1. Einführung und Überblick

The Analytical Engine is therefore a machine of the most general nature. Whatever formula it is required to develop, the law of its development must be communicated to it by two sets of cards. When these have been placed, the engine is special for that particular formula. The numerical value of its constants must then be put on the columns of wheels below them, and on setting the engine in motion it will calculate and print the numerical results of that formula.

Charles Babbage (1864)

1.1 Bedeutung und Grundprobleme der Informatik

Die Informatik erfährt ihre grundlegende und fächerübergreifende Bedeutung dadurch, daß sie mit dem Computer ein Werkzeug zur Verfügung hat, das erstens in seiner theoretischen Mächtigkeit nicht mehr übertroffen werden kann und zweitens in der Praxis universell anwendbar ist. Nach heutigem Wissen kann man alles, was sich mit irgendeinem irgendwie denkbaren Formalismus berechnen läßt, vom Prinzip her auch mit dem Computer berechnen, vorausgesetzt man hat genügend Speicher und Zeit zur Verfügung. Der Computer ist aber nicht nur eine Rechenmaschine, sondern ein Universalwerkzeug zur Informationsverarbeitung. Dies wird intuitiv klar, wenn man an die heute übliche digitale Kommunikation denkt, wo mit Medien wie CD, DVD, Digitalradio, Digitalfernsehen, ISDN oder Mobiltelefonen alle Information in Form von Zahlen, also eben digital, verarbeitet und übermittelt wird. Diese Bedeutung betont auch der europäische Begriff **Informatik** (*Informatics*, *Informatique*) im Gegensatz zum amerikanischen *Computer Science*.

Das Grundproblem der Informatik liegt in der praktischen Nutzung der theoretisch vorhandenen Fähigkeiten. Zunächst muß das Universalwerkzeug Computer durch Schreiben eines Programms zu einem Spezialwerkzeug für jede bestimmte Aufgabe – zur Berechnung einer bestimmten mathematischen Funktion – gemacht werden. Dieses Prinzip wurde bereits von Charles Babbage (1791–1871) in der ersten Hälfte des 19. Jahrhunderts als Verallgemeinerung von lochkartengesteuerten Webstühlen erdacht, bei denen jedem Kartensatz (Programm) ein Muster (Funktion) entspricht und jeder Fadenart (Daten) eine Ausführung (Wert) des Musters. Ada, Countess of Lovelace (1816–1852), sprach davon, daß die Maschine „algebraische

Muster webt“. Babbage hatte zunächst an einer Spezialmaschine zur Berechnung von Wertetabellen von Polynomen gearbeitet, der sog. *Difference Engine*, die er zur Berechnung nautischer Tabellen benutzen wollte. In der Folge erfand er mit seiner *Analytical Engine* das Prinzip einer Universalmaschine und erkannte, daß man sie mit geeigneten Programmen außer zum Tabellarisieren von Polynomen im Prinzip genauso gut zum Berechnen ihrer Nullstellen (Gleichungslösen) oder zum Berechnen von Zügen bei Brettspielen verwenden könnte. Babbage scheiterte bei der praktischen Realisierung (Babbage, 1864),[1] seine *Difference Engine* wurde aber 1991 schließlich doch noch gebaut (Swade, 2000).

Bereits im 17. Jahrhundert waren die Grundlagen für eine mechanische Ausführung der Grundrechenarten gelegt worden, zu der das Genie von Babbage die Programmierbarkeit hinzufügte (Williams, 1997). Ins Jahr 1623 datiert die Rechenmaschine des Tübinger Professors Wilhelm Schickard, die addieren und subtrahieren konnte und den Benutzer auch beim Multiplizieren unterstützte. Schickard, Professor für Mathematik, Astronomie und Alte Sprachen, wollte seinem Freund Johannes Kepler, der in Tübingen promoviert hatte, bei dessen Berechnungen helfen. Es wurden aber nur einzelne Exemplare der Maschine gebaut, die alle verloren gingen; das für Kepler bestimmte Exemplar der Maschine verbrannte noch in der Werkstatt. Als Schickard 1635 an der Pest starb, geriet seine Maschine für 3 Jahrhunderte in Vergessenheit.[2] Um seinem Vater bei der Buchhaltung zu helfen, entwickelte Blaise Pascal in den Jahren 1642–1645 eine Rechenmaschine, die addieren konnte und das Subtrahieren unterstützte. Da das Währungssystem dieser Zeit kompliziert war und die Maschine 8 Stellen hatte, war die Mechanik wesentlich aufwendiger als bei Schickard. Bis zum Jahre 1652 wurden etwa fünfzig jeweils leicht verschiedene Prototypen hergestellt, von denen einige erhalten sind. Interessanterweise vetreten damit die ersten Rechenmaschinen auch schon zwei Hauptanwendungen der Informatik, nämlich die Naturwissenschaften und das Geschäftsleben.

