Diplomarbeit

Ingo Linkweiler

Eignet sich die Skriptsprache Python für schnelle Entwicklungen im Softwareentwicklungsprozess ?

Eine Untersuchung der Programmiersprache Python
im softwaretechnischen und fachdidaktischen Kontext

14. November 2002

Fachbereich Informatik
Fachgebiet Didaktik der Informatik
der Universität Dortmund

Gutachter:
Prof. Dr. Sigrid Schubert
StD Dipl.-Inform. Ludger Humbert

Zusammenfassung

Die Frage nach geeigneten Programmiersprachen für den Informatikunterricht wird seit Jahrzehnten diskutiert. Dieser Bericht fasst die Ergebnisse einer Untersuchung von Programmiersprachen – und insbesondere der Sprache *Python* – unter dem Aspekt der schnellen Softwareentwicklung zusammen. Es zeigen sich Parallelen bei den Anforderungen an Programmiersprachen zur schnellen Softwareentwicklung zu den Anforderungen aus fachdidaktischer Sicht.

Danksagungen

Ich bedanke mich bei meinem Betreuer Ludger Humbert für die gute Zusammenarbeit und zahlreiche Vorschläge zur Diplomarbeit. Stefan Kirchberg und Michael Matschke danke ich für viel Unterstützung bei den Korrekturen, zahlreiche Ratschläge sowie die Einführung in TEX. Hilfe fand ich auch in der Newsgroup comp.lang.python beim Lösen einiger Softwareprobleme und die Einführung in GNU *gettext*.

Inhaltsverzeichnis

1 Einleitung und Motivation

Alle Sprache ist Bezeichnung der Gedanken.

Immanuel Kant (1724 – 1804)

In der vorliegenden Arbeit wird die Programmiersprache Python auf ihre Eignung für schnelle Softwareentwicklung untersucht.

Es werden Kriterien zur Beurteilung der Eignung einer Programmiersprache für den schnellen Softwareentwicklungsprozess entwickelt und exemplarisch auf die Programmiersprache Python angewandt. Dabei soll gezeigt werden, inwiefern die Sprache und dazu verfügbare Werkzeuge den Prozess unterstützen, angefangen bei der Erlernbarkeit der Sprache über die Modellierung des Problems sowie dessen Implementierung bis hin zur Dokumentation.

In diesem Zusammenhang werden Syntax und Semantik, Portabilität zwischen verschiedenen Systemen sowie Lösungsmöglichkeiten für konkrete Aufgabenstellungen anhand von Beispielen untersucht.

Es wird untersucht, welche Gemeinsamkeiten und Unterschiede zwischen den Kriterien für schnelle Softwareentwicklung und fachdidaktischen Anforderungen in der informatischen Bildung bestehen.

Guido van Rossum nennt in einem Interview [Rossum02] die erreichten Ziele bei der Entwicklung von Python:

1. Eignung als Sprache zur Ausbildung
2. Einfache, leicht erlernbare Syntax und Datenstrukturen
3. Große (Standard-) Bibliotheken mit GUI-Anbindung und gute Erweiterungsmöglichkeiten

In dieser Arbeit wird untersucht, inwiefern diese Eigenschaften zutreffen, und wo genau Vor- sowie Nachteile aus Sicht erfahrener Entwickler wie auch im fachdidaktischen Einsatz liegen.

Als Fallstudie werden im Rahmen dieser Arbeit mehrere Programme erstellt, welche die Möglichkeiten von Python in der Praxis bei Projekten unterschiedlicher Größe demonstrieren sollen. *SuM* ist eine Python-Implementierung der Ausbildungssoftware *Von Stiften und Mäusen* [CDH99]. *PyNassi* realisiert einen Editor für Struktogramme und demonstriert unter anderem die Möglichkeiten der Entwicklung grafischer Benutzungsoberflächen.

1.1 Strukturierung dieser Arbeit

Unter der Zielsetzung der schnellen Softwareentwicklung werden in Kapitel 1 zentrale Elemente der Programmiersprache Python vorgestellt.

In Kapitel 2 werden Grundlagen der schnellen Softwareentwicklung aufbereitet und Bezüge zum Einsatz in der informatischen Ausbildung hergestellt.

Kapitel 3 stellt vorbereitend die notwendigen Methoden und Kriterien zur Untersuchung der Programmiersprache Python bereit. Dabei werden typische Anforderungen an Entwicklungssysteme ermittelt, die schnelle Softwareentwicklung ermöglichen.

Kapitel 4 stellt die Überprüfung der einzelnen Kriterien an der Programmiersprache Python dar, welche hauptsächlich die schnelle Softwareentwicklung betreffen. Dazu zählen insbesondere Syntax und Semantik sowie die Ausstattung mit Bibliotheken.

Der Einsatz zu Ausbildungszwecken wird in Kapitel 5 thematisiert. Es werden Umsetzungsmöglichkeiten von Programmierparadigmen und typische Anforderungen der informatischen Bildung an eine Programmiersprache berücksichtigt.

Um die Praxisrelevanz der entwickelten und beschriebenen Kriterien zu prüfen, werden sie in Kapitel 6 auf Ergebnisse der Softwareentwicklung angewandt. Dabei finden sowohl Ergebnisse aus veröffentlichten empirischen Studien, aber auch die Erfahrungen mit Fallstudien im Rahmen dieser Arbeit Berücksichtigung.

Eine Zusammenfassung der Ergebnisse erfolgt abschließend in Kapitel 7.

Im Rahmen dieser Arbeit wurden einige Programme (siehe 3.2) und zahlreiche Beispiele erstellt. Des Weiteren wurden einzelne Themen auf Tagungen und Workshops vorgestellt und diskutiert. Auf der beiliegenden CD wurden Software und sämtliches Material aus dem Umfeld der Arbeit zusammengestellt. Da die Software noch weiterentwickelt wird, wurde eine Internetseite [Linkweiler02] erstellt, die über aktuelle Entwicklungen und neue Softwareversionen informiert.

1.2 Einordnung von Python

Python ist eine objektorientierte Programmiersprache und wird den interpretierten Skriptsprachen zugeordnet. Ihren Namen verdankt Python nicht der gleichnamigen Schlangenart, auch wenn diese inzwischen zu ihrem Symbol geworden ist, sondern der britischen Comedy-Gruppe *Monty Python*.

Begonnen wurde ihre Entwicklung 1989 von Guido van Rossum am CWI (Centrum voor Wiskunde en Informatica) in Amsterdam [CWI]. Skriptsprachen, die in dieser Zeit in der UNIX-Welt weit verbreitet waren, Compilersprachen (C, Modula3, Icon) sowie die weniger bekannte, aus dem Ausbildungsbereich stammende Sprache ABC übten Einfluss auf ihre Entstehung aus. Ursprünglich wurde Python in der Betriebssystemforschung eingesetzt, es wurde aber schnell auf unterschiedliche Plattformen portiert. Ein Entwicklungsbaum der Programmiersprachen befindet sich in Anhang A.9.

Python wird frei entwickelt und wurde lange Zeit durch die nichtkommerzielle PSA (Python Software Activity [PSA02]) unterstützt, inzwischen liegt diese Aufgabe in den Händen der ebenfalls nichtkommerziellen PSF (Python Software Foundation [PSF02]). Die PSF organisiert die Entwicklung und erstellt die kostenlose Kerndistribution von Python inklusive Bibliotheken, Sourcecode und Dokumentation. Python ist Open Source im Sinne der Open Source Initiative [OSI02]. Jeder kann sich an der Entwicklung beteiligen. Schon lange hat sich eine große Gemeinschaft gebildet; internationale Mailinglisten, Foren, Arbeitsgruppen und Konferenzen werden unter anderem auf den Internet-Seiten der PSA organisiert. Eine wichtige Rolle in der Entwicklung von Python tragen auch die SIGs (Python Special Interest Groups).

Wichtigster Startpunkt zum Kennenlernen von Python ist die Internetseite www.python.org, auf der aktuelle Entwicklungen erhältlich und Verweise zu allen wichtigen Python-Quellen hinterlegt sind.

Python ist für Programmierer anderer Sprachen leicht zu lesen. Eine umfangreiche Einführung würde den Rahmen dieser Arbeit sprengen, daher sei hier auf folgende Literatur verwiesen:

- *Mit Python programmieren* [HM99]
- *Python 2* [Lowis97]
- *Learning Python* [LA99]

2 Grundlagen

Die Grundlage des Erfolgs ist eine klare Linie mit hinreichend vielen Abzweigungen.

Helmar Nahr (*1931), Mathematiker

In diesem Kapitel werden allgemeine Eigenschaften von High-Level-Sprachen und Skriptsprachen vorgestellt. Im Anschluss erfolgt eine Einführung in Methoden der schnellen Softwareentwicklung. Abschließend wird der Einsatz von Programmiersprachen in der Ausbildung diskutiert.

Klassische Modelle der Softwaretechnologie eignen sich oft nur schlecht zur schnellen Softwareentwicklung. Ein häufig eingesetztes Verfahren ist hierbei das Wasserfallmodell [Balzert00]. Der Ablauf der Entwicklung erfolgt hier in getrennten Phasen.

Probleme:

1. Frühes und exaktes Festlegen der Anforderungen erschwert spontane Änderungen.
2. Es bestehen nur schlechte Dialogmöglichkeiten zum Auftraggeber bei ungenauen Angaben, da der Auftraggeber keinen Eindruck von der Software bekommen kann.[1]
3. Falls Änderungen der Anforderungen während des Entwicklungsprozesses auftreten, sind gegebenenfalls Rückschritte nötig.
4. Die Möglichkeiten der Aufwand- und Kostenabschätzung für das Gesamtprojekt sind ungenau.
5. Erst spät im Entwicklungsprozess entstehen lauffähige Versionen, die mit dem Kunden konkret diskutiert werden können.

In geeigneten Fällen kann schnelle Softwareentwicklung diese Probleme durch höhere Flexibilität vermeiden; wobei Skriptsprachen ein bewährtes Werkzeug der schnellen Softwareentwicklung sind.

[1] Wenn in dieser Arbeit die männliche Form verwendet wird, sind Frauen gleichermaßen gemeint, sofern nicht explizit Gegenteiliges behauptet wird.

2.1 Über High-Level-Sprachen und Skriptsprachen

In dieser Arbeit werden die Begriffe *High-Level-Sprache* und *Skriptsprache* immer wieder benutzt. Aus diesem Grund erfolgt an dieser Stelle eine kurze Erklärung der Begriffe:

High-Level-Sprache

Definition:

High-level-language (HLL): *A computer programming language that is primarily designed for, and syntactically oriented to, particular classes of problems and that is essentially independent of the structure of a specific computer or class of computers. Synonym high-order language.* [FS1037C]

Seit Entwicklung der ersten Programmiersprachen ist eine ständige Verbesserung und Abstraktion der Programmiersprachen erkennbar. Diese Entwicklung lässt sich in vier Generationen gliedern:

- **Die erste Generation:**
  Die ersten auf einem Prozessor basierenden Systeme wurden direkt in Maschinensprache programmiert. Jede Zeile Quelltext entspricht dabei genau einer Maschineninstruktion. Als Mittel für häufig gebrauchte Programmsegmente werden Makros eingesetzt.
- **Die zweite Generation:**
  Prozedurale Compilersprachen wie *Fortran, C* und *Pascal* erlauben die Strukturierung und Modularisierung von Software. Bibliotheken bieten dem Entwickler Sammlungen an Funktionalität. Allerdings ist diese Funktionalität meistens relativ starr, beispielsweise durch Beschränkung auf bestimmte Datentypen und Strukturen.
- **Die dritte Generation:**
  Objektorientierte Sprachen mit Klassen, polymorphen Methoden und abstrakten Datentypen erlauben flexibles Arbeiten durch universell nutzbare Bibliotheken.
- **Die vierte Generation:**
  Die sogenannten *High-Level-Sprachen* gelten als leicht erlernbar und bieten Ausnahmebehandlung, Fehlerverfolgung, Introspektion. Grundlegende Datenstrukturen wie Zeichenketten, Listen und Hashes sind bereits Bestandteil der Sprache. Der Entwickler muss sich um maschinenorientierte Strukturen wie Zeigerstrukturen und Speicherverwaltung nicht zu kümmern. Selbst Eigenschaften der Plattform wie Betriebssystem und GUI treten in den Hintergrund.

Skriptsprachen: Moderne Skriptsprachen wie *Perl*, *Tcl*, *Ruby* und *Python* zählen zu den Sprachen der vierten Generation. Doch was genau sind Skriptsprachen? Historisch gesehen enstanden Skriptsprachen aus einfachen Shell-Programmiersprachen wie *sh* und *awk*. Skripte ermöglichen Steueranweisungen zur Automatisierung unterschiedlichster Aufgaben wie beispielsweise Systemverwaltung sowie Konfiguration und Erstellung von benutzerdefinierten Abläufen. Bekannte Beispiele sind Batchdateien unter DOS oder JCL (Job Control Language) der IBM /360. Diese einfachen Skriptsprachen waren aber selbst den damals verfügbaren

Programmiersprachen bzgl. des Funktionsumfangs weit unterlegen, stellten aber oft eine Arbeitserleichterung dar.

Inzwischen wurden Skriptsprachen zu vollwertigen Programmiersprachen weiterentwickelt. Im Folgenden wird dargestellt, welche Eigenschaften eine moderne Skriptsprache kennzeichnen und wo Vorteile sowie Nachteile gegenüber anderen Sprachen wie beispielsweise den Compilersprachen *Pascal, C++, Eiffel* liegen.

Folgende technischen Merkmale kennzeichnen moderne Skriptsprachen:

- Nutzung eines Interpreters
  Bei Compilersprachen wird der Programmcode in einen in der Regel systemabhängigen Maschinencode übersetzt, der dann von Prozessor ausgeführt werden kann. Das erfordert zeitaufwendige Compiler- und Linkeraufrufe.
  Im Gegensatz dazu wird bei Interpretersprachen das Programm oft direkt, also ohne vollständige Übersetzung, ausgeführt. Die Übersetzung erfolgt während der Eingabe oder beim Starten des Programms in sogenannte Tokens oder einen Binärcode.
  Dadurch sind Interpreter zwar langsamer in der Programmausführung, ermöglichen aber die schnelle und direkte Ausführung eines Programms. Gerade in der Entwicklungsphase, in der häufig Änderungen am Programmcode vorgenommen werden, spart ein Interpreter sehr viel Zeit, da Änderungen direkt im Interpreter vorgenommen werden können.
  Ein wesentlicher Vorteil von Interpretern ist die Plattform-Unabhängigkeit. Da auf Maschinencode verzichtet wird, können beispielsweise unter UNIX entwickelte Skripte grundsätzlich auch unter Windows ausgeführt werden.

- Fehlertoleranz
  Der Interpreter einer Skriptsprache steht zwischen Programm und Betriebssystem / Hardware. Bei fehlerhafter Programmausführung kann der Interpreter diese Ausnahmen (Exceptions) kontrolliert abfangen und behandeln.

- Dialog mit dem Interpreter
  Einzelne Anweisungen können durch den Benutzer direkt im Interpreter eingegeben werden oder zumindest einfach an den Interpreter übergeben werden. So kann beispielsweise auch jede Routine manuell aufgerufen werden.

- Dynamik
  Die Übersetzung von Quelltexten ist während der Laufzeit möglich. So kann ein interpretiertes Programm während seiner Programmausführung Quelltext erzeugen, einlesen oder eingeben lassen und diesen direkt ausführen.

- Leichte Erlernbarkeit
  Anwender und nicht nur professionelle Entwickler sollten in der Lage sein, Skripte zu erstellen. Insbesondere eine einfache Syntax kann dazu beitragen.

- Erweiterbarkeit und Einbettbarkeit
  Skripte bieten die Möglichkeit, Module anderer Sprachen zu nutzen und auch umgekehrt aus anderen Sprachen, meist C/C++, genutzt werden zu können.

- Wiederverwendbarkeit
  Prozedurale und modulare Konzepte sowie leichte Erweiterbarkeit der Skripte sollen einen möglichst universellen Einsatz einmal erstellter Module erlauben.

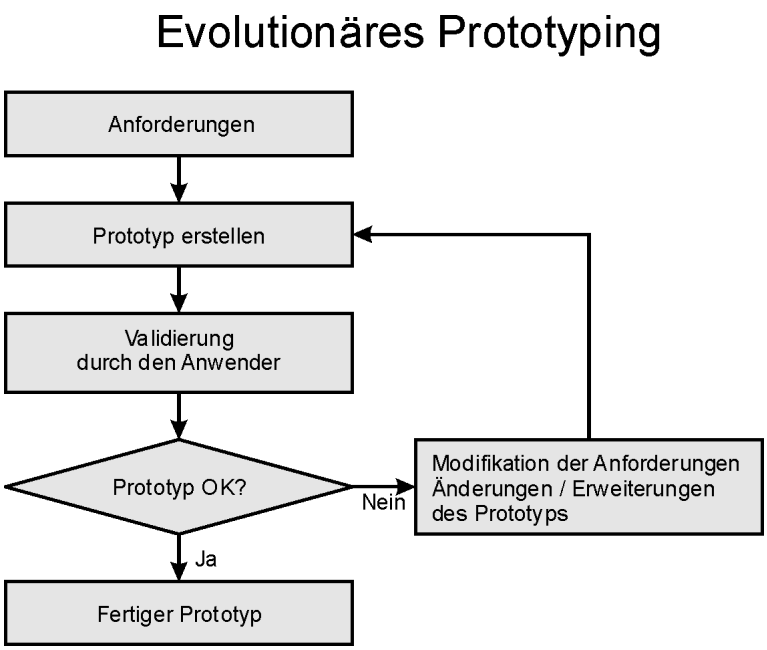


Abbildung 1: *Evolutionäres Prototyping*

- Modularität
  Skriptsprachen arbeiten modular, Programme können in einzelne Teilmodule zerlegt werden. Bibliotheken bieten Modulsammlungen für häufig wiederkehrende Aufgaben.

- High-Level-Sprache
  Skriptsprachen zählen zu den High-Level-Sprachen. Skriptsprachen bieten bereits grundlegende Datentypen wie Listen und enthalten eine Speicherverwaltung, die einzelne nicht mehr benötigte Objekte selbst freigibt (Garbage Collection). Diese Datentypen werden auch syntaktisch unterstützt, beispielsweise durch spezielle Literale und Operatoren. In *Low-Level-Sprachen* wie C und C++ müssen solche Datentypen durch komplexe Zeigerstrukturen implementiert oder externe Bibliotheken verwendet werden.

2.2 Konzepte der schnellen Softwareentwicklung

Nachfolgend werden Einsatzgebiete und Konzepte aus der Softwaretechnologie zur schnellen Softwareentwicklung vorgestellt.

Prototyping

Prototyping bezeichnet alle Aktivitäten zur Entwicklung eines Software-Produkts, welches dann auch Software-Prototyp genannt wird. Es dient als Kommunikationsmodell zwischen Auftraggeber und Entwickler sowie als experimentelles System zur Klärung der Benutzeranforderungen und Schnittstellen, der Systemarchitektur und verschiedener Lösungsansätze.

Prototyping eignet sich besonders, wenn der Auftraggeber die Anforderung nicht genau spezifizieren kann. Ein Prototyp kann bereits in einem sehr frühen Stadium der Entwicklung dem Auftraggeber einen Eindruck eines möglichen Endproduktes geben.

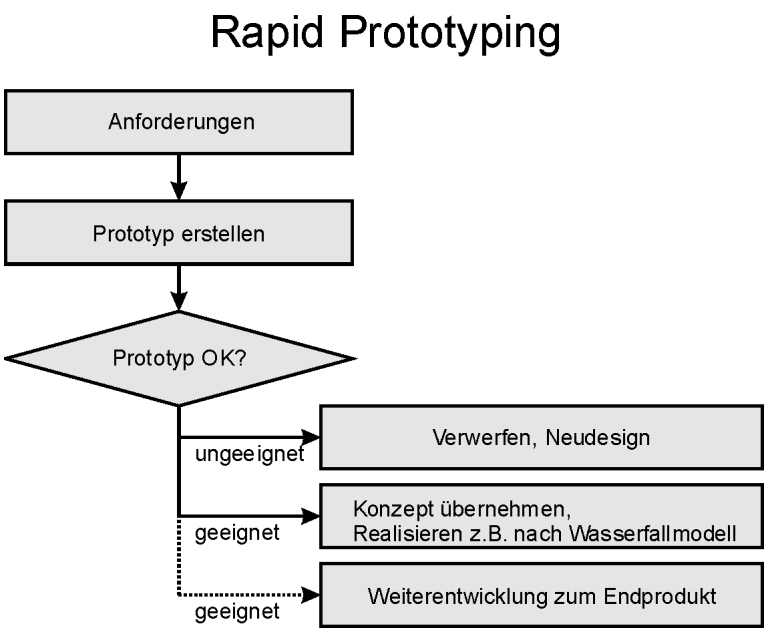


Abbildung 2: *Rapid Prototyping*

Rapid Prototyping

Der Begriff des *Rapid Prototyping* wurde von den Ingenieurwissenschaften in die Informatik übernommen. Es wird in [WJC93] definiert:

Rapid prototyping: *In a design process, early development of a small-scale prototype used to test out certain key features of the design. Most useful for large-scale projects.*

Ziel des Rapid Prototypings ist es, möglichst schnell einen Software-Prototypen zu erstellen, der die wichtigsten Funktionen des Endproduktes demonstriert. Dabei kann es sich beispielsweise um eine Benutzungsoberfläche handeln, die dem Auftraggeber eine Vorschau auf die endgültige Lösung geben kann. Einzelne Funktionen können in diesem Stadium noch durch Dummy-Funktionen[2] realisiert sein. Oft wird Rapid Prototyping auch eingesetzt, um einzelne Elemente eines geplanten großen Projektes zu testen. Dabei kann es sich beispielsweise um Machbarkeitsstudien sowie Strukturtests und Effizienzanalysen von Datenbanken und Kommunikationsabläufen handeln. Weitere denkbare Einsatzgebiete werden beispielsweise in [WJC93] vorgestellt. In vielen Fällen dient der Prototyp nur als Muster und wird anschließend verworfen. Dieses Vorgehen wird auch *Throw-Away-Prototyping* genannt.

In vielen Projekten ist es ein allerdings auch möglich, einen akzeptierten Prototypen zu einem vollwertigen Endprodukt weiterzuentwickeln, das sogenannte *Stepwise Refinement* (schrittweise Verfeinerung), was gegebenenfalls in mehreren Zyklen geschehen kann.

Der wichtigste technologische Aspekt beim Rapid Prototyping ist, ein Produkt möglichst schnell und kostengünstig herzustellen. Dieses wird unterstützt durch geeignete Werkzeuge und insbesondere auch die verwendete Programmiersprache.

[2]Funktionen, die eine mögliche Arbeitsweise simulieren

Vorteile:

- Frühzeitig durch Kunden evaluierbarer Prototyp
- Prototyp kann dem Kunden als Angebot vorgelegt werden
- Kosten müssen nicht bereits zu Beginn festgelegt werden, der reale Aufwand wird nach Prototyperstellung besser kalkulierbar
- Projektplanung kann in Teilprojekte zerlegt werden

Nachteile:

- Verwerfen des Prototypen, unvergütete Arbeit für den Entwurf des Prototypen
- Evolutionäre Entwicklung: unter Umständen fallen übermäßig viele Änderungen und Verwürfe an.

Die Ideen des Rapid Prototyping sind unter verschiedenen Gesichtspunkten in der Literatur dargestellt:

- *The Rapid Prototyping Model* [CU02]
- *Rapid Prototyping as an instructional design* [Hoffmann02]
- *Cognitive Approaches to Instructional Design* [Dunn02]
- *Software Engineering* [Sommer00]

Die dokumentierten Eigenschaften bestätigen die genannten Vorteile. Darüber hinaus wird auf methodische Varianten hingewiesen, die im Folgenden zusammenfassend dargestellt werden.

Code and Fix

Code and Fix bezeichnet das Lösen und Implementieren von Kleinstproblemen durch einen einzelnen Entwickler innerhalb kurzer Zeit. Das Vorgehen bleibt dem Entwickler selbst überlassen, typischerweise wird sehr früh mit der Implememtierung begonnen und anschließend Fehlersuche und Strukturverbesserung durchgeführt.

Abbildung 3: *Code and Fix*

Vorteile:

- Schnelle und unkomplizierte Lösung bei kleinen Aufgaben möglich.

Nachteile:

- Fehleranfällig, da Entwicklung und Kontrolle durch nur eine Person erfolgt.

Extreme Programming

Ein weiteres Verfahren der schnellen Softwareentwicklung wurde von Kent Beck [Beck99] propagiert.

Extreme Programming (kurz XP): *Eine kleine Gruppe (bis ca. 5 Personen) von Programmierern löst gemeinsam in relativ kurzer Zeit ein Problem.*

Das Vorgehen beim XP ähnelt dem *Evolutionären Prototyping*. Analyse, Entwurf, Implementierung und Test werden in einem kleinen Team kontinuierlich bzw. im schnellem Wechsel durchgeführt. Jeder der Entwickler ist an allen Phasen der Entwicklung beteiligt.

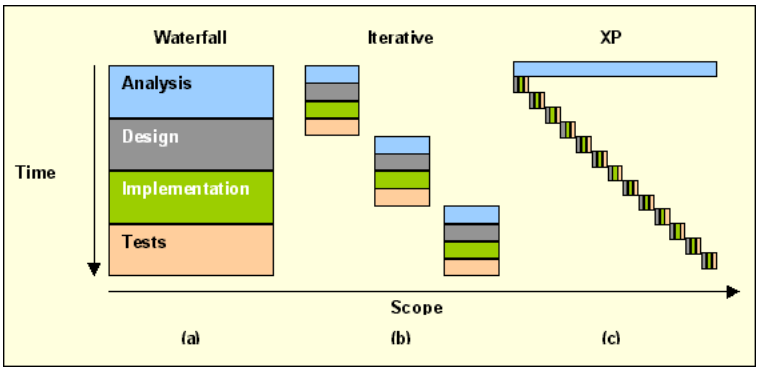


Abbildung 4: *Extreme Programming* [Acebal01]

XP ist geeignet für kleine Projekte in nicht sicherheitskritischen Bereichen. Es zeichnet sich aus durch:

- Verzicht auf Phasen der Softwareentwicklung
- keine aufwendige Analyse und Design, Analyse erfolgt parallel zur Implementierung
- Kleine Teams / Pair Programming Teams
- Intensive Kommunikation innerhalb des Entwicklerteams
- Systematisches *Evolutionäres Prototyping* mit Kontakt zum Auftraggeber
- Dokumentation erfolgt im Quelltext
- Testdurchläufe erfolgen schon während der Kodierung
- Testfälle werden zusammen mit dem Code geschrieben

In der Praxis hat sich XP bei überschaubaren Projekten als brauchbare Alternative erwiesen und ermöglicht oft schnelle und kostengünstige Entwicklungen. Dennoch wird XP zur Zeit noch heftig diskutiert.

Literatur:

- *Extreme Programming* *[Beck99]*
- *Extreme Programming Explained* *[Beck00]*
- *A new method of Software Development* *[Acebal01]*
- *Extremeprogramming.org* *[Wells02]*

Pair Programming

Definition:

Pair Programming: *Zwei Entwickler lösen gemeinsam ein Problem, häufig an nur einem Rechner. Die Vorgehensweise ist dabei vergleichbar zu Extreme Programming. Meistens übernimmt wechselweise einer der Entwickler die Programmierung, der andere die Kontrolle. Ideen zur Problemlösung werden gemeinsam ausgearbeitet.*

Vorteile:

- Gegenseitige Kontrolle
- Jeder Entwickler kennt sich gut im Projekt aus

Nachteile:

- Begrenzte Kapazität, da nur 2 Entwickler beteiligt sind

Pair Programming ist neben Gruppenarbeit eine an Schulen häufig anzutreffende Situation. Auch hier wird oft ein Rechner von zwei Schülern zur gemeinsamen Aufgabenlösung eingesetzt.

Literatur:

- *A new method of Software Development: eXtreme Programming* [Acebal01]
- *Pairprogramming, and Extreme Programming practice* [BSW02]

2.3 Softwareentwicklung in der Ausbildung

Für Lehr- und Lernprozesse[3] sind Problemstellungen typisch, die zentrale und allgemein bildende Konzepte der Informatik thematisieren. Die Implementierung der Modellierung solcher Probleme muss die engen zeitlichen Restriktionen[4] des Informatikunterrichts berücksichtigen. Aus diesem Grund sind eine hohe Modularisierung und kurze Programme anzustreben.

Da der Informatikunterricht typischerweise in 45- oder 90-Minuten-Abschnitten zergliedert ist (Einzelstunde/Doppelstunde), ist es nötig, die Umsetzung der Modellierung durch Lehrkräfte so planen zu können, dass maximale Modularisierung und Kapselung möglich ist. Diese Rahmenbedingungen führen zur Präferierung objektorientierter Modellierung, die einerseits den Schwerpunkt Analyse betonen, aber andererseits die Möglichkeiten zur unabhängigen Implementierung einzelner Klassen und zu unabhängigen Tests einzelner Klassen umfasst. Dies führt zu der Anforderung, dass die zu implementierenden Klassen arbeitsteilig erstellt werden.

Die Entwicklung erfolgt dabei in der Regel in kleinen Gruppen, vergleichbar zum *Extreme Programming*. Dies ist häufig darauf zurückzuführen, dass sich jeweils zwei Schüler[5] ein Informatiksystem zur Arbeit teilen müssen, beinhaltet aber zugleich die didaktische Gestaltung im Sinne

[3] In dieser Arbeit wird die fachdidaktische Eignung einer Programmiersprache zur Einführung in die Grundlagen der Programmierung thematisiert. In fortgeschrittenen Phasen der Ausbildung können sich weitere Anforderungen ergeben, welche in dieser Arbeit nur ergänzend dargestellt werden.

[4] typischerweise 2 – 3 Stunden / Woche

[5] Wenn in dieser Arbeit von Schülern und Lehrern gesprochen wird, sind Entwickler und Ausbilder an weiterführenden Schulen und Hochschulen damit nicht ausgeschlossen.

des *Pair Programmings,* um gerade bei der oben genannten arbeitsteiligen Implementierung qualitativ hochwertige Lösungen erzielen zu können. Ebenso wie bei schneller Softwareentwicklung sind möglichst einfache Änderungsmöglichkeiten an der Software gefragt.

Zum Erlernen der konkreten Sprache ist eine einfache Syntax und Semantik erforderlich, die möglichst bruchfrei und frei von überflüssigen Anweisungen ist. Eine Nähe zu Pseudocode oder einer natürlichen Sprache trägt dabei zur Lesbarkeit von Quelltexten bei. Die Sprache sollte Sprachstrukturen zur Verfügung stellen, die der menschlichen Denkweise und dem menschlichen Problemlöseverhalten angepasst sind.

Im Unterricht sollte aber nicht die konkrete Programmiersprache im Vordergrund stehen, sondern die allgemeinen Ideen des Prozesses von der Modellierung bis zur Implementierung.

Dazu gehört:

- die Modellierung des Problems als Ausschnitt aus der realen Welt.
- die Lösung eines Problems mit unterschiedlichen Paradigmen, einschließlich des Variantenvergleichs zwischen Lösungen unterschiedlicher Paradigmen. [Schubert00]
- das Verstehen von grundlegenden Typen und Datenstrukturen wie Feldern, Listen und Hashes. [Schwill02, Schwill93]
- das Verstehen und Austauschen der Ideen von Algorithmen. [Schwill02, Schwill93]
- die Schüler zu motivieren, sich mit der Informatik und der Programmierung zu beschäftigen.
- die didaktische Gestaltung, das heisst die lerngruppenbezogene Vereinfachung, ohne eine fachlich falsche Vorstellung zu entwickeln. [Humbert01]
- die Anbindung an die Lebens- und Erfahrungswelt der Schüler. [Humbert01]
- die Unterstützung von Gruppen- bzw. Projektarbeit. [Schwill02]

Auch wenn sich einige der Punkte primär auf die Gestaltung des Unterrichts beziehen, die in dieser Arbeit nicht thematisiert wird, kann der Unterricht durch die Wahl der Sprache sowie geeignete Lern- und Entwicklungsumgebungen mehr oder weniger unterstützt werden.

Die in Kapitel 3 aufzustellenden Kriterien werden daher neben sprachlichen Eigenschaften Werkzeuge für den Einsatz im Informatikunterricht berücksichtigen. Ferner wird untersucht, inwiefern die Anforderungen an eine Programmiersprache zur informatischen Bildung denen der schnellen Softwareentwicklung ähneln.

Weitere Literatur:

- *Why Java is not my favorite first-course language* [Boszormenyi98]

3 Vorgehensweise

Wer fremde Sprachen nicht kennt,
weiß nichts von seiner eigenen.

J. W. Goethe (1749 – 1832)

Im vorangegangenen Kapitel sind die Ziele der schnellen Softwareentwicklung erkannt worden. Es werden nun Kriterien gesucht, die eine Programmiersprache erfüllen sollte, um die Ziele möglichst produktiv zu erreichen.

Die Wahl der Sprache und der Werkzeuge kann hohen Einfluss auf die Produktivität haben. Nachfolgend wird ein Katalog von Kriterien erstellt, anhand dessen eine Programmiersprache auf ihre Eignung für schnelle Softwareentwicklung untersucht werden kann. Dabei soll nicht nur auf die Interessen professioneller Entwickler eingegangen werden, sondern auch auf die Einsatzmöglichkeiten im Ausbildungsbereich. Es zeigt sich, dass beide Gruppen zahlreiche gemeinsame Interessen haben, aber auch dass es jeweils Interessen gibt, die nur eine der beiden Gruppen hat. Es wird daher auch beachtet, ob es zu Interessenskonflikten kommen kann. Die Situation wird in Abbildung 5 dargestellt:

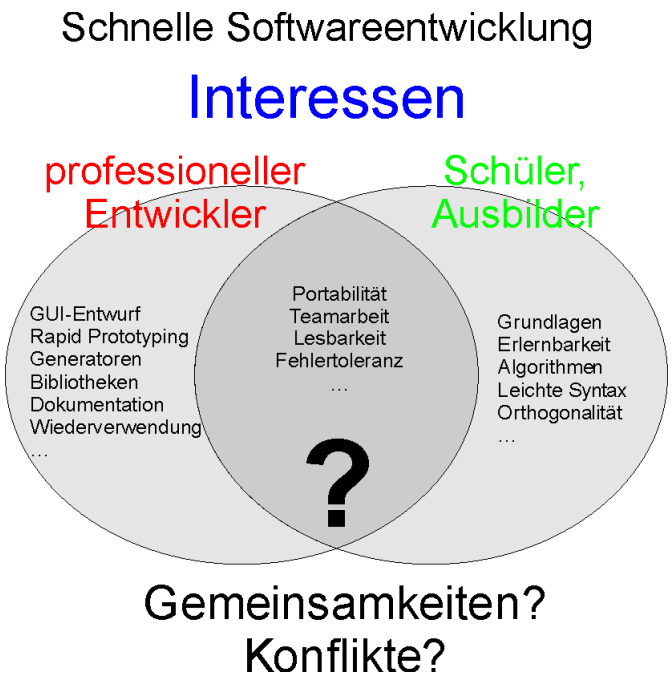


Abbildung 5: *Interessen bei schneller Softwareentwicklung*

Hypothese:

Es gibt viele Gemeinsamkeiten bei Kriterien für schnelle Softwareentwicklung und fachdidaktischen Anforderungen.

Daher werden in dieser Arbeit beide Merkmale parallel behandelt. Dabei erfolgt nur eine grobe Einordnung der Kriterien zu den jeweiligen Interessensgruppen, um eine Doppelaufzählung zu vermeiden.

Ein allgemeiner Katalog von Kriterien zur Bewertung von Sprachen wurde auf der Linux-Expo '01 vorgestellt [Pixel01]. Dieser Katalog umfasst insbesondere syntaktische Merkmale. Bezogen auf schnelle Softwareentwicklung und Fachdidaktik ergeben sich weitere Anforderungen, die nicht nur syntaktischer Art sind.

3.1 Aufstellen von Kriterien

Viele Anforderungen an eine Programmiersprache zur schnellen Softwareentwicklung ergeben sich aus den alltäglichen Erfahrungen eines Softwareentwicklers. Berücksichtigt werden sollen auch die zyklisch wiederholten Phasen der Entwicklung von Prototypen:

Kundenkommunikation, Projektplanung, Entwurf, Konstruktion, Erprobung.

In [Waclena97] und [Dressler01] wurden Kataloge von einigen allgemeinen Anforderungen an Programmiersprachen vorgestellt, wobei hier kein Bezug auf schnelle Softwareentwicklung erfolgt. Weitere Kriterien ergeben sich unmittelbar aus den Merkmalen der unterschiedlichen Softwareentwicklungsmethoden. Als weiterer Ansatzpunkt zum Auffinden von Anforderungen wurden vom Autor einige Softwareentwickler zu ihren Erfahrungen befragt. Im Folgenden werden die erarbeiteten Kriterien vorgestellt. Diese sind zunächst nur grob in allgemeine Kriterien und fachdidaktische Kriterien gegliedert. Es folgt eine tabellarische Darstellung mit einer genaueren Zuordnung der Anforderungen zu den jeweiligen Interessensgruppen.

Allgemeine Anforderungen:

Very High Level Language
: Es soll eine möglichst abstrakte und systemunabhängige Softwareentwicklung erfolgen sowie kompakter Quelltext durch Integration von Basisdatentypen möglich sein. Der Begriff *Very-High-Level-Language* schließt die meisten der nachfolgend genannten Merkmale ein.

Automatische Speicherverwaltung
: Der Entwickler muss sich nicht um das Reservieren und Freigeben von Speicher kümmern.

Portabilität und Dateizugriff
: Erstellte Programme sollten in verschiedenen Umgebungen lauffähig sein, mit keinen oder nur minimalen Änderungen. Oft steht bei der Entwicklung des Prototypen die Zielplattform noch nicht fest. Des Weiteren ist im im Kontext vernetzter und interaktiver heterogenener Informatiksysteme[6] Kompatibilität zwischen unterschiedlichen Plattformen allgemein wünschenswert. Auf allen Plattformen sollte uneingeschränkter, aber gleichzeitig kompatibler Zugriff auf das Dateisystem möglich sein. Ebenso wichtig ist eine Standardisierung der Sprache selbst.

[6]Informatiksystem wird in der vorliegenden Arbeit als eine spezifische Zusammenstellung von Hardware, Software und Netzwerkverbindungen zur Lösung eines Anwendungsproblems verstanden.

Einfache Syntax + Semantik

Die Sprache soll den Entwickler durch leicht lesbaren Code unterstützen. Lesbarkeit ist zwar ein subjektiver Begriff, allerdings kann eine klare und einheitliche Notationen das Verständnis erleichtern. In [Pixel01] wird in diesem Zusammenhang eine Reihe syntaktischer Sprachmerkmale vorgestellt.

Datentypen

Die Integration von Datentypen in die Syntax erlaubt schnellen Umgang mit häufig anzutreffenden Strukturen wie Strings, Listen und Hashes. Programmiersprachen imperativen Ursprungs (oft im Gegensatz zu funktionalen und prädikativen Programmiersprachen) arbeiten auf Datentypen, die Bestandteil der jeweiligen Sprachdefinition sind. Ausgehend von maschinennaher Typisierung (Integer, Real, Character) werden nach und nach die Kompositionstypen Datensatz (Record, zuerst in COBOL) und Liste (Array, imperativ zuerst in COBOL und FORTRAN) als Bestandteile in Programmiersprachen zur Verfügung gestellt. Da häufig mit Zeichenketten gearbeitet werden muss, wird für eine „unveränderliche Liste aus einzelnen Zeichen" ein besonderer Datentyp (Zeichenkette, engl. String) angeboten (Problem Orthogonalität). Für häufig auftretende Fragestellungen ist die direkte Unterstützung weiterer Datentypen wie Hashlisten (Wörterbuchmetapher) und komplexer Zahlen wünschenswert. Alle Datentypen sollten beliebig miteinander kombinierbar sein und zum Aufbau eigener Datenstrukturen auch rekursiv benutzt werden können (Orthogonalität).

Funktionskonzept

Datentypen sollten sowohl als Argument an Funktionen wie auch als Rückgabewert einer Funktion erlaubt sein. Das Funktionskonzept sollte nicht durch beschränkte Rekursion oder Parameterübergabe eingegrenzt sein. Funktionen selbst sollten als Parameter an Funktionen übergeben werden können, ebenso sollten Funktionen als Ergebnis eine Funktion liefern können. Des Weiteren sollten Funktionen lokal arbeiten und auch lokal definiert werden können. [Balzert00, Waclena97]

Objektorientierung

Objektorientierte Programmierung sollte in die Sprache integriert sein. Neben sauberer Realisierung objektorientierter Konzepte wie Vererbung, Mehrfachvererbung, Kapselung ist gleichzeitig eine möglichst einfache Deklaration frei von überflüssigen Spezifikationen wünschenswert. Eine automatische Speicherverwaltung auf Objekten und insbesondere die selbstständige Freigabe von freigewordenem Objektspeicher ist ebenfalls eine wichtige Eigenschaft.

Schneller GUI-Entwurf

Diese Anforderung ergibt sich aus der Definition des Prototypings. Der Entwurf von Benutzungsoberflächen ist ein wesentliches Ziel von Rapid Prototyping. Der Entwickler benötigt geeignete Bibliotheken und Gestaltungsmittel.

Geeignete Hilfsmittel und Werkzeuge

Die Sprache und ihre Entwicklungsumgebung sollen die schnelle Softwareentwicklung unterstützen. Zeitraubende Installationen, Konfigurieren der Entwicklungsumgebung, Erstellen von Projektdateien oder Makefiles sollte möglichst entfallen. Die schnelle Entwicklung von grafischen Benutzungsoberflächen für einen Prototyp sollte mit Entwurfswerkzeugen möglich sein. Auf die Sprache angepasste Editoren oder Entwicklungsumgebungen sollten die dazu nötige Arbeit unterstützen. Quelltextgeneratoren können bei

der Umsetzung vorhandener Modelle die Arbeit vereinfachen. Diese und weitere Werkzeuge erleichtern die Arbeit der schnellen Softwareentwicklung und werden daher auch als Kriterium aufgeführt.

Modularisierung und Wiederverwendbarkeit

Das "Zusammenstecken" von Software aus fertigen Komponenten (Baukastenprinzip) ist kostengünstig und spart Zeit. Möglichst große und vielseitige Bibliotheken bestimmen häufig die Geschwindigkeit der Softwareentwicklung. Kann ein Entwickler nach dem Baukastenprinzip auf fertige Bibliotheken zugreifen, erspart er sich zeitintensive Eigenentwicklungen. Hilfreich sind möglichst vielseitige, polymorphe Bibliotheken sowie gute Möglichkeiten der Softwaredokumentation. Ein Modulsystem ist für eine heutige Programmiersprache unentbehrlich und sollte ein gewisses Maß an Komfort und Sicherheit bieten. Schnelle sowie kostengünstige Verfügbarkeit, einfache Installation und gute Recherchemöglichkeiten zum Finden von Bibliotheken zu einem gegebenen Problem sind wichtige Faktoren.

Erweitern und Einbetten

(Extending und Embedding) Neben Bibliotheken ist auch die Nutzung externer Programme oder Module eine Anforderung, die im Alltag der schnellen Softwareentwicklung erforderlich ist. Skriptsprachen wurden ursprünglich zu diesem Zweck entwickelt. Auch umgekehrt sollten sich Skripte von anderen Sprachen aus nutzen lassen.

Dokumentation

Die Dokumentation erleichtert die Fehlersuche und Wartung eigener Software und erleichtert spätere Wiederverwertung bereits vorhandener Module. Dokumentation innerhalb des Quelltextes verschafft Entwicklern Einblick in die Funktionsweise fremder Quelltexte. Dokumentation als Bestandteil des Zielcodes – im Gegensatz zum Quelltext – vereinfacht die Nutzung fremder und gegebenenfalls geschützter Bausteine.

Softwaretest

Insbesondere bei der Arbeit mit Prototypen sind häufige Tests einzelner Komponenten wie auch des gesamten Prototyps nötig. Wie im Ingenieurwesen, aus dem der Begriff Prototyp entlehnt wurde, sind auch in der Softwareentwicklung Fehlerbeseitigungen, Anpassungen und Verbesserungen durch direktes Arbeiten am Prototypen erforderlich. Die Möglichkeiten der Fehlererkennung und des interaktiven Eingreifens zur Wartung durch einen Entwickler sind dabei wichtige Anforderungen an die Entwicklungsumgebung. Häufig besteht ein Produkt aus einzelnen Modulen; auch deren korrekte Funktion sollte unabhängig getestet werden können. Des Weiteren ist schnelle Ausführung ein wichtiges Kriterium, welches Entwickler im Alltag erwarten. Nach kleinen Änderungen sollte sich ein Programm ohne lange Wartezeiten ausführen lassen.

Fehlertoleranz

Während der experimentellen Arbeit an Prototypen werden häufig Testdurchläufe durchgeführt. Prototypen weisen anfangs oft noch zahlreiche Probleme und Fehler auf. Die Erkennung und Behandlungsmöglichkeit von sogenannten Ausnahmen (exceptions) ist daher ein wichtiges Merkmal bei der Auswahl einer Programmierumgebung. Zusätzliche Möglichkeiten, die Sicherheit zu erhöhen und Fehler zu erkennen, sind sogenannte Zusicherungen (assertions, vgl. [Eiffel02], [Meyer90]).

Fehlersuche und Fehlervermeidung

Häufig enthalten erste Implementierungen zu Beginn Fehler und Inkonsistenzen, verursacht beispielsweise durch Tippfehler oder falsche Annahmen. Programmiersprache und Entwicklungsumgebung sollten den Entwickler bei der Lokalisierung von Fehlern unterstützen. Im Fehlerfall sollte der Entwickler Möglichkeiten haben, betroffene Parameter und Objekte interaktiv während der Laufzeit zu untersuchen und kritische Programmteile genau zu beobachten. Es sollte sich zu jeder Zeit ermitteln lassen, welchen Typ ein Objekt hat, und welche Daten es enthält. Fehlervermeidung kann durch statische Typisierung und Zusicherungen erreicht werden, des Weiteren trägt auch die Syntax der Sprache dazu bei.

