Oliver Berse und Raimund Seisenberger

FLOYD

Einführung in die Programmierung von Adventurespielen
Version 3.3

Inhalt

1 Einleitung

Let Floyd launch the spacetruck? Please? Floyd has not crashed a truck in over two weeks! -- Steve Meretzky, *Stationfall*

Floyd ist eine Sprache für die Entwicklung von Adventurespielen. Mit Adventurespielen ist dabei die klassische Variante der Textadventures gemeint, die vom Spieler durch Eingabe einfacher Kommandos und Sätze gesteuert werden.

Wie alle anderen Spiele lassen sich auch Adventures mit einer normalen Programmiersprache wie C oder Java schreiben. Dabei müßte aber erst viel Code geschrieben werden, um die in jedem Adventure vorkommenden Aufgaben wie die Verarbeitung der Spielereingabe, die Simulation einer Spielwelt oder das Speichern und Laden von Spielständen zu erledigen.

Floyd nimmt dem Autor solche Routinearbeit ab und hilft ihm, sich auf die Umsetzung der Handlung zu konzentrieren. Floyd kann mit Spielereingaben umgehen, in denen mehrere Objekte oder Wörter wie "alle" und "außer" vorkommen. Eingaben können korrigiert oder rückgängig gemacht werden. In der Spielwelt können Räume, Verbindungen, Nichtspielercharaktere, Behälter, Fahrzeuge und viele weitere Objekte vorkommen, deren Verhalten sich leicht bestimmen läßt. Die Ein- und Ausgabe unterstützt Floyd mit Skripten, Menüs, Textfenstern, Grafiken und Mitschriften des Spielverlaufs.

Floyd verfügt über eine Standardbibliothek, in der wichtige Objekte der Spielwelt und einige nützliche Funktionen vordefiniert sind. Die Standardbibliothek besteht aus mehreren Quelltextdateien und läßt sich anpassen und erweitern.

Für deutschsprachige Spiele gibt es neben Floyd bereits seit längerem einige weitere Programmiersprachen und Autorensysteme. Die bekanntesten von ihnen sind die deutsche Version des Inform-Compilers von Graham Nelson und der Text-Adventure-Generator von Martin Oehm. Diese Systeme sind zweigeteilt, d.h. der Quelltext wird erst von einem Compiler in einen Zwischencode übersetzt, der dann mit einem Interpreter gespielt werden kann. Floyd interpretiert den Quelltext dagegen direkt, so daß Programmierer und Spieler das gleiche Programm benutzen können.

Um die Einarbeitung zu erleichtern, entspricht Floyd syntaktisch einer sehr stark vereinfachten Variante von C und ähnlichen Sprachen. Wer also schon ein wenig Programmiererfahrung hat, wird sich in Floyd schnell zurechtfinden.

Floyd ist Open Source und darf für nicht-kommerzielle Zwecke ohne Einschränkungen weiterverbreitet werden. Mit Floyd geschriebene Programme gehören ihren Autoren. Für eventuell durch Floyd verursachte Schäden (z. B. Kopfschmerzen um 3:00 Uhr morgens vor dem Rechner) und die Inhalte damit geschriebener Programme übernehmen die Autoren keine Verantwortung.

Interessierte finden unter www.oliver-berse.de auch den Quelltext, der verändert und weiterverbreitet werden darf. Fairerweise sollten darin die Namen der Autoren erhalten bleiben.

Diese Dokumentation behandelt nur die technische Seite der Programmierung mit Floyd. Bevor man sich an die Entwicklung eines Adventures macht, sollte man sich intensiv mit dem Design von Spielen beschäftigen. Sehr viele nützliche Hinweise finden sich z. B. in den folgenden beiden Texten:

Graham Nelson: The Craft of Adventure
www.ifarchive.org/if-archive/info/Craft.Of.Adventure.txt

1 Einleitung

Kevin Wilson: Whizzard's Guide to Text Adventure Authorship
www.ifarchive.org/if-archive/info/authorship-guide.base

Wer ein Spiel mit Floyd schreibt oder die Standardbibliothek verbessert, kann die Autoren gerne benachrichtigen, damit auf der Floyd-Website darauf hingewiesen werden kann. Wer Kontakt mit den Autoren aufnehmen möchte, sende bitte eine Mail an: mail AT oliver-berse PUNKT de

Bevor es jetzt losgeht, noch ein paar Worte zur Typographie dieser Dokumentation:

- Programmzeilen, die eingegeben werden können, werden eingerückt und durch Maschinenschrift gekennzeichnet.
- Bildschirmtexte, Formeln und Definitionen stehen in einzelnen Zeilen und werden ebenfalls eingerückt.
- Schlüsselwörter von Floyd, Namen von Variablen, Funktion und Dateien werden im Text kursiv geschrieben.
- Wichtige Hinweise, die helfen sollen, Fehler zu vermeiden, werden mit dem Wort **Achtung** eingeleitet.

2 Grundlagen

Bevor es um Adventures geht, klärt dieses Kapitel, welche Voraussetzungen Floyd stellt, wie ein Floyd-Programm ausgeführt wird und was seine wichtigsten Bestandteile sind.

2.1 Floyd und Java

Die ersten Versionen von Floyd waren in Delphi-Pascal geschrieben und liefen unter Windows und Linux. Die Version 3 ist eine vollständige Neuimplementierung in Java und läuft damit auf allen Betriebssystemen, für die es eine Java-Laufzeitumgebung (Java Runtime Environment, JRE) gibt. Floyd erfordert mindestens die JRE Version 1.5 (Java 5). Wenn auf Ihrem System noch kein Java installiert ist, findet sich die JRE für verschiedene Systeme unter:

http://www.java.com/de/download/manual.jsp

Starten läßt sich Floyd einfach mit einem Klick auf die Programmdatei *floyd.jar*.

2.2 Ein Programm schreiben und ausführen

Floyd-Programme lassen sich mit jedem beliebigen Texteditor schreiben und müssen mit der Dateiendung .floyd gespeichert werden. Für den Windows-Texteditor Textpad und die plattformunabhängige Entwicklungsumgebung Eclipse finden sich auf der Homepage auch Plugins, die eine Syntaxhervorhebung (d.h. die Markierung von Schlüsselwörtern) ermöglichen.

Programmdateien werden über das Menü *Datei/Öffnen* ausgewählt und gestartet. Alternativ kann eine Programmdatei auch per Drag & Drop in das Floyd-Fenster gezogen und gestartet werden. Tritt während der Ausführung ein Fehler auf, gibt Floyd eine Meldung und die Nummer der Zeile aus, in der der Fehler auftrat, und bricht das Programm ab (s. 8).

Fertige Programme lassen sich mit dem Menüpunkt *Optionen/Quelltext compilieren* übersetzen. Dabei wird eine neue Datei mit der Endung .floydc erzeugt und eine kurze Statistik über das Spiel ausgegeben. "Compilieren" ist dabei kein ganz passender Begriff für diesen Vorgang, denn der Quelltext wird nicht in eine Maschinensprache übersetzt. Wenn eine Programmdatei compiliert wird, wird der Quelltext vorverarbeitet, wobei Kommentare und überflüssige Leerzeichen entfernt und einige weitere Umwandlungen vorgenommen werden. Dabei wird der Quelltext auch in eine für den Spieler unlesbare Form übertragen.

Achtung: Werden Quelltextdateien zwischen verschiedenen Betriebssystemen ausgetauscht oder auf einer Website veröffentlicht, müssen sie mit einem Zeichensatz gespeichert werden, der auf allen Systemen gleich dargestellt wird. Das verhindert, daß die unter System A gespeicherten deutschen Umlaute unter System B als Zeichensalat angezeigt werden. Ein von allen gängigen Betriebssystemen und modernen Texteditoren unterstützter Zeichensatz ist UTF-8, der daher für den Datenaustausch zu empfehlen ist. Compilierte Programmdateien (.floydc) werden immer automatisch UTF-8 codiert gespeichert.

Ab Version 3.3 unterstützt Floyd auch den Java-Webstart. Der Interpreter und eine Spieldatei können damit über einen Weblink gestartet werden. Eine Anleitung dafür gibt es in Kapitel 9.

2.3 Das erste Programm

Floyd-Programme bestehen aus Funktionen. Jede Funktion kann mehrere Anweisungen enthalten, die nacheinander ausgeführt werden, wenn die Funktion aufgerufen wird. Die Programmausführung beginnt immer mit dem Aufruf einer Funktion namens *main()*, die in jedem Programm vorkommen muß. Das folgende Miniprogramm besteht nur aus der Funktion *main()* und einer *write*-Anweisung zur Ausgabe eines kurzen Textes.

```
void main() {
  write("Hallo Welt^");
}
```

void ist eines der Schlüsselwörter, die den Anfang einer Funktion markieren. *void* markiert eine Funktion, die nach ihrer Ausführung kein Ergebnis (einen Rückgabewert) zurückliefert.

In dem Klammernpaar hinter dem Funktionsnamen können die Argumente angegeben werden, die der Funktion beim Aufruf übergeben und von ihr weiterverarbeitet werden. Wenn das Klammernpaar leer ist, erwartet die Funktion keine Argumente (und es dürfen auch keine Argumente übergeben werden).

Alle zur Funktion gehörenden Anweisungen müssen schließlich noch von einem Paar geschweifter Klammern umschlossen und mit einem Semikolon voneinander getrennt werden. Die geschweiften Klammern und das Semikolon helfen dem Interpreter, einzelne Anweisungen und Programmteile auseinander zu halten. Werden sie an einer Stelle vergessen oder falsch gesetzt, führt das zu einem Fehler.

Leerzeichen, Tabulatorzeichen und Zeilenumbrüche dienen nur der besseren Lesbarkeit. Floyd akzeptiert z. B. auch die folgende Schreibweise:

```
void main() {write("Hallo Welt^"); }
```

Achtung: Die Schlüsselwörter von Floyd dürfen nicht als Namen neuer Funktionen benutzt werden, d.h. eine Funktionsdefinition wie

```
void write() {
}
```

führt zu einer Fehlermeldung.

Die folgende Tabelle zeigt alle in Floyd bekannten Schlüsselwörter:

```
abstract        addScores       array         box
break           case            class         count
day             default         do            else
firstWord       fetch           for           gender
getKey          getLong         getShort      getWord
halt            has             if            int
isfirst         issecond        location      menu
moveto          name            noun          object
objectsInside   pluralInside    pnoun         quit
random          return          room          scores
serial          setColor        setLong       setNoun
setPlayer       setShort        setTime       sizeof
split           startDaemon     startTimer    StatusLineFormat
stopDaemon      stopTimer       string        strlen
string          strstr          strrstr       substr
super           switch          time          toggle
verb            void            while         with
write
```

2.4 Texte

Die *write*-Anweisung gibt einen in doppelten Anführungszeichen stehenden Text aus. Alle in einem Programm vorkommenden Texte müssen in doppelten Anführungszeichen stehen. Dabei darf der Text auch mehrere Zeilen umfassen:

```
write("Erste Zeile^
```

```
Zweite Zeile^
Dritte Zeile^");
```

Wenn ein Text für eine Bildschirmzeile zu lang ist, fügt Floyd nach dem letzten in die Zeile passenden Wort automatisch einen Zeilenumbruch ein. Mit dem Caret-Zeichen ^ läßt sich ein Zeilenumbruch an beliebiger Stelle einfügen. Ohne dieses Zeichen würde der Text im letzten Beispiel in einer Zeile erscheinen.

Sollen in einem Text selbst doppelte Anführungszeichen vorkommen, muß an ihrer Stelle ein Backslash (\) geschrieben werden. Bei der Ausgabe wird der Backslash durch das doppelte Anführungszeichen ersetzt.

```
write("Der Arzt sagt zu deinem Wärter: \Passen Sie gut auf ihn auf\.");
```

Da der Backslash in normalen Texten extrem selten auftaucht, gibt es auch keine Möglichkeit, seine Ersetzung durch Anführungsstriche zu verhindern.

Textattribute und Farben

Einzelne Wörter lassen sich auch fett, kursiv, unterstrichen, durchgestrichen oder (ab 3.1) invers darstellen. Dafür wird die entsprechende Textstelle mit spitzen Klammern markiert:

<Format>Text</Format>

Format ist dabei ein einzelner Kleinbuchstabe und bestimmt das gewünschte Textattribut:

Buchstabe	Format
b	fett
i	kursiv
u	unterstrichen
s	durchgestrichen
r	invers

Ein Beispiel:

```
write("Eine <b>fett geschriebene</b> und eine <u>unterstrichene Textstelle</u>.^");
```

Dabei können die Formate auch kombiniert werden:

```
<b><u>unterstrichen und fett</u></b>
```

Wenn direkt auf das Ende einer Formatierung ein Zeilenumbruch folgt, wird in einigen Fällen die folgende Zeile nur halb dargestellt. Dieser Fehler läßt sich nicht immer reproduzieren und liegt vermutlich in der verwendeten Java-Komponente. Verhindern läßt er sich, indem das Zeichen ^ vor das Formatende gestellt wird:

```
<i>Text^</i>
```

Die Text- und Hintergrundfarben für die Statuszeile und die Textausgabe lassen sich mit der Anweisung

```
setColor(string StatusText, string StatusHintergrund, string AusgabeText,
   string AusgabeHintergrund)
```

einstellen, wenn im Menü *Optionen/Darstellung* die Option „Programme dürfen Farben ändern“ aktiv ist. Die Farbwerte werden dabei als RGB-Werte in Hexadezimalform angegeben. Je eine Hexadezimalzahl bestimmt in dem sechsstelligen Farbstring den Anteil für Rot, Grün und Blau: #000000 entspricht Schwarz und #FFFFFF Weiß. Wie in HTML müssen die Farbangaben mit dem Rautenzeichen # beginnen:

```
setColor("#000000", "#FFFFFF", "#CCCCCC", "#0000FF");
```

Bei inverser Darstellung werden die Vorder- und Hintergrundfarben vertauscht. Unter *Optionen/Darstellung* läßt sich auch eine Farbe für den Eingabetext einstellen, diese wird von der *setColor*-Anweisung immer gleich der Vordergrundfarbe eingestellt.

2.5 Namen von Variablen und Funktionen

Namen dürfen die Buchstaben a bis z (Klein- und Großschrift), die Ziffern 0 bis 9 und den Unterstrich enthalten und müssen mit einem Buchstaben oder dem Unterstrich beginnen. Umlaute dürfen in Namen nicht vorkommen.

Floyd unterscheidet zwischen Groß- und Kleinschreibung. Das bedeutet, daß alle Schlüsselwörter und Bezeichner wie Funktionsnamen und Variablen immer gleich geschrieben werden müssen. Das Schlüsselwort *write* beispielsweise muß als write eingetippt werden, nicht als Write oder WRITE. Daher sind auch RaumName, raumName, raumname und RAUMNAME die Namen von vier verschiedenen Variablen.

2.6 Kommentare

Um den Quelltext verständlicher zu gestalten, sollten wichtige Funktionen kommentiert werden. Floyd erlaubt sowohl einzeilige als auch mehrzeilige Kommentare. Einzeilige Kommentare werden mit einem doppelten Schrägstrich (Slash) eingeleitet und dürfen eine ganze Zeile einnehmen oder hinter einer Anweisung stehen:

```
// Ein einzeiliger Kommentar

/* Ein Kommentar über
    mehrere Zeilen */
```

Kommentare eignen sich auch, um etwa bei der Fehlersuche bestimmte Programmteile auszukommentieren.

2.7 Datentypen und Variablen

Variablen müssen in Floyd vor ihrer ersten Verwendung dem Interpreter bekannt gemacht (deklariert) werden. Dabei wird einfach der Datentyp der Variable gefolgt von ihrem Namen angegeben. Floyd kennt die drei Datentypen Integer, String und Objekt für die Verarbeitung von ganzen Zahlen, von Texten und von Klassen. Der Datentyp Objekt wird zusammen mit Klassen in Kapitel 3 behandelt.

Zahlen

Variablen, die ganze Zahlen (Integers) speichern sollen, werden mit dem Schlüsselwort *int* deklariert. Sie können Werte von -2147483648 bis +2147483647 aufnehmen.

```
int a,b,c;
```

Strings

Der in einem Adventure am häufigsten gebrauchte Datentyp ist der String (Zeichenkette) für die Aufnahme von Texten. Die Länge der darin gespeicherten Texte ist praktisch unbegrenzt.

```
string SpielerName;
```

Für die Verarbeitung von Strings kennt Floyd folgende Funktionen:

```
int strlen(string str)
```

Liefert die Zahl der Zeichen in einem String.

```
int strstr(string substr, string str)
```

Liefert die Position des ersten Vorkommens von *substr* in *str*. Das erste Zeichen in einem String hat den Index 0. Kommt *substr* nicht in *str* vor, gibt die Funktion also -1 zurück.

```
int strrstr(string substr, string str)
```

Liefert die Position des letzten Vorkommens von *substr* in *str*.

```
string substr(string str, int p, int z)
```

Liefert einen Teilstring aus *str* ab Position *p* mit z Zeichen. Schließlich überträgt die Funktion

```
split(string Quelle, string Trennzeichen)
```

einzelne Teilstrings aus *Quelle* in eine Liste von Strings, ein Array. Die einzelnen Teilstrings müssen in *Quelle* durch ein einzelnes Trennzeichen (z. B. ein Komma) getrennt sein. Diese Funktion war in Versionen vor 3.0 eine Anweisung, die eine als drittes Argument übergebene Stringvariable in ein Array umwandelte. Nähere Informationen über Arrays folgen in 2.9.

```
string s[];
s=split("Kluge,Foobar,Xyzzy", ",");
write(s[0]);
```

Anwendungsbeispiele für Stringfunktionen finden wir später in den Kapiteln 2.12 und 7.

2.8 Deklarationen und Zuweisungen

Bei der Deklaration dürfen Variablen auch konstante Werte zugewiesen werden. Geschieht das nicht, werden Integervariablen mit 0 und Stringvariablen mit einem Leerstring initialisiert.

```
int a=23, b, c=1;
string SpielerName="Gomi no sensei";
```

Namen von Variablen (und alle anderen Bezeichnernamen) dürfen aus Klein- und Großbuchstaben (ohne Umlaute), den Ziffern 0 bis 9 und dem Unterstrich _ bestehen. Das erste Zeichen muß ein Buchstabe sein. Die Namen von Schlüsselwörtern und Funktionen von Floyd (z. B. *write*, *main*) können nicht als Variablennamen genutzt werden.

Variablen dürfen in einem Programm außerhalb oder innerhalb von Funktionen deklariert werden. Eine innerhalb einer Funktion deklarierte Variable ist lokal und darf nur in derselben Funktion verwendet werden. Eine außerhalb jeder Funktion (und Klasse, s. Kapitel 3) deklarierte Variable ist eine globale Variable und darf in allen Funktionen verwendet werden.

Achtung: Ab Version 3.0 gibt es bei Zuweisungen eine Typüberprüfung. Variablen können also nur noch Variablen oder Konstanten gleichen Typs zugewiesen werden.

Variablen unterschiedlichen Typs lassen sich in der *write*-Anweisung mit dem Pluszeichen gemeinsam ausgeben (name ist ein Schlüsselwort, daher _name):

```
int alter=23;
string __name="Tom";
write(__name+" "+alter+"^");
```

Sollen Ergebnisse von Berechnungen zusammen mit Strings ausgegeben werden, müssen sie in Klammern gesetzt werden. Dies war in Versionen vor 3.0 nicht notwendig.

```
write((alter+5)+"^");
```

2.9 Arrays

Ein Array ist eine indizierte Liste von Werten des gleichen Datentyps. Eine Arrayvariable wird wie eine normale String- oder Integervariable deklariert. Ihrem Namen folgt ein eckiges Klammernpaar, in dem die Anzahl der zugehörigen Werte angegeben werden kann. Dabei darf ein Array immer nur einen Index (d.h. eine Dimension) haben.

```
int liste[5];
```

Diese Zeile deklariert die Integervariable *liste*, die bis zu fünf Werte aufnehmen kann. Die Indizierung beginnt immer bei 0. Um auf die einzelnen Werte zugreifen zu können, muß ihr Index angegeben werden.

```
liste[0]=1;
liste[1]=2;
```

Die einzelnen Werte einer Arrayvariablen können bei der Deklaration auch initialisiert werden:

```
int taler[2]=(5,10);
string numerale[3]=("eins","zwei","drei");
```

Die Funktion

```
int sizeof()
```

liefert den maximalen Index eines Arrays. Da der erste Index immer 0 ist, ist der maximale Index gleich der Anzahl der Werte minus eins.

Arrays können in Floyd zwar nur einen Index haben, mehrdimensionale Arrays lassen sich aber leicht simulieren. Dafür werden die Elemente eines mehrdimensionalen Arrays einfach in einem eindimensionalen Array aufgereiht. Wenn z. B. i und j den Spalten- und den Zeilenindex eines zweidimensionalen Arrays mit m Spalten angeben, dann hat das Element *a(i,j)* in einem eindimensionalen Array den Index n, wobei $n = j * m + i$.

2.10 Konstanten

In einem Programm häufig vorkommende Werte können als Konstanten deklariert werden, um sie bei einer Änderung nur an einer Stelle ändern zu müssen. Beim Einlesen des Quelltextes ersetzt der Interpreter jeden Konstantennamen durch seinen Wert. Konstanten werden mit dem Schlüsselwort #*define*, ihrem Namen und ihrem Wert deklariert:

```
#define MAX_LEBEN 3
```

Konstanten werden traditionell groß geschrieben, was aber keine Vorschrift ist. Wichtig ist, daß vor dem Zeichen # keine Leerzeichen stehen und die Zeile nicht mit einem Semikolon endet.

Eine Konstante namens SYSTEM wird vom Interpreter automatisch deklariert. Sie dient der Erkennung der verwendeten Floyd-Version: 0 = Version 2 unter Windows, 1 = Version 2 unter Linux, 3 = Version 3 (alle OS).

2.11 Ausdrücke und Operatoren

Ein Ausdruck ist eine Folge von Konstanten, Variablen und Funktionen, die mit Operatoren verknüpft sind und aus der der Interpreter einen Wert vom Typ Integer, String oder Objekt berechnen kann.