In den 1940er Jahren und danach baute der deutsche Ingenieur Konrad Zuse in der Isolation der Kriegsjahre eine Reihe von Rechenmaschinen, für deren Schalter er zunächst elektro-mechanische Relais verwendete. Er entwickelte eine eigene Programmiersprache, den sog. **Plankalkül** und zielte auf Anwendungen im Ingenieurwesen, z. B. für die Berechnung von Tragflächen. Leider kam auch später nie eine breite Verbindung zu der universitären Forschung zustande, so daß die Tragweite seiner Erfindung in Theorie und Anwendung lange Zeit unerkannt blieb. Die Ingenieurwissenschaften sind aber auch heute noch das dritte große Aanwendungsgebiet der Informatik.

Seit den 30er Jahren des 20. Jahrhunderts waren von Mathematikern wie Church, Kleene, Post, Gödel, Herbrand und Turing mehrere formale Berechnungsmodelle entwickelt und deren Stärken und Grenzen untersucht worden, denn man

[1] Er macht dafür mangelnden Weitblick bei den forschungsfördernden Stellen der Regierung verantwortlich, offenbar ein historisches Problem der Informatik.

[2] Neben realen Nachbauten der Schickardschen Rechenmaschine gibt es inzwischen auch einen „virtuellen Nachbau“ im Internet, für den die Möglichkeiten der Programmiersprache Java zu interaktiven Simulationen benutzt werden, siehe `http://www.gris.uni-tuebingen.de/projects/schickard/`.

wollte wissen, inwieweit die Mathematik mechanisierbar ist. Church entwickelte den λ-Kalkül (*lambda calculus*), der später zur Programmiersprache LISP und anderen sog. funktionalen Sprachen führte. Alan Turing entwarf seine **Universelle Turingmaschine** (UTM) als abstraktes, mathematisch präzises Konzept einer einfachen, offensichtlich baubaren Maschine zur Ausführung von Berechnungen (Turing, 1937a,b).[3] Im Jahre 1931 hatte Kurt Gödel gezeigt, daß es wahre Aussagen über Zahlen gibt, die nicht durch eine formale Anwendung eines Kalküls bewiesen werden können. Church und Turing zeigten, daß man auch nicht durch eine (endliche) Berechnung entscheiden kann, ob eine Aussage so überhaupt beweisbar sein wird oder nicht.

Zumindest aber erwiesen sich alle Berechnungsmodelle als äquivalent (gleich mächtig), so daß alle auf der UTM implementiert werden können. Heute wird allgemein die Hypothese von Alonzo Church akzeptiert, daß es kein vernünftiges Berechnungsmodell gibt, das mächtiger wäre als etwa λ-Kalkül oder UTM (Church, 1936). Das heißt, daß schon die UTM mit ihrer rudimentären Programmiersprache ohne jegliche Datenstrukturen theoretisch völlig ausreicht, jede Funktion zu berechnen, für die eine konstruktive Berechnungsvorschrift (Algorithmus) in irgendeinem vernünftigen Formalismus vorliegt. Heutige Computer kann man vom Prinzip her als hoch optimierte Varianten der UTM ansehen, die allerdings in der Praxis immer mit endlichem Speicher auskommen müssen. Mehr hierzu findet sich z. B. bei Engeler und Läuchli (1988); Hodges (1994); Hopcroft und Ullman (2000).

Einige der historischen Persönlichkeiten standen Pate für die Namen moderner Programmiersprachen. Niklaus Wirth schuf mit Pascal die erste für die Lehre der Informatik geeignete Programmiersprache. Der Vorname von Ada, Countess of Lovelace, wurde zur Benennung einer im militärischen Bereich weit verbreiteten Sprache verwendet, so wie der Vorname des amerikanischen Logikers Haskell B. Curry für die funktionale Sprache Haskell.

Aus dem Wunsch der Mechanisierung aller Berechnungen ergeben sich drei große Teilgebiete der Informatik, mit denen sich schon Babbage und Turing beschäftigt haben: Theorie, Praxis und Technik. **Theoretische Informatik** befaßt sich ebenso mit Fragen der prinzipiellen Berechenbarkeit wie mit der Konstruktion von Algorithmen und mit der Analyse ihres prinzipiellen Aufwandes; diese Thematik behandeln wir in Kapitel 3 und in Teil III. **Praktische Informatik** befaßt sich mit der Umsetzung der Theorie in praktisch nutzbare Softwaresysteme. Teilgebiete sind u. a. Softwaretechnik, Programmiersprachen und Übersetzerbau, Datenbanken, Betriebssysteme, Verteiltes Rechnen oder Computer-Graphik; dieses Buch gibt insbesondere in den Kapiteln 4 und 5 sowie in Teil II eine Einführung in objektorientierte Programmiersprachen und Softwaretechnik und in Teil III in die Programmierung von Algorithmen. **Technische Informatik** behandelt den Bau und die Organisation von Computer-Hardware zur Ausführung der Software; wir geben eine elementare Einführung in Kapitel 2. Manchmal nimmt man als weiteres Gebiet noch die **Angewandte Informatik** hinzu und versteht darunter die Anwendung von Metho-