Umstrukturierung (Refactoring)

Refactoring wird das Umstrukturieren von Software genannt, ohne dadurch das Verhalten nach außen zu ändern. Dies geschieht typischerweise zur Verbesserung von Lesbarkeit, Wartbarkeit oder bei allgemeinen Entwurfsänderungen. Prototypen entwickeln sich evolutionär, daher fallen häufig Änderungen im Sinne von Refactoring an [Kniesel00].

Nebenläufigkeit (Threading)

Neben der Synchronisation und Koordination der Prozesse ist insbesondere die Kopplung über Prozesskommunikation wie etwa TCP/IP, CORBA, JMS oder SOAP von Bedeutung, wozu geeignete Bibliotheken erforderlich sind.

Sonstiges

Von Entwicklern wurden weitere Anforderungen genannt, die nicht unter die bereits genannten Oberbegriffe eingeordnet werden können:

- Persistente Datenspeicherung
- nicht zuviel Einbußen an Leistung gegenüber anderen Sprachen
- Nutzung in Realzeitsystemen
- Eignung für verteilte Systeme (Agenten usw.)

Weitere Kriterien, bezogen auf die informatische Bildung:

Bei der Untersuchung einer Programmiersprache zur Eignung für die informatische Bildung sollen nicht alleine sprachliche Merkmale berücksichtigt werden. Weitere Anforderungen an Programmiersprachen zum Einsatz im Informatikunterricht werden beispielsweise in [Schubert00, Schubert01, Schwill02, Schwill93, Balzert79] genannt.

Um das "Klebenbleiben" an gerätespezifischen Kenntnissen zu Ungunsten des weiteren Verständnisses zu vermeiden, sollte unbedingt zunächst die Problemanalyse, Modellbildung und algorithmische Problemlösung an einer Vielzahl von Beispielen aus unterschiedlichen Problembereichen geübt werden. [...] Es kommt nicht darauf an, dass eine Programmiersprache in kürzester Zeit erlernbar ist, sondern dass sie problemadäquate Sprachstrukturen zur Verfügung stellt, die der menschlichen Denkweise und dem menschlichen Problemlöseverhalten angepasst sind. [Balzert79]

Die nachfolgend aufgestellten Kriterien sollen daher neben der Les- und Lernbarkeit einer Sprache auch den Übergang vom Problem zum Modell und schließlich zur Implementierung

mittels einer konkreten Zielsprache berücksichtigen. Dies schließt insbesondere auch die Umsetzung mehrerer Paradigmen ein, da Schüler im Zusammenhang mit der informatischen Bildung verschiedene Paradigmen kennenlernen sollten (vgl. [GI2000]).

Die fachdidaktischen Kriterien sind primär auf die Einführung in die Programmierung ausgerichtet. In weit fortgeschrittenen Phasen der Ausbildung, wie etwa systemnaher Programmierung, können sich weitere Anforderungen ergeben, die in dieser Arbeit nur teilweise angesprochen werden.

Paradigmen für unterschiedliche Lösungskonzepte:

Neben objektorientierten Ansätzen führen oft auch funktionale oder wissensbasierte Methoden zum Ziel. Beispiel:
Im Bundeswettbewerb Informatik hat ein Schüler die Lösung einer gestellten Aufgabe mit ein paar Zeilen Prolog eingereicht, war sich aber nicht sicher, ob diese Form der Lösung akzeptiert wird. Er hat daraufhin einen kleinen Prolog-Interpreter geschrieben und damit gezeigt, dass er das Problem auch imperativ lösen kann.
Schüler sollten die verschiedene Paradigmen kennenlernen und durch einen Variantenvergleich die Stärken und Schwächen der Nutzung eines bestimmten Paradigmas zur Lösung eines konkreten Problems einschätzen können. Eine typische Aufgabenstellung ist beispielsweise die Modellierung des Pizza-Beispiels [BS97] unter den unterschiedlichen Paradigmen.
Jedes dieser Paradigmen sollte sich sauber und bruchfrei in der eingesetzten Programmiersprache implementieren lassen. Dabei sollte die Syntax der Sprache die Integration der Paradigmen unterstützen [Humbert02]. Wichtig ist dabei eine klare und widerspruchsfreie Realisierung der Konzepte.

Die konkrete Programmiersprache ist die Schnittstelle zwischen Mensch und Maschine. Historisch gesehen wurden Programmiersprachen zum Austausch von Algorithmen unter Menschen entwickelt, um über Probleme und Lösungen zu diskutieren (vgl. Algol59). Gleichzeitig dienen Sprachen zur Vereinfachung der Programmierung von Computern. Erst später erfolgte die Nutzung zur Unterstützung der Modellierung in speziellen Anwendungsfeldern (vgl. XML).

Die Sprache ist lediglich das Mittel zum Zweck. Das Problem bzw. der Algorithmus soll in der informatischen Bildung im Vordergrund stehen, nicht die Vermittlung syntaktischer und semantischer Feinheiten.

Bezogen auf die Implementierungsphase werden in dieser Arbeit untersucht:

Einfache Lesbarkeit und Erlernbarkeit

Beim Erlernen einer konkreten Programmiersprache sind eine einfache Syntax und Semantik eines der wichtigsten Kriterien, die einen schnellen Zugang erlauben [Rossum02].

Ist die Sprache geeignet zum Austausch von Algorithmen?

Algol59 war die erste Austauschsprache für Algorithmen. Bis heute wird eine Mischung aus Ada, Algol, Pascal, Modula und Klartext als “Pseudocode” zum Austausch von Algorithmen in der Literatur verwendet [Schwill93].

Orthogonale Syntax

Eine klare Syntax ohne Sonderfälle und Ausnahmen erleichtert die Erlernbarkeit. Sie muss daher als eines der zentralen Kriterien angesehen werden. Ziel ist nach [Schwill02]

ein minimales System mit maximaler Komplexität, mit anderen Worten: Die Sprache muss alles Nötige bieten, aber Unnötiges vermeiden. Ein Gegenbeispiel sind C-Makros: Hier wird im Extremfall eine Subsprache definiert oder die Syntax der eigentlichen Sprache durch die lexikalischen Ersetzungen durch Makros unterlaufen. Ebenso wichtig ist die Konsistenz der Konstrukte, es sollte hier möglichst keine Varianten von gleichartigen Sprachelementen geben.

Klare Fehlermeldungen
Möglichst eindeutige Fehlermeldungen erleichtern die Fehlerbeseitigung (auch als Debugging bezeichnet) und insbesondere das schnelle Erkennen der Syntaxfehler, die besonders von Anfängern gemacht werden.

Probleme und Schwierigkeiten
Beim Erlernen der Programmierung werden in den unterschiedlichen Sprachen typische Fehler gemacht. Welche Probleme und Fehlvorstellungen haben Einsteiger in der konkreten Sprache?

Das Erlernen der Programmierung kann durch geeignete Werkzeuge erheblich erleichtert werden. Dabei müssen unterschiedliche Zielgruppen berücksichtigt werden: Im Informatikunterricht werden andere Anforderungen gestellt als in Schnellkursen für erfahrene Entwickler.

Eingabehilfen
In diesem Zusammenhang werden Editoren und Entwicklungsumgebungen auf ihre Einsatzmöglichkeiten untersucht.

Tutorials
Sind Anleitungen verfügbar, die einem Entwickler mit Erfahrungen in anderen Sprachen einen Schnelleinstig in die Programmiersprache ermöglichen?

Lernumgebungen
Es soll untersucht werden, ob geeignete Lernumgebungen und Modellwelten für den schulischen Einsatz vorhanden sind, beispielsweise zur Visualisierung von Algorithmen und Programmabläufen. Lassen sich typische Aufgabenklassen realisieren?

Motivation
Frühe Erfolgserlebnisse und Unterstützung der Experimentierfreudigkeit fördern schnelles Lernen. Hat die Programmiersprache den Reiz eines Bau- oder Experimentierkastens?

Gemeinsamkeiten und Differenzen:

Die oben erfolgte Unterteilung in Kriterien für schnelle Softwareentwicklung und Eignung zur informatischen Bildung stellt keine exklusive Zuordnung dar, sondern nur eine grobe Untergliederung.

Beispiel einer Gemeinsamkeit:

- Modularisierung mit exakten Schnittstellen ist zur schnellen Softwareentwicklung ebenso wichtig wie für den Projektunterricht. Modularisierung gestattet Wiederverwendung von erarbeiteter Funktionalität zu einem späteren Zeitpunkt. Damit sind natürlich auch alle zugehörigen Merkmale wie Selbstdokumentation für beide Anwendergruppen interessant.

Beispiel einer Differenz:

- Entwicklung von grafischen Benutzungsoberflächen ist eine der wichtigsten Anforderungen an die schnelle Softwareentwicklung. Im schulischen Einsatz spielt dieses Merkmal meist eine untergeordnete Rolle, da typischerweise erst die Grundlagen zum allgemeinen Verständnis im Vordergrund stehen.

Bei der Untersuchung der einzelnen Kriterien wird daher gegebenenfalls auf beide Interessensgruppen eingegangen.

Zusammenfassung:

Tabelle 1 enthält eine kompakte Übersicht über die zuvor aufgestellten Kriterien. Die zweite und dritte Spalte zeigen eine Abschätzung der Bedeutung der einzelnen Kriterien für die beiden überprüften Einsatzgebiete *schnelle Softwareentwicklung* und *informatische Bildung*. Die letzte Spalte verweist auf das oder die Kapitel, in denen das Kriterium behandelt wird.

Kriterien	für schnelle Entwicklung	für informatische Bildung	in Kapitel
Portabilität und Dateizugriff	✓	✓	4.2
Schneller GUI-Entwurf	✓	○	4.3
Geeignete Hilfsmittel und Werkzeuge	✓	✓	4.4
Einfache Syntax und Semantik	✓	✓	4.5
Integrierte Datentypen	✓	✓	4.5.4
Funktionskonzept	✓	✓	4.5.8
Modularisierung und Wiederverwendbarkeit	✓	✓	4.6
Refactoring	✓	✓	4.6
Dokumentation	✓	✓	4.7
Softwaretest	✓	✓	4.8
Fehlertoleranz	✓	✓	4.8
Fehlersuche und Vermeidung	✓	✓	4.9
Klare Fehlermeldungen	✓	✓	4.9
Nebenläufigkeit (Threading)	✓	✓	4.10
Persistente Datenspeicherung	✓	✓	4.10
Erweiterungsfähigkeit und Einbettung	✓	×	4.11
Automatische Speicherverwaltung	✓	✓	5.1
Objektorientierte Entwicklung	✓	✓	5.1.3
Funktionale Entwicklung	○	✓	5.1.2
Prädikative Entwicklung	○	✓	5.1.4
Einfache Lesbarkeit und Erlernbarkeit	○	✓	5.2
Austausch von Algorithmen	○	✓	5.3
Orthogonale Syntax	○	✓	5.3
Eingabehilfen	✓	✓	5.7
Lernumgebung	○	✓	5.6
Motivation	✓	✓	6.1
Leistung (Laufzeiteffizienz)	○	×	6.1

Legende: ✓ ist erforderlich, ○ ist nützlich, × ist unwichtig

Tabelle 1: *Kriterienkatalog*

Die Untersuchung von Python und Werkzeugen erfolgt in Kapitel 4. Die überwiegend fachdidaktischen Aspekte werden in Kapitel 5 überprüft.

Literatur:

- *Lehrbuch der Softwaretechnik* [Balzert00]
- *Objektorientierte Software-Entwicklung mit UML* [KR98]

3.2 Vorstellung der Fallstudien und Erhebungen

3.2.1 Projekte

Im Rahmen dieser Arbeit wurde vom Autor der praktische Einsatz von Python zur schnellen Softwareentwicklung an unterschiedlichen Projekten erprobt. Dadurch sollen unter anderem mögliche Vorzüge und Probleme im realen Einsatz aufgedeckt werden.

Projekt 1: SuM

Aufgabe:

- Implementierung und Test der Ausbildungssoftware *Von Stiften und Mäusen*.

Anforderungen:

- Im Handbuch [CDH99] sind die Spezifikationen in Form der fertigen Projektbeschreibung, dokumentierten Klassenbibliothek und Diagramme gegeben.

Projekt 2: PyNassi

Aufgabe:

- Planung, Implementierung und Test eines grafischen Editors für Struktogramme.

Anforderungen:

- Grafische Benutzungsoberfläche zur Erstellung von Struktogrammen
- Anlehnung an Vorgaben von *Nassi und Shneiderman* sowie DIN 66261
- Laden und Speichern der Struktogramme
- Eingabe von Anweisungen als Python-Quelltext
- Erzeugung von Python-Quelltext aus dem Struktogramm
- Interaktive Ablaufsbeobachtung am Struktogramm

3.2.2 Erhebungen

Es wurden einzelne Umfragen in Newsgroups, Mailinglisten sowie unter Lehrern und kommerziellen Entwicklern durchgeführt. Die Umfragen sind unter anderem wegen der relativ kleinen Teilnehmerzahl (<50) nicht repräsentativ. Vielmehr war Ziel der Umfragen das Einholen allgemeiner Meinungen und Erkennung genereller Tendenzen zu der jeweiligen Problematik.

Umfrage 1:

Ziel:

Ermitteln der subjektiven Lesbarkeit unterschiedlicher Programmiersprachen.

Umfrage 2:

Ziel:

Aufdecken von Probleme beim Einsatz von Python in der Ausbildung.

Vergleichsprojekt:

Problemstellung:

Herunterladen einer festen URL aus dem Internet und Ausgabe über den Standard-Ausgabekanal. Diese einfache Aufgabe soll in unterschiedlichen Programmiersprachen realisiert werden.

Die Erfahrungen mit der Realisierung der Projekte werden in Kapitel 6 zusammengefasst, ebenso werden dort die Ergebnisse der Erhebungen diskutiert.

4 Analyse der Spracheigenschaften von Python

The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.

Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one – and preferably only one – obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Tim Peters, Python-Entwickler

In diesem Kapitel werden die Eigenschaften der Programmiersprache Python ermittelt. Mit den zuvor aufgestellten Kriterien wird überprüft, inwiefern diese Merkmale den Anforderungen an eine Sprache zur schnellen Softwareentwicklung genügen, und welche Probleme möglicherweise vorhanden sind.

4.1 Komplexität im Verhältnis zur Quelltextlänge

Das Verhältnis der Quelltextlänge zu der Komplexität eines Probleme ist ein Maß für die Kompaktheit des Quelltextes. Zur Messung der Quelltextgröße sind unterschiedliche Methoden verbreitet, wobei diese sogenannten Softwaremetriken eine eigene Disziplin der Informatik sind. (siehe [Balzert00]):

Lines of Code (LOC)
Einfaches Zählen der Programmzeilen
Nachteil: Sehr ungenau, da stark abhängig von Notationsgewohnheiten der Entwickler

Function Points
Aufstellen eines Bewertungsmaßstabs für unterschiedliche Anweisungen und Operationen
Nachteil: Finden eines objektiven Maßstabs schwierig, Sprachunabhängigkeit der Metrik

Redundanzfreie Bytes
Zählen aller nicht redundanten Zeichen des Quelltextes (Redundant z. B. Leerzeichen,

Kommentare)
Nachteil: Begünstigt knappe und oft schwerer lesbare Schreibweise

Kompakte Quelltexte sind ein wichtiges Merkmal von High-Level-Sprachen. Eine Analyse für einige Programmiersprachen wurde von [Jo96] durchgeführt. Deutlich ist hier der Unterschied zwischen Sprachen unterschiedlicher Generationen erkennbar: In Sprachen neuerer Generationen ist erheblich weniger Quelltext nötig als in Sprachen der ersten oder zweiten Generation.

Die Größe des Quelltextes hängt in erster Linie von nachstehenden Faktoren ab:

- Hohes Abstraktionsniveau
- Syntax und Semantik der Sprache
- Nutzung des „Baukastenschemas": universelle Module und Bibliotheken

In den folgenden Unterkapiteln werden die Spracheigenschaften von Python untersucht, und es wird gezeigt, inwiefern diese Merkmale auf Python zutreffen. In Kapitel 6 werden Ergebnisse statistischer Analysen zur Komplexität gegebener Problemstellungen zusammengefasst.

4.2 Portabilität

Die Unterstützung möglichst vieler Plattformen ist im heute vielfach anzutreffenden heterogenen EDV-Umfeld beziehungsweise im Kontext vernetzter, interaktiver Informatiksysteme verschiedenster Ausprägungen (unterschiedliche Betriebssysteme, Prozessoren, Einsatzbreite) eine wichtige Eigenschaft einer Programmiersprache, um einen möglichst breiten und systemunabhängigen Einsatz zu erlauben. Python ist für eine sehr große Anzahl von Plattformen und Betriebssystemen verfügbar, teilweise in Varianten kombiniert mit Entwicklungsumgebungen. Nahezu alle Versionen sind als Quelltext erhältlich, ebenso sind für viele Systeme bereits fertig kompilierte Versionen (Binaries) bereitgestellt. Zum Teil verfügen diese Versionen über Installationsprogramme oder werden über die Standardmechanismen der jeweiligen Plattform installiert (RPM, MSI). Im Anhang A.2 befindet sich eine Übersicht der unterstützten Plattformen.

In der Python-Distribution ist bereits eine umfangreiche Bibliothek in Form von Python-Modulen enthalten. Die meisten sind universell auf allen Plattformen nutzbar, so dass die erstellten Skripte beziehungsweise Programme ohne Anpassungen verwendet werden können. Des weiteren sind zahlreiche Bibliotheken für eine Vielzahl von Aufgaben auf allen wichtigen Plattformen erhältlich. Das Spektrum reicht von Bibliotheken zur GUI-Gestaltung über Multimedia und Spielentwicklung bis hin zu verteilten Systemen. Dabei werden Standards bzgl. gängiger Dateiformate, Datenbanken, Netzwerkprotokolle, Logdateien und anderer Themen unterstützt. Eine exemplarische Besprechung einzelner Module ist in Abschnitt 4.10 zu finden. Umfangreiche Bibliotheken sind auf [Parn02] und [Python02] erhältlich.

Internationalisierung

In klassischen Sprachen belegt bei Strings jedes Zeichen nur 1 Byte, daher können nur maximal 256 Zeichen unterschieden werden. Für die Verarbeitung von internationalen Texten ist das

nicht ausreichend. Es gibt Zeichensätze wie die fernöstlicher Sprachen, die weit mehr Zeichen enthalten. Für dieses Problem gibt es eine Reihe von Lösungen; eine davon ist die Definition des sogenannten Unicode-Zeichensatzes [UniCon00], der alle Sprachen umfasst. Zeichenketten in Python können in Unicode-Notation erstellt werden. Dadurch können auch internationale Texte mit Python verarbeitet werden.

Die Erstellung mehrsprachiger Software ist auf mehreren Wegen möglich. Erste Ansätze wurden von [Lowis97] vorgestellt. Mit dem seit Python 2 verfügbaren Modul *gettext* lässt sich bereits in der Implementierungsphase mehrsprachige Softwareentwicklung vorbereiten. Hierzu müssen übersetzbare Textstellen mit einer Übersetzungsfunktion gekennzeichnet werden und das Modul *gettext* importiert werden. Mit den kostenlosen Programmen der auch für andere Sprachen nutzbaren GNU-gettext-Bibliothek wird aus den Quelltexten eine Wortliste extrahiert. Der Übersetzer kann nun ohne Änderungen am Quelltext die Übersetzung vornehmen, wobei es zahlreiche Programme zur Bearbeitung der Wortlisten gibt. Auch spätere Änderungen der Quelltexte sind möglich, denn aktualisierte Programmteile können in alte Wortlisten integriert werden. Dieses Verfahren wurde bei der Fallstudie *PyNassi* eingesetzt (siehe Abschnitt 6.3).

Nachträgliche Übersetzung bestehender Software ist beispielsweise durch Zusatzprogramme erreichbar, die direkt in den Quelltexten mit Wortlisten Ersetzungen vornehmen. Mit der Internationalisierung der Python-Software befasst sich unter anderem die *Internationalization-SIG (i18n)* [PSIG02]. Werkzeuge für eine vollständig mehrsprachige Entwicklung (Kommentare, Dokumentation, Bezeichner usw.) sind allerdings bisher nicht verfügbar.

Dateisysteme

Unterschiedliche Dateisysteme bereiten vielen systemnahen Programmiersprachen wie beispielsweise C++ Probleme. Insbesondere die Trennung von Unterverzeichnissen wahlweise durch "/" (UNIX), "\" (Windows) oder ":" (Mac OS X) bereiten einen Mehraufwand bei der Portierung. In Python wurde dieses Problem durch die Konvention gelöst, Verzeichnisse im Quelltext immer mit "/" zu trennen. Abhängig von der Systemumgebung ersetzt der Interpreter gegebenenfalls durch "\" oder ":"

Probleme bereiten derzeit noch unterschiedliche Zeilenende-Zeichen bei Übernahme von Dateien anderer Plattformen. In der kommenden Python-Version 2.3 sollen die unterschiedlichen Varianten erkannt und korrekt interpretiert werden.

4.3 Schneller GUI-Entwurf

Der schnelle Entwurf grafischer Benutzungsoberflächen (GUI) ist eines der wichtigsten Einsatzgebiete des Rapid Prototyping. Hat der Auftraggeber noch keine genaue Vorstellung oder nur ungenaue Angaben für einen Produktentwurf, bietet der Prototyp eine geeignete Diskussionsgrundlage. Änderungen müssen schnell möglich sein, und auch ein vollständiger Neuentwurf darf nicht viele Ressourcen in Anspruch nehmen. Nachfolgend werden Bibliotheken und Werkzeuge vorgestellt, die den Python-Entwickler bei dieser Aufgabe unterstützen, und deren Besonderheiten erläutert.

Im heutigen heterogenen Systemumfeld trifft der Entwickler auf unterschiedlichste Betriebssysteme mit grafischer Benutzerführung: Windows, X, MacOS und andere. Durch Einsatz

geeigneter GUI-Bibliotheken kann der Entwickler in High-Level-Sprachen unabhängig von der Plattform grafische Entwürfe erstellen. Kaum ein kommerzielles Programm kommt heute ohne grafische Benutzungsoberfläche aus. Python bietet dem Entwickler viele Möglichkeiten, um ein Programm mit einer grafischen Benutzungsoberfläche auszustatten. Im Folgenden werden bekannte Bibliotheken erläutert.

Tkinter ist eine Wrapper-Bibliothek, die auf dem Softwarepaket *Tcl/Tk* beruht. *Tcl* ist selbst eine Skriptsprache, *Tk* ein speziell für *Tcl* entwickeltes sogenanntes *Widget Set*, das zunächst für X11 verfügbar war. Der Python-Entwickler bekommt davon allerdings nichts zu sehen, da alle Tcl-Befehle von der Bibliothek erzeugt werden und an einen eingebetteten Tcl-Interpreter übergeben werden. Tcl/Tk ist sehr weit verbreitet und für viele Plattformen wie X, Windows und Macintosh verfügbar. *Tkinter* bietet eine gute Auswahl an Widgets inklusiv Geometrieverwaltern zum automatischen Plazieren der Widgets. *Tkinter* ist bereits in Pythons Basisbibliothek enthalten, daher entfallen lästige Installationsprozeduren. Ebenso ist bereits die Dokumentation in der Python-Anleitung integriert, so kommt der Entwickler für Python und Tk mit der internen Dokumentation aus. In beinahe jedem Python-Buch wird auf die Entwicklung von Anwendungen mit Tk eingegangen. Nur bzgl. Performance und Funktionalität muss der Entwickler Abstriche in Kauf nehmen: Andere Bibliotheken arbeiten schneller, da hier der Umweg über Tcl entfällt. Darüber hinaus bieten sie mehr Funktionen und sehen etwas moderner aus. Ein weiterer Kritikpunkt ist eine nicht saubere Objektorientierung, da oft freie Funktionen zum Einsatz kommen.

PyQt ist eine Python-Bibliothek zur Nutzung der *Qt*-Anwendungsschnittstelle, auf der auch der KDE-Desktop basiert. *Qt* ist das einzige kommerzielle Produkt zum Entwurf von grafischen Benutzungsoberflächen; für privaten Einsatz ist jedoch auch eine kostenlose Version mit leicht eingeschränktem Funktionsumfang und auf nichtkommerzielle Weitergabe beschränkter Lizenz erhältlich. Es bietet dafür weit mehr als reinen GUI-Entwurf, beispielsweise systemunabhängige Schnittstellen zu Datenbanken, zahlreiche Netzwerkfunktionen, Dateibearbeitung, Threading usw. *Qt* ist in Versionen für Windows, UNIX/X11 sowie Mac OS X erhältlich, wobei die Mac-Version zusammen mit *PyQt* schwierig zu installieren ist und derzeit noch Probleme bei Nutzung bestimmter Widgets hat. Diese Probleme sollen in Kürze behoben werden. *PyQt* überzeugt durch eine gut dokumentierte, objektorientierte Bibliothek. Ähnlich wie *PyGTK* nutzt es das Konzept der *Signals and Slots*, um auf Anwendungsereignisse zu reagieren.

Eine weitere Besonderheit von *Qt* ist die Eigenschaft, das typische Aussehen anderer Oberflächen simulieren zu können. So kann beispielsweise eine Anwendung unter Windows aussehen, als wäre sie mit *Motif* gestartet worden, oder umgekehrt. Auch völlig unabhängige sogenannte Styles sind wählbar.

WxPython ist eine Schnittstelle zur Nutzung der in C++ geschriebenen Bibliothek *WxWindows*. *WxWindows* wurde auf alle gängigen UNIX-Varianten mit *GTK* oder *Motif*, sowie für Windows, OS/2 und den Mac portiert. Die Schnittstellen von *WxPython* sind nahezu eine 1:1-Umsetzung der C++-Version, so dass sich Dokumentation und Nutzung in beiden Sprachen sehr ähneln. Die Erzeugung einzelner Widgets erfordert eine verhältnismäßig große Anzahl an teilweise optionalen Parametern; hier kann der Python-Entwickler von den einfachen Möglichkeiten der Parameterübergabe an Funktionen profitieren.

WxPython arbeitet durch systemnahe Nutzung von C++-Bibliotheken sehr schnell und fügt sich auch optisch gut in die jeweils genutzte Umgebung ein. Somit ist es eine interessante Alternative für Entwickler, die das etwas träge und altmodisch aussehende *TkInter* meiden

wollen.

PyGTK basiert auf *GTK+*. Es ist für UNIX entwickelt und später nach Windows portiert worden. *PyGTK* ist eine objektorientierte Schnittstelle zu *GTK+*. Benutzereingaben werden der Anwendung in Form von Ereignissen und Signalen mitgeteilt, das heißt, ein Widget kann bei Eintreten einer bestimmten Benutzeraktion (Signal) eine Funktion ausführen. Obwohl *GTK+* in C entwickelt wurde, stellt *PyGTK* eine durchdachte objektorientierte Schnittstelle für den Python-Entwickler dar. Angeboten werden zahlreiche Widgets inklusive Geometrieverwaltung.

Während alle vorgestellten Bibliotheken auf mehreren Plattformen laufen, ist *Pythonwin* ein reines Windows-Programm. Es basiert auf den *MFC* (Microsoft Foundation Classes) aus Visual C++, wobei ein großer Teil der C++-Funktionen auf Python abgebildet wird. Kenntnisse der MFC sind nötig, um grafische Anwendungen zu entwickeln. Selbst die Realisierung einfacher Benutzungsoberflächen ist deutlich aufwendiger als bei allen anderen Bibliotheken. Dafür enthält *Pythonwin* eine vollständige Entwicklungsumgebung mit guter Dokumentation der Bibliotheken. Die Möglichkeiten der MFC gehen weit über die reine Oberflächen-Entwicklung hinaus. Zahlreiche Möglichkeiten zur Nutzung von nahezu allen Windows-Diensten der MFC sind vorhanden.

Jython nimmt eine Sonderstellung ein: *Jython* ist ein vollständig in Java implementierter Python-Interpreter. Auf die Besonderheiten wird in Abschnitt 4.11.1 noch genauer eingegangen. Der Entwickler hat vollständigen Zugriff auf Java-Swing und AWT. *Jython* ist dadurch auf nahezu allen Java-tauglichen Plattformen lauffähig und kann mit den Java-Funktionen kombiniert werden. Problematisch ist die Leistung, da *Jython* um ein Vielfaches langsamer läuft als Python. Technisch bleibt der Entwickler auf die Fähigkeiten der virtuellen Java-Maschine beschränkt. Er muss also Abstriche bei Multimedia-Anwendungen machen, bekommt dafür aber die Vorzüge beispielsweise für Internet-Anwendungen.

AnyGUI nimmt ebenfalls eine Sonderstellung ein: Es enthält kein eigenes GUI, sondern ist eine Oberklasse für alle (!) zuvor vorgestellten Bibliotheken. Aus Sicht des Entwicklers existiert nur die *AnyGUI*-Bibliothek; diese erkennt dann selbständig, welche weiteren Bibliotheken installiert sind, und nutzt die jeweils optimale Variante. *AnyGUI* bietet Fenster, Menüzeilen sowie Dialoge mit diversen Buttons, Eingabefeldern und Auswahlmöglichkeiten. Der Funktionsumfang ist eine Schnittmenge der unterstützten Bibliotheken und kleiner als bei den einzelnen Bibliotheken, dafür aber universell auf allen Plattformen nutzbar. Ist keine Bibliothek installiert, werden sogar Dialoge in reiner Textform erzeugt. Der Leistungsverlust gegenüber direkter Nutzung der originalen Bibliotheken ist minimal.

PyUI ist ein GUI-Aufsatz für die Spiele- und Multimediabibliothek *Pygame*. Fenster und Dialoge werden von eigenen Darstellungsfunktionen erzeugt und wirken durch ein ungewöhnliches Dialogdesign sehr verspielt. Alle Dialoge und Fenster werden als Unterfenster des eigentlichen Anwendungsfensters erzeugt, welches zuvor auf eine feste Größe initialisiert werden muss. *PyUI* bietet aber alle üblichen Funktionen für Fenster und Dialoge und harmoniert mit den Grafik-Funktionen der Pygame-Bibliothek. *PyUI* ist objektorientiert; die Benutzereingaben werden dabei über Ereignisse (Callbacks) der Anwendung gemeldet. *PyUI* ist zwar relativ portabel, es hat aber durch sein ungewöhnliches Dialogdesign und das starre Hauptfenster bislang nur unter den Grafik- und Spielentwicklern Freunde gefunden.

Curses und *Textui* bieten einfache Dialogfunktionen auf einer reinen Textebene, wobei alle typischen GUI-Elemente wie Menüs, Eingabe- und Auswahldialoge vorhanden sind. *Pythondialog* basiert auf dem UNIX-Programm *Dialog* und ist ebenso textorientiert. Die Dialogfunktionen

sind hier sehr einfach, aber auch sehr leicht – meist mit nur wenigen (1 ... 3) Programmzeilen – zu verwenden. Es bietet sich zum Einsatz in Installations- und Konfigurationsskripten an.

Eine Übersicht der Bibliotheken zeigt Tabelle 2.

Bibliothek	**Objektorientierung**	**Portabilität**	**Leistung**	**Funktionen**
Tkinter	+	++	-	-
PyQt	++	+	++	++
WxPython	++	++	++	+
AnyGUI	++	++	+	-
PyGTK	+	-	+	+
PythonWIN	+	-	++	++
JavaSwing + Jython	+	++	-	++
PyUI	+	+	+	-
Curses / Textui	Nicht getestet	++	Nicht getestet	Nicht getestet
PythonDialog	+	-	-	-

Tabelle 2: *Vergleich GUI-Bibliotheken*

Werkzeuge für den GUI-Entwurf

Zur Erstellung komplexerer Benutzungsoberflächen gibt es für High-Level-Sprachen Werkzeuge, welche einfaches grafisches Gestalten ermöglichen. Per Mausklick können Rahmen, Schaltflächen, Eingabemasken und Auswahlfelder zu Dialogen kombiniert werden. Aus den erstellten Dialogen wird vom Werkzeug Quelltext erstellt.

Moderne Werkzeuge bieten neben dem reinen Dialogentwurf mehr oder weniger umfangreiche integrierte Entwicklungsumgebungen, welche meist Funktionen zur Quelltexteingabe und zum Debugging bieten. Durch Quelltexteingabe kann einzelnen Dialogelementen direkt im Editor ihre Funktionalität zugewiesen werden.

Ein einzelner Entwickler kann für kleine Anwendungen mit geeigneten Werkzeugen binnen weniger Stunden einen Prototyp einer grafischen Benutzerführung und Grundfunktionalität entwerfen. Für den Python-Entwickler stehen einige Werkzeuge zum GUI-Entwurf bereit:

- BlackAdder / Qt Designer
- Boa Constructor
- PythonWorks-Pro
- Microsoft Visual.Net Studio

Nachfolgend werden ausgewählte Python-Werkzeuge vorgestellt, welche auf unterschiedlichen Bibliotheken aufsetzen. Exemplarisch wurde mit allen Werkzeugen das Pizza-Beispiel (siehe Anhang und [BS97]) erstellt.

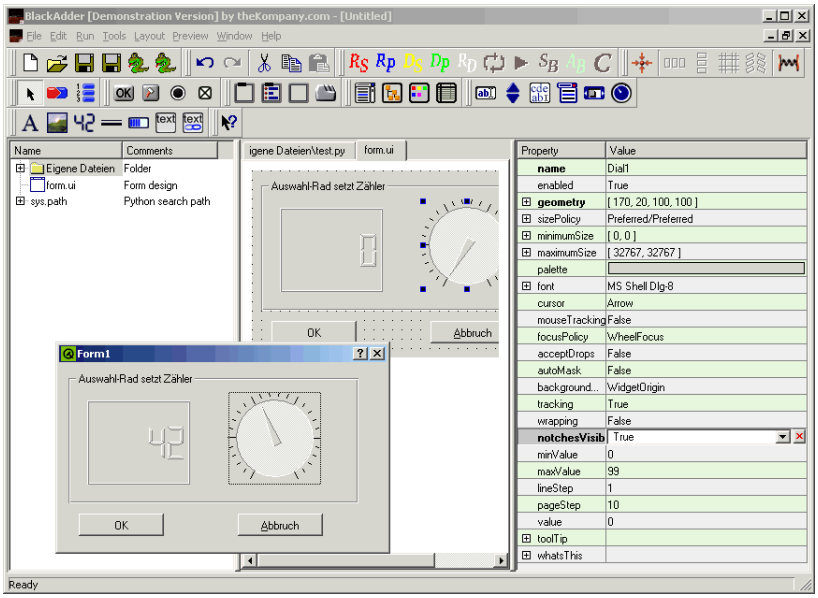


Abbildung 6: *BlackAdder*

BlackAdder

Mit *BlackAdder* [TheKompany] steht dem Python-Anwendungsprogrammierer eine Entwicklungsumgebung zum Erstellen von GUI-Anwendungen unter Qt beziehungsweise KDE zur Verfügung. *BlackAdder* wird kommerziell vertrieben und ist in Versionen für Windows und Linux erhältlich. *BlackAdder* ist eine visuelle Entwicklungsumgebung, bestehend aus Editor, Dialog-Designer, Python-Interpreter und Debugger. Des Weiteren enthält es eine umfangreiche Dokumentation zu Python und Qt. Funktionen zur Modellierung oder Dokumentation sind nicht integriert.

BlackAdder eignet sich zum schnellen Entwurf von Anwendungen mit grafischer Benutzerführung. Der Forumulardesigner macht einen ausgereiften Eindruck und ist leicht und schnell zu bedienen. *BlackAdder* unterstützt das Prinzip *Signals and Slots* und kann bereits beim Dialogentwurf auf bestimmte Ereignisse reagieren. Beispielsweise kann ein Slider (Schieberegler) direkt an ein Ziffern-Anzeigefeld gekoppelt werden. Auf diese Weise erstellte Dialoge können direkt zum Testen ausgeführt werden, ohne eine Zeile Quelltext schreiben zu müssen. Fertig entworfene Dialoge werden dann gespeichert, im Python-Programm importiert und dort mittels Subclassing um ihre Funktionalität erweitert. Entwickler, die auf den internen Python-Editor und Debugger verzichten können, bekommen mit dem *Qt Designer* eine vergleichbare, aber kostenlose Alternative im Rahmen der nichtkommerziellen Qt-Lizenz.

Boa Constructor

Der *Boa Constructor* [Booysen02] - kurz *Boa* - ist eine grafische, integrierte Entwicklungsumgebung ausschließlich für Python. Es läßt sich in die Kategorie der Visuellen Entwicklungspakete einordnen, ähnlich wie Microsofts *Visual Basic* oder Borlands *Delphi*. Die Entwicklung einer Anwendung besteht hier typischerweise aus zwei wesentlichen Schritten:

Zuerst erfolgt der Entwurf der Benutzungsoberfläche. Mit wenigen Mausklicks lassen sich einzelne Widgets (Texte, Schaltflächen, Menüs, Fenster usw.) zu Dialogen und schließlich der

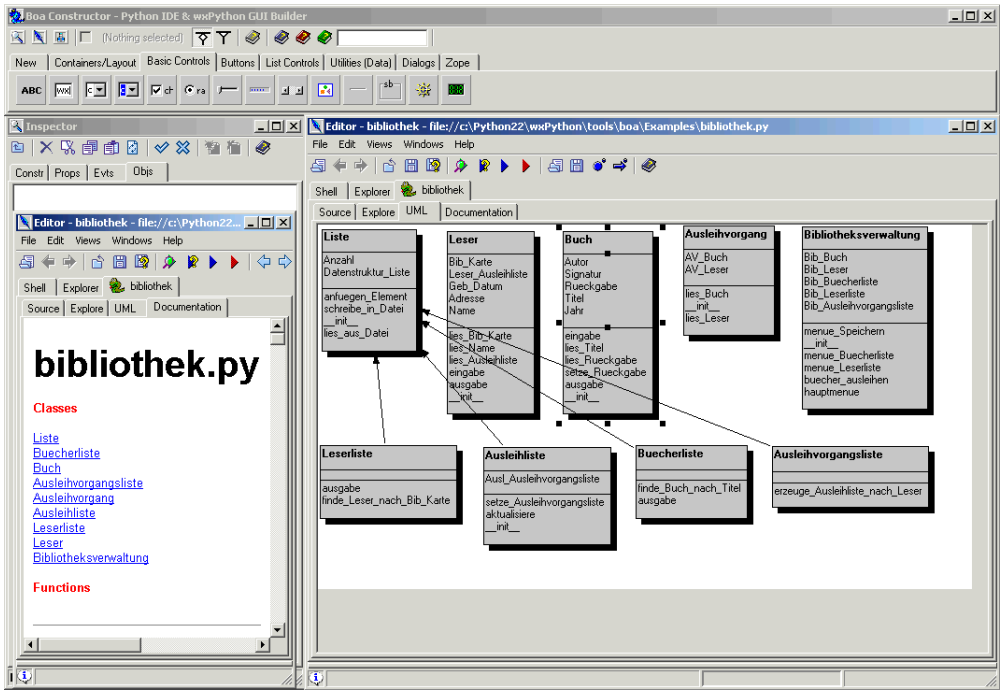


Abbildung 7: *Boa Constructor*

gesamten Benutzungsoberfläche kombinieren. Während des Entwurfs wird automatisch Quelltext zur Darstellung der Benutzungsoberfläche erzeugt. Im zweiten Schritt wird den Widgets dann nach und nach die Funktion zugewiesen.

Boa steht unter der *GNU General Public License* und ist im Internet kostenlos erhältlich. Es verwendet das GUI-Toolkit *WxPython* [Dunn02] und ist daher auf den Plattformen Windows, UNIX mit GTK/Motif/Lesstif sowie Macintosh nutzbar.

Boa unterstützt den Entwickler durch zahlreiche Funktionen während des gesamten Entwicklungsvorgangs. Nach Erstellen eines neuen Projekts lassen sich mit wenigen Schritten Dialoge erstellen. Änderungen an Dialogen werden sofort in den Programmcode übernommen. Jeder Dialog wird als eigenes Python-Modul realisiert. Einzelne Module können direkt in unterschiedlichen Ansichten dargestellt werden. Die Hierarchieansicht zeigt in einer übersichtlichen Baumdarstellung Klassen und Methoden. Beziehungen zwischen den Klassen werden in der UML-Darstellung veranschaulicht (siehe obige Abbildung). Jederzeit hat man Einblick in die Quelltext-Dokumentation. *Boa* übernimmt hierzu automatisch die sogenannten *Doc-Strings* aus dem Quelltext. Einzelne Module lassen sich mit einer Aufgabenliste (ToDo List) versehen, um schnell einen Einblick in den Stand eines Moduls zu bekommen. Die Ausführung der erstellten Programme ist direkt in *Boa* möglich. Dabei unterstützt ein Debugger die Fehlersuche.

Boa bietet sich für die Entwicklung von Anwendungen mit grafischer Benutzerführung durch einen einzelnen Entwickler oder ein kleines Team an. Die Planungs- und Modellierungsphase wird mit dem aktuellen Stand vernachlässigt, so ist der Entwickler in dieser Phase auf andere Werkzeuge angewiesen. Dafür erhält er allerdings ein starkes Werkzeug für die Implementierung und Dokumentation.

Boa befindet sich derzeit noch in der Entwicklung; die vorliegende Version 0.1 ist im Alpha-Stadium. Sie überzeugt bereits jetzt durch leichte Bedienung und ein durchdachtes Konzept, Mängel liegen im Detail. So fehlt beispielsweise eine Funktion zum UML-Export, und der

integrierte Debugger stürzte im Test vereinzelt ab. Ansonsten arbeitet *Boa* gut und ist mit den eingesetzten Test-Quelltexten hervorragend nutzbar. Wenn mit Rapid Prototyping das Ziel einer früh dem Kunden vorzeigbaren Benutzerführung verfolgt wird, ist *Boa* mit Python ein geeignetes Hilfsmittel.

PythonWorks-Pro

PythonWorks-Pro von der Firma *PythonWare* bietet dem Entwickler einen TkInter-Formulardesigner, einen modernen Editor sowie einen Debugger. Kommerzielle Versionen sind für Linux, Windows und Solaris verfügbar. Die Benutzerführung weicht sehr von bekannten Konzepten ab: Der auffällig farbenfrohe Arbeitsbereich ist aufgeteilt in Projektübersicht, Formulardesigner und Codeeingabe, wobei häufig zwischen unterschiedlichen Ebenen gewechselt werden muss. Nach kurzer Eingewöhnung lässt sich aber gut damit arbeiten.

Im Formulardesigner lassen sich per Drag & Drop[7] Dialoge gestalten, wobei direkt der zugehörige Python-Code und eine XML-Darstellung erzeugt werden. Einzelne Dialoge können direkt nach ihrem Entwurf durch Ausführung des so erstellten Quelltextes getestet werden. Die Umsetzung des Pizza-Beispieles [BS97] ist innerhalb weniger Minuten möglich.

Ergebnis:

Der Python-Entwickler hat die Qual der Wahl: Alle Bibliotheken haben ihre Stärken und Schwächen. *TkInter*, *WxPython* und *Jython* bieten besonders portable Anwendungsentwicklung und sind kostenlos. *PyQt* bietet neben GUI-Entwurf zahlreiche Zusatzfunktionen, für kommerziellen Einsatz muss allerdings eine Lizenz erworben werden. *PythonWin* ist die einzige Bibliothek, die systemnahe Dienste und Anwendungen unter Windows gestattet, bietet aber keinerlei Kompatibilität zu anderen Plattformen.

Die Nutzung der Bibliotheken erscheint einfacher als die äquivalenten Versionen von Perl und C++, die flexible Argumentübergabe an die Methoden ist hier ein großer Vorteil von Python. Ein `Hello-World`-Dialog ist in allen Bibliotheken mit 10–20 Anweisungen realisierbar. Einige Beispiele werden in [LF02] gezeigt, weitere liegen auch den Bibliotheken bei. Zum direkten Vergleich mit anderen Sprachen wie C++ kann beispielsweise die entsprechende Version von *Qt* oder *WxWindows* herangezogen werden, schon in der Dokumentation fallen hier die Unterschiede der Parameterübergabe auf.

Für Python stehen geeignete Werkzeuge zur Erstellung grafischer Benutzungsoberflächen unter den unterschiedlichen Bibliotheken[8] zur Verfügung, die verschiedenen Anforderungen gerecht werden. Der Funktionsumfang im Bereich GUI-Design ist bei allen Werkzeugen vergleichbar, so dass Zielplattformen und Funktionsumfang der Bibliotheken das primäre Entscheidungskriterium bei der Auswahl sind. Eine Übersicht zu den Funktionen der Werkzeuge zeigt Abb. 15.

[7] Drag & Drop bezeichnet das Verschieben von Objekten mit einem Zeigegerät wie einer Maus auf der Arbeitsumgebung

[8] Dies gilt auch für viele andere High-Level-Sprachen.

4.4 Werkzeuge zur Quelltexterzeugung

Bereits während der Planung und des Entwurfs eines Produktes ist die Verfügbarkeit von unterstützenden Hilfsmitteln ein wichtiger Aspekt. Sobald ein Programm mehr leisten soll als nur wenige Anweisungen auszuführen, nimmt die Modellierung der Problemstellung einen großen Teil der Entwicklungsarbeit ein. Modellierungsorientierte Entwicklung hilft, Anwendungen oder Probleme in dem Zustand darzustellen, in dem sie sich derzeit befinden oder in Zukunft befinden sollen. Ein gutes Modell kann die Kommunikation innerhalb des Programmierteams verbessern, erlaubt die Wiederverwendbarkeit von Code und führt zu einem wirtschaftlichen Entwicklungsprozess.

Ein zentraler Begriff im Themenbereich der Modellierung ist die *Unified Modeling Language*, kurz UML. Sie vereint programmiersprachenunabhängig grafische Darstellungsmöglichkeiten sowohl statischer wie auch dynamischer Strukturen.

Im Einzelnen:

- Konzepte zur Datenmodellierung (Entity Relationship Diagramme)
- Modellierung von Abläufen (Business Modeling, Work Flow)
- Objekt-Modellierung
- Modellierung von Komponenten

Die Sprache kann während des gesamten Softwareentwicklungszyklus über verschiedene Implementierungstechnologien hinweg eingesetzt werden. UML hat sich als Standardsprache zum Visualisieren, Spezifizieren, Konstruieren und Dokumentieren im Bereich der Softwareentwicklung durchgesetzt, auch wenn es Kritik bezüglich der Konsistenz der Sprache gibt.

