STAATSINSTITUT FÜR SCHULQUALITÄT
UND BILDUNGSFORSCHUNG
MÜNCHEN

Informatik am Naturwissenschaftlich-technologischen Gymnasium Jahrgangsstufe 12

München 2010

Erarbeitet im Auftrag des Bayerischen Staatsministeriums für Unterricht und Kultus

Leitung des Arbeitskreises:
Dr. Petra Schwaiger ISB

Mitglieder des Arbeitskreises:
Dr. Siglinde Voß Gymnasium Immenstadt
Andreas Wagner Goethe-Gymnasium Regensburg
Albert Wiedemann Erasmus-Grasser-Gymnasium München
Stefan Winter Gymnasium Vilshofen

Herausgeber:
Staatsinstitut für Schulqualität und Bildungsforschung

Anschrift:
Staatsinstitut für Schulqualität und Bildungsforschung
Abteilung Gymnasium
Schellingstr. 155
80797 München
Tel.: 089 2170-2304
Fax: 089 2170-2125
Internet: www.isb.bayern.de
E-Mail: petra.schwaiger@isb.bayern.de

Druck und Vertrieb:
Kastner AG – das medienhaus
Schlosshof 2-6
85283 Wolnzach
Internet: www.kastner.de

Inhaltsverzeichnis

Vorwort

Die vorliegende Handreichung enthält eine Aufarbeitung von informatischem Fachwissen der Jahrgangsstufe 12, didaktisch-methodische Überlegungen zu den einzelnen Lehrplaninhalten bzw. Hinweise zu deren Intention, zahlreiche Aufgaben mit Lösungen bzw. Lösungshinweisen sowie weiteres umfangreiches digitales Material auf der beiliegenden CD-ROM.

Aufbauend auf dem vorausgegangenen Informatikunterricht eignen sich die Schülerinnen und Schüler, die in der Jahrgangsstufe 12 das Fach Informatik belegen, weiterführende Konzepte und Methoden der Informatik an. Der Lehrplan sieht die vier Themengebiete „formale Sprachen“, „Kommunikation und Synchronisation von Prozessen“, „Funktionsweise eines Rechners“ und „Grenzen der Berechenbarkeit“ vor. Das Fachprofil nennt unter anderem folgende Zielsetzungen: „Ein fundiertes Verständnis für die prinzipielle Funktionsweise eines Rechners gewinnen die Schüler durch die genauere Beschäftigung mit dem Vorgang der Kommunikation mit der Maschine. Überlegungen zu den Grenzen der maschinellen Berechenbarkeit unterstützen sie bei der realistischen Einschätzung der tatsächlichen Möglichkeiten der Informationstechnologie, wie sie für ein selbstbestimmtes Leben und Arbeiten in unserer Informationsgesellschaft notwendig ist.“

Insgesamt konkretisieren die vorgestellten Konzepte Intention und Anforderungsniveau des Lehrplans, sodass die Auswahl von Aufgaben und Vertiefungsmöglichkeiten aus dem umfangreichen Angebot der Schulbücher unterstützt wird.

An dieser Stelle möchte ich es nicht versäumen, den Mitgliedern des Arbeitskreises für ihr enormes Engagement zu danken.

München, Juli 2010 Petra Schwaiger

Bemerkungen:

- In der Handreichung ist von „Schülerinnen und Schüler“ bzw. von „Lehrerinnen und Lehrer“ die Rede. In allen anderen Fällen wurde die weibliche Form der Kürze halber mitgedacht.
- Auf der Begleit-CD befinden sich die Dateien, auf die im Text verwiesen wird. Diese enthalten die Implementierungen und ggf. weitere Lösungshinweise zu den verschiedenen Kapiteln. Die Dateistruktur korrespondiert im Wesentlichen mit den übergeordneten Abschnitten der Handreichung.
- Darüber hinaus befindet sich die elektronische Farbversion des Handreichungstextes auf der Begleit-CD. Zu beachten ist, dass sich Hinweise auf verwendete Farbcodierungen auf die Farbversion der Handreichung bezieht. Entsprechende Textpassagen sind jedoch auch ohne Farbdruck verständlich.
- Die Handreichung sowie die auf der Begleit-CD enthaltenen Materialien stehen auch auf der Homepage des ISB (ISB-Startseite (www.isb.bayern.de) → Gymnasium → Fach Informatik → Publikationen) zur Verfügung.

1 Formale Sprachen (ca. 16 Std.)

Lp: Um die Möglichkeiten der Kommunikation zwischen Mensch und Maschine besser beurteilen zu können, betrachten die Schüler in Jahrgangsstufe 12 das dabei verwendete Hilfsmittel Sprache. Die Erkenntnis, dass für die Kommunikation mit einer Maschine exakte Vereinbarungen unentbehrlich sind, führt sie zum Begriff der formalen Sprache. Bei deren praktischer Anwendung wird den Jugendlichen bewusst, dass der Kommunikation Mensch-Maschine durch den nötigen Formalismus große Beschränkungen auferlegt sind.

1.1 Formalisierung von Sprachen

Lp: Bisher kennen die Schüler Sprachen vor allem als Mittel zur Kommunikation zwischen Menschen. Ihnen ist bekannt, dass eindeutiges gegenseitiges Verstehen nur dann gewährleistet ist, wenn sich die Kommunikationspartner auf eine gemeinsame Sprache einigen und die zu deren Gebrauch festgelegten Regeln einhalten. Im Rückblick auf das bisherige Arbeiten mit dem Computer wird ihnen deutlich, dass die Verständigung zwischen Mensch und Maschine ebenfalls einen Kommunikationsprozess darstellt, der ähnlichen Regeln unterliegt. Daher betrachten sie zunächst den strukturellen Aufbau einer ihnen bereits bekannten natürlichen Sprache sowie den Aufbau einer künstlichen Sprache. Die Jugendlichen lernen dabei, die Begriffe Syntax und Semantik einer Sprache zu unterscheiden.

1.1.1 Didaktische Hinweise

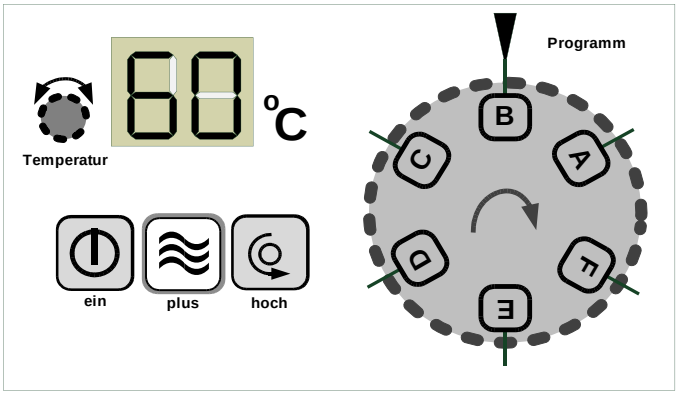
Den Schülerinnen und Schülern ist bereits bewusst, dass die Kommunikation in einer natürlichen Sprache zu Missverständnissen führt, wenn zwei Gesprächspartner denselben Satz unterschiedlich interpretieren. Die Ausdrucksmöglichkeiten einer natürlichen Sprache sind so groß, dass eine eindeutige Repräsentation der Bedeutung bestimmter Aussagen nicht immer möglich ist.

Die Verständigung mit Maschinen muss jedoch unmissverständlich sein. Daher verwendet man formale Sprachen, welche als eindeutig definierte Menge zugelassener Zeichenketten festgelegt sind, denen jeweils eine genau festgelegte Semantik zugeordnet ist. Diese Menge lässt sich im einfachsten Fall entweder direkt durch Aufzählung angeben, oder aber mithilfe von Regeln zur Bildung korrekter Zeichenketten.

Im nachfolgenden Abschnitt lernen die Schülerinnen und Schüler, dass bereits bei der Interaktion mit vielen Maschinen aus ihrem Erfahrungsbereich eine Art Sprache benutzt wird, bei der jedes „Wort“ (eine zulässige, aus den Zeichen der Sprache gebildete Zeichenkette, die korrekt interpretiert werden kann) eine Anweisung für eine eindeutige Abarbeitung der gewünschten Arbeitsschritte darstellt. Insbesondere ist dies bei Geräten mit einem breiten Einsatzbereich der Fall, bei denen der Anwender beispielsweise durch Schalt- bzw. Drehknöpfe den gewünschten Ablauf vorab einstellen kann.

1.1.2 Wörter aus einem Alphabet

Als Beispiel soll die Kommunikation einer Person mit einer einfachen Maschine, etwa einer Waschmaschine, betrachtet werden. Diese hat Knöpfe oder Drehschalter, die mit bestimmten Symbolen markiert sind.



Neben der Einstellung eines der sechs Waschprogramme (A, B, C, D, E, F) und der Waschtemperatur soll durch einen Knopfdruck zusätzlich eine erhöhte Wasserzufuhr bzw. eine extra hohe Drehschleuderzahl möglich sein. Um einen Waschgang schließlich starten zu können, muss am Ende der Einschaltknopf gedrückt werden. Drückt man die gewählten Optionen durch Aneinanderreihen der entsprechenden Symbole aus, so erhält man Zeichenketten, die die Grundlage der Sprache zur Kommunikation mit dieser Waschmaschine bilden.

So kann etwa mit der Zeichenkette 'B60' ausgedrückt werden, dass die Waschmaschine das Programm B mit einer Waschtemperatur von 60° und erhöhter Drehschleuderzahl startet.

Für die Beschreibung einer Sprache, mit deren Hilfe der Anwender alle möglichen Programmabläufe der Maschine notieren kann, ist zunächst eine Vereinbarung über alle zugrunde liegenden Zeichen notwendig. Die Menge der zugelassenen Symbole Σ bildet das **Alphabet** der verwendeten Sprache.

Am Beispiel der einfachen Waschmaschine ist mit Σ = {A, B, C, D, E, F, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, , , } ein mögliches Alphabet festgelegt.

1.1.3 Formale Sprachen

Nach der Festlegung des Alphabets können formale Sprachen als eindeutig definierte Menge zugelassener Zeichenketten von Symbolen aus einem Alphabet vereinbart werden. Diese Festlegung erfolgt entweder durch direkte Aufzählung oder mithilfe von Regeln.

Die **Worte einer formalen Sprache** sind Zeichenketten aus den Symbolen des Alphabets, die in dieser Sprache als korrekt definiert sind. Dabei ist der Begriff „Wort“ in diesem Kontext klar von der Bedeutung „Wort“ in natürlichen Sprachen zu unterscheiden. In der Informatik wird beispielsweise ein syntaktisch korrektes Programm als ein (einziges) Wort der zugehörigen (Programmier-)Sprache angesehen, während bei natürlichen Sprachen nur zusammenhängende korrekte Buchstabenketten (ohne Leerzeichen) als Wort verstanden werden.

Alle anderen Zeichenketten, die den Regeln nicht entsprechen, gehören nicht zur festgesetzten Sprache, bilden also keine Wörter der Sprache.

Die Regeln legen fest, wie man die Symbole des Alphabets der Reihe nach anordnen darf. Diese Regeln werden zunächst noch informell formuliert.

Am Beispiel der Sprache aller zulässigen Zeichenketten für die Darstellung von Anweisungen für die Waschmaschine könnten diese Regeln wie folgt aufgestellt werden:

(1) Starte mit dem Symbol für das Waschprogramm: A, B, C, D, E, F;

(2) Darauf folgen zwei der Ziffern 0, 1, 2, 3,..., 9 für die Waschtemperatur;

(3) Optional folgt ein Symbol , wenn man eine erhöhte Schleuderdrehzahl möchte;

(4) Optional folgt ein Symbol , wenn man eine erhöhte Wasserzufuhr möchte;

(5) Am Ende steht immer das Symbol .

Alle Zeichenketten, die nach den angegebenen Regeln aus dem vorgegebenem Alphabet gebildet sind, gehören zur Sprache der Anweisungen für die Waschmaschine. Man sagt: die Zeichenketten sind **syntaktisch korrekt** bezüglich dieser Sprache. Weitere Beispiele für syntaktisch korrekte Zeichenketten sind z.B. 'F95' oder 'C33'.

Nicht syntaktisch korrekt wäre z.B. 'K12', da K nicht zum Alphabet gehört, ebenso 'A40', da die Reihenfolge der Symbole nach obigen Regeln nicht eingehalten wurde.

1.1.4 Unterschied zwischen Syntax und Semantik

Die Semantik eines Wortes (einer syntaktisch akzeptierten Zeichenkette) gibt seine inhaltliche Bedeutung wieder. So gibt es etwa syntaktisch korrekte Wörter einer Sprache, die jedoch semantisch nicht sinnvoll sind, etwa das Wort 'A00⏻' aus dem eben angeführten Waschmaschinen-Beispiel. Obige Regeln sind zwar eingehalten, doch wird keine Waschmaschine die Wäsche bei 0°C waschen können, da eine Vorrichtung zur Abkühlung des Zulaufwassers sicher nicht eingebaut ist. Möglicherweise ist die Waschmaschine so gebaut, dass mit der Einstellung 'A00⏻' das Waschprogramm A mit der voreingestellten Temperatur des Kaltwassers durchlaufen wird. Durch Hinzunahme zusätzlicher Regeln könnte die Sprache für die Waschmaschine noch sinnvoller festgelegt werden. Beispielsweise könnte man die Temperaturangaben einschränken, indem man für die erste Stelle nur die Ziffern 1 bis 9 zulässt.

1.2 Aufbau formaler Sprachen

Lp: Anhand einfacher Beispiele wie Autokennzeichen, E-Mail-Adressen oder Gleitkommazahlen lernen die Schüler den Begriff der formalen Sprache als Menge von Zeichenketten kennen, die nach bestimmten Regeln aufgebaut sind.

1.2.1 Didaktische Hinweise

Die Schülerinnen und Schülern lernen, den Aufbau einer formalen Sprache mithilfe formaler Regeln zu definieren. Die Festlegung der Grammatik einer Sprache erfolgt mithilfe von Produktionsregeln. Dabei beschränkt man sich auf Produktionen, die auf der linken Seite nur aus einem einzigen Symbol (genauer: Variable) bestehen (kontextfreie Produktionen). Kontextsensitive Sprachen werden nicht behandelt.

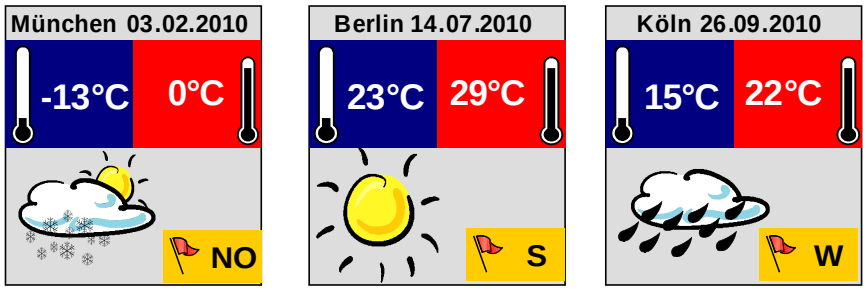
Zu beachten ist außerdem, dass für die Formulierung von Produktionen neben den eigentlichen Zeichen des Alphabets zusätzliche syntaktische Variable benötigt werden, welche nicht zu den Symbolen der Sprache gehören. Zur deutlichen Unterscheidung wird eine derartige syntaktische Variable im Folgenden in spitzen Klammern geschrieben, z.B. <s>. Damit können Bezeichner von syntaktischen Variablen auch aus mehreren Zeichen bestehen, z.B. <start>, solange sie als eine Einheit interpretiert werden.

Hinweis: In der vorliegenden Ausarbeitung werden nur Grammatiken mit genau einer Startvariablen betrachtet.

1.2.2 Produktionen zur Erzeugung von Wörtern einer Sprache

Zur Darstellung von Wetterinformationen für einen bestimmten Tag an einem bestimmten Ort werden im Allgemeinen mehrere Daten benötigt, wie etwa die Angabe der tiefsten und höchsten Tagestemperatur, der Wetterlage, der Luftfeuchtigkeit und der Windrichtung. Die aus Tageszeitungen bzw. Webseiten bekannten grafischen Wetterdarstellungen sollen beispielsweise durch ein Programm nach Eingabe einer entsprechenden Anweisung automatisch gezeichnet werden können.

Beispiele für Wörter einer solchen Sprache könnten etwa mit '-13°C0°C🌤NO' für einen kalten, teils sonnigen Wintertag mit Schneefall und Windrichtung Nord-Ost, '23°C29°C☀S' für einen sonnigen heißen Tag mit Windrichtung S oder etwa '15°C22°C🌧W' für einen bewölkten und regnerischen Tag mit Wind aus westlicher Richtung angegeben werden.



Für die Beschreibung der Sprache dieser nach dem angegebenem Schema gebildeten Wörter ist eine Aufzählung aller korrekten Wörter unhandlich, da ihre Anzahl unüberschaubar groß ist.

Übersichtlicher ist die Beschreibung der Struktur der zur Sprache gehörigen Wörter mithilfe von Oberbegriffen, deren Aufbau sukzessive immer detaillierter wiedergegeben wird, bis man schließlich eine aus lauter Zeichen des Alphabets bestehende Zeichenkette erhält.

Ein Vorschlag zur Beschreibung der Zusammensetzung von zulässigen Zeichenketten der Sprache der Wetterinformationen nach obigem Muster ist folgende zunächst in natürlicher Sprache formulierte Auflistung:

- Jede Wetterdarstellung besteht aus der Angabe von zwei Temperaturwerten (für die Tiefst- und Höchsttemperatur), der Wetterlage und Windrichtung.
- Eine Temperatur wird durch eine Zahl aus ein oder zwei Ziffern und der Einheit °C angegeben. Optional wird das Symbol – (für negative Temperaturen) vorangestellt.
- Die Wetterlage wird durch ein oder zwei Wettersymbole dargestellt.
- Ein Wettersymbol wird durch eines der Bildzeichen ☁, ☼, 🔥, ✳ repräsentiert.
- Die Windrichtung wird durch eine N-S-Richtung bzw. O-W-Richtung oder durch N-S-Richtung gefolgt von O-W-Richtung angegeben.
- Die N-S-Richtung wird durch den Buchstaben N oder S, die O-W-Richtung durch W oder O abgekürzt.

Eine Formalisierung der angegebenen Strukturbeschreibung wird mithilfe von **Produktionsregeln (Produktionen)** durchgeführt. Eine Produktion besteht dabei aus zwei Teilen, die durch einen Pfeil voneinander getrennt sind: Auf der linken Seite steht eine syntaktische Variable, auf der rechten Seite ein Term, der Zeichen des Alphabets und/oder wiederum eine oder mehrere syntaktischen Variable enthalten kann.

Damit lassen sich die Regeln für die Sprache der Wetterdarstellungen durch folgende 28 Produktionen angeben, wobei Nebenbedingungen, dass etwa die erste Temperaturangabe kleiner sein muss als die zweite, hier nicht berücksichtigt sind:

(1) <Wetter> → <Temperatur><Temperatur><Wetterlage><Windrichtung>

(2) <Temperatur> → – <Zahl>°C

(3) <Temperatur> → <Zahl>°C

(4) <Wetterlage> → <Wettersymbol>

(5) <Wetterlage> → <Wettersymbol><Wettersymbol>

(6) <Wettersymbol> → ☼

(7) <Wettersymbol> → 

(8) <Wettersymbol> → 

(9) <Wettersymbol> → 

(10) <Zahl> → <Ziffer>

(11) <Zahl> → <Ziffer><Ziffer>

(12) <Ziffer> → 0

(13) <Ziffer> → 1

(14) <Ziffer> → 2

(15) <Ziffer> → 3

(16) <Ziffer> → 4

(17) <Ziffer> → 5

(18) <Ziffer> → 6

(19) <Ziffer> → 7

(20) <Ziffer> → 8

(21) <Ziffer> → 9

(22) <Windrichtung> → <NS>

(23) <Windrichtung> → <OW>

(24) <Windrichtung> → <NS><OW>

(25) <NS> → N

(26) <NS> → S

(27) <OW> → W

(28) <OW> → O

1.2.3 Erzeugung von Wörtern einer Sprache

Zur Erzeugung eines Wortes für die Wetterdarstellung beginnt man mit der ersten Regel, also mit der Variablen <Wetter>. Diese Variable heißt **Startvariable** (im folgenden auch **Startsymbol** genannt).

Eine Regel wird nun auf einen Term angewendet, indem eine enthaltene syntaktische Variable innerhalb des Terms durch die rechte Seite der entsprechenden Regel ersetzt wird. Dabei gibt es oftmals mehr als eine Regel zur Ersetzung derselben syntaktischen Variable. Für die Erzeugung eines beliebigen Wortes der Sprache kann irgendeine dieser Regeln ausgewählt werden. Ausgehend von der Startvariablen werden nun so lange Produktionen angewendet, bis nur noch Zeichen des Alphabets übrig bleiben.

Alle Wörter, die sich nun mithilfe des angegebenen Regelsatzes erzeugen lassen, werden als zur Sprache gehörig definiert.

Will man nachprüfen, ob ein Wort zu einer bestimmten Sprache gehört, muss eine Folge von Regeln gefunden werden, die aus dem Startsymbol schrittweise durch Anwendung der Regeln in geeigneter Reihenfolge das entsprechende Wort liefert. Eine derartige schrittweise

Erzeugung eines Wortes heißt auch **Ableitung** eines Wortes. Beispielsweise kann die Zeichenkette '-13°C0°C☼✷NO' wie folgt abgeleitet werden:

Regel	Fortschritt der Ableitung
(1)	<Wetter> → <Temperatur><Temperatur><Wetterlage><Windrichtung>
(2)	→ –<Zahl>°C<Temperatur><Wetterlage><Windrichtung>
(11)	→ –<Ziffer><Ziffer>°C<Temperatur><Wetterlage><Windrichtung>
(13)	→ –1<Ziffer>°C<Temperatur><Wetterlage><Windrichtung>
(15)	→ –13°C<Temperatur><Wetterlage><Windrichtung>
(3)	→ –13°C<Zahl>°C<Wetterlage><Windrichtung>
(10)	→ –13°C<Ziffer>°C<Wetterlage><Windrichtung>
(12)	→ –13°C0°C<Wetterlage><Windrichtung>
(5)	→ –13°C0°C<Wettersymbol><Wettersymbol><Windrichtung>
(6)	→ –13°C0°C☼<Wettersymbol><Windrichtung>
(9)	→ –13°C0°C☼✷<Windrichtung>
(24)	→ –13°C0°C☼✷<NS><OW>
(25)	→ –13°C0°C☼✷N<OW>
(28)	→ –13°C0°C☼✷NO

Die Reihenfolge der Regelanwendung ist dabei nicht genau festgelegt. Im vorliegenden Beispiel wurden die Termersetzungen von links nach rechts vorgenommen (Linksableitung).

1.2.4 Nichtterminale und Terminale

Da alle syntaktischen Variablen im Laufe der Ableitung eines Wortes ersetzt werden, bezeichnet man diese Symbole auch als **Nichtterminale**. Dagegen heißen alle Symbole, die zum Alphabet Σ der formalen Sprache gehören, **Terminale**, weil sie nicht weiter ersetzt werden.

1.2.5 Zusammenfassen von Regeln

Eine kompaktere Auflistung des Regelsatzes erhält man, wenn alle Regeln mit identischer linker Seite zusammenfasst werden. Die jeweiligen rechten Seiten werden durch einen senkrechten Strich | (im Sinne von "oder") getrennt. Im Beispiel können die 28 einzelnen Regeln aus Abschnitt 1.2.2 zu folgenden 9 Regeln reduziert werden:

(1) <Wetter> → <Temperatur><Temperatur><Wetterlage><Windrichtung>

(2) <Temperatur> → – <Zahl>°C | <Zahl>°C

(3) <Wetterlage> → <Wettersymbol> | <Wettersymbol><Wettersymbol>

(4) <Wettersymbol> → ☼ | ☁ | 🔥 | ✳

(5) <Zahl> → <Ziffer> | <Ziffer><Ziffer>

(6) <Ziffer> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(7) <Windrichtung> → <NS> | <OW> | <NS><OW>

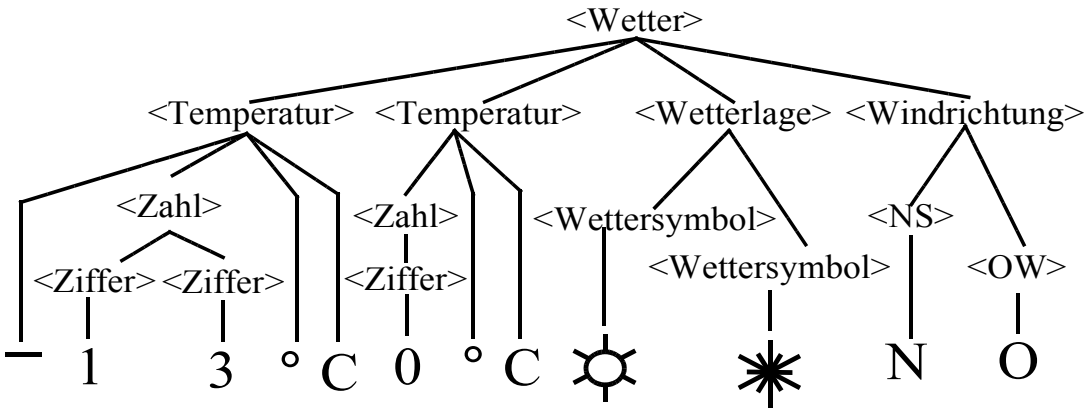
(8) <NS> → N | S

(9) <OW> → O | W

1.2.6 Ableitungsbäume

Eine übersichtliche grafische Darstellung der Ableitungsstruktur eines Wortes bietet ein Ableitungsbaum, der die möglichen Ableitungen eines Wortes unabhängig von der Reihenfolge der Regelanwendung übersichtlich darstellt. Die Wurzel stellt die Startvariable dar. Jeder Knoten hat so viele Kindknoten, wie die entsprechende Regel auf der rechten Seite Symbole (Terminale oder Nichtterminale) hat. Notiert man sie von links nach rechts in derselben Reihenfolge, so bilden die Blätter des Ableitungsbaums schließlich das abgeleitete Wort aus lauter Symbolen des Alphabets.

Das folgende Beispiel zeigt den Ableitungsbaum für das Wort '−13°C0°C☼✳NO'.



1.2.7 Grammatik einer formalen Sprache

Die Schülerinnen und Schüler sollen erkennen, dass eine formale Sprache durch die Angabe der **Grammatik** G definiert ist, welche sowohl die Syntax als auch Semantik der Sprache festlegt. Dabei besteht eine Grammatik aus folgenden Komponenten:

- der Menge der syntaktischen Variablen V (die Menge der Nichtterminale),
- dem Alphabet Σ der Sprache (Menge der Terminalsymbole),
- der Menge der Produktionen P,
- der Startvariablen $S \in V$.

Dabei dürfen Σ und V kein gemeinsames Element haben, d. h. $\Sigma \cap V = \{\}$.

Die formale Sprache legt alle Zeichenketten fest, die nur aus Terminalsymbolen aus Σ bestehen und ausgehend vom Startsymbol S mithilfe einer endlichen Anzahl von Anwendungen von Regeln aus P abgeleitet werden können.

1.3 Notationsformen formaler Sprachen

Lp: Zur Beschreibung (...) (von) Regeln verwenden (...) (Schülerinnen und Schüler) Textnotationen oder Syntaxdiagramme und können damit analog zu den natürlichen Sprachen Grammatiken für formale Sprachen definieren.

1.3.1 Didaktische Hinweise

Neben der in Kapitel 1.2 vorgestellten Möglichkeit, formale Sprachen mithilfe von Produktionsregeln einer Grammatik zu definieren, gibt es noch andere Alternativen.

Eine Möglichkeit ist die Formulierung der Regeln in der so genannten Backus-Naur-Form (BNF). Es wird empfohlen, die erweiterte Backus-Naur-Form (EBNF) zu verwenden, die zwar nicht mächtiger als die BNF ist, jedoch eine kompaktere Darstellung der Regeln ermöglicht. Zudem existiert für die EBNF ein festgelegter Standard. Anmerkung: Die EBNF ist von der ISO standardisiert unter der Nummer ISO/IEC 14977:1996(E).

Bei Sprachen mit nicht zu umfangreichem Regelsatz können Syntaxdiagramme eine übersichtliche grafische Darstellung der Produktionsregeln gewährleisten.

In diesem Abschnitt wird das leere Wort mit dem Symbol ε erstmals als sinnvolle Vereinfachung zur Notation von Regeln dargestellt.

1.3.2 Erweiterte Backus-Naur-Form

An einem Beispiel soll die Textnotation EBNF dargestellt und schrittweise verfeinert werden. Generell werden alle Terminalsymbole mit Hochkommas versehen. Dies hat auch den Vorteil, dass ein Leerzeichen durch ' ' repräsentiert werden kann. Auf die spitzen Klammern bei den Nichtterminalen kann dadurch verzichtet werden. Jede Regel wird mit einem Strichpunkt abgeschlossen.

Als textuelle Notation kommt die EBNF in der Darstellung den Produktionsregeln bei der Darstellung von Alternativen sehr nahe: Anstelle des Produktionspfeils schreibt man lediglich ein Gleichheitszeichen für die Zuordnung (EBNF, Variante 1).

Zusätzlich können in EBNF Optionen durch eckige Klammern und Ausdrücke, die ausgelassen oder beliebig oft wiederholt werden dürfen, mit geschweiften Klammern dargestellt werden (EBNF, Variante 2). Bei dieser kompakteren Variante der EBNF kann zudem auf rekursive Konstruktionsregeln (wenn z.B. auf der rechten Seite einer Regel dasselbe Nichtterminal steht wie auf der linken Seite) verzichtet werden, um eine Sprache mit unendlich vielen Wörtern darstellen zu können.

Die folgende Tabelle veranschaulicht am Beispiel der Sprache der Zahldarstellungen natürlicher Zahlen, wie aus den bekannten Produktionsregeln zunächst eine erste grobe Form einer EBNF gewonnen und diese dann unter Zuhilfenahme der oben angegebenen Darstellungsmöglichkeiten verfeinert wird:

Produktionsregeln
<Zahl> → <PositiveZahl> | <NegativeZahl> | 0
<PositiveZahl> → <ZifferNichtNull><Ziffernfolge>
<NegativeZahl> → -<PositiveZahl>
<Ziffernfolge> → <Ziffer><Ziffernfolge> | ε
<Ziffer> → <ZifferNichtNull> | 0
<ZifferNichtNull> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

EBNF (Variante 1)

```
Zahl = PositiveZahl | NegativeZahl | '0' ;
PositiveZahl = ZifferNichtNull Ziffernfolge ;
NegativeZahl = '-' PositiveZahl ;
Ziffernfolge = Ziffer Ziffernfolge | ε ;
Ziffer = ZifferNichtNull | '0' ;
ZifferNichtNull = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
```

EBNF (Variante 2)

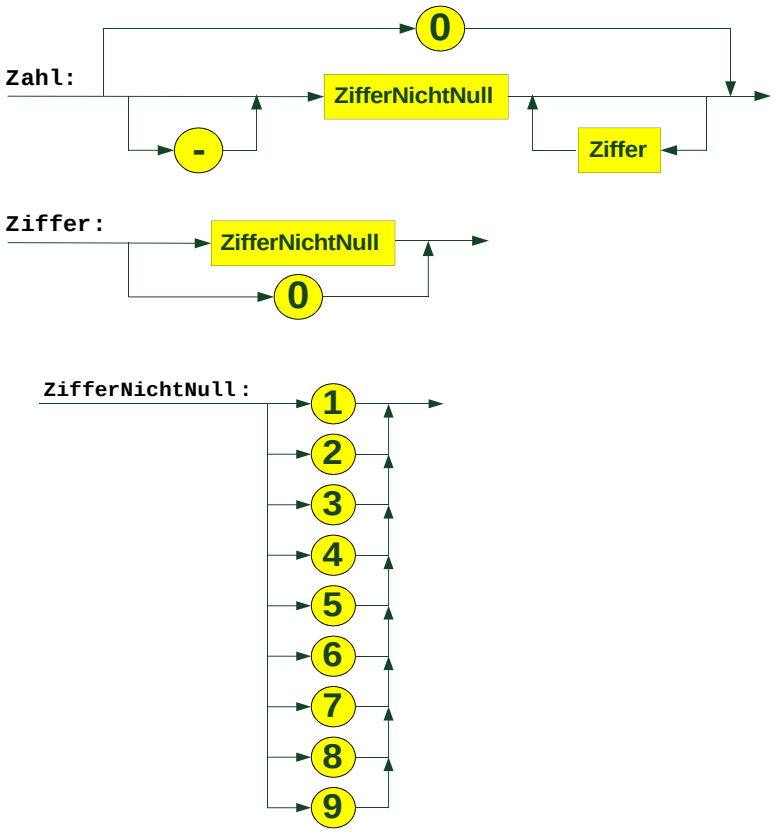
```
Zahl = ['-'] ZifferNichtNull { Ziffer } | '0' ;
Ziffer = ZifferNichtNull | '0' ;
ZifferNichtNull = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
```

1.3.3 Syntaxdiagramme

In Syntaxdiagrammen werden die Regeln einer Grammatik grafisch dargestellt. Für jede Regel gibt es ein eigenes Syntaxdiagramm, das mit dem Nichtterminal der linken Seite der entsprechenden Regel bezeichnet wird.

Man unterscheidet zwei verschiedene Arten von Knoten: Terminale werden durch Kreise, Nichtterminale durch Rechtecke repräsentiert. Verfolgt man einen Pfad vom Start- zum Endpfeil, so erhält man durch Aneinanderreihen der in den Knoten dargestellten Symbole eine mögliche rechte Seite, die in der entsprechenden Regel vom jeweiligen Nichtterminal abgeleitet ist.

Alternativen werden durch zusätzliche Pfade im Syntaxdiagramm dargestellt. Bei Optionen ist einer der alternativen Pfade ein Pfad ohne Symbol. Wiederholungen werden durch einen Rückwärtspfeil realisiert.



1.4 Erkennung formaler Sprachen

Lp: Die Zweckmäßigkeit der streng formalen Beschreibung formaler Sprachen zeigt sich den Jugendlichen bei der automatischen Überprüfung der syntaktischen Korrektheit von Zeichenketten mithilfe von endlichen Automaten.

1.4.1 Didaktische Hinweise

Die Schülerinnen und Schüler lernen, dass besonders einfach strukturierte formale Sprachen (nämlich die regulären Sprachen) mittels endlicher Automaten beschrieben werden können. Im Gegensatz zu den bereits bekannten Zustandsdiagrammen gibt es bei den endlichen Automaten keine bedingten Übergänge.

Im Vordergrund stehen dabei deterministische endliche Automaten (DEA). Da in manchen Anwendungsfällen die Modellbildung durch einen nichtdeterministischen endlichen Automaten (NEA) für Schülerinnen und Schüler naheliegender bzw. intuitiver realisierbar ist, ist auch die Behandlung nichtdeterministischer endlicher Automaten im Unterricht an ausgewählten Beispielen sinnvoll und möglich. Keinesfalls sollte von Schülerinnen und Schülern jedoch die systematische Konstruktion äquivalenter deterministischer endlicher Automaten aus nichtdeterministischen Automaten verlangt werden. Ebenso ist die Einschränkung auf deterministische endliche Automaten erforderlich, wenn es um die Transformation des Automatenmodells in ein Programm geht.

Ziel dieses Abschnitts ist die Behandlung erkennender Automaten, welche lediglich feststellen, ob ein bestimmtes Wort (Eingabewort) zu der durch den Automaten repräsentierten Sprache gehört oder nicht. Die Thematisierung von Automaten mit einer Ausgabe ist durch den Lehrplan nicht gefordert.

1.4.2 Endliche Automaten

Endliche, erkennende Automaten bestehen aus endlich vielen Zuständen und verarbeiten eine aus endlich vielen Zeichen bestehende Zeichenkette, deren Symbole Elemente des Eingabealphabets Σ sind. Die Abarbeitung des Eingabewortes erfolgt dabei zeichenweise. Hierbei beginnt der Automat an einem eindeutig festgelegten Startzustand (gekennzeichnet durch einen Anfangspfeil) und beendet die Verarbeitung an dem Zustand, von dem aus kein weiterer Verarbeitungsschritt möglich ist. Falls dieser zuletzt erreichte Zustand ein festgelegter Endzustand (gekennzeichnet durch eine doppelte Umrandung) ist und die Zeichenkette ganz abgearbeitet ist, gehört das überprüfte Wort zur Sprache, sonst nicht.

Als mögliches Einstiegsbeispiel aus der Erfahrungswelt der Schülerinnen und Schüler kann die Erkennung maschinenlesbarer Strichcodes durch Scannerkassen gewählt werden.

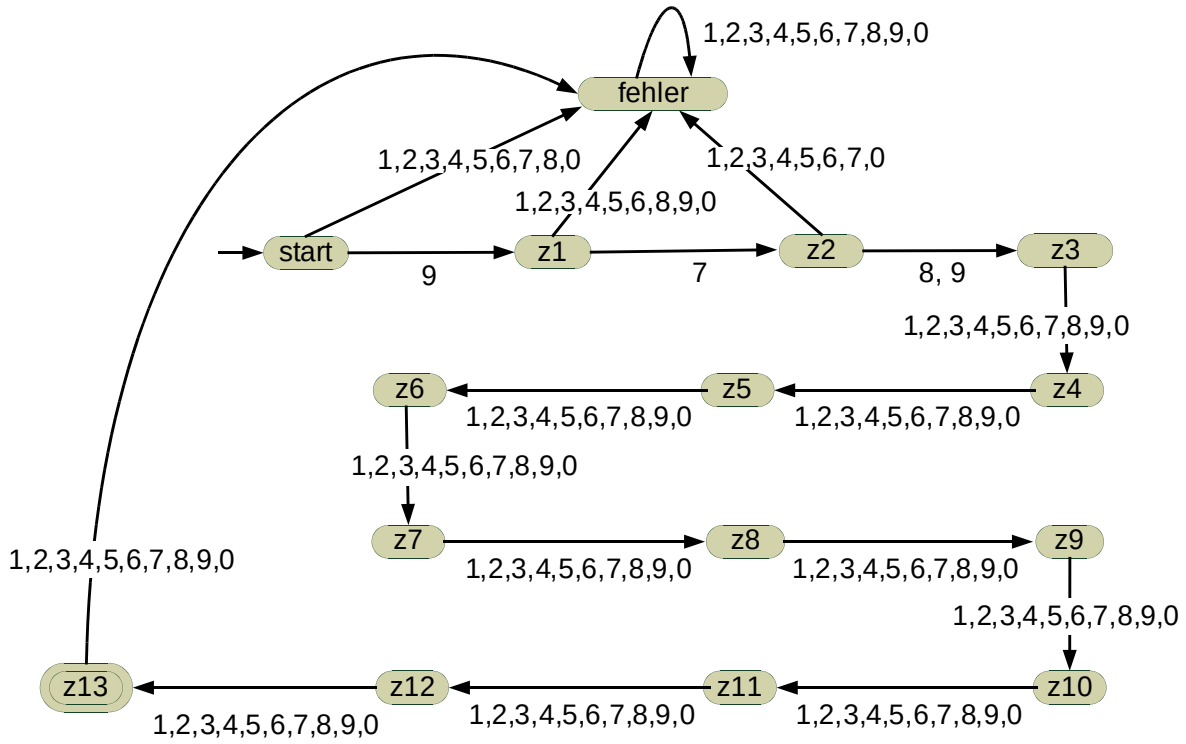
Auf nahezu allen Warenpackungen sind so genannte EAN-Nummern (EAN steht für European Article Number) als Strichcodes aufgedruckt, die z.B. im Fall der EAN-13-Nummern für die ISBN von Bücher eine bestimmte Struktur aufweisen.

