Eine Visuelle Programmiersprache: Prograph

Björn Kolbeck - bjko@cs.tu-berlin.de
Eduard Weiss - eduardw@cs.tu-berlin.de
Holger Endert - endert@cs.tu-berlin.de

Visuelle Sprachen und Entwicklungsumgebungen
Wintersemester 2004/2005
TU-Berlin

11. Februar 2005

Inhaltsverzeichnis

1 Einleitung - was ist Prograph **2**

1.1 Visuelle Programmiersprache 2
1.2 Datenflußorientierte Sprache 2
1.3 Objektorientierte Sprache 3

2 Die Entwicklungsumgebung **3**

3 Sprachelemente **5**

3.1 Klassen 5
3.1.1 Klassendefinition und Vererbung 5
3.1.2 Attribute 6
3.1.3 Methoden 7
3.2 Universals 8
3.3 Persistents 9
3.4 Datenfluss - Implementierung der Methoden 10
3.4.1 Aufbau von Methoden 10
3.4.2 Die Arität einer Methode 10
3.4.3 Aufruf von Methoden und vordefinierten Operationen . 10
3.4.4 Erstellung und Manipulierung von Objekten 12
3.4.5 Schleifen 13
3.4.6 Listen und Listeniteration 15
3.4.7 Fallunterscheidung 17

4 Implementierungsbeispiel **18**

4.1 Programmaufbau 18
4.2 Ablauf 20
4.3 `start` 20
4.4 `auswahl` 20
4.5 `auswahlEntscheiden` 21
4.6 `getAdressListe` 22
4.7 `Adressbuch/einfuegen` und `Adressbuch/eingabe` 22
4.8 `Adressbuch/ausgabe` und `printAdresse` 23
4.9 `Adressbuch/aendern` 23
4.10 `Adressbuch/loeschen` 24
4.11 `Adressbuch/suchen` und `Adresse/passt` 24

5 Fazit und Bewertung **25**

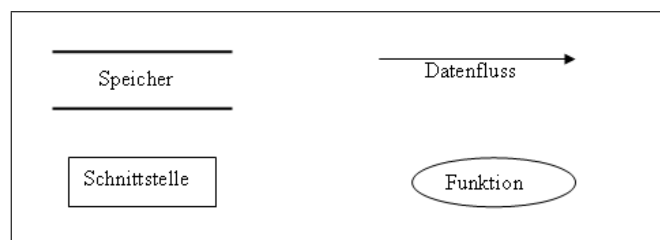

Abbildung 1: Elemente von Datenflussdiagrammen

1 Einleitung - was ist Prograph

Die Sprache PROGRAPH wurde von der Pictorius Group entwickelt. Sie unterstützt eine völlig neue Art und Weise der Entwicklung von Softwareapplikationen auf dem Macintosh und auf Windows-Plattformen. PROGRAPH vereinigt drei wichtige Programmierkonzepte in sich: Visuelle, objektorientierte und datenflußorientierte Programmierung. Außerdem stellt es eine komfortable Entwicklungsumgebung zur Verfügung.

1.1 Visuelle Programmiersprache

Bei PROGRAPH handelt es sich um eine visuelle Programmiersprache im eigentlichen Sinne. Es gibt zwei verschiedene Arten von Icons in PROGRAPH. Die einen dienen zur grafischen Darstellung von Programmelementen in der Programmierumgebung. Die anderen sind Programmierelemente. Nur mit Hilfe dieser Symbole kann die Funktion einer Methode beschrieben werden. Sie ersetzen den Programmtext der traditionellen Programmiersprachen. Die Symbole werden später genauer erläutert.

1.2 Datenflußorientierte Sprache

Die Struktur einer Methode ähnelt einem Datenfluß-Diagramm. Die Ähnlichkeit besteht im Aufbau: Jede Methode hat ein Speicher, Datenfluss, Schnittstellen und Funktionen. Einige Nachteile bei Datenfluß-Diagrammen gegenüber einer Programmiersprache sind, dass sie keinen Kontrollfluss modellieren. Außerdem sind zyklische Strukturen und programmiertechnische Details, z.B. Fallunterscheidung, nicht darstellbar. Damit Prograph als eine Programmiersprache angesehen wird, haben die Entwickler diese Konzepte integriert.

1.3 Objektorientierte Sprache

PROGRAPH unterstützt objektorientiertes Programmieren. Klassen bestehen aus Attributen und Methoden. Dem Programmierer steht eine vordefinierte Klassen- und Methodenbibliothek zur Verfügung. So kann eine Applikation komfortabel auf diese aufsetzen. Allerdings besteht PROGRAPH nicht auf einen objektorientieren Progammierstil, sondern ermöglicht vielmehr auch eine prozedurale Programmierung. Dabei wurde nicht, wie bei anderen Hybridsprachen (z.B. Turbo-Pascal) ausgehend von einer prozeduralen Programmiersprache, ein objektorientierter Zusatz beigefügt, beispielsweise durch Typdefinitionen. Hier wurde der objektorientierte Ansatz so erweitert, dass auch eine mehr oder weniger prozedurale Programmierung möglich ist. So können z.B. neben den Methoden in den einzelnen Klassen auch universelle Methoden erstellt werden, die keiner Klasse zugeordnet werden.
Ein wesentlicher Unterschied zwischen PROGRAPH und anderen objektorientierten Sprachen besteht darin, dass Nachrichten nicht an Objekte, sondern an die jeweilige Klasse geschickt werden. Dies führt dazu, dass man das aktuelle Objekt als Parameter übergeben oder auf andere Weise herausfinden muss, während es z.B. in C++ als impliziter, erster Parameter immer vorhanden ist (this).
Die Objekte spielen in PROGRAPH eine bedeutende Rolle. Sie sind aber nicht die aktiven Elemente, wie man sie sich in der rein objektorientierten Programmierung vorstellt. So werden in PROGRAPH einige wichtige Prinzipien der Objektorientierung verletzt, z.B. Information-hiding über private Datenelemente, und eine reine Programmierung ist in diesem Stil kaum bzw. nur mit Einschränkungen möglich.

2 Die Entwicklungsumgebung

Prograph beinhaltet ein Interpreter, grafischen Debugger, Tutorials und Hilfen. Die Struktur eines Prograph-Programms wird durch eine Fenster-Hierarchie gebildet. Ein Programm ist ein Projekt. Ein Projekt enthält mehrere Sections. Diese sind vergleichbar mit Packages in Java. Jede Section besteht aus Classes, Universals und Persistents. Elemente des Programms sind durch Symbole dargestellt.
Eine Beschreibung und Tutorials, wie man Prograph benutzt, bekommt man mit der Software ausgeliefert. Aber sie sind nicht besonders hilfreich. Es reicht gerade noch um einfache Programme zu implementieren. Vielleicht hängt es damit zusammen, dass seit 1998 an Prograph nicht mehr entwickelt wurde [1].

[1] Zumindest war es nicht möglich, eine aktuellere Version zu bekommen

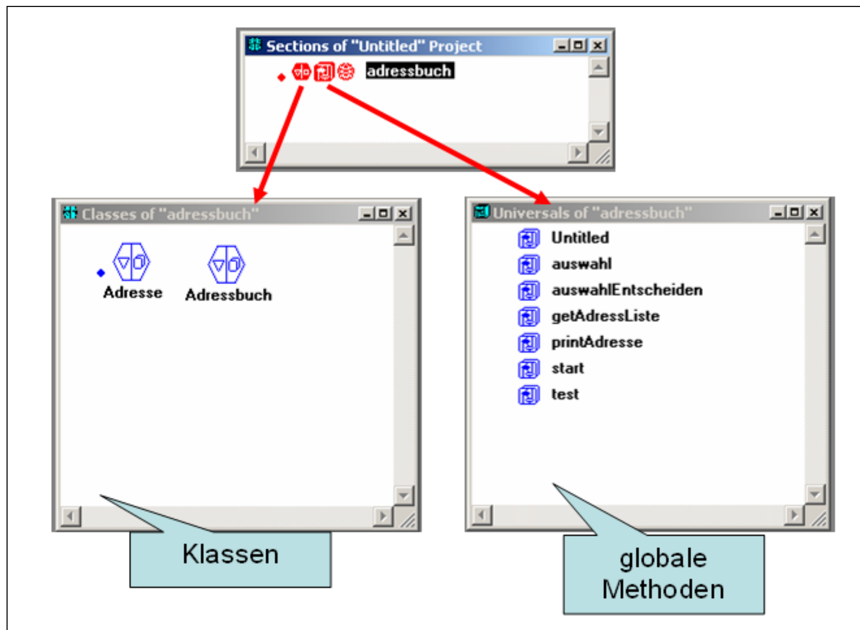

Abbildung 2: Fensterstruktur der Prograph-IDE

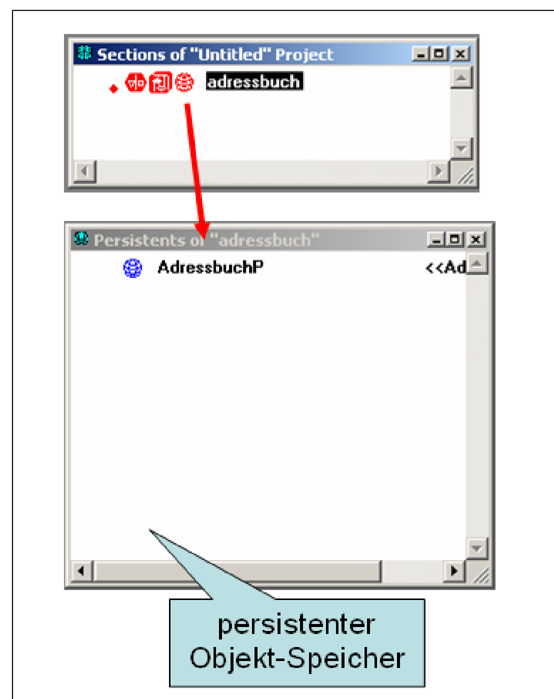

Abbildung 3: Fensterstruktur der Prograph-IDE

3 Sprachelemente

Im hier folgenden Kapitel werden alle wichtigen Sprachelemente von Prograph beschrieben. Als Vergleich dieser Elemente wird beispielhaft versucht, die Prograph-Konstrukte in eine textuelle Sprache abzubilden, um die Semantik der Sprache besser verständlich zu machen. Begonnen wird mit der Erklärung der Klassen und den dazugehörigen Eigenschaften. Anschließend werden Methoden erläutert, wobei zunächst nicht zwischen universellen oder Klassenmethoden unterschieden wird, da der grundsätzliche Aufbau bei beiden gleich ist. Nur der Aufruf der Methoden unterscheidet sich geringfügig.