Nachfolgend werden einige Modellierungswerkzeuge vorgestellt, welche den Entwickler in der Modellierungsphase unterstützen und zusätzliche Hilfsmittel für die spätere Realisierung mit Python bieten. Eine Übersicht weiterer Werkzeuge und ihrer Möglichkeiten ist im Anhang 5.8 aufgeführt. Die genannten GUI-Toolkits wie *Qt*, *Tk* und *WxPython* wurden bereits in Kapitel 4.3 vorgestellt.

Diagrammeditor Dia

Dia [Dia02] ist ein auf GTK+ [GTK02] basierendes Diagramm-Entwurfswerkzeug, freigegeben unter der GPL (GNU General Public License). Neben dem Quelltext sind Binaries für Linux und Windows kostenlos verfügbar. *Dia* eignet sich zum Entwurf vieler Arten von Diagrammen, insbesondere aber für Entity-Relationship-Diagramme, Klassendiagramme, Flussdiagramme und Netzwerkdiagramme. Die erstellten Diagramme können unter anderen im XML-Format gespeichert werden. Dies ermöglicht die Interoperabilität mit anderen Werkzeugen.

Zur Konstruktion von UML-Diagrammen und zur späteren Weiternutzung des Entwurfs in Programmiersprachen hat sich *Dia* bewährt. Die Gestaltung der Diagramme erfolgt auf einer grafischen Arbeitsfläche, per Drag & Drop lassen sich Klassendefinitionen, Beziehungen zwischen Klassen sowie optische Gestaltungsmittel kombinieren. Neben Darstellungsmerkmalen wie beispielsweise der Farbe werden den Klassendefinitionen Attribute und Methoden mit

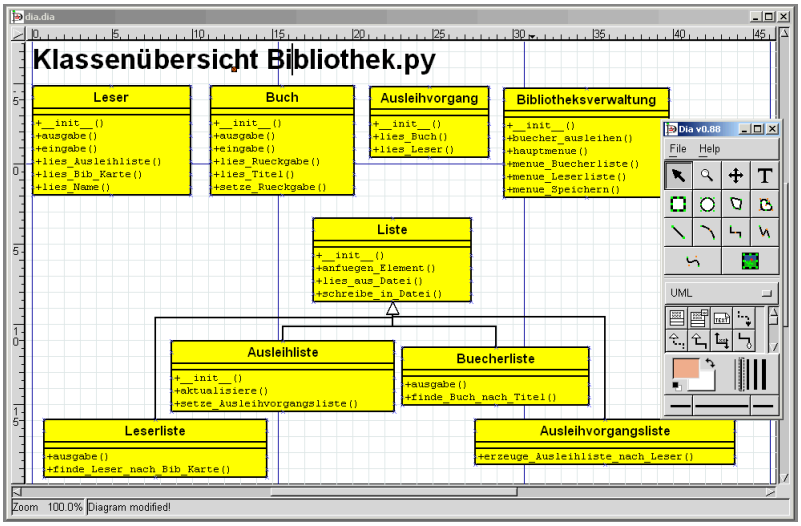


Abbildung 8: *Dia*

ihren Parametern hinzugefügt. *Dia* ist ein universelles Werkzeug und nicht an die Nutzung bestimmter Programmiersprachen gebunden. Daher bietet es auch Gestaltungsmöglichkeiten an, die, abhängig von der Programmiersprache, später möglicherweise nicht genutzt werden können. Ebenso lassen sich Elemente verschiedenartiger Diagramme zu sinnlosen Entwürfen kombinieren, denn eine automatische Kontrolle erfolgt in der Entwurfsphase nicht.

Der Funktionsumfang von *Dia* beschränkt sich hauptsächlich auf den Entwurf und die Gestaltung von Diagrammen. Die Funktionalität lässt sich allerdings durch Nutzung von Zusatzprogrammen erweitern: Es existieren bereits einige Werkzeuge zur Konvertierung der Diagramme in nutzbaren Code und umgekehrt. Neben SQL-Datenbanken lassen sich Coderahmen für gängige Programmiersprachen erstellen. *Dia2Code* [Ohara02] erzeugt aus mit *Dia* erstellten UML-Klassendiagrammen ein Programmgerüst in einer wählbaren Sprache; neben C++ und Java wird auch Python unterstützt.

Ebenso kann mit *Happydoc* [Hellmann02] aus vorhandenem Code eine Klassenübersicht als UML-Diagramm im *Dia*-Format gewonnen werden. Details zu *Happydoc* werden im Abschnitt zur Dokumentation (4.7) noch vorgestellt.

Wenn *Dia* zusammen mit diesen Zusatzprogrammen eingesetzt wird, ist es ein gutes Werkzeug für kleine Projekte und bietet sich auch zur Nutzung im Informatikunterricht an. Es ließ sich problemlos mit dem während dieser Arbeit angefallenen Code nutzen und erscheint zum Entwurf und zur Untersuchung von kleineren Projekten mit Python geeignet. Möglichkeiten zum Mehrbenutzerbetrieb und zur Versionsverwaltung sind nicht vorgesehen, wodurch es für große Entwicklungsteams nur eingeschränkt verwendbar ist.

ObjectDomain-R3

ObjectDomain-R3 [OD02] ist ein kommerzielles Modellierungswerkzeug zur professionellen Softwareentwicklung. Es dient ähnlich wie *Dia* zur visuellen Gestaltung aller Abläufe bei der Softwareentwicklung, insbesondere beim objektorientierten Modellieren.

Im Vergleich zu *Dia* bietet es weit mehr Möglichkeiten; als besondere Merkmale seien hier Mehrbenutzer-Fähigkeit, umfangreiche Dokumentationsmöglichkeiten und vollautomatisches

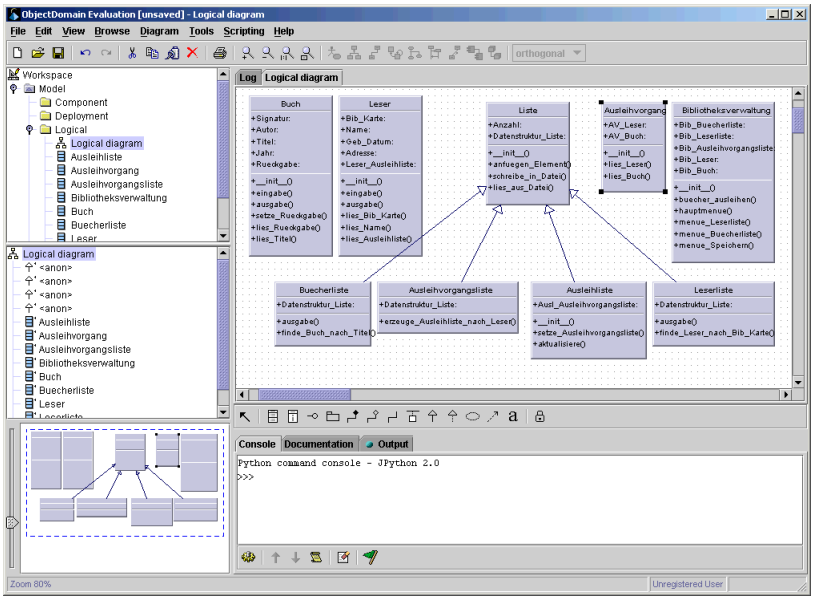


Abbildung 9: *ObjectDomain-R3*

Diagrammlayout genannt. Funktionalität zur Codeerzeugung und Diagrammerstellung aus vorhandenem Code sind bereits für Python und andere Sprachen wie Java, C++, IDL integriert.

Die Installation unter Windows wie auch unter Linux gestaltet sich dank Installationsprogramm sehr einfach. *ObjectDomain-R3* basiert auf Java und ist daher auf vielen Plattformen nutzbar. Ein vollständiger Pythoninterpreter ist mittels *Jython* integriert. Über eine Kommando-Konsole ist sowohl die Manipulation der erstellen Modelle mittels Skripten möglich wie auch der Test von Quelltexten. Durch Versionsverwaltung und Mehrbenutzeroption bietet sich auch ein Einsatz bei größeren Projekten an.

Die Einarbeitung in die Benutzung erfolgte ohne Probleme und ist relativ zügig möglich. Die Tests mit den Beispielprogrammen verliefen bei *ObjectDomain-R3* ohne Beanstandung.

Ergebnis:

Die vorhandenen Werkzeuge für Python erlauben es, Modelle zu erstellen und aus diesen Modellen Quelltext zu erzeugen. Unterschiede liegen in der Ausrichtung auf unterschiedliche Zielgruppen.

Dia bietet unter einer schnell erlernbaren Benuzungsoberfläche alle wichtigen Funktionen zur computergestützten Modellierung und späteren Nutzung des Modells in der Implementierungsphase. Es bietet sich damit zum Einsatz in der Ausbildung bzw. Informatikunterricht und zur Nutzung in kleinen Projekten an. Dagegen ist *ObjectDomain-R3* auf den professionellen Entwickler ausgerichtet. Zur Einarbeitung in die Nutzung sind einige Arbeitsstunden erforderlich, dafür erhält der Entwickler aber ein Werkzeug mit vielen Funktionen: eine skriptbare, integrierte Entwicklungsumgebung, integrierter Interpreter und viel mehr. Durch die Menge an Funktionen ist es für Neulinge oder Ausbildungszwecke jedoch nicht geeignet. Alle Programme sind derzeit nur in englischer Sprache verfügbar, dies sollte bei der Nutzung im Informatikunterricht berücksichtigt werden.

Literatur:

- *Riverbank Computing: Python Qt* [Riverbank02]
- *Programming with Qt* [Dal02]
- *GUI-Programming with Python / Qt Edition* [Rempt01]
- *WxPython* [Dunn02]
- *Python Programming on Win32* [Hammond00]
- *Jython Essentials* [Pedroni02]
- *Python and Tkinter* [Grayson00]
- *Anygui* [AnyGUI02]
- *Jython Home Page* [Jython02]

4.5 Syntax und Semantik

Nachfolgend werden Syntax und Semantik von Python untersucht und Besonderheiten im Vergleich zu anderen Programmiersprachen gesucht. Insbesondere soll geprüft werden, welche Möglichkeiten für die Erstellung kompakten aber verständlichen Quelltextes geboten werden. Zum Vergleich werden wegen ihrer Bekanntheit[9] häufig die Sprachen C++, Java, Perl, sowie Pascal-Dialekte und funktionale Sprachen wie SML herangezogen.

4.5.1 Blockbildung

In den meisten Sprachen erfolgt die Bildung von Anweisungsblöcken durch besondere Strukturelemente, beispielsweise `{}` in Perl, C++ und Java, `BEGIN .... END` in Pascal. Dagegen werden in Python Blöcke nur durch ihre Einrückung gekennzeichnet. Alle gleich weit eingerückten Anweisungen gehören zu einem Block, das Blockende wird durch Ausrückung gekennzeichnet. Daher sind auch keine Blockende-Markierungen wie `ENDIF, ENDFOR, }` nötig. Des Weiteren enthält jede Zeile genau eine Anweisung, so dass auch keine Anweisungstrennzeichen wie Semikola nötig sind.

```
if i==1:
  anweisung1()
  anweisung2()
else:
  anweisung3()
  anweisung4()
```

Dadurch wird der Programmierer zu einem sauberen Quelltextlayout gezwungen und die Lesbarkeit des Quelltextes erhöht. (Motto: WYSIHIIP = What You See Is How It Is Parsed)

Beispiel:

Betrachtet man nur kurz das folgende Programm in C und Python, wird durch die Formatierung suggeriert, beide Programme würden zehnmal die Funktion `Ausgabe` aufrufen.

```
/* C */
for(i=0,i<10,i++);
  Ausgabe(i);

# Python
for i in range(10):
  Ausgabe(i)
```

[9] bezogen auf ihre aktuelle Verbreitung im Informatikunterricht und in der Hochschule

Im C-Programm ist das aber nicht der Fall: das gesetzte Semikolon nach der Schleife schließt bereits die Schleife ab, die Ausgabe wird danach nur einmal aufgerufen. Während in den meisten Sprachen das Einrücken nur zum schönen Stil gehört, um die Lesbarkeit zu verbessern, ist korrektes Einrücken in Python Pflicht. Dafür wird auf blockbildende Sprachelemente verzichtet.

Eine Gefahr dieses Verfahrens liegt im Detail: Das Einrücken darf sowohl durch Leerzeichen wie auch durch Tabulatoren erfolgen. Werden Tabulatoren und Leerzeichen zusammen verwendet, ist die Einrückung nicht mehr eindeutig, da die Größe eines Tabulators in Leerzeichen nicht festgelegt ist. Moderne Editoren und integrierte Entwicklungsumgebungen können dieses Problem lösen.

Manche Entwickler, die bisher andere Programmiersprachen genutzt haben, kritisieren die Blockbildung durch Einrücken als sehr ungewohnt und möglicherweise fehleranfälliger als die klassische Blockbildung: ein unbeabsichtigtes Leerzeichen könne die Struktur des gesamten Quelltextes verändern. Noch existieren keine genauen Untersuchungen, allerdings sind auch bei der klassischen Blockbildung durch unbeabsichtigte Zeichen Fehler möglich, wie etwa obiges C-Beispiel zeigt. Des Weiteren sind Fehler durch falsche Einrückung im Quelltext schneller erkennbar als beispielsweise ein vergessenes Semikolon. Die in Abschnitt 6.1 vorgestellten Studien zeigen, dass die Anzahl an Fehlern in Python geringer ist als in vielen anderen Sprachen.

Ergebnis:

Vorteile:

- Das Einrücken erzwingt gut lesbaren Quelltext
- Fehler leicht erkennbar

Probleme:

- Verwechslungsgefahr von Tabulatoren und Leerzeichen
- Individuelle Gestaltungsspielräume werden eingeengt
- Bei sehr langen, tief geschachtelten Programmen geht die Übersicht verloren
- Einrückung für manche Umsteiger ungewohnt
- mangelnde Unterstützung durch syntaxgesteuerte Editoren (entsprechend Klammersprung-Funktion)

4.5.2 Strings

Für Stringliterale (Zeichenketten) sind in Python mehrere Varianten vorgesehen. Einfache Zeichenketten werden durch einfache oder doppelte Anführungszeichen eingefasst, das jeweils andere Zeichen darf innerhalb des Strings verwendet werden:

```
S1 = " Monty sagt 'Hallo'. "
S2 = ' Monty sagt "Hallo". '
```

Sonderzeichen und Steuerzeichen werden wie in C durch den Backslash "\" markiert, gefolgt von einem Steuerzeichen oder dem oktalen Code.

```
S3 = 'Zeile1 \n Zeile2'
```

Sollen keine Steuerzeichen verwendet werden, können sogenannte Raw-Strings eingesetzt werden, indem dem String ein "r" vorangestellt wird:

```
S4 = r'Hier sind Zeichen wie \ und / erlaubt.'
```

Lange Zeichenketten können über mehrere Quelltextzeilen verteilt sein. Sie werden in dreifache Anführungszeichen (einfach oder doppelt) eingefasst.

```
Langertext=""" Fischer Fritze fischt frische Fische.
Frische Fische fischt Fischer Fritze."""
```

Python unterscheidet nicht zwischen Zeichen und Zeichenketten, jedes einzelne Zeichen ist eine Zeichenkette. Dadurch erspart der Programmierer quelltextaufblähende Konvertierungen. Die Länge von Zeichenketten ist unbegrenzt und muss zuvor nicht wie beispielsweise in Pascal oder C festgelegt werden. Die Speicherverwaltung des Python-Interpreters erledigt selbständig die optimale Ausnutzung des Speicherplatzes.

Strings werden in Python genauso wie immutable[10] Sequenzen behandelt. Dies hat zur Folge, dass alle auf Sequenzen definierten Funktionen und Methoden auch auf Strings anwendbar sind. Hier liegt ein grosser Vorteil von Python: Neben einfachem Umgang mit Strings stellt die Darstellung als „Liste aus einzelnen Zeichen" ein orthogonales Konzept dar, welches nur selten anzutreffen ist (vgl. 5.3). In den meisten Sprachen sind Strings ein eigener, isolierter Datentyp, auf dem eigene Funktionen oder Methoden definiert sind. Insbesondere sind in Python sowohl auf Strings als auch auf Listen und Tupel Addition und Multiplikation definiert. Dadurch lassen sich beispielsweise schnell Vielfache von Zeichenketten erzeugen.

```
>>> s = 'Hallo! '
>>> print s+s
Hallo! Hallo!
>>> print '-'*20
--------------------
```

Weiterhin sind in Python Unicode-Strings möglich; dies wird im Kontext der Internationalisierung noch genauer verdeutlicht.

Auf Strings sind zahlreiche Methoden (wie `upper, lower, replace, find, index` und andere) definiert, die in der für Methoden üblichen Notation `string.methode()` aufgerufen werden. Auch auf Literale ist diese Notation anwendbar, beispielsweise `"Hallo Welt!".upper()`. Es gibt jedoch einen Stilbruch bezüglich der objektorientierten Notation:

```
>>> print len(s)
7
```

[10] immutable (engl.) = unveränderbar

Die Länge einer Zeichenkette wird über eine freie Funktion erfragt. Seit Python 2.1 sind zu diesem Zweck auch Methoden definiert, die Notation als Systemfunktion in der Form `s.__len__()` stellt allerdings eine ungewöhnliche Schreibweise dar. Vergleichbare Probleme bestanden in älteren Python-Versionen bei der Dateiverwaltung, hier ist `open` als Funktion deklariert, welche ein Dateiobjekt liefert. Inzwischen existiert hier eine alternative Notation als `file()`.

In Python kann aus nahezu jedem Objekt eine Darstellung als String gewonnen werden; dazu ist die Funktion `str(objekt)` vorgesehen. Das Objekt kann beliebiger Art sein, angefangen bei einfachen Zahlen und Variablen, bis hin zu Funktionen und Klassendeklarationen. Für Klassen kann hierzu von Programmierer gegebenenfalls die Methode `str` neu definiert werden. Besonders die interaktive Fehlersuche und Tests werden dadurch erleichtert.

Zur Ausgabe, Kombination und Formatierung von Strings und Werten stehen dem Entwickler zwei Alternativen zur Auswahl. Im einfachen Fall können zur Ausgabe mit `print` Strings und Werte mit Kommata getrennt übergeben werden. Alternativ ist eine Formatierung mit unterschiedlichen Platzhaltern möglich, wie es unter anderem aus C bekannt ist. In diesen Beispiel ist `%g` Platzhalter für eine dem String nachgestellte Gleitkommazahl.

```
>>> x=3.14
>>> print "x ist",x
x ist 3.14
>>> print "x ist %g" % x
x ist 3.14
```

Beide Varianten sind in ähnlicher Form auch in anderen Sprachen anzutreffen und haben sich dort bewährt.

Vorteile:

- Direkter Einsatz von Anführungszeichen im String ist möglich
- Behandlung von Strings wie Listen
- Polymorphie durch Addition und Multiplikation
- Umwandlung aller Objekte in Strings möglich
- Keine Unterscheidung von Zeichen und Zeichenketten, Umwandlungen entfallen
- Keine Längenbeschränkungen
- Automatische Verwaltung des Speicherbedarfs

Nachteile:

- Viele Variationsmöglichkeiten zur Darstellung von Strings wirken anfangs verwirrend.
- Mischung von Methoden und Funktionen.

4.5.3 Elementare Datentypen

Jede Programmiersprache verfügt über grundlegende Datentypen wie Zahlen und Zeichenketten. High-Level-Sprachen verfügen, verglichen mit anderen modernen Sprachen, über eine größere Auswahl elementarer Datentypen.

In Python sind folgende Zahlentypen definiert:

- Ganze Zahlen (*int*, mindestens 32 Bits)
- Grosse ganze Zahlen (*long,* nahezu beliebig groß)
- Gleitkommazahlen (*float*, entspricht C-Typ double)
- Komplexe Zahlen (*complex*)

Weitere elementare Datentypen in Python sind Strings, Tupel, Listen, Dictionaries, sowie interne Spezialtypen für Funktionen, Klassen und Quelltext-Objekte. Der Datentyp *Boolean* existiert in expliziter Form noch nicht, stattdessen wird bisher 0 für „falsch" und 1 für „wahr" verwendet. In der kommenden Python-Version sollen *Booleans* geführt werden.

```
zahl = 3.141
tupel = (7,42)
text = 'Hallo.'
komplex = complex(5,-2)
liste = [1,2,3,4]
dict = {1: 'abc', 2: 'def'}
```

Durch Aufnahme dieser Datentypen als Bestandteil der Sprache vereinfachen sich die Ausdrucksmöglichkeiten und die Darstellung im Quelltext. Mit nur einer Zeile Quelltext können so Listen oder andere Strukturen notiert werden, wozu in Java beispielsweise das fünf- bis zehnfache an Zeilen nötig ist (vgl. 4.5.4) . Diese Ausdrucksmöglichkeiten führen zu einer umfangreicheren syntaktischen Beschreibung. Verglichen mit C ist die Syntax von Python, dargestellt in ihrer Backus-Naur-Form, daher auffällig umfangreich. Bei Blick auf andere moderne Sprachen erscheint die Syntax von Python jedoch nicht ungewöhnlich umfangreich.

Komplexe Zahlen und Grundoperationen sind bereits in Python als elementarer Datentyp mit allen Grundoperationen integriert. Dadurch wird die schnelle Entwicklung naturwissenschaftlicher Anwendungen erleichtert und der Quelltext ist leichter lesbar als bei Verwendung von Bibliotheken. Ebenso interessant und in nur wenigen Sprachen zu finden sind die großen ganzen Zahlen. Der Wertebereich ist nahezu beliebig groß und nur durch den Arbeitsspeicher beschränkt. Dadurch bietet sich der Einsatz zum Beispiel bei Verschlüsselungsalgorithmen an, da hier oft mit großen ganzen Zahlen gerechnet wird. Zur Erweiterung der Funktionalität der internen Operationen empfiehlt sich die Bibliothek *Numeric Python* (*NumPy*). Sie bietet hocheffiziente mathematische Funktionen und eignet sich sehr gut zum Rechnen mit großen Datenmengen wie Matrizen und Feldern (siehe 4.10).

Perl bietet im Vergleich zu Python nur drei Gruppen von Datentypen: Skalare, Listen und Dictionaries. Der Datentyp wird durch ein Präfix explizit gekennzeichnet. Es gibt keine unterschiedlichen Zahlentypen, es wird auch nicht direkt zwischen Strings und Zahlen unterschieden. Die Unterscheidung erfolgt erst bei der Interpretation, wozu bestimmte Operatoren eingesetzt werden müssen.

Ergebnis:

Die Integration elementarer Datentypen macht die Nutzung spezieller Bibliotheken überflüssig und bewirkt eine einfache Darstellung im Quelltext. Komplexe und große ganze Zahlen lassen sich mit den gewohnten Operatoren nutzen, dadurch ist weniger Quelltext nötig. Insgesamt wird die schnelle Softwareentwicklung dadurch begünstigt.

4.5.4 Basisdatentypen: Tupel, Listen und Hashes

Die Integration von Basisdatentypen in die Syntax einer Sprache kann die Lesbarkeit der Quelltexte erheblich verbessern: Es müssen keine Bibliotheken zur Nutzung importiert werden, des Weiteren können Strukturen wie Listen direkt im Quelltext notiert werden. Python bietet neben der Integration der wichtigsten Basisdatentypen auch interessante Möglichkeiten des Datenzugriffs.

Tupel sind Gruppierungen beliebiger Objekte zu einer festen, unveränderbaren Einheit. Sie kommen immer dann zum Einsatz, wenn ein Paar oder eine konstante Menge von Objekten gemeinsam verwaltet oder als Parameter übergeben wird. Häufigstes Einsatzgebiet sind Vektoren und Positionsangaben.

```
>>> v = (10,20,30)
>>> print v[0],v[1],v[2]
10,20,30
```

Einzelne Elemente eines Tupels lassen sich über ihren Index in eckigen Klammen auslesen. Tupel sind unveränderbare Folgen, daher ist eine Änderung einzelner Elemente über Ihren Index nicht möglich. Ein Merkmal der Programmiersprache Python ist das sogenannte *Tupel-Unpacking* bzw. die *Mehrfachzuweisung*. Auf der einen Seite einer Zuweisung können mehrere Variablen stehen, auf der anderen ein Tupel. Die eine Seite wird dann komponentenweise der anderen zugewiesen. Dadurch sind mit einer einfachen Notation bequem Vertauschungen und Permutationen erzeugbar.

```
>>> (a,b,c) = 1,2,3
>>> print a,b,c
1 2 3
>>> (a,b) = (b,a)
>>> print a,b,c
2 1 3
```

Tupel-Unpacking darf auch im Kopf von Funktionen eingesetzt werden und bietet so einen schnellen Zugriff auf Elemente eines Tupels. Nur sehr wenige, meist funktionale Sprachen wie ML und Lisp erlauben den Einsatz von Tupeln und bieten dabei erweiterte und leicht lesbare Zuweisungsmöglichkeiten sowie die direkte Nutzung von Tupeln im Interpreter.

Die Notation der Tupel in SML ist vergleichbar mit der von Python:

```
- val x=(1,2);
val x = (1,2) : int * int
- val (a,b)=x;
val a = 1 : int
val b = 2 : int
```

Tupel stellen eine elegante Möglichkeit zur Verfügung, Wertepaare in Quelltext darzustellen und lassen sich im Quelltext gut lesbar darstellen. Sie erlauben einfache und kompakte Parameterübergabe und schnelle Erzeugung von Permutationen sowie Vektoroperationen.

Listen sind im Gegensatz zu den unveränderbaren Tupeln veränderbare und nahezu beliebig große Folgen von Objekten. In Python werden Listen in eckigen Klammern notiert. Der Typ der Elemente innerhalb einer Liste ist beliebig, und nicht wie etwa in SML auf Elemente gleichen Typs eingeschränkt. Dieses ermöglicht beispielsweise Listen von unterschiedlichen Grafikobjekten zu erstellen und mit einem Iterator die einzelnen Objekte zu durchlaufen.

```
>>> liste = [42, 3.141]
>>> liste.append("Hallo")
>>> [1,2,3]+[4,5,6]
[1, 2, 3, 4, 5, 6]
>>> [1,2,3]*2
[1, 2, 3, 1, 2, 3]
```

Neben dem Einsatz von üblichen Methoden wie Einfügen von Elementen, Zählen, Suchen und Sortieren ist auch die Nutzung der Operatoren `+` und `*` vorgesehen. Die Addition ist als Konkatenation zweier Listen definiert und die Multiplikation einer Liste mit einer ganzen Zahl n als n-fache Wiederholung. Einzelne Elemente einer Liste lassen sich wie auch bei Tupeln und Strings über ihren Index ansprechen. Im Gegensatz zu Tupeln ist bei Listen auch das schreibende Ersetzen von Elementen über ihren Index erlaubt.

Die genannten Möglichkeiten erlauben die sehr kompakte Notation und leicht lesbare Darstellung von Listen. In den meisten objektorientierten Sprachen sind Listen nur als Bibliothek integriert. Dadurch ist die direkte Notation von Listen im Quelltext nicht möglich; eine Literal-Notation ist syntaktisch nicht vorgesehen. Das Erstellen einer Liste erfordert nach der Erzeugung einer leeren Listeninstanz viele einzelne Aufrufe der Einfüge-Methode.

Neben der Erzeugung und Darstellung von Listen bietet Python mit dem sogenannten *Slicing* ein elegantes Konzept, auf Teile von Listen zuzugreifen. Slicing ist eine Erweiterung des Zugriffs über Indizes. Es wurde die Notation in Form `liste[von:bis]` eingeführt, um auf einen Ausschnitt aus einer Liste zuzugreifen. Werden `von` oder `bis` nicht angegeben, werden dafür Anfang oder Ende der Liste genommen. Negative Indizes kennzeichnen einen Zugriff relativ zum Listenende. (-1 = letzter, -2 = vorletzter usw.)

```
>>> liste = range(6)
>>> print liste
[0,1,2,3,4,5]
>>> liste[2:5]
[2, 3, 4]
>>> liste[:3]
[0, 1, 2]
>>> liste[-2:]
[4, 5]
>>> liste[1:3]=[21,22,23]
>>> print liste
[0, 21, 22, 23, 3, 4, 5]
```

Slicing kann in gleicher Form auf alle Arten von Folgen angewendet werden, also auch auf Tupel und Strings. Um das Konzept der Indizierung beim Slicing zu erläutern, ist es (insbesondere

für Einsteiger) notwendig, den Sinn des zweiten Index zu verstehen: Die Notation `l[2:5]` lässt vermuten, dass hier die Elemente 2 bis inklusive 5 ausgegeben werden. Tatsächlich beginnt die Indizierung bei Null, und der rechte Wert ist exklusive des letzten (hier fünften) Elements. Aus didaktischer Sicht ist diese Konvention problematisch, etwas mehr Nähe zur mathematischen Schreibweise wäre hier sinnvoll. Diese Form der Notation ist in Python einheitlich für alle Sequenzen und erlaubt die Ermittlung der Sequenzlänge durch Differenzbildung der Grenzwerte.

Die Darstellung von Listen in Python wird von vielen Entwicklern als sehr angenehm beurteilt. Sie ist einfacher als beispielsweise in Lisp, in der Darstellung vergleichbar mit SML, und bietet durch Slicing einfacheren Zugriff.

Listen in Lisp:

```
> (append '((a b)) '(1 2 3))
((A B) 1 2 3)
```

Listen in SML:

```
- val liste=[1,2,3];
val liste = [1,2,3] : int list
```

Tabelle 3 gibt einen Überblick über Basisdatentypen[11] einiger Programmiersprachen[12]:

Sprache	Tupel als Basistyp	Listen als Basistyp	Leicht lesbar
Python	+	+	+
ML	+	+	+
Lisp	+	+	−
Ada	−	−	+
Java	−	−	+
Perl	−	+	−
Eiffel	−	−	+
Tcl	+	+	+

\+ Basistyp vorhanden, − Basistyp nicht vorhanden

Tabelle 3: *Basistypen*

Alle Datentypen in Python sind beliebig kombinierbar und verschachtelbar: Listen, Hashes und Tupel können Objekte aller Art enthalten. Dies ist längst nicht selbstverständlich, beispielsweise können in Perl Arrays keine Arrays enthalten. Selbst die einzelnen Elemente einer Liste etc. können unterschiedlichen Typs sein und zur Laufzeit gegebenenfalls geändert werden.

[11] In Tcl erfolgt keine Unterscheidung von Tupeln und Listen.

[12] Die Einschätzung der Lesbarkeit gibt hier die Meinung des Autors wieder und berücksichtigt die Einschätzung von Entwicklern, die bereits mit anderen Sprachen gearbeitet haben. Siehe auch Abbildung 12.

Ergebnis:

Die Integration von Basisdatentypen wie Listen, Tupel und Hashes[13] ist in Python in einer leicht lesbaren Notation gelungen. Die Erzeugung von Datenstrukturen ist schnell und einfach in einer einzigen Programmzeile möglich. Umfangreiche Manipulationsmöglichkeiten durch Slicing, sowie die Möglichkeit, Operatoren zu nutzen, erlauben einfache und elegante Nutzung dieser Datentypen.

Ungewohnt erscheint Umsteigern von anderen Sprachen die Bereichsdarstellung beim Slicing exklusive des letzten Elements. Dieses Konzept ist allerdings widerspruchsfrei bei allen Datenstrukturen einsetzbar, so dass es keinen negativen Einfluss innerhalb der Sprache hat.

4.5.5 Typisierung und Variablenkonzept

Es werden drei Konzepte der Typisierung unterschieden: Statisch, dynamisch und implizit dynamisch. Variablen sind in Python dynamisch typisiert, das heißt eine Variable kann jederzeit jeden beliebigen Datentyp annehmen. Variablen müssen nicht wie beispielsweise in Java oder C++ deklariert werden, sondern werden durch Zuweisen eines Wertes oder Objektes erzeugt. Durch weitere Zuweisungen können einer Variablen auch Werte anderen Typs zugewiesen werden. Zugriffe auf zuvor nicht definierte Variablen verursachen einen Fehler.

```
>>> x=42
>>> print x
42
>>> x="Hallo"
>>> print x
Hallo
```

Die fehlende Typdeklaration bringt viele Vorteile, aber auch Probleme: Im Sinne von Rapid Prototyping ist es sicher eine Erleichterung. Der Quelltext wird kürzer und übersichtlicher. Polymorphe Operatoren wie etwa + können ohne Typprüfung genutzt werden.

Aufwendige Typumwandlungen (Typecasts), wie sie beispielsweise in C nötig sind, können entfallen, der Interpreter nimmt nötige Typumwandlungen zwischen Zahlentypen weitgehend selbständig vor.

```
>>> x=10
>>> y=3
>>> x/y
3
>>> x2=10.0
>>> x2/y
3.3333333333333
>>> type(x)
<type 'int'>
>>> type(x2)
<type 'float'>
```

[13] In dieser Arbeit wird der Begriff *Hash* als Synonym für gestreute Speicherverfahren (Hashlisten/Dictionaries) verwendet.

Der Benutzer sollte Kenntnis davon haben, dass es verschiedene Typen gibt und jedes Objekt/jede Variable auch einen Typ besitzt. Insbesondere muss er auf korrekten Umgang mit Integerwerten bei der Division achten. Durch Import der Pseudo-Bibliothek *future* kann das Verhalten der Division eingestellt werden. Sollen explizit Typumwandlungen erfolgen, stehen hierzu Funktionen wie `int, float, str, eval` bereit.

Erwähnenswert ist die Eigenschaft von Python, Zahlenüberläufe (Overflows) zu verhindern. Gerät man an die Grenze der meist 32 Bit breiten Zahlendarstellung, wird automatisch auf das nahezu unbegrenzte Long-Format (L) konvertiert. Dabei handelt es sich um eine interne Darstellung unter Nutzung des gesamten verfügbaren Arbeitsspeichers.

```
>>> gross=1000000
>>> gross*gross
1000000000000L
```

Wie an obigen Beispielen zu sehen, erfolgt die Zuweisung eines Wertes an eine Variable wie in C mit dem Operator "`=`", hingegen erfolgen Vergleiche auf Gleichheit mit "`==`". Python-Kritiker bemängeln diese Notation, da die Verwechselung mit der Bedeutung aus der Mathematik zu Fehlinterpretationen verleiten kann und dadurch Neulinge der Programmierung verwirren könnte. Besser wäre aus fachdidaktischer Sicht möglicherweise die Notation "`:=`" aus Pascal oder Eiffel.

Im Gegensatz zu C ist eine Zuweisung ein Statement (Anweisung), nicht ein Ausdruck. Daher benötigt jede Wertzuweisung eine eigene Programmzeile, dabei kann Mehrfachzuweisung und Tupelnotation verwendet werden. Damit wird nicht nur die Lesbarkeit verbessert, sondern auch der Interpreter in die Lage versetzt, versehentliche Vertauschungen von "`=`" und "`==`" sicher zu erkennen.

```
a = b = c = 99
x,y,z = 7,8,9
```

Groß- und Kleinschreibung von Bezeichnern wird in Python berücksichtigt und führt zu verschiedenen Bezeichnern (`A` und `a` sind also zwei verschiedene Bezeichner) (engl. *case sensitive*). Erlaubte Variablennamen enthalten Buchstaben, Ziffern und den Unterstrich. Etwas Vorsicht ist daher bei Wertzuweisungen nötig: Da Variablen nicht deklariert werden, kann es zu unbeabsichtigten Fehlzuweisungen kommen. Syntaktisch lässt sich diese Problem ohne vorausgehende Deklaration prinzipbedingt in keiner Sprache lösen. Seit Python 2.2 ist dieses Problem zumindest für Klassen durch das optionale Attribut `__slots__` vermeidbar[14]. Alternativ bietet sich der Einsatz eines Quelltext-Prüfers wie *PyLint* [Dotfunk02] an, der Quelltexte nach ungenutzten Zuweisungen durchsuchen kann.

Variablen (wie auch Attribute) gelten in Python, sofern nicht explizit durch die `global` Anweisung anders festgelegt, immer lokal, also nur innerhalb der erzeugenden Funktion oder Methode. Soll auf Variablen aus anderen Modulen oder Objekten zugegriffen werden, wird der Modul- oder Objektname getrennt durch einen Punkt vorangestellt. Diese Art der Notation wird in vielen objektorientierten Sprachen eingesetzt und wird allgemein als gut lesbar empfunden.

[14]Hier kann der Entwickler eine Liste gültiger Methoden- und Attributsnamen festlegen.

Im Gegensatz zu Python werden in Perl Variablen für Zahlen und Strings immer in der Form `$name` notiert. Die Variablen sind typlos, die Interpretation erfolgt anhängig von den angewendeten Operatoren. Der Perl-Entwickler muss also wissen, zu welchem Zweck er die Variablen einsetzt. Problematisch ist dabei die viel größere Anzahl nötiger Operatoren. Für Entwickler anderer Sprachen ist die Notation ohne Kenntnisse der Perl-Syntax teilweise unlesbar.

Ergebnis:

Die dynamische Typisierung erspart dem Entwickler aufwendige Typumwandlungen. Dieses Konzept geht allerdings zu Lasten der Typsicherheit und steht im Gegensatz zu einer sehr strengen Typprüfung, wie sie beispielsweise in Algol oder Ada erfolgt [Barnes84]. In Python kann vom Datentyp her nicht zwischen Währungen, Metriken und anderen Zahlentypen unterschieden werden. Als Ausweg bietet sich der Einsatz von Klassen zusammen mit überladenen Operatoren an. Klassen können auf diese Weise zur strengen Typprüfung eingesetzt werden. Alle Standardtypen können ab Python 2.2 auch zur Typerweiterung und somit als Oberklasse oder Superklasse benutzt werden. In Prototypen und Experimentierumgebungen kann auf diese Sicherheit verzichtet werden und dadurch die Entwicklung vereinfacht werden.

Aus fachdidaktischer Sicht kann die dynamische Typisierung die Einführung in die Programmierung erleichtern, da sowohl Typdeklarationen wie auch Konvertierungen entfallen können. Erst in weit fortgeschrittenen und speziellen Lernbereichen, etwa bei systemnaher Programmierung oder Implementierung von Datenstrukturen wie etwa Listen mit Realisierung einer eigenen Speicherverwaltung, ist Python als Sprache weniger geeignet. Python ist als High-Level-Sprache konzipiert, dagegen ist die systemnahe Programmierung mit Low-Level-Sprachen wie C besser realisierbar.

Vorteile:

- Dynamische Typisierung
- Polymorphie von Operatoren
- Variablen als Platzhalter für alle Arten von Objekten, inklusive Funktionen und Methoden
- automatische Typwandlung
- Überlaufbehandlung
- Lokaler Geltungsbereich der Variablen
- weniger Quelltext als in Sprachen mit Typdeklaration
- schnellere Softwareentwicklung

Nachteile:

- Notation der Zuweisung mit `=` statt `:=`
- Gefahr der unbeabsichtigten Zuweisung oder Doppelnutzung von Bezeichnern (vermeidbar)
- Gefahr vergessener Typumwandlungen bei Integer-Division (vermeidbar)

4.5.6 strukturierte Typen und Referenzen

Die Syntax Pythons sieht keine direkte Umsetzung von Datenstrukturen wie etwa ein `struct` in C oder ein `RECORD` in Pascal vor. Es gibt auch keine Zeiger in der Form, wie sie in diesen Low-Level-Sprachen anzutreffen sind. Elementare Datenstrukturen wie Listen, Tupel und Hashes sind dagegen bereits Bestandteil der Sprache.

Zur Umsetzung von zusammengesetzten Strukturen wie etwa Bäumen stehen dem Entwickler andere (und oft bessere) Möglichkeiten zur Verfügung: Eine Möglichkeit besteht in der Umsetzung von Strukturen als Klasse, wobei die Attribute der Klasse den Einträgen eines Records entsprechen. Die Notation für Zuweisung und Referenz entspricht der gewohnten Schreibweise aus dem klassischen `record`.

```
class Tree:
  left = None
  right = None
  value = 0
>>> t=Tree()
>>> t.value=42
```

Zeiger und Speicherreservierungen sind in Python nicht erforderlich. Den Attributen `left` und `right` können direkt Objekte zugewiesen werden. Alternativ lässt sich auch das elementar vorhandene Hash als Struktur benutzen:

```
>>> person={}
>>> person["Name"]="Marvin"
>>> person["Alter"]="33"
>>> print person
{'Name': 'Marvin', 'Alter': '33'}
```

Für reguläre Strukturen wie beispielsweise Vektoren oder Matrizen bieten sich ferner auch Tupel und Listen an.

Vorteilhaft gegenüber klassischen Strukturen ist hier die Polymorphie, da der Typ der Attribute beziehungsweise Elemente nicht festgelegt ist. Ferner sind die Strukturen auch zur Laufzeit problemlos erweiterbar. Bei Änderungen oder Erweiterungen von Strukturen muss keine Definition geändert werden, wie es beispielsweise in `struct` in C oder `RECORD` in Pascal der Fall wäre.

Eine Gefahr ist wieder die ungewollte Zuweisungsmöglichkeit nicht vorgesehener Attribute bei Klassen und Dictionaries. Soll das verhindert werden, kann bei Klassen der Zugriff über Methoden geregelt werden, die gegebenenfalls die Gültigkeit der Parameter testen. Seit Python 2.2 kann zudem durch den Einsatz des optionalen Attributs `__slots__` dieses Problem bei Klassen vermieden werden, indem eine Liste gültiger Attributs- und Methodennamen definiert (quasi deklariert) wird.

Zeiger (Pointer) sind in High-Level-Sprachen wie Python durch Referenzen auf Objekte ersetzt. Auf Referenzen sind keine arithmetischen Operationen erlaubt, der Zugriff erfolgt ausschließlich per Dereferenzierung. Dadurch werden die Unsicherheiten bei der Arbeit mit Zeigern umgangen. Auch die Notation ist simpler, da es nicht mehrere unterschiedliche, nebeneinander nutzbare Notationen für Referenz und referenziertes Objekt gibt, wie beispielsweise in C++ (`int, int&, int*`)

Ergebnis:

Vorteile:

- Klassen und Hashes flexibler als feste Datenstrukturen
- Kompakter Quelltext
- Leicht erweiterbar, auch zur Laufzeit
- Polymorphie
- Verzicht auf Zeiger, dadurch einfachere und eindeutige Notation

Nachteile:

- Gefahr unbeabsichtigter Fehlzuweisungen (vermeidbar).
- Keine statische Kontrolle

4.5.7 Kontrollstrukturen

Bedingte Anweisungen

Bedingte Anweisungen sind in Python wie in vielen Sprachen durch `if ... elif ... else` realisiert:

```
if x==1:
  print "eins"
elif x==2:
  print "zwei"
else:
  print "unbekannt"
```

An dieser Stelle fällt der überflüssig erscheinende Doppelpunkt hinter der Bedingung auf, da die Blockbildung bereits durch die Einrückung vollständig definiert ist. Zweck dieser Konstruktion soll die Möglichkeit für erfahrene Programmierer sein, alternativ direkt hinter der Bedingung eine einzelne Anweisung angeben zu können. So kann etwas kompakterer Quelltext geschrieben werden:

```
if x==1: print "eins"
elif x==2: print "zwei"
else: print "unbekannt"
```

Auf Neulinge kann diese Variationsmöglichkeit verwirrend wirken; ein vergessener Doppelpunkt stellt derzeit eine Fehlerquelle dar, obwohl der Interpreter dies theoretisch als gültige Notation erkennen könnte.

Perl-Programmieren ist die `unless` Anweisung bekannt. Hierbei handelt es sich um ein negiertes `if`. Es kann in Python durch `if not` ersetzt werden. Bei der Konzeption der Programmiersprache Python wurde versucht, auf redundante, alternative Darstellungen eines Programmiersprachenkonstrukts zu verzichten.

Python erlaubt bei Vergleichen Intervallabfragen. Derartige Abfragen sind in nur wenigen Sprachen möglich, obwohl sie leicht verständlich sind und einen zweiten Vergleich ersparen.

```
if 5 <= x <= 10 :
  print "x liegt im Intervall [5,10]"
```

Mehrfachverzweigungen, bei denen mehrere Fälle einer Bedingung getestet werden, sind in Python nicht vorgesehen. Es existiert keine Konstruktion wie `case` in Pascal oder `switch` in C. Der Programmierer muss hier auf eine Folge von `elif`-Klauseln ausweichen. Hierin liegt ein von Umsteigern anderer Sprachen häufig genannter Kritikpunkt; über eine mögliche Erweiterung der Sprache wird derzeit unter den Entwicklern offen diskutiert.

Zyklen

Python kennt zwei Arten von Zyklen: Die bedingte Schleife `while` und die Iteration mit `for`.

Die Schleife mit `while` ist eine Schleife mit Eintrittsbedingung und entspricht der klassischen Form dieser Schleife, wie sie in nahezu jeder Sprache anzutreffen ist. Optional kann in Python wie bei Verzweigungen (`if...else`) auch bei Schleifen ein `else`-Zweig angegeben werden, der einmalig ausgeführt wird, wenn die Eintrittsbedingung nicht erfüllt ist. Des weiteren kann der Körper der Schleife durch die Anweisungen `break` verlassen und `continue` neu gestartet werden.

```
x=1
while x<=3:
  print x
  x = x+1
else:
  print "x>3"
```

Eine direktes Äquivalent für `repeat...until` existiert in Python nicht. Auch hier beschlossen die Python-Entwickler, dass eine Anweisung zur Definition eines Zyklus genügt. Daher muss an dieser Stelle auf eine korrespondierende Konstruktion mit einer Abbruch-Anweisung zurückgegriffen werden:

```
while 1:
  bla()
  if fertig: break
```

In vielen klassischen Sprachen iteriert eine `for`-Anweisung über einen Zahlenbereich. In Python kann im Unterschied dazu über beliebige Arten von Folgen iteriert werden. Eine Iteration besteht aus einer Laufvariablen sowie einer beliebigen Folge in Form eines Tupels oder einer Liste. Die Elemente der Folge dürfen beliebigen Typs sein, beispielsweise Zahlen oder Objekte:

```
for obj in objektliste:
  obj.tuwas()
namen = {'Ingo':30, 'Stefan':32, 'Kai':27, 'Michael':29}
for key in namen:
  print key, namen[key]
```

Seit Python 2.2 können dabei auch Iteratoren und Generatoren verwendet werden, wobei Iteratoren bereits von zahlreichen Basisdatentypen wie Listen und Hashes unterstützt werden.

