Zu den wichtigsten Neuerungen im Bereich der Programmiertechniken und Softwaretechnologie gehört, als Weiterentwicklung der prozeduralen Programmierung, die objektorientierte Programmierung. Der folgende Artikel stellt anhand eines einfachen Programmbeispieles das objektorientierte Paradigma vor.

Objektorientierte Programmierung

Objektorientierung ist zuallererst eine Methode zu programmieren und deshalb ein Werkzeug für Entwickler. Der entscheidende Vorteil der Objektorientierung liegt in der Wiederverwendbarkeit von Software. Durch Modularität kann bestehende Software ohne erneutes Kompilieren und Testen in neuen Programmen genutzt werden, bestehende Programme können, ohne die Gesamtstruktur zu belasten, durch Ändern einzelner Softwarebestandteile weiterentwickelt werden. Andere Vorteile, wie z. B. eine saubere Strukturierung, sind zwar bedeutsam, aber nicht maßgebend. Für die Softwareentwicklung bedeutet Objektorientierung also Zeitgewinn und damit Einsparen von Kosten. So wird sie auch verstanden, wenn sie z. B. zum Überwinden der Softwarekrise angeboten wird.

Anwendern hingegen, denen normalerweise verschlossen bleibt, wie eine Software programmiert ist, bietet ein objektorientiertes Programm keine direkten Vorteile. Dennoch wird auch hier der Begriff »Objektorientierung« inflationär verwendet: Wunderdinge können (mal wieder) erwartet werden. Man sollte sich davon nicht beeindrucken lassen. Es ist eigentlich egal, wie ein Programm entwickelt wurde, wenn es nur einwandfrei funktioniert.

DER AUTOR

Werner Lonsing, Jahrgang 1962, Stuium der Kunstgeschichte und Architektur, Mitarbeit an Forschungsprojekten der FH Dortmund und TH Darmstadt, beschäftigt sich seit 2 Jahren mit der objektorientierten Programmierung unter Nextstep.

Die Denkweise

Unabhängig davon lassen sich zwei Beobachtungen machen: Einerseits gibt es nach wie vor kaum vollständig objektorientiert entwickelte Programme, andererseits werden viele Programme, die schon immer mit Datensätzen arbeiteten, rückwirkend als objektorientiert deklariert.

Unter Programmierern ranken sich zwei Legenden um die objektorientierte Programmierung. Die erste ist, daß es Jahre dauert, bis jemand die Grundsätze einigermaßen verstanden hat. Die zweite besagt, daß objektorientiertes Programmieren nichts anderes als sauberes prozedurales Programmieren ist.

Beides ist nicht zutreffend. Die Objektorientierung ist vor allem eine neue Denkweise zum Lösen von Problemen, ein anderer Standpunkt. Als solche fordert sie, natürlich, ein gewisses Umdenken. Das ist aber bei weitem nicht so schwierig, wie oft dargestellt.

Der Ansatz

Der objektorientierte Denkansatz verspricht Softwareentwicklern, schnell und einfach Programme zu erstellen, die leicht gelesen und verstanden werden können. Dazu werden die Programme in Teile aufgelöst, die einzeln programmiert werden. Programmierte Objekte funktionieren relativ autark und sind prinzipiell stabiler als Programme. Fertigprogrammiert sind sie einfach zu handhaben und können in mehreren Programmen verwendet werden.

Die Versprechen werden an vier Merkmale gebunden:

Modularität

Programme werden in kleine, leicht verständliche Teile gegliedert, die einfach zu kombinieren sind.

Eindeutigkeit

Jeder Parameter und jedes Programmfragment taucht einmal und nur einmal in einem Programm auf.

Kapselung

Informationen in einem Programm werden dort behalten, wo sie auch manipuliert werden.

Abstraktion

Details der Implementierung bleiben anderen Teilen des Programms verborgen, so daß Detailänderungen keinen Einfluß auf das Programm haben. Die Teile sind abstrakt.

Dem Anspruch wird dadurch genügt, daß Vereinbarungen getroffen werden, die syntaktisch kaum ins Gewicht fallen, die aber, weil eine andere Denkweise dahintersteht, die Struktur eines Programmes ändern und damit dem Programm selbst ein anderes Erscheinungsbild geben.

Objekte und Klassen

Wie der Name vorgibt, bestehen objektorientierte Programme aus Objekten. Ein Objekt ist eine selbstständige Programmiereinheit. Es enthält einen Satz Daten in einer fixen Struktur und die zur Bearbeitung der Daten erforderlichen Operationen, die Methoden. Unterschieden werden Objekte nach Klassen. Eine Objektklasse ist ein Prototyp für Objekte mit gleicher Datenstruktur und gleichen Methoden. Ein einzelnes Objekt einer Klasse wird als Instanz bezeichnet, die Daten, die es enthält, als Instanzvariablen. Da die Variablen eines Instanzobjektes ausschließlich vom Objekt selbst genutzt werden und dem Rest des Programms verborgen bleiben, können andere Objekte Da-

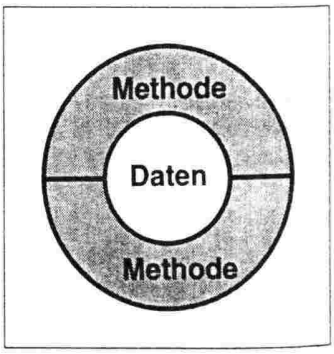


Abb. 1: Ein Objekt: Die Daten sind vollständig von Methoden umschlossen

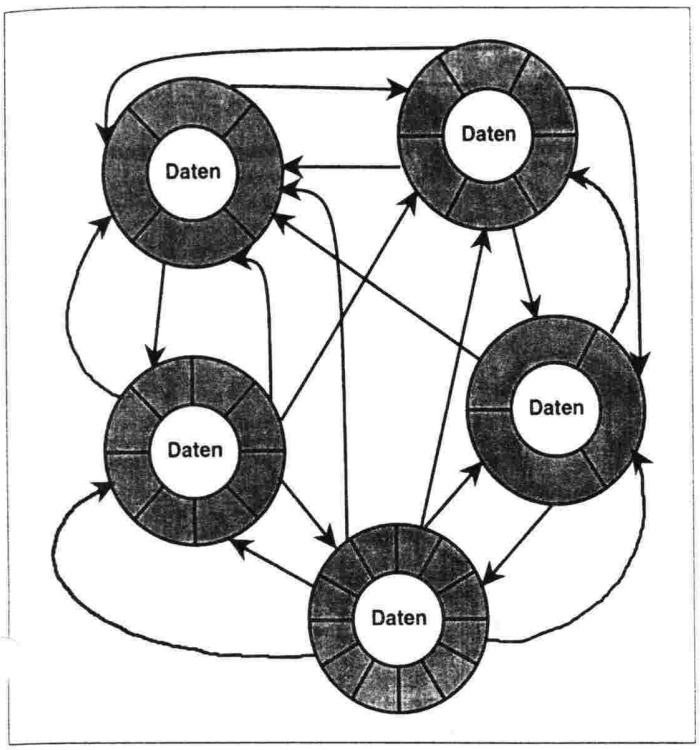


Abb. 2: Nachrichten zwischen Objekten

ten nur mit dafür implementierten Methoden erhalten. Diese Eigenschaft von Objekten bezeichnet man als Kapselung (Abb. 1). Ebenso können innerhalb eines Objektes nur klassenspezifische Methoden eingesetzt werden. Methoden anderer Klassen sind nicht zugänglich und können auch irrtümlich nicht aufgerufen werden.

Nachrichten

ypischerweise besteht ein objektorientiertes Programm aus einer Anzahl miteinander verbundener Objekte, die sich gegenseitig zum Lösen isolierter Teilaufgaben aufrufen (Abb. 2). Der Aufruf eines Objektes durch ein anderes wird, unabhängig davon, ob Daten transferiert werden, als Nachricht (engl.:message) bezeichnet. Um Nachrichten zu senden, haben Objekte Instanzvariablen zum Speichern von Adressen der Objekte, die Nachrichten erhalten sollen. Diese Verbindungen der Objekte untereinander bestimmen ganz wesentlich die Struktur eines Programmes.

