Friedrich-Schiller-Universität Jena

Fakultät für Mathematik und Informatik

Diplomarbeit

im Studiengang Informatik

Entwurf und Implementierung einer visuellen Programmiersprache für den Einsatz in Schulen

Betreuer: Prof. Dr. Wilhelm R. Rossak, Prof. Dr. Michael Fothe

Jena, den 14. Dezember 2004

Vorgelegt von:
Lutz Kohl
Otto-Schott-Str.16
07745 Jena
E-Mail: mail@lutzkohl.de
Matrikelnummer: 54815

Inhaltsverzeichnis

Abbildungsverzeichnis

Danksagung

Bedanken möchte ich mich bei allen, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben. Besonders sind an dieser Stelle meine Betreuer Prof. Dr. Michael Fothe und Prof. Dr. Wilhelm R. Rossak zu nennen, die mir während der gesamten Zeit – manchmal fordernd, manchmal noch rechtzeitig bremsend – mit konstruktiven Vorschlägen, Gesprächen und Ideen zur Seite standen. Die von ihnen initiierten Studienarbeiten von Herrn Alexander Lärz und Frau Susanne Fritsch sind weitere Faktoren, die maßgeblich zum Fortschritt des entwickelten Systems beigetragen haben. Ich bedanke mich bei allen Lehrern, die Anforderungen nannten und die entstandene visuelle Programmiersprache kritisch testeten. Des Weiteren halfen mir Prof. Dr. Jürgen F. H. Winkler, Prof. Dr. Werner Hartmann und Dr. Wolfram Amme mit Diskussionen und Literaturhinweisen. Außerdem war jede Form der von den fleißigen Lesern geübten Kritik sehr hilfreich.

1 Einleitung

Programmieren zu lernen ist schwer. Wenn noch das Erlernen von Syntax hinzukommt, erscheinen dem Anfänger die Hürden oft unüberwindbar. Syntaxfehler frustrieren und lenken von den eigentlichen Algorithmen, die es zu verstehen gilt, ab. Durch visuelle Programmierung ist es mit Hilfe von Bausteinen, die aufgrund ihrer Form nur in richtiger Art und Weise miteinander verbunden werden können, möglich, Syntaxfehler weitestgehend zu vermeiden (vgl. [Ke 03] S. 9).

Mit dieser Diplomarbeit wird der Versuch unternommen, eine visuelle Programmiersprache für den Einsatz in Schulen zu entwickeln. In einer vorangegangenen Studienarbeit wurden hierfür theoretische Grundlagen gelegt und bestehendes Material wurde gesichtet. Außerdem sind durch Interviews mit Lehrern Anforderungen an eine solche Sprache zusammengetragen worden (vgl. [Ko 04] S. 45-52). Da keines der in der Studienarbeit untersuchten Systeme die Wünsche der Befragten erfüllte[1], sollte eine visuelle Programmiersprache für Programmieranfänger geschaffen werden, die sich für den Einsatz in der Schule eignet.

Hierfür wird im ständigen Kontakt mit Lehrern ein Prototyp implementiert. Während dieser Phase werden die Anforderungen konkretisiert und eventuell verändert. Anschließend soll mit den Methoden der objektorientierten Analyse und des objektorientierten Designs ein System entworfen und in der Programmiersprache Java implementiert werden. Danach erfolgt ein stichprobenartiger Test des Systems.

Die Frage, ob das erstellte Produkt wirklich für den Einsatz in der Schule geeignet ist, kann im Rahmen dieser Arbeit nur ansatzweise diskutiert werden. Es wird aber zumindest ein weiterer Weg aufgezeigt, auf welche Art und Weise die visuelle Programmierung in den Informatikunterricht an den allgemein bildenden Schulen Einzug halten kann.

[1] Eines der größten Probleme ist, dass mit den vorliegenden Systemen nur vereinzelte Teile des Lehrplans erfüllt werden können.

2 Anforderungen

Im Rahmen einer Studienarbeit wurden durch Interviews mit Thüringer Lehrern Anforderungen für eine visuelle Programmiersprache, die in Schulen eingesetzt werden soll, erhoben. Diese werden im Folgenden noch einmal kurz zusammengefasst (vgl. [Ko 04] S. 45-52). Die zu erstellende visuelle Programmiersprache soll:

- per Drag and Drop zu bedienen sein,
- visuell erstellte Programme speichern und laden können,
- Oberon-2- und Pascal-Code generieren,
- zwischen lokalen und globalen Variablen unterscheiden,
- dem Benutzer ermöglichen, selbst Bausteine zu erstellen,
- das Realisieren der im Thüringer Lehrplan angegebenen Algorithmen ermöglichen,
- sowie ein Tutorial und eine Homepage zur Unterstützung anbieten.

Diese Anforderungen sollen im Folgenden als Ausgangspunkt dienen, an dem sich zunächst die Entwicklung des Prototyps orientieren soll.

3 Geplanter Ablauf

Um die Durchführbarkeit des Projektes zu analysieren und erste Erfahrungen mit der Implementierung einer visuellen Programmiersprache zu gewinnen, soll während der ersten Phase der Diplomarbeit ein vertikaler Prototyp entwickelt werden. Dieser soll entsprechend der Sprachdefinition von PL/0 entworfen werden (vgl. [Wi 86] S. 45-49). Für die Verwendung dieser Programmiersprache sprechen verschiedene Gründe:

- Die Sprache ist so einfach gehalten, dass Niklaus Wirth bereits 1970 in seinen Vorlesungen PL/0 - Compiler zu Übungszwecken bauen ließ.
- Andererseits können mit ihr die wichtigsten Grundprinzipien von Programmiersprachen dargestellt werden (vgl. [Wi 86] S. 45).
- PL/0 ist den Sprachen Oberon-2 und Pascal im Aufbau ähnlich, dies ermöglicht eine Wiederverwendung der Elemente des Prototyps.

- Der Verzicht auf verschiedene Datenstrukturen und die Verwendung von parameterlosen Prozeduren vereinfacht den Bau des Prototyps.
- Mit der Programmiersprache PL/0 können für den Schulalltag praxisrelevante einfache Programme wie z. B. die Berechnung des größten gemeinsamen Teilers programmiert werden. Dadurch ist es möglich, auch schon direkt nach der Entwicklung des Prototyps die didaktischen Aspekte der visuellen Programmiersprache zu analysieren und zu testen.

Nach der Fertigstellung des Prototyps sollen die Anforderungen noch einmal überprüft und gegebenenfalls überarbeitet werden.

In der Phase 2 der Diplomarbeit soll anhand der Ergebnisse von Phase 1 eine visuelle Programmiersprache für den Einsatz in Schulen entworfen und implementiert werden. Hierfür sollen die Methoden der objektorientierten Analyse, des objektorientierten Designs und der objektorientierten Programmierung angewendet werden.

In der sich anschließenden Phase 3 wird das entwickelte System mit Lehrern und Schülern kurz getestet werden. Auftretende Probleme sollen nach Möglichkeit gleich behoben werden.

Außerdem werden in dieser Arbeit Fehler beim Erstellen von Programmen kategorisiert und es wird überprüft, welche Fehler durch das entwickelte System verhindert werden können. Dann soll diskutiert werden, inwieweit Anforderungen an eine Programmiersprache für den Anfangsunterricht in Schulen erfüllt worden.

Abschließend soll ein Ausblick für weitere Entwicklungen in Verbindung mit der erstellten visuellen Programmiersprache gegeben werden.

4 Entwicklung eines vertikalen Prototyps

Ein vertikaler Prototyp zeichnet sich dadurch aus, dass einzelne Elemente einer Software funktionell auf allen Ebenen ausprogrammiert sind. (vgl. [Ba 98] S. 116) Gerade durch die Wahl der Programmiersprache PL/0 für den Prototyp ist eine funktionelle Auswahl bereits im Vorfeld getroffen. Die gewählten Sprachelemente sollen vollständig implementiert werden. Sie können dann in Phase 2 der Diplomarbeit in abgewandelter Form wieder verwendet werden.

4.1 Beschreibung des entwickelten Prototyps „Puck“

In Phase 1 der Diplomarbeit wurde, aufbauend auf der Syntax von PL/0, ein Prototyp entwickelt, der bereits eine eingeschränkte Funktionalität der zu erstellenden visuellen Programmiersprache bietet. Um die Ergebnisse der Arbeit besser testen zu können, wurden die entwickelten Programmierbausteine und die Codegenerierung an die Sprache Oberon-2 angepasst. Hierdurch war es möglich, bereits mit dem Prototyp verschiedene Programme zu erstellen und auszuprobieren. Eine große Anzahl von Fehlern konnte dadurch schon frühzeitig erkannt und beseitigt werden.

Der Prototyp wurde in Anlehnung an Shakespeares Sommernachtstraum „Puck“ genannt, denn der Kobold Puck ist in diesem klassischen Stück der kleine Gehilfe des großen Königs der Elfen, Oberon. Über ihn wird gesagt:

„Doch wer dich freundlich grüßt, dir Liebes tut,

Dem hilfst du gern, und ihm gelingt es gut.“ ([Vi 71] S. 62)

Außerdem wurde ein 1985 von der Voyager 2 entdeckter Mond des Planeten Uranus Puck genannt. Dieser ist neben dem zehnmal so großen Uranusmond Oberon[2], nach dem auch die gleichnamige Programmiersprache benannt wurde, einer von 15 Monden des 2,9 Milliarden Kilometer von der Sonne entfernten Planeten, die alle nach Figuren aus Shakespearestücken bezeichnet wurden.

[2] Der Uranusmond Oberon wurde bereits 1787 durch William Herschel entdeckt.

4.2 Die Oberfläche des Prototyps

Auch wenn die Bausteine im Prototyp noch recht eckig und unförmig erscheinen, so ist es doch möglich, mit ihnen Programme zu erstellen. Die verschiedenen Elemente können nur zusammengefügt werden, wenn die Verbindungsstücke formschlüssig ineinander passen. Im Folgenden soll kurz auf die drei Bereiche eingegangen werden, in denen sich die Programmierung in Puck abspielt. Abbildung 1 zeigt die Oberfläche beim Erstellen eines Programms zur Berechung von Potenzen. Bereich 1 wird als Bausteinquelle bezeichnet. In ihm können sich mehrere Container[3] befinden. Dies sind aufklappbare Grafikelemente – hier dunkelgraue Rechtecke – die verschieden Bausteine[4] enthalten.

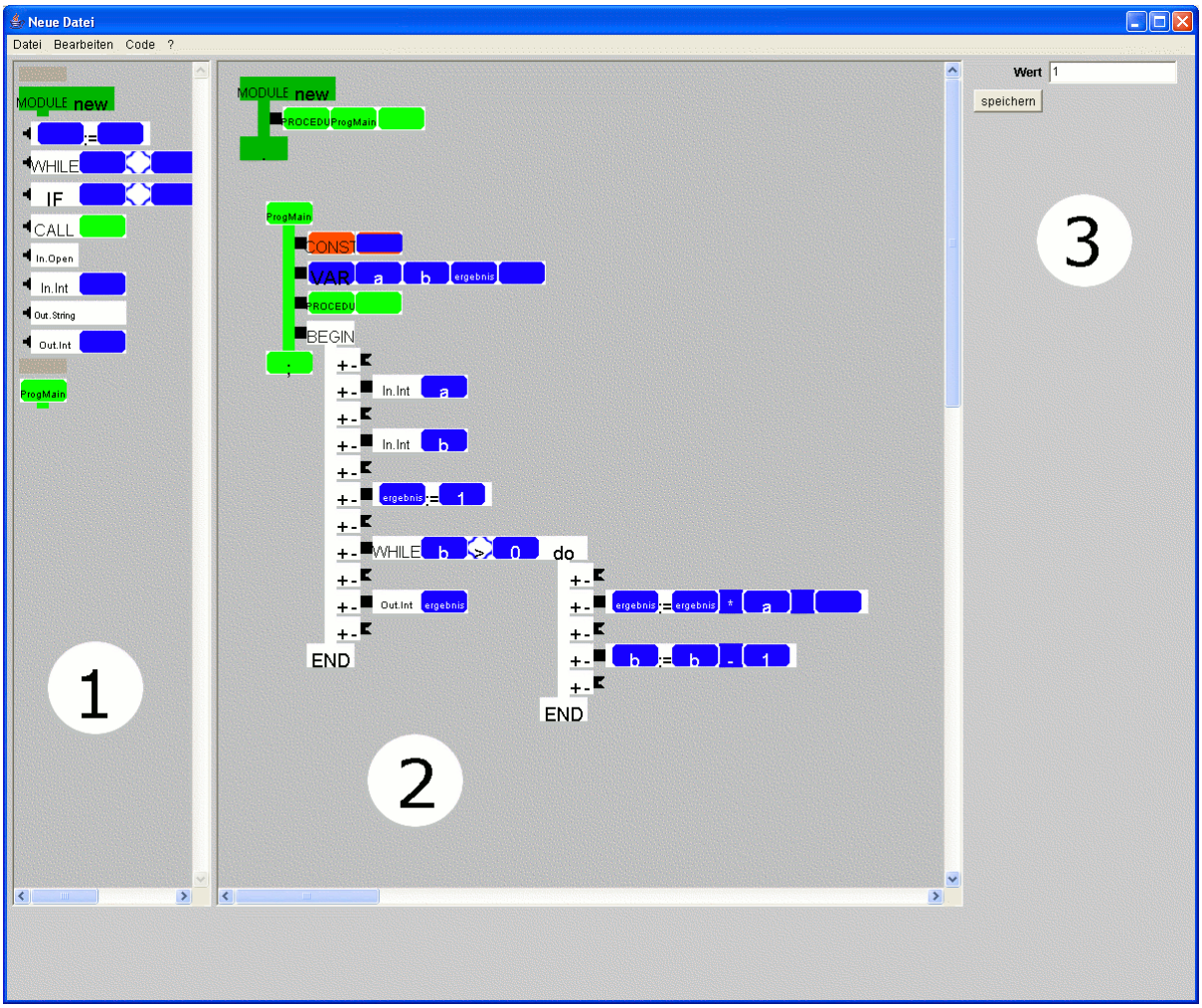


Abbildung 1: Die drei Bereiche der Oberfläche des Prototyps

[3] Mit der Bezeichnung Container sind hier Objekte der Klasse „SourceDrawer“ gemeint (vgl. [Lä 04] S. 26-28).

[4] Mit „Baustein“ werden im Folgenden alle Objekte von Klassen bezeichnet, die das Verhalten von Fragment-ImageContainer erben (vgl. [Lä 04] S. 20-26).

Ein Baustein besteht aus verschiedenen Teilbildern[5] und Anschlussstellen[6] für andere Bausteine. Er ist das Grundelement der Drag and Drop Operationen in der verwendeten Bibliothek, die im Rahmen einer Studienarbeit an der Friedrich-Schiller-Universität Jena entwickelt wurde (vgl. [Lä 04]).

Unter dem ersten Container in Abbildung 1 sind die Moduldefinition sowie die in diesem Prototyp implementierten Anweisungs-Bausteine bereitgestellt. Der zweite Container beinhaltet alle im Programm erstellten Prozeduren. Alle Bausteine aus dem Bereich 1 können mit der Maus in den Arbeitsbereich, der in Abbildung 1 mit einer 2 markiert ist, gezogen werden. Beim Loslassen der Maustaste wird eine Kopie erstellt. Die Bausteine können mit ihren verschiedenen Anschlussstellen miteinander verbunden und somit zu einer vollständigen Prozedur zusammengefügt werden. Durch einen Klick mit der linken Maustaste auf bestimmte Teilbilder des Bausteins wird in Bereich 3 eine Attributtabelle geöffnet. In dieser können Werte für die verschiedenen Attribute eingegeben und gespeichert werden.

Des Weiteren beinhaltet die Oberfläche noch eine Menüleiste. Unter den Elementen „Datei“ wurden Standardfunktionen, wie „Neue Datei“, „Speichern“ und „Laden“ implementiert. Da sich die Funktionalität dieser Menüpunkte nicht von anderen Programmen unterscheidet, sollen diese hier nicht weiter erläutert werden. Der Punkt „Bearbeiten“ enthält keine Implementierung. Unter dem Menüleistenelement „Code“ befindet sich der Punkt „Generiere Oberon Code“, auf dessen Aktivierung sich ein Fenster mit einem Textfeld öffnet. In diesem ist eine textuelle Repräsentation des erstellten Programms in der Programmiersprache Oberon-2 dargestellt. Der so erstellte Code kann markiert, kopiert und anschließend in eine Oberon-2-Quelldatei eingefügt werden. Zum Test der so erstellten Oberon-2-Dateien wurde die Programmierumgebung „POW!“ eingesetzt[7]. Hinter dem letzten Element der Menüleiste verbirgt sich eine Hilfefunktion. Diese ist nicht mit Inhalten gefüllt. Alle Elemente der Menüleiste können auch über „Shortcuts“ erreicht werden.

---

[5] Mit „Teilbilder“ werden im Folgenden alle Objekte von Klassen bezeichnet, die das Verhalten von FragmentImage erben (vgl. [Lä 04] S. 16-20).

[6] Mit „Anschlussstelle“ werden im Folgenden alle Objekte von Klassen bezeichnet, die das Verhalten von FragmentImageInterface erben (vgl. [Lä 04] S. 14-16).

[7] Die Programmierumgebung „POW!“ kann unter [www1] bezogen werden.

4.3 Überarbeitung der Anforderungen

Die Entwicklung des Prototyps zeigte verschiedene Schwierigkeiten sowie Stellen, an denen Erweiterungen möglich waren. Somit war es am Ende von Phase 1 der vorliegenden Arbeit an der Zeit, die gestellten Anforderungen noch einmal zu überprüfen und kritisch zu hinterfragen. Im Gespräch innerhalb der Arbeitsgruppe „Visualisierung im Informatikunterricht Jena“[8] und mit den Betreuern dieser Arbeit sind, in Anlehnung an den Prototyp, folgende neue Anforderungen für das Endprodukt entstanden.

- Die Stärke der Syntaxunterstützung beim Erstellen eines ausführbaren Programms soll einstellbar sein.
    - Bei eingeschalteter Syntaxunterstützung werden statische semantische Fehler verhindert[9]. So werden z. B. verwendete Variablen automatisch umbenannt, wenn der Name in der Deklaration der Variablen verändert wird.
    - Obwohl diese Überprüfung zur semantischen Analyse gezählt wird (vgl. [Cl 03] S. 593), soll sie hier dennoch unter dem Punkt „Syntaxunterstützung“ aufgeführt werden, da eine andere Bezeichnung die Benutzer zu der falschen Annahme verleiten könnte, dass mit Puck nur semantisch korrekte Programme erstellt werden könnten.
- Pascal-Code soll nicht generiert werden[10].
- Eine Projektarbeit, die sich mit der Erstellung einer Hilfe und eines Tutorials beschäftigt, soll für Lehramtsstudenten in der Abteilung für Didaktik der Mathematik und Informatik der Friedrich-Schiller-Universität Jena ausgeschrieben werden.
- Innerhalb zweier Doppelstunden soll die erstellte visuelle Programmiersprache mit Schülern getestet werden.
- Eine kontextsensitive Hilfe soll zur Verfügung gestellt werden.
- Bei eingeschalteter Syntaxunterstützung sollen – SOWEIT MÖGLICH – alle Syntaxfehler sowie das falsche Verwenden von Variablen und Parametern verhindert werden.

---

[8] Diese Arbeitsgruppe beschäftigt sich mit Möglichkeiten der Visualisierung im Informatikunterricht. Sie ist in Zusammenarbeit zwischen der Friedrich-Schiller-Universität Jena, dem ThILLM Bad Berka und verschiedenen Jenaer Gymnasien gegründet wurden.

[9] Eine Klassifizierung von Fehlern wird in Kapitel 7.1 dargestellt.

[10] In Thüringer Schulen wird derzeit überwiegend die Programmiersprache Oberon-2 eingesetzt.

- Die Fehlerbehandlung soll auch Thema der Diplomarbeit sein.
- Die Datentypen Integer und Boolean sollen implementiert werden.
- While-, Repeat- und For-Schleife sollen als Bausteine entwickelt werden.
- Prozeduren mit Parameterübergabe sollen implementiert werden. Hierbei soll zwischen Wert- und Referenzparametern unterschieden werden.

Diese Anforderungen werden nun in Phase 2 dieser Arbeit umgesetzt.

5 Die visuelle Programmiersprache Puck

Die im Rahmen dieser Arbeit erstellte visuelle Programmiersprache „Puck“ soll im Folgenden erläutert werden. Hierfür wird der Leser im Kapitel 5.1 in die Bedienung der verschiedenen Funktionen der Programmiersprache anhand von Beispielen eingeführt. Dabei wird lediglich vorausgesetzt, dass schon einmal in einer imperativen Programmiersprache gearbeitet wurde. Danach soll auf die entwickelten Pakete und Klassen eingegangen werden, um den Lesern die Möglichkeit zu geben, das System zu verstehen, zu verbessern und zu erweitern. Da es kein Ziel dieser Arbeit ist die Begriffswelt der objektorientierten Programmierung zu vermitteln, wird in Kapitel 5.2 vorausgesetzt, dass der Leser mit dieser vertraut ist.

5.1 Die Bedienung von Puck

5.1.1 Die Installation

Puck ist in Java implementiert. Dies hat den Vorteil der Plattformunabhängigkeit, wodurch die erstellte visuelle Programmiersprache auf Linux, Windows und anderen Systemen lauffähig ist, wenn die entsprechende Java-Laufzeitumgebung installiert ist. Aufgrund von Problemen mit der Abwärtskompatibilität der neuesten Java-Versionen wird das Puck-System mit der Laufzeitumgebung jre1.4.2_01 ausgeliefert. Bereits ab Version 1.4.2_06 treten Probleme auf, die wegen des zeitlichen Aufwandes erst nach dieser Arbeit behoben werden können[11].

Installationsanweisung Windows 98, 2000, XP:

1. JRE1.4.2_01 installieren[12].
2. Eventuell Computer neu starten.
3. Die Datei „Puck.jar“ und den Ordner „lib“[13] zusammen an einen beliebigen Ort auf die Festplatte kopieren[14].

---

[11] Das Benutzen von „int.class“ führt in den Drag and Drop Bibliotheken von Java 1.4.2_06 und höher zu einer Exception. Dies kann durch das konsequente Verwenden der Klasse Integer anstatt int verhindert werden.

[12] Dieser Schritt entfällt, wenn auf dem Computer schon eine entsprechende Java-Laufzeitumgebung installiert ist. Die Installationsdatei für java 1.4.2_01 ist auf der beiliegenden CD zu finden. Aktuellere Versionen der Java Laufzeitumgebung sind unter [www7] bereitgestellt.

[13] Im Ordner „lib“ befindet sich die Hilfe zum System.

[14] Der Inhalt der zur Arbeit gehörenden CD ist in Anhang 1 dargestellt. Das System kann auch von CD gestartet werden. Dann werden aber die Optionen nicht abgespeichert und langsame CD-Laufwerke können zu längeren Ladezeiten führen.

4. Die Datei „Puck.jar“ mit einem Doppelklick öffnen[15].
5. Das entwickelte System erzeugt nur Quellcode für die Programmiersprache Oberon-2 und keine ausführbaren Dateien. Deshalb sollte, damit die Programme ausprobiert werden können, auf dem Computer auch das POW!-System – ein Editor mit Compiler für die Sprache Oberon-2 – installiert sein[16].
    - Die POW!-Installationsdateien auf die Festplatte kopieren;
    - POW32_30b.exe starten und den Installationsanweisungen folgen;
    - Den Quelltext der entwickelten Programme als „.mod“-Datei abspeichern;
    - Die abgespeicherte Datei im POW!-System öffnen und mit der Tastenkombination „Strg“ + „F9“ ausführen;

[15] Die Datei ist vom Typ „Executeable Jar File“ und somit unter Windows Systemen direkt zu öffnen. Linux-Benutzer müssen eventuell den Dateityp „.jar“ mit dem Aufruf „java –jar“ verknüpfen.

[16] Die Installationsdateien für POW! sind auf der beiliegenden CD enthalten.

5.1.2 Das erste Programm „Hallo Thüringen“

Nach dem Programmstart öffnet sich die Oberfläche, die in Abbildung 2 dargestellt ist. Auf der linken Seite befindet sich die Bausteinquelle. Hier sind die vordefinierten Anweisungen, wie Input, Output oder Zuweisung, und die Prozeduren, wie die bereits zur Verfügung stehende Startprozedur „ProgMain“, zu finden. Jedes Oberon-2-Programm, das in der Programmieroberfläche POW! entwickelt wird und ausführbar sein soll, besitzt eine öffentliche „ProgMain“-Methode, die beim Abarbeiten als Erstes aufgerufen wird.

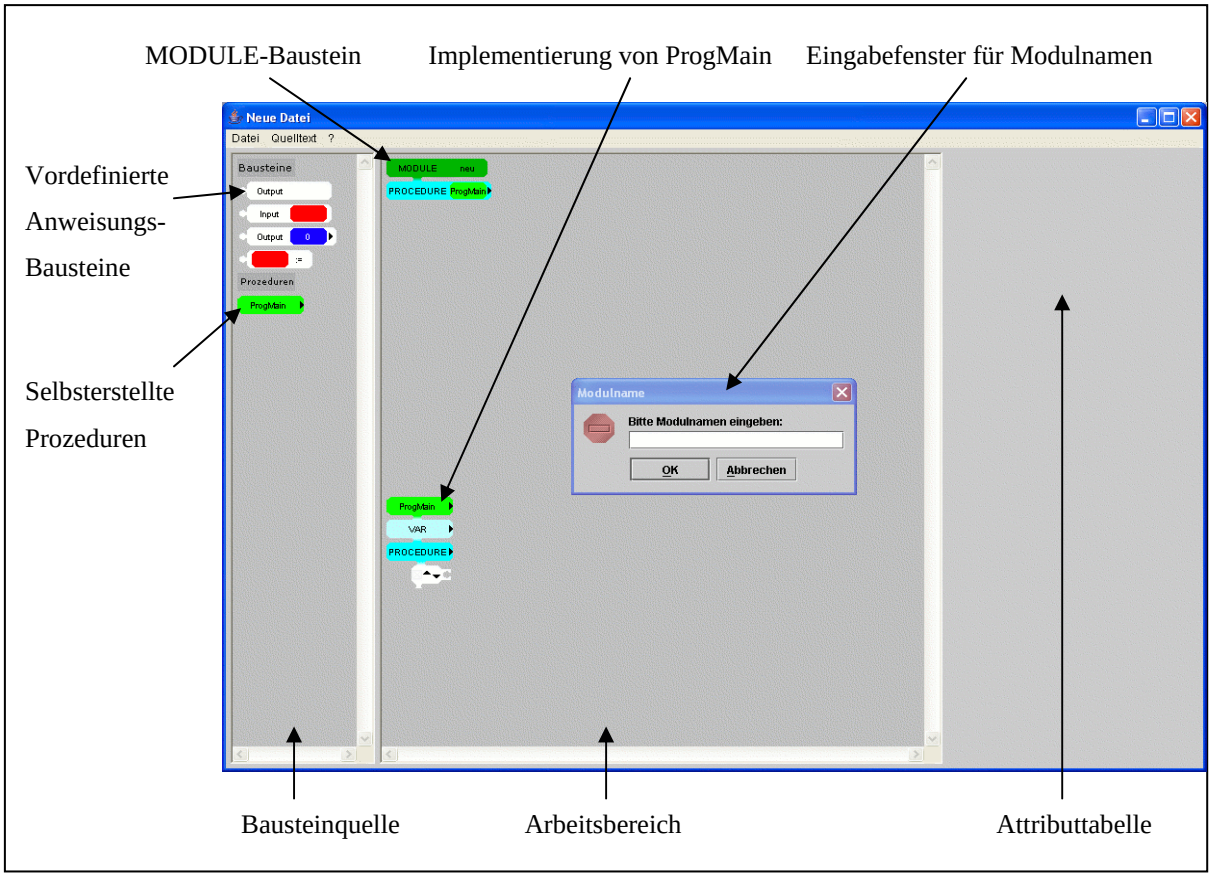


Abbildung 2: Puck beim Programmstart

Direkt nach dem Programmstart wird der Benutzer aufgefordert, einen Modulnamen einzugeben. Somit muss für jedes Programm bereits am Anfang ein Name festgelegt werden, der dann als Modulname, als Dateinamenvorschlag für das Speichern des visuellen Programms und für das Speichern des Quelltextes verwendet wird. Für das Beispielprogramm bietet sich hier „HalloThueringen“ an.

Wenn ein gültiger Oberon-2-Modulname eingegeben wurde, so findet sich dieser rechts von dem Schlüsselwort „MODULE“ in dem dunkelgrünen Baustein wieder. Soll ein neuer Name

eingegeben werden, so muss der Benutzer den alten Namen mit der linken Maustaste anklicken, ihn in der Attributtabelle verändern und mit der Return-Taste oder dem „übernehmen“-Button die Veränderung speichern.

Um nun das erste Programm zu erstellen, muss der vordefinierte Textausgabe-Baustein – der weiß ist und die Aufschrift Output trägt – per Drag and Drop von der Bausteinquelle in den Arbeitsbereich bewegt werden[17]. Jetzt sollte sich eine Kopie des Bausteines im Arbeitsbereich befinden. Diese hat an der linken Seite eine Anschlussstelle, die formschlüssig zu der rechten Seite der Anweisungsfolge an der Implementierung von „ProgMain“ passt. Nun muss der Benutzer den Textausgabe-Baustein im Arbeitsbereich mit der linken Maustaste anklicken, an den Anschlussstelle ziehen und die Maustaste wieder loslassen.

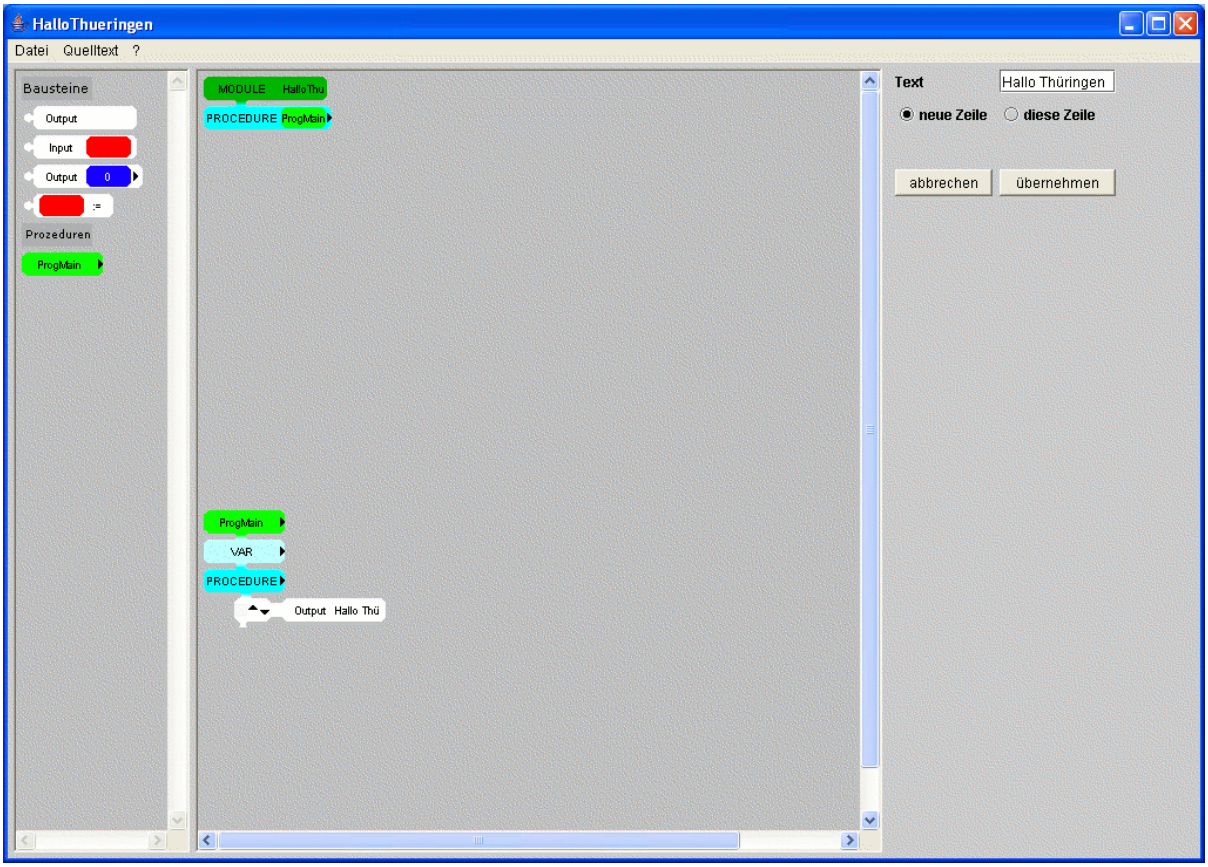


Abbildung 3: Das erste Programm "Hallo Thüringen"

[17] Hierfür muss der Benutzer den Mauszeiger über den Baustein bringen, die linke Maustaste drücken, den Baustein an einen freien Platz im Arbeitsbereich bewegen und die Maustaste wieder lösen. Dieser Vorgang wird im Folgenden mit „Drag and Drop“ bezeichnet.

Um die Ausgabe nun noch mit dem richtigen Text zu füllen, wird der Textausgabe-Baustein mit der linken Maustaste angeklickt. Hierdurch öffnet sich dessen Attributtabelle, in die der Text „Hallo Thüringen“ eingegeben werden kann. Abbildung 3 zeigt das fertige Programm.

Der Benutzer kann sich über den Punkt der Menüleiste „Quelltext“ -> „Quelltext anzeigen“ den Oberon-2-Quelltext in einem Fenster anzeigen lassen. Zum Ausprobieren muss der generierte Quelltext mit dem Menüpunkt „Quelltext“ -> „Quelltext in Datei speichern“ in eine „.mod“-Datei abgespeichert werden. Diese kann dann mit dem POW!-System geöffnet, kompiliert und ausgeführt werden. Der Name der Datei muss dem Modulnamen entsprechen, da das Kompilieren des Programms sonst zu einem Fehler führt. Abbildung 4 zeigt das in POW! erstellte Programm mit der Ausgabe „Hallo Thüringen“.