Zur einfachen Erzeugung ganzzahliger Zahlenfolgen steht die Funktion `range` zur Verfügung:

```
for i in range(5):
  print i
```

An Stelle der Laufvariablen `i` kann auch ein Tupel benutzt werden, dann kommt das mächtige Verfahren der *Tupelzuweisung* zum Einsatz (siehe 4.5.4).

Seit Python 2.0 existiert mit den sogenannten *List Comprehensions* ein Verfahren zur schnellen Definition von Mengen. Zahlreiche Beispiele sind in der Dokumentation enthalten, zur Vollständigkeit sei auch hier eines gezeigt:

```
>>> [i*i  for i in range(6) ]
[0,1,4,9,16,25]
```

Genau wie bei einfachen Zyklen kann bei der Iteration auch mit den Anweisungen `break` und `continue` gearbeitet werden, um den Ablauf der Iteration vorzeitig zu unterbrechen oder mit den nächsten Element fortzufahren.

Ergebnis:

Fallunterscheidungen und Zyklen lassen sich in Python übersichtlich und gut lesbar darstellen. Einzig die Doppelpunkt-Notation stellt eine Fehlerquelle dar, bietet aber gleichzeitig Variationsmöglichkeiten für erfahrene Programmierer.

Positiv fällt die Iteration über alle Arten von Sequenzen und Listen auf, ebenso die erweiterten Kontrollmöglichkeiten (`else`) bei Zyklen. Dagegen werden manche Programmierer die fehlenden Mehrfachverzweigungen vermissen.

4.5.8 Funktionen

Eine Funktion wird in der Programmiersprache Python beschrieben durch ihren Namen, eine Parameterliste sowie ihren Funktionsrumpf. Sie arbeitet mit den Parametern und liefert mit der `return`-Anweisung ein Ergebnis zurück.

Die Parameter einer Funktion sind in Python dynamisch typisiert. Dadurch arbeiten Funktionen vollkommen polymorph. Solange alle Operationen innerhalb der Funktion auf den Parametern definiert sind, liefert die Funktion ein Ergebnis, anderenfalls wird eine Exception erzeugt. Dabei sind alle in Python definierbaren Typen und Objekte *First-Class-Values*, das heißt, auch komplexere Typen wie Listen sind als Parameter verwendbar.

```
def addiere(a,b):
  return a+b
>>> addiere(3,5)
8
>>> addiere("Hallo ","Welt")
'Hallo Welt'
>>> addiere("Dummfug", 42)
TypeError: cannot concatenate 'str' and 'int' objects
```

Möchte der Entwickler sicherstellen, dass ein Parameter einen bestimmten Typ hat, kann dies über die `assert`-Anweisung erreicht werden. Diese bietet gegenüber einer statischen Typprüfung den Vorteil, auch mehrere Typen für einen Parameter erlauben zu können oder den Typ zweier Parameter zu vergleichen. Für den Fehlerfall kann der Entwickler eine optionale Fehlermeldung als Klartext definieren, wodurch die Fehlersuche erleichtert wird. Der Nachteil dieses Verfahrens liegt darin, dass Typfehler erst zur Laufzeit bei Aufruf der Funktion erkannt werden.

In Python wurden die Möglichkeiten der Funktionsdefinition stark erweitert. So wurde die Möglichkeit von Standardparametern und offenen Argumentlisten eingeführt. Dadurch ist die Anzahl an Argumenten variabel. Einzelne Parameter der Funktion können bei ihrer Definition mit Standardwerten vorbelegt werden. Wird bei Aufruf der Funktion ein Parameter nicht übergeben, wird hier der Standardwert genommen. Auf diese Weise ist der Entwickler flexibler als durch den Einsatz von *Curry*-Funktionen[15]. Im Aufruf darf die Reihenfolge der Parameter verändert werden, wenn sie entsprechend der Funktionsdefinition benannt werden.

```
def demo1(a,b=10,c=20,d=30):
  print "a=%s ; b=%s ; c=%s ; d=%s" % (a,b,c,d)
>>> demo1(1,2)
a=1 ; b=2 ; c=20 ; d=30
>>> demo1(1,d="Hallo")
a=1 ; b=10 ; c=20 ; d=Hallo
```

Offene Argumentlisten bieten sich an, wenn die mögliche Zahl der Parameter groß ist und nicht zuvor festgelegt werden kann. Dieser Mechanismus ist vergleichbar mit den Parametern bei Aufruf von Programmen über eine Shell.

In Python werden bei der Funktionsdefinition zwei weitere Parameter `*b` und `**c` angegeben. Alle bei Aufruf der Funktion zusätzlich übergebenen Parameter werden in Form eines Tupels in `b` gespeichert. Parameter, bei denen auch ein Variablenname angegeben wird, werden in Form eines Dictionaries in `c` übergeben.

```
def demo2(a,*b,**c):
  print "a=%s ; b=%s ; c=%s" % (a,b,c)
>>> demo2(1,2,3,4,name="Marvin")
a=1 ; b=(2, 3, 4) ; c={'name': 'Marvin'}
```

Die Einsatzmöglichkeiten von Standardparametern und offenen Argumentlisten sind vielfältig. Insbesondere bei der Programmierung von grafischen Benutzungsoberflächen werden intensiv Standardparameter zur Erzeugung von Objekten angewendet. Hier spart der Entwickler die Übergabe vieler optionaler Parameter ein und kommt so mit weniger Quelltext schneller zum Ziel.

Alle in einer Funktion erzeugten Variablen gelten lokal, also nur innerhalb der Funktion. Ebenso sind empfangene Parameter bei den Basistypen (int, float, complex) lokal. Die explizite Übergabe von Zeigern oder variablen Parametern zur Rückgabe von Ergebnissen ist nicht vorgesehen. Allerdings werden Listen und Objekte von Python automatisch als Referenz übergeben, so dass sich Änderungen innerhalb der Funktion auch außerhalb bemerkbar machen.

[15] *Curry*-Funktion: Aus der funktionalen Programmierung: Vereinfachung einer Funktion durch Definition einer neuen Funktion, welche die ursprüngliche Funktion mit teilweise vordefinierten, konstanten Parametern aufruft.

Dieses Verhalten ist bei Objekten durchaus logisch und korrekt. Bei Listen besteht aber die Gefahr, dass der Entwickler auch hier Lokalität annimmt. Auch in der Informatik-Ausbildung kann die Grenze zwischen Lokalität und Referenz bei Parametern zu Verwirrungen führen. Zur Rückgabe von Ergebnissen sollte daher immer die `return`-Anweisung benutzt werden.

Gegenüber Perl bietet Python hier bessere Möglichkeiten: In Perl werden Argumente ohne Namen in Form einer Liste übergeben. Im Vergleich zu Python erscheint diese Notation unleserlich und ist durch Vertauschungsgefahr eine häufige Fehlerquelle.

Vorteile:

- Polymorphie
- Optionale und flexible Typprüfung
- Möglichkeit, Standardparameter zu setzen
- Offene Argumentlisten
- Vereinfachte und schnellere Entwicklung möglich
- Leicht lesbare Syntax

Nachteile:

- Typprüfung erfolgt erst zur Laufzeit
- Mögliche Probleme bei Übergabe von Parametern als Referenz

4.5.9 Ausnahmebehandlung

Im Ablauf eines Programms kann es immer wieder zu nicht vorgesehenen Fehlern beziehungsweise zu Abweichungen vom normalen Programmablauf kommen. Die möglichen Ursachen sind vielseitig:

- Nicht erfüllte Annahmen
- Typfehler (falsche Übergabe von Parametern)
- Schreibfehler (wie Vertauschung von Parametern, Tippfehler)
- Unbedachte Fehleingaben des Anwenders (unter anderem Strings statt Zahlen)
- Fehler seitens des Systems (beispielsweise Festplatte voll, Fehler im Dateisystem)
- Mathematische Fehler (Division durch Null etc.)

In solchen Fällen ist es wünschenswert, dass ein Programm nicht unkontrolliert abbricht, sondern den Anwender über das Problem informiert und dem Entwickler Möglichkeiten zum Eingrenzen des Fehlers bietet. Erstmalig in Eiffel wurde das Konzept der Annahmen, Kontrollen und Ausnahmen in einer objektorientierten Sprache eingesetzt [Eiffel02, Monninger93]. Allerdings wurde schon in Ada ein ähnliches Verfahren definiert, und bereits Konrad Zuses *Plankalkül* [Zuse45] sah sogenannte *Konditionen* vor.

Annahmen und Kontrollen (*assertions*) beruhen auf den Ingenieurwissenschaften. Sie sind eine Methode der Qualitätssicherung, indem möglichst viele Parameter vor ihrer Nutzung auf Korrektheit und Plausibilität geprüft werden. Eine Ausnahme (*exception*) tritt ein, sobald ein Fehler des regulären Ablaufs festgestellt wurde, wobei zwischen erwarteten und nicht erwarteten Ausnahmen unterschieden wird. Im Fehlerfall kann der Entwickler festlegen, wie weiter vorgegangen werden soll. In Python sind hierzu die Anweisungen bzw. Schlüsselwörter `assert, try, except, finally` und `raise` vorgesehen.

`assert` prüft Voraussetzungen und löst im Fehlerfall eine Ausnahme aus. `try` führt einen Block von Anweisungen aus. Tritt bei der Ausführung des Anweisungsblocks ein Fehler auf, kann der Entwickler die Behandlung einer erwarteten Ausnahme *A* mit `except A` definieren. Nicht erwartetete Ausnahmen lassen sich durch ein allgemeines `except` abfangen. Mit `raise` wird gezielt eine Ausnahme erzeugt, wobei der erkannte Fehler als Klartext übergeben werden kann und so eine leichtere Fehlererkennung möglich ist.

```
try:
  datei=open(dateiname)
except IOError:
  print "Fehler beim Öffnen der Datei",dateiname
except:
  raise "unerwarteter Fehler in Modul test"
```

Obiger Quelltext erkennt Dateifehler und meldet diese dem Benutzer. Andere Fehler, beispielsweise ein unerwarteter Variablentyp als Dateiname, verursachen eine Ausnahme.

Die Ausnahmebehandlung ist eine sehr bequeme Möglichkeit für den Entwickler, Fehler zu erkennen und abzufangen. Bei experimentellem und evolutionärem Prototyping können unvollständige oder fehlerhafte Programmstücke gezielt umgangen oder korrekte Arbeit simuliert werden. Allerdings birgt Ausnahmebehandlung bei falscher Anwendung auch Gefahren:

```
try:
  eine_funktion()
except:
  pass #alle Fehler ignorieren
```

Tritt nun im Funktionsaufruf ein Fehler auf, wird er nicht erkannt, das Programm arbeitet weiter, als wenn nichts geschehen wäre. Es besteht also das Risiko, durch zu pauschale Fehlerbehandlung unerwartete Fehler zu übersehen. Bei Einsatz der Ausnahmebehandlung sollte daher gezielt auf erwartbare Fehler reagiert werden und unerwartete Ereignisse gemeldet werden.

Pythons dynamische Eigenschaften bergen hier die gleichen Risiken wie bei Variablen und Typen (siehe oben). Da Exceptions nicht vorab deklariert werden müssen und auch Codeabschnitte nicht vorher dahingehend gekennzeichnet werden müssen, welche Exceptions sie erzeugen können (vgl. beispielsweise `raises`-Klausel in Java), ist es nicht möglich, vorab die Gesamtstruktur der Ausnahmebehandlung auf Plausibilität zu prüfen.

Ergebnis:

Durch die Ausnahmebehandlung in Python kann der Entwickler gezielt erwartbare und unerwartete Fehler abfangen. Auch die Strukturierung des Programms wird gegenüber einer Fehlervermeidung über einfache Fallunterscheidungen übersichtlicher. Bei unvorsichtigem Einsatz können indes Fehler übersehen werden; der Entwickler muss sich dieser Problematik bewusst sein.

4.5.10 Zusammenfassung

Die Syntax von Python unterstützt die schnelle Softwareentwicklung durch die vorgestellten Konzepte. Insbesondere die integrierten Basisdatentypen, Ausnahmebehandlung und die komfortable Parameterübergabe an Funktionen tragen dazu bei, mit wenig durch die Syntax wohlstrukturiertem Quelltext viel zu erreichen.

Einige prinzipbedingte Konflikte wurden erkannt und können zusammenfassend folgendermaßen charakterisiert werden: Auf der einen Seite erlauben Konzepte wie Polymorphie, Dynamik und Änderungsmöglichkeiten zur Laufzeit schnelles Arbeiten am Prototypen. Andererseits bedeutet diese Vorgehensweise den Verzicht auf Sicherheitsmerkmale wie strenge Typprüfung und sichere Kapselung von Modulen. Somit eignet sich Python gut als Werkzeug für Prototyping, aber weniger in sicherheitskritischen Bereichen.

Aus fachdidaktischer Sicht bietet Python gut lesbaren Code und ist leicht erlernbar. Durch Integration grundlegender Datentypen kann man sich auf Konzepte der Programmierung konzentrieren und braucht nicht in einem frühen Stadium bereits Zeigerstrukturen und Speicherverwaltung einzuführen. Als Nachteil hat sich besonders die fehlende annehmende Schleife und die Grenzenbehandlung bei der Indizierung erwiesen.

4.6 Wiederverwendbarkeit

Um Kosten und Zeit zu sparen, ist die Wiederverwendbarkeit von bereits erstellten Komponenten von Bedeutung. Das Zusammenbauen von Software aus fertigen Komponenten ist schnell und preiswert. Daher sollte jedes Modell daraufhin überprüft werden, ob bereits vorhandene oder erwerbbare Module, Klassen oder Subsysteme wiederverwertet werden können.

In der Softwaretechnik haben sich Verfahren bewährt, die eine Wiederverwendung erleichtern:

- Modularisierung / Bibliotheken
- Polymorphie
- objektorientierter Entwurf
- Entwurfsmuster
- Frameworks

Bei der Wiederverwendung von Software ist der Einsatz von Quelltextdokumentation von zentraler Bedeutung, auf die Möglichkeiten wird weiter unten eingegangen. Zunächst wird die technische Umsetzung der Wiederverwertung in Python untersucht.

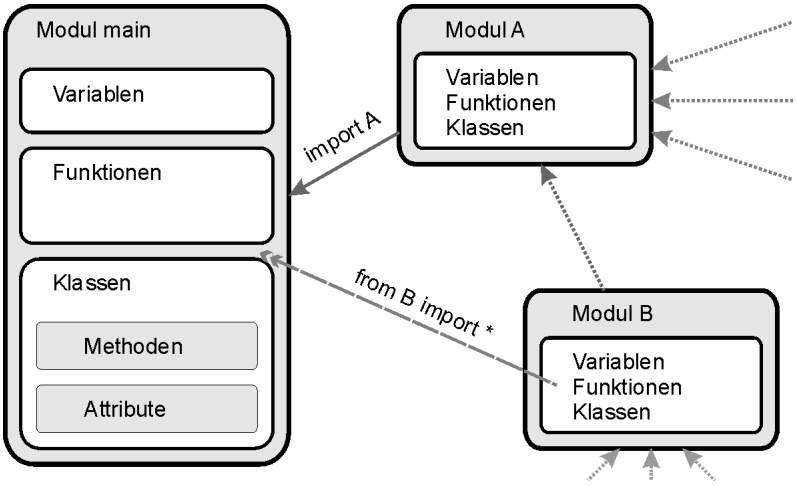


Abbildung 10: *Module in Python*

4.6.1 Modularisierung

Module sind Programmteile, die in eigenen Dateien stehen, um sie in mehreren Programmen verwenden zu können. Die Aufteilung eines Projektes in einzelne Module ist eine schon seit längerer Zeit eingesetzte Methode, die Produktivität der Softwareentwicklung zu erhöhen [Balzert00, Parnas01]. Die wesentlichen Vorzüge sind:

- verbesserte Strukturierung der Software
- Aufteilung in Teilprojekte für unabhängige Entwickler
- separate Testbarkeit einzelner Module
- Wiederverwendbarkeit einzelner Module in anderen Projekten
- Bibliothekskonzept für häufig genutzte Module

In Python kann jedes Modul globale und lokale Variablen, Funktionen und Klassen mit ihren Methoden und Attributen enthalten. Über die `import`-Anweisung kann ein Modul andere Module oder Teile davon übernehmen.

In Python gibt es drei Möglichkeiten, wie ein Modul (beispielsweise `main`) andere Module importieren und auf dessen Variablen und Funktionen zugreifen kann. Der Quelltext

```
import A
A.funktionA()
```

lädt das Modul `A` und stellt es unter diesem Namen im aktuellen Namensraum zur Verfügung. Der Aufruf von Funktionen (und ebenso Variablen sowie Klassen) erfolgt in der Notation `A.funktionsname()`, der Aufruf von `A.irgendwas` greift auf Interna von `A` zu.

Dagegen wird mit

```
from B import *
funktionB()
```

das Modul `B` geladen und alle seine Elemente im aktuellen Namensraum global zur Verfügung gestellt. Somit wird das aktuelle Modul um den importierten Inhalt des Moduls `B` ergänzt. Der Aufruf einer Funktion erfolgt direkt über ihren Namen. Sind gleichnamige Funktionen bereits zuvor definiert, werden diese durch die Definition in `B` ersetzt. Diese Art des Modulimports birgt die Gefahr, unbeabsichtigt vorhandene Definitionen zu verändern. Es wäre sogar möglich, dass Modul `B` Operatoren wie die Addition umdefiniert und dadurch unbeabsichtigte Seiteneffekte erzeugt oder sogar ein Programm unbrauchbar machen kann.

Andererseits liegen hier auch interessante Möglichkeiten: Beispielsweise lassen sich Debugger oder gezielt definierte Systemfunktionen importieren. Im informatischen Ausbildungsbereich kann beispielsweise eine auf den Kurs angepasste Lernumgebung importiert werden. Einige dieser Lernhilfen werden später noch vorgestellt (siehe 5.6).

Die Konstruktion

```
from C import funktionC
funktionC()
```

entspricht der Variante `B`, nur wird hier nur eine einzelne Funktion (beziehungsweise Variable oder Klasse) des Moduls importiert. Sowohl Variante `B` wie auch `C` legen im Gegensatz zu `A` lokale *Kopien* der importierten Funktionen und der enthaltenen Variablen an. In Anhang A.6 wird der Unterschied an einem Beispiel verdeutlicht, dieser ist insbesondere von Einsteigern nicht leicht zu verstehen. Durch Verzicht auf modulübergreifende globale Variablen kann dieser Problematik jedoch lange ausgewichen werden.

Die Variante `B` gilt unter Entwicklern als schlechter Stil. Sie widerspricht dem Konzept, dass ein Modul gekapselt sein sollte. In der Softwaretechnik [Balzert00] definiert man ein Modul `M` durch seine Schnittstelle und den Rumpf. Die Schnittstelle spezifiziert die zur Verfügung stehenden Dienste, im Rumpf ist der Dienst implementiert. Über den Import wird eine Benutzbarkeitsbeziehung zu dem zu nutzenden Modul hergestellt, wobei das Modul eine abgeschlossene Einheit bilden soll.

Auch wenn ein Modul mit Variante `A` importiert wird, ist es nicht sicher gekapselt. Es besteht in Python die Möglichkeit, bereits importierte Dienste eines Moduls neu zu definieren:

```
A.funktionA = neue_Funktion()
```

Damit wird die Definition der `FunktionA` in Modul `A` durch `neue_Funktion()` ersetzt. Hier liegt ein Konflikt zwischen zwei Konzepten vor: Einerseits die sichere Kapselung, andererseits die Möglichkeit, gezielt Module zu verändern. Letzteres hilft dem Entwickler bei schneller Softwareentwicklung, da sogar zur Laufzeit eines Programms Funktionen und Module geändert werden können. Vorteilhaft ist das beispielsweise bei der Fehlersuche oder der experimentellen Änderung von Prototypen im Interpreter. Den Python-Entwicklern ist dieser Konflikt bekannt, über mögliche Lösungen wird derzeit diskutiert.

Das Modulkonzept von Python unterscheidet sich etwas von dem Modulkonzept anderer Sprachen wie beispielsweise Perl. Perl erlaubt sowohl dem Modul als auch dem importierenden Programm eine Entscheidung, welche Funktionen beziehungsweise Namen im importierenden Programm ohne Qualifizierung verwendbar sein sollen. Perl bietet dadurch etwas mehr Sicherheit als Python.

Module in Perl müssen als solches durch ihre Dateiendung `.pm` gekennzeichnet sein. Hier ist Python gegenüber Perl flexibler: Ein Modul unterscheidet sich von einem Programm nur durch die Verwendung, nicht durch speziellen Code. Ein Python-Skript kann erkennen, ob es als Modul oder als Hauptprogramm gestartet wird. Dadurch kann Test- und Demonstrationscode in ein Python-Modul eingebaut werden, der nur ausgeführt wird, wenn das Modul direkt gestartet wird. Er muss nicht entfernt werden, wenn die Funktionalität eines Moduls genutzt werden soll.

Bei objektorientierter Programmierung können Klassen beliebig in Python-Module verteilt werden, auch mehrere Klassen können in einem Modul definiert werden. In Perl entspricht eine Klasse immer einem Modul.

Zur Installation eines Moduls sind in Python mehrere Verfahren möglich:

1. Manuelle Installation durch Kopieren der Modul-Dateien in die Python-Verzeichnisse
2. Installation als `.rpm`-Paket beziehungsweise über ein Installationsprogramm
3. Nutzung des genormten *Distutils*, die Installation erfolgt über ein Installationsskript

Die letzten beiden Varianten sind auch für unerfahrene Benutzer leicht durchführbar, Modulentwickler sollten daher möglichst eines dieser Verfahren nutzen.

Ergebnis:

Vorteile:

- Hohe Flexibilität
- Unterstützt Prototyping, Experimente und Fehlersuche
- Einfache Erweiterungsmöglichkeiten des Basis-Systems
- Klassen beliebig in Module verteilbar
- Test- und Demonstrationscode ist auf Modulebene integrierbar

Nachteile:

- Unterschiedliche Methoden des Modulimports
- Unsichere Kapselung
- Risiko unbeabsichtigter Neudefinition

4.6.2 Polymorphie

Polymorphie bedeutet, dass eine Funktion oder ein ganzes Modul auf unterschiedlichen Objekttypen arbeiten kann. In der objektorientierten Programmierung kann dieselbe Nachricht an Objekte verschiedener Klassen gesendet werden, welche sie dann unterschiedlich interpretieren.

Wie bereits bei den Datentypen und Funktionen gezeigt (siehe 4.5.8), arbeitet Python so weit wie möglich polymorph. Algorithmen, Klassen und ganze Module sind dadurch universell nutzbar und können leicht wiederverwertet werden. Beispielsweise kann ein einmal implementierter Sortieralgorithmus Objekte aller Art bearbeiten, sofern eine Ordnungsrelation (Halbordnung) bezüglich der Objekte definiert ist.

Im Gegensatz zu Python ist Perl hier weniger flexibel. Skalare, Listen und Dictionaries unterscheiden sich bereits syntaktisch durch das Präfix. Für Vergleiche gibt es keinen universellen Operator, es muss abhängig vom Objekttyp mit unterschiedlichen Operatoren gearbeitet werden.

Python ist bzgl. Polymorphie ähnlich leistungsstark wie die funktionale, jedoch statisch typisierte Sprache SML. Erst bei Ausführung eines Skriptes erfolgt die Typauswertung. Solange alle Methoden beziehungsweise Operatoren auf einen Typ definiert sind, ist ein Skript ausführbar.

Bei der Definition eigener Klassen ist wie in Eiffel das Überladen von Operatoren möglich. Das heißt, es können eigene Methoden für Operatoren wie Addition, Vergleich und Komplement definiert werden.

Ergebnis:

Vorteile:

- Methoden- und Funktionsdefinitionen vollkommen unabhängig vom Objekttyp
- Überladen von Operatoren möglich
- Hoher Level von Polymorphie erleichtert Wiederverwertbarkeit

4.6.3 Objektorientierter Entwurf

Python wurde als objektorientierte Sprache konzipiert und bietet somit alle erforderlichen Voraussetzungen. Auf die objektorientierte Programmierung mit Python wird in Abschnitt 5.1.3 genauer eingegangen.

4.6.4 Refactoring

Refactoring wird das Umstrukturieren von Software genannt, ohne das Verhalten nach außen zu ändern. Dies geschieht typischerweise zur Verbesserung von Lesbarkeit, Wartbarkeit oder bei allgemeinen Entwurfsänderungen. Prototypen entwickeln sich evolutionär, daher fallen häufig Änderungen im Sinne von Refactoring an.

Typische Änderungen sind:

- Aufteilung eines Moduls in Teilmodule
- Komposition oder Aufteilen von Methoden
- Verlagern von Methoden in andere Klassen
- Strukturierung von Daten
- Vereinfachung von Fallunterscheidungen

Daraus ergeben sich Anforderungen an die Entwicklungsumgebung, die denen des Prototypings ähneln: Veränderungen an Klassen werden durch Entwicklungsumgebungen unterstützt, die schnell eine Klassenübersicht erzeugen können, beispielsweise in Form eines integrierten Klassenbrowsers. Python unterstützt Refactoring durch dynamische Attribute; so lassen sich schnell Klassen ändern und erweitern, ohne gleichzeitig Probleme strenger Typisierung beachten zu müssen.

4.6.5 Entwurfsmuster

Entwurfsmuster sind ein Konzept zur Lösung von wiederkehrenden Entwurfsproblemen der objektorientierten Softwareentwicklung. Eine gute allgemeine Einführung in die Thematik ist in [GHJV96] zu finden. Ein Muster beschreibt ein beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem. Die Lösungen sind wiederverwendbar und durch Objekte und Schnittstellen beschrieben.

Muster können grob in drei Kategorien eingeordnet werden: *Erzeugungsmuster*, *Strukturmuster* und *Verhaltensmuster*. Bekannte Vertreter sind die *Abstrakte Fabrik*, der *Dekorierer* und der *Beobachter*. Implementierungen vieler in [GHJV96] vorgestellten Muster sind im *Python Cookbook* von *ActiveState* [PCOOK02] zu finden.

Als Beispiel befindet sich mit `singleton.py` im Anhang A.6 eine Implementierung einer *Queue* (Warteschlange) in Form eines *Singletons*. Ein *Singleton* ist ein Erzeugungsmuster, welches sicherstellt, dass eine Klasse genau ein Exemplar besitzt. Wird versucht, mehrere Exemplare zu erzeugen, kann die Klasse selbst den Zugriff regeln, und beispielsweise globalen Zugriff auf dieselbe Instanz erlauben oder einen Fehler melden. Ersteres realisiert das Beispielprogramm: Auch wenn mehrere Instanzen der Queue erzeugt werden, handelt es sich faktisch um dieselbe Queue. Dies ist vergleichbar etwa mit einer systemweiten Druckerwarteschlange, auf die mehrere Anwendungen Zugriff haben.

4.6.6 Zusammenfassung

Python unterstützt wie auch andere moderne Sprachen einen modularen, objektorientierten Softwareentwurf. Dynamische Typisierung und Polymorphie vereinfachen gegenüber statisch typisierten Sprachen die universelle Nutzbarkeit erstellter Module. Wiederverwertung von Software bedeutet aber auch, die nötigen Softwarekomponenten zur richtigen Zeit zu finden und ihre Funktionsweise zu verstehen. Die Möglichkeiten der Quelltextdokumentation spielt dabei eine wesentliche Rolle, wie in nächsten Abschnitt gezeigt wird.

4.7 Dokumentation

Neben separat verfügbaren Dokumentationen zur globalen Funktionsweise eine Moduls ist auch die Dokumentation innerhalb des Quelltextes Bestandteil der Softwareentwicklung. Die Dokumentation sollte aus folgenden Gründen bereits während der Softwareentwicklung erfolgen:

- Entwicklungsentscheidungen werden festgehalten, um bei Modifikationen bereits gemachte Erfahrungen auswerten zu können.
- Sofortiges Festhalten der Entscheidungen stellt sicher, dass keine Information verloren geht und bereits während der Entwicklung verfügbar ist.
- Im Quelltext integrierte Dokumentation reduziert den Aufwand zur Gesamtdokumentation.
- Die Einarbeitung in fremde Quelltexte und Module wird erleichtert.
- Die Wartbarkeit des Produkts wird verbessert.
- Die Wiederverwendung wird erleichtert.

In den meisten Programmiersprachen erfolgt die Dokumentation über besonders gekennzeichnete Kommentarzeilen. Dabei wird typischerweise die Parameterübergabe und die Arbeitsweise einer Funktion direkt zusammen mit den Funktionskopf ausführlich dokumentiert. Innerhalb einer Funktion werden Dokumentationstexte nur noch zu Erläuterung einzelner Schritte genutzt.

In Python sind zwei Notationen für Kommentare vorgesehen: Kurze Kommentare beginnen mit `#` und können an das Ende einer Zeile angehängt werden. Längere Kommentare werden als String in ein oder drei Anführungszeichen eingefasst und dürfen im letzteren Fall auch über mehrere Zeilen reichen.

```
>>> class spam:
...   "Demonstration der Klassendokumentation"
...   def methode(self,arg):
...     "Parameter: arg = Dieses Argument wird ausgegeben"
...      print arg #ein kurzer Kommentar
>>> spam.__doc__
'Demonstration der Klassendokumentation'
>>> spam.methode.__doc__
'Parameter: arg = Dieses Argument wird ausgegeben'
```

Module, Klassen, Methoden und Funktionen interpretieren einen Kommentarstring vor dem ersten inneren Ausdruck als einen ihnen zugeordneten Dokumentationstext, in Python *Docstring* genannt. Dieser String kann so als Dokumentation für das Objekt herangezogen werden. Der Zugriff kann über das für alle diese Strukturen vordefinierte `__doc__`-Attribut erfolgen. Weitere Informationen sind interaktiv über die `help()`-Funktion aufrufbar.

Der Entwickler hat während der Laufzeit die Möglichkeit, direkt im Interpreter die Dokumentation zu nutzen. Bei der experimentellen Arbeit an Prototypen ist dieser Zugriff schneller und einfacher als eine aufwendige Suche in einer externen Dokumentation oder eine Suche im

Quelltext. Auch in bereits kompilierten Modulen, bei denen der Quelltext nicht vorliegt, ist diese Art des Zugriffs möglich.

Zur Erstellung einer vollständigen Quelltextdokumentation eines Produktes kann das Pythonskript *Happydoc* herangezogen werden [Hellmann02]. Happydoc liest rekursiv ein gesamtes Quelltextverzeichnis ein und erstellt anhand der Funktionsköpfe sowie der Dokumentationsstrings eine vollständige Beschreibung des Gesamtprojekts. Die Dokumentation umfasst alle Module, Klassen, Methoden und Funktionen und kann in gängigen Formaten wie HTML, XML oder als UML-Diagramm ausgegeben werden.

Dem Python-Entwickler stehen zahlreiche weitere Werkzeuge zur Erstellung von Dokumentationen zur Verfügung, Tabelle 4 stellt die bekanntesten dar. Eine vollständige und aktuelle Übersicht ist im Internet auf [Parn02] einsehbar. JavaDoc stellt hier das Vorbild dar, welches von den Python-Versionen bezüglich Funktionsumfang bisher noch nicht erreicht worden ist.

Name	**Zweck**
Dbdoc	Analysiert Datenbanken (Oracle und PostgreSQL) und erzeugt eine HTML-Dokumentation
Happydoc	Analysiert Python-Quelltext und erstellt eine vollständige Dokumentation als HTML oder XML
Pythondoc	Analysiert Python-Quelltext und erstellt eine vollständige Dokumentation als HTML, XML und anderen Formaten
GenDoc	Erzeugt aus Python-Quelltext HTML oder ASCII-Dokumentationen.

Tabelle 4: *Dokumentationswerkzeuge*

Ergebnis:

Vorteile:

- Dokumentation in sowohl als Bestandteil des Quelltextes wie auch in kompilierten Modulen integriert
- Schneller Zugriff direkt im Interpreter
- Einfache Erstellung einer Gesamtdokumentation möglich

4.8 Softwaretest

Vielseitige und interaktive Testmöglichkeiten sind ein typisches Merkmal interpretierter Skriptsprachen. Insbesondere bei der Erstellung von Prototypen werden Experimente und interaktive Tests eingesetzt, ebenso stellen zyklische Testphasen einen Bestandteil der Methode des Extreme Programming dar (siehe 2.2). Interpretersprachen unterstützen die Änderung einzelner Funktionen während der Laufzeit und erlauben sofortigen Test, ohne das gesamte Projekt inklusive Testprozeduren neu kompilieren zu müssen.

Pythons Modulkonzept unterstützt den Gebrauch von Testcode auf besondere Weise. Da Python Hauprogramme nicht von Modulen und Bibliotheken unterscheidet, kann jedes Modul wie ein eigenständiges Programm genutzt werden und mit einem eigenen Test- oder Beispielcode versehen werden. Dazu ist in Python eine einfache Möglichkeit vorgesehen, wie ein Skript erkennen kann, ob es als importiertes Modul oder als eigenständiges Programm gestartet wird. Am Ende des Skripts wird folgendes Statement eingefügt:

```
if __name__=="__main__":
  starten()
```

Dabei ist `starten()` eine beliebige Funktion des Moduls, in welcher der Testcode implementiert werden kann. Die Variable `__name__` enthält entweder den Namen eines Moduls oder den String `"__main__"`, wenn das Modul direkt gestartet wurde.

Praktisch jedes existierende Python-Bibliotheksmodul nutzt diese Möglichkeit, um mit einfachen Beispielen die Funktionsweise der realisierten Schnittstellen zu demonstrieren und um durch Testfälle die korrekte Funktion zu prüfen. Somit erlaubt dieses Verfahren neben der Testmöglichkeit auch die Unterstützung der Dokumentation; fremden Entwicklern wird das Verständnis der Funktionsweise durch funktionsfähige Beispiele erleichtert.

Realisiert ein einzelnes Modul beispielsweise einen grafischen Benutzungsdialog, kann der Dialog direkt ausgeführt werden, ohne ihn in das vollständige Programm zu integrieren. Entspricht das Ergebnis nicht den Erwartungen des Entwicklers oder Anwenders, wird im Interpreter die fehlerhafte Funktion geändert und erneut ohne lange Kompilierzeit getestet (Evolutionäres Prototyping).

In den Python-Bibliotheken ist das spezielle Modul *unittest* enthalten. Dabei handelt es sich um eine Umsetzung der Testumgebung *Junit* von Kent Beck aus Java beziehungsweise Smalltalk, welche sich als de-facto-Standard in diesen Sprachen bewährt hat. Eine vollständige Funktionsübersicht ist in der Python-Dokumentation enthalten.

Ergebnis:

Vorteile:

- Einfache interaktive Testmöglichkeiten zur Laufzeit des Programms
- Unterstützt evolutionäres und Experimentelles Prototyping
- Testcode und Beispiele direkt in Module integrierbar
- Umsetzungen bewährter Testumgebungen aus anderen Sprachen sind vorhanden

4.9 Fehlersuche

Kaum ein Programm ist direkt nach der Implementierung fehlerfrei. Mögliche Fehlerquellen und die gezielte Behandlung von Ausnahmen wurden bereits vorgestellt. Im Folgenden wird untersucht, wie Python auf unterschiedliche, nicht erwartete Fehler reagiert und den Entwickler bei der Fehlersuche (Debugging) unterstützt.

Syntaxfehler meldet der Python-Interpreter als sogenanntes *Traceback* direkt nach der Eingabe beziehungsweise beim Importieren von Modulen. Dabei wird die fehlerhafte Quelltextzeile auszugsweise ausgegeben, der erkannte Fehler markiert und der Modulname mit zugehöriger Zeilennummer angefügt. Syntaxfehler lassen sich so schnell lokalisieren und beseitigen. Zugriffe auf nicht definierte Symbole (Variablennamen, Module, Funktions- und Klassendefinitionen) werden von Python erst zur Laufzeit erkannt. Sobald ein Symbol definiert wird, wird es in den Namensraum eingefügt und steht von diesem Zeitpunkt an zur Verfügung. Ist ein Symbol nicht im Namensraum eingetragen, meldet Python beim Zugrifssversuch einen `NameError` beziehungsweise `AttributeError`. Ebenso werden Typinkompatibilitäten bei Zuweisungen erst zur Laufzeit erkannt und unter Angabe der Typen gemeldet.

Sehr ungewöhnlich bei der Typprüfung verhalten sich Vergleichsoperatoren: Vergleiche unterschiedlicher Typen führen nicht zu Typfehlern:

```
>>> 5<"Hallo"
1
>>> 5+"4"
File "<interactive input>", line 1, in ?
TypeError: unsupported operand types for +: 'int' and 'str'
>>> 5<"4"
1
```

Ursache dafür ist die in Python implizit definierte Ordnung auf Objekttypen; so ist ein Integer immer kleinerer Ordnung als ein String. Nach Auskunft der Python-Entwickler (in der Newsgroup comp.lang.python) soll damit das einfache Aufbauen von Binärbäumen aus beliebigen Objekten vereinfacht werden. Aus fachdidaktischer Sicht ist dieses Verhalten problematisch, da unbeabsichtigte Vergleiche von Strings und Zahlen möglicherweise nicht im Sinne des Entwicklers bearbeitet werden, wie es im obigen Beispiel illustriert wurde.

Lokalisierung von Fehlern, die vom Python-Interpreter gemeldet werden, ist auch von ungeübten Entwicklern leicht möglich: Der Interpreter liefert aussagekräftige Fehlermeldungen, wie folgendes Beispiel zeigt:

```
Traceback (most recent call last):
File "modul1.py", line 6, in ?
modul2.produkt("Hallo","Welt")
File "modul2.py", line 3, in produkt
return a*b
TypeError: unsupported operand type(s) for *: 'str' and 'str'
```

Im Python-Interpreter sind umfangreiche Möglichkeiten zur Fehlersuche integriert, und bereits die von System ausgegebenen Fehlermeldungen sind sehr aussagekräftig. Der Entwickler erfährt die Programmzeile inklusive Quelltextauszug, den Modulnamen und den eigentlich aufgetretenen Fehler. Zusätzlich wird ein Ablaufprotokoll der letzten wichtigen Statements ausgegeben. Über den Interpreter lassen sich dann die Variablen genauer untersuchen und einzelne Funktionen mit veränderten Parametern ausführen. Funktionen können umdefiniert und der Programmablauf wieder aufgenommen werden.

In Python sind über die Module *traceback* und *inspect* zahlreiche interne Funktionen des Python-Interpreters nutzbar, über die mit wenigen Programmzeilen ein leistungsfähiger Debugger implementiert werden kann: Die Module ermöglichen detailliert Einblick in die Tiefen

des Interpreters und geben beispielsweise Zugriff auf die gerade ausgeführte Programmzeile sowie den vollständigen Namensraum. Wie einfach damit ein Debugger realisiert werden kann, demonstriert das Modul *debugger.py* in dem zu dieser Arbeit erstellten Projekt *PyNassi*: In diesem Projekt werden einige Elemente, die als Vorstudie für einen Debugger dienen, realisiert. Sie erlauben das Verfolgen des Ablaufs und geben aussagekräftige Mitteilungen, die es gestatten, den Programmablauf detailliert zu inspizieren. Allerdings werden keine Änderungen während des Ablaufs ermöglicht. Die Möglichkeiten des Debuggings machen sich alle integrierten Entwicklungswerkzeuge zu nutze. Bereits das in Python enthaltene *Idle* erlaubt schrittweisen Programmablauf mit genauer Quelltextinspektion; *WingIDE* ermöglicht darüberhinaus die Analyse von verteilt arbeitenden Programmen. Einige integrierte Entwicklungswerkzeuge werden in Kapitel 5.8 vorgestellt.

Möchte ein Entwickler gezielte Information über den Ablauf eines Programmes erhalten, ohne in den Ablauf einzugreifen, werden häufig Logdateien eingesetzt. Für die Ausgabe und Formatierung werden in anderen Sprachen wie Java bereits Module zur Ausgabe der Meldungen in bewährten Notationen eingesetzt. Mit *log4py* [Preishuber02] steht dem Python-Entwickler eine Umsetzung des aus Java bekannten *log4j* zur Verfügung. Es unterstützt Protokollierung in Dateien sowie über TCP/IP mit konfigurierbaren Ausgabeformaten und Detailstufen.

Ein Vorteil bei Interpretersprachen ist die Möglichkeit, den Programmablauf jederzeit unterbrechen zu können, um einzelne Variablen und Objekte zu untersuchen. Bei der Definition von Klassen kann in Python eine Methode `__repr__` zur Darstellung des Objektes als Text definiert werden. Damit wird insbesondere die Fehlersuche vereinfacht und die direkte Ausgabe von Objektattributen ermöglicht.

```
class Vector:
  "Eine Vektorklasse"
  def __init__(self,x,y):
    self.x=x
    self.y=y
def __repr__(self):
  return "Vector (%s, %s )" % (self.x, self.y)
>>> v=Vector(3,-1)
>>> print v
Vector (3, -1 )
```

Wird keine Textdarstellung definiert, enthält die Ausgabe den Objekttyp und eine interne Adresse:

```
<__main__.Vector instance at 0x010BC410>
```

Während der Laufzeit hat der Entwickler Zugriff auf Komponenten der Implementierung. Mit einer Vielzahl von Methoden wie `__dict__`, `__class__`, `__name__`, `type` lassen sich Klassen und Objekte untersuchen (die sog. Introspektion).

```
>>> v.__class__.__name__
'Vector'
>>>v.__class__.__dict__
{'__module__': '__main__',
'__repr__': <function __repr__ at 0x010C56C0>,
'__init__': <function __init__ at 0x010B50C8>,
'__doc__': "Eine Vektorklasse"}
```

Ergebnis:

Python bietet interaktive Fehlersuche auf einem hohen Niveau. Die genauen Fehlermeldungen machen den Einsatz eines Debuggers zur Fehleranalyse oft schon überflüssig. Zum schrittweisen Beobachten des Programmablaufs sind bereits enthaltene Werkzeuge gut geeignet, die durch kostenlose Werkzeuge insbesondere bezüglich des Bedienungskomforts noch erweitert werden können.

Vorteile:

- Detaillierte Fehlerausgaben des Interpreters
- Interaktive Testmöglichkeiten
- Im Interpreter integrierte Debugging-Funktionen
- Introspektion zur genauen Analyse von Objekten
- Zuverlässige Erkennung von Syntaxfehlern
- Zugriff auf interne Zustände des Interpreters
- Leistungsfähige und kostenlose Entwicklungsumgebungen mit Debugger verfügbar
- Übliche Schnittstellen zur Erstellung von Logdateien während des Programmablaufs

Nachteile:

- Fachdidaktisch nicht sinnvoller Vergleich inkompatibler Typen wird toleriert
- Viele Fehler sind prinzipbedingt durch die Dynamik erst durch Laufzeittests auffindbar

4.10 Bibliotheken

Zur möglichst schnellen Lösung eines softwaretechnisches Problems wird ein Entwickler versuchen, dieses Problem in Teilprobleme zu zerlegen und anschließend schauen, ob es für diese Teilprobleme schon fertige Lösungen in Form von Modulen bzw. Bibliotheken gibt. Die Menge und Auswahl an Bibliotheken und deren Qualität[16] ist für die unterschiedlichen Bereiche das primäre Kriterium für einen Entwickler, daher wird in den nächsten Abschnitten auf die wichtigsten Einsatzgebiete genauer eingegangen.

Schnelle und kostengünstige Verfügbarkeit sind zwei weitere Anforderungen bei der schnellen Softwareentwicklung. Der Entwickler muss in möglichst kurzer Zeit seine vorhandenen Bibliotheken durchsuchen können wie auch Zugriff auf externe Suchfunktionen für Bibliotheken haben.

In Python ist bereits eine sehr umfassende und gut dokumentierte Bibliothek für zahlreiche Probleme enthalten. Eine Übersicht der Bibliotheken ist in der englischen Python-Dokumentation enthalten, deutsche Literatur ist ebenso verfügbar [Lundh01]. Findet der Entwickler unter

[16] Qualität ist zu verstehen bzgl. Vollständigkeit, Fehlerfreiheit, Performance, Dokumentation...

den Systembibliotheken keine geeignete Lösung, kann er auf eine große Sammlung meist kostenloser Bibliotheken zugreifen. Nahezu alle Bibliotheken – derzeit knapp 2000 – sind im Internet auf [Parn02] registriert. Mit geeigneten englischen Suchanfragen oder über eine Auswahl nach Kategorien wird der Entwickler hier fast immer schnell fündig. Ein vollständiges deutsches Register ist derzeit nicht vorhanden.

Python kann bezüglich der Anzahl[17] an verfügbaren Bibliotheken nicht ganz mit Perl mithalten, da Perl hier durch seine frühere Verbreitung einen zeitlichen Vorsprung von mehreren Jahren hat und Perl schon lange insbesondere für CGI-Skripte und andere Netzwerkaufgaben im Einsatz ist. Die Suche nach Bibliotheken und Skripten erfolgt für Perl über *CPAN Search* (Comprehensive Perl Archive Network Search) [CPAN02]. Auch hier erfolgen Anfragen überwiegend mit englischen Suchbegriffen oder über eine Auswahl nach Kategorien. Inhaltlich werden von den Bibliotheken beider Sprachen vergleichbare Bereiche abgedeckt, der Perl-Entwickler hat jedoch eine größere Auswahl an Alternativen.

Die Situation im Vergleich mit Java ist ähnlich, auch in Java ist die Zahl verfügbarer Bibliotheken größer. Der Java-Entwickler ist allerdings auf die technischen Möglichkeiten der virtuellen Java-Maschine beschränkt, systemnähere Module wie beispielsweise diverse GUIs kann der Java-Entwickler nicht nutzen.