Die Tabelle zeigt die in Floyd bekannten Operatoren:

Operator	Beispiel	Rang	Bedeutung
++	++a, a++	6	Prä- oder Post-Inkrement
--	--a, a--	6	Prä- oder Post-Dekrement
!	!a	6	logische Negation
~	~a	6	bitweise Negation
*	a*b	5	Multiplikation
/	a/b	5	Ganzzahldivision a durch b
%	a%b	5	Rest einer Ganzzahldivision (Modulo)
+	a+b	4	Addition
-	a-b	4	Subtraktion
==	a==b	3	Test auf Gleichheit
!=	a!=b	3	Test auf Ungleichheit
<<	a<<b	3	Shift um b Bits nach links
>>	a>>b	3	Shift um b Bits nach rechts
&&	a && b	2	logisches UND
&	a & b	2	bitweises UND
||	a || b	1	logisches ODER
|	a | b	1	bitweises ODER
^	a ^ b	1	bitweises XOR
?:	a ? b:c	1	Wenn a, dann b, sonst c
<=	a<=b	0	Test, a kleiner oder gleich b
>=	a>=b	0	Test, a größer oder gleich b

*=	a*=b	0	zuweisende Multiplikation a=a*b
/=	a/=b	0	zuweisende Ganzzahldivision a=a/b
=	a%=b	0	zuweisendes Modulo a=a%b
-=	a-=b	0	zuweisende Subtraktion a=a-b
+=	a+=b	0	zuweisende Addition a=a+b
=	a=b	0	Zuweisung von b zu a

Zuweisungen mit Bitoperationen und den Kommaoperator von C kennt Floyd nicht.

Achtung: Der Zuweisungsoperator = darf nicht mit dem Vergleichsoperator == verwechselt werden.

Alle logischen Operatoren und Vergleiche liefern als Ergebnis entweder eine 1 (wahr) oder 0 (falsch). Alle anderen Werte werden ebenfalls als wahr interpretiert. Die aus anderen Sprachen bekannten Werte *true* und *false* können mit Konstanten abgebildet werden:

```
#define TRUE  1
#define FALSE 0
```

Die in Spielen am häufigsten gebrauchten Operatoren sind && (UND), || (ODER) und ! (NICHT). Mit ihnen lassen sich komplexe Vergleiche ausdrücken, von deren Ergebnis das weitere Spielgeschehen abhängt. Der Operator && liefert 1, wenn beide Operanden ungleich 0 sind. Der Operator || liefert 1, wenn wenigstens einer seiner Operanden ungleich 0 ist. Der Operator ! arbeitet mit nur einem Operanden und kehrt dessen Wert um. Hat z. B. die Variable *a* einen Wert ungleich 0, dann ist !a gleich 0. Umgekehrt wird aus 0 durch !0 eine 1.

Die Auswertungsreihenfolge der einzelnen Operatoren ergibt sich aus ihrem Rang in der Tabelle, wobei Operatoren mit höherem Rang eher ausgewertet werden. Die Reihenfolge läßt sich wie üblich mit runden Klammern beeinflussen:

```
ergebnis=3+4*7;    // 31
ergebnis=(3+4)*7;  // 49
```

Floyd kennt den Konditionaloperator, der mit drei Operanden arbeitet. Der erste Operand muß eine Zahl (bzw. ein numerischer Ausdruck) sein. Das Ergebnis des Konditionaloperators hängt vom Wert des ersten Operanden ab. Ist er ungleich 0, wird der Wert des zweiten Operanden zurückgegeben, andernfalls der Wert des dritten Operanden.

```
string s;
s=(2<8) ? "kleiner" : "größer";
```

Der Operator ++ addiert 1 zu dem Wert seines einzigen Operanden (d.h. er inkrementiert ihn). Der Operand muß dabei eine Integervariable sein. Wenn der Operator ++ vor seinem Operanden steht, inkrementiert er ihn zuerst und liefert dann dessen erhöhten Wert zurück (Prä-Inkrement). Steht er hinter dem Operanden, gibt er erst dessen unveränderten Wert zurück und erhöht ihn dann (Post-Inkrement). Der Operator -- subtrahiert 1 vom Wert seines Operanden, verhält sich aber sonst gleich wie der Inkrementoperator. Die Operatoren ++ und -- werden häufig verwendet, um Zählervariablen zu verändern.

```
int a=5, b;
```

```
b=a++;      // b=5, a=6
b=a--;      // b=6, a=5
b=--a;      // b=4, a=4
```

Nur sehr selten benötigt werden die Bitoperatoren, die mit den einzelnen Bits ihrer Operanden arbeiten. Die Operanden müssen Integervariablen oder Konstanten sein. Wie die Operatoren & (UND), | (ODER) und ^ (XOR) die 32 Bits ihrer Operanden einzeln verknüpfen, zeigen folgende Tabellen:

	0	1
0	0	0
1	0	1
	UND	

	0	1
0	0	1
1	1	1
	ODER	

	0	1
0	0	1
1	1	0
	XOR	

Der Operator ~ invertiert die Bits seines Operanden, aus 0 wird 1 und umgekehrt. Aufgrund der Binärdarstellung von Zahlen mit Vorzeichen ist die bitweise Negation einer Zahl gleichbedeutend damit, ihr Vorzeichen umzudrehen und 1 zu subtrahieren. Wie das Zeichen ^, das als XOR-Operator dient und in Strings einen Zeilenumbruch markiert, kommt dem Operator ~ später im Zusammenhang mit der *with*-Anweisung noch eine andere Bedeutung zu.

Mit den Operatoren << (Links-Shift) und >> (Rechts-Shift) läßt sich das Bitmuster der Binärdarstellung einer Zahl nach links oder rechts verschieben. Eine Verschiebung um ein Bit nach links ist gleichbedeutend mit einer Multiplikation mit 2. Eine Verschiebung um zwei Bits entspricht einer Multiplikation mit 4 usw. Umgekehrt ist eine Verschiebung nach rechts gleichbedeutend mit einer Ganzzahldivision.

2.12 Funktionen

Neben der Funktion *main()*, bei der jedes Programm beginnt, werden sich in einem Adventure viele weitere Funktionen finden. Soll eine Funktion einen Wert zurückgeben, muß statt des Schlüsselwortes *void* nur der Typ des Rückgabewerts *(int, string* oder *object*) angegeben werden. Der Rückgabewert wird mit der *return*-Anweisung zurückgeliefert. Diese Anweisung muß in jeder Funktion vorkommen, die einen Rückgabewert hat. In *void*-Funktionen darf sie dagegen nicht vorkommen.

```
// liefert den groesseren von zwei Werten
int max(int a, int b) {
   return((a>b) ? a : b);
}

// liefert Namen eines Wochentages (0-6)
string wochentag(int tag) {
   string namen[7]=("Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag",
      "Samstag", "Sonntag");
   return(namen[tag]);
}

// liefert Zahl der Minuten seit Mitternacht als Uhrzeit
string uhrzeit(int minuten) {
   int stunden;
   stunden=minuten/60;
   minuten=minuten-stunden*60;
   return(stunden+":"+minuten);
}
```

Achtung: Ab Version 3.0 beendet die *return*-Anweisung eine Funktion. Zeilen nach *return* werden also nie ausgeführt. Vor 3.0 bestimmte *return* nur den Rückgabewert und die Funktion wurde erst mit der abschließenden Klammer beendet.

Ab Version 3.1 sind die Klammern bei der *return*-Anweiung optional. Neben der Variante

```
return(1);
```

ist jetzt auch, wie in anderen Sprachen, die Variante

```
return 1;
```

möglich.

Aufgerufen werden Funktionen einfach mit ihrem Namen und ihren in Klammern gesetzten Argumenten.

```
void main() {
  int x;
  x=max(6,2);
  write(wochentag(x)+" "+uhrzeit(870));
}
```

In Kapitel 2.8 wurde schon gezeigt, wie sich mehrdimensionale Arrays simulieren lassen. Die folgenden beiden Funktionen setzen und lesen Werte in einem zweidimensionalen Integer-Array:

```
#define SPALTEN 8
int a[64]; // 8*8 Elemente

void setA(int i, int j, int wert) {
 a[j*SPALTEN+i]=wert;
}

int getA(int i, int j) {
   return(a[j*SPALTEN+i]);
}

void main() {
   setA(3,6,12);
   write(getA(3,6));
}
```

Ein Anwendungsbeispiel für die schon in Kapitel 2.6 vorgestellten Stringfunktionen von Floyd ist die Funktion

```
string upper_lower(string s0, int upper, int first) {
   string a0="ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ",
   a1="abcdefghijklmnopqrstuvwxyzäöü",s1;
   int i,j,k,n;
   n=(first) ? 1 : strlen(s0);
   while (i<n) {
         s1=substr(s0,i,1);
         k=(upper) ? strstr(s1,a1) : strstr(s1,a0);
         if (k>=0) {
            s1=substr(s0,0,i);
            s1+=(upper) ? substr(a0,k,1) : substr(a1,k,1);
            s1+=substr(s0,i+1,strlen(s0));
            s0=s1;
         }
```

```
        i++;
    }
    return(s0);
}
```

Sie wechselt die Groß- und Kleinschreibung in einem String. Der Parameter *s0* ist dabei der zu ändernde String. *upper* bestimmt, ob Klein- in Großbuchstaben (1) oder Groß- in Kleinbuchstaben (0) geändert werden sollen. Mit dem Wert 1 für *first* wird nur der erste Buchstabe in *s0* geändert, was für klein geschriebene Wörter praktisch ist, die bei der Ausgabe sowohl mitten im Satz als auch am Satzanfang stehen können. Auf die verwendete *while*-Anweisung wird im nächsten Kapitel eingegangen.

Die Zahl der Argumente beim Aufruf einer Funktion muß mit der in ihrer Deklaration übereinstimmen.

Die Wertübergabe bei Funktionsaufrufen erfolgt immer durch einen "call by value". So wird bei dem Aufruf *wochentag(x)* der Wert der Variablen *x* in die Variable *tag* der Funktion *wochentag()* kopiert. Die Funktion *wochentag()* kann daher nicht den Wert der Variablen *x* in der Funktion *main()* verändern.

Ab Versionen 3.0 dürfen in den Indexklammern eines Arrays neben Konstanten und Ausdrücken (*a[2], a[i], a[n+1]*) auch Rückgabewerte von Funktionen (*a[funktion()]*) stehen.

2.13 Kontrollstrukturen

Um Anweisungen zu wiederholen oder nur unter bestimmten Bedingungen auszuführen, kennt Floyd die gleichen Kontrollstrukturen wie C.

while, do-while und for

Die Anweisungen *while*, *do-while* und *for* wiederholen eine oder mehrere Anweisungen abhängig von einer Laufbedingung. Die Anweisungen *while* und *for* testen diese Bedingung zu Beginn eines Durchlaufs, die *do-while*-Anweisung am Ende eines Durchlaufs. Die zu wiederholenden Anweisungen müssen dabei immer zwischen geschweiften Klammern stehen.

Die *while*-Anweisung führt die Anweisungen zwischen den geschweiften Klammern solange aus, wie ihre Bedingung wahr ist.

```
// Syntax der while-Schleife
while (Ausdruck) {
   Anweisungen
}

// Beispiel für eine while-Schleife
int anzahl=0;
while (anzahl<5) {
   anzahl++;
}
```

Die *for*-Anweisung faßt die Laufbedingung und die Veränderung einer Zählervariablen zusammen.

```
// Syntax der for-Schleife
for (Ausdruck1; Ausdruck2; Ausdruck3) {
   Anweisungen
}
```

Ausdruck1 initialisiert die Laufvariable und wird nur einmal ausgeführt. Ergibt *Ausdruck2* den Wert 0, so wird die Schleife beendet. Ergibt *Ausdruck2* einen Wert ungleich 0, so findet ein Schlei-

fendurchlauf statt und *Ausdruck3* wird ausgeführt. Die Schleife startet anschließend wieder mit der Auswertung von *Ausdruck2*.

```
// eine einfache for-Schleife
int anzahl;
for (anzahl=1; anzahl<10; anzahl++) {
   write(anzahl+"^");
}
```

Im Gegensatz zu den *while-* und *for-*Schleifen wird die Laufbedingung bei der do-whileSchleife erst am Ende eines Durchlaufs getestet. Das bedeutet, daß die abhängigen Anweisungen in jedem Fall mindestens einmal ausgeführt werden.

```
// Syntax der do-while-Schleife
do {
   Anweisungen
} while(Ausdruck);

// eine do-while-Schleife
int anzahl;
do {
   anzahl++;
} while(anzahl<10);
```

Schleifen dürfen beliebig tief verschachtelt werden, d. h. in den abhängigen Anweisungen einer Schleife darf wieder eine Schleife vorkommen.

Achtung: Die *while-, do-while-* und *for-* Schleifen produzieren Endlosschleifen, wenn ihre Abbruchbedingung nie erfüllt wird. Hier hilft meist nur noch, Floyd über die Taskbar zu beenden (Rechtsklick, schließen oder beenden).

if und switch

Die *if*-Anweisung erlaubt es, eine oder mehrere Anweisungen nur dann auszuführen, wenn eine bestimmte Bedingung wahr ist. Es sind zwei Formen der *if*-Anweisung möglich. Die erste ist:

```
if (Bedingung) {
   Anweisungen
}
```

Die Anweisungen zwischen den geschweiften Klammern werden nur ausgeführt, wenn der Ausdruck einen Wert ungleich 0 liefert.

```
if (OrkName=="") {
   OrkName="Ozmoz";
}
```

Wenn die Bedingung nicht wahr ist, wird das Programm nach der *if*-Anweisung fortgesetzt. Mit dem Schlüsselwort *else* ist es möglich, Anweisungen nur dann ausführen zu lassen, wenn die Bedingung nicht erfüllt ist.

```
if (Bedingung) {
  Anweisungen
}
else {
   alternative Anweisungen
}
```

Ein Beispiel:

```
if (OrkName=="") {
   write("Du begegnest einem der unzähligen namenlosen Orks.^");
}
else {
   write("Du wirst von dem grausamen "+OrkName+" überrascht.^");
}
```

In Spielen müssen oft mehrere alternative Fälle unterschieden werden, so daß in den *else*-Teil weitere *if*-Anweisungen eingefügt werden.

```
if (Ausdruck1) {
  if (Ausdruck2) {
   // 1.Fall   }
 else {
   // 2.Fall   }
}
else {
   if (Ausdruck3) {
     // 3.Fall     }
   else {
     // diesem Fall ist schwer zu folgen     }
 }
}
```

Verschachtelte if-Anweisungen werden schnell unübersichtlich. Eine übersichtlichere Fallunterscheidung bei mehr als zwei Fällen ermöglicht die *switch*-Anweisung. Sie vergleicht den Wert eines Ausdrucks mit mehreren Konstanten.

```
// Syntax der switch-Anweisung
switch (Ausdruck) {
 case(Fall1);
     Anweisungen
     break;
  case(Fall2);
     Anweisungen
     break;
  default;
     Anweisungen
}
```

Fall1 und *Fall2* dürfen Konstanten oder Variablen sein:

```
case(7); case("Text"); case(var);
```

Das Ergebnis des Ausdrucks wird mit den *case*-Markern verglichen. Stimmt der Wert des Ausdrucks mit einer der Konstanten oder Variablen überein, werden alle Anweisungen ab diesem *case* ausgeführt.

Wenn nur die Anweisungen des passenden *case*-Markers ausgeführt werden sollen, muß die *break*-Anweisung eingefügt werden, die die *switch*-Anweisung beendet. Die *break* Anweisung muß dabei nicht die letzte Anweisung des entsprechenden *case*-Markers sein, folgende Anweisungen werden einfach ignoriert.

Stimmt der Wert des Ausdrucks mit keiner der *case*-Konstanten überein, so werden die Anweisungen nach der *default*-Marke ausgeführt, sofern diese vorhanden ist.

Der Ausdruck der *switch*-Anweisung wird meist einen Integerwert liefern. Er kann aber auch vom Typ String oder Objekt sein.

Der Ausdruck kann auch mit mehreren Konstanten verglichen werden, die zur gleichen Fallbehandlung führen.

```
int n=3;

switch (n) {
   case(1);
   case(2);
      write("n ist gleich 1 oder 2");
      break;
   case(3);
      write("n ist genau gleich 3");
      break;
   default;
      write("n liegt nicht zwischen 1 und 3");
}
```

2.14 Direktiven

Viele Compilersprachen kennen Direktiven genannte Anweisungen, mit denen sich die Übersetzung eines Programms steuern läßt. Sie bestimmen, welche anderen Dateien noch in den Quelltext eingebunden oder welche Programmteile nur unter bestimmten Bedingungen übersetzt werden.

Floyd ist eine Interpretersprache, kennt aber trotzdem einige Direktiven. Sie beginnen mit dem Zeichen # und werden vor dem Start von Programmen im Quelltext ausgeführt.

Achtung: Vor dem # Zeichen dürfen keine Leerzeichen stehen und die Zeile darf nicht mit einem Semikolon enden.

In compilierten Programmen werden Direktiven ignoriert. Die am häufigsten gebrauchte Direktive ist *#define*, die Sie schon bei der Definition von Konstanten kennengelernt haben.

Einbinden von Dateien

Floyd-Programme können aus mehreren Quelltextdateien bestehen. Häufig benötigte Funktionen, Konstanten und globale Variablen können in separaten Dateien gespeichert und in das Hauptprogramm eingebunden werden. Mit der Direktive

```
#include <Dateiname>
```

wird eine Datei in den Quelltext aufgenommen. Wenn mit dem Dateinamen nicht auch ein Pfad angegeben wird, sucht Floyd die Datei im aktuellen Programmverzeichnis. Liegt die einzubindende Datei in einem anderen Ordner, kann ab Version 3.1 auch ein relativer Pfad angegeben werden:

```
#include <stdlib/stdconst.floyd>
```

oder

```
#include <../stdlib/stdconst.floyd>
```

Dabei werden einzelne Ordner unabhängig vom Betriebssystem immer mit einem Slash (/) getrennt.

Eingebundene Dateien sollten die Endung .floyd haben und selbst keine weiteren Dateien einbinden. Wenn eine Quelltextdatei, die andere Dateien einbindet, compiliert wird, werden die eingebundenen Dateien nicht mehr benötigt, um die compilierte Datei auszuführen.

Wenn ein Programm in mehrere Dateien aufgeteilt wird, müssen Abhängigkeiten zwischen den Dateien in der Reihenfolge der *include*-Anweisungen berücksichtigt werden, s. auch 3.2.

Bedingte Ausführung

Der Interpreter kann Teile des Quelltextes ignorieren. Das kann z.B. sinnvoll sein, wenn bestimmte Programmteile für einen Betatest nicht in das fertige Spiel aufgenommen werden sollen. Der fragliche Teil des Quelltextes wird mit der Direktive

```
#ifdef SYMBOL
```

eingeleitet und endet mit der Direktive

```
#endif
```

Die #*ifdef*-Direktive prüft, ob *SYMBOL* bereits mit #*define* definiert wurde. Ist das nicht der Fall, wird der Quelltext bis #*endif* ignoriert. Umgekehrt prüft die Direktive

```
#ifndef SYMBOL
```

ob, das *SYMBOL* noch nicht definiert wurde. Ist es bereits definiert, wird der Quelltext bis #*endif* ignoriert.

Bei *SYMBOL* kann es sich um eine Konstante oder einfach um einen definierten Bezeichner ohne Wert handeln:

```
#define EINE_KONSTANTE 23
#define NUR_BEZEICHNER
```

Ein Bezeichner ohne Wert macht Sinn, wenn er nur für #*ifdef* und #*ifndef* verwendet wird.

3 Klassen

Klassen sind in Floyd das wichtigste Sprachelement zur Programmierung von Adventures. Alle Elemente der Spielwelt sind Klassen: alle Gegenstände, Räume, Türen, Nichtspielercharaktere (NPCs) und auch der Spielercharakter selbst. Es lassen sich in Floyd einfache Programme ohne Klassen schreiben, aber richtige Spiele sind nur mit Klassen möglich.

Genauer ist eine Klasse in Floyd eine Datenstruktur, die über Funktionen und Variablen verfügt und einen kleinen Teil der Spielwelt simuliert. Sie reagiert auf Eingaben des Spielers und bestimmt selbständig, was er mit ihr machen kann.

In dem Spiel *Der Nebelmond* gibt es z.B. ein Kabel, mit dem sich ein Roboter und ein Computerinterface verbinden lassen. Die entsprechende Klasse weiß, welches ihrer Kabelenden bereits im Roboter oder im Interface steckt. Sie benachrichtigt andere Klassen, wenn die Verbindung steht und antwortet auch auf Versuche des Spielers, sie mit anderen Dingen als dem Roboter oder dem Interface zu verbinden.

3.1 Wie Klassen deklariert werden

Die Deklaration einer Klasse beginnt mit dem Schlüsselwort *class*, dem der Klassenname folgt. Zwischen geschweiften Klammern folgen die Variablen und Funktionen der Klasse.

```
class MeineKlasse {

   // Variablen
   int a, b;

   // Funktionen
   void mach_was() {
      write("a ist "+a+" und b ist "+b);
   }
}
```

Die zu einer Klasse gehörenden Funktionen werden in anderen Programmiersprachen Methoden der Klasse genannt, um sie von globalen Funktionen zu unterscheiden. Auch im weiteren Text soll dieser Begriff benutzt werden.

3.2 Zugriff auf die Daten einer Klasse

Auf die Variablen der Klasse kann in allen ihren Methoden zugegriffen werden. Um von außerhalb der Klasse auf ihre Variablen und Methoden zugreifen zu können, muß ihnen der Klassenname und ein Punkt vorangestellt werden.

```
void main() {
  if (MeineKlasse.a>0) {
    MeineKlasse.mach_was();
  }
}
```

In den meisten anderen Programmiersprache, die Klassen kennen, gibt es Möglichkeiten, den Zugriff auf einzelne Variablen und Methoden einer Klasse von außen zu verhindern. Dieser Zugriffsschutz wird Datenkapselung genannt und soll die Unabhängigkeit der Klasse vom Rest des Programms sicherstellen. Floyd kennt keine zugriffsgeschützten Variablen oder Methoden. Damit eine Klasse dennoch kontrollieren kann, wie auf ihre Variablen zugegriffen wird, kann die Veränderung von Variablen über Methoden erfolgen.

```
class Uhr {
```

```
   int stunde, minute;

   void zeigeZeit() {
      write(stunde+":"+minute);
   }

/* Kontrolliert Zuweisung an stunde und minute und
korrigiert ungültige Werte */

   void setzeZeit(int s, int m) {
      if (s<0) {
         s *= -1;
      }
      if (m<0) {
         m *= -1;
      }
      stunde = s % 24;
      minute = m % 60;
   }

}

void main() {
   Uhr.setzeZeit(30,-15);
   Uhr.zeigeZeit();
}
```

Achtung: Wenn Klassen in einzelne Dateien ausgelagert und mit *include* in das Hauptprogramm eingefügt werden, müssen Abhängigkeiten zwischen den Klassen berücksichtigt werden, d. h., wenn Klasse A in ihrem Code auf Klasse B zugreift, muß B auch vor A eingebunden werden. Wenn zwei Klassen gegenseitig aufeinander zugreifen, müssen sie in der gleichen Datei stehen.