[3] Siehe auch `http://www.turing.org.uk`. Die Maschine wird von einem endlichen Automat gesteuert und speichert Programme und Daten auf einem beliebig langen Band.

den der Informatik auf andere Wissenschaften, etwa die Wirtschafts-, Medien-, Geo- oder die Lebenswissenschaften. In jüngster Zeit hat die Anwendung auf die Lebenswissenschaften das Fach der Bio-Informatik hervorgebracht, das vielfach schon in eigenen Studiengängen gelehrt wird. Denn genau so, wie man physikalische Zusammenhänge ohne Berechnungen durch Computer nicht in aller Tiefe verstehen kann, so benötigt man auch für ein umfassendes Verständnis etwa des Aufbaus und der Wirkung von Genen und Proteinen die Ergebnisse von Rechenverfahren (Algorithmen), die in der Praxis nur ein Computer ausführen kann.

Im Einzelnen verschwimmen oft die Grenzen zwischen den Teilgebieten; schon die Pioniere Babbage und Turing haben sowohl Rechenverfahren als auch Programme als auch Hardware entworfen und sich um neue Anwendungen bemüht. So geht es beispielsweise im Fach „Symbolisches Rechnen“ um die Implementierung mathematischer Rechenregeln von Algebra und Logik in Computer-Algebra Systemen und in automatischen Beweisern. Die theoretische Seite befaßt sich mit der Entwicklung konstruktiver Lösungsvorschriften für mathematische Probleme, die praktische Seite befaßt sich u. a. mit den speziellen Problemen der Repräsentation mathematischer Objekte (z. B. Polynome), mit geeigneten Implementierungstechniken, der Umsetzung in Java (siehe `java.math`) oder mit Anwendungen wie Systemen für Berechnungen im Ingenieurwesen oder zur formalen Verifikation von Software; siehe hierzu etwa Fleischer *et al.* (1995); Bibel und Schmitt (1998a,b,c).

Dieses Buch versteht sich vor allem als eine Einführung in das objektorientierte Programmieren der praktischen Informatik, behandelt aber so viel Theorie wie für ein grundsätzliches Verständnis der Probleme nötig ist, deren Lösung hier anhand von Java vorgeführt wird. Eine umfassendere Darstellung der mathematischen Grundlagen der Informatik findet sich bei Wolff *et al.* (2004). Das Zusammenspiel von Theorie und Praxis sieht man besonders gut anhand der Verifikation von Programmen (Kap. 3 und 17) und anhand der Konstruktion und Programmierung von Algorithmen (Kap. 3 und Teil III).

Der Informatik sind, bei aller vorhandenen Theorie, immer auch die tatsächliche Konstruktion und die Tauglichkeit von Lösungen in der Praxis und im Dienste von Anwendungen wichtig. Dadurch geht es immer auch um Effizienz und Kosten, und somit nicht nur um akademisch elegante Lösungen, sondern auch um tragfähige Lösungen im komplexen praktischen Umfeld.

Die Programme, die bei Babbage noch auf Lochkarten und bei Turing noch auf einem einfachen Lochstreifen (oder Band) gestanzt waren, werden heute auf Magnetplatten und in Halbleiterspeichern mit Kapazitäten im Gigabyte-Bereich gehalten. Dabei herrscht schon für minimale Speichergrößen von einigen hundert Byte eine praktisch unübersehbare Vielfalt von möglichen Programmen, von denen die meisten natürlich nicht das jeweils Gewünschte tun. Zudem stellt sich selbst für theoretisch korrekte Programme das Problem von ausreichendem Speicher und Zeit, das heißt der nötigen Effizienz der Lösung.

In der Praxis ist das Hauptproblem die Bewältigung der auftretenden Komplexität bei dem Entwurf und der Realisierung von Lösungen. Hier setzt in der modernen Software-Entwicklung die Objekttechnologie an, die Strukturen einführt, um

das Zusammenspiel von Funktionen und Daten (die den beiden Kartenstapeln von Babbage entsprechen) geeignet zu organisieren, damit auch große Software für den Menschen durchschaubar bleibt.

Wir wollen uns nun anhand von zwei abstrakten Gedankenspielen die theoretische Mächtigkeit und Bedeutung mathematischer Funktionen sowie die theoretische Vielfalt von Lösungen der Informatik noch einmal eindrücklicher vergegenwärtigen. Das erste Gedankenspiel illustriert die praktische Bedeutung des Computers, das zweite u. a. die Wichtigkeit von Organisation und Planung bei der Erstellung von Software.

