Harro von Lavergne

Objekte, Klassen, Module, Kontrakte und Komponenten

Mit dem gewachsenen Einfluß der objektorientierten Programmierung (OOP) in der Software-Erstellung haben sich zunehmend auch die Grenzen dieses Paradigmas gezeigt. Seit einigen Jahren verlagert sich daher das Interesse zu einer umfassenderen Sichtweise, die unter dem Schlagwort komponentenorientierte Programmierung (COP) bekanntgeworden ist.
Die folgenden Darlegungen fassen an Hand bekannter Programmbausteine die Entwicklungslinien von den Konzepten der Strukturierten Programmierung und Schrittweisen Verfeinerung über das Parnas-Modell der Datenabstraktion zur objektorientierten Programmierung zusammen und stellen die wesentlichen Merkmale der komponentenorientierten Programmierung sowie die Beziehungen zwischen beiden Paradigmen dar.
An einem exemplarischen Beispiel wird demonstriert, wie Komponentenorientierung auch im Unterricht der Sekundarstufe II vermittelt werden kann.

Komponentenklassenobjektinstanz - Objektklasseninstanzkomponente - Klasseninstanzkomponentenobjekt ...
Wieviele Permutationen von vier Objekten gibt es?

Obwohl sich inzwischen der Nebel über der Objektwelt ein wenig gehoben hat, sind leider auch heute noch in nicht wenigen Darstellungen der Grundlagen objektorientierter Programmierung wenn nicht solche so doch zumindest ähnliche Wortgetüme zu lesen. Ich unterstelle, daß der jeweilige Autor diese nicht schlicht aus irgendeiner seiner Quellen übernommen hat, sondern daß für ihn - hoffentlich - klare Vorstellungen der damit verbundenen Inhalte existieren. Es ist allerdings für die Kommunikation zwischen Menschen, die sich mit ein und derselben Sache beschäftigen irgendwann nötig, eine gemeinsame Sprache zu finden, was in der Regel heißt, Inhalte und Bezeichner zu präzisieren und zu vereinheitlichen.

Ich möchte hier darstellen, welche Begriffe sich in den Publikationen der vergangenen Jahre als wesentlich herausgebildet haben und diese an Beispielen erläutern. Dazu ist es erst einmal notwendig, die Bedeutung jedes einzelnen Bestandteils der eingangs stehenden Wortgebilde zu klären - im Anschluss stellt sich möglicherweise heraus, daß Zusammensetzungen wie die genannten Beispiele oder auch solche wie *Klassenkomponente* oder *Objektinstanz* keinerlei Neuigkeiten oder Verbesserungen des Verständnisses ergeben, vielmehr zur Verwirrung beitragen.

Aber nicht nur bei Zusammensetzungen wie den oben stehenden Begriffsbildungen, selbst bei den einzelnen Bestandteilen dieser Konstruktionen, insbesondere dem Begriff des Objekts, gibt es derartig viele verschiedene Vorstellungen, daß wie in der Religion jeder sich eine ihm passende aussuchen und dabei zahlreiche Gleichgesinnte finden kann.

In einer über mehrere Jahre geführten intensiven Diskussion (Nachzulesen in der renommierten Computerzeitschrift SD-Magazine [SD-M]) haben bekannte Autoren versucht, den Wirrwarr zu entflechten und bei Reduktion auf das Wesentliche eine Klärung der Zusammenhänge zu erreichen. Dabei haben sich die Begriffe Objekt, Klasse, Modul und Komponente als Kern herauskristallisiert.

Was sind Objekte?

Objekte sind - mit wenigen Ausnahmen - dynamische, nur zur Laufzeit eines Programms existierende Entitäten. Jedes Objekt hat eine eindeutige Identität und einen zu jeder Zeit eindeutig definierten, wenn auch veränderbaren Zustand. Jedes Objekt stellt eine Kapsel der in ihm enthaltenen Daten und Operationen dar, die seine Identität bilden. Da Objekte Repräsentationen einer ihnen zugrunde liegenden Verallgemeinerung ihrer Eigenschaften sind, müssen sie erzeugt werden, bevor sie agieren können. Für jedes zu erzeugende Objekt muß es ein Muster geben, nach dem es entsteht. Im wesentlichen existieren zwei Arten von Mustern - Prototypen und Klassen

- Ein Prototyp ist ein unveränderbares Objekt, das in seinen Eigenschaften weitgehend mit dem zu erzeugenden Objekt übereinstimmt, wobei dessen Eigenschaften bei der Erzeugung des individuellen Objekts über benut-

zerdefinierbare Parameter angepasst werden können. Prototypen sind eine der oben erwähnten Ausnahmen, als Objektmuster liegen sie in gespeicherter Form vor, sie sind persistent.

– Eine Klasse ist eine Abstraktion der Objekteigenschaften, die die möglichen Zustände und Operationen aller aus ihr erstellten Objekte bestimmt, sie selbst ist kein Objekt, sondern nur ein Erzeugungsmuster, ein Konstruktionsplan. Jedes von der Klasse abgeleitete Objekt ist eine Realisierung des Plans, eine Instanz der Klasse.

Unabhängig von seiner Herkunft muß jedes Objekt erzeugt und initialisiert werden. Dies kann entweder durch eine statische Prozedur geschehen, solche Prozeduren werden als Konstruktoren bezeichnet, oder durch ein spezielles, als Objektfabrik bezeichnetes Verfahren. Dabei kann es sich um ein bereits existierendes Fabrikobjekt (factory object) handeln oder um eine in einem Objekt residierende Fabrikmethode (factory method). Die in Objektfabriken verwendeten Objekte sind eine zweite der oben erwähnten Ausnahmen; sie sind wie Prototypen persistent, darüber hinaus sind sie im Gegensatz zu "normalen" Objekten und Prototypen unveränderbar, sie haben also weder eine dynamische Existenz, noch gehen von ihnen außer der Objekterzeugung irgendwelche Aktionen aus.

Objekte im komponentenorientierten Programmentwurf

Einzelne Objekte sind bis auf seltene Ausnahmefälle nicht "lebensfähig". Zur Erzeugung sinnvoller Ergebnisse benötigt ein Programm im Allgemeinen das Zusammenwirken mehrerer verschiedener Objekte. Dies Zusammenwirken erfordert in praktisch allen Fällen einen uneingeschränkten Zugriff auf Einzelheiten der kooperierenden Objekte. Entsprechend müssen die erzeugenden Muster - Klassen oder Prototypen - wechselseitig Kenntnis voneinander haben und Zugang zu den von ihnen erzeugten Objekten ermöglichen, eine Notwendigkeit, für die rein objektorientierte Programmierung (OOP) kein überzeugendes Verfahren anbietet.