3.3 Der Datentyp Objekt

Neben Integers und Strings kennt Floyd noch Variablen vom Typ Objekt. Sie sind Platzhalter für Klassen und werden mit dem Schlüsselwort *object* deklariert.

```
class Taschenlampe {
   string _name="Taschenlampe";
   int brenntRunden=7;
}

class Fackel {
   string _name="Fackel";
   int brenntRunden=4;
}

void main() {
   object a,b,c;
   a=Taschenlampe;
   b=Fackel;
   if (a.brenntRunden<b.brenntRunden) {
      c=a;
      a=b;
      b=c;
   }
   write("Die "+a._name+" brennt länger als die "+b._name);
}
```

Bei der Deklarationen kann einer Objektvariablen auch schon eine Klasse zugewiesen werden:

```
object a = Taschenlampe;
```

Solange einer Objektvariablen keine Klasse zugewiesen ist, hat sie den Wert NULL. Dieser Wert bedeutet einfach "steht für keine Klasse" und kann der Variablen auch zugewiesen werden.

Für Objektvariablen gibt es die Kontrollanweisung *fetch()*.

```
fetch(Objektvariable, Ausdruck, int Reichweite) {
  Anweisungen
}
```

Diese Anweisung ist eine Schleife über alle Klassen, mit denen der *Ausdruck* wahr wird. Sie setzt für die *Objektvariable* alle im Programm deklarierten Klassen, die einen Namen haben, ein und führt die Anweisungen in ihrem Rumpf für jede Klasse, mit der der *Ausdruck* wahr wird, einmal aus. Die *Objektvariable* muß hierfür auch im *Ausdruck* vorkommen.

Das folgende Beispiel gibt die Namen aller Klassen aus, deren Variable *brenntRunden* größer als zwei ist. Die Anweisung *setShort()* bestimmt den Namen einer Klasse und wird in Abschnitt 3.5 genauer vorgestellt.

```
class Taschenlampe {
   setShort("-Taschenlampe");
   int brenntRunden=7;
}

class Fackel {
   setShort("-Fackel");
   int brenntRunden=4;
}

class Kerze {
   setShort("-Kerze");
   int brenntRunden=2;
}

void main() {
   object x;
   fetch (x, x.brenntRunden>2, 1) {
      write(x+"^");
   }
}
```

Der Parameter *Reichweite* bestimmt, welche Klassen in einem Spiel berücksichtigt werden. Hat *Reichweite* den Wert 0, wird der Ausdruck nur mit den Klassen getestet, die sich mit dem Spieler in einem Raum befinden. Für den Wert 1 werden alle im Programm deklarierten Klassen berücksichtigt. Ist das Programm kein Spiel (wie die bisherigen Beispiele), bleibt *Reichweite* unberücksichtigt. In der Praxis reicht es meist, die Anweisung auf die Klassen in der Nähe des Spielers zu beschränken.

Die Zahl der Klassen, für die der Ausdruck wahr wird, kann bereits in der Schleife mit der Systemvariablen

```
int count
```

ermittelt werden. Das ist möglich, weil der Interpreter bereits vor dem ersten Schleifendurchlauf, alle passenden Klassen sucht.

Übrigens macht der Datentyp *object* auch den Unterschied zwischen Floyd und anderen objektorientierten Sprachen deutlich. In den meisten anderen objektorientierten Sprachen sind Klassen

Vorlagen für mehrere Objekte, die alle über die gleichen Methoden und Variablen verfügen, sich aber in ihren Werten voneinander unterscheiden können. In Adventures kommen die meisten Objekte der Spielwelt aber nur einmal vor. Daher gibt es in Floyd keinen Unterschied zwischen Klassen und Objekten.

Seit Version 3 sind auch Funktionsketten möglich, d. h. über Funktionen, die ein Objekt zurückgeben, kann direkt auf die Methoden und Variablen des Objekts zugegriffen werden:

```
funktion().methode()
funktion().variable
```

3.4 Attribute

Neben Variablen und Methoden können Klassen auch über Attribute verfügen. Attribute dienen der Darstellung dichotomer Eigenschaften und lassen sich nur auf ihr Vorhandensein überprüfen. Eine Klasse kann ein bestimmtes Attribut haben oder nicht haben.

Wenn in einem Spiel z.B. eine Kaffeemaschine vorkommt, kann die entsprechende Klasse das Attribut *eingeschaltet* bekommen, wenn der Spieler sie einschaltet. Die Attribute *Wasser* und *Pulver* lassen sich setzen, wenn der Spieler Wasser und Kaffeepulver einfüllt.

Um einer Klasse ein Attribut hinzuzufügen, wird die Anweisung

```
with(Attributname);
```

verwendet. Für Attributnamen gelten die gleichen Regeln wie für Variablennamen. Mit der Anweisung *with* können auch mehrere durch Kommata getrennte Attribute gleichzeitig hinzugefügt werden.

```
with(A1, A2, ..., An);
```

Ein Attribut läßt sich wieder entfernen, indem in der *with*-Anweisung seinem Namen das Zeichen ~ (Tilde) vorgestellt wird.

```
with(~Attributname);
```

Übrigens ist es kein Fehler, wenn versucht wird, ein nicht vorhandenes Attribut zu entfernen oder ein Attribut zweimal hinzuzufügen (es ist immer nur einmal vorhanden).

Die *with*-Anweisung gehört zu den wenigen Anweisungen, die außerhalb einer Methode direkt in die Klasse geschrieben werden dürfen. Sie wird dann beim Programmstart ausgeführt.

```
class MeineKlasse {
   with(MeinAttribut);
   void mach_was() {
      with(~MeinAttribut);
   }
}
```

Mit der Funktion

```
int has(Attributname)
```

läßt sich feststellen, ob eine Klasse über ein bestimmtes Attribut verfügt. Die Funktion liefert entweder 1 (wahr) oder 0 (falsch) zurück. Für das Attribut können keine Variablen oder Konstanten angegeben werden.

Schließlich setzt oder löscht die Anweisung

```
toggle(Attributname)
```

noch ein Attribut, abhängig davon, ob es schon vorhanden ist. Ein bereits vorhandenes Attribut wird entfernt, ein noch nicht vorhandenes hinzugefügt.

3.5 Namen

Damit der Spieler eine Klasse in der Eingabe erwähnen kann, muß sie einen Namen erhalten. Der Name der Klasse im Spiel ist von dem hinter dem Schlüsselwort *class* vergebenen Namen (dem internen Namen) unabhängig und wird mit der Anweisung

```
setShort(string Name)
```

bestimmt. In dem String können auch mehrere durch Kommata getrennte Namen (Synonyme) angegeben werden. In diesem Fall kann sich der Spieler mit allen Namen auf die Klasse beziehen. Im Spiel gibt Floyd aber immer nur den ersten Namen aus. Der Index des vom Spieler eingegebenen Namens in der Liste der Klassennamen wird mit der Funktion

```
int name()
```

ermittelt. Der erste Name hat den Index 0. Hat eine Klasse also z.B. die Namen Pudel, Hund und Tier, liefert *name()* den Wert 2, wenn der Spieler die Klasse mit "Tier" erwähnt. Zusätzlich muß Floyd mitgeteilt werden, ob der Name ein Femininum, Maskulinum, Neutrum, ein Eigenname oder ein Pluralname ist. Dafür werden dem Namen folgende Zeichen vorgestellt:

Zeichen	Geschlecht	Beispiel
+	Maskulinum	+Wagen
-	Femininum	-Kristallkugel
$	Eigenname	$Tom
&	Pluralname	&Leute

Beginnt ein Name mit keinem dieser Zeichen, gilt er als Neutrum. Eine Klasse darf auch Namen unterschiedlichen Geschlechts haben:

```
setShort("Auto, +Wagen, -Limousine");
```

Eigennamen mit den Endbuchstaben a, e und i werden bei ihrer Deklination als weibliche, alle anderen als männliche Namen behandelt. Das Geschlecht des ersten Klassennamens läßt sich mit der Funktion

```
int gender()
```

ermitteln. Sie liefert den Wert einer der hierfür in *stdconst.floyd* deklarierten Konstanten

```
G_MASC    0
G_FEM     1
G_NEUTRA  2
G_NAME    3
G_PLURAL  4
```

Namen werden von Floyd dekliniert, um sie in der Spielereingabe in unterschiedlichen Formen erkennen zu können. Floyd orientiert sich bei der Deklination eines Namens an seinem Geschlecht und an seinen letzten beiden Silben. Das funktioniert bisher mit allen getesteten deut-

schen Hauptwörtern. Bei vielen Fremdwörtern funktionieren die Deklinationsregeln leider nicht, z.B. kann der Roboter im *Nebelmond* auch englisch Robot genannt werden. Wird er mit einem Artikel im Akkusativ erwähnt ("u den Robot"), erkennt ihn Floyd nicht. Dieses Problem läßt sich ab Version 3.1 mit einem Fragezeichen hinter dem Namen verhindern:

```
setShort("+Robot?");
```

Das Fragezeichen verhindert die Deklination. Nur im Genitiv bekommt der Name dann noch ein s oder ein Hochkomma angehängt. Bei Eigennamen ($) und Pluralnamen (&) wird das Fragezeichen nicht berücksichtigt.

Ein Name kann auch ein Adjektiv enthalten. Es wird in seiner Grundform gefolgt von einem Stern vor den Namen geschrieben:

```
setShort("+rot* Ball");
```

Auch zusammengesetzte Namen, z. B. *Quell des Lebens*, werden richtig dekliniert. Wenn in ihnen ein Adjektiv vorkommt, z. B. *Wächter des schwarzen Turms*, sollte es immer ausgeschrieben und nicht mit einem Stern angegeben werden, da es als Teil des Namens nie seine Form ändert.

Mehrere gleichartige Objekte

Ein Klassenname mit Adjektiv wird in der Eingabe auch erkannt, wenn der Spieler das Adjektiv weg läßt. Hierbei kann es zu Mehrdeutigkeiten kommen, wenn sich mehrere Objekte nur durch ihre Adjektive im Namen unterscheiden (ein roter und ein blauer Ball). Wenn in diesem Fall die betroffenen Objekte den gleichen Pluralnamen bekommen, kann Floyd den Spieler fragen, welches der Objekte er meint:

```
class ball_1:stditem {
   setShort("+rot* Ball,&Bälle");
   moveto(raum);
}

class ball_2:stditem {
   setShort("+blau* Ball,&Bälle");
   moveto(raum);
}
```

Wenn sich beide Bälle im gleichen Raum befinden und der Spieler die Farbe nicht erwähnt, bittet Floyd um eine genauere Angabe:

>x den Ball
Ich bin mir nicht sicher, was du meinst: den roten Ball oder den blauen Ball?

>den blauen Ball
Daran ist nichts Besonderes zu entdecken.

Die Antwort des Spielers ist hierbei nicht auf die in der Frage genannten Objekte beschränkt. Er kann auch mit irgendeinem anderen erreichbaren Objekt antworten oder eine ganz neue Anweisung eingeben.

Objekte mit einem gemeinsamen Pluralnamen lassen sich auch gemeinsam manipulieren:

>nimm die Bälle
Du trägst jetzt den blauen Ball.
Du trägst jetzt den roten Ball.

```
>lege die Bälle ab
Du legst den blauen Ball ab.
Du legst den roten Ball ab.
```

Damit der Parser auch Mengenangaben versteht („nimm zwei Bälle“), müssen in *stdconst.floyd* für die gewünschten Verben neue Satzschablonen angelegt werden:

```
verb("nimm|nehme #number #noun",A_TAKE,0);
```

Der Parser überträgt die angegebene Zahl in die Systemvariable *number* und versucht, die jeweilige Aktion entsprechend oft auszuführen.

Objekte mit unterschiedlichen Adjektiven und gleichen Pluralnamen sind in Floyd die einzige Möglichkeit, mehrere gleichartige Dinge zu simulieren. Die Unterscheidung von Objekten durch Positionsangaben wie in dem Satz „nimm den Ball auf dem Tisch (und nicht den unter dem Stuhl)“ ist leider noch nicht möglich (mit Ausnahme von Satzschablonen, die auf eine bestimmte Angabe passen).

Anreden und Titel

Wenn der Name die Anreden bzw. den Titel Mr., Mrs., Ms. oder Dr. enthält kann der Spieler ihn auch mit dem Punkt eingeben. Vor der Auswertung der Eingabe werden Punkte hinter diesen Zeichenfolgen entfernt, so daß sie nicht als Satztrennzeichen dienen. Damit der Name vom Parser ohne Punkt erkannt, aber immer mit Punkt ausgegeben wird, sollte zuerst die Version mit Punkt angegeben werden:

```
setShort("$Mr. Bond, $Mr Bond");
```

Die Ausgabe von Klassennamen mit getShort()

Um den Klassennamen auszugeben, gibt es zwei Möglichkeiten. Die erste ist die Funktion

```
string getShort(int Kasus, int Artikel)
```

Der Parameter *Kasus* bestimmt den gewünschten Fall und muß einer der in *stdconst.floyd* deklarierten Konstanten C_xxx entsprechen:

```
C_NOM 0  // Nominativ
C_ACC 1  // Akkusativ
C_GEN 2  // Genitiv
C_DAT 3  // Dativ
```

Der Parameter *Artikel* bestimmt, ob der Name ohne (0), mit einem bestimmten („der“, 1) oder einem unbestimmten Artikel („ein“, 2) ausgegeben wird.

Das folgende Programm gibt zwölf Formen des Namens "roter Löwe" aus:

```
#include <stdconst.floyd>

class loewe {
   setShort("+rot* Löwe");
}

void main() {
   int k,a;
   for (k=C_NOM; k<=C_DAT; k++) {
```

```
      for (a=0; a<3; a++) {
         write(loewe.getShort(k,a)+"^");
      }
   }
}
```

Die Ausgabe von Klassennamen in Strings

Die zweite, etwas flexiblere Methode zur Ausgabe von Klassennamen besteht darin, sie direkt in den Text einzufügen. Dafür wird in den String die Zeichenfolge

```
<[~][!]Artikel Klasse [VerbSingular VerbPlural]>
```

eingefügt. Die Angaben in den eckigen Klammern sind optional. Der Artikel muß immer im Maskulinum stehen, da sich nur hier die Fälle eindeutig unterscheiden lassen. Wird dem Artikel eine Tilde vorgestellt, wird er nicht mit ausgegeben. Ein Ausrufezeichen gibt eine Verneinung im entsprechenden Fall aus. Der Artikel kann auch groß geschrieben werden, wenn er an einem Satzanfang steht.

Artikel	Nominativ	Akkusativ	Genitiv	Dativ
bestimmt	der	den	des	dem
unbestimmt	ein	einen	eines	einem
ohne Artikel	~der	~den	~des	~dem
verneinend	!ein	!einen	!eines	!einem

Die Verneinung funktioniert auch mit bestimmten Artikeln.

Nach dem Artikel folgt die Klasse oder eine Objektvariable. Wenn mit einer Objektvariablen Singular- und Pluralnamen ausgegeben werden sollen, denen ein Verb folgt, können optional noch beide Verbformen angegeben werden.

```
class apfel {
   setShort("+rot* Apfel");
}

class kiste {
   setShort("-Kiste");
}

class fahrrad {
   setShort("Fahrrad");
}

class leute {
   setShort("&Leute");
}

class tom {
   setShort("$Tom");
```

```
}

class lara {
   setShort("$Lara");
}

void main() {
   object x;
   fetch(x, x!=NULL, 0) {
      write("<Der x>, der Name <des x>, <~der x>, <!der x>,
         <der x wird werden> gelesen.^");
   }
}
```

Personalpronomen

Das zum Klassennamen passende Personalpronomen liefert die Methode

```
string pnoun(int Kasus, int Großschreibung)
```

Der *Kasus* bestimmt wie bei *getShort()* den gewünschten Fall. Wenn *Großschreibung* den Wert 1 hat, wird der erste Buchstabe groß geschrieben.

	Nominativ	Akkusativ	Genitiv	Dativ
Maskulinum	er	ihn	seine	ihm
Femininum	sie	sie	ihre	ihr
Neutrum	es	es	seine	ihm
Plural	sie	sie	ihre	ihnen

Alle Pronomen enden im Genitiv auf e. Hier muß eventuell noch die passende Endung mit ausgegeben werden.

3.6 Beschreibungen

Wenn der Spieler eine Raumbeschreibung lesen oder eine bestimmtes Objekt genauer untersuchen möchte, wird die Beschreibung der entsprechenden Klasse ausgegeben. Dieser Text wird mit der Anweisung

```
setLong(string Beschreibung)
```

bestimmt. Wenn die Klasse während des Spiels ihr Äußeres ändert, kann die Beschreibung mit *setLong()* jederzeit angepaßt werden. Die Funktion

```
string getLong()
```

gibt den aktuellen Beschreibungstext zurück.

Oft muß neben der Objektbeschreibung noch eine Information über den Zustand (z.B. ein- oder ausgeschaltet) oder eine Auflistung des Inhalts von Behältern oder Räumen ausgegeben werden. Dafür gibt es in den Dateien *stditem.floyd* und *stdroom.floyd* der Standardbibliothek entsprechende Funktionen, über die wir im Kapitel 5 noch mehr erfahren werden.

3.7 Positionen und Behälter

Jede Klasse hat eine bestimmte Position in der Spielwelt. Objekte befinden sich z. B. in bestimmten Räumen, ein Taler in einer Geldbörse oder ein Kuchen in einem Ofen, der sich selber wieder in einer Küche befindet.

Mit der Anweisung

```
moveto(object Position)
```

läßt sich eine Klasse in eine andere Klasse bewegen. So können in einem Spiel Objekte einfach in Kisten oder andere Räume verfrachtet werden.

Um festzustellen, wo sich eine Klasse gerade befindet, können die Funktionen

```
object location()
```

und

```
object room()
```

benutzt werden. Sie liefern die Behälterklasse. Der Unterschied zwischen ihnen ist, daß *location()* den ersten Behälter liefert, während *room()* den äußersten Behälter zurückgibt. Im Falle des Kuchens, der sich in einem Ofen befindet, der wiederum in einer Küche steht, liefert *kuchen.location()* also den Ofen und *kuchen.room()* die Küche. Wenn eine Klasse nur einen Behälter besitzt, führen *location()* und *room()* natürlich zum gleichen Ergebnis.

Ob sich eine Klasse in einer bestimmten Behälterklasse befindet, stellt die Funktion

```
int in(object Behälter)
```

fest. Sie berücksichtigt wie *location()* nur den ersten Behälter und liefert entweder 1 oder 0. Die Funktion

```
int items()
```

ermittelt die Zahl der Klassen in einem Behälter. Die folgende Funktion *list()* liefert den Inhalt eines Behälters und aller darin befindlichen Behälterklassen. Dafür ruft sie sich rekursiv selber auf. Eine ausführlichere Version dieser Funktion findet sich in der Datei *stdlist.floyd*.

```
class schatzkammer {
   setShort("-Schatzkammer");
}

class krone {
   setShort("-Krone");
   moveto(schatzkammer);
}

class truhe {
   setShort("-Truhe");
   moveto(schatzkammer);
}

class taler {
   setShort("+Taler");
   moveto(truhe);
}
```

```
void list(object behaelter) {
   object x;
   fetch(x, x.in(behaelter), 0) {
      write(x+"^");
      if (x.items()>0) {
         list(x);
      }
   }
}

void main() {
   list(schatzkammer);
}
```

Wie in dem Programm zu sehen ist, gehört auch *moveto()* neben *with()*, *setShort()* und *setLong()* zu den Anweisungen, die außerhalb einer Methode direkt im Klassenrumpf ausgeführt werden können.

Manchmal sollen bestimmte Objekte nirgendwo vom Spieler zu finden sein oder erst nach einem bestimmten Ereignis in die Spielwelt eingefügt werden. In solchen Fällen bietet es sich an, einfach einen Behälter (eine Art schwarzen Zylinder) zu schaffen, der nicht mit der Spielwelt verbunden ist. Alle Objekte, die vor dem Spieler versteckt werden sollen, können dann in ihm platziert oder wieder aus ihm hervorgezaubert werden.

Achtung: Es ist auch möglich, Objekte in NULL zu positionieren oder Räume ineinander zu verschachteln. So etwas sollte aber vermieden werden, da diese Fälle in der Standardbibliothek nicht berücksichtigt sind.

3.8 Die Initialisierung

Variablen, Attribute, Namen und die Beschreibung und Position einer Klasse können im Klassenrumpf außerhalb von Methoden deklariert bzw. bestimmt werden. Die entsprechenden Anweisungen werden beim Programmstart einmal ausgeführt, um die Klasse zu initialisieren.

Wenn bei der Initialisierung einer Klasse auch Methoden ausgeführt oder Ausdrücke ausgewertet werden sollen, muß die Klasse eine Methode *void init()* besitzen. Diese Methode wird beim Programmstart automatisch ausgeführt und kann beliebige Ausdrücke, Methoden- und Funktionsaufrufe enthalten.

```
class pistole {
   void init() {
      object fundort;
      setShort("-Pistole,-Waffe");
      fundort=(random(2)==0) ? schlafzimer : badezimmer;
      moveto(fundort);
   }
}
```

Die Funktion

```
int random(int n)
```

liefert hierbei eine Zufallszahl zwischen 0 und *n*-1.

3.9 Vererbung

Eine neue Klasse kann bei ihrer Deklaration von einer bereits bestehenden Klasse abgeleitet werden. Sie besitzt dann die gleichen Variablen, Attribute und Methoden wie die Klasse, von der sie abgeleitet wurde. Die abgeleitete Klasse wird Nachfahre ihrer Superklasse genannt.

Diese Ableitung von Klassen wird in anderen Programmiersprachen auch Vererbung genannt und bietet die Möglichkeit, viele Klassen mit ähnlichen Eigenschaften zu erzeugen, ohne die einzelnen Variablen, Attribute und Methoden jedesmal neu deklarieren zu müssen. Wenn in einem Spiel z.B. mehrere Objekte vorkommen, die sich öffnen, schließen, ein- oder ausschalten lassen, müssen die entsprechenden Klassen nur von der in *stdobject.floyd* deklarierten Klasse *stdobject* (oder einem ihrer Nachfahren) abgeleitet werden. Sie verfügen dann automatisch über den für diese Aktionen notwendigen Code.