Denken in Objekten

Ein großer, impliziter Vorteil der Objektorientierung ist das Verständnis von Objekten als agierenden oder handelnden Objekten. In einem prozeduralen Programm, in dem Daten und Funktionen überall verstreut werden können, werden Fragen nach dem Programmablauf abstrakt formuliert: »In welcher Prozedur wird was erledigt?«, und: »Wo sind die Daten?« In einem Programm, in dem Objekte kleine Einheiten mit Daten und Funktionen bilden, kann subjektivierend formuliert werden: »Das Objekt macht das, und ein anderes Objekt macht jenes.«. Eine entsprechende Frage lautet dann: »Wer macht das?« Hinzu kommen sehr plastische Vorstellungen von Objekthierarchien: »Wer ist hier der Chef?«, oder: »Darf der das?«

Die Subjektivierung von Objekten ist eine wertvolle Denkhilfe, mit der komplexe Vorgänge vereinfacht dargestellt werden können. Die Darstellungsweise hat aber ihre Grenzen, und sie ist falsch. Wenn in den folgenden Zeilen trotzdem vereinfacht »Objekte handeln«, ge-

schieht das mit dem Wissen, daß es nicht so ist.

Ein kleines Beispiel

Eigentlich alle Situationen können objektorientiert beschrieben werden.

Eine Hand hält ein Glas. Sie läßt es los. Das Glas fällt herunter und zerbricht.

Es gibt die Klassen `'Hand'` und `'Glas'`. Eine Instanz von `'Hand'` mit der Variable `'(object)etwas'` und den Methoden `'greife: und Halte:'` und `'lasseLos'` ist `'meineHand'`, von `'Glas'` mit `'gravitiere:'` und `'zerbreche:'` ist es `'meinGlas'`. Von außen kommt die Nachricht `'[meineHand greife:meinGlas und-Halte:(BOOL)fest]'`. Die Hand 'hält' `'meinGlas'`. Irgendwann kommt die Nachricht `'[meineHand lasseLos]'`, und die Hand 'läßt' das Glas los.

Nun ist das Glas an der Reihe. Es erhält die Nachricht `'gravitiere:'` und sendet sich daraufhin selbst `'zerbreche:'` mit `'[self löscheMich]'`. Das Glas als logisches Objekt existiert nicht mehr. Die Nachrichten und Aktionen sind schlüssig und entsprechend dem physikalischen Vorbild logisch definiert, es funktioniert so.

Man kann, wenn es weitergehen soll, `'zerbreche:'` erweitern mit `'[self alles-Scherben]'`. In der Methode würden einige Instanzen der Klasse `'Scherbe'` gebildet und sinnvoll parametrisiert. Die Klasse `'Hand'` würde um die Methode `'tutWeh:'` mit `'[self lasseLos]'` erweitert, sie würde in `'greife:undHalte'` entsprechend dem Parameter `'fest'` aufgerufen. Der Funktionalität sind keine Grenzen gesetzt.

Genauso können die Klassen erweitert werden: `Hand`, wenn es sinnvoll ist, in `LinkeHand` und `RechteHand`, `Glas` in `Weinglas`, `Sektglas`, `Plastikglas`, `Zahnputzglas`, `Bierglas` usw. Entsprechend spezifizierende Methoden sind

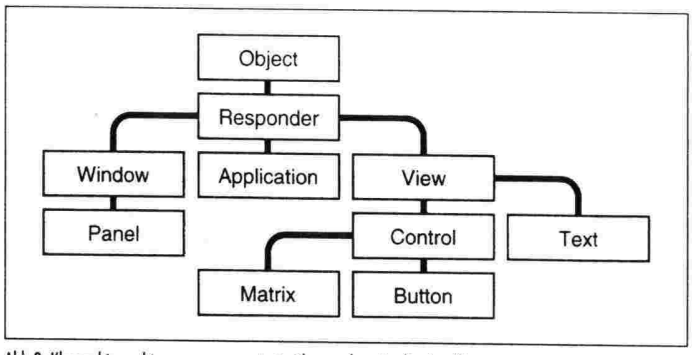


Abb.3: Klassenhierarchie: vorprogrammierte Klassen des »Application-Kit«

denkbar, in `Plastikglas` müßte z. B. `zerbreche:` modifiziert werden.

Was aber, wenn `meineHand` `meinGlas` auf einem `Tisch` losläßt, `meineHand` aber nicht 'weiß', daß ein `Tisch` unter dem `Glas` ist, es auch nicht zu wissen braucht. Für `meinGlas` gilt dasselbe. Die vollständigste Lösung wäre, `Glas`, `Tisch` und `Boden` dreidimensional zu definieren und in `gravitiere:` alle in Frage kommenden Objekte auf eine Kollision nach unten überprüfen, die Aufprallgeschwindigkeit nach Entfernung und Erdbeschleunigung rechnen und danach eventuell `zerbreche:` aufrufen.

Es ist dabei ein Unterschied, ob logische oder physikalische Objekte betrachtet werden. Das Verhalten logischer Objekte wird vollständig durch die Programmierung determiniert, nicht durch die Physik. Wenn logische Objekte zufälligerweise physikalischen ähnlich sind, ist das ein dankenswerter Umstand, der beim Betrachten eines Problemes hilft und zur Lösung führen kann. Er ist aber nicht die Lösung. Gerade bei der Objektorientierung führt Leichtfertigkeit in der Betrachtungsweise schnell zu falschen Ergebnissen.

Es ginge also auch anders: `meinGlas` könnte mit der Methode `[self alleAchtung: self]` die anderen Objekte benachrichtigen, die daraufhin selbst prüfen, ob sie für eine Kollision in Frage kommen. Solche Konstruktionen werden häufig vorgestellt, um daran zu demonstrieren, daß Objekte faktisch alles können, wenn sie nur entsprechend programmiert sind. Sie unterschlagen jedoch wichtige Details. Entweder »kennt« `meinGlas` alle Objekte und sendet allen eine Nachricht oder es gibt andere, abstrakte Objekte zur Steuerung. Die erste Vorstellung ist unrealisitsch, da so allen Objekten alle anderen bekannt sein müssen. Mit jedem neuen Objekt würde der Speicherbedarf exponential ansteigen. Im zweiten Fall müssen alle am Szenario beteiligten Objekte unbedingt vorgestellt werden, sonst werden die Objekte als omnipotent charakterisiert. Das ist schlichtweg falsch und läuft auf die Irreführung der Anwender hinaus.

Und abschließend: Was passiert, wenn in `zerbreche:` stünde: `[meinAutofahrLos]`? Wer macht da was mit wem?

Eine andere Frage ist: Welche Rolle spielt der Rechner? Logische Objekte haben irgendwo auch eine physikalische Repräsentation. Vielleicht anders gefragt: Was hat der Bildschirm damit zu tun, wenn auf ihm ein Glas zerbricht? Es ist beim objektorientierten Programmieren zwingend erforderlich, auch die Ein- und Ausgabe zu berücksichtigen und damit den Rechner. Er muß mit Objekten (nicht: 'als') in die objektorientierte Welt eingebunden werden.

Dieser Umstand wird leider häufig unterschlagen, und er ist mit ein Grund dafür, daß viele Leute von der Objektorientierung wissen, aber selten Kenntnisse der Funktionsweise haben. Deshalb wird, bevor die objektorientierte Programmierung am Beispiel vorgestellt wird, die verwendete Programmiersprache Objective-C und das dazugehörige Betriebssystem Nextstep erläutert. Auf diese Umgebung beziehen sich, auch wenn sie allgemeingültigen Charakter haben, die zur Erläuterung notwendigen Definitionen.

Programmiersprache

Objective-C ist, soweit sich das beurteilen läßt, eine der einfachsten der auf C basierenden objektorientierten Programmiersprachen. Die Erweiterungen zum ANSI-C Standard beschränken sich auf die Syntax zum Definieren von Klassen und zum Übermitteln von Nachrichten. Sie ermöglichen eine vollständig objektorientierte Programmierung einschließlich dynamischer Typisierung und Laufzeitbindung ohne Verzichte der Funktionalität einer hohen Programmiersprache.