Die Diskussion weist daher seit geraumer Zeit über Objekte hinaus. Bereits 1994 stellte Jon Udell im Computermagazin BYTE fest: *Object technology failed* [Ude94], der Titel eines 1995 erschienenen Aufsatzes von D. Gruntz lautete: *Objects are not enough* [Gru95] und Broy/Siedersleben [BrS02] kommen sogar zu dem Schluss, daß Objekttechnologie bei der Erstellung großer Softwaresysteme hinderlich sei. Selbst wenn man nicht soweit gehen möchte ist festzustellen, daß das Schlüsselwort für die Erstellung anwendungsorientierter Programme nicht Objekt sondern Komponente heißt, wobei Objekte - richtig verstanden - in der komponentenorientierten Programmierung (COP, oft auch als CBD: Component Based Development bezeichnet) eine wesentliche Rolle spielen können.

Als Komponente wird dabei ein Kollektiv erzeugender Muster bezeichnet, das eine in sich geschlossene, im Rahmen eines entsprechenden Grundsystems funktionsfähige Einheit darstellt. Wesentlicher Aspekt von Komponenten ist ihre Abgeschlossenheit, sie interagieren niemals direkt mit ihrer Außenwelt. Alle Dienste, die sie anbieten oder benötigen, sind ausschließlich durch spezielle Zugangspunkte (hot spots) definiert - die Schnittstellenspezifikationen der einzelnen Komponenten. Eine Schnittstelle gibt an, welche Vorleistungen die Komponente für ihr Funktionieren benötigt und welche Ergebnisse sie unter diesen Bedingungen liefert; die Schnittstelle ähnelt damit einem Kontrakt, der die zu erbringenden Leistungen des Kontraktnehmers und die zu erfüllenden Vorbedingungen spezifiziert. Aus diesem Grund wird komponentenorientierte Programmierung häufig auch als Programmieren mit Verträgen (programming by contract) [GHJ98] bezeichnet. In dieser Sichtweise ist eine Komponente eine reine black box, eine besonders rigide Anwendung des Prinzips der Datenkapselung und Datenabstraktion - zumindest sollte sie es sein, leider ist das nicht immer der Fall.

Datenkapselung und Datenabstraktion

Datenkapselung und Datenabstraktion sind nicht neu, bekannt ist die von D. L. Parnas bereits 1972 [Par72] beschriebene Methode der abstrakten Spezifikation eines Programms, bei der dessen Funktion in einer mehr oder minder formalen Weise dargestellt wird, die von den Details der Programmimplementierung absieht und die Daten zusammen mit den auf ihnen operierenden Prozeduren in den Vordergrund stellt.

Als wichtigste programmtechnische Realisierungen dieser Methode entstanden die Programmiersprache ADA und die aus PASCAL entwickelte Sprache MODULA-2. In MODULA-2 ist das Geheimnisprinzip durch die Aufteilung eines Programms in einen öffentlichen Teil (definition module) und einen privaten Teil (implementation module) verwirklicht. Der öffentliche Teil stellt die Schnittstellenbeschreibung des Programms dar, die Präsentation derjenigen Daten und Prozeduren, die durch andere Programme genutzt werden können. Damit ist ein solcher, allgemein Modul genannter Programmbaustein eine in sich abgeschlossene funktionale Einheit. Die wichtigsten modular realisierbaren Programmbausteine sind als *Abstrakte Datenstruktur* (ADS) bzw. *Abstrakter Datentyp* (ADT) bekannt. Beide stellen Zusammenfassungen von Daten(-werten) und den auf ihnen definierten Operationen dar. Der Unterschied besteht im wesentlichen darin, daß von einer abstrakten Datenstruktur immer nur jeweils ein Exemplar in einer Systemumgebung existieren kann, bei einem abstrakten Datentyp dagegen gleichzeitig mehrere verschiedene Objekte in derselben Umgebung vorhanden sein können.

Abstrakte Datenstrukturen und abstrakte Datentypen

In der Nachfolge von MODULA-2 sind weitere PASCAL ähnliche Programmiersprachen entstanden. Die anschließenden Beispiele verwenden eine der neuesten dieser Sprachen, COMPONENT PASCAL. Die abgedruckten Schnittstellendefinitionen sind anders als in MODULA-2 keine gesonderten Definitionsmodule - diese gibt es in COMPONENT PASCAL nicht - vielmehr werden die öffentlichen Teile eines Programms innerhalb der Implementierung mit Exportmarken versehen und vom Compiler in der ausführbaren Codedatei entsprechend gekennzeichnet, Klienten haben dadurch automatisch Zugriff auf die Exporte, nicht aber auf die Implementierung. Gedruckte Schnittstellendefinitionen lassen sich mit dem systemeigenen object-browser erstellen, sie dienen in COMPONENT PASCAL nur zu Dokumentationszwecken.

Den folgenden Programmbeispielen liegen bekannte Algorithmen für die Implementierung linearer Listen als Stapel (stack) beziehungsweise Schlange (queue) zugrunde, die Schnittstellen sind - soweit möglich - ähnlich gehalten, da nicht die Algorithmen, sondern die Gemeinsamkeiten und Unterschiede von ADS, ADT, Klassen und Komponenten von Interesse sind.

```
DEFINITION FILOsADS;

    PROCEDURE AnzahlElemente (): INTEGER;
    PROCEDURE EntferntesElement (): INTEGER;
    PROCEDURE ErstesElement (): INTEGER;
    PROCEDURE Leer (): BOOLEAN;
    PROCEDURE NächstesElement (): INTEGER;
    PROCEDURE Stapeln (Inhalt: INTEGER)

END FILOsADS.


DEFINITION FILOsADT;

    TYPE
        Stapel = POINTER TO LIMITED RECORD
            AnzahlElemente: PROCEDURE (stapel: Stapel): INTEGER;
            EntferntesElement: PROCEDURE (stapel: Stapel): CHAR;
```

```
            ErstesElement: PROCEDURE (stapel: Stapel): CHAR;
            Leer: PROCEDURE (stapel: Stapel): BOOLEAN;
            NächstesElement: PROCEDURE (stapel: Stapel): CHAR;
            Stapeln: PROCEDURE (stapel: Stapel; Inhalt: CHAR)
        END;

    PROCEDURE NeuerStapel (OUT stapel: Stapel);

END FILOsADT.
```

Die beiden vorstehenden Schnittstellen zeigen die öffentlichen Teile zweier Implementationen eines Stapels als ADS beziehungsweise als ADT. Beide Implementationen sind statisch in dem Sinn, daß als Elemente jedes Stapels nur feste Grundeinheiten - ganze Zahlen bzw. Zeichen - zugelassen sind. *FILOsADS* exportiert ausschließlich Zugriffsprozeduren für die Verwaltung der Stapelelemente, das eigentliche Stapelobjekt ist verborgen, es existiert nur ein einziges Mal implizit in der Systemumgebung. Dagegen enthält *FILOsADT* einen explizit deklarierten und exportierten Typ *Stapel*, der als (Zeiger auf einen) Verbund mit Prozedurfeldern implementiert ist. Klienten des Moduls können gleichzeitig mehrere, von einander unabhängige Stapelobjekte erzeugen. In *FILOsADT* müssen deshalb im Gegensatz zu *FILOsADS* die Zugriffsprozeduren explizite Parameter des Typs *Stapel* enthalten.