Anders sieht die Situation gegenüber meisten anderen Sprachen aus. Typischerweise sind Entwickler mit C++, Delphi oder Eiffel auf meist teure, kommerzielle Lösungen angewiesen, wo der Java-, Perl- oder Python-Entwickler sich aus kostenlosen Quellen bedienen kann. Die Standard C++-Bibliotheken sind durch strengere Typisierung und fehlende Dynamik deutlich komplizierter zu handhaben und weniger flexibel.

Python-Basisbibliotheken

Die Python-Basisbibliotheken sind in der Dokumentation in Kategorien gegliedert, wie in Tabelle 5 dargestellt wird.

Weitere systemspezifische Bibliotheken sind teilweise in Distributionen mit integrierter Entwicklungsumgebung wie beispielsweise *ActivePython* enthalten und auch separat verfügbar. Die *Win32 Extension*s enthalten Schnittstellen zu den Systemfunktionen von Microsoft Windows. Die *Win32 Com Extensions* enthalten Erweiterungen zur COM- Unterstützung.

Für Macintosh-Rechner existieren mit den *MacPython*-Modulen Schnittstellen zu den Mac-Systemfunktionen wie dem Finder, EasyDialogs und Kommunikationsfunktionen.

Die mitgelieferten Systembibliotheken bieten dem Entwickler eine ungewöhnlich große Auswahl. In den nächsten Abschnitten wird eine Auswahl von Bibliotheken aus unterschiedlichen Kategorien vorgestellt, um die einfache Nutzung exemplarisch zu demonstrieren.

Literatur:

- *Python Standard-Bibliothek* [Lundh01]

Strings: Reguläre Ausdrücke

Reguläre Ausdrücke werden in der Programmierung größtenteils als Suchmuster für Strings verwendet. Es gibt verschiedene, leicht unterschiedliche Notationen zum Schreiben regulärer

[17] Die exakte Anzahl war leider nicht feststellbar.

Kategorie	**Module (Auswahl)**
Strings	Reguläre Ausdrücke, C-Strings, ...
Diverses	Mathematische Funktionen, Modultest
Allgemeine Betriebssystemfunktionen	Datei- und Prozessverwaltung, Terminal, ...
Spezielle Betriebssystemfunktionen	Signals, Sockets, Threading, ZIP-Archive, ...
UNIX-Spezifische Dienste	POSIX, Pipes, Shell, ...
Python Debugger	Debugger zur Fehlersuche
Profiler	Zeitmessung, Laufzeitoptimierung, ...
Internetprotokolle	HTTP, FTP, Mail, Webbrowser, Webserver, ...
Internet Datenbehandlung	Mail-Parser, MIME, Mailbox, Multifile, ...
Strukturierte Datenformate	HTML, XML, SGML, ...
Multimedia	Audio- und Bilddatenzugriff
Kryptografie	RSA, HMAC, ROTOR, ...
Grafische Benutzungsschnittstellen	Tkinter
Eingeschränkte Programmausführung	Rechtevergabe für Quellcodes (rexec)
Python-Sprachdienste	Parser, Compiler, Distutils
SunOS- + IRIX-spezifische Dienste	Audio, GL, ...
MS Windows-spezifische Dienste	Audio, Registryzugriff

Tabelle 5: *Basis-Bibliotheken*

Ausdrücke, wobei die auch in Perl verwendete Notation inzwischen am weitesten verbreitet ist [Freidl02].

Seit Python 2.0 können reguläre Ausdrücke über das Modul *re* in Python eingesetzt werden. Die Bearbeitungsmöglichkeiten von Strings durch Einsatz von Slicing erlauben dabei eine besonders einfache Nutzung. Reguläre Ausdrücke werden von Python intern zu einem endlichen Automaten[18] kompiliert, der auch auf sehr großen Strings schnell und effizient arbeitet.

```
>>> import re
>>> txt="blabliblabla 192.168.0.1 blabliblabla"
>>> treffer=re.compile("[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*").search(txt, 1)
>>> treffer.group()
'192.168.0.1'
```

Diverse Dienste: Iteration über Textdateien

Das einfache Iterieren über Zeilen einer Textdatei wird in Skripten oft benötigt. Python (ab Version 2.2) bietet hier durch die integrierte Iteration eine elegante Lösung:

```
datei=open("test.txt")
for zeile in datei:
  print zeile
datei.close()
```

[18]Es gibt Algorithmen basierend auf endlichen Automaten, die Stringsuche sehr effizient in linearer Rechenzeit mit o(Stringlänge+Suchtextlänge) erlauben.

Allgemeine Systemfunktionen: Gepufferte Verzeichnisdienste

Das Modul *dircache* liest mit nur einem Funktionsaufruf ein Verzeichnis des Dateisystems in einen Puffer und gibt eine Liste der gefundenen Dateien und Verzeichnisse zurück:

```
>>> import dircache
>>> dateiliste=dircache.listdir('/')
```

Python profitiert hier von den in die Sprache integrierten Listen sowie durch Verzicht auf unnötige Initialisierungen, wie es beispielsweise in C++ und Java der Fall wäre.

Erweiterte Systemfunktionen: Nebenläufigkeit (Threading)

Über das Modul *threading* hat der Entwickler die Möglichkeit, Funktionen als Thread zu starten. Zur Synchronisation sind Locks beziehungsweise Semaphore vorgesehen. Für einfache parallel laufende Prozesse existiert mit dem *timer* eine einfache Unterklasse der Klasse *threading*. Listenoperationen sind in Python atomar[19], so kann hier auf Semaphore verzichtet werden.

Internet-Protokolle

Die schnelle und einfache Entwicklung von Speziallösungen für Netzwerk- und Internetanwendungen ist heute ein wichtiges Einsatzgebiet von Skriptsprachen. Noch ist Perl die derzeit meist eingesetzte Sprache, jedoch entdecken immer mehr Entwickler die von Python gebotenen Möglichkeiten. Die zu erstellenden Programme sind typischerweise recht klein und werden oft von nur einem oder zwei Programmierern (Pair Programming Team, vgl. 2.2) in kurzer Zeit implementiert.

Pythons Basis-Bibliothek kann hier ihre Leistungsfähigkeit zeigen. Für alle bekannteren Protokolle existieren Module, die unabhängig von der Plattform eingesetzt werden können:

- TCP/IP
- FTP
- NNTP
- HTTP
- E-Mail (POP, SMTP, MAPI)
- Internet-Datenbehandlung: Parser für HTML, MIME, Email
- SGML- / XML-Verarbeitung
- Kommunikation mit Webbrowsern
- CGI-Entwicklung

[19] atomar = können unabhängig von mehreren Prozessen genutzt werden

- Ein vollständig in Python implementierter HTTP-/Webserver

Die Dokumentation und einfache Beispielprogramme sind bereits in die Python-Dokumentation integriert.

Im Folgenden zeigen einige Beispiele, wie einfach mit Python die Nutzung von Internetdiensten möglich ist.

Beispiel: Lesen einer Webseite per HTTP:

```
import urllib
datei = urllib.urlopen("http://ddi.cs.uni-dortmund.de")
text = datei.read()
datei.close()
```

Beispiel: Senden einer Nachricht per SMTP:

```
import smtplib
def eingabe(prompt):
    return raw_input(prompt).strip()
smtphost = eingabe("Adresse des Mailservers:")
fromaddr = eingabe("Von........: ")
toaddrs = eingabe("An.........: ")
subject = eingabe("Betreff....: ")
msgtext = eingabe("Nachricht..: ")
# Erzeuge den Mail-Header
msg = ("From: %s\r\nTo: %s\r\nSubject:%s\r\n\r\n" % (fromaddr, toaddrs, sub-
ject) )
# Nachricht anhängen
msg = msg + msgtext
server = smtplib.SMTP(smtphost)
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Auch hier ist der eigentliche Protokollanteil nur vier Programmzeilen lang, die restlichen Zeilen dienen der Dateneingabe. Für die meisten modernen Programmiersprachen existieren Bibliotheken mit vergleichbarer Funktionalität, deren Handhabung allerdings häufig aufwendiger ist.

Auch wenn diese Beispiele nur einen sehr kleinen Ausschnitt der gesamten Bibliothek zeigen, ist zu erkennen, dass auf unnötige Vorbereitungen verzichtet werden kann. Hier kann der Python-Entwickler besonders von den dynamischen Datentypen und der integrierten Speicherverwaltung profitieren. Für das Aufnehmen des Seitentextes im ersten Beispiel muss beispielsweise zuvor kein Objekt als Platzhalter erzeugt werden. Strings und Sequenzen sind bei allen Protokollen geeignete Platzhalter für Eingaben und Rückgaben beliebiger Größe. Ein weiterer Vorteil: Die Python-Bibliotheken sind kostenlos und bereits in der Standard-Distribution enthalten.

Im Vergleich zu anderen aktuellen Skriptsprachen bietet nur Perl eine deutlich größere und ebenfalls kostenlose Bibliothek. Die Nutzung der Perl-Bibliotheken ist wegen des Typkonzeptes und der namenlosen Parameterübergabe allerdings weniger flexibel und schlechter lesbar.

Literatur:

- Python Standard-Bibliothek [Lundh01]
- Python 2 [LF02]

Web-Applikationen mit Zope

Zur professionellen Entwicklung internetbasierender Anwendungen existiert mit *Zope* [Zope02] ein leistungsfähiges Werkzeug. Es enthält zahlreiche Werkzeuge und Module, welche die Entwicklung von Web-Anwendungen vereinfachen, wie beispielsweise einen integrierten Mail- und Webserver, Anbindung an Datenbanken, eine internetbasierte Entwicklungsumgebung und vieles mehr. Primäre Einsatzgebiete sind alle Anwendungen, bei denen ein Nutzer über das Internet mit einer zentralen Anwendung arbeiten soll, wie beispielsweise Suchmaschinen und internetgestützte Datenbanken.
Zope ist eines der umfangreichsten verfügbaren Werkzeuge auf dem Gebiet der Web-Programmierung über Skriptsprachen. Primär wurde *Zope* für Python entwickelt, inzwischen werden weitere Programmiersprachen wie Perl unterstützt. *Zope* ist ein Open-Source-Projekt, daher ist der Einsatz kostenlos. Für Linux, UNIX, Windows, und MacOS X sind angepasste Versionen verfügbar.

Datenbearbeitung: XML

Zum Zugriff auf XML-Daten haben sich in anderen Sprachen zwei Schnittstellen etabliert: SAX (Simple API for XML) und DOM (Document Object Model). Diese beiden Schnittstellen wurden auch nach Python portiert. Als weitere Schnittstelle kann die *xmllib* verwendet werden, die in Anlehnung an die bestehenden HTML-Bibliotheken entwickelt wurde.

Multimedia

Pythons Basisbibliothek hält bereits einfache Funktionen zur Bilddatenbearbeitung und zur Audiowiedergabe bereit (Module *audioop* und *imageop*) Für reine Multimedia-Anwendungen und Spiele gibt es einige Zusatzmodule, wobei hier die Bibliothek *pygame* den größten Funktionsumfang bietet und bereits in vielen Anwendungen und Spielen eingesetzt wird.

Pygame arbeitet auf den Plattformen Windows, MacOS X, BeOS, FreeBSD, IRIX und Linux. Es setzt auf der verbreiteten SDL-Multimedia-Bibliothek auf und ist dadurch aus Entwicklersicht vollkommen hardwareunabhängig. Dennoch bietet es sehr schnelle Nutzung der Multimedia-Hardware. Unterstützt werden 2D- und 3D-Bildschirmausgaben, Sprites, MP3- und CD-Wiedergabe, Bildbearbeitung, Abfrage von Eingabegeräten wie Joysticks sowie Audio- und Videowiedergabe. Zahlreiche Beispielprogramme wie beispielsweise das schön animierte *Pygris* zeigen die Leistungsfähigkeit und die einfache Nutzung dieser Bibliothek. An *Pygame* wird deutlich, dass auch die Programmierung schneller Spiele in einer High-Level-Sprache wie Python möglich ist.

Datenbanken

Zur Nutzung von Datenbanken steht dem Entwickler eine gute Auswahl an Bibliotheken zur Verfügung. Es gibt zahlreiche Varianten an Schnittstellen zu einfachen Objektspeichern und relationalen Datenbanken, von denen die bekanntesten in Tabelle 6 aufgelistet sind.

Die meisten Bibliotheken sind schon seit längerer Zeit erhältlich, in vielen Anwendungen erprobt und gut dokumentiert. Die Anzahl der bei [Parn02] registrierten Datenbankbibliotheken und auch die Zahl der unterschiedlichen Datenbanken liegt auf dem Niveau von Perl (Perl: 81,

Python-Modul	**Erläuterung**
Anydbm	Allgemeine Schnittstelle zu UNIX-DBM Datenbanken
PySQLLite	Wrapper für SQLite
SQLBuilder	Erzeugt SQL-Ausdrücke
PyPGSQL2	Anbindung an Postgre SQL
DMTools	Anbindung an MySQL
Python-DBCore, Dbfreader	Anbindung an DBASE
Sybasemodule	Anbindung an Sybase SQL
DCOracle, Cx_Oracle	Anbindung an Oracle
mxODBC	DBI 2.0 kompatible ODBC Datenbankanbindung
Informixdb13	Anbindung an Informix

Tabelle 6: *Datenbankmodule*

Python: 76; Stand Juni 2002). Bei Python kommen noch die bereits in den Basisbibliotheken vorhandenen *DBM-* sowie *XML*-Module hinzu. Viele Datenbanken bilden den Inhalt einzelner Datensätze auf ein Python-Dictionary ab, so dass der Entwickler hier einfache Möglichkeiten zum Datenzugriff hat.

Mathematik und Statistik

Für mathematische und statistische Aufgabenstellungen sind einige hundert freie Bibliotheken vorhanden, beispielsweise *stats* für statistische Funktionen, *disipy* zur Visualisierung. Für eine vollständige Aufzählung fehlt hier der Platz, daher sei hier auf die Rubrik *Math* in [Parn02] verwiesen. Das Angebot ist insgesamt sehr groß und deckt weite Bereiche der Mathematik, Physik, Chemie und weiterer Naturwissenschaften ab. Ebenso sind Schnittstellen zu verbreiteten Programmen wie *Mathematica* und *Gnuplot* erhältlich.

Einen besonderen Bekanntheitsgrad hat die Bibliothek *numeric* erlangt. *Numeric* ist in C implementiert und erweitert Python um hocheffiziente Funktionen für Berechnungen an Feldern und Matrixen. Damit läßt sich die Interaktivität und die elegante Syntax von Python mit der nötigen Performance für umfangreiche mathematische Berechnungen kombinieren. Mit diesen Bibliotheken wird im professionellen Umfeld gearbeitet, beispielsweise vom Fermilab [Fermilab02].

Messen, Steuern und Regeln

Die Nutzung externer Hardware bereitet in Python keine Probleme. Es gibt eine gute Auswahl an Modulen für allgemeine und spezielle Hardware. Beispielsweise existieren spezielle Schnittstellen zu Navigationsgeräten, speicherprogrammierbaren Steuerungsgeräten usw. So setzt beispielsweise die Firma *gluIT* [Gluit02] Python-Module zur Überwachung von Containerkränen am Hamburger Containerterminal Altenwerder ein.

Robotik und Simulation

Rene Liebscher [Liebscher00] hat ein System zur Simulation von Robotern erstellt. Sein Projekt zeigt gleichzeitig die Möglichkeiten der Simulation, kombiniert mit einer grafischen dreidimen-

sionalen Visualisierung. Weitere Bibliotheken gestatten die Simulation von Automaten und Turing-Maschinen sowie einigen speziellen biologischen und chemischen Vorgängen [Parn02].

Realzeitsysteme

Zum Einsatz unter dem Realzeitsystem QNX RTP existieren angepasste Python-Versionen, die auf [Schwill02] verfügbar sind.

Besonderheiten

Eine besonders zu erwähnende Eigenschaft von Python ist das Modul *pickle*. Dieses Modul gestattet mit wenigen Aufrufen das Schreiben ganzer Objekte und Objektstrukturen in Dateien. Das Modul *pickle* speichert rekursiv ein Objekt und alle Attribute ab, genaugenommen wird es serialisiert und an einen geeigneten Ausgabekanal (z.B. auch über eine TCL/IP-Verbindung) gesendet. Dazu sind lediglich vier Anweisungen nötig:

```
datei=open("mein_obj.dat","w")
p=pickle.Pickler(datei)
p.dump(mein_objekt)
datei.close()
```

Zum Lesen einer so gesicherten Objektstruktur genügt ein einzeiliger Aufruf der Methode *Unpickler*. Es wird dann ein neues Objekt mit allen Attributen erzeugt. Die Objekte werden in einer ASCII-Codierung gespeichert und sind daher prinzipiell unabhängig von genutzten Python-Interpreter und der Plattform. Bei Migration von Windows nach UNIX zeigt sich derzeit noch ein Problem: Die Zeilenenden werden in Windows mit `<CR><LF>` markiert, während UNIX beim Lesen nur ein `<CR>` erwartet. Aus diesem Grund müssen derzeit Windows-Dateien erst in ein UNIX-konformes Format umgewandelt werden. Eine Lösung dieses Problems wird für die kommende Python-Version erwartet.

Nur sehr wenige Sprachen unterstützen direkt das Speichern und Lesen von Objekten. Für den Entwickler ist diese Eigenschaft eine gewaltige Erleichterung. Das rekursive Sichern von Objekten bedeutet in klassischen Programmiersprachen einen extrem hohen Programmieraufwand, da jedes Attribut sowie die Verschachtelung der Objekte einzeln gesichert werden muss. Hierzu sind viele Ausgabeanweisungen und Codierungsfunktionen erforderlich. Oft ist dabei für die Sicherungsmethoden einzelner Objekte mehr Quelltext nötig, als für alle anderen Methoden zusammen. *Pickling* ist hier eine geschätzte Erleichterung für den Entwickler und kann die Softwareentwicklung erheblich beschleunigen.

Ergebnis:

Python enthält eine sehr gute Basisbibliothek und kann gut mit externen Bibliotheken erweitert werden. Umfangreiche Systembibliotheken aus allen Anwendungsgebieten, insbesondere der Netzwerkprogrammierung, sind kostenlos verfügbar und gut in die Python-Dokumentation integriert. Die Menge verfügbarer Bibliotheken ist sehr groß, als Skriptsprache bietet derzeit nur Perl eine deutlich größere Auswahl.

Die schnelle Softwareentwicklung wird nicht nur durch die gute Auswahl an Bibliotheken unterstützt, sondern auch durch Nutzung sprachlicher Eigenschaften. So kann durch häufige

Nutzung von Standardparametern auf oft unnötige Parameterübergaben oder Definition vieler Einzelfunktionen verzichtet werden. Pickling ist ebenfalls eine Erleichterung, die nur wenige Sprachen[20] als Bibliothek bieten.

4.11 Kopplungen

Kopplungen verschiedener Module, Systeme und Programmiersprachen erfolgen häufig, um die jeweiligen Vorteile der unterschiedlichen Sprachen zu nutzen oder zur Anbindung an bestehende Systeme. Nachfolgend werden Besonderheiten und Möglichkeiten von Python diskutiert.

4.11.1 Jython=Java+Python

Jython (früher *JPython*) ist ein Implementierung des Python-Interpreters in Java [Jython02]. Es ist damit möglich, innerhalb der Java-Umgebung Jython-Skripte auszuführen und Jython-Bibliotheken zu verwenden, und so Python-Funktionalität der Java-Umgebung hinzuzufügen. Umgekehrt lassen sich aus Jython auch die Java-Klassen nutzen; so können beispielsweise *AWT* und *Swing* verwendet werden. Auch lässt sich Jython zur interaktiven Fehlersuche innerhalb von Java einsetzen.

Jython ist vollkommen hardware-unabhängig. Die Jython-Skripte werden in echten Java-Bytecode übersetzt, und sogar eine Kompilierung zu statischem Bytecode ist möglich, um beispielsweise Beans oder Applets zu erstellen.

Über zwei Probleme sollte der Entwickler sich vor dem Einsatz von Jython allerdings klar sein:

1. Die Nutzung systemnaher Bibliotheken wie Grafik-Bibliotheken ist in der gekapselten Java-Box nicht möglich.
2. Die Geschwindigkeit des Jython-Interpreters beträgt nur etwa 10–20% einer normalen Python-Umgebung.

Dagegen ist Jython eine elegante Lösung, wenn beispielsweise durch Projektvorgaben Java verlangt wird. Dann lassen sich gleichzeitig die Vorteile der schnellen Softwareentwicklung durch Python zu nutzen.

4.11.2 Erweitern und Einbetten

Erweitern und Einbetten *(Extending* und *Embedding)* ist das Kombinieren einer Programmiersprache mit anderen Sprachen, um zusätzliche Funktionalität bereitzustellen. Eine solche Kombination kann in zwei Richtungen erfolgen: Entweder man erweitert Python, so dass Funktionen der anderen Programmiersprache in Python erreichbar sind, oder man erweitert das Programm in der anderen Sprache um einen eingebetteten Python-Interpreter.

Typischerweise geschieht dies durch ein separates Modul mit dem Ziel, leistungskritische oder systemnahe Programmteile in einer schnellen und systemnahen Programmiersprache zu implementieren. Dagegen werden die restlichen Programmteile in einer High-Level-Sprache wie

[20] Java bietet ebenfalls die sogenannte Serialization

Python realisiert, um hier von den Vorteilen der schnelleren Entwicklung und den typischerweise kompakteren Quelltexten zu profitieren.

Python bietet die Möglichkeit, Module in C / C++ zu nutzen (Extending); das Vorgehen wird beispielsweise in [LF02] ausführlich beschrieben. Zur Konvertierung von Objektdaten in für C geeignete Formate und umgekehrt sind geeignete Funktionen vorhanden. Möglich ist das Erzeugen von Funktionen, Klassen und sogar die Definition neuer Typen.

Ein Beispiel ist die Grafikbibliothek *PyGame*. Die zeitkritischen Grafikfunktionen sind in C implementiert, für den Python-Entwickler werden einfach nutzbare Schnittstellen und Klassen in einem separaten Modul zur Verfügung gestellt.

Das Einbetten (Embedding) von Python-Skripten in andere Anwendungen erfolgt typischerweise durch Einbindung des Python-Interpreters und der Python-Bibliothek in den C / C++ -Quelltext. Der Interpreter wird von dem aufrufenden Programm initialisiert und das gewünschte Python-Skript gestartet.

4.11.3 Prozess-Kommunikation

Der Daten- oder Nachrichtenaustausch zwischen den eigenständigen Komponenten einer Software erfolgt häufig über Prozesskommunikation. Dazu werden typischerweise entweder sequenzielle Kommunikation oder Bibliotheken zum gemeinsamen Objektzugriff eingesetzt.

Zur sequentiellen Kommunikation können einfache Sockets bzw. TCP/IP-Verbindungen verwendet werden. Darüber hinaus kann zusammen mit Java unter Jython auch *JMS (Java Messaging Service)* zum Einsatz kommen. Die erforderlichen Bibliotheken für die serialisierte Datenkommunikation sind bereits in der Python- bzw. Jython-Distribution enthalten.

Gemeinsamer Zugriff oder Austausch von Objektdaten erfolgt über bekannte Schnittstellen wie *CORBA (Common Object Request Broker Architecture)* oder *SOAP (Simple Object Access Protocol)*. Diese bieten zahlreiche Dienste an, um Objekte und Methoden gemeinsam zu nutzen. Unter Python existieren hierzu jeweils mehrere Bibliotheken, wie beispielsweise *Fnorb* oder *Soapy*.

Als (sehr gute) Alternative bieten sich Frameworks[21] wie beispielsweise *Twisted, ACE* und *OSE* an, welche ebenfalls Anbindungen an objektorientierte Dienste wie *SOAP* und weitere Kommunikationsdienste bereitstellen. Alle genannten Bibliotheken sind auf [Parn02] registriert.

4.12 Zusammenfassung

In diesem Kapitel wurde Python anhand der zuvor aufgestellten Kriterien untersucht, welche überwiegend die schnelle Softwareentwicklung betreffen. Es zeigte sich, dass Python den Anforderungen gerecht wird. Besondere Vorzüge liegen in der Syntax und Semantik, wozu insbesondere die Basisdatentypen sowie die dynamische Typisierung beitragen. Auch die vielseitigen Möglichkeiten des GUI-Entwurfs und hervorragende Testmöglichkeiten zeichnen Python aus. Ebenso positiv ist die umfangreiche Bibliothek zu werten, wobei insbesondere die Module für Netzwerk- und Internetprogrammierung genannt werden müssen.

[21] Framework = Coderahmen, mit dem ermöglicht wird, eine Klasse von Problemen zu lösen, indem gezielt Methoden überschrieben (=gefüllt) werden müssen, die applikationsspezifisch ausgeprägt sind. Erste Frameworks waren Lisa Toolkit (Macintosh) und MacApp für die Macintosh-Benutzungsoberfläche.

Die größten Kritikpunkte sind die Vergleichsmöglichkeit inkompatibler Typen sowie Detailprobleme in der Notation. Letztere sollten nicht überbewertet werden, dürfen aber, speziell im Kontext der informatischen Bildung, auch nicht ignoriert werden. In Kapitel 7 werden in einer Übersicht alle geprüften Kriterien einer Gesamtbewertung unterzogen.

5 Python aus fachdidaktischer Sicht

Der einen Sprache bedarf man, um sich verständlich zu machen, der anderen, um sich selbst zu verstehen.

Elazar Benyoëtz (*1937)

Die bisherigen Schwerpunkte der Analyse lagen weitgehend auf softwaretechnischen Aspekten der Sprache, wie Syntax, Funktionsumfang und anderen Spracheigenschaften. In diesem Kapitel wird näher auf die praktische Anwendung der Sprache eingegangen, wobei besonders auf fachdidaktische Merkmale Wert gelegt wird. Die wesentlichen zu untersuchenden Aspekte sind:

- Unterstützung unterschiedlicher Paradigmen der Programmierung
- Einfache Lesbarkeit
- Orthogonalität: einfache, klare und widerspruchsfreie Konzepte
- So einfach wie möglich: Verzicht auf Unnötiges
- Schnelle Erlernbarkeit
- Eingabehilfen bei der Implementierung
- Verfügbarkeit von Lernwerkzeugen zur Visualisierung

5.1 Umsetzung von Paradigmen der Programmierung

In diesem Abschnitt wird untersucht, inwiefern es mit Hilfe der Programmiersprache Python möglich ist, unterschiedliche Programmierparadigmen (Programmierstile) umzusetzen. Python wurde als moderne, objektorientierte Programmiersprache konzipiert. Gleichzeitig bietet es als Skriptsprache – im Unterschied zu rein objektorientierten Sprachen (wie Java oder Smalltalk) direkte Unterstützung prozeduraler und damit imperativer Konzepte. Funktionale Programmierkonstrukte, die durch Lisp, Scheme oder auch Mathematica breiter bekannt (und benutzt) wurden, sind ebenfalls in Python integriert. Über Bibliotheken lassen sich darüber hinaus auch prädikative Programmierkonzepte einsetzen.

Eine allgemeine Einführung mit Musteraufgaben zum Verständnis unterschiedlicher Paradigmen gibt [BS97]. Dort wird unter anderem das bekannte Pizza-Beispiel behandelt und eine Realisierung mit unterschiedlichen Paradigmen durchgespielt.

5.1.1 Imperative / prozedurale Programmierung

Erste Programmiersprachen wie Konrad Zuses Plankalkül [Zuse45] und der Lambda-Kalkül von Alonzo Church entstanden zunächst nur auf dem Papier. Imperative Programmiersprachen wie FORTRAN und COBOL waren die ersten weiter verbreiteten und auf einem Computer nutzbaren höheren Programmiersprachen. Die Methode der imperativen Programmierung ist älter als die ersten technisch realisierten Computersysteme.

Imperative Programmiersprachen zeichnen sich durch eine enge Anlehnung an die sogenannte *von-Neumann-Rechnerarchitektur* aus, die auf der Idee eines Speichers mit Daten und Instruktionen, einer Steuer- und einer Verarbeitungseinheit basiert. Die Anlehnung beruht vor allem in der sequentiellen schrittweisen Ausführung von Instruktionen (Befehlen, Anweisungen) mit der Möglichkeit der Wiederholung bestimmter Befehlssequenzen sowie der Zuordnung von Speicherzellen zu Werten. Über Zuweisungsoperationen können Werte abgelegt, modifiziert und abgerufen werden. Eine Variable repräsentiert den Inhalt einer oder mehrerer Speicherzellen. Die *von-Neumann-Rechnerarchitektur* entspricht dem Aufbau praktisch aller gängigen Prozessoren und läßt sich maschinennah in imperative und prozedurale Sprachen nutzen[22].

Der prozedurale Programmierstil erweitert die imperative Programmierung um Abstraktionsmechanismen zur Bildung von Prozeduren (Unterprogrammen). Zu einem gegebenen Problem wird eine Lösungsbeschreibung im Form sich untereinander aufrufender Prozeduren entwickelt. Die Prozeduren selbst enthalten Anweisungen zur Daten-Ein- und -Ausgabe, Variablenzuweisungen und Aufrufe anderer Prozeduren. Aus der prozeduralen Programmierung ergibt sich das sogenannte EVA-Konzept (Eingabe, Verarbeitung, Ausgabe):

$$\text{Dateneingabe} \Rightarrow \text{Prozedur} \Rightarrow \text{Datenausgabe}$$

Bekannte Vertreter der prozeduralen Programmiersprachen sind FORTRAN, COBOL, BASIC, Ada, C, Pascal, Modula-2 und Oberon.

In Python kann diese Art der Programmierung direkt und bruchlos umgesetzt werden. Als Beispiel ein kurzes prozedural realisiertes Programm zur Berechnung der Fakultät einer ganzen Zahl:

```
# Berechnung der Fakultät einer ganzen Zahl,
# prozedural gelöst durch eine Scheife.
def fakultaet(n):
  assert n>=0
  r=1
  for i in range(2,n+1):
    r=r*i
  return r
eingabe=input("Berechne die Fakultaet von:")
ergebnis=fakultaet(eingabe)
print "Die Fakultaet von",eingabe,"ist",ergebnis
```

Der Vorteil einer rein prozeduralen Programmierweise gegenüber streng objektorientierter Sicht liegt häufig in der Einfachheit. Schreibaufwendige Initialisierungen können entfallen. So ist beispielsweise das klassische *Hello-World*-Beispiel mit Basic oder Python in einer Zeile implementiert, während in Java schon für solch einfache Aufgaben Klassen nötig sind:

```
class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

[22] Diese Sicht ist sehr deutlich von Bauer (1997) kritisiert worden, der fragt, ob es sinnvoll war, sich so lange an dem konkreten Maschinenmodell zu orientieren statt sich stärker auf die zu modellierenden Gegenstände zu beziehen.

In diesem Beispiel fallen zahlreiche Schlüsselwörter wie `class`, `public`, `static` an, welche sich fachdidaktisch schwierig rechtfertigen lassen: Diese müssen erklärt werden, damit die Schüler wissen, wie sie eine Lösung für ein Problem formulieren (vgl. [Church41]).

Ergebnis:

Prozedurales Programmieren ist bei Kleinstaufgaben oft einfacher als objektorientiertes Vorgehen. Für einfache Aufgaben nach dem EVA-Prinzip fällt in Python praktisch kein sogenannter "Overhead" an, das heißt, es müssen keine überflüssigen Befehle geschrieben werden. Skripte fallen dadurch kürzer aus und sind auch von Laien für einfache Aufgaben schnell nutzbar.

5.1.2 Funktionale Programmierung

Der Grundgedanke des funktionalen Programmierstils ist die Funktion. Dabei erfolgt zu einem gegebenen Problem die Lösungsbeschreibung als Geflecht von Funktionen. Standardmäßig eingesetzt wird dabei das mächtige Mittel der Rekursion. Die funktionale Programmierung beruht auf einer mathematischen Definition, dem Lambda-Kalkül. Variablen repräsentieren nicht den Inhalt einer Speicherzelle, sondern sind an genau einen Wert *gebunden*. Klassische Beispiele für die funktionale Programmierung sind SML und Lisp. Elemente eines imperativen Programmierstils sind in einem gewissen Maße auch in diesen Sprachen enthalten.

Alle wichtigen Ideen der funktionalen Programmierung lassen sich in Python einsetzen. Am wichtigsten ist hier die Rekursion:

```
# Rekursive Berechnung der Fakultaet einer ganzen Zahl.
def fakultaet_rekursiv(n):
  assert n>=0
  if n>1:
    return n*fakultaet_rekursiv(n-1)
  else:
    return 1
```

Eine weitere Eigenschaft vieler (aber nicht aller) funktionaler Programmiersprachen ist die Polymorphie. Eine polymorphe Funktion arbeitet nicht auf einem bestimmten Datentyp, sondern kann mit beliebigen sinnvollen Eingaben umgehen, sofern alle verwendeten Unterfunktionen ebenfalls die Eingabe verarbeiten können.

```
>>> def verdopple(x):
...   return x*2
>>> verdopple(5)
10
>>> verdopple(3.141)
6.282
>>> verdopple("Hallo! ")
'Hallo! Hallo! '
```

In Python sind alle Argumente typfrei, man benötigt keine Unterscheidungen, wie sie beispielsweise in C++ zur Überladung von Funktionen nötig sind.

Mit dem sogenannten λ-Ausdruck lassen sich in Python anonyme Funktionsobjekte definieren. Dieser Begriff stammt von Church aus dem *Lamba-Kalkül* bei der Definition funktionaler Sprachen.

```
>>> dopple=lambda x: 2*x
>>> dopple
<function <lambda> at 0x010B35E8>
>>> dopple(11)
22
```

Wie eine Funktion hat ein λ-Ausdruck Parameter (hier `x`) und einen Funktionskörper (hier `2*x`).

In allen funktionalen Sprachen gibt es wenigstens die drei Standard-Operationen `apply`, `map` und `reduce`. Auch in Python sind diese Basisoperationen definiert, zusätzlich noch die Funktion `filter`.

`apply` wendet eine Funktion auf einen Parameter an:

```
>>> apply(dopple,[44])
88
```

`map` arbeitet ähnlich, es wendet eine Funktion auf eine Folge von Parametern an:

```
>>> map(dopple,[42,3.141,"Hallo"])
[84, 6.282, 'HalloHallo']
```

`reduce` arbeitet auf zweistelligen Funktionen:

```
>>> reduce(lambda x,y: x+y, [1,2,3,4,5])
15
```

Die Funktion ist hier `x+y`; diese wird zuerst auf die ersten beiden Werte der Liste angewendet. Der Reihe nach wird dann das Ergebnis mit dem jeweils nächsten Element verknüpft.

`filter` wendet auf jeden Parameter eine boolesche Funktion an und liefert nur die Argumente zurück, für die das Ergebnis wahr ist:

```
>>> filter(lambda x: x<10, [5,9,15,7,42])
[5, 9, 7]
```

Damit stehen in Python bereits mit integrierten Mitteln typische Merkmale einer funktionalen Sprache zur Verfügung. Die Syntax bei der Funktionsdefinition ähnelt der der funktionalen Programmiersprache SML.

Eine Erweiterung der funktionalen Möglichkeiten bietet das *Xoltar Toolkit* [Keller01]. Es stellt weitere funktionale Elemente wie Funktionen für Komposition und die sogenannten *Curry-Funktionen* sowie eifrige und verzögerte Auswertung bereit. Die folgenden Beispiele stellen nur einen kleinen Teil der Möglichkeiten des *Xoltar-Toolkits* dar.

Curry-Funktionen

Die Curry-Funktion ersetzt Parameter einer vorgegebenen Funktion durch Standardwerte:

```
def addiere2Werte(x, y):
  return x + y
inkrement = curry(addiere2Werte, 1)
>>> inkrement(5)
6
```

Komposition

Die Komposition entspricht dem mathematischen ∘-Operator. $f \circ g$ bedeutet: f wird angewandt auf das Ergebnis von g.

```
def f(x):
  return x*2
def g(x):
  return x+1
h=compose(f,g)
```

`h(9)` entspricht nun `f(g(9))`, also `f(10)`, und liefert `20`.

Einen besonderen Stellenwert nimmt die *Lazy Evaluation* (verzögerte Auswertung) ein. Verzögerte Ausdrücke werden erst berechnet, wenn sie im Programm benutzt werden. Viele funktionale Programmiersprachen wie SML bieten diese Möglichkeit. Auf diese Weise können auch unendliche Mengen definiert und auf diese Mengen Operationen angewandt werden, wie im Folgenden gezeigt wird. Die Definition einer (unendlichen) Liste aller Fakultätszahlen ist beispielsweise folgendermaßen möglich:

```
def fakultaet(index, seq):
  if index == 0:
    return 1
  else:
     return seq[index -1] * index
fakultaeten = LazyTuple(itemFunc = fakultaet)
>>> print fakultaeten[2:4]
LazySlice (2, 6, 24, ..)
>>> print fakultaeten[5]
120
>>> print lazyfilter(lambda x: x<100,fakultaeten)
LazyTuple (1, 1, 2, 6, 24)
>>> print len(fakultaeten)
RuntimeError: Non-terminating structure, cannot evaluate len().
```

Die Liste `fakultaeten` ist unendlich und wird verzögert ausgewertet. Erst wenn auf ein einzelnes Element der Liste zugegriffen wird, wird es erzeugt.

Pizza-Fallstudie:

Die direkte Umsetzung des Pizza-Beispiels [BS97] für das funktionale Paradigma in Python ist nicht ohne Zuhilfenahme einiger Änderungen möglich. Brennwald macht intensiv Gebrauch von Pattern-Matching, welches in der aktuellen Python-Version noch nicht innerhalb der Syntax realisiert ist. Als Äquivalent bietet Python die Umsetzung mittels Tupel-unpacking zusammen mit Fallunterscheidungen an, auch wenn dies syntaktisch weniger schön erscheint. Eine weitere Alternative ist die Klasse *attempt* aus dem *Xoltar Toolkit*, aber auch diese Variante stellt eher eine Notlösung dar.

Weiterhin werden in Brennwalds Beispiel aufgezählte Typen eingesetzt, welche in Python nur über Klassen simulierbar sind. Damit ist zumindest hier ein Bruch zum rein funktionalen Programmieren nötig. Eine rein funktionale Python-Implementierung unter Einsatz von Hashes befindet sich auf der CD zu dieser Arbeit.

Ergebnis:

Zusammen mit dem *Xoltar Toolkit* bietet Python alle wesentlichen Möglichkeiten der funktionalen Programmierung. Somit eignet sich Python zum Vermitteln funktionaler Programmierideen. Es können dabei jederzeit prozedurale Elemente mitgenutzt werden, was in rein funktionalen Sprachen oft nicht möglich ist.

Pattern-Matching und aufgezählte Typen sind Merkmale einiger funktionaler Sprachen, die in Python nicht direkt umgesetzt werden können. Alternative Implementierungen sind jedoch möglich, wenn auch nicht immer bruchfrei.

Eine interessante alternative Möglichkeit zur funktionalen Programmierung mit Python ist der mit nur wenigen Seiten Quelltext geschriebene Lisp-Interpreter von Chris Meyers [Meyers01].

Vorteile:

- Alle Grundlagen funktionaler Sprachen können mit Python eingesetzt werden
- Polymorphie wie in SML
- Integration von Listen und Tupeln übertrifft die Möglichkeiten von SML
- Gute Einbindung der funktionalen Elemente in die Python-Syntax
- Sequenzen sind wie Listen nutzbar
- Verzögerte Auswertung kann eingesetzt werden

Nachteile:

- Derzeit ist kein Pattern-Matching möglich
- Zur Simulation statischer Typen ist Rückgriff auf Klassen erforderlich

Literatur:

- *The Xoltar Toolkit* [Keller01]
- *Functional Programming in Python* [Mertz01]

5.1.3 Objektorientierte Programmierung

Der objektorientierte Programmierstil beruht auf der Vorstellung, dass der zu modellierende Weltausschnitt durch eine Menge von Objekten dargestellt werden kann, die miteinander über Nachrichten kommunizieren. Die Objekte haben einen Zustand, dargestellt durch die konkreten Attributwerte, und werden von außen mittels Nachrichten aktiviert. Dadurch können sowohl der Zustand des Objekts geändert werden, als auch andere Objekte benachrichtigt werden. Das Methodenspektrum eines Objekts bestimmt, wie auf Nachrichten, die das Objekt erhält, reagiert wird. Ein Zugriff auf die Objektdaten ist ausschließlich über die Objektmethoden möglich[23].

[23] Das Senden einer Nachricht an ein Objekt ist äquivalent zum Methodenaufruf. Es handelt sich um unterschiedliche Sichtweisen der objektorientierten Modellierung.

Mit Hilfe sogenannter Vererbungsstrukturen stellen Klassen (als Vorlagen – Baupläne für konkrete Objekte) in Hierarchieebenen, Attribute und Methoden für konkrete Objekte, welche dann instanziiert[24] werden, zur Verfügung. Der objektorientierte Stil ist verbunden mit Konzepten wie Modularisierung, Geheimnisprinzip und abstrakter Datentyp.

Typische objektorientierte Programmiersprachen sind Simula, Smalltalk, Eiffel, C++, Java und Python.

Python ist als moderne, objektorientierte Sprache konzipiert worden. In den folgenden Abschnitten werden kurz die Besonderheiten objektorientierter Programmierung mit Python vorgestellt. Lehrmaterial zur objektorientierten Programmierung mit Python ist bereits genügend vorhanden (beispielsweise [LF02]).

Klassen und Objekte

Objekte einer Klasse haben Eigenschaften, die sogenannten Attribute. Des Weiteren hat ein Objekt Methoden; diese sind an das Objekt gebundene Funktionen. Objekte können als Abstraktion real existierender Dinge gesehen werden. Beispielsweise wird im Folgenden ein Tier mit den Eigenschaften *Name* und *Alter* betrachtet:

```
class Tier:
  def __init__(self, name):
    self.name = name
    self.alter = 1
  def setze_alter(self, wert):
    self.alter = wert
  def gib_Daten_aus(self):
    print "Mein Name ist ",self.name
    print "Ich bin", self.alter, "Jahr(e) alt."
```

Hier wird eine Klasse *Tier* mit den Methoden `__init__`, `setze_alter` und `gib_Daten_aus` sowie den Attributen `name` und `alter` definiert. Der Ausdruck `self` ist das Objekt selbst, es wird in Python bei Methoden immer als erster Parameter angegeben. Um innerhalb einer Methode das Attribut `x` eines Objekts auf den Wert `w` zu setzen oder neu zu definieren, wird der Ausdruck `self.x=w` benutzt. Umgekehrt wird einer Variablen `v` durch `v=self.x` der Inhalt des Attributes `x` zugewiesen. Die besondere Methode `__init__` eines Objektes ist ein optionaler Konstruktor, der bei der Erzeugung einer Instanz des Objektes aufgerufen wird. Es bietet sich an, hier alle Attribute eines Objektes zu definieren und mit Standardwerten zu belegen.

Hier liegt eine weitere Besonderheit von Python: Neue Attribute müssen nicht wie beispielsweise in Java extra deklariert werden, sondern werden direkt bei der ersten Zuweisung erzeugt. Dahinter steht das Konzept der *dynamischen Attributierung*. Zur Laufzeit sind jederzeit neue Attribute definierbar, ohne an der Definition des Objektes Änderungen vornehmen zu müssen. Bei der schnellen Softwareentwicklung spart der Entwickler auf diese Weise Zeit und viele Zeilen Quelltext. Es ist in Python nicht nötig (bzw. auch nicht möglich), den Typ einzelner Attribute festzulegen. Genauso wie bei Variablen gilt das Prinzip der *dynamischen Typisierung*, das heißt, jedes Attribut ist Platzhalter für beliebige Typen.

Um ein Exemplar eines Tieres zu erzeugen, wird die Klasse wie eine Funktion mit den Parametern der Funktion `__init__` aufgerufen:

[24] sprich: nach diesen Bauplänen angefertigt

```
>>> MeinTier=Tier("Mikey")
```

Methoden und Attribute eines Objektes werden in der Notation *Objektname.Methode* aufgerufen:

```
>>> MeinTier.setze_alter(13)
>>> MeinTier.gib_Daten_aus()
Mein Name ist Mikey
Ich bin 13 Jahr(e) alt.
```

Vererbung

Ein wichtiges Konzept der objektorientierten Programmierung ist die Vererbung. Oft stellt man fest, dass eine Menge von Objekten in etlichen Attributen und Methoden gleich ist und sich nur in speziellen Details unterscheidet. Es können dann ausgehend von einer Basisklasse Spezialisierungen definiert werden, wobei die spezialisierte Klasse alle Eigenschaften der Basisklasse erbt. Innerhalb der Spezialisierung können neue Attribute und Methoden definiert, und vorhandene neu definiert werden. Beispielsweise können `Hund` und `Katze` als Spezialisierung von `Tier` angesehen werden, und es soll für jedes Tier eine eigene Methode `gib_laut` festgelegt werden:

```
class Hund(Tier):
  def gib_laut(self):
    print "Wuff"
class Katze(Tier):
  def gib_laut(self):
    print "Miau"
meinHund = Hund("Bello")
meineKatze = Katze("Mieze")
```

`Hund` und `Katze` erben alle Attribute und Methoden der Klasse `Tier` und werden um die Methode `gib_laut` erweitert.

Aufrufe von Methoden sind in Python, im Gegensatz beispielsweise zu C++, immer virtuell. Sei `Vater` eine Basisklasse und `Sohn` erbt Methoden von `Vater`. Wird dann eine Methode einer Instanz von `Sohn` aufgerufen, wird immer die Methode des Sohnes benutzt, sofern nicht explizit die Methode des Vaters aufgerufen wird. Python-Programme sind nicht wie beispielsweise C++ statisch kompiliert. Erst zur Laufzeit wird der Aufruf von Methoden geprüft und beispielsweise bei Aufruf einer undefinierten Methode der Ausnahmefehler `NameError` verursacht.

Eine Klasse kann auch von mehreren Basisklassen abgeleitet werden, es wird dann von Mehrfachvererbung gesprochen. Das Diagramm 11 stellt die Situation dar.

`Flugtier` mit der Methode `fliegen` und `Lauftier` mit der Methode `laufen` sind Spezialisierungen der Klasse `Tier`. Die Klasse `Ente` erbt die Eigenschaften beider Klassen.