1.1.1 Die Bedeutung des Berechnens von Funktionen

> *That the whole of the developments and operations of analysis are now capable of being executed by machinery.*
>
> *Charles Babbage (1864)*

Zwei Teilprobleme sind bei Problemlösungen der Informatik von ganz besonderer Bedeutung: das Berechnen von Funktionen (mittels Algorithmen) und das Modellieren und Realisieren von Daten und ihren Wechselbeziehungen (mittels Datenstrukturen). Historisch gesehen stand die Berechnung von mathematischen Funktionen lange Jahre im Vordergrund, nicht zuletzt weil die Informatik wesentlich von Mathematikern mit geschaffen wurde. Mit Funktionen sind hier mathematische Abbildungen von natürlichen Zahlen auf natürliche Zahlen gemeint; die Inkrementfunktion, die zu jeder Zahl ihren Nachfolger berechnet, ist ein ganz einfaches Beispiel. Wenn heute die Hälfte aller deutschen Haushalte einen PC besitzen, dann vermutlich aber nicht, weil sie vorderhand mathematische Funktionen berechnen wollen, sondern weil sie Texte verarbeiten, Musik und Videos speichern und abspielen oder per E-Mail kommunizieren wollen.

Eine erste wichtige Bemerkung zur Bedeutung von Zahlen und Funktionen ist, daß Zahlen auch als Repräsentanten (Codierungen) allgemeiner Symbole stehen können. Historisch benutzte man nach dem Mathematiker Kurt Gödel benannte Gödelisierungen, heute benutzt man standardisierte Codes wie z. B. ASCII. Wie wir in Kapitel 2 ausführen werden, entspricht jeder Zahl im Rechner genau ein Bitmuster, und Bitmuster können durch Vereinbarung geeigneter Codierungen wieder z. B. Schriftzeichen, (Gleit-)Kommazahlen und andere Symbole repräsentieren. Somit können wir auch allgemeine Texte und insbesondere auch Computerprogramme als Zahlen auffassen. Damit bekommen mathematische Abbildungen die Bedeutung allgemeiner Zuordnungen, wie z. B. die Zuordnung von Wörtern zu Wörtern in einem Wörterbuch, oder von Wörtern zu Zahlen in einem Telefonbuch. An dieser Stelle können wir den Computer als Universelle Turingmaschine schon für das technisch-wissenschaftliche Rechnen und für das Büro einsetzen.

Die universelle Bedeutung des Rechners als *Kommunikationsinstrument* ergibt sich aber erst daraus, daß sich für alle praktischen Fälle auch jede analoge elektromagnetische oder akustische Welle durch eine Folge von Zahlen repräsentieren

läßt. Dieses Prinzip nutzen die modernen digitalen Kommunikationsmittel wie CD, DVD, Digitalradio, Digitalfernsehen oder ISDN. Bilder und Töne, die in Zahlenform vorliegen, können mit dem Computer be- und verarbeitet werden, man kann sie also also z. B. speichern, kopieren, verschlüsseln, verändern etc.

Zur Wandlung in digitale Form (**Digitalisierung**) tastet man z. B. eine analoge Schwingung in regelmäßigen Abständen (also mit einer bestimmten Abtastfrequenz) ab und merkt sich dabei jeweils ihre Stärke als Zahl. Die Folge der Zahlen (in Abb. 1.1 unter den Abtaststellen angegeben) ist der **Puls** (*pulse*). Dabei macht man zwei Fehler: man tastet nur endlich viele Werte ab und man erfaßt jeden Wert nur mit einer bestimmten Genauigkeit. (Bei CD-Technologie tastet man mit 44,1 kHz ab, also 44.100 mal pro Sekunde und erfaßt jeden Wert mit 16 Bits, d. h. man erkennt nur $2^{16} = 65\,536$ verschiedene Signalstärken.)

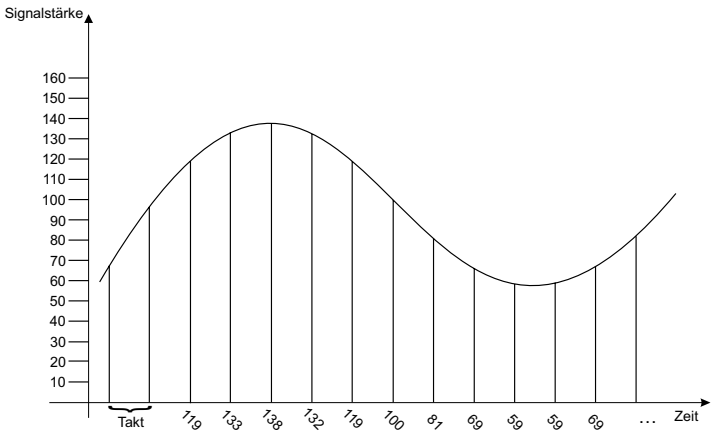


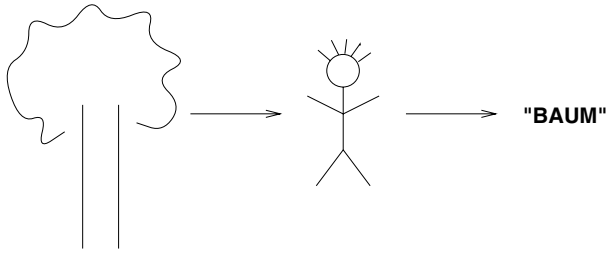
Abb. 1.1. Digitalisierung mit Pulscodemodulation