Will nun ein Klient von *FILOsADT* andere Objekte als Zeichen - beispielsweise reelle Zahlen - verwalten, müssen der Quelltext geändert und das Modul neu kompiliert werden. Falls die Verwaltung von Zeichenstapeln weiterhin möglich sein soll, muß für den neuen Elementtyp das entsprechende Modul neben den bisherigen in der Systemumgebung vorliegen, dies muß für jeden weiteren gewünschten Elementtyp wiederholt werden. Binnen kurzer Zeit kann es so zu einer Flut gleichzeitig existierender Stapelmodule kommen, die sich lediglich in dem Typ der verwalteten Elemente unterscheiden, während der Stapel selbst unverändert bleibt.

Vererbung und Polymorphie - Klassen

Eine Lösung für dieses "Überschwemmungsproblem" bietet die objektorientierte Programmierung, bei der ein wesentlicher Aspekt in dem Prinzip der Vererbung zu sehen ist. Während ältere Programmierkonzepte die Optimierung eines einzelnen Programms betonten (bekannt als structured programming und stepwise refinement), stellt OOP den Aspekt der Wiederverwendbarkeit in den Vordergrund. Vererbung ermöglicht es, Programme auf "Vorrat" zu schreiben, als Halbfabrikate, die von Klienten entsprechend den individuellen Anforderungen ergänzt werden können, wobei anders als bei einer ADS oder einem ADT der Quelltext des Basismoduls nicht geändert oder neu kompiliert werden muß. Der Erfolg der OOP beruht auf dieser als Polymorphie bekannten Möglichkeit der dynamischen Typerweiterung. Polymorphie bedeutet, daß Objekte einer Erweiterungsklasse an jeder Stelle verwendet werden können, an der Objekte der Basisklasse verwendbar sind.

```
DEFINITION FILOsPoly;

    TYPE
        Element = POINTER TO ABSTRACT RECORD
            (element: Element) NächstesElement (): Element, NEW
        END;

        Stapel = POINTER TO LIMITED RECORD
            (stapel: Stapel) AnzahlElemente (): INTEGER, NEW;
            (stapel: Stapel) EntfernteElement (): Element, NEW;
            (stapel: Stapel) ErstesElement (): Element, NEW;
            (stapel: Stapel) Leer (): BOOLEAN, NEW;
            (stapel: Stapel) Stapeln (element: Element), NEW
        END;

    PROCEDURE NeuerStapel (OUT stapel: Stapel);

END FILOsPoly.
```

Die Schnittstelle des Moduls *FILOsPoly* stellt eine Realisierung des Vererbungskonzepts vor. Die auffallendste Änderung ist die zusätzliche Einführung der Klasse *Element* anstelle des in *FILOsADT* verwendeten konkreten Elementtyps *CHAR*.

Klassen sind in COMPONENT PASCAL nicht besondere Sprachkonstrukte, sondern existieren als prinzipiell erweiterbare Verbundtypen (Datentyp *RECORD*), wobei die Erweiterbarkeit durch Attribute modifiziert werden kann, die folgende Eigenschaften haben

Attribut	erweiterbar	allozierbar
– (final)	Nein	Ja
EXTENSIBLE	Ja	Ja
ABSTRACT	Ja	Nein
LIMITED	Im definierenden Modul	Im definierenden Modul

Mit den dargestellten Attributen ist es möglich, Klassen für spezielle Zwecke zu konstruieren und zu exportieren, ohne die in vielen anderen Programmiersprachen mit der Veröffentlichung verbundenen Sicherheitsrisiken einzugehen. Die Zusammenstellung der Klassenattribute zeigt, daß die Klasse *FILOsPoly.Stapel* wie auch schon *FILOsADT.Stapel* auf Grund des *LIMITED*-Attributs von einem Klienten des Moduls *FILOsPoly* nicht erweitert werden kann, sie ist dadurch vor unerwünschten Veränderungen geschützt.

Dagegen ist die für Klienten interessante Klasse *FILOsPoly.Element* durch das Attribut *ABSTRACT* als erweiterbar gekennzeichnet. Objekte der Klasse *FILOsPoly.Element* sind auf Grund des *ABSTRACT*-Attributs nicht allozierbar, Klienten können diese Basisklasse nur nutzen, indem sie sie in konkreten Unterklassen erweitern, z. B. zu einer Klasse *IntElement* oder zu einer Klasse *TextElement*. Eine abstrakte Klasse ist in erster Linie eine Schnittstellenspezifikation, die die Eigenschaften und Fähigkeiten der Klasse innerhalb ihres Basismoduls dokumentiert.

Trotzdem muß eine abstrakte Klasse nicht völlig abstrakt sein, sie kann Datenfelder und Operationen enthalten, die bereits wesentliche Eigenschaften ihrer Objekte definieren. Klassen wie *FILOsPoly.Element* werden wegen ihrer Erweiterbarkeit zu beliebigen Unterklassen in der Literatur manchmal auch als generisch bezeichnet. In ADA, C++, EIFFEL oder SMALLTALK existiert Generizität als eigenes Sprachkonstrukt, in anderen Sprachen läßt sich Generizität mit abstrakten Erweiterungsklassen simulieren [Mey86], [Szy02].

Durch die Einführung der Klasse *FILOsPoly.Element* sind folgende Ziele erreicht. Klienten des Moduls *FILOsPoly* sind im allgemeinen nicht an einer Veränderung der Klasse *FILOsPoly.Stapel* interessiert, vielmehr unterstellen sie, daß die Stapelverwaltung korrekt implementiert ist, sie benötigen lediglich die Möglichkeit, ein Objekt der Klasse *Stapel* zu erzeugen und ihm die Verwaltung der klientenspezifischen Elemente zu übergeben. Die Klasse *Stapel* kann vom Implementierer des Moduls *FILOsPoly* vollständig ausprogrammiert und gegen Veränderungen durch Klienten geschützt werden. Auf Grund des *LIMITED*-Attributs können von Klienten weder Unterklassen (Typerweiterungen) deklariert, noch unmittelbar Objekte der Klasse alloziert werden. Die Allokation eines Stapelobjekts ist nur innerhalb des die Klasse *Stapel* enthaltenden Moduls *FILOsPoly* möglich. Klienten müssen Stapelobjekte über die vom Modul *FILOsPoly* exportierte Prozedur *NeuerStapel (OUT stapel: Stapel)* erzeugen, bei der es sich um eine Konstruktor-Prozedur handelt. Mit dem Konstruktor kann *FILOsPoly* die korrekte Initialisierung der Stapelobjekte garantieren, Programmsicherheit und Programmstabilität werden dadurch wesentlich erhöht.