Python unterstützt auch derartige Konstruktionen:

```
class Ente(Flugtier, Lauftier):
  def gib_laut(self):
    print "Quaak"
>>> meineEnte.fliegen()
```

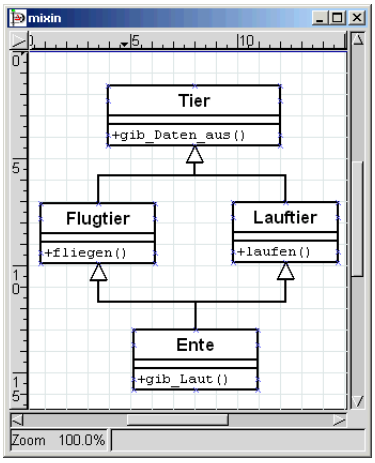


Abbildung 11: *Mehrfachvererbung*

Oft gibt es Situationen, in denen Mehrfachvererbung oder Mixin-Klassen[25] vorteilhaft sind. Mixin-Klassen ergeben sich häufig bei der intuitiven Modellierung von Klassenhierarchien. Während Mehrfachvererbung in Compilersprachen oft zu Problemen führt, kann Python zur Laufzeit mögliche Konflikte lösen. Wenn Mehrfachvererbung benutzt wird, nimmt Python das erste Auftreten eines Attributes oder einer Methode bei einer Breitensuche durch den Baum aller Oberklassen. Bei Implementierungen in vielen Sprachen wie beispielsweise C++ muss hier auf ein kompliziertes Regelwerk ausgewichen werden, was einen deutlich erhöhten Entwicklungsaufwand erfordert. Python kann in diesem Fall den Aufwand reduzieren und die Softwareentwicklung deutlich beschleunigen.

Speicherverwaltung

In Python ist es nicht nötig, sich mit der Speicherverwaltung zu beschäftigen. Python erkennt selbst, ob ein Objekt nicht mehr benötigt wird, indem es intern einen Referenzzähler führt. Ohne dass der Entwickler oder Anwender etwas davon merkt, wird regelmäßig eine sogenannte *Garbage Collection* (=Abfallbeseitigung) durchgeführt. Objekte, auf die es keine Referenzen mehr gibt, werden dabei aus dem Speicher beseitigt. Die Speicherverwaltung arbeitet verglichen mit Java schnell und macht sich auch bei speicherintensiven Anwendungen nicht durch lange Programmpausen, wie sie etwa in Java oft zu beobachten sind, bemerkbar. Probleme bereitete bis zu Python 2.1 die Erkennung zirkulärer Referenzen; dieses Problem wurde mit Python 2.2 beseitigt und soll in zukünftigen Versionen noch weiter optimiert werden.

Polymorphie

Die Polymorphie ist eine der Stärken der objektorientierten Programmierung. Unabhängig vom Typ eines Objektes kann eine gemeinsame Methode aufgerufen werden. In klassischen, prozeduralen Programmiersprachen sind hierzu umfangreiche Fallunterscheidungen nötig.

Genau wie bei den Funktionen sind auch die Parameter von Methoden polymorph. Ausprägungen aller vorhandenen Datentypen, auch Funktionen und Objekte, können als Parameter eingesetzt werden.

[25] Mixin-Klassen erweitern eine Klasse um zusätzliche Funktionalität.

Das Geheimnisprinzip

Hinter dem Geheimnisprinzip der objektorientierten Programmierung steckt eine einfache Idee: Attribute und nur intern genutzte Methoden einer Klasse sollen für die Außenwelt unsichtbar sein. In Python sind zunächst alle Attribute und Methoden öffentlich. Auch außerhalb einer Methode kann jederzeit lesend und schreibend auf alle Attribute zugegriffen werden. Darüber hinaus erlaubt das Prinzip der dynamischen Attributierung auch jederzeit die Definition neuer Attribute.

```
>>> print meinHund.name
Bello
>>> meinHund.name="Wuffi"
>>> meinHund.gewicht=25
```

Der Vorteil von öffentlichen Attributen liegt in der Einsparung spezieller Methoden zum Lesen und Schreiben der Attribute. Bei Testläufen im Interpreter oder Debugger kann das Programm jederzeit angehalten werden. Der Programmierer kann dann einzelne Objektattribute lesen, ändern oder neu definieren, und den Programmablauf anschließend fortsetzen. Öffentliche Attribute bergen aber auch Risiken, etwa dass unbeabsichtigt Attribute geändert werden können. Als Ausweg lassen sich deshalb einzelne Attribute und Methoden als geheim deklarieren. Alle Attribute, deren Bezeichner mit einem doppelten Unterstrich beginnt, sind private Klassenobjekte und außerhalb der Klasse nicht erreichbar.

Ergebnis:

Python bietet alle Merkmale einer modernen und objektorientierten Sprache, wobei auf strenge Sicherheitsmechanismen zugunsten schnellerer Programmierung verzichtet wird.

Vorteile:

- Integrierte Speicherverwaltung (Garbage Collection)
- Volle Polymorphie
- Dynamisch typisierte Attribute
- Attribute müssen nicht definiert werden, Definition auch zur Laufzeit möglich
- Hierarchiesuche bei Mehrfachvererbung

Nachteile:

- Parameter `self` muss bei allen Methoden angegeben werden
- Unbeabsichtigte Attributsdefinition ist problematisch

5.1.4 Prädikative und wissensbasierte Programmierung

Der *prädikative* (auch *logisch* oder *deklarativ* genannte) Programmierstil basiert auf der folgenden Idee: Der Programmierer deklariert die Fakten und Eigenschaften des Problems, für das eine Lösung gesucht wird. Diese Information wird von der Inferenzkomponente[26] des Systems dazu benutzt, um eine Lösung zu finden. In der prädikativen Programmierung erfolgt die Problembeschreibung in einem logischen Formalismus wie beispielsweise dem Prädikatenkalkül. Notiert werden Fakten und Regeln. Das Variablenkonzept ist in der prädikativen Programmierung ähnlich dem der funktionalen Programmierung: Variablen sind an einen Wert gebunden, Zuweisungen finden nicht statt. Die bekannteste prädikative Programmiersprache ist Prolog. Eine einfache Implementierung von Prolog in Python hat kürzlich Chris Meyers [Meyers02] vorgestellt. Python bietet mit seinen Basisbibliotheken keine Möglichkeiten der prädikativen Programmierung. Es gibt aber bereits Bibliotheken, die deklaratives Programmieren auch mit Python ermöglichen:

Holmes [LRLW01] ist ein in Python implementiertes Expertensystem. Es wird als Modul importiert und kann auf zweierlei Arten verwendet werden. Primär bietet Holmes einen Interpreter mit einer eigenen Eingabekonsole. Es lassen sich interaktiv mit einer verständlichen Syntax Fakten und Regeln definieren und anschließend Abfragen durchführen. Sollen größere Mengen an Fakten verarbeitet werden, können diese in einer Datei vorbereitet und eingelesen werden. Holmes wird in dieser Form der Forderung nach einer eigenständigen prädikativen Sprache gerecht. Soll eine Integration in Python-Programme erfolgen, können alternativ sämtliche Anweisungen und Abfragen statt über die Konsole über äquivalente Funktionen erfolgen. Eine direkte Integration in die Python-Syntax ist nicht vorgesehen. Holmes ist geeignet, Grundlagen und Ideen der prädikativen Softwareentwicklung einzuführen und innerhalb von Python zu nutzen; jedoch sind die Möglichkeiten der Anwendung nur knapp dokumentiert, und es sind nur wenige Beispiele vorhanden. Seit drei Jahren ist keine Weiterentwicklung erfolgt, daher sind hier auch keine Verbesserungen zu erwarten.

Logilab Constraint Package [Logilab02] ist ein vollständig in Python implementiertes und nutzbares Modul zur wissensbasierten Modellierung von Problemen. Die Fakten werden in Form von Variablen und Bedingungen zu einer Matrix zusammengefasst und an die Inferenzkomponente übergeben. Eine kurze Dokumentation und Beispielcode ist vorhanden. Das Modul ist vollständig in Python geschrieben und kommt, im Gegensatz zu den anderen vorgestellten Modulen, ohne Simulation oder Integration einer anderen Sprache wie Prolog aus. Die Syntax ist jedoch kompliziert und zur Einführung in prädikative Entwicklung ungeeignet.

PyLog [Delord02] ist eine direkt in Python entwickelte und nutzbare Bibliothek. Sie unterstützt das Erzeugen logischer Terme, Variablen und Atome und kann diese später vereinfachen und auflösen. Terme und Variablen sind als Klasse dargestellt, so dass direkt Ausdrücke in logischer Notation geschrieben werden können. *PyLog* fügt sich gut in das objektorientierte Python-Konzept ein; ebenso erscheint die Notation geeignet, einfache logische Probleme zu formulieren. Des Weiteren kann *PyLog* verwendet werden, um Prolog-Programme in ein ausführbares Python-Skript zu übersetzen. Die Dokumentation der Software beschränkt sich derzeit allerdings auf zwei einfache Beispiele, ein wenig Zeit zum Experimentieren muss der Nutzer daher investieren. Im fachdidaktischen Einsatz ist die formelartige Notation jedoch problematisch, daher ist *PyLog* hier nicht empfehlenswert.

Semnet ist eine kleines, aber zweckmäßiges Modul zur Darstellung von Wissen als *Semanti-*

[26] Inferenz = logische Schlussfolgerung

sches Netzwerk. Über die drei Klassen *Fact*, *Entity* und *Relation* wird dem System Wissen vermittelt, und es kann anschließend über Funktionen befragt werden. Positiv fällt an *Semnet* die einfache Syntax auf.

Neural Integrator gestattet das Erstellen neuronaler[27] Netze. Es besteht aus einer integrierten Entwicklungsumgebung, in der sich Zellen zu einem neuronalen Netz kombinieren lassen. Dabei lassen sich eigene Zelltypen mittels C++ oder Python definieren und unterschiedliche Modelle neuronaler Netze nutzen. Die für Berechnungen erforderlichen Programmteile sind in C geschrieben, daher wird eine gute Rechenleistung erzielt.

In der Entwicklung befinden sich weitere Projekte: *Formula* ist ein Projekt zur prädikativen Programmierung mit Python, allerdings liegen derzeit noch keine öffentlichen Versionen vor. *CLIPS-Wrapper* [Clips01] stellt eine Verbindung zu dem von der NASA entwickelten Expertensystem *CLIPS* her. Problematisch ist hier die Fehlersuche, da sich Fehlermeldungen von *CLIPS* in Python nicht auswerten lassen.

Ergebnis:

Die Möglichkeiten direkter prädikativer Programmierung mit Python sind mit den aktuellen Bibliotheken noch eingeschränkt. Keines der Pakete erlaubt eine gut lesbare Notation direkt im Python-Interpreter. Zum fachdidaktischen Einsatz eignet sich *Holmes* als einziges Paket mit einer angemessenen Syntax, welche hier über einen eigenen Interpreter erreicht wird, gleichzeitig bietet es Integrationsmöglichkeiten in Python-Programme.

Erfahrenen Entwicklern ist die Integration prädikativer Konzepte in bestehende prozedurale Skripte eingeschränkt mit *PyLog* oder dem *Logilab Constraint Package* möglich, wobei beide Projekte erst am Anfang ihrer Entwicklung stehen. Alle derzeit erhältlichen Module befinden sich in frühen oder experimentellen Entwicklungsstadien und sind mangels Dokumentationen und Beispielen nur bedingt für praktischen Einsatz geeignet. Die Entwickler und die Python-Gemeinschaft arbeiten derzeit an der Weiterentwicklung vorhandener Module wie auch an einigen neuen Projekten, so dass die derzeitigen Mängel möglicherweise bald behoben sein werden. Derzeit gestattet nur *Holmes* eine syntaktisch saubere prädikative Entwicklung.

Gut hingegen ist die Anbindung an externe Datenbanken durch zahlreiche Bibliotheken, wie bereits im Abschnitt zu Datenbanken in 4.10 gezeigt wurde. Ebenso ist das Erstellen neuronaler Netze mit den *Neural Integrator* auf praxisgerechte Weise möglich.

5.1.5 Zusammenfassung – Paradigmen mit Python

Obwohl heute das Gros der Softwareentwicklung nach dem objektorientierten Paradigma erfolgt, sollten die anderen Paradigmen nicht vernächlässigt werden. Zahlreiche Klassen von Problemen können funktional oder prädikativ angemessener formuliert werden.

Anzustreben sind Programmiersprachen, die Lösungen für verschiedene der hier diskutierten Probleme bieten. Von derart breit angelegten Sprachen werden dann in Projekten konkrete Profile angewandt, das heisst, durch Konventionen und darauf abgestimmte Werkzeuge wird die Mächtigkeit der Sprache wieder eingeschränkt. [JH02]

Python bietet die Möglichkeit, alle Paradigmen der Softwareentwicklung unter dem Dach einer Programmiersprache zu nutzen. Damit wird Python der Forderung nach einer breit angelegten

[27] Die Bezeichungen *neurale Netze* und *neuronale Netze* werde in der Literatur oft gleichbedeutend verwendet.

Sprache weitgehend gerecht. Nur die prädikativen Entwicklungsmöglichkeiten sind derzeit, verglichen mit rein prädikativen Sprachen wie Prolog, eingeschränkt. Sie lassen sich aber bereits jetzt in prozedurale und objektorientierte Programmteile einbetten.

Als wichtigste Vorteile der Integration aller Paradigmen in eine Programmiersprache lassen sich eine durchgehend einheitliche Arbeitsumgebung und einfacher Datenaustausch nennen. Bei Nutzung unterschiedlicher Sprachen für die jeweiligen Paradigmen ist der Datenaustausch zwischen den verschiedenen Entwicklungsumgebungen problematisch, da geeignete Schnittstellen gefunden und erlernt werden müssen.

Die Idee, alle Paradigmen innerhalb einer Sprache zu nutzen, birgt aber auch Probleme, was insbesondere fachdidaktische Aspekte betrifft. Es besteht die Gefahr, dass die einzelnen Paradigmen nicht als eigenständige Konzepte erkannt werden. Diese kann zur häufigen Mischung unterschiedlicher Paradigmen und auch zum Rückfall auf gewohnte Entwicklungsmethoden führen. Es müssen daher geeignete Methoden gefunden werden, alle Paradigmen zu motivieren und eine klare Trennung der Paradigmen zu erreichen. Ziel muss dabei sein, die Einsatzgebiete sowie die Vor- und Nachteile der Paradigmen zu erkennen. Hier besteht noch Forschungsbedarf: Es gibt derzeit zu wenig Erfahrungen, ob die Integration der Paradigmen innerhalb einer Sprache im fachdidaktischen Umfeld problemlos möglich ist.

5.2 Umfrage zur Lesbarkeit von Programmiersprachen

Lesbarkeit in Bezug auf eine Programmiersprache ist ein Begriff, der sich nur schwierig definieren lässt. Jeder Mensch hat eigene Vorstellungen, wann er einen Quelltext als gut oder weniger gut lesbar empfindet. Des Weiteren ist die Lesbarkeit vorhandenen Codes stark vom Programmierer anhängig, denn in nahezu jeder Sprache lässt sich unlesbarer Code erstellen. Trotzdem gibt es offenbar gemeinsame Faktoren, welche die allgemeine Lesbarkeit günstig beeinflussen. Um einen allgemeinen Trend erkennen zu können, wurden im Rahmen dieser Arbeit über 30 Studenten und Programmierer in Universitäten, Newsgroups und Unternehmen zur Lesbarkeit von Programmiersprachen befragt. Dabei wurden zur näheren Erläuterung des Begriffs *Lesbarkeit* die folgenden Leitfragen formuliert:

1. Kann jemand, der die Sprache nie zuvor gesehen hat, verstehen, was ein Algorithmus macht, ohne in das Referenzbuch zu schauen?
2. Kann ein Entwickler ein Programm schreiben und es nach sechs Monaten noch verstehen?
3. Kann eine Sprache benutzt werden, um Grundlagen der Programmierung zu erklären und die Arbeitsweise einfacher Algorithmen zu zeigen?
4. Wie ist die allgemeine Meinung über die Sprache?

Zur Auswahl standen folgende Sprachen, wobei auch weitere Sprachen genannt werden konnten (zusätzlich genannte Sprachen in Klammern):

ABC, Ada, APL, Beta, C++, COBOL, Delphi, Eiffel, Forth, FORTRAN, Java, Lisp, Modula, Pascal, Perl, Prolog, Python, SML, Scheme, Smalltalk, Tcl, Visual Basic, andere (Algol60, Basic, C, Haskell, Ruby, Ocaml).

Die Bewertung erfolgte auf einer Skala von 5 (für sehr gut lesbar) bis 1 (sehr schlecht lesbar oder unlesbar). Ziel der Befragung ist die Exploration von Einstellungen bezüglich des unscharfen

Begriffs der Lesbarkeit im Vergleich verschiedener Programmiersprachen. Zur Quantifizierung wurden die Standardabweichung und das geometrische Mittel der Ergebnisse ermittelt. Die Streuung innerhalb der Einzelergebisse ist gering und zeigt, dass nahezu alle befragten Entwickler eine vergleichbare Meinung zu den einzelnen Sprachen haben. Die Einzelergebnisse der Umfrage befinden sich im Anhang A.8.

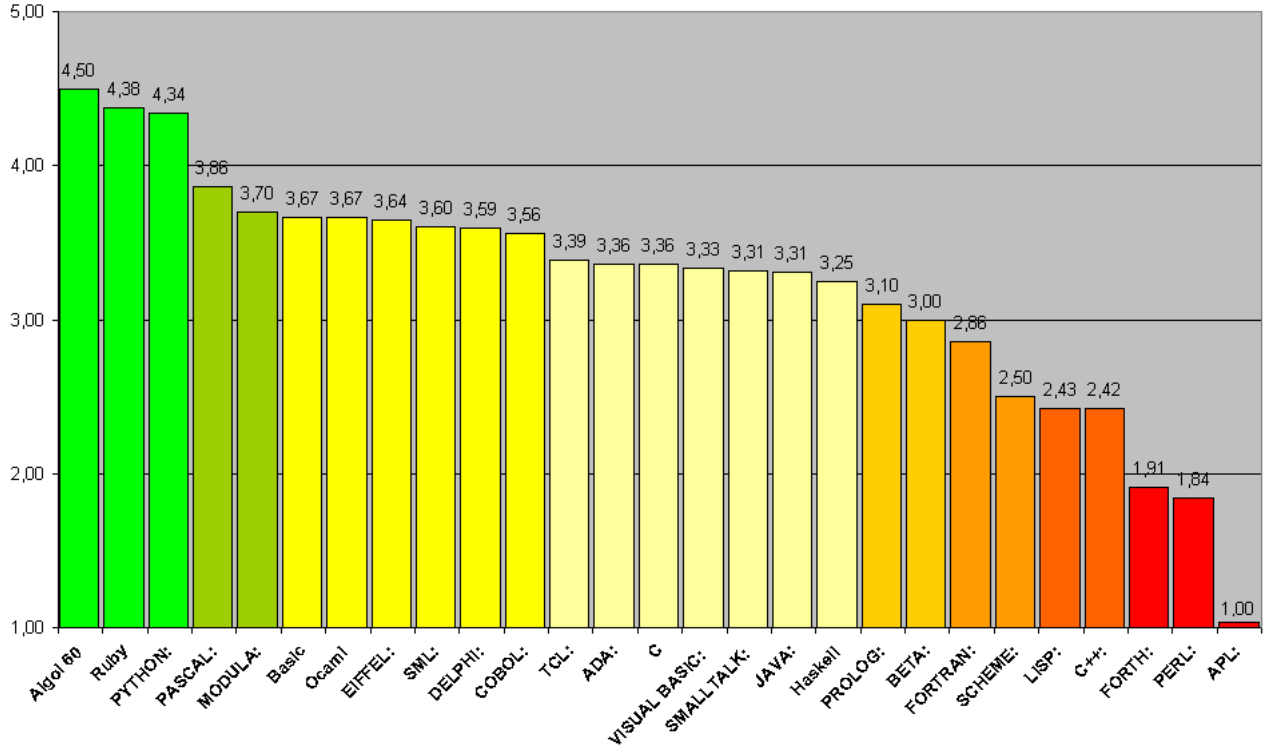


Abbildung 12: *Ergebnisse der explorativen Befragung zur Lesbarkeit*

Eine Übersicht der Umfrageergebnisse ist in Abbildung 12 dokumentiert. Demnach gelten Algol, Modula, Pascal, Python, Ruby und SML als besonders gut lesbar, während APL und Perl als schwierig zu lesen beurteilt wurden. Als wünschenswerte Anforderungen an eine Sprache wurde von einigen Befragten ergänzt:

- Die Schlüsselwörter einer Sprache sollten kurz, aber in ihrer Bedeutung leicht verständlich sein. Die Programmiersprache APL zeichnet eine äußerst kompakte Schreibweise aus, die zur Konsequenz hat, dass ca. 60 ausgewiesene Symbole zur Umsetzung benötigt werden. Sind die Symbole nicht bekannt, bleibt auch ein gut strukturiertes Programm so unlesbar wie Hieroglyphen. Pascal und Python verwenden englische Kurzformen wie `if`, `else`, `for`, `while`, womit eine Nähe zu einer allgemein bekannten natürlichen Sprache besteht.

- Parameterübergaben an Funktionen sind den meisten Menschen aus der Mathematik bekannt. Pascal, Eiffel, SML und Python haben hier eine vergleichbare Notation, während Perl eine ungewöhnliche Form der namenlosen Parameterübergabe einsetzt. Die Wiedererkennung einer bekannten Notation kann nach Ansicht der Befragten die Lesbarkeit verbessern. Die Befragten gaben an, dass die Anzahl der Argumente variabel sein sollte und die Argumente eindeutige Namen haben sollten. Funktionen sollten dabei möglichst

lokal arbeiten und unabhängig von globalen Zuständen sein, wie es aus der Mathematik bekannt ist.

- Zur leichteren Erkennung der Programmstruktur werden optische Gestaltungsmittel wie eingerückte Blöcke und Kommentarzeilen eingesetzt. In Python ist Einrücken zwingend erforderlich, wodurch die Lesbarkeit im Allgemeinen verbessert wird (siehe 4.5).

- Die Sprache sollte einheitliche und eindeutige Notationsformen bieten und möglichst ohne überraschende Sonderfälle und Ausnahmen arbeiten. Diese Eigenschaften werden unter dem Begriff *Orthogonalität* im nächsten Abschnitt behandelt.

5.3 Orthogonalität der Sprache

Weniger ist manchmal mehr.

Ludwig Mies van der Rohe (1886 – 1969),
Architekt

Die Orthogonalität einer Programmiersprache kann (in Anlehnung an [Schwill02]) durch die folgenden Eigenschaften charakterisiert werden:

1. Die Sprache soll sich beschränken auf das nötigste Minimum für den jeweiligen Einsatzzweck. Überflüssige Anweisungen sind zu vermeiden, um so einfach wie möglich ein Ziel erreichen zu können.

2. Syntax und Notation der Sprache sollen frei von Widersprüchen sein.

Eine differenzierte Darstellung der syntaktischen Struktur der Programmiersprache Python wurde bereits in Kapitel 4 vorgestellt. Im Folgenden werden die dort dokumentierten Ergebnisse auf das Kriterium der Orthogonalität bezogen.

ad 1: So einfach wie möglich zum Ziel

Eine Programmiersprache sollte möglichst auf unnötige Anweisungen und Deklarationen verzichten können. C und C++ sind Beispiele für Sprachen, in denen ein erheblicher "Ballast" an zusätzlichen und oft überflüssig erscheinenden Deklarationen nötig ist. Um Schülern, die eine erste Modellierung implementieren sollen, die selbständige Erarbeitung zu gestatten, müssen vor diesem Schritt die dazu unabdingbaren programmiersprachlichen Elemente bekannt sein. Gerade im Anfangsunterricht ist es nicht einfach, die Bedeutung der Schlüsselworte `void`, `public`, `static` zu fundamentieren. Ebenso sind bei rein prozeduraler Nutzung objektorientierter Sprachen oft Initialisierungen von Objekten nötig, die Schülern zunächst unverständlich sind. Das *Hello-World*-Beispiel (vgl. Abschnitt 5.1.1) in Java zeigt diese Problematik, in vielen anderen Sprachen sieht es ähnlich aus.

Python gestattet einfache und interaktive Programmierung ohne diese redundanten Anteile. Die Anweisungen `print` und `input` sind direkt ohne weitere Zusätze nutzbar. Auch die Definition von Unterprogrammen sowie Funktionen ähnelt der mathematischen Definition einer Funktion und kommt ohne Zusätze von Typinformation und Geltungsbereichen aus.

Vollkommen frei von überflüssigen Notationen ist Python allerdings auch nicht. Manche Entwickler kritisieren die erzwungene Angabe von `self` als ersten Parameter im Kopf der Methodendefinition. Diese Angabe ist immer erforderlich, obwohl sie eindeutig ist und keine praktische Funktion erfüllt. Es soll dadurch kenntlich gemacht werden, dass das Objekt selbst innerhalb der Methode lokal zur Verfügung steht. Aus fachdidaktischer Sicht erscheint diese Entscheidung sinnvoll, auch wenn sie zu etwas mehr Schreibarbeit führt.

Ergebnis:

Python ist, bezogen auf prozedurale, objektorientierte und funktionale Entwicklung, weitgehend frei von überflüssigen Deklarationen. Lediglich der Parameter `self` bei Methodendefinitionen und die bereits erwähnte Doppelpunkt-Notation bei Schleifen und Fallunterscheidungen sind unnötige Spracheigenschaften, die einem Entwickler regelmäßig begegnen.

ad 2: Orthogonalität bezüglich Syntax und Notation:

Programmiersprachen sollten frei von Widersprüchen in Syntax und Semantik sein. Vor allem in Lehr- oder Lernprozessen führen Inkonsistenzen zu unnötigen Lernwiderständen. Es gibt zahlreiche Beispiele für nicht orthogonale Spracheigenschaften anderer Programmiersprachen:

- C: Makros definieren eine Subsprache, die unter Umständen unverständlich ist
- Perl: Klammern werden für Gruppierungen, Listen und Funktionsaufrufe verwendet
- Perl: Geschweifte Klammern definieren Blöcke und Hashes
- Perl: `if ... then ... else` liefert Werte, ist aber kein Ausdruck
- Pascal: Arrays sind nicht als Parameter nutzbar
- Pascal: `writeln` und `readln` akzeptieren offene Parameterlisten, aber keine anderen Funktionen erlauben dies
- Basic: Sowohl für Zuweisungen als auch Vergleiche wird `=` eingesetzt

Orthogonalität gilt aber nicht nur für die Sprache selbst, sondern auch für die Notation in verwendeten Bibliotheken. Auch die Syntax und Semantik der Programmiersprache Python ist nicht frei von einigen der Orthogonalität abträglichen Eigenschaften; Details dazu wurden bereits in Kapitel 4 diskutiert:

Objektzugriff:
Alle Zugriffe auf Objekte sollten in Form von Methoden erfolgen. Einige Zugriffe erfolgen aber auch über Funktionen, wie `len()` auf Strings und Listen, `open()` auf Dateien. Seit Python 2.1 sind die Probleme durch Einführung neuer Methoden behoben, ohne die Kompatibilität einzuschränken.

Vererbung:
Ableitung neuer Klassen sollte von allen Typen und Klassen möglich sein. Seit Python 2.2 kann auch aus Basistypen wie Integer eine neue Klasse erstellt werden (sogenannte *New Style Classes*). Durch Vererbung können Standarddatentypen um Eigenschaften und Methoden erweitert werden. Es lassen sich beispielsweise Klassen für Währungen oder Metriken einführen und somit die Typsicherheit erhöhen, oder eine Klasse rationaler Zahlen konstruieren.

Zeichenketten:
Zur Notation von Zeichenketten gibt es zahlreiche Varianten. Insgesamt kann das als Vorteil für erfahrene Programmierer gesehen werden. Auf Anfänger wirkt diese Vielfalt oft verwirrend. Als Ausweg bietet sich an, in Anfängerkursen konsequent nur eine der möglichen Alternativen zu nutzen.

Funktionsdefinition:
Python arbeitet durch das Konzept der Einrückung weitgehend ohne spezielle Zeichen zur Markierung von Zeilenenden oder Block-Enden. Bei der Definition von Funktionen sowie zum Einleiten von Schleifen und Fallunterscheidungen muss allerdings explizit ein Doppelpunkt gesetzt werden.

Namensgebung:
Die Syntax für Bezeichner in Programmiersprachen stellt einen Prüfstein für die Orthogonalität dar. Um diese nicht unnötig mit Kontexteigenschaften zu erweitern, werden häufig Richtlinien vereinbart, welche die Schreibweise selbstgewählter Bezeichner nach einem Schema vorsehen. Häufig wird daher die folgende Konvention verwendet:

- Klassennamen beginnen mit großen Buchstaben.
- Methoden und Attribute werden klein geschrieben.
- Statt Leerzeichen wird der Unterstrich eingesetzt oder im Wort großgeschrieben (`einWert` bzw. `ein_wert`)
- Modulnamen und Bibliotheken werden durchgehend klein geschrieben.

Einige der Autoren der Module der Python-Basisbibliothek verstoßen gegen diese Konventionen. Beispiele: Vereinzelt sind Modulnamen groß geschrieben und zusammengesetzte Namen ohne Unterstrich anzutreffen. (Beispiele: *py_compile* gegenüber *pydoc*, *HTMLParser* gegenüber *htmllib*)

Ergebnis:

Mit Ausnahmen in der Namensgebung kann Python als orthogonale Sprache angesehen werden. Kritiken aus der Nutzergemeinschaft wurden und werden von den Python-Entwicklern ernst genommen und in neueren Versionen berücksichtigt, sofern Sprachänderungen nicht zu Kompatibilitätsproblemen führen. Hier ist die offene Entwicklung von Python, die jedem interessierten Entwickler die Möglichkeit zur Verbesserung der Sprache gibt, von Vorteil.

5.4 Lernen von Algorithmen

Neben der Modellierung ist das Erlernen von grundlegenden Algorithmen eines der wichtigsten fachdidaktischen Ziele. In der Literatur wird dabei häufig Pascal-ähnlicher Pseudocode zur programmiersprachenunabhängigen Notation von Algorithmen eingesetzt. Python wird hier der Forderung nach einer Pseudocode-artigen Syntax weitgehend gerecht, wozu auch der in Abschnitt 5.3 angesprochene Verzicht auf Deklarationen beiträgt. Daher sind zahlreiche Algorithmen, wie etwa Dijkstras Algorithmus für kürzeste Wege, in Python-Notation ebensogut verständlich wie in der üblichen Pseudocode-Version. In Abb. 13 und Tab. 7 werden die drei Darstellungsformen Struktogramm[28], Pseudocode und Python-Quelltext am Schulbeispiel des euklidischen Algorithmus gezeigt, um die Ähnlichkeiten zu veranschaulichen. Dijkstra und zahlreiche weitere Algorithmen sind beispielsweise im Python Cookbook (vgl. [PCOOK02]) exemplarisch umgesetzt.

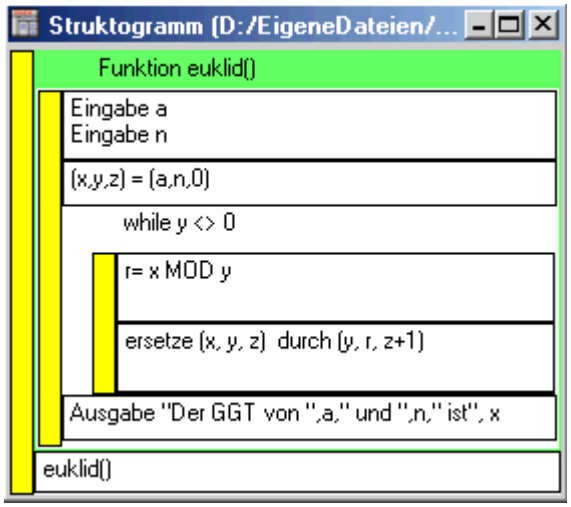


Abbildung 13: *euklidischer Algorithmus zur Berechnung des ggT*

```
Algorithmus Euklid:
  Eingabe a
  Eingabe n
  (x,y,z) := (a,n,0)

  solange y <> 0
  begin
    r := x MOD y
    ersetze (x,y,z) durch (y,r,z+1)
  end

  Ausgabe "GGT von " a " und " n "= " x
```

```
def Euklid():
  a = input("a = ")
  n = input("n = ")
  (x,y,z) = (a,n,0)

  while y <> 0:
    r = x % y
    (x,y,z) = (y,r,z+1)

  print "GGT von ",a," und ",n,"= ", x
```

Tabelle 7: *euklidischer Algorithmus: Pseudocode und Python*

5.5 Typische didaktische Problemfelder

Auf der Ebene atomarer Operationen in Programmiersprachen stellt – wie oben dargestellt – das Kriterium Orthogonalität den Schlüssel zur Erzielung eines widerspruchsfreien Erzeugen-

[28] erstellt mit PyNassi

densystems zur Verfügung. Auch die Einlösung dieser Forderung verhindert nicht, dass Schüler Fehlvorstellungen oder ungeeignete mentale Modelle aufbauen. Eine mögliche Ursache für dieses Problem ist in der Notwendigkeit zu sehen, durch eine (möglichst) einfache Syntax ein Maximum an Ausdrucksfähigkeit erzielen zu wollen. Deshalb ist bei der Modellierung darauf zu achten, dass die gleiche Schreibweise, wie sie beispielsweise aus der Mathematik bekannt ist ("=") in der Informatik eine völlig andere Bedeutung erhalten hat. Wird dieser Unterschied nicht thematisiert, so wird unnötigerweise Missinterpretationen Vorschub geleistet.

Es hat sich in der Vergangenheit gezeigt, dass es häufig zu Fehlvorstellungen beim Variablen-, Feld- und Funktionskonzept kommt (vgl. [Balzert79]). Zuweisungen von Variablen (`a = b`) werden oft statt als Kopie des Inhalts als Gleichung oder "Verschieben des Inhalts" missinterpretiert. Eine von der Mathematik leicht abweichende Notation kann zumindest die Assoziation zur Gleichung verhindern. In Algol wurde bereits mit der Darstellung der Zuweisung durch `":="` dieses Problem erkannt und erfolgreich gelöst.

Arrays (Felder) bereiten Anfängern oft Probleme, insbesondere wenn sie als Ersatz für Listen herangezogen werden müssen: Zum Speichern und Suchen von Einträgen in einem Array müssen zahlreiche Ansätze gleichzeitig verstanden werden: Variablenkonzept, die Verwaltung der Listengröße, unterschiedliche Schleifen mit Fallunterscheidungen und Abbruchbedingungen. Listen bieten fertige Methoden zum Einfügen und Zugriff auf Elemente. Durch den in Python möglichen Zugriff über den Index lassen sich Arrays, sofern sie denn wirklich als solche gebraucht werden, durch Listen simulieren.

Das Funktions- und Prozedurkonzept ähnelt in Python sehr der mathematischen Definition einer Funktion oder Formel. Durch konsequent lokale Variablennutzung werden die typischen Probleme bei Einsatz globaler Variablen vermieden.

Eine weitere Fehlvorstellung liegt in der Ansicht, Computer könnten intuitiv agieren. Bei der Umsetzung von Alltagssprechweisen zu Algorithmen entstehen dabei oft unerwartete Ausdrücke, beispielsweise:

```
Gift = 'Kadmium' or 'Blei'
if eingabe == Gift:
  print 'ist giftig'
```

Dieser Quelltext ist dabei auch noch syntaktisch korrekt, erfüllt aber sicher nicht die Erwartungen, die der Schüler hatte. Derartige Probleme kann auch der Python-Interpreter nicht verhinden, an dieser Stelle sind die Fähigkeiten eines guten Ausbilders nötig. Auf weitere Probleme, die beim praktischen Einsatz der Programmiersprache Python auftreten, wird in Kapitel 6 differenziert eingegangen.

5.6 Lernhilfsmittel und Schulungsumgebungen

Um die informatische Modellierung zielgerichtet unterrichtlich umsetzen zu können, ist es notwendig, geeignete Lernhilfen nutzen zu können. Dabei kommt der Unterstützung möglichst verschiedener Zugangsmöglichkeiten zum Problembereich eine herausragende Bedeutung zu. Der zu modellierende Bereich sollte sich besonders eignen, um die für wichtig erachteten Konzepte zu erarbeiten, darzustellen und implementieren zu können. Zum Einsatz von Python in der informatischen Ausbildung steht dem Lehrer bereits ein gutes Angebot an Hilfsmitteln für

anschaulichen Unterricht zur Verfügung, von denen nachfolgend einige ausgewählte Beispiele vorgestellt werden:

Zur Einführung in die objektorientierte Modellierung und Programmierung wird (in NRW und Bremen) häufig die Klassenbibliothek *Von Stiften und Mäusen* [CDH99] eingesetzt. Hier wird dem Schüler eine grafische Umgebung mit den Klassen `Stift`, `Maus`, `Bildschirm` und `Anwendung` geboten, von denen sich weitere Klassen ableiten lassen. Basierend auf diesem Konzept wurden zahlreiche Musteraufgaben entwickelt, die im Informatikunterricht der gymnasialen Oberstufe eingesetzt werden. Im Rahmen dieser Diplomarbeit wurde eine Python-Implementierung der Klassenbibliothek *Von Stiften und Mäusen* namens *SuM* erstellt, welche im Kapitel 6.2 als Fallstudie genauer beschrieben wird.

Das Ergebnis einer weiteren Fallstudie (*PyNassi*) unterstützt die Erstellung und Änderung von Struktogrammen und visualisiert dynamische Abläufe auf diesen Struktogrammen. Das Ziel dieser Fallstudie bestand darüber hinaus darin, im Kontext der Diplomarbeit die Anforderungen an schnelle Softwareentwicklung beispielhaft umzusetzen. Mit *PyNassi* lassen sich Struktogramme erstellen, dabei können Anweisungen als Python-Quelltext eingefügt werden. Das fertige Struktogramm kann dann zu einem Python-Programm übersetzt werden und der Ablauf des Programms anschließend am Struktogramm beobachtet werden. Struktogramme sind zur graphischen Darstellung von Abläufen in Algorithmen entwickelt worden. Sie eignen sich besonders, um strukturiertes prozedurales oder funktionales Denken – programmiersprachenunabhängig – auf der anschaulichen Ebene zu unterstützen. Die Umsetzung von Struktogrammen in konkrete imperative und objektorientierte Sprachelemente ist mit einfachen Mitteln möglich. *PyNassi* bietet sich hier als Übergang zur konkreten Implementierung am Computer an.

Zur prozeduralen Modellierung wurde (ab 1981) eine einfache Modellwelt unter den Namen *Karel der Roboter, Niki* oder *Hamster* zum Einsatz im Informatikunterricht zur Verfügung gestellt. Der Einsatz dieser Modellwelt ist beispielsweise in Bayern für den zukünftig verpflichtenden Informatikunterricht in der Sekundarstufe I des Gymnasiums vorgesehen (vgl. [Hubwieser02]). Für Python sind Umsetzungen dieser Modellwelt verfügbar, beispielsweise das von Schülern entwickelte *PyKarel* oder *Niko* [Kokavecz01]. Bestimmte Probleme der prozeduralen Modellierung können mit dieser Modellwelt unterrichtlich umgesetzt werden. Dabei ist zu beachten, dass Schüler in der Sekundarstufe II (Oberstufe) solche "Spielwelten" häufig als kognitive Unterforderung betrachten und sich damit nicht ernsthaft mit den zugrundeliegenden Konzepten auseinanderzusetzen bereit sind.

Zur Vorstellung von Algorithmen, die auf Graphen arbeiten, wurde die Lernsoftware *Gato* entwickelt [Schliep02]. *Gato* bietet einfache Möglichkeiten, Graphen und darauf arbeitende Algorithmen zu definieren. Die Darstellung der Graphen wird von *Gato* automatisch übernommen, wobei sich der Ablauf der Algorithmen parallel am Quelltext und am Graphen darstellen lässt.

Zwei neuartige Ansätze zur objektorientierten Einführung in die Programmierung mit dreidimensionaler, animierter Computergrafik stellen *Vpython* [Scherer02] und *Alice* [Alice02] dar. Es sind allerdings ein räumliches Vorstellungsvermögen beziehungsweise mathematische Grundlagen der linearen Algebra erforderlich, so dass der schulische Einsatz nur in der Oberstufe oder mit sehr interessierten Schülern möglich ist. Beide Projekte bieten unter diesen Vorraussetzungen einen geeigneten Zugang zur Computergrafik und Animation und gestatten frühe Erfolgserlebnisse.

PyCard [PyCard02] eignet sich als Einführung in die Erstellung von Anwendungen mit grafischen Benutzungsoberflächen. Es entstand in Anlehnung an Apples *HyperCard* [Apple02].

Zahlreiche lernorientierte Beispielanwendungen veranschaulichen die Benutzung der auf leichte Programmierung optimierten Bibliothek, welche selbst auf *wxPython* aufsetzt. Die Durchsicht der Konzepte und Beispiel, die mit diesem Ansatz verbunden sind, vermittelt den Eindruck einer vor allem für den Informatikanfangsunterricht interessanten Bibliothek. Die Einbindung netzwerkbezogener Elemente und Dienste kommt neueren fachdidaktischen Forderungen nach der Thematisierung vernetzter Strukturen sehr entgegen (vgl. [HS02] oder [Humbert01])

Für Entwickler mit Erfahrungen in anderen Programmiersprachen gibt es zahlreiche Einführungen, die in kurzer Zeit die Grundlagen der Python-Entwicklung vermitteln. Bereits das in der Python-Dokumentation enthaltene englischsprachige Tutorial ist geeignet, um alle Möglichkeiten der Python-Entwicklung zu erlernen. Eine Übersicht weiterer Einführungen ist auf der Python-Webseite [Python02] zu finden.

Eine besonders interessante Einführung ist das Open-Book-Projekt *How to Think Like a Computer Scientist*. Diese Einführung wird zur Zeit in zahlreiche (natürliche) Sprachen übersetzt, und ist außerdem auch für andere Programmiersprachen (derzeit Java und C++) erhältlich [DEM01]. Auf der Webseite [Burley02] werden zusätzlich interaktive Testmöglichkeiten für Python-Quelltexte angeboten. Es muss also keinerlei lokale Installation vorgenommen werden.

Weitere auf spezielle Anwendergruppen optimierte Kurse mit zugehörenden Bibliotheken sind etwa *BioPython* [Biopy02] zur Genanalyse und M. Williams *Handbook of the Physics Computing* [Williams02].

5.7 Eingabewerkzeuge

Geeignete Eingabewerkzeuge in Form von speziell auf Python optimierten Editoren und integrierten Entwicklungsumgebungen können die Entwicklung erheblich vereinfachen. Nachfolgend werden einige nützliche Eigenschaften aktueller Eingabewerkzeuge gezeigt.

Syntax-Hervorhebung, Integration der Compiler beziehungsweise Interpreteraufrufe sowie Protokollierung der Fehlerausgaben an den Standard-Ausgabe- und Fehlerkanal mit Übernahme in die Benutzungsoberfläche bieten inzwischen nahezu alle besseren Editoren und IDEs. Ebenso erleichtern automatische Vervollständigung der Schlüsselwörter sowie automatisches Einrücken die Eingabe.

Eine wirklich neue und sehr nützliche Eigenschaft integrierter Entwicklungsumgebungen wie *PythonWin* ist die Möglichkeit der Auflistung und Vervollständigung von Methoden und Attributen bei Objekten. Dabei sind nicht nur Schlüsselwörter aus der Syntax der Sprache und der Systembibliotheken nutzbar, sondern auch Attribute zuvor selbst definierter Objekte.

Hier machen die Werkzeuge Gebrauch von der Objekt-Introspektion, also der Möglichkeit, zur Laufzeit Angaben zu Objekten zu erhalten. Besonders Entwickler, die noch nicht lange mit Python arbeiten, werden diese Eigenschaft schätzen, um schnell eine Liste aller Methoden auf ein Objekt zu bekommen.

Ergebnis:

Im praktischen Einsatz erspart die halbautomatische Vervollständigung einige Schreibarbeit und viele Blicke in die Dokumentation. Somit ist sie für schnelle Softwareentwicklung ein nützliches und in dieser Form neues Hilfsmittel, welche von einigen modernen Werkzeugen bereits geboten wird.

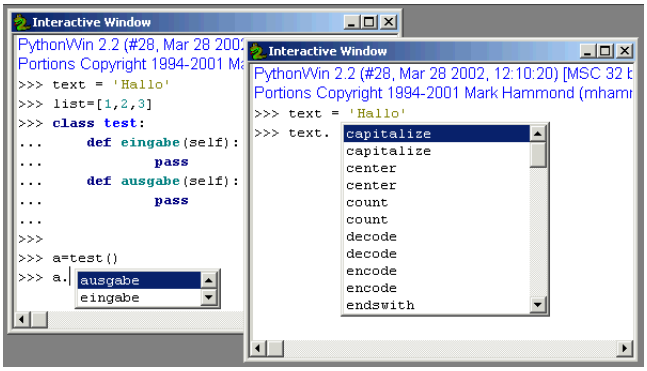


Abbildung 14: *Vervollständigung von Methoden- und Attributbezeichnern*

5.8 Integrierte Entwicklungsumgebungen

In Abschnitt 4.4 und Anhang A.3 wurden einige integrierte Entwicklungsumgebungen vorgestellt und ihre möglichen Einsatzgebiete besprochen. Es zeigte sich, dass dem Python-Entwickler sowohl in der Entwurfsphase wie auch zur Implementierung und Dokumentation eine breite Auswahl an Werkzeugen zur Verfügung steht. Bereits die kostenlosen Werkzeuge bieten einem kleinen Entwicklerteam genügend Möglichkeiten, die Arbeit an Projekten zu erleichtern.

Für welche Werkzeuge und Entwicklungsumgebung sich ein Entwickler entscheiden wird, hängt letztendlich auch vom Einsatzgebiet und der Zielplattform ab. Insbesondere zum Entwurf von Oberflächen findet der Entwickler mit *PythonWorks* für *Tkinter*, *BlackAdder* für *Qt* und *Boa* für *WxPython* mögliche Hilfsmittel.

