};
```

Die Klassendefinition zeigt, daß wir von der Klasse `Hofzwerg` erben. Es ist vielleicht ganz hilfreich, wenn Sie Ihre Aufmerksamkeit auf den Konstruktor und auf die Methoden richten, die sämtlich als `inline`-Vereinbarungen angegeben sind.
Für die erstklassigen Hofzwerge bleiben die Werte von Konstanten zu definieren. Das geschieht hier:

```
const int HofzwergErsterKlasse::FestGehalt = 35;
const int HofzwergErsterKlasse::Zusatzfreibetrag = 3;
```

3.3.2 Die Klasse `ZeitArbeiter`

Die Realisierung der Klasse für die Zeitarbeiter ist ein wenig umständlicher, als dies bei den erst- oder zweitklassigen Zwergen der Fall gewesen ist. Das liegt daran, daß mehr Details zu berücksichtigen sind, die durch eine überlegte Mischung zwischen Konstanten und rein virtuellen Funktionen realisiert werden können. Die Einzelheiten sind hierdoch nicht weiter überraschend. Die rein virtuelle Methode `DerZusatfreibetrag` kann schon hier implementiert werden, da alle Angaben vorhanden sind. Wir führen allerdings eine rein virtuelle Methode `DerStundenlohn` ein. Das Ergebnis dieser Methode soll den Stundenlohn für die jeweilige Klasse angeben. Wir tun dies nach der jetzt sattsam bekannten Realisierung, daß wir die Angaben zum Stundenlohn hier schon verwenden können, ohne daß der Wert bekannt

sein muß. Diese Realisierung spricht auch dafür, die auf der Hand liegende Überlegung durch Konstanten zu verwerfen. Die Klasse `ZeitArbeiter` läßt sich nun einfach realisieren (vgl. Code in Implementierung 3.4).

Implementierung 3.4 Vereinbarung der Klasse `ZeitArbeiter`

```
class ZeitArbeiter: public Hofarbeiter {
  private:
       static const int ZusatzFreibetrag;
  protected:
       int StundenZahl;
       int DerZusatzfreibetrag() {return ZusatzFreibetrag;}
  public:
       ZeitArbeiter(char * t):Hofarbeiter(t) {}
       virtual int DerStundenlohn() = 0;
       int Gehalt() {return DerStundenlohn() * DieStundenZahl();}
       void SetzeStunden(int v)  {StundenZahl = v; }
       int DieStundenZahl() {return StundenZahl;}
};
```

Der Vollständigkeit halber sei die Definition der einzigen Konstanten angegeben. Sie bezieht sich auf die Klasse `ZeitArbeiter`.

```
const int ZeitArbeiter::ZusatzFreibetrag = 4;
```

Der Rest der Implementierung ist ziemlich nahe liegend.
Als erste Spezialisierung dieser Klasse der Zeitarbeiter betrachten wir die Kammerkalligraphen. Diese Klasse ist nicht abstrakt, wir müssen die bislang virtuell gebliebenen Methoden und die spezifischen Konstanten definieren. Das haben wir bei Hofzwergen auch so gemacht. Die Vorgehensweise läßt sich auch hier gut an der entsprechenden Stelle in der Klassifikationshierarchie studieren, so daß wir hierauf nicht näher eingehen müssen. Die Klasse ist wie folgt definiert:

```
class Kammerkalligraph: public ZeitArbeiter {
  private:
       static const int Stundenlohn;
  public:
       Kammerkalligraph(char *, int);
       int DerStundenlohn() {return Stundenlohn;}
};
```

Dabei ist zum einen der Konstruktor zu formulieren

```
Kammerkalligraph::Kammerkalligraph(char * t, int w):
                                                ZeitArbeiter(t) {
       StundenZahl = w;
}
```

und wir sollten uns zum anderen über den Wert der Konstanten `Stundenlohn` Gedanken machen, der nach der Formulierung des Problems wie folgt gesetzt werden kann:

```
const int Kammerkalligraph::Stundenlohn = 2
```

Wir haben die Hofzwerge zweiter Klasse und die Hofpompfünebristen jetzt nicht an dieser Stelle realisiert. Die Formulierungen sind völlig analog zu den hier angegebenen Definitionen, sie werden daher an dieser Stelle nicht angegeben. Freilich ist die geneigte Leserin dazu eingeladen, diese Klasse zu implementieren.

3.4 Was haben wir denn jetzt daraus gelernt?

Fassen wir kurz zusammen, was wir in dieser Diskussion der Hofzwerge gelernt haben: Wir haben auf der Grundlage der Problembeschreibung eine Klassifikationshierarchie konstruiert, wobei wir die uns zur Verfügung stehenden Daten und das Verhalten, das später in Methoden umgesetzt wird, zugrundelegen konnten.
Dies ist ein Entwurfsschritt, der in jedem größeren Projekt durchgeführt werden muß. Meist ist es bei einer umfangreicheren Aufgabenstellung erforderlich, eine separate Phase, die *Anforderungsanalyse* vor diese Phase des Entwurfs der Klassifikationshierarchie zu stellen, um die Anforderungen an das zu konstruierende System exakt zu fassen. Das ist hier nicht nötig gewesen, weil die Aufgabenstellung schon so formuliert werden konnte, daß sie die Anforderungen sozusagen gleich mitformuliert hat.
Im Laufe der Diskussion haben wir recht systematisch aus der Klassifikationshierarchie eine Klassenhierarchie gemacht, wobei jedem Knoten im Klassifikationsbaum eine Klasse entsprochen hat, und die Baumstruktur in eine Vererbungsstruktur übertragen worden ist. Die Übersetzung hat sich nicht als voll mechanisierbar erwiesen, weil wir bei Attributen und Methoden unter anderem die Zugriffsspezifikationen nicht in die Klassifikationshierarchie aufnehmen konnten und weil wir keine Aussage über die Abstraktheit von Methoden gemacht haben.
In dieser Diskussion ist auch deutlich geworden, daß die Klassifikationshierarchie, wie wir sie betrachtet haben, lediglich die statischen Aspekte der Modellierung erfaßt. Über die dynamischen Aspekte gibt sie keine Auskunft; sie werden meist separat modelliert.

3.5 Der Zahlmeister der Hofburg

Wir haben oben gesagt, daß wir dem Zahlmeister der Hofburg helfen wollen, wenn er gern wissen möchte, was monatlich an Gehältern, Steuern und Freibeträgen zusammenkommt. Die Idee bei der Ermittlung dieser Daten ist klar. Man faßt die Bediensteten der Hofburg in einer verketteten Liste zusammen, iteriert über diese verkettete Liste und gewinnt so die erwünschten Daten. Spätestens an dieser Stelle wird sich die Flexibilität des verwendeten objektorientierten Zugangs erweisen.

3.5.1 Eine Alternative?

Lehnen wir uns einen Augenblick zurück und überlegen, was wir zu tun hätten, wenn wir nicht die Möglichkeit gehabt hätten, Klassen durch Vererbung zu konstruieren. Wir hätten andere Wege finden müssen, um die entsprechenden Stellenbeschreibungen umzusetzen. Es böte sich in einer solchen Situation an, eine `struct` zu konstruieren und die Stellenbeschreibungen

jeweils mit einer Kennung zu kodieren. Unser Zahlmeister würde über die Liste der Instanzen dieser `struct` iterieren und seine Daten anhand der Kennung zu extrahieren versuchen. Das ist der gängige Weg, gegen den wenig einzuwenden ist, außer daß der hier vorgeschlagene objektorientierte Weg ohne Kennung arbeitet (die konkrete Art jeder Stelle, also das manifeste *So-Sein* des Hofzwergs wird zur Laufzeit festgestellt). Es wird sich dann zeigen, daß bei einer Erweiterung der Stellenhierarchie eine Änderung der Iteration über die Liste überhaupt nicht notwendig ist!
Das steht im krassen Gegensatz dazu, daß wir im alternativen Fall erhebliche Vorkehrungen treffen müssen, um auch wirklich alle Stellen im Code zu bedenken, an denen wir Änderungen durchführen müßten. Das gilt insbesondere dann, wenn wir neue Stellentypen hinzufügen. Nun mag Ihnen das Problem der Übersichtlichkeit an dieser Stelle nicht so gravierend erscheinen. Man soll jedoch von komplexeren Klassifikationshierarchien gehört haben, die mehr als sieben Knoten umfassen. Dann, so können Sie sich vorstellen, greift das Argument der Übersichtlichkeit nachhaltiger.

3.5.2 Konstruktion der Liste

Wir werden eine Liste aus den entsprechenden Bediensteten konstruieren. Dazu konstruieren wir eine Klasse, die — Sie werden es kaum glauben — den Namen `HofburgListe` erhalten wird. Die Operationen auf dieser Liste sind weitgehend kanonisch. Daher können wir diese Diskussion auchin der gebotenen Kürze durchführen.
Zunächst überlegen wir, welche Attribute diese neue Klasse haben sollte. Wir kommen auf die folgenden Attribute:

```
Hofarbeiter  * Angestellter;
HofburgListe * weiter;
HofburgListe * Kopf;
HofburgListe * AktuellesElement;
```

Die Attribute `Kopf` und `AktuellesElement` dienen dazu, den Anfang der Liste zu verzeichnen, beziehungsweise über die Liste zu iterieren.
Als Methoden sehen wir die folgenden vor:

- Einfügen eines Elements in die Liste (was in der *realen* Welt dem Engagieren eines Hofarbeiters entspricht);
- Starten der Iteration über die Liste;
- Inspektion des aktuellen Elements mit Extraktion der gesuchten Werte;
- Weiterschalten in der Liste, bis das Ende erreicht ist.

Die Klasse `HofburgListe` ist in Implementierung 3.5 dargestellt. Ich möchte nun nicht alle Methoden für diese Klasse im Detail diskutieren, vielmehr einige Hinweise zu ihrer Implementierung geben:

- Der Konstruktor initialisiert alle Zeiger zu `NULL`.
- Die Methode `StartIteration` arbeitet so, daß das Attribut `AktuellesElement` auf `Kopf` gesetzt wird.

Implementierung 3.5 Vereinbarung der Klasse `HofburgListe`

```
class HofburgListe {
  private:
       //Attribute
  public:
       HofburgListe() {
                  // inline
                  };
       void Engagieren(Hofarbeiter *);
       void StartIteration() {/* inline */}
       Hofarbeiter *  DieserAngestellte() {
            // inline
            }
       int WeiterSchalten();
};
```

- Die Methode `DieserAngestellte`, die zur Inspektion des gegenwärtig aktuellen Elementes dient, gibt `AktuellesElement->Angestellter` als Wert zurück.

Das Engagieren eines Hofarbeiters wird durch die gleichnamige Methode realisiert, vgl. Implementierung 3.6. Es handelt sich hier um das Einfügen an den Anfang einer verketteten Liste, wie man es ja gründlich bei der Einführung von Listen übt. kennengelernt haben.

Implementierung 3.6 Engagieren eines Hofarbeiters

```
void HofburgListe::Engagieren(Hofarbeiter * wen) {
     HofburgListe * HilfsKopf = new HofburgListe();
     HilfsKopf->Angestellter = wen;
     HilfsKopf->weiter = Kopf;
     Kopf = HilfsKopf;
  }
```

Die Methode `Weiterschalten` dient zur Mitteilung, ob die Liste bereits zu Ende durchlaufen worden ist oder nicht. In dem Fall, daß wir am Ende der Liste angelangt sind, gibt die Methode den Wert `-1` aus, sonst `0`, wie Implementierung 3.7 zeigt.

3.5.3 Erzeugung der Liste

Unglücklicherweise können wir uns der Gehaltslisten der Hofburg nicht mehr bedienen, weil die k.u.k.-Monarchie seit mehr als achtzig Jahren von der Bildfläche verschwunden ist. Die Frage stellt sich, wie die Liste denn nun entstehen kann, und hier greifen wir auf einen Zugang zurück, der für *Simulationen* üblich ist. Wir erzeugen zufällige Zahlen, schauen uns jede der erzeugten Zahlen genauer an und konstruieren daraus ein Element der Liste.
Die Erzeugung von Zufallszahlen ist fast eine schwarze Kunst: Man muß hier bei einer Folge von ganzen Zahlen erzeugen, die zufällig aussehen, bei denen also keine wie auch immer geartete Gesetzmäßigkeit erkennbar ist. Die Diskussion von Algorithmen zur Erzeugung solcher

Implementierung 3.7 Iteration über die `HofburgListe`