Die 13-stellige EAN-13 ist wie folgt aufgebaut:

- Die ersten beiden Ziffern sind 97, gefolgt von 8 oder 9.
- Danach folgt eine Kennung für das jeweilige Land, z.B. 0 und 1 für den englischsprachigen, 2 für den französischsprachigen, 3 für den deutschsprachigen Raum, 88 für Italien, 99953 für Paraguay.
- Anschließend folgt eine Ziffernfolge für die Kennung des Verlags, gefolgt von einer verlagsinternen Titelnummer, wobei die Anzahl der Ziffern dann genau 12 betragen muss.

– Zum Schluss wird eine Prüfziffer angegeben, die das Erkennen von Lesefehlern möglich machen soll. (Hinweis: Die Prüfziffer z_{13} an der 13. Stelle ist korrekt, wenn gilt: $(z_1+z_3+z_5+z_7+z_9+z_{11}+z_{13}+3\cdot(z_2+z_4+z_6+z_8+z_{10}+z_{12}))\ mod\ 10=0$, wobei z_i die Ziffer an der i-ten Stelle bedeutet.)

Die Arbeitsweise eines einfachen erkennenden Automaten, der nur die Länge der EAN-Nummer und die ersten drei Ziffern, nicht jedoch die Prüfziffer kontrollieren kann, wird am vorgegebenen Automatenmodell anhand verschiedener Eingabewörter erläutert, genauer: Der Automat kontrolliert zwar, ob die Prüfziffer vorhanden ist (syntaktische Korrektheit); den Wert der Prüfziffer (Semantik des Wortes) kann dieser jedoch nicht verifizieren.



Der Automat hat insgesamt 15 Zustände mit *start* als Startzustand und *z13* als einzigem Endzustand. An den Übergängen stehen als auslösende Aktionen Symbole aus dem Alphabet {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}. Vereinfachend werden alle Zeichen, die von einem Zustand aus zum gleichen Folgezustand führen, über nur einen Transitionspfeil durch Kommas getrennt angegeben. Beispielsweise erfolgt aus dem Zustand *start* ein Zustandsübergang in den Zustand *fehler*, falls das aktuelle Symbol eines der Zeichen '1', '2', '3', '4', '5', '6', '7', '8' oder '0' ist.

Der Automat startet im Zustand *start*.

– Bei Eingabe der Zeichenkette '9741018995322' beispielsweise wird derjenige Zustandsübergang ausgelöst, der mit dem Zeichen '9' markiert ist. Der Automat gelangt somit in den Zustand *z1*. Das nächste Zeichen des Eingabewortes ist '7'. Im Beispiel wird der Automat also im zweiten Schritt den Zustand *z2* einnehmen. Das dritte Zeichen '4' jedoch macht bereits an dieser Stelle deutlich, dass das Eingabewort sicher nicht zur oben angegebenen Sprache gehören kann. Der Zustandsübergang führt den Automaten in den Zustand *fehler*, der kein Endzustand ist. Diesen Zustand kann der Automat nicht mehr verlassen, da von diesem keine Transition wegführt. Der Automat akzeptiert also das Wort '9741018995322' nicht.

– Bei Eingabe der Zeichenkette '9783127316681' werden der Reihe nach die Zustände *start*, *z1*, *z2*, *z3*, *z4*, *z5*, *z6*, *z7*, *z8*, *z9*, *z10*, *z11*, *z12*, *z13* eingenommen. Da das Eingabewort vollständig abgearbeitet und der letzte eingenommene Zustand ein Endzustand ist, gehört das Wort '9783127316681' zur beschriebenen Sprache.

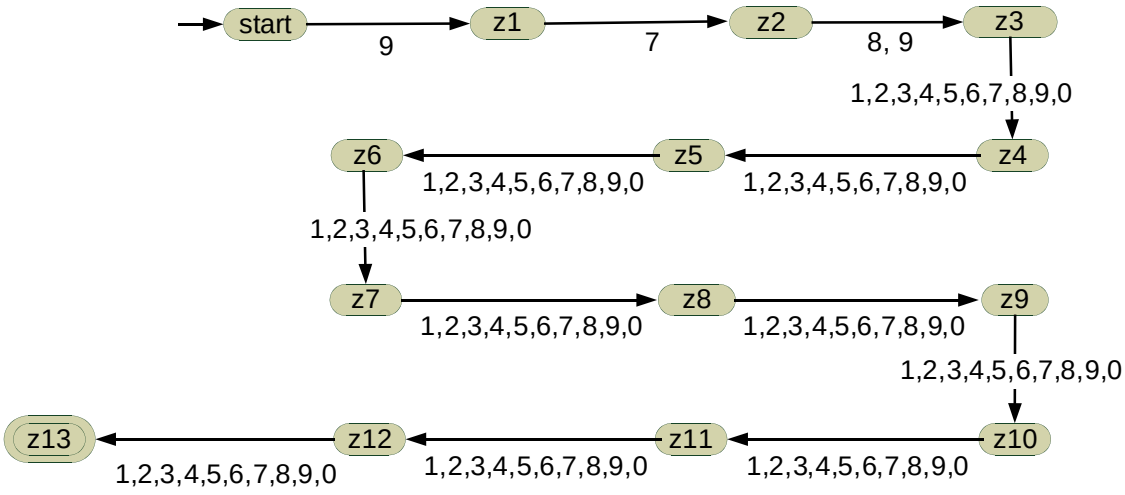
– Verfolgt man dagegen die erreichten Zustände bei der Eingabe der Zeichenkette '97830601434603', so erkennt man, dass der Automat im Endzustand *z13* noch nicht alle Zeichen abgearbeitet hat: das letzte Zeichen '3' führt den Automaten vom Endzustand *z13* weg in den Zustand *fehler*. Der Automat stoppt hier also in einem Zustand, der nicht Endzustand ist. Somit gehört das entsprechende Wort nicht zur angegebenen Sprache, da es zu lange ist.

1.4.3 Vereinfachte Darstellung endlicher Automaten

Im unter Punkt 1.4.2 dargestellten Beispiel führt von jedem Zustand aus zu jedem Zeichen des angegebenen Alphabets {1, 2, 3, 4, 5, 6, 7, 8, 9, 0} eine Transition weg, da so jedes beliebige Wort aus Symbolen des Alphabets abgearbeitet werden kann. Besonders bei Automaten, die bei mehr als drei (wie oben) Übergängen nur bestimmte Symbole des Alphabets zulassen, wird die Darstellung sehr schnell unübersichtlich. Daher vereinbart man folgende Vereinfachung:

Man notiert nur diejenigen Transitionen und Zustände, die für die Erkennung aller korrekten Wörter notwendig sind. Fehlerzustände lässt man also mit den dort endenden Zustandsübergängen weg. Wenn also bei der Abarbeitung eines Wortes von einem Zustand aus keine Transition mit dem aktuellen Zeichen existiert, so gehört dieses Wort nicht zur Sprache.

Im obigen Beispiel vereinfacht sich der Automat wie folgt:



1.4.4 Formale Definition determistischer endlicher Automaten

Wenn bei einem Automaten in jedem Zustand jedes Zeichen des Alphabets höchstens eine Transition auslösen kann, spricht man von einem **deterministischen endlichen Automaten** (DEA). Wenn es jedoch mindestens einen Zustand gibt, von dem aus ein Eingabezeichen eine Transition zu verschiedenen Folgezuständen auslösen kann, ist der Automat nichtdeterministisch (NEA). Die Behandlung von NEAs ist laut Lehrplan nicht verlangt. Daher wird im folgenden Kapitel darauf nicht eingegangen.

Jeder deterministische endliche Automat wird durch folgende fünf Komponenten beschrieben:

1. eine endliche Menge Z von Zuständen,

   (im Beispiel: Z = {*start*, *z1*, *z2*, *z3*, *z4*, *z5*, *z6*, *z7*, *z8*, *z9*, *z10*, *z11*, *z12*, *z13*, *fehler*})

2. ein endliches Eingabealphabet Σ,

   (im Beispiel: Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

3. einen Startzustand,

   (im Beispiel: *start*)

4. eine Menge E von Endzuständen,

   (im Beispiel: E = {*z13*})

5. eine zweistellige Übergangsfunktion δ, die jedem möglichen Paar aus Zustand und Eingabezeichen einen Folgezustand zuordnet,

   (im Beispiel: δ (*start*, '9') = *z1* oder δ (*start*, '3') = *fehler)*.

Aus jedem DEA lässt sich relativ leicht die zugehörige Grammatik entwickeln, die genau die Sprache beschreibt, welche der Automat akzeptiert. Dazu führt man folgende Schritte durch:

1. die Menge der Zustandsbezeichner Z wird zur Menge der Nichtterminalen V,

   im Beispiel V = {start, z1, z2, z3, z4, z5, z6, z7, z8, z9, z10, z11, z12, z13, fehler};

2. das Eingabealphabet Σ wird zur Menge der Terminale Σ,

   im Beispiel Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

3. der Startzustand wird zum Startsymbol S

   im Beispiel S = <start>;

4. jede Funktionsanwendung der Übergangsfunktion δ (*z1*, 'a') = *z2* wird durch die Produktionsregel <z1> → 'a' <z2> ersetzt,

   im Beispiel etwa <start> → '9' <z1>, <start> → '3' <fehler> oder <z12> → '0' <z13>;

5. für jeden Endzustand *ze* legt man zusätzlich eine Produktion fest, bei deren Anwendung das Nichtterminal <ze> durch das "leere" Wort ε ersetzt wird: <ze> → ε,

   im Beispiel: <z13> → ε.

Man kann also zu jedem endlichen deterministischen Automaten eine Grammatik für die Sprache angeben, die der Automat akzeptiert. Dies ist umgekehrt nicht immer möglich: Es gibt Sprachen, die nicht durch einen endlichen Automaten beschrieben werden können, vgl. Kapitel 1.4.5.

Sprachen, die durch endliche Automaten beschrieben werden können, bilden also eine echte Teilmenge aller möglichen Sprachen. Sie werden auch **reguläre Sprachen** genannt. Für die regulären Sprachen gilt, dass alle Produktionen in der Form

<Nichtterminal> → 'Terminalsymbol' <Nichtterminal> bzw.

<Nichtterminal> → 'Terminalsymbol'

notiert werden können.

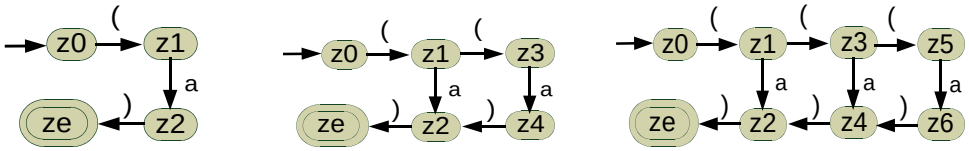
1.4.5 Grenzen endlicher Automaten

Wie in Kapitel 1.4.4 bereits bemerkt, gibt es formale Sprachen, welche nicht von einem endlichen Automaten erkannt werden können. Beispielsweise können hier Sprachen genannt werden, welche beliebig tief geschachtelte Klammerterme enthalten.

Vereinfacht soll nun die Sprache L betrachtet werden, welche alle Wörter enthält, die das Symbol 'a' beliebig oft, aber mindestens einmal umklammern, also L = {(a), ((a)), (((a))), ...}. Diese Sprache kann nicht durch einen endlichen Automaten beschrieben werden.

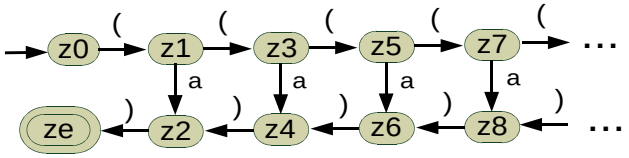
Zur Begründung dieser Behauptung werden zunächst die Teilsprachen L_n von L definiert, welche das Symbol 'a' höchstens n-mal umklammern, also L_1 = {(a)}, L_2 = {(a), ((a))}, L_3 = {(a), ((a)), (((a)))}, usw.

Zu diesen Teilsprachen L_n lässt sich jeweils ein endlicher Automat zeichnen, wie die folgenden drei Abbildungen zeigen:



Für n=1 werden also vier Zustände, für n=2 sechs Zustände und für n=3 acht Zustände gebraucht. Für jede zusätzliche Klammerebene müssen zwei weitere Zustände hinzukommen, da sich der Automat über die Zustände „merkt“, wie viele Klammern bereits offen sind und daher wieder geschlossen werden müssen. Allgemein benötigt man im Falle n insgesamt 2n+2 Zustände. Jede der Sprachen L_n ist also regulär.

Wenn nun aber die Anzahl der Klammerebenen nicht beschränkt sein soll, wie etwa bei der Sprache L, würde man unendlich viele Zustände benötigen, wie folgende Abbildung andeutet:



Die Menge der Zustände endlicher Automaten muss jedoch laut Definition endlich sein. Dies macht intuitiv klar, dass die Sprache L nicht durch einen endlichen Automat beschrieben werden kann. Die Sprache L ist also keine reguläre Sprache.

1.5 Implementierung erkennender Automaten

Lp: Endlicher Automat als geeignetes Werkzeug zur Syntaxprüfung für reguläre Sprachen; Implementierung eines erkennenden Automaten

1.5.1 Didaktische Hinweise

Für die Implementierung werden deterministische endliche Automaten durch spezielle Zustandsübergangsdiagramme (ohne bedingte Übergänge und ohne Ausgabe) repräsentiert. Abhängig vom jeweiligen Zustand führt die auslösende Aktion (das Lesen des nächsten Zeichens des Eingabewortes) je nach Symbol eventuell zu einem anderen Folgezustand. Daher verwendet man als Kontrollstruktur eine Fallunterscheidung über die einzelnen Zustände, die jeweils eine Fallunterscheidung über die Eingabezeichen des Alphabets enthalten.

Zu beachten ist, dass bei der Implementierung der Fehlerzustand in jedem Fall berücksichtigt werden sollte.

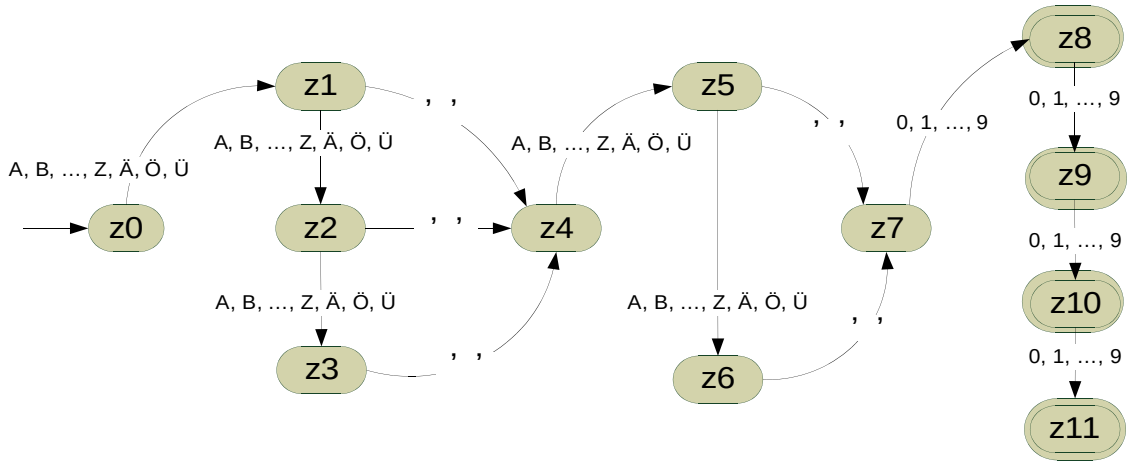
1.5.2 Formulierung des Algorithmus

Die Schülerinnen und Schüler arbeiten zunächst mit einem bereits gegebenen Automatenmodell und formulieren einen Algorithmus in Pseudocode. Als Beispiel soll ein Automat herangezogen werden, der überprüft, ob ein eingegebenes Wort dem Format eines KFZ-Kennzeichens entspricht, d. h. wie folgt aufgebaut ist:

1. das Wort beginnt mit einer Buchstabenkombination aus ein bis drei Großbuchstaben;
2. anschließend folgt ein Leerzeichen;
3. danach schließt sich eine Buchstabenkombination aus ein oder zwei Großbuchstaben an;
4. wiederum folgt ein Leerzeichen;
5. das Wort endet mit einer Zahl aus ein bis maximal vier Ziffern.

Hinweis: Hier wird lediglich gefordert, dass die zugelassenen Wörter dem Format eines KFZ-Kennzeichens entsprechen, d. h. sie umfassen auch Buchstabenkombinationen, die in der Realität nicht zugelassen sind, wie z. B. 'ÄÄÄ Ü 0000'.

Der unten abgebildete endliche Automat erkennt genau die durch die fünf Regeln beschriebenen Wörter.



Der erkennende Automat hat in der vereinfachten Darstellung (d. h. ohne Fehlerzustand) 12 Zustände *z0*, *z1*, ..., *z11*. Dabei sind *z8*, *z9*, *z10*, *z11* allesamt Endzustände, da die Autonummer ein- bis vierstellig sein darf. Für die Implementierung nimmt man als Fehlerzustand den Zustand *z12* hinzu.

Im Anfangszustand *z0* führt nur einer der Großbuchstaben aus der Menge {A, B, ..., Z, Ä, Ö, Ü} zum Folgezustand *z1*. Bei allen anderen Eingabezeichen ist an dieser Stelle schon klar, dass die entsprechende Zeichenkette kein gültiges KFZ-Kennzeichen sein kann. Aus dem Modell wird ebenso deutlich, dass im Zustand *z11* kein weiteres Eingabezeichen erlaubt ist. Dies realisiert man dadurch, dass der Automat in den Fehlerzustand *z12* übergeht, falls im Zustand *z11* ein beliebiges Zeichen eingegeben wird.

Ein Algorithmus, der innerhalb einer Methode *zustandWechseln* abgearbeitet wird, könnte wie folgt beschrieben werden:

```
Methode zustandWechseln(eingabezeichen)
        wenn zustand ist gleich z0
                wenn eingabezeichen aus {A, B, C, D, E, ..., Z, Ä, Ü, Ö}:
                        zustand = z1
                sonst
                        zustand = z12
                endewenn
        sonst
        wenn zustand ist gleich z1
                wenn eingabezeichen aus {A, B, C, D, E, ..., Z, Ä, Ü, Ö}
                        zustand = z2
                sonst
                wenn eingabezeichen ist gleich ' ':
                        zustand = z4
                sonst
                        zustand = z12
                endewenn
        sonst
        . . .  // Fälle für die Zustände z2 bis z10
        wenn zustand ist gleich z11
                wenn eingabezeichen ist beliebig
                        zustand = z12
                endewenn
        endewenn
endeMethode
```

Für die Untersuchung, ob eine eingegebene Zeichenkette ein gültiges KFZ-Kennzeichen darstellt, wird noch eine Methode *wortUntersuchen* definiert, welche die zu untersuchende Zeichenkette übernimmt und der Reihe nach jedes Zeichen des übergebenen Wortes von vorne nach hinten abarbeitet, indem sie diese jeweils der Methode *zustandWechseln* übergibt. Als Ergebnis wird wahr zurückgegeben, wenn sich am Ende der Methodenausführung der endliche Automat in einem der Zustände *z8*, *z9*, *z10* oder *z11* befindet.

In Pseudocode könnte diese Methode wie folgt formuliert werden:

```
Methode wortUntersuchen(kennzeichen){
        kennzeichenIstKorrekt = falsch
        wiederhole vom ersten bis zum letzten Zeichen von kennzeichen
                zustandWechseln(zeichen)
        endewiederhole
        wenn zustand ist gleich z8 oder z9 oder z10 oder z11
                kennzeichenIstKorrekt = wahr
        endewenn
        zustand = z0
        gib den Wert von kennzeichen IstKorrekt zurück
endeMethode
```

1.5.3 Implementierung

Nachfolgend wird eine Implementierung der Klasse AUTOKENNZEICHEN in der Programmiersprache Java gezeigt:

```
public class Autokennzeichen {
   private int zustand;
   public Autokennzeichen() {
      zustand = 0; // Anfangszustand z0
   }
   public boolean wortUntersuchen(String kennzeichen){
      boolean kennzeichenIstKorrekt=false;
      for (int i=0;i<kennzeichen.length();i++){
         zustandWechseln(kennzeichen.charAt(i));
      }
      if ((zustand==8) || (zustand==9) || (zustand==10) || (zustand==11)){
         kennzeichenIstKorrekt=true;
      }
      zustand = 0; // Anfangszustand z0
      return kennzeichenIstKorrekt;
   }
   private void zustandWechseln(char eingabezeichen) {
      switch (zustand) {
         case 0 : {
            switch (eingabezeichen) {
               case 'A':
               case 'B':
               ...
               case 'Ö':
               case 'Ü': {zustand = 1;}  break;
               default : {zustand = 12;} break;
            }
         }
         break; // ende case 0

         case 1 : {
            switch (eingabezeichen) {
               case 'A':
               case 'B':
               ...
               case 'Ö':
               case 'Ü': {zustand = 2;}  break;
               case ' ': {zustand = 4;}  break;
               default : {zustand = 12;} break;
            }
         }
         break; // ende case 1

         ... // Fälle für die Zustände 2, 3, 4, 5

         case 6 : {
            switch (eingabezeichen) {
               case ' ': {zustand = 7;}  break;
               default : {zustand = 12;} break;
            }
         }
         break; // ende case 6

         ... // Fälle für die Zustände 7, 8, 9, 10

         case 11 : {
            switch (eingabezeichen) {
                default : {zustand = 12;} break;
            }
         }
         break; // ende case 11

        } // ende switch zustand
    } // ende void zustandWechseln
}
```

2 Kommunikation und Synchronisation von Prozessen (ca. 20 Std.)

Lp: Beim weltweiten Austausch von Information spielt die Kommunikation zwischen vernetzten Rechnern eine entscheidende Rolle. Die Schüler lernen, dass es hierzu fester Regeln für das Format der auszutauschenden Daten sowie für den Ablauf des Kommunikationsvorgangs bedarf. Der gemeinsame Zugriff auf Ressourcen führt sie zum Problem der Kommunikation und Synchronisation parallel ablaufender Prozesse, bei dessen Lösung die Jugendlichen erneut den Anwendungsbereich ihrer Modellierungstechniken erweitern.

Bei jedem der nachfolgenden Kapitel wird eine kurze inhaltliche Zusammenfassung der Grundlagen gegeben. Diese inhaltlichen Grundlagen sind unabhängig von der Umsetzungsreihenfolge im Unterricht und gehen an manchen Stellen etwas tiefer als im Unterricht benötigt, wenn vermutet werden kann, dass hier Schülerfragen auftreten.

In den Abschnitten „Umsetzungshinweise“ werden einerseits zur Unterrichtsunterstützung geeignete Programme vorgestellt, andererseits javaspezifische Hinweise zu den auf der CD zur Verfügung gestellten Projekte gegeben.

Darüber hinaus werden im Anhang dieses Kapitels hilfreiche Praxistipps angeführt.

2.1 Topologie von Rechnernetzen; Internet als Kombination von Rechnernetzen

Lp: Topologie von Rechnernetzen (Bus, Stern, Ring); Internet als Kombination von Rechnernetzen

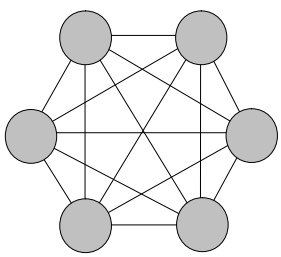
2.1.1 Inhaltliche Grundlagen

Die Verbindung mehrerer Rechner zu einem Netzwerk kann verschiedene Strukturen (Topologien) aufweisen.

Die wichtigsten Grundstrukturen (Bus-, Stern- oder Ringtopologie) sollen in diesem Kapitel besprochen werden. Für ein möglichst gut funktionierendes Netzwerk können dabei folgende Bewertungsaspekte für die verschiedenen Topologien dienen (siehe auch Übersichtstabelle):

- **Störanfälligkeit**: Wie störanfällig ist das Netzwerk beim Ausfall eines Verbindungspartners (Rechners) bzw. bei einer Trennung einer Verbindungsleitung?
- **Systempflege**: Wie groß ist der Aufwand bei der Fehlersuche? Ist eine einfache Erweiterbarkeit des Netzes gegeben?
- **Kosten:** Wie viele Leitungen werden benötigt? Wie hoch sind die Kosten spezieller Komponenten?
- **Übertragungsrate:** Wie gut ist die Übertragungsrate bzw. wie verändert sich diese, wenn sich die Anzahl der Verbindungsteilnehmer erhöht? Ist gleichzeitiges Senden und Empfangen möglich oder kommt es möglicherweise zu sogenannten „Kollisionen“?
  Anmerkung: Die genaue Analyse der Übertragungsrate erfordert sehr viel Spezialwissen und physikalische Kenntnis!

Direkte Verbindung aller beteiligter Rechner (Vermaschtes Netz)



Der naive Versuch, ausgehend von zwei Rechnern, alle Rechner eines Netzwerks direkt zu verbinden, zeigt sehr schnell, dass eine viel zu große Anzahl von Leitungen erforderlich wäre.

Bei sechs Rechnern benötigt man z. B. schon 5 + 4 + 3 + 2 + 1 = 15 Verbindungsleitungen. Für n Rechner würde man $(n-1)+(n-2)+....+1=\frac{n\cdot(n-1)}{2}$ Verbindungsleitungen benötigen, da bei jedem zusätzlichen i-ten Rechner (i – 1) Leitungen hinzukommen.

Eine solche Netzwerkstruktur ist zwar sehr sicher, da ein Ausfall eines Verbindungspartners andere Verbindungen nicht beeinflusst, zur Verbindung einer großen Anzahl von Rechnern innerhalb eines Netzwerks aufgrund der vielen benötigten Verbindungsleitungen allerdings nicht geeignet.

Adressierung von Rechnern

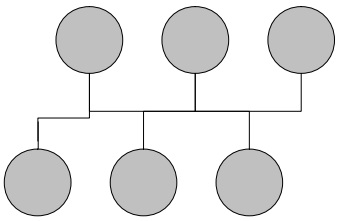
Wenn Rechner nicht mehr direkt miteinander verbunden werden, sondern sich wie in den nachfolgenden Netzwerktopologien das Verbindungsmedium teilen, muss jeder Rechner im Netzwerk einen eindeutigen Bezeichner besitzen.

Im Kontext des Internets werden als Bezeichner sogenannte IP-Adressen verwendet. Diese bestehen aus einer 32-Bit-Zahl, die aus Lesbarkeitsgründen byteweise (Zahlen zwischen 0 und 255 und durch Punkt getrennt) dargestellt wird:

So ergibt sich aus der binär dargestellten Adresse $01111111000000000000000000000001_2$ bzw. 2130706433 in dezimaler Schreibweise die Darstellung 127.0.0.1 der IP-Adresse für den lokalen Rechner.

Weil IP-Adressen für Menschen schlecht lesbar sind, können diesen IP-Adressen über sogenannte DNS-Server lesbare Bezeichnungen zugeordnet werden und umgekehrt. Wird eine Anfrage (beispielsweise durch einen Webseitenaufruf) an die Adresse „alp.dillingen.de“ gestartet, so erfolgt zunächst eine Namensauflösung durch einen DNS-Server, danach werden die Datenpakete an die richtige IP-Adresse (194.95.207.10) über verschiedene Stationen gesandt (siehe auch Abschnitt 2.1.2).

Bustopologie

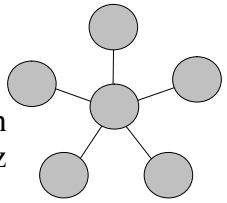


Sollen Rechner möglichst einfach und kostengünstig miteinander verbunden werden, so kann die Bustopologie verwendet werden. Weil dabei alle Netzwerkgeräte an ein gemeinsames Übertragungsmedium angeschlossen sind, muss ein gleichzeitiges Senden von Daten (Kollision) vermieden werden. Jedes angeschlossene Gerät prüft, ob die Leitung zur Übertragung frei ist, und beginnt zu senden. Falls ein weiteres Gerät gleichzeitig den Sendevorgang beginnt, kommt es zur Kollision, da alle Geräte ein gemeinsames Kabel benutzen. Diese Kollision muss im Netzwerk erkannt und der Sendevorgang wiederholt werden. Bei hohem Netzwerkverkehr steigt die Anzahl der Kollisionen überproportional, was zu einer starken Verminderung der tatsächlich übertragenen Daten führt.

Die Nachricht wird an alle Teilnehmer übertragen und muss von allen angeschlossenen Teilnehmern gelesen werden können, damit diese entscheiden können, ob die Nachricht für sie selbst bestimmt ist.

Vertiefende Information: Dadurch ist sogenanntes Sniffing, also das Mithören des Netzwerkverkehrs, möglich!

Sterntopologie

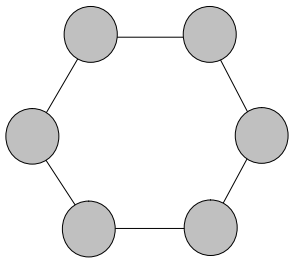


Jeder Rechner (jedes Endgerät) wird direkt mit einem zentralen Knoten verbunden. Selbst wenn ein Endgerät ausfällt oder nur zeitweise mit dem Netz

verbunden ist, hat dies keine Auswirkungen auf das gesamte Netzwerk. Fällt allerdings der Knoten aus, so ist das gesamte Netzwerk gestört.

Vertiefende Information: In einem in Schichten aufgebauten Netzwerk muss klar werden, auf welcher Schicht die Topologie gesehen wird. In der Literatur wird hier häufig zwischen logischer und physischer Ebene unterschieden. Bei der twisted-pair Ethernetverkabelung über einen Hub oder Switch entsteht physikalisch ein Stern, auf logischer Ebene jedoch ein Bus oder Punkt-zu-Punkt-Verbindungen.

Token-Ring



Im Token-Ring sind Rechner ringförmig miteinander verbunden. Es treten keine Kollisionen auf, da ein Rechner nur dann senden darf, wenn er ein spezielles Datenpaket (das Token) besitzt, das von der Kontrolleinheit des Rings zugeteilt wird. Deshalb bleibt auch bei hohen Lasten die Übertragungsrate konstant. Da eine Nachrichtenübertragung allerdings höchstens so lange dauern kann, wie das Signal zum Umlauf um den Ring benötigt, liegt die Übertragungsrate unterhalb der eines Bussystems.

Vertiefende Information: Wird das Netzwerk unterbrochen, so kann der Token nicht mehr rundum laufen. Die Netzwerkübertragung erfolgt dann über die verbleibende Verbindung mit hin- und herlaufendem (pendelndem) Token.

Im tabellarischen Überblick sind exemplarisch vorteilhafte Aspekte mit [+] und nachteilige mit [-] gekennzeichnet:

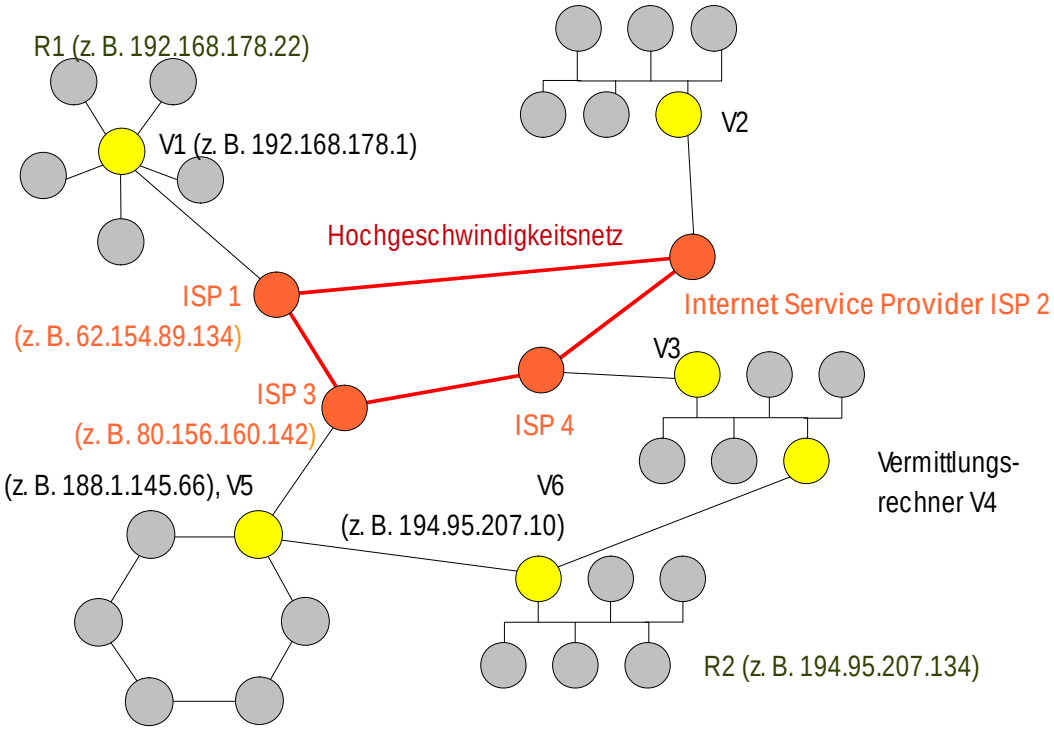
	Bus	Stern	Token-Ring
Ausfallsicherheit	[+] Ausfall eines Gerätes hat für die Funktionalität des Netzwerkes keine Konsequenzen [-] Störung des Übertragungsmediums an einer einzigen Stelle im Bus (defektes Kabel) blockiert den gesamten Netzstrang	[+] Der Ausfall eines Endgerätes hat keine Auswirkung auf den Rest des Netzes [-] Durch Ausfall des zentralen Knotens wird Netzverkehr unmöglich	[+] Pendelbetrieb bei Unterbrechung einer Leitungsverbindung ist möglich
Kosten, Erweiterbarkeit	[+] geringe Kosten, da nur geringe Kabelmengen und keine aktiven Netzwerkkomponenten nötig; leicht erweiterbar	[+] geringe Kosten, da nur geringe Kabelmengen und keine aktiven Netzwerkkomponenten nötig; leicht erweiterbar	[-] teure Komponenten
Sicherheit	[-] Datenübertragungen können leicht abgehört werden (Sniffing)		[-] Datenübertragungen können leicht abgehört werden (Sniffing)
Übertragungsrate	[-] Reduktion der Übertragungsrate bei hohem Netzwerkverkehr durch Kollisionen	[-] niedrige Übertragungsrate bei vielen Hosts, wenn ein Hub benutzt wird	[+] gleichzeitiges Senden wird durch Token vermieden, dadurch garantierte Übertragungsbandbreite
Anwendungsbereiche	kleine Netzwerke	Anschluss von Haushalten an Verbindungsrechner des Providers, Heimnetzwerke, Computerräume	Rechnernetze in Universitäten, werden aus v.a. Kostengründen nach und nach durch andere Strukturen abgelöst.

Internet als Kombination von Rechnernetzen

Da jede der oben beschriebenen Strukturen Vor- und Nachteile hat, verwendet man eine Kombination der verschiedenen Topologien, um weltweit Millionen von Rechnern zum Internet zu verbinden. Dabei werden die Rechnernetze (Teilnetze) über Verbindungsrechner (sogenannte Router) miteinander verbunden. Das dadurch entstehende globale Netz ist besonders ausfallsicher, da durch geschickte Verbindung der Teilnetze alternative Datenwege entstehen.

Verbindungsrechner sind für die Vermittlung und Weiterleitung (Routing) der Datenpakete zuständig und können mit Postämtern verschiedener Zustellbezirke (eigenes Teil- bzw. Subnetz) verglichen werden:

Ein Datenpaket startet zunächst im eigenen Subnetz. Anhand der IP-Adresse kann der Vermittlungsrechner erkennen, ob das Datenpaket für ein fremdes Subnetz bestimmt ist, und gegebenenfalls weiterleiten (Vergleich: Das Postamt eines Zustellbezirks leitet einen Brief an eine Adresse eines anderen Zustellbezirks weiter an das zuständige Postamt). Gibt es alternative Weg durch das Netz, so versucht man, Wege mit starker Netzlast zu vermeiden. Auch bei Störungen reagieren die Router und schicken die Datenpakete über alternative Wege im Netz. So können zum Beispiel, wie in der nachfolgenden Graphik dargestellt, Daten vom Rechner R1 über V1, ISP 1 (Internet Service Provider), ISP 3, V5, V6 zu R2 oder von R1 über V1, ISP1, ISP2, ISP4, V3, V4, V6 zu R2 geschickt werden.



Vertiefende Information: Das Hochgeschwindigkeitsnetz, das die Internetserviceprovider verbindet (in der Graphik rot gezeichnet), wird oft auch als „Backbone“ (= Rückgrat) bezeichnet. Auf die Einführung dieses zusätzlichen Begriffs wird hier bewusst verzichtet.