3.1 Klassen

Klassen dienen, wie in allen anderen objektorientierten Programmiersprachen auch, zur Modellierung von Datenstrukturen und deren Verhalten. Elementare Konzepte wie Vererbung, Polymorphie und Verschattung spielen dabei eine zentrale Rolle.

3.1.1 Klassendefinition und Vererbung

Wie in Kapitel 2 bereits zu sehen war, befinden sich die Klassen eines Prograph-Programms im Klassenfenster der jeweiligen Section. In diesem Fenster werden alle enthaltenen Klassen durch sechseckige Figuren dargestellt, die vertikal durch eine Linie in zwei Teile geteilt sind. Im linken Teil der Klasse befindet sich ein nach unten zeigendes Dreieck, welches für die Attribute steht, und im rechten Teil ein verkleinertes Datenfluss-Symbol, hinter welchem sich die Klassen-Methoden verbergen. Unter dem Sechseck befindet sich der Name der Klasse, welcher in dem Projekt eindeutig sein muss. Doppelklickt man in eine freie Fläche des Fensters, erstellt man eine neue leere Klasse, doppel-klickt man in eines dieser Teile einer existierenden Klasse, so öffnet sich das dazu gehörige Fenster für Attribute oder Methoden. Jede neu erstelle Klasse besitzt automatisch einen Default-Konstruktor, der ein Objekt mit Default-Attributwerten erstellt. Ein davon abweichender Konstruktor kann ebenfalls implementiert werden, dies wird jedoch im Methoden-Fenster einer Klasse getan.
Vererbung wird durch eine blaue Linie dargestellt, die von der unterseite der Superklasse zur oberseite der Subklasse verläuft. Die Richtung der Vererbung ist also nur von den Ankerpunkten der Vererbungslinien abzulesen, was bei komplexen Klassenstrukturen zu Verwirrung führen kann. Mehrfachvererbung ist, wie z.B. bei Java auch, mit Prograph nicht möglich. Ein Beispiel für Klassen und Vererbung ist in Abb. 4 zu sehen. Es existieren drei Klassen,

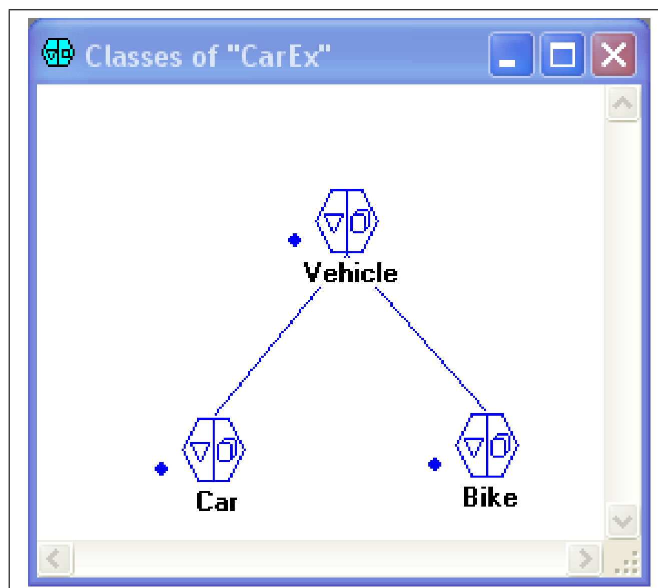

Abbildung 4: Klassen und Vererbung

Vehicle, *Car* und *Bike*, wobei die beiden letzteren von der ersten erben.

3.1.2 Attribute

Attribute einer Klasse sind in einem separaten Fenster angeordnet. Prograph Klassen können statische (Klassen-Attribute) und Instanz-Attribute besitzen. Statische Attribute haben genau den gleichen Wert für alle Objekte einer Klasse, sie entsprechen also dem *static*-Prinzip von Klassen-Attributen in Java, während die Werte vor Instanz-Attribute für jedes Objekt unterschiedlich sein können. Dargestellt werden sie durch untereinander liegende Auflistung im Attributfenster, wobei für statische und Instanz-Attribute jeweils ein eigener Bereich existiert, welcher durch eine grüne horizontale Linie von dem anderen abgegrenzt wird.
Im oberen Bereich sind die statischen Attribute organisiert. Sie werden durch ein kleines Sechseck (analog zu den Klassen selbst) gefolgt von einem Namen und einem Defaultwert dargestellt. Den Typ des Attributs gibt man an, indem man mit einem Doppel-klick auf das entsprechende Sechseck ein weiteres Fenster öffnet. Dort kann man aus einer Liste entweder einen Basistyp oder eine Klasse auswählen. Im Unteren befinden sich die Instanz-Attribute, welche durch ein nach unten zeigendes Dreieck gefolgt von einem Namen und einem Defaultwert dargstellt werden. Den Typ hierfür gibt man genau wie

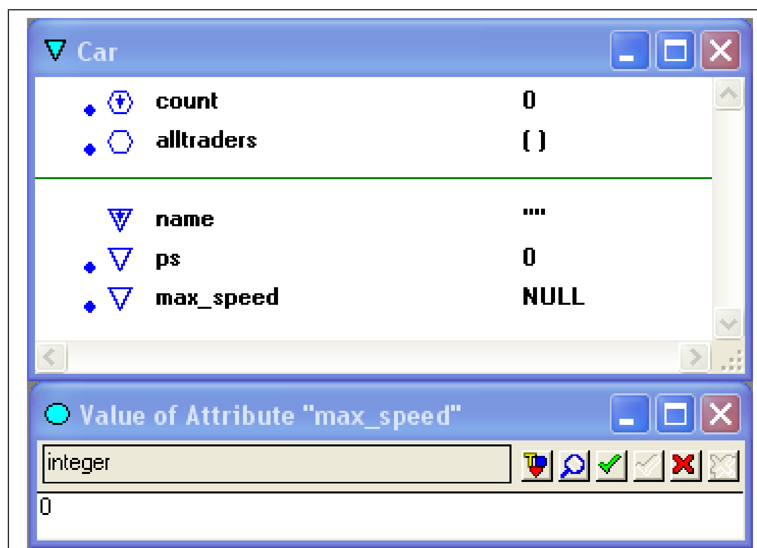

Abbildung 5: Attribute der Klasse Car

bei den statischen Attributen an.
Ist ein weiteres Dreieck in dem ursprünglichen Symbol (Dreieck oder Sechseck) enthalten, so ist das Attribut von einer Superklasse geerbt. Dann kann der Defaultwert überschrieben (verschattet) werden. Der Zugriff auf Attribute erfolgt nur über *getter*- und *setter*-Methoden, welche sofort nach Erstellung eines Attributs verfügbar sind. Diese können auch überschrieben werden, was im Methodenfenster gemacht werden kann. In Abb. 5 sind die statischen und Instanz-Attribute der Klasse *Car* zu sehen. Sowohl *count* als auch *name* sind von der Klasse *Vehicle* geerbt, und enthalten einen nach unten gerichteten Pfeil im entsprechenden Symbol. Die anderen Attribute sind in der Klasse selbst definiert. Im unteren Teil der Abbildung sieht man ein Value-Fenster für das Attribut *max_speed*, indem der Typ und der Defaultwert festgelegt werden kann.