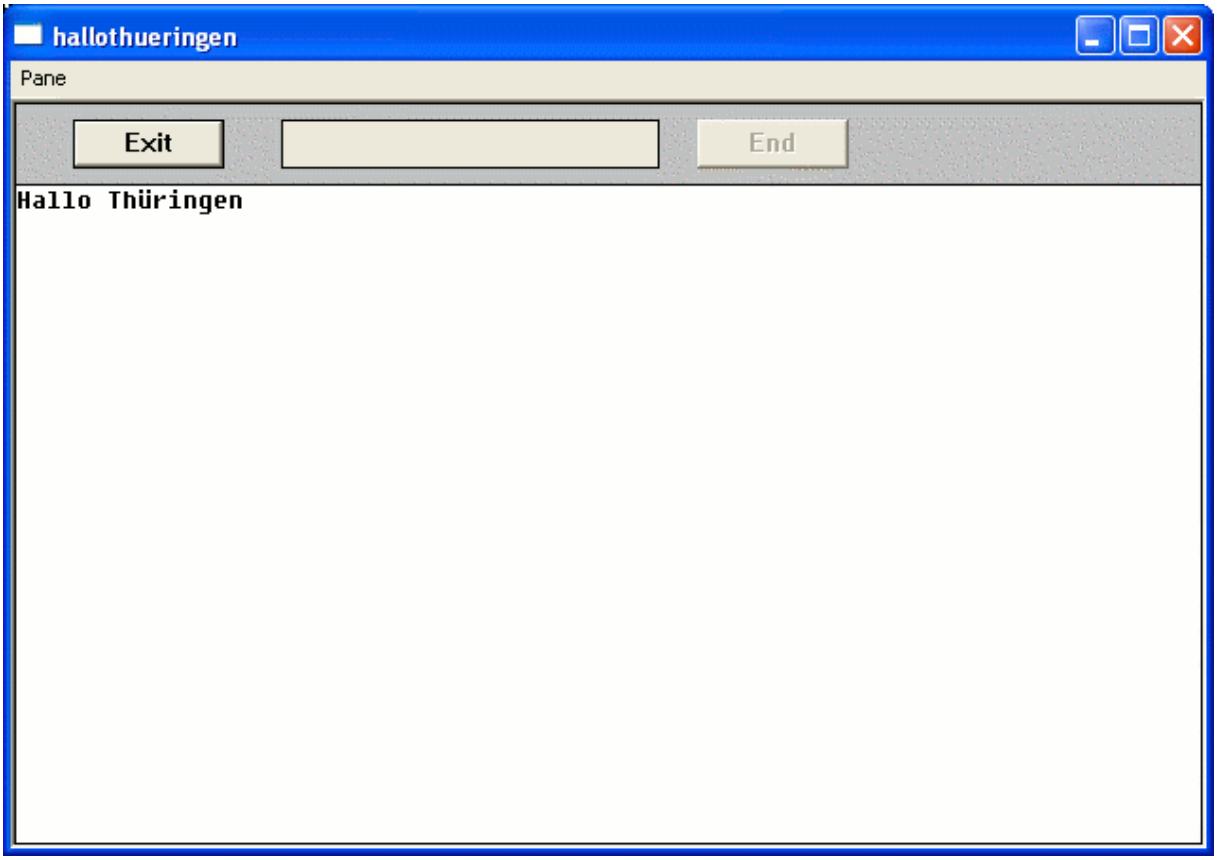


Abbildung 4: Ausgabe des in POW! kompilierten Programms "Hallo Thüringen"

5.1.3 Grundlagen

Um Lehrern das Erklären des Puck-Systems zu vereinfachen, wurde auf Anraten einer Testperson ein Merkblatt erstellt, welches Grundlagen beim Arbeiten mit „Puck“ vermitteln und Fragen, die erfahrungsgemäß bei Anfängern auftreten, beantworten soll[18]. An dieser Stelle soll kurz auf die Punkte des Merkblattes eingegangen werden.

Abbildung 5: Unterscheidung zwischen den Datentypen Integer und Boolean

Wie in Abbildung 5 zu sehen ist, sind Integer-Variablen oder -Werte blau, Boolean-Variablen oder -Werte dunkelgrau dargestellt. Somit kann der Benutzer mit einem Blick sehen, welchen Datentyp eine Variable hat. Dies ist hilfreich, um auch bei komplexeren Programmen den Überblick über die Datentypen der Variablen zu behalten, ohne jeweils zu deren Deklaration scrollen zu müssen.

Abbildung 6: Rot markierte Elemente in Puck

Ein rot markiertes Element in einem Baustein zeigt an, dass noch etwas fehlt. Wenn der Schüler am Anfang auf diesen Sachverhalt hingewiesen wurde, kann er jeweils vor dem Kompilieren eines Programms schnell überprüfen, ob dieses schon vollständig ist.

[18] Das Merkblatt für Anfänger ist in Anhang 2 abgebildet.

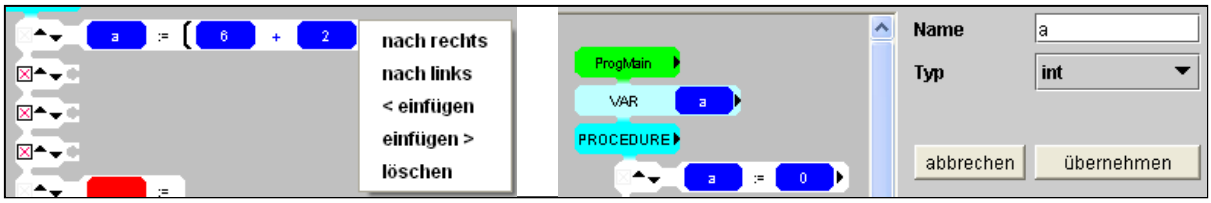


Abbildung 7: Verwendung von Kontextmenüs und Attributtabelle

Mit der rechten Maustaste öffnet man Kontextmenüs, wenn diese verfügbar sind, mit der linken die Attributtabelle eines Bausteins, falls diese vorhanden ist. In der Testphase stellte sich heraus, dass die Benutzer beim Erstellen der ersten Programme oft wussten, ob sie ein Kontextmenü oder eine Attributtabelle erwarteten, aber nicht wussten, welche Maustaste hierfür zu drücken war.

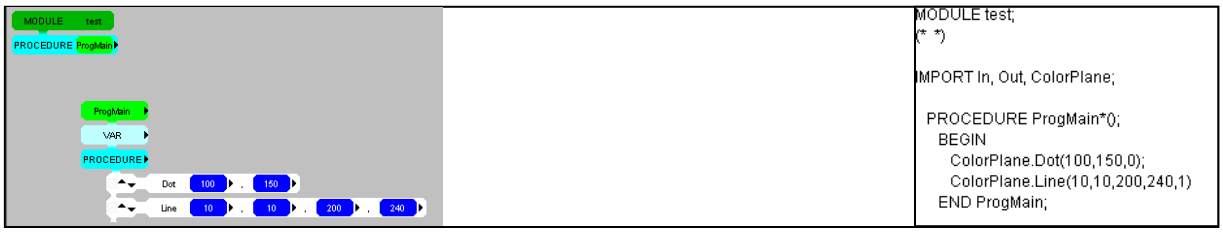


Abbildung 8: Speichern des visuellen Programms und des Quelltextes

Unter „Datei“ kann man das visuelle Programm speichern, unter „Quelltext in Datei speichern“ kann man den Quelltext speichern. Es stellte sich heraus, dass dem Benutzer teilweise nicht klar ist, dass es ein Unterschied ist, das visuelle Programm oder den generierten Quelltext zu speichern. Ungeübte versuchen oft, den generierten Quelltext der abgespeicherten „.mod“-Datei in eine visuelle Repräsentation zu laden. Um Enttäuschungen zu vermeiden, sollte bei der Verwendung von Puck mehrmals darauf hinweisen werden, dass das visuelle Programm unter „Datei“ -> „Speichern“ abgespeichert werden muss, wenn dieses später mit „Datei“ -> „Öffnen“ wiederherstellbar sein soll.

An allen Stellen, an denen Ausdrücke verwendet werden, muss mindestens ein Operand des richtigen Datentyps stehen. Dieser kann nicht gelöscht werden. Wenn der einzige Operand eines Ausdruckes gelöscht werden würde, so wäre der gesamte Ausdruck gelöscht und somit

hätte der Benutzer einen Syntaxfehler produziert. Dies soll aber verhindert werden. Jeder Operand kann immer in alle anderen möglichen Operanden umgeformt werden[19].

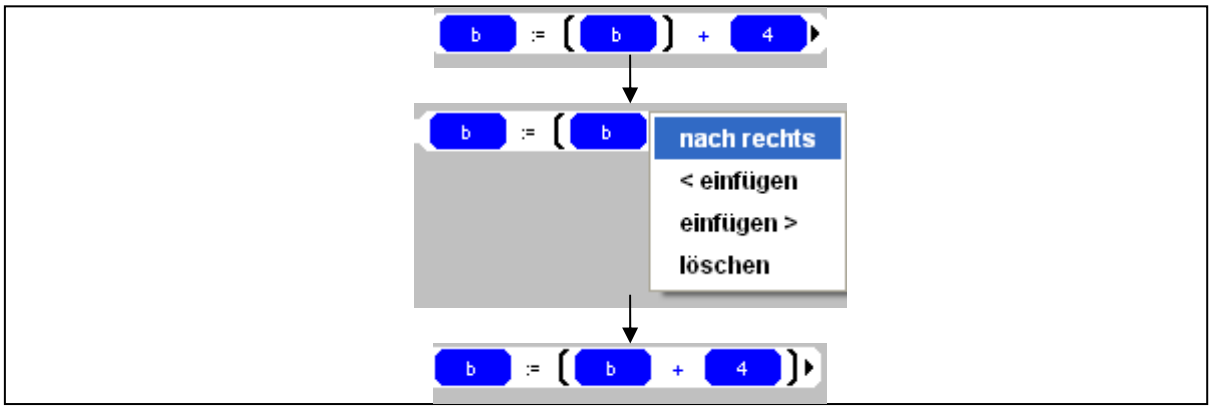


Abbildung 9: Klammersetzung

Klammern werden mit dem Menüpunkt „klammern“ um einen Operanden gesetzt und dann mit Hilfe des Kontextmenüs an die richtige Stelle bewegt. Mit Hilfe dieses Systems entstehen stets Ausdrücke mit korrekter Klammersetzung. Die Menüpunkte „nach rechts“ und „nach links“ erscheinen nur dann, wenn eine Klammer auch wirklich in die entsprechende Richtung bewegt werden kann. Außerdem wird beim Löschen einer linken Klammer die zugehörige rechte mitgelöscht und umgekehrt.

5.1.4 Hilfe

Das Hilfe-System befindet sich zur Zeit der Abgabe dieser Arbeit noch im Anfangsstadium. Im Rahmen einer Projektarbeit der Abteilung für Didaktik der Mathematik und Informatik werden die Hilfe-Seiten zum Puck-System entwickelt. Sobald die Arbeiten abgeschlossen sind, werden diese nachträglich ins System integriert. Eine kontextsensitive Hilfe zu einem bestimmten Baustein kann aktiviert werden, wenn sich die Maus über diesem befindet und die „F1“-Taste gedrückt wird.

[19] Einzige Ausnahme ist der Vergleich zweier Integer-Werte in einem Boolean-Ausdruck. Hier muss, um den Vergleich zu ersetzen, ein weiterer Boolean-Operand eingefügt werden. Anschließend kann der Vergleich gelöscht werden.

5.1.5 Optionen

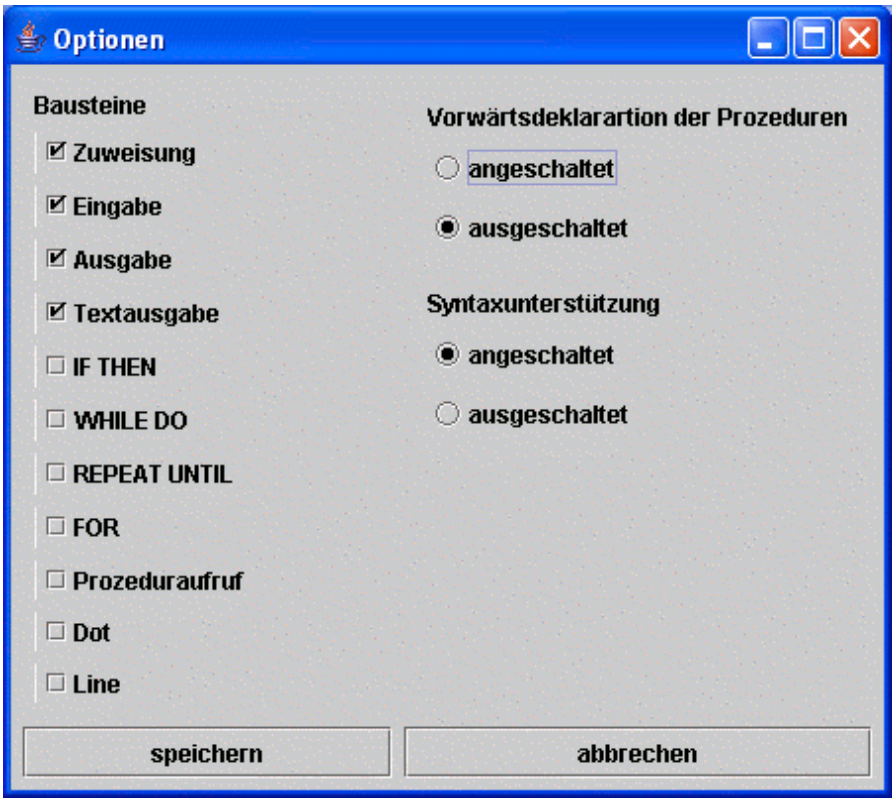


Abbildung 10: Einstellmöglichkeiten in den Optionen

Die Optionen sind über den Menüpunkt „?“ -> „Optionen“ oder die Tastenkombination „Strg“ + „T“ zu öffnen. Das Fenster, welches in Abbildung 10 dargestellt ist, ist in zwei Bereiche unterteilt. Auf der linken Seite kann eingestellt werden, welche Anweisungs-Bausteine in der Bausteinquelle vorhanden sein sollen. Die rechte Seite bietet die Möglichkeit, die Syntaxunterstützung sowie die Vorwärtsdeklaration der Prozeduren ein- oder auszuschalten. Bei aktivierter Syntaxunterstützung wirken sich Veränderungen an der Deklaration von Parametern, Variablen und Prozeduren auch auf die Verwendungen derselben aus. Das heißt, dass sich z. B. die Verwendung einer Variable a, wenn diese den neuen Namen „Ergebnis“ zugeordnet bekommt, auch in „Ergebnis“ wandelt. Ein Ausschalten der Syntaxunterstützung kann eine Herausforderung für Benutzer sein, die schon gut mit dem System vertraut sind, um den Übergang zum textuellen Programmieren vorzubereiten. Bei eingeschalteter Vorwärtsdeklaration wird im Quelltext jede Prozedur am Anfang des umschließenden Blockes deklariert, damit sie dann im gesamten Block bekannt ist.

5.1.6 Variablen

Im Puck-System können nur Integer- und Boolean-Variablen verwendet werden. Auf weitere Datentypen und Konstanten wurde bewusst verzichtet, um den Programmieranfänger nicht zu überfordern. Außerdem können mit ganzen Zahlen und Wahrheitswerten schon viele Aufgaben gelöst werden. Das so gewonnene Grundverständnis kann anschließend mit einer textuellen Programmiersprache vertieft werden.

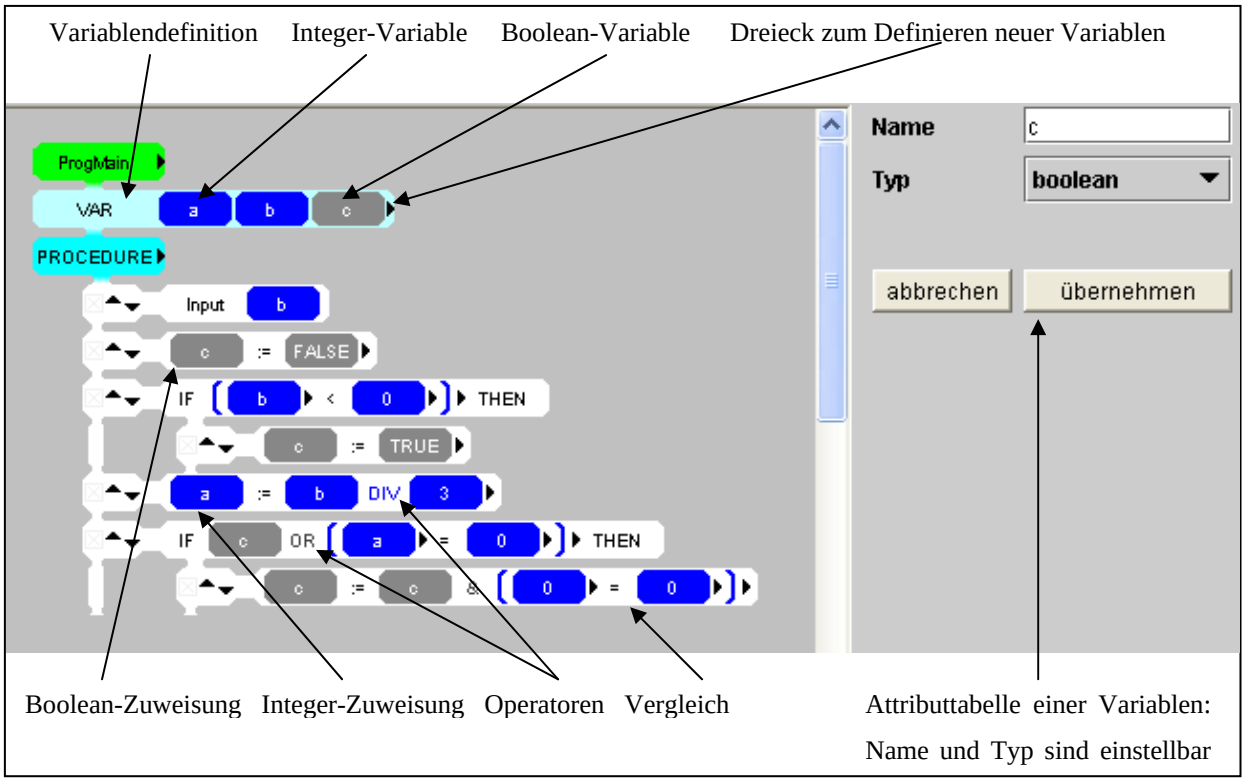


Abbildung 11: Verwendung von Variablen

Variablen können, wie in Abbildung 11 dargestellt, innerhalb von Prozeduren in dem hellblauen VAR-Baustein definiert werden. Immer dann, wenn der Benutzer den schwarzen Pfeil auf der rechten Seite der Variablendefinition mit der linken Maustaste anklickt, wird eine neue Variable eingefügt. Diese hat den Typ Integer und ist mit einem Buchstaben bezeichnet, der im Kontext noch nicht als Name vergeben wurde. Die Attribute „Name“ und „Typ“ können in der Attributtabelle verändert werden. Hierfür muss der Benutzer die entsprechende Variable in der Variablendeklaration anklicken und nach der Veränderung der Werte den „übernehmen“-Button drücken. Wenn die Syntaxunterstützung eingeschaltet ist, verändert sich nicht nur die Variablendefinition, sondern auch jedes Vorkommen der Variablen im Programm. Es kann passieren, dass Operanden in Ausdrücken nach einer Typveränderung ge-

löscht werden, da die Variablen im Kontext ihrer Verwendung mit dem neuen Typ keinen Sinn ergeben würden. Dies ist so gewollt und garantiert, dass im Programm auf diese Weise keine Fehler durch Typkonflikte entstehen können.

5.1.7 Ausdrücke

Ausdrücke werden in Puck, genau wie in anderen Programmiersprachen, an unterschiedlichen Stellen genutzt. So befindet sich auf der rechten Seite einer Zuweisung ein Ausdruck. Ein Vergleich besteht aus zwei einfachen Ausdrücken und einem Vergleichsoperator, eine Bedingung ist ein Ausdruck und auch bei Prozeduraufrufen mit Wertparametern werden diese eingesetzt.

Innerhalb eines Ausdruckes gibt es verschiedene wechselseitig voneinander abhängige Elemente: Operatoren, Variablen und Werte müssen zum jeweiligen Typ passen, verändert und wieder gelöscht werden können. Außerdem gibt es Klammern, die immer korrekt gesetzt sein sollten. Im Folgenden wird die Verwendung von Integer- und Boolean-Ausdrücken erklärt.

5.1.7.1 Integer-Ausdrücke

Ein Integer-Ausdruck ist im einfachsten Fall eine ganze Zahl. Deshalb findet der Benutzer an allen Stellen, an denen er einen Integer-Ausdruck einzusetzen hat, die Zahl 0 vor, die er mit Hilfe der Attributtabelle in jeden anderen zulässigen Wert umformen kann[20]. Wenn er die 0 mit der rechten Maustaste anklickt, öffnet sich, wie in Abbildung 12 dargestellt, ein Kontextmenü, in welchem sich verschiedene Möglichkeiten zur Auswahl bieten.

[20] Der Datentyp Integer umfasst in der Programmiersprache Oberon-2 den ganzzahligen Bereich von -32768 bis +32767.

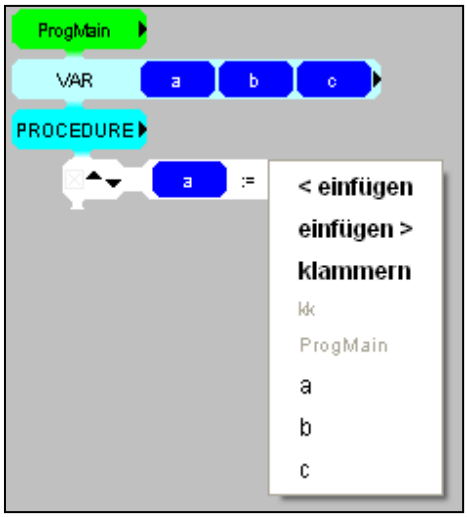


Abbildung 12: Verwendung von Integer-Ausdrücken

Mit „einfügen >“ und „ < einfügen“ kann links bzw. rechts von der 0 ein weiterer Integer-Operand mit einem Operator eingefügt werden. Ein Klick auf den Menüeintrag „klammern“ bewirkt, dass der Operand – in diesem Fall die 0 – in Klammern gesetzt wird. Die weiteren Menüelemente beschreiben übergeordnete Blöcke[21] mit den von ihnen definierten Integer-Variablen und Parametern[22], durch die der Operand ersetzt werden kann. Im Kontextmenü einer Variablen innerhalb eines Integer-Ausdrucks findet sich außerdem noch der Eintrag „Zahl“, durch den der Benutzer die Variable wieder durch eine Zahl ersetzen kann. Besteht ein Ausdruck aus mehreren Operanden, so können diese einzeln mit Hilfe der rechten Maustaste gelöscht werden[23]. Als Operatoren sind nur „+“, „-“, „*“, „DIV“ und „MOD“ zugelassen. Diese könne auch über ein Kontextmenü ausgewählt werden.

5.1.7.2 Boolean-Ausdrücke

Boolean-Ausdrücke bestehen nach ihrer Konstruktion, aus dem Wert „TRUE“ oder „FALSE“. In den Schleifen sind die Werte standardmäßig so gesetzt, dass keine Endlosschleife entsteht, solange am Ausdruck nichts verändert wurde. Ein Kontextmenü zeigt die bekannten Menüeinträge „< einfügen“, „einfügen >“ und „klammern“, die mit der gleichen Funktionalität belegt sind, wie bei den Integer-Ausdrücken.

---

[21] Blöcke sind in einem Kontextmenü hellgrau dargestellt und können nicht ausgewählt werden.

[22] Variablen, Parameter und Prozeduren sind in den Kontextmenüs unterhalb des Blockes angeordnet, in dem sie definiert wurden.

[23] Beim Löschen eines Operanden wird jeweils auch ein Operator gelöscht.

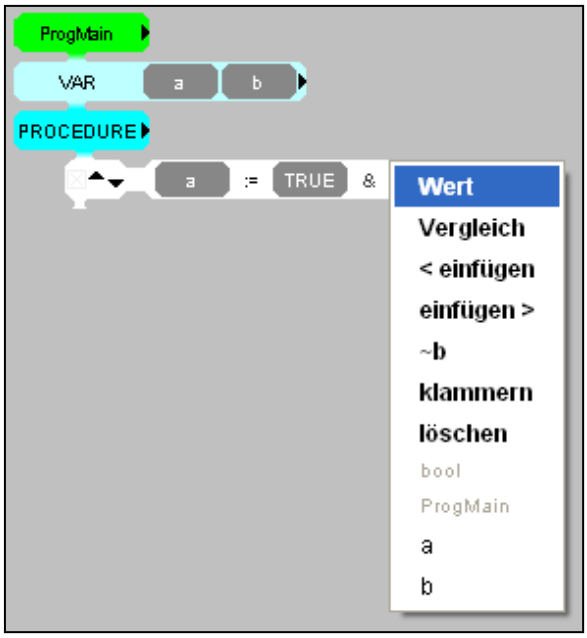


Abbildung 13: Verwendung von Boolean-Ausdrücken

Außerdem kann eine Variable über „Wert“ durch die Konstante „TRUE“ bzw. „FALSE“ ersetzt werden. Der Wert kann mit Hilfe einer Auswahl in der Attributtabelle geändert werden. In Abbildung 13 findet sich auch der Menüeintrag „~b“, der den ausgewählten Operanden „b“ negiert und im erstellten Quelltext in Klammern setzt. Variablen und Parameter werden wie in den Integer-Ausdrücken verwendet. Des Weiteren gibt es im Kontextmenü den Eintrag „Vergleich“. Hiermit kann eine logische Variable oder ein Boolean-Wert durch einen Vergleich zweier Integer-Ausdrücke ersetzt werden[24]. Diese werden dann von einem Operator getrennt und von zwei blauen Klammern umschlossen in den Boolean-Ausdruck eingefügt. Über das Kontextmenü des Vergleichsoperators kann dieser ausgetauscht sowie der gesamte Vergleich gelöscht oder in Klammern gesetzt werden. Die beiden Operatoren „&“ und „OR“ zur Verknüpfung von Boolean-Operanden können über das Kontextmenü ausgewählt werden.

5.1.7.3 Klammern

Klammern werden sowohl in Integer- als auch in Boolean-Ausdrücken im Kontextmenü der Operanden über den Menüpunkt „klammern“ gesetzt. Um nicht nur eine Variable oder einen Wert in Klammern setzen zu können, ist es möglich, Verschiebungen vorzunehmen. Dies geschieht, indem der Benutzer mit der rechten Maustaste auf eine Klammer klickt und den Ein-

[24] Im visuellen Programm, sowie im generierten Quellcode wird ein Vergleich immer in Klammern gesetzt, um ihn als eigenständigen Boolean-Operanden zu kennzeichnen.

trag „nach rechts“ beziehungsweise „nach links“ auswählt[25]. Diese Menüelemente sind nur vorhanden, wenn ein Verschieben der Klammer in die entsprechende Richtung auch zulässig ist. Außerdem ist es möglich, links und rechts von Klammern jeweils einen dem Datentyp entsprechenden Operanden und Operator einzufügen.

5.1.8 Anweisungs-Bausteine

In der Bausteinquelle des Puck Systems gibt es elf Anweisungs-Bausteine, die über Checkboxen im Menü „Optionen“ zu- beziehungsweise weggeschaltet werden können. Somit kann der Lehrer seinen Schülern vorgeben, mit welchen Konstrukten jeweils gearbeitet werden darf. In einer Anfangsstunde reichen vielleicht die Bausteine Eingabe, Ausgabe und Zuweisung für ein einfaches Programm oder Line und FOR für das Erstellen von Fadengrafiken. Später, wenn das Thema Schleifen abgeschlossen ist und Rekursion behandelt werden soll, werden vielleicht nur Prozeduraufruf, IF-Anweisung, Zuweisung, Eingabe und Ausgabe zur Verfügung gestellt. Somit wird der Schüler nicht sofort mit der Komplexität einer gesamten Programmiersprache überlastet, sondern lernt die einzelnen Elemente Schritt für Schritt kennen.

Alle Anweisungs-Bausteine haben an der linken Seite eine puzzleähnliche Anschlussstelle, mit der sie in eine Anweisungsfolge eingehängt werden können. Anweisungsfolgen können mit Hilfe eines Klicks der linken Maustaste auf das nach unten gerichtete Dreieck erweitert werden. Ein weiteres Sequenzelement unterhalb des angeklickten erscheint. Ein Klick auf das nach oben gerichtete Dreieck erstellt ein neues Sequenzelement oberhalb des angeklickten. Ein Klick auf das rote Kreuz löscht das Element. Gibt es in einer Anweisungsfolge nur ein Element oder ist das Element mit einem Baustein verbunden, so ist das rote Kreuz hellgrau dargestellt und ein Löschen unmöglich. Im Folgenden sollen die elf Anweisungs-Bausteine kurz vorgestellt werden.

5.1.8.1 Wertzuweisung

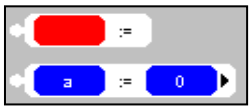


Abbildung 14: Der Wertzuweisungs-Baustein

[25] Abbildung 9 zeigt, wie eine Klammer verschoben werden kann.

Die Wertzuweisung ist ein elementarer Bestandteil einer imperativen Programmiersprache. Mit ihr wird der Variablen auf der linken Seite der Wert des Ausdruckes auf der rechten Seite zugewiesen. Ist die Variable über das entsprechende Kontextmenü eingestellt, so wird auch die rechte Seite der Zuweisung mit einem Ausdruck des richtigen Datentyps initialisiert[26]. Ändert sich der Typ der Variablen, so wird ein neuer Ausdruck mit dem neuen Datentyp erzeugt. Die untere Zuweisung in Abbildung 14 generiert den Quelltext:

```
a:=0
```

5.1.8.2 Eingabe

Abbildung 15: Der Input-Baustein

Mit dem Input-Baustein kann ein Programmierer den Benutzer seines Programms zu einer Eingabe auffordern. Der eingegebene Wert wird dann in einer Variablen gespeichert und kann so weiter verarbeitet werden. Entsprechend des Standardmoduls „In“ können in Oberon-2 vom Benutzer nur Integer-Werte eingegeben werden. Wenn eine Boolean-Eingabe gewünscht ist, so muss diese mit Hilfe einer Integer-Eingabe emuliert werden. Da eine Variable, in der der eingegeben Wert gespeichert werden soll, angegeben werden muss, ist das entsprechende Feld rot gekennzeichnet. Ist der Input-Baustein mit der Anweisungsfolge einer Prozedur verknüpft, so können im Kontextmenü alle lokal sichtbaren Integer-Variablen und Parameter ausgewählt werden. Um dem Benutzer des Programms zu verdeutlichen, was an der jeweiligen Stelle im Einzelnen gewünscht ist, gibt es eine Beschreibung des einzugebenden Wertes, die nach einem Klick mit der linken Maustaste auf das Wort „Input“ in der Attributtabelle eingegeben werden kann. Der generierte Oberon Quelltext für die untere Input-Anweisung in Abbildung 15 lautet:

```
In.Prompt('ganze Zahl:');
In.Int(a)
```

[26] Variablen können nur dann in eine Zuweisung eingesetzt werden, wenn diese mit einer Prozedur verknüpft ist, weil Prozeduren, Parameter und Variablen immer nur innerhalb eines Kontextes bekannt sind.

