auch als normale Liste verwendet werden könnte, dann aber nicht zu entscheiden wäre, welche Methode (`Eintragen()` oder `EintragenSortiert()`) anzuwenden ist.

So versieht man die `Eintragen()` Methode von `Liste` mit einer neuer Implementierung und *Polymorphismus* erlaubt es, daß jeweils die richtige Methode aufgerufen wird. Dieses Vorgehen wird im Allgemeinen als das *Überladen der Methode* bezeichnet. In unserem Beispiel veranlaßte der Aufruf von `Suchen()` an ein Objekt der Klasse `SortierteListe` die Ausführung der entsprechenden Implementierung der Klasse `Liste`, weil ja Subklassen die Funktionalität ihrer Vorfahren erben, während ein Aufruf von `Eintragen()` glücklicherweise die anders implementierte Methode von `SortierteListe` aktiviert.

Leider kann man damit noch nicht alle Fälle richtig behandeln. Oft hantiert man mit Zeigern auf Objekte anstatt mit ihnen selbst. Durch die Typkompatibilität kann jetzt ein Superklasse-Zeiger auch auf Objekte der Subklassen gesetzt werden. Dieses würde nun unweigerlich bei indirekten Methodenaufrufen zu Problemen führen. Habe ich beispielsweise einen `Liste`-Zeiger auf eine sortierte Liste gesetzt und rufe im Weiteren über diesen Zeiger indirekt die Methode `Eintragen()` auf, ist das Verhalten nicht selbstverständlich definiert. Soll die Methode von `Liste`, weil entsprechender Zeiger ja ein `Liste`-Zeiger ist, ausgeführt werden oder doch die von `SortierteListe`, weil dieser Zeiger ja effektiv auf ein solches Objekt zeigt. Um diese Doppeldeutigkeit aufzulösen, gibt es *virtuelle Methoden*, die garantieren, daß jeweils die Version der effektiv angesprochenen Klasse aktiviert wird. Deklarierten wir also in unserem Beispiel die Methode `Eintragen()` einer Liste, ob sortiert oder nicht, virtuell, würde ein indirekter Aufruf dieser Methode über einen `Liste`-Zeiger, jeweils die Variante des Objekts aktivieren, auf das dieser Zeiger gerichtet wurde — also nicht immer die `Liste`-Version dieser Methode.

Abstrakte Klassen

Die virtuellen Methoden führen uns nun aber letztlich noch zu einem anderen wichtigen Aspekt der Objektorientierung, den *abstrakten Klassen*. Vielfach zeigt sich nämlich, daß Superklassen, die nahe der Wurzel der Klassenhierarchie stehen, eher den Charakter von Abstraktionen der darunter liegenden Klassen bekommen und selbst gar nicht in Verwendung treten. Um wieder auf das Datenbankbeispiel zurück zu kommen, kann man sich hier die Klasse `Datensatz` vorstellen, die als sie selbst, also ohne konkrete Ausprägung, keinen Sinn macht, denn niemand wird einen abstrakten Datensatz instanzieren wollen und können.

Trotzdem ist so eine Klasse sinnvoll. Andere Klassen, die sich auf Datensätze generell beziehen, können diese Klasse als Interface-Definition zu beliebigen Klassen, die von `Datensatz` abgeleitet sind, verwenden. So eine abstrakte Klasse definiert nur das Interface, deklariert somit bloß virtuelle Methoden, für die es keine Implementierung gibt. Derartige virtuellen Methoden nennt man *pure virtual functions*. Klarerweise wird eine abstrakte Klasse auch ganz gewöhnliche, nicht virtuelle Methoden zur Verfügung stellen, für die allerdings eine passende Implementierung vorliegen muß, die dann an die konkreten Nachfahren vererbt wird. Als weiteres Beispiel soll hier eine abstrakte Superklasse `Term` angeführt werden, die das Interface zur Methode `getValue()` als pure virtual function realisiert. Jede von `Term` abgeleitete konkrete Klasse muß nun diese Methode entsprechend ihrer Bedeutung überladen, was zur Folge hat, daß dann auf sämtliche Terme die Methode `getValue()` angewandt werden kann, ohne berücksichtigen zu müssen, um was für einen Term es sich handelt.

lich irrelevant. Dies stellt eine gewaltige Verminderung der berühmt-berüchtigten Seiteneffekte dar, die bei Verwendung des prozeduralen Ansatzes so viele Probleme schaffen.

Die Objektorientierung verlangt jedoch einen gewissen Wandel im Planen und Entwickeln. Nun geht man von den Klassen aus und denkt "Was für Funktionalität stelle ich meinen Benutzern zur Verfügung?". Man geht nicht mehr her und plant bzw. schreibt eine Prozedur, die ein per Parameter spezifiziertes Array sortiert, sondern das Array selbst kann mittels einem Methodenaufruf dazu veranlaßt werden, sich zu sortieren. Man denkt also generell in kleineren und deshalb besser überschaubaren Teilen.