3.1.3 Methoden

Klassen-Methoden sind ähnlich organisiert wie Attribute, nur dass es keine Unterscheidung zwischen statischen und Instanz-Methoden gibt. Vielmehr sind alle hier definierten Methoden statisch. Der Unterschied zu den Universals besteht lediglich darin, dass sie logisch zu einer bestimmten Klasse gehören. Neben den *normalen* Methoden können jedoch speziellere implementiert werden. Dies sind zum einen die Konstruktoren der Klasse, zum anderen die *getter*- und *setter*-Methoden, die dazu dienen, die Default- Im-

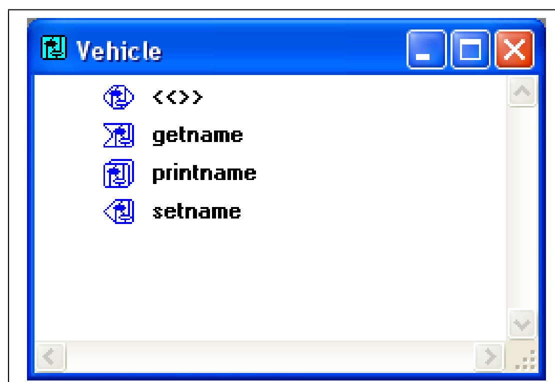

Abbildung 6: Methoden und Konstruktor der Klasse Car

plementierungen zu überschreiben.
In dem Methoden-Fenster sind die einzelnen Methoden wieder untereinander aufgelistet, indem das jeweilige Methodensymbol (unterschiedlich für die jeweiligen Methodenvarianten) gefolgt von einem Methodennamen abgebildet ist. Die Arität (Parameter und Rückgabewert) der Methode gibt man hier noch nicht an. Sie wird bei der Definition des Methodenrumpfes erstellt, wie später gezeigt wird. In Abb. 6 ist ein Konstruktor (der Name muss <<>> lauten), ein Getter, ein Setter und eine normale Methode der Klasse *Vehicle* zu sehen. Jeder Typ hat ein anderes Symbol, welches ähnlich zu den Symbolen ist, dass beim Aufruf der Methoden in den Datenflussdiagrammen verwendet wird. Der Konstruktor ist wieder ein Sechseck, diesmal mit Datenfluss-Symbolen innerhalb, der Getter ist ein Balken mit ausgeschnittenem Dreieck auf der linken Seite, der Setter mit ausgeprägtem Dreieck, und die normale Methode ist ein einfacher Balken, jeweils ebenfalls mit Datenfluss-Symbolen innerhalb.

3.2 Universals

Universals sind Methoden, die keinen logischen Bezug zu irgendeiner Klasse haben, und deswegen nicht als Klassenmethode implementiert werden sollen. Die Definition von Universals erfolgt analog zu der Definition der anderen Methoden auch, sodass hier keine weiteren Erläuterungen folgen. Das Fenster mit allen Universals einer Section öffnet sich nach Doppel-klick auf das mittlere Symbol einer Section. In Abb. 7 ist ein Universals-Fenster mit einer Methode zu sehen, die zu keiner Klasse logisch gehört [2].

[2] Obwohl es in Java eine Klasse zur Erstellung von Zufallszahlen gibt

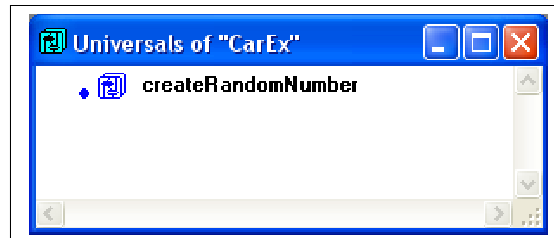

Abbildung 7: Ein Universals-Fenster

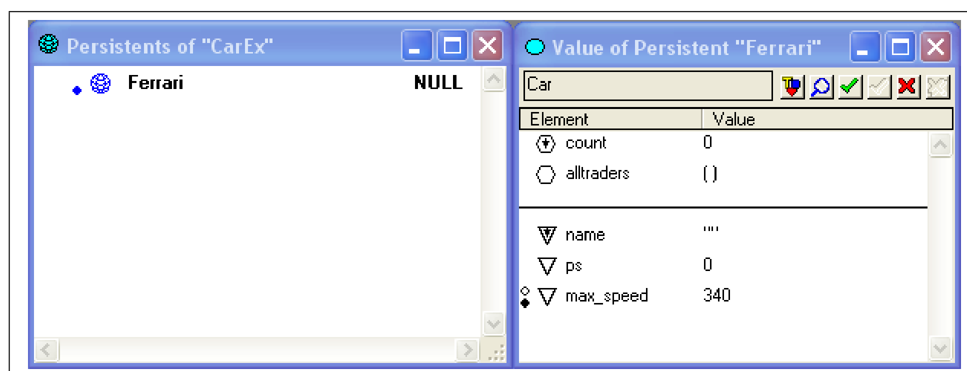

Abbildung 8: Ein Persistents-Fenster

3.3 Persistents

Persistents sind, wie der Name schon andeutet, persistente Objekte, also eine Art objektorientierte Datenbank, in der man Objekte dauerhaft abspeichern kann. Jedes darin definierte Objekt behält seinen Zustand bei, wenn das Programm beendet und neu gestartet wird[3]. Um persistente Objekte zu Erzeugen, öffnet man das Persistenz-Fenster durch Doppel-klick auf das rechte Symbol (Kreis) der Section. Dort kann man Objekte definieren und deren Attribute belegen, indem man auf die entsprechenden Symbole klickt, woraufhin sich wieder ein Value-Fenster öffnet. In Abb. 8 ist ein Persistents-Fenster zu sehen, welches ein Objekt vom Typ *Car* mit dem Namen *Ferrari* enthält. Dieser Name ist für die Referenzierung wichtig. Außerdem sieht man im linken Teil das Value-Fenster, mit dem man das Objekt manipulieren kann.

[3]Dies funktioniert allerdings nur, wenn man den Interpreter verwendet. Beim Benutzen von compilierten Projekten muss man eine eigene Persistenz- Schnittstelle implementieren

3.4 Datenfluss - Implementierung der Methoden

Während bisher die statischen Elemente eines Prograph-Programms beschrieben worden sind, folgen nun die dynamischen Aspekte, also die Implementierung von Methoden. Wie bereits erwähnt, wird dies in Prograph durch Verwendung von Datenfluss-Diagrammen realisiert, die um spezielle programmiersprachliche Konstrukte erweitert worden sind.

3.4.1 Aufbau von Methoden

Methoden in Prograph haben alle den gleichen Aufbau, sodass hier nicht zwischen Universal, Klassenmethode, Getter, Setter oder Konstruktor unterschieden wird. Um eine Methode zu implementieren, öffnet man mit einem Doppel-klick auf das entsprechende Symbol ein Methoden- Fenster, welches initial nur zwei blau und diagonal gestreifte Balken enthält (siehe Abb. 9). Dies sind der Eingabebalken und der Ausgabebalken, über welche die Parameter und die Rückgabewerte in bzw. aus die Methode *fließen*. Zwischen den beiden Balken steht dann der visuelle Code, der die Parameter verarbeitet (falls vorhanden), und die Ausgabewerte generiert. Der Datenfluss geht also immer von oben (vom Eingabebalken) nach unten (zum Ausgabebalken).

3.4.2 Die Arität einer Methode

In jeder Programmiersprache hat jede Methode sowohl eine Liste von Parametern, als auch einen oder mehrere Rückgabewerte. Dieser Tupel wird als Arität bezeichnet. In Prograph bestimmt man diese, indem man die Ein- und Ausgabebalken einer Methode manipuliert. Wie in Abb. 9 zu sehen ist, befinden sich unterhalb des Eingabebalkens zwei Punkte, sog. Datenknoten. Diese markieren die Eingabeparameter, die dort referenziert werden können. Am Ausgabebalken befindet sich ebenfalls ein Datenknoten, der einen Rückgabewert darstellt. Die Typen der Parameter sind leider nicht fest wählbar, vielmehr muss man wissen, mit welchen Argumenten man eine Methode aufruft, und welche Rückgabewerte sie generiert.

3.4.3 Aufruf von Methoden und vordefinierten Operationen

Um innerhalb einer neu definierten Methode den visuellen Code zu implementieren, muss man meist andere Methoden aufrufen, oder aber eine Operation aus dem vordefinierten Fundus von Prograph verwenden. Dazu zählen Operation zum Rechnen, zum Arbeiten mit Listen, Standard-Überprüfungen, Dialoge, und viele andere, damit diese nicht immer neu implementieren werden müssen. Um eine vordefinierte Operation aufzurufen, erzeugt man