5.1.8.3 Ausgabe

Abbildung 16: Der Output-Baustein

Der Output-Baustein gibt dem Benutzer den Wert eines Integer-Ausdrucks auf dem Bildschirm aus. Ein Klick auf das Wort „Output“ öffnet die Attributtabelle, in die noch eine Beschreibung eingegeben werden kann, welche dann vor dem Ausdruck ausgegeben wird. Wenn der Baustein mit der Anweisungsfolge einer Prozedur verknüpft ist, kann der Ausdruck, wie in Kapitel 5.1.7.1. beschrieben, verändert werden. Der untere Output-Baustein in Abbildung 16 generiert den Quelltext:

```
Out.String('Das Ergebins ist: ');
Out.Int(a MOD 3,0)
```

5.1.8.4 Textausgabe

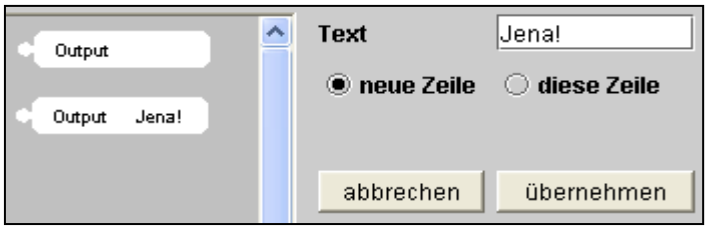


Abbildung 17: Der Textausgabe-Baustein

Mit dem Textausgabe-Baustein ist es dem Programmierer möglich, dem Benutzer seines Programms einen Text auf dem Bildschirm auszugeben. Dieser ist über die Attributtabelle einzugeben. Des Weiteren kann noch mittels der Auswahl „neue Zeile“ ein Zeilenvorschub bewirkt werden. Die ersten Buchstaben des auszugebenden Textes werden hinter dem Wort „Output“ angezeigt. Der untere Textausgabe-Baustein in Abbildung 17 generiert den Quelltext:

```
Out.String('Jena!');
Out.Ln
```

5.1.8.5 IF-Anweisung

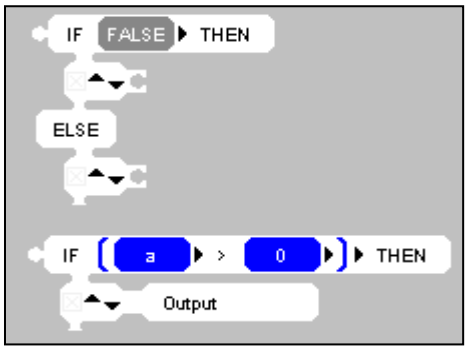


Abbildung 18: Der IF-THEN-Baustein

Der IF-THEN-Baustein beschreibt eine Verzweigung im Programm. Je nachdem, ob die Bedingung, welche durch einen Boolean-Ausdruck beschrieben wird, wahr oder falsch ist, wird die Anweisungsfolge des THEN-Zweiges oder die des ELSE-Zweiges abgearbeitet. Wie in Abbildung 18 unten zu sehen ist, kann in der Bedingung auch ein Vergleich zweier Integer-Ausdrücke eingesetzt werden. Über das Kontextmenü der Wörter IF, THEN und ELSE kann der ELSE-Zweig aus- beziehungsweise eingeschaltet werden. Die Anweisungsfolgen können erweitert werden. Das untere Beispiel in Abbildung 18 generiert den Quelltext:

```
IF (a  >  0) THEN
  Out.String('');Out.Ln
END
```

5.1.8.6 REPEAT-Anweisung

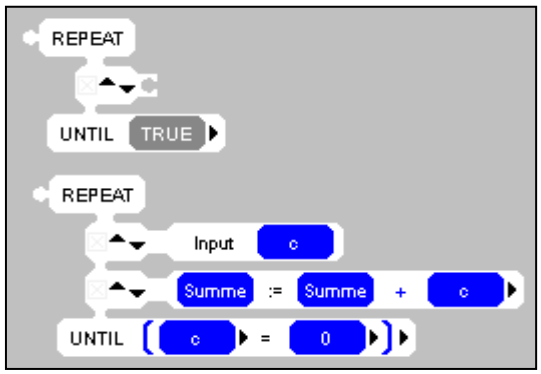


Abbildung 19: Der REPEAT-UNTIL-Baustein

Bei der REPEAT-Schleife wird eine Anweisungsfolge mindestens einmal ausgeführt. Ist die nachgestellte Bedingung nicht erfüllt, so wird die Anweisungsfolge wieder ausgeführt und so weiter. Das untere Beispiel in Abbildung 19 zeigt eine REPEAT-Anweisung, mit deren Hilfe

eine Folge von Zahlen eingelesen und aufsummiert wird. Erst wenn eine Null eingegeben wurde, bricht die Schleife ab. Der erzeugte Quelltext hierfür ist:

```
REPEAT
  In.Int(c);
  Summe:=Summe+c
UNTIL (c = 0)
```

5.1.8.7 WHILE-Anweisung

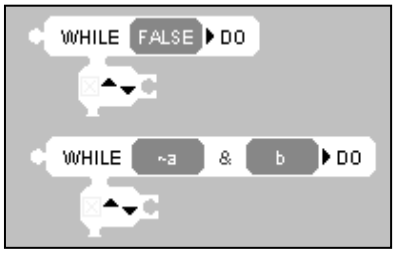


Abbildung 20: Der WHILE-DO-Baustein

Der WHILE-DO-Baustein beschreibt eine Schleife, bei der die Anweisungen des Rumpfes wiederholt werden, solange die vorangestellte Bedingung wahr ist. Die untere WHILE-DO-Anweisung in Abbildung 20 erzeugt folgenden Quelltext:

```
WHILE ~a & b DO
END
```

5.1.8.8 FOR-Anweisung

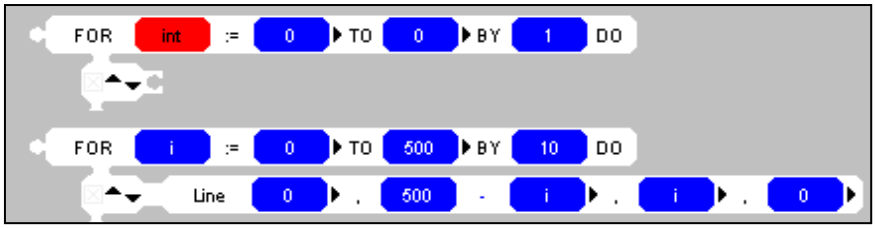


Abbildung 21: Der FOR-Baustein

Soll der Wert einer Laufvariablen von einem Anfangswert bis zu einem Endwert in jedem Schleifendurchlauf erhöht bzw. verringert werden, dann ist es sinnvoll, den FOR-Baustein zu verwenden. Der Programmierer muss an der rot gekennzeichneten Stelle eine Integer-Variable einsetzen, die dann bei jedem Schleifendurchlauf um die Zahl nach BY hochgezählt wird. Diese kann über die Attributtabelle verändert werden. Der Startwert kann in einem Integer-

Ausdruck links, der Endwert rechts des Wortes „TO“ angegeben werden. Die zweite FOR-Schleife in Abbildung 21 würde also 51-mal durchlaufen, wobei die Variable i dabei die Werte 0, 10, 20, 30, 40, 50, …470, 480, 490, 500 annimmt. Hierfür wird der folgende Quelltext generiert:

```
FOR i:=0 TO 500 BY 10 DO
  ColorPlane.Line(0,500-i,i,0,1)
END
```

5.1.8.9 Prozeduraufruf

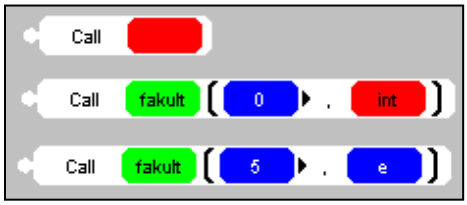


Abbildung 22: Der Prozeduraufruf-Baustein

Für Prozeduraufrufe wird der Baustein mit der Beschriftung „Call“ benutzt. Das rot markierte Feld zeigt dem Benutzer, dass der Prozeduraufruf-Baustein unvollständig ist. Hier muss über das Kontextmenü noch die Prozedur, die aufgerufen werden soll, ausgewählt werden. Hat diese Parameter, so stehen jene, genau wie bei der textuellen Programmierung, hinter dem Prozedurnamen, durch Kommas getrennt, in Klammern. Für Parameter mit Wertübergabe sind Ausdrücke des entsprechenden Datentyps voreingestellt. Parameter mit Referenzübergabe sind durch ein rotes Feld markiert, welches signalisiert, dass an dieser Stelle noch eine Variable des richtigen Typs angegeben werden muss.

Verändert der Benutzer die Signatur[27] einer Prozedur bei eingeschalteter Syntaxunterstützung, so verändert sich auch der Prozeduraufruf. Dementsprechend müssen die Werte der hinzugekommenen Parameter, nach einer Veränderung der aufgerufenen Prozedur, eingestellt werden.

Wenn innerhalb einer Prozedur eine weitere aufgerufen wird, die erst nach dieser definiert ist – dies kann bei falscher Reihenfolge der Prozedurdeklaration oder bei sich gegenseitig aufrufenden Prozeduren passieren – dann würde dieser Aufruf zu einem Syntaxfehler führen. Um dies zu vermeiden, gibt es in Oberon-2 die Vorwärtsdeklaration (vgl. [Mü 95] S. 105). Wenn

[27] Die Signatur besteht aus Anzahl und Typ der Parameter sowie evtl. dem Rückgabetyp einer Prozedur.

diese Option eingeschaltet ist, wird im Quelltext der entsprechenden Code generiert. Der untere Prozeduraufruf aus Abbildung 22 erstellt den Quelltext:

```
fakult(5,e)
```

5.1.8.10 Dot

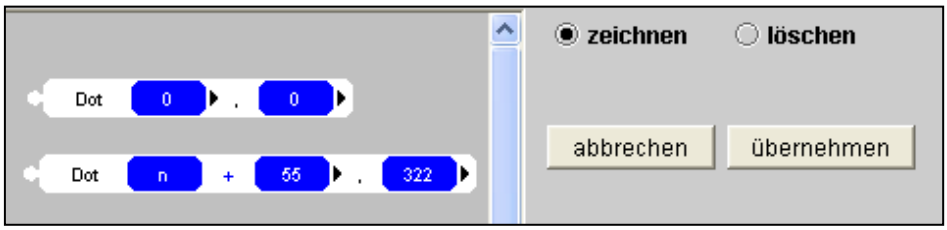


Abbildung 23: Der Dot-Baustein

Der Dot-Baustein zeichnet oder löscht einen Punkt im Ausgabefenster. Die Koordinaten, die dafür benötigt werden, können in Integer-Ausdrücken angegeben werden. In der Attributtabelle kann der Benutzer wählen, ob an der angegebenen Stelle ein Punkt gezeichnet oder gelöscht werden soll. Mit Hilfe dieses Befehls sind grafische Ausgaben möglich und der Benutzer kann sich beispielsweise mathematische Funktionen in einem Koordinatensystem Punkt für Punkt anzeigen lassen. Die Verwendung von Dot impliziert einen automatischen Import des ColorPlane-Moduls im Quelltext, welches für die grafische Ausgabe verantwortlich ist. Der generierte Quelltext für den unteren Dot-Baustein in Abbildung 23 ist[28]:

```
ColorPlane.Dot(n+55,322,1)
```

[28] In der Dot-Anweisung wird dem dritten Parameter eine 1 übergeben. Dies bedeutet, dass der Punkt gezeichnet werden soll. Wenn in der Attributtabelle „löschen“ ausgewählt wurde, wird an dieser Stelle eine 0 übergeben.

5.1.8.11 Line

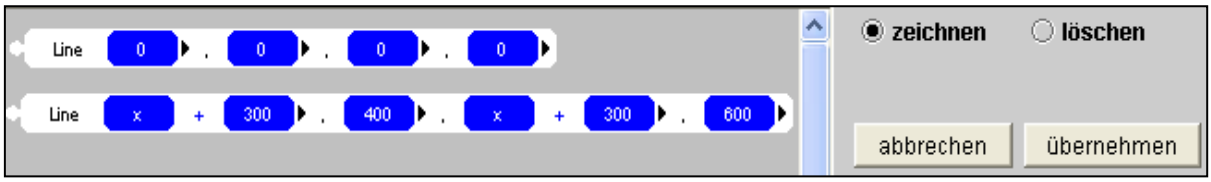


Abbildung 24: Der Line-Baustein

Mit dem Line-Baustein ist es dem Puck-Benutzer möglich, eine Linie auszugeben. Hierfür müssen Anfangs- und Endpunkt – jeweils durch x- und y-Koordinate – mit Hilfe von vier Ausdrücken angegeben werden. Außerdem kann wie beim Dot-Baustein über die Attributtabelle eingestellt werden, ob die Linie gezeichnet oder gelöscht werden soll. Auch die Verwendung des Line-Bausteins impliziert einen Import des ColorPlane-Moduls. Die zweite Linie in Abbildung 24 generiert den Quelltext[29]:

```
ColorPlane.Line(x+300,400,x+300,600,1)
```

5.1.9 Das Modul

In Oberon ist ein Modul eine Sammlung von Deklarationen für Konstanten, Typen, Variablen und Prozeduren mit einer initialisierenden Anweisungsfolge (vgl. [Mü 95] S. 105-110). Außerdem kann ein Modul die exportierten Teile eines anderen Moduls importieren. Um dieses komplexe Gebilde didaktisch sinnvoll zu reduzieren, können in einem Modul-Baustein in Puck lediglich Prozeduren deklariert werden. Dies reicht für einen Anfänger zum Erlernen imperativer Programmierung aus.

Abbildung 25: Verwendung eines Moduls

In Abbildung 25 ist ein dunkelgrüner Modul-Baustein mit der türkisfarbenen Prozedurdeklaration dargestellt. Rechts neben dem Schlüsselwort „MODULE“ befindet sich der Modulname. Dieser kann nach einem Klick mit der linken Maustaste in der Attributtabelle verändert

[29] In der Line-Anweisung wird dem fünften Parameter eine 1 übergeben. Dies bedeutet, dass die Linie gezeichnet werden soll. Wenn in der Attributtabelle „löschen“ ausgewählt wurde, wird an dieser Stelle eine 0 übergeben.

werden. Außerdem kann an dieser Stelle noch eine Beschreibung des Programms, die sich im Quelltext als Kommentar wiederfindet, angegeben werden. Der um die Endung „.mod“ erweiterte Modulname wird dem Benutzer beim Speichern des Quelltextes als Dateiname vorgeschlagen. Diesem Vorschlag sollte gefolgt werden, denn ein Oberon-2-Programm ist in POW! nur dann ausführbar, wenn Datei- und Modulname übereinstimmen. Jedes Programm in Puck besitzt genau ein Modul, welches auch nicht gelöscht werden kann. In der Prozedurdeklaration können mit Hilfe des schwarzen Pfeils neue Prozeduren erstellt, über das Kontextmenü alte Prozeduren gelöscht und über die Attributtabelle bestehende Prozeduren verändert werden.

In Abbildung 25 wurde zu der öffentlichen Standardprozedur „ProgMain“, die den Einstiegspunkt in jedes Oberon-2-Programm in der Programmierumgebung POW! darstellt, noch eine weitere mit dem Namen „hanoi“ hinzugefügt. In der Attributtabelle einer Prozedur kann mit der Auswahl zwischen „öffentlich“ und „privat“ eingestellt werden, ob diese vom Modul exportiert werden soll. Parameter werden erst bei der Implementierung einer Prozedur angegeben. Um Verwechselungen und Problemen bei der Verwendung von Prozeduren vorzubeugen, kann der Benutzer in Puck nur Prozeduren mit unterschiedlichen Namen erzeugen. Die „ProgMain“-Prozedur ist bei einem neuen Programm schon vorhanden und weitere Prozeduren werden immer links von ihr eingefügt. Somit wird „ProgMain“ immer als letztes deklariert und kennt deshalb alle vor ihr implementierten Prozeduren auch ohne eingeschaltete Vorwärtsdeklaration. Das in Abbildung 25 dargestellte Modul generiert folgenden Quelltext:

```
MODULE TuermeHanoi;
(*  *)
IMPORT In, Out;
  PROCEDURE hanoi*();
    BEGIN
    END hanoi;
  PROCEDURE ProgMain*();
    BEGIN
    END ProgMain;
END TuermeHanoi.
```

5.1.10 Prozeduren

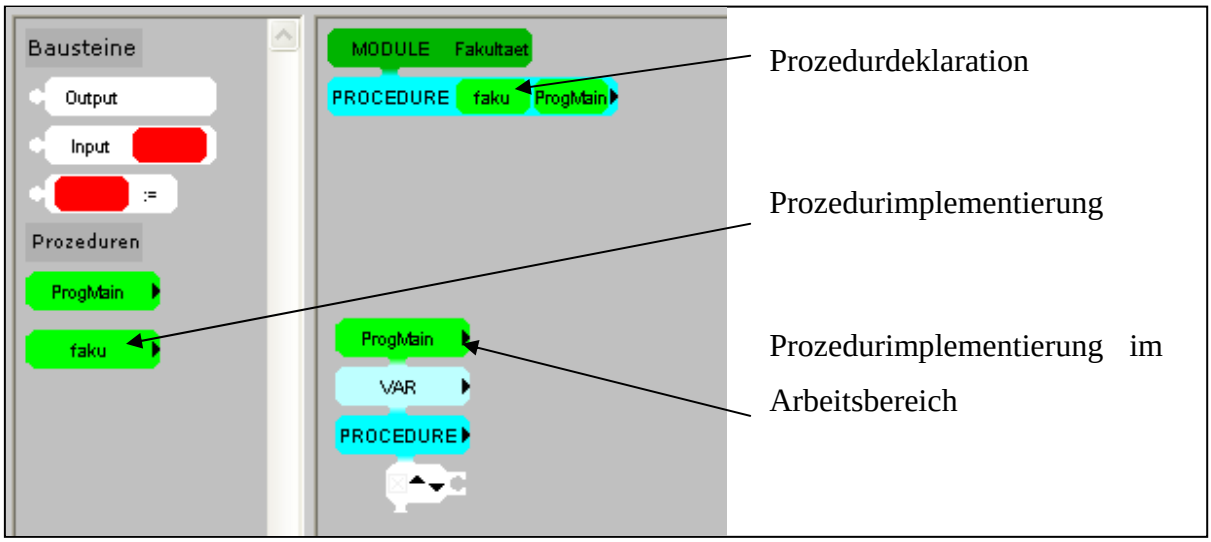


Abbildung 26: Deklaration von Prozeduren

Prozeduren können wie in Abbildung 26 in einem Modul oder in einer Prozedur deklariert werden. Sie erscheinen dann in der Bausteinquelle unter dem zweiten Container hellgrün. Möchte der Benutzer die Prozedur implementieren, so muss er sie mittels Drag and Drop in den Arbeitsbereich ziehen. Jede Prozedurimplementierung kann im Gegensatz zu den Anweisungs-Bausteinen nur einmal im Arbeitsbereich vorhanden sein. Wenn nach dem Programmstart ein Modulname eingegeben wurde, so befindet sich außer dem entsprechenden Modul-Baustein auch die Prozedur „ProgMain“ im Arbeitsbereich, da diese in jedem Programm, das ausführbar sein soll, implementiert sein muss.

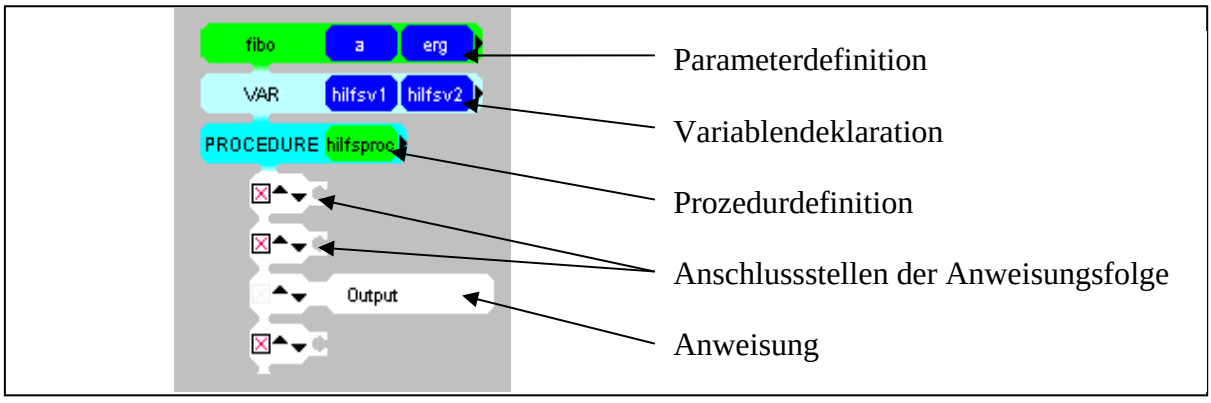


Abbildung 27: Verwendung von Prozeduren

Die Implementierung einer Prozedur beinhaltet den Namen der Prozedur auf einem hellgrünen Untergrund. Hier können durch das kleine Dreieck auf der rechten Seite auch Parameter definiert werden. Die Variablendeklaration befindet sich auf hellblauem Grund direkt unterhalb des Prozedurnamens. Auch Variablen werden bereitgestellt, indem der Benutzer mit der

linken Maustaste das entsprechende schwarze Dreieck anklickt. Damit direkt nach einer Deklaration das Programm noch syntaktisch korrekt ist, bekommen Variablen und Parameter einen Kleinbuchstaben als Namen zugeordnet. Außerdem können innerhalb einer Prozedurimplementierung noch weitere Prozeduren definiert werden. Dies funktioniert ähnlich zum Modul-Baustein, mit der Ausnahme, dass die hier deklarierten Prozeduren nicht exportiert werden können. Unterhalb der Prozedurdefinition befindet sich eine Anweisungsfolge.

Alle in einer Prozedur deklarierten Elemente, hiermit sind Parameter, Variablen und Prozeduren gemeint, können über das Kontextmenü, das mit der rechten Maustaste erreichbar ist, gelöscht werden. Für die Prozedur in Abbildung 27 wird folgender Quelltext generiert:

```
PROCEDURE fibo*(a: INTEGER; VAR erg: INTEGER);
    VAR
      hilfsv1: INTEGER;
      hilfsv2: INTEGER;
    PROCEDURE hilfsproc();
      BEGIN
      END hilfsproc;

    BEGIN
      Out.String('');Out.Ln
    END fibo;
```

5.1.11 Parameter

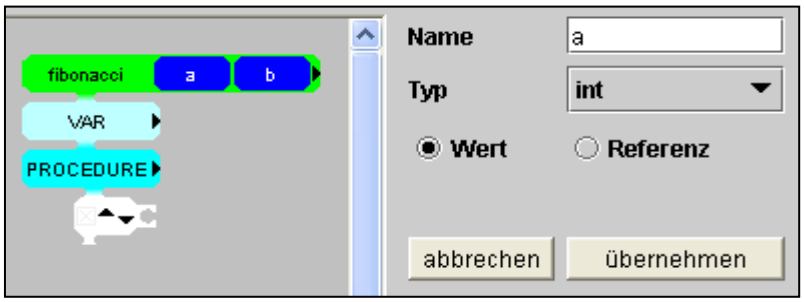


Abbildung 28: Deklaration von Parametern

In einer Prozedurimplementierung können mit Hilfe des schwarzen Dreiecks – rechts vom Namen – Parameter definiert werden. Diese werden fortlaufend mit Kleinbuchstaben bezeichnet. Wie in Abbildung 28 zu sehen ist, können über einen Klick auf die Parameter in der Attributtabelle Name, Typ und Übergabemechanismus verändert werden. Wertparametern wird beim Prozeduraufruf ein Ausdruck, Referenzparametern eine Variable des entsprechenden Typs übergeben.

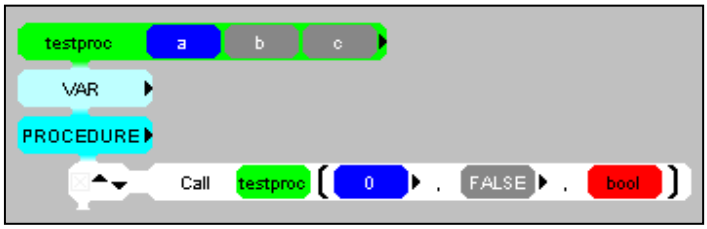


Abbildung 29: Verwendung von Parametern

In Abbildung 29 ist die Implementierung einer unvollständigen Prozedur veranschaulicht, die sich selbst aufruft. Die drei verwendeten Parameter sollen kurz erläutert werden: Wie an den Farben zu sehen ist, ist die Variable a vom Datentyp Integer, b und c sind vom Typ Boolean. Der Prozeduraufruf enthält zwei Ausdrücke für die Wertparameter a und b. Das rote Feld an der Stelle des dritten Parameters zeigt an, dass hier eine Variable vom Typ Boolean eingesetzt werden muss, da c als Referenzparameter übergeben wird. Für die Prozedur in Abbildung 29 wird folgender Quelltext generiert:

```
PROCEDURE testproc*(a: INTEGER; b: BOOLEAN; VAR c: BOOLEAN);
  BEGIN
    testproc(0,FALSE,bool)
  END testproc;
```

5.1.12 Beispielprogramme

Im Folgenden werden einige Programme vorgestellt, die die Möglichkeiten des Puck-Systems zum Ausdruck bringen sollen. Hierbei wird jeweils kurz auf die grafische Repräsentation und die Ausgabe des Programms eingegangen. Der von Puck generierte Quelltext der Programme ist in Anhang 3 dargestellt.

5.1.12.1 Fibonacci-Zahlen

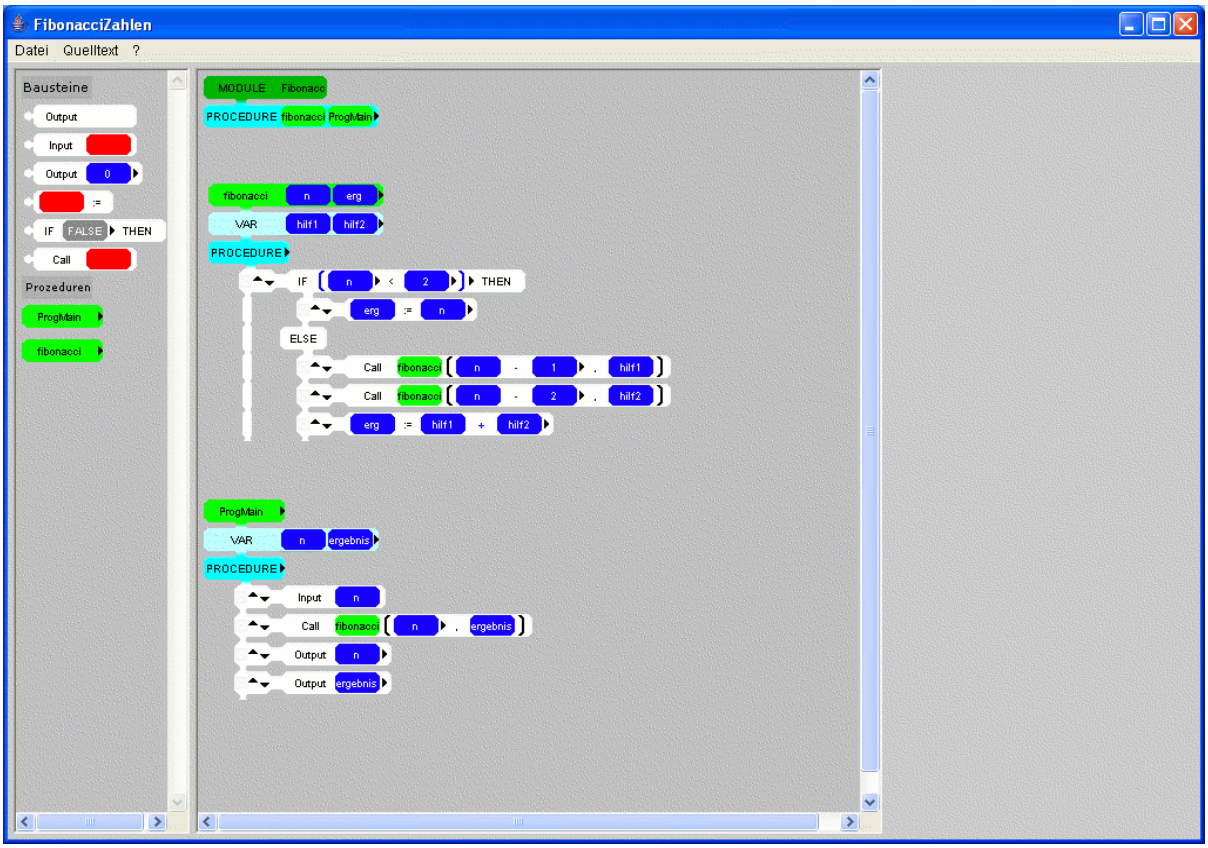


Abbildung 30: Das visuelle Beispielprogramm „Fibonacci-Zahlen“

Die Berechnung der Fibonacci-Zahlen ist ein beliebtes Beispiel zur Einführung der Rekursion im Informatikunterricht[30]. Das Programm in Abbildung 30 liest eine Zahl n ein, berechnet mit Hilfe der im Modul definierten Prozedur „fibonacci“ die n-te Fibonacci-Zahl und gibt dann beide Zahlen mit einem entsprechenden Text aus. Für diese Prozedur wurden zwei Parameter definiert. Der Wertparameter „n“ gibt an welche Fibonacci-Zahl berechnet werden soll und in dem Referenzparameter „erg“ wird das Ergebnis der Berechnung zurückgegeben. Um die

[30] Die Folge der Fibonacci-Zahlen ist eines der sieben Beispiele für rekursive Funktionen in ([Fo 02a] Kapitel 1.11), die alle mit der visuellen Programmiersprache Puck umgesetzt werden können.

Abbruchbedingung der Fibonacci-Zahlen fib(0) = 0 und fib(1) = 1 zu implementieren wird, wenn das übergebene n kleiner als 2 ist, der Wert von n selbst als Ergebnis festgelegt. Ansonsten ergibt sich das Ergebnis aus der Summe der beiden Hilfsvariablen „hilf1“ und „hilf2“, in denen die (n-1)-te und (n-2)-te Fibonacci-Zahl rekursiv berechnet wurden. Abbildung 31 zeigt die Ausgabe des mit POW! kompilierten Programms bei der Eingabe der Zahl 20. Der generierte Quelltext ist in Anhang 3.1 abgebildet.

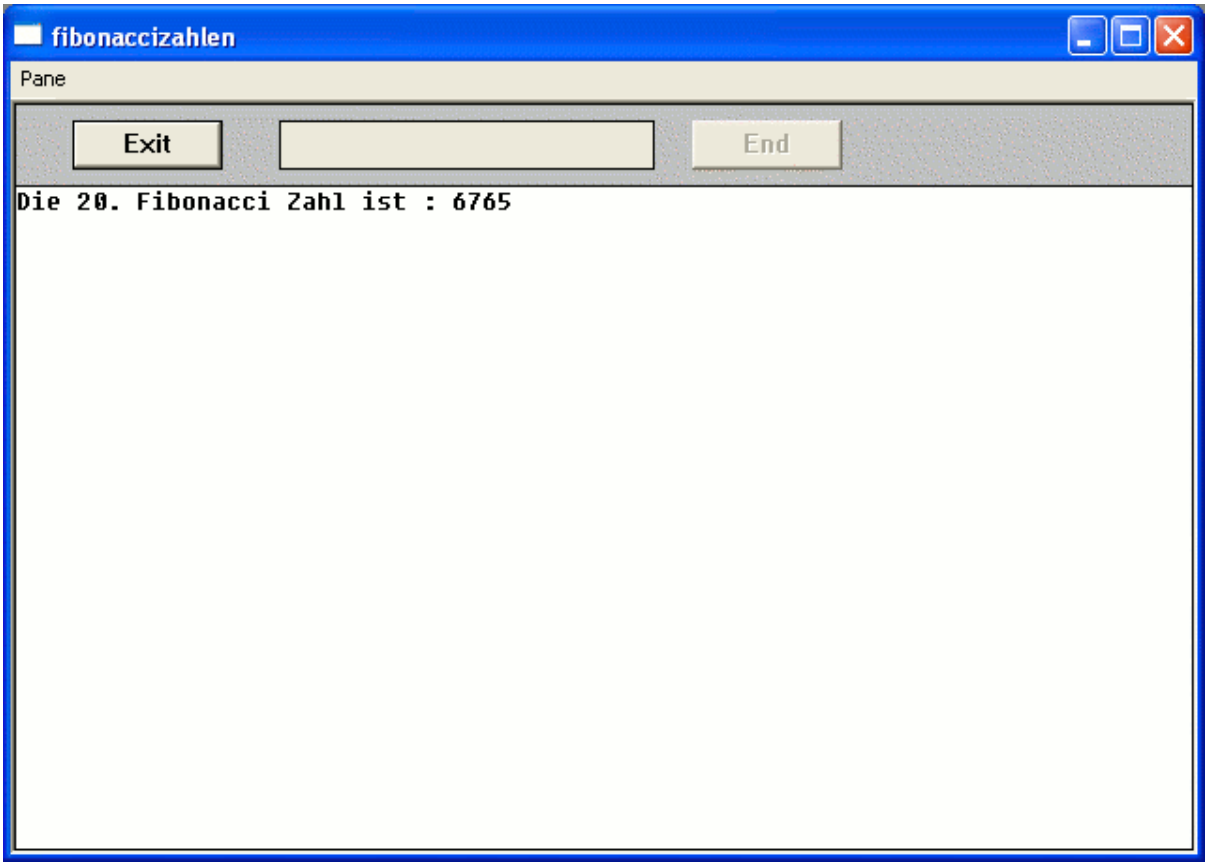


Abbildung 31: Ausgabe des mit POW! kompilierten Programms „Fibonacci-Zahlen“

5.1.12.2 Türme von Hanoi

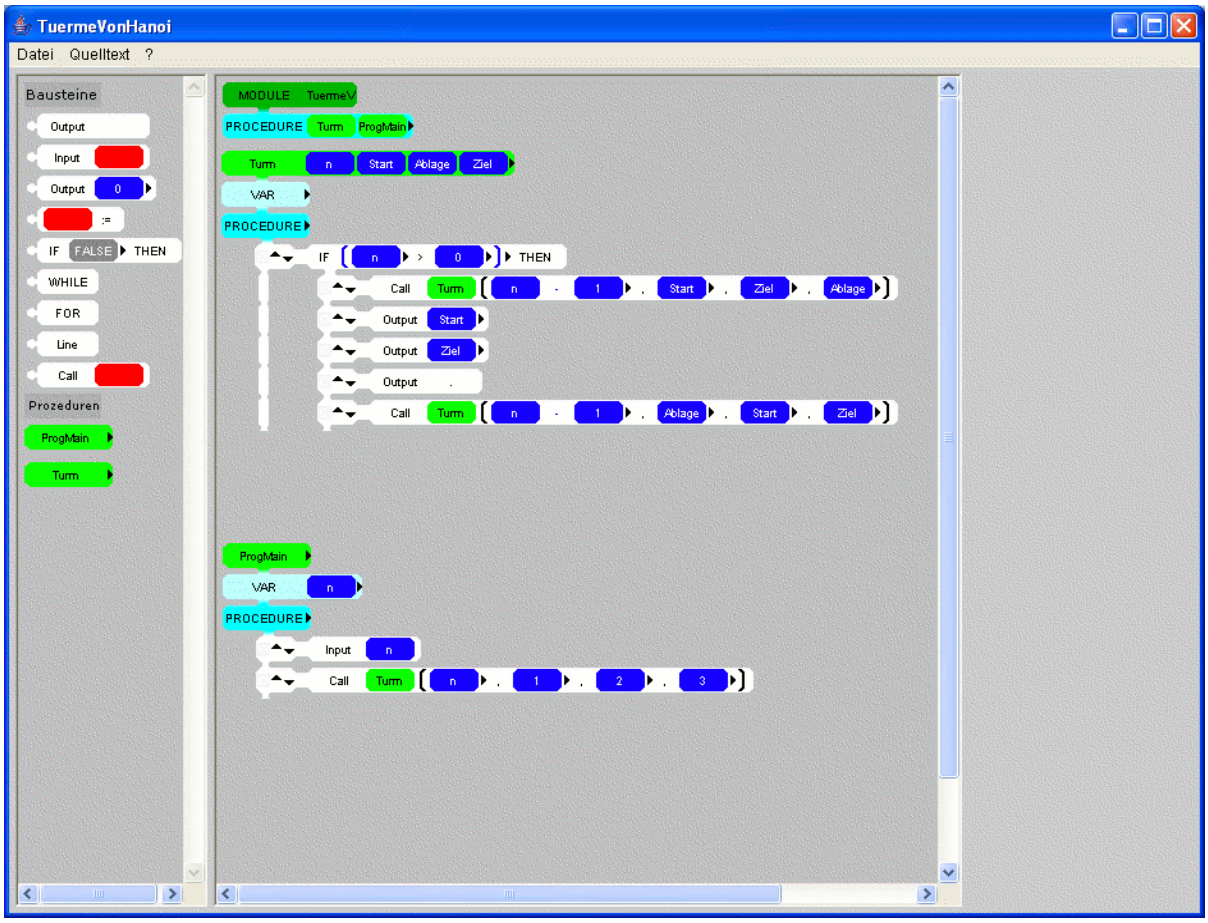


Abbildung 32: Das visuelle Beispielprogramm „Türme von Hanoi“

Laut Informatik-Duden demonstriert das Türme-von-Hanoi-Problem die Eleganz und Kürze rekursiver Lösungsalgorithmen (vgl. [Cl 03] S. 681). Die Prozedur „Turm“ in Abbildung 32 wird im Hauptprogramm nur einmal mit der eingegebenen Scheibenanzahl und den Turmnummern 1, 2 und 3 aufgerufen[31]. Wenn die Scheibenanzahl größer als 0 ist, wird ein Turm der Höhe n-1 vom Start mit Hilfe des Zieles auf die Ablage gebracht, dann wird die unterste Scheibe vom Start zum Ziel befördert und am Ende wird der Turm der Höhe n-1 von der Ablage mit Hilfe des Starts zum Ziel gebracht. Somit wird das Problem einen Turm der Höhe n zu bewegen solange auf das Problem einen Turm der Höhe n-1 zu bewegen reduziert, bis das n irgendwann 0 ist und nichts mehr gemacht werden muss. Abbildung 33 zeigt die Ausgabe des in POW! kompilierten Programms für die Turmhöhe 3. Der generierte Quelltext ist in Anhang 3.2 abgebildet.