```
int HofburgListe::WeiterSchalten() {
    if (AktuellesElement->weiter == NULL)
       return -1;
    else  {
       AktuellesElement = AktuellesElement->weiter;
       return 0;
  }
}
```

Zahlen ist recht tiefliegend und soll hier nicht geführt werden. Wenn Sie's interessiert, ist [6] eine erstklassige Referenz.
Uns kommt es lediglich darauf an, daß die Sprache einen Zufallszahlengenerator `rand` zur Verfügung stellt, eine Funktion mit der Signatur `int rand()`. Jeder Aufruf dieser Funktion erzeugt eine neue, zufällig aussehende Zahl. Wir berechnen den Divisionsrest bei der Division dieser Zahl durch 4 und nehmen diesen Divisionsrest zum Anlaß, einen entsprechenden Hofzwerg zu erzeugen und in unsere Liste einzufügen. Dies ist in der Funktion `Erzeugen` näher beschrieben, die Sie in der Implementierung 3.8 finden können.

Implementierung 3.8 Simulierte Erzeugung der Liste aller Bediensteten

```
HofburgListe * Erzeugen(int j) {
  HofburgListe * dieListe = new HofburgListe();
  for (int t = 0; t < j; t++) {
       int zufall = rand();
       char * nme = new char (5);
       int KammerStunden = zufall % 82;
       int HofpompStunden = zufall % 44;
       Hofarbeiter * einArbeiter;
       itoa(zufall, nme, 10);
       int sw = zufall % 4;
       if (sw == 0)
              einArbeiter  = new HofzwergErsterKlasse(nme);
       else if (sw == 1)
              einArbeiter  = new HofzwergZweiterKlasse(nme);
       else if (sw == 2)
              einArbeiter  = new Kammerkalligraph(nme, KammerStunden);
       else if (sw == 3)
              einArbeiter  = new Hofpompfuenebrer(nme, HofpompStunden);
       dieListe->Engagieren(einArbeiter);
       return dieListe;
    }
}
```

Da wir für jeden Bediensteten einen Namen brauchen, nehmen wir die erzeugte Zahl noch

einmal her und verwandeln sie in eine Zeichenkette. Dies geschieht durch die Funktion `itoa`, die ganze Zahlen in die entsprechenden Zeichenketten verwandelt und in jeder Bibliothek vertreten ist.
Das Hauptprogramm sieht dann wie in Implementierung 3.9 aus. Der wesentliche Punkt in diesem Hauptprogramm findet sich in der Iteration über die Liste. Sie sehen, daß die konkrete Ausprägung der Klasse des gerade betrachteten Listenobjektes keine Rolle bei der Berechnung der Gehaltssumme spielt. Vielmehr werden die entsprechenden Funktionen zur Berechnung des Gehalts, der Steuer und des Freibetrags aufgerufen. Durch die dynamische Bindung sind wir in der Lage, die *richtigen* Versionen der entsprechenden Funktionen aufzurufen.

Implementierung 3.9 Freut sich der Zahlmeister?