2.1.2 Umsetzungshinweise

Um die Netzwerkinformationen bei einem Rechner mit dem Betriebssystem Windows zu ermitteln, startet man das Programm „cmd.exe“ und gibt dort den Befehl „ipconfig“ ein. Man erhält man eine Übersicht über alle Netzwerkadapter des Computers:

```
Drahtlos-LAN-Adapter Drahtlosnetzwerkverbindung:

   Medienstatus. . . . . . . . . . . . : Medium getrennt
   Verbindungsspezifisches DNS-Suffix:

Ethernet-Adapter LAN-Verbindung:

   Verbindungsspezifisches DNS-Suffix:
   Verbindungslokale IPv6-Adresse  . : fe80::4c49:c13:ac0c:43e1%12
   IPv4-Adresse  . . . . . . . . . . : 192.168.178.35
   Subnetzmaske  . . . . . . . . . . : 255.255.255.0
   Standardgateway . . . . . . . . . : 192.168.178.1
```

Im Beispiel ist der Adapter für die Drahtlosnetzwerkverbindung nicht verbunden. Der Adapter für die LAN-Verbindung ist mit dem Router (hier als Standardgateway bezeichnet) 192.168.178.1 verbunden. Die eigene Adresse des Rechners im Netz ist 192.168.178.35. Nachrichten, die an diesen Rechner geschickt werden, müssen als Empfänger die IP-Adresse 192.168.178.35 enthalten.

Hinweis: Die Adressen beginnend mit 192.168... sind typisch für einfache Heimnetzwerke, die über einen Router mit dem Internet verbunden werden.

Mit dem Befehl „tracert“ kann man den Ablauf einer Sendung von Daten (hier nach alp.dillingen.de) verfolgen.

```
C:\Users\Admin>tracert alp.dillingen.de

Routenverfolgung zu alp.dillingen.de [194.95.207.10] über maximal 30 Abschnitte:

  1     1 ms     1 ms     1 ms  fritz.fonwlan.box [192.168.178.1]
  2    45 ms    44 ms    45 ms  217.0.116.92
  3    44 ms    44 ms    45 ms  217.0.70.122
  4    54 ms    55 ms    54 ms  1-eb2-i.L.DE.NET.DTAG.DE [62.154.89.134]
  5    55 ms    60 ms    59 ms  80.156.160.142
  6    74 ms    97 ms    96 ms  xr-aug1-ge6-1.x-win.dfn.de [188.1.145.66]
  7    70 ms    68 ms    71 ms  kr-afldil2.x-win.dfn.de [188.1.232.50]
  8    70 ms    68 ms    69 ms  bndlg-fw01-transfer.bndlg.de [194.95.207.108]
  9    70 ms    66 ms    65 ms  alp.dillingen.de [194.95.207.10]
```

Die wichtigsten Stationen im obigen Beispiel sind:

Der DSL-Router im Heimnetzwerk (192.168.178.1), der mit dem Verbindungsrechner des Providers (217.0.116.92) verbunden ist, die Verbindung über das DFN (deutsches Forschungsnetz nach Augsburg und Dillingen), zum dortigen Provider (Bürgernetz Dillingen) und von dort in die Akademie.

Hinweise:

- Zur beispielhaften Visualisierung kann das in 2.1.1 skizzierte Netz verwendet werden, dort sind die gleichen IP-Adressen eingetragen.
- Unter Linux können die Befehle „ifconfig“ und „traceroute“ verwendet werden.

2.1.3 Möglicher Unterrichtablauf

Topologie von Rechnernetzen, IP-Adresse, Internet

Ausgehend vom Aufruf einer Webseite wird die Frage untersucht, auf welchem Weg die Information zum Rechner des „Internetnutzers“ gelangt. Dass dazu eine Anforderung vom Clientrechner an den Webserver gesendet werden muss, wird zunächst nicht weiter vertieft. Interessant ist, von wo und über welchen Weg die Daten kommen, die der Browser zur Darstellung einer Webseite benötigt. Im einfachsten Fall handelt es sich um eine Webseitendatei, die auf dem

Webserver eines Providers liegt. Zur Erarbeitung der verschiedenen Stationen auf dem Weg der Webseitendatei wird der Datenweg quasi „rückwärts“ vom Clientrechner aus betrachtet.

Im Unterrichtsgespräch werden mögliche Stationen identifiziert. Befindet sich z. B. der Clientrechner im Computerraum der Schule, so geht der Weg zumindest über den Router der Schule zum Internet Service Provider der Schule und von dort weiter zum Provider, der den Webserver betreibt.

Zur vorläufigen Bestätigung kann nach der einführenden Diskussion über den Befehl „tracert“ (alternativ „traceroute“ unter Linux) der tatsächliche Weg gezeigt und anhand der dort sichtbaren IP-Adressen die notwendige Adressierung (auch des eigenen Rechners) festgehalten werden. Auch die erfolgte Namensauflösung über eine (nicht sichtbare) Anfrage an einen DNS-Server wird hier thematisiert. Da das Konsolenfenster die IP-Adresse des eigenen Rechners nicht zeigt, wird diese anschließend ermittelt und der Befehl „ipconfig“ besprochen.

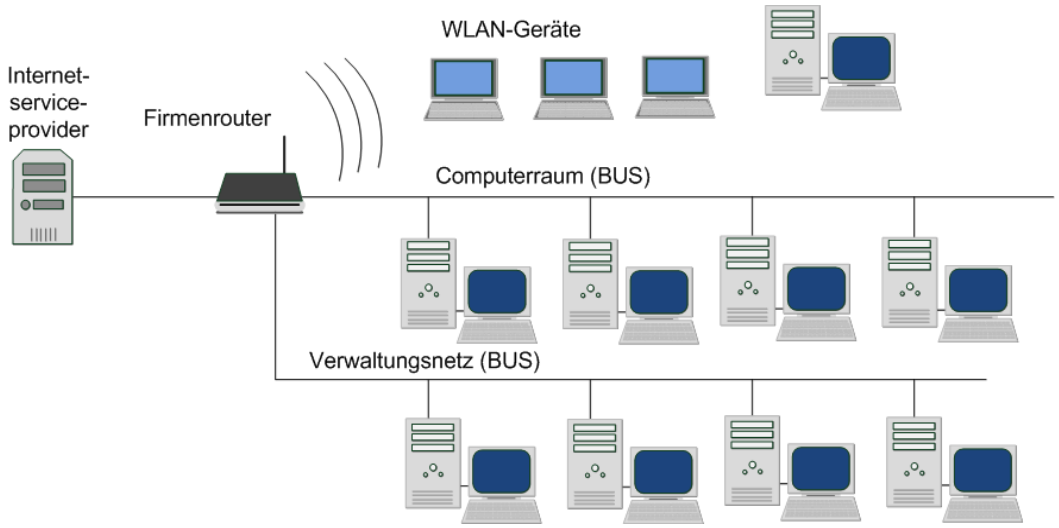
Die Frage nach Performanz (Kollisionsvermeidung und Übertragungsrate) und Ausfallsicherheit (Ausgangspunkt der Entwicklung des Internets war der „Kalte Krieg“) motiviert die Auseinandersetzung mit der netzartigen Struktur des Internets. Ausgehend von einer „naiven“ Vorstellung einer linearen Verbindung bis zum Webserver können Nachteile der dabei entstehenden Strukturen erkannt und diskutiert werden. Die dabei entstehende Skizze enthält bereits die Topologien „Stern“ (Anbindung an der Internet Service Provider) und „Bus“ (typisches Beispiel Computerraum). Die wenig häufige Ringtopologie muss möglicherweise extra motiviert werden.

Folgende Aufgaben können schließlich im Unterricht besprochen werden:

Aufgabe 1: Entwurf eines Firmennetzwerks

Als Systembetreuer einer Firma sollen Sie für den künftigen Neubau die Vernetzung der Firmenrechner planen. Aus Sicherheitsgründen sollen dort die Teilnetze der Verwaltung und die existierenden Computerräume nicht direkt verbunden sein. Zusätzlich gibt es in den Gängen der Firma Infoterminals, die per WLAN mit dem Firmenrouter verbunden sind. Skizzieren Sie ein mögliches Firmennetzwerk.

Lösungsvorschlag zu Aufgabe 1:



Aufgabe 2: Traditionelle Kommunikationswege

In einer Firma gibt es üblicherweise zwei Wege zur Information der Mitarbeiter, die alle ein persönliches Postfach besitzen.

Erster Weg: Beim sogenannten Rundlauf eines Dokumentes wird eine Mitteilung mit einem Deckblatt versehen, auf dem in einer Liste alle Adressaten verzeichnet sind, und in das Postfach des ersten Adressaten gegeben. Dieser bestätigt durch Unterschrift seine Kenntnisnahme und gibt die Mitteilung in das Postfach des nächsten Adressaten.

Zweiter Weg: Den betroffenen Mitarbeitern wird jeweils eine Kopie in ihr Postfach gelegt.

a) Diskutieren Sie Vor- und Nachteile beider Vorgehensweisen.

b) Vergleichen Sie die Kommunikation von Rechnern, die über einen gemeinsamen Bus verbunden sind, mit dem Rundlauf und zeigen Sie Analogien und Unterschiede auf.

Lösungsvorschlag:

zu a)

Rundlauf	Kopien
– geringer Materialaufwand – Weiterleitung ist mit Aufwand für den Mitarbeiter verbunden – mögliche Unterbrechung, wenn ein Mitarbeiter nicht weiterleitet – höherer Aufwand beim Erstellen der Adressatenliste – höherer Zeitbedarf, bis alle Mitarbeiter informiert sind. – Bestätigung der Kenntnisnahme liegt an Ende vor.	– hoher Materialaufwand – die Information wird sicher und quasi gleichzeitig zugestellt, aber es erfolgt keine Kontrolle der Kenntnisnahme – schnelle Information der Mitarbeiter, falls diese regelmäßig ihr Postfach kontrollieren.

zu b)

– Auch bei der Bustopologie führt (wie beim Rundlauf) eine Unterbrechung im Übertragungsmedium zum Verlust der Information. Die Rechner hinter der Unterbrechungsstelle sind nicht mehr erreichbar.

– Die Nachricht enthält normalerweise nur einen Empfänger, kann aber von allen Rechnern gelesen werden. Nachrichten an mehrere Rechnern müssen mehrfach verschickt werden.

  Vertiefender Hinweis: Spezielle Broadcastingverfahren machen es möglich, dass die Nachricht an alle angeschlossenen Rechner adressiert werden kann, ohne die Nachricht in Kopie mehrfach zu verschicken. Dies wird jedoch hier nicht weiter problematisiert.

– Damit der sendende Rechner überprüfen kann, ob die Nachricht korrekt zugestellt wurde, bekommt er vom Empfänger eine Bestätigung.

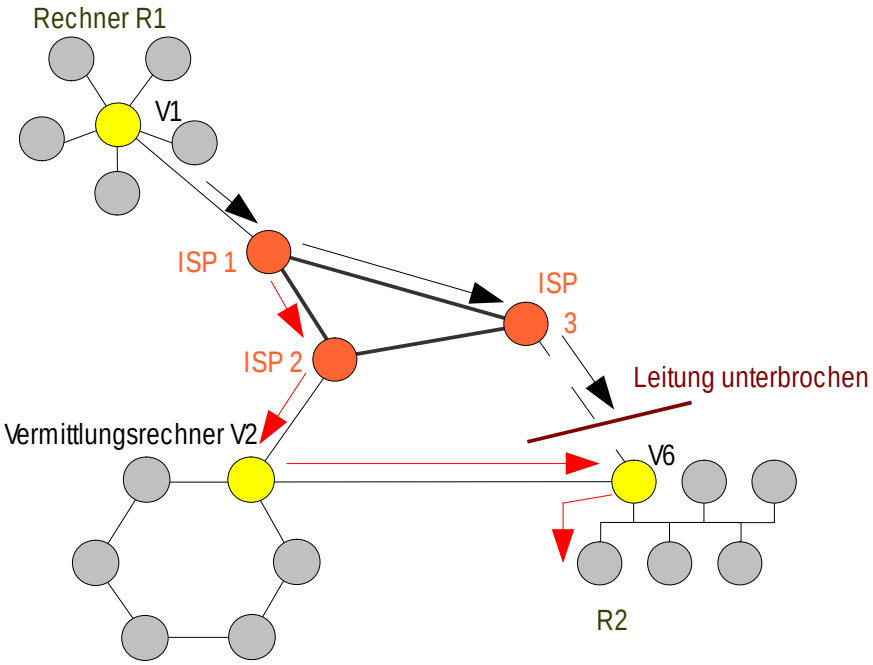
– Mehrere Rundläufe können gleichzeitig stattfinden, die Größe eines Postfachs ist jedoch genauso beschränkt wie die Bandbreite beim Bus.

Aufgabe 3: Verbindungsausfall

Erläutern Sie anhand einer geeigneten Skizze eines Ausschnitts aus dem Internet, wie Vermittlungsrechner trotz des Ausfalls einer Leitung die Verbindung zweier Rechner ermöglichen.

Lösungvorschlag:

In der nachfolgenden Abbildung kann eine Nachricht von Rechner R1 über den Internetserviceprovider 1 (ISP 1) über den Internetserviceprovider ISP 2 und die Vermittlungsrechner V2 und V6 zum Rechner R2 gelangen, falls die Verbindung von Internetserviceprovider ISP 3 zum V6 unterbrochen ist.



2.2 Kommunikation zwischen Prozessen

Lp: Aus vielen Bereichen der Computernutzung wie beispielsweise dem Homebanking oder einem Buchungssystem für Reisen ist den Schülern die Notwendigkeit zur Zusammenarbeit mehrerer Computer bekannt. Sie sehen ein, dass zur korrekten Verständigung von Computern spezielle formale Regeln (Protokolle) existieren müssen. Anhand der Kommunikation in Rechnernetzen erfahren die Jugendlichen, dass es sinnvoll ist, Kommunikationsvorgänge in verschiedene, aufeinander aufbauende Schichten aufzuteilen.

Als grundlegendes Beispiel wird ein Chatserver betrachtet. Ausgehend von der Anwendungserfahrung einer Chatsitzung werden die für die Kommunikation zwischen Prozessen notwendigen Bedingungen studiert.

Die Aufgaben im Verlauf des Kapitels können dabei schrittweise zur Implementierung von Chatserver und Chatclient-Programm führen. Eine vollständige Implementierung kann jedoch nur für besonders leistungsstarke Klassen bzw. zur Binnendifferenzierung empfohlen werden. Die Verwendung der beiliegenden Programme und deren Diskussion ist jedoch auch ohne Implementierung möglich.

Begriffsverwendungen

Prozess: Die in der Informatik gebräuchliche Definition eines Prozesses als eines laufenden Programms wird in diesem Kapitel etwas weiter gefasst. Ein Prozess wird allgemein als Abfolge verschiedener Aktivitäten eines Objekts (Person, Webserver usw.) verstanden.

Server und Client: Die in diesem Kapitel verwendeten Begriffe „Server“ und „Client“ meinen die Programme, die auf den jeweiligen Rechnern (oder auch auf ein und demselben Rechner) laufen.

Beteiligte Personen werden mit „User“ bezeichnet. Die Computer, auf denen die Programme laufen, werden mit „Server-Rechner“ und „Client-Rechner“ bezeichnet.

2.2.1 Dienste, Ports und Protokolle

2.2.1.1 Inhaltliche Grundlagen

In diesem Kapitel wird die Kommunikation von beteiligten Prozessen bei den verschiedenen Diensten des Internets untersucht. Die für die einzelnen Dienste typisch genutzten Ports und Protokolle stehen dabei im Vordergrund.

Dienste des Internets, Ports

Die weltweite Vernetzung von vielen Computern ist die Grundlage für viele verschiedene Dienste, wie z. B. E-Mail, Dateiübertragung über FTP (File Transfer Protokoll), Telefonieren über VOIP (Voice over IP). Auch das World Wide Web, kurz WWW, ist ein Dienst des Internets, obwohl man umgangssprachlich diesen Dienst oft als „das Internet“ bezeichnet. Dieser Internetdienst erlaubt die Kommunikation eines Clientrechners unter Verwendung eines Browsers mit einem Webserver, der Webseiten ausliefert.

Um zwischen Rechnern gleichzeitig mehrere Verbindungen aufbauen zu können, müssen die einzelnen Verbindungen vom Kommunikationspartner identifiziert werden können. Dies geschieht über sogenannte Ports. Dabei handelt es sich um Nummern, die zusätzlich zur Adresse angegeben werden. Die Portnummer kann mit der Angabe einer Zimmernummer oder einer Abteilung einer Firma zusätzlich zur Briefadresse verglichen werden. So findet beispielsweise durch Angabe „Abteilung Kundenbetreuung Zi. 08.15“ der Brief den Weg zum richtigen Sachbearbeiter und wird dem Kundenbetreuungsdienst der Firma zugeordnet.

Für die verschiedenen Dienste des Internets werden üblicherweise festgelegte Ports verwendet. Typische Beispiele zeigt folgende Tabelle:

Port	Dienst
20,21	Dateitransfer (Datentransfer vom Server zum Client und umgekehrt)
25	E-Mail-Versand
80	Webserver
110	E-Mail-Abruf
443	Webserver mit verschlüsseltem Zugang
3306	Zugriff auf MySQL-Datenbanken

Protokolle

Zur Kommunikation sind zusätzlich zur Verbindung der Rechner via Internet Protokolle nötig, die das Datenformat und den Ablauf einer Kommunikation zwischen zwei Rechnern festlegen. Eine gängige Definition ist beispielsweise:

> In der Informatik und in der Telekommunikation ist ein **Protokoll** eine Vereinbarung, nach der die Kommunikation zwischen zwei Parteien abläuft. Ein Protokoll ist eine formale Sprache, bei der die Syntax auch den Ablauf der Kommunikation definiert.

Hinweis: Eine formale Sprache umfasst sowohl die Syntax als auch die Semantik der Kommunikation (siehe Kapitel 1.2.7).

Die Beschreibung des Datenformats eines Protokolls kann mit der erweiterten Backus-Naur-Form (EBNF, siehe Kapitel 1.3.2), die Beschreibung eines beispielhaften Ablaufs mit einem Sequenzdiagramm erfolgen.

Hinweis: Im Zusammenhang mit dieser grundlegenden Einführung des Begriffs wird bewusst noch nicht auf das Schichtenmodell eingegangen. Die Thematik wird in Kapitel 2.2.4 ausführlich behandelt, dabei werden auch schichtenspezifische Protokolle diskutiert.

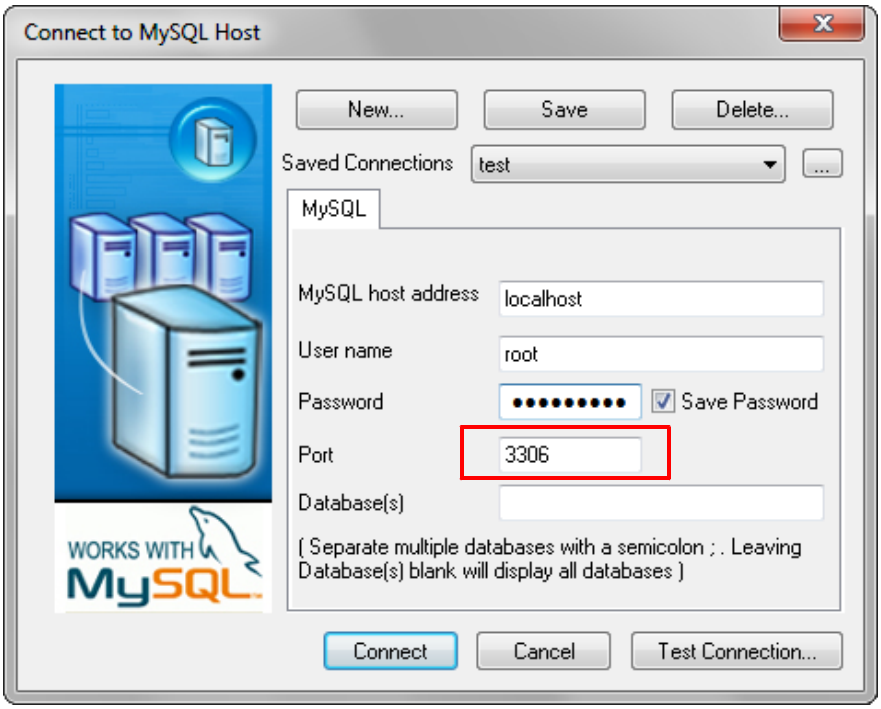
Der bereits angesprochene Vergleich mit der Kundenbetreuungsabteilung einer Firma kann an dieser Stelle erweitert werden:

Dienst	World-Wide-Web	Kundenbetreuung
Adressierung	IP-Adresse	Briefadresse
Port	80	Adresszusatz, z. B. Zimmernummer
Protokoll	HTTP	Die benötigten Daten werden durch ein Formular erfasst. Das Verfahren wird durch die Allgemeinen Geschäftsbedingungen der Firma geregelt (Nachbesserungsfristen, Garantiezeit), sodass der Dienst von jedem Sachbearbeiter nach bestimmten Regeln ausgeführt wird.

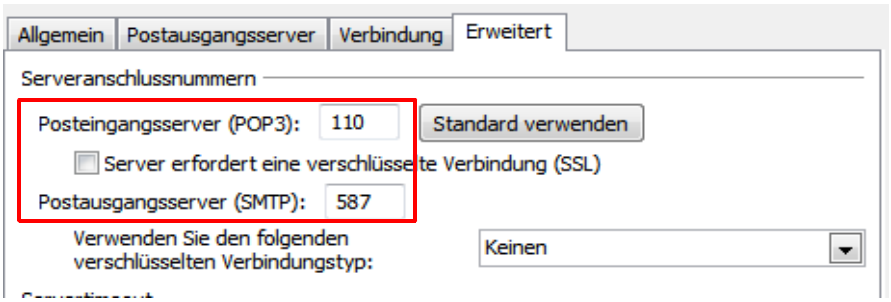
2.2.1.2 Umsetzungshinweise

Die Kenntnis der typisch verwendeten Ports der verschiedenen Dienste des Internets ist beim Konfigurieren der entsprechenden Programme hilfreich. Beispiele entsprechender Einstellungen könnten sein:

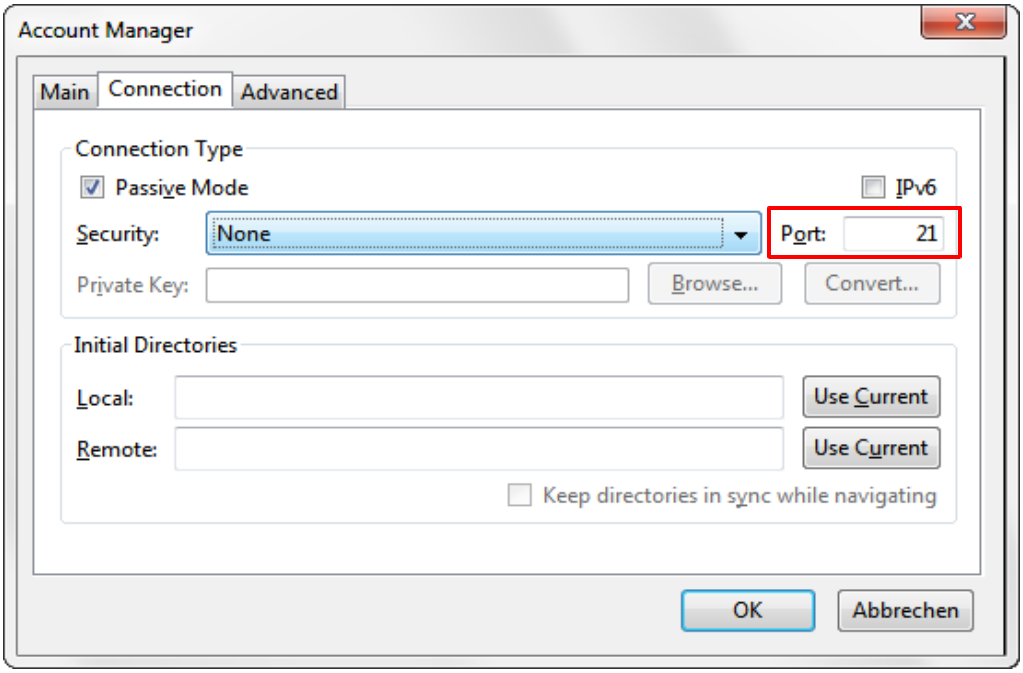
– Ein SQL-Client, wie er möglicherweise auch schon in der Jahrgangsstufe 9 beim Thema Datenbanken verwendet wurde:



– Ein E-Mail Programm:
Hinweis: In diesem Beispiel ist abweichend vom typischen Port 25 für die Verbindung zum Postausgangsserver der Port 587 eingetragen, da der Port 25 im Firmennetzwerk durch eine Firewall gesperrt ist.



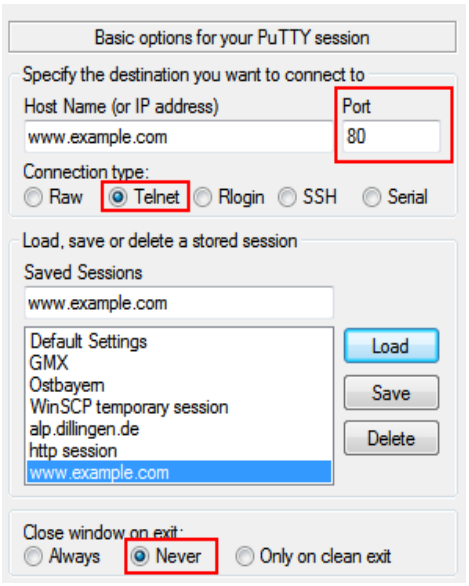
– Ein Programm zum Übertragen von Dateien mittels FTP:



Telnet-Client Putty als Werkzeug zur Untersuchung eines Webservers

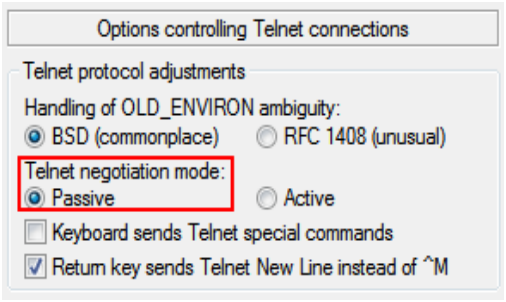
Zur Untersuchung des HTTP-Protokolls ist das Programm telnet.exe, das unter Windows standardmäßig zur Verfügung steht, nur bedingt geeignet. Bei Verwendung von telnet.exe sind die eigenen Eingaben per Voreinstellung im Konsolenfenster nicht sichtbar. Es wird deshalb empfohlen, als Telnet-Client das Programm „Putty“ zu verwenden, das kostenlos für Windows und Linux verfügbar ist (Download unter: http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html). „Putty“ kann zudem die Verbindungsdaten abspeichern, was das erneute Verbinden zum WWW-Server erleichtert.

Für den Webseitenaufruf von „www.example.com“ unter Verwendung von Putty sind folgende Schritte nötig:



Nachdem das Programm gestartet ist, kann man die Verbindungsdaten eingeben.

Hinweis: Je nach verwendeten Router kann es nötig sein, dass in der Abteilung „Telnet“ noch der **Passive Modus** eingeschalten werden muss, damit die Authentifizierung korrekt erfolgen kann.



Nach dem Öffnen der Verbindung fordert man eine Webseite mit dem GET-Befehl an:

HTTP Befehl — angeforderte Datei — verwendetes Protokoll

```
GET /index.html HTTP/1.1
Host: example.com
```

Anfrage

Antwort

```
HTTP/1.1 200 OK
Date: Fri, 26 Feb 2010 13:21:03 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT
ETag: "24ec5-1b6-4059a80bfd280"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

Header der zurückgelieferten Datei

```
<HTML>
<HEAD>
  <TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing &quot;example.com&quot;,
&quot;example.net&quot;,
  or &quot;example.org&quot; into your web browser.</p>
<p>These domain names are reserved for use in documentation and are not availabl
e
  for registration. See <a href="http://www.rfc-editor.org/rfc/rfc2606.txt">RFC
  2606</a>, Section 3.</p>
</BODY>
</HTML>
```

Inhalt der zurückgelieferten Datei

Die Antwort des Webservers besteht aus einem Header mit Informationen über den Sendezeitpunkt, Webserversoftware und über das nachfolgende Dokument (Länge, Codierung usw.) sowie dem eigentlichen Inhalt der Webseite.

Kommunikation mit einem Mailserver via Telnet

Zur Untersuchung des Protokolls bei E-Mail kann das Programm „putty“ oder das Programm „telnet“ verwendet werden.

1. Zur Herstellung einer Verbindung zum E-Mail-Server gibt man im Konsolenfenster (bei Windows cmd.exe aufrufen) den Befehl telnet <Adresse des Pop3-Servers> 110 ein.
2. Nun können verschiedene Anweisungen ausgeführt werden. Die folgende Tabelle zeigt einige Möglichkeiten:

POP3-Anweisung	Beschreibung
USER username	wählt den Benutzernamen „username“ bzw. das Benutzerkonto auf dem E-Mail-Server.
PASS passwort	übergibt das Passwort „passwort“ in Klartext.
QUIT	beendet die Verbindung zum POP3-Server . Alle gespeicherten Änderungen werden dann ausgeführt (Zum Beispiel der Befehl DELE).
LIST (n)	listet alle E-Mails auf. liefert die Anzahl und die Größe der (n-ten) E-Mail(s).
STAT	liefert den Status der Mailbox, u.a. die Anzahl aller E-Mails im Postfach und deren Gesamtgröße (in Byte).
RETR n	liefert die n-te E-Mail aus.
DELE n	löscht die n-te E-Mail.
NOOP	Dieser Befehl dient zur Aufrechterhaltung der Verbindung. Der Server würde sonst die Verbindung nach einem bestimmten Zeitintervall trennen.
RSET	setzt die aktive Verbindung zurück (setzt alle DELE Kommandos zurück).

Beispiel: Abrufen einer E-Mail

```
+OK GMX POP3 StreamProxy ready
USER max.muster@gmx.de
+OK May I have your password, please?
PASS 74jfzsj389
+OK Mailbox locked and ready
LIST
+OK
1 1745
2 1016496
3 19406
.
RETR 1
+OK
Return-Path:

       .....(technische Informationen zur E-mail.....)

From: "Andreas Wagner"
To: max.muster@gmx.de
Subject: test
Mime-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-Disposition: inline
X-GMX-Antivirus: 0 (no virus found)
X-GMX-Antispam: 0 (Mail was not recognized as spam);
 Detail=5D7Q89H36p6i75npGen84eVAEFK/syJmQWfUJhD4BPL42vD+2ZuqoFSikZQiMYiOsSe7p
 SYKeehYeY/pC/61vSJiYuLlK/gn7NmaJC9RKxsrIbx8cvvG23XQhj1AgqgYmh0p5SQrYOlhI1457
 m3/Gw==V1;
X-GMX-UID: d3KUeHxxPTRtAtoVMzIwIQYxc2tpZAvB

Hallo Andi

.
QUIT
+OK GMX POP3 server signing off
```

2.2.1.3 Möglicher Unterrichtsablauf

Dienste des Internets, Ports

Zunächst werden im Unterrichtsgespräch die verschiedenen Nutzungsmöglichkeiten des Internets gesammelt und anschließend den jeweiligen Diensten des Internets zugeordnet. Aus eigener Erfahrung wissen die Schülerinnen und Schüler, dass man z. B. gleichzeitig im World Wide Web surfen, E-Mails empfangen und eine VOIP-Schaltung nutzen kann. Damit dies möglich ist, haben die Datenpakete, die den Clientrechner erreichen, eine zusätzliche Information zur Unterscheidung des Kommunikationskanals – die Portnummer, die in den Einstellungen der jeweiligen Clientprogramme entsprechend konfiguriert werden kann. In der Grundeinstellung sind dort meist die typischen Portnummern (siehe Tabelle in 2.2.1.1) voreingestellt. Für einige ausgewählte Programme werden diese Einstellungen gesucht, notiert und den jeweiligen Diensten zugeordnet.

Protokolle

Am Beispiel eines Chatservers (🖫 p10_chatserver) wird die Kommunikation zwischen mehreren Rechnern demonstriert und die Notwendigkeit von Protokollen für diese Kommunikation erarbeitet.

Nach einer Chatsitzung mit der gesamten Klasse werden folgende Fragen thematisiert und damit die Begriffe IP-Adresse, Port, Server und Client vertieft:

- Welche Rechner sind beteiligt?
  Die Rechner der Chatteilnehmer und der Rechner, auf dem der Chatserver läuft.
- Wie unterscheiden sich die Rollen, die die Rechner innerhalb der Chatsitzung spielen?
  Aufgabe des Chatprogramms am sogenannten Clientrechner (kurz Chatclient) ist das Senden der vom Chatteilnehmer eingegebenen Botschaft zum Serverrechner und das Empfangen von Botschaften vom Serverrechner.

Das Chatprogramm am Serverrechner (kurz Chatserver) hält pro Chatclient je eine Verbindung, empfängt Botschaften von einem Chatclient und schickt diese an einen oder mehrere Clients weiter.

- Auf welchem Weg gelangt die Botschaft eines Chatteilnehmers zum Ziel?
  Es besteht keine direkte Verbindung zwischen den Clients. Botschaften werden über den Chatserver zum jeweiligen Adressaten gesandt.
- Welche Informationen sind dazu nötig?
  Im vorliegenden Beispiel ist vereinbart, dass eine Botschaft an alle Chatteilnehmer verteilt wird, in diesem Fall genügt der Text der Botschaft.
- Wieso müssen zu Beginn der Sitzung Verbindungsdaten eingegeben werden und wozu dienen diese Informationen?
  Diese Informationen dienen dazu, die Verbindung zum Chatserver aufzubauen. Die IP-Adresse legt den Rechner fest, der als Chatserver fungiert. Die Vereinbarung eines freien Ports ermöglicht eine Verbindung, auch wenn noch weitere Dienste des Internets am Server- oder Clientrechner genutzt werden.

Die unterschiedlichen Rollen (Client- und Serverrechner) werden bereits beim Programmstart deutlich. Um eine Verbindung zum Server aufzubauen, muss die IP-Adresse des Servers bekannt sein und man muss sich auf einen (freien) Port einigen, über den die Kommunikation stattfinden soll. Diese Informationen müssen nach Programmstart (p10_starteChatclient.bat) eingegeben werden (🖫 p10_chatserver).

Die Definition des Begriffs Protokoll wird durch die Bearbeitung der nachfolgenden Aufgaben erarbeitet.

Aufgabe 1: Beschreiben Sie den Ablauf einer Chatsitzung

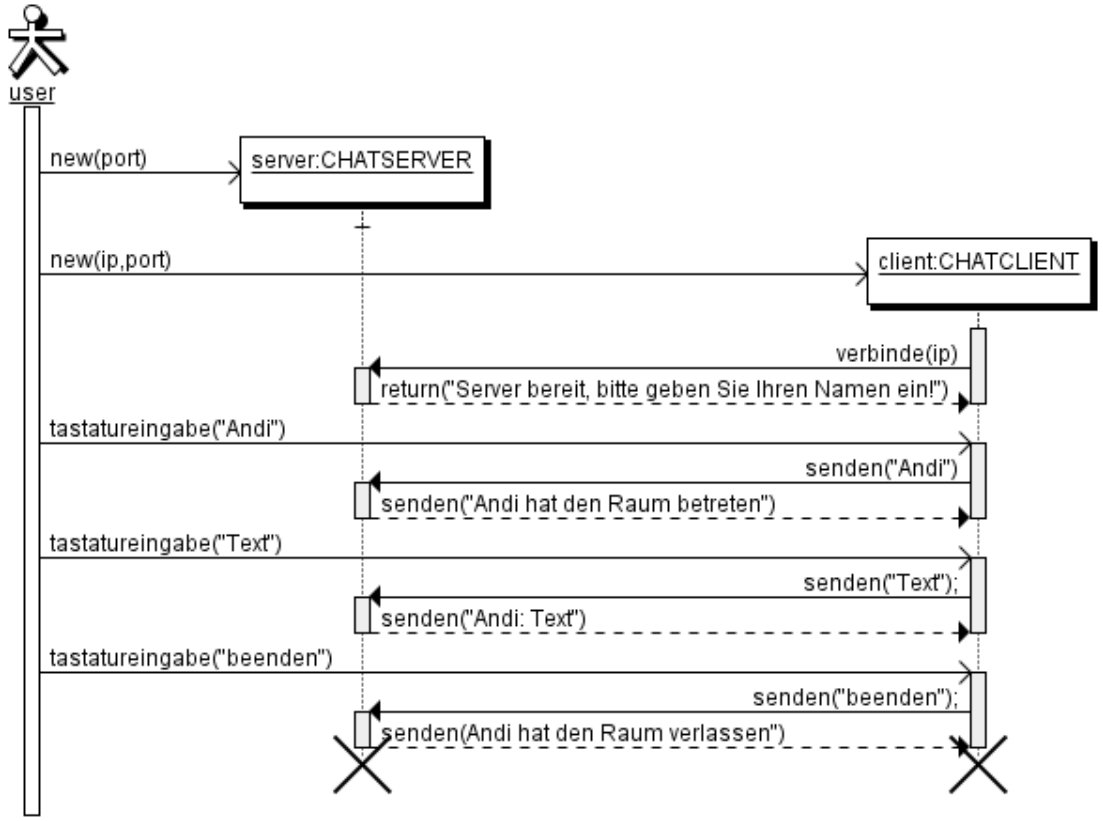
a) in Worten;

b) mithilfe eines Sequenzdiagramms.

Lösungsvorschläge zu Aufgabe 1:

a) Beschreibung des Ablaufs einer Chatsitzung
   - Starten des Clients (Eingabe von IP-Adresse des Servers und Portnummer)
   - Anmelden am Server mit Namen
   - Bestätigung des Servers
   - Senden einer Mitteilung, Empfangen von Mitteilungen

b) Darstellung einer möglichen Kommunikation mithilfe eines Sequenzdiagramms:



Aufgabe 2:

Beschreiben Sie die verwendeten Daten und deren Formate

a) in Worten

b) unter Verwendung der EBNF

für den vorliegenden Chatserver

Lösungsvorschlag zu b)

Anmeldung:

```
Ziffer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Portnummer = Ziffer [ Ziffer [ Ziffer [ Ziffer [ Ziffer ] ] ] ];
Zahl = Ziffer [ Ziffer [ Ziffer ] ];
IP-Adresse = Zahl '.' Zahl '.' Zahl '.' Zahl;
Zeilenende = '<CR>';
```

Mitteilung:

```
Zeichen = 'A' .. 'Z' | 'a' .. 'z';
Zeichenkette = Zeichen { Zeichen };
Clientname = Zeichenkette;
Botschaft = Zeichenkette;
Mitteilung = Botschaft Zeilenende;
Befehl = 'beenden';
```

Hinweis: Nach dieser EBNF wäre auch eine Adresse wie 456.999.999.111 gültig. Bei solchen Adressen kommt eine Verbindung nicht zustande. Eine Einschränkung auf gültige IP-Adressen würde eine umfangreichere EBNF erfordern.

Die Aufgaben verdeutlichen, dass zur Beschreibung der Kommunikation zwischen zwei Prozessen (insbesondere zweier Programmabläufe) die Beschreibung der verwendeten Datenformate und des

Kommunikationsablaufs gehört. In einem Protokoll werden entsprechende Regeln zur Kommunikation festgelegt. Ausgehend von einer Besprechung der Aufgaben und gegebenenfalls der Untersuchung eines Seitenaufrufs von „www.example.com“ oder der Lösung der Aufgaben zur Chatsitzung wird die Definition eines Protokolls formuliert.

Am Beispiel der Kommunikation mit einem Mailserver via Telnet werden die gefundene Definition und die Begriffe „Datenformat“ und „Kommunikationsablauf“ vertieft.

Aufgabe 3: Datenformate und Ablauf bei E-Mail

Beschreiben Sie die verwendeten Datenformate und den Ablauf beim Abrufen einer E-Mail.

Lösungsvorschlag:

Datenformate:

```
Zeichen = 'A' .. 'Z';
Zeichenkette = Zeichen { Zeichen };
Ziffer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Nummer = Ziffer { Ziffer };
Username = Zeichenkette;
Passwort = Zeichenkette;
Emailadresse = Zeichenkette '@' Zeichenkette '.' Zeichenkette;
Befehle = 'USER' Username | 'PASS' Passwort | 'QUIT' |
          'LIST' [Nummer] | 'STAT' | 'RETR' Nummer | 'DELE' Nummer |
          'NOOP' | 'RSET';
```

Ablauf:

User	Mailserver
Verbindungsdaten zum Mailserver in Clientprogramm eingeben	Mailserver bereit
Username eingeben	Username akzeptiert
Passwort eingeben	Passwort ist akzeptiert, Mailbox ist bereit
Auflisten der vorhandenen E-Mails mit „LIST“	gibt Liste der E-Mails aus
Die erste E-Mail in der Liste mit „RETR 1“ anfordern	Mailserver liefert E-Mail aus
Die Verbindung durch Eingabe von „QUIT“ beenden.	Bestätigung, dass Verbindung beendet wird und Verbindung beenden.

Mögliche weitere Aufgaben:

1. Welche Protokolle nutzen die Ihnen bekannten Dienste des Internets?

Lösungsvorschlag:

Dienst	Protokoll
Dateitransfer (Datentransfer vom Server zum Client und umgekehrt)	FTP
E-Mail-Versand	SMTP
Webserver	HTTP
E-Mail-Abruf	POP3
Webserver mit verschlüsseltem Zugang	HTTPS

2. Im Bereich der Diplomatie wird oft auch der Begriff Protokoll verwendet. Welche Unterschiede und Gemeinsamkeiten lassen sich im Vergleich zur Definition eines Protokoll in der Informatik finden?

Lösungsvorschlag:

Das Protokoll im Bereich der Diplomatie umfasst auch Vereinbarungen, die nicht unmittelbar der Kommunikation der beteiligten Parteien dienen. So sind z. B. über die Regeln der verbalen Kommunikation hinaus Kleiderordnung, Tischordnung und Rangordnungen genau festgelegt.

2.2.2 Umsetzung einer Client-Server-Kommunikation

In diesem Kapitel wird eine erste Client-Server-Kommunikation betrachtet. Dabei soll durch die Untersuchung (bzw. wahlweise auch Implementierung) von Protokollen bisher Gelerntes vertieft werden. Die Analyse bzw. der Test des ersten naiven Implementierungsversuchs eines Servers und dessen Verbesserung zeigt die Notwendigkeit einer Parallelisierung auf und motiviert so das nächste Kapitel. Die Diskussion kann unter Verwendung der beiliegenden Javaprogramme in Abhängigkeit vom Leistungsstand der Klasse auch völlig ohne Implementierungen durch Schülerinnen und Schüler erfolgen (Hinweise an jeweiliger Stelle).

2.2.2.1 Umsetzungshinweise

Die vorliegende JAVA-Implementierung (🖫 p01_wiegehts) der ersten Client-Server-Kommunikation verwendet die drei Klassen SERVER, CLIENT und SERVERVERHALTEN, um ein einfaches Gesprächsprotokoll zu ermöglichen.

Server- und Clientprozess im Pseudocode für das vorliegende Beispiel

Unabhängig von der verwendeten Programmiersprache gliedert sich eine erste naive Implementierung des Serverprozesses und des Clientprozesses im vorliegenden Beispiel in wenige Teile, die im Pseudocode formuliert werden können:

Serverprozess

Starten des Servers

Warten auf eine Clientverbindung und bei Verbindungsaufnahme Herstellen der Verbindung
tue
    Client-Botschaft lesen;
    Antwort ermitteln und senden;
solange Antwort nicht gleich „Auf Wiedersehen!“

Beenden der Clientverbindung
Stoppen des Servers

Clientprozess

Verbindung zum Server herstellen

wiederhole solange gelesene Server-Botschaft nicht gleich „Auf Wiedersehen“
    neue Botschaft von Tastatur lesen und an Server senden
Endewiederhole

Beenden der Verbindung zum Server

Zustandsdiagramm des Servers

Die Antwort des Servers auf die Clientbotschaft wird im Objekt der Klasse SERVERVERHALTEN ermittelt, die das Zustandsdiagramm (siehe Kapitel 2.2.2.2) der Kommunikation abbildet. Dadurch bleibt der Code übersichtlich und spätere Änderungen am Serververhalten können leicht als Zusatzaufgabe implementiert werden.

Verwendung der JAVA-Klassen ServerSocket und Socket

Um mit dem Server kommunizieren zu können, verwendet das Clientprogramm die Klassen BufferedReader zum Lesen des eingehenden Datenstroms und PrintWriter zur Ausgabe der Daten zum Server. Zur Erstellung dieser Objekte wird im Konstruktor als Parameter der Output- bzw. Inputstream eines Objekts der Klasse Socket benötigt, die in Java eine bidirektionale Schnittstelle für die Netzwerkkommunikation bereitstellt.

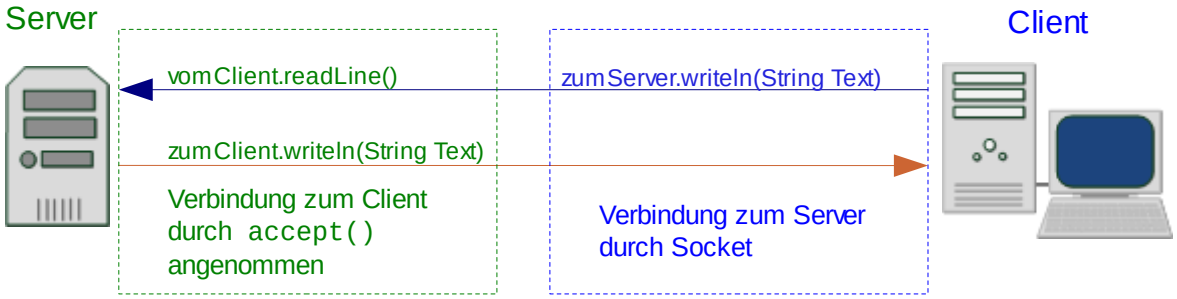
Verbindung zum Server im Clientprogramm:

```
//Verbindung über Port 4444 zur IP-Adresse „192.168.178.22“
clientSocket = new Socket(„192.168.178.22“, 4444);
//Objekte zum Lesen und Schreiben über die Verbindung erzeugen
zumServer =new PrintWriter(clientSocket.getOutputStream(), true);
vomServer =new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
```

Die Verbindung zum Server wird durch den Aufruf des Socketkonstruktors hergestellt, falls im Netz auf dem Rechner mit der angegebenen IP-Adresse ein Programm auf eine eingehende Verbindung über den angegebenen Port wartet und dieser Port noch frei ist. Im Serverprogramm wird dazu die Klasse ServerSocket verwendet. Die Methode *accept()* eines Objekts der Klasse ServerSocket wartet auf die eingehenden Verbindungen unter der beim Aufruf des Konstruktors festgelegten Portnummer. Nimmt der Client nun eine Verbindung auf, so liefert *accept()* den Clientsocket zurück, mit dessen Hilfe die jeweiligen Datenströme für In- und Output genutzt werden können.

Verbindung zum Client im Serverprogramm:

```
serverSocket = new ServerSocket(4444);//ServerSocket auf Port 4444 erzeugen
clientSocket = serverSocket.accept(); //auf die Verbindung warten
//Objekte zum Lesen und Schreiben über die Verbindung erzeugen
zumClient =new PrintWriter(clientSocket.getOutputStream(), true);
vomClient =new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
```



2.2.2.2 Praktische Umsetzung

Die Schülerinnen und Schüler untersuchen eine einfache Client-Server-Kommunikation. Dabei wird das zugrunde liegende Protokoll herausgearbeitet und anschließend untersucht, wie solche Protokolle implementiert werden können. Die strengen Regeln, nach denen das Gespräch zwischen Server und Client abläuft, werden zunächst in Form eines Zustandsdiagramms für den Server verdeutlicht und eine Serversitzung exemplarisch in einem Sequenzdiagramm festgehalten.

Beispielsweise in Zweierteams aufgeteilt erhalten die Schülerinnen und Schüler dazu folgende Aufgabe:

Aufgabe 1: Das Projekt „Hallo, wie geht's?“ (🖫 p01_wiegehts)

Für diese Aufgabe sind zwei verschiedene Rechner nötig.

a) Ermitteln Sie die IP-Adresse der beiden Rechner.

b) Starten Sie auf einem der beiden Rechner das Programm p01_starteServer.bat und wählen Sie einen Port, über den die Kommunikation stattfinden soll.

c) Starten Sie auf dem anderen Rechner das Programm p01_starteClient.bat und geben Sie die notwendigen Informationen zur Verbindung zum Server ein.

d) Erkunden Sie die Arbeitsweise des Servers und stellen diese in einem Zustandsdiagramm dar.

e) Zeichnen Sie ein beispielhaftes Sequenzdiagramm für eine Sitzung.

f) Welche Vereinbarungen wurden im vorliegenden Beispiel getroffen bzw. welche Regeln müssen, angefangen vom Start der Programme p01_starteServer.bat und p01_starteClient.bat, eingehalten werden, damit ein Gespräch geführt werden kann?

Nach der Untersuchung der Kommunikation bei „Hallo, wie geht's?“ wird der beiliegende Implementierungsvorschlag (🖫 p01_wiegehts) mit den Jugendlichen unter Berücksichtigung der in obiger Aufgabe gestellten Fragen besprochen. Dabei kann eine Formalisierung des Ablaufs in Pseudocode zum Codeverständnis beitragen. Die in Java vorhandenen Socketklassen und die beteiligten Streams werden erläutert. Die beiden Input-Streams *tastatur* und *vomServer* in der Klasse CLIENT müssen dabei genau unterschieden werden.

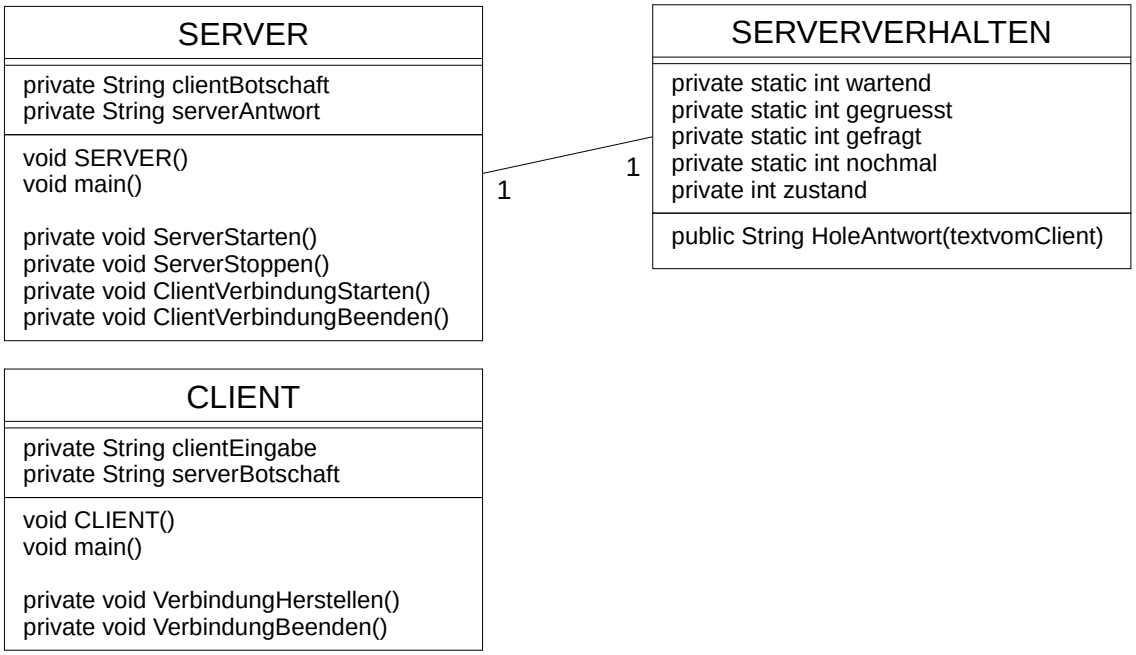
Besondere Beachtung verdient die Implementierung des Serververhaltens (Gesprächsregeln) in der Klasse SERVERVERHALTEN, die die modellierten Zustände des Servers repräsentiert. In Hinblick auf eine flexible Erweiterung des Serververhaltens wird diese Kapselung empfohlen. Die Klassen SERVER und CLIENT bleiben dadurch in den nächsten Programmbeispielen im Wesentlichen unverändert.

Hinweis: Die Gesprächsregeln sind Teil des Protokolls auf der Anwenderschicht. Dass Protokolle schichtenspezifisch sind, wird jedoch erst im Kapitel 2.2.3 diskutiert.

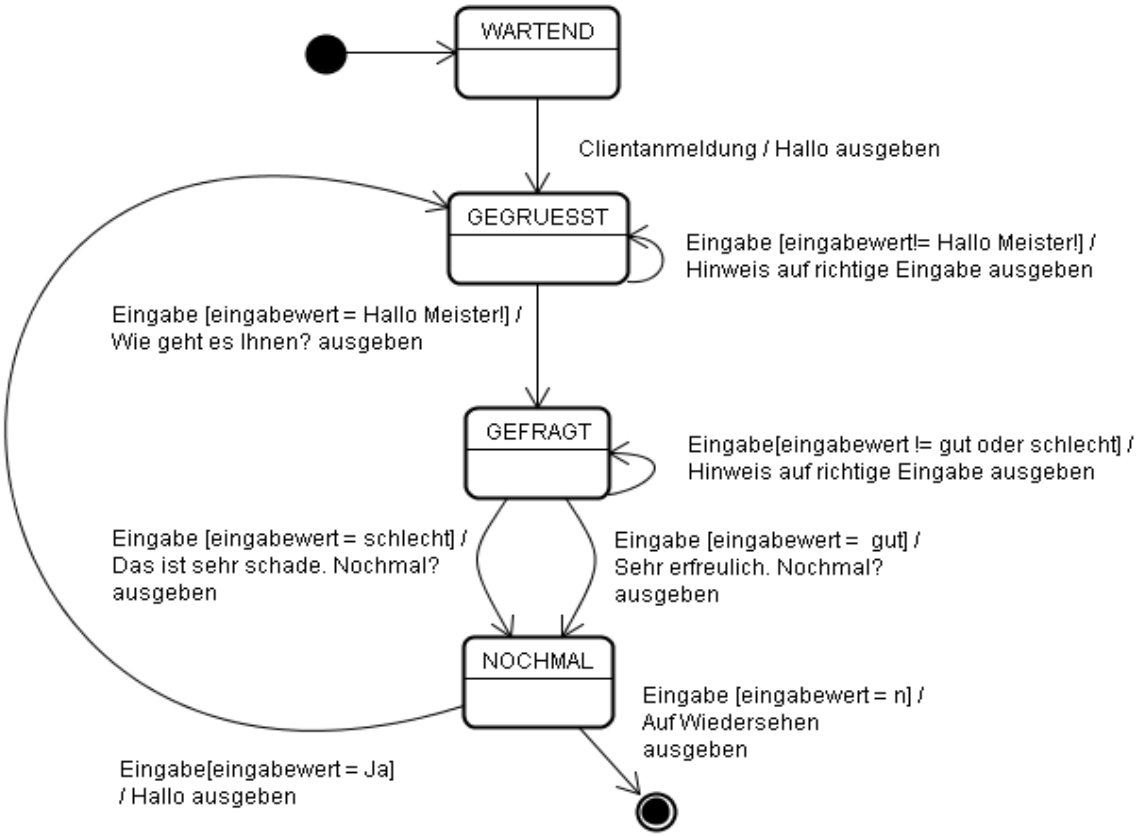
Lösungsvorschläge und Erläuterungen zu Aufgabe 1:

Im Projekt „Wie geht's“ (🖫 p01_wiegehts) wird eine einfache Kommunikation zwischen einem Server und *einem* Client implementiert. Die Implementierung der Gesprächsregeln als Teil des Protokolls erfolgt in der Klasse SERVERVERHALTEN.

Folgende Klassen werden verwendet:



Die Klasse SERVERVERHALTEN implementiert das folgende Zustandsdiagramm:



Das Protokoll umfasst im vorliegenden Beispiel mehr als die eigentliche Unterhaltung. Angefangen von der Verbindungsaufnahme (im vorliegenden Fall gehört bereits die Angabe von IP-Adresse und

Port-Nummer beim Start des Client-Programms bzw. des Serverprogramms dazu) über die geführte Unterhaltung bis hin zum Beenden des Programms sind Syntax, Semantik und Ablauf geregelt.

Datenformate

EBNF für das Beispiel „Wie geht's“

```
Ziffer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Portnummer = Ziffer [ Ziffer [ Ziffer [ Ziffer [ Ziffer ] ] ] ];
Zahl = Ziffer [ Ziffer [ Ziffer ] ];
IP-Adresse = Zahl '.' Zahl '.' Zahl '.' Zahl;
Zeilenende = '<CR>';
Mitteilung = Zeichenkette Zeilenende;
Servertext = 'Hallo!' | 'Wie geht es Ihnen?' | 'Sie muessen mich mit "Hallo
    Meister!" begruessen! Probieren Sie es nochmal!' | 'Sehr erfreulich. Nochmal
    (Ja)?' | 'Das ist sehr schade. Nochmal (Ja)?' | 'Sie koennen nur mit "gut"
    oder "schlecht" antworten. Wie geht es Ihnen?' | 'Auf Wiedersehen!';
Servermitteilung = Servertext Zeilenende;
```

Ablauf

Nach einer korrekten Anmeldung durch einen geregelten Start von Server und Client (auch die Reihenfolge ist vorgegeben und ist Teil des Protokolls!) kann das Gespräch zwischen User und Server stattfinden. Die Gesprächsregeln sind sehr genau festgelegt. Nachdem der Server vom Client mit „Hallo Meister!“ begrüßt wurde (auf andere Eingaben reagiert der Server mit einem Hinweis), stellt der Server die Frage „Wie geht es Ihnen?“ Die zugelassenen Antworten „gut“ oder „schlecht“ kommentiert der Server und fragt, ob die Kommunikation beendet werden soll. Auf alle andere Antworten reagiert der Server wiederum mit einem Hinweis.

Implementierung der Gesprächsregeln

Die jeweilige Antwort des Servers wird in der Methode *HoleAntwort()* des SERVERVERHALTEN-Objekts des Servers ermittelt. Die Klasse SERVERVERHALTEN implementiert dazu einen Automaten mit den Zuständen *wartend*, *gegruesst*, *gefragt* und *nochmal*.

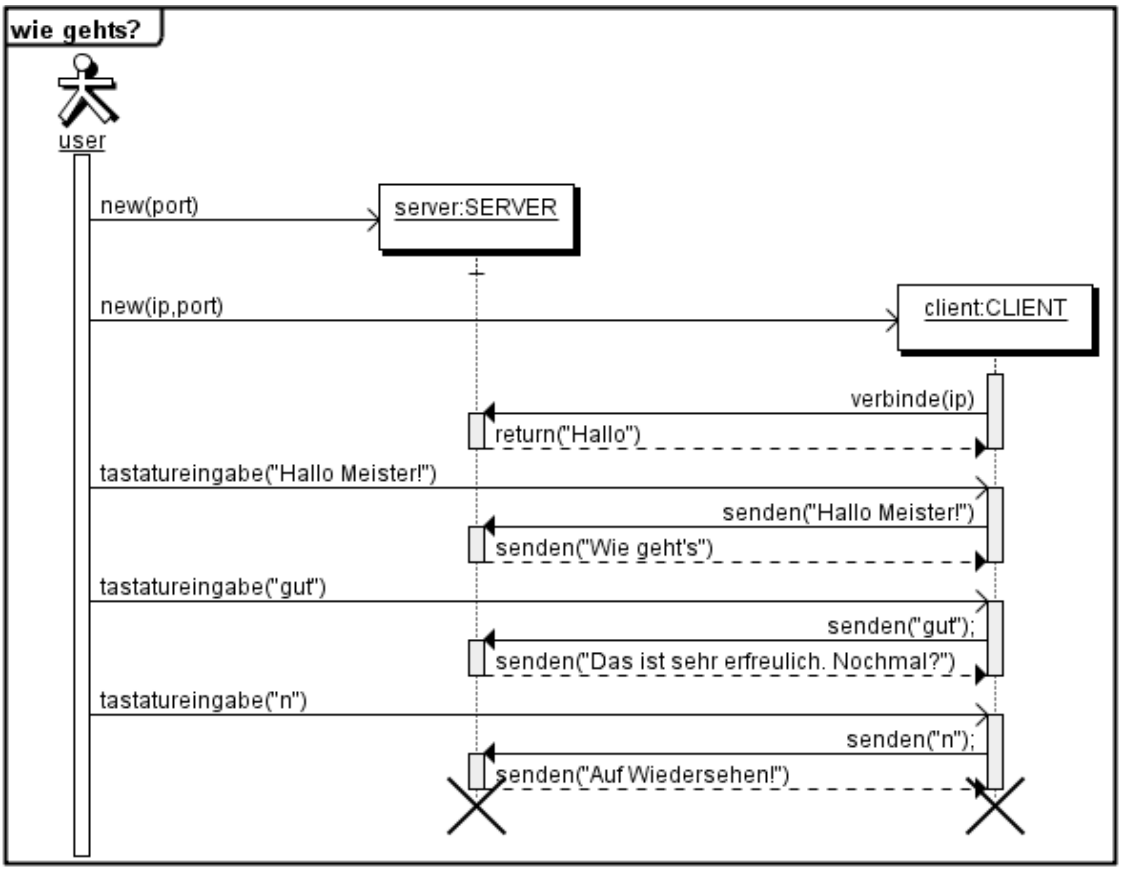
Zustandsdiagramm des Servers

Implementierung der entsprechenden Zustände in der Methode *HoleAntwort* der Klasse SERVERVERHALTEN:

```
public String HoleAntwort(String textvomclient) {
String ausgabe = null;

switch (zustand) {

    case wartend:
        ausgabe = "Hallo!";
        zustand = gegruesst;
        break;

    case gegruesst:
        if (textvomclient.equalsIgnoreCase("Hallo Meister!")) {
            ausgabe = "Wie geht es Ihnen?";
            zustand = gefragt;
        } else {
            ausgabe = "Sie muessen mich mit \"Hallo Meister!\" begruessen! "
                    + "Probieren Sie es nochmal!";
        }
        break;

    case gefragt:
        if (textvomclient.equalsIgnoreCase("gut")) {
```

```
            ausgabe = "Sehr erfreulich. Nochmal (Ja)?";
            zustand = nochmal;
        } else if (textvomclient.equalsIgnoreCase("schlecht")) {
            ausgabe = "Das ist sehr schade. Nochmal (Ja)?";
            zustand = nochmal;
        } else {
            ausgabe = "Sie koennen nur mit \"gut\" oder \"schlecht\" antworten. "
                    + "Wie geht es Ihnen?";
        }
        break;

    case nochmal:
        if (textvomclient.equalsIgnoreCase("Ja")) {
             ausgabe = "Hallo!";
             zustand = gegruesst;
        } else {
            ausgabe = "Auf Wiedersehen!";
            zustand = wartend;
        }
        break;
    }
return ausgabe;
}
```

Beispielhafter Ablauf einer Sitzung:



Nach der Besprechung verwenden die Schülerinnen und Schüler den vorhandenen Quellcode als Grundlage für einige kleine Änderungen. Es bietet sich an, die veränderte Implementierung der Gesprächsregeln im Zustandsdiagramm zu dokumentieren.

Je nach Leistungsstand der Klasse können die notwendigen Quelltextänderungen entweder im Unterrichtsgespräch ausgehend von einem veränderten Zustandsdiagramm diskutiert oder von den Schülerinnen und Schülern selbst implementiert werden.

Aufgabe 2: Veränderung des Protokolls

Verwenden Sie den Quellcode des Projekts p01_wiegehts, um folgende Veränderungen im Programm zu implementieren:

a) Das Verhalten des Servers soll so verändert werden, dass auf die Frage „Wie geht es Ihnen“ auch die Antwortoption „geht so“ zugelassen wird, auf die der Server mit „das hört sich nicht so toll an“ antwortet. Verändern Sie das Zustandsdiagramm entsprechend und passen Sie den Quellcode der Klasse SERVERVERHALTEN an.

b) Nach der Antwort auf die Frage „Wie geht es Ihnen“ soll die Frage „Heute schon gelacht?“ mit den Antwortoptionen „ja“ und „nein“ eingefügt werden. Verändern Sie die Implementierung der Klasse SERVERVERHALTEN und diskutieren Sie die notwendigen Veränderungen im Zustandsdiagramm.

Lösungshinweis: 🖫 p02_wiegehts2

Zur zusätzlichen Vertiefung eignet sich die Aufgabe der Implementierung einer Wetterauskunft. Auch hier bleiben die Implementierungen der Klassen SERVER und CLIENT im Wesentlichen noch unverändert. In der Klasse WETTERVERHALTEN werden jedoch zur Auffrischung der Java-Kenntnisse mehr Anforderungen gestellt. Sie muss komplett neu implementiert werden.

Aufgabe: Implementierung einer Wetterauskunft (🖫 p03_wetterauskunft**)**

Ein Wetterauskunftsterminal meldet sich mit „Bereit, bitte geben Sie Ihren Namen ein“, falls ein Client die Verbindung aufgenommen hat. Nach Eingabe des Namens *name* wird der Client mit „Hallo *name*“ begrüßt und nach einer Stadt gefragt. Für die Städte „Mailand“, „Madrid“, „Berlin“, „London“ liegen Wetterdaten vor (in der Simulation wird eine Wetterlage zufällig ausgewählt). Für alle anderen Städte antwortet der Server „Für diese Stadt liegen leider keine Wetterdaten vor!“ Anschließend fragt der Server, ob die Verbindung geschlossen werden soll oder die Wetterdaten einer anderen Stadt ermittelt werden sollen.

Entwerfen Sie ein Zustandsdiagramm für den Server und setzen Sie das Serververhalten nach obiger Beschreibung in der Klasse WETTERVERHALTEN um.

Lösungshinweis: 🖫 p03_wetterauskunft

2.2.3 Schichtenmodell

Lp: Anhand der Kommunikation in Rechnernetzen erfahren die Jugendlichen, dass es sinnvoll ist, Kommunikationsvorgänge in verschiedene, aufeinander aufbauende Schichten aufzuteilen.

2.2.3.1 Inhaltliche Grundlagen

Schichtenmodell

Mithilfe eines Schichtenmodells kann man komplexe und aufwendige Arbeitsabläufe strukturieren, die bei der Kommunikation zwischen Prozessen auftreten.

Die Idee eines Schichtenmodells besteht darin, dass jede Schicht ihre spezielle Aufgabe hat und die Schichten grundsätzlich voneinander unabhängig sind, dabei aber über definierte Schnittstellen miteinander kommunizieren.

Jede Schicht kommuniziert über ein **schichtenspezifisches Protokoll** mit der entsprechenden Schicht auf dem anderen System **(logischer Datenfluss)**, indem es Daten an die darunterliegende Schicht weiterleitet **(physikalischer Datenfluss)** bzw. von ihr erhält.

Ein einfaches grundlegendes Schichtenmodell für die Kommunikation kann folgendermaßen aussehen:

Schicht	Aufgabe
Anwendungsschicht	Nutzung des Anwendungsprogramms
Transportschicht	Zuverlässige Übertragung, Vollständigkeit der Daten
Vermittlungsschicht	Weitervermittlung von Paketen und die Wegewahl (Routing)
Netzzugangsschicht	Technik zur Datenübertragung

Die genaue Kenntnis der Schichtung des im Bereich der Netzwerkkommunikation häufig verwendeten ISO/OSI Modells oder des TCP/IP-Protokollstapels ist laut Lehrplan nicht gefordert. Die verwendeten Protokolle und die Aufgaben der einzelnen Schichten können aber durchaus im Rahmen von Beispielaufgaben besprochen werden.

Vertiefende Hinweise: In den Beispielimplementierungen in Java (z.B. p10_chatserver) umfasst das Protokoll neben den Regeln zur Verbindungsaufnahme (richtiger Start des Programms, Anmeldung) auch Festlegungen in Bezug auf die Datenübertragung. Die Java-Klassen ServerSocket und Socket implementieren das sogenannte TCP/IP-Protokoll und garantieren damit die vollständige und fehlerfreie Übertragung der Daten in der richtigen Reihenfolge. Im Gegensatz dazu ist z.B. beim UDP-Protokoll (User Datagram Protocol), das häufig zur Übertragung von Bild und Ton eingesetzt wird, die Übertragung der Datenpakete in der richtigen Reihenfolge nicht zugesichert. Kommt z.B. bei der Übertragung eines Videos ein Datenpaket, das zu einem früheren Zwischenbild gehört, zu spät an, so wird es einfach verworfen, da es bei einer solchen Übertragung wichtiger ist, den zeitlichen Ablauf eines Films korrekt wiederzugeben, als dass etwa einige Einzelbilder fehlen.

2.2.3.2 Möglicher Unterrichtsablauf

Ausgehend von folgender Einstiegsaufgabe wird ein erstes Modell einer Kommunikation in Schichten entwickelt:

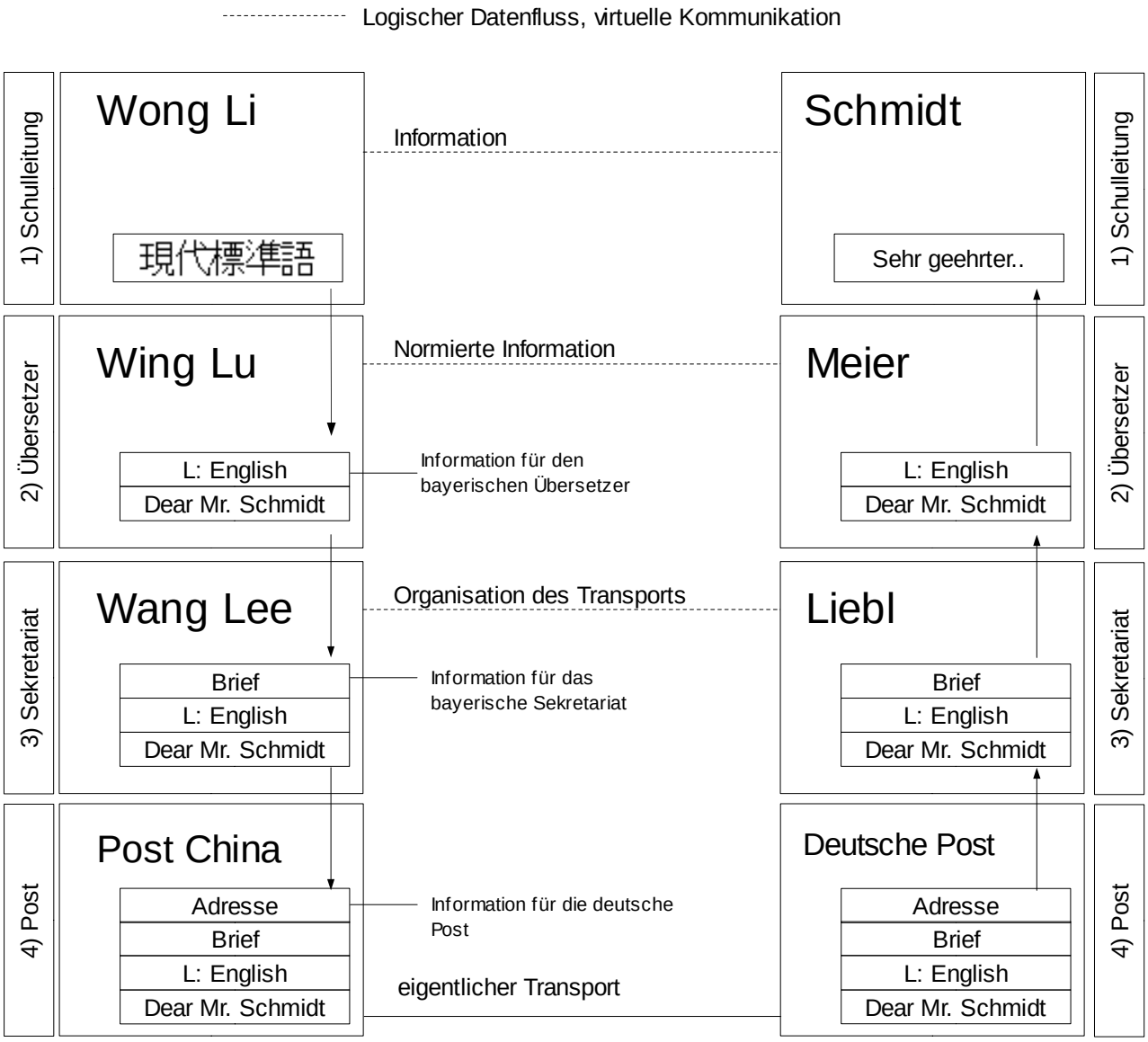
Aufgabe 1: E-Mail-Kommunikation

Im Rahmen eines Schüleraustauschprojekts will die Schulleiterin Wong Li einer japanischen Schule mit dem Schulleiter Herrn Schmidt eines bayerischen Gymnasiums kommunizieren. Glücklicherweise verfügen beide Schulleiter über Kollegen, die die jeweilige Landessprache ins Englische übersetzen können. Deren Übersetzungen werden über das Sekretariat per Post an die jeweilige Schule versandt.

a) Stellen Sie in einem Diagramm die jeweiligen Kommunikationspartner gegenüber und beschreiben Sie die Regeln, nach denen die Partner kommunizieren.

b) Über welche Stationen gelangt eine Nachricht von Frau Wong an Herrn Schmidt? Welche Fähigkeiten müssen dabei die beteiligten Personen mitbringen, welche Informationen geben sie an den jeweiligen Adressaten weiter?

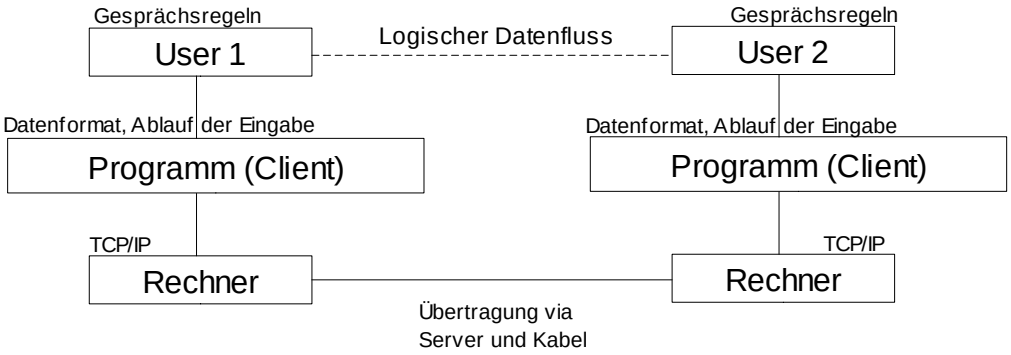
Die Diskussion über die beiden Teilaufgaben könnte zu folgendem Modell führen:



Es lassen sich in diesem Beispiel vier Schichten erkennen. Jede Schicht erledigt dabei eine ihr fest zugewiesene Aufgabe und stellt entsprechend festgelegte Schnittstellen zu den Schichten über und unter ihr bereit. Vom eigentlichen Weg der Daten braucht die Schicht dabei nichts zu wissen. Eine Schicht dieses sogenannten Protokollstapels kommuniziert in einem **schichtspezifischen Protokoll** mit der entsprechenden Schicht auf dem anderen System **(logischer Datenfluss),** indem es Daten an die darunter liegende Schicht weiterleitet **(physikalischer Datenfluss)** bzw. von ihr erhält.

Nach der Besprechung der Aufgabenlösung kann das Schichtenmodell allgemein definiert werden.

Zur Vertiefung wird die Chatkommunikation unter dem Aspekt des Schichtenmodells untersucht.



Im Unterrichtsgespräch werden die beteiligten Schichten erarbeitet und ihre jeweiligen Aufgaben identifiziert. Der Aspekt der schichtenspezifischen Protokolle wird durch die Gegenüberstellung von „Gesprächsregeln“ als Teil des Protokolls der Anwenderschicht und des TCP/IP-Protokolls, mit dem die Rechner miteinander kommunizieren, deutlich. Im vorliegenden Beispiel beinhaltet das Protokoll auf oberster Ebene die Gesprächsregeln (Netiquette), die beim Chatten zu beachten sind. Auf Ebene des Programms (ebenfalls Anwendungsschicht) sind Datenformate und Eingabeablauf geregelt. Auf der untersten Schicht sorgt ein entsprechendes Protokoll für die zuverlässige und fehlerfreie Übertragung der Daten (Transport-, Vermittlungs- und Netzzugangsschicht).

Hinweis: Für die Erläuterung des TCP/IP-Protokolls reicht es aus, die Aufgaben zu erarbeiten, die für eine fehlerfreie Übertragung nötig sind (Transport, Vermittlung, Übertragung). In der Aufgabe zu TCP/IP-Prokollstapel kann dieses Wissen noch vertieft werden. Auf die Einzelheiten des Protokolls braucht nicht eingegangen zu werden.

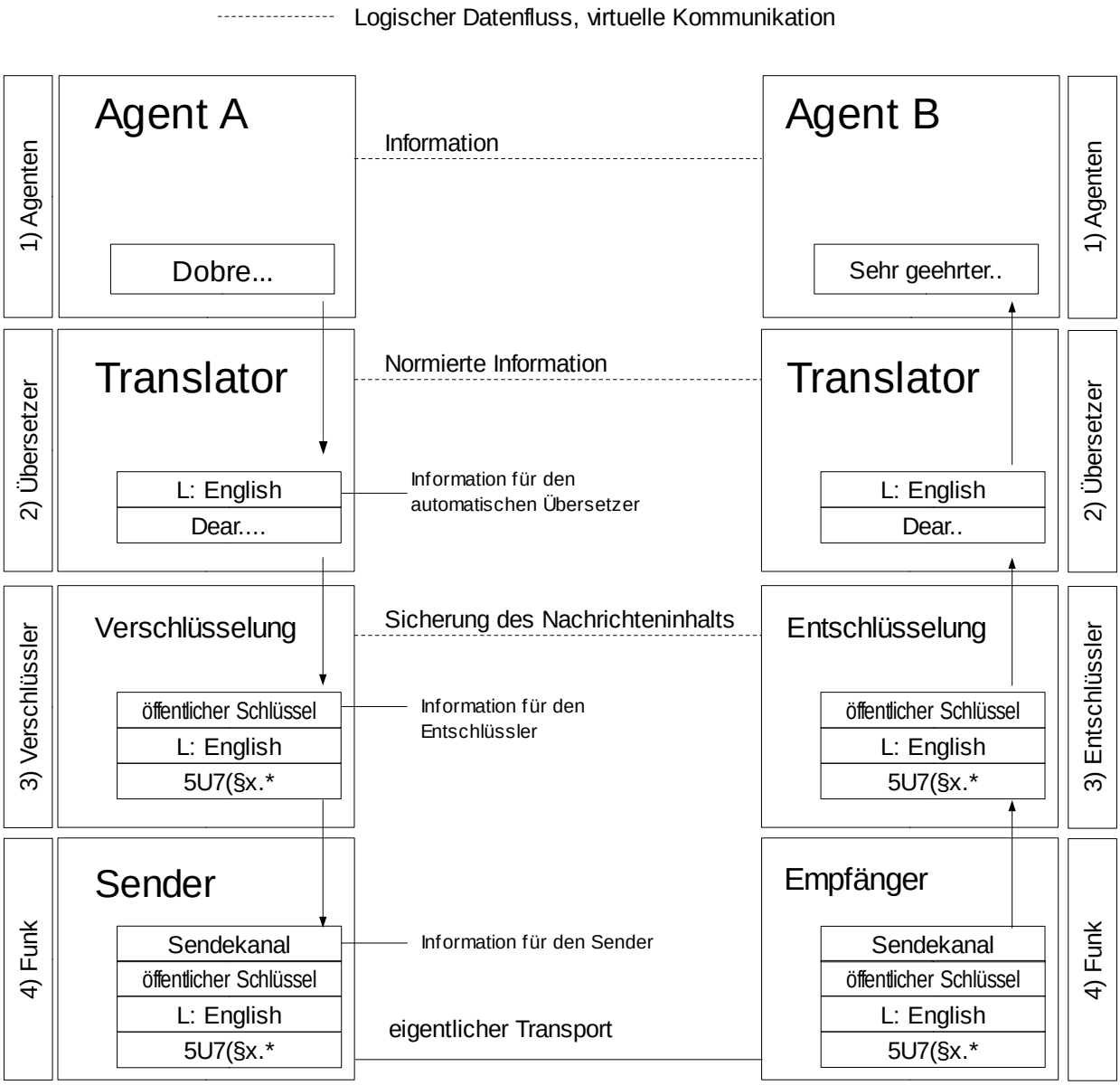
Mögliche weitere Aufgaben:

Aufgabe: Verschlüsselte Kommunikation

Zwei Geheimagenten verschiedener Länder kommunizieren über abhörsichere Kommunikationsgeräte. Die eingegebene Nachricht wird automatisch ins Englische übersetzt und verschlüsselt übertragen.

Geben Sie ein Schichtenmodell mit (mindestens) 4 Schichten für diesen Kommunikationsablauf an und erklären Sie anhand dieses Modells beispielhaft den Ablauf der Kommunikation zwischen den Partnern.

Lösungsvorschlag:



Aufgabe: TCP/IP-Protokollstapel

Ein bewährtes Schichtenmodell zur Beschreibung von Kommunikation innerhalb von Rechnernetzen ist der sogenannte TCP/IP-Protokollstapel. Recherchieren Sie, welche Schichten zur Beschreibung der Kommunikation verwendet werden. Welche Aufgaben haben dabei die einzelnen Schichten? Nennen Sie Protokolle, die auf den jeweiligen Schichten verwendet werden.

Lösungsvorschlag:

Schicht	Aufgabe	Protokolle
Anwendungsschicht	Protokolle, die mit Anwendungsprogrammen zusammenarbeiten	HTTP, FTP, SMTP, POP, Telnet
Transportschicht	Zuverlässige Übertragung, Vollständigkeit der Daten	TCP

Vermittlungsschicht	Weitervermittlung von Paketen und die Wegewahl (Routing)	IP (Ipv4, IPv6)
Netzzugangsschicht	Technik zur Datenübertragung	Ethernet, Token Bus, Token Ring, FDDI

2.3 Modellierung einfacher paralleler Prozesse

Lp: Die Schüler erkennen z. B. bei der Analyse der Aufgaben eines Mailservers, dass Rechner anfallende Aufträge oft parallel bearbeiten und dennoch einen geordneten Ablauf gewährleisten müssen.

2.3.1 Inhaltliche Grundlagen

Im folgenden wird unter „Prozess“ ein ablaufender Teil eines Programms verstanden. Laufen mehrere Programme oder Teilprogramme gleichzeitig, so spricht man von **parallelen Prozessen**.

Um Wartezeiten des Prozessors zu nutzen, können bei allen Betriebssystemen mehrere Prozesse parallel ausgeführt werden. Man unterscheidet dabei zwischen der parallelen Ausführung von Programmen (schwergewichtige Prozesse) und von Programmteilen (leichtgewichtige Prozesse). Der Wechsel zwischen leichtgewichtigen Prozessen (sogenannten Threads) erfolgt sehr schnell. Der Zugriff auf gemeinsame Ressourcen ist möglich.

Hinweis: In Abhängigkeit des Prozessors und des jeweiligen Betriebssystems laufen Prozesse oft „quasiparallel“ ab. Die Nutzung des Prozessors durch die Programme erfolgt sequentiell, die Zuteilung von Prozessorzeit an die laufenden Prozesse erfolgt jedoch wechselweise unter Verwendung betriebssystemspezifischer Schedulingstrategien, so dass der Eindruck entsteht, Programme oder Programmteile würden parallel laufen. Im folgenden Kapitel werden solche Prozesse unabhängig von der Umsetzung im jeweiligen Betriebssystem kurz als „parallel“ bezeichnet.

Unter **nebenläufigen Prozessen** versteht man parallele Prozesse, die sich möglicherweise gegenseitig beeinflussen, weil sie auf gemeinsame Ressourcen zugreifen.

Hinweis: In der Literatur werden die Begriffe „nebenläufig“ und „parallel“ oft nicht klar unterschieden bzw. in unterschiedlicher Bedeutung verwendet.

2.3.2 Umsetzungshinweise

Mit der Klasse Thread kann in Java ein leichtgewichtiger paralleler Prozess innerhalb eines Programms leicht realisiert werden. Dazu muss die Methode *run()* einer eigenen von Thread abgeleiteten Klasse implementiert werden.

Beispiel:

```
class MeinThread extends Thread // damit lassen sich eigene
                                // Thread-Objekte erstellen
{
    // Nötige Attribute, Konstruktor, Hilfsmethoden etc.
    public void run ()   // Hier beginnt der Thread seine Arbeit
    {
        // Arbeitsauftrag des Threads
    }
}

//Erzeugen und aktivieren des Threads
Thread t = new MeinThread ();   // Das Objekt wird angelegt
t.start (); // Der Thread wird in die Verwaltung aufgenommen; sobald
            // er an der Reihe ist, wird mit Ausführung
            // der Methode run begonnen
```

2.3.3 Möglicher Unterrichtsablauf

In den bisherigen Versionen der Klasse SERVER kann sich immer nur ein Client mit dem Server verbinden, da am Server nur ein Prozess für den (exklusiven!) Client aktiv ist.

In diesem Kapitel wird die Implementierung der Client-Server-Kommunikation fortgesetzt. Der Server wird weiter verbessert, so dass auch mehrere Clients parallel bedient werden können. In der nachfolgenden Aufgabe werden sich die Schülerinnen und Schüler dieses Problems bewusst. Es kann auch nur Teilaufgabe 1a) und 1d) unter Verwendung des fertigen Programms (🖫 p04_mehrclients_loesung) bearbeitet werden, wenn man auf Programmieraufgaben verzichten möchte.

Beim Test des Programms „Wetterauskunft“ in der nachfolgenden Aufgabe erkennen die Schülerinnen und Schüler, dass eine parallele Bearbeitung von Anfragen zum Wetter notwendig ist, und machen sich in kleinen Implementierungsaufgaben mit dem Quelltext des Programms vertraut. Die Implementierungen bereiten zudem die Parallelisierung des Serverprozesses vor, die anschließend im Unterrichtsgespräch gemeinsam durchgeführt bzw. studiert werden kann.

Aufgabe 1: Mehrere Clients (🖫 p04_mehrclients_aufgabe)

Für diese Aufgabe sind drei verschiedene Rechner nötig. Arbeiten Sie in Gruppen.

a) Testen Sie den Wetterauskunftsserver (🖫 p03_wetterauskunft) unter Verwendung von drei Rechnern. Starten Sie dazu auf einem Rechner das Serverprogramm (p03_starteServer.bat) und verbinden Sie sich über einen weiteren Rechner mit der Wetterauskunft. Was passiert, wenn sich ein zweiter Client anmelden möchte? Erläutern Sie die Problematik mithilfe eines Sequenzdiagramms.

b) Die vorliegende Implementierung „Wetterauskunft“ soll so verändert werden, dass der Client durch die Eingabe von „beenden“ jederzeit beendet werden kann. Die Methode *HoleAntwort()* soll dazu unabhängig vom Zustand die Anfrage des Clients mit dem Stopp-Signal „Server [stopClient]:“ beantworten. Verwenden Sie dazu die Quellcodevorgabe (🖫 p04_mehrclients_aufgabe)

c) Ergänzen Sie den Quellcode der unvollständig implementierten Klasse SERVER2 (🖫 p04_mehrclients_aufgabe) an den gekennzeichneten Stellen und zeichnen Sie ein Klassendiagramm.

d) Testen Sie das Programmverhalten Ihrer Implementierung, wenn sich zunächst nacheinander zwei Clients (client1 und client2) anmelden und anschließend client1 beendet wird.

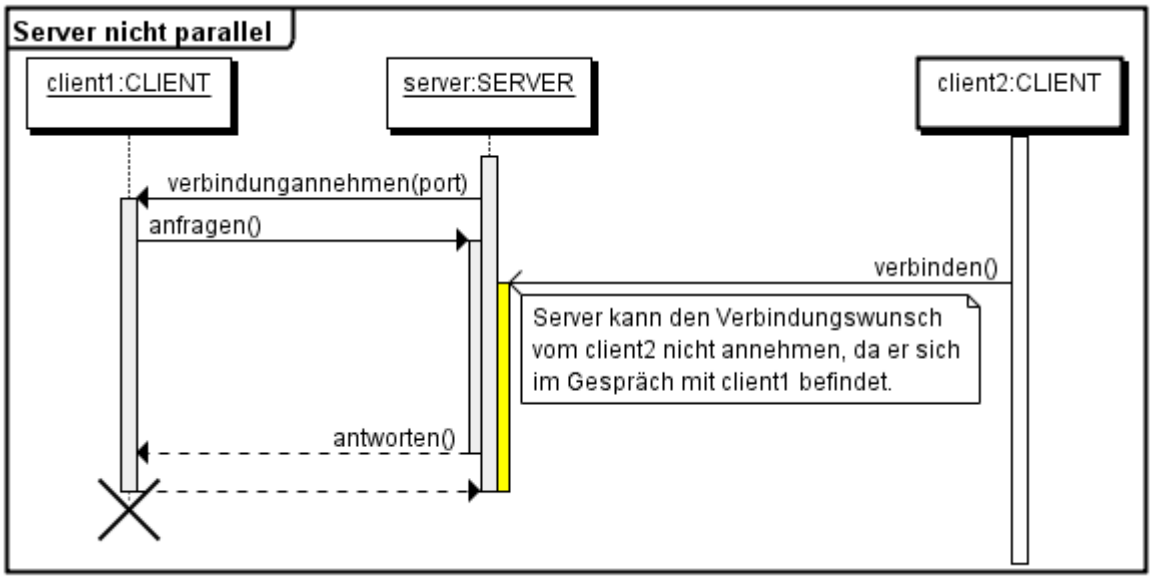
Lösungshinweise: (🖫 p04_mehrclients_loesung)

zu a) Ein zweiter Client kann sich beim Programm „Wetterauskunft“ nicht mit dem Server verbinden, weil der Server die Verbindungen nicht parallel bearbeiten kann. Da beim Beenden des verbundenen Clients auch der Server beendet wird, scheitert die Verbindung des zweiten Clients, auch wenn der verbundene Client die Verbindung freigibt.

zu d) Durch das implementierte Stoppsignal wird die Clientverbindung beendet, ohne dass das Serverprogramm beendet wird. Ein Clientprogramm, das auf eine Verbindung wartet, kann nun die Verbindung zum Server aufnehmen.

Hinweis: Bei allen nachfolgenden Implementierungen soll künftig die Serververbindung des Clients beendet werden, falls die empfangene Serverbotschaft mit der Zeichenfolge „Server [stopClient]“ beginnt.

Die Schülerinnen und Schüler erkennen schnell, dass es darauf ankommt, dass der Server nach einer Verbindungsannahme sofort wieder für eine weitere Verbindung zur Verfügung steht, da es sonst zu langen Wartezeiten kommt, weil die Clientanfragen nur nacheinander abgearbeitet werden können.



Die eigentliche Arbeit erledigt der Server innerhalb der Wiederholung im Konstruktor und ist währenddessen nicht für eine neue Verbindung bereit. Die zu wiederholende Sequenz muss also künftig ausgelagert sein. Alle Attribute und Methoden(-teile), die einem Client zugeordnet sind, werden dazu in einer Klasse CLIENTPROZESS gekapselt.

Als Aufgabe oder im Unterrichtsgespräch erfolgt der Umbau des Servercodes. Im Quelltext der Klasse SERVER (🖫 p04_mehrclients_loesung) sollen die Schülerinnen und Schüler zunächst die clientspezifischen Elemente (im Lösungsvorschlag gelb) markieren. Die Variable *zustand* und das WETTERVERHALTEN-Objekt darf dabei nicht vergessen werden. Das Warten auf einen neuen Client wird in die Methode *AufNeuenClientWarten()* übertragen. Bei diesen Überlegungen wird auch deutlich, warum die unterschiedlichen Aufgabenteile (Annehmen und Verwalten der Verbindungen, Arbeit mit den Verbindungen) auf zwei Klassen aufgeteilt werden sollten.

Der (teilweise) bearbeitete Quelltext der Klasse SERVER könnte so aussehen:

- Alle markierten Zeilen (gelb) werden in die Klasse CLIENTPROZESS übertragen (Sequenzanteile in die dortige Methode *run()*), die von der Klasse Thread erbt, damit die Methode *run()* nebenläufig ausgeführt werden kann.
- **Fett** markierte Teile müssen angepasst werden: das Stop-Signal für das Beenden des Clientprozesses lautet künftig **„Server[stopClient]“**.
- *Kursiv* markierte Teile werden nicht mehr benötigt

```
//Schnittstelle zur Netzwerkprotokoll-Implementierung...
ServerSocket serverSocket = null; //.. des Servers
Socket clientSocket = null;       //.. des Clients
PrintWriter zumClient = null;      //Datenstrom zum Client
BufferedReader vomClient = null;  //Datenstrom vom Client

WETTERVERHALTEN2 kkp; //Zustandsdiagramm des Clients
String clientBotschaft = null; //Botschaft von Client zum Server
String serverAntwort = null;   //Botschaft vom Server zum Client

public SERVER2() throws IOException {
```

```
        ServerStarten();
        ClientVerbindungStarten(); //auf Client warten und verbinden

        do {//lesen und antworten

            clientBotschaft = vomClient.readLine();
            serverAntwort = kkp.HoleAntwort(clientBotschaft);
            zumClient.println(serverAntwort);

            if (serverAntwort.startsWith("Server[stopClient]:")) {
                ClientVerbindungBeenden();  //Verbindung schließen
                ClientVerbindungStarten();  //auf neue Verbindung warten
            }

        } while (!serverAntwort.startsWith("Server[stopServer]:"));

        ClientVerbindungBeenden();
        ServerStoppen();
    }

    private void ClientVerbindungStarten() throws IOException {

        clientSocket = serverSocket.accept();  //Warten auf die Verbindung
        zumClient = new PrintWriter(clientSocket.getOutputStream(), true);
        vomClient = new BufferedReader(new InputStreamReader
                              (clientSocket.getInputStream()));
        //Protokoll-Klasse zur Ermittlung der Serverantworten
        kkp = new WETTERVERHALTEN2();
        //Begrüßung
        serverAntwort = kkp.HoleAntwort("");
        zumClient.println(serverAntwort);
        System.out.println("Client verbunden");
    }

    private void ClientVerbindungBeenden() throws IOException {..}
```

Der Programmteil der Klasse SERVER reduziert sich dabei deutlich; das SERVER-Objekt braucht in einer Methode *AufNeuenClientWarten()* nur noch auf einen neuen Client zu warten und für diesen einen neuen Thread zu starten.

Der Umbau des Servercodes kann im Unterrichtsgespräch diskutiert und auch durchgeführt werden. Da der Clientcode unverändert bleibt, können sich die Schülerinnen und Schüler sofort mit dem Server verbinden und die Implementierung testen (🖫 p05_mehrclientsparallel).

Hinweis: In der vorliegenden Implementierung wird der Einfachheit halber auf ein Server-Endesignal verzichtet. Für eine Implementierung wäre die Einführung einer Variablen nötig, dessen Wert in Abhängigkeit der vom Client gesendeten Nachricht durch den jeweiligen Clientprozess zugewiesen wird. Die Umsetzung eignet sich als Zusatzaufgabe im nächsten Kapitel, weil dadurch möglicherweise ein weiterer kritischer Abschnitt entsteht.

Anschließend kann der Kern der jetzigen Implementierung des Servers in Pseudocode formuliert und mit dem ersten naiven Ansatz aus Kapitel 2.2.2.1 verglichen werden.

erster Ansatz	Servercode nach Umbau
Starten des Servers Warten auf eine Clientverbindung und Herstellen der Verbindung wiederhole Client-Botschaft lesen; Antwort ermitteln und senden; bis Server-Endesignal gesetzt Clientverbindung beenden Server stoppen	Starten des Servers wiederhole immer Warten auf eine Clientverbindung und Herstellen der Verbindung Auslagern der Verbindung in Thread endewiederhole

Auch die frühen Implementierungen des Serververhaltens der Beispiele p01_wie gehts und p02_wiegehts2 sind leicht auf den neuen Server anpassbar und könnten als Zusatzaufgabe umgesetzt werden.

Dass Clientanfragen durch Auslagerung der Kommunikationsbehandlung in einen parallel aufgeführten Programmteil „quasi“ gleichzeitig behandelt werden können, ist auch für Buchungssysteme wichtig. Im Unterschied zur behandelten „Wetterauskunft“ greifen die Clientprozesse im Buchungssystem des Projekt p06_platzbuchung lesend und schreibend auf die Ressource *platzanzahl* zu.

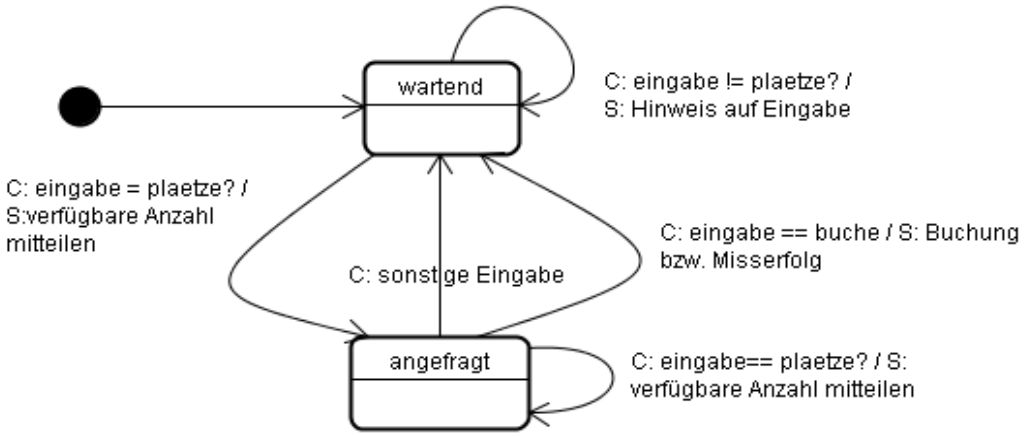
Zur Vertiefung implementieren die Schülerinnen und Schüler ausgehend vom Quelltext der Wetterauskunft ein Platzbuchungssystem und sind dadurch bereits mit dem Motivationsbeispiel des nächsten Kapitels vertraut, mit dessen Hilfe die Probleme beim gemeinsamen Zugriff durch parallele Prozesse auf Ressourcen aufzeigt werden.

Aufgabe 2: Buchungssystem (🖫 p06_platzbuchung_aufgabe)

Über einen Server soll eine Platzbuchung nach folgenden Regeln vorgenommen werden können:

Nach der Verbindungsherstellung *muss* der Client mindestens einmal die Anzahl der freien Plätze (Attribut *plaetzevorhanden* des Servers) durch Senden der Zeichenkette „plaetze?“ erfragen, erst anschließend kann ein Platz gebucht (sende „buche“) werden. Nach einer Buchung kann erst erneut gebucht werden, wenn wiederum mindestens einmal eine Anfrage nach den verfügbaren Plätze erfolgt ist.

a) Verwenden Sie die Zustände wartend und angefragt und implementieren Sie ausgehend von der KLASSE WETTERVERHALTEN2 der vorherigen Aufgabe eine Klasse PLATZBUCHUNG nach folgendem Zustandsdiagramm.



b) Ergänzen Sie den Quellcode der Klasse SERVER4 des Projekts p06_platzbuchung_aufgabe. Auf das Attribut *platzanzahl* des Servers soll über die Methode *PlaetzeVerfuegbar()* zugegriffen werden. Die eigentliche Buchung erfolgt über die Methode *PlaetzeBuchen(int anzahl)*, die den Wert *false* zurückgibt, falls die Buchung nicht möglich ist.

c) Testen Sie Ihre Implementierung. Kann Ihrer Meinung nach eine Überbuchung auftreten?

d) Zusatzaufgabe für besonders Schnelle: Erweitern Sie die Implementierung so, dass ein gebuchter Platz wieder storniert werden kann, solange der Client nicht beendet wird.

Die Schülerinnen und Schüler können einsehen, dass vor der eigentlichen Aktion (Buchung oder Stornierung) die Anzahl der freien Plätze nochmals überprüft werden muss. Diese Überprüfung bewahrt noch nicht vor Synchronisationsproblemen, gibt aber einen ersten Hinweis auf die Problematik, die im nächsten Kapitel durch eine Buchungssimulation gezeigt wird.

Lösung: 🖫 p06_platzbuchung_loesung

Zusätzliche Aufgabenmöglichkeiten und Implementierungen

Falls man bei besonders schnellen und leistungsstarken Schülergruppen die Implementierung des Chatservers weiter studieren bzw. umsetzen möchte, können weitere Aufgaben bearbeitet werden. Dieser Teil ist jedoch keineswegs verpflichtend.

Aufgabe 3 führt die maximale Anzahl an Verbindungen als weitere Ressource ein, die von den parallelen Clientprozessen verwendet wird. Falls diese Aufgabe bearbeitet wird, so kann bei der Behandlung von Synchronisationsproblemen im nächsten Kapitel zusätzlich auf dieses Beispiel zurückgegriffen werden.

In der Aufgabe 4 wird die Notwendigkeit einer Parallelisierung auf Clientseite problematisiert, die für ein funktionierendes Chatprogramm notwendig ist.

Aufgabe 3: maximale Anzahl von Verbindungen (🖫 zp1_maximaleclientanzahl)

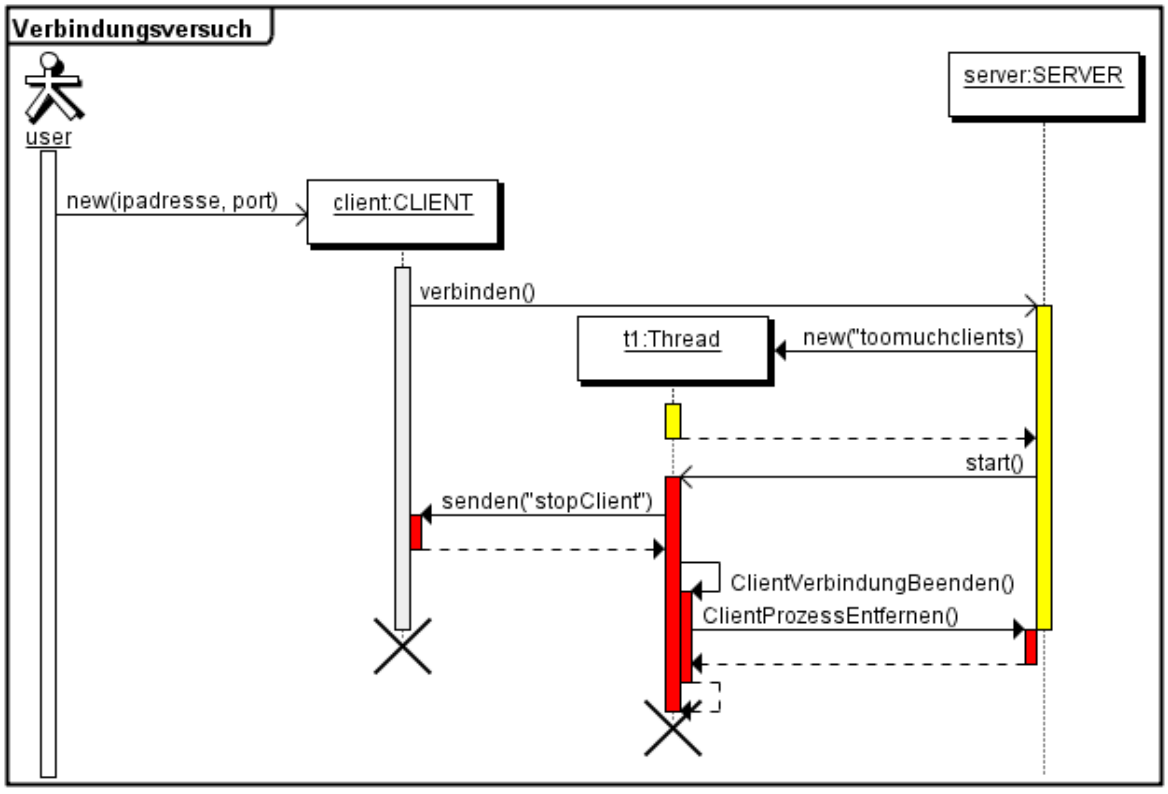
In der bisherigen Programmversion (p05_mehrclientsparallel) ist dem Server nicht bekannt, wie viele Clients bereits mit ihm verbunden sind. Dies könnte zu Engpässen der Ressourcen führen, wenn sich eine große Anzahl von Clients mit dem Server verbindet. Deshalb wird in der Praxis die Anzahl der Verbindung beschränkt.

Erweitern Sie die Klasse SERVER so, dass

- der Server die Anzahl der Verbindungen in einem Attribut *clientanzahl* speichert;
- der Server bei mehr als 2 (zu Testzwecken, später mehr) Verbindungen keine weitere Verbindung mehr zulässt (die Verbindung wird zunächst aufgebaut, der Client über die maximale Anzahl benachrichtigt und anschließend die Verbindung durch Senden der Antwort „Server [stopClient]“ beendet);
- der Server beim Verbinden und Beenden eines Clients die aktuelle Zahl der Verbindungen ausgibt.

Ein Verbindungsversuch läuft prinzipiell folgendermaßen ab:

Damit der Server dem Client mitteilen kann, dass keine (dauerhafte) Verbindung aufgebaut werden kann, muss der Client mit dem Server kurzzeitig verbunden sein. In der vorliegenden Implementierung sendet der Server nach Überprüfung der Verbindungsanzahl sofort das Stoppsignal zum Client, der sich darauf hin beendet. Der serverseitige Prozess des Clients trägt sich selbst aus der Clientprozessliste aus und wird dann beendet.



Stopp-Bedingungen für den Client und den Clientprozess des Servers:
Der Client und der Clientprozess soll künftig immer gestoppt werden, wenn die Serverantwort mit „Server[stopClient]“ beginnt.

Lösungshinweis: 🖫 zp1_maximaleclientanzahl

Aufgabe 4: Modellierung eines Chatservers (🖫 p10_chatserver)

Da bei einer Chatsitzung nicht bekannt ist, wann eine Nachricht vom Server eintrifft, muss der Client stets empfangsbereit sein. Bei der Realisierung eines Chatservers ist deshalb eine weitere Parallelisierung auf Clientseite nötig. Betrachtet man den Quellcode der Klasse CLIENT, so fällt

auf, dass das Clientprogramm entweder auf die Tastatureingabe oder auf eine Servernachricht wartet.

Modellieren Sie ausgehend vom vorliegenden Buchungsserver einen Chatserver.

Erstellen Sie dafür

a) ein Klassendiagramm der beteiligten Klassen;

b) ein Sequenzdiagramm, das den vergeblichen Anmeldeversuch eines Clients zeigt (maximale Zahl der Verbindungen erreicht);

c) ein Sequenzdiagramm eines erfolgreichen Anmeldeversuchs eines Clients;

d) ein Zustandsdiagramm für einen Clientprozess.

Folgende Aspekte sollen bei der Modellierung berücksichtigt werden:

- Eine Nachricht soll immer an alle Chatteilnehmer versandt werden.
- Für einen Clientprozess gibt es die Zustände: wartend, angemeldet.
- Ein Client soll einen eindeutigen Namen haben; erst nach der Anmeldung, bei der ein gültiger Name angeben werden muss, kann gechattet werden.
- Ein Client muss zu jeder Zeit Servernachrichten empfangen können.
- Es gibt eine Grenze für die Anzahl der Verbindungen.

Lösungshinweis: 🖫 p10_chatserver

2.4 Synchronisation von Prozessen

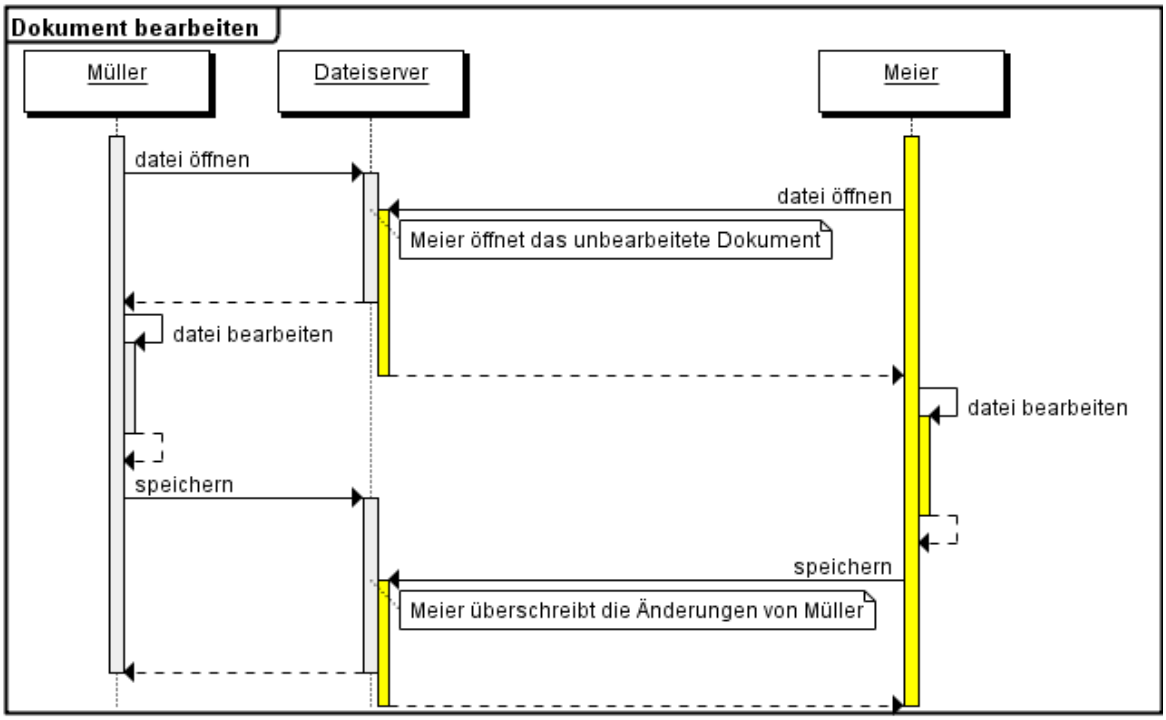
> Lp: [Die Schülerinnen und Schüler] erarbeiten Lösungsansätze für die Synchronisation derartiger Vorgänge an Beispielen aus dem täglichen Leben wie der Regelung des Gegenverkehrs durch eine einspurige Engstelle. Beim Übertragen dieser Ansätze auf die Prozesse im Computer wird ihnen bewusst, dass es bei jedem dieser Verfahren bestimmte Sequenzen gibt, die nicht von mehreren Prozessen gleichzeitig ausgeführt werden dürfen.

2.4.1 Kritischer Abschnitt, Monitore

2.4.1.1 Inhaltliche Grundlagen

Der Zugriff verschiedener Prozesse auf gemeinsame Ressourcen muss geregelt werden, damit es nicht zu Inkonsistenzen kommt.

Beispiel: Zwei Sekretärinnen sollen gemeinsam an einem Dokument arbeiten, das auf einem Server liegt. Wird die Veränderung des Dokuments (=Ressource) von beiden nicht abgesprochen (oder durch entsprechenden Schreibschutz geregelt), so könnte es vorkommen, dass eine Sekretärin die Änderungen der anderen überschreibt. Ein Sequenzdiagramm verdeutlicht das Problem.



Die einfachste Lösung des Problems besteht darin, dass die Datei exklusiv einer Bearbeiterin oder einem Bearbeiter zugeteilt wird, also nur einmal geöffnet und nur wechselseitig bearbeitet werden kann.

In der Informatik wird der Programmteil, der von verschiedenen Prozessen nur im wechselseitigem Ausschluss ausgeführt werden darf, als kritischer Abschnitt bezeichnet.

Als **kritischer Abschnitt** wird ein Programmabschnitt eines Prozesses bezeichnet, in dem Betriebsmittel (z.B. Daten, Verbindungen, Geräte usw.) verwendet werden und der nicht zeitlich verzahnt zu Programmabschnitten anderer Prozesse ausgeführt werden darf, die die gleichen Betriebsmittel ebenfalls verwenden. Andernfalls kann es zu inkonsistenten Zuständen der Betriebsmittel kommen.

Für den wechselseitigen Ausschluss von Prozessen im kritischen Abschnitt stellen höhere Programmiersprachen sogenannte **Monitore** zur Verfügung. Ein Monitor besteht aus einem oder mehreren Programmabschnitten, die immer nur exklusiv von einem einzigen Thread ausgeführt werden dürfen. Wollen weitere Threads diese Programmteile betreten, müssen sie warten. Im Falle des aktiven Wartens versuchen wartende Threads immer wieder aktiv, den Monitor zu betreten. Beim passiven Warten werden die Threads dagegen in einen Wartezustand versetzt und benachrichtigt, wenn der aktuell im Monitor befindende Thread den kritischen Abschnitt verlässt. Sobald ein Thread den Monitor betreten hat, werden die weiteren Bewerber um den Monitor abgewiesen und erneut in den Wartezustand versetzt.

Verklemmung

Falls der Zugriff auf mehr als eine Ressource synchronisiert werden muss, kann es zur Verklemmung (Dead Lock) kommen. Dabei sichern sich verschiedene Prozesse zunächst nur einen Teil der Ressourcen, die sie für ihre Aufgabe benötigen. Eine Verklemmung tritt auf, wenn die

Prozesse weitere zur Beendigung ihrer Arbeit nötige Ressourcen nicht bekommen können, ihrerseits aber die bereits reservierten Ressourcen nicht wieder frei geben.

Dies kann in folgender Definition formalisiert werden:

Verklemmung (Dead Lock):
Eine Menge von Prozessen $\{P_1,...,P_n\}$ sind in einer **Verklemmung**, wenn jeder dieser Prozesse auf ein Ereignis wartet, das nur von einem anderen Prozess (aus dieser Menge) ausgelöst werden kann.

Hinweis: Das Ereignis ist typischerweise die Freigabe einer benötigten Ressource.

2.4.1.2 Umsetzungshinweise

Buchungssimulation

Das Buchungssystem des vorherigen Kapitels kontrolliert die Anzahl der verfügbaren Plätze. Bevor ein Platz gebucht werden kann, wird seine Verfügbarkeit überprüft.

Für die Simulation (🖫 p07_buchungssimulation) wird eine leicht modifizierte Version des Buchungssystems aus dem vorherigen Kapitel (🖫 p06_platzbuchung_loesung) verwendet, bei der die Tastatureingabe in der Klasse CLIENT durch eine zufällige Auswahl aus den Eingaben „plaetze?“ und „buche“ ersetzt wurde.

Veränderungen in der Klasse CLIENT:

```
String[] eingaben = {"plaetze?", "buche"};

//Simulation !!! statt clientEingabe
int index = (int) Math.round(Math.random());
clientEingabe = eingaben[index];
//
```

In der Methode *PlaetzeBuchen()* des Servers wird die Anzahl der verfügbaren Plätze periodisch auf den Wert *platzkontingent* gesetzt, falls keine Plätze mehr verfügbar sind. Damit startet die Simulation quasi wieder „von vorne“. Außerdem wird ein verzögerter Zugriff auf die Platzanzahl simuliert:

Veränderungen in der Klasse SERVER:

```
public boolean PlaetzeBuchen(int anzahl) {

        boolean plaetzebuchbar = (anzahl <= plaetzevorhanden);

        // das Ermitteln, ob Plaetze buchbar sind braucht Zeit ...
        try {
                    Thread.sleep((int)(Math.random() * 100));
        } catch (InterruptedException e) { }

        if (plaetzebuchbar) {
            plaetzevorhanden = plaetzevorhanden - anzahl;
            System.out.println(plaetzevorhanden + " Plaetze vorhanden.");

            if (plaetzevorhanden < 0) {//bei Überbuchung abbrechen.
                System.exit(1);
            }
            if (plaetzevorhanden == 0) {//Kontingent wieder auffüllen
                plaetzevorhanden = platzkontingent;
            }
            return true;
        } else {
            return false;
        }
    }
```

Hinweise:

- Die zusätzlich hinzugekommenen Zeilen sind farblich hinterlegt.
- Thread.sleep(zeitangabe) muss nicht die angegebene Zeit lang warten, sondern kann im Extremfall auch sofort zurückkehren, falls das System nicht ausgelastet ist.

Monitorkonzept

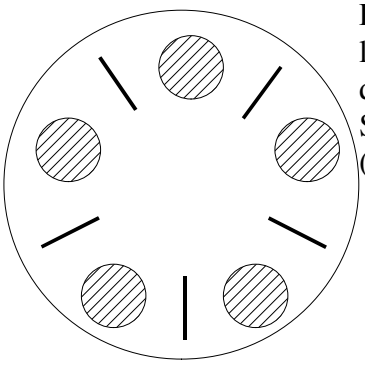
In Java ist das Monitorkonzept auf Objektebene implementiert, d. h. ein Objekt kann den wechselseitiger Ausschluss bei Zugriffen auf eine gemeinsam genutzte Datenstruktur garantieren. Die Deklaration der zu schützenden Methoden als „synchronized“ genügt. Synchronisierte Methoden benötigen in der Regel wesentlich mehr Ausführungszeit als nicht synchronisierte Methoden, man wird also möglichst nur die Methoden schützen, die kritische Abschnitte enthalten.

Will man, wie im Beispiel des Chatservers und den Zusatzprojekten zp2_maximaleclientanzahl_mit_Liste und zp3_platzbuchung_mit_maxclientanzahl, die Referenzen auf die verbundenen Clients in einem Feld speichern, ergibt sich ein weiteres Synchronisationsproblem. Da sich in den vorliegenden Implementierungen die Clientprozesse selbst wieder aus diesem Feld austragen, darf der Zugriff auf dieses Feld nur im wechselseitigem Ausschluss erfolgen. Die Methoden *ClientProzessHinzufuegen* und *ClientProzessEntfernen* müssen deshalb ebenfalls synchronisiert werden.

Simulation „Die speisenden Philosophen“:

An einem runden Tisch sitzen fünf Philosophen, die abwechselnd speisen und denken.

Ein Philosoph kann nur zu speisen beginnen, falls er das rechts und links von ihm liegende Stäbchen von den vorhandenen fünf Stäbchen an sich nehmen konnte.



Es kommt zur Verklemmung, wenn *jeder* der Philosophen z. B. das links von ihm liegende Stäbchen bereits an sich genommen hat und auf das rechts vom ihm liegende Stäbchen wartet. In der beiliegenden Simulation führt das dazu, dass jeder der Philosophen nur noch „denkt“ (das genommene Stäbchen ist vor dem Philosophennamen vermerkt).