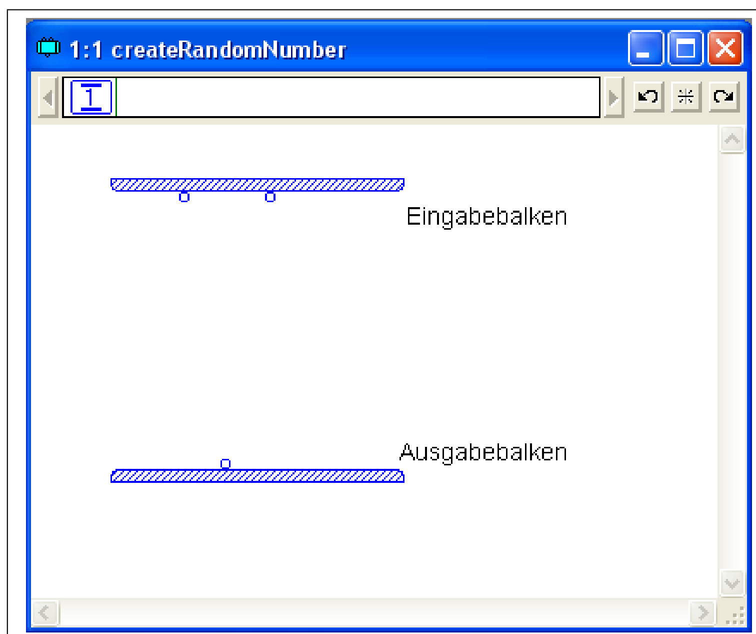

Abbildung 9: Eine leere Methode

durch Doppel-klick in das Methoden-Fenster ein Operations-Symbol (ein blauer Balken), und gibt den entsprechenden Namen ein, der aus der Dokumentation gewonnen werden kann. Ist dieser korrekt, so erzeugt die IDE automatisch die benötigte Anzahl der Ein- und Ausgabedatenknoten dieser Operation. Der Aufruf von selbst erstellen Universals funktioniert genau in der gleichen Weise. Für einen Klassenmethoden-Aufruf gibt es mehrere Möglichkeiten. Die einfachste besteht darin, den Klassennamen getrennt von einem / und den Methodennamen in das Symbol zu schreiben. Dabei wird genau die in der angegebenen Klasse definierte Methode aufgerufen (obwohl das übergebene Objekt vom Typ einer Subklasse sein kann). Will man genau die Methode aufrufen, die zu dem übergebenen Objekt passt, kann man den Methodennamen einfach durch voranstellen von // in das Symbol schreiben. Dann versucht Prograph die Methode zu finden, die in der Klasse des übergebenen Objekts definiert ist. Falls in der Klasse des übergebenen Objekts keine derartige Methode definiert ist, kann mit voranstellen von / die Methode der in der Vererbungshierarchie nahesten Superklasse gesucht und aufgerufen werden. Dies wird als Polymorphie bezeichnet. Beispiele zu den Methodenaufrufen sind in Abb. 10 zu sehen.

3.4 Datenfluss - Implementierung der Methoden

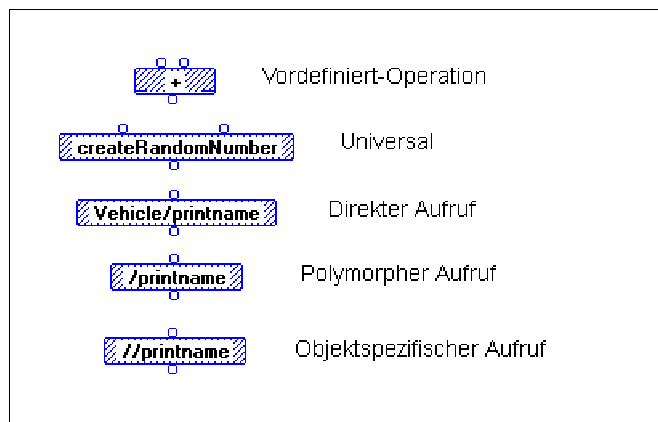

Abbildung 10: Unterschiedliche Aufrufe von Methoden

3.4.4 Erstellung und Manipulierung von Objekten

Der Methodenrumpf wird dann innerhalb der Ein- und Ausgabebalken definiert. Das Grundprinzip besteht darin, Elemente über Datenverbindungen so miteinander zu verknüpfen, dass ein Datenfluss vom Eingabebalken bis zum Ausgabebalken besteht. Zunächst werden Elemente zum Erstellen eines Objekts, und dessen Getter und Setter-Methoden dargestellt. Diese sind in Abb. 11 zu sehen. Eine Objekt-Instanz wird durch einen Balken dargestellt, der eine Spitze zu beiden Seiten besitzt, und den Klassennamen enthält (ohne eckige Klammern). Getter und Setter sind durch die gleichen Symbole dargestellt, wie sie bei der Deklaration der Methoden im Methoden-Fenster der Klasse zu sehen waren, also Balken mit entweder ausgeschnittener Spitze (Getter) oder angefügter Spitze (Setter), und dem Namen des Attributs. An diesen Elementen sind ebenfalls Datenknoten an der Ober- und Unterseite vorhanden. Die Knoten der Oberseite sind wieder Eingabewerte, die (meist) benötigt werden, die der Unterseite Ausgabewerte, die nach dem Ausführen zurückgegeben werden.

Um die Funktionsweise zu beschreiben wird die Methode aus Abb. 12 betrachtet. Wie schon gesagt, verläuft der Datenfluss von oben nach unten, und zwar entlang der Datenverbindungen. Jedes enthaltene Element einer Methode kann dann ausgeführt werden, wenn alle eingehenden Daten vorhanden sind. In der Abbildung kann also als erstes sowohl das Objekt erstellt werden (Eingabewert ist hier nicht erforderlich), als auch die Aktion *ask* ausgeführt werden, da ihr eine konstante übergeben wird (Konstanten sind durch einen unterstrichenen Wert dargestellt, und besitzen immer nur einen Ausgabeknoten). Die anderen Operationen benötigen hingegen Einga-

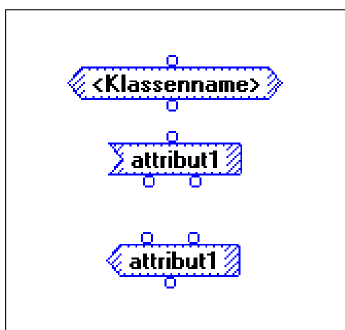

Abbildung 11: Symbole für Instanziierung, Getter und Setter

bewerte, die erst noch erzeugt werden müssen, deswegen können diese nicht ausgeführt werden. Ist z.B. das Objekt erstellt, kann anschließend der Getter das Alter der Person abfragen, und weiterleiten. Dies geschieht in dem rechten Ausgabeknoten, während im linken das Objekt weitergeleitet wird. Die Plus-Operation kann dann noch nicht schalten, erst wenn *ask* einen Wert geliefert hat (ask ist eine Build-In Operation, die einen Eingabedialog öffnet, und den Eingabewert weiterleitet). Ist die Plus-Operation durchgeführt, fließt das Ergebnis in den Setter für das Alter, welcher das Objekt und den neuen Wert bekommt, und als Ausgabe das modifizierte Objekt liefert. Dieses fließt abschließend in den Ausgabebalken und ist somit der Rückgabewert der Methode. Die tatsächliche Abfolge der Operationen ist im Prinzip unbestimmt. Die einzige Synchchronisation erfolgt in diesem Beispiel durch die Abhängigkeiten, die durch die Ein- und Ausgabewerte der Operationen entstehen. Es könnte also zuerst das Objekt erstellt werden und dann die *ask*-Operation aufgerufen werden, oder aber umgekehrt. In diesem Beispiel ist dies jedoch egal.

Zur Verdeutlichung soll an dieser Stelle Java-Code angegeben werden, der das Verhalten der Methode nachbildet. Dies ist in Abb. 13 zu sehen.

3.4.5 Schleifen

Schleifen sind eines der Dinge, die mit Datenfluss selbst nicht modelliert werden, weswegen in Prograph dazu eine neue Syntax definiert wurde. Das Prinzip ist folgendes: Man kann eine Methode in Schleife laufen lassen, indem man die Ausgabewerte gleich wieder in die entsprechenden Eingabewerte reinschreibt. Symbolisiert wird dies durch rückwärts gerichtete Pfeile anstelle von normalen Datenknoten für die Parameter und Ausgabewerte einer Methode (siehe Abb. 14).

3.4 Datenfluss - Implementierung der Methoden

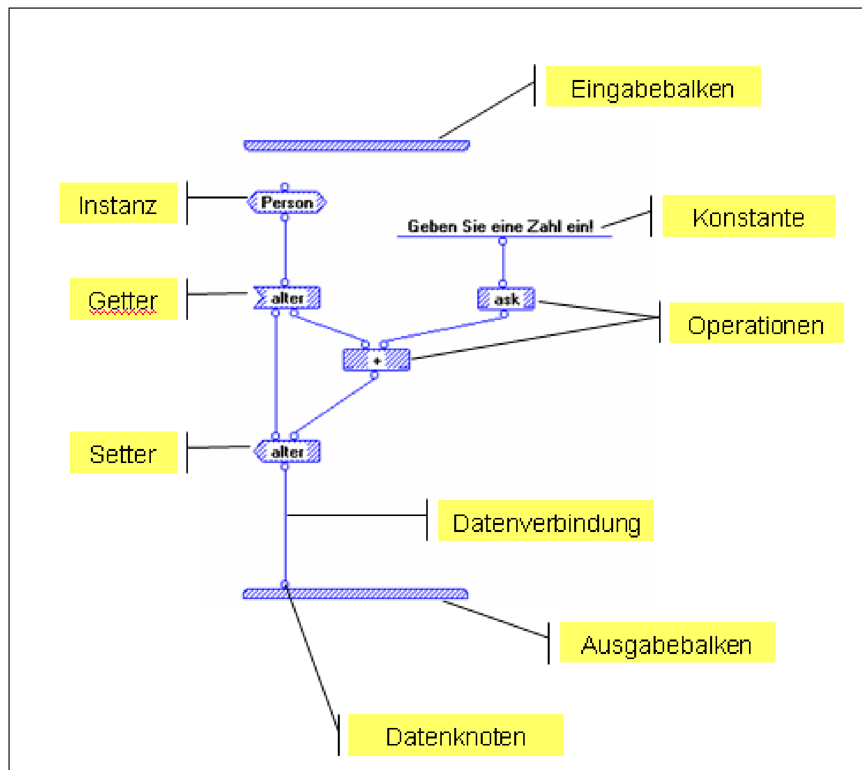

Abbildung 12: Erstellung eines Objekts, Getter und Setter

```
Person p = new Person();
int tmp = p.getAlter();
int eingabe = ask(..);
p.setAlter(tmp+eingabe);
return p;
```