```
main() {
       HofburgListe *eineListe ... ;
       eineListe->StartIteration();
       Hofarbeiter * hoferl  = eineListe->DieserAngestellte();
       long int gehaltsSumme = hoferl->Gehalt();
       long int steuerSumme  = hoferl->Steuer();
       long int freiBetragsSumme = hoferl->Freibetrag();
       while (eineListe->WeiterSchalten() == 0) {
            hoferl            = eineListe->DieserAngestellte();
            gehaltsSumme     += hoferl->Gehalt();
            steuerSumme      += hoferl->Steuer();
            freiBetragsSumme += hoferl->Freibetrag();
            }
     // Ausdruck
}
```

Die oben angedeutete Einführung einer Kennung für jede Stellenart in einem alternativen Zugang ohne Vererbung, die an die Stelle der Ausnutzung dynamischer Bindung treten könnte, würde zu einer recht umständlichen Formulierung führen. Die entsprechende Klasse muß insbesondere an die Anforderungen der einzelnen Stellenarten angepaßt werden. Da, wie wir gesehen haben, die Berechnung der entsprechenden Daten nicht gleichförmig erfolgt, andererseits alle Möglichkeiten vorgehalten werden müssen, führt dies zu einer höllisch unübersichtlichen Klassenformulierung. Das ist insgesamt fehleranfällig, nicht änderungsfreundlich und, wenn man sich die Klasse genauer ansieht, auch wesentlich weniger verständlich als bei der gewählten Formulierung.

3.5.4 Änderungsfreundlich?

Wir demonstrieren die Änderungsfreundlichkeit unseres Zugangs durch die Einführung einer neuen Stellenart, nämlich der wohlbekannten und berühmten Klasse der `GrossZwerge`. Großzwerge sind insbesondere Hofzwerge, stellen aber in der Hierarchie der Hofzwerge etwas Besonderes dar, was durch ihre Gehaltszahlung manifestiert wird. Wir berücksichtigen diese Hofzwerge in der Hierarchie, indem wir ein geeignetes Blatt mit den entsprechenden Angaben in die Klassifikationshierarchie einfügen. Aber ein Schritt nach dem anderen.
Wir müssen natürlich zunächst einmal ein wenig genauer bestimmen, durch welche Angaben

die Großzwerge denn nun genau charakterisiert sind. Die **I**nteressenvertretung der **Z**werge **b**ei **H**ofe (**IZbH**) hat weder Mühen noch Kosten gescheut, den Tarifvertrag für Großzwerge wie folgt zu gestalten:

- Das Festgehalt beträgt 52 Gulden, der Zusatzfreibetrag beträgt 6 Gulden.
- Großzwerge gehören zur Klasse der Hofzwerge (was sich wie selbstverständlich anhört: Was denken Sie, welche erbitterten Verhandlungen durch die Vertreter der **IzbH** notwendig waren! Einige Zwerge wollten sogar wachsen, um ihren Proteststatus zu verdeutlichen).

Sehen wir uns die Klassifikationshierarchie an, so wird ein Knoten *Großzwerg* als Sohn des *Hofzwerg*-Knotens eingeführt, vgl. Abbildung 4.

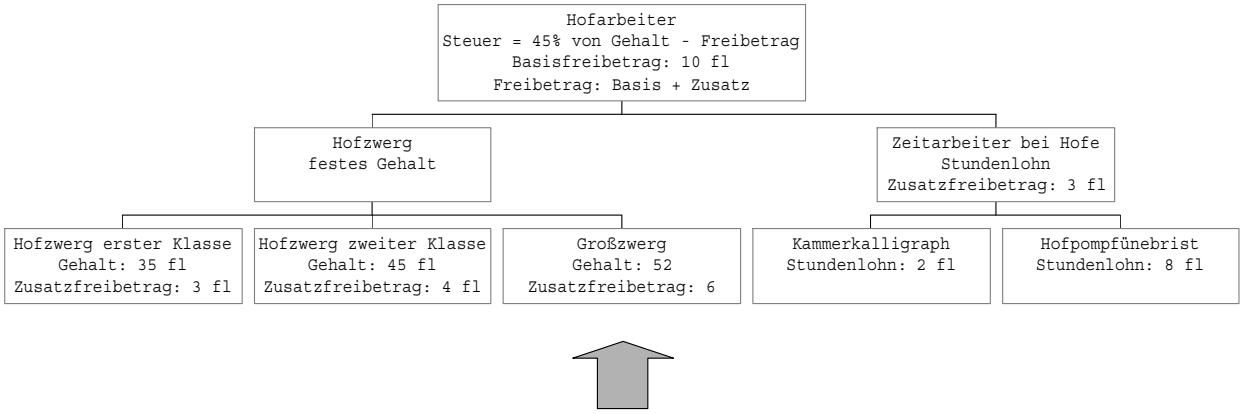

Abbildung 4: Erweiterte Klassifikationshierarchie

Auf der Grundlage unserer Überlegungen definieren wir jetzt in Anlehnung auf die Hofzwerge erster und zweiter Klasse eine Klasse `GrossZwerg`, in der wir die Angaben für die Großzwerge formulieren. Diese Klasse ist zusammen mit der Angabe der entsprechenden Konstanten in Implementierung 3.10 notiert. Das Hauptprogramm selbst wird nicht verändert, wir können jeden Großzwerg in die Liste einfügen und bei der Iteration über die Liste die Angaben über unsere Großzwerge berechnen.
Wir haben also durch die Verwendung unseres objektorientierten Ansatzes einen änderungsfreundlichen Entwurf geschaffen. Diese Änderungsfreundlichkeit wird durch unser Beispiel nachhaltig belegt.
Überlegen Sie noch einmal die Alternativen. Wenn wir die Großzwerge in einer Situation eingeführt hätten, in der wir die Stellen durch entsprechende Kennungen charakterisiert hätten, so hätten wir

1. die entsprechende Klasse modifizieren müssen, indem wir eine neue Kennung eingefügt hätten;
2. Methoden der Klasse verändern müssen, um den geänderten Rahmenbedingungen Rechnung zu tragen;

Implementierung 3.10 Die weithin sichtbare Klasse **GrossZwerg**