Wie bereits erwähnt bedeutet Polymorphie, daß Objekte einer Erweiterungsklasse an jeder Stelle verwendet werden können, an der Objekte der Basisklasse verwendbar sind. Polymorphie ermöglicht es der Klasse *FILOsPoly.Stapel*, Operationen auf Objekten der Klasse *FILOsPoly.Element* zu implementieren, sie zum Beispiel auf dem Stapel abzulegen oder daraus zu entfernen. Ein Stapelobjekt kann zur Laufzeit des Programms jedes beliebige, ihm (und dem Programmierer des Basismoduls) völlig unbekannte Objekt irgendeiner Unterklasse von *FILOsPoly.Element* soweit verarbeiten, wie die Eigenschaften dieses Objekts bereits in der Basisklasse definiert sind. Voraussetzung dieser aus dem Vererbungskonzept resultierenden polymorphischen Flexibilität ist die Fähigkeit, verschiedene Programmteile nicht wie früher üblich rein statisch zur Kompilationszeit zusammenzufügen (early binding), sondern dynamisch erst zur Laufzeit eines Programms (late binding). Nicht die in vielen eher folkloristisch-philo-

sophischen Darstellungen betonte Einheit von Daten und Methoden sondern die Verbindung von Polymorphie und später Bindung ist die wesentliche Grundlage für den Erfolg der objektorientierten Programmierung gewesen.

Klassen- und Methodenattribute

Anders als die objektzentrierten Operationen der Module *FILOsADS* und *FILOsADT* sind die Stapeloperationen im Modul *FILOsPoly* nicht als gewöhnliche Prozeduren bzw. Prozedurfelder, sondern als typgebundene Prozeduren realisiert (typgebundene Prozeduren sind in COMPONENT PASCAL erkennbar an dem vor dem Prozedurnamen stehenden Empfängerparameter [receiver parameter]). In rein klassenbasierten Programmiersprachen wie EIFFEL, JAVA oder SMALLTALK existieren ausschließlich typgebundene Prozeduren, in OOP-Terminologie werden solche Prozeduren Methoden genannt, in Hybridsprachen wie COMPONENT PASCAL ist der Terminus Methode umfassender, in ihnen sind generell die auf einem erweiterbaren ADT definierten Operationen als Methoden anzusehen, unabhängig von der Art der Implementierung. Sowohl *FILOsADT* als auch *FILOsPoly* arbeiten also mit Methoden auf Objekten der Klasse *Stapel*.

Im Gegensatz zu den objektzentrierten Methoden der Klasse *FILOsADT.Stapel*, die - bei unveränderter Signatur - in jedem einzelnen von der Klasse abgeleiteten Objekt verschieden sein können, sind die typgebundenen Methoden des Moduls *FILOsPoly* klassenzentriert, sie können von Klienten nicht für einzelne Objekte zur Laufzeit eines Programms ausgetauscht werden. Ein Austausch typgebundener Prozeduren ist nur möglich, indem diese in einer Unterklasse der Basisklasse überschrieben werden. Nun besitzt die Klasse *FILOsPoly.Stapel* das Attribut *LIMITED*, sie ist in einem Klientenmodul nicht erweiterbar, die vorgegebenen Stapeloperationen sind - absichtlich - gegen Veränderungen geschützt. Dagegen ist die Klasse *FILOsPoly.Element* als *ABSTRACT* deklariert. Sofern Klienten Objekte der Klasse allozieren wollen, muß diese in einer Unterklasse erweitert werden. In dieser Unterklasse könnte die Methode *(element: Element) NächstesElement (): Element, NEW* entsprechend dem Gesagten überschrieben, also durch eine klientenspezifische Variante ersetzt werden. An dieser Stelle ist zu beachten, daß sich in COMPONENT PASCAL die Eigenschaften typgebundener Prozeduren ähnlich wie bei Klassen durch Attribute modifizieren lassen, wobei es insgesamt vier verschiedene Attribute gibt

Attribut	**Bedeutung**
-	Methode kann aufgerufen, aber nicht überschrieben werden (finale Methode)
EXTENSIBLE	Methode kann aufgerufen und überschrieben werden
ABSTRACT	Methode kann nicht aufgerufen, muß überschrieben werden
EMPTY	Methode kann aufgerufen und überschrieben werden

Auf Grund der Attribuierung ist die Methode *(element: Element) NächstesElement (): Element, NEW* final, sie kann trotz der Erweiterbarkeit der Klasse *FILOsPoly.Element* nicht in einer Unterklasse redefiniert werden, in der Unterklasse lassen sich lediglich Datenfelder und zusätzliche, klientenspezifische Methoden ergänzen, die Methode *NächstesElement* ist gegen Veränderungen durch Klienten geschützt.

Datensicherheit

Die in COMPONENT PASCAL existierenden Möglichkeiten, Klassen und Methoden durch Attribute gezielt auf den jeweiligen Verwendungszweck zuschneiden zu können, sind wesentliche Grundlagen für die Erstellung von Komponenten, bei denen es darauf ankommt, den Aspekt Sicherheit der Daten und der Verarbeitungsmethoden mit maximaler Flexibilität zu verbinden.

Mit der Flexibilisierung von Programmen durch objektorientierte Programmierung geht eine gewollte partielle Veröffentlichung ihres Innenlebens einher, wobei wichtige Teile des Programms - in erster Linie die Methoden und die zu verarbeitenden Daten - gegen unkontrollierte Veränderung von aussen geschützt bleiben sollen. In vielen Programmiersprachen läßt sich der Schutz der Daten nur dadurch erreichen, daß diese verborgen bleiben, eine oft

zitierte Regel fordert deshalb aus gutem Grund, keine globalen Variablen zu exportieren. Auf diese Weise ist eine Bearbeitung der Daten ausschließlich über exportierte Zugriffsprozeduren möglich, wie es in den bisherigen Modulschnittstellen gezeigt wurde. Derartige Zugriffsprozeduren sind sinnvoll und unumgänglich, wenn es darum geht Daten zu verändern, ein schreibender Datenzugriff kann damit unter Kontrolle gehalten werden. Das Verfahren ist allerdings in den Fällen unnötig aufwendig, in denen es ausschließlich um einen lesenden Datenzugriff geht. Zugriffsprozeduren für solche Zwecke erinnern an die Kanonen-Spatzen-Methode. Sowohl der Programmieraufwand als auch die durch den Prozeduraufruf entstehenden Laufzeitkosten lassen sich vermeiden

```
DEFINITION FIFOsPoly;

  TYPE
    Element = POINTER TO ABSTRACT RECORD
      NächstesElement-: Element
    END;

    Schlange = POINTER TO LIMITED RECORD
      ErstesElement-: Element;
      Elementzahl-: INTEGER;
      Anhängen: PROCEDURE (schlange: Schlange; element: Element);
      Entfernen: PROCEDURE (schlange: Schlange; OUT kopf: Element);
      Leer: PROCEDURE (schlange: Schlange): BOOLEAN;
      Anzeigen: PROCEDURE (schlange: Schlange)
    END;

  PROCEDURE NeueSchlange (OUT schlange: Schlange);

END FIFOsPoly.
```