Abbildung 13: Java-Code zum ersten Beispiel

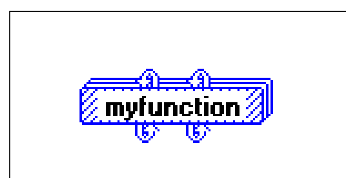

Abbildung 14: Schleifenaufruf einer Methode

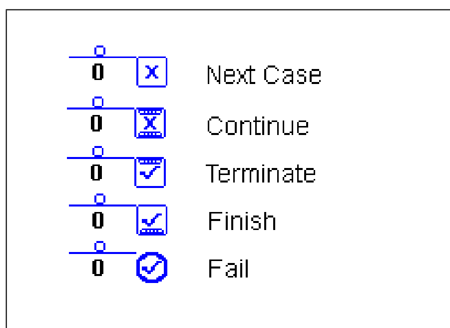

Abbildung 15: Match-Vergleiche mit Meta-Aktionen

Die Methode wird dann zunächst in einer Endlosschleife aufgerufen. Um die Iteration zu stoppen, muss in der Methode ein Abbruch definiert werden. Dazu gibt es ebenfalls ein weiteres Element, einen sog. Match-Konstrukt, mit dem Werte verglichen werden können, und je nach Ausgang des Vergleichs kann eine Meta-Aktion ausgeführt werden. Meta-Aktionen sind *Terminate*, *Continue*, *Finish*, *Fail* und *Next Case*. *Terminate* ist vergleichbar mit einer Java *break*-Anweisung, *Continue* entspricht ebenfalls dem *continue* aus Java. *Finish* ist wie *Terminate*, nur dass der aktuelle Vorgang noch einmal ausgeführt wird. *Fail* wird verwendet, um eine Exception zu werfen, und *Next Case*, um eine Fallunterscheidung einzuleiten. In Abb. 15 sind alle diese Match-Konstruke zu sehen. Innerhalb des Match-Elements steht der Wert, mit dem der Eingabewert verglichen wird. Steht ein X in dem Feld rechts, bedeutet dies, die Meta-Aktion wird bei Ungleichheit aufgerufen, steht ein Häkchen in dem Feld, wird die Meta-Aktion bei Gleichheit aufgerufen. Eine weitere Besonderheit der Match-Konstrukte ist, dass sie immer als erstes ausgeführt werden.

In Abb. 16 ist ein Beispiel für eine Schleife zu sehen. Innerhalb der Methode *test* wird die Methode *fakultaet* iteriert aufgerufen, welche die Fakultät eine Zahl berechnet. Die Eingabewerte sind ein Zählerwert, welcher hier mit 5 initialisiert und in jedem durchlauf um eins dekrementiert wird, und ein Ergebniswert, der das aktuelle Produkt der bisher betrachteten Zählerwerte enthält. Die Abbruchbedingung ist in der Methode *fakultaet*, und besagt, dass die Iteration stoppen soll (mit *Terminate*), wenn der Zählerwert gleich 0 ist. In Abb. 17 ist Java-Code abgebildet, der die Funktionsweise nachbildet.

3.4.6 Listen und Listeniteration

Listen in Prograph lassen sich sehr einfach konstruieren. Eine Konstante als Liste wird einfach erzeugt, indem man die Werte (Objekte) einfach in Klam-

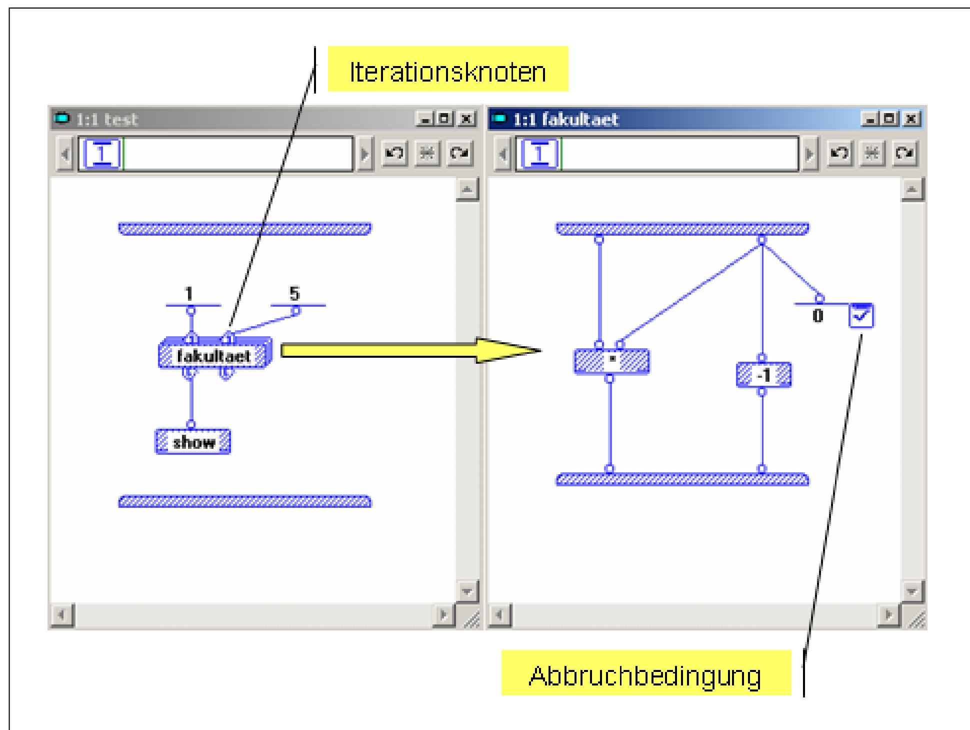

Abbildung 16: Beispiel einer Schleife zur Berechnung der Fakultät

```
int fak = 1;
int zaehler = 5;
while (true) {
  //start "fakultaet"
  if (zaehler == 0)
    break;
  fak = fak*zaehler;
  zaehler = zaehler-1;
  //ende "fakultaet"
}
show(fak);
```

Abbildung 17: Java-Code zum Schleifen Beispiel

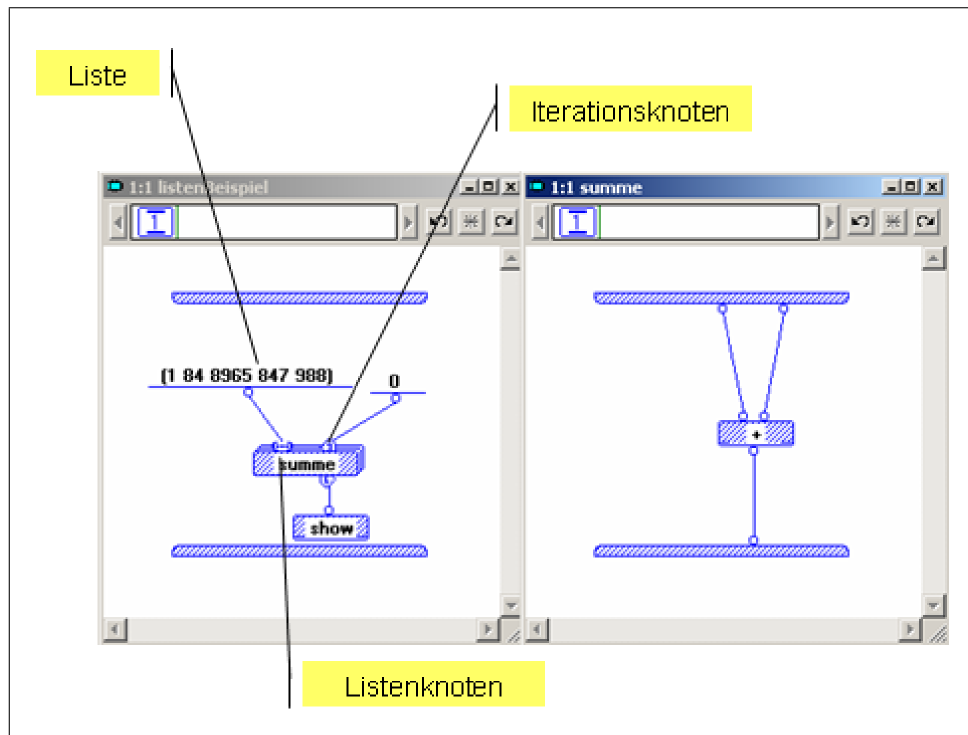

Abbildung 18: Beispiel einer Listeniteration

mern angibt. Zusätzlich unterstützt Prograph die Iteration über Listen, welche wieder an bereits vorhandenen Methoden ausgeführt werden kann. Dazu definiert man dann einen Listeniterationsknoten (analog zum Schleifenknoten), und übergibt diesem eine Liste. Dadurch wird die Methode mit jedem Listenelement genau einmal aufgerufen, eine Abbruchbedingung ist nicht notwendig (kann aber verwendet werden).

In Abb. 18 sieht man eine Listeniteration über die Methode *summe*, welche eine Eingabeliste von Integerwerten bekommt. Der Iterationsknoten ist durch zwei Klammern mit Punkten innerhalb gekennzeichnet. Zusätzlich wird während der Iteration der zweite Ausgabewert wieder als Eingabewert übergeben, sodass die Summer der Liste berechnet werden kann. Die Operation *show* gibt das Ergebnis auf einem Dialog aus.