```
class GrossZwerg : public Hofzwerg {
  private:
       static const int FestGehalt;
       static const int Zusatzfreibetrag;
  public:
       GrossZwerg(char * t): Hofzwerg(t) { }
       int Gehalt()
            { return FestGehalt; }
       int DerZusatzfreibetrag()
            { return Zusatzfreibetrag; }
};

const int GrossZwerg::FestGehalt = 52;
const int GrossZwerg::Zusatzfreibetrag = 6;
```

3. das Hauptprogramm, nämlich die Iteration über die Liste, um die Untersuchung eines neuen Falls erweitern müssen.

Dies sind in der Tat vielfältige und fehlerträchtige Modifikationen.

4 Schlußbetrachtung

Das betrachtete Problem erforderte die Formulierung einer Klassifikationshierarchie und ihre Transofmration in die Klassenhierarchie einer objektorientierten Programmiersprache. Es entstand eine Hierarchie von Klassen, die durch Vererbung miteinander in Beziehung stehen. Bei der Konstruktion einer Liste von Objekten, die durchaus verschiedenen Klassen angehören können, zeigte sich der Vorteil des dynamischen Bindens, da stets dieselben Methoden aufgerufen wurden, die in unterschiedlichen Objekten unterschiedlich arbeiten. Dieses polymorphe Verhalten wird durch die Zugehörigkeit der entsprechenden Klassen zu einer einzigen Klasse, in der die fraglichen Methoden als rein virtuelle Methoden vereinbart sind, ermöglicht.
Die Daten sind relativ einfach strukturiert, die einzige Relation zwischen den Klassen ist die Spezialisierung. In komplexeren Problemstellungen herrschen weniger einfache Verhältnisse vor, so daß sich auch die Datenmodellierung mit feineren Instrumenten wappnen muß. Ein gegenwärtig in Literatur und Praxis intensiv diskutierter Ansatz wird durch `UML`, die *Unified Modeling Language*, realisiert (vgl. [5]). Auch sie steht in engem Verhältnis zum objektorientierten Paradigma.

Literatur

[1] Heide Balzert. *Lehrbuch der Datenmodellierung — Analyse und Entwurf.* Spektrum Akademischer Verlag, Heidelberg und Berlin, 1999.

[2] Stefan Dißmann und Ernst-Erich Doberkat. *Einführung in die objektorientierte Programmierung mit Java.* R. Oldenbourg-Verlag, München und Wien, 1999.

[3] Ernst-Erich Doberkat. *Das siebte Buch: Objektorientierung mit C++*. B. G. Teubner, 2000.

[4] Ernst-Erich Doberkat und Stefan Dißmann. *Einführung in die objektorientierte Programmierung mit BETA*. R. Oldenbourg-Verlag, München und Wien, 1997.

[5] Martin Hitz und Gerti Kappel. *UML@Work*. dpunkt.verlag, Heidelberg, 1999.

[6] Donald E. Knuth. *The Art of Computer Programming*, Band II: Seminumerical Algorithms. Addison-Wesley, Reading, Mass., 1993.

[7] O. L. Madsen, B. Møller-Pedersen und K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[8] Fritz von Herzmanovsky-Orlando. *Sämtliche Werke*. Residenz-Verlag, Salzburg und Wien, 1991. Lizenzausgabe bei Zweitausendeins, Frankfurt am Main, 1995.