[31] Auf einen Algorithmus mit nur drei Parametern (vgl. [Cl 03] S. 681) wurde verzichtet, weil nach Meinung des Autors die Prozedur auf die angegebene Art leichter zu verstehen ist.

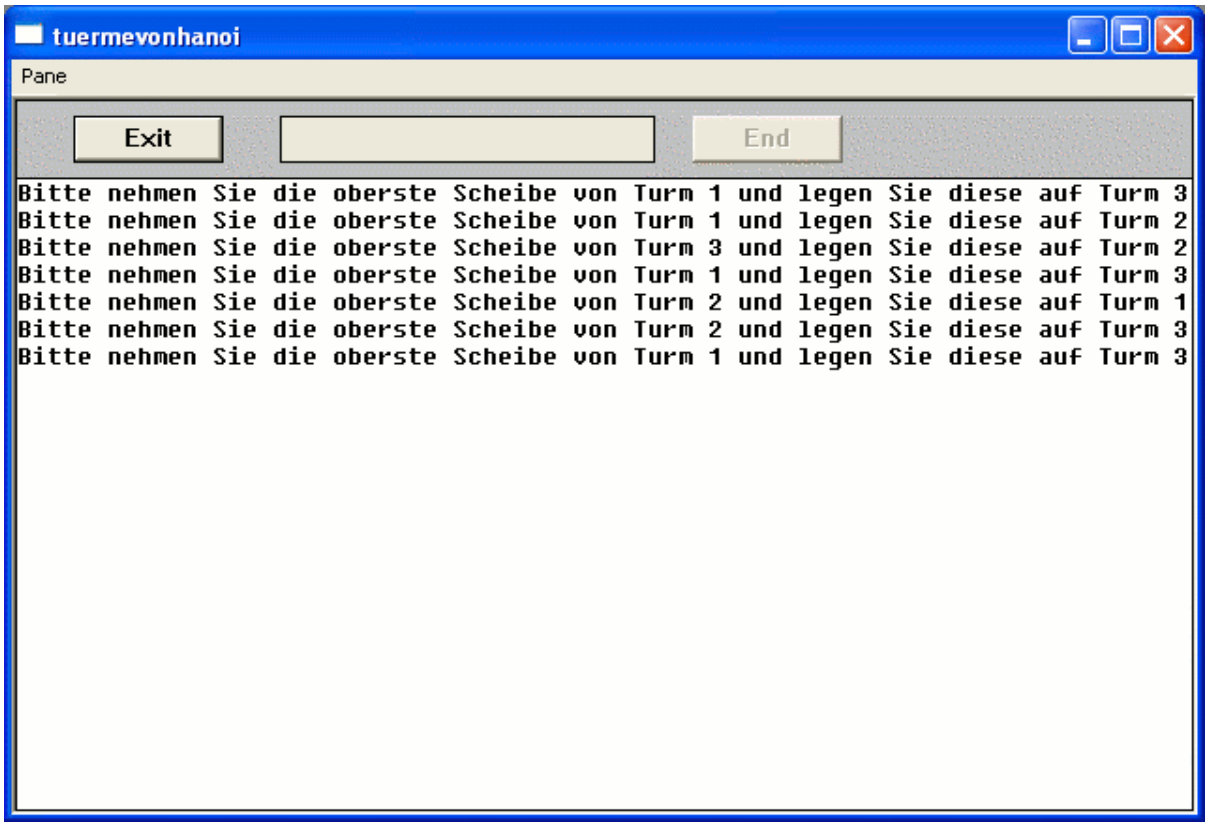


Abbildung 33: Ausgabe des mit POW! kompilierten Programms „Türme von Hanoi“

5.1.12.3 Fadengrafik

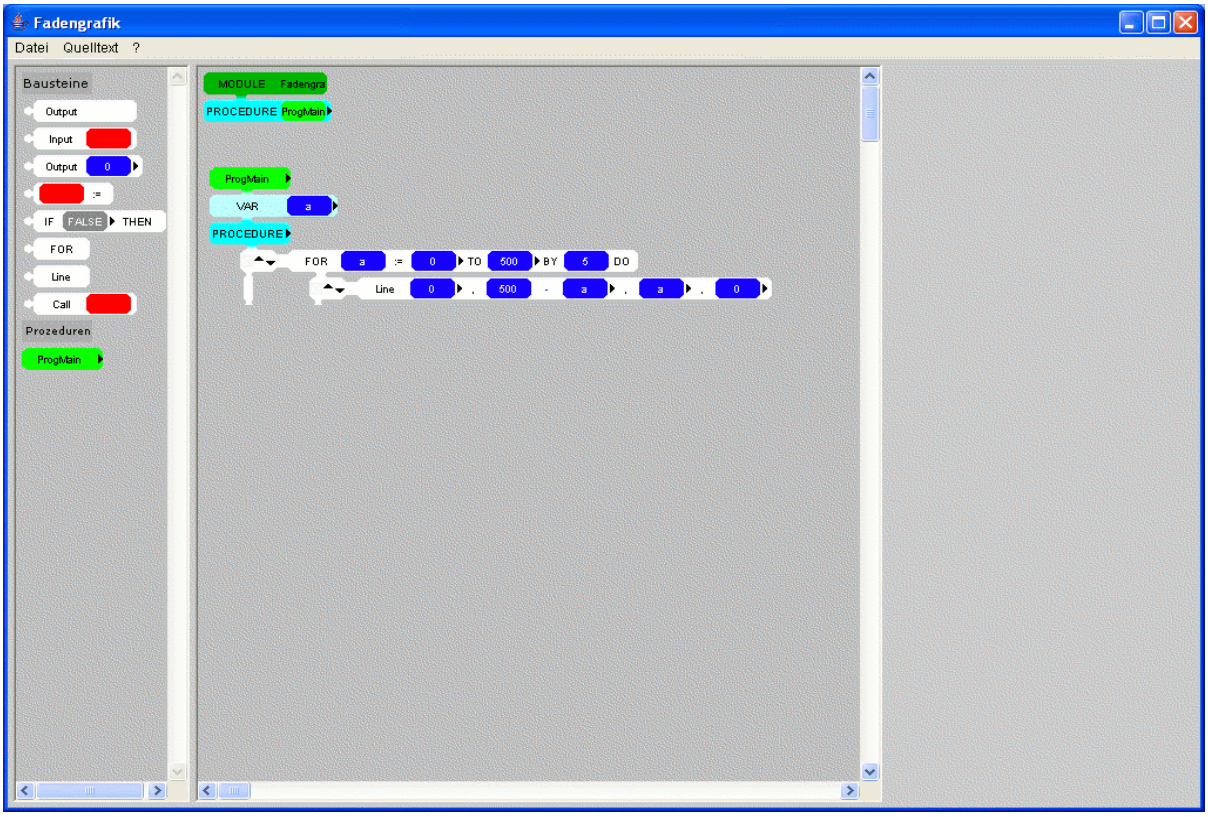


Abbildung 34: Das visuelle Beispielprogramm „Fadengrafik“

In Abbildung 34 ist ein Beispiel für ein einfaches Programm dargestellt, welches eine Fadengrafik ausgibt. In der Prozedur „ProgMain“ wird lediglich eine FOR-Schleife abgearbeitet, in der der Wert der vorher definierte Variable a von 0 bis 500 in 5-er Schritten hochgezählt wird. Im Schleifenrumpf wird jeweils eine Linie mit von a abhängigen Koordinaten gezeichnet. Die Ausgabe des Programms ist in Abbildung 35 dargestellt. Sie enthält eine Menge von Linien, die so angeordnet sind, dass sie eine Rundung darstellen. Der generierte Quelltext ist in Anhang 3.3 zu finden.

Dieses Beispiel soll demonstrieren, dass Bausteine, welche eine grafische Ausgabe erzeugen, für den Start in die Programmierung durchaus geeignet sind. Die Schüler werden motiviert und außerdem kann die Notwendigkeit der Verwendung von Schleifen und anderen Konstrukten veranschaulicht werden[32].

[32] Alle Schüler wollen die Grafik auf ihrem Bildschirm sehen, aber kein Schüler möchte 101 Line-Bausteine einzeln einfügen.

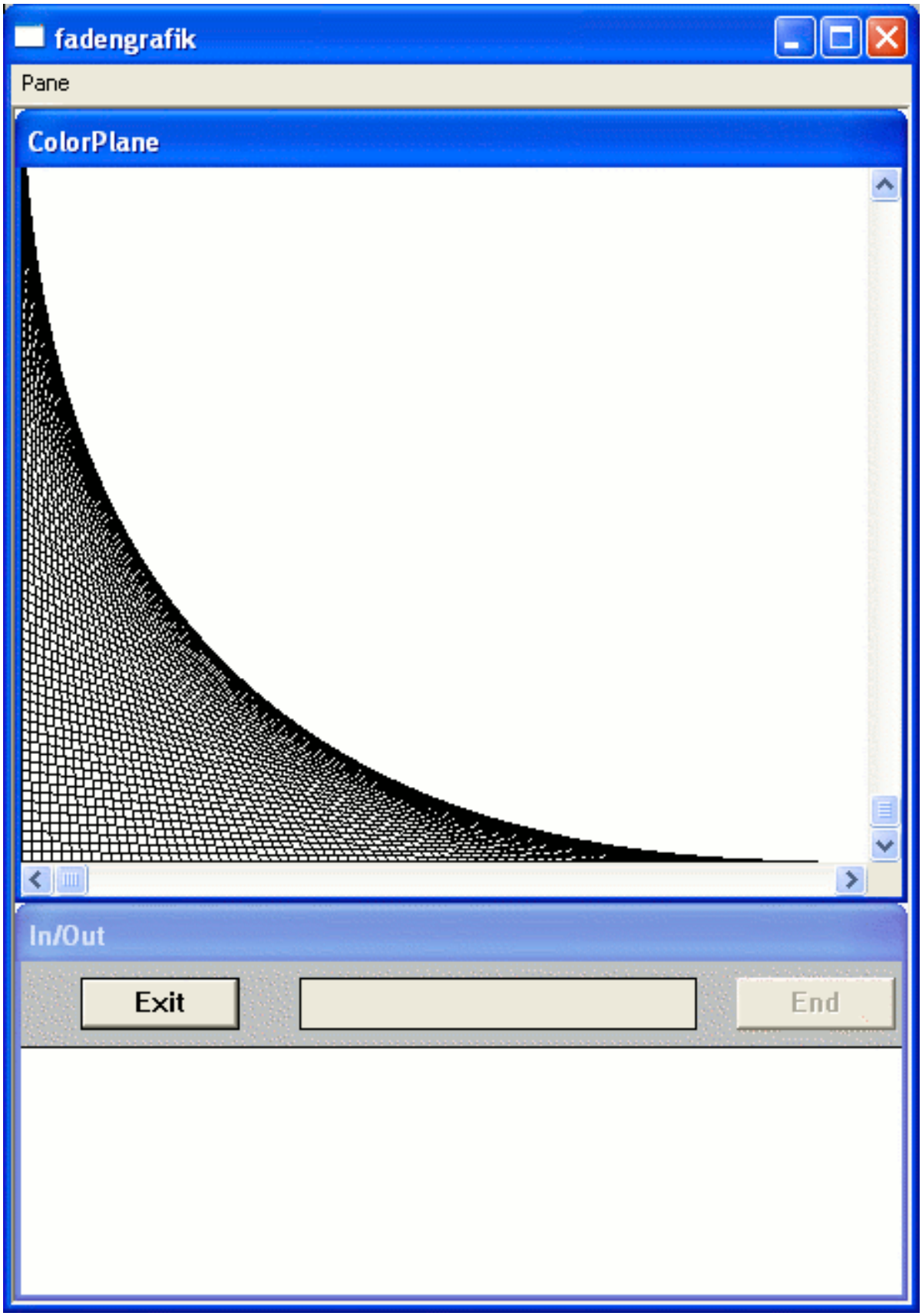


Abbildung 35: Ausgabe des mit POW! kompilierten Programms „Fadengrafik“

5.1.12.4 Mathematische Funktionen zeichnen

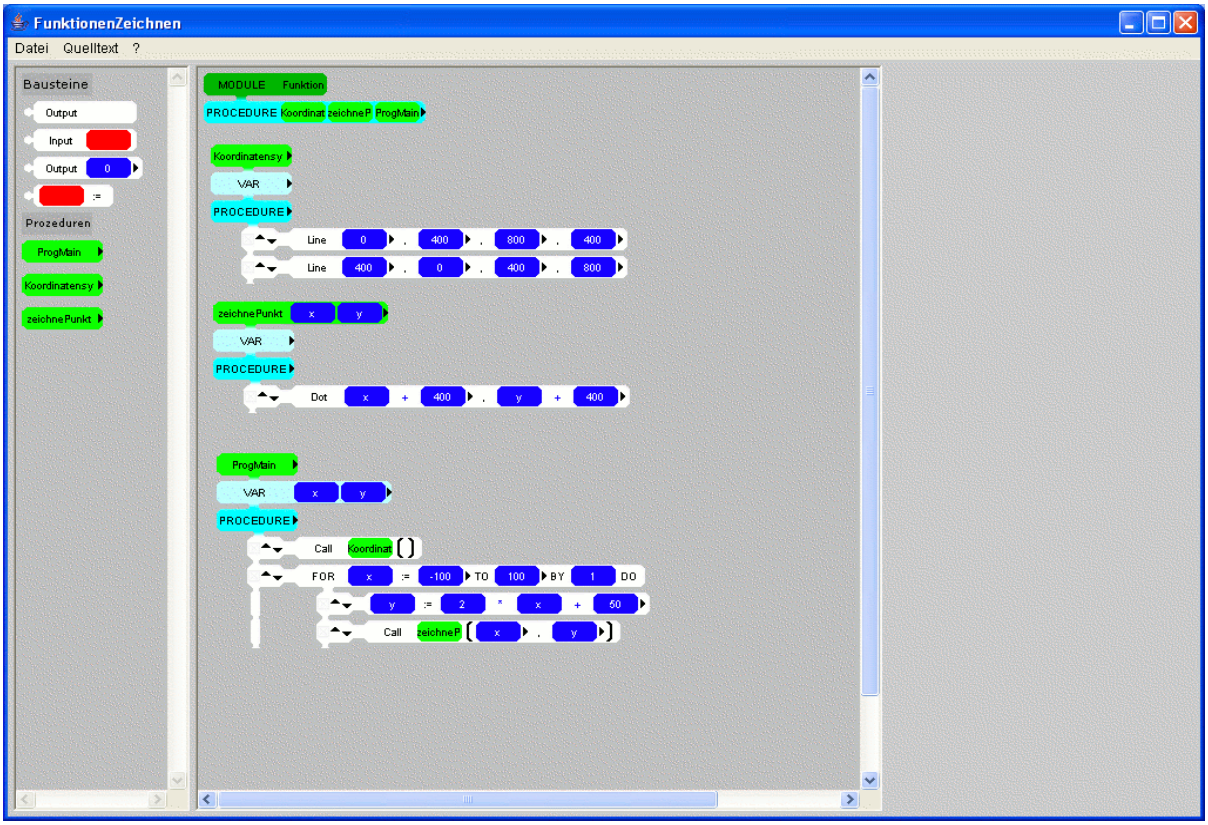


Abbildung 36: Das visuelle Beispielprogramm „Mathematische Funktionen zeichnen“

Mit dem Programm in Abbildung 36 können mathematische Funktionen gezeichnet werden. Es wurden zwei Prozeduren deklariert: Die Prozedur KoordinatensystemZeichnen zeichnet zwei Linien, welche das Ausgabefenster in die vier Quadranten aufteilen. Die Prozedur zeichnePunkt zeichnet einen Punkt in das Ausgabefenster, dessen Koordinaten als Parameter übergeben werden.

Im Hauptprogramm wird nun nur noch eine FOR-Schleife mit den gewünschten x-Werten durchlaufen. Im Schleifenrumpf wird die Funktion durch eine Wertzuweisung beschrieben, die dem y einen Wert in Abhängigkeit vom jeweiligen x übergibt. Anschließend wird die Prozedur zeichnePunkt mit den Parametern x und y aufgerufen. Die Ausgabe des Koordinatensystems mit der Geradengleichung y=2*x + 50, die durch das Programm erzeugt wird, ist in Abbildung 37 dargestellt. Das Programm kann so erweitert werden, dass der Benutzer selbst die Parameter der Geradengleichung eingeben kann.

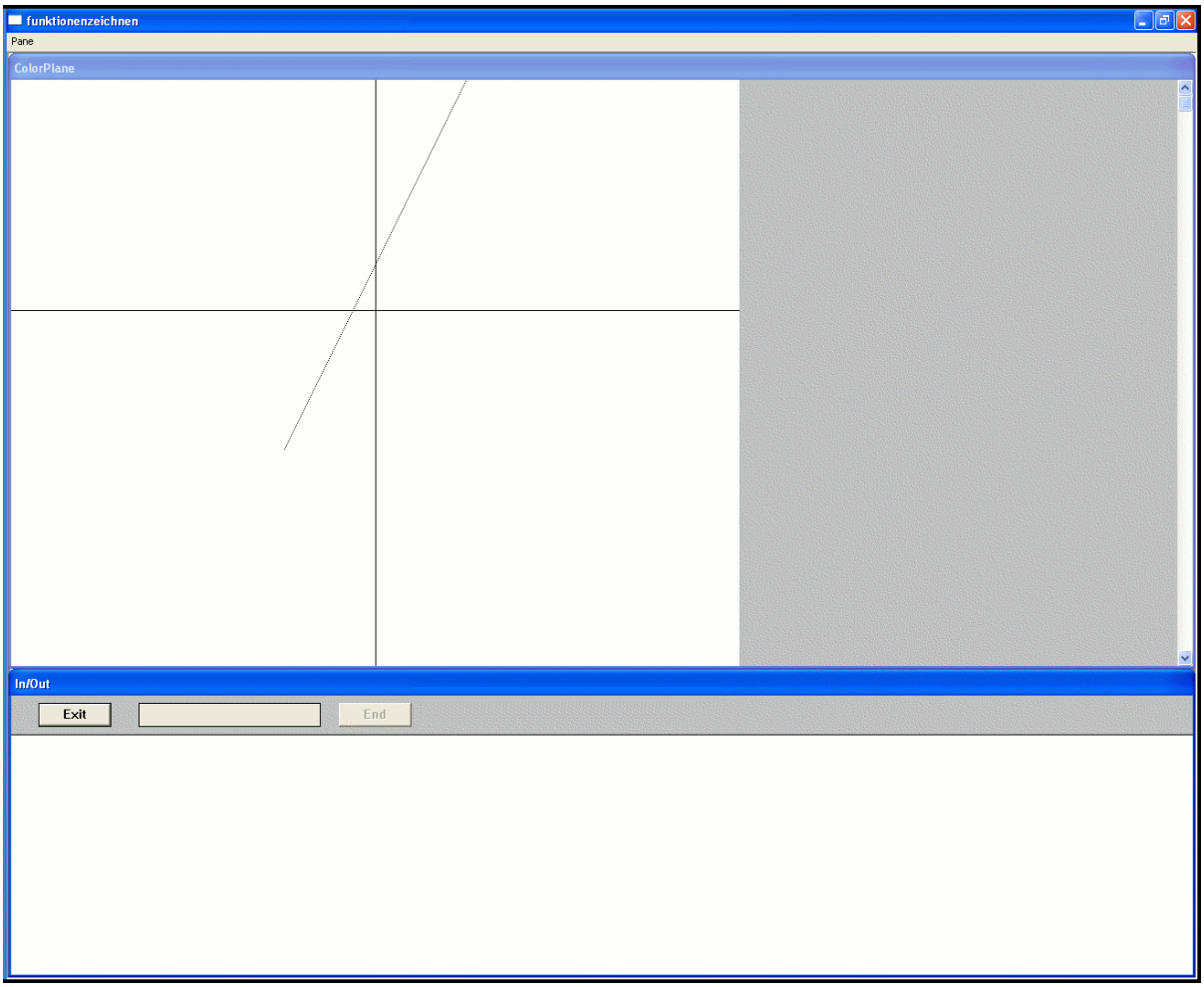


Abbildung 37: Ausgabe des mit POW! kompilierten Programms „Mathematische Funktionen zeichnen“

5.1.12.5 Größter gemeinsamer Teiler nach Nicomachos

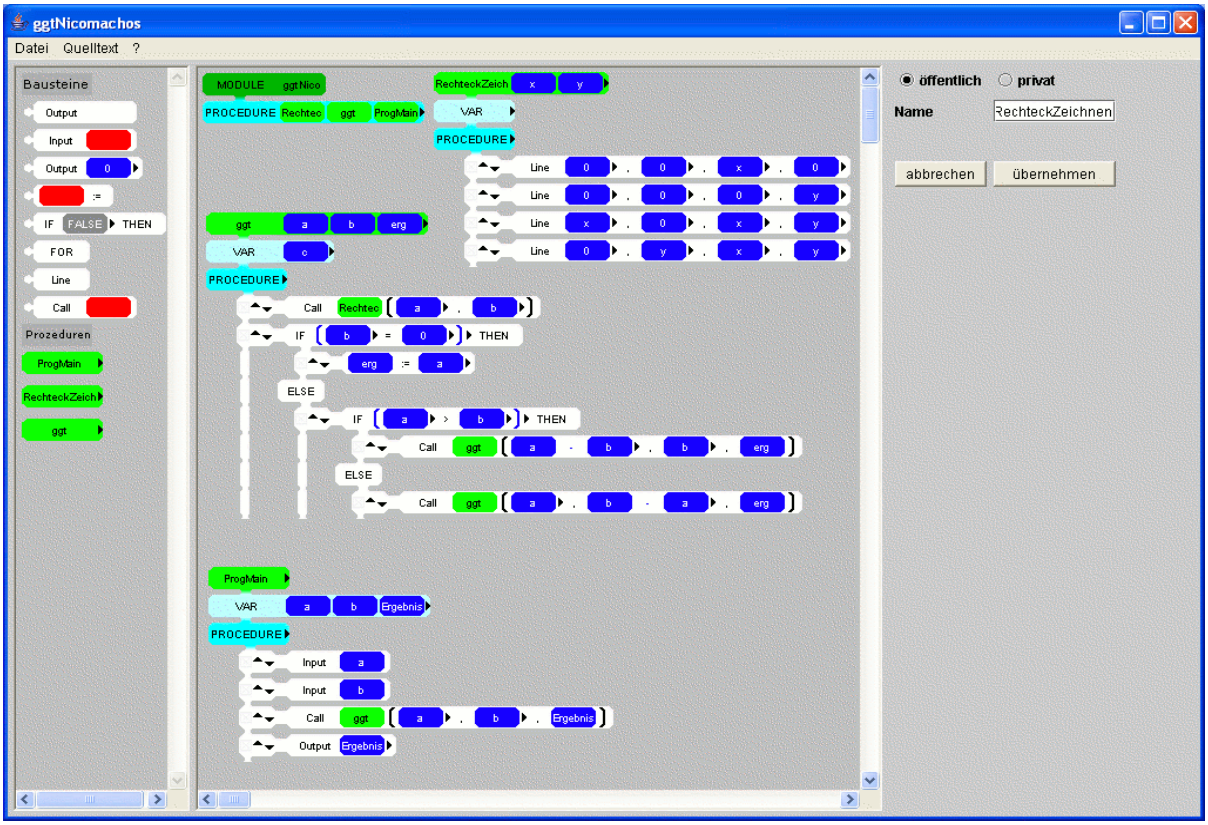


Abbildung 38: Das visuelle Beispielprogramm „Größter gemeinsamer Teiler nach Nicomachos“

In Abbildung 38 ist ein Programm dargestellt, das den größten gemeinsamen Teiler nach der Subtraktionsmethode bestimmt und nebenbei Rechtecke zeichnet. Diese stellen eine grafische Repräsentation des Problems dar, denn der ggt zweier Zahlen a und b kann geometrisch als das Aufteilen eines Rechteckes mit den Seitenlängen a und b in gleiche Quadrate mit der größten möglichen ganzzahligen Seitenlänge aufgefasst werden. Abbildung 39 zeigt die Ausgabe des Programms bei der Eingabe der Zahlen 606 und 170.

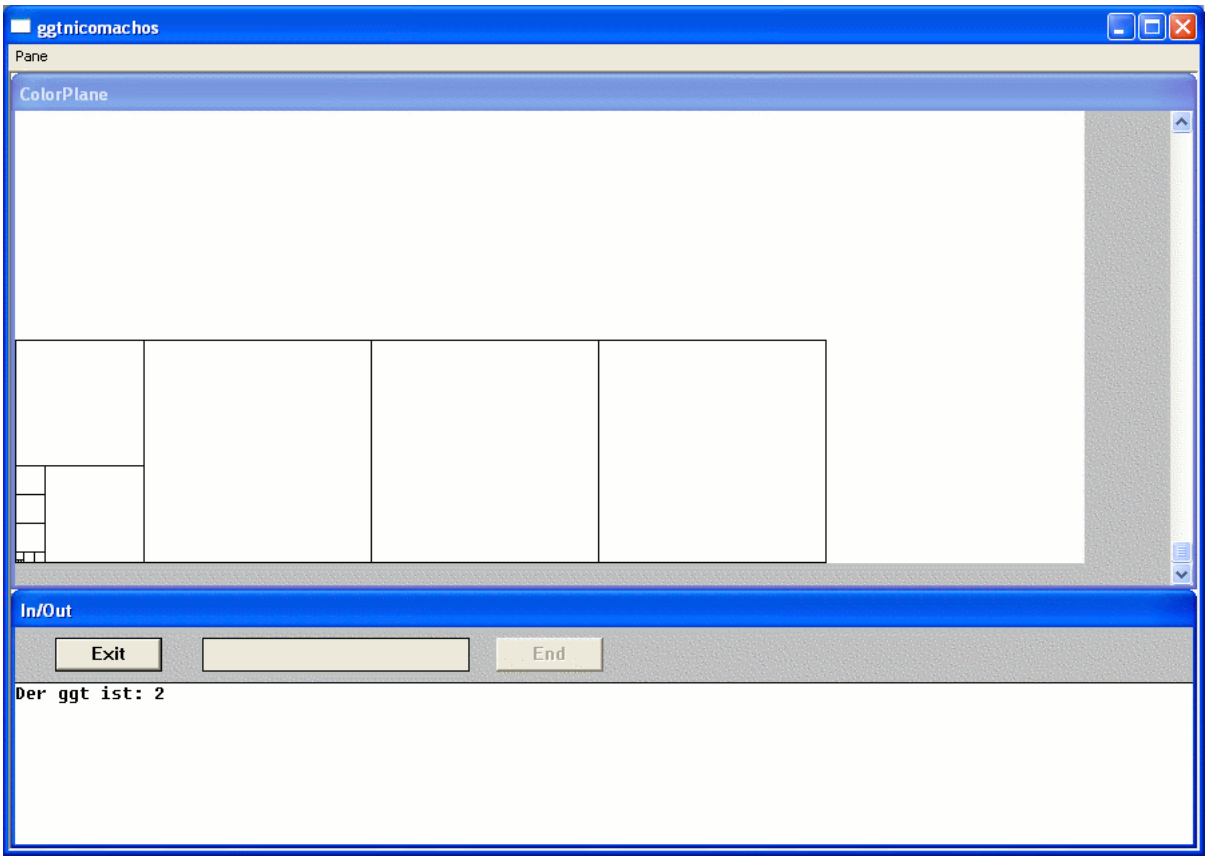


Abbildung 39: Ausgabe des mit POW! kompilierten Programms „Größter gemeinsamer Teiler nach Nicomachos“

5.2 Die Implementierung von Puck

Da der Prototyp noch mit kleinen Fehlern, Lücken und programmiertechnischen „Unschönheiten“ behaftet war, wurde er nicht weiterentwickelt. Das System wurde, mit den gewonnenen Erfahrungen, von Grund auf neu implementiert. Dadurch konnten viele zuvor kompliziert gelöste Teile erheblich vereinfacht werden. Des Weiteren wurden im Endprodukt Entwurfsmuster eingesetzt, wodurch die Gesamtarchitektur verbessert werden konnte.

Die Testphase hat außerdem gezeigt, dass das entstandene System variabel und erweiterbar ist. So konnten viele Änderungsvorschläge schnell umgesetzt werden[33].

Das entstandene System besteht aus 95 Klassen, die in 7 Pakete zusammengefasst sind[34]. Da eine detaillierte Beschreibung den Rahmen dieser Arbeit sprengen würde, sollen an dieser Stelle die Klassen nur kurz erläutert werden. Der interessierte Leser sei auf die Dokumentation verwiesen, in der die öffentlichen Attributen und Methoden aller Klassen kurz beschrieben sind. Da dies ein „Open Source“-Projekt ist, kann auch der Quelltext mit den entsprechenden Kommentaren eingesehen werden. Somit sind genügend Möglichkeiten gegeben, um sich in das System einzuarbeiten und Erweiterungen vorzunehmen.

Die Klassendiagramme in diesem Kapitel sind mit einem Programm aus dem Quelltext des erstellten Produktes generiert worden[35]. Auf Kardinalitäten und Bezeichnungen an Assoziationen und Vererbungspfeilen wurde verzichtet, da diese in den Klassendiagrammen mehr zur Unübersichtlichkeit als zum Verständnis des Systems beigetragen hätten. In welchem Verhältnis die Klassen der einzelnen Pakete zueinander stehen, kann in diesem Kapitel nachgelesen und im Quelltext nachvollzogen werden.

---

[33] Die Änderungswünsche eines Lehrers füllten zwei A4-Seiten. Alle diese Vorschläge wurden innerhalb von drei Stunden implementiert.

[34] Ein Paket mit 12 Klassen wurden aus der Studienarbeit „Entwicklung einer dynamischen Drag & Drop Umgebung zum Erstellen von visuellen Programmen“ übernommen (vgl. [Lä 04]).

[35] Zur Generierung der Klassendiagramme wurde das Programm „Together - Eclipse Edition“ von der Firma Borland eingesetzt. Während der Entwurfsphase wurde mit handschriftlichen Diagrammen gearbeitet.