Das folgenden Beispiel deklariert eine Superklasse monster mit den wichtigsten Eigenschaften eines Monsters. Von dieser Klasse werden dann weitere spezialisierte Gegner abgeleitet.

```
class monster {
   int trefferpunkte, maxSchaden;
   void angreifen() {
      // Code fuer einen Angriff auf den Spieler
   }
}

class kobold:monster {
   void init() {
      setShort("+Kobold");
      trefferpunkte=5;
      maxSchaden=3;
   }
   void verfolgeSpieler() {
      // Code, der den Kobold den Spieler suchen laesst
   }
}

class magier:monster {
   void init() {
      setShort("+Magier");
      trefferpunkte=8;
      maxSchaden=6;
   }
   void zauber() {
      // Code fuer eine Magie-Attacke auf den Spieler
   }
}
```

Wenn eine Superklasse wie in diesem Beispiel nur als Vorlage für weitere Klassen dient und selbst nicht im Spiel auftauchen soll, kann sie auch als abstrakte Klasse deklariert werden:

```
class abstract monster {
}
```

Abstrakte Klassen gehören nicht zur Spielwelt und werden bei der Auswertung der Spielereingaben nicht berücksichtigt. Der Spieler kann sich daher nicht auf sie beziehen. Auch die *fetch*-Anweisung ignoriert abstrakte Klassen.

Oft ist es notwendig, festzustellen, von welcher Superklasse eine Klasse abstammt. Hierfür gibt es das Schlüsselwort *super*. Es liefert die entsprechende Superklasse zurück, wenn die Klasse einen Vorfahren hat. Gibt es keine Superklasse, wird NULL zurückgegeben.

```
kobold.super  // liefert monster
monster.super // liefert NULL
```

Die Objektvariable *this* liefert die aktuelle Klasse. Wenn von einer Superklasse mehrere Nachfahren abgeleitet werden, kann mit *this* in einer vererbten Methode festgestellt werden, zu welchem Nachfahren sie gehört.

3.10 Überladen von Methoden

Von einer Superklasse geerbte Methoden können überladen werden. Dabei wird die geerbte Methode in einer abgeleiteten Klasse neu deklariert, um so das Verhalten der Klasse beim Aufruf dieser Methode zu verändern.

Um bei unseren Monstern zu bleiben, leiten wir von *magier* einen Drachen ab, der ebenfalls zaubern kann, aber natürlich mit anderen Sprüchen und anderer Wirkung.

```
class drachen:magier {
   void init() {
      setShort("+Drache"); // Namen werden nicht vererbt!
      trefferpunkte=15;
      maxSchaden=10;
   }
   void zauber() {
      // Code fuer die Magie-Attacke des Drachen
   }
}
```

Da *zauber()* hier den Code von *magier* vollständig ersetzt, hätte der Drache auch direkt von *monster* abgeleitet werden können. Beim Überladen von Methoden kann bestehender Code aber auch erweitert werden:

```
class drachen:magier {
   void zauber() {
      if (random(2)==0) {
         // Der Drache benutzt die weniger gefaehrlichen
         // Sprueche eines Magiers
         super.zauber();
      }
      else {
         // oder seine eigene Zauberei
      }
   }
}
```

Über das Schlüsselwort *super* kann beim Überladen von Methoden also immer auf den geerbten Code zurückgegriffen werden.

4 Die Spielereingabe

4.1 Ein erster Dialog

Den bisherigen Beispielen fehlt noch der für Adventures typische Dialog zwischen Spielereingabe und Textausgabe. Um die Interaktion mit dem Spieler zu starten, sind die folgenden Schritte notwendig.

Zunächst müssen die acht Dateien der Standardbibliothek am Programmanfang eingebunden werden. Sie enthalten abstrakte Klassen, die als Vorlagen für Räume, Gegenstände und Lebewesen in einem Adventure dienen. Die Datei *stdconst.floyd* enthält einige wichtige Konstanten und muß immer als erste inkludiert werden. Ausführlichere Informationen zu den einzelnen Dateien gibt es in Kapitel 5.

```
#include <stdconst.floyd>
#include <stdobject.floyd>
#include <stdcreature.floyd>
#include <stdlist.floyd>
#include <stdroom.floyd>
#include <stditem.floyd>
#include <stdexit.floyd>
#include <stderror.floyd>
```

Wenn die Standardbibliothek eingebunden ist, kann die Spielwelt definiert werden. Für das einfachste mögliche Beispiel genügen ein Raum und ein Spieler.

```
class raum:stdroom {
   setShort("-Höhle");
   setLong("Eine Höhle ohne Ausgang^");
}

class spieler:stdcreature {
   setLong("Du bist nur ein namenloser Beispiel-Held.^");
}
```

In der *main*-Funktion muß Floyd jetzt noch mitgeteilt werden, welche Klasse als Spieler dienen soll und wo sie sich bei Spielbeginn befindet.

```
void main() {
   // setPlayer() bestimmt die Spielerklasse
   setPlayer(spieler);

   spieler.moveto(raum);

   /* Die erste Raumbeschreibung wird nicht automatisch ausgegeben,
      weshalb in main() auch noch die dafür zuständige Methode description
      aufgerufen werden kann. */
   raum.description();
}
```

Mit *setPlayer()* kann die Spielerklasse auch während des Spiels geändert werden. Die Systemvariable

```
object player
```

enthält immer die aktuelle Spielerklasse.

Achtung: Alle Klassen, die mit *moveto()* in der Spielerklasse abgelegt werden, befinden sich im Inventar, gehören also zu den Objekten, die der Spieler mit sich herumträgt. Natürlich können auch NPCs über ein Inventar verfügen.

Die Funktion *main()* ist auch der geeignete Platz, um einen Prolog, eine Titelzeile und einen Copyright-Hinweis auszugeben.

```
Höhle
Eine Höhle ohne Ausgang

>gehe nach osten
Hier geht's nicht weiter.

>untersuche mich
Du bist nur ein namenloser Beispiel-Held.

>warte
Die Zeit vergeht ...

>ende
```

Dieser erste Dialog ist noch lange kein richtiges Spiel. Bevor er aber ausgebaut werden kann, müssen wir verstehen, wie eine Spielereingabe verarbeitet wird.

4.2 Der Parser

Die Verarbeitung der Eingabe (das sogenannte Parsen) funktioniert in Floyd ganz ähnlich wie in anderen Autorensystemen auch. Die Spielereingabe wird mit einer Liste von bekannten Sätzen (einem Lexikon) verglichen. Wird sie darin gefunden, führt das Spiel eine dem passenden Lexikoneintrag zugeordnete Aktion aus.

Ein neuer Satz wird mit der Anweisung

```
verb(string Satzschablone, int Aktionsnummer, int istMetaverb)
```

in das Lexikon eingetragen. Die Satzschablone besteht aus Wörtern und/oder Platzhaltern für Wörter und wird mit der Spielereingabe verglichen. Wie eine Satzschablone genau aussieht, wird im nächsten Kapitel beschrieben.

Wenn die Spielereingabe mit der Schablone übereinstimmt, wird den in der Eingabe erwähnten Klassen die *Aktionsnummer* mitgeteilt, damit sie auf die Spielereingabe reagieren können. Auf die Verarbeitung der Aktionen wird in Kapitel 4.4 genauer eingegangen.

Der Parameter *istMetaverb* kann die Werte 0 und 1 annehmen und bestimmt, ob nach dieser Eingabe Züge und Spielzeit fortgezählt werden sollen, was bei den sogenannten Metakommandos, die das Spielgeschehen nicht direkt beeinflussen (z.B. "inventar", "objekte" und "version"), nicht notwendig ist. Bei ihnen kann dieser Parameter auf 1 gesetzt werden.

Achtung: Floyd vergleicht die Eingabe mit den Satzschablonen in der Reihenfolge der *verb*-Anweisungen im Quelltext. Wenn eine passende Schablone gefunden wird, werden alle folgenden Schablonen ignoriert. Daher kann es zu unerwartetem Fehlverhalten kommen, wenn mehrere Schablonen auf die gleiche Eingabe passen.

In der Datei *stdconst.floyd* sind die in Spielen am häufigsten vorkommenden Kommandos bereits definiert. Sie dienen

- der Bewegung des Spielers in die acht Richtungen der Kompaßrose sowie nach oben und unten.
- dem Umgang mit Objekten (untersuchen, nehmen, weglegen, öffnen, schließen etc.)
- der Interaktion mit NPCs (u.a. fragen und geben).
- der Anzeige von Statusinformationen (Punkte, Inventar) und ähnlichen Dingen, die den Spielverlauf nicht beeinflussen (die sogenannten Metakommandos)

Die Aktionsnummern sind in *stdconst.floyd* der Übersichtlichkeit halber als Konstanten mit dem Präfix A_ deklariert und reichen von 0 bis über 50. Um in der Standardbibliothek noch Platz zu lassen, sollten neue Aktionsnummern erst bei 100 beginnen.

Der Akteur

Der Spieler kann Anweisungen an NPCs richten, die dann von ihnen ausgeführt werden. Dafür muß die Eingabe mit dem Namen eines NPC gefolgt von einem Komma beginnen.

```
>roboter, gehe nach norden
```

Um festzustellen, wer eine Aktion ausgelöst hat, gibt es die Systemvariable *actor*. Sie enthält entweder die Spielerklasse, dann ist sie gleich *player*, oder die Klasse eines NPC. Wenn der Spieler eine Anweisung an einen NPC richtet, wird nur der Teil der Eingabe nach dem ersten Komma im Lexikon gesucht.

Seit Version 3.2 ist es nicht mehr notwendig, die Reflexivpronomen "mich", "mir", "dich" und "dir" als Namen anzugeben, damit Anweisungen wie

```
>roboter, gib mir die batterie
>computer, schalte dich aus
```

verstanden werden. Stößt der Parser z.B. auf das Wort "mich" und findet es nicht in der Liste der Klassennamen, wird es automatisch als Platzhalter für die Spielerklasse erkannt. Dabei stehen "mich" und "mir" immer für den Spieler, während "dich" und "dir" immer auf den Akteur verweisen.

Achtung: Bei der wörtlichen Rede muß der Text nach dem Namen des angesprochenen NPC in doppelten Anführungszeichen stehen (troll, "Hallo"). Der Akteur, d.h. der Redner, ist dabei immer der Spieler.

4.3 Satzschablonen

Satzschablonen bestimmen, welche Eingaben von einem Spiel verstanden werden. Eine Satzschablone kann aus einem einzelnen Wort, einem Platzhalter oder einer Liste von durch Leerzeichen getrennten Wörtern und Platzhaltern bestehen.

Ein Wort in der Schablone entspricht immer genau dem gleichen Wort in der Spielereingabe. Alternative Wörter können mit dem Zeichen | getrennt werden. Zu beachten ist, daß Floyd mit einem e endende Verben auch erkennt, wenn der Spieler die Form ohne e eingibt. Voraussetzung dafür ist, daß in der Schablone die Form mit e angegeben wird. Die Schablone

```
verb("laufe|renne|gehe weg",A_LAUFWEG,0)
```

erkennt also die Eingaben "laufe weg", "lauf weg", "renne weg" usw.

Achtung: Weil die Spielereingabe vor ihrer Auswertung in Kleinbuchstaben umgewandelt wird, dürfen auch in den Satzschablonen nur Kleinbuchstaben vorkommen.

Platzhalter stehen für einzelne Wörter oder Klassen in der Eingabe. Sie bestimmen, was der Spieler in der Eingabe erwähnen kann und beginnen immer mit dem Zeichen #.

Floyd kennt elf verschiedene Platzhalter:

#quote Ein oder mehrere Wörter in doppelten Anführungszeichen. Mit *#quote* wird in *stdconst.floyd* wörtliche Rede erkannt. Der Text zwischen den Anführungszeichen kann mit der Systemvariablen *quote* ermittelt werden.

#topic Akzeptiert alle Wörter bis zu dem ersten Wort nach dem Platzhalter. Die Schablone

```
verb("tippe #topic",A_TIPPE,0)
```

erkennt Eingaben mit beliebigen Wörtern nach "tippe". Die Schablone

```
verb("sage #topic zu dem zauberer",A_REDE,0)
```

akzeptiert beliebige Wörter zwischen "sage" und "zu". Alle von *#topic* erkannten Wörter finden sich in der Systemvariablen *topic*. Ihr Inhalt kann mit den Stringfunktionen von Floyd (s. Kapitel 2.6) weiterverarbeitet werden.

Wenn ein oder mehrere beliebige Wörter erkannt werden sollen, können also sowohl *#quote* als auch *#topic* verwendet werden. Beide Platzhalter eignen sich z.B. für die Angabe von Themen, über die sich der Spieler in einem Buch oder Computer informieren kann. Für Eingaben mit wörtlicher Rede wird in der Standardbibliothek aber *#quote* statt *#topic* verwendet, weil ohne Anführungszeichen nicht zwischen normalen Anweisungen und wörtlicher Rede unterschieden werden kann.

Der Platzhalter *#topic* eignet sich auch zum Einlesen unterschiedlicher Telefonnummern, Kennzeichen, Mailadressen und anderer Codes, die immer einer gleichen Syntax folgen. Im *Nebelmond* muß der Spieler verschiedene Koordinatentripel eingeben, die immer aus drei Zahlen beliebiger Länge und zwei Leerzeichen bestehen. Die entsprechende *verb*-Anweisung ist einfach:

```
define A_TYPE 100
verb("tippe #topic",A_TYPE,0);
```

Den eingegebenen Text liefert die Systemvariable *topic*. Mit den in Kapitel 2.7 vorgestellten Stringfunktionen kann jetzt geprüft werden, ob ein Koordinatentripel eingegeben wurde:

```
int istTripel() {
   int i,j,ok=1;
   string s1,s2;
   for (i=0;i<strlen(topic);i++) {
      s1=substr(topic,i,1);
      if (strstr(s1,"0123456789 ")<0) {
         ok=0;
      }
```

```
      if (s1==" " && s2!=" ") {
         j++;
      }
      s2=s1;
   }
   if (ok && j==2) {
        write("Das neue Sprungziel ist "+topic+".^");
   }
   else {
      write("Das ist kein Koordinatentripel.^");
   }
   return(ok);
}
```

#number Akzeptiert ganze Zahlen aus dem Integerbereich und die Numerale zwei bis zehn. Der Zahlenwert wird in der Systemvariablen *number* gespeichert.

#noun Akzeptiert eine oder mehrere Klassen, die für den Spieler erreichbar sind. Die Schablone

```
verb("nimm|nehme #noun",A_TAKE,0)
```

erkennt z.B. "nimm die Axt" oder "nehme die Karte und den Schlüssel", wenn die entsprechenden Klassen für den Spieler erreichbar sind. Erreichbar ist eine Klasse, wenn sie sich mit dem Spieler in einem hellen Raum befindet und nicht in einem geschlossenen Behälter verborgen ist. Dinge im Inventar sind auch bei Dunkelheit erreichbar. Ob eine Klasse x für einen Akteur erreichbar ist, überprüft die Methode

```
int scope(object x)
```

in *stdcreature.floyd* bei jedem Vergleich eines Platzhalters mit einer Klasse. Sie liefert entweder 1 (erreichbar) oder 0 (nicht erreichbar).

#oos (out of scope) Dies ist der weitreichendste Platzhalter. Er akzeptiert jede Klasse, die mit einem Namen referenziert werden kann, unabhängig davon, ob sie für den Akteur erreichbar ist. Das ist z. B. praktisch für Gespräche über Objekte, die der Spieler kennt, aber gerade nicht erreichen kann. Auch NPCs lassen sich hiermit aus anderen Räumen herbeirufen.

#single Akzeptiert nur eine einzelne Klasse, z.B. "nimm die Axt", nicht aber "nimm die Axt und das Holz".

#multi Akzeptiert nur mehrere Klassen, z.B. "lege die Axt, das Schwert und den Rucksack ab".

#held Akzeptiert nur Klassen im Inventar.

#reachable Akzeptiert nur Klassen, die sich nicht im Inventar befinden.

#inside Erfordert noch einen zweiten Platzhalter für Klassen und akzeptiert eine Klasse, wenn sie sich in der vom anderen Platzhalter erkannten Klasse befindet. Z.B. kann die Schablone

```
verb("nimm #inside aus #single",A_RAUS,0)
```

die Eingabe "nimm die Münze aus dem Beutel" erkennen.

#function Diesem Platzhalter müssen ein Gleichheitszeichen und der Name einer globalen Funktion folgen, der Floyd eine in der Eingabe erwähnte Klasse übergibt. Die Funktion muß 0 oder 1

zurückgeben und entscheidet damit, ob die Klasse akzeptiert wird. Im folgenden Beispiel lassen sich alle Klassen tragen, die ein Gewicht kleiner zehn haben:

```
verb("trage #function=test()",A_TRAGE,0)

int test(object x) {
   return(x.gewicht<10);
}
```

Achtung: alle Platzhalter für Klassen erkennen auch automatisch einen dem Klassennamen vorgestellten Artikel. So erkennt "nimm #noun" sowohl "nimm Axt" als auch "nimm die Axt". In Satzschablonen sollten daher keine Artikel auftauchen.

In einer Satzschablone können maximal zwei Platzhalter vorkommen. Wenn eine Schablone mit der Eingabe übereinstimmt, legt der Parser zwei interne Listen namens *first* und *second* mit den zu den Platzhaltern passenden Klassen an. Enthält eine Schablone mehr als zwei Platzhalter, werden die Klassen aller weiteren Platzhalter in *second* gespeichert. Mit den beiden Methoden

```
int isfirst()
int issecond()
```

läßt sich feststellen, ob eine Klasse in einer der beiden Listen auftaucht. Für eine in der entsprechenden Liste auftauchende Klasse liefern sie den Wert 1. Kommt die Klasse nicht in der Eingabe vor, wird 0 zurückgegeben.

```
// Schleife ueber alle zum ersten Platzhalter passenden Klassen
fetch (x, x.isfirst(), 0) {
}
```

Die folgende Tabelle zeigt einige typische Satzschablonen und die entsprechenden Werte für *actor* und die Listen *first* und *second*.

Schablone	Eingabe	actor	first	second
nimm #noun	nimm die Axt	player	Axt	-
nimm #noun	nimm Axt und Münze	player	Axt, Münze	-
lege #noun in #single	lege die Axt in die Kiste	player	Axt	Kiste
lese #noun	Frank, lese die Karte	Frank	Karte	-
#quote	Frank, "Hallo"	player	-	-

Übrigens ist es gleichgültig, in welchem Kasus ein Klassenname in der Eingabe auftaucht. Der Parser vergleicht jede Deklination mit der Eingabe, so daß auch grammatikalisch falsche Sätze wie "fütter dem Hund" erkannt werden. Da viele Namen in mehreren Fällen die gleiche Form haben, läßt sich leider nicht eindeutig feststellen, in welchem Fall ein Name eingegeben wurde.

Es ist praktisch kaum notwendig, auf die einzelnen Wörter der Eingabe zurückzugreifen. Sollte das doch einmal notwendig sein, kann dafür die Funktion

```
string getWord()
```

verwendet werden. Sie liefert nacheinander alle vom Spieler eingegebenen Wörter und Kommata.

```
string s;
firstWord();
do {
   s=getWord();
   write(s+" ");
} while (s!="");
```

Die Anweisung *firstWord()* setzt den internen Wortzähler zurück und läßt *getWord()* wieder das erste Wort liefern.

In seltenen Fällen muß festgestellt werden, ob der Spieler eine bestimmte Klasse bereits erwähnt hat, z.B. für eine Liste bereits manipulierter Objekte. Diese Frage beantwortet die Methode

```
int used()
```

Sie liefert 1 (bereits erwähnt) oder 0 (noch nicht erwähnt) zurück. Natürlich muß eine Klasse, die noch nicht erwähnt wurde, dem Spieler nicht unbekannt sein.

Fehlermeldungen

Wenn die Eingabe zu keiner Schablone paßt, ruft der Parser die Funktion

```
void parserError(int errno, string lastWord, object lastClass)
```

auf, die sich in der Datei *stderror.floyd* findet. *errno* zeigt an, wie weit der Parser mit der Eingabe gekommen ist und bestimmt die Fehlermeldung. *lastWord* und *lastClass* enthalten das Wort oder die Klasse, ab der die Eingabe mit keiner Schablone mehr übereinstimmt. Wenn die Meldungen umformuliert werden, sollte ihr Sinn dabei natürlich erhalten bleiben.

Weil der Parser immer nur zu der ganzen Eingabe eine passende Schablone sucht, kann es vorkommen, daß eine Schablone wegen eines einzigen unpassenden Wortes nicht erkannt wird, dieses Wort aber in ganz anderen Schablonen durchaus bekannt ist. Daher bedeutet eine Meldung wie "Ich kenne das Wort x nicht" nicht, daß das Wort x in allen Schablonen unbekannt ist.

In *stderror* findet sich auch die Anweisung

```
setNoun(object x)
```

Sie bestimmt die Klasse, auf die sich der Spieler mit einem Pronomen beziehen kann. Normalerweise ist es immer die Klasse, deren Name in der letzten Eingabe mit nur einer Klasse erwähnt wurde. So kann der Spieler sich also immer auf die zuletzt erkannte Klasse beziehen.

4.4 Aktionen

Wenn Floyd eine zu der Eingabe passende Satzschablone gefunden hat, wird in den in *first* und *second* vorkommenden Klassen die Methode

```
int onAction(int action)
```

mit der entsprechenden Aktionsnummer aufgerufen. Jede von einer Klasse der Standardbibliothek abgeleitete Klasse verfügt über diese Methode. In ihr wird die ausgelöste Aktion behandelt. Mit dem Rückgabewert teilt die Methode dem Parser mit, ob sie die Aktion abgearbeitet hat (1) oder nicht (0). Wird eine Aktion von keiner der in der Eingabe erwähnten Klassen behandelt, oder kommen darin gar keine Klassen vor, ruft Floyd die Methode *onAction()* in der Spielerklasse auf. Liefert auch sie 0 zurück, gibt der Parser die Meldung „Das geht leider nicht“ aus.