Nachrichten

Auffälligste Neuerung ist das Versenden einer Nachricht: `[einObjekt tuWas]`. In eckige Klammern wird dabei zuerst das Objekt `'einObjekt'` genannt und dann die Methode `'tuWas'` aufgerufen. Optionale Parameter werden durch angefügte Doppelpunkte übergeben: `[einObjekt machWas:daraus]`.

Datentypen

Einige Datentypen sind neu definiert. Die wichtigsten sind `id`, ein Zeiger auf die Datenstruktur eines Objektes, `Class`, ein Zeiger auf die Klassenstruktur eines Objektes, und `SEL`, ein Zeiger auf eine Methode.

Objekte

Ein Objekt in einem Programm wird durch eine Variable, einem Zeiger vom Typ `id` repräsentiert und damit aufgerufen. Der Zeiger kann wie alle Zeiger in C zugewiesen werden. Der Datentyp `id` enthält keine weiteren Informationen über ein Objekt.

Klassen

Jedes Objekt hat eine Instanzvariable vom Typ `Class`, die

auf seine Objektklasse zeigt. Die Variable wird bei der Initialisierung des Objektes gesetzt, und, solange das Objekt logisch existiert, nicht mehr geändert.

Methoden

Methoden, die sich ansonsten nur unwesentlich von herkömmlichen Prozeduren in C unterscheiden, werden indirekt während des Programmlaufes mit dem Datentyp `SEL` (= `@selector`, eine Compiler-Direktive) adressiert. Der Datentyp `SEL` kann nur als Konstante zugewiesen werden.

Definieren der Klassen

In Objective-C werden Objekte durch die Definition ihrer Klasse definiert. Die Methoden werden implementiert und zusammen mit den Instanzvariablen deklariert. Von der kompilierten Version des Objektes, dem Klassenobjekt, werden die Instanzen der Klasse gebildet. Ein Objekt in einem Programm funktioniert erst, wenn es als Instanz einer Klasse gebildet ist.

Klassenhierarchie

Alle Klassen in Objective-C stammen von einer einzigen Klasse, `'Object'`, ab. Sie werden als hierarchischer Baum dargestellt (Abb. 3). Jede Klasse, außer `'Object'`, hat eine Superklasse, der sie entstammt, und jede Klasse kann Superklasse für eine beliebige Anzahl von Unterklassen werden. Eine neu definierte Klasse muß Unterklasse einer bestehenden Klasse sein und wird durch Deklarieren der Superklasse in die Hierarchie eingeordnet.

Vererbung

Wird eine Instanz von einem Klassenobjekt neu gebildet, erhält es die dort deklarierten Variablen und Methoden. Darüber hinaus erhält es die Methoden und Variablen der Superklasse seines Klassenobjektes und aller weiteren Klassen der Hierarchie einschießlich der Wurzelklasse. Jede Klasse erbt alle Variablen und Methoden ihrer Superklasse. Die Instanz einer Klasse ist immer auch Instanz der Superklasse.

Es gibt zwei kleine Ausnahmen. Als `'static'` oder `'private'` deklarierte Instanzvariablen werden nicht vererbt, und Methoden können in einer Unterklasse überschrieben werden. Das geschieht einfach durch Verwenden des gleichen Namens. Die Methode kann erweitert, ersetzt oder auch stillgelegt werden. Bei verschiedenen Instanzen einer Superklasse wird so die gleiche Methode unterschiedlich implementiert, eine Möglichkeit, die mit dem Polymorphismus interessant ist.

abstrakte Superklassen

Oft ist es sinnvoll, eine Klasse zu definieren, die als Superklasse für eine Reihe sehr ähnlicher Objekte fungiert, deren wesentliche Merkmale aber in spezifischen Unterklassen definiert sind. Die Instanz einer abstrakten Superklasse, z. B. `'Object'`, ohne Erweiterung ist sinnlos.

Delegierte

Wenn die Funktionalität einer Klasse sehr variabel sein muß und nicht durch neue Klassen erweitert werden soll, können Methoden eines Objektes einem anderen delegiert werden. An den Aufrufen anderer Objekte

ändert sich dadurch nichts. Das delegierte Objekt kann geändert werden, ohne das delegierende Objekt zu beeinflussen.

Wurzelklasse

Obwohl strenggenommen keine sprachliche Eigenschaft, muß die Wurzelklasse `'Object'` doch hinzugefügt werden, weil alle neuen Klassen Erweiterungen von ihr sind. Eine Programmierung in Objective-C außerhalb dieser Klasse ist unmöglich. Hier sind alle Variablen deklariert, die jedes Objekt als Objekt benötigt, z. B. der Zeiger auf die Klasse und die Methoden zur Verwaltung von Objekten wie `new:` oder `free:`.

Aktionen

Eine Aktion ist die infolge einer Nachricht ausgeführte Methode eines Objektes. Aktionen verlaufen immer nach dem Schema: Nachricht – Aktion – Nachricht.

Laufzeitbindung

Anders als in einer prozeduralen Programmiersprache, bei der eine Funktion, ihre Argumente und ihr Aufruf während der Kompilierung verbunden werden, wird der Aufruf einer Methode mit einem empfangenden Objekt erst verbunden, wenn im Programmablauf eine Nachricht gesendet wird. Dieses Verhalten objektorientierter Programme wird als Laufzeitbindung bezeichnet.

Kapselung

Wird eine Methode durch eine Nachricht aufgerufen, hat sie nur Zugriff auf die Instanzvariablen des Objektes, dem sie angehört. Informationen des sendenden Objektes können als Parameter übergeben werden. Informationen anderer Objekte müssen via Nachricht abgerufen werden.

Dynamische Typisierung

Da der Zeiger auf ein Objekt keine Informationen über das Objekt enthält, insbesondere nicht, welcher Art es ist, muß das Objekt sie selbst liefern. Dafür hat jedes Objekt eine Methode, die den Klassenzeiger abgibt. Wenn innerhalb des Programms Informationen über die Spezifikation eines Objektes benötigt werden, wird diese Methode aufgerufen. Objekte werden so erst während des Programmlaufes typisiert.

Polymorphismus

Wie schon erwähnt, kann ein Objekt nur die in seiner Klasse definierten Methoden verwenden, selbst wenn in einer anderen Objektklasse eine Methode mit gleichem Namen definiert ist. Das bedeutet, daß zwei Objekte auf den Aufruf einer Methode unterschiedlich reagieren. Diese Eigenschaft, die im Design eines objektorientierten Programms eine wichtige Rolle spielt, wird als Polymorphismus bezeichnet.

Definieren einer Klasse

Objektorientiertes Programmieren besteht größtenteils aus dem Schreiben des Quelltextes für neue Objekte, dem Definieren neuer Klassen. In Objective-C wird eine Klasse in zwei Teilen definiert, einem Deklarationsteil, in dem die Methoden und Variablen deklariert sind und die Superklasse benannt wird, und einem Implementierungsteil für den eigentlichen Programmtext, die implementierten Methoden. Beide Teile werden üblicherweise als separate Dateien angelegt, ebenso, wie für jede neu zu definierende Klasse eigene Dateien angelegt werden sollten.

Der Deklarationsteil wird 'Interface' genannt, weil er von allen Objekten benutzt wird, die die Klasse benutzen. Von jeder Klasse, die von Objekten einer neu zu definierenden Klasse Nachrichten erhalten soll, müssen die Deklarationen bekannt sein. Deshalb müssen die Interfacedateien aller auch nur erwähnten Klassen ähnlich wie Standardbibliotheken importiert werden.

Die Methoden dagegen bleiben anderen Objekten verborgen, weshalb die Implementierung nicht offengelegt zu werden braucht. Es ist bei Next gängige Praxis, zu den mitgelieferten Klassenobjekten nur die Interfacedateien, nicht aber die Methoden als Quelltext beizufügen.