5.2.1 Das Paket *system*

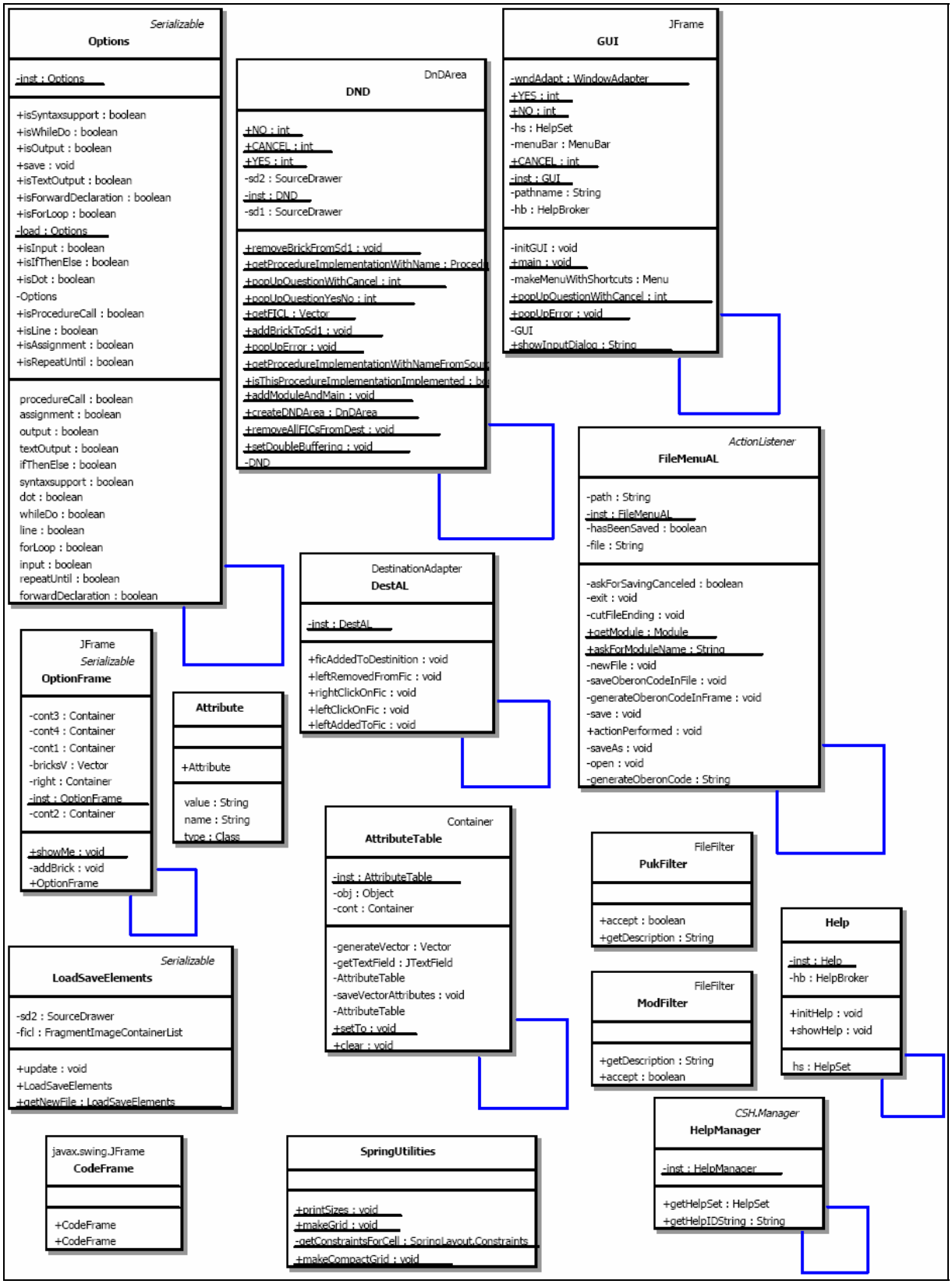


Abbildung 40: Klassendiagramm des Paketes *system*

In dem in Abbildung 40 dargestellten Paket *system* sind alle Klassen vereint, die für die Grundstruktur des Systems verantwortlich sind. Im Klassendiagramm gibt es nur wenige Assoziationen. Die meisten Klassen besitzen eine Verknüpfung die auf sie selbst verweist. Dies wird von der *getInst*-Methode des Singleton-Musters verursacht.

Das Singleton-Entwurfsmuster gewährleistet, dass eine Klasse nur eine einzige Instanz besitzt und stellt eine globale Zugriffsmöglichkeit für diese zur Verfügung (vgl. [Ga 97] S. 127-134). Dies hat nicht nur den Vorteil, dass Mehrfachinstanzierungen mit großem Speicherbedarf entfallen, sondern auch, dass sich die Anzahl der Parameter in Prozeduraufrufen des Paketes *system* im Vergleich zum Prototypen verringert[36], da auf die einzige Instanz einer Singleton-Klasse mittels einer statischen Klassenmethode getInst() zugegriffen wird. Durch die Verwendung dieses Musters konnte im Vergleich zum ersten Prototyp eine erhebliche Vereinfachung des gesamten Systems erreicht werden.

Die *Main*-Methode der Klasse *GUI* ist der Einstiegspunkt in das Puck-System. Hier wird die Instanz von *GUI* initialisiert und mit ihr auch die verschiedenen anderen Klassen des Systems. Die Oberfläche ist in drei Bereiche unterteilt (vgl. Abbildung 2). Auf der linken Seite befindet sich die Bausteinquelle, die direkt in dem Konstruktor von *GUI* bereitgestellt wird. In der Mitte befindet sich der Arbeitsbereich, welcher durch die Instanz der Klasse *DND* realisiert ist. Hier können die verschiedenen Bausteine zu einem Programm verknüpft werden. Die Klasse *DND* erweitert *DnDArea* aus dem Paket *newdnd* und enthält somit eine *Fragment-ImageContainerList*, in der die verschiedenen Bausteine verwaltet werden (vgl. [Lä 04] S. 20-26, 38-40). Des Weiteren beinhaltet *DND* noch eine Sammlung von statischen Methoden, die das Arbeiten mit den Bausteinen vereinfachen. Für die Überwachung von Ereignissen im Arbeitsbereich ist die Klasse *DestAL*[37] verantwortlich. Sie erbt das Verhalten von *DestinationAdapter* aus dem Paket *newdnd* (vgl. [Lä 04] S. 34-37). Das Auftreten eines Ereignisses in *DND* wird von *DestAL* registriert und an die richtigen Objekte weitergeleitet.

---

[36] Im Prototyp mussten bei Ereignissen, jeweils die Objekte der Systemklassen, wie z. B. das *DnDArea*-Objekt, den Prozeduren übergeben werden, die das jeweilige Ereignis behandelten. Damit konnten diese z. B. das Kontextmenü in die *DnDArea* zeichnen. Durch die Verwendung des Singleton-Entwurfsmusters fällt diese Übergabe der Systemklassen weg. Denn es existiert ja nur ein Objekt der Klasse *DND*, auf welches an allen Stellen des Programms über die statische *getInst*-Methode zugegriffen werden kann.

[37] Die Bezeichnung der Klasse *DestAL* ist die Kurzschreibweise für *DestinationActionListener*. Die Abkürzung AL wird auch in anderen Klassen äquivalent verwendet.

Auf der rechten Seite der Oberfläche befindet sich die Attributtabelle. Diese wird nach einem Klick mit der linken Maustaste auf ein *FI*-Objekt mit Inhalt gefüllt, falls das angeklickte Objekt vom Benutzer veränderbare Attribute enthält[38].

Attribute sind Objekte der Klasse *Attribute* und bestehen aus einem Namen, einem Typ und einem Wert. Aus diesen drei Elementen werden die Felder der Attributtabelle erzeugt. Hier kann nun der Wert des Attributes vom Benutzer verändert werden. Nachdem dieser dann den „übernehmen"-Button gedrückt hat, wird die Methode *saveVectorAttributes* in dem *AttributeTable*-Objekt aufgerufen und zulässige Werte werden in dem zugehörigen Objekt gespeichert.

Außer den beschriebenen drei Bereichen befindet sich auch noch eine Menüleiste an der oberen Kante der Benutzeroberfläche. Diese wird im Konstruktor der Klasse *GUI* erzeugt und alle Elemente werden mit dem *FileMenuAL*-Objekt verbunden, welches auf die Aktivierung der Menüpunkte reagiert. So werden beim Speichern, Laden, Öffnen und Anlegen einer neuen Datei die entsprechenden Methoden in der Instanz der Klasse *FileMenuAL* aufgerufen. Für die Serialisierung und die Wiederherstellung des Programmzustandes in bzw. aus einer externen Datei wurde eine Klasse *LoadSaveElements* implementiert. In ihr werden die *FragmentImageContainerList* des *DND*-Objektes und der zweite *SourceDrawer* der *GUI*-Instanz gespeichert. Über diese beiden Elemente, werden auch alle anderen Objekte eines visuellen Programms mit (de-) serialisiert. Zu speichernde Dateien werden mit der Endung „.puk" versehen um sie von anderen Programmen zu unterscheiden. Hierfür war die Implementierung der Klasse *PukFilter* notwendig.

Unter dem Menüpunkt Quelltext gibt es die beiden Optionen „Quelltext anzeigen" und „Quelltext in Datei speichern". Beim Anzeigen des Quelltextes öffnet sich ein Fenster – eine Instanz der Klasse *CodeFrame* – die den übergebenen Code anzeigt. Das speichern des Quelltextes in eine Datei wird direkt im *FileMenuAL*-Objekt ausgeführt. Die Datei wird unter einem Namen bestehend aus dem Modulnamen und der Endung „.mod" abgespeichert. Diese Konventionen sind nötig um die Datei in POW! ausführen zu können. Ähnlich wie beim Abspeichern der „.puk"-Dateien musste auch hier eine Klasse *ModFilter* implementiert werden.

Der dritte Punkt in der Menüleiste ist „?". Hier finden sich Hilfe und Optionen zum System. Die Hilfe ist mit dem JavaHelp-System implementiert (vgl. [www4]). Bei diesem müssen die HTML-Dateien mit den Inhalten des Hilfesystems erstellt und dann in verschiedenen XML-Dateien den jeweiligen Klassen zugeordnet werden. Das so erstellte Hilfesystem kann auch

[38] Für Attribute, die vom Benutzer verändert werden können, sind spezielle Vereinbarungen, bezüglich der automatischen Erstellung einer Attributtabelle, getroffen wurden. Diese sind in Kapitel 5.2.2.2. erläutert.

kontextsensitiv genutzt werden. Hierfür muss der Benutzer die „F1“-Taste drücken, während er sich mit dem Mauszeiger direkt über dem Objekt befindet, zu dem er eine Hilfe haben möchte. Die Klasse *HelpManager* ist für die Verwaltung der kontextsensitiven Aktionen verantwortlich[39]. Die Hilfe wird beim Aufruf des Konstruktors der Klasse *GUI* initialisiert. Beim Anklicken des Menüelements „Puck Hilfe“ wird lediglich die einzige Instanz der *Help*-Klasse mit der *showHelp*-Methode angezeigt.

Die Optionen des Systems wurden mit Hilfe von zwei Klassen implementiert. Die Klasse *OptionFrame* zeigt ein Fenster, in dem die verschiedenen Optionen eingestellt werden können. Das Layout wird mit Hilfe der Klasse *SpringUtilities* verwaltet[40]. Beim Drücken des „speichern“-Buttons werden die aktuellen Einstellungen in ein Objekt der Klasse *Options* geschrieben, welches sich selbst bei Veränderungen in die Datei „PuckOptions.opt“ abspeichert. Beim erneuten Öffnen der Optionen werden die Einstellungen im *OptionFrame* aus dem *Options*-Objekt geladen. Dieses wiederum versucht bei seiner Initialisierung auf die Datei „Options.opt“ zuzugreifen. Damit werden die eingestellten Optionen auch beim nächsten Programmstart beibehalten. Die verschiedenen Objekte, die auf die Einstellungen in den Optionen angewiesen sind, überprüfen diese jeweils am *Options*-Objekt.

[39] Detaillierte Angaben zur Implementierung eines dynamischen HelpManagers findet man unter ([www5] S. 90-99).

[40] Die Verwendung des SpringLayouts unter Java, sowie die Klasse *SpringUtilities* ist auf [www6] dargestellt.

5.2.2 Das Paket *basis*

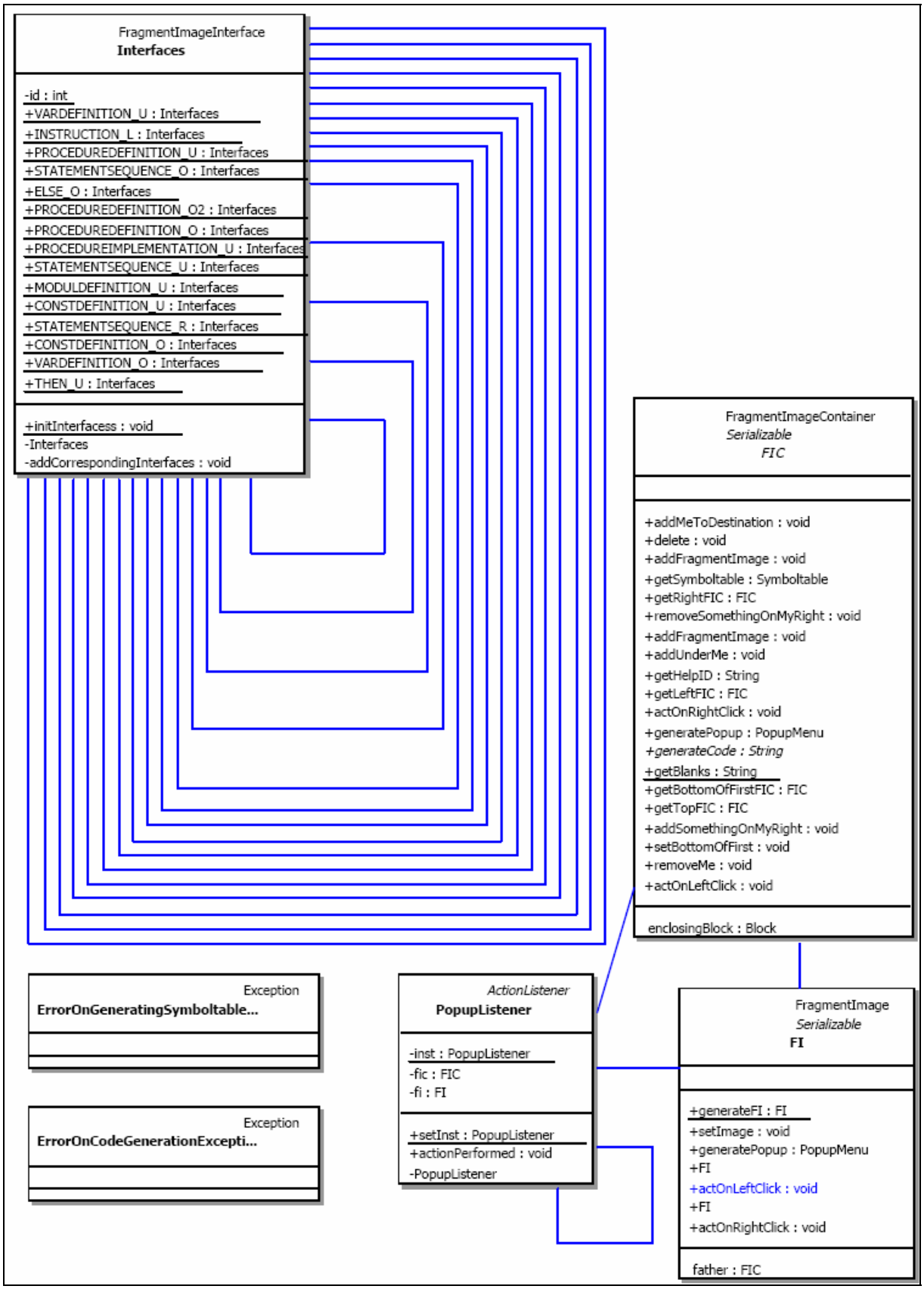


Abbildung 41: Klassendiagramm des Paketes *basis*

In dem in Abbildung 41 dargestellten Paket *basis* befinden sich die Klassen, die das Grundverhalten der Bausteine regeln. *FI*, *FIC* und *Interfaces* bilden die Oberklassen aller Bausteinelemente, wobei ein *FIC* einen Baustein beschreibt, der aus verschiedenen Teilbildern – Objekten der Klasse *FI* – besteht[41]. Um verschiedene Bausteine miteinander zu verknüpfen, gibt es Anschlussstellen, die, um eine Klassenhäufung zu vermeiden, alle als statische Variable in der Klasse *Interfaces* implementiert sind[42]. Die Implementierungen der drei Klassen *FIC*, *FI* und *Interfaces* sollten im Zusammenhang mit der jeweiligen Oberklasse aus dem Paket *newdnd* analysiert werden (vgl. [Lä 04] S. 14-26). Abbildung 42 zeigt verschiedene Bausteine mit ihren Teilbildern und den Anschlussstellen.

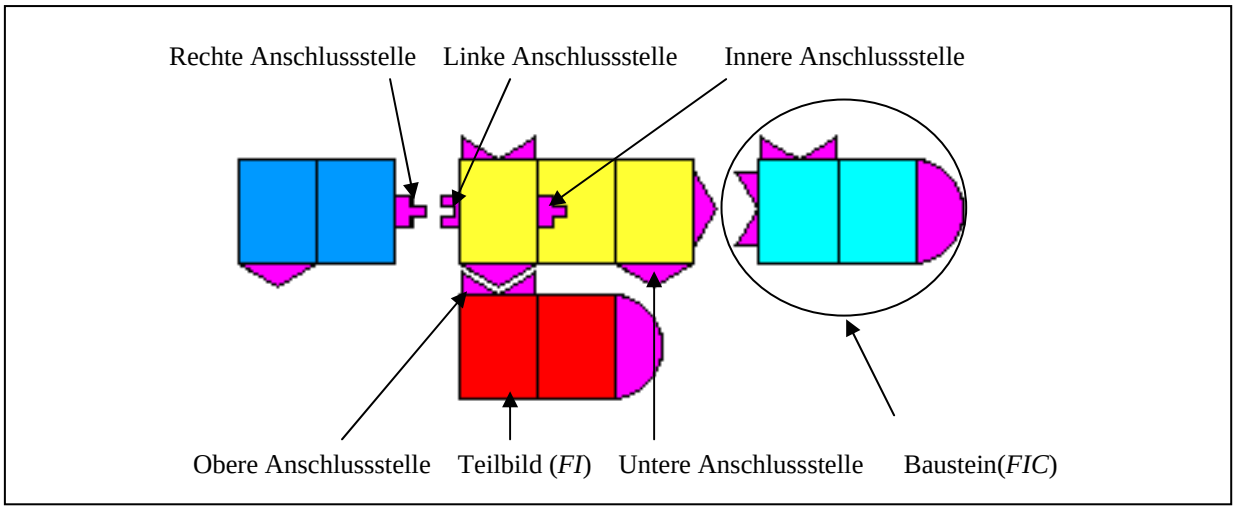


Abbildung 42: Verwendung der Klassen FIC, FI und Interfaces (vgl. [Lä 04] S. 14)

Des Weiteren befinden sich in dem Paket *basis* noch zwei Exceptions, die beim fehlerhaften Erstellen des Quelltextes oder der Symboltabelle aufgerufen werden. Außerdem gibt es eine Klasse *PopupListener*. Wenn ein Nutzer mit der rechten Maustaste auf ein *FIC*-Objekt klickt, dann wird in den meisten Fällen ein Popup-Menü erstellt, aus dem der Benutzer verschiedene Möglichkeiten wählen kann. Der *PopupListener* beobachtet das Popupmenü und ruft bei der Aktivierung eines Elementes die Methode *actOnRightClick* im jeweiligen *FIC* auf. Dort ist dann die gewünschte Funktionalität implementiert.

---

[41] *FI*, *FIC* und *Interfaces* erben von den Klassen *FragmentImage*, *FragmentImageContainer* und *FragmentImageInterface* aus dem Paket *newdnd*. Die Namen sind Kurzschreibweisen der jeweiligen Oberklasse. Die Klasse *Interfaces* beschreibt keine Schnittstelle (Interface in Java), sondern die visuellen Elemente, mit deren Hilfe in Puck Bausteine miteinander verknüpft werden können (vgl. Abbildung 42).

[42] Durch die als statische Attribute definierten Anschlussstellen entstehen die vielen Selbstreferenzen der Klasse *Interfaces* im Klassendiagramm von *basis*.

Im Folgenden sollen die drei Hauptklassen *FIC*, *FI* und *Interfaces* genauer erläutert werden, denn diese bilden das Grundgerüst aller Bausteine und sind somit besonders für Leser, die das System weiterentwickeln wollen, interessant.

5.2.2.1 Die Klasse *FIC*

Die Klasse *FIC* erweitert die Klasse *FragmentImageContainer* und stellt spezielle Methoden für die Bausteine, die in Puck benötigt werden, zur Verfügung. Diese erben das Verhalten von *FIC*, erweitern es, überschreiben eventuell einige der Methoden und können so auf gleiche Aufrufe unterschiedlich reagieren. Für alle Ereignisse, die jeweils von der Klasse *DestAL* registriert und weitergeleitet werden, gibt es in *FIC* eine Standardimplementierung, welche nur von Bausteinen überschrieben wird, die spezifisch auf ein bestimmtes Ereignis reagieren müssen. Im Folgenden sollen Attribute und Methoden von *FIC* erläutert werden.

Jedes *FIC* hat ein Attribut *enclosingBlock* über dessen Getter- und Setter-Methode auf den jeweiligen umschließenden Block zugegriffen werden kann[43]. Dies ist sehr nützlich für die Symboltabellengenerierung eines *FIC*-Objektes, mit deren Hilfe die in einem Baustein sichtbaren Variablen, Parameter und Prozeduren bestimmt werden. Denn dafür muss die entsprechende Symboltabelle des umschließenden Blockes generiert werden. Dieser ruft wiederum die *getSymboltable*-Methode des ihn umschließenden Blockes auf usw. bis das Modul gefunden ist. Um zu garantieren, dass immer der richtige umschließende Block angegeben ist, muss beim Verschieben der Bausteinen von einem Block zum anderen jeweils das *enclosingBlock*-Attribut richtig gesetzt werden.

Die statische Methode *getBlanks* bekommt durch den Integerparameter i eine Zahl übergeben und gibt einen String zurück. In diesem String befinden sich genau i Leerzeichen. Diese Methode wird in der Codegenerierung benötigt, um die Einrückung um eine bestimmte Anzahl von Leerzeichen zu realisieren.

Mit *getHelpID* wurde eine Methode bereitgestellt, in der zu jedem Baustein eine Help-ID angegeben werden kann. Dies ist nötig um bei der kontextsensitiven Hilfe jedem Baustein eine bestimmte Seite der Hilfe zuzuordnen. Hierfür muss in der XML-Datei für die Hilfeverwaltung die Help-ID mit der entsprechenden UML der Hilfeseite verknüpft werden.

---

[43] Für eine Anweisung ist der umschließende Block derjenige, in dem die Anweisung verwendet wird. Eine Prozedur wird von dem Block umschlossen, der die Prozedur definiert.

getRightFIC gibt das mit dem rechten Interface verknüpfte *FIC* zurück.

Die Methode *setBottomOfFirst* bekommt ein *Interfaces*-Objekt übergeben und fügt dieses dem *FIC* unterhalb des ersten *FI* hinzu. Somit kann dieser Baustein vom Benutzer mit einem anderen, der über eine passende Anschlussstelle an der Oberseite verfügt, verbunden werden.

In der Methode *actOnRightClick* sind die Befehle implementiert, welche bei einem Klick auf ein Element des Kontextmenüs ausgeführt werden sollen. Der Parameter *actionCommand* gibt an, welches Element des Menüs gewählt wurde. Außerdem wird noch das Teilbild übergeben, auf welches geklickt wurde. Somit kann unterschieden werden, welches *FI* ausgewählt wurde, und in besonderen Fällen dessen spezielle *actOnRightClick*-Methode aufgerufen werden.

Die Methode *actOnLeftClick* wird immer dann aufgerufen, wenn mit der linken Maustaste auf ein *FIC* geklickt wurde. Das ausgewählte *FI* wird übergeben und die Standardimplementierung ruft die Methode *actOnLeftClick* in diesem *FI* auf.

Es wurde eine abstrakte Methode *generateCode* vereinbart, welche jede Unterklasse von *FIC* implementieren muss. Somit ist *FIC* abstrakt und es können keine Objekte dieser Klasse erzeugt werden, denn es wäre ja nicht möglich, für ein Objekt dieser abstrakten Oberklasse aller Bausteine, speziellen Code einer Programmiersprache zu generieren. *generateCode* bekommt zwei Integerparameter übergeben. Der erste beinhaltet die Anzahl der Leerzeichen, die für die Einrückung des Quelltextes notwendig sind. Der zweite Integerparameter bestimmt die Programmiersprache, für welche Quelltext generiert werden soll. Im Rahmen dieser Arbeit war es aufgrund des engen Zeitrahmens nicht möglich, Quelltext für verschiedene Programmiersprachen zu generieren. Dies wäre aber eine durchaus wünschenswerte Erweiterung des Systems.

Die Methode *addUnderMe* ist eine Spezialisierung der Methode *addUnderAndRechain*. Es wird das als Parameter übergebene *FIC* unter dem ersten Bild des *FIC*-Objektes angehängt, dessen *addUnderMe*-Methode aufgerufen wurde.

addSomethingOnMyRight wird von der Klasse *DestAL* aufgerufen, sobald diese beobachtet, dass vom Benutzer etwas an der rechten Anschlussstelle des *FIC*-Objektes angehängt wurde. Die Methode hat standardmäßig eine leere Implementierung, da nur spezielle Bausteine auf dieses Ereignis reagieren müssen[44].

[44] Die Methode *addSomethingOnMyRight* wird von der Klasse *StatementSequenceElement* aus dem Paket *bricks* genutzt.

Durch die Methode *getSymboltable* soll die Symboltabelle eines *FIC* zurückgegeben werden. Die Methode muss natürlich in den Bausteinen entsprechend der Symboltabelle die generiert werden soll, überschrieben werden. Standardmäßig wird das Ergebnis der *getSymboltable*-Methode des *FIC*, das mit der oberen Anschlussstelle verbunden ist, zurückgegeben.

Die Methode *addFragmentImage* aus *FragmentImageContainer* wurde in *FIC* überschrieben, um beim Hinzufügen eines Teilbildes zu einem Baustein, automatisch das *Father*-Attribut des *FI* auf das *FIC* zu setzen, zu dem es hinzugefügt wurde. Somit kennt jedes *FI* das *FIC*, zu dem es gehört.

getBottomOfFirst gibt das mit der unteren Anschlussstelle des ersten Teilbildes eines Bausteins verbundene *FIC* zurück.

Die Methode *removeMe* ist eine Spezialisierung von *removeUnderAndRechain* (vgl. [Lä 04] S. 23). Das *FIC*, dessen *removeMe*-Methode aufgerufen wurde, wird gelöscht. Anschließend wird das *FIC* oberhalb des gelöschten mit dem unterhalb des gelöschten verknüpft.

Die Methode *generatePopup* wird von einem *DestAL*-Objekt aufgerufen, wenn mit der rechten Maustaste auf ein *FIC* geklickt wurde. Ein Popup wird zurückgegeben, welches dann vom *DestAL*-Objekt angezeigt wird. Standardmäßig befindet sich ein Element mit der Aufschrift „löschen“ in dem Popupmenü. Die Methode muss von Bausteinen, welche andere Menüelemente benötigen, überschrieben werden.

Die Methode *getLeftFIC* gibt das mit der linken Anschlussstelle verknüpfte *FIC* zurück.

removeSomethingOnMyRight wird in einem FIC von der Instanz der Klasse *DestAL* aufgerufen, sobald diese beobachtet, dass vom Benutzer etwas von der rechten Anschlussstelle des *FIC* entfernt wurde. Die Methode hat standardmäßig eine leere Implementierung, da nur spezielle Bausteine auf dieses Ereignis reagieren müssen[45].

Die Methode *delete* entfernt ein *FIC*-Objekt aus dem Arbeitsbereich. Wenn beim Löschen noch weitere Aufgaben übernommen werden sollen, muss diese Methode überschrieben werden.

Die Methode *getTopFIC* gibt das mit der oberen Anschlussstelle verknüpfte *FIC* zurück.

---

[45] Die Methode *removeSomethingOnMyRight* wird von der Klasse *StatementSequenceElement* im Paket *bricks* genutzt.

5.2.2.2 Die Klasse *FI*

Die Klasse *FI* erweitert *FragmentImage* aus dem Paket *newDnD* (vgl. [Lä 04] S. 16-20). Ein *FI*-Objekt ist ein Teilbild in einem Baustein. Auch *FI* hat verschiedene Methoden, die Reaktionen auf Ereignisse beinhalten und in den Unterklassen mit speziellen Implementierungen überschrieben werden können.

Um auf Attribute einer Unterklasse von *FI* über die Attributtabelle zugreifen zu können, ist eine Vereinbarung getroffen worden: Alle Attribute, deren Namen mit einem Unterstrich beginnen und die über die standardisierten Getter- und Settermethoden verfügen, werden automatisch bei einem Klick mit der linken Maustaste auf das gewählte *FI* in der Attributtabelle dargestellt[46]. Die hier vom Benutzer gemachten Veränderungen werden nach einem Klick auf den „übernehmen“-Button der Attributtabelle automatisch in das gewählte *FI*-Unterklassen-Objekt gespeichert.

Die Klasse *FI* besitzt zwei Konstruktoren, die je einen String übergeben bekommen, der angibt, wo sich die Datei mit dem Bild befindet, welches das *FI* darstellt. Der Quellpfad des Bildes wird auf die Bedürfnisse innerhalb einer „.jar“ Datei angepasst. Dem zweiten Konstruktor kann man zusätzlich noch ein *FIC* übergeben. Hier wird das *Father*-Attribut dann auf das übergebene *FIC* gesetzt.

Zusätzlich gibt es noch eine statische Methode, die ein *FI* zu einer Bilddatei generiert. Diese Methode heißt *generateFI*.

Die Methode *generatePopup* generiert für ein spezielles *FI* ein Popup-Menü. Diese Methode wird meist von der *generatePopup*-Methode des *FIC*, welches das angeklickte *FI* enthält, aufgerufen. Somit können für unterschiedliche Teilbilder in einem Baustein verschiedene Popup-Menüs erzeugt werden.

actOnRightClick ist ähnlich wie *generatePopup* eine Methode, die eine spezielle Reaktion eines Teilbildes in einem Baustein hervorrufen kann. Im Parameter *actionCommand* wird der durch ein Popup gewählte Menüeintrag übergeben. Um spezielle Reaktionen auf eine Auswahl in Popupmenüs hervorzurufen muss die Methode in den Unterklassen überschrieben werden.

Die Methode *actOnLeftClick* wird von *FIC* bei einem Linksklick auf ein *FI* aufgerufen. Sie generiert die Attributtabelle auf der rechten Seite der Benutzeroberfläche. Wenn bei einem

[46] Die automatische Generierung der Attributtabelle funktioniert in der vorliegenden Implementierung nur für Attribute bestimmter Datentypen, kann aber beliebig erweitert werden.

Klick mit der linken Maustaste etwas anderes passieren soll, muss die Methode überschrieben werden.

setImage ruft die gleichnamige Methode aus der Oberklasse *FragmentImage* mit dem auf die „.jar“-Datei angepassten Quellpfad auf und weist dem *FI* so ein neues Bild zu.

5.2.2.3 Die Klasse *Interfaces*

Die Klasse *Interfaces* erbt das Verhalten und die Attribute von *FragmentImageInterface* (vgl. [Lä 04] S. 14-16). Objekte dieser Klasse unterscheiden sich nur durch die mit ihnen verknüpfbaren Anschlussstellen und die Bilder, aus denen sie bestehen. Um eine Häufung von *Interfaces*-Unterklassen, die sich nur in zwei Attributen unterscheiden würden, zu verhindern, wurden die verschieden Anschlussstellen als statische Klassenvariablen bereitgestellt.

Die Methode *initInterfaces* initialisiert die statischen Attribute. Wenn eine neue Anschlussstelle entstehen soll, so wird hier die neue statische Klassenvariable mit dem richtigen Bild initialisiert. Anschließend können mit der Methode *addCorrespendingInterfaces* Anschlussstellen angegeben werden, welche dann vom Benutzer an dieses angedockt werden können.

5.2.3 Das Paket *symboltable*

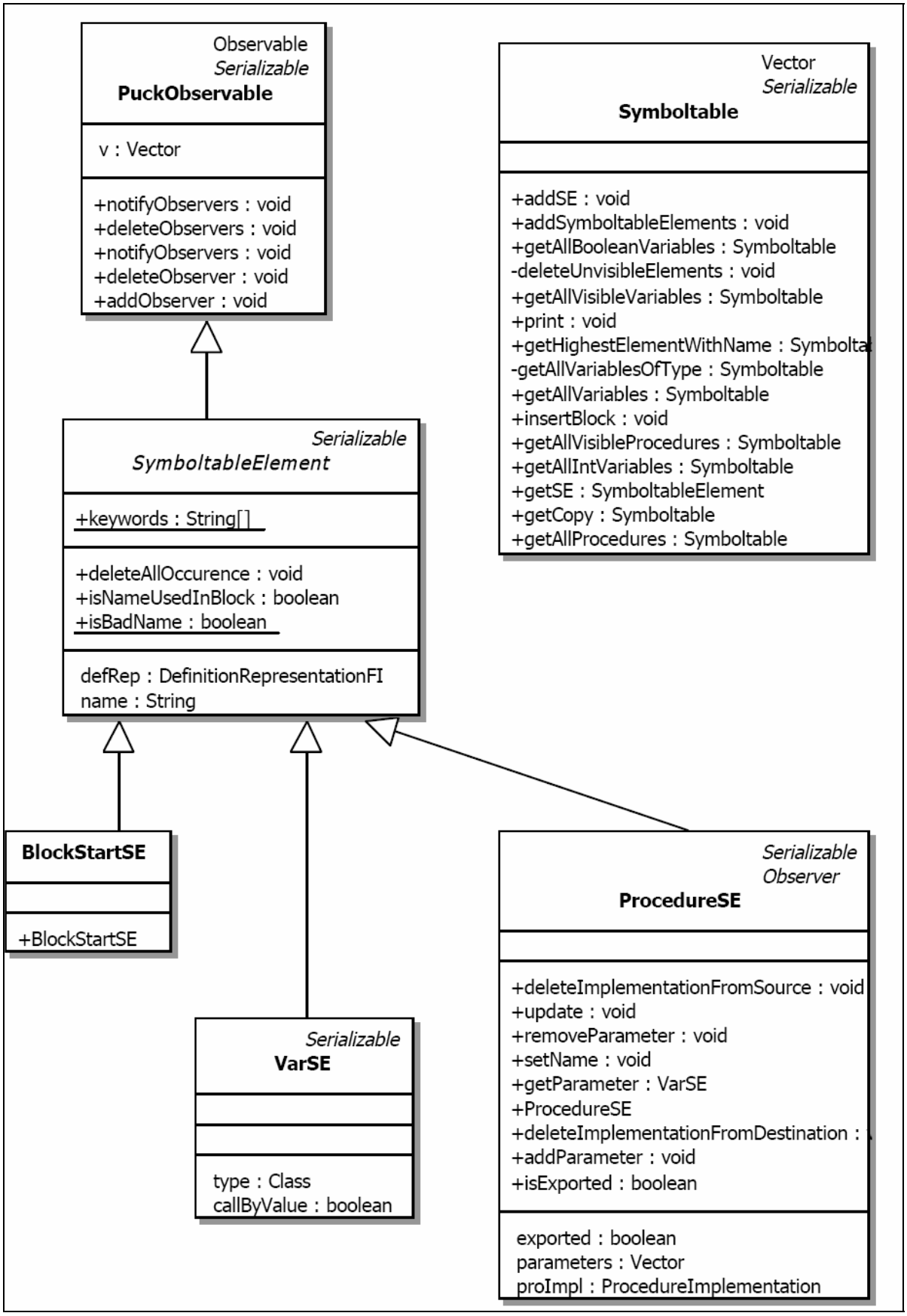


Abbildung 43: Klassendiagramm des Paketes *symboltable*

Das Paket *symboltable*, dessen Klassendiagramm in Abbildung 43 zu finden ist, enthält die Symboltabelle und die verschiedenen Elemente, die in dieser gespeichert werden können. In einer Symboltabelle sind die Bezeichner, die innerhalb eines Programms angelegt werden mit den zugehörigen Informationen gespeichert. Da sich in Puck Veränderungen der Bezeichner auch auf deren Verwendung auswirken sollen, werden die Symboltabellenelemente von deren Repräsentationen im Programm überwacht. Es war nötig die in Java enthaltene Klasse *Observable* zu überschreiben, da die Standardimplementierung nicht die Schnittstelle *Serializeable* implementiert. Dies ist aber nötig, da die Symboltabellenelemente sonst nicht abgespeichert werden können. In der Implementierung von *PuckObservable* werden die beobachtenden Elemente in einen *Vector*-Objekt gespeichert und beim Alarmieren – ein Aufruf der Methode *notifyObservers* – wird jeweils überprüft, ob auch alle Elemente des *Vector* als Beobachter angemeldet sind. Da die Java Standardklasse *Vector* serialisierbar ist, besteht so die Möglichkeit, auch die Beobachter eines Symboltabellenelements zu speichern.