Da in *onAction()* meistens auf mehrere Aktionen reagiert werden muß, wird in der Standardbibliothek die passende Behandlungsroutine immer mit der *switch*-Anweisung ausgewählt. Die Methode *onAction()* ist also (fast) immer wie folgt aufgebaut:

```
int onAction(int action) {
   switch(action) {
      case(A_xxx);
         // Code für Aktion A_xxx ...
         return(1);

      case(A_yyy);
         // Code für Aktion A_yyy ...
         return(1);

      default;
         // wenn die Aktion nicht von dieser Klasse behandelt wird
         return(0);
   }
}
```

Meistens wird die Methode *onAction()* einer Standardklasse nur um neue Aktionen erweitert. Im *default*-Abschnitt muß dann eine nicht behandelte Aktion an die Superklasse weitergegeben werden:

```
default;
   // wenn die Aktion von der Superklasse behandelt wird
   return(super.onAction(action));
```

Achtung: In den Dateien der Standardbibliothek werden bei der Verarbeitung von Aktionen in manchen Fällen auch weitere Aktionen ausgelöst. Z. B. wird bei der Reaktion auf A_EXAMINE und A_LOOKON in *stditem.floyd* auch A_LOOKINTO ausgelöst, wenn das entsprechende Objekt ein offener oder transparenter Behälter ist. Bei solchen Aktionen muß darauf geachtet werden, daß keine Endlosschleifen entstehen. Endlosschleifen können auftreten, wenn ein Objekt die Aktionen A und B behandelt, A auch B auslösen kann und in der Standardbibliothek bei der Verarbeitung von B wieder A ausgelöst wird. Im *Nebelmond* wird z. B. in der Gleiterklasse eine Endlosschleife mit der Dummy-Variablen *examine* verhindert, die festhält, ob A_LOOKINTO bereits ein A_EXAMINE vorausging.

before und beforeAction

Wenn ein Spiel bereits vor dem Aufruf von *onAction()* auf eine Eingabe reagieren soll, können die Funktionen

```
string before(string input)
```

und

```
void beforeAction(int action)
```

verwendet werden. Wenn es eine globale Funktion *before()* gibt, wird die Spielereingabe vor ihrer Auswertung an diese Funktion als Argument *input* übergeben. Der Parser arbeitet anschließend mit dem von *before()* zurückgegebenen String weiter. Mit *before()* ist es also möglich, die Spielereingabe zu verändern. Sinnvoll kann das z.B. in Fällen sein, in denen der Spieler unter einem Fluch steht, der ihn bestimmte Kommandos oder Richtungsangaben vertauschen läßt. Zu beachten ist dabei, daß die Eingabe immer schon in Kleinbuchstaben übergeben wird und *before()* für jeden Satz in der Eingabe einmal aufgerufen wird, wenn in der Eingabe mehrere durch Punkte getrennte Sätze oder Kommandos vorkommen.

Wenn in einer Klasse die Methode

```
void beforeAction(int Aktionsnummer)
```

definiert ist, wird sie nach der Auswertung der Eingabe und vor dem Aufruf von *onAction()* mit einer Aktionsnummer aufgerufen. Der Aufruf von *beforeAction()* erfolgt auch, wenn die entsprechende Klasse nicht in der Eingabe vorkommt. So können Klassen auch reagieren, wenn der Spieler mit ihnen eine bestimmte Aktion *nicht* ausführt. In *Download* wird z.B. der Telefonhörer vor irgendeiner anderen Ausgabe automatisch wieder aufgelegt, wenn der Spieler keine Telefonnummer wählt.

4.5 Attribute der Standardklassen

Die Klassen der Standardbibliothek verwenden eine Reihe von Attributen, um bestimmte Eigenschaften und Zustände von Klassen (z.B. eingeschaltet, offen) zu markieren. Diese Attribute werden bei der Verarbeitung von Aktionen gesetzt oder gelöscht. Die Standardbibliothek kennt die folgenden Attribute. In Klammern ist angegeben, in welchen Klassen das Attribut berücksichtigt wird. Da von *stdobject* alle anderen Klassen der Standardbibliothek abgeleitet sind, werden ihre Attribute auch in den anderen Klassen berücksichtigt.

closeable (stdobject) Die Klasse kann geschlossen werden.

container (stditem) Der Spieler kann in die Klasse andere Objekte hineinlegen.

detail (stditem) Verhindert die Beschreibung der Klasse bei *A_EXAMINEALL* ("untersuche alles"). Die Klasse kann z.B. eine Schraube an einer großen Maschine sein. Wenn die Maschine untersucht wird, wird auch die Schraube ausführlich erwähnt, so daß sie nicht zweimal beschrieben werden muß.

enterable (stditem) Der Spieler kann die Klasse mit "rein" und "raus" betreten und verlassen.

exit (stdexit) Markiert alle Verbindungen zwischen Räumen (Türen, Durchgänge).

hidden (stditem, stdexit) Die Klasse wird in der automatischen Raumbeschreibung nicht erwähnt. Die automatische Raumbeschreibung wird von der Methode *description()* in *stdroom* ausgegeben und listet alle in einem Raum befindlichen Objekte auf. Wenn die Klasse aber schon in der mit *setLong()* bestimmten Beschreibung auftaucht oder aus anderen Gründen nicht erwähnt werden soll, kann sie mit *hidden* versteckt werden.

*light (*std*room, stditem)* Die Klasse stellt eine Lichtquelle dar. Alle Räume werden automatisch mit *light* deklariert, so daß der Spieler in ihnen sehen kann. Wenn der aktuelle Raum keine Lichtquelle ist und sich auch keine in ihm befindet, steht der Spieler in der Dunkelheit und kann Objekte außerhalb seines Inventars normalerweise nicht erreichen.

linefeed (stdroom) Dient ab Version 3.0 der Ausgabe einer zusätzlichen Leerzeile zwischen der Raumbeschreibung und der Liste der im Raum befindlichen Objekte ohne *hidden*. Dieses Attribut ist sinnvoll, wenn auch die Absätze der Raumbeschreibung mit Leerzeilen getrennt sind.

living (stdcreature) Die Klasse stellt ein Lebewesen dar. Ob es lebt oder tot ist, muß im Spiel mit einem weiteren Attribut markiert werden.

lockable (stdobject) Die Klasse kann verschlossen werden

locked (stdobject) Die Klasse ist verschlossen

moveable (stditem) Die Klasse stellt ein Fortbewegungsmittel dar, das sich mit dem Spieler bewegt, wenn er sich darin befindet.

on (stditem) Die Klasse ist eingeschaltet.

open (stdobject) Die Klasse ist geöffnet.

openable (stdobject) Die Klasse läßt sich öffnen

scenery (stdroom) Alle Räume werden hiermit markiert.

supporter (stditem) Die Klasse dient als Ablage. Andere Objekte können draufgelegt und runtergenommen werden.

switchable (stditem) Die Klasse kann ein- und ausgeschaltet werden.

takeable (stditem) Der Spieler kann die Klasse aufnehmen, im Inventar tragen und sie wieder ablegen.

transparent (stditem) Objekte in einem transparenten Behälter sind für den Spieler sichtbar.

underground (stditem) Die Klasse hat einen Untergrund, so daß der Spieler unter sie schauen kann.

unique (stditem) Die Klasse wird in Listen (*stdlist*) mit einem bestimmten Artikel ausgegeben.

visited (stdroom) Markiert alle bereits besuchten Räume.

wearable (stditem) Die Klasse kann vom Spieler an- und ausgezogen werden, dient also als Kleidungsstück.

worn (stditem) Markiert Kleidungsstücke im Inventar, die der Spieler angezogen hat. Das An- und Ausziehen erledigen die Aktionen *A_WEAR* und *A_DISROBE*. Bei der Auflistung des Inventars erscheint nach dem Kleidungsstück die Meldung "angezogen". Dieser Text steht in der Variablen *robe* in *stditem* und muß manchmal angepaßt werden (z.B. sollte bei einem Helm besser "aufgesetzt" erscheinen).

Eine Berührung

Ein einfaches Beispiel ist die Aktion *A_TOUCH*, die in *stdconst.floyd* für die Berührung beliebiger Objekte deklariert ist:

```
#define A_TOUCH 54

verb("berühre #single", A_TOUCH, 0);
verb("fasse #single an", A_TOUCH, 0);
```

Damit sich in einem Spiel auch alle möglichen Dinge berühren lassen, wird die Aktion in der Klasse *stdobject* behandelt, die Superklasse aller anderen Klassen ist.

```
case(A_TOUCH);
   s=(has(living)) ? "<Der this> scheint davon nicht sehr angetan.^":
      "Nichts passiert.^";
   write(s);
   return(1);
```

Wenn wir in ein Spiel ein Feuer einbauen, sollte es bei einer Berührung natürlich eine andere Meldung ausgeben. Dafür wird die Feuerklasse einfach von *stdobject* abgeleitet und *onAction()* überladen:

```
class feuer:stdobject {
   setShort("Feuer");

   int onAction(int action) {
      switch(action) {
         case(A_TOUCH);
            write("Ahhrg, du verbrennst dir die Finger.^");
            return(1);

         default;
            // Alle anderen Aktionen werden von stdobject behandelt
            return(super.onAction(action));
      }
   }
}
```

4.6 Aktionen mit mehreren Klassen

Wenn in einer passenden Satzschablone zwei Platzhalter für Klassen vorkommen, ist es oft nicht notwendig, die Aktion von beiden Klassen ausführen zu lassen. So erkennt z.B. die Schablone

```
verb("lege #held in #single",A_DROPINTO,0);
```

die Eingabe "Lege den Dolch und die Münze in die Truhe" und löst die Aktion *A_DROPINTO* in den Klassen Dolch (*first*), Münze (*first*) und Truhe (*second*) aus. Der Dolch und die Münze müssen aber nur einmal in die Truhe befördert werden. Wenn einer der beiden Platzhalter nur für eine einzelne Klasse steht, was meistens der Fall ist, braucht die Aktion nur von der zu #single gehörenden Klasse ausgeführt zu werden. *A_DROPINTO* wird z.B. in *stditem* nur für die Klasse in *second* ausgeführt:

```
int onAction(int action) {
   object x1,x2;
   x1=this;
   switch(action) {
      case(A_DROPINTO);
         // lege alle x2 in x1
         if (x1.issecond() && x1.has(container)) {
            if (x1.has(open) || actor.in(x1)) {
               fetch(x2,x2.isfirst(),0) {
                  if (x2.has(takeable) && x2!=x1) {
                     if (x2.in(actor) || x2==actor) {
                        write("<Der actor> <legst legt>
                           <den x2> in <den x1>.^");
                        if (x2.has(worn)) {
                           x2.with(~worn);
                        }
                        x2.moveto(x1);
                     }
                     else {
                        write("<Der actor> <mußt muß> <den x2> dafür erst aufnehmen.^");
                     }
                  }
                  else {
                     write("<Der actor> <kannst kann> <den x2> nicht
                        in <den x1> legen.^");
                     if (x1==x2) {
                        halt(1);
                     }
                  }
               }
               if (count==0) {
```

```
            write("Ich kann hier nichts in <den x1> legen.^");
         }
      }
      else {
         write("<Der x1 muß müssen> erst geöffnet werden.^");
         halt(2);
      }
      return(1);
   }
   break;
```

Die *halt*-Anweisung unterdrückt den Aufruf von *onAction()* in weiteren Klassen. *halt(1)* verhindert die Aktion in allen weiteren Klassen der Listen *first* und *second*. *halt(2)* verhindert die Aktion nur in allen weiteren Klassen von *second*. So verhindert *halt(1)* in diesem Beispiel die doppelte Ausgabe von "Du kannst x nicht in x legen" nach dem unsinnigen Versuch, den Behälter in sich selbst zu legen.

Wenn beide Platzhalter mehrere Klassen aufnehmen, können nach dem ersten Aufruf von *onAction()* alle weiteren mit einem *halt(1)* verhindert werden. Weil die *halt*-Anweisung aber die Listen *first* und *second* löscht, kann auch, wenn die Listen erhalten bleiben sollen, mit einer Dummy-Variablen leicht festgestellt werden, ob die Aktion bereits ausgeführt wurde:

```
int first=1;

int onAction(int action) {
   switch(action) {
      case(A_irgendwas);
         if (first) {
            // Code fuer diese Aktion
            first=0;
         }
         return(1);
   }
}
```

Eine Taschenlampe

Die Aktion *A_DROPINTO* läßt sich auch für eine Taschenlampe verwenden, in die eine Batterie eingelegt werden muß, bevor sie sich einschalten läßt. Wenn der Spieler die Batterie aus der Lampe nimmt, sollte ihr Licht wieder ausgehen:

```
class batterie:stditem {
   setShort("-Batterie");
   with(takeable);
   int onAction(int action) {
      switch(action) {
         case(A_TAKE);
         case(A_TAKEOUT);
            lampe.with(~light);
            lampe.with(~on);
         default;
            return(super.onAction(action));
      }
   }
}
```

Für das Ein- und Ausschalten von Objekten sind in *stdconst* bereits folgende Aktionen und Verben definiert:

```
#define A_SWITCHON  30
#define A_SWITCHOFF 31
verb("schalte|mache #noun ein|an",A_SWITCHON,0);
verb("schalte|mache #noun aus|ab",A_SWITCHOFF,0);
```

A_SWITCHON und *A_SWITCHOFF* werden in stditem behandelt. Die Taschenlampe wird also von *stditem* abgeleitet und berücksichtigt in *onAction()* beim Einschalten noch die Batterie und das Licht:

```
class lampe:stditem {
   setShort("-Taschenlampe,-Lampe");
   with(takeable,switchable,openable,closeable,container);

   int onAction(int action) {
      switch(action) {
         case(A_SWITCHON);
            if (batterie.in(lampe)) {
               with(light);
               return(super.onAction(action));
            }
            else {
               write("Nichts passiert^");
               return(1);
            }
            break;

         case(A_SWITCHOFF);
            with(~light);

         default;
            return(super.onAction(action));
      }
   }
}
```

Wir können jetzt das Licht im Raum abschalten und dem Spieler die Taschenlampe und die Batterie mitgeben.

```
void main() {
   raum.with(~light);
   setPlayer(spieler);
   spieler.moveto(raum);
   lampe.moveto(spieler);
   batterie.moveto(spieler);
   raum.description();
}
```

Jetzt kann der Spieler das Licht einschalten:

Du kannst deine eigene Hand nicht vor den Augen sehen, so finster ist es hier.

>öffne die lampe
Du öffnest die Taschenlampe.

>lege die batterie in sie
Du legst die Batterie in die Taschenlampe.

>schalte die taschenlampe ein
Die Taschenlampe ist jetzt eingeschaltet.

5 Die Spielwelt

In diesem Kapitel werden wir die Bestandteile der Spielwelt und die zugrundeliegenden Standardklassen genauer kennenlernen. Dabei soll ein kleines (winziges) Spiel mit dem Namen *Das Gewand der Finsternis* geschrieben werden. Es handelt sich dabei um eine Umsetzung des englischen Originals *Cloak of Darkness* von Roger Firth, von dem es bereits Versionen für verschiedene Autorensysteme gibt. Sinn des Spiels ist es eigentlich, den Vergleich von Autorensystemen zu erleichtern. Die englischen Versionen finden sich unter

http://www.firthworks.com/roger/cloak/index.html

Das Spiel besteht nur aus drei Räumen und drei Objekten:

- Das Spiel beginnt im leeren Foyer eines alten Opernhauses. Zwei Türen führen nach Westen und Süden. Eine dritte Tür im Norden kann nicht passiert werden.
- Die Bar liegt südlich vom Foyer und ist anfangs dunkel. Kehrt der Spieler hier nicht sofort nach Norden zurück, erscheint eine Warnung, wonach er in der Dunkelheit etwas durcheinanderbringen wird.
- Die Garderobe befindet sich im Westen des Foyers. Hier ist ein Kleiderhaken an der Wand befestigt.
- Der Spieler trägt einen schwarzen, lichtabsorbierenden Mantel, den er an den Haken in der Garderobe hängen kann.
- Kehrt der Spieler ohne Mantel in die Bar zurück, ist sie beleuchtet. Auf dem Boden findet sich eine Nachricht in Sägemehl gekritzelt.
- Der Spieler kann die Nachricht lesen und damit das Spiel beenden. Die Nachricht lautet, je nachdem wie oft der Spieler bereits darübergelaufen ist: "Du hast gewonnen" oder "Du hast verloren".

5.1 Räume

Alle Räume der Spielwelt werden von der Standardklasse *stdroom* abgeleitet. Die Räume im Gewand brauchen zunächst nur einen Namen und eine statische Beschreibung. In der Bar schalten wir dann noch das Licht aus:

```
class garderobe:stdroom {
   setShort("-Garderobe");
   setLong("An den Wänden diese kleinen Raumes befanden sich offenbar einmal Reihen
      von Kleiderhaken aus Messing, von denen jetzt nur noch einer geblieben ist.
      Der Ausgang ist eine Tür im Osten.^");
}

class bar:stdroom {
   setShort("-Bar");
   setLong("Die Bar ist nicht halb so prächtig, wie du annahmst, nachdem du das Foyer
      im Norden gesehen hast. Sie ist völlig leer, bis auf das Sägemehl am Boden,
      in dem du eine hingekritzelte Nachricht erkennen kannst.^");
   with(~light);
}
```

Im Norden des Foyers befindet sich der Eingang der Oper, durch den der Spieler aber nicht wieder hinausgelangen soll. Immer wenn der Spieler eine Richtungsangabe eingibt, unternimmt die Standardbibliothek folgende Schritte:

1. In der Klasse des Akteurs (normalerweise die Spielerklasse) wird die der angegebenen Richtung entsprechende Aktion ausgelöst. Die von Richtungsangaben ausgelösten Aktionsnummern sind in *stdconst.floyd* als Konstanten mit dem Präfix D_ (für Direction) deklariert und reichen von null bis neun (acht Himmelsrichtungen plus oben und unten). Der Ausdruck *action<10* stellt also immer fest, ob sich der Akteur bewegen möchte. Neben den Richtungsangaben ist in *stdconst.floyd* auch noch die Aktion A_GOTO (11) definiert, die den Spieler zu einem benachbarten und bereits bekannten Raum bewegt. Wenn der angegebene Raum gefunden wird, wird in *stdcreature* eine Bewegung in die entsprechende Himmelsrichtung ausgeführt.

2. Die *onAction*-Methode von *stdcreature* merkt sich die aktuelle Position des Akteurs in der Objektvariablen *from*. Mit *actor.from* läßt sich also immer feststellen, von wo der Akteur einen Raum betreten hat.

3. In der aktuellen Raumklasse wird anschließend die Methode

```
void goto(int dir, object x)
```

mit der entsprechenden Aktionsnummer (*dir*, z.B. *D_NORTH*) und der Klasse des Akteurs (*x*) aufgerufen. Wenn es in der gewünschten Richtung eine (nicht verschlossene) Verbindung zu einem anderen Raum gibt, wird der Akteur dorthin bewegt, andernfalls wird die Meldung "Hier geht's nicht weiter" oder "X muß erst geöffnet werden" ausgegeben.

Um die Bewegung des Spielers zu blockieren, müssen wir also im Foyer *goto()* überladen:

```
class foyer:stdroom {
   void init() {
      setShort("Foyer der Oper");
      setLong("Du stehst in einer großen, prächtig ausgestatteten Halle
         in rot und gold. Funkelnde Kronleuchter hängen von der Decke herab.
         Der Eingang liegt im Norden, weitere Türen befinden sich im Süden
         und Westen.^");
   }
   void goto(int dir, object x) {
      if (dir!=D_NORTH) {
         super.goto(dir,x);
      }
      else {
         write("Du bist doch gerade erst angekommen, und draußen scheint das
            Wetter ohnehin nur noch schlechter zu werden.^");
      }
   }
}
```

Die Aktion *A_GOTO* läßt den Spieler mit der Eingabe „Gehe zu Raumname“ sein Ziel ohne eine Richtungsangabe bestimmen. Der angegebene Raum wird dabei von der Methode

```
int searchRoom(object goal)
```

in *stdroom.floyd* gesucht. Die Methode gibt die Nummer der entsprechenden Himmelsrichtung zurück. Ob der Spieler nur bereits besuchte Räume, alle bekannten Räume oder nur den aktuellen Raum erwähnen kann, bestimmt Zeile 42 in *stdcreature.floyd* :

```
ok=x.has(visited);
```

Wird die Zeile in *ok=1* geändert, können alle bekannten Räume erwähnt werden. Mit *ok=0* akzeptiert der Parser nur noch den Namen des aktuellen Raums.

Raumbeschreibungen

Die mit *setLong()* bestimmte Raumbeschreibung wird von der Methode

```
void description()
```

ausgegeben, wenn die Raumklasse das Attribut *light* besitzt. Wenn der Raum dunkel ist, wird der Text der Variablen *darkness* ausgegeben. Die Standardmeldung "Du kannst deine eigene Hand nicht vor den Augen sehen, so finster ist es hier" kann also mit

```
darkness="meine dunkle Beschreibung";
```

verändert werden. Ob sich im aktuellen Raum eine Lichtquelle befindet, wird in *stdroom* mit der Methode

```
int isLight()
```

ermittelt. Sie liefert entweder 1 (hell) oder 0 (dunkel). Nach dem mit *setLong()* bestimmten Text wird noch eine Liste von Objekten, die sich im Raum befinden und nicht versteckt sind (~ *hidden*), ausgegeben ("Weiterhin siehst du hier ...").

Wenn der Raum bereits besucht wurde, er also das Attribut *visited* hat, wird nur der Raumname ausgegeben und die Objektliste beginnt mit "Du siehst hier ...". Dieses Verhalten kann mit dem Kommando "ausführlich" geändert werden, das die in *stdconst.floyd* definierte Variable

```
int isverbose;
```

auf 1 setzt und damit immer für ausführliche Beschreibungen sorgt.