Das **Abtasttheorem** von Nyquist und Shannon besagt nun, daß man aus dem Puls die ursprüngliche Schwingung mit einer bis zur halben Abtastfrequenz reichenden Genauigkeit wieder rekonstruieren kann; lediglich darüber liegende höhere Frequenzen werden abgeschnitten. Da Computer heute mit Taktfrequenzen bis in den Gigahertz-Bereich hineinreichen und genügend Speicherplatz vorhanden ist, ist die Digitalisierung in der Praxis oftmals völlig verlustfrei. (Für den Audio Bereich erhält man aus 44,1 kHz Abtastfrequenz eine Tonrekonstruktion bis 22,05 kHz, wobei das menschliche Gehör nur von ca. 20 Hz bis 20 kHz reicht; der Quantisierungsfehler durch die Beschränkung auf 16 Bits entspricht einem Rauschen an der Grenze des Hörbaren.)

Zur digitalen Übertragung oder Speicherung der Schwingung speichert man bei der **Pulscodemodulation** (**PCM**) die Folge der Abtastwerte, beim Verfahren der **differentiellen PCM** die Folge der Differenzen der Abtastwerte und bei der **Del-**

tamodulation eine Folge von 1-bit Werten, die angibt, ob die Schwingungskurve steigt oder fällt; siehe hierzu auch (Bauer und Goos, 1991; Tanenbaum, 1997).

Nun machen wir ein weiteres Gedankenexperiment. Ein Beobachter sieht einen Baum, erkennt diesen und sagt daraufhin „Baum“. Dabei wandelt er (durch Denken) einen Sinnes-Eindruck in einen Ausdruck um.



Wie wir soeben bemerkt haben, können diese Eindrücke und Ausdrücke digitalisiert, d. h. in Zahlenform repräsentiert werden. In unserem Beispiel haben wir damit das Phänomen des Umwandelns eines Sinneseindrucks in einen entsprechenden Ausdruck als Abbildung von Zahlen auf Zahlen, d. h. als mathematische Funktion, beschrieben (ohne damit das Denken irgendwie erklärt zu haben)!

Diese Funktion ist offensichtlich berechenbar, denn der Mensch hat die Abbildung ja konstruktiv vorgenommen. Falls es gelingt, diese Abbildung in einer Turingmaschine zu programmieren, würden wir dann sagen, daß diese Maschine (künstlich) intelligent ist? Alan Turing, der bereits die Bedeutung seiner Maschine als universeller Informationsverarbeiter erkannt hatte, schlug hierzu seinen Turing-Test vor (Turing, 1950): Wenn ein Außenstehender das Ein-/Ausgabeverhalten einer Maschine nicht von dem eines Menschen unterscheiden kann, muß man die Maschine für intelligent halten, auch wenn sie evtl. ihre Ergebnisse auf völlig andere Art berechnet als der Mensch.

Auf die vielfältigen Debatten, die die Frage aufgeworfen hat, ob *alle* geistigen Fähigkeiten des Menschen sich im Rahmen des theoretisch Berechenbaren bewegen oder nicht, wollen wir an dieser Stelle nicht weiter eingehen. Auch der Turing-Test und die Frage nach seiner Aussagekraft sind Gegenstand vielfältiger Diskussionen. Weitere Informationen finden sich z. B. bei Penrose (1996) oder im Internet unter `http://www.turing.org.uk`.

1.1.2 Das Problem der Komplexität

Programmers are always surrounded by complexity; we cannot avoid it.
C. A. R. Hoare (1981)

Jedes Softwaresystem kann rein theoretisch als eine Funktion aufgefaßt werden, die aus digitalisierten Eingaben digitalisierte Ausgaben berechnet. An dieser Stelle begegnen wir aber einem Phänomen, das für die Informatik äußerst bedeutsam ist: die außerordentliche Komplexität der Modellierungs- und Lösungsmöglichkeiten.

Zur Illustration betrachten wir wieder ein stark vereinfachendes Beispiel: Wir vergleichen die Anzahl verschiedener Bilder auf einem Schwarzweiß- und einem Farbbildschirm, sowie die Anzahl verschieden beschriebener Schreibmaschinenseiten mit der Zahl von Wasserstoffatomen, die im bekannten Weltall Platz haben.

Ein Computerbildschirm hat ca. eine Million Bildpunkte. In Schwarzweiß ergeben sich also $2^{1\,000\,000}$, bei 256 Farben $2^{8\,000\,000}$ mögliche Bilder.