Die vorstehende Schnittstelle des Moduls *FIFOsPoly* zeigt eine Implementation der Listenverwaltung (hier als Schlange statt als Stapel realisiert), in der gegenüber *FILOsPoly* alle Methoden für nur lesenden Datenzugriff fehlen. Statt dessen exportiert Modul *FIFOsPoly* neben den zur Klasse *FIFOsPoly.Schlange* gehörenden Methoden *Anhängen*, *Entfernen*, *Leer* und *Anzeigen* die ebenfalls zu dieser Klasse gehörenden Datenfelder *ErstesElement*, *Elementzahl* sowie das zur Klasse *FIFOsPoly.Element* gehörende Datenfeld *NächstesElement*. Die Minusmarken an den Bezeichnern weisen darauf hin, daß es sich um schreibgeschützte Exporte handelt, Klienten können nur lesend auf die Daten zugreifen. Datenfelder schreibgeschützt veröffentlichen zu können erhöht die Effizienz von Programmen und trägt auf elegante Weise dazu bei, sie transparent zu machen, ohne ihre Integrität und die Sicherheit der Daten zu gefährden.

Und nicht zuletzt erübrigt schreibgeschützter Export auch den manchmal fast erbittert geführten Streit, ob die Präsentation des nächsten Elements Aufgabe der Schlange ist oder aber dem jeweils aktuellen Element überlassen bleibt - ob also, wie im Modul *FILOsPoly*, die Prozedur *NächstesElement* eine Methode der Klasse *Element* oder ob sie, wie im Modul *FILOsADT*, eine Methode der Klasse *Stapel* sein sollte. An Stelle des Aufrufs einer kompletten Methode genügt dem Klienten bei schreibgeschütztem Export eine kurze Abfrage des in kanonischer Weise der Klasse *Element* zugeordneten Attributs *NächstesElement*.

Was ist eine Komponente - und was nicht

Obwohl der Begriff der Klasse seit langem existiert und in Bezug auf die wesentlichen Merkmale inzwischen weitgehend Einigkeit herrscht, gibt es auf Grund der spezifischen Eigenarten der jeweils zu Grunde liegenden Programmiersprachen Unterschiede und Meinungsverschiedenheiten, die im Zusammenhang mit Komponenten zunehmend wichtig werden. Einer der kontroversen Punkte in der aktuellen Diskussion ist die Frage, ob Klassen Komponenten sind oder nicht.

Auch wenn in seltenen Fällen eine Komponente aus einer einzelnen Klasse bestehen kann, die Komponente in einem derartigen Fall also aussieht wie eine Klasse, sind Komponenten keine "Über"-Klassen, vielmehr sind die Konzepte von Klassen und Komponenten zueinander orthogonal [Szy92]. Es ist möglich, daß eine Komponente Klassen enthält, aber es ist keineswegs notwendig. Ein Beispiel für klassenlose Komponenten sind die vielen, seit

langem existierenden mathematischen Bibliotheken, die oftmals in FORTRAN - einer gewiss klassenlosen Programmiersprache - geschrieben wurden, ein weiteres Beispiel sind VB-Controls, Kontrollelemente für graphische Benutzeroberflächen, die in VISUAL BASIC erstellt sind, einer Sprache, die in ihren Anfängen ebenfalls klassenlos gewesen ist.

Komponenten mit oder ohne Klassen und Objekte zu erstellen ist deshalb möglich, weil Komponenten als *plug-ins*, die mit ihrer Umgebung ausschließlich über Schnittstellen kommunizieren können, das Prinzip der Datenkapselung rigide befolgen müssen. In welcher Weise die Implementation der Schnittstellen erfolgt, ist einer Komponente in keiner Weise entnehmbar, sie kann in BASIC-Spaghetti oder in ASSEMBLER, in strukturiertem PASCAL, in einer klassenbasierten Sprache wie JAVA oder EIFFEL, aber auch in einer modularen Sprache wie ADA, OBERON, COMPONENT PASCAL oder C# erstellt worden sein. Weiterhin ist es möglich, daß eine Komponente aus mehreren, in verschiedenen Programmiersprachen und Programmierstilen erstellten (Sub-)Komponenten zusammengesetzt ist, die selbstverständlich untereinander ebenfalls nur über Schnittstellen kommunizieren können.

Damit entsteht die Frage, welche Softwarekonstrukte als Komponenten in Frage kommen. Szyperski nennt drei wichtige Kriterien, die ein Softwarebaustein erfüllen muß, um als Komponente gelten zu können [Szy02].

Eine Komponente
- ist eine selbständige Einheit
- wird von unabhängigen Nutzern verwendet
- besitzt keinen von außen wahrnehmbaren Zustand

Entsprechend dem ersten Kriterium muss eine Komponente von ihrer Umgebung abgeschirmt, sie muß gekapselt sein, eine Komponente kann folglich nur als Ganzes, nicht in Einzelteilen existieren. Die im zweiten Kriterium genannten unabhängigen Nutzer haben keinerlei Kenntnis der inneren Struktur einer Komponente, sie kennen ausschließlich die in der Spezifikation genannten Eigenschaften, den Kontrakt, der entsprechend präzise deklariert, welche Bedingungen die Komponente voraussetzt, um die ebenso klar im Kontrakt deklarierten Leistungen erbringen zu können.

Daraus resultiert, daß eine Komponente zwei Schnittstellen hat, eine auf der Eingangsseite (incoming interface) und eine auf der Ausgangsseite (outgoing interface). Die Ausgangsschnittstelle entspricht der von Klassen und Modulen gewohnten Schnittstelle, sie spezifiziert die den Klienten zugänglichen Leistungen, während die Eingangsschnittstelle die für das vertragsgemäße Funktionieren der Komponente notwendigen Voraussetzungen spezifiziert.

Eine derartige Spezifikation der Vorbedingungen ist an sich auch bei Klassen und Modulen nötig, wird aber selten explizit durchgeführt und es gibt bisher keine allgemein akzeptierte Darstellungsform. Bei Komponenten, die in einem von ihrer Erstellung unabhängigen Kontext verwendbar sein sollen, ist die Spezifikation zwingend erforderlich, ein Anwender, der die Wahl zwischen mehreren gleichwertigen Komponenten hat, wird sich zweifellos für diejenige entscheiden, die beide Seiten des Kontrakts am klarsten präsentiert.