Der objektorientierten Programmierung entspricht diese Trennung von Interface und Implementierung: Sind einmal die Interaktionen eines Objektes in einem Programm determiniert, die Methoden also vollständig deklariert, kann die Implementierung weiter verändert werden, ohne eine andere Klasse, also einen anderen Programmteil zu beeinflussen.

Wie alles funktioniert

Aus den Quelltexten werden vom Übersetzer die Klassenobjekte erzeugt. Wie in C aus den Funktionen werden aus den Methoden lauffähige Prozeduren gebildet. Dazu wird in jedem Klassenobjekt ein Zeiger auf die Superklasse mitgeführt und ein 'dispatch table' erstellt, in dem die Namen der Methoden mit Zeigern auf die Prozeduren assoziiert sind. Nachrichten, z. B. `[einObjekt machWas:daraus]`, werden zu einem nur dem Compiler möglichen Aufruf der Funktion `obj-msgSend ()` konvertiert, entsprechend dem Beispiel: `obj-msgSend( einObjekt, machWas, daraus)`. Die Instanzvariablen werden zu den in C bekannten Strukturen umgeformt.

Beim Programmstart werden die Klassenobjekte wie die Prozeduren eines prozeduralen Programmes als Laufteil geladen. Von den Objekten werden dann nach Bedarf Instanzen gebildet, d. h., von einer Methode zum Bilden von Instanzen, die jedes Klassenobjekt hat, wird für die Datenstruktur Speicherplatz beschafft und der Zeiger auf die Struktur übergeben. Der Zeiger ist vom Typ `id`. Die erste Komponente der Struktur, der Zeiger auf die Klasse, wird auf das herstellende Klassenobjekt ausgerichtet. Weitere Komponenten der Struktur bzw. Instanzvariablen werden objektspezifisch initialisiert.

Beim Senden von Nachrichten wird die Funktion `obj-msgSend ()` aufgerufen. Zuerst findet sie anhand der Klassenobjekte die Prozedur, also die Methode des empfangenden Objektes, auf das sich die Nachricht bezieht. Danach wird die Prozedur mit der Datenstruktur ihres Objektes und eventuellen Parametern aufgerufen. Abschließend liefert die Funktion das Resultat der Prozedur als eigenes Resultat an das aufrufende Objekt zurück.

Objektorientierte Programme haben wegen ihrer Struktur ein etwas ungewöhnliches Laufzeitverhalten. Weil anfänglich nur die allernotwendigsten Objekte initialisiert werden, ist ein Programm rasch präsent. Dafür muß jedes neue Objekt neu geladen werden, was etwas Zeit kosten kann. Darüber hinaus wird das Laufzeitverhalten durch Speichern benutzter Methoden in einem Cache und anderer Kleinigkeiten optimiert. Die Programme laufen sich nach allen Regeln erst einmal warm.

Historie

Smalltalk

Die objektorientierten Syntaxerweiterungen von Objective-C sind Smalltalk entlehnt, einer Programmiersprache, an und mit der viele Konzepte der Objektorientierung entwickelt wurden. Die Sprache entstand Ende der sechziger Jahre/Anfang der siebziger Jahre und ist heute in mehreren Dialekten verbreitet.

Entwicklung

Entwickelt wurde Objective-C ursprünglich von Stepstone für einen zweistufigen Compiler. Die erste Stufe übersetzt den Quelltext in reines C, das in der zweiten Stufe von jedem C-Compiler in Maschinencode übersetzt werden kann. Damit ist die Portierbarkeit auf jedes C unterstützende System gewährleistet.

Systemspezifische Komponenten, wie z. B. eine grafische Benutzeroberfläche, werden so aber nicht unterstützt. Der Objective-C-Compiler von Next ist deswegen einstufig und realisiert den Zugriff auf systemspezifische Komponenten über vorprogrammierte Klassen, außerhalb dieser Welt kann er nicht genutzt werden.

Die beiden Implementierungen unterscheiden sich also, anders, als man vielleicht vermutet, in ihren Klassen, insbesondere der jeweiligen Wurzelklasse. Unterschiede der Syntax sind demgegenüber nicht evident.

Eine objektorientierte Programmiersprache ist allgemein mehr als andere Programmiersprachen abhängig von ihrem Umfeld. Es existiert in aller Regel ein Stamm fertiger Objekte, auf die zugegriffen wird. Das entspricht einer der Intentionen der Objektorientierung, nämlich der unabhängigen Wiederverwertbarkeit von Programmcode. Eine dagegen auf die Programmiersprache beschränkte Objektorientierung reduziert deren Vorteile auf Programminterna.

Unter Nextstep reichen die Konsequenzen noch weiter. Die grafischen Bedienelemente der Benutzeroberfläche sind Objekte, ihre Klassen stehen vorprogrammiert zur Verfügung. Programme, die die grafische Benutzeroberfläche nutzen wollen, müssen deshalb zumindest teilweise in Objective-C programmiert werden.

Das Betriebssystem

Viele Entscheidungen, die normalerweise beim Kompilieren getroffen werden, bleiben bei einem objektorientierten Programm offen und werden erst während der Ausführung durch ein unterstüzendes Betriebssystem festgelegt. Die Programmiersprache Objective-C in der Version von Next muß daher im Zusammenhang mit dem objektorientierten Betriebssystem Nextstep gesehen werden.

Nextstep gliedert sich in vier Teile: einen Unix-Kern, Hintergrundprozesse, Applikationen und fertige Objekten.

Unix

Kern von Nextstep ist das mit Unix 4.3 BSD uneingeschränkt kompatible Betriebssystem Mach. Statt des herkömmlichen, in seiner mehr als zwanzigjährigen Geschichte immer erweiterten Unix-Kerns enthält es einen Multi-Thread-Kern, der nur die elementarsten Funktionen wie Steuerung der Prozessoren, virtuelle Speicherverwaltung und Datenaustausch zwischen einzelnen Prozessoren regelt und daher sehr klein ist. Die normalerweise im Kern enthaltenen Unix-Dienste werden in separierten Programmen bereitgestellt.

Dadurch wird vor allem durch die Interprozeßkommunikation eine konsistente Programmierschnittstelle geschaffen, die der Objektorientierung sehr zugute kommt. Das Unix bleibt in der Regel auch dem Programmierer verborgen und wird mit Objekten vom »Workspace-Manager« bedient.

Hintergrundprozesse

Bei allen Unix-Derivaten laufen im Hintergrund Prozesse wie 'Print-Server' oder 'File-Server' zum Steuern der Ein- und Ausgabe im System, so auch bei Nextstep. Von Interesse ist der 'Window-Server'. Er kontrolliert alle Benutzerereignisse wie Maus- oder Tastaturaktionen und verwaltet die Fenster. Der 'Window-Server' definiert die Schnittstelle zwischen der Benutzeroberfläche mit ihren Objekten und dem Unix-Kern. Für alle grafischen Ausgabemedien ist im 'Window-Server' ein 'Display-PostScript-Interpreter' integriert.

Zu den Prozessen gehören die Bibliotheken, mit denen in einem Programm Funktionen angesprochen werden, so z. B. die Postscript-Bibliothek.

Applikationen

Der Benutzer bedient das System an der Oberfläche mit einigen Applikationen. Zu nennen ist hiervon der immer präsente 'Workspace-Manager' zum Arbeiten mit Dateien und Verwalten der Prozesse. Das Programm unterscheidet sich von anderen Programmen lediglich dadurch, daß es nicht terminiert werden kann.

Fertige Objekte

Die Beschreibung des Hauptmerkmales eines objektorientierten Betriebssystems soll sich hier auf ein paar Klassen und einige Besonderheiten beschränken.

Direkt unter der Wurzelklassen findet sich für alle Objekte, die auf Ereignisse reagieren sollen, die abstrakte Superklasse `'Responder'`, deren Unterklassen `'View'`, `'Window'` und `'Application'` alles Nötige zur grafisch unterstützten Programmsteuerung leisten (s. Abb. 3).