Wieviel Möglichkeiten gibt es, eine Seite Text zu schreiben (z. B. Programmtext)? Auf eine DIN A4 Seite passen zunächst etwa 2000 Zeichen. Für jede Zeichenstelle kann man aus $2^8 = 256$ Zeichen wählen. Also erhält man $(2^8)^{2000} = 2^{8\cdot 2000} = 2^{16\,000}$ „mögliche“ Texte; $2^{16\,000} = (2^{10\cdot 1600}) \approx 10^{3\cdot 1600} = 10^{4800}$.

Im Vergleich dazu: Wieviele Wasserstoffatome passen ins Weltall? Wir nehmen hierzu wie allgemein üblich an, daß das Weltall vor 15 Milliarden Jahren durch einen Urknall entstanden ist. Es kann sich maximal mit Lichtgeschwindigkeit ausdehnen, nimmt also maximal eine Kugel mit einem Radius von 15 Milliarden Lichtjahren ein. Damit erhalten wir für den Durchmesser D_{W} des Weltalls

$$\begin{aligned} D_{\mathrm{W}} &= 2 \cdot (15 \cdot 10^9) \text{ Lichtjahre} \\ &= (30 \cdot 10^9) \cdot (10\,000 \cdot 10^9)\text{km} \\ &= 300\,000 \cdot 10^{18}\text{km} \; = 3 \cdot 10^5 \cdot 10^{18} \cdot 10^3\text{m} \\ &= 3 \cdot 10^{26}\text{m}. \end{aligned}$$

Daraus ergibt sich für das Volumen des Weltalls $V_{\mathrm{W}} \approx (3 \cdot 10^{26})^3\text{m}^3 = 27 \cdot 10^{78}\text{m}^3 \approx 10^{79}\text{m}^3$.

Im Vergleich dazu gilt für den Durchmesser D_{H} eines Wasserstoff-Atoms $D_{\mathrm{H}} \approx 10^{-10}\text{m} = 1\text{Å}$. Hieraus ergibt sich ein Volumen $V_{\mathrm{H}} \approx 1\text{Å}^3 = 10^{-30}\text{m}^3$.

Es haben also maximal $\frac{V_{\mathrm{W}}}{V_{\mathrm{H}}}$ Atome im Weltall Platz, wobei $\frac{V_{\mathrm{W}}}{V_{\mathrm{H}}} \approx \frac{10^{79}}{10^{-30}} = 10^{109} \approx 2^{362}$. Man beachte, daß $10^{4\,800} = 10^{109} \cdot 10^{4\,691}$!

In der Informatik haben wir es also mit einer kombinatorischen Explosion von Möglichkeiten zu tun, da wir keinen herkömmlichen physikalischen Restriktionen bei der Kombination unterliegen. Oft scheitern mathematisch einfach erscheinende Lösungswege an der praktischen Komplexität. Zum Beispiel können theoretisch alle Bäume in den $2^{1\,000\,000}$ möglichen Bildern auf einem Bildschirm durch eine Funktion b (mit endlichem Definitionsbereich!) erkannt werden, die jedes Baum-Bild auf 1 und jedes andere Bild auf 0 abbildet – aber diese Funktion kann ohne weitere Information praktisch nicht realisiert oder gespeichert werden.

1.2 Konzeption des Buches

Das zentrale Thema der Informatik ist es, geeignete Konzepte zur Strukturierung der ungeheuren Vielfalt möglicher Problemlösungen zu finden. Für unsere Überarbeitung der Vorlesung Informatik I/II an der Universität Tübingen und der Vorlesung Informatik I an der Universität Bonn haben wir darum einen Ansatz gewählt,

der explizit bemüht ist, die Gewichte zwischen den Ansprüchen der akademisch-grundlegenden Seite und der industriellen Praxis auszutarieren. Wir behandeln in diesem Buch hierzu objektorientierte Softwaretechnik, also objektorientierte Analyse- und Modellierungsmethoden für Software sowie Programmiermethoden, Datenstrukturen und allgemeine Problemlösungsmethoden (Algorithmen) aus objektorientierter Sicht. Die Verwendung von Industrie-Standards wie der Modellierungssprache UML und der Programmiersprache Java vermitteln unmittelbar praxisrelevantes Wissen. Theoretische Begründungen des Stoffes unter Einschluß von Themen wie formale Verifikation von Programmen sowie ein ausführlicher mathematischer Anhang schaffen eine dauerhafte Wissensbasis für das in der Informatik unumgängliche lebenslange Lernen.