3.4.7 Fallunterscheidung

Das letzte umfangreichere Programmkonstrukt sind Fallunterscheidungen. Da Prograph datenfluss-, und nicht kontrollflussorientiert ist, sind diese Konstrukte etwas umständlicher. Die bisher betrachteten Beispiele hatten keine

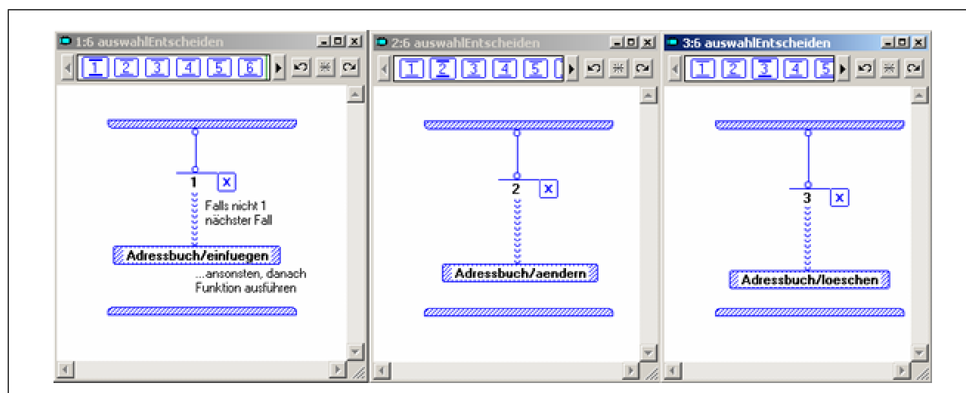

Abbildung 19: Beispiel Fallunterscheidung

Fallunterscheidung, jedenfalls keine, in der dadurch bedingt unterschiedlicher eigener Code ausgeführt wird (nur eine Abbruchbedingung ist bisher gezeigt worden). Das Prinzip ist, dass jede Methode eine unterschiedliche Anzahl an möglichen Fällen besitzt, deren Reihenfolge relevant ist. Jeder Fall wird in einem eigenen Fenster definiert. Mit der Match-Anweisung *Next Case* kann dann bei erfolgreicher Überprüfung in den jeweils nächsten Fall gesprungen werden. Dazu ist ein Beispiel in Abb. 19 zu sehen. Dort ist eine Methode zu sehen, die insgesamt 6 Fälle unterscheidet, wovon drei zu sehen sind. In jedem Fall wird zunächst eine Bedingung überprüft, und falls diese nicht zutrifft, wird in den nächsten Fall gesprungen. In den Fenstern ist im oberen Bereich zu sehen, welcher der Fälle dargestellt ist, indem in einer Liste von blau umrahmten Zahlen der aktive fett hervorgehoben ist. Weitere Besonderheit dieses Beispiels ist, dass ein sog. Synchro-Link vom Match zum Methodenaufruf vorgibt, dass die Match-Anweisung vor der anderen ausgeführt werden muss. Die aufgerufene Methode ist dabei aus der Klasse Adressbuch, was durch voranstellen des Klassennamens getrennt durch einen Schrägstrich definiert wird. In Abb. 20 ist zu sehen, wie ein entsprechender Java-Code aussehen könnte.

4 Implementierungsbeispiel

4.1 Programmaufbau

Das Beispielprogramm `Adressbuch` modelliert ein sehr rudimentäres Adressbuch ohne grafische Benutzeroberfläche (diese ist nur in der kommerziellen Version von Prograph enthalten). In Fenster (1) in Abb. 21 sind die Module

```
String eingabe;
//wird beim Funktionsaufruf übergeben
if (eingabe == "1") {
Adressbuch.einfuegen();
} else if (eingabe == "2") {
Adressbuch.aendern();
} else if (eingabe == "3") {
Adressbuch.loeschen();
} ...
```

Abbildung 20: Java-Code zum Fallunterscheidungs-Beispiel

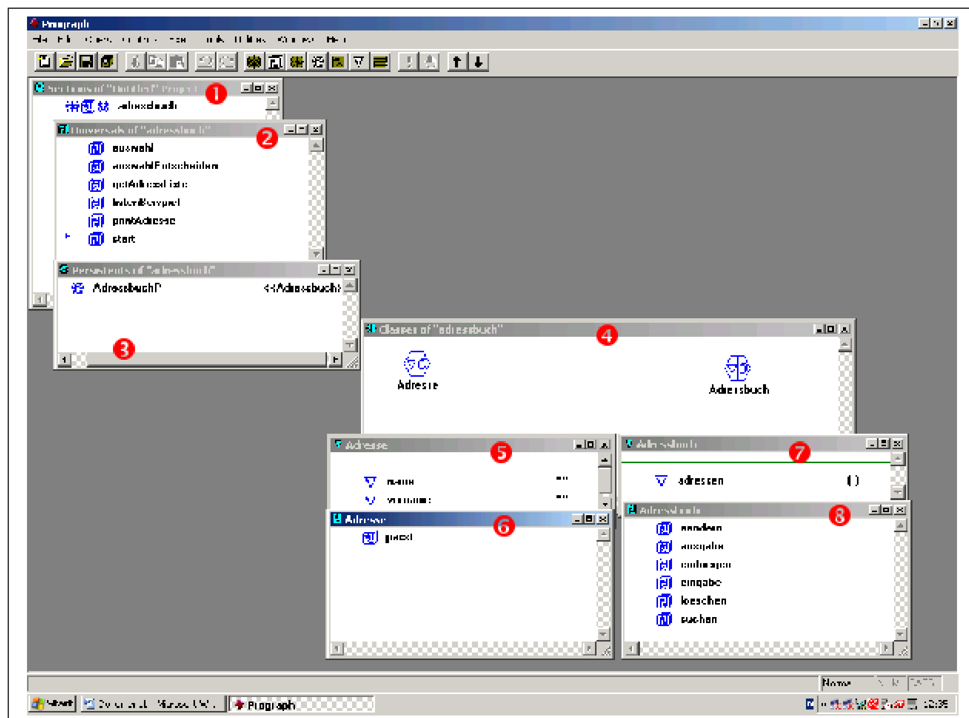

Abbildung 21: Programmaufbau

(`Sections`) des Programms aufgelistet. Das Adressbuch besteht nur aus einem Modul `adressbuch`. Die Module bestehen wiederum aus drei Teilen, den Klassen, globalen Methoden (`Universals`) und einem persitenten Objektspeicher (`Persistents`). Im Fenster sind die globalen Methoden aufgelistet. Das Fenster (3) zeigt die gespeicherten Objekte, hier das Adressbuch mit den Einträgen. In (4) sind die Klassen gezeigt. Die Klasse Adresse kapselt die Einträge, die Klasse Adressbuch enthält die eigentliche Funktionalität. In den Fenstern (5) und (7) sind die Attribute, in den Fenster (6) und (8) die Methoden der Klassen gelistet.

4.2 Ablauf

Die Methoden und deren Funktion wird anhand des Programmablaufs (Kontrollfluss) erklärt. Hier werden allerdings nicht alle Funktionen ausführlich erklärt. Detaillierte Ausführungen beschränken wir hier auf die Methoden, die interessante oder eigenwillige Sprachkonstrukte von Prograph illustrieren.

4.3 `start`

Die Methode start (siehe Abbildung) ruft in einer Schleife die Methode **`auswahl`** auf, bis der Benutzer `Q` zum Beenden eingibt. In dem `Match` wird das Programm terminiert. Die Fahne symbolisiert einen Breakpoint.

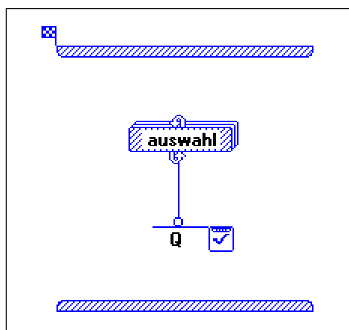

Abbildung 22: Methode `start`

4.4 `auswahl`

Hier wird dem Benutzer zuerst eine Liste an möglichen Funktionen angezeigt, dann wird er nach seiner Auswahl gefragt.
Der Synchro-Link stellt sicher, dass die Frage erst nach Ausgabe der Liste

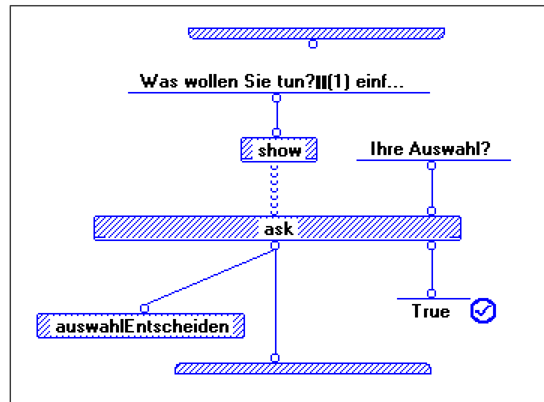

Abbildung 23: Methode `auswahl`

erfolgt. Dann wird mit dem Match auf `True` geprüft, ob der Benutzer abgebrochen hat. Zuletzt wird die Benutzereingabe an die Kontroll-Methode `auswahlEntscheiden` übergeben.