In der Symboltabelle sind die Symboltabellenelemente auch in einem *Vector*-Objekt zusammengefasst. Für das Einfügen, Löschen und Herausfiltern besonderer Elemente sind spezielle Methoden implementiert worden, die den Umgang mit der Symboltabelle vereinfachen.

Die Symboltabellenelemente werden in drei Unterklassen unterschieden: Ein Element der Klasse *BlockStart* signalisiert, dass ein neuer Block beginnt. In *VarSE* sind alle Variablen und Parameter zusammengefasst[47]. Ein Objekt der Klasse *ProcedureSE* steht für eine Prozedur.

Die Symboltabelle ist nach dem Vorbild der „pulsierenden Liste“ (vgl. [Wi 86] S. 57-58) entwickelt worden, hierdurch könnte, wenn ein visuell erstelltes Programm in einem „Debugmodus“ ausgeführt würde, der aktuelle Inhalt der Symboltabelle ständig angezeigt werden. Das würde das Verständnis des Programmablaufes, das Testen und das Finden von Fehlern vereinfachen. In der vorliegenden Implementierung kommt kein Pulsieren zustande, da die Symboltabelle ja immer nur statisch für eine bestimmte Stelle des Programms generiert wird und nicht dynamisch während des Parsens lokale Parameter hinzugefügt und gelöscht werden müssen. Eine solche Erweiterung der Symboltabelle ist jedoch problemlos möglich.

Symboltabellen werden in Puck hauptsächlich für die Auswahl von Variablen, Parametern oder Prozeduren in den Anweisungen verwendet. Beim Einfügen von neuen Elementen wird überprüft, ob dies auch wirklich möglich ist. So muss z. B. ein Variablenname mit einem Buchstaben beginnen, danach dürfen Buchstaben und Zahlen folgen und außerdem darf der

[47] Die Buchstaben *SE* innerhalb der Klassenbezeichnung sind eine Kurzschreibweise für *SymboltableElement.*

Name im gewählten Kontext noch nicht vergeben sein. Bei Verstößen gegen diese Regeln öffnet sich ein Fenster mit einer erklärenden Fehlermeldung. Somit wird ein falsches Anlegen von Symboltabellenelementen schon im Ansatz verhindert.

5.2.4 Das Paket *expression*

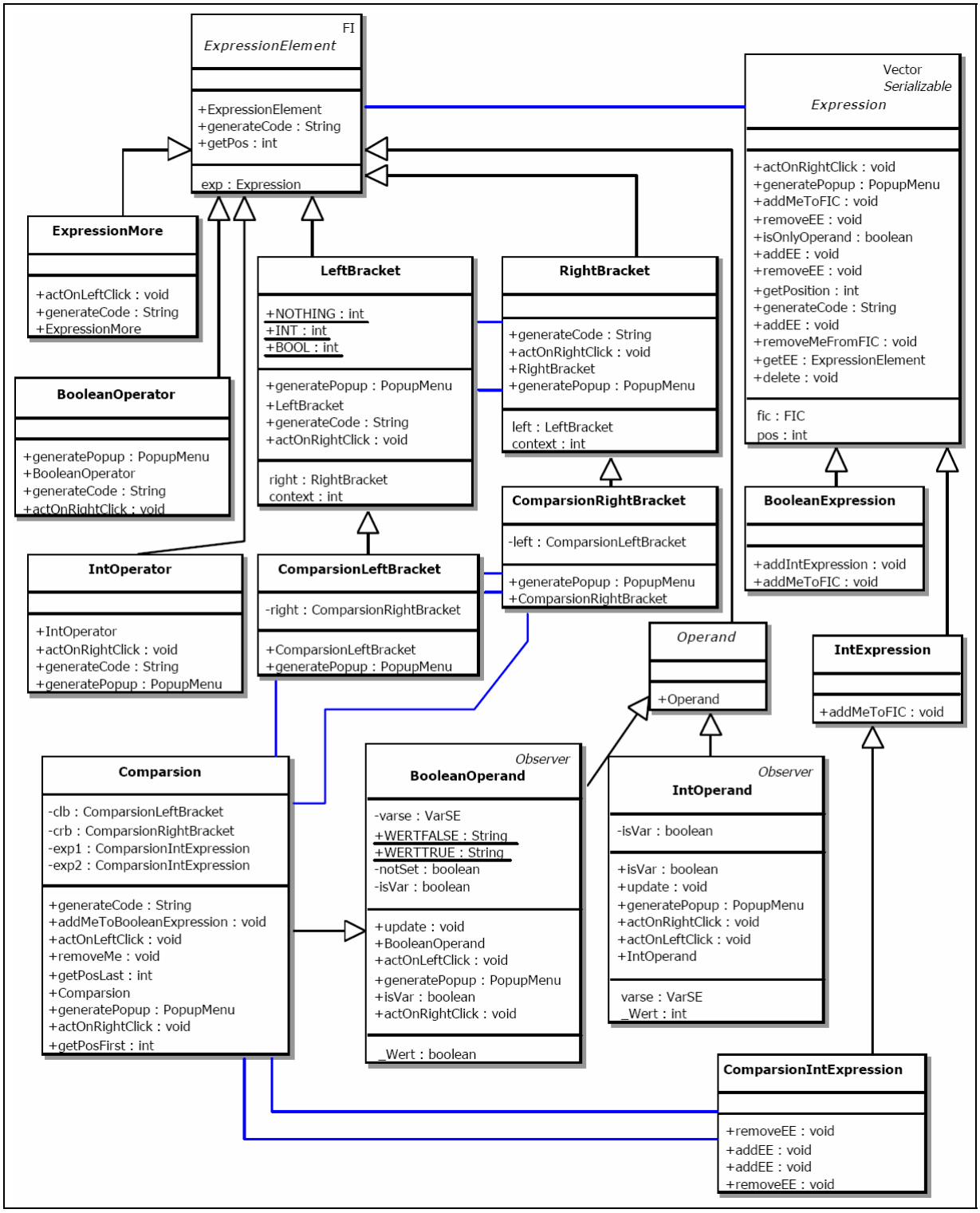


Abbildung 44: Klassendiagramm des Paketes *expression*

Eine *Expression*[48] beschreibt einen Ausdruck, bestehend aus verschiedenen *ExpressionElement*-Objekten, der mit Hilfe der Klasse *Vector* implementiert ist. Diese erben Attribute und Methoden von *FI*. Jedes *Expression*-Objekt kann mit der Methode *addMeToFIC* in ein *FIC* eingefügt werden. Hierfür werden alle *ExpressionElement*-Objekte des Ausdruckes einzeln als Teilbilder zu dem Baustein hinzugefügt. Innerhalb der Klasse *Expression* gibt es verschiedene Methoden, die den Umgang mit deren Elementen vereinfachen. Im Puck-System sind die Klassen *BooleanExpression* und *IntegerExpression* implementiert, die beide das Verhalten der abstrakten Oberklasse *Expression* erben und für einen Ausdruck des jeweiligen Datentyps stehen. In *Expression* sind auch Methoden wie *generateCode* oder *actOnRightClick* vorhanden, da das Verhalten des Ausdruckes meist nicht vom *FIC*, das diesen enthält gesteuert wird, sondern vom Ausdruck selbst.

ExpressionElement ist die Oberklasse aller Elemente, die Teil eines Ausdruckes sein können. Ein *ExpressionMore*-Objekt stellt einen Pfeil nach rechts dar, der auf einen Mausklick einem Ausdruck einen weiteren Operator und einen Operanden hinzufügt. *BooleanOperator* und *IntOperator* sind Klassen, welche die Operatoren der verschiedenen Datentypen repräsentieren. Bei diesen Klassen kann per Rechtsklick ein typgerechter Operator ausgewählt werden.

Des Weiteren gibt es noch verschiedene Klammer-Klassen. *LeftBracket* und *RightBracket* stehen für normale Klammern innerhalb eines Ausdruckes. Sie können verschoben und gelöscht werden. Eine öffnende Klammer kennt jeweils die zugehörige schließende und umgekehrt. *ComparsionLeftBracket* und *ComparsionRightBracket* schließen einen Vergleich innerhalb eines Boolean-Ausdrucks ein. Sie können im Ausdruck nicht verschoben werden und sind außerdem Grenzen für alle anderen Klammern, die jeweils beide entweder innerhalb oder außerhalb des Vergleichs liegen müssen.

Operand ist die Oberklasse aller Operanden, von ihr erben *IntOperand* und *BooleanOperand*. Ein Operand in einem Integer-Ausdruck kann ein Wert, eine Integer-Variable oder ein Integer-Parameter sein. Alle drei Möglichkeiten sind innerhalb der Klasse *IntOperand* realisiert, da sonst bei einer Veränderung, z. B. von einem Zahl zu einer Variablen, ständig Objekte gelöscht und neu angelegt werden müssten. Das Attribut *isVar* zeigt an, ob das *IntOperand*-Objekt aktuell ein Wert oder eine Variable/Parameter ist. Der Wert kann mit Hilfe der Attri-

[48] Expression ist das englische Wort für Ausdruck. Die Mischung von deutschen Namen mit den englischen Namen der Java-Standardklassen und des Paketes *newdnd* hätte den Quelltext unnötig verkompliziert. Alle Pakete, Klassen, Methoden und Attribute in Puck wurden soweit möglich mit englischen Wörtern bezeichnet. Einzige Ausnahme sind die vom Benutzer veränderbaren Attribute, da deren Name in der Attributtabelle verwendet wird.

buttabelle gesetzt werden. Eine Variable wird mit Hilfe des Kontextmenüs ausgewählt[49]. Wenn dies geschehen ist, so wird das *varse*-Attribut im Operanden auf das *VarSE*-Objekt in der Symboltabelle gesetzt, welches die ausgewählte Variable repräsentiert. Außerdem wird der *IntOperand* als Beobachter des Symboltabellenelements registriert, um so immer über Veränderungen an diesem informiert zu werden.

Die Klasse *BooleanOperand* ist der Klasse *IntOperand* sehr ähnlich. Der Wert kann hier nur „TRUE“ und „FALSE“ annehmen und die Variablen oder aktuellen Parameter müssen vom Typ *boolean* sein. Des Weiteren kann ein *BooleanOperand* auch ein Vergleich sein. Dieser wird mit der Klasse *Comparsion* realisiert, welche den Vergleichsoperator darstellt und zwei Objekte der Klasse *ComparsionIntExpression* sowie die Vergleichsklammern *Comparsion-LeftBracket* und *ComparsionRightBracket* rechts und links von sich initialisiert.

[49] Wie Ausdrücke verwendet werden ist in Kapitel 5.1.7. beschrieben.

5.2.5 Das Paket *bricks*

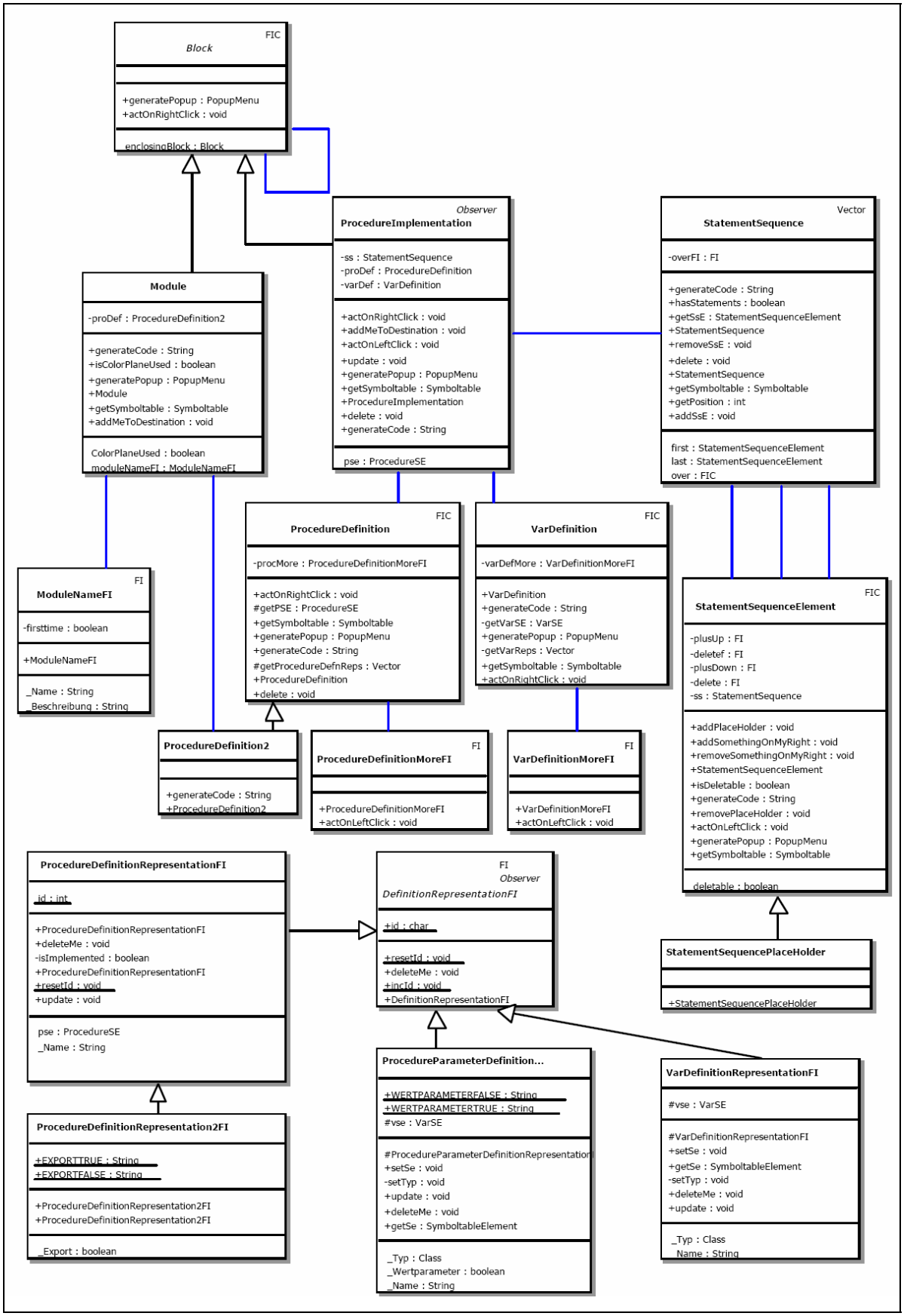


Abbildung 45: Klassendiagramm des Paketes *bricks*

In dem Paket *bricks* sind die Grundbausteine der visuellen Programmiersprache Puck implementiert. Dazu zählen das Modul und Prozeduren mit ihren Anweisungsfolgen[50], Parameter-, Variablen- und Prozedurdefinitionen. Das Klassendiagramm des Paketes *bricks* ist in Abbildung 45 dargestellt.

Die Klassen *Module* und *ProcedureImplementation* erben beide das Verhalten von *Block*. Diese abstrakte Oberklasse besitzt ein Attribut *enclosingBlock*, welches auf das Element zeigt, in dem dieser Block definiert wurde. Dies ist nötig, um beim Erzeugen von Symboltabellen auch auf den umschließenden Block zugreifen zu können.

Eine Instanz der Klasse *Module* beinhaltet ein *ModuleNameFI*-Objekt, in dem der Modulname eingetragen werden kann, und ein *ProcedureDefinition2*-Objekt zum Definieren von Prozeduren[51]. Auf die Definition globaler Variablen und eine Anweisungsfolge im Modul wurde bewusst verzichtet, um die Komplexität zu reduzieren und den Benutzer von Anfang an im prozeduralen Denken zu schulen.

Die Implementierung einer Prozedur wird durch die Klasse *ProcedureImplementation* realisiert. Diese beinhaltet jeweils ein Attribut der Klasse *ProcedureDefiniton*, *VarDefinition* und *StatementSequence*. Durch das *ProcedureDefinition*-Objekt können Prozeduren, durch das *VarDefiniton*-Objekt Variablen definiert werden. Beide Klassen besitzen ein Teilbild mit einem schwarzen Dreieck, welches es dem Benutzer ermöglicht, durch einen Klick mit der linken Maustaste eine neue Prozedur bzw. eine neue Variable zu definieren. In einem *StatementSequence*-Objekt sind mehrere *StatementSequenceElement*-Objekte zusammengefasst. Diese stellen die Anschlussstellen für Anweisungen bereit. Die Klasse *StatementSequence* besitzt die Attribute *overFI* und *over* in denen der Baustein und das spezielle Teilbild gespeichert sind, unterhalb derer das *StatementSequence*-Objekt eingefügt wurde. Des Weiteren sind in den Attributen *first* und *last* das erste und das letzte Element der Anweisungsfolge gespeichert. Die verschiedenen Methoden ermöglichen das Hinzufügen und Löschen der Elemente. Ein Objekt der Klasse *StatementSequenceElement* beinhaltet vier *FI*-Attribute, die jeweils unterschiedlich auf einen Klick mit der linken Maustaste reagieren. Ein Klick auf das Teilbild *plusUp* fügt oberhalb, ein Klick auf *plusDown* unterhalb des Angeklickten ein neues *StatementSequenceElement*-Objekt ein. Ein Element einer Anweisungsfolge kann nur dann ge-

---

[50] Anweisungsfolgen sind mit Hilfe der Klassen *StatementSequence* und *StatementSequenceElement* implementiert.

[51] *ProcedureDefinition2* ist eine Unterklasse von *ProcedureDefinition*, die an die Bedürfnisse eines Moduls angepasst ist. Eine Prozedur, die in einem Modul definiert ist, kann öffentlich oder privat sein, diese Unterscheidung gibt es bei einer Definition innerhalb einer Prozedur nicht.

löscht werden, wenn es nicht das einzige Element ist und nicht mit einer Anweisung verknüpft ist. Je nachdem ob ein Element löschbar ist beinhaltet es das Teilbild *delete* oder das Teilbild *deletef* und reagiert entsprechend auf einen Klick. Wenn eine Anweisung mehr als eine Zeile in Anspruch nimmt, so wird entsprechend der Zeilenanzahl eine Menge von *StatementSequencePlaceHolder*-Objekten eingefügt. Die Anzahl dieser Elemente passt sich dynamisch der Zeilenanzahl der Anweisung an. Das Löschen und Einfügen neuer Elemente durch den Benutzer ist bei den Objekten der Klasse *StatmentSequencePlaceHolder* nicht möglich, da rechts von ihnen ja auch kein Platz vorhanden ist.

Die abstrakte Klasse *DefinitionRepresentationFI* ist die Oberklasse aller Vereinbarungen[52]. Sie besitzt eine statische Klassenvariable *id*, die für Bezeichnungen von Objekten verantwortlich ist. Da auch direkt nach der Vereinbarung einer Variablen oder eines Parameters die Syntax korrekt sein soll, wird mit Hilfe von *id* automatisch ein Name generiert. Dazu wird der jeweilige Charakter in *id* durch die Methode *incId* um einen Buchstaben hoch gezählt, bis ein möglicher Name erreicht wurde. Nach Erreichen des ‚z' wird wieder bei ‚a' begonnen[53]. Wenn der Benutzer eine neue Datei erzeugt, wird durch *resetId* der Wert wieder auf das ‚a' gesetzt. Außerdem implementiert die Klasse *DefinitionRepresentation* das Interface *Observer*, da jede Definition einer Variablen, eines Parameters oder einer Prozedur mit einem Symboltabellenelement verknüpft ist, auf dessen Veränderungen auch die Repräsentation dieses Elementes in der Definition reagieren muss.

Objekte der Klasse *VarDefinitionRepresentationFI* definieren eine Variable mit einem Namen und einem Typ. Die entsprechenden Attribute finden sich in der Klasse und können vom Benutzer mit Hilfe der Attributtabelle verändert werden. Des Weiteren besitzt jedes *VarDefinitionRepresentationFI*-Objekt eine Verbindung zu einem *VarSE*-Objekt – der Repräsentation dieser Variablen in der Symboltabelle – um Veränderungen daran vornehmen zu können. Die Klasse *ProcedureParameterDefinitionRepresentationFI* erlaubt es, Parameter für eine Prozedur zu vereinbaren. Außer Name und Typ besitzen die Parameter noch ein weiteres Attribut, nämlich *_Wertparameter*. Hierdurch kann der Benutzer in der Attributtabelle den Übergabemechanismus des Parameters festlegen. Um bei Boolean-Attributen nicht nur die Werte „TRUE" und „FALSE" anzeigen zu können, wurde eine Vereinbarung getroffen: Eine stati-

---

[52] Mit Vereinbarungen sind an dieser Stelle die Deklarationen von Variablen, Parametern und Prozeduren gemeint.

[53] Wenn der Benutzer alle Buchstaben des Alphabetes verwendet hat, kann er keine neuen Variablen mehr anlegen. Aber in einem Anfängerprogramm sollte es keine 26 Variablen geben, die alle nur aus einem Buchstaben bestehen.

sche Klassenvariable kann definiert werden. Diese muss mit dem Namen des Booleanattributes – ohne den Unterstrich am Anfang – beginnen und mit dem Wort „TRUE“ oder „FALSE“ enden. Dabei muss der so zusammengesetzte Name des Attributes komplett aus Großbuchstaben bestehen und der Datentyp muss String sein. Der Wert dieser Variable wird dann in der Attributtabelle als Auswahlmöglichkeit angezeigt. Die Verwendung einer Auswahl wird am gerade vorgestellten Beispiel in Abbildung 46 dargestellt[54].

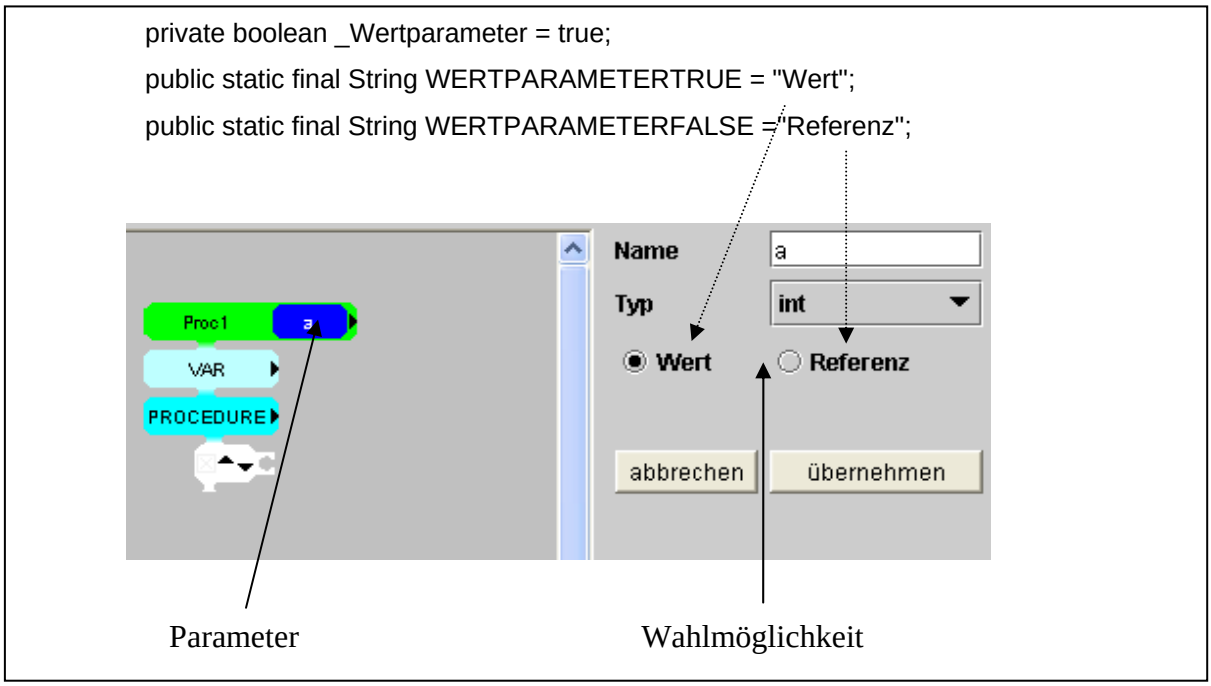


Abbildung 46: Beispiel für das Erstellen einer Auswahl in der Attributtabelle

Die Klasse *ProcedureDefinitionRepresentationFI* repräsentiert die Definition einer Prozedur. Diese besitzt einen Namen und eine Verknüpfung zum zugehörigen Symboltabellenelement. Beim Erstellen wird ein Standardname bestehend aus „Proc“ und einer darauf folgenden Zahl generiert. Diese wird hierbei solange erhöht, bis ein im Kontext gültiger Name entstanden ist. Die Klasse *ProcedureDefinitionRepresentation2FI* erweitert *ProcedureDefintionRepresentationFI*, indem sie einer Prozedur noch das Attribut *_Export* hinzufügt, durch welches der Benutzer einstellen kann, ob die Prozedur auch außerhalb des Moduls sichtbar sein soll.

[54] Auf die Getter- und Setter-Methode für das Attribut *_Wertparameter* wurde in der Darstellung verzichtet.

5.2.6 Das Paket *instructions*

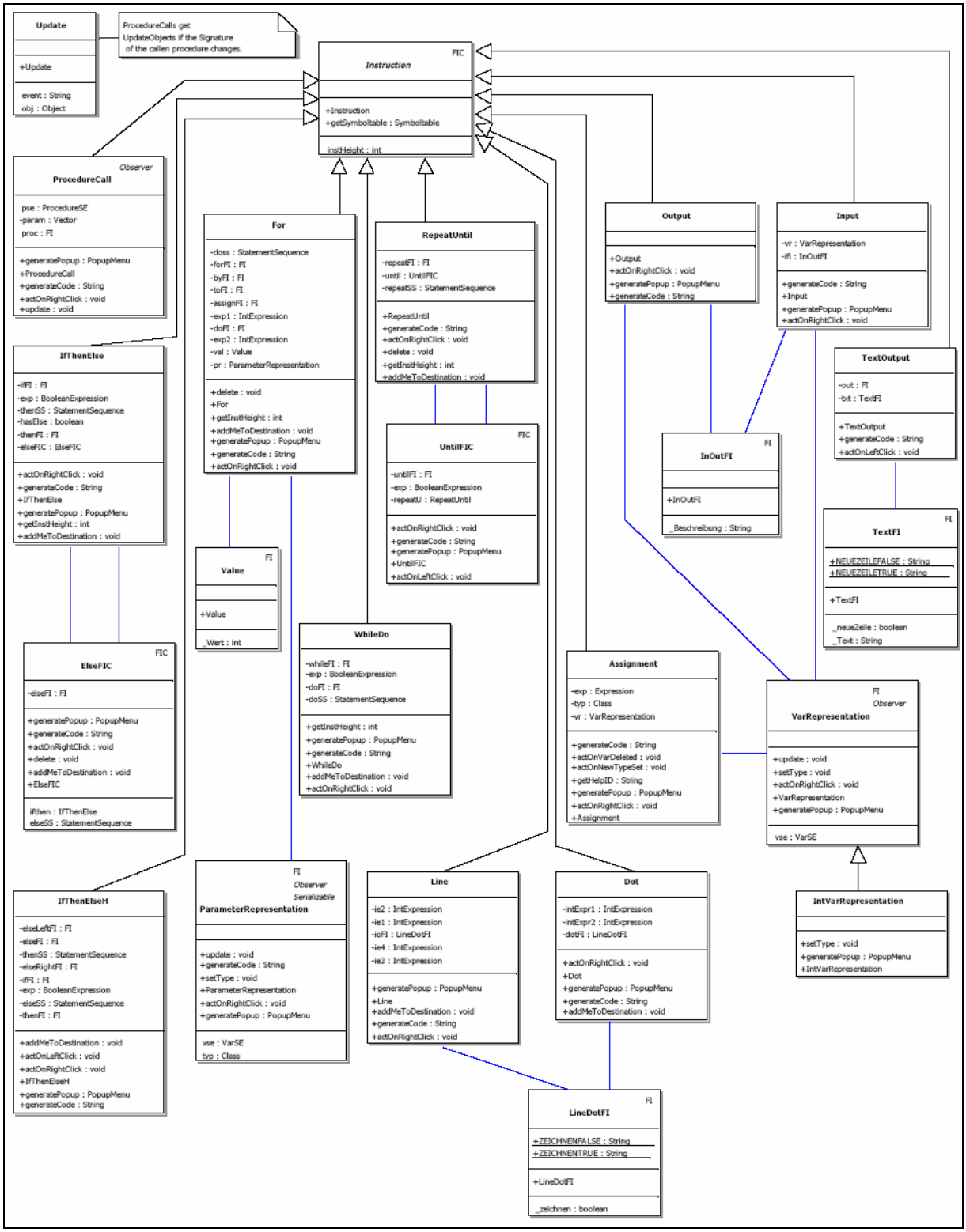


Abbildung 47: Klassendiagramm des Paketes *instructions*

Das Klassendiagramm des Paketes *instructions* ist in Abbildung 47 dargestellt. Es beinhaltet alle Anweisungen, die dem Benutzer in Puck zur Verfügung gestellt werden. Die abstrakte Oberklasse *Instruction* erbt die Methoden und Attribute von *FIC*. Sie enthält einen Konstruktor, welcher alle Objekte mit einem Teilbild initialisiert und eine Anschlussstelle auf der linken Seite der Anweisung bereitstellt. Hierdurch kann jedes *Instructions*-Objekt vom Benutzer mit einer Anweisungsfolge verbunden werden. Außerdem hat jede Instanz der Klasse *Instruction* eine bestimmte Höhe, die im Attribut *instHeight* gespeichert ist. Dies ist nötig, um bei Anweisungen, die mehr als eine Zeile benötigen, entsprechend viele Platzhalter[55] in der *StatementSequence* einzufügen. Die Methode *getSymboltable* gibt die Symboltabelle des mit der linken Anschlussstelle verknüpften Bausteins zurück. Dies kann, aufgrund der Verknüpfungsregeln, nur ein *StatementSequenceElement*-Objekt sein.

Die Anweisungs-Klassen in diesem Paket erben alle das Verhalten und die Attribute von *Instruction*. Die verschiedenen Prozeduren, wie *generateCode*, *actOnrightClick*, *addMeToDestination* und so weiter sind entsprechend des gewünschten Verhaltens der Bausteine implementiert. Außerdem haben die Anweisungs-Klassen verschiedene Attribute, die jeweils spezielle Teilbilder, mit der Anweisung verknüpfte *StatementSequence*-Objekte oder verwendete Ausdrücke beinhalten. Die anweisungsspezifischen Teilbilder sind teilweise als Klassen implementiert, um so dem Benutzer die Möglichkeit zu geben, über die Attributtabelle die Werte verschiedener Attribute zu verändern. Repräsentationen von Symboltabellenelementen implementieren die Schnittstelle *Observer* und werden so durch ihre *update*-Methode über Veränderung informiert. Die Klasse *Update* wird benötigt, um bei den Änderungen des Symboltabellenelementes einer Prozedur genau unterscheiden zu können, welcher Parameter gelöscht, hinzugefügt oder verändert wurde.

Im Folgenden soll am Beispiel der Dot-Anweisung, die einen Punkt zeichnet oder löscht, kurz das Erstellen von Bausteinen – speziell Anweisungs-Bausteinen – erläutert werden. Die Abbildung 48, am Ende des Kapitels, zeigt den fertigen Baustein mit seinen Elementen.

[55] Mit der Bezeichnung Platzhalter sind Objekte der Klasse *StamentSequencePlaceHolder* gemeint. Deren Verwendung wurde in Kap 5.2.5. behandelt.

5.2.6.1 Erstellen eines einfachen Anweisungs-Bausteins am Beispiel der Klasse *Dot*

Der Dot-Baustein soll den Aufruf der Oberon-2-Prozedur „ColorPlane.Dot(x,y,b)“ darstellen. Diese hat drei Parameter: x und y geben die Koordinaten des Punktes an und können als Integer-Ausdrücke dargestellt werden, das b kann den Wert 0 oder 1 übergeben bekommen, wobei 0 für das Löschen und 1 für das Zeichnen des Punktes steht. Hier bietet sich eine Auswahl in der Attributtabelle an.