Im Gewand sind alle Raumbeschreibungen statisch. Manchmal muß die Beschreibung aber geändert werden, wenn sich im Raum etwas verändert hat. Eine neue Beschreibung kann wieder mit *setLong()* bestimmt werden. Am einfachsten geht das, wenn die Methode *description()* überladen wird, um die passende Beschreibung zu bestimmen. Als Beispiel ändern wir bei jeder Beschreibung die Farben des Foyers:

```
void description() {
   string s;
   switch (random(3)) {
      case(0);
         s="rot und gold";
         break;

      case(1);
         s="schwarz und blau";
         break;

      case(2);
         s="grün und purpurrot";
   }
   setLong("Du stehst in einer großen, prächtig ausgestatteten Halle
      in "+s+". Funkelnde Kronleuchter hängen von der Decke herab.
      Der Eingang liegt im Norden, weitere Türen befinden sich im
      Süden und Westen.^");
   super.description();
}
```

5.2 Verbindungen, Türen und Schlüssel

Die Klasse *stdexit* verbindet zwei Räume, so daß der Spieler vom einen in den anderen gelangen kann. Wenn die Verbindung einfach nur einen offenen Durchgang darstellen soll, braucht sie nur ohne weitere Angaben von *stdexit* abgeleitet zu werden.

```
class foyer2garderobe:stdexit {
   with(hidden);
}

class foyer2bar:stdexit {
   with(hidden);
}
```

Um ihre Auflistung nach der Beschreibung zu verhindern, kann der Name weggelassen oder das Attribut *hidden* gesetzt werden. Verbunden werden zwei Räume mit der Methode

```
void addExit(int dir, object nextroom, object exit)
```

in *stdroom*. *dir* ist eine der Richtungskonstanten aus *stdconst.floyd* und bestimmt die Lage des zweiten zu verbindenden Raums, der mit *nextroom* angegeben wird, *exit* gibt die zwischen den Räumen liegende Verbindung an. Um im Gewand das Foyer mit den beiden anderen Räumen zu verbinden, erweitern wir die *init*-Routine in *foyer* wie folgt:

```
void init() {
   setShort("Foyer der Oper");
   setLong("Du stehst in einer großen, prächtig ausgestatteten Halle
      in rot und gold. Funkelnde Kronleuchter hängen von der Decke
      herab. Der Eingang liegt im Norden, weitere Türen befinden
      sich im Süden und Westen.^");
   addExit(D_WEST, garderobe, foyer2garderobe);
   addExit(D_SOUTH, bar, foyer2bar);
}
```

Die Methode *addExit()* trägt die angegebene Verbindung in das Objektarray *exits[]* von *stdroom* ein. Dieses Array hat für jede Richtung (null bis neun) einen Eintrag (eine Verbindung oder NULL), so daß sich leicht feststellen läßt, ob in einer bestimmten Richtung ein anderer Raum liegt. Verbindungen merken sich mit den Variablen *from* und *to*, welche beiden Räume sie verbinden. Dabei haben die Namen *from* und *to* keine Bedeutung, der Spieler kann auch von *to* nach *from* gehen.

Türen

Manchmal soll der Spieler erst eine Tür oder ein ähnliches Hindernis öffnen, um in einen anderen Raum zu gelangen. Die Verbindung muß also zu öffnen und zu schließen sein. Hierfür muß der Verbindung zuerst ein Name gegeben werden, damit der Spieler sie erwähnen kann. Ob der Spieler die Verbindung öffnen und schließen kann, bestimmen die Attribute *openable* und *closeable*. Der Zustand der Verbindung wird mit dem Attribut *open* bestimmt, das eine offene Verbindung markiert. Das Attribut *open* wird bei der Deklaration von *stdexit* gesetzt und muß in einer abgeleiteten Klasse entfernt werden, wenn die Verbindung anfangs geschlossen sein soll:

```
class foyer2garderobe:stdexit {
   setShort("-Tür");
   with(hidden, openable, closeable, ~open);
}
```

Für Schlüssel (Chipkarten, Äxte etc.) mit denen sich auf- und abschließbare Objekte öffnen lassen, ist in *stdobject* die Objektvariable *key* deklariert. Ihr kann eine von *stditem* abgeleitete Klasse zugewiesen werden, mit der sich die Verbindung auf- und zuschließen läßt. Zusätzlich muß in der

Verbindung noch das Attribut *lockable* gesetzt sein. Schließlich bestimmt das Attribut *locked* noch, ob die Verbindung abgeschlossen ist:

```
class schluessel:stditem {
   setShort("+Schlüssel");
   with(takeable);
   moveto(spieler);
}

class foyer2garderobe:stdexit {
   setShort("-Tür");
   with(hidden,openable,closeable,~open,lockable,locked);
   void init() {
      key=schluessel;
   }
}
```

Für den Umgang mit Dingen, die sich öffnen, schließen, ab- und wieder aufschließen lassen sind in *stdconst.floyd* die Aktionen *A_OPEN*, *A_CLOSE*, *A_LOCK*, und *A_UNLOCK* deklariert. Weil sich nicht nur Türen schließen und öffnen lassen, werden diese Aktionen bereits in *stdobject* verarbeitet. Sie können von folgenden Sätzen ausgelöst werden:

```
verb("öffne #noun",A_OPEN,0);
verb("mache #noun auf",A_OPEN,0);

verb("schließe #noun",A_CLOSE,0);
verb("mache #noun zu",A_CLOSE,0);

verb("schließe|verschließe #noun mit #held",A_LOCK,0);
verb("schließe #noun mit #held ab",A_LOCK,0);
verb("schließe #noun ab|zu",A_LOCK,0);
verb("verschließe #noun",A_LOCK,0);

verb("öffne #noun mit #held",A_UNLOCK,0);
verb("schließe #noun mit #held auf",A_UNLOCK,0);
verb("schließe #noun auf",A_UNLOCK,0);
```

Die Tür zur Garderobe ermöglicht jetzt z.B. folgenden Dialog:

>i
Du trägst: einen Schlüssel

>x tür
Die Tür ist geschlossen.

>öffne die tür
Die Tür ist verschlossen.

>schliesse die tür mit dem schlüssel auf
Du schließt die Tür auf.

>öffne die tür
Du öffnest die Tür.

>verschliesse die tür
(mit dem Schlüssel)
Du verschließt die Tür.

>w
Die Tür muß erst geöffnet werden.

>schliesse die tür auf
(mit dem Schlüssel)
Du schließt die Tür auf.

>w
(Du öffnest erst die Tür)
Garderobe
An den Wänden diese kleinen Raumes befanden sich offenbar einmal Reihen von Kleiderhaken aus Messing, von denen jetzt nur noch einer geblieben ist. Der Ausgang ist eine Tür im Osten.

Wie aus dem Dialog hervorgeht, wird eine Tür beim Aufschließen nicht automatisch geöffnet. Um das zu erreichen, muß bei der Aktion *A_UNLOCK* in *stdobject* nur ein *with(open)* nach dem *with(~locked)* eingefügt werden.

5.3 Einfache Gegenstände

Die meisten in einem Spiel vorkommenden Dinge, die der Spieler manipulieren kann, werden von *stditem* abgeleitet. Auch die in den nächsten beiden Kapiteln vorgestellten Behälter und Fahrzeuge werden mit *stditem* erzeugt. Für das Gewand brauchen wir einen Mantel, einen Haken und eine Nachricht:

```
class mantel:stditem {
   setShort("+Mantel,+Samtmantel");
   setLong("Ein schöner Samtmantel, mit Satin durchzogen und von
      Regentropfen leicht benetzt. Er ist so tief schwarz, daß es
      fast scheint, als schlucke er das umgebende Licht.^");
   with(takeable,wearable,worn);
   moveto(spieler);
   int amHaken,first=1;
   int onAction(int action) {
      switch(action) {
         case(A_HANGUP);
            write("Du hängst den Mantel an den Haken.^");
            with(~worn,hidden);
            moveto(garderobe);
            amHaken=1;
            bar.with(light);
            if (first) {
               addScores(1);
               first=0;
            }
            return(1);
         case(A_DROP);
         case(A_DISROBE);
            if (room()==garderobe) {
               write("Warum willst du den Mantel auf den Boden
                  legen, wenn es hier doch einen Kleiderhaken gibt?^");
            }
            else {
               write("Dies ist nicht der beste Ort, um einen
                  schönen Mantel herumliegen zu lassen.^");
            }
            return(1);
            case(A_TAKE);
            case(A_WEAR);
         if (amHaken) {
            amHaken=0;
```

```
            with(~hidden);
            bar.with(~light);
         }
         default;
            return(super.onAction(action));
      }
   }
}
```

Die Aktion *A_HANGUP* wird ausgelöst, wenn der Spieler den Mantel an den Haken hängen möchte:

```
#define A_HANGUP 100;
verb("hänge #single an #single",A_HANGUP,0);
verb("hänge #single an|am #single auf",A_HANGUP,0);
```

Alle anderen Aktionen sind bereits in *stdconst.floyd* definiert. Wenn der Spieler den Mantel zum ersten Mal aufhängt, erhält er mit der Anweisung *addScores()* einen Punkt.

```
class haken:stditem {
   setShort("+Kleiderhaken,+Haken,+Messinghaken");
   with(hidden);
   moveto(garderobe);
   int onAction(int action) {
      string s;
      switch(action) {
         case(A_EXAMINE);
            s="Es ist nur ein einfacher Kleiderhaken aus
               Messing, ";
            s+=(mantel.amHaken) ? "an dem ein Mantel hängt.^" :
               "der an die Wand geschraubt ist.^";
            setLong(s);
         default;
               return(super.onAction(action));
         }
   }
}
```

Die Beschreibung des Hakens wird hier dynamisch angepaßt, wenn der Spieler den Haken untersucht.

```
class nachricht:stditem {
   setShort("-gekritzelt* Nachricht,Sägemehl");
   with(hidden);
   moveto(bar);
   int n;
   int onAction(int action) {
      string s;
      switch(action) {
         case(A_EXAMINE);
         case(A_READ);
            if (n<2) {
               addScores(1);
               write("Die Nachricht lautet ...^
                  *** Du hast gewonnen ***^");
               }
               else {
                     write("Die Nachricht wurde zertrampelt. Du
                        kannst gerade noch folgende Worte
                        entziffern:^*** Du hast verloren ***^");
               }
```

```
            quit;
         default;
            return(super.onAction(action));
      }
   }
}
```

Die Variable *n* hält fest, wie oft der Spieler im Dunkeln über die Nachricht gelaufen ist. Die *quit*-Anweisung beendet den Dialog mit dem Spieler und bietet ihm die Optionen, das Spiel neu zu beginnen oder das Spiel zu beenden.

Ein kleines Problem bleibt aber noch: In der dunklen Bar soll sich der Spieler nur nach Norden bewegen können. Unternimmt er etwas anderes, zertrampelt er die Nachricht am Boden. Also muß vor der Ausführung irgendeiner Aktion in *spieler* und *mantel* noch geprüft, werden, ob der Spieler sich in der Bar befindet und diese dunkel ist. Um den entsprechenden Code nicht doppelt schreiben zu müssen, setzen wir ihn in eine globale Funktion und machen von ihrem Rückgabewert die weitere Ausführung einer Aktion abhängig:

```
int vorher(int action) {
   if (player.location()!=bar || bar.has(light) || action==D_NORTH) {
      return(1);
   }
   else {
      if (action<11) {
         write("Es ist keine gute Idee, im Dunkeln
            herumzutappen.^");
         nachricht.n+=2;
      }
      else {
         write("Im Dunkeln? Du könntest hier leicht
            etwas durcheinander bringen.^");
         nachricht.n+=1;
      }
      return(0);
   }
}
```

Jetzt muß die Funktion *vorher()* noch in der *onAction*-Methode von *spieler* und *mantel* berücksichtigt werden:

```
class spieler:stdcreature {
   int onAction(int action) {
      if (vorher(action)) {
         return(super.onAction(action));
      }
      else {
         return(1);
      }
   }
}

class mantel:stditem {
   setShort("+Mantel,+Samtmantel");
   setLong("Ein schöner Samtmantel, mit Satin durchzogen und von
      Regentropfen leicht benetzt. Er ist so tief schwarz, daß es
      fast scheint, als schlucke er das umgebende Licht.^");
   with(takeable,wearable,worn);
   moveto(spieler);
   int amHaken,first=1;
   int onAction(int action) {
```

```
      if (vorher(action)) {
         // Code fuer einzelne Aktionen
      }
      else {
         return(1);
      }
   }
}
```

Die Funktion *vorher()* hat übrigens nichts mit der Funktionalität von *before()* zu tun, die ja bereits vor der Auswertung der Eingabe aufgerufen wird. Statt *vorher* in *onAction* aufzurufen, könnte ihr Rückgabewert aber auch schon in *beforeAction*-Methoden des Spielers und des Mantels einer Integervariablen zugewiesen werden, die dann in *onAction()* geprüft wird.

Wenn solche Überprüfungen vor oder auch nach der Ausführung von Aktionen in mehreren Klassen durchgeführt werden müssen, empfiehlt es sich, eine abstrakte Klasse mit einer entsprechend angepaßten *onAction*-Methode zu schreiben und alle weiteren Klassen auf sie aufzubauen. Die abstrakte Klasse kann dabei natürlich auch selbst von einer Standardklasse abgeleitet werden.

Wenn wir jetzt noch eine *main*-Funktion und einen kurzen Vorspann schreiben, kann das Gewand der Finsternis schon gespielt werden.

```
void main() {
   setPlayer(spieler);
   spieler.moveto(foyer);
   write("Das Gewand der Finsternis^Ein einfaches Beispiel für
      Interactive Fiction^Version "+serial()+" Geschrieben 2003
      von Oliver Berse^^Du eilst durch die verregnete Novembernacht
      und bist froh, die strahlenden Lichter der Oper zu sehen. Es
      ist überraschend, daß hier nicht mehr Menschen sind, aber was
      erwartest du von so einem einfachen Beispiel ... ?^^");
   foyer.description();
}
```

Die Funktion

```
string serial()
```

liefert das Datum der letzten Kompilierung im Format JJMMTT und kann als Versionsnummer dienen. Wenn das Programm noch nicht compiliert wurde, liefert sie die Zeichenfolge "000000".

Das Spiel hält sich stark an die englischen Versionen und kann an einigen Stellen natürlich noch verbessert werden. Versuchen Sie doch z.B. einmal zu verhindern, daß der Spieler den Mantel mehrmals hintereinander an den Haken hängen kann.

5.4 Behälter

In Behälter kann der Spieler Dinge hineinlegen und wieder herausnehmen, sie lassen sich auch ab- und wieder aufschließen. Für Behälter können wieder die schon von Türen bekannten Attribute *openable*, *closeable*, *open*, *lockable* und *locked* verwendet werden. Zusätzlich müssen sie über das Attribut *container* verfügen:

```
class kaestchen:stditem {
   setShort("Holzkästchen, Kästchen");
   with(takeable, container, openable, closeable);
}
```

Für den Umgang mit Behältern sind in *stdconst.floyd* folgende Aktionen und Verben definiert:

```
#define A_DROPINTO      16
#define A_DROPALLINTO   47
#define A_TAKEOUT       13
#define A_TAKEALLOUT    48

verb("lege #held in #single", A_DROPINTO, 0);
verb("lege alles in #single", A_DROPALLINTO, 0);
verb("nimm|nehme #noun aus #single", A_TAKEOUT, 0);
verb("nimm|nehme alles aus #single", A_TAKEALLOUT, 0);
```

Bei *A_TAKEOUT* läßt sich auch #inside statt #noun verwenden, die Überprüfung, ob sich die angegebene Klasse tatsächlich in einem Behälter befindet, kann dann in der *onAction*-Methode von *stditem* entfallen. Der Unterschied ist nur, daß sich die Fehlermeldung ändert, wenn eine Klasse sich nicht im Behälter befindet. Die Fehlermeldung von #inside kommt aus *stderror.floyd*:

>nimm die platine und das kabel aus dem container
Das Kabel muß sich hierfür in einem Behälter befinden.

Die Fehlermeldung bei Verwendung von #noun kommt aus der *onAction*-Methode von *stditem*, da erst hier geprüft wird, ob die zu #noun passende Klasse sich im Behälter befindet:

>nimm kabel und platine aus container
Das Kabel befindet sich nicht in dem Container.
Die Platine befindet sich nicht in dem Container.

Achtung: Auch in einigen anderen Satzschablonen läßt sich #noun durch einen genaueren Platzhalter ersetzen, um einige Überprüfungen in *onAction()* zu sparen. Hier ist die Standardbibliothek also noch etwas uneinheitlich.

In das oben definierte Kästchen legen wir jetzt noch einen Knochen:

```
class knochen:stditem {
   setShort("+Knochen");
   with(takeable);
   moveto(kaestchen);
}
```

Und ermöglichen damit folgenden Dialog:

Ein Raum
Weiterhin siehst du hier ein Holzkästchen.

>x kästchen
Das Holzkästchen ist geschlossen.

Informationen über den Zustand des Behälters (offen oder geschlossen) werden nur ausgegeben, wenn keine Beschreibung angegeben wurde. Daher sollten Behälter, die geöffnet und geschlossen werden können, auf *A_EXAMINE* immer mit einer dynamischen Beschreibung reagieren, die ihren Zustand berücksichtigt.

>öffne das kästchen
Du öffnest das Holzkästchen.
In dem Holzkästchen befindet sich ein Knochen.

Der Inhalt eines Behälters wird, sofern die einzelnen Klassen nicht mit *hidden* versteckt wurden, nach dem Öffnen aufgelistet. Die Anzahl nicht versteckter Klassen in einem Behälter kann mit der Funktion

```
int objectsInside(object c)
```

ermittelt werden. Vor Version 3.0 war diese Funktion in stdlist.floyd definiert, seit 3.0 ist sie eine Systemfunktion. Die Funktion

```
items()
```

zählt dagegen immer auch die versteckten Klassen.

>schau in das kästchen
In dem Holzkästchen befindet sich ein Knochen.

>x kästchen
Daran ist nichts Besonderes zu entdecken.
In dem Holzkästchen befindet sich ein Knochen.

>schliesse das kästchen
Du schließt das Holzkästchen.

Wenn einem Behälter das Attribut *transparent* gegeben wird, ist sein Inhalt auch im geschlossenen Zustand sichtbar. Das gilt natürlich auch für eventuell darin befindliche Lichtquellen.

Begehbare Behälter

Der Spieler kann sich auch selbst in einen Behälter begeben (z.B. in eine Telefonzelle). Der Behälterklasse muß dafür nur das Attribut *enterable* gegeben werden. Mit den Verben

```
verb("rein", A_ENTER, 0);
verb("raus", A_EXIT, 0);
verb("steige ein", A_ENTER, 0);
verb("steige aus", A_EXIT, 0);
verb("betrete #reachable",A_GOINTO,0);
verb("steige|gehe in|ins #reachable",A_GOINTO,0);
```

der Standardbibliothek kann der Spieler den Behälter jetzt betreten und verlassen. Zusätzlich kann der Behälter auch das Attribut *openable* haben. Der Spieler kann dann nur rein- und rausgelangen, wenn der Behälter geöffnet ist.

Eine weitere Möglichkeit, den Spieler selbst in einen Behälter zu befördern, besteht darin, ihm das Attribut *takeable* mitzugeben. Er kann sich dann mit der Eingabe "lege dich in X" selbst in einen Behälter legen. Das Attribut *takeable* wird bei der Bearbeitung von *A_DROPINTO* in *stditem* für alle Klassen verlangt, die sich in Behälter legen lassen. Der Behälter sollte auch in diesem Fall das Attribut *enterable* haben, damit der Spieler mit "raus" wieder hinausgelangt.

Das Betreten und Verlassen von Behältern übernimmt die Methode

```
void enterOrExit(int action, object x)
```

in *stdcreature*. Ihr wird die Aktionsnummer (*A_ENTER, A_EXIT* oder *A_GOINTO*) und die Behälterklasse übergeben.

Neben "rein" und "raus" lassen sich selbstverständlich noch weitere Verben für das Betreten von Behältern definieren.

Achtung: Bei der Behandlung von *A_ENTER, A_EXIT* und *A_GOINTO* in *stdcreature* gibt es die Abfrage

```
if (!x.in(this)) {
}
```

Sie verhindert, daß sich der Spieler in einen Behälter begibt, den er selber trägt. Eine solche Verschachtelung von Klassen kann den Interpreter bei der Ausgabe des Inhalts von Behältern ins Nirwana reißen.

Ablagen

Eine besondere Form von Behältern sind Ablagen. Der Spieler kann Dinge auf sie legen und von ihnen herunternehmen. Ablagen werden in *stditem* ganz ähnlich wie normale Behälter behandelt. Eine von *stditem* abgeleitete Klasse wird mit dem Attribut *supporter* zu einer Ablage. Mit den Aktionen *A_DROPON* und *A_TAKEFROM* lassen sich andere Klassen darauf ablegen oder entfernen. Klassen, die sich auf der Ablage befinden, sind wie bei normalen Behältern einfach in ihr positioniert. Daher sollte eine Klasse auch nicht gleichzeitig die Attribute *container* und *supporter* bekommen, da *stditem* nicht feststellt, welche Dinge sich in und welche sich auf der Klasse befinden. Mit den in *stditem* definierten Strings *s_support* (Singular) und *p_support* (Plural) kann bestimmt werden, wie sich Dinge auf der Ablage befinden, ob sie liegen, sitzen oder stehen.