Für die komponentenorientierte Programmerstellung ergeben sich aus den bisherigen Darlegungen zwei Konsequenzen. Zum Einen sollte der Entwickler eines komplexen Softwaresystems, das aus vielen Subkomponenten zusammengesetzt ist, bei der Wahl dieser Subkomponenten weitgehend ungebunden sein. Die Zielplattform der von ihm entwickelten Komponente sollte ihm in Bezug auf die jeweilige Programmiersprache der einzelnen Subkomponenten möglichst keine Vorschriften machen. In dieser Beziehung ist Microsofts neue *.Net*-Umgebung vorbildlich (näheres s. Kasten).

Zum Zweiten muß die Laufzeitumgebung einer Komponente prinzipiell offen sein, Komponenten sollen und müssen während des Programmbetriebs geladen und entfernt werden können, ohne daß es zu einer Störung der übrigen Programmabläufe kommt. Eine Segmentierung des Arbeitsspeichers, bei der in jedem Segment ein in sich geschlossenes und von der Außenwelt abgekapseltes Programm läuft, ist weder möglich noch sinnvoll.

Diese Offenheit hat ihren Preis - oder besser: ihre Vorzüge. Da sich eine globale Analyse des Datenraums, wie sie bei abgeschlossenen Programmen möglich und für die manuelle Speicherverwaltung nötig ist, in einer offenen Systemumgebung nicht durchführen lässt, erfordern Komponenten ein automatisches Speichermanagement - einen

Garbage Collector - ein Vorzug, den absturzgeplagte Programmierer und Anwender gleichermaßen schätzen werden. Außerdem muß die Systemumgebung dynamisches Laden und Binden von Komponenten unterstützen, auch in verteilten Netzen. Dies setzt voraus, daß eine Komponente die dafür notwendigen Metadaten enthält und es setzt voraus, daß die jeweilige Laufzeitumgebung diese Metadaten abrufen und auswerten kann.

JAVA versus .NET

Praktisch alle heutzutage existierenden Programme sind ohne eine Laufzeitumgebung, die ihnen die notwendigen Basisdienste zur Verfügunge stellt, nicht denkbar. So benötigen JAVA Programme das JAVA runtime environment (JRE), Windows Programme erforderten bisher COM oder COM+ als Hintergrund für ihre Ausführung, in Zukunft wird es .NET (gesprochen: dot net) mit der common language runtime (CLR) sein.

Ähnlich wie die JAVA runtime ist .NET eine Laufzeitumgebung für Programme, die in einer Metasprache vorliegen. Anders aber als bei der JAVA runtime, die nur Programme in JAVA-Bytecode verarbeitet, wobei dieser während der Ausführung interpretiert oder von einem just-in-time (JIT) Compiler übersetzt wird, was die Langsamkeit von JAVA-Programmen erklärt, sind .NET-Programme vollständig kompiliert und laufen entsprechend schnell.

Die Kompilation von .NET Programmen erfolgt wie bei JAVA in zwei Schritten. Im ersten Schritt wird der Quellcode, der in einer beliebigen der inzwischen ca. 30 auf .NET portierten Programmiersprachen erstellt sein kann, in CIL (Common Intermediate Language) übersetzt, dabei werden zahlreiche Metainformationen generiert, die für das Laden und Binden in der Laufzeitumgebung nötig sind. Gleichzeitig bieten Metainformationen die Möglichkeit, sich in jeder der .NET Sprachen die Schnittstellen der einzelnen Komponenten so anzeigen zu lassen, wie es in dieser Sprache üblich ist, der Anwender einer Komponente kann diese also verwenden, als ob sie in der von ihm benutzten Sprache entstanden wäre.

Im zweiten Schritt wird das Metakompilat aus CIL in den von der jeweiligen Hardware ausführbaren Code übersetzt. Ein .NET Programm ist also einerseits in (fast) jeder Programmiersprache erstellbar und kann andererseits - als CIL Code - für (praktisch) jede existierende Hardware kompiliert werden.

Von Klassen über Module zu Komponenten

Aus der dritten von Szyperski genannten Komponenteneigenschaft folgt unmittelbar, daß die nicht selten zu findende Identifizierung von Objekten als Komponenten (*Komponentenobjekt, Objektkomponente usw.*) grundsätzlich nicht möglich ist, jedes Objekt besitzt einen eindeutig definierten, von außen wahrnehmbaren Zustand, es kann folglich keine Komponente darstellen. Eine Komponente enthält auch keine Objekte (mit Ausnahme der erwähnten Prototypen oder Fabrikobjekte), sie kann und wird aber in vielen Fällen über Objekte, die während ihrer Aktivierungszeit erzeugt werden, mit ihrer Außenwelt kommunizieren.

Daraus ergibt sich die Frage, ob Klassen - als zustandslose Erzeugendenmuster von Objekten - Komponenten sein können. Bis auf exotische Ausnahmen ist eine einzelne Klasse als Komponente nicht geeignet, eine Klasse und die aus ihr erzeugten Objekte sind im allgemeinen auf das Zusammenwirken mit anderen Klassen und ihren Objekten angewiesen. Die angeführten Beispiele in den Modulen *FILOsPoly* und *FIFOsPoly* zeigen, daß schon in einfachen Fällen ein Software-Baustein, der in einer geeigneten Umgebung im Stil einer Komponente verwendet werden kann, aus mehr als einer Klasse zusammengesetzt ist, in der Regel wird eine funktionsfähige Komponente in gleicher Weise mehr als eine Klasse enthalten.

Diese Betrachtungsweise führt zu dem eingangs zitierten Parnas-Modell der modularen Abstraktion. Module können - als Kapseln für Klassen, aber auch für klassenlosen, konventionellen Programmiercode - Komponenten sein. Sie sind in einer geeigneten Systemumgebung (deren Spezifikation Bestandteil der Eingangsschnittstelle ist), *pluggable* und sie kommunizieren mit ihrer Umgebung ausschließlich über wohldefinierte Schnittstellen. In diesem Sinn kann ein Modul eine Komponente sein (s. dazu a. [Szy00]), die Module *FILOsPoly* und *FIFOsPoly* können