- `'View'` ist eine abstrakte Superklasse. Instanzen ihrer Unterklassen können als einzige Objekte auf dem Bildschirm zeichnen. Mit der Fähigkeit, Ereignisse zu bearbeiten, ist die Klasse designiert für spezifische Erweiterungen. In vielen Programmen sind Instanzen neuer Unterklassen von `'View'` die eigentlich ausführenden Objekte.
- `'Window'` ist eine Klasse zum Manipulieren von Fenstern. Jedes Objekt korrespondiert mit einem physikalischen Fenster des 'Window-Servers' auf dem Bildschirm. Die Klasse definiert einen Rahmen und verwaltet die `'View'`-Objekte, die innerhalb des Rahmens zeichnen.
- `'Application'` definiert Objekte mit der Umgebung zum Ausführen eines Programmes. Ein Programm hat genau eine Instanz der Klasse. Es verwaltet die Fenster und stellt die Verbindung zum Window-Server. Das Objekt ist so wichtig, daß es global mit `'NXApp'` aufgerufen wird.

Die Klassen `'Window'` und `'Application'` werden regelmäßig durch Delegierte erweitert.

Einige besondere Klassen sind Unterklassen von `'Panel'` wie `'FontPanel'`, `'PrintPanel'` oder `'ColorPanel'` (Abb. 4). Diese – immer erweiterbaren – Klassen sind weit ausprogrammiert und stellen komfortable Instrumente zum Bedienen häufiger Programmfunktionen wie 'Drucken' oder 'Sichern' dar. Je Applikation kann nur jeweils eine Instanz einer solchen Klasse gebildet werden.

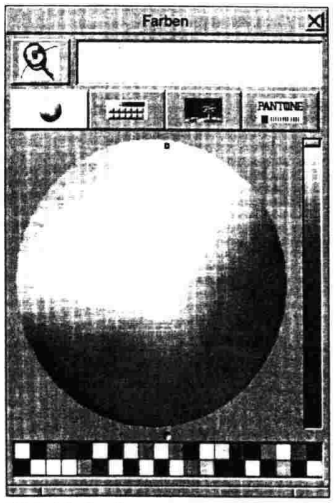


Abb.4: ColorPanel

Insgesamt werden von Next mehr als 100 vorprogrammierte Klassen geliefert, davon ca. die Hälfte zum Steuern der Programme. Die Klassen werden als Bausätze zusammen mit entsprechenden Werkzeugen zur Verfügung gestellt.

Programmlauf mit Objekten

Die Interaktion zwischen den Programmen mit ihren Objekten und dem Betriebssystem ist objektorientiert anders und bedarf der Beschreibung. Deshalb soll exemplarisch der Ablauf eines Programmaufrufes und eines Ereignisses die inhärente Funktionalität von Objekten verdeutlichen.

Programmaufruf

Wenn ein Benutzer ein Programmsymbol angeklickt hat und das Programm laufen soll, erstellt der Workspace-Manager dem Programm eine Instanz der Klasse `'Application'`. Dieses Objekt erstellt sich dann unter anderem je eine Instanz der Klasse `'Speaker'` und `'Listener'` zum Nachrichtenaustausch mit anderen Applikationen. Der `'Speaker'` sendet Nachrichten an andere Applikationen, der `'Listener'` empfängt sie. Danach wird in aller Regel eine Interfacedatei mit dem Hauptmenue und eventuellen Fenstern geladen. Ist die Prozedur beendet, sendet das Objekt dem 'Workspace-Manager' die Nachricht, daß es aktiviert ist und aktives Programm sein möchte. Wird es das aktive Programm, erscheint sein Hauptmenue auf dem Bildschirm, und sein oberstes Fenster wird aktiv. Abschließend startet das Objekt eine Schleife zum Abfragen von Ereignissen.

Ereignis

Tritt ein Ereignis ein, meldet es der 'Window-Server' zuerst dem aktiven Programm, das die Nachricht zu seinem aktiven Fenster weiterleitet. Das Fenster sendet die Nachricht an alle von ihm kontrollierten `'View'`-Objekte. Betrifft das Ereignis eines der Objekte, führt es eine Aktion aus und antwortet dem Fenster. Das Fenster antwortet der Applikation. Mit der Antwort der Applikation, dem 'Window-Server', ist die Aktion beendet.

Antwortet das Fenster nicht, sendet die Applikation die Nachricht allen Fenstern seiner Liste. Antwortet eines, wird es aktiviert, und die Aktion kann ausgeführt werden. Kann die aktive Applikation nicht antworten, weil keines der Fenster antwortet, sendet der 'Window-Server' dem 'Workspace-Manager' eine Nachricht. Der sendet daraufhin eine Nachricht an alle Applikationen. Antwortet eine Applikation, wird diese aktiviert, und die Aktion kann ausgeführt werden.

Antwortet keine Applikation, wurde der Bildschirmhintergrund angeklickt.

Bei alledem enthalten die aufrufenden Objekte keine Informationen über Art und Zweck eines Ereignisses oder einer Aktion. Sie erwarten nur die Antwort, ob ihre Nachricht akzeptiert wird. Die antwortenden Objekte haben und bekommen keine Information des aufrufenden Objektes (außer der, daß es sie aufgerufen hat).

Die beiden Beispiele zeigen, wie in einem objektorientierten Programm in sich abgeschlossene Einheiten den Programmablauf regeln. Gleichzeitig zeigen sie – natürlich – exemplarisch das Hin und Her von Nachrichten, das die Objektorientierung kennzeichnet.

Entwicklung

Das Betriebssystem Nextstep wurde 1988 zusammen mit dem ersten Rechner der Firma Next vorgestellt, die 1985 von Steve Jobs, einem der Erfinder der Apple-Computer, gegründet wurde. Es ist seitdem das einzige objektorientierte Betriebssystem, das einem größeren Benutzerkreis zugänglich ist.

Ursprünglich nur auf den Next-Workstations mit Motorola-Prozessoren lauffähig, ist das Betriebssystem seit Frühjahr 1993 mit der Einführung von Nextstep 3.0 auch auf Rechnern mit Intel-Prozessoren (486 SX aufwärts) verfügbar. Ab Mitte 1994 soll das System für die Hewlett-Packard-Rechner der Apollo-9000/700-Serie zur Verfügung stehen.

Im Mai 1993 wurde die Hardware-Produktion von Next eingestellt.

Ein Beispiel

Vorgestellt wird ein kleines Programm, mit dem auf dem Bildschirm einfache Zeichnungen gemacht werden können. Als Eingabegerät wird die Maus verwendet.

Konzept

Erster und wichtigster Schritt ist das Festlegen einer Klassenhierarchie. Dazu bedarf es genauer Vorstellungen der Anforderungen an die Funktionalität einer Klasse und den notwendigen Daten.

Zugleich muß die Verwendbarkeit vorhandener Klassen sondiert und eingeplant werden. Erst danach können neue Klassen definiert werden.

Ein Entwurf

Eine Zeichnung entsteht durch Aneinanderfügen grafischer Elemente wie Punkt, Strich oder auch Fläche mit einem Zeichengerät auf einer Zeichenfläche.

Da die Zeichenfläche ein Teil des Bildschirmes ist, muß mindestens eine Instanz von

View gebildet werden. Die Abbildungsfunktionen für Bildschirm und Drucker sind damit vorhanden. Ebenso liegen die Ereignisfunktionen vor, weshalb das Zeichengerät, die Maus, vorerst der Zeichenfläche zugeordnet wird.

Eine Zeichnung besteht aus mehreren Zeichenelementen, die die Zeichnung vollständig beschreiben. Sie kann als Liste von Zeichenelementen verstanden werden.