Vererbung

Das mit der Klasse verbundene wohl wichtigste Prinzip ist die *Vererbung*. Man spricht davon, *Subklassen* von einer *Superklasse abzuleiten*, wodurch sich oft eine hierarchische Struktur von Klassen ergibt. Klassen nun zu einander in eine Verwandtschaftsbeziehung zu bringen, hat mehrere Gründe, aber immer handelt es sich darum, reale Beziehungen nachzubilden. So kann man sich in einer Personaldatenbank einer großen Firma die Klasse `Beschäftigter` vorstellen, die alle Daten repräsentiert, die für alle Beschäftigten benötigt werden. Beschäftigt diese Firma nun, zum Beispiel, auch Kraftfahrer, bietet sich eine Subklasse `Kraftfahrer` an, die zusätzliche Informationen, die nur für diese Kategorie von Beschäftigten gespeichert werden sollen, repräsentiert. Ebenso motivierten sich demnach Subklassen `Bürokraft`, `Techniker`, usw.

Ein Vorteil dieser Struktur ist der, daß alle Funktionalität, die für Beschäftigte im Allgemeinen gilt und deshalb als Methoden der Klasse `Beschäftigter` vorhanden ist, nicht von jeder Subklasse neu implementiert werden muß, sondern von der Superklasse verwendet wird. Dieser Vorteil ist auch als *implementation sharing* bekannt. Ein weiterer Vorteil, der sich aus dieser Struktur ergibt ist jener, daß überall dort, wo weitergehende Funktionalität auf Beschäftigten realisiert ist, natürlich auch mit Kraftfahrern, Technikern und anderen speziellen Beschäftigten vorgegangen werden kann. Dieser Umstand ergibt sich aus der *Typkompatibilität* zwischen Subklassen und Superklasse. Im Klartext heißt das: "Wenn ich eine Verarbeitungsfunktion für Beschäftige habe, kann ich damit genauso Techniker, Kraftfahrer oder andere spezielle Beschäftigte bearbeiten.".

Polymorphismus und virtuelle Methoden

Führt man nun diese Überlegungen weiter, so sieht man, daß Vererbung auch zur Spezialisierung verwendet wird. Als Beispiel realisiere die Superklasse `Liste` eine lineare Liste bestimmter Daten mit der dazugehörigen Funktionalität Eintragen, Suchen und Löschen. Leitet man nun eine Subklasse `SortierteListe` ab, die wie der Name schon ausdrückt auch eine Liste repräsentiert, die allerdings immer sortiert ist, sieht man, daß nunmehr die Methode `Eintragen()` ganz anders implementiert werden muß. Sicher könnte man hierfür eine eigene Methode `EintragenSortiert()` schreiben, aber dies würde dem objektorientierten Prinzip der Typkompatibilität widersprechen, weil wie oben beschrieben, jede sortierte Liste

nutzer dann in Anspruch nehmen kann. Man kann sich das recht gut als *message passing* vorstellen. Man schickt an ein Objekt eine Nachricht, die es veranlaßt, in gewünschter Weise mit den ihm eigenen Daten umzugehen. Diese Nachrichten, die also die Funktionalität einer Klasse repräsentieren, nennt man *Methoden* oder *member functions*. Dazu zählen vor allem die Zugriffsfunktionen auf die eigentlichen Daten, die man *Instanzvariable* oder *member variables* nennt, sowie diverse Aktionen auf ihnen. Als Beispiel kann man sich eine Datenbankklasse vorstellen, an die man Anfragen (Queries) verschiedener Art stellen kann.

Datenkapselung und Datenkonsistenz

Man betrachtet oft die von einer Klasse zur Verfügung gestellten Methoden als das *Interface der Klasse*. Jeder Zugriff auf die zu Grunde liegenden Daten, sei er nun lesend oder schreibend, muß über diese Methoden erfolgen, woraus sich eine *Kapselung der Daten* ergibt. Dies kann ausgenutzt werden, um die Kontrolle über den Zugriff auf die Daten zu behalten. Somit kann zum Beispiel gesteuert werden, wer welche Instanzvariable verändern darf.

Vielfach gliedern sich diese internen Daten einer Klasse weiter in verschieden Teile (meist auch wieder Objekte), die in ihrer Gesamtheit den Zustand (Wert) der Klasse repräsentieren. Unterliegen diese Teile gewissen Abhängigkeiten — also wenn beispielsweise gewisse Werte von Teil A einer Klasse bestimmte Werte von Teil B derselben Klasse bedingen — dann liegt es bloß im Design der entsprechenden Klasse, ob Konsistenz innerhalb ihrer Daten gewährleistet werden kann. Keine der Aktionen, die auf so einer Klasse dann ausgeführt werden, kann Inkonsistenz der internen Daten verursachen.