Mit Java liegt (erstmalig seit Pascal) heute wieder eine Sprache vor, die sowohl konzeptionell auf der Höhe der Zeit ist als auch breite industrielle Anwendung findet und sich gleichwohl für die Anfängerausbildung an der Hochschule eignet. Sehr wertvoll an Java ist die Vielzahl nützlicher Standards und Werkzeuge, die die Sprache umgeben. Zunächst ist es ungeheuer hilfreich, daß es bei Java weitgehend unerheblich ist, auf welchem Rechner mit welchem Compiler ein Programm ausgeführt wird – die Ergebnisse sind stets die gleichen. Mittlerweile existieren auch kostenlose und offene integrierte Entwicklungsumgebungen (IDE) wie Netbeans oder Eclipse, mit denen eine plattformunabhängige professionelle Programmierung möglich ist. Sodann hat Java eine Fülle standardisierter Schnittstellen und Bibliotheken für wichtige Probleme wie graphische Oberflächen, Parallelausführung, Netzwerk- und Datenbankverbindungen und viele andere mehr. In der industriellen Praxis reduziert dies alles die Systemvielfalt, für die Grundvorlesung Informatik eröffnet es neue Möglichkeiten, z. B. die Behandlung von graphischen Oberflächen oder von Themen des Software Engineering.

1.2.1 Aufbau des Buches

Dieses Buch behandelt sowohl allgemeine Grundlagen der Informatik als auch speziell das Programmieren mit Java. Es ist in vier Teile gegliedert: **(I) Grundkonzepte**, **(II) Sprachkonzepte**, **(III) Algorithmen** und **(IV) Theorie**. Teil I gibt unabhängig von konkreten Rechnern oder Programmiersprachen einen Überblick über wesentliche Grundkonzepte von Hardware und Software. Teil II behandelt die Konzepte objektorientierter Programmiersprachen konkret anhand von Java. Teil III behandelt die Theorie und Praxis der Konstruktion von Algorithmen mit Java. In Teil IV sind theoretische Grundlagen zusammengefaßt.

Ein Vorteil von Java ist die konsequente Objektorientierung. Dies schafft aber für ein Lehrbuch Probleme, da Objekte ein relativ fortgeschrittenes Konzept sind, ohne das man in Java aber wenig tun kann. Anders als bei einer Verwendung von C++ kann man nicht ohne weiteres zuerst einen „C-Teil“ (ohne Objekte) behandeln und dann den „C++-Teil“ (die Objekt-Erweiterungen). Wir lösen das Problem durch einen Kunstgriff: In Teil I lernen wir zunächst abstrakte Objekttechnik kennen, u. a. anhand von UML, in Teil II lernen wir dann Java verstehen und programmieren.

Teil I: Grundkonzepte. Wir beginnen in Kap. 2 mit einem Überblick über die Prinzipien von Hardware- und Software-Architektur mit einem Schwerpunkt auf PC-Systemen. Die Behandlung von Zahldarstellungen und Konversionsmethoden führt in natürlicher Weise zum Begriff des Algorithmus hin. Im folgenden Kap. 3 führen wir in dieses Konzept ein und untersuchen gängige Sprachkonzepte zur Beschreibung von Algorithmen einschließlich von Flußdiagrammen, die wir in der Form von UML Aktivitätsdiagrammen verwenden. Bereits hier, mit den ersten Beschreibungen von elementaren Algorithmen, führen wir die Methode von Floyd zur Verifikation ein. Danach geben wir in Kap. 4 eine abstrakte und sprachunabhängige Einführung in Datenstrukturen, die in natürlicher Weise zum Konzept des Objekts führt. In Kap. 5 geben wir anhand von Klassendiagrammen in UML eine Einführung in die Konzepte von Klassen, Objekten, abstrakten Datentypen sowie dem objektorientierten Software-Entwurf als zentrale Strukturierungsmittel der Software-Erstellung. Alle Themen von Teil I werden in den späteren Teilen nochmals anhand von Java aufgenommen und wesentlich vertieft. Sie werden hier in kompakter Form erstmals vorgestellt, damit wir die programmiersprachlichen Teile nicht durch grundlegende Konzepte wie Analyse und Entwurf oder Verifikation zerdehnen müssen und damit wir später unabhängig vom Fortschritt in Java schon das wesentliche Arsenal der abstrakten Konzepte kennen.

Teil II: Sprachkonzepte und Java. Wir behandeln in Kap. 6 zunächst den „C-Teil“ von Java (inklusive Arrays und Strings, da die Konzepte der Objekttechnik bereits aus Teil I bekannt sind). Kapitel 7 führt dann Klassen und dynamische Datentypen (Listen, Stacks etc.) ein und Kap. 8 behandelt höhere Konzepte wie Vererbung und virtuelle Funktionen. Kap. 9 führt anhand von AWT in graphische Oberflächen (GUI) ein und behandelt zwei größere Programmierbeispiele, die sowohl UML-basierte Modellierung aus Teil I als auch große Teile des Stoffes aus Teil II anwenden.

Teil III: Algorithmen. Wir behandeln die Theorie und Praxis von Algorithmen: Entwurf, Komplexitätsanalyse und Implementierung von Standard-Algorithmen wie Suchen, Sortieren, Baum-Algorithmen und Hash-Verfahren, sowie weiterführenden höheren Datenstrukturen wie Bäume und Hash-Tabellen. Hier werden sowohl die objektorientierten Programmierverfahren als auch die höheren Datentypen (Listen, Stacks, Arrays) aus Teil II angewendet.