In der Entwurfsphase bietet *Dia* bei kleinen Projekten bereits gute Unterstützung. Bei Planung größerer Projekte kann *ObjectDomain-R3* dank Versionsverwaltung und Mehrbenutzereignung geeigneter sein.

Bei der Wahl eines Werkzeugs zur reinen Implementierung und Quelltexteingabe ohne Hilfsmittel zur GUI-Erstellung ist die Auswahl groß, selbst wenn noch ein Debugger und eine Python-Shell gefordert sind. *IDLE*, *ActivePython*, *Komodo* und die *Wing-IDE* leisten hier Vergleichbares. Letztere bietet sich besonders in Netzwerkumgebungen an und ist die einzige Entwicklungsumgebung, die problemlos mit allen GUI-Bibliotheken genutzt werden kann. *ActivePython* wird sicher den Windows-Anwender ansprechen, während *Komodo* “mehrsprachigen” Programmierern gefallen wird.

Eine Übersicht der Leistungsmerkmale zeigt Tabelle 15:

	UML-Darstelng Import / Export	Klassen-Browser	Doku	GUI-Erstellung	Multi-User	Modellierung	Quelltext-Editor / Ausgabe / Import	Debugger	Versions-Verwaltung	Weitere Sprachen
Projektplanung										
Dia mit Erweiterungen	D,I,E	-	+	-	-	+	A	-	-	+
PyNassi	-	-	-	-	-	-	A,E	+	-	-
Object Domain R3	D,I,E	+	+	-	+	+	A,E,I	+	+	+
(Rational Rose)	D,I,E	+	+	-	+	+	A	-	+	+
PyUT	D,I,E	-	-	-	-	+	A,I	-	-	+
Thorn	D	-	-	-	-	+	A,E	-	-	-
Leo	-	-	+	-	+	-	A	-	+	-
Integrierte Entwicklung										
Boa Constructor	D	+	+	WxPython	-	-	E	+	-	-
HappyDoc	E	-	+	-	-	-	-	-	-	+
Idle	-	+	-	-	-	-	E	+	-	-
BlackAdder	-	-	-	Qt	-	-	A,E	-	-	Ruby
Kommodo	-	-	-	-	-	-	E	+	-	+
WingIDE	-	+	+	-	+	-	A,E	-	+	+
ActivePython	-	-	-	-	-	-	E	+	-	-
PythonWorksPro	-	-	-	Tkinter	+	-	+	+	+	-
PyCrust	-	-	+	-	-	-	+	-	-	-

Abbildung 15: *Entwicklungswerkzeuge*

Erläuterung der Spalten:

UML: Das Produkt kann UML darstellen (D), importieren (I) und exportieren (E).
Klassenbrowser: Es kann eine Übersicht der Klassen oder Methoden dargestellt werden.
Doku: Die Erstellung einer Quelltextdokumentation wird unterstützt.
GUI-Erstellung: Der Entwurf von Formularen für das angegebene Modul ist möglich.
Multi-User: Das Produkt unterstützt die Bearbeitung durch mehrere Benutzer.
Modellierung: Planung und Entwurf von Strukturen in Form von Diagrammen ist vorgesehen.
Quelltexteingabe: E = Editor für Quelltext ist integriert, A= Quelltexterzeugung, I=Quelltextimport mit Analyse
Debugger: Fehlersuche und Ablaufanalyse auf Quelltextebene ist möglich.
Versionsverwaltung: Es lassen sich Produktänderungen protokollieren.
Andere Sprachen: Neben Python werden die aufgeführte oder weitere Programmiersprachen unterstützt.

5.9 Zusammenfassung

In diesem Kapitel wurde Python anhand der zuvor aufgestellten Kriterien untersucht, welche überwiegend fachdidaktische Anforderungen betreffen. Es zeigte sich, dass Python den Anforderungen in vielen Bereichen gerecht wird, auch wenn einige Probleme erkannt wurden.

Python eignet sich gut, um die unterschiedlichen Paradigmen der Programmierung innerhalb einer Programmiersprache zu nutzen. Dabei lassen sich prozedurale, objektorientierte und funktionale Konzepte sehr gut sowohl einzeln wie auch in Kombination untereinander nutzen. Prädikative Entwicklung ist auf unterschiedliche Weise ebenfalls möglich, ist allerdings an objektorientierte Ansätze gebunden oder erfolgt über eingebettete Fremdinterpreter.

Python kann als sehr gut les- und erlernbare Sprache angesehen werden und erweist sich mit nur wenigen Ausnahmen (siehe 5.3) als weitgehend orthogonal. Zur Unterstützung konzeptionellen Lernens ist die Orthogonalität der Programmiersprache damit als wichtige Vorbedingung erfüllt. Ansätze, die konzeptionelles Lernen durch die Beschränkung auf Modellwelten unterstützen, sind in signifikanter Anzahl auch für Python verfügbar. Des Weiteren existieren

diverse Entwicklungsumgebungen, welche den unterschiedlichen Anforderungen sowohl von Anfängern als auch professionellen Entwicklern gerecht werden.

6 Python in der Praxis

Meine Sprache ist allzeit simpel, enge und plan.
Wenn man einen Ochsen schlachten will,
so schlägt man ihm gerade vor den Kopf.

Georg Christoph Lichtenberg (1742 – 1799)

In diesem Kapitel werden Erfahrungen von Python-Entwicklern, die durch Befragung im Zuge dieser Arbeit erfaßt wurden, vorgestellt. Es beinhaltet darüber hinaus die Zusammenfassung von Erkenntnissen, die durch empirische Untersuchungen in der Literatur dokumentiert wurden und enthält ferner die in Kapitel 3 erwähnten Fallstudien, welche vom Autor im Rahmen dieser Arbeit durchgeführt wurden.

Im Anhang A.5 werden einige Kommentare von Entwicklern zu Python vorgestellt. Zusammenfassend lässt sich feststellen, dass Python besonders wegen guter Lesbarkeit und Erlernbarkeit geschätzt wird. Die Motivation beim Arbeiten mit Python ist sehr hoch, und die Arbeit führt schnell zum Ziel. Nachfolgend sollen diese Erfahrungen genauer untersucht werden.

6.1 Ergebnisse empirischer Untersuchungen

Nachdem Python bezüglich der in Kapitel 4 überprüften Kriterien und subjektiver Einschätzungen befragter Entwickler für die schnelle Softwareentwicklung geeignet erscheint, wird dies mit Hilfe empirischer Untersuchungen überprüft.

In [Prechelt00] wird ein empirischer Vergleich von C, C++, Java, Perl, Python, Rexx und Tcl durchgeführt. Dabei wurden je Sprache cirka 20 Entwickler mit unterschiedlichen Erfahrungen zur Lösung einer geeigneten, nichttrivialen Problemstellung in den genannten Sprachen beauftragt. Untersucht werden Entwicklungszeit, Laufzeit und Programmgröße der jeweiligen Lösungen in den jeweiligen Sprachen.

Es zeigte sich:

1. Python und Perl liegen bei dem gestellten Problem im Punkt Geschwindigkeit im mittleren Bereich, nur C und C++ waren etwa um Faktor 4 schneller. Innerhalb der Skriptsprachen waren Perl und Python am schnellsten.

2. Bezüglich der Programm-Länge erreichte Python das beste Ergebnis, knapp gefolgt von Perl. Java sowie C / C++ waren um den Faktor 3 länger.

3. Skript-Lösungen und insbesondere Python waren geringfügig fehlerfreier und zuverlässiger als die Varianten in C / C++.

4. Die Entwicklungszeit war bei Python etwa um den Faktor 3 kürzer als bei Java, C und C++.

5. Skriptsprachen sind auch zur Bearbeitung sehr großer Datenmengen geeignet.

In [Waclena97] erfolgt ein Vergleich der Leistungsmerkmale unterschiedlicher Programmiersprachen. Über einen Bewertungsmaßstab werden überwiegend syntaktische Merkmale verglichen. In dieser Statistik liegen OCAML, Python, SML und Haskell auf den vorderen Plätzen. Die Auswertung ist bereits relativ alt, mit den gesetzten Maßstäben würde die aktuelle Version von Python (aber auch ein Teil der weiteren Sprachen) besser abschneiden, da zahlreiche Neuerungen in die Sprachen integriert wurden und Bibliotheken wie Sprachen verbessert sowie beschleunigt wurden.

Lingyun Wang und Phil Pfeiffer führen in [WP02] eine qualitative Fallstudie durch, indem sie selbst zwei Programme in den drei Sprachen Perl, Python und Tcl implementieren.

Sie stellen fest:

1. Die Entwicklungszeit in Python ist etwa 10 – 30 % kürzer als in Perl und Tcl.
2. In Python werden deutlich weniger Implementierungsfehler gemacht als in Perl und Tcl.

Doug Bagley sammelt in [Bagley01] für 30 Programmiersprachen Implementierungen für jeweils 25 bekannte Algorithmen und führt Messungen zur Laufzeit, Speichernutzung und Quelltextgröße in Lines of Code durch. Der Autor vermeidet in seiner Dokumentation allerdings eine wertende Darstellung. Die dokumentierten Ergebnisse lassen jedoch folgende Schlüsse zu:

1. Python liegt bezüglich Geschwindigkeit und Speichernutzung im Mittelfeld. Vorteile liegen bei String- und Hashbenutzung, die Objektverwaltung ist dagegen vergleichsweise langsam.
2. Python ist insbesondere bei den komplexeren Problemen den jeweils schnellsten Implementierungen nur um einen relativ kleinen Faktor (3 – 5) unterlegen. Bei relativ prozessornahen Benchmarks muss es sich allerdings mit Faktoren >10 geschlagen geben.
3. Python zählt bezüglich der Codegröße bei den komplexeren Problemen zu den kompaktesten Sprachen und übertrifft insbesondere Compilersprachen um den Faktor 3 – 20.

Ergebnis:

Die Untersuchungen zeigen, dass moderne objektorientierte Skriptsprachen bezüglich der Entwicklungszeit und Quelltextgröße den restlichen Sprachen meist deutlich überlegen sind. Innerhalb der Skriptsprachen gehört Python dabei zu den jeweils besten Sprachen.

Im Hinblick auf die Laufzeit nimmt Python innerhalb der Skriptsprachen eine gute Position ein, ist aber im Vergleich zu kompilierten Sprachen in Benchmarks unterlegen. Bei realen, großen Programmen macht sich dies allerdings weniger bemerkbar als bei rein algorithmischen Benchmarks. Viele Bibliotheksfunktionen arbeiten intern sehr schnell, so dass Laufzeitverluste durch den Interpreter teilweise vernachlässigt werden können. Bei der Entwicklung von Anwendungen, die nicht auf optimale algorithmische Leistung angewiesen sind, macht sich der Unterschied daher nur unwesentlich bemerkbar.

Insgesamt decken sich die Ergebnisse mit den Erfahrungen, die in den Fallstudien des Autors beobachtet wurden. Diese werden in den anschließenden drei Abschnitten diskutiert.

6.2 Fallstudie SuM

Ziel des ersten Software-Projektes im Rahmen dieser Arbeit war die schnelle Implementierung der Ausbildungssoftware *Von Stiften und Mäusen*. Eine genaue Beschreibung der zu erstellenden Klassenbibliothek inklusive Anwendungsbeispielen als Pseudocode lag in Form der Dokumentation der Software vor [CDH99]. Im Vordergrund stand somit die direkte Implementierung des gegebenen Konzeptes. Der fertige Quelltext wurde aber später auch zum Test von Dokumentationswerkzeugen und zur Erstellung von Klassendiagrammen herangezogen.

Zunächst galt es, eine geeignete Bibliothek für die Grafikausgabe auszuwählen. Voraussetzungen waren Lauffähigkeit auf möglichst vielen Plattformen und gute Grafikleistungen auch auf langsameren Computern. Des Weiteren sollte die Bibliothek die Möglichkeit des Threadings bieten, um eine gleichzeitige Nutzung von Interpreter und Grafikausgabe zu gestatten.

Letzteres Kriterium erwies sich als relativ hart. Von den in Abschnitt 4.3 vorgestellten Bibliotheken wurden nur noch *WxWindows* und *PyGame* in die engere Auswahl gezogen. Wegen der erstaunlich hohen Grafikleistung, die beispielsweise von den beiliegenden Demonstrationen gezeigt wird, und einer einfacheren Bibliotheksstruktur wurde *PyGame* bevorzugt. *PyGame* ist gut dokumentiert, einzelne Grafikfunktionen lassen sich ohne Probleme direkt im Interpreter testen.

Die Notation der einzelnen Klassenmethoden sollte einerseits weitgehend der Definition in der Klassenbibliothek entsprechen, andererseits aber auch keine Python-Konventionen (siehe Abschnitt 5.3) verletzen. Aus diesem Grund wurde entschieden, Klassennamen immer groß zu schreiben, und Methodennamen in Verbform klein. Aus mehreren Wörtern zusammengesetzte Methodennamen wurden ohne Trennzeichen zusammengeschrieben, wobei ein zweites Substantiv gegebenenfalls groß geschrieben wurde. Die Namen und Reihenfolge der Parameter von Methoden wurden exakt beibehalten.

Zur Eingabe der Quelltexte wurde ein Texteditor mit Syntaxunterlegung und Tabulatorwandlung verwendet. Die Implementierung erfolgte *Bottom up* nach der *Code & Fix* Methode (siehe Abschnitt 2.2). Zunächst wurde die Klasse Bildschirm implementiert, wobei im Konstruktor ein Thread für die automatische Bildschirmaktualisierung realisiert werden musste. Bei der Implementierung der Klassen *Stift* und *Maus* fiel auf, dass für eine saubere Umsetzung eine („kennt-") Beziehung zur Klasse *Bildschirm* nötig ist, da ansonsten mit globalen Zuständen gearbeitet werden muss. Infolgedessen wurde die Klassenbeschreibung korrigiert und die Autoren des Konzeptes über diesen Mangel benachrichtigt. Die Klassen *Stift, Maus, Tastatur* und *Bildschirm* waren innerhalb eines Arbeitstages funktionsfähig implementiert und mittels der integrierten Testfunktionen getestet.

Einen weiteren Arbeitstag nahmen die restlichen Klassen in Anspruch. Anschließende Tests mit Beispielen aus der Literatur erfolgten direkt im Interpreter. Hier konnten die von Python gebotenen Vorteile (vgl. 4.8) zum schnellen Testen gut ausgenutzt werden. In den Probeläufen wurde dabei ein erster Fehler der Implementierung erkannt: Winkelangaben wurden im Uhrzeigersinn statt ihm entgegengesetzt behandelt. Korrektur und ein erneuter Test erfolgten ohne Neustart direkt im Interpreter, anschließend wurde im Editor der Quelltext korrigiert. Implementierungen der Beispiele und zahlreiche Funktionstests nahmen einen weiteren Arbeitstag in Anspruch.

Die dynamische Typisierung erwies sich in diesem Projekt als angenehm. Durch die genaue Projektdefinition waren Typfehler nahezu ausgeschlossen. Die Dynamik ersparte viele Zeilen zur Deklaration von Variablen, Attributen und Parametern. In der Klasse Bildschirm wurde

häufig von der Ausnahmebehandlung (siehe auch Abschnitt 4.5.9) Gebrauch gemacht, um Fehler bei Bibliotheksaufrufen zu erkennen und abzufangen.

Nützlich bei der Bibliothek waren auch die Funktionsdefinitionen, die oft mit Standardparametern vorbelegt sind. Lange und zumeist überflüssige optionale Parameterangaben konnten so weggelassen werden.

Insgesamt nahm die gesamte Implementierung etwas mehr als zwei Tage in Anspruch. Da ursprünglich mehr Zeit für das Projekt eingeplant war, wurde die restliche Zeit für eine Erweiterung des Konzeptes genutzt. Es fällt auf, dass zahlreiche Beispiele aus der Dokumentation bewegte Objekte verwenden. Aus diesem Grund wurde eine neue Klasse *Sprite* definiert, welche die Möglichkeit bietet, zuvor entworfene Figuren auf dem Bildschirm zu animieren. Dabei wurden Methoden zum Setzen der Position und des Drehwinkels realisiert. Mit einfachen Erweiterungen der Klasse können beispielsweise vom Schüler die aus der sogenannten *Turtle-Grafik* bekannten Methoden umgesetzt werden.

Zur Dokumentation der Quelltexte und zur Erstellung eines Klassendiagramms wurde das Python-Programm *Happydoc* eingesetzt. Es lieferte eine brauchbare und gut formatierte Dokumentation aller Klassen und Module im HTML-Format. Bessere Ergebnisse hätte die Verwendung von Formatierungskonventionen der Dokumentationsstrings im Quelltext geliefert, was in diesem Projekt jedoch nicht eingesetzt wurde.

Nachdem das Projekt fertiggestellt war, ergaben sich noch einige unerwartete Probleme: Die verwendete Bibliothek *PyGame* konfligiert mit der integrierten Entwicklungsumgebung *PythonWin*. Auf diese Problematik wird in Abschnitt 6.5 genauer eingegangen.

Zusammenfassung:

Die Umsetzung des Projektes verlief erstaunlich schnell. Eingeplant war eine Woche Arbeitszeit, zu gleichen Teilen für Bibliotheksauswahl und Einarbeitung, Implementierung, sowie Test und Quelltextdokumentation. Bereits nach einem kurzen Blick in die Beispielprogramme und die Dokumentation konnten einfache Grafiken ausgegeben werden, so dass am selben Tag mit der Implementierung begonnen wurde und am dritten Tag das Projekt in der geplanten Form fertiggestellt war. Zusammenfassend kann die Implementierung der Klassenbibliothek *Von Stiften und Mäusen* als erfolgreiches Beispiel einer schnellen Softwareentwicklung durch einen einzelnen Programmierer angesehen werden.

Erweiterungen:

Sicherlich werden bei einem ersten praktischen Einsatz noch Verbesserungsvorschläge und Probleme seitens der Lehrer und Schüler festgestellt. Die Quelltexte wurden gut dokumentiert, so dass eigene Verbesserungen einfach möglich sind. Auch wird versucht werden, im Anschluss an diese Arbeit das Projekt weiter zu betreuen und Probleme beim Einsatz in unterschiedlichen Umgebungen zu beseitigen.

6.3 Fallstudie PyNassi

Ziel des zweiten Projekts war der Entwurf eines Prototypen und die anschließende vollständige Implementierung eines Editors für Struktogramme (siehe Abschnitt 3.2). Nach ersten

Vorüberlegungen zum Design der Benutzungsoberfläche und zu Möglichkeiten der Eingabe sowie Darstellung der Struktogramme musste eine Bibliothek ausgewählt werden.

Anforderungen an die Bibliothek waren moderne Widgets, Kompatibilität, die Möglichkeit eine Zeichenfläche (Canvas) zu nutzen und ein Hilfsmittel zum Dialogentwurf. In die engere Auswahl wurden daher *wxPython* und *PyQt* genommen. Wegen des auf den ersten Blick recht gelungenen *Qt Designers* und vorhandener Beispielquelltexte fiel die Entscheidung auf *PyQt*.

Zunächst wurden einige einfache Testdialoge und Anwendungen erstellt, um die Arbeit mit der Bibliothek und dem Designer zu erlernen. Nach drei Tagen war das Grundgerüst eines Testprogramms fertiggestellt, das alle benötigten Widgets demonstrierte. Nun begann der Entwurf des ersten Prototypen: Im *Qt Designer* wurden komplexere Dialoge entworfen und dann in Python-Quelltext übersetzt. Einfachere Dialoge wurden direkt als Klasse, also ohne Nutzung des Designers, implementiert. Hierbei war es sehr vorteilhaft, die einzelnen Dialoge direkt im Interpreter starten zu können, ohne auf die Gesamtanwendung angewiesen zu sein. Die Dialoge wurden dann mit dem Testprogramm zu einer einfachen Anwendung kombiniert. Damit war der erste startbare, aber noch funktionslose Prototyp fertiggestellt. Das nach einer Woche vorgestellte Ergebnis stellte Entwickler wie auch Testpersonen zufrieden.

In der nächsten Phase wurden die eigentlichen Struktogramm-Elemente erstellt. Dazu wurden direkt aus einfachen Entwurfsskizzen der drei Statement-Arten (Anweisung, Block, Fallunterscheidung) die erforderlichen Klassen implementiert. Von zwei experimentell erfolgten Entwürfen wurde der erste wegen einiger Entwurfsprobleme verworfen. Der verbleibende Entwurf wurde in einer neuen, erbenden Klasse um Methoden für Ausgaben in ein Canvas erweitert und schließlich in den Entwurfsdialog integriert. Dieser wurde erst einzeln getestet und dann mit der Gesamtanwendung kombiniert. Das Ergebnis wurde den Anwendern vorgestellt und als Prototyp akzeptiert.

Die bisher erstellten Quelltexte waren bereits so gut strukturiert, dass sie für den Ausbau zum Endprodukt weiterverwendet werden konnten. Mängel in der Quelltextdokumentation wurden korrigiert, experimentelle Funktionen entfernt und mit *Happydoc* eine vorläufige Klassenübersicht erstellt. Darauf basierend wurden Klassen für weitere Statement-Arten (Schleife, Rückkehranweisung, Funktion und Klasse) spezifiziert und die Anforderungen an das Endprodukt festgelegt (Laden und Speichern, Quelltexterstellung, Quelltexteditor, Debugger).

Mit diesen Anforderungen wurde die Ausgabeklasse um die Fähigkeit zur Quelltexterzeugung erweitert sowie neue Klassen (Editor, Debugger und Debugger-Ausgaben) geplant. Die Erweiterung der Statements inklusive Quelltextausgabe war nach zwei Arbeitstagen fertiggestellt. Als Editor wurde ein vorhandenes Qt-Widget als “Fertigteil” genutzt. Die Entwicklung des Debuggers wurde zunächst vom Projekt getrennt und separat als eigenes Programm entwickelt und getestet. In einem Team hätte die Entwicklung des Debuggers unabhängig und parallel erfolgen können (vgl. Folgeabschnitt 6.4). Das fertige Modul konnte dann problemlos eingebunden werden.

Zum Laden und Speichern der Struktogramme wurde *Pickling* eingesetzt. Die erforderlichen Methoden umfassen nur wenige Programmzeilen und waren in weniger als einer Stunde fertiggestellt.

Das Produkt wurde einem Testanwender übergeben und nach einigen kleinen Änderungen allgemein vorgestellt. In einem letzten Schritt wurde eine Hilfsfunktion und eine Anwenderdokumentation erstellt sowie die Quelltextdokumentation aktualisiert.

Im ersten Testeinsätzen zeigten sich noch einige unerwartete Probleme: Die aktuellen *PyQt*-Bibliotheken lassen sich auf MacOS nur mit Problemen installieren, da fertig kompilierte Ver-

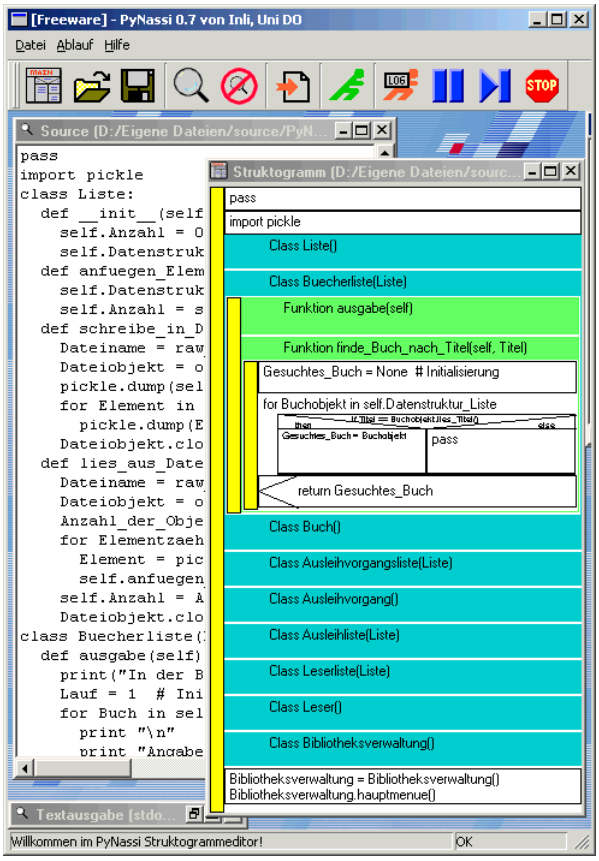


Abbildung 16: *PyNassi*

sionen derzeit noch nicht vorliegen und die manuelle Installation sehr aufwendig ist. Einzelne Widgets wie beispielsweise Icons und bestimmte Schriftarten funktionieren unter MacOS noch nicht korrekt. Die Windows- und Linux-Version arbeitet dagegen problemlos. Die kostenlose, nicht ganz aktuelle Version von *Qt* zeigte zunächst noch einige Redraw-Probleme, eine Versionsupdate hat diesen Fehler behoben.

Das eingesetzte *pickle*-Modul arbeitet zwar auf jedem System korrekt, ein Datenaustausch zwischen unterschiedlichen Systemen ist wegen unterschiedlicher Zeilenende-Zeichen aber problematisch. Zukünftige Python-Versionen sollen das Problem der Übernahme von Textdateien zwischen UNIX und Windows umgehen können.

Ergebnis:

PyNassi war innerhalb von drei Wochen fertiggestellt. Die Entwicklung in der ersten Phase kann als erfolgreiches Beispiel einer Softwareentwicklung per Rapid Prototyping gesehen werden: Innerhalb weniger Tage war eine präsentable Benutzungsoberfläche und Grundfunktionalität des Programms realisiert.

Der Endausbau war eher eine Mischform zwischen Evolutionärem Prototyping und Extreme Programming: mehrere kurze Zyklen von Planung, Entwicklung und Test kombiniert mit Dialogen zum Anwender. Python unterstützte die einzelnen Entwicklungsschritte ausgezeichnet.

Neben syntaktischen und semantischen Merkmalen haben sich in diesem Projekt die unabhängigen und interaktiven Testmöglichkeiten einzelner Module als vorteilhaft erwiesen. Eben-

so gut gelang die Integration des Debugger-Teilprojekts. Ein besonders wichtiges Merkmal in diesem Projekt war die Möglichkeiten der dynamischen Quelltexterzeugung und Ansteuerung des Interpreters sowie die Introspektion: Ohne diese Eigenschaften wären viele Merkmale der Software nicht oder nur mit gewaltigem Mehraufwand realisierbar gewesen.

Insgesamt würde ein vergleichbares Softwareprojekt in klassischen, kompilierten objektorientierten Programmiersprachen wie C++ oder Java, selbst bei Nutzung visueller Entwicklungsumgebungen, deutlich mehr Entwicklungszeit benötigen.

6.4 Teilprojekt Debugger

Ziel des Projektes war die Entwicklung eines modularen Debuggers für Python-Quelltext. Der Debugger sollte einen als Parameter übergebenen Quelltext als eigenen Thread starten, zeilenweise bearbeiten und Statusmeldungen in eine globale Liste schreiben. Das fertige Modul sollte sich später in *PyNassi* oder andere Projekte integrieren lassen, aber auch eigenständig verwendet werden können.

Die Python-Dokumentation gab über die internen Vorgänge des Python-Interpreters und über vorhandene Debugging-Funktionen ausreichende Angaben. Mit einigen interaktiven Experimenten im Interpreter wurde die Funktionsweise der einzelnen Module erprobt, welche zu diesem Zeitpunkt zum Teil noch unbekannt waren. Die Module *threading* und *time* waren bereits aus dem *SuM*-Projekt bekannt. Zum Verständnis der Module *traceback* und *inspect* konnten im Interpreter kurze Testfunktionen und Beispiele ausprobiert werden.

Erhebliche Erleichterungen ergaben sich durch Einsatz der Python-Funktionen `exec` und `compile`. Mit nur einem Funktionsaufruf kann interaktiv Python-Quelltext innerhalb eines Python-Programms verarbeitet werden. Dies kann wahlweise in einem eigenen Kontext oder dynamisch im gleichen Kontext wie das aufrufende Programm geschehen. Funktionen und Quelltexte können als Objekt behandelt werden und somit auch Parameter für Funktionen und Methoden sein. In Compilersprachen ist dies technisch unmöglich, und kaum eine andere Interpretersprache bietet derartig vielseitige Möglichkeiten. Zunächst war geplant, den Quelltext zeilenweise mit `exec`-Anweisungen auszuführen und dann den aktuellen Status zu ermitteln. Es zeigt sich aber, dass die Funktionen der Module *traceback* und *inspect* genau diese Funktionalität bereits für beliebige Quelltexte anbieten. Daher musste nur noch die parallele Bearbeitung per Thread sowie die Datenauswertung beziehungsweise Kommunikation realisiert werden.

Die ersten Experimente mit den genutzten Modulen benötigten noch etwas mehr als einen Tag. Dagegen war die endgültige Implementierung und der Test nach wenigen Stunden abgeschlossen. Der gesamte Quelltext des Projektes inklusive interner Dokumentation und zweier Testfälle umfasst weniger als 150 Programmzeilen.

6.5 Probleme bei Implementierung der Fallstudien

Während der Erstellung dieser Arbeit traten einige unerwartete Probleme bei der Nutzung von Python, Werkzeugen und Bibliotheken auf. Dieser Abschnitt soll erkannte Schwierigkeiten beschreiben, die nicht in den erstellten Kriterienkatalog passen, oder nur in einzelnen Versionen beobachtet wurden.

Kompatibilitätsprobleme der GUI-Bibliotheken

Es zeigte sich erwartungsgemäß, dass es mit wenigen Ausnahmen nicht möglich ist, mehrere unterschiedliche grafische Umgebungen innerhalb einer Anwendung zu nutzen. Dieses Problem erscheint zunächst nebensächlich. Allerdings ist es oft auch nicht möglich, innerhalb einer integrierten Entwicklungsumgebung, die selbst ein bestimmtes GUI nutzt, eine andere GUI-Bibliothek zu verwenden, ohne die Funktionalität der IDE zu beeinträchtigen. Einzig *Tkinter* scheint mit allen integrierten Entwicklungsumgebungen zu harmonieren. Umgekehrt ist *WingIDE* die einzige Umgebung, die auch andere GUIs zulässt. Als Ursache dieses Problems wurde die Ereignisbearbeitung erkannt. Innerhalb einer Anwendung dürfen nicht mehrere unabhängige Ereignisschleifen gestartet werden.

Bei der Nutzung von *Qt* in *PyNass*i wurden anfangs Redraw-Probleme und Abstürze beobachtet. Die Probleme waren den Autoren des PyQt-Moduls bekannt und sind mittlerweile behoben. MacOS X wird von *PyQt* in der vorliegenden Version nicht voll unterstützt, an einer Anpassung wird gearbeitet.

Probleme bei nachträglichen Erweiterungen

Bei der nachträglichen Implementierung der Druckfunktionen stellte sich heraus, dass ein *Canvas* (Vektor-Zeichenfläche) keine Druckfunktionalität besitzt. Daher war eine zusätzliche Implementierung der Grafikausgabe über einen *Painter* (Pixelzeichenfläche) nötig. Dieses unerwartete Problem hätte durch eine zeitige Einplanung der Druckfunktionalitäten vermieden werden können und zeigt exemplarisch die Grenzen der schnellen Softwareentwicklung: Nachträgliche Erweiterungen sind zwar oft problemlos möglich, führen aber zu nicht unerheblichen Änderungen des Entwurfs mit erhöhtem Implementierungsaufwand.

Internationalisierung

Neben der Druckfunktion wurde auch die Möglichkeit zur Internationalisierung erst später ergänzt. Die erforderlichen Änderungen der Quelltexte waren schnell durchgeführt: Markierung aller Textstellen mit der Übersetzungsfunktion `_()` und Einbindung des Moduls *gettext.*

Quelltext ohne Übersetzung	Quelltext mit Übersetzung
`print 'Hello World'`	`print _('Hello World')`

Tabelle 8: *Übersetzung mit gettext*

Probleme bereitete anschließend die korrekte Installation und Nutzung der GNU-Werkzeuge unter Windows, diese waren zunächst nur in einer UNIX-Variante auffindbar. In der kostenlosen *Cygwin*-Distribution wurde schließlich eine fertig compilierte Windows-Version gefunden. Die Dokumentation der Python-*gettext*-Bibliothek beschränkt sich auf die Beschreibung und Beispiele zu den Methoden. Hinweise zur Installation, zum Einrichten der gewünschten Verzeichnisstruktur und zur korrekten Nutzung der Programmteile über die Kommandozeile mussten aus der GNU-Dokumentation zusammengesucht werden. Hier besteht noch dringender Bedarf nach einer Dokumentation, die alle Schritte beschreibt. Ist das Vorgehen dann einmal bekannt, bereitet die weitere Arbeit keine Probleme. Zur Bearbeitung der Übersetzungstexte stehen sowohl für Windows wie auch UNIX leicht bedienbare Programme zur Verfügung.

6.6 Kritische Meinungen von Python-Anwendern

Nachteile einer Sprache oder eines Entwicklungssystems zeigen sich häufig erst während der intensiven Nutzung. Im folgenden Abschnitt werden Ergebnisse dokumentiert, die von Python-Entwicklern und Lehrkräften auf Nachfrage mitgeteilt wurden.

1. Zitat:
   If it's not popular, it can't be any good"; (read between the lines: "if it didn't come from Microsoft, it's no good"); or the even deadlier slag "it won't help our grads get a job". This leads to pushback from other teachers and some students who feel I'm teaching something totally irrelevant and "out in left field". Visual Basic, C, C++, Java and Turing are the alternatives proposed.

2. Dynamische Typisierung erleichtert zwar den Einstieg in die Programmierung, führt aber bei Wechsel auf streng typisierte Sprachen wie Java oder C zu Problemen.

3. Schüler sind nicht auf die Problematik der Zeilentrennung und Blockbildung in anderen Sprachen vorbereitet.

4. Einem Lehrer fehlte nach Wechsel von Pascal auf Python eine Funktion zum Einlesen von Wertepaaren. In einer langen Diskussion in der Mailingliste wurden alternative Möglichkeiten erarbeitet.

5. Der Interpreter akzeptiert keine einzelnen, mit Leerzeichen eingerückten Anweisungen.

6. Unvermeidbare Ungenauigkeiten durch die Konvertierung zwischen dezimaler Darstellung und interner binärer Darstellung von Fließkommazahlen werden oft als Fehler von Python gedeutet, beispielsweise liefert `4/5.0` die Ausgabe `0.80000000000000004`

7. Listen und Hashes als Parameter von Funktionen arbeiten anders als von manchen erwartet.

8. Die unterschiedlichen Möglichkeiten des Modulimports und der Klassenerzeugung (mal mit, mal ohne Modulname) wirken anfangs verwirrend.

9. Von einem Lehrer wurde mehrfach folgender Fehler bei der Blockbildung durch Einrücken beobachtet:

```
if x == y :
  do some stuff
  else :
    # do something else
# Sorry, I couldn't get it to work. It kept giving me errors
```

10. Die Datums- und Zeitbibliothek ist im Vergleich zu Perl primitiv.

11. Bei Nutzung gleichnamiger globaler und lokaler Variablen ist oft schwierig zu erkennen, ob nun die globale oder lokale Variable abgesprochen wird. Der Geltungsbereich wirkt unklar und kann zu unerwarteten Vertauschungen oder Fehlern führen.

12. Es fehlt für Python eine integrierte Entwicklungsumgebung wie etwa für Delphi, in der man mit wenigen Mausklicks ein Programm mit GUI erstellen kann.

Kommentar:

Die ersten vier genannten Kritikpunkte sind durch Festhalten an alten Vorstellungen, Gewohnheiten und Konzepten begründet.

Problem 5 bis 7 können als Mangel von Python anerkannt werden, eine Änderung sollte den Entwicklern vorgeschlagen werden.

Problem 8 und 9 sind ein typisches Anfängerproblem. Vergleichbare Probleme sind in anderen Sprachen auch vorhanden, wobei Notationsprobleme bei der Blockbildung in anderen Sprachen vermutlich weit häufiger vorkommen als in Python.

Problem 10 lässt sich durch geeignete Bibliotheken umgehen.

Über Problem 11 wird derzeit in Entwicklerkreisen diskutiert. Es bleibt abzuwarten, welche Ergebnisse dieser Diskussionsprozess hat.

Zu 12: In dieser Arbeit wurden zahlreiche integrierte Entwicklungsumgebungen vorgestellt, die Delphi kaum nachstehen. Des Weiteren sollte diskutiert werden, ob das „Zusammenklicken" eines Programms Ziel des Unterrichts sein sollte, denn es besteht hier die Gefahr, dass wichtige Konzepte der Informatik wie Modellierung und Algorithmik nicht ausreichend behandelt werden.

7 Zusammenfassung und Ausblick

Die Sprache ist ein unvollkommenes Werkzeug.
Die Probleme des Lebens sprengen alle Formulierungen.

Antoine de Saint-Exupéry (1900 – 1944),
französischer Humanist, Romancier, Erzähler und Flieger

In dieser Arbeit wurde Python auf die Eignung zur schnellen Softwareentwicklung und für den Einsatz zur informatischen Bildung untersucht. Dazu wurde ein Kriterienkatalog aufgestellt, anhand dessen Python analysiert wurde.

Kriterien	für schnelle Entwicklung	für informatische Bildung	Eignung von Python
Portabilität und Dateizugriff	✓	✓	++
Schneller GUI-Entwurf	✓	○	++
Geeignete Hilfsmittel und Werkzeuge	✓	✓	++
Einfache Syntax und Semantik	✓	✓	++
Integrierte Datentypen	✓	✓	++
Funktionskonzept	✓	✓	++
Modularisierung und Wiederverwendbarkeit	✓	✓	++
Refactoring	✓	✓	++
Dokumentation	✓	✓	++
Softwaretest	✓	✓	++
Fehlertoleranz	✓	✓	++
Fehlersuche und Vermeidung	✓	✓	++
Klare Fehlermeldungen	✓	✓	++
Nebenläufigkeit (Threading)	✓	✓	+
Persistente Datenspeicherung	✓	✓	++
Erweiterungsfähigkeit und Einbettung	✓	×	++
Automatische Speicherverwaltung	✓	✓	++
Objektorientierte Entwicklung	✓	✓	++
Funktionale Entwicklung	○	✓	+
Prädikative Entwicklung	○	✓	-
Einfache Lesbarkeit und Erlernbarkeit	○	✓	++
Austausch von Algorithmen	○	✓	++
Orthogonale Syntax	○	✓	++
Eingabehilfen	✓	✓	++
Lernumgebung	○	✓	+
Motivation	✓	✓	++
Leistung (Laufzeiteffizienz)	○	×	○

✓ist erforderlich, ○ ist nützlich, × ist unwichtig

++ sehr gut geeignet, + gut geeignet, ○ ist geeignet, - eingeschränkt geeignet, – nicht geeignet

Tabelle 9: *Auswertung*

Tabelle 9 zeigt eine Übersicht der überprüften Kriterien. Es erfolgt eine Darstellung der Bedeutung der einzelnen Kriterien für die beiden überprüften Einsatzgebiete sowie die in dieser Arbeit ermittelte Beurteilung der jeweiligen Eigenschaft in Python. Die Beurteilung erfolgte durch Vergleiche der Merkmale zu anderen Sprachen sowie Auswertung empirischer Untersuchungen, Umfragen und Fallstudien.

Es konnte in dieser Arbeit gezeigt werden, dass Python den gestellten Kriterien weitestgehend gerecht wird. Die Fallstudien und empirischen Untersuchungen belegen, dass Python auch im praktischen Einsatz eine geeignete Sprache ist.

Besondere Stärken liegen in der einfachen, Pseudocode-ähnlichen Syntax und Semantik, der Integration vieler Datentypen mit dynamischer Typisierung sowie in der Funktions- und Bibliotheksnutzung. Die Syntax erweist sich weitestgehend als widerspruchsfrei und orthogonal, womit ein wichtiges Kriterium für die Auswahl einer Programmiersprache erfüllt ist.

Zusätzlich zeigt sich die Integration mehrerer Paradigmen der Modellierung unter dem Dach einer Sprache als besonders interessante Eigenschaft von Python. Dies ist bezüglich der prozeduralen, objektorientierten und funktionalen Umsetzung gut gelungen. Prädikative Entwicklung ist grundsätzlich möglich, allerdings derzeit noch nicht auf dem Niveau einer rein prädikativen Programmiersprache wie Prolog. Hier besteht noch Entwicklungsbedarf, jedoch wird an diesen Projekten bereits intensiv gearbeitet. Zu möglichen Problemen im fachdidaktischen Einsatz, beispielsweise durch Mischungsgefahr der Paradigmen, besteht noch Forschungsbedarf.

Für beide Interessensgruppen (professionelle Entwickler und Fachdidaktik) sind geeignete Werkzeuge und Lernumgebungen in Form von Tutorien und Modellwelten für den gesamten Softwareentwicklungsprozess vorhanden. Zusätzlich wurden sie noch mit den in dieser Arbeit entwickelten Fallstudien *PyNassi* und *SuM* ergänzt.

Es wurden auch Bereiche identifiziert, in denen Python nur bedingt geeignet ist. So bleibt die systemnahe Programmierung Low-Level-Sprachen wie C vorbehalten, wobei die Nutzung von systemnah implementierten Modulen Teil des Konzeptes von Python ist. Des Weiteren gibt es Bereiche, in denen automatische Speicherverwaltung und dynamische Typisierung nicht erwünscht sind. Jedoch sind diese Gebiete ausserhalb der untersuchten Arbeitsfelder der schnellen Softwareentwicklung und (einführenden) informatischen Bildung einzuordnen.

Insgesamt hat sich in dieser Arbeit gezeigt:

- **Die Anforderungen an eine Programmiersprache zur schnellen Softwareentwicklung und zur informatische Bildung stimmen in vielen Bereichen überein. Es ist möglich, beiden Anforderungen gleichermaßen gerecht zu werden.**
- **Python und verfügbare Werkzeuge eignen sich sowohl zur schnellen Softwareentwicklung als auch zum Einsatz in der informatischen Bildung.**

Ausblick:

Bereits eine große und wachsende Zahl von Unternehmen und Entwicklern setzt Python zur schnellen Softwareentwicklung ein. Die in diesem Bereich erfassten Erfahrungen sind sehr positiv und bestätigen die Ergebnisse dieser Arbeit. Im informatisch-didaktischen Bereich wurden die Stärken von Python zur Einführung in die Programmierung erkannt. Es ist daher wahrscheinlich, dass Python in Zukunft auch hier verbreitet eingesetzt wird; die positive Bewertung von Python in dieser Arbeit trägt vielleicht auch einen Teil dazu bei.

Die Autoren von Python und der zugehörigen Bibliotheken sowie Werkzeuge arbeiten mit viel persönlichem Einsatz an Verbesserungen ihrer Produkte. Das Open-Source-Konzept erlaubt es jedem interessierten Entwickler, sich daran zu beteiligen. So ist es wahrscheinlich, dass erkannte Probleme in naher Zukunft behoben werden und auch die Lücken bei der prädikativen Softwareentwicklung noch geschlossen werden.

Allerdings soll diese Arbeit nicht dazu verleiten, Python als die einzige „ideale" Programmiersprachen anzusehen. Einerseits werden auch andere Sprachen einem großen Teil der gestellten Kriterien gerecht, andererseits gibt es insbesondere aus fachdidaktischer Sicht noch offene Fragen:

1. Werden bei Integration unterschiedlicher Paradigmen in eine Programmiersprache die jeweiligen Konzepte als eigenständige Idee erkannt?
2. Wie aufwendig ist das spätere Erlernen „niedriger" Programmiersprachen wie C++ mit strenger Typisierung und ohne Speicherverwaltung?

Erst der praktische Einsatz im Unterricht wird diese Fragen beantworten können. Insbesondere in den Niederlanden wird Python bereits häufig als erste Programmiersprache eingesetzt, mit überwiegend positiven Erfahrungen sowohl seitens der Lehrer als auch der Schüler.

Kommentar und Schlusswort:

Es fällt meist nicht schwer, Ausbilder und Schüler von den Fähigkeiten der Programmiersprache Python zu überzeugen. Immer wieder ist jedoch die Frage nach dem praktischen Nutzen zu hören: *Der Markt sucht derzeit Entwickler für XYZ, und XYZ ist weit verbreitet. Warum soll ich dann Python lehren (bzw. lernen)? Ich möchte auch auf den Markt vorbereiten.*

Wie diese Arbeit zeigt, eignet sich Python hervorragend zum Erlernen von Konzepten. Sind Konzepte einmal verstanden, fällt die Adaption auf andere Sprachen leicht. Programmiersprachen kommen und gehen, die grundlegenden Konzepte hingegen bleiben aber gleich. Daher sollte eine Programmiersprache gewählt werden, welche die Konzepte unterstützt, und nicht primär auf den aktuellen Markt geachtet werden. In wenigen Jahren können die Marktanforderungen sich komplett wandeln. Bereits heute werden aufgrund der gewachsenen Bedeutung der schnellen Softwareentwicklung Entwickler in diesen Bereichen gesucht, daher sollte man die zukünftige Bedeutung von Skriptsprachen wie Python nicht unterschätzen.

Es bleibt daher zu hoffen, dass Entscheidungsträger die Vorteile von Python und vergleichbaren Sprachen gegenüber klassischen Sprachen erkennen und die Ergebnisse dieser Arbeit bei ihrer Auswahl berücksichtigen.

A Anhang

A.1 Testfälle

Mit den folgenden Testfällen wurden die Entwurfswerkzeuge geprüft:

- **Bibliothek**
  Das Bibliotheksbeispiel wurde von Frau Prof. Schubert am Lehrstuhl DDI der Universität Dortmund in Lehrveranstaltungen genutzt. Es modelliert objektorientiert eine einfache Buchausleihe mit Verwaltung von Lesern und Büchern. Wegen seiner überschaubaren Objektstruktur eignet es sich als einfacher Testfall für Diagrammeditoren und Werkzeuge mit Quelltextanalyse.

- **PyNassi**
  PyNassi macht umfangreich Gebrauch von Vererbung und im Quelltext integrierter Dokumentation. Es wurde daher als Testfall für Quelltextanalyse und zur Dokumentationserstellung eingesetzt.

- **Mixin**
  Mixin-Klassen werden nur von wenigen Programmiersprachen unterstützt. Daher wurden Diagrammeditoren auch mit dieser Art von Klassenstruktur getestet.