Ein Zeichenelement hat grafische und geometrische Komponenten. Grafische Komponenten wie Farbe oder Strichstärke sind Attribute des Zeichengerätes. Geometrische Komponenten wie Strich oder Kreis werden mit Hilfe des Zeichengerätes interaktiv erstellt. Zumeist werden einfache Funktionen wie Kreis um Punkt A oder Linie von Punkt A nach Punkt B benutzt. Betrachtet man diese Funktionen ebenfalls als Attribute, können die geometrischen Komponenten auf die Lageinformation der Punkte reduziert werden. Das sind genau die Informationen, die mit den Ereignisfunktionen der Zeichenfläche von dem Mauszeiger der grafischen Oberfläche geliefert werden.

Mit einer Funktion als Attribut ist in Objective-C nichts anderes als eine spezifizierende Methode gemeint. Die Zeichenelemente werden daher in mehreren Klassen mit entsprechenden Funktionen aus der Postscript-Bibliothek definiert. Die

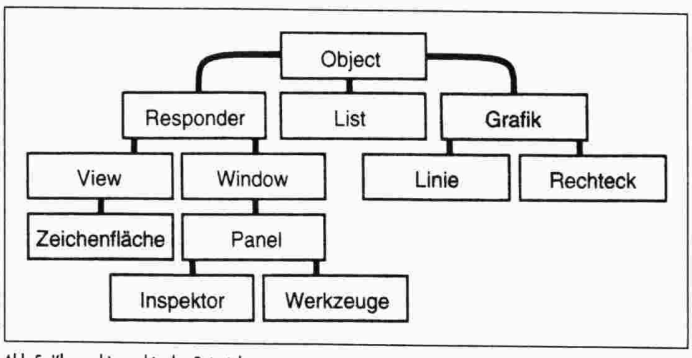


Abb.5: Klassenhierarchie des Beispielprogrammes

Klassen sind Unterklassen einer abstrakten Superklasse, die direkt der Wurzelklasse entstammt. Hier werden allgemeine Eigenschaften der Zeichenelemente bestimmt. Wenn keine systemspezifischen Funktionen benutzt werden, ist das sehr vorteilhaft für die Unabhängigkeit der eigenen Klassenhierarchie.

Der Entwurf könnte so funktionieren. Tauchten jetzt Widersprüche auf, wäre es allemal besser, eine neue Hierarchie zu entwerfen, als anfangen zu programmieren.

Neue Klassen

Steht der Entwurf, werden die Klassen definiert. `Zeichenfläche` ist eine Unterklasse von `View`, `Grafik` eine Unterklasse von `Object`, `Linie`, `Kreis`, `Rechteck` etc. sind Unterklassen von `Grafik`. `Zeichnung` braucht nicht definiert werden, dafür gibt es bereits die vorprogrammierte Klasse `List` zum Verwalten von Objekten (Abb. 5).

Zeichenfläche

Die Zeichenfläche hat die Kontrolle der grafischen Ein- und Ausgabe. Zum Zeichnen muß die Zeichnung bekannt sein, darum wird eine Instanzvariable `(id)zeichnung` deklariert. Zugleich muß die Größe der Fläche auf dem Monitor bekannt sein, also `(NXRect)ausschnitt`. Zur Eingabe werden, für das gerade in Arbeit befindliche Zeichenelement, die Variablen `(Class)element` und `(id)grafik` deklariert.

Die in `View` vordefinierte Methode `drawSelf:` wird so erweitert, daß allen Instanzen von `Grafik` die gleiche Nachricht, `zeichne:`, gesendet wird (Polymorphismus), dazu kommen die notwendigen Ereignisfunktionen.

Alle wichtigen Daten sind nun von diesem Objekt aus verfügbar, darum werden hier auch die Methoden `oeffnen:` und `sichern:` mit den entsprechenden Klassen `OpenPanel` und `SavePanel` entwickelt. Mehr soll das Programm nicht können.

Die Superklasse

Bis auf die unterschiedlichen Zeichenfunktionen werden alle Methoden in der Klasse `Grafik` definiert. Das sind die Methoden, die durch Ereignisse gesteuert werden, z. B. `schiebe:` oder `wähle:`, und die Methoden zum Setzen der attributiven Werte, z. B. `setzeLinienBreite:`. Zusätzlich wird die Methode `zeichne:imAusschnitt` ohne Inhalt definiert.

Alle Variablen für die attributiven Werte werden deklariert, z. B. `(NXColor)linienfarbe`. Dazu kommen zwei Punkte, `(NXPoint)punktA` und `(NXPoint)punktB`. Die meisten geometrischen Grundformen sind mit zwei oder drei Punkten vollständig definiert. Da hier der Einfachheit halber alle Elemente orthogonal angeordnet werden, genügen zwei Punkte für `Linie`, `Kreis` (besser `Ellipse`) und `Rechteck`.

Die Klassen `Freihandlinie` und `Polygonzug` benötigen mehr als zwei Punkte, aber als Anfangs- und Endpunkt werden die Variablen ebenfalls verwendet (Abb. 6).

Unterklassen

Die Methode `zeichne:` wird um die spezifischen Zeichenfunktionen erweitert. `Linie` zeichnet einen Strich von A nach B. `Rechteck` zeichnet ein `(NXRect)rechteck` mit der Diagonale von A nach B. `Ellipse` zeichnet eine Ellipse, die von einem Rechteck mit der Diagonale AB umschrieben wird.

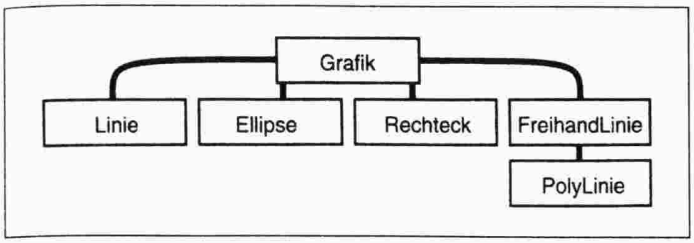


Abb.6: Unterklassen der Superklasse 'Grafik'

Freihandlinie und Polygonzug sind umfangreicher zu bearbeiten. Der Unterschied liegt in der Eingabe. Während Polygonzug eine Linie von Punkt zu Punkt zeichnet, die mit der Maus einzeln eingegeben werden, zeichnet Freihandlinie jeden Punkt, über den die gedrückte Maus geführt wird. Da so Lücken auftreten können, werden trotzdem nicht die Punkte, sondern die Verbindungslinien gezeichnet, die Methode zeichne: ändert sich nicht.

Noch eine Superklasse?

Eine Methode zum dynamischen Speichern von Punkten ist für beide Klassen wichtig (Abb. 7). Das führt zur Frage, ob die Methode in einer Klasse, Freihandlinie, definiert wird und die andere Klasse, Polygonzug, sie erbt. Als Alternative wäre eine abstrakte Superklasse, Linienzug, mit zwei Unterklassen zu definieren (Abb. 8).

Die Entscheidung ist einfach. Wenn absehbar ist, daß die Methode nur in den bekannten zwei Klassen genutzt wird, sollte man keine Superklasse definieren und unterschiedliche Methoden einfach überschreiben. Hier rechtfertigt der Mangel an Übersichtlichkeit noch keine differenzierte Hierarchie. Das Beispiel bleibt, auch wegen zeichne: auf diesen Fall beschränkt (s. Abb. 6).

Sind weitere Elemente mit mehreren Punkten notwendig, oder ist nicht absehbar, ob neue Klassen, etwa für Schraffuren, verwendet werden sollen, definiert man besser eine Superklasse.