Damit der Baustein über die Anschlussstelle einer Anweisung und die vorgefertigten Methoden verfügt, muss er Verhalten, Attribute und den Konstruktor von *Instruction* erben:

```
public class Dot extends Instruction
{
        public Dot()
        {
                super();
        }
}
```

Für die Integer-Ausdrücke, die die Werte für die Koordinaten darstellen, und das Bild, welches mit „Dot“ beschriftet sein soll, werden *intExpr1*, *intExpr2* und *dotFI* als Attribute vereinbart. Da auf diese von außen nicht zugegriffen werden muss, sind sie als „private“ deklariert. Im Konstruktor wird das Teilbild *dotFI* nun initialisiert und zum Baustein hinzugefügt. Hierbei muss die *ErrorWhileLoadingException* abgefangen werden. Diese wird nur dann ausgelöst, wenn die entsprechende Bilddatei nicht gefunden wurde. Des Weiteren wird noch ein *FI* mit dem Bild „instruction3.gif“ angefügt, das den abgerundeten Abschluss des Bausteines realisiert.

```
public class Dot extends Instruction
{
	private LineDotFI dotFI;
	private IntExpression intExpr1;
	private IntExpression intExpr2;
	public Dot()
	{
		super();
		try
		{
			dotFI = new LineDotFI();
			dotFI.setName("Dot");
			this.addFragmentImage(dotFI);
			this.addFragmentImage(FI.generateFI("gif/instructions/instruction3.gif"));
		} catch (ErrorWhileLoadingException e)
		{
			System.out.println("Exception in Class " + this.getClass() + " Exception: " + e);
		}
	}
}
```

Wird der Dot-Baustein vom Benutzer in den Arbeitsbereich gezogen, so wird die Methode *addMeToDestination* aufgerufen. Um der Anweisung bei diesem Vorgang die beiden Integer-Ausdrücke und ein Komma zwischen diesen hinzuzufügen ist es nötig, *addMeToDestination* zu überschreiben. Nach der Initialisierung werden *intExpr1* und *intExpr2* durch die Methode *addMeToFIC* dem Baustein an der jeweils richtigen Stelle hinzugefügt. Zwischen beiden wird noch ein Teilbild *comma* eingefügt.

```
public void addMeToDestination()
{
	intExpr1 = new IntExpression();
	intExpr2 = new IntExpression();
	intExpr1.addMeToFIC(this, 2);
	FI comma = FI.generateFI("gif/instructions/instruction4.gif");
	comma.setName(",");
	this.addFragmentImage(comma, 4);
	intExpr2.addMeToFIC(this, 5);
  }
```

Damit die Dot-Anweisung auf einen Klick mit der rechten Maustaste richtig reagiert, muss die Methode *generatePopup* aus *FIC* überschrieben werden. Diese bekommt ein *FI*-Objekt übergeben, das anzeigt, auf welches Teilbild geklickt wurde. Für die ersten beiden Bilder wird die *generatePopup*-Methode aus *FIC* aufgerufen, die ein Popup mit dem Eintrag „löschen“ erzeugt. Für alle anderen Elemente wird die *generatePopup*-Methode des jeweiligen Teilbildes aufgerufen. Somit werden für die Integer-Ausdrücke die jeweils richtigen Popups erstellt.

```
public PopupMenu generatePopup(FI fi)
{
	if (this.getPosition(fi) > 1)
		return fi.generatePopup();
	else
		return super.generatePopup(fi);
}
```

Die Methode *actOnRightClick* funktioniert ähnlich. Hier wird noch ein String mit dem *actionCommand* übergeben, über den festgestellt werden kann, welches Element aus dem PopupMenü ausgewählt wurde.

```
public void actOnRightClick(String actionCommand, FI fi)
{
	if (this.getPosition(fi) > 1)
		fi.actOnRightClick(actionCommand);
	else
		super.actOnRightClick(actionCommand, fi);
}
```

Die Methode *generateCode* ist für das Erstellen des Quelltextes zuständig. Sie bekommt zwei Parameter übergeben, *blanks* gibt die Anzahl der Leerzeichen an, um die die Anweisung eingerückt werden soll, *language* gibt die Programmiersprache an, für die Quelltext generiert werden soll. Zuerst wird eine Variable *code* angelegt, die einen leeren String enthält, zu dem nach und nach die Elemente einer „ColorPlane.Dot“-Anweisung in Oberon-2 hinzugefügt werden. Die Anweisung *FileMenuAL.getModule().setColorPlaneUsed(true)* setzt ein Boolean-Attribut innerhalb des Module-Bausteins auf „TRUE“. Dieses signalisiert, dass bei dem erstellten Programm das ColorPlane-Modul importiert werden muss. Die Methode *getBlanks* gibt einen String mit der gegeben Anzahl an Leerzeichen zurück.

Nachdem der Text „ColorPlane.Dot(“ zum Inhalt der Variable *code* hinzugefügt wurde, folgt der Aufruf der *generateCode*-Methoden in den Integer-Ausdrücken. Am Ende wird je nach-

dem, wie das BooleanAttribut *_zeichnen* in dem Bild *LineDotFI* gesetzt ist, noch eine „1“ oder eine „0“ für den letzten Parameter der Oberon-2-Prozedur eingesetzt.

```
public String generateCode(int blanks, int language) throws ErrorOnCodeGenerationException
{
	String code = "";
	if (language == 0)
	{
		FileMenuAL.getModule(). setColorPlaneUsed(true);
		code += getBlanks(blanks);
		code += "ColorPlane.Dot(";
		code += intExpr1.generateCode(blanks, language) + "," ;
		code += intExpr2.generateCode(blanks, language);
		code += dotFI.get_zeichnen() ? ",1)" : ",0)";
	}
	return code;
}
```

Außer dem Baustein muss auch noch das Teilbild erstellt werden, bei welchem durch den Benutzer eingestellt werden kann, ob der zu erstellende Punkt gezeichnet oder gelöscht werden soll. Hierfür muss eine *FI*-Unterklasse mit einem boolean-Attribut angelegt werden, auf das über die Attributtabelle zugegriffen werden kann. Da dieses Teilbild sowohl in der Klasse *Dot* als auch in *Line* verwendet werden kann, wird die zu erstellende Klasse *LineDotFI* genannt.

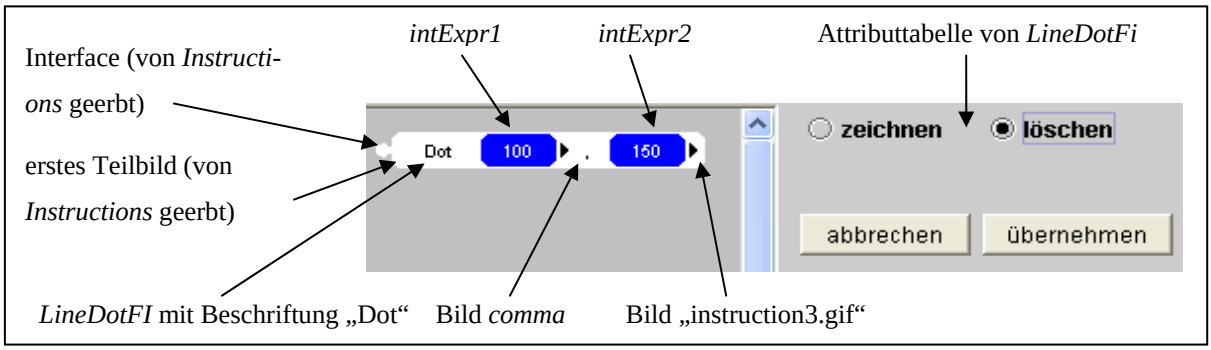


Abbildung 48: Die Elemente des Dot-Bausteins

```
public class LineDotFI extends FI
{
	public static String ZEICHNENFALSE = "löschen";
	public static String ZEICHNENTRUE  = "zeichnen";
	private boolean _zeichnen = true;

	public LineDotFI() throws ErrorWhileLoadingException
	{
		super("gif/instructions/instruction2.gif");
	}

	public boolean get_zeichnen()
	{
		return _zeichnen;
	}
	public void set_zeichnen(boolean zeile)
	{
		_zeichnen = zeile;
	}
}
```

Da ein einfaches „TRUE“ und „FALSE“ in der Attributtabelle den Benutzer an dieser Stelle verwirren würden, sind mit Hilfe der Variablen *ZEICHNENFALSE* und *ZEICHNENTRUE* auch noch die Bezeichnungen „löschen“ und „zeichnen“ für die Auswahl vereinbart worden[56]. Der Konstruktor initialisiert das Bild mit der Datei „instructions2.gif“. Für den Zugriff der Attributtabelle auf die *_zeichnen*-Variable wurden die Standard Getter- und Settermethoden implementiert. Abbildung 48 zeigt eine Dot-Anweisung mit den verschiedenen Elementen. Wenn diese mit einer Prozedur verknüpft wird, generiert sie den Quelltext:

```
ColorPlane.Dot(100,150,0)
```

[56] Die detaillierte Beschreibung einer Auswahl in der Attributtabelle ist in Kapitel 5.2.5 dargestellt.

6 Eine erste Erprobung in der Schule

Um Probleme und Verbesserungsmöglichkeiten im Puck-System zu finden, wurde es mit fünf Lehrern getestet. Fehler wurden sogleich verbessert und die Vorschläge der Pädagogen wurden – soweit möglich – sofort umgesetzt. Das so gereifte Produkt wurde dann im Unterricht einer zehnten Klasse des Carl-Zeiss-Gymnasiums Jena eingesetzt. Die Schüler hatten bereits Vorkenntnisse in Oberon-2, die es erleichterten, das System zu testen, ohne die Grundprinzipien der Programmierung erklären zu müssen. Innerhalb der ersten Stunde wurde Puck anhand eines Programms, welches die Summe von n Zahlen berechnet, vorgestellt. Die Schüler bekamen anschließend den Auftrag, das vorgezeigte Programm nachzubauen, um sich mit dem System vertraut zu machen. Darauf folgend sollten sie ein neues Programm erstellen, das die Fakultät einer Zahl iterativ berechnet.

Nach einer kurzen Eingewöhnungsphase hatten sich die Schüler an Puck gewöhnt und konnten das geforderte Programm umsetzen. Hierbei war es interessant zu beobachten, dass die Gymnasiasten schneller und effektiver durch Ausprobieren lernten, mit dem System umzugehen, als die Lehrer in der vorangegangenen Testphase. Außerdem halfen sich die Probanden gegenseitig, wenn der Lehrende mit Einzelnen beschäftigt war.

Um die Schüler mit dem Prozedurkonzept in Puck vertraut zu machen, wurde in der zweiten Stunde das Thema „Rekursion“ behandelt. Ein Programm zum rekursiven Berechnen der Fakultätsfunktion wurde vorgeführt. Nachdem die Schüler dieses selbst implementiert hatten, folgte die rekursive Berechnung der Summe der ersten n Zahlen. Diese Aufgabe musste wieder eigenständig gelöst werden. Interessierte Schüler blieben noch länger[57] und implementierten ein Programm zur rekursiven Berechnung der Fibonacci-Zahlen. Insgesamt wurde das System in der ersten Doppelstunde sehr gut angenommen[58].

Zu Beginn der zweiten Unterrichtseinheit – eine Woche später – wurde an der Tafel kurz der Line-Baustein mit seinen Parametern vorgestellt. Danach bekamen die Schüler die Aufgabe, zwölf vorgegebene Fadengrafiken mit Hilfe von Schleifen zu programmieren. Der Umgang mit dem Puck-System brauchte nicht wiederholt zu werden. Der Vorschlag, dass Parameter auch im generierten Quelltext verändert werden könnten, wurde nur von einem Schüler angenommen.

---

[57] Die Veranstaltung fand an einem Nachmittag statt.

[58] Am Ende der Stunde fragte ein Schüler, ob Puck jetzt immer auf ihren Rechnern installiert bleibe und sie auch weiterhin damit arbeiten könnten.

Im zweiten Teil der Doppelstunde wurde im Gespräch ein Algorithmus für den größten gemeinsamen Teiler (ggT) entwickelt. Nach einer Wiederholung zum ggT, in der auf verschiedene Methoden der Berechnung eingegangen wurde, entschieden sich die Schüler, den ggT mit Hilfe der Subtraktionsmethode zu programmieren. Alle setzten das gemeinsam entwickelte Programm auf dem Computer mit Hilfe von Puck um. Am Ende der Stunde folgte noch eine kurze Präsentation eines im gleichen System erstellten ggT-Programms, das die Berechnung ausführt und auch die grafische Repräsentation – das Unterteilens eines Rechteckes in eine maximale Anzahl gleichgroßer Quadrate – visualisiert[59].

Die Schüler arbeiteten motiviert mit dem System und wussten aufgrund ihrer Erfahrung mit Oberon-2 auch die Vorteile gegenüber textuellen Programmiersprachen zu schätzen. Nahezu alle erstellten Programme waren ausführbar[60], auch wenn sie manchmal noch logische Fehler enthielten und nicht die richtigen Ergebnisse lieferten. Die Installation auf den Rechnern funktionierte problemlos[61].

Insgesamt kann dieser kurze, stichprobenartige Test positiv gewertet werden. Gerade das Implementieren von Programmen, die nur skizzenartig an der Tafel entworfen wurden, konnte von allen Gymnasiasten fast ohne weitere Anleitung des Lehrers bewältigt werden. Die Schüler mussten nicht mit frustrierenden Syntaxfehlern umgehen und die Bedienung des Systems funktionierte problemlos.

Trotzdem wäre es wünschenswert, Puck in einem größeren Rahmen zu testen[62]. Hierfür ist eine Klasse mit Programmieranfängern geeignet. Folgenden Fragestellungen wären im Rahmen einer Evaluation interessant:

- Wie kommen Schüler ohne Programmiererfahrung mit dem System zurecht?
- Inwieweit beschäftigen sich die Probanden mit dem generierten Quelltext und verstehen diesen?
- Wie schwer fällt den Schüler der Übergang zum textuellen Programmieren?
- Wird der Lehrer entlastet?
- Kann mehr Stoff im Unterricht vermittelt werden?

---

[59] Das hier vorgestellte Programm wurde in dieser Arbeit in Kapitel 5.1.12.5 bereits erläutert.

[60] Einzige Ausnahme: Ein Modul mit einem zu langen Dateinamen konnten in POW! unter Windows 98 nicht ausgeführt werden.

[61] In der Schule wurden Personalcomputer mit den folgenden technischen Daten verwendet: Prozessor: 1 GHz, Arbeitspeicher: 128 MB, Betriebssystem Windows 98.

[62] Eine größere Testphase hätte den zeitlichen Rahmen dieser Arbeit gesprengt.

7 Fehler beim Erstellen von Programmen

7.1 Klassifizierung von Fehlern

In [www3] wird darauf hingewiesen, dass die Fehlerklassen beim Erstellen von Programmen teilweise nicht scharf voneinander abgrenzbar sind. Dies kommt daher, dass sich die Klassifizierungen in der Literatur am Compilerbau orientieren[63]. So werden die Fehler danach unterschieden, in welcher Phase des Kompilierens sie behandelt werden. Da aber gleiche Fehler in unterschiedlichen Phasen erkannt werden können, gibt es keine strikte Abgrenzung. Trotzdem soll im Folgenden eine kurze Klassifizierung zusammengetragen und mit selbstgewählten Beispielen veranschaulicht werden[64].

7.1.1 Lexikalische Fehler

Lexikalische Fehler entstehen, wenn ungültige Zeichen verwendet werden. Falsch geschriebene Schlüsselwörter, Operatoren oder Bezeichner können zu Wörtern führen, die in der verwendeten Programmiersprache nicht erlaubt sind. Dies kann schon während der lexikalischen Analyse als Fehler in einem Programm erkannt werden. Um aber sinnvolle Fehlermeldungen zu erstellen, werden nur wenige Fehler während der lexikalischen Analyse erkannt, da diese am Anfang des Kompilierens steht und deshalb nur wenige Informationen über das Programm bekannt sind. In [www3] werden lexikalische Fehler nicht einzeln klassifiziert, denn sie können auch während der folgenden Phasen des Kompiliervorgangs erkannt werden.

Beispiele:

VAR #loops:INTEGER;	Das Zeichen # ist an der verwendeten Stelle nicht zugelassen. #loops ist kein gültiges Wort in Oberon-2.
a:= 3² + 5;	Das Zeichen „²“ ist in der Sprache Oberon-2 nicht bekannt.

[63] Klassifizierungen aus Sicht des Compilerbaus sind in ([AH 00] S.161) und ([Lo 94] S.28-30) dargestellt.

[64] Die gewählten Beispiele beziehen sich im Folgenden auf die Programmiersprache Oberon-2.

7.1.2 Syntaxfehler

Verstößt ein Programm gegen die Regeln der Grammatik einer Sprache, so spricht man von Syntaxfehlern. Diese können durch die falsche Anordnung oder das Weglassen von Token[65] entstehen.

Beispiele:

a:=3 b:=a+5;	Der wohl typischste Syntaxfehler: nach der ersten Zuweisung wurde das Semikolon vergessen.
MODULE FunktionenZeichnen; PROCEDURE ProgMain*(); END ProgMain; BEGIN	Hier fehlt das END, welches das Ende des Moduls anzeigt.
IF a>0 ELSE a:=3 THEN a:=-3;	Bei dieser Anweisung wurden THEN und ELSE vertauscht.
a:=5+3*a);	Hier fehlt die öffnende Klammer.
a:=5 * + 3;	Zwischen dem * und dem + fehlt ein Operand.

[65] Als Token werden im Compilerbau syntaktische Elemente wie „BEGIN“, „;“, „33“ oder „summe“ bezeichnet, die dem Parser vom Scanner übergeben werden.

7.1.3 Semantische Fehler

Von semantischen Fehlern wird gesprochen, wenn trotz korrekter Syntax Anweisungen im jeweiligen Kontext ungültig oder nicht ausführbar sind.

7.1.3.1 Fehler der statischen Semantik

Statisch semantische Fehler können schon während des Kompilierens festgestellt werden. Sie betreffen meist Abhängigkeiten zwischen entfernt liegenden Quelltextabschnitten (vgl. [www3]).

Beispiele:

VAR fib:INTEGER ; BEGIN fib(233); …	Der Bezeichner fib wurde als Integer-Variable definiert und als Prozedur verwendet.

VAR radius,flaeche:INTEGER;
BEGIN
flaeche:=3.14 * radius * radius;
	Der Ausdruck auf der rechten Seite der Zuweisung ist durch den verwendeten Wert 3.14 nicht vom Datentyp Integer. Ein Typkonflikt entsteht.

VAR IF:INTEGER;	Laut Syntax ist die Verwendung des Bezeichners „IF“ nicht verboten. Trotzdem würde eine Variable mit diesem Namen einen Fehler verursachen, da „IF“ ein Oberon-2 Schlüsselwort ist.
PROCEDURE zeichne*(); … zeichne(100,100);	Hier wurde eine Prozedur „zeichne“ parameterlos deklariert. Der Aufruf enthält aber Parameter.

7.1.3.2 Fehler der dynamischen Semantik

Dynamisch semantische Fehler werden erst während der Laufzeit des Programms erkannt. Sie werden deshalb auch als Laufzeitfehler bezeichnet. In einigen Programmiersprachen können sie im Programm mit Hilfe von Ausnahmenbehandlungen abgefangen werden[66].

Beispiele:

In.Int(c); a:=3 DIV c;	Wird für die Variable c eine 0 eingegeben, so folgt in der nächsten Anweisung eine Division durch 0, die das Programm zum Abbruch bringt.

VAR a,b:INTEGER;
BEGIN
a:=30000;
b:=3*a; …
	Obwohl schon zur Ausführungszeit sicher ist, dass bei der Zuweisung zu der Variablen b eine Bereichsüberschreitung auftritt, wird der vorliegende Quellcode richtig übersetzt und ausgeführt[67]

7.1.4 Logische Fehler

Wenn das entwickelte Programm aufgrund falscher Implementierung unerwünscht reagiert, so wird von logischen Fehlern gesprochen. Diese können zu einem Programmabsturz oder einfach nur zu falschen Ergebnissen führen. Einen Überblick über logische Fehler gibt [Gr 90].

Beispiel:

PROCEDURE ProgMain*();
VAR
i: INTEGER;
faku: INTEGER;
BEGIN
faku:=1;
FOR i:=0 TO 7 BY 1 DO
faku:=faku*i
END;
Out.Int(faku,0)
END ProgMain;
	Die angegebene Prozedur soll die Fakultät von 7 berechnen. Da die FOR-Schleife mit 0 als Startwert beginnt, wird das Ergebnis faku mit 0 multipliziert und behält diesen Wert auch in allen weiteren Schleifendurchläufen.
	Die Zahl 0 wird als Ergebnis ausgegeben.

[66] Z. B. gibt es in der Programmiersprache Java „Exceptions“, mit deren Hilfe Laufzeitfehler bearbeitet werden können.

[67] Die größte Zahl des Datentyps INTEGER in der Programmiersprache Oberon-2 ist 32767.

7.2 Fehler bei Anfängern

Im Folgenden werden die Ergebnisse einer Studie zu Syntaxfehlern von Ripley und Druseikis vorgestellt. In dieser wurden die Fehler analysiert, die von Anfängern beim Erstellen von Pascal-Programmen gemachten wurden (vgl. [Ri 78]).

Die genannten Autoren haben festgestellt, dass nur 59,8 % der von den Testpersonen erstellten Programme syntaktisch und semantisch korrekt waren[68]. Wenn Fehler auftraten, dann meist vereinzelt[69]. Zu den 413 näher betrachteten Syntaxfehlern konnte der Compiler nur 136 korrekte Fehlermeldungen ausgeben[70]. Dies zeigt, dass Syntaxfehler durchaus ein Problem für Anfänger sind und die Fehlermeldungen des Compilers bei deren Beseitigung oft nicht weiterhelfen. Besondere Probleme bereiteten die Interpunktion und die richtige Verwendung der Begin-End-Strukturen.

Obwohl sich die Compiler in den letzten Jahren weiterentwickelt haben, ist die komplizierte Syntax von Programmiersprachen immer noch ein Grund für viele Fehler bei Anfängern. Dies wird auch in ([Ke 03] S. 8) mit dem Satz „One of the largest and most frustrating challenges for novice programmers is syntax.“ bestätigt. Besonders bei Anfängern sollte doch eine „Frustration“ verhindert werden.

Durch die Verwendung von visuellen Programmiersprachen ist es möglich, Syntaxfehler fast vollständig zu vermeiden. Inwieweit dies im Puck-System gelungen ist, soll im nächsten Kapitel analysiert werden.

---

[68] In der Studie wurden nur Fehler betrachtet, die das Ausführen des Programms verhinderten. Von 589 getesteten Programmen waren 237 fehlerhaft.

[69] 79,2 % der Anweisungen, die Fehler beinhalteten, hatten nur einen, 12,9 % hatten zwei Fehler.

[70] Von 413 Fehlermeldungen wurden 136 als „Accurate“, 260 als „Poor“ und 17 als „Incorrect“ eingeordnet.

7.3 Fehler in Puck

Im Puck-System wurde besonderen Wert darauf gelegt, gerade solche Fehler, die auf mangelnde Kenntnis der Syntax einer Programmiersprache zurückzuführen sind, zu verhindern.

Lexikalische Fehler sind in Puck nicht möglich, denn die textuelle Repräsentationen von Anweisungen, Prozeduren und Modulen werden vom System automatisch generiert. Bei vom Benutzer verwendeten Bezeichnern wird direkt nach ihrer Eingabe überprüft, ob diese der Sprachdefinition von Oberon-2 entsprechen und eventuell wie in Abbildung 49 eine erklärende Fehlermeldung ausgegeben.

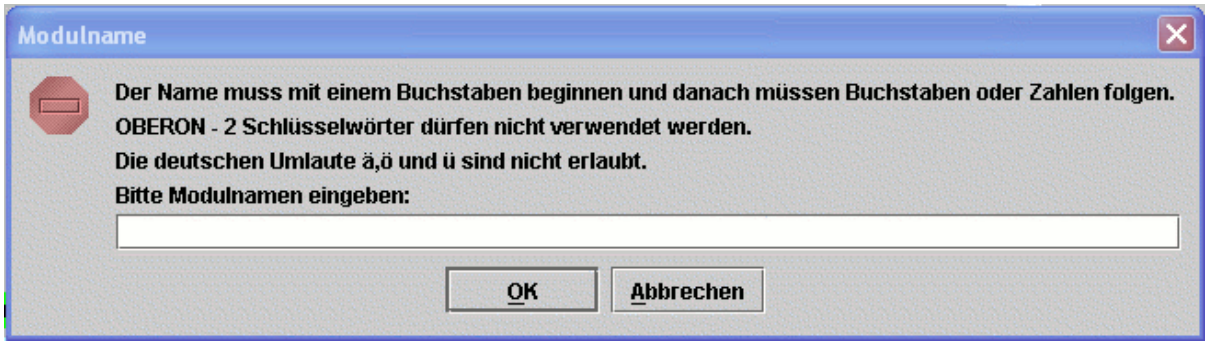


Abbildung 49: Fehlermeldung bei Eingabe eines nicht erlaubten Modulnamens

Auch Syntaxfehler werden von Puck verhindert[71]: Semikolon, BEGIN, END, IF, THEN, MOD, DIV und andere Token der Programmiersprache werden vom System selbst generiert und können dadurch nicht an einer falschen Stelle gesetzt oder weggelassen werden. Ebenso werden die richtige Verwendung von Klammern sowie die korrekte Reihenfolge von Operanden und Operatoren in Ausdrücken vom System erzwungen und können somit nicht zu Fehlern führen. Abbildung 50 zeigt ein in Puck erstelltes Programm mit dem dazugehörigen generierten fehlerfreien Oberon-2-Quelltext.

[71] Die einzige Möglichkeit mit Puck einen Fehler zu erzeugen ist das Verwenden von Hochkomma in Textausgaben.

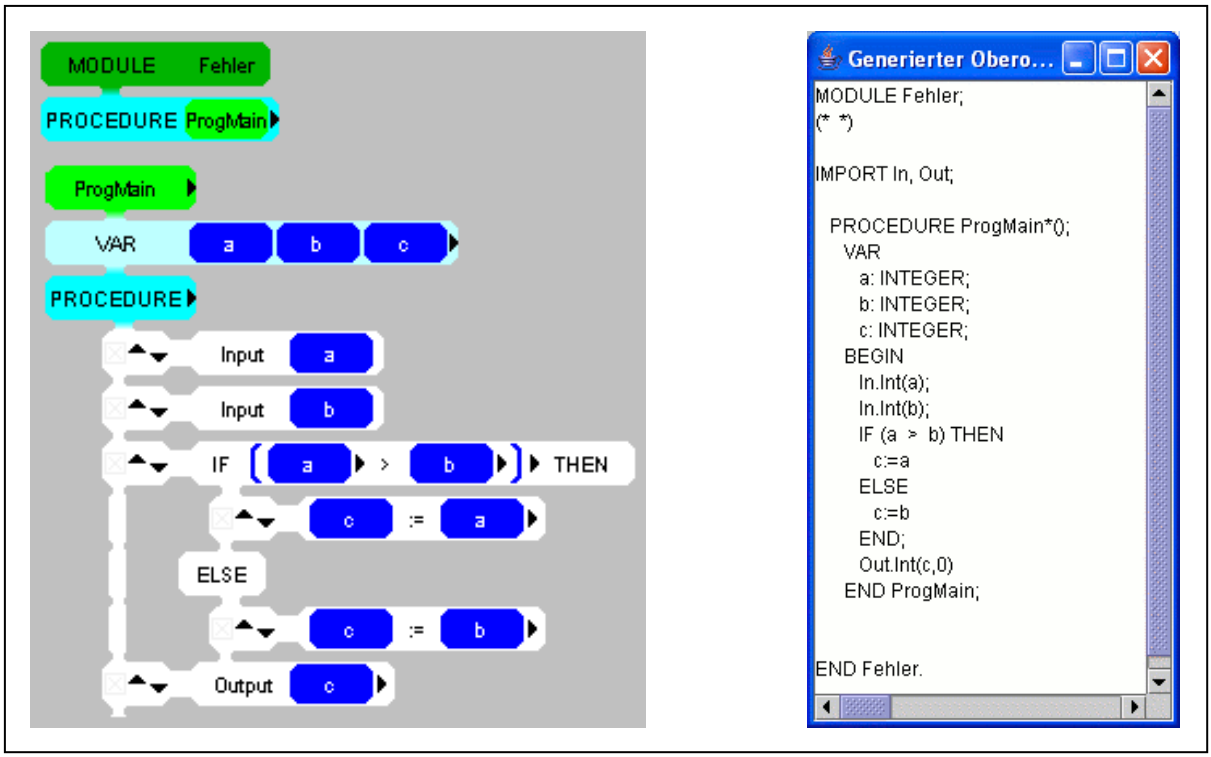


Abbildung 50: Ein mit Puck erstelltes Programm mit fehlerfrei generiertem Quelltext

Statisch semantische Fehler werden bei eingeschalteter Syntaxunterstützung zum Teil abgefangen. So kann in einer Anweisung nur auf im entsprechenden Kontext bekannte Variablen und Prozeduren zugegriffen werden. Bei der Anweisung der Prozedur „ProgMain“ in Abbildung 51 kann nur der Variablen a ein Wert zugewiesen werden, da keine anderen Variablen oder Parameter definiert sind.

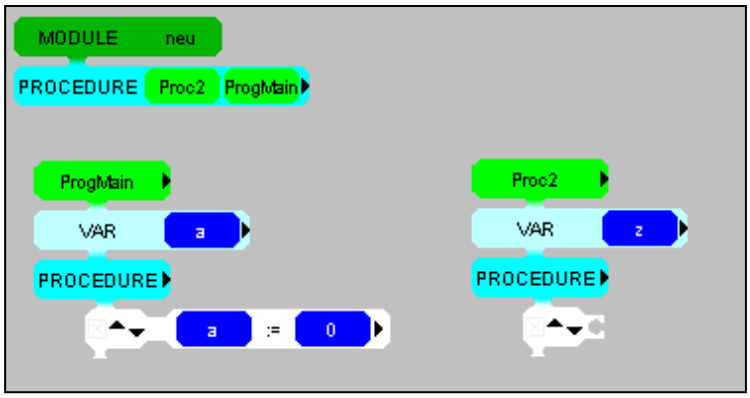


Abbildung 51: Vermeiden von statisch semantischen Fehlern in Puck

Das Verschieben einer Anweisung von einer Prozedur in eine andere kann allerdings zu statisch semantischen Fehlern führen. Wird die Zuweisung „a:=0“ aus Abbildung 51 an die Prozedur „Proc2“ angehangen, so wird dort, wie in Abbildung 52 zu sehen ist, die Variable a

verwendet, obwohl sie gar nicht definiert ist. Dies führt dazu, dass beim Kompilieren ein Fehler ausgegeben wird.

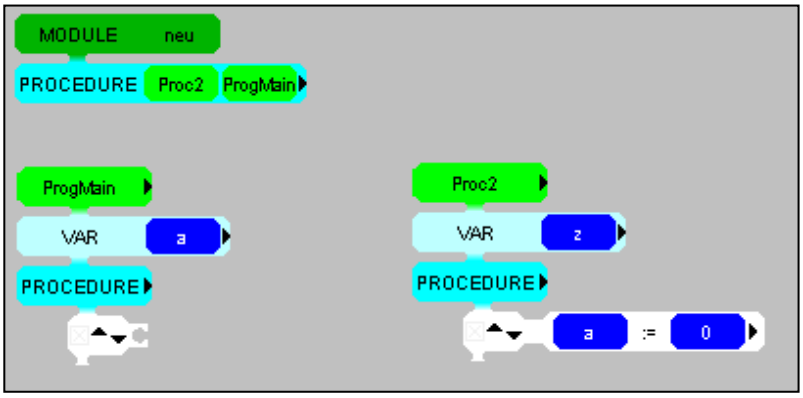


Abbildung 52: Statisch semantische Fehler, die von Puck nicht verhindert werden

Somit wird dem Benutzer aber ermöglicht, eine Anweisung unverändert in einem neuen Kontext einzufügen und dann anzupassen. Er kann die verwendeten Variablen, Parameter und Prozeduren, die nun nicht mehr gültig sind, durch im neuen Kontext vorhandene ersetzen. Die Operatoren, Klammern und sonstige Kontextunabhängige Elemente müssen so nicht neu erstellt werden.

An allen Stellen, an denen ein Element rot markiert ist, fehlt noch etwas. Das rote Feld in Abbildung 53 zeigt an, dass in der FOR-Schleife eine Laufvariable eingesetzt werden muss. Solange dies nicht geschehen ist, wird der generierte Quelltext an der entsprechenden Stelle nur das Wort „int“ enthalten und so einen Fehler beim Kompilieren auslösen.

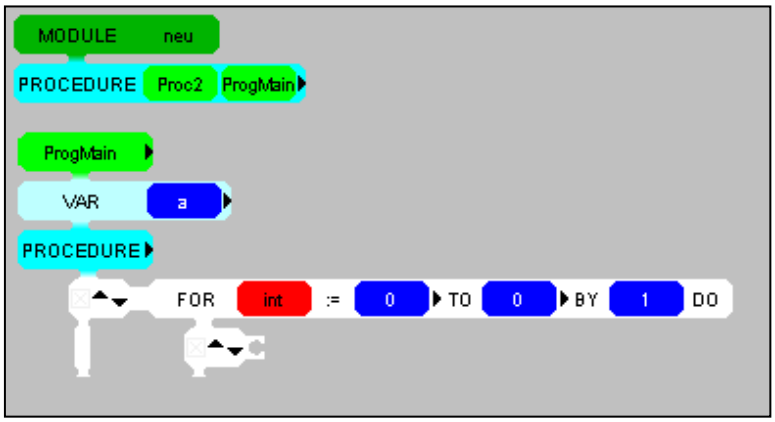


Abbildung 53: Beispiel für mögliche Fehler in Puck durch rot markierte Elemente

Da dynamisch semantische Fehler erst zur Ausführungszeit gefunden werden können, ist ein Abfangen mit dem Puck-System nicht realisiert worden. Ähnlich verhält es sich mit logischen Fehlern. Um diese zu vermeiden, müsste die Programmiersprache die gewünschten Ergebnisse des zu überprüfenden Programms kennen.