Mengen, Größen und Gewichte

Die Standardbibliothek berücksichtigt beim Umgang mit Behältern keine Maximalwerte für die Menge, die Größe oder das Gewicht von Objekten, die in einen Behälter gelegt werden können. Hierfür kann eine abstrakte Klasse von *stditem* abgeleitet werden, die vor Ausführung von *A_DROPINTO* und *A_DROPALLINTO* einen entsprechenden Test mit den hineinzulegenden Klassen durchführt. Die maximale Kapazität der folgenden Behälterklasse läßt sich einfach mit den Variablen *maxInhalt*, *maxGroesse* und *maxGewicht* bestimmen. Eine weitere Variable *gewicht* ist notwendig, um das Eigengewicht und das Gewicht des bereits vorhandenen Inhalts festzustellen.

```
class abstract behaelter:stditem {
   with(takeable,container,openable,closeable);
   int maxInhalt,maxGroesse,maxGewicht,gewicht;

   /* prueft Gewicht und Groesse von x */
   int test(object x) {
      int ok=1;
      if (maxInhalt==objectsInside(this)) {
         write("In <den this> paßt nichts mehr hinein.^");
         ok=0;
      }
      gewicht+=x.gewicht;
      if (maxGewicht<gewicht && ok) {
         write("<Der x> ist zu schwer.^");
         gewicht-=x.gewicht;
         ok=0;
      }
      if (maxGroesse<x.groesse && ok) {
         write("<Der x> ist zu groß.^");
         ok=0;
      }
      return(ok);
   }

   /* ermittelt Eigengewicht+Gewicht von Inhalt */
   void ermittelGewicht() {
      object x;
      fetch(x,x.in(this) && !x.has(hidden),0) {
         gewicht+=x.gewicht;
      }
```

```
   }
   int onAction(int action) {
      object x;
      int aufnehmen=1,ag;

      switch(action) {
         case(A_DROPINTO);
            if (issecond() && !isfirst()) {
               ag=gewicht;
               ermittelGewicht();
               fetch(x,x.isfirst(),0) {
                  aufnehmen=(aufnehmen) ? test(x) : 0;
               }
               gewicht=ag;
               if (aufnehmen) {
                  super.onAction(action);
               }
               else {
                  halt(1);
               }
            }
            return(1);

            case(A_DROPALLINTO);
               if (isfirst()) {
                  ag=gewicht;
                  ermittelGewicht();
                  fetch(x,x.in(actor) &&
                     x.has(takeable) &&
                     !x.has(hidden) && x!=this,0) {
                        aufnehmen=(aufnehmen) ? test(x) : 0;
                  }
                  gewicht=ag;
                  if (aufnehmen) {
                     super.onAction(action);
                  }
                  else {
                     halt(1);
                  }
               }
               return(1);

            default;
               return(super.onAction(action));
      }
   }
}
```

Alle Klassen, die in den Behälter hineingelegt werden sollen, benötigen jetzt die Variablen *groesse* und *gewicht*:

```
class beutel:behaelter {
   void init() {
      setShort("+Beutel");
      with(open);
      moveto(raum);
      maxInhalt=1;
      maxGewicht=8;
      maxGroesse=3;
   }
}
```

```
class knochen:stditem {
   setShort("+Knochen");
   with(takeable);
   moveto(raum);
   int gewicht=1, groesse=1;
}
class schwert:stditem {
   setShort("Schwert");
   with(takeable);
   moveto(raum);
   int gewicht=5, groesse=3;
}
```

Die Behälterklasse macht es notwendig, daß alle angegebenen Objekte in den Behälter passen. Paßt nur eines nicht hinein, wird *onAction()* in *stditem* nicht aufgerufen, so daß kein einziges Objekt in den Behälter gelangt. Soll dieses Verhalten geändert werden, muß der Code für *A_DROPINTO* und *A_DROPALLINTO* aus *stditem* übernommen und für jede erfolgreich getestete Klasse ausgeführt werden.

In vielen Spielen wird nur die Kapazität des Inventars begrenzt, so daß der Spieler nur *n* Dinge mit sich tragen kann. Hierfür wird einfach eine abstrakte Klasse von *stditem* abgeleitet und die Aktion *A_TAKE* abgefangen, wenn der Spieler das Objekt aufnehmen möchte. Alle anderen Aktionen zum Aufnehmen von Objekten (*A_TAKEALL, A_TAKEOUT, A_TAKEALLOUT* und *A_TAKEEXCEPT*) werden auf *A_TAKE* zurückgeführt.

```
class abstract maxI:stditem {
   with(takeable);
   int onAction(int action) {
      switch(action) {
         case(A_TAKE);
            if (objectsInside(actor)==actor.maxInventar) {
               write("<Der actor> <kannst kann> <den this>
                  nicht mehr aufnehmen.^");
               return(1);

            }
         default;
            return(super.onAction(action));
      }
   }
}
```

Die Spielerklasse (und die jedes anderen Akteurs) braucht jetzt nur noch ihre maximale Kapazität in der Variablen *maxInventar* anzugeben.

```
class spieler:stdcreature {
   int maxInventar=4;
}
```

5.5 Transporter

Transporter bewegen sich mit dem Spieler, wenn er sich in oder auf ihnen befindet. Wenn der Spieler sich mit einem Transporter bewegt, wird in der Statuszeile neben dem Raumnamen die Meldung "in Transportername" ausgegeben. Ein Transporter wird von *stditem* abgeleitet und muß das Attribut *moveable* bekommen. Damit er sich mit dem Spieler bewegt, muß sich die Spielerklasse in der Transporterklasse befinden.

Ob in der Statuszeile "in einem X" (Auto) oder "auf einem X" (Fahrrad) erscheint, wird mit dem Attribut *enterable* bestimmt. Ist es vorhanden, kann der Spieler mit "rein" und "raus" (*A_ENTER,*

A_EXIT) den Transporter betreten und verlassen. Wenn der Spieler sich nur auf einen Transporter setzen (stellen, legen) soll, müssen dafür neue Verben definiert werden.

In der Methode *isLight()* von *stdroom* ist definiert, daß der Spieler nichts sehen kann, wenn er sich in einem geschlossenen Behälter ohne Lichtquelle befindet. Die Transporterklasse sollte also auch noch das Attribut *open* oder *transparent* bekommen. Ob der Transporter beide Attribute oder nur eines bekommt, ist von dem gewünschten Fortbewegungsmittel abhängig. Ein Auto ist *transparent* (Fenster) und kann geöffnet und geschlossen werden. Hier wären also auch noch *openable* und *closeable* angebracht. Einem Pferd hingegen braucht nur *open* mitgegeben zu werden.

Folgendes Beispiel läßt den Spieler auf einem Einrad fahren:

```
#define A_RAUF    100
#define A_RUNTER  101

verb("steige auf|aufs #single",A_RAUF,0);
verb("steige von|vom #single",A_RUNTER,0);
verb("steige von|vom #single runter",A_RUNTER,0);

class rad:stditem {
   setShort("Einrad,Rad");
   with(moveable,open);

   int onAction(int action) {
      switch(action) {
         case(A_RAUF);
            if (!actor.in(rad)) {
               write("<Der actor> sitzt jetzt auf dem Rad.^");
               actor.moveto(rad);
               with(hidden);
            }
            else {
               write("<Der actor> sitzt doch schon auf dem Rad.^");
            }
            return(1);

         case(A_RUNTER);
            if (actor.in(rad)) {
               write("<Der actor> <steigst steigt> vom Rad.^");
               actor.moveto(rad.location());
               with(~hidden);
            }
            else {
               write("<Der actor> sitzt nicht auf dem Rad.^");
            }
            return(1);

         default;
            return(super.onAction(action));
      }
   }
}
```

Wenn der Spieler auf das Rad steigt, wird es versteckt, um dem Spieler nicht nach jedem Raumwechsel mitzuteilen, daß er hier noch ein Einrad sieht. Wenn das Rad das Attribut *enterable* bekommt, kann die Behandlung von *A_RAUF* durch den Aufruf von

```
spieler.onAction(A_ENTER)
```

ersetzt werden. In *stdcreature* wird dann auch der passende Text "Du befindest dich jetzt auf dem Rad" ausgegeben. Wenn die Behandlung von *A_RUNTER* aber durch einen Aufruf von *A_EXIT* ersetzt wird, gibt *stdcreature* den etwas unpassenden Text "Du verläßt das Rad" aus.

Achtung: Wenn der Spieler sich mit einem Transporter bewegen kann, muß genau darauf geachtet werden, wann *location()* und wann *room()* verwendet wird. Gerade Ausdrücke wie *player.location()* können leicht die Transporterklasse statt des erwarteten Raums liefern.

Wenn der Spieler sich ohne Transporter bewegt, wird für jeden Raumwechsel die Methode *goto()* in *stdroom* aufgerufen (s. Kapitel 5.1). In *stditem* gibt es diese Methode auch. Sie ruft aber nur wieder *goto()* in *stdroom* auf. Diese Methode kann in *stditem* überladen werden, wenn die Bewegung eines Transporters eingeschränkt werden soll, z.B. sollten Boote nur im Wasser schwimmen und Fahrräder keine Treppen hochfahren können.

```
class boot:stditem {
   setShort("Boot");
   with(moveable,enterable);

   void goto(int dir, object who) {
      object r;
      int ok=1;
      r=room();
      if (r.exits[dir]!=NULL) {
         r=r.exits[dir];
            r=(r.from==location()) ? r.to : r.from;
            if (!r.has(wasser)) {
               write("Hier schwimmt's sich schlecht.^");
               ok=0;
            }
         }
         if (ok) {
            r=location();
            r.goto(dir,this);
         }
   }
}
```

Nicht alle Objekte, mit denen der Spieler den Ort wechseln kann, müssen das Attribut *moveable* haben und ihm folgen. Z.B. können Aufzüge und Teleporter den Spieler in einen anderen Raum bewegen, ohne daß er dafür eine Himmelsrichtung angeben muß.

5.6 Lebewesen

Die Spielerklasse und die Klassen aller NPCs (Non Player Characters) werden von *stdcreature* abgeleitet. Mit der Anweisung

```
setPlayer(object Spieler)
```

kann die Spielerklasse jederzeit geändert werden. Die Spielerklasse muß dabei von *stdcreature* abgeleitet sein. Mit der Systemvariablen

```
object player
```

kann die Spielerklasse nachträglich festgestellt werden. Ob die aktuelle Aktion vom Spieler oder einem NPC ausgeführt wird, beantwortet die Systemvariable

```
object actor
```

Sie enthält nach jeder Eingabe die Klasse des Akteurs.

Anweisungen an NPCs

Wenn der Spieler einem NPC eine Anweisung gibt (z.B. "Troll, nimm die Krone"), wird die entsprechende Aktion in der NPC-Klasse ganz normal ausgeführt. Ein Unterschied besteht nur in der darauf folgenden Meldung der Standardbibliothek ("Der Troll nimmt die Krone" statt "Du nimmst die Krone").

Soll ein NPC auf eine Aktion anders reagieren als die Spielerklasse, kann die Methode

```
int onOrder(int action)
```

in *stdcreature* überladen werden. Sie wird immer vor der Ausführung einer Anweisung an einen NPC aufgerufen und hat normalerweise den Rückgabewert 0, was bedeutet, daß die Aktion vom NPC ausgeführt werden kann. Wenn sie 1 zurückgibt, wird die Aktion abgebrochen.

Angenommen, ein Troll soll sich weigern, dem Spieler irgendein Objekt zu geben, dann kann dieses Verhalten wie folgt umgesetzt werden.

```
class troll:stdcreature {
   setShort("+Troll");
   int onOrder(int action) {
      object x;
      int ok;
      switch(action) {
         case(A_GIVEME);
            fetch(x,x.isfirst()) {
               if (x.in(troll)) {
                  write("<Den x> scheint der Troll nicht mehr hergeben zu  wollen.^");
                  ok=1;
               }
            }
            return(ok);
         default;
            return(0);
      }
   }
}
```

Die Methode *onOrder()* findet sich auch in *stdobject*, hat dort aber nur die Funktion, unsinnige Anweisungen an unbelebte Objekte abzufangen.

Der Spielerwechsel

Das folgende Beispiel ermöglicht es dem Spieler, zwischen zwei Personen zu wechseln.

```
#define A_ALTEREGO 100
verb("alter ego",A_ALTEREGO,0);

class abstract spieler:stdcreature {
   int onAction(int action) {
      object alter;
      switch(action) {
         case(A_ALTEREGO);
            alter=(player==jekyll) ? hyde : jekyll;
            write("Du schlüpfst jetzt in die Haut von <~der alter>^");
            setPlayer(alter);
            return(1);
         default;
            return(super.onAction(action));
      }
```

```
   }
}

class jekyll:spieler {
   setShort("$Dr. Jekyll, $Jekyll");
}

class hyde:spieler {
   setShort("$Mr. Hyde, $Hyde");
}

class RaumA:stdroom {
   setShort("A");
}

void main() {
   setPlayer(hyde);
   hyde.moveto(RaumA);
   jekyll.moveto(RaumA);
}
```

Der Code läßt sich natürlich auch erweitern, um den Spieler z.B. mehr als zwei Mitglieder einer Rollenspiel-Party übernehmen zu lassen. Wenn sich alte und neue Spielerklasse nicht im gleichen Raum befinden, muß beim Wechsel noch eine der beiden Klassen in einen anderen Raum befördert werden.

5.7 Essen und Trinken

Objekte mit den Attributen *edible* und *potable* können vom Spieler gegessen bzw. getrunken werden. Für Nahrungsmittel sind in *stdconst* die Aktionen *A_EAT* und *A_DRINK* definiert, die in *stditem* behandelt werden:

```
verb("iß|esse|verspeise #held",A_EAT,0);
verb("trinke #held",A_DRINK,0);
```

In *stditem* ist die Integervariable *food* definiert, mit der sich angeben läßt, nach wie vielen Runden die Nahrung aufgebraucht ist. Objekte, die aufgegessen oder ausgetrunken wurden, können in ein Versteck verschoben werden, um sie aus der Spielwelt zu entfernen. Ob ein Nahrungsobjekt aufgebraucht wurde, läßt sich nach jeder Runde mit einem Daemon (s. 6.2) überprüfen:

```
class apfel:stditem {
   setShort("+Apfel");
   moveto(spieler);
   with(edible, takeable);

   void init() {
      food=3;
      startDaemon();
   }

   void daemon() {
      if (food==0) {
         moveto(versteck);
         stopDaemon();
      }
   }
}
```

6 Der Spielverlauf

Der Spieler kann den Zustand der Spielwelt normalerweise in jeder Runde verändern. Manche Veränderungen finden aber auch unabhängig vom Spieler statt, und manchmal soll der Spieler auch gar nicht sofort bemerken, was er verändert.

6.1 Die Spielzeit

Manche Adventures erfordern vom Spieler die Lösung einer Aufgabe in einer bestimmten Zeit (z.B. Infocoms Witness). In Floyd ist die Spielzeit unabhängig von der realen Zeit und wird mit einer 24 Stunden Uhr gemessen. Sie kann mit der Anweisung

```
setTime(int Zeit, int n)
```

eingestellt werden. Der Parameter *Zeit* gibt die Zahl der Minuten seit Mitternacht an und läßt sich mit der Formel

```
Zeit = Stunden * 60 + Minuten
```

ermitteln. 9 Uhr 10 entspricht so z.B. einem Zeitwert von 550. Ein Beispiel für die umgekehrte Berechnung der Uhrzeit aus der Zahl der Minuten findet sich in Kapitel 2.10. Der Parameter *n* bestimmt, wie schnell die Zeit im Spiel vergeht. Bei einem negativen Wert von *n* vergeht in *n* Runden (Spielereingaben) eine Minute. Bei einem positiven Wert von *n* vergehen in einer Runde *n* Minuten.

Ermitteln läßt sich die Zeit mit der Funktion

```
int time()
```

Sie liefert die Spielzeit in Minuten. Wenn 24 Stunden vergangen sind und die Uhr zurückgestellt wird, ist es natürlich auch einfach, Tage zu zählen. Daher liefert die Funktion

```
int day()
```

die Zahl der Tage, die der Spieler schon in der Spielwelt verbracht hat.

6.2 Zeitschalter und Dämonen

Nach der Verarbeitung der Spielereingabe und den daraus folgenden Ausgaben können in jeder Klasse Zeitschalter und Dämonen ausgeführt werden. Ein Zeitschalter ist eine Methode, die *n* Runden nach ihrer Aktivierung ausgeführt wird. Ein Dämon ist eine Methode, die nach ihrer Aktivierung am Ende jeder Runde ausgeführt wird.

Zeitschalter

Die Methode für einen Zeitschalter muß

```
void timeout()
```

heißen und wird mit der Anweisung

```
startTimer(int n)
```

aktiviert, wobei *n* die Anzahl der Runden angibt, nach der die Methode ausgeführt werden soll. Mit der Anweisung

```
stopTimer
```

kann ein Zeitschalter auch vorzeitig beendet werden, ohne daß *timeout()* ausgeführt wird. Bestes Beispiel für einen Zeitschalter ist eine Zeitbombe, die eine bestimmte Zeit nach ihrer Aktivierung explodiert:

```
class kabine:stdroom {
  setShort("-Kabine");
  setLong("Deine Kabine ist mit allen Annehmlichkeiten ausgestattet,
     die ein kaiserlicher Palastkreuzer zu bieten hat. Besonders auffällig
     ist die kleine Metallkugel auf dem Bett, die du sofort als
     Positronenbombe erkennst.^");
}

class bombe:stditem {
   setShort("-Metallkugel,-Bombe");
   with(hidden,takeable,switchable);
   moveto(kabine);

   int onAction(int action) {
      int result;

      switch(action) {
         case(A_SWITCHON);
            result = super.onAction(action);
            if (has(on)) {
               startTimer(3);
            }
            return result;

         case(A_SWITCHOFF);
            result = super.onAction(action);
            if (!has(on)) {
               stopTimer();
            }
            return result;

         case(A_EXAMINE);
            if (has(on)) {
               write("Ein kleiner Schalter in der Kugel blinkt
                  in einem roten Licht.^");
            }
            else {
               write("In der Kugel befindet sich ein kleiner
                  Schalter.^");
            }
            return 1;

         default;
            return(super.onAction(action));
      }
   }

   void timeout() {
      write("Die Positronenbombe explodiert ...^");
      quit;
   }
}
```

Kabine

Deine Kabine ist mit allen Annehmlichkeiten ausgestattet, die ein kaiserlicher Palastkreuzer zu bieten hat. Besonders auffällig ist die kleine Metallkugel auf dem Bett, die du sofort als Positronenbombe erkennst.

>u metallkugel
In der Kugel befindet sich ein kleiner Schalter.

>schalte die bombe ein
Die Metallkugel ist jetzt eingeschaltet.

>schalte die bombe aus
Die Metallkugel ist jetzt ausgeschaltet.

>schalte die bombe ein
Die Metallkugel ist jetzt eingeschaltet.

>warte
Die Zeit vergeht ...

>warte
Die Zeit vergeht ...

>warte
Die Zeit vergeht ...
Die Positronenbombe explodiert ...

Dämonen

Wenn NPCs sich selbständig durch die Spielwelt bewegen sollen, sind Dämonen (daemons) angesagt. Die Methode, die nach jeder Runde in Klassen mit aktiven Dämonen aufgerufen wird, ist

```
void daemon()
```

Für die Bewegung eines NPC kann in ihr einfach seine Position geändert werden. Im folgenden Beispiel läuft eine Maus unabhängig vom Spieler durch drei Räume. Die zu durchlaufenden Räume werden in einem Pfad-Array gespeichert, in dem die Maus unermüdlich auf und ab läuft. Dabei wird in der Methode *daemon()* auch geprüft, ob 1. die Maus in den Raum läuft, in dem sich auch der Spieler befindet, oder ob 2. die Maus den Spieler gerade verlassen hat. Schließlich kann der Spieler auch noch in den Raum eintreten, in dem sich die Maus gerade befindet. Um ihm diesen Fall mitzuteilen, muß die Mausposition nach jedem Raumwechsel in der Spielerklasse abgefragt werden.

```
class raumA:stdroom {
   setShort("Erster Raum");
   setLong("...^");
}

class raumB:stdroom {
   setShort("Zweiter Raum");
   setLong("...^");
}

class raumC:stdroom {
   setShort("Dritter Raum");
   setLong("...^");
}
```

```
class A2B:stdexit {
}

class B2C:stdexit {
}

class spieler:stdcreature {
   int onAction(int action) {
      return(super.onAction(action));
      if (action<11 && maus.room()==spieler.room()) {
         write("Du begegnest einer Maus.^");
      }
   }
}

class maus:stdcreature {
   setShort("-Maus");
   moveto(raumC);
   object pfad[3]=(raumA,raumB,raumC);
   // raum: aktueller Raum im Pfad
   // richtung: 0=vorwärts 1=rückwärts
   int raum,richtung;
   void init() {
      raum=3;
   }
   void daemon() {
      int raum_alt;
      raum_alt=raum;
      raum=(richtung==0) ? raum+1 : raum-1;
      if (raum<0 || raum>2) {
         raum=1;
         richtung=1-richtung;
      }
      moveto(pfad[raum]);
      if (pfad[raum]==spieler.room()) {
         write("Eine Maus kommt in den Raum gelaufen.^");
      }
      if (pfad[raum_alt]==spieler.room()) {
         write("Die Maus verläßt dich.^");
      }
   }
}

void main() {
   setPlayer(spieler);
   spieler.moveto(raumA);
   raumA.addExit(D_NORTH,raumB,A2B);
   raumB.addExit(D_EAST,raumC,B2C);
   maus.startDaemon();
   raumA.description();
}
```

Die Anweisung *startDaemon()* sorgt dafür, daß die Methode *daemon()* nach jeder Runde aufgerufen wird. Mit der Anweisung

```
stopDaemon
```

wird der Dämon wieder angehalten. Die Begegnung der Maus könnte jetzt so aussehen:

Erster Raum ...

```
>n
zweiter Raum ...
Eine Maus kommt in den Raum gelaufen.

>z
Die Zeit vergeht ...
Die Maus verläßt dich.

>o
Dritter Raum ...

>w
Zweiter Raum
Du siehst hier eine Maus.
Die Maus verläßt dich.

>s
Erster Raum

>z Die Zeit vergeht ...
Eine Maus kommt in den Raum gelaufen.
```

Damit die Maus nicht zu schnell durch die Räume flitzt und den Spieler nicht schon wieder verläßt, wenn er ihr gerade begegnet, kann in die Klasse maus ein zusätzlicher Zähler eingebaut werden, der in der Methode *daemon()* hochgezählt wird. Ein Raumwechsel findet dann erst nach einer bestimmten Zahl von Runden statt. Diese Variante gleicht einem Zeitschalter, der in der Methode *timeout()* neugestartet wird.

Zeitschalter und Dämonen können mit der Anweisung *halt(3)* auch eine Runde ausgesetzt werden. Auf die Listen der in der Eingabe erwähnten Klassen hat die *halt*-Anweisung mit dem Parameter 3 keine Auswirkung.

6.3 Punkte

Der Punktestand des Spielers kann mit der schon im Gewand verwendeten Anweisung

```
addScores(int scores)
```

verändert werden. Normalerweise wird eine Veränderung sofort in der Statuszeile angezeigt.

Komfortabler für den Spieler ist es aber, wenn er auch mit einer Meldung über die neuen Punkte informiert wird und mit einem besonderen Kommando auch sehen kann, für welche Aufgaben er wie viele Punkte bekommen hat. Auch wäre es praktisch, automatisch zu verhindern, daß der Spieler für eine Aktion, die er mehrmals durchführen kann, auch mehrmals Punkte erhält.