Teil IV: Theorie. Hier haben wir elementare Mathematik und theoretische Grundlagen der Informatik zum Nachschlagen zusammengefaßt. Außerdem wird hier, als Anwendung der mathematischen Logik, ausführlich die Verifikation von Programmen mit dem Hoare-Kalkül behandelt, nachdem Floyd's halb-formale Methode und Schleifeninvarianten schon von Teil I bekannt sind. Eine weit ausführlichere Behandlung der für die Informatik relevanten Diskreten Mathematik und Logik findet sich bei Wolff *et al.* (2004).

1.2.2 Hinweise für Dozenten

Dieses Buch ist insbesondere zur Verwendung als Lehrbuch in der Einführungsvorlesung Informatik I / II an Universitäten und Fachhochschulen gedacht. Foliensätze zur bisherigen zweiten und zur vorliegenden dritten Auflage finden sich unter

`www-sr.informatik.uni-tuebingen.de/InfoBuch` .

Für die begleitenden Mathematik-Vorlesungen empfiehlt sich das neue Lehrbuch „Mathematik für Informatik und BioInformatik“ von Wolff *et al.* (2004), das schon weitgehend mit den Vorschlägen der GI für die Mathematik der neuen Bachelor-Studiengänge in Informatik kompatibel ist.

Im akademischen Jahr 2003/04 haben wir in Tübingen zum ersten Mal zusammen mit dem Buch von Anfang an die neue integrierte Entwicklungsumgebung Eclipse eingesetzt. Dieses Konzept mit dem Namen FOOD (*foundations of object oriented development*) wurde 2003 mit einem *IBM Eclipse Innovation Award* ausgezeichnet und hat sich in der Praxis bestens bewährt. Eclipse ist eine quell-offene professionelle Entwicklungsumgebung, die für viele Kombinationen von Hardware und Betriebssystemen verfügbar und problemlos zu installieren ist. Eclipse wird von den im Programmieren erfahrenen Studierenden als professionelles Tool akzeptiert, unterstützt aber auch die Anfänger (z. B. durch *syntax-aware editing*) und überfordert sie keineswegs. Darüber hinaus homogenisiert Eclipse die heterogene Landschaft an Hardware und Betriebssystemen, sodaß man Anweisungen und Hinweise zum Programmieren nur ein einziges Mal verfassen muß.

Unter Verwendung aller vier Teile deckt dieses Buch praktisch die ersten zwei Semester des Informatikstudiums ab. In diesem Fall kann man bei einem theoriebetonten Vorgehen mit Stoff der Kapitel 15 und 16 aus Teil IV beginnen und danach die Teile I, II und III in Folge behandeln. Das Kapitel 17 über Korrektheit von Unterprogrammen kann dann unmittelbar nach dem Abschnitt über Unterprogramme in Kap. 6 behandelt werden.

Alternativ kann man den Schwerpunkt auf die Teile II (Java) und III (Algorithmen) legen und nach Bedarf mit Material aus Teil I (Grundkonzepte) und Teil IV (Theorie) anreichern. Bei einem gestrafften Vorgehen bleiben dann noch ca. 4–6 Wochen Vorlesungszeit am Ende übrig, für die sich u. a. folgende Optionen als Alternativen anbieten:

1. Eine Einheit zu Übersetzerbau und virtuellen Stackmaschinen (Wirth, 1995; Lindholm und Yellin, 1996). Alle hierfür nötigen Algorithmen und Datenstrukturen (insbesondere Bäume und Hash-Tabellen) werden in Teil III eingeführt.
2. Eine Einführung in C++ anhand der Differenz zu Java. (Zum Beispiel Zeigervariablen, Variablen allgemeiner Referenzstufe, Objektvariablen der Referenzstufe Null, Objekte auf dem Stack, Destruktoren, Templates.)
3. Mit Java eröffnet sich die Möglichkeit einer homogenen themenübergreifenden Einführung in die Informatik. Eine Einheit über Systemkonzepte in Java könnte ausgehend von Kap. 2 in Teil I Themen von Betriebssystemen und Datenbanksystemen aufgreifen, wie z. B. Files, Threads of Control, Remote Method Invo-

cation (RMI), Netzverbindungen, Objekt-Serialisierung und Persistenz, sowie Datenbanken (JDBC), vgl. (Hendrich, 1997; Kredel und Yoshida, 2002).

Für die Vorlesung in Tübingen 2003/04 wurden im ersten Semester die Teile I (Grundkonzepte) und II (Sprachkonzepte) in der Reihenfolge der vorliegenden dritten Auflage durchgenommen. In der ersten Hälfte des zweiten Semesters folgte Teil III, und in der zweiten Semesterhälfte wurden die obigen Themen 1 und 2 behandelt.