4.5 `auswahlEntscheiden`

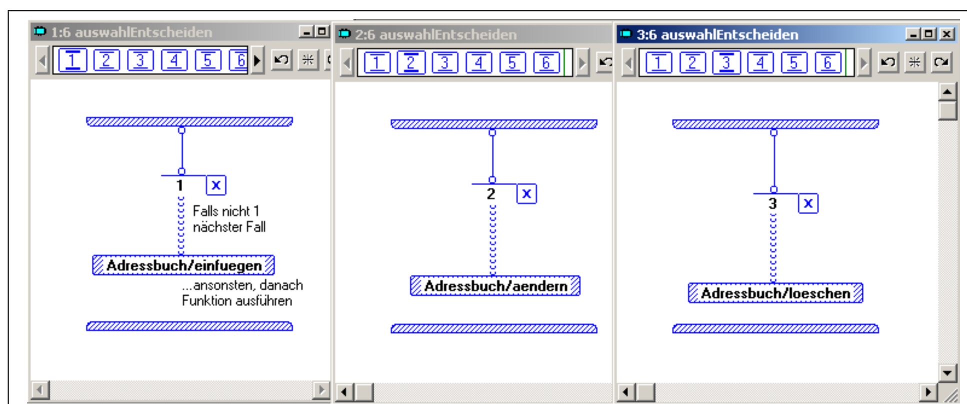

Abbildung 24: Methode `auswahlEntscheiden`

In dieser Methode wird eine mehrfache Fallunterscheidung modelliert. In einer traditionellen Programmiersprache würde man hierfür ein "switch"-Konstrukt verwenden. In Prograph können jedoch nur Fallunterscheidungen mit einem Wechsel zum nächsten Fall, jedoch nicht zu einem beliebigen Fall, modelliert werden. Daher ergäbe sich folgende "Übersetzung" der Prograph-Methode:

```
String eingabe; //wird beim Funktionsaufruf übergeben
if (eingabe == "1") {
    Adressbuch.einfuegen();
} else if (eingabe == "2") {
    Adressbuch.aendern();
} else if (eingabe == "3") {
    Adressbuch.loeschen();
}
...
```

4.6 getAdressListe

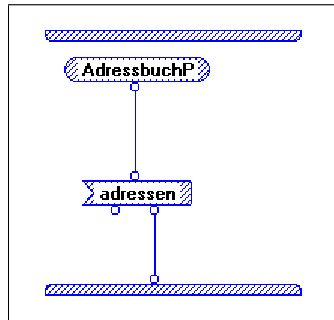

Abbildung 25: Methode `getAdressListe`

Dies ist eine Hilfsroutine, die das Objekt `AdressbuchP` aus dem persistenten Speicher lädt, dann die Liste mit den Adressen über eine Get-Operation herauskopiert und diese dann zurückgibt.

4.7 Adressbuch/einfuegen und Adressbuch/eingabe

Die Methode `Adressbuch/einfuegen` fügt einen Eintrag in das Adressbuch ein. Zuerst wird der Benutzer in der Methode `Adressbuch/eingabe` gebeten, Vor- und Nachname einzugeben. Gibt der Benutzer einen leeren String ein, so wird der Vorgang abgebrochen. Danach wird eine neue Instanz von Adresse generiert und mittels Settern mit den eingegebenen Informationen gefüllt. Dann wird das Adressbuch aus dem persistenten Speicher geladen und das neue Adresse-Objekt wird mittels der Operation `attach-r` hinten an die Liste angehängt. Zuletzt wird die neue Liste in das persistente Objekt kopiert und dieses wieder gespeichert.
Eine textuelle Übersetzung könnte z.B. so aussehen:

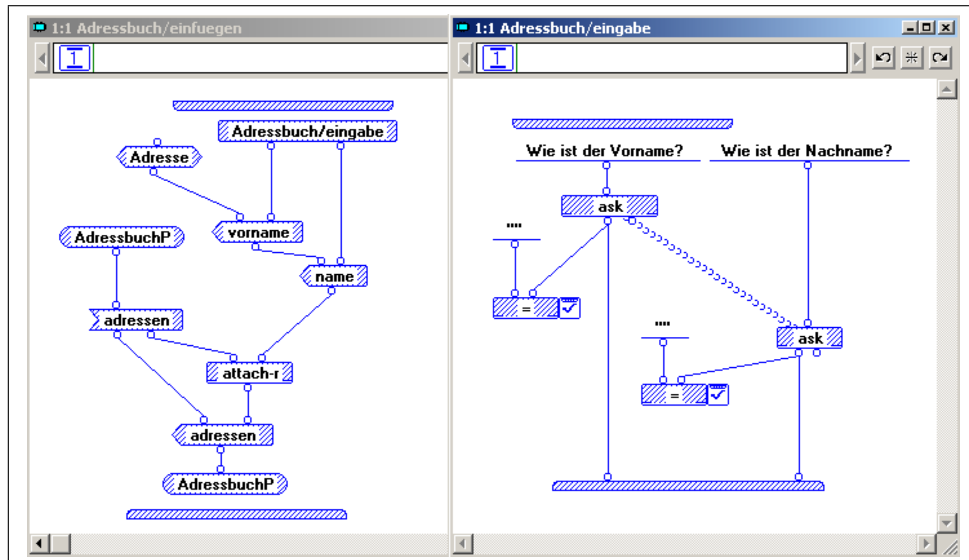

Abbildung 26: Methoden `Adressbuch/einfuegen` und `Adressbuch/eingabe`

```
String vorname,name;
Adressbuch.eingabe(&vorname,&name);
Adresse neue = new Adresse();
neue.setVorname(vorname);
neue.setName(name);
Adresse addr[] = AdressbuchP.getAdressen();
addr.add(neue);
```

4.8 Adressbuch/ausgabe und printAdresse

Die Methode `Adressbuch/ausgabe` erstellt eine Liste mit allen Adressen und zeigt diese dann dem Benutzer an. Die universelle bzw. globale Methode `printAdresse` hängt ein Adresse-Objekt an einen String an. Die Funktion `join` hängt die einzelnen Strings in der Reihenfolge aneinander. Der Funktion `printAdresse` werden der String und ein Zähler als Iterationsknoten übergeben, zusätzlich wird die Adress-Liste als Listenknoten übergeben.

4.9 Adressbuch/aendern

Diese Methode dient dem Ändern von Einträgen. Zuerst werden dem Benutzer die Einträge angezeigt (Reihenfolge wird durch den Synchro-Link sichergestellt). Dann muss der Benutzer den gewünschten Eintrag als Zahl

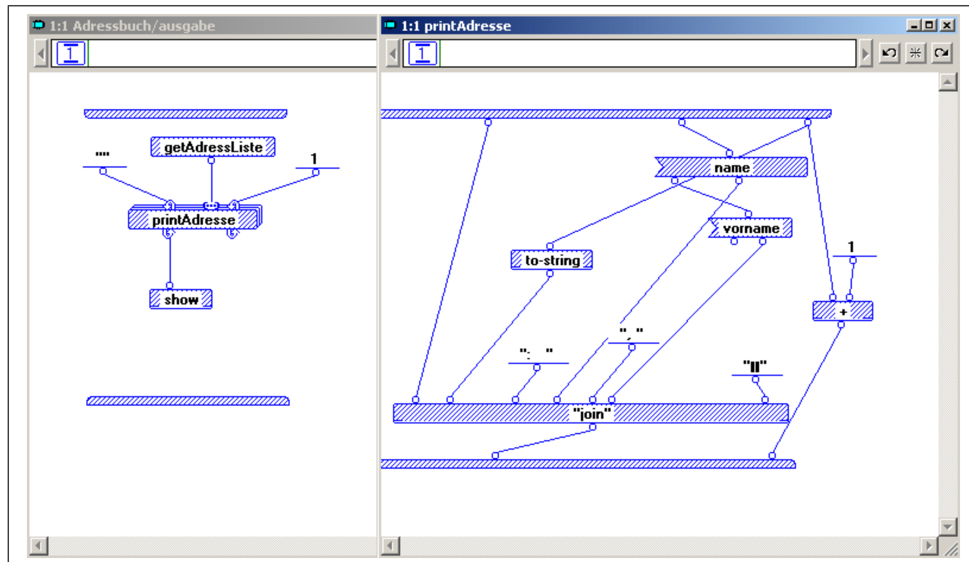

Abbildung 27: Methoden `Adressbuch/ausgabe` und `printAdresse`

eingeben. Unter (1) wird dann geprüft, ob die eingegebene Zahl zu groß ist. Bei (2) wird geprüft, ob die Zahl kleiner als 1 ist. In beiden Fällen wird zum nächsten Fall gesprungen und eine Fehlermeldung ausgegeben.
In (3) wird der Eintrag aus der Adressliste extrahiert. Dann kann der Benutzer die neuen Daten eingeben. Schließlich wird unter (4) der Eintrag in der Liste überschrieben und die Liste wieder im persistententen Speicher gesichert.

4.10 Adressbuch/loeschen

Diese Methode funktioniert analog zur vorherigen, nur dass hier der gewählte Eintrag mittels der `detach-nth` Funktion aus der Liste gelöscht wird. Außerdem wird hier, vor der Bereichsüberprüfung der der eingegebenen Zahl, geprüft, ob der Benutzer wirklich eine Zahl eingegeben hat. Dazu dient die `integer?` Funktion, diese springt zum nächsten Fall, wenn der Benutzer etwas anderes eingegeben hat. Die Synchro-Links stellen sicher, dass zuerst `integer?` ausgeführt wird.