Um die Erhaltung der Konsistenz innerhalb einer Klasse zu gewährleisten, gibt es unter anderem[2] spezielle Methoden, den *Konstruktor* und *Destruktor* einer Klasse. Mittels dem Konstruktor, der automatisch bei jeder Objektinstanzierung ausgeführt wird, kann erreicht werden, daß ein Objekt gleich bei seiner Instanzierung in einen konsistenten Zustand versetzt wird. Zerstören die gewöhnlichen Methoden einer Klasse nicht diese innere Konsistenz, was letztlich Kriterium eines guten Designs ist, bleibt das Objekt während seiner "Lebenszeit" in diesem erwünschten Zustand. Abschließend sorgt bei der Demontage eines Objekts der Destruktor dafür, daß ein während der "Objektlebenszeit" womöglich aufgebauter, komplexer innerer Zustand Objekts ordnungsgemäß aufgeräumt wird. Die Notwendigkeit eines Destruktors wird vor allem bei verketteten Strukturen deutlich, wo einmal angelegte Daten auch am Schluß wieder aus dem Speicher beseitigt werden müssen.

Modularisierung

Weiters ergibt sich aus dem Prinzip der Klasse, daß nun weitgehend modular programmiert und entwickelt werden kann. Schließlich braucht man ja nur das Interface zu den jeweiligen Klassen spezifizieren, wie dann die einzelnen Methoden implementiert sind, bleibt meist ziem-

---

2 Die hier angesprochenen Möglichkeiten der Objektorientierung, *pre-* und *postconditions* für Methoden sowie gewisse *invariants*, die über die "Lebenszeit" eines Objekts halten müssen, zu spezifizieren, können hier nicht näher behandelt werden.

Grundlagen der Objektorientierten Programmierung

Helwig Löffelmann[1]

Institut für Computergraphik, Technische Universität Wien
A-1040 Karlsplatz 13/186/2

Einleitung

Objektorientierung ist in den letzten Jahren ein wichtiges Schlagwort im Bereich Softwareentwicklung geworden. Und obwohl dieses Prinzip noch ziemlich jung ist, hat es sich bereits auf vielen Gebieten gegenüber herkömmlichen Ansätzen durchgesetzt. Sowohl in der Wirtschaft als auch in der Wissenschaft spannt sich hierfür ein weites Anwendungsgebiet.

Auch wenn es in der Zukunft wohl immer Bereiche geben wird, innerhalb derer man besser einen logischen oder funktionalen Ansatz wählt, oder solche, wo maschinennahes Entwickeln von Nöten ist, so wird doch das prozedurale Programmier- und Designprinzip eher zurückweichen und noch mehr Platz für die Objektorientierung machen als dies ohnehin bereits geschehen ist. Dies kann auch angenommen werden, wenn man beachtet, daß heutzutage bereits sehr viel Software existiert, die in prozeduralen Programmiersprachen wie C, COBOL und FORTRAN geschrieben wurde und weiter verwendet wird, deshalb auch gewartet werden muß, erweiterbar bleiben soll, usw.

Weil nun die Objektorientierung eher den Bereich der bislang prozeduralen Programmiersprachen erobert, also als eine Verbesserung dieser Methoden angesehen werden kann, sollen im Weiteren die sich ergebenden Unterschiede zwischen diesen beiden Kategorien herausgestrichen werden.

Das fundamentale Prinzip der Klasse

Mit der Klasse als Programmier- und Designprinzip steht und fällt die Objektorientierung. Waren bisher Daten und Funktionalität getrennt, so bilden jetzt Dateneinheiten zusammen mit der zu ihnen assoziierten Funktionalität *Klassen*. Man könnte also formulieren “Die Daten werden Herr ihrer selbst.”. Klassen repräsentieren demnach einen Typ, von dem konkrete *Objekte instanziert* werden können. Als Beispiel kann man hier ins Tierreich abschweifen, wo der Begriff der Klasse recht ähnlich in Verwendung ist. Die "Objekte" Bello und Struppi wären somit "Instanzen" der "Klasse" Hund.

Es gibt also nicht mehr die Abfolge von Prozeduren, die ihrerseits auf globalen oder als Parameter übergebene Daten arbeiten, sondern Daten bieten Funktionalität an, die man als Be-

---

1 mailto:helwig@cg.tuwien.ac.at, http://www.cg.tuwien.ac.at/~helwig/