- **Pizza**
  Ein Beispiel, welches oft im Ausbildungsbereich verwendet wird: Eine Pizza soll bestellt werden, hierbei soll Auswahl zwischen verschiedenen Größen und Kombinationen von Belägen möglich sein. Hier wurde nicht auf fertigen Quelltext zurückgegriffen, sondern versucht, diese Problemstellung zu modellieren und im Entwicklungssystem umzusetzen. So wurde das Pizza-Beispiel sowohl zum Test von IDEs wie auch der unterschiedlichen Paradigmen verwendet. [BS97]

A.2 Plattformen

Python ist für folgende Plattformen verfügbar:

- **Wichtigste Varianten**
  Windows 95, 98, ME, NT, 2000, XP
  Windows NT/Alpha
  Windows CE
  MacOS X
  Linux
  Solaris
  diverse UNIX-Derivate
  Java (via Jython)

- **Weitere Varianten**
  DOS
  Windows 3.1
  OS/2

Amiga
BeOS
QNX
VMS
Psion
Acorn
VxWorks
IBM AS/400
Playstation
Sharp Zaurus
Palm OS
Virtuelle Maschine

A.3 Integrierte Entwicklungsumgebungen

IDLE

IDLE ist eine von Python-Entwickler Guido van Rossum entwickelte integrierte Entwicklungsumgebung für Python und ist bereits im Python-Programmpaket enthalten. Der Funktionsumfang ist minimal aber zweckmäßig: Editor mit Syntax-Hervorhebung, Python-Shell und Debugger. *IDLE* verwendet nur Python und Tkinter und ist daher auf nahezu allen Plattformen nutzbar. Spezielle Funktionen zur Modellierung oder Dokumentation sind nicht enthalten, durch den eingebauten Modul- und Klassenbrowser wird allerdings die Nutzung von (fremden) importierten Modulen und Klassen unterstützt.

Zusammen mit Zusatzprogrammen wie *Dia* kann *IDLE* eine gute Alternative zu den großen Entwicklungsumgebungen sein. Durch übersichtliche und nicht überladene Menüs bietet sich der Einsatz zu Ausbildungszwecken an.

Komodo

Eine weitere kommerzielle Entwicklungsumgebung stellt ActiveState mit *Komodo* [Komodo02] zur Verfügung. *Komodo* enthält einen Editor mit Syntax-Hervorhebung, Projektübersicht, Debugger und Syntax-Prüfung. Interessant ist *Komodo* wegen seiner Unterstützung zahlreicher Programmiersprachen: Neben Python sind auch Perl, PHP, Tcl, XSLT und zahlreiche andere Sprachen unter Linux und Windows nutzbar. Funktionen zur Modellierung oder Dokumentation sind nicht vorgesehen, hier ist der Entwickler auf weitere Werkzeuge angewiesen.

ActivePython

ActiveState stellt mit *ActivePython* [ASPY02] eine ausgereifte IDE für Windows und Linux kostenlos zur Verfügung. Es enthält in Ergänzung zur Python-Kerndistribution Module zur XML-Verarbeitung, die *Zlib* zur Datenkompression, eine Sammlung an Win32-Modulen von Mark Hammond mit Unterstützung für COM, ASP und Windows MFC.

Das Installationsprogramm installiert die vollständige IDE inklusive Python, so ist man auf die Kerndistribution nicht mehr angewiesen. Die IDE vereint einen Editor mit Syntax-Hervorhebung mit Interpreter, Debugger sowie umfangreicher Online-Dokumentation.

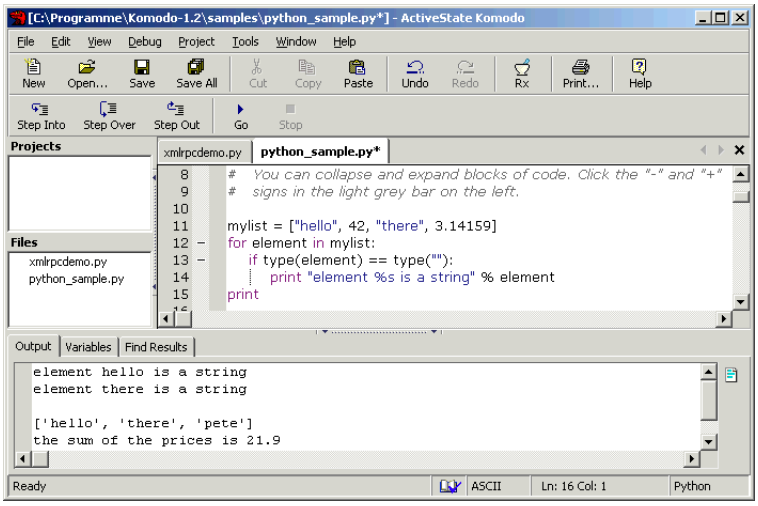


Abbildung 17: *Komodo*

Durch die umfangreiche Ausstattung mit Bibliotheken und zugehöriger Dokumentation spricht *ActivePython* besonders Entwickler unter Microsoft Windows an, denn aufwendiges Installieren vieler Softwarepakete kann entfallen.

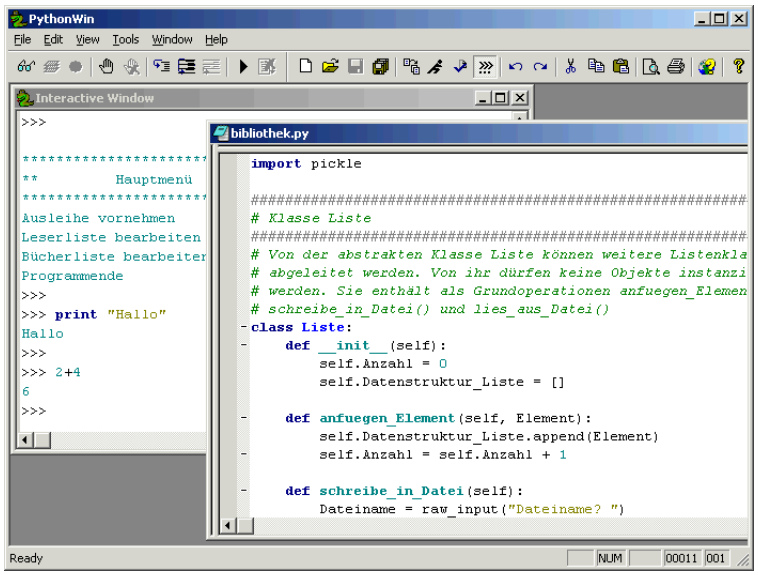


Abbildung 18: *ActivePython*

Wing-IDE

Die kommerzielle Entwicklungsumgebung *Wing-IDE* [Archaeopteryx02] bietet neben einem mehrsprachenfähigen Editor mit Quelltext-Browser und Interpreter einen netzwerktauglichen Debugger. Letzterer ist die Stärke dieses Produktes: Der Debugger unterstützt Fehlersuche in nicht lokal laufenden Programmen, überwacht Netzwerkverbindungen und eingebettete Skripte. Darüber hinaus arbeitet *Wing-IDE* als einzige Entwicklungsumgebung problemlos mit

GUI-Anwendungen aller Bibliotheken zusammen, darunter *Tkinter*, *Pygtk*, *PyQt* und *wxPython*. Damit wird *Wing-IDE* zu einem geeigneten Werkzeug für Entwickler von CGI-Skripten und Anwendungen in Client-Server-Umgebungen. Zu erwähnen ist noch die integrierte Projektverwaltung, die eine Entwicklung durch mehrere Benutzer ermöglicht, sowie die Möglichkeit, auch andere Programmiersprachen nutzen zu können.

Nach Meinung des Autors ist die Wing-IDE die derzeit beste Wahl für erfahrene Entwickler, sofern auf integrierte Modellierungs- und GUI-Gestaltungshilfen verzichtet werden kann. Dafür bekommt der Entwickler den derzeit besten Python-Debugger und ein ausgereiftes Werkzeug, welches gut mit allen Bibliotheken und anderen Werkzeugen wie beispielsweise dem *Qt Designer* harmoniert.

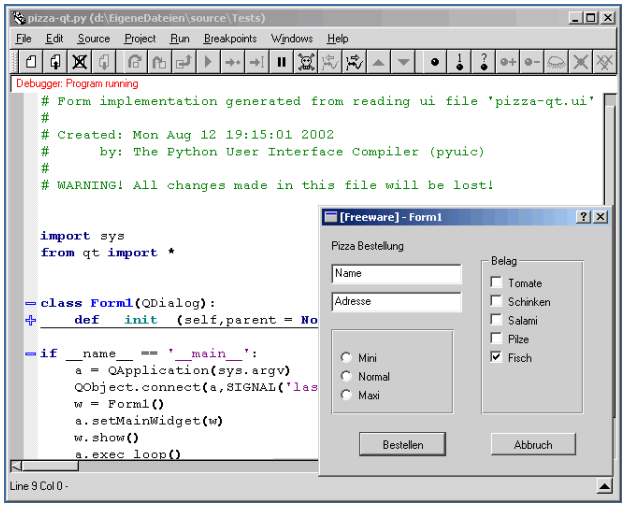


Abbildung 19: *Wing-IDE*

PyCrust

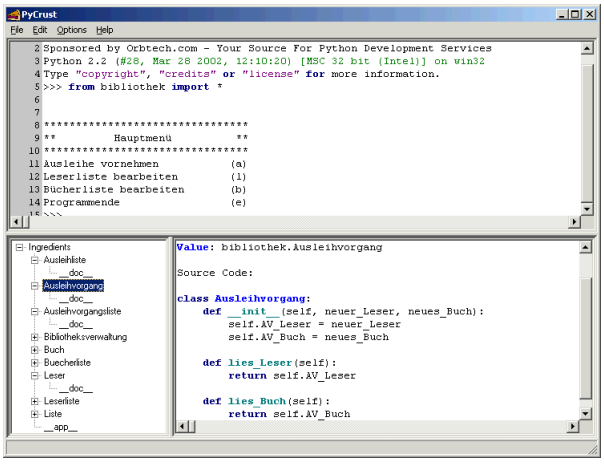


Abbildung 20: *PyCrust*

PyCrust [Pobrien02] ist eine in Python geschriebene Shell mit Pythoninterpreter. Es verwendet das GUI *wxPython*. *PyCrust* integriert den Interpreter in eine grafische Benutzungsoberfläche unter *wxPython* und erweitert die Möglichkeiten der Codeeingabe. Neben verbesserten Eingabefunktionen inklusive automatischer Vervollständigung bietet es einen ständigen Überblick über alle geladenen Klassen und ihre Dokumentation.

A.4 Weitere Python-Werkzeuge

PyUT

Von einer Schweizer Studentengruppe wurde *PyUT* [Dutoit02] entwickelt, ein einfacher Editor für UML-Klassendiagramme. Der Funktionsumfang umfasst die Gestaltung der Diagramme sowie Import und Export in den Formaten XMI, XML, JPEG und Postscript.

Das Programm ist modular aufgebaut und beherrscht Quelltexterzeugung für C++, Java, Python und SQL sowie Rückwärtsanalyse für Java und Python.

PyUT wurde vollständig in Python programmiert. Da der Quelltext verfügbar ist und die Schnittstellen zu Modulen gut dokumentiert wurden, sind eigene Erweiterungen möglich. *PyUT* ist mehrsprachenfähig und unterstützt derzeit Englisch und Französisch. Die Entwicklung ist noch nicht abgeschlossen, aber bereits jetzt erscheint das Programm für praktische Einsätze gut geeignet.

Thorn

Thorn ist ein UML Diagrammentwurfswerkzeug, speziell entworfen für Open-Source-Projekte. Als Dateiformat wird XML verwendet. Thorn unterstützt Scripting mit Jython.

Leo

Leo [Ream02] ist eine Projektverwaltung, geschrieben in Python mit dem GUI *Tkinter*. Es besteht aus einem Editor sowie einem Browser für Projekte, Programme, Klassen und andere Daten. Es soll Entwurf, Codierung, Test und Fehlersuche erleichtern. Leo unterstützt XML und ist zu *Noweb* sowie *Cweb* kompatibel.

Angekündigte Projekte

Weitere Projekte befinden sich derzeit in Entwicklung.

- *Umlaut* ist als einfaches Diagramm-Zeichenwerkzeug konzipiert.
- *Pyidea* soll Musterentwurf erlauben und Anbindung an *Swing* sowie *wxWindows* bekommen.

A.5 Kommentare von Python-Anwendern

Eine Auswahl zahlreicher Kommentare auf Anfragen in Newsgroups und unter Entwicklern:

- *Syntax is a big plus for Python. Functions have named arguments (no need for @ unpacking). Lists and dictionaries (associative lists) can contain arbitrary items including other lists and dictionaries. The syntax for objects and exception handling are clear and readable.*
- *Python übt auf mich den Reiz eines Fischertechnik-Baukastens aus. Immer wieder entdeckt man neue Möglichkeiten. Alles passt hervorragend zueinander. So wie man sein Fischertechnik nach und nach um Ausbaukästen erweiterte, findet man jeden Tag ein neues interessantes Python-Merkmal oder Modul. (S. Kirchberg, Softwareentwickler)*
- *I think Perl's syntax is probably a bit more efficient for quickie text processing scripts, but it starts to get in the way for larger things. By comparison, Python is good for small scripts and also good for larger tasks. It scales nicely and is readable.*
- *Nothing is perfect. Things I don't like about Python include (everybody will have their own list, of course):*
  - *The type/class unification problem. This is much improved in recent Python but still has further to go.*
  - *There is a sometimes uncomfortable (to me) mix of global functions, operators, etc., meaning more trips to the docs than I'd like. For instance collection classes don't have a* `len` *or* `length` *method, you call the global function* `len` *on them. As a result, it is sometimes challenge finding the way to do something. Still, it is better than it was. At one time there was a string module full of functions rather than methods on string objects.*
- *Wenn es nicht besondere Gründe gibt, Perl zu verwenden, sollte man ein Projekt mit Python in Angriff nehmen.*
- *I was generating working code nearly as fast as I could type. When I realized this, I was quite startled. An important measure of good language design is how rapidly the percentage of missteps of this kind falls as you gain experience with the language. When you're writing working code nearly as fast as you can type and your misstep rate is near zero, it generally means you've achieved mastery of the language. But that didn't make sense, because it was still day one and I was regularly pausing to look up new language and library features!* [Raymond00]
- *Ja, auch unsere Programmierer dokumentieren nicht vernünftig, aber da wir mit Python eine "lesbare" Programmiersprache gewählt haben, erklärt sich der Quellcode von selbst." (W. Appelt, Fraunhofer Inst.)*

A.6 Quelltexte

Beispiel zum Modulimport:

```
# Dies ist Modul n.py
# mit einer globalen Variable "var"
var = 1
def pv():
  print "im Untermodul: var ist", var
```

Variante 1

```
import n
print "vorher:", n.var
n.var = 17
n.pv()
print "nachher", n.var
```

Variante 2

```
from n import *
print "vorher:", var
var = 17
pv()
print "nachher:", var
```

Variante 1 liefert die Ausgabe

```
vorher: 1
im Untermodul: var ist 17
nachher: 17
```

Variante 2 liefert die Ausgabe

```
vorher: 1
im Untermodul: var ist 1
nachher: 17
```

In Variante 1 ändert `n.var=17` die Variable innerhalb des Moduls, daher ist sowohl aus Sicht des Moduls wie auch des außerhalb `n.var` nun `17`.

In Variante 2 setzt `var=17` eine lokale Kopie. Innerhalb des Moduls ist daher der Wert der hier gleichnamigen internen Variable nach wie vor `1`.

Quelltext singleton.py

```
# Implementierung einer Warte-Schlange (Queue) als Singleton
class Singleton:
    """ Ein Python Singleton """
    class __impl:
        """ Implementierung des Singleton """
        def __init__(self):
            print "Erzeuge neue Schlange"
            self.items=[]
        def append(self,item):
            self.items.append(item)
        def get(self):
            if self.items==[]:
              return None
            else:
              r=self.items[0]
              self.items=self.items[1:]
              return r
        def spam(self):
            """ Test Methode, liefert Singleton id """
            return id(self)
    # Speicher fuer die Instanz-Referenz
    __instance = None
    def __init__(self):
        """ Erzeuge neue Singleton Instanz """
        # Prüfe, ob es schon eine Instanz gibt
        if Singleton.__instance is None:
            # Erzeuge Instanz
            Singleton.__instance = Singleton.__impl()
        # Speichere Instanz-Referenz als einziges Mitglied
        self.__dict__['_Singleton__instance'] = Singleton.__instance
    def __getattr__(self, attr):
        """ Leite Zugriff an Implementierung weiter """
        return getattr(self.__instance, attr)
    def __setattr__(self, attr, value):
        """ Leite Zugriff an Implementierung weiter """
        return setattr(self.__instance, attr, value)
# Test
s1 = Singleton()
print id(s1), s1.spam()
s2 = Singleton()
print id(s2), s2.spam()
s1.append(1)
s1.append(2)
s1.append(3)
print s2.get()
print s2.get()
print s1.get()
```

Quelltext Permutation

```
# Diese Funktion liefert zu einer Sequenz und einem Wert index die <in-
dex>te Permutation.
# Es gibt n! Permutationen.
# Aus: [PCOOK02]
def getPerm(seq, index):
    "Liefert die <index>te Permutation von <seq>"
    seqc = list(seq[:])
    seqn = [seqc.pop()]
    divider = 2
    while seqc:
        index, new_index = index//divider, index%divider
        seqn.insert(new_index, seqc.pop())
        divider += 1
    return seqn
```

Quelltext zum objektorientierten Pizza-Beispiel pizza-oo.py

```
# Pizza-Beispiel
# Objekt-orientierter Ansatz, nach [Brennwald97]
# Python-Version von Ingo Linkweiler 07/2002
class Zugaben:
  def setze_Zustand(self,zustandneu):
    self.zustand=zustandneu
class Tomate(Zugaben):
  def schaelen(self):
    self.setze_Zustand("geschält")
  def hacken(self):
    self.setze_Zustand("gehackt")
class Mozzarella(Zugaben):
  def zerkleinern(self):
    self.setze_Zustand("zerkleinert")
class Ingredienzen:
  # Attribute: tomate, kaese, gewuerz
  def __init__(self):
    self.tomaten=[]
    self.kaese=[]
    self.gewuerz=""

  def wuerzen(self,gewuerzt_mit):
    self.gewuerz=gewuerzt_mit

  def einkaufen(self):
    for i in range(2):
      zutat=Tomate()
      zutat.setze_Zustand("frisch")
      self.tomaten.append(zutat)
      zutat=Mozzarella()
      zutat.setze_Zustand("frisch")
      self.kaese.append(zutat)
```

```
  def vorbereiten(self):
    for i in self.tomaten:
      i.schaelen()
      i.hacken()
    for i in self.kaese:
      i.zerkleinern()
class Pizzateig:
  def einkaufen(self):
    self.radius=5
    self.dicke=4.5
    self.zustand="geknetet"

  def ausrollen_auf(self, sollradius, solldicke):
    while (self.radius<sollradius) and (self.dicke>solldicke):
      self.dicke=self.dicke / 1.5
      self.radius=self.radius+2
      self.zustand="ausgerollt"

class Pizza:
  def Anfangszustand(self,neuzustand):
    self.zustand=neuzustand
  def belegen(self, teig, ingredienzen):
    self.margheritaTeig = teig
    self.beigaben = ingredienzen
    self.zustand = "belegt"
    self.beigaben.wuerzen("Pfeffer")
  def backen(self, dauer, temperatur):
    self.ofentemperatur = temperatur
    self.backzeit=0
    while self.backzeit < dauer:
      self.backzeit = self.backzeit + 1
    self.zustand="gebacken"
# Programm-Aufruf:
teig = Pizzateig()
teig.einkaufen()
zutaten=Ingredienzen()
zutaten.einkaufen()
margherita=Pizza()
margherita.Anfangszustand("unbearbeitet")
zutaten.vorbereiten()
teig.ausrollen_auf(15,0.5)
margherita.belegen(teig, zutaten)
margherita.backen(20,250)
print "Fertig"
```

A.7 Anforderungen an Programmiersprachen

Es folgt eine Auswahl von Lehrern genannter Anforderungen an Programmiersprachen. Diese wurden in Umfragen sowie Diskussionen ermittelt und wurden bei der Aufstellung des Kriterienkataloges berücksichtigt.

- Beitrag zur Allgemeinbildung
- Vermeidung spezieller Syntaxregeln
- Bezahlbare Softwarepakete
- Typenstrenge (FH / Universität) *
- Unterstützung der Sprache durch mehrere Hersteller
- Automatische Speicherverwaltung für Systemprogrammierung an Universitäten nicht wünschenswert *
- Syntaktische Sauberkeit
- Saubere Trennung von Vererbungssichtbarkeit und Modul / Paketsichtbarkeit *
- Verschleierung von Konzepten durch die Syntax
- Vermeidung der Notationsoptimierung auf Kosten der Verständlichkeit (z. B. C)
- Leichter, schneller Einstieg und schnell sichbare Resultate
- Vermeidung von Schreibarbeit, aber Nachvollziehbarkeit eines jeden Mausklicks (bzgl. IDEs)
- Einbettung in eine gute Entwicklungsumgebung
- Unabhängigkeit von Betriebssystem und Hardware
- Textuelle Sprachen sind in der Einführung oft unwichtig, Modellierung ist auch über UML möglich

* Diese Anforderungen sprechen möglicherweise gegen die Nutzung von Python. Dies wird bei der Diskussion der Kriterien in dieser Arbeit berücksichtigt.

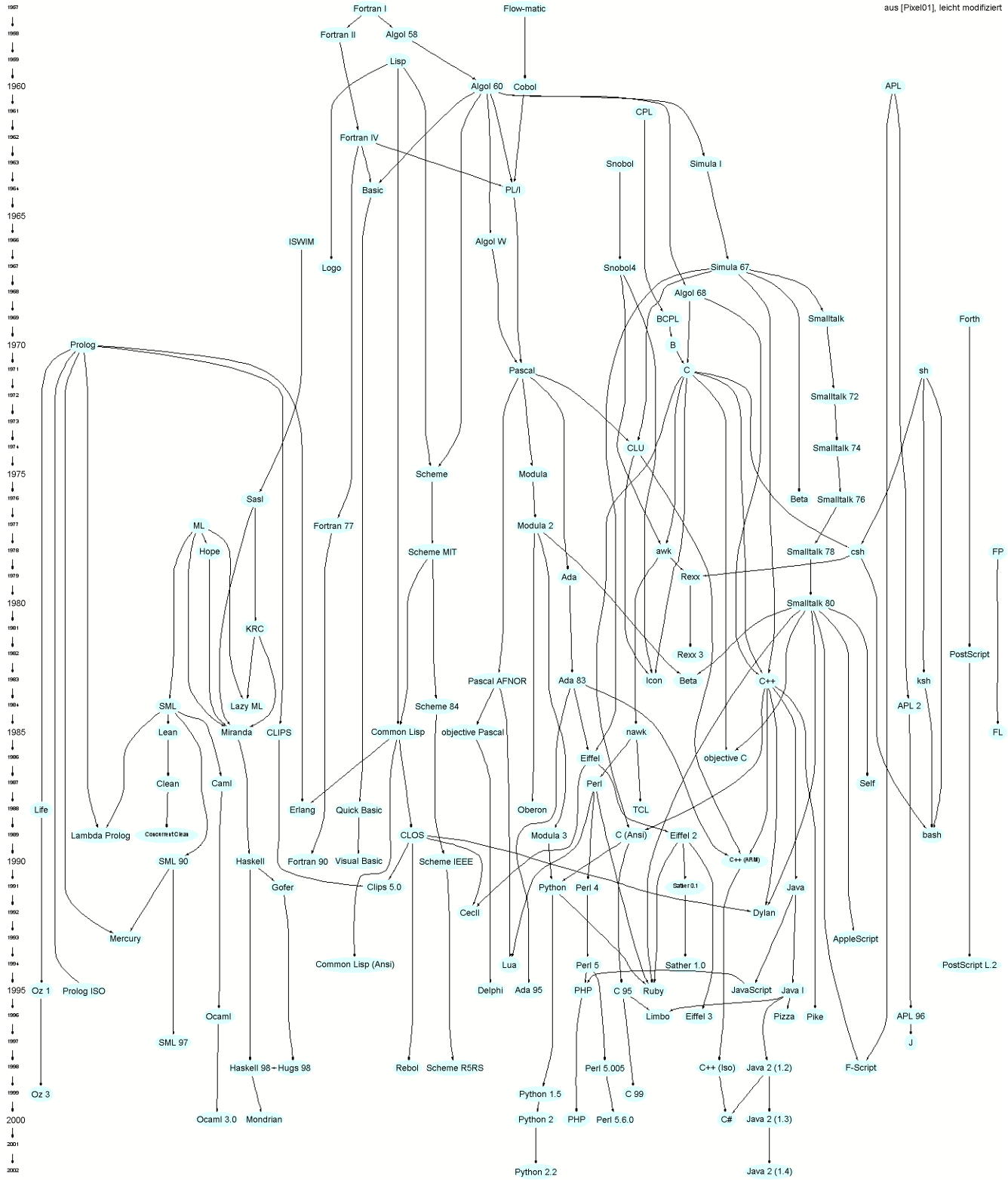
A.8 Umfrageergebnisse

Sprache	**Arithmetischer Mittelwert**	**Geometrisches Mittel**	**Anzahl Wertungen**	**Std.-Abweichung**
Ada	2,62	2,37	11	1,21
APL	5,0	5,0	10	0,0
Beta	3,0	2,93	5	0,71
C++	3,58	3,52	26	0,64
Cobol	2,44	2,25	9	0,88
Delphi	2,41	2,3	11	0,66
Eiffel	2,36	2,28	7	0,63
Forth	4,09	3,97	11	0,94
Fortran	3,14	3,05	14	0,77
Java	2,69	2,62	26	0,62
Lisp	3,58	3,4	20	1,04
Modula	2,3	1,95	15	1,36
Pascal	2.14	1,98	22	0,77
Perl	4,16	4,12	25	0,61
Prolog	2,9	2,76	10	0,99
Python	1,66	1,51	19	0,75
SML	2,4	2,22	10	0,84
Scheme	3,5	3,3	13	1,19
Smalltalk	2,69	2,54	8	0,96
Tcl	2,61	2,47	18	0,85
Visual Basic	2,67	2,47	15	0,98
C	2,64	2,46	7	0,94
Ruby	1,88	1,73	4	0,85
Ocaml	2,33	2,29	3	0,58
Basic	2,33	2,08	3	1,15
Algol60	1,5	1,44	3	0,4

1 = Sehr gut lesbar – 5 = nahezu unlesbar.

Tabelle 10: *Umfrage zur Lesbarkeit von Programmiersprachen*

A.9 Diagramm zur Entwicklung der Programmiersprachen



Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Acebal01] Acebal, Cesar F.; Cueva, Lovelle: *A New Method of Software Development - eXtreme Programming.* In: Upgrade, `http://www.upgrade-cepis.org/issues/2002/2/upgrade-vIII-2.html` . Frankfurt: Cepis - 10/2002

[Alice02] Alice: *Free, interactive 3D Graphics.* Carnegie Mellon University - `http://www.alice.org` - 06/2002

[AnyGUI02] Hetland, Magnus Lie: *Anygui.* `http://anygui.sourceforge.net/` - 06/2002

[Apple02] Apple Computer: *Hypercard.* `http://www.apple.com/hypercard/` - 10/2002

[Archaeopteryx02] Archaeopteryx Software: *Wing IDE for Python.* `http://archaeopteryx.comwingide/` - 05/2002

[ASPY02] Active State: *ActivePython*, `http://www.activestate.com/Products/ActivePython` - 05/2002

[Bagley01] Bagley, Doug: *The Great Computer Language Shootout.* `http://www.bagley.org/~doug/shootout/` - 07/2001

[Barnes84] Barnes, J. G. P.: *Programming in Ada.* Zweite Auflage Boston: Addison-Wesley Publishing Company - 1984, ISBN 0201137992

[Balzert79] Balzert, Helmut: *Informatik II, Lösungsband mit methodisch-didaktischer Einführung.* München: Hueber Holzmann Verlag - 1979

[Balzert83] Balzert, Helmut: *Von Problem zum Programm Hauptband.* Zweite Auflage, München: Hueber Holzmann Verlag - 1983

[Balzert00] Balzert, Helmut: *Lehrbuch der Software-Technik 1+2.* Heidelberg: Spektrum Akademischer Verlag - 2000

[Beck00] Beck, Kent: *Extreme Programming, Das Manifest.* Zweite Auflage, Bonn: Addison Wesley - 09/2000, ISBN 3827317096

[Beck99] Beck, Kent: *Extreme Programming Explained.* Zweite Auflage, Bonn: Addison Wesley - 10/1999, ISBN 0201616416

[Biopy02] BioPython Project: `www.biopython.org` - 10/2002

[Booysen02] Booysen, Riaan: *Boa Constructor.* `http://boa-constructor.sourceforge.net/` - 10/2002

[Boszormenyi98] Böszörmenyi, Laszlo: *Why Java is not my favorite first-course language*, in: Software - Concepts & Tools. Berlin: Springer Verlag - 1998, ISSN 0945-8115

[BS97] Brennwald, Daniel; Stamm, Christoph: *Gruppenunterricht zum Thema Paradigmen von Programmiersprachen.* Technische Hochschule Zürich. `http://www.educeth.ch/informatik/puzzles/paradigmen/docs/para.pdf` - 07/2002

[BSW02] Baldassarre, Teresa; Succi, Giancarlo; Williams, Laurie: *Pair Programming, an Extreme Programming Practice.* North Carolina State University. `http://pairprogramming.com/` - 06/2002

[Burley02] Burley, Brent: *Python Bibliotheca, Python resources for teachers and students*, `http://www.ibiblio.org/obp/pyBiblio/interactive.php` - 06/2002

[Church42] Church, Alonzo: *Introduction to Mathematical Logic.* 1944, ISBN 0527027294

[CS01] Claus, Volker; Schwill, Andreas: *Duden Informatik, ein Fachlexikon für Studium und Informatik.* 3. Auflage, Meyers Lexikon. Mannheim, Leipzig, Wien, Zürich: Bibliographisches Institut - 2001

[Clips01] Vladimir: *CLIPS Wrapper.* `http://starship.python.net/crew/gandalf/DNET/AI/` - 06/2002

[CPAN02] CPAN: *Comprehensive Perl Archive Network Search.* `http://search.cpan.org/` - 06/2002

[CU02] *The Rapid Prototyping Model.* Concordia University, ETEC 568/668. `http://www.arts.ualberta.ca/~tchao/rpwebsite/definition.html` - 07/2002

[CWI] Centrum voor Wiskunde en Informatica, Amsterdam. `www.cwi.nl` - 10/2002

[CDH99] Czischke, Dick, Hildebrecht, Humbert, Ueding, Wallos: *Von Stiften und Mäusen.* Bönen: Kettler - Verlag für Schule und Weiterbildung - 1999

[Dal02] Dalheimer, Matthias Kalle : *Programming with Qt.* Köln: O'Reilly Verlag - 02/2002, ISBN 1565925882

[Ohara02] O'Hara, Javier : *Dia2Code Homepage.* `http://dia2code.sourceforge.net/` - 05/2002

[Delord02] Delord, Christophe : *PyLog.* `http://christophe.delord.free.fr/en/pylog/logic.html` - 06/2002

[Dia02] *Dia, a drawing program.* `http://www.lysator.liu.se/~alla/dia/` - 04/2002

[DEM01] Downey, Allen; Elkner, Jeff; Meyers, Chris: *How to Think Like a Computer Scientist*: *Learning with Python.* Green Tea Press - 2001

[Dotfunk02] DOTFUNK.COM *Projektseite Pylint etc.* `http://www.dotfunk.com/projects/` - 07/2002

[Dressler01] DRESSLER, Prof. Dr. P.: *Software Engineering.* Skriptum FH Augsburg, Seite 104ff, `http://www.fh-augsburg.de/informatik/vorlesungen/swe/skript` - 12/2001

[Dutoit02] DUTOIT, Cédric: *PyUt.* `http://sourceforge.net/projects/pyut/` - 06/2002

[Dunn02] DUNN, Robin: *wxPython v.2.3.* `http://sourceforge.net/projects/wxpython` - 05/2002

[Eiffel02] ISE EIFFEL, *In a Class by Itself.* `http://www.eiffel.com` - 06/2002

[Fermilab02] FERMI NATIONAL ACCELERATOR LABORATORY. Batavia. `www.fnal.gov` - 08/2002

[Freidl02] FREIDL, Jeffrey E.F.: *Mastering Regular Expressions.* O'Reilly UK - 07/2002, ISBN 0596002890

[FS1037C] *Federal Standard 1037C,* General Service Administration - 08/1996

[GI2000] *Empfehlung der Gesellschaft für Informatik e.V. für ein Gesamtkonzept zur informatischen Bildung an allgemeinbildenden Schulen.* Informatik Spektrum DDI `http://ddi.cs.uni-dortmund.de/ddi_bib/gi_empfehlung/gesamt2000/gesamtkonzept-26-9-2000.pdf` - 09/2000

[GHJV96] GAMMA, HELM, JOHNSON, VLISSIDES: *Entwurfsmuster.* Bonn: Addison-Wesley - 1996

[Gluit02] GLUIT SOFTWARE: `www.gluit.de`, Castrop-Rauxel - 2002

[Grayson00] GRAYSON, John E.: *Python and Tkinter.* Connecticut: Manning Publications - 01/2000

[GTK02] PYGTK GUI. `http://www.daa.com.au/james/pygtk/` - 06/2002

[Hammond00] HAMMOND, Mark: *Python Programming on Win32.* Köln: O'Reilly Verlag - 01/2000, ISBN 1565926218

[Hellmann02] HELLMANN, Doug: *HappyDoc Source Documentation.* `http://happydoc.sourceforge.net/` - 02/2002

[HM99] HIMSTEDT, Tobias; MÄTZEL, Klaus: *Mit Python programmieren.* Heidelberg: Dpunkt Verlag - 1999, ISBN 3920993853

[Hightower02] HIGHTOWER, Richard: *Python Programming with the Java Class Libraries.* Bonn: Addison-Wesley - 2002, ISBN 0201616165

[Hoffmann02] HOFFMANN, Joe; MARGERUM-LEYS, Jon: *Rapid Prototyping as an instructional design.* `http://www-personal.umich.edu/~jmargeru/prototyping` - 05/2002

[HS02] HUMBERT, Ludger; SCHUBERT Sigrid: *Fachliche Orientierung des Informatikunterrichts in der Sek. II* . Report 771 Universität Dortmund - 02/2002

[Hubwieser02] HUBWIESER, Peter: `http://seciii.cs.uni-dortmund.de/web/ps-object.htm#hubwieser` . Universität Dortmund - 07/2002

[Humbert01] HUMBERT, Ludger: *Informatik lehren – Zeitgemäße Ansätze zur Qualifikation von Schülern.* `http://ddi.cs.uni-dortmund.de:8000/ddi_bib/forschung/pub/Informatik-lehren.pdf` - Universität Dortmund - 07/2002

[Humbert02] HUMBERT, Ludger: *Welche Programmiersprache unterstützt meine Konzepte für den Informatikunterricht?*. Universität Dortmund - 07/2002

[JH02] JÄHNICHEN, Stefan; HERMANN, Stephan: *Was bitte bedeutet Objektorientierung?*. Informatik Spektrum 08/2002, Berlin: Springer Verlag, TU-Berlin - 08/2002

[Jython02] *Jython Home Page*. `http://www.jython.org/` - 06/2002

[Jurgens00] JÜRGENS, Manuela: *LaTeX, eine Einführung*. A/026/0003, Fernuniversität Hagen - 2000

[KR98] KAHLBRAND, ROSSUM: *Objektorientierte Software-Entwicklung mit UML*. Berlin: Springer Verlag - 02/1998, ISBN 354063309X

[Keller01] KELLER, Bryn: *Xoltar Toolkit*. `http://sourceforge.net/projects/xoltar-toolkit/` - 06/2001

[Kniesel00] KNIESEL, Dr. Günter: *Refactoring,* Kapitel 2. Universität Bonn. `http://www.informatik.uni-bonn.de/III/lehre/vorlesungen/SWT/WS2000/resourcen/` - 07/2000

[Kokavecz01] KOKAVECZ, Dr. Bernd: *Mit leichten Schritten in die objektorientierte Programmierung*. `http://www.b.shuttle.de/b/humboldt-os/python/` - 12/2001

[Komodo02] ACTIVE STATE: *Komodo*. `http://aspn.activestate.com/ASPN/` - 05/2002

[LA99] LUTZ, Mark; ASCHER, David: *Learning Python*. O'Reilly UK - 04/1999, ISBN 01565924649

[Liao99] LIAO, Luby: *Python as the Pascal of 2000s*. `http://www.sandiego.edu/~liao/python.pdf` . University of San Diegu - 07/2002

[Liebscher00] LIEBSCHER, Rene: *Konzeption und Aufbau eines modularen Systems zur Simulation von Robotern*. `http://www.informatik.htw-dresden.de/~iwe/Belege/Liebscher/diplom/diplom.pdf` - TH Dresden - 08/2000

[Linkweiler02] LINKWEILER, Ingo: *Aktuelle Informationen zu dieser Diplomarbeit*. `http://www.ingo-linkweiler.de/diplom` - 11/2002

[LF02] Löwis, Martin von; Fischbeck, Nils: *Python 2*. Zweite Auflage, München: Addison Wesley - 2002, ISBN 382731691X

[Logilab02] Fayolle, Alexandre: *Logilab Constraint Package*. `http://www.logilab.org/python-logic/constraint.html` - 06/2002

[Lundh01] Lundh, Fredrik : *Python Standard-Bibliothek*. Boston: O'Reilly - 05/2001, ISBN 96000960

[LRLW01] Lutz, Mark; Rossum, Guido van; Lewin, Laura; Willison, Frank: *Programming Python*. O´Reilly UK - 03/2001, ISBN 0596000855

[Lowis97] Löwis, Martin von: *Internationalizing Python*. Proceedings of the 6th International Python Conference. `http://www.python.org/workshops/1997-10/proceedings/` - Humboldt-Universität zu Berlin - 10/1997

[Mertz01] Mertz, David: *Functional Programming in Python*. IBM Corporation. `http://www-106.ibm.com/developerworks/library/l-prog.html` - 03/2001

[Meyer90] Meyer, Bertrand: *Objektorientierte Softwareentwicklung*. München: Hanser Verlag - 1990, ISBN 3446157735

[Meyers01] Meyers, Chris: *Lisp with Python*. `http://www.ibiblio.org/obp/py4fun/lisp/lisp.html` - 06/2002

[Meyers02] Meyers, Chris: *Prolog with Python*. `http://www.ibiblio.org/obp/py4fun/prolog/prolog.html` - 06/2002

[Monninger93] Monninger, Frieder: *Eiffel: Objektorientiertes Programmieren in der Praxis*. Hannover: Heise Verlag - 1993

[OD02] ObjectDomain: *ObjectDomain R3*. `http://www.objectdomain.com/domain/index.html` - 05/2002

[Opferkuch01] Opferkuch Stefan: *Validierung objekt-orientierter Software*. `http://studweb.studserv.uni-tuttgart.de/studweb/users/inf/inf24723/hase01/Ausabeitung.html` - 10/2001

[OSI02] Open Source Initiative: *The Open Source Definition*. `http://www.opensource.org/docs/definition.php` - 07/2002

[Parn02] Vaults of Parnassus: `http://www.vex.net/parnassus/` - 07/2002

[Parnas01] Parnas, David L.: *Software Fundamentals*. Boston: Addison Wesley - 04/2001

[Padawitz98] Padawitz, Peter: *Einführung in das funktionale Programmieren* - Skriptum. Universität Dortmund. `http://ls5-www.cs.uni-dortmund.de/padawitz.html` - 1998

[Pixel01] Pixel: *Programming Language Study*. Paris: Linux Expo Paris. `http://merd.net/pixel/language-study/` - 2001

[PCOOK02] Active State: *Python Cookbook*. http://aspn.activestate.com/ASPN/Cookbook/Python - 06/2002

[Pobrien02] Pobrien: *PyCrust*. http://sourceforge.net/projects/pycrust/ - 05/2002

[Pedroni02] Pedroni, Samele: *Jython Essentials*. O'Reilly UK - 03/2002, ISBN 0596002475

[PB96] Pomberger, Blaschek: *Grundlagen des Software Engineering*. München: Hanser Verlag - 1996, ISBN 3446186905

[Prechelt00] Prechelt, Lutz: *An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl*. http://www.ipd.uka.de/~prechelt/Biblio/jccpprtTR.pdf - 03/2000

[Preishuber02] Preishuber, Martin: *Log4Py*. http://sourceforge.net/projects/log4py/ - 02/2002

[PSA02] Python Software Activity: http://www.python.org/psa - 05/2002

[PSF02] Python Software Foundation: http://www.python.org/psf/ - 05/2002

[PSIG02] Python Special Interest Groups: http://www.python.org/sig/ - 05/2002

[PyCard02] Python Card: http://pyhoncard.sourceforge.net - 08/2002

[Python02] Python Language Website: http://www.python.org/ - 07/2002

[Rational02] Rational Software: *Rational Rose V.2001*. http://www.rational.com/products/rose/ - 05/2002

[Raymond00] Raymond, Eric S.: *Why Python?* In: Linux Journal. http://www.linuxjournal.com/article.php?sid=3882 . Seattle - 05/2000

[Ream02] Ream, Edward K.: *LEO*. http://personalpages.tds.net/~edream/front.html - 07/2002

[Reissing00] Reissing, Ralf: *Extremes Programmieren*. Informatik Spektrum, Ausgabe 23. Universität Stuttgart - 04/2000

[Rempt01] Rempt, Boudewijn: *GUI Programming with Python: Qt Edition*. OpenDocs - 2002, ISBN 09700330044

[Riverbank02] Riverbank Computing: *Python Qt*. http://www.riverbankcomputing.co.uk/ - 06/2002

[Rossum00] van Rossum, Guido: *IDLE Homepage*. http://www.python.org/idle/doc/ - 05/2000

[Rossum02] van Rossum, Guido: *Benevolent Dictator of Life*. The Perl Review. http://www.theperlreview.com/Issues/The_Perl_Review_0_2.pdf - 04/2002

[Scherer02] Scherer, David: *Vpython: 3D Programming for Ordinary Mortals*. http://www.vpython.org/ - 06/2002

[Schliep02] Schliep, Alexander: *Gato*. http://www.zpr.uni-koeln.de/~gato/ - Universität Köln - 07/2002

[Schubert00] Schubert, Sigrid: *Einführung in die Didaktik der Informatik*, Skript zur Vorlesung. Universität Dortmund, Lehrstuhl Didaktik der Informatik. http://ddi.cs.uni-dortmund.de/lehre/grundstudium/ - 10/2000

[Schubert01] Schubert, Sigrid: *Fundamentale Ideen der Informatik,* Folien zur Vorlesung. Universität Dortmund, Lehrstuhl Didaktik der Informatik. http://ddi.cs.uni-dortmund.de/lehre/hauptstudium/ws2001/ - 08/2001

[SBB02] Schulz-Zander, Renate; Brauer, Wilfried; Burkert, Jürgen; Heinrichs, U.; Hilty, Lorenz M.; Hölz, I.; Keidel, K.; Klages, Albrecht; Koerber, Bernhard; Meyer, M.; Peschke, Rudolf; Pflüger, Jörg; Reineke, Vera; Schubert, Sigrid: *Veränderte Sichtweisen für den Informatikunterricht GI Empfehlungen für das Fach Informatik in der Sekundarstufe II allgemeinbildender Schulen*. In: TROITZSCH, Klaus G. (Hrsg.): *Informatik als Schlüssel zur Qualifikation*. Berlin, Heidelberg: Springer, 1993 (Informatik aktuell). Gesellschaft für Informatik e.V., S. 205–218

[Schwill93] Schwill, Andreas: *Fundamentale Ideen der Informatik*. Fachbereich Informatik, Universität Oldenburg. http://www.informatikdidaktik.de/Forschung/Schriften/ZDM.pdf - 1993

[Schwill02] Schwill, Andreas: *Programmiersprachen im Informatikunterricht*. Fachbereich Informatik, Universität Oldenburg. http://www.informatikdidaktik.de/Forschung/Schriften/PSimUnterrMatNatTag.pdf - 2002

[Steinhoff02] Steinhoff, Armin: *Python ports for QNX*. http://sourceforge.net/projects/pyqnx - 07/2002

[Sommer00] Sommerville, Ian: *Software Engineerin*. 6. Auflage, UK: Addison-Wesley - 2000, ISBN 020139815X

[TheKompany] The Kompany: *BlackAdder*. http://www.thekompany.com/products/blackadder/ - 05/2002

[UniCon00] The Unicode Consortium; Joan Aliprand; Julie Allen; Rick McGowan; Joe Becker; Michael Everson; Mike Ksar; Lisa Moore; Michel Suignard; Ken Whistler; Mark Davis; Asmus Freytag; John Jenkins: *The Unicode Standard*. Addison Wesley - 02/2000

[Waclena97] Waclena, Keith: *Programming Languages: What's wrong with them*. The University of Chicago. http://www.lib.uchicago.edu/keith/crisis/ - 07/2002

[WP02] Wang, Lingyun; Pfeiffer, Phil: *A Qualitative Analysis of the Usability of Perl, Python, and Tcl.* East Tennessee State University - `http://www.python10.com/p10-papers/14/index.htm` - 2002

[Wells02] Wells, Don: *Extreme Programming, a Gentle Introduction.* `http://www.extremeprogramming.org/` - 03/2002

[Williams02] Williams, Michael: *Handbook of the Physics Computing.* University of Oxford - `http://users.ox.ac.uk/~sann1276/handbook/` - 06/2002

[WJC93] Wilson, Brent; Jonassen, David; Cole, Peggy: *Cognitive Approaches to Instructional Design.* - The ASTD handbook of instructional technology. `http://carbon.cudenver.edu/~bwilson/training.html` - 1993

[Zope02] Zope Community: *The Zope Framework.* `http://www.zope.org` - 08/2002

[Zuse45] Zuse, Konrad: *Plankalkül.* `http://www.zib.de/zuse/Inhalt/Texte/Chrono/40er/Pdf/0233.pdf` - 1945

* * *