4.11 Adressbuch/suchen und Adresse/passt

Die Methode `Adressbuch/suchen` fragt den Benutzer zuerst nach dem Suchkriterium und iteriert dann über die Adressliste mittels der `Adresse/passt-`

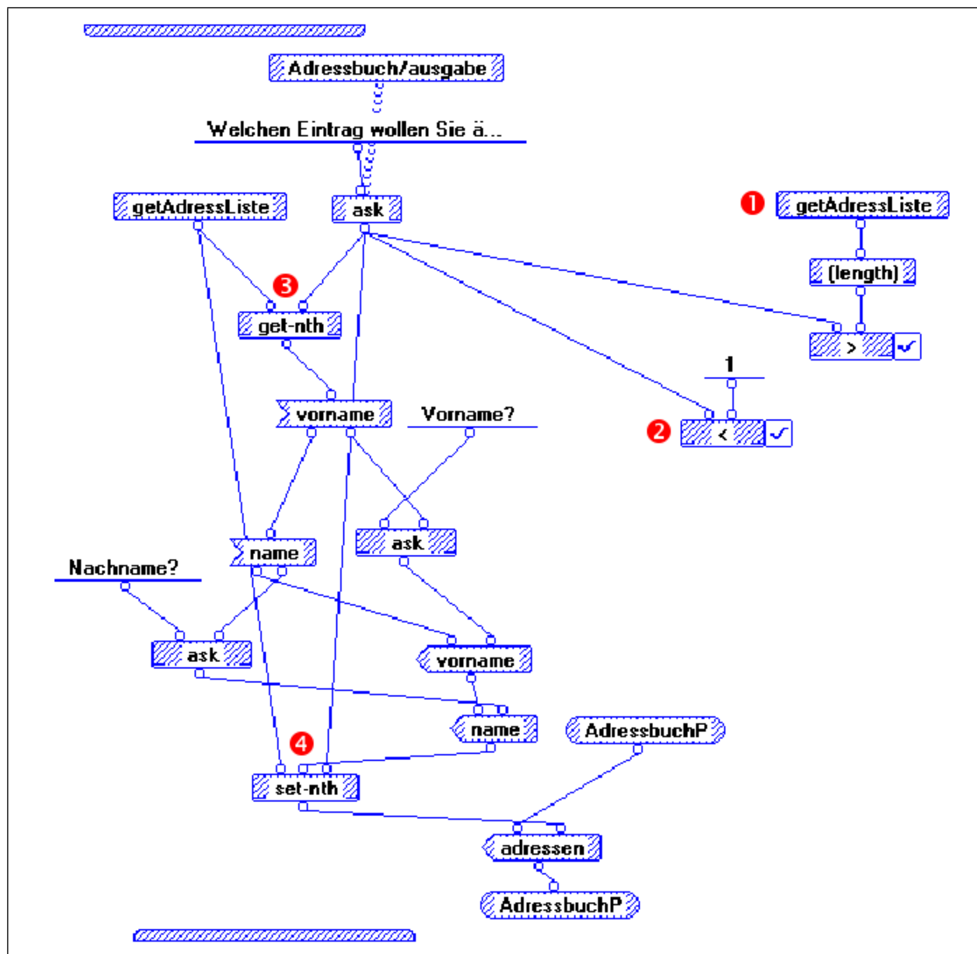

Abbildung 28: Methode `Adressbuch/aendern`

Methode. Dann gibt sie die Liste mit den passenden Einträgen aus. Die `Adresse/passt`-Methode prüft jeweils, ob der Suchbegriff im Namen oder Vornamen enthalten ist. Trifft das zu, so wird der aktuelle Eintrag (zweiter Eingabeparameter) an die Liste (erster Parameter) angehängt. Der dritte Parameter (Suchbegriff) wird immer auch ausgegeben, damit dieser auch beim nächsten Aufruf zur Verfügung steht.

5 Fazit und Bewertung

Abschließend soll der Fokus auf die Vor- und Nachteile von Prograph gesetzt werden. Von Vorteil sind sicherlich die visuellen Aspekte, die die Programme

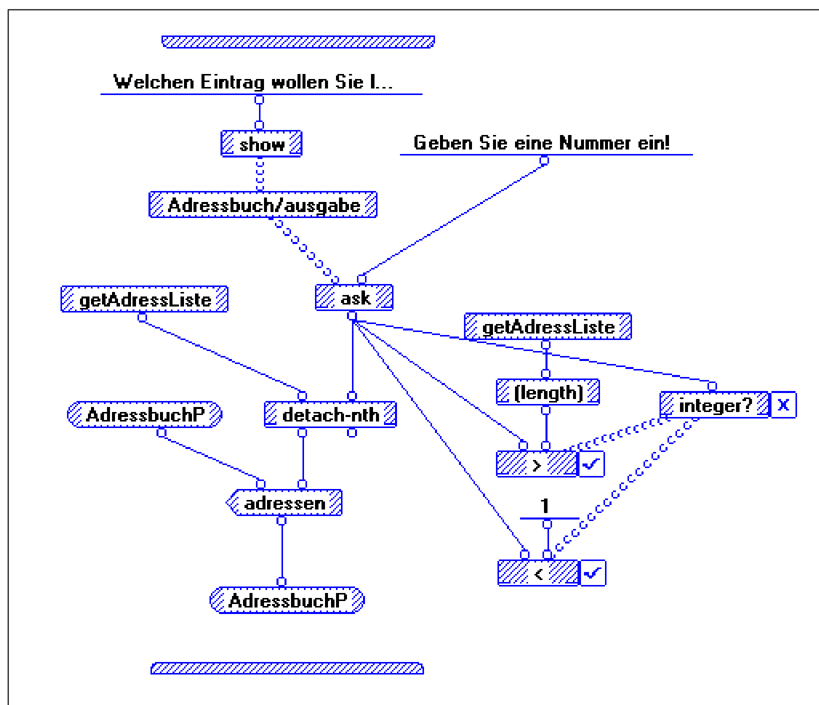

Abbildung 29: Methode `Adressbuch/loeschen`

nachvollziehbarer und leichter verständlich machen sollen, da die Bedeutung der Symbole intuitiv ist. Das visuelle Editieren verhindert auch, dass man gegenüber textuellen Sprachen weniger Syntaxfehler macht, und wie in [4] zu sehen ist, sind mehrere umfangreiche Programme mit Prograph erstellt worden, was den Schluß zulässt, dass alles, was mit textuellen Sprachen möglich ist, auch mit Prograph implementiert werden kann.

Nachteilig wirkte sich jedoch die nicht vorhandene Typisierung von Methoden aus. Viele Fehler werden diesbezüglich erst zur Laufzeit, und nicht zur Compilezeit gefunden. Der Kontrollfluss ist ebenfalls schlecht umgesetzt. Jeder Fall einer bedingten Anweisung muss in einem separatem Fenster implementiert werden, worduch der Code im Zusammenhang nicht gut visualisiert wird. Des weiteren wäre es wünschenswert, Modellierungen mit gängigen Techniken, wie z.B. UML vornehmen zu können, da diese inbesondere in der Informatik einen hohen Bekanntheitsgrad besitzen. Um Transparenz zu gewährleisten, wäre es zusätzlich eine gute Alternative, wenn Quellcode einer bekannten Sprache generiert werden könnte. Damit wäre der Programmierer in der Lage, alle Feinheiten, die textuelle Sprachen zulassen, nachträglich einzufügen.

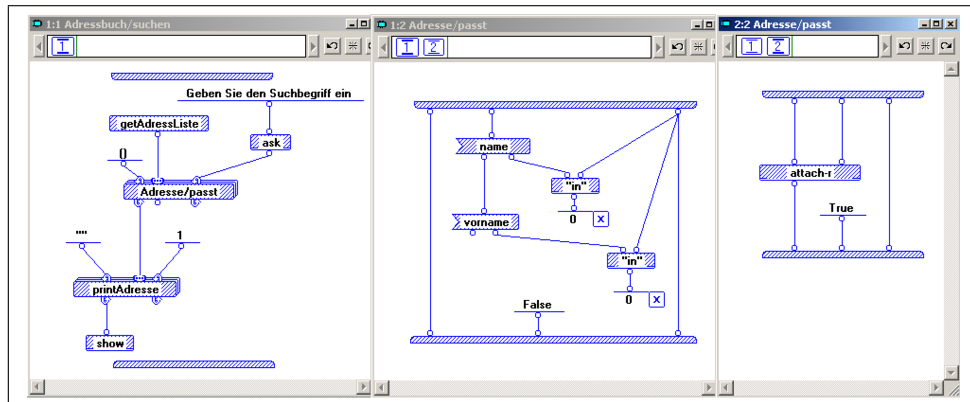

Abbildung 30: Methoden `Adressbuch/suchen` und `Adresse/passt`

Literatur

[1] Prograph Tutorial, Pictorius Inc., 2000, Halifax, Canada

[2] Prograph User Reference, Pictorius Inc., 2000, Halifax, Canada

[3] Website der Fa. Pictorius http://192.219.29.95 (zur Zeit offline)

[4] Website von Triteria, http://www.tritera.com/prograph.html