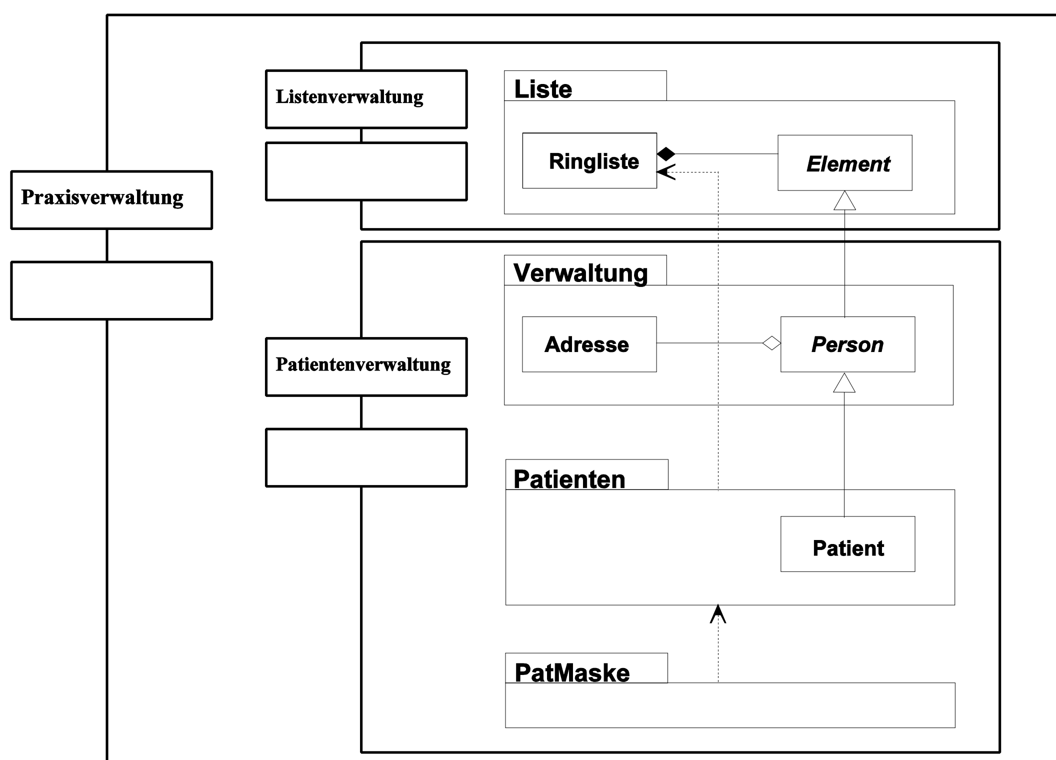
als Listenkomponenten angesehen werden, die die geordnete Verwaltung beliebiger Objekte ermöglichen. Module, wie sie in COMPONENT PASCAL oder MODULA-2 existieren (units in DELPHI, packages in ADA, assemblies in C#), erfüllen die für Komponenten genannten Bedingungen.

Einen wichtigen, bisher nicht genannten Aspekt von Komponenten unterstützen Module jedoch nicht, den Aspekt der externen Resourcen. Eine "ausgewachsene" Komponente enthält neben dem ausführbaren Code und den von der Laufzeitumgebung benötigten Metadaten weitere Resourcen, die für ihre Aktivierung erforderlich sind, beispielsweise Sprachdateien, mit denen die Eigenschaften der Komponente länderspezifisch angepaßt werden können, Dialogboxen für die Daten-Ein- und -Ausgabe usw. Eine Komponente wird also in der Regel mehr sein als ein Modul, obwohl Ähnlichkeiten bestehen, ebenso wie ein Modul in der Regel mehr ist als eine Klasse, obwohl auch zwischen diesen Ähnlichkeiten bestehen können.

Komponentenorientierte Programmierung im Informatikunterricht

Im folgenden sei am Ergebnis eines Informatikkursprojekts, in dem eine fiktive Praxisverwaltung entworfen und realisiert wurde, der Aufbau einer Komponente dargestellt.

In dem hier gezeigten Endzustand ermöglicht die Komponente *Praxisverwaltung* das Anzeigen, Anlegen, Ändern und Löschen einzelner Datensätze in einer Patientendatei, die auf einem externen Medium gespeichert ist, außerdem Durchblättern der Datei, Suchen eines Datensatzes sowie Anzeigen und Ausdrucken der gesamten Datei. (Eine ausführliche Projektdarstellung und die Entwicklungsergebnisse sind beim Autor erhältlich.)



Softwarekomponente *Praxisverwaltung*

Der Abbildung (diese verwendet UML-Notation - für eine Kurzdarstellung s. [HvL04], Anhang D; umfassender [Oose]) lässt sich entnehmen, daß die Komponente *Praxisverwaltung* aus zwei inneren Komponenten - *Listenverwaltung* und *Patientenverwaltung* - besteht. Benötigte globale Resource (incoming interface) der *Praxisverwaltung*

ist das Komponentengerüst (component framework) *BlackBox*. Nicht in der Abbildung gezeigt sind die oben erwähnten sonstigen Ressourcen wie Sprach- und Menüdateien, Dialogboxen etc[1].

Die innere Komponente *Listenverwaltung* besteht aus dem Modul (package) *Liste* als Kompositum der limitierten Klasse *Ringliste* und der abstrakten Klasse *Element*. Das Modul *Liste* ähnelt im Aufbau dem Modul *FIFOsPoly*, die eigentliche Listenverwaltung ist jedoch als ringförmige, doppelt verzeigerte Liste implementiert. Das Modul *Liste* ist entsprechend den von Szyperski genannten Kriterien eine Komponente, es besitzt als Modul keinen von außen wahrnehmbaren Zustand, es ist unabhängig von der Entwicklung der *Praxisverwaltung* entstanden und es wird von Nutzern verwendet, die von ihm keinerlei Kenntnis außer seiner Ausgangsschnittstelle haben

```
DEFINITION Liste;

    TYPE
        Element = POINTER TO ABSTRACT RECORD
            NächstesElement-, VorigesElement-: Element;
            . . .
        END;

        Ringliste = POINTER TO LIMITED RECORD
            ErstesElement-, LetztesElement-: Element;
            Elementzahl-: INTEGER;
            (Liste: Ringliste) Einfügen (element: Element; Schlüssel: LONGINT), NEW;
            (Liste: Ringliste) EntferntesElement (Schlüssel: LONGINT): Element, NEW;
            (Liste: Ringliste) GesuchtesElement (Schlüssel: LONGINT): Element, NEW;
            (Liste: Ringliste) InitElement (element: Element), NEW;
            . . .
        END;

    PROCEDURE NeueListe (OUT liste: Ringliste);

END Liste.
```

Auf ein für Komponenten charakteristisches Detail der vorstehenden Schnittstelle möchte ich hinweisen. Die vollständige Schnittstelle enthält mehr als die hier zu sehenden Exporte, gezeigt sind lediglich die von der verwendenden Komponente *Patientenverwaltung* benötigten Dienste, es handelt sich also - genau genommen - nicht um die Ausgangsschnittstelle der *Listenverwaltung*, sondern lediglich um ihren von der Komponente *Patientenverwaltung* verwendeten Teil, deren Eingangsschnittstelle. Eine solche Auswahl ist für Komponenten typisch, eine verwendende Komponente wird im konkreten Einzelfall fast immer nur eine Teilmenge der angebotenen Dienste nutzen.