Mit Puck können lexikalische und syntaktische Fehler nahezu vollkommen, statisch semantische teilweise vermieden werden. Somit wurde eine große Hürde für Anfänger, nämlich die der Frustration durch Syntaxfehler, genommen. Der Anwender kann sich auf das Erlernen der Konstrukte imperativer Programmierung konzentrieren und braucht nicht auf Semikolons, BEGIN- und END-Strukturen oder die richtige Schreibweise von Schlüsselwörtern zu achten. Wenn später eine textuelle Programmiersprache erlernt werden soll, so sind zumindest schon die verschiedenen Konstrukte bekannt. Somit wird der anfängliche „Schock“ bei Programmierneulingen, der bei [Ro 03] auf das Bewältigen verschiedener schwieriger Aufgaben zurückgeführt wird, entschärft, indem das Erlernen der Syntax bereits von „Puck“ übernommen wurde[72].

[72] In ([Ro 03] S.148) sind folgende schwierige Aufgaben für Programmieranfänger angegeben: generelle Orientierung, wofür Programme da sind und was man mit ihnen machen kann; ein imaginärer Maschinenbegriff: Ein Computermodell, das mit der Programmausführung verbunden ist; Notation des Programms, das heißt Syntax und Semantik einer Programmiersprache; Schemas und Pläne zum Lösen von Aufgaben; praktische Fähigkeiten wie Planen, Entwickeln, Testen, Fehlersuchen.

8 Eignung von Puck für den Einsatz in Schulen

Im Folgenden soll anhand von zwei Quellen diskutiert werden, inwieweit Puck als Programmiersprache für Einsteiger bzw. als Programmiersprache für die Schule geeignet ist.

8.1 Eignung als eine Programmiersprache für Einsteiger

Anforderungen an eine Programmiersprache für Einsteiger sind anschaulich in ([Re 04] S. 10-12) zusammengefasst. Die dort diskutierten Punkte sollen im Folgenden für das erstellte System Puck analysiert werden.

- Komplexität reduzieren

In der vorliegenden Implementierung wird nicht nur die Komplexität des speziellen Computers, des Betriebssystems und der Ein- und Ausgabegeräte reduziert, es wird sogar von der Programmiersprache Oberon-2 abstrahiert, indem ein visuelles System geschaffen wurde, bei dem nur geringe Kenntnisse des Schülers über die Syntax der Programmiersprache nötig sind.

- Komplexität verstecken

Damit ist eine Behandlung von „Maschinenaktionen“ als „black boxes“ gemeint, das heißt, dass es für den Programmierer nicht wichtig ist, wie komplexe Befehle funktionieren, sondern nur, was sie bewirken. Dieses Prinzip findet sich in dem vorliegenden System Puck wieder. Der vom Programm erstellte Oberon-2-Code enthält die Komplexität einer Programmiersprache. Der Benutzer des Systems muss diesen Quelltext und den Vorgang der Generierung aber nicht im Detail verstehen, er braucht ihn nur vom Computer ausführen zu lassen, um sein erstelltes Programm zu testen. Um einen Umstieg auf die textuellen Programmiersprachen zu vereinfachen ist es dem Anwender in Puck möglich, sich den Quelltext anzeigen zu lassen. Somit kann für den fortgeschrittenen Benutzer aus der „black box“ des textuellen Programms eine „glass box“ werden.

- Visualisierung

Die visuelle Darstellung in Puck hilft dem Nutzer, die Struktur und Semantik eines Programms besser zu verstehen. Auch wenn eine Visualisierung der Programmausführung in Puck aufgrund des begrenzten Zeitrahmens dieser Arbeit noch nicht vorgesehen ist, so wäre

dies doch eine wünschenswerte Erweiterung, die das Verständnis der Vorgänge, die in einem Computer ablaufen, bei den Benutzern von Puck erhöhen würde.

- Kleiner Sprachumfang

In ([Re 04] S. 11) heißt es: „Für erste Programmierschritte empfiehlt sich eine Programmiersprache mit kleinem, rasch überblickbarem Sprachumfang. Die Sprache sollte Schritt für Schritt erlernt werden können.“ Diese Forderung ist in der erstellten visuellen Programmiersprache durch die wenigen Bausteine erfüllt, die außerdem noch in einem Optionen-Menü zu- und weggeschaltet werden können.

- Einfache Programmierumgebung

Das einfache Programmiersystem, das mit Puck geschaffen wurde, hilft erheblich dabei, Fehler zu vermeiden[73]. Einerseits können nur Bausteine zusammengefügt werden, die auch laut Syntax zusammenpassen, andererseits werden typische Anfängerfehler, wie vergessene Semikolons oder falsche Klammersetzung, vollständig verhindert.

- Alltagsorientierte Aufgabenstellung

Die Aufgaben, die mit Puck bewältigt werden können, sind genau diejenigen, die auch bei einem normalen Einstieg in die Programmierung mit einer textuellen Programmiersprache bearbeitet werden[74]. Außerdem ist es leicht möglich, das System so zu erweitern, dass durch wenige neu zu erstellende Bausteine ein kleines, sich auf dem Bildschirm bewegendes Objekt, sei es nun ein Roboter, eine Schildkröte oder ein Marienkäfer, mit Hilfe von Puck programmiert werden kann[75].

[73] Welche Fehler mit Puck vermieden werden können ist ausführlich in Kapitel 7.3 dargestellt.

[74] In [Fo 02a] wird eine Einführung in die Programmiersprache Python für die Schule vorgestellt. Diese ist in vier Bereiche aufgeteilt. Alle Aufgaben des ersten Bereiches können mit Puck gelöst werden.

[75] Die Begriffe Roboter, Schildkröte und Marienkäfer beziehen sich auf bestehende Programmiersysteme mit visueller Repräsentation der Programmausführung.

8.2 Eignung als eine Programmiersprache für die Schule

Für die Wahl einer Programmiersprache für die Schule können nach [Fo 02b] die folgenden Kriterien herangezogen werden:

- Die Programmiersprache eignet sich auf Grund ihrer Komplexität für den Schulunterricht.
- Mit der Programmiersprache lassen sich Probleme aus verschiedenen Gebieten bearbeiten.
- Vor dem breiten Einsatz einer Programmiersprache wurden Erfahrungen an einzelnen Schulen gewonnen.
- Für die Schülerinnen und Schüler liegen geeignete Unterrichtsmaterialien vor.
- Auf der Grundlage der am allgemein bildenden Gymnasium erlernten Programmiersprachen soll ein Einarbeiten in andere Sprachen möglich sein.
- Das Programmiersystem sollte kostenlos verfügbar sein.

Im Folgenden soll dargestellt werden, inwieweit die oben angegebenen Kriterien erfüllt sind bzw. noch erfüllt werden können.

Das entwickelte Programmiersystem ist mit den wenigen Bausteinen von geringer Komplexität. Dies lässt es für Programmieranfänger geeignet erscheinen. Für komplexe Programme, die sich auch in verschiedenen Länderlehrplänen wieder finden, werden Datenstrukturen wie Arrays, Records oder sogar Objekte benötigt[76]. Deshalb sollte „Puck“ in der Schule hauptsächlich für den Einstieg genutzt werden. Um die ganze Bandbreite der Programmierung unterrichten zu können, kann Puck mit der textuellen Programmiersprache Oberon-2 kombiniert werden. Durch den generierten Quelltext gewöhnt sich der Schüler schon während des Umgangs mit der visuellen Programmiersprache an die textuelle Repräsentation des Programms.

Die Beispiele in Kapitel 5.1.12 zeigen, dass sich mit Puck nicht nur mathematische Probleme wie das Berechnen der Fibonacci-Zahlen lösen lassen, sondern z. B. auch Fadengrafiken gezeichnet werden können. Da das System aber für den Einstieg in die Programmierung geeignet sein sollte, waren Einschränkungen, wie z. B. die Verwendung von nur zwei Datentypen nötig, um Anfänger nicht zu überlasten. Wird Puck für die Einführung in die Programmierung

[76] Im Lehrplan des Landes Thüringen sind z. B. Zeigertypen im Themenbereich „Listen und Bäume“ des Leistungskurses Informatik der 11. Klasse vorgesehen (vgl. [Th 99] S. 36).

verwendet, so können anschließend mit Oberon-2 Aufgaben aus verschiedenen Gebieten bearbeitet werden. Da auch andere Programmiersprachen eine Pascal-ähnliche Syntax haben, fällt ein Übergang zu Programmiersystemen wie z. B. Delphi, ebenfalls nicht schwer.

Erfahrungen mit dem Einsatz von Puck in der Schule liegen derzeit nur punktuell vor[77]. Eine längere Erprobungszeit mit der Umsetzung von Verbesserungsvorschlägen wäre wünschenswert.

Auch Unterrichtsmaterialien zum Puck-System sind derzeit noch nicht vorhanden. Diese könnten nach einer Testphase erstellt werden.

Da Konstrukte wie Variablen und Prozeduren in allen imperativen Programmiersprachen vorkommen, ist der Schüler in der Lage, sich auch in andere Systeme einzuarbeiten.

Puck wird unter der „GNU General Public License“ (vgl. [www2]) veröffentlicht und den interessierten Schulen kostenlos zur Verfügung gestellt.

[77] Ein kurzer Test in einer Schule ist in Kapitel 6 beschrieben.

9 Zusammenfassung

Visuelle Programmiersprachen sind einfach zu erlernen und motivieren insbesondere Einsteiger, die schneller mit grafischen als mit textuellen Elementen umgehen können. Außerdem können frustrierende Syntaxfehler nahezu vollständig vermieden werden. Der größte Nachteil visueller Programmiersprachen, komplexe Inhalte unübersichtlich und schwer durchschaubar darzustellen, tritt bei Anfängern nicht auf, da deren Übungsprogramme nur einen geringen Umfang haben[78].

In der vorliegenden Arbeit wurde, nach einer vorherigen Erhebung von Anforderungen, eine visuelle Programmiersprache für den Einsatz in Schulen entwickelt. Hierfür wurde ein Prototyp implementiert, auf dessen Basis ein Produkt geschaffen wurde, das sich an den Wünschen der befragten Lehrer orientiert.

Mit Hilfe des so entstandenen Systems können Anfänger ein Programm per Drag and Drop entwickeln. Hierfür wurden verschiedene visuelle Elemente zur Verfügung gestellt. Elf Anweisungs-Bausteine können unter dem Menüpunkt „Optionen“ zu- und weggeschaltet werden. Dadurch bekommt der Benutzer die Möglichkeit, das System Schritt für Schritt, also Anweisung für Anweisung, zu erkunden. An den Bausteinen können mit Hilfe von Attributtabellen und Kontextmenüs spezifische Einstellungen vorgenommen werden. Zu einem visuell erstellten Programm kann sich der Benutzer den Quelltext in der Sprache Oberon-2 generieren und anzeigen lassen. Außerdem ist es möglich, den so erstellten textuellen Code in einer „.mod“-Datei zu speichern, die dann mit der Programmierumgebung POW! geöffnet, kompiliert und ausgeführt werden kann. Die Quelltextgenerierung wurde exemplarisch für die Programmiersprache Oberon-2 implementiert, da diese an den meisten Thüringer Schulen eingesetzt wird. Es ist mit vertretbarem Aufwand möglich, textuellen Code für eine andere Programmiersprache generieren zu lassen.

Die Verwendung des entwickelten Systems und seine Implementierung stellt das Kapitel 5 ausführlich dar. Der Verlauf einer Testphase mit Schülern wird in Abschnitt 6 beschrieben. In Kapitel 7 wurde auf Fehler, die beim Erstellen von Programmen auftreten, eingegangen und gezeigt, inwieweit diese bei der Verwendung der visuellen Programmiersprache Puck vermieden werden. Die Frage, ob das System für den Einsatz in Schulen geeignet ist, wurde in Kapitel 8 anhand von zwei Quellen diskutiert.

---

[78] Vor- und Nachteile visueller Programmiersprachen sind in ([Sc 98] S. 69-109) und ([Ko 04] S. 7-11) ausführlich zusammengestellt.

Des Weiteren liegt dieser Arbeit eine CD bei, die das gesamte entwickelte System mit Klassendiagrammen, Dokumentation und Beispielen enthält[79]. Außerdem sind die Quellcodes auf der CD enthalten, so dass die visuelle Programmiersprache weiterentwickelt und an spezielle Anforderungen angepasst werden kann.

In Puck ist es möglich, mit nur wenigen Bausteinen einfache Programme zu erstellen. Somit können Benutzer die Verwendung von Variablen, Prozeduren und Anweisungen erlernen, ohne gleich mit der komplexen Syntax einer Programmiersprache konfrontiert zu werden. Bei anspruchsvolleren Aufgaben muss der Anfänger auf eine textuelle Programmiersprache umsteigen, mit welcher er sich vorher durch das ständige Generieren von Quelltext vertraut machen kann.

10 Ausblick

Trotz positiver Reaktionen aus der Lehrerschaft kann das Produkt einer Diplomarbeit nicht mit der Arbeit professioneller Softwarefirmen konkurrieren. Deshalb werden im Folgenden einige Punkte aufgezählt, die den Rahmen dieser Arbeit gesprengt hätten, aber trotzdem Thema beziehungsweise Ziel weiterer Entwicklungen sein könnten.

- Das erstellte Produkt konnte nur kurz getestet werden. Eine weitere Testphase in einer Schule könnte sich an diese Arbeit anschließen, damit Probleme im Umgang mit dem System analysiert und beseitigt werden können[80]. Außerdem wäre es im Rahmen dieser Testphase möglich, Unterrichtsmaterialien zu erstellen.
- Um die Zeitspanne zu verlängern, in der die visuelle Programmiersprache im Unterricht eingesetzt werden kann, wäre es nötig, z. B. weitere einfache Datentypen sowie Funktionsprozeduren bereitzustellen.
- Für die Realisierung komplexerer Programme wäre es von Vorteil, wenn der Platz im Arbeitsbereich besser ausgenutzt werden könnte. Es erscheint zweckmäßig, dem Benutzer die Möglichkeit zu geben, fertig implementierte Prozeduren zusammenzuklappen.

[79] Der Inhalt der CD ist in Anhang 1 beschrieben.

[80] Bei der Evaluierung einer solchen Testphase wären die in Kapitel 6 vorgestellten Fragen von Interesse.

- Um die Eingabezeit für Ausdrücke zu verkürzen wäre es sinnvoll, für fortgeschrittene Benutzer eine weitere Möglichkeit zu schaffen, bei der ein Ausdruck textuell eingegeben werden kann, aber trotzdem auf Syntaxfehler überprüft wird.
- Durch die Übersetzung des visuellen Programms in andere Sprachen als Oberon-2 könnte eine größere Zielgruppe erreicht werden. Hierdurch wäre es außerdem möglich, im Unterricht auf verschiedene textuelle Programmiersprachen einzugehen.
- Ein Button, über den der erstellte textuelle Code direkt kompiliert und ausgeführt werden kann, würde den Umweg über ein anderes System ersparen. Hierdurch wären die Schüler allerdings nicht mehr mit der textuellen Programmiersprache konfrontiert und so würde der Umstieg wahrscheinlich schwerer fallen.
- Wenn das visuell erstellte Programm mit Puck kompilierbar und ausführbar ist, wäre auch ein Debugmodus wünschenswert. Dieser könnte mit einer visuellen Repräsentation der Symboltabelle den Speicherinhalt anschaulich darstellen und so die Programmanalyse vereinfachen.
- Um das System besser an spezielle Anforderungen anpassen zu können, wäre es wünschenswert, für Module, die in einer textuellen Programmiersprache erstellt wurden, Bausteine zu generieren, beziehungsweise Bausteine mit XML zu beschreiben. Somit könnte das System, ohne spezielle Kenntnisse über dessen Implementierung, erweitert werden.
- Eine Möglichkeit, die Konzepte der Objektorientierung vollständig mit Hilfe von visuellen Programmiersprachen umzusetzen, wäre die Kombination des entwickelten Systems mit einem Klassendiagrammeditor. Die dort deklarierten Methoden könnten dann mit Hilfe von Puck implementiert werden.
- Um die entwickelte visuelle Programmiersprache auch über den deutschsprachigen Raum hinaus bekannt zu machen, wäre es sinnvoll, Menüs und Optionen in mehreren Sprachen bereitzustellen.

In dieser Arbeit ist es gelungen, eine leicht zu erweiternde, visuelle Programmiersprache für den Einsatz in Schulen zu erstellen. Diese sollte in Zusammenarbeit mit Lehrern und Schülern ausführlich getestet werden, um herauszufinden, welche der aufgezeigten Möglichkeiten der Weiterentwicklung sinnvoll umzusetzen sind.

Literatur

[AH 00] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: Compilers: Principles, Techniques, and Tools; Reading; 2000.
ISBN 0-201-10194-7

[Ba 98] Balzert, Helmut: Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung; Berlin; 1998. ISBN 3-8274-0065-1

[Cl 03] Claus, Volker; Schwill, Andreas: Duden Informatik: Ein Fachlexikon für Studium und Praxis; 3.Auflage; Mannheim; Leipzig; Wien; Zürich; 2003.
ISBN 3-411-10223-0

[Fo 02a] Fothe, Michael: Problemlösen mit Python / Thüringer Institut für Lehrerfortbildung, Lehrplanentwicklung und Medien (Hrsg.): Reihe Materialien; Heft 72; Bad Berka; 2002.
ISSN: 0944-8705

[Fo 02b] Fothe, Michael: Informatische Bildung an den allgemein bildenden Schulen in Thüringen; In: Konzepte zur informatischen Bildung an allgemein bildenden und berufsbildenden Schulen (Workshop); Dresden; 2002.
http://osg.informatik.tu-chemnitz.de/EFI/workshop2002/programm/fothe.pdf
Stand: 14.12.2004

[Ga 97] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns: Elements of Reusable Object-Oriented Software; 11. Auflage; Reading; 1997.
ISBN 0-201-63361-2

[Gr 90] Grams Timm: Denkfallen und Programmierfehler; P. Schnupp; H.Strunz (Hrsg.); Berlin; 1990.
ISBN 3-540-52039-2

[Ke 03] Kelleher, Caitlin; Pausch Randy: Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers; Pittsburgh; 2003.
http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-137.pdf
Stand: 14.12.2004

[Ko 04] Kohl, Lutz: Studienarbeit: Konzepte der visuellen Programmierung und ihrer Einsatzmöglichkeiten an Schulen; nicht erschienen; 2004.

[Lä 04] Lärz, Alexander: Studienarbeit: Entwicklung einer dynamischen Drag & Drop Umgebung zum Erstellen von visuellen Programmen; nicht erschienen; 2004.

[Lo 94] Louden, Kenneth C.: Programmiersprachen: Grundlagen, Konzepte, Entwurf; Kempten; 1994.
ISBN: 3-929821-03-6

[Mü 95] Mühlbacher, Jörg R.; Leisch, Bernhard; Kreuzeder, Ulrich: Programmieren mit Oberon-2 unter Windows; München; Wien; 1995.
ISBN: 3-446-18406-6

[Re 04] Reichert, Raimond; Nievergelt, Jürg; Hartmann, Werner: Programmieren mit Kara: Ein spielerischer Zugang zur Informatik; Berlin; Heidelberg; New York; 2004.
ISBN 3-540-40362-0

[Ri 78] Ripley, G. D.; Druseikis, F. C.: A statistical analysis of syntax errors; Computer Languages 3; 1978; S. 227-240.

[Ro 03] Robins, Anthony; Rountree, Janet; Rountree, Nathan: Learning and Teaching Programming: A Review and Discussion; In: Computer Science Education; 2003; Vol. 13; Nr. 2; S. 137-172.

[Sc 98] Schiffer, Stephan: Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten; Bonn; 1998.
ISBN: 3-8273-1271-X

[Th 99] Thüringer Kultusministerium (Hrsg.): Lehrplan für das Gymnasium Informatik; Erfurt; 1999.

[Vi 71] Victor, Walther: Shakespeare – Ein Lesebuch für unsere Zeit; 12. Auflage; Berlin; Weimar; 1971.

[Wi 86] Wirth, Niklaus: Compilerbau: Eine Einführung; 4.Auflage; Stuttgart; 1986.
ISBN 3-519-32338-9

[www1] Downloadadresse POW!
http://www.fim.uni-linz.ac.at/pow/Download.htm; Stand: 14.12.2004

[www2] GNU General Public License
http://www.gnu.org/copyleft/gpl.html; Stand: 14.12.2004

[www3] Klassifikation von Fehlern (FH München, FB 07 Informatik/Mathematik; Schiedermeier - Vorlesung "Programmieren I")
http://www.informatik.fh-muenchen.de/~schieder/programmieren-1-ws96-97/fehlerarten.html; Stand: 14.12.2004

[www4] The Java Help System
http://java.sun.com/products/javahelp/; Stand: 14.12.2004

[www5] Java Help System User's Guide
http://java.coe.psu.ac.th/Extension/JavaHelp2.0/javahelp-2_0-guide.pdf; Stand: 14.12.2004

[www6] How to use SpringLayout
http://java.sun.com/docs/books/tutorial/uiswing/layout/spring.html; Stand: 14.12.2004

[www7] Download Java 2 Platform, Standard Edition, Version 1.4.2
http://java.sun.com/j2se/1.4.2/download.html; Stand: 14.12.2004

Anhang 1 – CD Inhalt

Beispiele

Im Ordner Beispiele befinden sich verschiedene ".puk"-Dateien. Diese können über den Menüpunkt "Datei" - > "Öffnen" mit dem Programm "Puck.jar" geöffnet werden.

Dokumentation

Im Ordner Dokumentation sind drei Unterordner zu finden.

Unter „Dokumentation der Klassen“ befindet sich eine Sammlung von HTML-Dateien, die die entwickelten Klassen mit ihren Methoden ausführlich beschreiben. Einstiegspunkt ist die Datei "index.html"

Unter „Klassendiagramme“ befinden Sich die Klassendiagramme im ".pdf"-Format. Diese Dateien können mit dem Adobe Reader dargestellt werden. Im Einzelnen sind hier folgende Dateien zu finden:

basis.pdf

bricks.pdf

expressions.pdf

instructions.pdf

Packages.pdf

symboltable.pdf

system.pdf

Unter „UML Dokumentation“ ist die Dokumentation der Klassen im HTML-Format mit den Klassendiagrammen als Scaleable-Vector-Graphics kombiniert. Um SVG-Dateien mit einem Browser darstellen zu können muss ein Plugin installiert werden. Dieses Plugin ist für den Internet Explorer unter Installation bereitgestellt. Der Einstiegspunkt für die UML Dokumentation ist die Datei "index.html". Hier kann mit Hilfe der Links in der textuellen Dokumentation, der Klassendiagramme oder der Navigationsleiste durch die Dokumentation des gesamten Projektes gesurft werden.

Falls beim Öffnen der „HTML“-Dateien im oberen Bereich keine Klassendiagramme zu sehen sind, so ist das erforderliche Plugin nicht installiert.

Installation

Im Ordner Installation sind drei Unterordner zu finden.

Im Ordner „IE Plugin zum betrachten von SVG-Dateien“ befindet sich die Installationsdatei mit der ein Plugin für den Internet Explorer installiert werden kann. Mit dessen Hilfe ist es möglich, SVG-Dateien in HTML-Seiten mit dem Internet Explorer darzustellen.

Unter „java 1.4.2_01“ befindet sich die Java-Laufzeitumgebung mit dessen Hilfe das Projekt entwickelt wurde. (Achtung die ausgelieferte Version von "Puck" funktioniert noch nicht mit neueren Java Versionen.)

Im Ordner „POW!“ sind die Installationsdateien für die Oberon-2 Entwicklungsumgebung für Windows zu finden, die installiert werden muss, wenn die in Puck erstellten Programme ausgeführt werden sollen. (Achtung die Dateien in diesem Ordner bitte zuerst auf die Festplatte kopieren und dann die Installation starten!)

lib

Im Ordner „lib“ befinden sich zwei Dateien, die für die Hilfe im Puck-System verantwortlich sind. Wenn sich eine Kopie dieses Ordners an der gleichen Stelle wie die Datei "Puck.jar" befindet, so kann beim Ausführen des Programms auf die Hilfe zugegriffen werden. (Die Inhalte der Hilfeseiten werden in einer Projektarbeit entwickelt und sind auf der CD nicht enthalten.)

Weiterentwicklung

Unter dem Ordner Weiterentwicklung befindet sich das gesamte entwickelte Projekt mit allen Quelltexten. Bei der Entwicklung wurde mit der Programmierumgebung Eclipse gearbeitet. Die Verzeichnisstruktur entspricht einem Eclipse-Projekt.

Die Datei "Copying.txt"

In der Datei „Copying.txt“ sind die Lizenzvereinbarungen für Puck zu finden. Puck wird unter der GNU-General Public License veröffentlicht.

Die Datei „Entwurf und Implementierung einer visuellen Programmiersprache für den Einsatz in Schulen.pdf“

In dieser Datei befindet sich diese Arbeit.

Die Datei "Grundlagen beim Programmieren mit Puck.doc"

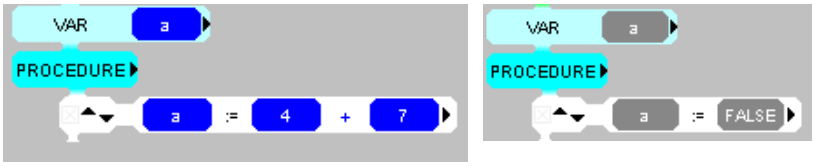
Die Datei „Grundlagen beim Programmieren mit Puck.doc“ enthält eine Seite voll Informationen, die beim Umgang mit Puck zu beachten sind. Diese Seite sollte ein Programmierneuling beim Start mit Puck immer griffbereit haben.

Die Datei "Puck.jar"

Die Datei „Puck.jar“ ist das Hauptprodukt dieser Arbeit / CD. Sie kann in Windows-Systemen mit einem Doppelklick geöffnet werden, wenn eine Java-Laufzeitumgebung installiert ist. (Unter Linux kann die Datei mit dem Konsolenaufruf „java -jar Puck.jar“ geöffnet werden.)

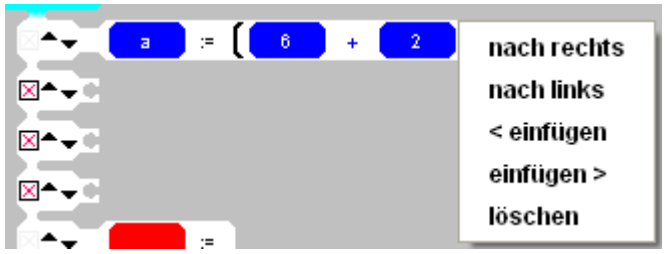
Anhang 2 – Grundlagen beim Programmieren mit Puck

1. Integer-Variablen oder -Werte (ganze Zahlen) sind blau, Boolean-Variablen oder -Werte (wahr oder falsch) sind dunkelgrau dargestellt.

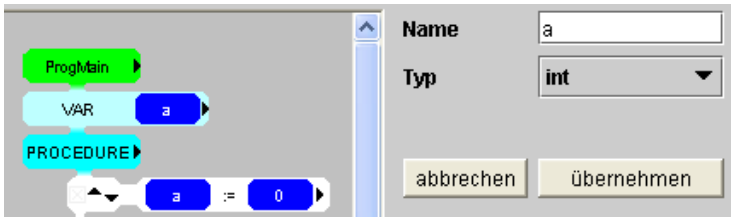


2. Ein rot markiertes Element in einem Baustein zeigt an, dass noch etwas fehlt.

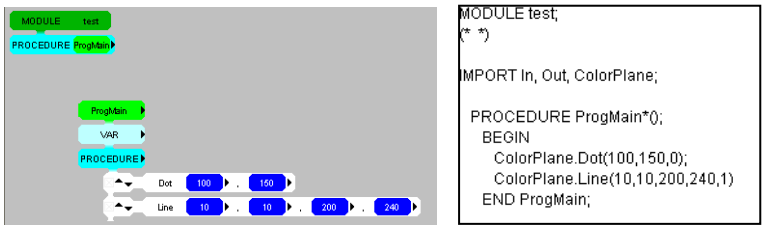
3. Mit der rechten Maustaste öffnet man Kontextmenüs, wenn diese verfügbar sind.



4. Mit der linken Maustaste öffnet man die Attributtabelle des Elements, falls diese vorhanden ist.

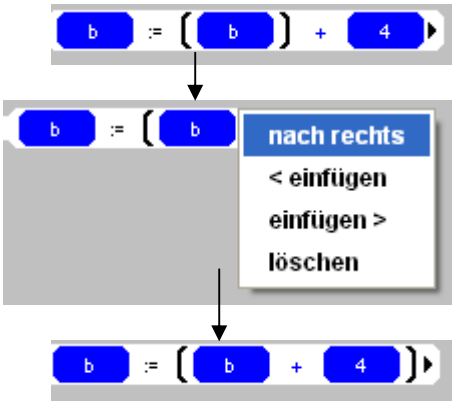


5. Unter „Datei“ kann man das visuelle Programm speichern, unter „Quelltext in Datei speichern“ kann man den Quelltext speichern.



6. An allen Stellen, an denen Ausdrücke verwendet werden, muss mindestens ein Operand des richtigen Datentyps stehen. (Dieser kann nicht gelöscht werden.)

7. Klammern werden mit dem Menüpunkt „klammern“ um einen Operanden gesetzt und dann mit Hilfe des Kontextmenüs an die richtige Stelle bewegt.



Anhang 3 – Generierter Quelltext der Beispielprogramme

Das Programm „Fibonacci-Zahlen“

```
MODULE FibonacciZahlen;
(*  *)

IMPORT In, Out;


  PROCEDURE fibonacci*(n: INTEGER; VAR erg: INTEGER);
    VAR
      hilf1: INTEGER;
      hilf2: INTEGER;
    BEGIN
      IF (n  <  2) THEN
        erg:=n
      ELSE
        fibonacci(n-1,hilf1);
        fibonacci(n-2,hilf2);
        erg:=hilf1+hilf2
      END
    END fibonacci;

  PROCEDURE ProgMain*();
    VAR
      n: INTEGER;
      ergebnis: INTEGER;
    BEGIN
      In.Prompt('ganze Zahl:');
      In.Int(n);
      fibonacci(n,ergebnis);
      Out.String('Die ');
      Out.Int(n,0);
      Out.String('. Fibonacci Zahl ist : ');
      Out.Int(ergebnis,0)
    END ProgMain;


END FibonacciZahlen.
```

Das Programm „Türme von Hanoi“

```
MODULE TuermeVonHanoi;
(*  *)

IMPORT In, Out;

  PROCEDURE Turm*(n: INTEGER; Start: INTEGER; Ablage: INTEGER; Ziel: INTEGER);
    BEGIN
      IF (n  >  0) THEN
        Turm(n-1,Start,Ziel,Ablage);
        Out.String('Bitte nehmen Sie die oberste Scheibe von Turm ');
        Out.Int(Start,0);
        Out.String(' und legen Sie diese auf Turm ');
        Out.Int(Ziel,0);
        Out.String('.');Out.Ln;
        Turm(n-1,Ablage,Start,Ziel)
      END
    END Turm;

  PROCEDURE ProgMain*();
    VAR
      n: INTEGER;
    BEGIN
      In.Int(n);
      Turm(n,1,2,3)
    END ProgMain;


END TuermeVonHanoi.
```

Das Programm Fadengrafik

```
MODULE Fadengrafik;
(*  *)

IMPORT In, Out, ColorPlane;

  PROCEDURE ProgMain*();
    VAR
      a: INTEGER;
    BEGIN
      FOR a:=0 TO 400 BY 5 DO
        ColorPlane.Line(0,400-a,a,0,1)
      END
    END ProgMain;


BEGIN
  ColorPlane.Open;
END Fadengrafik.
```

Das Programm „Funktionen Zeichnen“

```
MODULE FunktionenZeichnen;
(*  *)

IMPORT In, Out, ColorPlane;

  PROCEDURE KoordinatensystemZeichnen*();
    BEGIN
      ColorPlane.Line(0,400,800,400,1);
      ColorPlane.Line(400,0,400,800,1)
    END KoordinatensystemZeichnen;

  PROCEDURE zeichnePunkt*(x: INTEGER; y: INTEGER);
    BEGIN
      ColorPlane.Dot(x+400,y+400,1)
    END zeichnePunkt;

  PROCEDURE ProgMain*();
    VAR
      x: INTEGER;
      y: INTEGER;
    BEGIN
      KoordinatensystemZeichnen();
      FOR x:=(-100) TO 100 BY 1 DO
        y:=2*x+50;
        zeichnePunkt(x,y);
        IF (x MOD 10 = 0) THEN
          Out.String('x= ');
          Out.Int(x,0);
          Out.String('                y= ');
          Out.Int(y,0);
          Out.String('');Out.Ln
        END
      END
    END ProgMain;

BEGIN
  ColorPlane.Open;
END FunktionenZeichnen.
```

Das Programm „Größter gemeinsamer Teiler“

```
MODULE ggtNicomachos;
(*  *)

IMPORT In, Out, ColorPlane;

  PROCEDURE RechteckZeichnen*(x: INTEGER; y: INTEGER);
    BEGIN
      ColorPlane.Line(0,0,x,0,1);
      ColorPlane.Line(0,0,0,y,1);
      ColorPlane.Line(x,0,x,y,1);
      ColorPlane.Line(0,y,x,y,1)
    END RechteckZeichnen;

  PROCEDURE ggt*(a: INTEGER; b: INTEGER; VAR erg: INTEGER);
    VAR
      c: INTEGER;
    BEGIN
      RechteckZeichnen(a,b);
      IF (b = 0) THEN
        erg:=a
      ELSE
        IF (a  >  b) THEN
          ggt(a-b,b,erg)
        ELSE
          ggt(a,b-a,erg)
        END
      END
    END ggt;

  PROCEDURE ProgMain*();
    VAR
      a: INTEGER;
      b: INTEGER;
      Ergebnis: INTEGER;
    BEGIN
      In.Prompt('Zahl1: ');
      In.Int(a);
      In.Prompt('Zahl2: ');
      In.Int(b);
```

```
        ggt(a,b,Ergebnis);
        Out.String('Der ggt ist: ');
        Out.Int(Ergebnis,0)
      END ProgMain;

BEGIN
  ColorPlane.Open;
END ggtNicomachos.
```

Erklärung

Hiermit erkläre ich, dass ich diese Diplomarbeit selbstständig verfasst und dabei keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Jena, den 14.12.2004 Lutz Kohl