Im *Nebelmond* gibt es für die Punkteverwaltung eine abstrakte Klasse. Sie enthält eine Kurzbeschreibung der einzelnen Aufgaben und weiß, wieviele Punkte es für jede Aufgabe gibt.

```
#define MAX_AUFGABEN 3
class abstract scoring {
   // Punkte für einzelne Aufgaben
   int punkte[MAX_AUFGABEN]=(2,4,6);
   // Einzelne Aufgabe schon geloest? 0=nein 1=ja
   int geloest[MAX_AUFGABEN];
   // Beschreibung einzelner Aufgaben
   string aufgaben[MAX_AUFGABEN]=(
      "Du hast Aufgabe A erledigt (",
```

```
        "Du hast Aufgabe B erledigt (",
        "Du hast Aufgabe C erledigt (");

      // Punkte fuer Aufgabe n vergeben
      void plus(int n) {
         if (!geloest[n]) {
            addScores(punkte[n]);
            geloest[n]=1;
            if (isverbose) {
               write("^*** Damit hast du dir "+punkte[n]+" Punkte verdient ***^");
            }
         }
    }

    // Liste erledigter Aufgaben
    void list() {
      if (scores()>0) {
         int i,s;
         for(i=0;i<MAX_AUFGABEN;i++) {
           s+=punkte[i];
           if (geloest[i]) {
             write(aufgaben[i]+punkte[i]+" Punkte) ");
           }
         }
         write("^Damit hast du in "+moves()+" Zügen "+scores()+
             " Punkte von "+s+" möglichen Punkten erreicht.^");
      }
      else {
         write("Mit 0 Punkten bist du leider noch ein sehr
            erfolgloser Abenteurer.^");
      }
   }
}
```

Immer wenn der Spieler für eine bestimmte Aufgabe Punkte erhalten soll, wird ihre Methode *plus()* mit der Nummer der jeweiligen Aufgabe aufgerufen. Die Klasse merkt sich dann, daß diese Aufgabe erledigt ist. Die Methode *list()* liefert schließlich eine Liste der erledigten Aufgaben. Dabei liefern die Funktionen

```
int scores()
```

und

```
int moves()
```

die Zahl der Punkte und Züge. Die Benachrichtigung des Spielers ist hier von der in *stdconst.floyd* definierten Variablen *isverbose* abhängig. Sie können natürlich auch eine eigene Variable einfügen. Die Benachrichtigung ist dann unabhängig davon, ob der Spieler sich auch ausführliche Raumbeschreibungen ausgeben läßt.

6.4 Spielstände

Wenn eine Spielstanddatei (.floyds) geladen wird, wird die Raumbeschreibung ausgegeben, um den Spieler über den Zustand der Spielwelt zu informieren. Dieses Verhalten kann mit der Systemvariablen

```
string load;
```

geändert werden. Sie bestimmt das Kommando, das nach dem Laden eines Spielstandes ausgeführt wird und hat die Voreinstellung "u" für die Raumbeschreibung.

6.5 Aufrufe

Die folgende Liste zeigt noch einmal alle Funktionen, die vom Interpreter nach einer Eingabe aufgerufen werden. Die Reihenfolge der Aufrufe entspricht der Reihenfolge in der Tabelle.

Kontext	Funktion	Optional	Beschreibung
global	`string before(string Eingabe)`	ja	Die Auswertung der Eingabe wird mit dem zurückgegeben String fortgesetzt.
Klasse	`void beforeAction(int action)`	ja	Wird auch in nicht in der Eingabe vorkommenden Klassen ausgeführt.
Klasse	`int onOrder(int action)`	nein	Bricht angesprochener NPC die Aktion ab? (1=ja, 0=nein, Aktion kann ausführt werden)
Klasse	`int onAction(int action)`	nein	Wird in den in der Eingabe vorkommenden Klassen aufgerufen. Der Rückgabewert gibt an, ob die Aktion bearbeitet wurde (0=nein, 1=ja). Der Aufruf weiterer *onAction()* kann mit der *halt*-Anweisung verhindert werden. Wird die Aktion von keiner anderen Klasse verarbeitet, wird *onAction()* in der Spielerklasse aufgerufen.
Klasse	`void daemon()`	nein	Aufruf in allen Klassen mit aktivem Daemon.
Klasse	`void timeout()`	nein	Aufruf in allen Klassen mit abgelaufenem Timer.

7 Eingabe und Ausgabe

7.1 Listen

Wenn der Spieler sein Inventar anzeigen läßt oder nach Raumbeschreibungen die im Raum befindlichen Objekte aufgelistet werden, wird auf Funktionen in *stdlist.floyd* zurückgegriffen, die automatisch Auflistungen von Klassennamen erzeugen.

```
void shortList(object c, int casus, int m)
```

Diese Funktion gibt die Namen der in *c* befindlichen Klassen untereinander im Fall *casus* aus. Der Inhalt von Behältern wird um *m* Leerzeichen eingerückt. Mit dieser Funktion wird in *stdcreature* das Inventar ausgegeben.

```
void longList(object c, int casus)
```

Diese Funktion gibt die Namen der in *c* befindlichen Klassen im Fall *casus* getrennt von Kommata und einem "und" in einem Satz aus. *longList()* wird in *stdroom* für Raumbeschreibungen verwendet. Wenn die Liste nicht direkt ausgegeben werden soll, kann die Funktion

```
 string strLongList(object c, int casus)
```

verwendet werden, die dieselbe Liste als String zurückgibt.

```
string contentList(object c, string vsingular, string vplural, int pronomen)
```

Diese Funktion liefert eine Liste der in *c* befindlichen Objekte als String zurück. Die Parameter *vsingular* und *vplural* können die Singular- bzw. Pluralform eines Verbs enthalten, das angibt, "wie" sich die Objekte in dem Behälter befinden (z.B. "liegen", "sitzen" oder einfach "befinden"). Der Parameter *pronomen* bestimmt, ob die Liste mit dem Behälternamen ("In X", *pronomen*=0) oder einem stellvertretenden Pronomen ("In ihm/ihr", *pronomen*=1) eingeleitet wird. Zusätzlich wird noch berücksichtigt, ob sich der Spieler selbst in dem Behälter befindet. In diesem Fall lautet die Einleitung "Neben dir". *contentList()* ist hilfreich, wenn die Liste selber Teil einer längeren Beschreibung sein soll. Im *Nebelmond* wird z. B. der Inhalt der Hängematte mit *contentList()* in ihre Beschreibung eingefügt:

```
string s="... In einer Ecke der Kabine baumelt eine Hängematte. ";
if (objectsInside(matte)>0) {
  s+=contentList(matte,"liegt","liegen",1)+".^";
}
s+="Rechts führt ein Durchgang in den Laderaum.^";
setLong(s);
```

Die Funktion

```
int objectsInside(object c)
```

ist eine interne Floyd-Funktion und liefert die Zahl aller in *c* befindlichen Objekte, die nicht mit dem Attribut *hidden* versteckt sind.

Seit Version 3.3 läßt sich mit dem Attribut *unique* bestimmen, ob eine Klasse mit bestimmten oder unbestimmten Artikel aufgelistet wird. Klassen mit diesem Attribut erscheinen in Listen mit

einem bestimmten Artikel. Das kann nützlich sein, um bestimmte Objekte hervorzuheben: „In der Truhe befinden sich ein Apfel, ein Zahnrad und *die* grüne Statue des Baal“.

7.2 Die Statuszeile

In Floyd ist die Statuszeile immer in eine linke und eine rechte Hälfte geteilt. In jeder Hälfte können unterschiedliche Informationen ausgegeben werden. Normalerweise erscheinen links der aktuelle Raumname und rechts die Punkte und Züge des Spielers. Der Text der linken Seite wird von der Methode

```
string StatusLine()
```

in *stdroom* und *stditem* bestimmt. Diese Methode wird in jeder Runde in der aktuellen Raumklasse aufgerufen und prüft, ob der Spieler etwas sehen kann. Sie gibt dann entweder den Raumnamen oder das Wort "Dunkelheit" zurück. Wenn sich der Spieler in einem Fahrzeug befindet, wird zuerst *StatusLine()* in der Transporterklasse aufgerufen, um die Meldung "in einem Transporter" zu erzeugen. Wenn *StatusLine()* in *stdroom* überladen wird, kann auf der linken Seite der Statuszeile beliebiger Text ausgegeben werden. Das Format der rechten Seite läßt sich mit der Anweisung

```
StatusLineFormat(int format)
```

bestimmen. Der Parameter *format* kann folgende Werte annehmen: (1) Punkte und Züge werden im Format "p/z" angezeigt. (2) Punkte und Züge werden im Format "Punkte:p Züge:z" angezeigt. (3) Die Spielzeit wird im Format "HH:MM" angezeigt. (4) Es wird der Text der globalen Stringvariable *stlmessage* ausgegeben, die bereits vom System deklariert ist. (5) Der Text von *StatusLine()* wird zentriert angezeigt. Die genaue Ausrichtung des zentrierten Textes funktioniert nur mit einer nichtproportionalen Schrift wie Courier.

Da der Text der rechten Seite mit der Variablen *stlmessage* bestimmt werden kann, ist er im Gegensatz zum Text der linken Seite unabhängig von der Position der Spielerklasse, die bestimmt, in welcher Raum- oder Transporterklasse die Methode *StatusLine()* zur Textausgabe aufgerufen wird.

Neben der Statuszeile läßt sich auch der Text in der Titelzeile des Fensters anpassen. Hierfür gibt es die Systemvariable

```
string title
```

Sie kann z. B. den Spieltitel aufnehmen. Wird *title* kein Wert zugewiesen, erscheint in der Titelzeile der Dateiname.

7.3 Textfenster

Manchmal sollen dem Spieler kurze Texte (z.B. Zitate) angezeigt werden, die aber nicht im normalen Text auftauchen brauchen. Hierfür kann die Anweisung

```
box(string s)
```

verwendet werden. Sie gibt eine oder mehrere durch das Caret-Zeichen getrennte Textzeilen umrahmt in der oberen Bildschirmhälfte aus.

```
box("Erste Zeile^
   Zweite Zeile^
   Dritte Zeile");
```

Die weitere Abarbeitung des Programms wird durch die Ausgabe nicht unterbrochen.

7.4 Grafik

Zusätzlich zum Text kann Floyd auch Grafiken darstellen. Alle Grafiken, die in einem Spiel erscheinen sollen, müssen im PNG-Format vorliegen. Sie werden in eine Floyd-Grafikdatei kopiert und können im Spiel über eine Bildnummer angezeigt werden. Mit folgenden Schritten lassen sich Grafiken in ein Spiel einbinden:

1. Menüpunkt *Optionen/Grafik* kopieren: In dem Dateidialog können mehrere PNG-Dateien ausgewählt werden (Mehrfachauswahl mit *Strg*- oder *Apfel*-Taste).

2. Im Floyd-Programmordner (oder im Home-Ordner des Benutzers, wenn im Programmordner nicht geschrieben werden kann) wird die Datei *neu.images* angelegt. Sie muß in den Namen der Spieldatei (z.B. nebelmond.images) umbenannt und in den Ordner der Spieldatei verschoben werden.

3. Wenn die Datei neu.images angelegt wird, gibt Floyd die Namen der einzelnen Grafiken und eine zugehörige Nummer aus. Diese Grafiknamen und ihre Nummern müssen als Konstanten in die Spieldatei eingefügt werden, damit Floyd über eine Bildnummer auf die entsprechende Grafik zugreifen kann.

Wenn z. B. drei PNG-Grafiken mit den Namen *agent.png*, *babelfish.png* und *caligula.png* ausgewählt werden, gibt Floyd folgende Konstanten aus:

```
#define AGENT 0
#define BABELFISH 1
#define CALIGULA 2
```

Wenn diese Zeilen in die Programmdatei eingefügt werden, kann mit der Anweisung

```
showImage(int Bildnummer)
```

jetzt die entsprechende Grafik angezeigt werden.

7.5 Die Tastatur

In der Spielereingabe können die Klein- und Großbuchstaben von A bis Z plus den deutschen Umlauten, der Punkt, das Komma, doppelte Anführungsstriche und das Leerzeichen vorkommen. Diese Zeichen können auch mit der Funktion

```
string getKey()
```

von der Tastatur gelesen werden. Die Funktion läßt den Interpreter auf einen Tastendruck warten und gibt ein einzelnes Zeichen zurück. Dabei werden Großbuchstaben (außer Umlauten) automatisch in Kleinbuchstaben umgewandelt. Eine einfache Ja-Nein-Abfrage ist mit dem folgenden Beispiel möglich:

```
int ja_nein() {
   string t;
   do {
      t=getKey();
   } while (t!="j" && t!="n");
   write(t+"^");
   return(t=="j");
}
```

Da die *getKey*-Funktion keine Sondertasten wie [Entf] berücksichtigt, läßt sie sich nur eingeschränkt für die Eingabe mehrerer Zeichen verwenden.

7.6 Menüs

Viele Spiele verwenden Menüs, um den Spieler zu Hintergrundinformationen oder Hilfestellungen zu führen. In Floyd gibt es dafür die Funktion

```
int menu(int Titel, string Auswahl)
```

Der Parameter *Auswahl* enthält die einzelnen Menüpunkte, die durch das Caret-Zeichen voneinander getrennt werden. Der Parameter *Titel* bestimmt, ob der Text bis zum ersten Caret in *Auswahl* einen normalen Menüpunkt (0) oder eine über dem Menü auszugebende Titelzeile (1) darstellt. Die Funktion liefert die Nummer des ausgewählten Menüpunkts aus Auswahl zurück. Der erste Punkt hat dabei immer die Nummer 1. Ein Rückgabewert von 0 bedeutet, daß das Menü mit [Esc] abgebrochen wurde.

```
void main() {
   int m;

   do {
      m=menu(1,"Menütitel^
         Punkt A^
         Punkt B^
         Punkt C");

      switch (m) {
         case(1);
            write("Text für den ersten Punkt ...^");
            break;

         case(2);
            write("Text für den zweiten Punkt ...^");
            break;

         case(3);
            write("Text für den dritten Punkt ...^");
            break;

      }

      if (m>0) {
         write("^Weiter");
         getKey();
      }
   } while(m>0);
}
```

Wenn die einzelnen Menüpunkte nur Text ausgeben, kann die *switch*-Anweisung durch ein Array ersetzt werden:

```
void main() {
   int m;
   string auswahl[3]=(
      "Text für den ersten Punkt ...^",
      "Text für den zweiten Punkt ...^",
      "Text für den dritten Punkt ...^");

   do {
      m=menu(1,"Menütitel^
         Punkt A^
         Punkt B^
         Punkt C");
      if (m>0) {
```

```
         write(auswahl[m-1]);
         write("^Weiter");
         getKey();
      }
   } while(m>0);
}
```

Gibt ein Untermenü nur Text aus, läßt es sich wie folgt umsetzen:

```
void warte(int a) {
   if (a>0) {
      write("^Weiter");
      getKey();
   }
}

void main() {
   int m,n;
   string sub[2]=(
      "Text für ersten Unterpunkt ...^",
      "Text für zweiten Unterpunkt ...^");

   do {
      m=menu(0,"Punkt A^Punkt B^");

      switch (m) {
         case(1);
            write("Text für den ersten Punkt ...^");
            break;
         case(2);
            do {
               n=menu(0,"Unterpunkt A^Unterpunkt B^");
               write(sub[n-1]);
               warte(n);
            } while(n>0);
      }
      warte(m==1);
   } while(m>0);
}
```

Bei Untermenüs muß darauf geachtet werden, nach welchen Menüpunkten auf einen Tastendruck gewartet wird. Wenn ein Untermenü abgebrochen wird, braucht in der Hauptschleife nicht mehr auf einen Tastendruck gewartet zu werden.

Hilfestellungen

Menüs eigen sich auch für Hilfestellungen zu den einzelnen Rätseln im Spiel. Dabei gibt es zu jedem Rätsel mehrere Antworten, die immer genauer werden und dem Spieler nach und nach angezeigt werden, so daß das Spiel nicht zu schnell an Reiz verliert. Zuerst aufgetaucht sind solche Hilfestellungen in den Spielen von Infocom, wo sie auch Invisiclues genannt wurden.

Die einzelnen Fragen lassen sich leicht mit der *menu*-Anweisung auswählen. Die Antworten kommen in ein Array und werden nach einem Tastendruck ausgegeben.

```
void main() {
   int m,n;
   // Antworten auf drei Fragen
   string antworten[9]=(
      "Antwort 1 zur Uhr", "Antwort 2 zur Uhr",
      "Antwort 1 zu Merlin", "Antwort 2 zu Merlin",
```

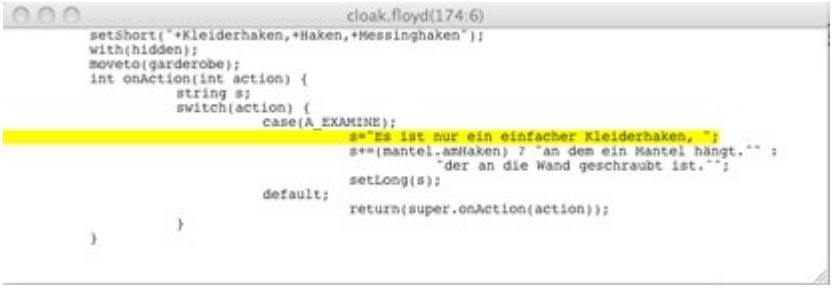
```
      "Antwort 3 zu Merlin", "Antwort 4 zu Merlin",
      "Antwort 1 zu Gonzo", "Antwort 2 zu Gonzo",
      "Antwort 3 zu Gonzo");
   // Erster Index einzelner Fragen
   int start[3]=(0,2,6);
   // Antworten pro Frage
   int anzahl[3]=(2,4,3);
   string t;

   do {
      m=menu(0,"Was mache ich mit der Uhr?^
         Was sage ich zu Merlin?^
         Wie werde ich Gonzo los?");

      if (m>0) {
         write(anzahl[m-1]+" Tips  Zurück mit Z^");
         n=start[m-1];
         do {
            t=getKey();
            if ((t!="z") && (n-start[m-1]<anzahl[m-1])) {
               write(antworten[n++]+"^");
            }
         } while(t!="z");
      }
   } while(m>0);
}
```

8 Der Debugger

Um die Fehlersuche zu erleichtern, verfügt Floyd über einen Debugger, mit dem sich der Programmablauf verfolgen und die Werte von Variablen überprüfen lassen. Aktiviert wird der Debugger mit dem Menüpunkt *Optionen/Debugger.* Wenn er aktiv ist, erscheint hier ein Häkchen. Um Spielern nicht zuviel zu verraten, arbeitet der Debugger nur mit Quelltextdateien.

Wenn der Debugger aktiv ist und ein Programm gestartet wurde, erscheinen zwei zusätzliche Fenster. Im ersten Fenster wird die gerade ausgeführte Programmzeile angezeigt. Mit dem Menüpunkt *Optionen/Quelltextanzeige* kann auch der Pfad zu einem externen Editor angegeben werden, der die Anzeige des Quelltextes übernehmen soll. Bei der Formulierung des Ausdrucks für den Aufruf können die Variablen $FILE und $LINENR benutzt werden.

Unter Windows kann dann z. B. für den freien Editor Notepad++ der Aufruf folgendermaßen aussehen: C:\Program Files\Notepad++\notepad++.exe -n$LINENR $FILE



Das zweite Fenster dient der Steuerung des Debuggers und der Ausgabe verschiedener Meldungen. Im Feld *Zeile* kann eine Zeilennummer angegeben werden. Mit dem Button *Setzen* wird der Zeile ein Haltepunkt zugefügt, bei dem der Debugger die Programmausführung anhält. Wenn ein Häkchen bei *StdLib* gesetzt wird, können Haltepunkte auch in eingebundenen Dateien (meist die der Standardbibliothek) gesetzt werden. Mit dem Button *Laufen* wird das Programm fortgesetzt, bis der nächste Haltepunkt erreicht wird. Der Button *Einzelschritt* führt das Programm Anweisung für Anweisung aus und ermöglicht so eine genaue Ablaufkontrolle.

Ist ein Haltepunkt erreicht, können lokale und globale Variablen und Klassen ausgewählt und zur Anzeige hinzugefügt werden. Für diese werden dann bei jedem neuen Halt die Werte aktualisiert.

Im Feld *Ausgabe* erscheinen verschiedene Logmeldungen, die Floyd während der Ausführung erzeugt (z. B. gerade ausgeführte Anweisung, Dateinamen oder Fehlermeldungen von Java Exceptions).

Bestehende Haltepunkte können wieder ausgewählt und entfernt werden. Wenn das Debugfenster verschwindet, weil bei der Ausführung kein Haltepunkt mehr erreicht wurde, kann das Menü *Optionen/Debugger* zweimal nacheinander ausgewählt werden, um den Debugger wieder sichtbar zu machen.

9 Webstart

Ab Version 3.3 unterstützt Floyd den Java-Webstart. Der Interpreter und ein Spiel können so über einen Weblink gestartet werden (vorausgesetzt, auf dem PC des Benutzers ist Java installiert). Auf dem Webserver muß dafür neben der Spieldatei eine mit einem Zertifikat signierte JAR-Datei liegen. Um ein eigenes Zertifikat zu erzeugen und die Jar-Datei von Floyd zu signieren, werden die Programme *keytool* und *jarsigner* benötigt, die mit dem JDK installiert werden.

Das Programm *keytool* fordert zur Eingabe einiger Daten auf und erzeugt ein Zertifikat:

```
keytool -selfcert -validity 730 -alias CAFloydKeys -keystore CAFloydKeyStore
```

Validity gibt die Gültigkeit des Zertifikats in Tagen an. Die Namen für *alias* und *keystore* können frei gewählt werden.

Jetzt kann die JAR-Datei signiert werden:

```
jarsigner -keystore CAFloydKeyStore floyd33.jar CAFloydKeys
```

Auf dem Webserver muß jetzt noch die folgende XML-Datei mit der Endung .jnlp gespeichert werden. Wird sie verlinkt, startet sie Floyd mit der in *argument* angegebenen Spieldatei:

```
<?xml version="1.0" encoding="UTF-8"?>
  <jnlp  spec="1.0+"
   codebase="http://www.domainname.de/webstart/"
   href="floyd.jnlp">
 <information>
   <title>Floyd</title>
   <vendor>Name des Autors</vendor>
   <homepage href="http://www.domainname.de"/>
   <description>Floyd</description>
   <description kind="short">Web Start Beispiel</description>
   <icon href="floyd3.png"/> <icon kind="splash" href="floyd3.png"/>
   <offline-allowed/>
 </information>
 <security>
   <all-permissions/>
 </security>
 <resources>
   <j2se href="http://java.sun.com/products/autodl/j2se" version="1.5+"/>
   <jar href="floyd33.jar"/>
 </resources>
 <application-desc>
   <argument>http://www.domainname.de/webstart/spieldatei.floydc</argument>
 </application-desc>
</jnlp>
```

10 Index