Weitere Klassen

Die attributiven Werte werden mit Hilfe einer neuen Klasse, Inspektor, gesetzt, die Unterklasse von Panel ist und im Entwurf nicht auftaucht. Eine Instanz sendet seine Nachrichten an eine Zeichenfläche. Die Zeichenelemente bleiben damit unter der Kontrolle nur eines Objektes. Mit den verschiedenen Bedienelementen werden entsprechende Methoden wie setzeLinienBreite: oder setzeFuellFarbe: aufgerufen (Abb. 9).

```
- mehrSpeicher
/* Der Speicher wird in Brocken angefordert.  */

{
    int i, neueGroesse;

    neueGroesse = laenge + BROCKEN_GROESSE;
    if (punkte) {
        NX_ZONEREALLOC([self zone], punkte, float, neueGroesse << 1);
        NX_ZONEREALLOC([self zone], userPathOps, char, neueGroesse);
    } else {
        NX_ZONEMALLOC([self zone], punkte, float, neueGroesse << 1);
        NX_ZONEMALLOC([self zone], userPathOps, char, neueGroesse);
    }
    for (i = neueGroesse - 1; i >= laenge; i--) {
        userPathOps[i] = dps_rlineto;
    }

    return self;
}
```

Abb.7: Methode zum dynamischen Speichern von Linienpunkten

Als letzte Klasse wird, zum Auswählen der Zeichenelemente und ebenfalls als Unterklasse von Panel, Werkzeuge definiert. Wird ein Knopf gedrückt, wird eineZeichenfläche ein Klassenzeiger in einer Nachricht gesandt, der (Class)element zugewiesen wird (Abb. 10).

Die neuen Unterklassen können ohne Schwierigkeiten in die Hierarchie eingefügt werden, sie beeinflussen den Entwurf nicht (s. Abb. 5).

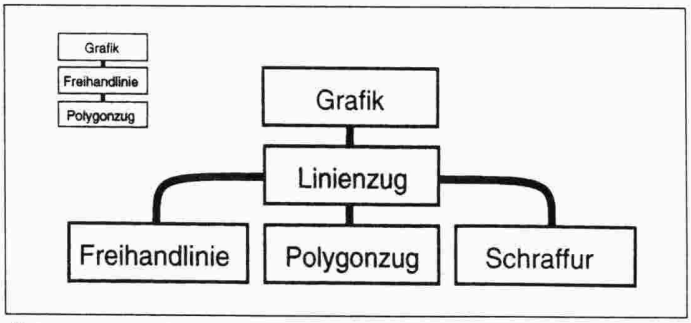


Abb.8: Varianten einer Klassenhierarchie

Objekthierarchie

Sind alle Klassen programmiert, kann schematisch bestimmt werden, welches Objekt welche Aufgabe übernimmt. Dazu werden die Objekte mit den wichtigsten Verbindungen skizziert (Abb. 11). Solche Skizzen sind wichtig für den Entwurf des Programmes, sie werden, während die Klassen definiert werden, häufig modifiziert.

Ablauf

Wenn die Objekte zusammengefügt sind, kann gezeigt werden, wie das Programm funktioniert.

Will ein Benutzer zeichnen, wird mit dem ersten Mausklick (Nachricht von NXApp:[eineZeichenfläche mouseDown:-x:y]) auf der Zeichenfläche eine Instanz von List gebildet und meineZeichnung zugewiesen. Dann wird eine Instanz der Klasse entsprechend in element gebildet (beim Start Linie) und grafik zugewiesen. Ist das Element fertig gezeichnet (mouseUp:), wird grafik in meineZeichnung gespeichert. Beim nächsten Mausklick wird eine neue Instanz von Grafik entsprechend element gebildet.

Muß die Zeichnung neu gezeichnet werden, weil sich vielleicht der Ausschnitt verändert hat, erhält eineZeichenfläche eine entsprechende Nachricht von NXApp. In der damit aufgerufenen Methode drawSelf: wird allen Objekten in meineZeichnung die Nachricht [grafik zeichne:ausschnitt] gesendet. Die Objekte prüfen, ob sie in den Ausschnitt passen, und 'zeichnen sich dann selbst' entsprechend ihrer Definition und ihren Werten. Ähnliches geschieht beim Speichern: Alle Objekte werden aufgerufen und speichern ihre Werte 'selbst'.

Die übrigen Objekte zum Steuern des Programmes werden durch das Hauptmenue aufgerufen. Der Benutzer klickt in einen entsprechenden Eintrag, z. B. 'Werkzeuge', und das Objekt 'dieWerkzeuge' erscheint auf dem Bildschirm. Gleiches gilt für die anderen Funktionen.

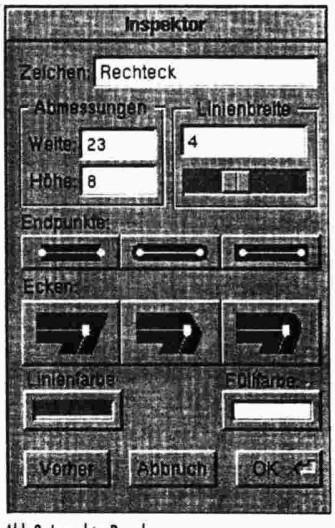
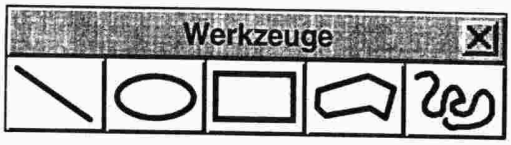


Abb.9: InspektorPanel

Abb.10: WerkzeugePanel



ColorPanel (s. Abb. 4) hat eine Besonderheit: Das Objekt sendet keine Nachricht, wenn eine Farbe gewählt ist. Daher wird es nicht der Zeichenfläche, sondern NXApp beigeordnet. Die Farbe wird entweder von einem anderen Objekt abgefragt, oder sie wird mit der Maus als kleine Farbfläche zum Zeichenelement oder zum Inspektor transportiert.

Konsequenzen

Anwendern bietet die objektorientierte Programmierung vorderhand keine Vorteile. Die Art der Programmierung ist einem Programm in der Regel nicht anzumerken. Das Beispiel funktioniert genauso wie ein Dutzend oder mehr anderer Zeichenprogramme auch.

Die Vorteile der Objektorientierung liegen in der Softwareentwicklung (wenn es diesen Begriff dann noch gibt). Alle Klassen können prinzipiell und isoliert erweitert werden, ohne das Funktionieren des Programmes zu gefährden. Grafik, die dafür prädestinierte Klasse, kann mit relativ wenig Aufwand um Klassen für Schraffuren, Texte oder auch Bilder erweitert werden. Aber auch NXApp kann erweitert werden, beispielsweise zum Verwalten mehrerer Zeichnungen und Fenster, oder Zeichenfläche könnte ein Raster mit den dazugehörigen Methoden selbst steuern. Die Weiterentwicklung eines Programmes wird durch die Erweiterbarkeit der Klassen wesentlich vereinfacht.

Andere Vorteile zeigen sich im Vergleich mit prozeduralen Programmen: Die konditionierten Bewertungen (if...else) und Fallunterscheidungen (case) fehlen fast vollständig. Wäre das Programm prozedural programmiert, gäbe es einen Datenpool mit allen Daten einer Zeichnung. Würde ein Zeichenelement herausgegriffen und sollte manipuliert werden, bestimmte man mit Hilfe konditionierter Bewertungen genau, um was für ein Element es sich handelte. Erst danach würde das Element manipuliert.

Objektorientiert fragt 'eineZeichnung' alle Objekte seiner Liste, ob bestimmte Eigenschaften für sie gelten. Die Objekte vergleichen in einer Bewertung die angefragten Daten mit ihren eigenen und antworten entsprechend. Danach wird nichts mehr kontrolliert, die Objekte sind für ihre Methoden selbst verantwortlich. (Aufmerksame Leser werden es bemerkt haben: Tatsächlich finden, wenn auch verborgen, mehr Bewertungen statt, nämlich in jedem Objekt eine.)

Ein weiterer Vorteil ist die Kürze der Quelltexte für die einzelnen Klassen und die sich daraus ergebende Übersichtlichkeit.

Es gibt natürlich auch Nachteile. Ein Problem sind die Schwierigkeiten, die immer wieder mit den ererbten Methoden der Klassen auftauchen. Anders als mit Funktionen einer Bibliothek, die einmal eingebunden immer präsent sind, verliert man bei Klassen in der sechsten und siebten Generation schon mal den Überblick, welche Methoden vorhanden sind. Besonders unangenehm ist es, wenn eine ganz oben in der Hierarchie implementierte Methode in einer der nachfolgenden Generationen überschrieben ist. Man kann sie nicht benutzen. Solche Schwierigkeiten sind glücklicherweise selten und lassen sich durch entsprechende Werkzeuge beseitigen.