Diese Tatsache verweist auf einen der wesentlichsten Aspekte von Komponenten, ihre Wiederverwendbarkeit. Eine Komponente ist wiederverwendbar, wenn sie wiederverwendet wird (a component is reusable if it is reused). Um wiederverwendet zu werden, muß eine Komponente hinreichend viele verschiedene Dienste anbieten, darf aber andererseits nicht überladen sein, das "Gewicht" einer Komponente (component weight) ist eine für ihren Erfolg kritische Größe.

Für die Listenkomponente gelten alle im Zusammenhang mit dem Modul *FIFOsPoly* genannten Eigenschaften, die Klasse *Ringliste* ist limitiert, Klienten können via Konstruktor ein Listenobjekt erzeugen - das Modul *Patienten* unterhält eine entsprechende Beziehung zur Klasse *Ringliste* - und die in der Schnittstelle gezeigten Dienste abrufen. Außerdem können sie Unterklassen der abstrakten Klasse *Element* deklarieren - in der Komponente *Patienten-*

[1] Einige Leser werden vielleicht in der Darstellung eine Fensterklasse vermissen, wie sie in vielen anderen Systemumgebungen für die Ein- und Ausgabe der Daten via Dialogboxen nötig ist. Derartige Klassen und die häufig für ihre korrekte Verwendung benötigten umfangreichen Detailkenntnisse sind im *BlackBox*-Framework nicht erforderlich, die entsprechenden Dienste sind vollständig in den *BlackBox*-Systembibliotheken gekapselt.
Wer sich dafür und für das damit zusammenhängende MVC-Muster sowie für die resultierende, sehr viel einfachere Erstellungsweise von Dialogboxen und Textfenstern im *BlackBox*-Framework interessiert, findet Darstellungen in [HvL99], [HvL04] oder [Pfi01].

verwaltung sind dies die abstrakte Klasse *Person* und die von ihr abgeleitete konkrete Klasse *Patient*, Objekte dieser Klasse können polymorphisch von einem Listenobjekt verwaltet werden. Dies zeigt einen anderen wesentlichen Aspekt von Komponenten, obwohl Komponenten rigide gekapselt sind, können Abhängigkeiten über Komponentengrenzen hinweg durch Aggregationen und Vererbungsbeziehungen entstehen.

Die Komponente *Patientenverwaltung* zeigt neben der Tatsache, daß eine Komponente häufig aus mehr als einem Modul besteht, einen weiteren wichtigen Aspekt: nicht jedes Einzelmodul ist eine sinnvolle Komponente. Die Aufspaltung der *Patientenverwaltung* in drei Module entstand im Rahmen der Projektarbeit aus der Erkenntnis, daß eine verallgemeinerte Version der Software nicht nur die Verwaltung von Patienten sondern auch anderer Personen erlauben sollte, die abstrakte Klasse *Verwaltung.Person* ermöglicht dies. Anders aber als im Fall des hinreichend generellen Moduls *Liste* ist die Funktionalität des Moduls *Verwaltung* trotz seiner Abgeschlossenheit nicht umfassend genug, um es als Komponente ansehen zu können, obwohl das strukturell möglich wäre. Noch deutlicher wird dies an den beiden übrigen Modulen der *Patientenverwaltung* - *Patienten* und *PatMaske*. Sie sind - trotz modularer Abgeschlossenheit - untereinander und mit dem Modul *Verwaltung* so eng verzahnt, daß sie definitiv keine Komponenten sein können.

Das Modul *Patienten* stellt den eigentlichen Kern der Komponente *Patientenverwaltung* dar, es enthält bis auf die von der Listenkomponente gelieferten Dienste die gesamte in der Projektbeschreibung dargestellte Funktionalität der Patientenverwaltung, auf deren Details soll hier jedoch nicht weiter eingegangen werden, da sie für die Erläuterung der Komponentenstruktur der *Praxisverwaltung* nicht wesentlich sind.

Dagegen besteht ein wichtiger Aspekt des modularen Aufbaus der Komponente *Patientenverwaltung* in der Tatsache, daß das Modul *Patienten* nicht direkt mit der Aussenwelt kommuniziert. Diese Aufgabe fällt dem Modul *PatMaske* zu, das programmtechnisch die Ausgangsschnittstelle (outgoing interface) der gesamten Komponente *Praxisverwaltung* enthält. Auf eine Erläuterung des Moduls kann hier aus den genannten Gründen ebenfalls verzichtet werden.

Da es sich bei der Ausgangsschnittstelle im Fall der *Praxisverwaltung* um die Schnittstelle zu einem humanen Klienten (Nutzer) handelt, sei diese abschließend in der angemessenen Form als Dialogbox gezeigt

Literatur

[BrS02] Broy, M.; Siedersleben. J.: Objektorientierte Programmierung und Softwareentwicklung. Informatik Spektrum, Band 25, Heft 1, Februar 2002

[GHJ98] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Reading, Addison-Wesley, 1998

[Gru95] Gruntz, D.: Objects are not enough. The Oberon Tribune, Vol 1, No 2, 1995

[HvL99] von Lavergne, H.: Visuelle Welt - Schön. The missing link. LOG IN 19. Jg (1999), H. 5. (s. auch: http://www.lahini.de/)

[HvL04] von Lavergne, H.: Component Pascal. Ein Tutorium, 2004 (http://www.lahini.de/)

[Mey86] Meyer, B.: Genericity versus inheritance. Proceedings OOPSLA '86, SIGPLAN Notices, vol. 21 (11), 1986

[Oose] http://www.oose.de/downloads/uml-2-Notations-uebersicht-oose.de.pdf

[Par72] Parnas, D. L.: A Technique for Software Module Specification with Examples. CACM 15 (12), 1972

[Pfi01] Pfister, C.: Component Software: A Case Study Using BlackBox Components. Oberon Microsystems, 2001

[SD-M] Meyer, B.; Szyperski, C. u. a.: Beyond Objects. Kolumne der Zeitschrift SD-Magazine, 1999 - 2002. (http://www.sdmagazine.com/)

[Szy92] Szyperski, C.: Import is not inheritance - Why we need both: Modules and Classes. Proceedings, Sixth European Conference on Object Oriented Programming (ECOOP '92), Utrecht, The Netherlands. Springer Verlag, Lecture Notes in Computer Science No. 615, 1992

[Szy00] Szyperski, C.: Modules and Components - Rivals or Partners? Böszörmenyi, L., Gutknecht, J., Pomberger, G. (Hrg.): The School of Niklaus Wirth. dpunkt.verlag, Heidelberg, 2000

[Szy02] Szyperski, C.; Gruntz, D.; Murer, S.: Component Software - Beyond Object-Oriented Programming. 2nd edition. Addison-Wesley, Harlow, UK, 2002

[Ude94] Udell, J.: Componentware. BYTE Magazine, 19(5), 1994

Erschienen in:
LOG IN 131/132, 2004