Andere Probleme sind weniger einfach zu beseitigen. Die Umstellung zur Objektorientierung bereitet vielen Entwicklern Schwierigkeiten. Die Konventionen, eigentlich erfunden, um das Programmieren einfacher und intuitiver zu machen, sind zu konkret: Die Parallelität von Objekthierarchien und sozialen Hierarchien z. B. ist nicht allen behaglich. Diesem Punkt sollte Aufmerksamkeit gewidmet werden, wenn behauptet wird, Objekte könnten alles.

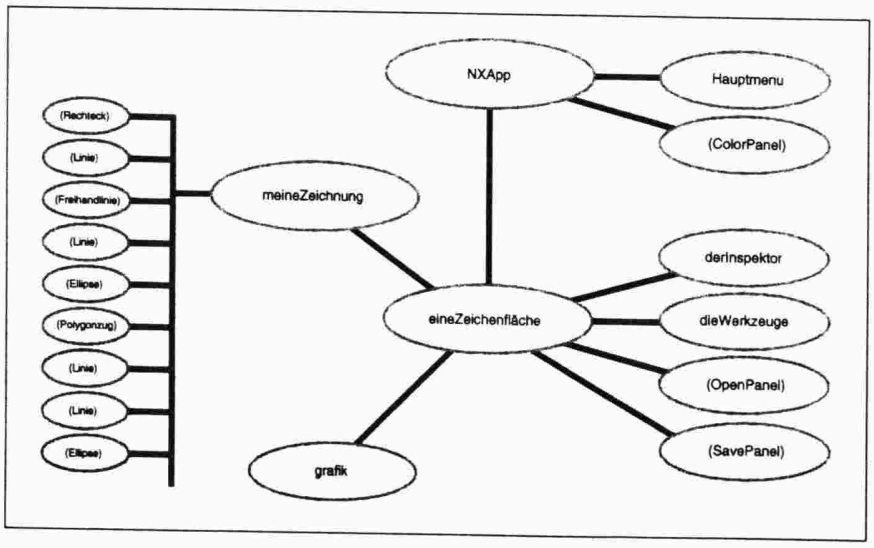
Ein anderer Punkt ist die Art der Programmierung: Es ist nicht möglich, wie gewohnt bei Null anzufangen. Die erste neue Zeile Programmtext erweitert immer nur eine schon vorprogrammierte Klasse. Viele Entwickler fühlen sich dadurch in ein Korsett gepreßt. Sie wissen nicht genau, was die Klassen machen, und haben deswegen Bedenken, neuen Text zu formulieren.

Tatsächlich zwingt die Klassenstruktur niemanden, eine Zeile Programmtext weniger zu schreiben. Klassen können fast beliebig erweitert werden. Hat man diese Schwierigkeiten überwunden, entpuppt sich das Korsett als Gehhilfe in der Art von Siebenmeilenstiefeln. Es ist in den Klassen vieles detailliert vor- oder ausprogrammiert, getestet und lauffähig, das einfach benutzt werden kann.

Ein Blick in die Zukunft

Der objektorientierten Programmierung gehört sicher ein Teil der Zukunft. Dafür sprechen schon allein ihre originären Vorteile gegenüber der prozeduralen Programmierung. Gleichfalls dafür spricht die Entwicklung objektorientierter Betriebssysteme bei heutigen Marktführern als Basis wirklich effizienter objektorientierter Programme. Genannt seien hier das Projekt 'Cairo' von Microsoft und das Projekt 'Taligent' der Allianz von IBM und Apple mit dem objektorientierten Betriebssystem 'Pink'. Wenn diese Zukunft eintrifft, werden sich schon heute beobachtbare Entwicklungen verstärken, die die Trennung zwischen Softwareanbieter und Anwender verschieben oder gar ganz verwischen. Für das Betriebssystem Nextstep gibt es neben konventionellen Programmen mittlerweile einzelne Klassen und Klassenbibliotheken fertig programmiert zu kaufen. Damit wird es Anwendern ermöglicht, aus einer Reihe ge-

Abb.11: Objekthierarchie des Beispielprogrammes



kaufter Objekte eigene, bedarfsorientierte Lösungen zu schmieden. Softwarehersteller diversifizieren sich in Hersteller von Programmen und Hersteller von Klassen, aus Anwendern können potentielle Entwickler werden. Das ist nach den Insellösungen und den integrierten Gesamtlösungen sicher eine interessante Variante.

Gegen die Objektorientierung spricht ihre derzeitige Verbreitung. Die Ursachen dafür liegen wahrscheinlich tiefer, als zugegeben wird, und haben wenig mit der vorgeschoben erscheinenden, mangelnden Anwenderakzeptanz zu tun. Deutlich wurde das bei den Schwierigkeiten, die beim Transferieren der Datenbestände relationaler Datenbanken in objektorientierte Datenbanken auftraten und die ein wenig die Fachpresse belebten.

Ähnliche, noch größere Schwierigkeiten werden bei den Programmen erwartet. Es wird nicht möglich werden, prozedurale Programme unverändert in objektorientierte zu konvertieren. Die den Programmierweisen zugrundeliegenden Konzeptionen sind zu unterschiedlich, als daß eine vernünftige Lösung praktikabel erscheint. Wenn prozedurale Programme konvertiert werden müssen, können ihre sinnvollen und gut funktionierenden Teile in objektorientierte Programme eingebaut werden. Vorhandene Prozeduren würden zu Methoden neuer Objekte. Für solche Strategien bieten Programmiersprachen wie Objective-C, in denen die Quelltextkompatibilität zum Konzept gehört, die weitreichendsten Möglichkeiten.

Der Anspruch der Objektorientierung, mit der Verbindung von Daten und Funktionen in kleinen Programmbausteinen einfacher und schneller neue Programme entwickeln zu können, sollte nicht darauf reduziert werden, Konzepte der prozeduralen Programmierung zu wiederholen. Eine neu angewendete Technologie bietet immer auch neue Möglichkeiten. Daher sollte es Ziel sein, die neuen Werkzeuge, die die objektorientierte Programmierung bietet, zu nutzen, um anwenderorientierte, benutzerfreundliche Software zu entwickeln. Daran wird ihr Erfog gemessen werden.

Die Entwicklung der Softwaretechnologie bleibt bei der Objektorientierung nicht stehen. Eine konsequente Weiterentwicklung der objektorientierten Programmierung wird unter dem Stichwort 'Adaptive Software' präsentiert. Der Zugriff auf Objektkomponenten ist unabhängig von der Klassenstruktur, die Funktionalität der Anwendungen wird so von der tatsächlichen Struktur der Klassen und ihren Erweiterungen entkoppelt. Bis eine solche Technologie aber das Bauwesen betrifft, wird gewiß noch einige Zeit verstreichen. □

Literatur:

Kernighan, Brian W.und Ritchie, Dennis M.: Programmieren in C. München, Wien 1983.

Mühnke, Susanne: Der zweite Frühling, S.10ff in Workout 1/91. Hamburg 1991.

Perkins, Charles L.: Getting with the Program, S.30ff in Nextworld 2/92. San Francisco 1992.

Pinson, Lewis J.und Wiener, Richard S.: Objective C: object-oriented programming techniques. Reading, Menlo Park, New York, Don Millls usw. 1991.

Next Computer Inc. (Hrsg.): Next Operating System Software. Reading, Menlo Park, New York, Don Mills usw. 1991.

Next Computer Inc. (Hrsg.): Programmings and the Objective-C-Language: Release 3. Reading, Menlo Park, New York, Don Mills usw. 1992.

Riem, Philip: Große Schritte nach vorn, S.28 in Macup 8/93. Hamburg 1993.

Schicke, Harald: NeXT – einer für alle. 1. Aufl. Würzburg 1991.

Wagner, Michael P.: Insider-Treffen, S.28 in c't 7/93. Hannover 1993.

Wagner, Michael P.: Sonnige Zeiten für Objekte, S.12 in c't 8/93. Hannover 1993.A