```
[L] Philosoph 4 denkt
[L] Philosoph 3 denkt
[L] Philosoph 0 denkt
[L] Philosoph 2 denkt
[L] Philosoph 1 denkt
[L] Philosoph 4 denkt
```

Die Verklemmung kann nur gelöst werden, falls einer der Philosophen sein Stäbchen wieder her gibt, obwohl er auf das zweite Stäbchen wartet. Eine Verklemmung ließe sich nur verhindern, wenn der Philosoph beide Stäbchen ausschließlich gleichzeitig anfordern würde und beide Stäbchen auch nur gleichzeitig zugeteilt würden.

Hinweis: Die Möglichkeit einer Verklemmung bei Verwendung mehrerer Ressourcen wird an dieser Stelle nur angesprochen. Keinesfalls sollen für das beiliegende Programm Lösungen zur Vermeidung der Verklemmung implementiert werden. Eine weitere typische Deadlock-Situation findet man bei „Rechts vor Links“ im Straßenverkehr.

2.4.1.3 Möglicher Unterrichtsablauf

Mit Hinweis auf das Platzbuchungsprogramm aus dem vorherigen Kapitel werden die Schülerinnen und Schüler mit der Simulation vertraut gemacht. Für die weitere Vorgehensweise reicht die

Erläuterung aus, dass die Benutzereingaben (plaetze? und buche) zufallsgesteuert vom Clientprogramm simuliert werden und die Buchungen zurückgesetzt werden, wenn die Plätze ausgebucht sind, damit der Buchungsvorgang von Neuem beginnen kann. Der Quellcode muss hierbei nicht analysiert werden.

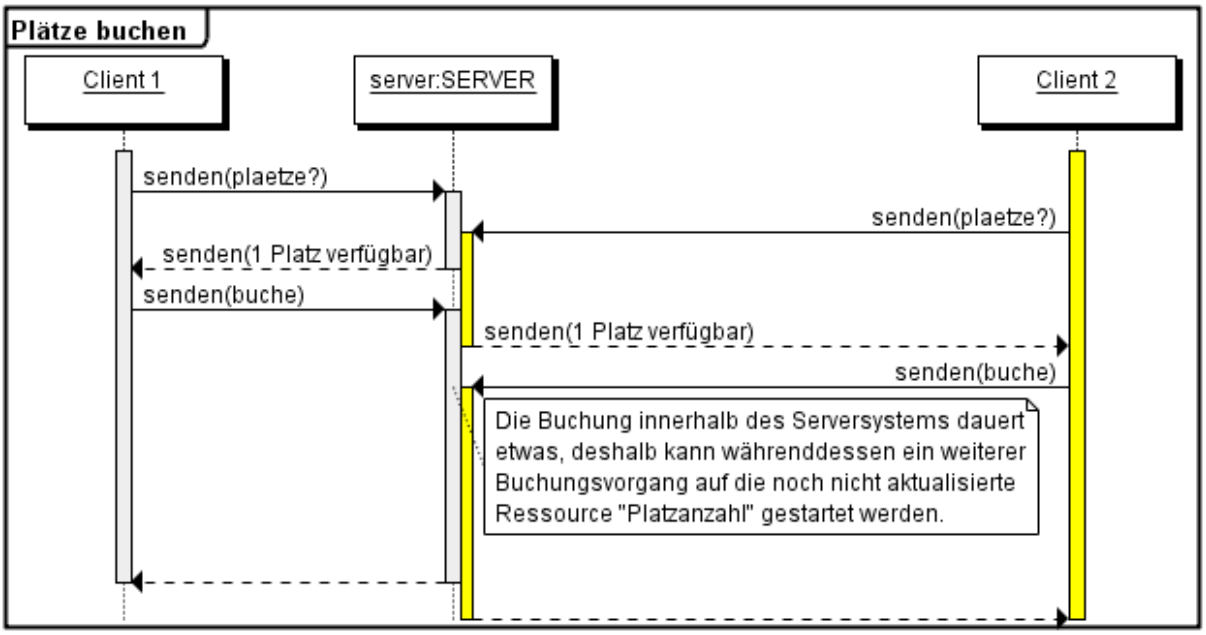
In der Simulation wird die korrekte Arbeitsweise des Buchungssystems getestet.

Die Simulation (🖫 p07_buchungssimulation) bricht mit einer Überbuchung ab, wenn man mehrere Clients mit dem Server verbindet. Im Beispiel buchen zwei Clients je einen Platz, obwohl keine Plätze mehr verfügbar sind (0 Plätze verfügbar). Die Statusmeldung zeigt deshalb eine vorhandene Platzzahl von -1.

```
5 Plaetze vorhanden.
4 Plaetze vorhanden.
3 Plaetze vorhanden.
2 Plaetze vorhanden.
1 Plaetze vorhanden.
-1 Plaetze vorhanden.
Überbuchung ! Beenden mit Return...
-1 Plaetze vorhanden.
Überbuchung ! Beenden mit Return...
```

Die Schülerinnen und Schüler erkennen, dass die Ursache des Problems darin besteht, dass verschiedene Threads (CLIENTPROZESS) auf die gleiche Ressource (Plaetze) zugreifen wollen.

Ein Sequenzdiagramm kann dies verdeutlichen:



Die Kennzeichnung der Methode *PlaetzeBuchen()* mit dem Schlüsselwort „**synchronized**“ verhindert, dass mehrere Prozesse diese Methode gleichzeitig betreten können. Eine Überbuchung ist dadurch künftig ausgeschlossen.

Nachdem sich die Schülerinnen und Schüler überzeugen konnten, dass die Wiederholung der Buchungssimulation mit verändertem Quelltext offensichtlich nicht mehr zu einer Überbuchung führt, wird die Definition eines kritischen Abschnitts formuliert (siehe inhaltliche Grundlagen).

Da in Programmiersprachen (wie z. B. Java), die das Monitorkonzept implementieren, ein wechselseitiger Ausschluss von Prozessen leicht zu realisieren ist, kommt es vor allem darauf an, die kritischen Abschnitte zu erkennen. Dies soll nochmals bei der Bahnplatzbuchung geübt werden.

Identifizierung des kritischen Abschnitts bei der Bahnplatzbuchung

Aus Gründen der Übersichtlichkeit wird im Beispiel p08_bahnsimulation auf die Client-Server-Kommunikation verzichtet. Nach der Demonstration des Programms sollen die Schülerinnen und Schüler den kritischen Abschnitt identifizieren, den Quellcode verändern und sich davon überzeugen, dass in der korrigierten Version keine Doppelbuchung auftritt. Die Prozesse der Kunden rufen die Methode *nochFrei()* und *getName()* auf. In der Methode *nochFrei()* befindet sich der kritische Abschnitt.

Aufgabe 1: Bahnsimulation

Untersuchen Sie den Quellcode des Programms Bahnsimulation (🖫 p08_bahnsimulation) und identifizieren Sie den kritischen Abschnitt des Programms. Korrigieren Sie den Quellcode und überzeugen Sie sich anschließend, dass keine Doppelbuchungen mehr auftreten. Begründen Sie, wieso es ohne Maßnahmen zu Fehlbuchungen kommen kann.

Lösungshinweise:

In der vorliegenden Simulation ist der Zugriff auf die freien Plätze nicht synchronisiert. Nach einem Durchlauf findet man im Buchungsprotokoll doppelte Belegungen:

```
Kunde 1: Platz 34 in der DB reserviert
Kunde 2: Platz 35 in der Vogtlandbahn reserviert
Kunde 1: Platz 35 in der Vogtlandbahn reserviert
Kunde 2: Platz 36 in der Vogtlandbahn reserviert
Kunde 1: Platz 36 in der Vogtlandbahn reserviert
```

Synchronisiert man die Methode *nochFrei()* der Klasse ZUG, so tritt keine Doppelbuchung mehr auf.

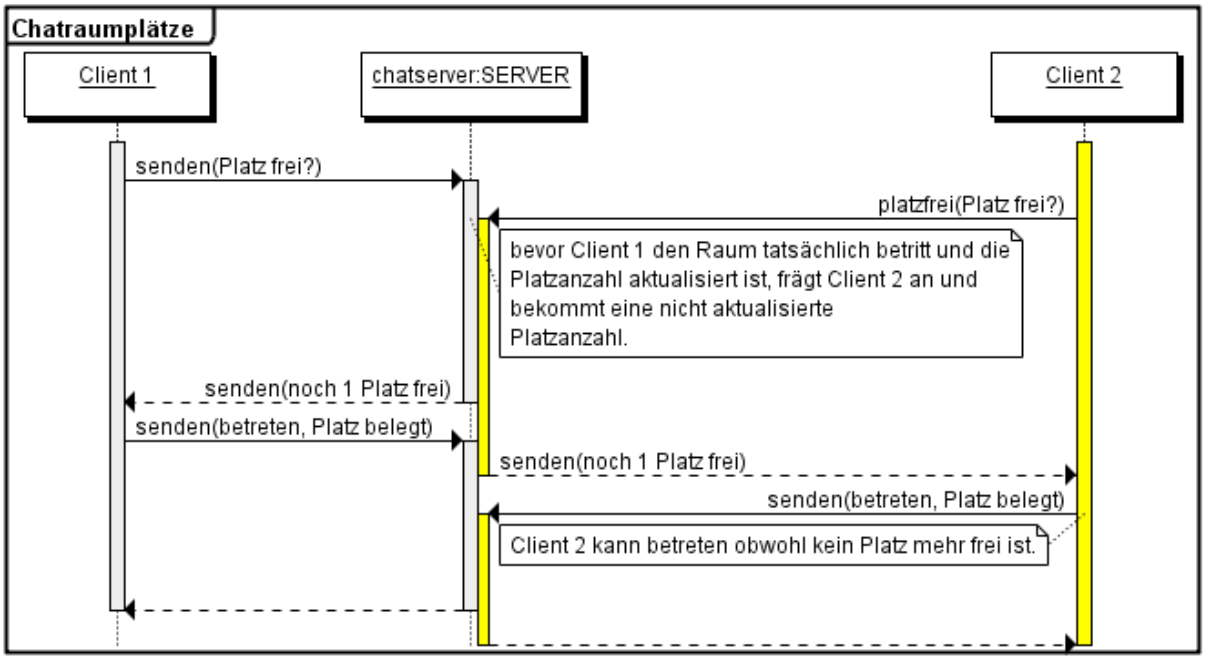
Ein ähnliches Problem tritt auf, falls die Anzahl der Teilnehmer innerhalb eines Chatraums begrenzt ist. Dies kann zur weiteren Vertiefung untersucht werden.

Aufgabe 2: Chaträume mit einer festen Anzahl von freien Plätzen

Die Anzahl der Plätze in einem Chatraum soll auf 20 begrenzt werden. Obwohl durch das Programm die Anzahl der Teilnehmer kontrolliert wird, stellen die Entwickler in der Testphase des Programms fest, dass plötzlich mehr als 20 Teilnehmer in einem Raum vorhanden sind.

Erläutern Sie unter Verwendung eines Sequenzdiagramms und eines geeigneten Codefragments (Pseudocode genügt), wie es dazu kommen kann, dass mehr als 20 Teilnehmer in den Raum gelangen, und welche Maßnahmen dagegen ergriffen werden können.

Lösungshinweise:



Kritischer serverseitiger Abschnitt im Pseudocode:

falls noch Plätze im Chatraum frei sind
füge Client zu den Chatraummitgliedern hinzu
aktualisiere Anzahl der freien Plätze
endefalls

Falls die Aufgabe 4 im Kapitel 2.3.3 behandelt wurde, kann optional bei der Besprechung der Lösung von Aufgabe 2 die vorliegende Implementierung zur Begrenzung der Anzahl der Clientverbindungen (🖫 zp2_maximaleclientanzahl_mit_Liste) untersucht werden. Bei der Prüfung des Serverquelltextes könnte man in der Methode *ClientProzessHinzufuegen()* einen kritischen Abschnitt finden:

```
private void ClientProzessHinzufuegen(Socket clientSocket) throws IOException {

if (clientprozesse.size() < maximaleclientanzahl) {

    CLIENTPROZESSZ1 clientprozess = new CLIENTPROZESSZ1(clientSocket, this, "");
    clientprozess.start();
    clientprozesse.add(clientprozess);
    System.out.println(clientprozesse.size() + "");

} else {...}
```

Ab hier ist die Listenlänge erst wieder aktuell!

Tatsächlich erfolgt hier ein nicht synchronisierter Zugriff auf die Liste der Clients. In der Zeit zwischen der Prüfung der Listenlänge und dem Hinzufügen in die Liste könnte sich ein anderer Client eingetragen haben. Außerdem können sich die Clients beim Beenden jederzeit selbständig aus der Liste austragen. Dadurch besteht die Gefahr eines inkonsistenten Zustands der Clientliste. Die Methode *ClientProzessHinzufuegen()* muss synchronisiert werden. Aber auch die Methode *ClientProzessEntfernen()* muss synchronisiert sein:

```
public synchronized void ClientProzessEntfernen(CLIENTPROZESS clientprozess) {

    clientprozesse.remove(clientprozess);
```

```
    System.out.println("Clientverbindung beendet ");
    System.out.println(clientprozesse.size() + " Verbindung(en)");

}
```

Die Methoden *remove* und *add* der Clientliste dürfen ebenfalls nicht parallel ausgeführt werden. Dies ist hier durch den Monitor der umgebenden Methoden sichergestellt.

Zugriff auf mehr als eine Ressource (Busplatz und Stadtbesichtigung)

In Anlehnung an die Platzbuchung wird folgendes Problem untersucht und anschließend der Begriff Verklemmung erläutert.

Bei einer Busreise können wie in der vorherigen Aufgabe Plätze im Bus gebucht werden. Der Veranstalter bietet auch eine Besichtigung am Reiseziel an, die extra gebucht werden muss. Während ein Reiseteilnehmer den letzten Busplatz buchen konnte, hat sich ein anderer Teilnehmer derweil den letzten Platz der Stadtbesichtigung reserviert. Beide Teilnehmer wollen aber mit dem Bus fahren und an der Stadtbesichtigung teilnehmen. Obwohl nun Busplatzkontingent und Stadtbesichtigungsplätze komplett vergeben sind, kann kein korrekter Abschluss des Buchung erfolgen. Der Busplatzbesitzer gibt den Platz nicht her und versucht ständig einen Platz für die Stadtbesichtigung zu erhalten und umgekehrt.

Den Schülerinnen und Schülern wird klar, dass das Problem durch die Notwendigkeit der Reservierung zweier Ressourcen verursacht wird.

Dies wird in der Definition eines Deadlocks verallgemeinert.

Hinweis: In der Literatur findet man häufig zusätzliche Bedingungen, die erfüllt sein müssen, damit eine Verklemmung auftreten kann. Eine umfassende Behandlung dieser Bedingungen bzw. Strategien zur Verhinderung und Vermeidung von Verklemmungen ist nicht erforderlich. Es reicht, wenn der Begriff „Verklemmung“ an Beispielen erläutert und darüber hinaus allgemein definiert wird.

Speisende Philosophen

Auch der Philosoph benötigt zwei Ressourcen (linkes und rechtes Stäbchen), die nacheinander angefordert werden, um zu essen. Bevor man die Simulation (🖫 p09_speisendePhilosophen) vorführt, sollen die Schülerinnen und Schüler die Möglichkeit der Verklemmung erkennen und die Verklemmungssituation näher beschreiben.

2.4.2 Erzeuger-Verbraucher-Problem, aktives und passives Warten

2.4.2.1 Inhaltliche Grundlagen

Falls ein Prozess vom Ergebnis eines anderen Prozesses abhängig ist, müssen die Prozesse neben der möglicherweise notwendigen Synchronisation von Zugriffen auf eine gemeinsame Ressource auch aufeinander warten können. Wird zum Beispiel bei der Platzvergabe auch eine Stornierung zugelassen, so könnte ein anderer Kunde zum Zuge kommen, sobald ein Platz frei wird. Dabei kann je nach Kundenverhalten zwischen aktivem und passivem Warten unterschieden werden: Aktives Warten liegt vor, wenn der Kunde ununterbrochen von sich aus wiederholte Platzanfragen stellt. Beim passiven Warten würden alle wartenden Kunden vom Buchungsserver benachrichtigt werden, falls ein Platz frei geworden ist.

Als Standardbeispiel zweier Prozesse, die aufeinander warten müssen, findet man in der Literatur das „Erzeuger-Verbraucher-Problem“:

In einem System nutzen verschiedene Prozesse eine gemeinsame Datenstruktur (Speicher), in der die sogenannten Erzeugerprozesse Elemente ablegen können. Verbraucherprozesse entnehmen der Datenstruktur diese Elemente und verarbeiten sie. Sie müssen warten, bis entsprechende Elemente zur Weiterverarbeitung vorliegen. Erzeugerprozesse hingegen können ihre Produkte nur ablegen, falls die Datenstruktur nicht bereits eine maximale Anzahl von Elementen (Speicher ist voll) enthält, sonst müssen sie ebenfalls warten.

2.4.2.2 Umsetzungshinweise

Das Erzeuger-Verbraucher-Modell kann bei folgendem Produktionsablauf betrachtet werden:

Zwei Roboter arbeiten nebenläufig innerhalb einer Produktionskette. Zur Übergabe der Ware von Roboter E (Erzeuger) an Roboter V (Verbraucher) wird ein Abstellplatz verwendet, der genau einen Artikel aufnehmen kann.

Da die Arbeitsgeschwindigkeit der Roboter auch noch von anderen Randbedingungen abhängt, kann es unterschiedlich lange dauern, bis Roboter V die Ware abholt. Da der Abstellplatz derweil belegt ist, muss Roboter E warten, bis er die neu erzeugte Ware dort zur Übergabe abstellen kann. Umgekehrt muss auch der Roboter V warten, bis auf dem Abstellplatz eine Kiste steht, die er weiterverarbeiten kann.

Zur Modellierung einer Simulation werden sowohl beim aktiven (🖫 p11_aktivesWarten) als auch beim passiven (🖫 p12_passivesWarten) Warten die Klassen ERZEUGER (Roboter E), VERBRAUCHER (Roboter V) und die KLASSE ABSTELLPLATZ verwendet.

Aktives Warten:

Falls die Methoden *abholen()* und *abstellen()* der Klasse ABSTELLPLATZ im Falle einer erfolgreichen Aktion wahr zurückgeben, setzen die Roboter ihre Arbeit fort. Andernfalls probieren sie erneut, ob sie die Ware abstellen bzw. abholen können.

ERZEUGER	VERBRAUCHER

erzeugen();
 	
wiederhole immer
 	
  wenn ablegen() wahr liefert
  erzeugen();
  endewenn
 	
endewiederhole
	
	wiederhole immer
	
	  wenn abholen() wahr liefert
	  weiterverarbeiten();
	  endewenn
	
	endewiederhole

ABSTELLPLATZ	
Methode ablegen	Methode abholen

wenn platzbelegt den Wert falsch hat
  platzbelegt = wahr
  gib wahr zurück *//wurde abgelegt*
endewenn
 	
gib falsch zurück
	wenn platzbelegt den Wert wahr hat
	  platzbelegt = falsch
	  gib wahr zurück *//wurde abgeholt*
	endewenn
	
	gib falsch zurück

Betrachtet man die Möglichkeit, dass mehrere Erzeuger oder mehrere Verbraucher vorhanden sind, so wird schnell klar, dass der Zugriff auf die Variable *platzbelegt* (allgemein auf die Ressource) synchronisiert erfolgen muss. Der kritische Abschnitt beginnt wie in den Beispielen zuvor mit der Prüfung des Variablenwerts von *platzbelegt*.

Passives Warten:

Aktives Warten kostet Rechenzeit und beansprucht dadurch unnötig Systemressourcen. Deshalb ist es besser, sich die wartenden Interessenten (ähnlich einer Warteliste) zu merken und zu benachrichtigen, falls die benötigte Ressource zur Verfügung steht. Dazu wird das Monitorkonzept um die Methoden *Warten()* und *Benachrichtigen()* erweitert.

Der Aufruf der Methode *Warten()* innerhalb des Monitors schickt den Thread in einen Wartezustand, der Thread wird auf die Liste der wartenden Threads gesetzt.

Der Aufruf der Methode *Benachrichtigen()* benachrichtigt alle wartenden Prozesse, dass sich der Ressourcenzustand geändert hat. Sobald der Monitor freigegeben ist, kann ein Prozess aus der Warteliste Zugriff auf die Ressource erhalten. Die anderen Prozesse werden durch den Aufruf der Methode *Warten()* wieder in den Wartezustand versetzt, falls die Ressource bereits erneut belegt ist.

ERZEUGER	VERBRAUCHER

wiederhole immer
 	
erzeugen();
ablegen();
 	
endewiederhole
	wiederhole immer
	
	abholen();
	weiterverarbeiten();
	
	endewiederhole

ABSTELLPLATZ	
Methode ablegen	Methode abholen

wiederhole solange platzbelegt den Wert wahr hat
warten();
ende wiederhole
 	
platzbelegt = wahr;
benachrichtigen();
	wiederhole solange platzbelegt den Wert falsch hat
	warten();
	endewiederhole
	
	platzbelegt = falsch;
	benachrichtigen();

2.4.2.3 Möglicher Unterrichtsverlauf

Ausgehend vom Beispiel der Platzbuchung werden mögliche Szenarien für die Platzvergabe im Falle einer Stornierung erarbeitet:

Beim **aktiven Warten** erhält derjenige den Platz, dessen Anfrage zufällig als erste nach der Stornierung eintrifft. Dazu ist es nötig, dass der Kunde die Anfrage ständig wiederholt.

Beim **passiven Warten** werden die Interessenten in eine Liste vorgemerkt und beim Freiwerden eines Platzes benachrichtigt.

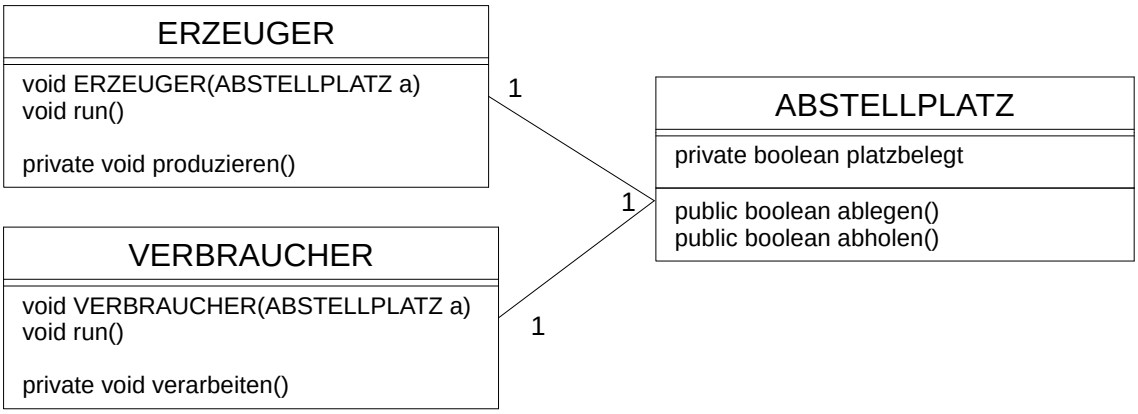
Zur weiteren Vertiefung der Begriffe wird anschließend nun das Beispiel zweier nebenläufig arbeitender Roboter innerhalb einer Produktionskette untersucht (siehe Abschnitt 2.4.2.1). Es werden zunächst nur ein Erzeuger (Roboter E) und ein Verbraucher (Roboter V) betrachtet.

Aufgabe 1: Aktives Warten (🖫 p11_aktivesWarten_aufgabe)

Zwei Roboter arbeiten innerhalb eines Produktionsablaufes zusammen: Dazu verpackt Roboter E die Ware in einen Karton und legt diesen auf einem Übergabeplatz ab. Von dort wird der Karton vom Roboter V abgeholt, etikettiert und weiter auf Paletten verladen. Da die Arbeitsgeschwindigkeit beider Roboter noch von anderen Randbedingungen abhängt, muss die Übergabe zeitlich geregelt werden.

a) Formulieren Sie in Worten, wie die Übergabe zeitlich geregelt werden kann. Wer muss dabei auf wen warten?

b) Im folgenden soll die Belegung des Übergabeplatzes modelliert werden. Die Klassen der Roboter E (Klasse ERZEUGER) und V (Klasse VERBRAUCHER) haben folgendes Klassendiagramm, der Abstellplatz wird mithilfe der Klasse ABSTELLPLATZ modelliert, für die Ihnen der Quellcode vorliegt. Ergänzen Sie den Quellcode der Methode *run()* der Klassen ERZEUGER und die Klasse VERBRAUCHER (🖫 p11_aktivesWarten_aufgabe).



c) Warten die Roboter aktiv oder passiv aufeinander? Erläutern Sie Ihre Antwort.

d) Erkunden Sie das Verhalten des Programms, falls mehrere Objekte der Klasse VERBRAUCHER eingesetzt werden. Erläutern Sie mögliche Ursachen für Programmfehler.

Lösungshinweise zur Aufgabe 1: (🖫 p11_aktivesWarten_loesung)

Das beiliegende Programm p11_AktivesWarten_aufgabe ist ein erster Implementierungsversuch zur Simulation des obigen Problems und wird als Quelltextfragment eingesetzt.

In der Aufgabe sollen die Schülerinnen und Schüler den Quelltext ergänzen und die fehlende Synchronisation der Methoden *abholen()* und *ablegen()* in dieser Implementierung erkennen und beheben. Werden *mehrere* Roboter eingesetzt, um die Ware am Abstellplatz abzuholen, führt dies schnell zur fehlerhaften Ausführung im Programm:

Es wird die Kiste zweimal hintereinander abgeholt bzw. abgelegt!

```
Platz wurde belegt
Platz wurde frei
Platz wurde frei
Platz wurde belegt
Platz wurde belegt
```

Nach der Identifikation des kritischen Abschnitts werden die Methoden *ablegen()* und *abholen()* synchronisiert. Die Schülerinnen und Schüler überzeugen sich nun von der korrekten Arbeitsweise des Programms, indem sie mehrere Erzeuger und Verbraucher einsetzen.

Das Programm soll nun so verändert werden, dass die Roboter passiv aufeinander warten (🖫 p12_passivesWarten):

In Java hat jedes Objekt mit der Objektmethode *wait()* die Möglichkeit, Threads, die sich im aktuellen Monitor befinden, in den Wartezustand zu versetzen. Soll der Zugriff durch die wartenden Threads wieder möglich sein, so ruft das Objekt die Methode *notify()* oder *notifyAll()* auf. Der Aufruf von *notify()* bewirkt, dass genau einer der wartenden Threads benachrichtigt wird und den Zugriff auf den Monitor erhält. Warten mehrere Threads, so wird dieser zufällig ausgewählt. Beim Aufruf von *notifyAll()* werden alle wartenden Threads aus dem Wartezustand geholt, der Scheduler des Betriebssystems entscheidet dann, welcher Thread als erster Rechenzeit der CPU zugeteilt bekommt und damit den Monitor als erster betreten kann. Hat ein Thread den Monitor betreten, so werden die restlichen Threads wieder in die Warteposition versetzt, sobald sie den Zugriff auf den Monitor versuchen.

Der Zugriff auf den Lagerplatz erfolgt im Beispiel über die Methoden *ablegen()* und *abholen()*:

```
public synchronized void ablegen() {

    while (platzbelegt) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    platzbelegt = true;
    System.out.println("Platz wurde belegt");
    notifyAll();
}

public synchronized void abholen() {

    while (!platzbelegt) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    platzbelegt = false;
    System.out.println("Platz wurde frei");
    notifyAll();
}
```

Eine weitere Implementierungsvariante kann für besonders gute Schülerinnen und Schüler darin bestehen, dass mehrere Erzeuger und Verbraucher, die durch ein Attribut *name* unterschieden werden, eingesetzt werden. Sowohl Erzeuger und Verbraucher könnten zudem eine verschiedene Anzahl an Kisten anliefern bzw. abholen (Abstellplatz nimmt beispielsweise bis zu 5 Kisten auf, Abholer können zwei oder auch drei Kisten mitnehmen).

Anhang zu Kapitel 2: Praxis-Tipps

A1 Verbindung mit Mailserver via Telnet

Bei den Betriebssystemen Windows Vista und höher muss der Telnet-Client extra aktiviert werden. Anleitungen dazu findet man im Internet.

A2 Implementierung einer Client-Server-Kommunikation

A2.1 Allgemeine Tipps und Vorgehensweise bei den Implementierungen von Client-Server-Kommunikation

- **Vorsicht „Firewall“:** Möglicherweise verhindert eine rechnereigene Firewall, dass sich zwei Rechner verbinden können. Weichen Sie gegebenenfalls auf andere Ports aus (oder schalten Sie die Firewall aus).
- **Zwei BlueJ-Instanzen:** Bei der Demonstration von Server- und Clientprogrammen auf nur einem Rechner durch den Lehrer muss das Programm BlueJ zweimal gestartet werden, da pro BlueJ-Instanz nur ein Konsolenfenster zur Verfügung steht. In Programmen wie z. B. Netbeans ist dies nicht nötig.
- **Javakonsolen-Problem:** Startet man ein executable JAR mit einem Doppelclick unter Windows, so wird standardmäßig *javaw.exe* verwendet, um das Programm auszuführen. Dabei erscheint kein Konsolenfenster. Deshalb werden zum Starten der Programme Batchdateien verwendet.
- **Problem mit Umlauten:** Bei der Kommunikation sollte möglichst auf Umlaute verzichtet werden, da es zu Konvertierungsproblemen kommen kann. Java codiert die Zeichen nach ISO-8859-1 (bzw. Unicode), während die Dos-Eingabeaufforderung aber Codepage 850 codierte Zeichen ausgibt. Die beiligenden Batchdateien verwenden deshalb die Option -DfileI.encoding=cp850 zum Start der Javaprogramme.

A2.2 Übersicht über den Zusammenhang der beiliegenden Implementierungen der Client-Server-Projekte

p01_wiegehts	SERVER	SERVERVERHALTEN	CLIENT
p02_wiegehts2	SERVER Umbenennung von SERVERVERHALTEN zu SERVERVERHALTEN2	SERVERVERHALTEN2 verändertes Serververhalten lt. Aufgabe 2, Kapitel 2.2.2.2	CLIENT unverändert
p03_wetterverhalten (optionale Implementierung)	SERVER Umbenennung von SERVERVERHALTEN2 zu WETTERVERHALTEN	WETTERVERHALTEN	CLIENT unverändert
p04_mehrclients_aufgabe	SERVER2 für Quelltextergänzung	WETTERVERHALTEN2 für Quelltextergänzung	CLIENT2 für Quelltextergänzung
p04_mehrclients_loesung	SERVER2 Stopbedingung für Client: „Server[stopClient]:“	WETTERVERHALTEN2 Antwort auf „beenden“: „Server[stopClient]:“	CLIENT2 Bedingung zum Beenden: „Server[stopClient]:“
p05_mehrclientsparallel	SERVER3 Auslagerung der Serververarbeitung in einen nebenläufigen Prozess unter Verwendung der Klasse CLIENTPROZESS.	WETTERVERHALTEN2 unverändert	CLIENT2 unverändert
p06_platzbuchung_aufgabe	SERVER4	PLATZBUCHUNG	CLIENT2

	Implementierung der Methoden *PlaetzeVerfuegbar()* *PlaetzeBuchen()* neues Attribut plaetzevorhanden	Implementierung der Methode *HoleAntwort()*	unverändert
p06_platzbuchung_loesung	SERVER4 erhält die Methoden *PlaetzeVerfuegbar()* *PlaetzeBuchen()* und das Attribut plaetzevorhanden	PLATZBUCHUNG Die Methode *HoleAntwort ()* ruft erstmalig auch Methoden des Servers auf	CLIENT2 Unverändert
p08_buchungssimulation	SERVER5 Das Platzkontingent in der Methode *PlaetzeVerfuegbar()* wird immer wieder aufgefrischt	PLATZBUCHUNG unverändert	CLIENT3 Die Eingabe des Benutzers wird zufallsgesteuert simuliert.
p10_chatserver	CHATSERVER Unterscheidung zwischen angemeldeten Usern und Anzahl der Verbindungen Neue Methoden: *SpitznameVorhanden()* *PrivateNachrichtSenden()* *PrivateNachrichtSenden()* mit weiterer Signatur *AnAlleSenden()* CLIENTPROZESS Attribut spitzname *SpitznameHolen()* *Senden()*	Das Verhalten wird komplett in die Klasse CLIENTPROZESS übertragen	CHATCLIENT Der Prozess zum Senden der Tastatureingabe wird erst gestartet, wenn die maximale Verbindungsanzahl nicht überschritten ist. Empfängt die Servernachrichten. CLIENTSENDER Dieser nimmt Tastatureingabe entgegen und sendet diese.

Optionale Programme

zp1_maximaleclientanzahl	SERVERZ3 Der Server erhält das Attribut maximaleclientanzahl und lässt nicht mehr Verbindungen zu. CLIENTPROZESSZ1	WETTERVERHALTENZ2 bei der Servernachricht „toomuchclients“ gibt die Methode HoleAntwort() als Antwort die Abbruchbedingung „Server [stopClient]: ...“ zurück	CLIENT2 unverändert
zp2_maximaleclientanzahl_mit_Liste	SERVERZ4 Der Server verwaltet die Clientprozesse in einer Liste CLIENTPROZESSZ1	WETTERVERHALTENZ2 Unverändert	CLIENT2 unverändert
zp3_platzbuchung_mit_maxclientanzahl	SERVERZ5 Der Server erhält das Attribut maximaleclientanzahl und lässt nicht mehr Verbindungen zu und verwaltet ein Platz-	PLATZBUCHUNG2 bei der Servernachricht „toomuchclients“ gibt die Methode HoleAntwort() als Antwort die Abbruchbedingung „Server [stopClient]: ...“ zurück	CLIENT2 unverändert

zp1_maximaleclientanzahl	SERVERZ3 Der Server erhält das Attribut maximaleclientanzahl und lässt nicht mehr Verbindungen zu. CLIENTPROZESSZ1	WETTERVERHALTENZ2 bei der Servernachricht „toomuchclients“ gibt die Methode HoleAntwort() als Antwort die Abbruchbedingung „Server [stopClient]: ...“ zurück	CLIENT2 unverändert
	kontingent CLIENTPROZESSZ2 WETTER-VERHALTENZ2 wird zu PLATZBUCHUNG2		

3 Funktionsweise eines Rechners (ca. 17 Std.)

Lp: Am Modell der Registermaschine lernen die Schüler den grundsätzlichen Aufbau eines Computersystems und die Analogie zwischen den bisher von ihnen verwendeten Ablaufmodellen und Maschinenprogrammen kennen. So wird ihnen auch bewusst, dass Möglichkeiten und Grenzen theoretischer algorithmischer Berechnungsverfahren für die reale maschinelle Verarbeitung von Information ebenfalls gelten.

Beispiele zeigen den Schülern, wie einfache Algorithmen auf systemnaher Ebene durch Maschinenbefehle realisiert werden können. Dabei beschränken sich Anzahl und Komplexität der benutzten Maschinenbefehle auf das für die Umsetzung der gewählten Beispiele Wesentliche. Für das Verstehen des Programmablaufs ist insbesondere die in Jahrgangsstufe 10 erlernte Zustandsmodellierung eine große Hilfe. Zur Überprüfung ihrer Überlegungen setzen die Schüler eine Simulationssoftware für die Zentraleinheit ein, die die Vorgänge beim Programmablauf veranschaulicht.

- Aufbau eines Computersystems: Prozessor (Rechenwerk, Steuerwerk), Arbeitsspeicher, Ein- und Ausgabeeinheiten, Hintergrundspeicher; Datenbus, Adressbus und Steuerbus
- Registermaschine als Modell eines Daten verarbeitenden Systems (Datenregister, Befehlsregister, Befehlszähler, Statusregister); Arbeitsspeicher für Programme und Daten (von-Neumann-Architektur), Adressierung der Speicherzellen
- ausgewählte Transport-, Rechen- und Steuerbefehle einer modellhaften Registermaschine; grundsätzlicher Befehlszyklus
- Zustandsübergänge der Registermaschine als Wirkung von Befehlen
- Umsetzung von Wiederholungen und bedingten Anweisungen auf Maschinenebene

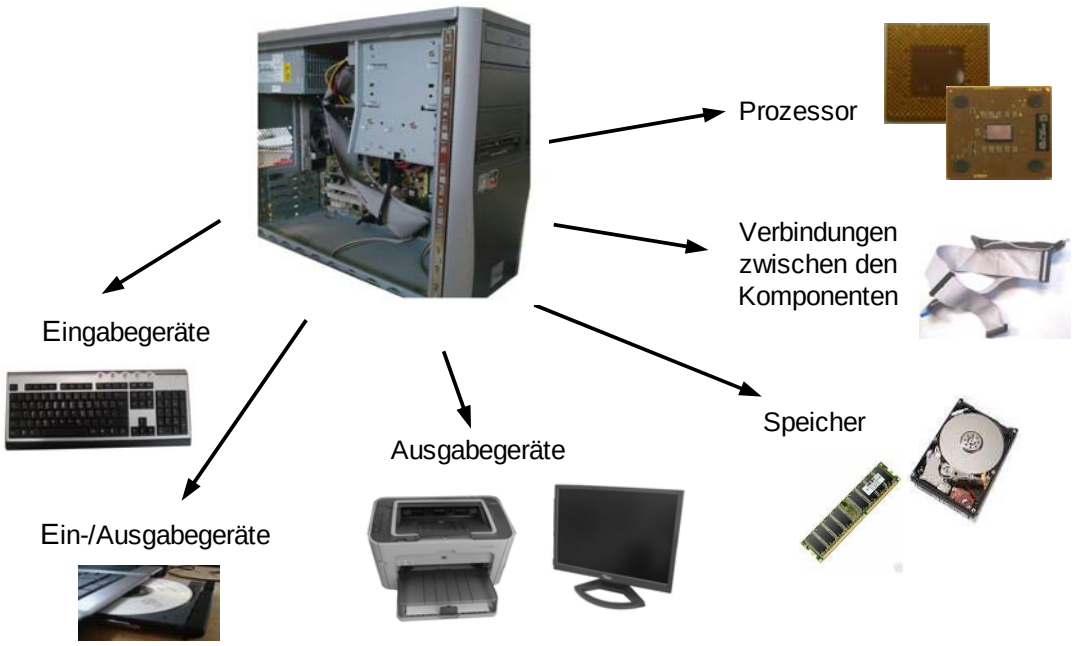
Hinweis: Die Lösungshinweise bzw. die Lösungen zu Aufgaben dieses Kapitels finden sich im entsprechenden Unterordner der Begleit-CD.

3.1 Einstiegs- und Motivationsszenarios

Um das „Innere“ eines Rechners zu erkunden, ist es nahe liegend, ein echtes Rechnersystem zu analysieren. Dabei bieten sich beispielsweise folgende Möglichkeiten – auch in Kombination – an:

3.1.1 Einstieg 1: Praktisches Zerlegen eines Rechnersystems

Die Begutachtung und Sortierung der Bauteile in Gruppen führt die grundlegenden Bestandteile eines Rechners und deren grundlegendes Zusammenwirken plastisch vor Augen.



Klar wird hier insbesondere, dass es Komponenten gibt, die mehrere Funktionen übernehmen, beispielsweise die Festplatte als Speicher, als Eingabe- und als Ausgabegerät.

3.1.2 Einstieg 2: Analyse einer Werbeanzeige

Auch die Analyse einer Werbeanzeige führt auf die typischen Bestandteile eines modernen Rechners. Hier wird ein Schwerpunkt naturgemäß auch auf die technischen Daten eines PCs gelenkt. Diese sollten nicht vertieft werden, ein grundlegendes Ansprechen zentraler Aspekte, wie beispielsweise Größenangaben bei Speichermedien, ist aber durchaus sinnvoll.

3.1.3 Einstieg 3: Wie rechnen Kinder 4 + 3?

Ein auf höherer Ebene einfach erscheinendes Problem kann auf niedriger Ebene doch mit einem größeren Aufwand verbunden sein.

Eine erste Sensibilisierung der Schülerinnen und Schüler für den „Aufwand“ bei der Umsetzung einer offensichtlich trivial erscheinenden Problemlösung auf Maschinenebene kann beispielsweise durch die Fragestellung „Wie rechnen Kinder 4 + 3?“ versucht werden. Diese Problemstellung kann sicher mit den Schülerinnen und Schüler aufgrund ihrer Erfahrung, sei es beispielsweise durch Erinnerung an die eigene Kindheit oder Beobachtungen an jüngeren Geschwistern oder Kindern des Bekanntenkreises, gut diskutiert werden.

Vorschulkinder wenden oft das so genannte „gezählte Rechnen“ an. Dabei handelt es sich um Strategien, einfache Rechenprobleme unter Zuhilfenahme der Finger zu lösen.

Sollen beispielsweise Kinder die Aufgabe 4 + 3 lösen, so nutzen sie oft das so genannte Weiterzählen: Dabei werden zuerst vier Finger (1. Summand) gezeigt und dann sukzessive 3 Finger dazu genommen. Dabei wird meist mitgezählt: 5, 6, 7.

Diese Strategie erinnert an folgenden Algorithmus:

```
a = 4
b = 3
wiederhole solange b > 0
        erhöhe den Wert von a um 1
        erniedrige den Wert von b um 1
endewiederhole
```

Grundsätzlich kann man damit aber zeigen, dass die (eigentlich einfache) Rechenaufgabe durch Anwenden einfacherer Operationen, hier in der Form vom „+1“ und „-1“, gelöst wird.

Hinweis: Das obige Beispiel kann auch einen anderen, mehr technisch orientierten Aspekt vor Augen führen. Ein wichtiger Punkt bei der Gestaltung eines Rechners, insbesondere des Prozessors (siehe Kapitel 3.3), ist die Schnelligkeit. Bei Untersuchungen in den 70er Jahren stellte sich u.a. heraus, dass etwa 80% aller Berechnungen einer typischen Anwendung mit etwa 20% der im Prozessor vorhandenen Befehle ausgeführt werden. Nahe liegend ist es deshalb u.a., den Befehlssatz eines Prozessors möglichst gering zu halten. Dies wird in der so genannte RISC-Architektur (Reduced Instruction Set Computing) ausgenutzt. Bei einem reduzierten Befehlssatz kann ein Befehl nicht durch ein Mikroprogramm, sondern durch einen Mikrobefehl (also sozusagen fest verdrahtet) ausgeführt werden. Dies führt zu großen Geschwindigkeitsvorteilen. Es stellt sich dabei aber sofort die Frage, wie man kompliziertere Befehle dann mithilfe des reduzierten Befehlssatzes „simulieren“ kann. Der gerade vorgestellte Algorithmus macht das an einem einfachen Beispiel klar. Die Addition zweier Zahlen kann mithilfe der einfacheren Anweisungen „erhöhe Wert um 1“ bzw. „erniedrige Wert um 1“ ausgedrückt werden.

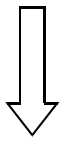
3.2 Vorschlag bezüglich der Themenreihenfolge

Ausgehend von den Einstiegs- und Motivationsszenarios, insbesondere 3.1.1 und 3.1.2, bietet sich folgende Vorgehensweise an:

1. Aufbau eines (modernen) Computersystems mit Schwerpunkt auf die Komponenten

Abstraktion der Komponenten

2. Von-Neumann-Architektur



3. Registermaschine als Modell eines datenverarbeitenden Systems

4. Anwendung der Registermaschine:

Umsetzung einfacher Algorithmen auf maschinennaher Ebene

3.3 Aufbau eines (modernen) Computersystems mit Schwerpunkt auf den Komponenten

Lp: Grundlegende Kenntnisse über den Aufbau eines Rechners und seiner prinzipiellen Funktionsweise helfen den Schülern, den Kommunikationsvorgang mit einer Maschine besser zu verstehen.

Bisher nutzten Schülerinnen und Schüler den Rechner als Black Box, mit dem Information dargestellt, verarbeitet und ggf. auch transportiert werden kann. Die Kommunikation zwischen Nutzer und Maschine war dabei mithilfe auf die Problemstellung zugeschnittener Werkzeuge möglich. Letztendlich wurde aber die eigentliche Umsetzung dieser Kommunikation nicht

hinterfragt. In engem Zusammenhang mit Kommunikation steht dabei auch die Verarbeitung der Information, die dem Rechner übergeben wird.

Ein erster Schritt zum Verständnis des Kommunikationsvorgangs mit einer Maschine sowie der Verarbeitung der durch Kommunikation übermittelten Information kann darin bestehen, in das Innere eines Rechners zu sehen, um einen grundlegenden Überblick über die Schnittstelle Mensch - Maschine und die rechnerinterne Informationsverarbeitung zu erhalten.

3.3.2 Inhaltliche Grundlagen

Im Folgenden wird ein Überblick über die im Rahmen dieses Abschnitts zu vermittelnden inhaltlichen Kenntnisse gegeben.

Hinweis: Insbesondere bei den Begrifflichkeiten gibt es in der Literatur unterschiedliche Interpretationen, auf die ebenfalls kurz eingegangen wird.

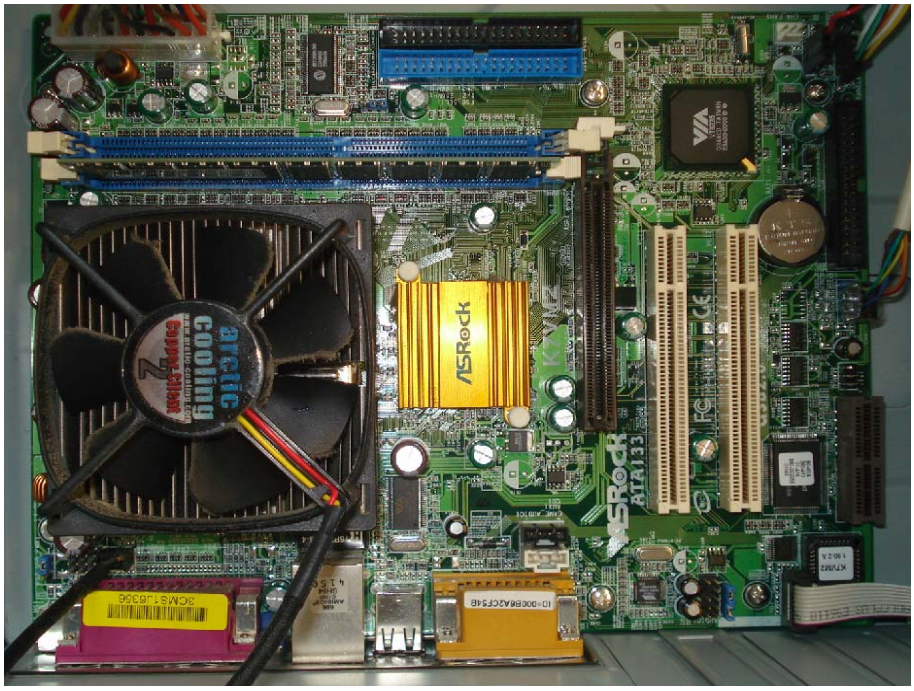
Ein **Computersystem** besteht grundsätzlich aus der **Zentraleinheit** und den **Peripheriegeräten,** kurz Peripherie genannt**.**

Hinweis: Sehr oft wird anstatt Zentraleinheit auch der Begriff Rechner verwendet. Diese Bezeichnung ist etwas irreführend, weil man sich unter einem Rechner das eigentliche Gerät vorstellt, in das bereits einige typische Peripheriegeräte, wie Festplatte oder DVD-Laufwerk, integriert sind.

Zur **Peripherie** gehören die Komponenten, die an die Zentraleinheit angeschlossen werden können. Man unterscheidet prinzipiell drei Gruppen:

1. Eingabegeräte, beispielsweise Tastatur, Maus, Scanner und Mikrophon;
2. Ausgabegeräte, wie Bildschirm, Drucker und Lautsprecher;
3. Ein-/Ausgabegeräte, beispielsweise Festplattenlaufwerke, optische Laufwerke (CD, DVD), Modems und Netzwerkkarten.

Die **Zentraleinheit** befindet sich im Prinzip auf der Hauptplatine (auch Mainboard oder Motherboard bezeichnet) eines Rechners.



typisches Motherboard

Diese Einheit umfasst

1. Mikroprozessor (CPU = Central Processing Unit)

   Er ist das Kernstück des Rechners und dient zur Verarbeitung von Daten, die sich im Speicher befinden. Letztendlich ist er damit für die Ausführung der Programme sowie die Steuerung und Verwaltung der Hardware verantwortlich.

2. (Haupt-) Speicher (Arbeitsspeicher, RAM = Random Access Memory)

   Dieser enthält die auszuführenden Programme bzw. Programmteile sowie die Daten, die gerade benötigt werden. Beim Ausschalten des Rechners gehen die in diesem Speicher befindlichen Daten verloren.

   Hinweis: Sollen Daten dauerhaft gespeichert werden, so müssen sie auf geeigneten Peripheriegeräten, beispielsweise Festplatte oder USB-Stick, abgelegt werden.

3. Bus (Systembus)

   Dieser verbindet die einzelnen Funktionseinheiten der Zentraleinheit sowie die Schnittstellen zur Ein- und Ausgabe und ermöglicht damit grundsätzlich die Kommunikation.

4. Schnittstellen

   Diese ermöglichen (in Form von Buchsen) den Anschluss der Peripheriegeräte. Der Zugang zum Bus wird dabei durch so genannte Controller (Steuerungselektronik) geregelt.

5. ROM-Speicher (ROM = Read-only-Memory, BootROM, bei (älteren) IBM-PCs oft auch noch BIOS genannt)

   Dieser (nur lesbare) Speicher enthält elementare Programme, die beim Einschalten des Rechners automatisch ausgeführt werden. Dazu gehören insbesondere die Überprüfung der Hardwarekomponenten sowie das Laden des Betriebssystems.

Hinweise:

– Der Begriff Zentraleinheit findet sich in der Literatur in unterschiedlicher Ausprägung. Neben der oben aufgeführten Darstellung wird unter der Zentraleinheit oft nur die Kombination Steuerwerk/Leitwerk oder Steuerwerk/Leitwerk/Speicher verstanden (siehe auch Kapitel 3.4 Von-Neumann-Architektur).

– Seit 2008 ist mit EFI (Extensible Firmware Interface) eine BIOS-Weiterentwicklung auf dem Markt, die sich langsam durchsetzt. Es unterstützt 64-Bit-Systeme und ist damit den Anforderungen der neuen Hardware besser gewachsen.

3.3.3 Didaktische Überlegungen, möglicher Unterrichtsablauf

Unter Nutzung von Einstieg 1 und/oder Einstieg 2 (siehe Kapitel 3.1) ist eine zügige Behandlung der Thematik möglich. Insbesondere der erste Einstiegsvorschlag, also die Untersuchung eines realen Rechnersystems, eignet sich gut für entdeckenden Unterricht.

Eine vertiefende Betrachtung der Bestandteile der Zentraleinheit, insbesondere von Mikroprozessor, Hauptspeicher und Bus, ist an dieser Stelle noch nicht sinnvoll. Der strukturelle Aufbau eines Speichers wird bei Behandlung der von-Neumann-Architektur ausreichend behandelt. Die grundsätzliche Struktur eines Mikroprozessors sowie dessen Arbeitsabläufe, insbesondere im Zusammenwirken mit dem Systembus, werden im Rahmen der Beschäftigung mit der Registermaschine schülergerecht aufbereitet.

Die Erwähnung des Caches als Zwischenspeicher, der schneller, aber dafür teurer als der Hauptspeicher ist, ist möglich, insbesondere wenn das untersuchte System einen entsprechenden Baustein aufweist.

Eine zu technische Ausrichtung des Kapitels in Form einer Gerätekunde ist zu vermeiden.

3.4 Von-Neumann-Architektur

Ausgehend von der eher praktischen Ausrichtung in Abschnitt 3.3 wird nun das grundlegende Modell gängiger Rechnerarchitekturen behandelt.

3.4.1 Inhaltliche Grundlagen

John von Neumann (1903-1957) entwickelte 1945 im Rahmen seiner Arbeit „First Draft of a Report on the EDVAC“ (EDVAC: Electronic Discrete Variable Automatic Computer) auf theoretisch-mathematischer Ebene eine grundlegende Architektur für ein Rechnermodell, auf das sich fast alle heutigen Computer stützen. Seine seinerzeit neuartige Idee war, dass Programme nicht mehr fest vorgegeben sind (beispielsweise durch Verdrahtung oder Einlesen von Lochkarten), sondern sich ebenso wie die zu bearbeitenden Daten im Speicher des Rechners befinden. Damit wird die Struktur des Rechners unabhängig von dem zu bearbeitenden Problem, was – beispielsweise im Gegensatz zu einem einfachen Taschenrechner, der nur mit vorgegebenen Rechenoperationen arbeiten kann – zu einer Art Universalrechner führt. Dieses Computermodell ist heutzutage unter dem Namen **von-Neumann-Rechner** oder **von-Neumann-Architektur** bekannt.

Der von-Neumann-Rechner basiert auf folgenden wesentlichen Prinzipien:

1. Der Rechner besteht aus fünf Grundeinheiten:
   - Rechenwerk (ALU - Arithmetic Logic Unit)

     Dieses führt einfache arithmetische und logische Operationen auf den Daten durch.
   - Steuerwerk (Leitwerk, Control Unit)

     Als Herz des Rechners ist es zuständig für die Koordination und Steuerung der Komponenten. Seine Hauptaufgabe besteht in der Bearbeitung der Anweisungen des auszuführenden Programms.
   - Speicherwerk (Speicher)

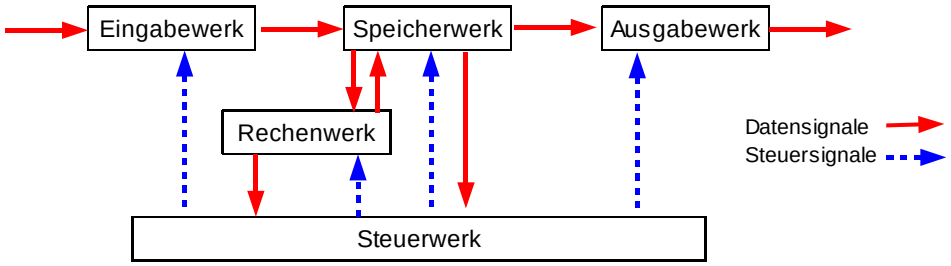
     Darin werden sowohl die Programme als auch die (zu bearbeitenden) Daten abgespeichert.
   - Eingabewerk

     Damit wird die Eingabe von Programmen bzw. Daten in den Speicher ermöglicht und gesteuert.
   - Ausgabewerk

     Dieses ist für die gesteuerte Ausgabe von Daten aus dem Speicher nach außen zuständig.
2. Der Speicher ist in gleichgroße Zellen unterteilt, die fortlaufend nummeriert sind. Über die Nummer, genannt Adresse, kann man auf den Inhalt dieser Speicherzelle zugreifen.
3. Programme und Daten sind binär codiert. Sie werden in demselben Speicher abgelegt.
4. Aufeinander folgende Befehle eines Programms werden in aufeinander folgenden Speicherzellen abgelegt und nacheinander abgearbeitet. Mithilfe von Sprungbefehlen kann jedoch von der linearen Bearbeitungsfolge abgewichen werden.

5. Die elementaren Operationen sind Befehle, darunter u.a.
   - arithmetische Befehle, beispielsweise Addieren oder Multiplizieren;
   - logische Befehle, beispielsweise Vergleiche;
   - Transportbefehle, beispielsweise zum Transport von Befehlen und Daten vom Speicher zum Rechenwerk und umgekehrt;
   - Sprungbefehle (mit oder ohne Bedingung).

Das nachfolgende Schaubild visualisiert die Struktur des von-Neumann-Rechners. Die Pfeile für die Steuer- und Datensignale deuten die grundsätzliche Kommunikation zwischen den Komponenten an.

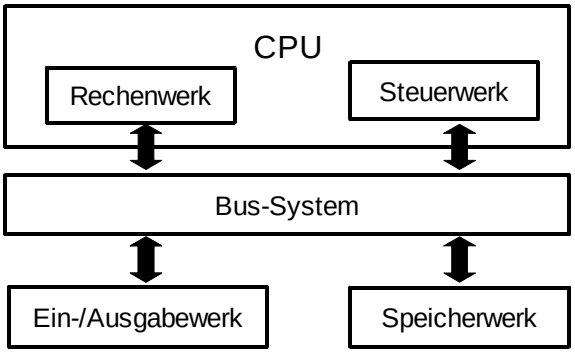


Die Daten, genauer die eigentlichen Daten und die Programme, werden über das Eingabewerk in den Speicher gebracht. Das Steuerwerk ist für die Abarbeitung des im Speicher liegenden Programms verantwortlich. Die Befehle werden dazu vom Steuerwerk schrittweise geholt und interpretiert. Das Rechenwerk führt dann die entsprechenden Anweisungen aus und muss dazu gegebenenfalls auf Daten im Speicher zugreifen. Ergebnisse werden in der Regel wieder in den Speicher geschrieben. Daten des Speichers können dann über das Ausgabewerk nach außen gegeben werden.

Bemerkungen zur Darstellung in der Literatur

1. Es gibt unterschiedliche Strukturdarstellungen des von-Neumann-Rechners. Diese Unterschiede ändern an der Grundstruktur der Architektur nichts. Sie resultieren meist daraus, dass die Von-Neumann-Architektur nur ein theoretisches Modell angibt. Bei der grafischen Darstellung eines Von-Neumann-Rechners werden oft Modifikationen und Weiterentwicklungen, die sich in der praktischen Umsetzung als naheliegend und zweckmäßig erwiesen haben, integriert. Folgende typische Varianten findet man in der Literatur:
   - Eingabewerk und Ausgabewerk werden zum Ein-/Ausgabewerk zusammengefasst. Dies ist in der Praxis vor allem bei den Peripheriegeräten die Regel, bei denen ein wechselseitiger Datenaustausch möglich ist, also bei Speichermedien, Netzwerkkarten usw.
     Das Ein-/Ausgabewerk wird häufig auch als Schnittstelle bezeichnet. Mehr aus technischer Sicht sind auch die Begriffe Steckplatz oder Port (z. B. AGP - Accelerated Graphics Port für Grafikkarten oder USB - Universal Serial Bus) zu finden.
   - Es wird der Begriff der Zentraleinheit eingeführt. Darunter wird eine Einheit verstanden, die Steuerwerk und Rechenwerk (oder oft auch Steuerwerk, Rechenwerk und Speicher) zusammenfasst.
   - Als weitere Grundeinheit wird der Bus (auch Bus-System genannt) eingeführt. Darunter wird ein Verbindungssystem zur Datenübertragung verstanden, auf das alle Komponenten zugreifen können.

Folgendes Schaubild, das man in der Literatur sehr häufig findet, berücksichtigt die eben aufgeführten Punkte:



2. Bei modernen Rechnern befinden sich für gewöhnlich alle Grundeinheiten des von-Neumann-Rechners auf dem Mainboard.
3. Unter einem etwas weiteren Blickwinkel wird oft auch der Rechner selbst, d.h. die Maschine, als von-Neumann-Rechner gesehen. In diesem Zusammenhang kann die Speicherhierarchie Cache – Hauptspeicher (Arbeitsspeicher) – Hintergrundspeicher als Weiterentwicklung des von-Neumann-Konzepts gesehen werden. Der Cache ist ein spezieller Puffer-Speicher, der zwischen dem Hauptspeicher und dem Prozessor liegt und – vereinfacht ausgedrückt – häufig benutzte Daten speichert. Damit können (langsame) Zugriffe auf den Hauptspeicher reduziert werden. Im Vergleich zu Haupt- und Hintergrundspeicher ist der Cache schneller, aber auch teurer.
4. Die Abarbeitung eines Programms erfordert einen ständigen Datenaustausch zwischen Rechen- und Steuerwerk und dem Speicher, der gänzlich über das Verbindungssystem abgewickelt wird. Bei früheren Rechnern waren Rechen- und Leitwerk die langsamsten Komponenten, so dass die Datenbereitstellung durch das Verbindungssystem kein Problem war. Dieser Sachverhalt änderte sich aber durch die Entwicklung immer schnellerer Prozessoren. Das Verbindungssystem wurde nun zum Engpass zwischen der CPU und dem Speicher. Dieser Sachverhalt wird als Von-Neumann-Flaschenhals bezeichnet.

   In der Praxis versucht man dieses Problem durch Einführung von Caches und die hardwaremäßige Unterstützung von Parallelverarbeitung abzuschwächen.

3.4.2 Didaktische Hinweise, möglicher Unterrichtsablauf

Ausgehend von einer Internetrecherche bei Computerherstellern und -anbietern und einer kurzen Analyse von Datenblättern und Rechnerbeschreibungen kann den Schülerinnen und Schülern bewusst gemacht werden, dass der in Abschnitt 3.3 aufgezeigte Rechneraufbau typisch für heutige Systeme ist.

Dies führt zwangsläufig zur Frage, ob modernen Rechnern eine gemeinsame Idee zugrunde liegt. Die Von-Neumann-Architektur liefert die Antwort.

Für die Erarbeitung dieser Architektur bietet sich folgender Weg an:

In einem ersten Schritt befasst man sich mit den Grundbestandteilen des von-Neumann-Rechners. Diese Funktionseinheiten lassen sich schlüssig aus der Analyse moderner Rechner (im Rückblick auf Abschnitt 3.3) abstrahieren:

Bestandteile eines modernen Rechners	Von-Neumann-Rechner
Prozessor	Rechenwerk, Steuerwerk
Schnittstellen mit Controller für Ein-/Ausgabe	Eingabewerk, Ausgabewerk
Speicher	Speicherwerk
Verbindungen zwischen den Komponenten	Signal- und Datenwege bzw. Bus (abhängig von dem Schaubild, das im Unterricht verwendet werden soll)

Hinweis: Rechen- und Steuerwerk lassen sich dabei aus den Aufgaben des Prozessors (Verarbeitung und Steuerung) erschließen.

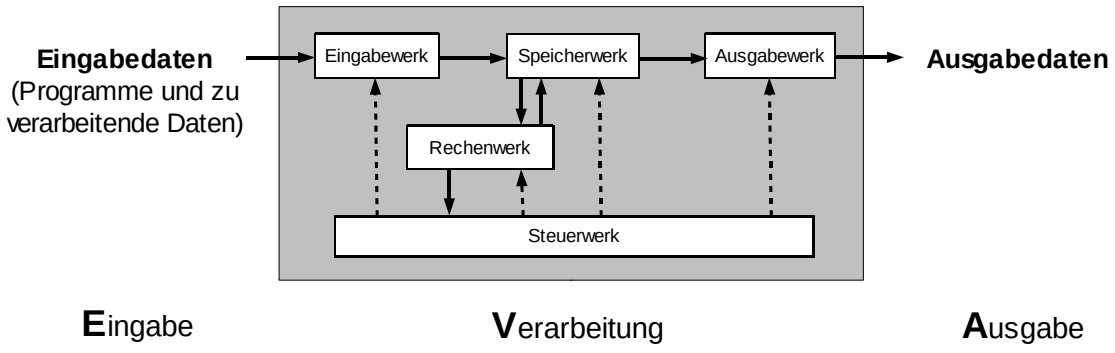
Diese entsprechen, falls man bei den Schnittstellen speziell Ein- und Ausgabe unterscheidet, den fünf typischen Funktionseinheiten des Von-Neumann-Rechners.

Die weiteren Prinzipien kann man beispielsweise in einer kleinen Gruppenarbeit durch die Schülerinnen und Schüler selbst erarbeiten lassen. Dazu werden an die Gruppen Schlagwörter aus der Von-Neumann-Architektur (wie Speicheraufbau, elementare Rechenoperationen, binäre Codierung usw.) verteilt, die von den Teams dann recherchiert und in einem gemeinsamen Plakat, einem elektronischen Dokument o.ä. präsentiert werden.

Die binäre Codierung von Befehlen und Daten erfordert eine zumindest grundlegende Einführung in das Binärsystem sowie in einfache Codierungsprinzipien. Dazu kann auf ein Arbeitsblatt zurückgegriffen werden, das auf der Handreichungs-CD unter Kapitel_3/3.4.2 zur Verfügung steht. Die zugehörigen Lösungen können während der Arbeitsphase zur selbstständigen Ergebniskontrolle im Klassenraum ausgelegt werden.

3.5 Die Registermaschine als Modell eines datenverarbeitenden Systems

Der letzte Abschnitt hat mit der von-Neumann-Architektur eher einen statischen Blick auf einen Rechner geliefert. Zudem ist klar geworden, dass entsprechende Maschinen auf dem EVA-Prinzip basieren:



In einem weiteren Schritt interessiert nun die generelle Verarbeitung von Daten in der Maschine. Dabei ist u. a. die Frage zu klären, wie beispielsweise Programme, die in einer höheren Programmiersprache geschrieben sind, vom binär-orientierten Rechner überhaupt verstanden werden.

Zu diesem Zweck befassen sich die Schülerinnen und Schüler mit einer Registermaschine. Dabei handelt es sich um ein Maschinenmodell, das nach den Prinzipien der von-Neumann-Architektur aufgebaut ist und damit ein grundlegendes Modell für heutige Rechner darstellt.

3.5.1 Inhaltliche Grundlagen

3.5.1.1 Aufbau einer Registermaschine

Registermaschinen finden sich in der Literatur in verschiedensten Varianten. Der Name dieser Maschine weist auf die Bedeutung der Register in diesem Modell hin. Dabei handelt es sich im Originalmodell um die Speicherzellen zur Abspeicherung der Daten. In einer an die Praxis angepassten Variante versteht man darunter – in Abgrenzung zu den eigentlichen Speicherzellen des (Haupt-)Speichers – Hilfsspeicherzellen, die in die Zentraleinheit der Registermaschine integriert sind und auf die ein schneller Zugriff möglich ist.

Für den Unterricht ist eine sehr einfache Registermaschine ausreichend, die im Folgenden umrissen wird.

Die Zentraleinheit der Maschine besteht aus

- dem Rechenwerk
- dem Steuerwerk
- einem Akkumulator (Datenregister), im Folgenden abgekürzt mit AK

  Dabei handelt es sich um ein Register, das eine ganze Zahl speichern kann. Diese Zahl kann einerseits ein Operand für eine Operation des Rechenwerks, andererseits aber auch das Ergebnis einer solchen Rechenoperation sein.
- einem Befehlszählregister (Befehlszähler, Programm Counter), im Folgenden abgekürzt mit BZ

  Diese Hilfsspeicherzelle enthält die Speicheradresse des Befehls, der als nächstes vom Steuerwerk zur Ausführung abgerufen wird.
- einem Befehlsregister, im Folgenden abgekürzt mit BR

  Darin ist der momentan ausgeführte Befehl gespeichert.
- Statusregister (Flagregister), im Folgenden abgekürzt mit SR

  In diesem Register werden bestimmte Informationen über das Ergebnis der jeweils letzten Rechenoperation hinterlegt. Dabei hat jedes Bit des Registers, auch Flag genannt, eine bestimmte Bedeutung. Folgende Flags sind u.a. typisch:

  - N-Flag (Negativ-Flag): Dieses wird gesetzt, wenn das Ergebnis der letzten vom Rechenwerk ausgeführte Rechenoperation negativ war;
  - Z-Flag (Zero-Flag, 0-Flag): Das Bit wird gesetzt, wenn das Ergebnis null war;
  - O-Flag (Overflow-Flag): Im Falle eines Überlaufs wird dieses Flag gesetzt. Ein Überlauf tritt dann auf, wenn das Ergebnis so groß ist, dass es nicht mehr in dem verfügbaren Speicherplatz abgespeichert werden kann.

  Hinweis: Ein Flag ist damit eine binäre Variable, welche als Hilfsmittel zur Kennzeichnung bestimmter Zustände benutzt wird.

Die Befehle und die zu verarbeitenden Daten sind, orientiert an der von-Neumann-Architektur, im selben Speicher abgelegt. Die einzelnen (gleich großen) Speicherzellen enthalten damit entweder

einen Befehl oder (als zu verarbeitende Daten) eine ganze Zahl. Die Speicherzellen sind, beginnend bei 0 oder 1, durchnummeriert. Die Nummer wird als Adresse der Speicherzelle bezeichnet.

Hinweise:

- Über die Größe einer Speicherzelle werden keine Aussagen gemacht. Es wird grundsätzlich davon ausgegangen, dass diese für die Aufnahme des Befehls oder der Zahl ausreichend sind. Insbesondere muss die Überlauf-Problematik, die bei Rechenoperationen auftreten kann, nicht zwingend angesprochen werden.
- Die im Speicher liegenden Daten sind gemäß der von-Neumann-Architektur eigentlich binär codiert. Im Hinblick auf die Nutzung der Registermaschine zur Veranschaulichung der Vorgänge des Programmablaufes werden Befehle und Daten im „Klartext“, angegeben. In diesem Sinne befindet man sich hier auf der Ebene eines Assemblers.
- Bei der Abspeicherung von Befehlen und zu bearbeitenden Daten in einem gemeinsamen Speicher ist folgende Problematik zu beachten: Bei Unachtsamkeit ist es durchaus möglich, dass mit Befehlen Daten überschrieben werden und umgekehrt. Das kann zum Verlust von Daten oder auch zum Programmabsturz führen. In realen Systemen müssen also Mechanismen eingebaut werden, die dieses Überschreiben verhindern. Dies wird in einigen Registermaschinen-Modellen dadurch gelöst, dass man den Speicher für das Programm und den Speicher für die Daten optisch trennt. Bei den Speicherzellen für die Daten wird dann oft allgemein von Datenregistern gesprochen. Prinzipiell besteht aber ansonsten kein Unterschied zu der oben dargestellten Variante des gemeinsamen Speichers. Beide Modelle sind im Kontext des Unterrichts der Jahrgangsstufe 12 gleichwertig.

Die einzelnen Komponenten sind durch ein spezielles Verbindungssystem, dem Bus oder System-Bus, verbunden. Dabei handelt es sich um eine Menge parallel laufender Datenleitungen, auf die jede der daran angeschlossenen Komponenten lesend oder schreibend zugreifen kann. Dabei werden drei Busteile mit unterschiedlichen Aufgaben unterschieden:

1. Datenbus

   Damit werden die Daten zwischen den Komponenten übertragen. Pro Arbeitstakt (siehe Steuerbus) kann der Inhalt einer Speicherzelle übertragen werden.

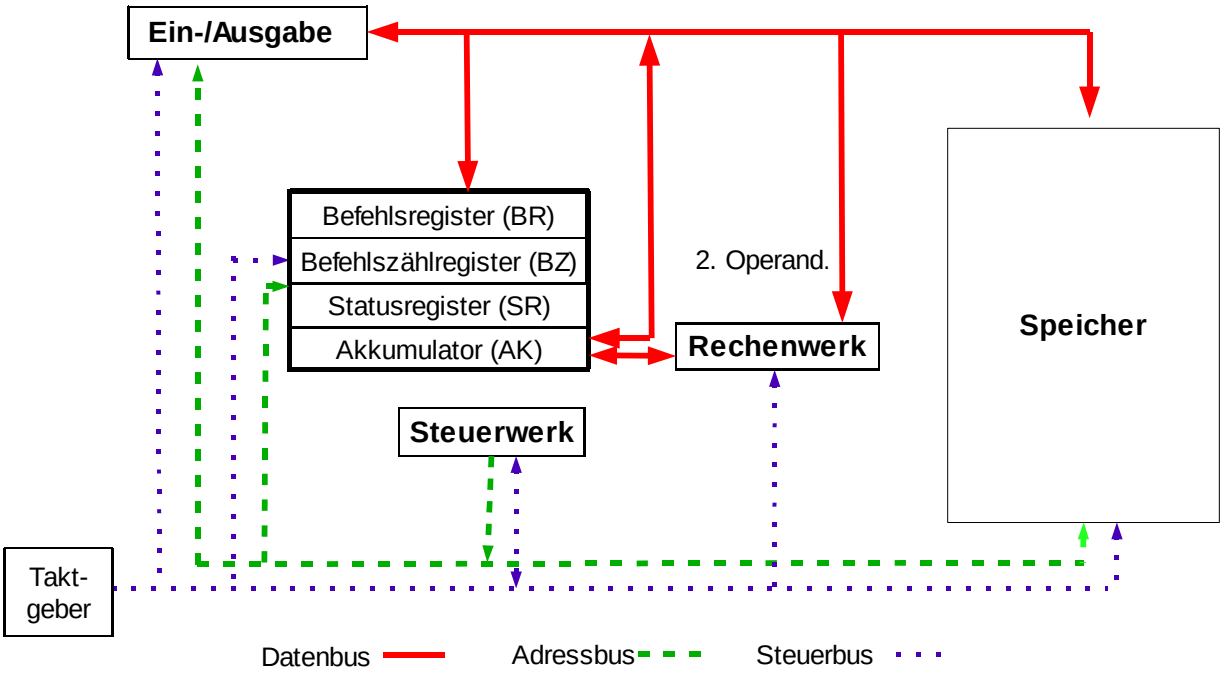
2. Adressbus

   Dieser Bus ist für die Übertragung von Speicheradressen zuständig.

3. Steuerbus

   Über diesen Bus wird das gesamte Bussystem gesteuert. U. a. wird die Lese- und Schreibsteuerung beim Datenbus, also die Richtung der Datenübertragung, festgelegt. Weiterhin wird dieser Bus vom Taktgeber zur Übertragung des Arbeitstaktes genutzt. Durch den Arbeitstakt, der die Anzahl der Arbeitsschritte per Zeiteinheit festlegt, können die einzelnen Komponenten synchronisiert werden.

Die folgende Abbildung zeigt den Aufbau einer Registermaschine im Überblick. Der Bus ist in die drei Bestandteile zerlegt, dabei geben die Pfeile die hauptsächlichen Arbeitsrichtungen an. Aus Übersichtlichkeitsgründen wurden die einzelnen Komponenten des Bussystems grafisch getrennt.



3.5.1.2 Der Befehlssatz der Registermaschine

Jede Registermaschine hat einen festgelegten Befehlssatz. Dieser hängt prinzipiell vom Prozessor bzw. der Registermaschine ab.

Ein Befehl besteht aus einem Operationsteil und einem Operandenteil.

- Der Operationsteil gibt in Form von Befehlsbezeichnungen, wie `add` für addieren oder `load` für (in den Akkumulator) laden, die auszuführende Operation an. Die Befehlsbezeichnungen sind nicht einheitlich definiert, sondern maschinenabhängig. Meist handelt es sich aber um selbsterklärende Namen.
- Der Operandenteil beinhaltet die Operanden als direkte Werte oder Adressen, wo die zugehörigen Operanden zu finden sind. Je nach Art der Maschine kann eine unterschiedliche Anzahl von Adressen angegeben werden. Bei 3-Adress-Befehlen wird beispielsweise die Adresse des ersten Operanden (oder ein konkreter Wert), die Adresse des zweiten Operanden (oder ein konkreter Wert) und die Adresse der Speicherzelle, in der das Ergebnis abgespeichert wird, angegeben.

  Für die Umsetzung im Unterricht reicht eine 1-Adress-Registermaschine vollkommen aus. Die Arbeit mit (maximal) 1-Adress-Befehlen hat insbesondere bei Operationen mit zwei Operanden zur Folge, dass nur ein Operand, genauer gesagt, dessen Speicheradresse oder ein konkreter Wert, angegeben werden kann. Als weiterer Operand wird dann grundsätzlich der Inhalt des Akkumulators verwendet. Dabei wird der Akkumulatorinhalt in fast allen Fällen als erster Operand (d.h. 1. Summand, Minuend usw.) interpretiert.

Der Befehlsname inklusive Operandenadresse wird in einer Speicherzelle abgespeichert.

Hinweis: Die obige Festlegung wird in der Praxis häufig folgendermaßen umgesetzt: Bei einer Speicherzelle, die aus n Bits besteht, werden die ersten m Bits für die Abspeicherung des Befehlsbezeichners, die (n-m) restlichen Bits für den Operanden verwendet. Ein ebenfalls oft umgesetztes Verfahren ist es, für den Befehlsbezeichner sowie für die Adresse des Operanden jeweils eigene Speicherzellen zu verwenden.

Im Folgenden ist ein für den Unterricht sinnvoll eingegrenzter und für alle Lehrplananforderungen ausreichender Befehlssatz, geordnet nach Befehlsgruppen, angegeben; xx steht dabei für die jeweilige Speicheradresse. Mit Ausnahme der Sprungbefehle wird der Befehlszähler nach Ausführung der Ausweisung um einen konstanten Wert erhöht.

Hinweis: Benötigt der Befehl inklusive Operandenteil eine Speicherzelle, so wird der Befehlszähler um 1 erhöht. Sind Operations- und Operandenteil in zwei hintereinander liegenden Speicherzellen abgespeichert, wird der Befehlszähler um 2 erhöht.

Statt der Formulierung "Speicherzelle mit der Speicheradresse xx" wird im Folgenden vereinfacht "Speicherzelle xx" verwendet.

a) Arithmetische Befehle

`add xx` Der Wert aus Speicherzelle xx wird zum Wert im Akkumulator addiert und das Ergebnis im Akkumulator abgelegt.

`sub xx` Der Wert aus Speicherzelle xx wird vom Wert im Akkumulator subtrahiert und das Ergebnis im Akkumulator abgelegt.

`mul xx` Der Wert aus Speicherzelle xx wird zum Wert im Akkumulator multipliziert und und das Ergebnis im Akkumulator abgelegt.

`div xx` Der Wert im Akkumulator wird durch den Wert aus Speicherzelle xx dividiert und das Ergebnis im Akkumulator abgelegt.

In vielen Befehlssätzen finden sich zusätzlich folgende Befehle:

- `inc` und `dec`, die den Wert im Akkumulator um 1 erhöhen bzw. erniedrigen;
- `addi y`, `muli y` usw.: Dabei wird als 2. Operand nicht der Wert aus einer Speicherzelle, sondern die ganze Zahl y verwendet.

b) Transportbefehle

`loadi y` Die ganze Zahl y wird in den Akkumulator geladen.

`load xx` Der Wert aus Speicherzelle xx wird in den Akkumulator geladen.

`store xx` Der Wert aus Akkumulator wird in die Speicherzelle xx geschrieben.

c) Sprungbefehle

Man unterscheidet bedingte und unbedingte Sprungbefehle.

`jump xx` In den Befehlszähler wird xx als Speicheradresse des nächsten auszuführenden Befehls gesetzt.

Bei den (bedingten) Sprungbefehlen muss man prinzipiell zwei verschiedene Ansätze unterscheiden.

Erster Ansatz: Die bedingten Sprungbefehle orientieren sich am Inhalt des Akkumulators.

`jumpz xx` Falls der Wert im Akkumulator 0 ist, wird der Befehlszähler auf xx gesetzt. (z steht für 0 (zero).)

`jumpn xx` Falls der Wert im Akkumulator negativ ist, wird der Befehlszähler auf xx gesetzt. (n steht für negativ.)

`jumpp xx` Falls der Wert im Akkumulator positiv ist, wird der Befehlszähler auf xx gesetzt. (p steht für positiv.)

Optional, aber nicht notwendig, sind weiterhin Sprungbefehle wie beispielsweise `jumpnz xx`.

Dieser Ansatz wird zum Beispiel in der Praxis beim 8051-Assembler umgesetzt. Seine Befehlsliste findet man unter http://www.htwm.de/mcls/deutsch/helpsys/t_as2.htm

Zweiter Ansatz: Die bedingten Sprungbefehle orientieren sich am Inhalt des Statusregisters. Der Inhalt dieses Statusregisters wird nach jeder Operation aktualisiert.

`jumpz xx` Springe zu Speicherzelle xx, falls das Z-Flag gesetzt ist, d.h. das Ergebnis der letzter Operation 0 war.

`jumpn xx` Springe zu Speicherzelle xx, falls nur das N-Flag gesetzt ist, d.h. das Ergebnis der letzten Operation negativ war.

`jmpp xx` Springe zu Speicherzelle xx, falls weder das N- noch das Z-Flag gesetzt ist, d.h. das Ergebnis der letzter Operation positiv war.

Hier sind ebenfalls wieder erweiterte Sprungbefehle vorstellbar, die aber für die Umsetzung im Unterricht nicht notwendig sind.

Dieser Ansatz wird von den meisten heute gängigen Prozessoren umgesetzt, beispielsweise bei dem 8086/88-Assembler (http://www.i8086.de/asm/8086-88-asm.html)

Prinzipiell sind beide Ansätze gleichwertig. Am Programm selbst kann man nicht erkennen, welcher Ansatz der verwendeten Registermaschine zugrunde liegt.

In einigen Befehlssätzen findet man auch den Compare-Befehl `comp xx`. Dabei wird der Wert des Akkumulators mit dem Wert der Zelle mit der Adresse xx verglichen und je nach Ergebnis die Flags gesetzt. Um den Vergleich durchzuführen, wird intern der Wert, der in Speicherzelle xx abgespeichert ist, vom Akkumulatorwert subtrahiert. Der Vorteil dieses Ansatzes ist, dass damit die Sprungbefehle gesteuert werden können, ohne den Inhalt des Akkumulators zu verändern. Dies erspart in vielen Fällen des erneute Laden von Werten in den Akkumulator.

d) Sonstige Befehle

`end` Ende der Programmabarbeitung

Hinweis: Der Befehl `end` hält das Programm an, damit wird also das Ende der Programmabarbeitung festgelegt. In vielen Fällen wird dieser Befehl am Schluss des Programms stehen. Im Kontext bedingter Anweisungen und Wiederholungsanweisungen ist es aber durchaus vorstellbar, dass der Befehl nicht am Ende, sondern innerhalb des Assemblerprogramms auftaucht.

3.5.1.3 Der Befehlszyklus der Registermaschine

Die Abarbeitung der einzelnen Befehle eines Programms orientiert sich an einem festen Schema, das solange wiederholt wird, bis das Programmende, d.h. also das Schlüsselwort `end`, erreicht ist. Man spricht dabei vom Befehlszyklus. Der Prozess der Befehlsverarbeitung bei Von-Neumann-Rechnern wird auch Von-Neumann-Zyklus genannt.

Hinweis: Jeder Schritt des Befehlszyklus (auch oft Phase des Befehlszyklus genannt) benötigt im Allgemeinen einen Takt. Die Taktung wird, wie in Abschnitt 3.5.1.1 beim Systembus angedeutet, durch einen Taktgeber über den Steuerbus vorgegeben.

Schritt

wiederhole bis das Ende der Programmabarbeitung erreicht

1 Befehl aus dem Speicher holen

2 Befehl decodieren und ggf. Operandenadressen errechnen

3 Operanden holen

4 Befehl ausführen und ggf. Ergebnis speichern

5 Befehlszähler erhöhen

endewiederhole

Hinweise:

– Je nach Differenzierung werden unterschiedlich viele Einzelphasen unterschieden. Sehr oft findet man ein 3-Schritt-Modell folgender Art:

  1. Befehl aus Hauptspeicher holen
  2. Befehl decodieren, d.h. auswerten
  3. Befehl ausführen

  Zudem wird oft die Erhöhung des Befehlszählers in einer früheren Phase durchgeführt.

  Weiterhin findet man beim 5-Schritt-Modell oft auch folgende Phasenbezeichnungen:

  1. Fetch
  2. Decode
  3. Fetch operands
  4. Execute
  5. Update instruction pointer

– Wichtig im Hinblick auf den Unterricht ist die Erkenntnis, dass es einen vorgegebenen Befehlszyklus gibt, die genaue Ausprägung ist nicht so sehr entscheidend.

Im Folgenden wird der Befehlszyklus an einem einfachen Beispiel illustriert.

Hinweis: Die Befehle (d.h speziell der Operationsteil wie der Operandenteil) werden nicht im Binärcode, also auf der Ebene der Maschinensprache (siehe Abschnitt 3.5.1.4), sondern im für den Menschen verständlichen Assemblercode angegeben, da das für die grundsätzliche Betrachtung keine Rolle spielt.

Gegeben ist das folgende Programm:

```
1 loadi 5
2 add 11
3 store 12
4 end
```

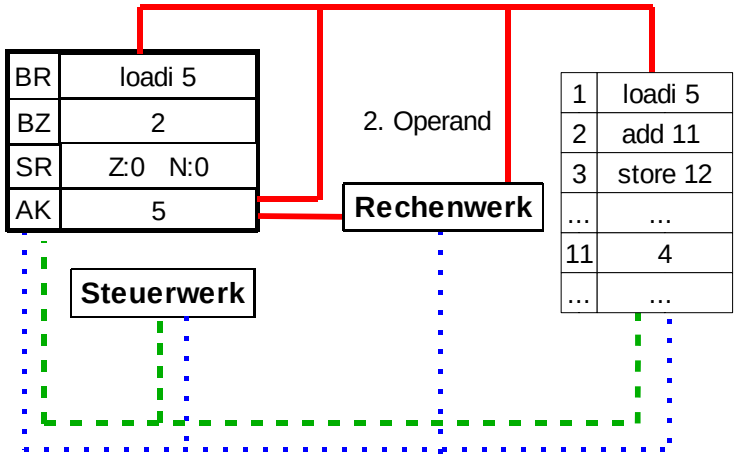
Hinweis: Vor den Befehlen wird hier grundsätzlich jeweils die Adresse der Speicherzelle angegeben, in welcher der nachfolgende Befehl abgespeichert ist. Dies erleichtert in späteren

Kapiteln insbesondere den Umgang mit Sprüngen. Ein zusätzlicher Vorteil für die Diskussion im Unterricht ist die dadurch implizit eingeführte Nummerierung der Programmzeilen.

Nach Abarbeitung des Befehls `loadi 5` liegt beispielshaft folgende Situation vor:

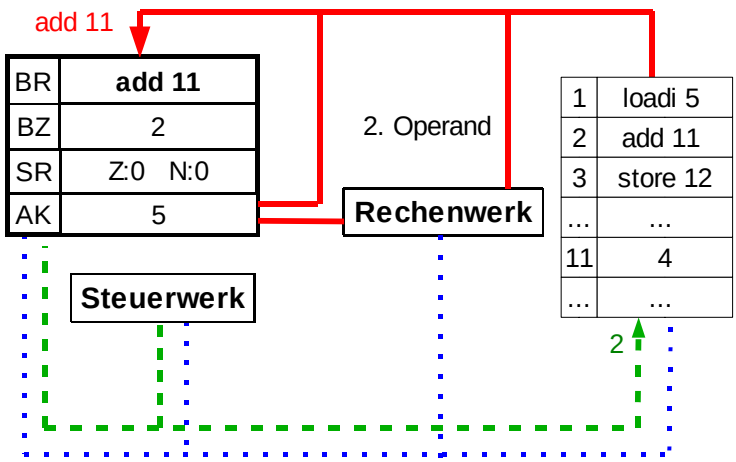
- Das Befehlsregister BR enthält noch den Befehl `loadi 5`.
- Der Befehlszähler BZ enthält die Adresse der Speicherzelle mit dem im nächsten Befehlszyklus auszuführenden Befehl, beispielsweise `add 11`.
- Der Akkumulator AK enthält den Wert 5.
- Die Flags des Flagregisters SR sind gesetzt. Da der Akkumulatorinhalt der letzten Operation 5 ist, wird weder das Zero-Flag noch das Negativ-Flag gesetzt, d.h. der Wert muss positiv sein.

Die nachfolgende Grafik zeigt zusammenfassend die vorliegende Ausgangssituation. (Pfeile werden beim Bus nur noch an den Stellen verwendet, an denen Übertragungen stattfinden.)



Phase 1: Holen des Befehls aus dem Hauptspeicher

Der Befehl, dessen Speicheradresse im Befehlszähler angegeben ist, wird aus dem Speicher geholt. Dazu wird zuerst mittels der im BZ enthaltenen Speicheradresse (hier: 2) über den Adressbus die entsprechende Speicherzelle im Speicher bestimmt. Der betreffende Befehl (hier `add 11`) wird anschließend über den Datenbus in das Befehlsregister (BR) geladen.



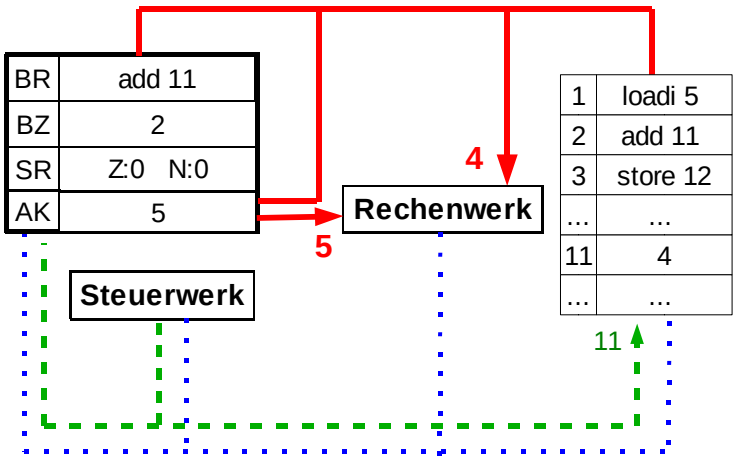
Phase 2: Dekodieren des Befehls

Der Befehl im Befehlsregister wird dekodiert, d.h. das Steuerwerk interpretiert, um welchen Befehl es sich handelt. Gegebenenfalls müssen noch die Operandenadressen berechnet werden, wenn z. B. symbolische Adressierung verwendet wird.

Hinweis: Da numerische Adressen für Menschen umständlich handhabbar sind, werden oft symbolische Adressen verwendet. Es handelt sich hierbei um "sprechende" Namen, die beim Übersetzen in Maschinensprache (siehe Abschnitt 3.5.1.4) automatisch in Adressen umgewandelt werden können.

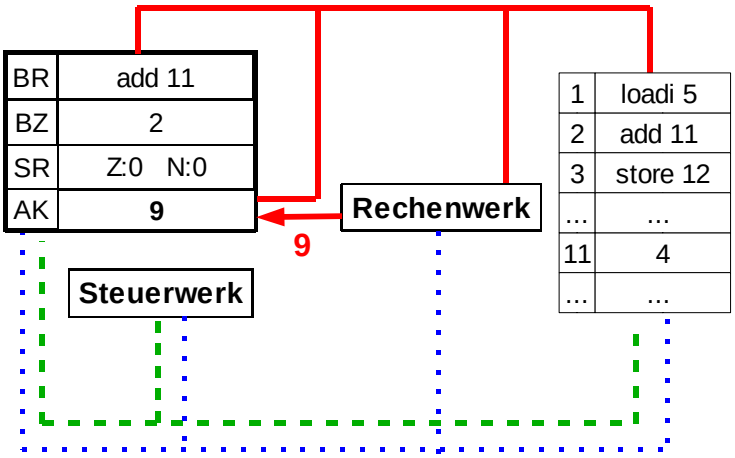
Phase 3: Holen der Operanden

Der erste Operand wird dem Rechenwerk über den Akkumulator zur Verfügung gestellt, die Speicherzelle des zweiten Operanden wird über den Adressbus bestimmt und der Wert dieser Speicherzelle dem Akkumulator als zweiten Operanden über den Datenbus zur Verfügung gestellt.



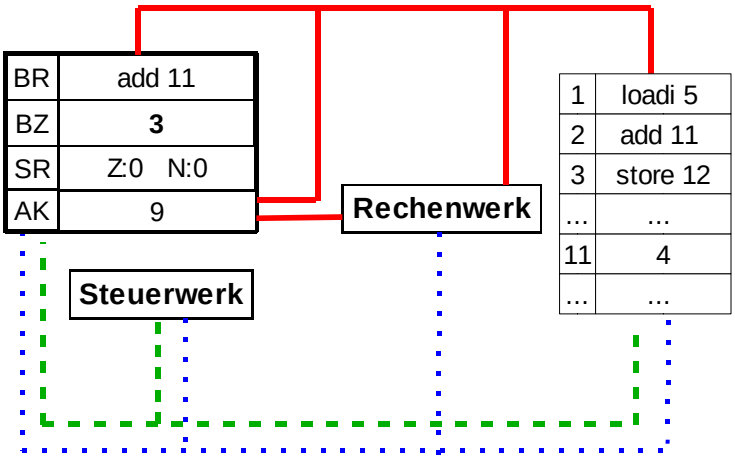
Phase 4: Ausführen des Befehls und Speichern des Ergebnisses

Die Operation, hier die Addition der Werte 5 und 4, wird ausgeführt und das berechnete Ergebnis im Akkumulator abgelegt.



Phase 5: Erhöhen des Befehlszählers

Da kein Sprungbefehl vorliegt, wird der Befehlszähler um 1 erhöht.



Der Befehlszyklus ist abgeschlossen, eine neuer Befehlszyklus wird begonnen. Dieser Vorgang wird solange wiederholt, bis das Ende des Programms erreicht ist.

3.5.1.4 Maschinensprache und Assemblersprache

Da sich die dem Informatikunterricht zugrunde liegende Registermaschine am von-Neumann-Rechner orientiert, sind Befehle und Daten im Speicher eigentlich binär codiert, also als Folge von Nullen oder Einsen abgelegt.

Diese Darstellung ist für den Menschen nicht lesbar, die Maschine kann diese Zeichenfolgen aber interpretieren. Diese Interpretation basiert auf der sogenannten Maschinensprache.

Die Maschinensprache bezeichnet ein System von Instruktionen, die der jeweilige Prozessor direkt ausführen kann. Vereinfacht gesagt sind der Umfang und die Ausprägung des Instruktionssatzes von den im Prozessor umgesetzten Schaltnetzen abhängig. Jede zur Verfügung stehende Instruktion ist durch eine festgelegte Bitfolge von Nullen und Einsen ansprechbar.

Hinweis: Schaltnetze sind ein Teilbereich der technischen Informatik. Eine verständlich aufbereitete Einführung findet man in Form eines E-Learning-Materials der Universität Passau (http://schueler-uni.fim.uni-passau.de).

Zum besseren Umgang mit der Maschinensprache werden die zulässigen Bitfolgen, die Maschinenbefehle, Adressen usw. mit symbolischen, für den Menschen verständlichen Namen belegt. Diese Befehle werden Assemblerbefehle genannt; sie ermöglichen die Implementierung von Assemblerprogrammen. Mit einem speziellen prozessorspezifischen Werkzeug, dem Assembler, kann ein Assemblerprogramm dann meist nahezu eins zu eins in ein entsprechendes Maschinenprogramm umgewandelt werden.

Beispiel: Es sei angenommen, dass für jeden Befehl eine Speicherzelle der Größe 2 Byte zur Verfügung steht. Davon sind beispielsweise die ersten 6 Bit für den Operationsteil, die restlichen 10 Bit für den Operandenteil, d.h. die Zahl oder Adresse, reserviert.

Mit den 6 Bit können $2^6 = 64$ verschiedene Befehle und mit 12 Bit $2^{12} = 4096$ verschiedene Zahlen bzw. Adressen codiert werden.

Jedem zur Verfügung stehenden Maschinenbefehl ist ein entsprechender Assemblerbefehl zugeordnet, beispielsweise:

Maschinenbefehl	Assemblerbefehl
010101	loadi
110111	add
111000	store
111111	end
...	...

Das folgende Assemblerprogramm soll nun in ein Maschinenprogramm übersetzt werden.

```
loadi 5
add 11
store 12
end
```

Dazu werden nur die Befehle gemäß der Codierungstabelle und die Angaben im Operandenteil in Binärzahlen „übersetzt“.

Der Befehl `loadi 5` wird umgewandelt:

- `loadi` ist durch die Folge `010101` codiert;
- Die Zahl 5 lautet in Binärschreibweise `101`; da für den Operandenteil 10 Bits zur Verfügung stehen, werden 7 führende Nullen ergänzt, damit steht `0000000101` für 5.

Der Maschinenbefehl hat demnach die Form: `0101010000000101`. Damit stellt sich das Maschinenprogramm, bei dem der Operationsteil aus Übersichtlichkeitsgründen fett geschrieben ist, insgesamt folgendermaßen dar:

0101010000000101 **110111**0000001011 **111000**0000001100 **111111**0000000000

3.5.2 Didaktische Überlegungen - Einsatz von Simulationssoftware

Lp: Zur Überprüfung ihrer Überlegungen setzen die Schüler eine Simulationssoftware für die Zentraleinheit ein, die die Vorgänge beim Programmablauf veranschaulicht.

In der Mittelstufe werden die im Unterricht entwickelten Modelle grundsätzlich in einem Informatiksystem umgesetzt, um u. a. die Tragfähigkeit des Modells „testen“ zu können. Ebenso ist es nun auch im Rahmen des Lehrplanabschnitts 12.3 zweckmäßig, die Richtigkeit von auf Assemblerebene entwickelten Algorithmen mithilfe einer Simulationssoftware zu überprüfen. Daneben sollten auch die dabei ablaufenden Vorgänge, also beispielsweise die über das Bussystem ablaufende Kommunikation, visualisiert werden können. Im Internet finden sich einige Simulationen, die im Unterricht mehr oder weniger gut eingesetzt werden können. Bei der Auswahl der Simulationssoftware für den Unterricht sind folgende Aspekte zu berücksichtigen:

- Jede Simulationssoftware nutzt einen spezifischen Befehlssatz. Dieser Befehlssatz sollte in etwa den im Abschnitt 3.5.1 vorgestellten Befehlsumfang enthalten.

  Oft stellen die Werkzeuge einen weitaus größeren Befehlssatz zur Verfügung, der auf keinen Fall in seiner gesamten Bandbreite besprochen und ausgenutzt werden sollte. Grundsätzlich ist es im Rahmen von Übungsaufgaben immer möglich, den Befehlssatz zu erweitern oder einzuschränken.

Zwei Beispiele seien dazu angedacht:

- Einschränkung des Befehlssatzes:

  Es wird bewusst auf den Multiplikationsbefehl verzichtet und ein Algorithmus zur Simulation des `mul`-Befehls entworfen. Die Idee ist hier, das Produkt in eine Summe mit lauter gleichen Faktoren umzuwandeln.

- Erweiterung des Befehlssatzes:

  Manche Assemblersprachen stellen einen Befehl (oft unter dem Operationsnamen inc) zur Verfügung, der den Wert des Akkumulators um 1 erhöht. Dieser Befehl erleichtert insbesondere die Umsetzung von Wiederholungsanweisungen. Man könnte beispielsweise zuerst ein Assemblerprogramm entwerfen, das diesen Befehl simuliert, und anschließend den entsprechenden, von der Software zur Verfügung gestellten Befehl für weitere Aufgaben zulassen.

- Eine Simulation auf Assemblerebene ist für das im Unterricht beabsichtigte Grundlagenverständnis durchaus ausreichend. Die Visualisierung der Maschinenebene, also die Darstellung der Befehle und Daten in Binärcode, ist bei der Simulation nicht unbedingt erforderlich. Trotzdem muss den Schülern das Grundprinzip, insbesondere die Abgrenzung von Assembler– und Maschinensprache, bekannt sein.
- Die bei der Abarbeitung eines Befehls ablaufenden Vorgänge zwischen Rechenwerk/Steuerwerk und Speicher sollten geeignet visualisiert werden können.
- Die Möglichkeit eines Moduswechsels (Einzelschritt, Befehl, Nonstop) ist hilfreich. Insbesondere der Einzelschrittmodus erleichtert das Nachvollziehen der doch teilweise komplizierten Vorgänge, insbesondere bei Sprüngen.

Zum Zeitpunkt der Handreichungsdrucklegung war die Menge der verfügbaren Simulationsprogramme relativ klein. Folgende Programme erscheinen für den Unterrichtseinsatz vernünftig bis sehr gut verwendbar:

- MiniMaschine (Downloadmöglichkeit unter www.schule.bayern.de/Bluejprojekte)

  Diese Simulation wurde von StD Albert Wiedemann entwickelt. Die ALU, also das Rechenwerk, ist durch das für die ALU typischerweise verwendete Zeichen **Y** dargestellt, welches die zwei Operandeneingänge und den eindeutigen Ergebnisausgang symbolisiert. Das Statusregister ist umgesetzt.

  Die Simulation arbeitet auf Assemblerebene, zeigt aber auch die codierte Ablage (Dezimalzahl statt Binärzahl) im Speicher. Symbolische Adressierung ist möglich.

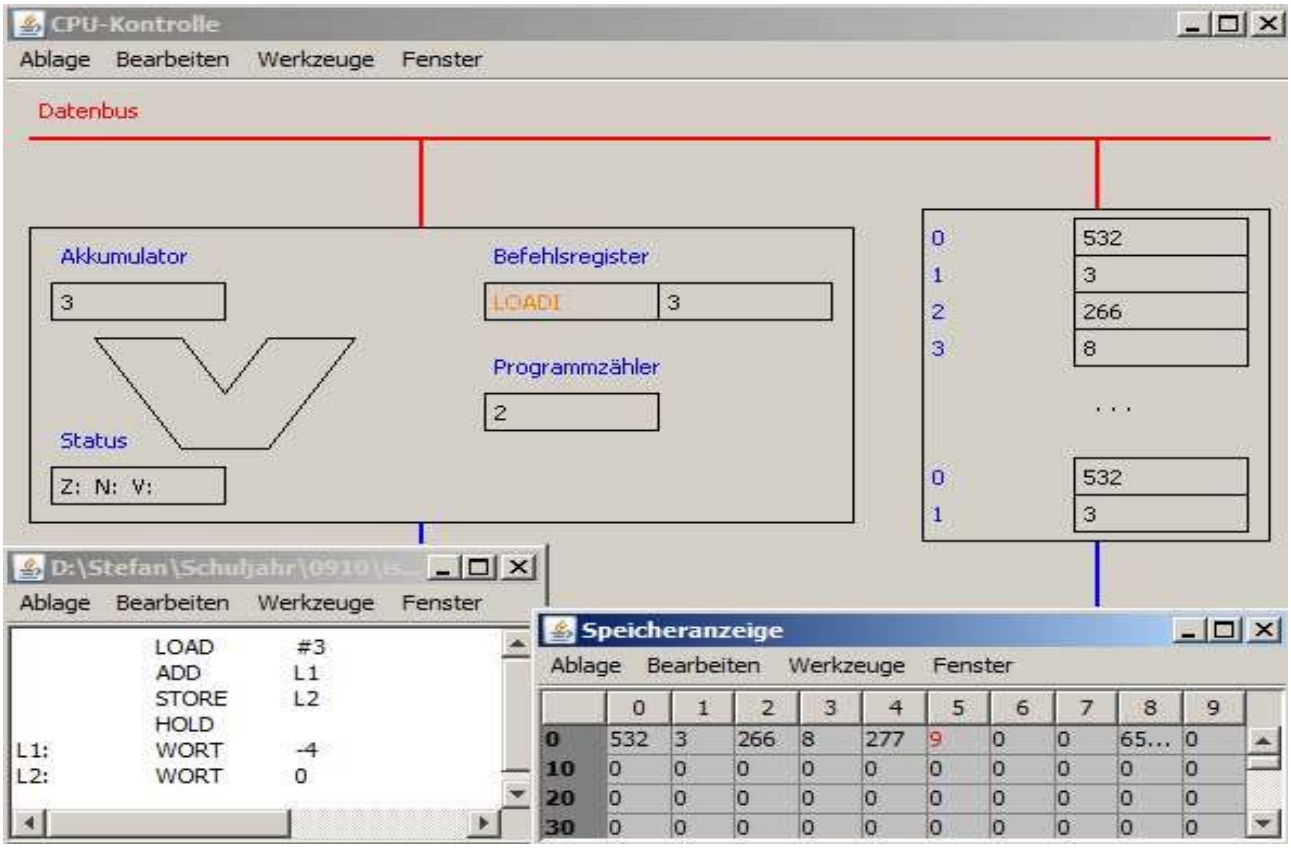
  Der Ablauf beim Befehlszyklus wird visualisiert, dabei wird zwischen Adress- und Datenbus unterschieden.

  Ein Programm wird im integrierten Editor entwickelt und nach dem Assemblieren im Speicher abgelegt. Ein direktes Manipulieren der im Speicher abgelegten Daten ist nicht möglich. Die Programme können gespeichert und ausgedruckt werden. Das Einfügen von Kommentaren ist möglich.

  Es werden 1-Adress-Befehle verwendet. Für den Befehlsbezeichner und für den Operanden wird je eine Speicherzelle verwendet; die Länge des Gesamtbefehls beträgt 2, dementsprechend wird der Befehlszähler bei jeder Durchführung des Befehlszyklus generell um 2 erhöht.

Die Simulationssoftware enthält auch einen Hochsprach-Compiler. Mit diesem kann gezeigt werden, dass eine in einer Hochsprache programmierte Kontrollstruktur automatisiert in ein entsprechendes Assembler- bzw. Maschinenprogramm umgesetzt werden kann.

Das Programm ist in Java geschrieben und damit auf allen gängigen Rechner lauffähig.



– RiSa (Downloadmöglichkeit unter www.gym-vilshofen.de → Fachbereiche → Informatik)

Das Programm wurde von Stefan Wessely im Rahmen der schriftlichen Hausarbeit im Studienseminar Informatik 2008/2010 am Gymnasium Vilshofen entwickelt.

Die Software basiert auf einer Analyse der Vor- und Nachteile von Simulationsprogrammen, die zum Zeitpunkt der Verfassung der Arbeit im WWW verfügbar waren. Dabei werden insbesondere die Schwächen von ZESI (siehe nachfolgender Spiegelstrich) aufgegriffen und weiterentwickelt. Auch RiSa erleichtert die Programmentwicklung durch die Möglichkeit von Quelltextkommentaren.

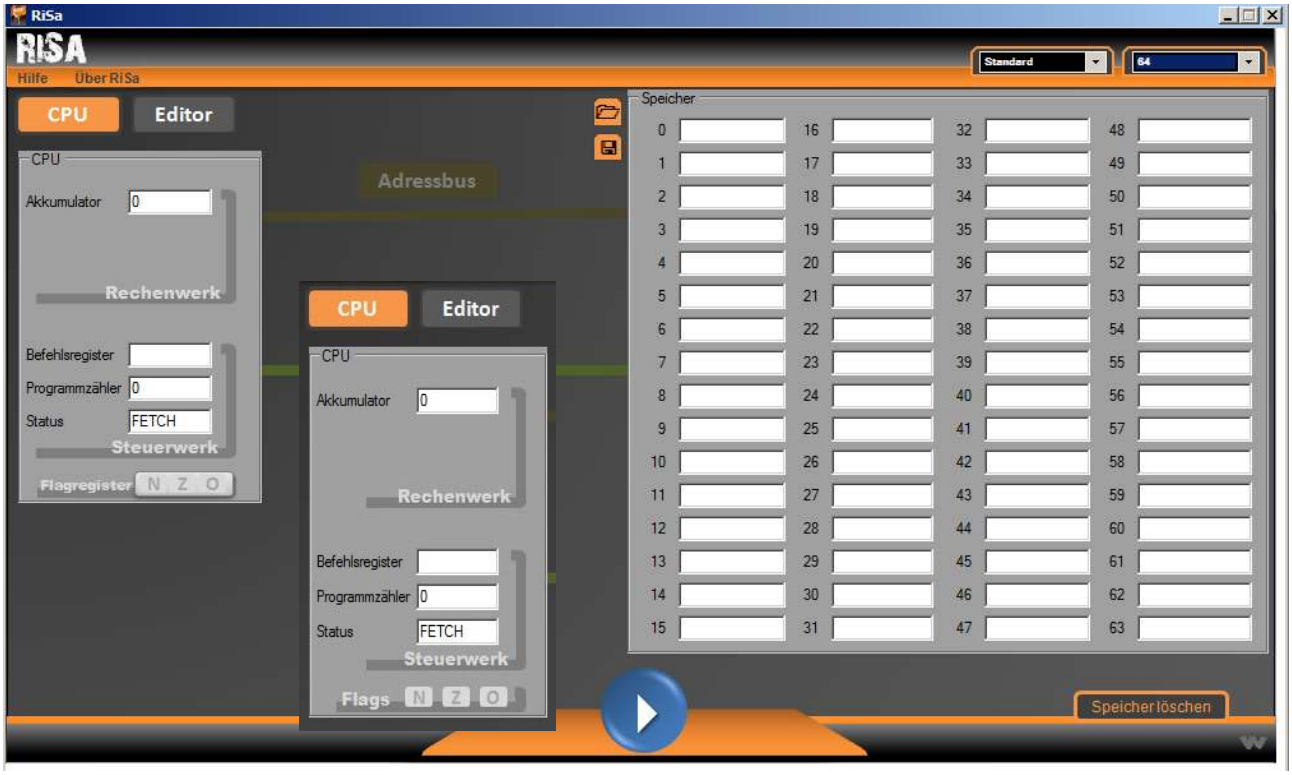
Die Simulation arbeitet auf Assemblerebene, die Ablage im Speicher erfolgt nicht codiert. Symbolische Adressierung ist derzeit noch nicht möglich.

Der Ablauf beim Befehlszyklus wird visualisiert, dabei wird zwischen Adress- und Datenbus unterschieden.

Ein Programm wird im integrierten Editor entwickelt und kann dann in den Speicher zum Programmtest abgelegt werden. Ein direktes Manipulieren der im Speicher abgelegten Daten ist nicht möglich. Die Programme können gespeichert werden. Das Einfügen von Kommentaren ist möglich.

Es werden 1-Adress-Befehle verwendet, mit einem erweiterten Befehlssatz ist auch das Experimentieren mit 2-Adress-Befehlen möglich. Für den Befehlsbezeichner und für den Operanden wird insgesamt eine Speicherzelle verwendet, dementsprechend wird der Befehlszähler bei jeder Durchführung des Befehlszyklus generell um 1 erhöht.

Das Programm ist mit C# geschrieben und derzeit nur auf Windows-Systemen einsetzbar. Zur Drucklegung der Handreichung wurde an einer Portierung auf andere Systeme gearbeitet. Aktuelle Versionen finden sich unter der angegebenen Internetadresse.



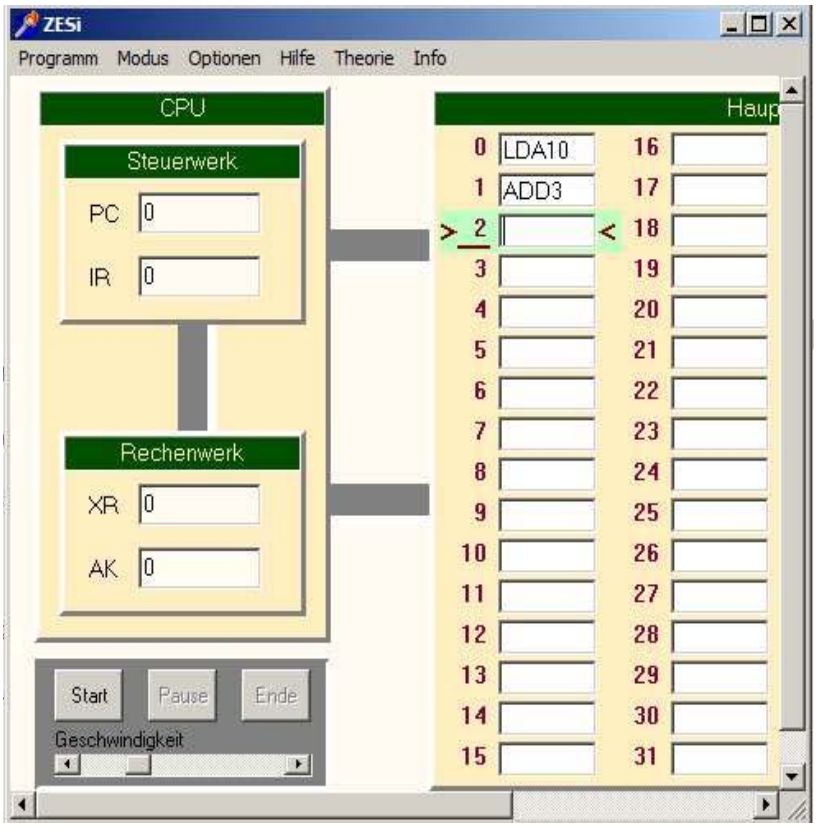
– ZESI - Zentraleinheit-Simulation (Downloadmöglichkeit unter www.zum.de unter dem Stichwort zesi)

  Die Simulation arbeitet auf Assemblerebene, die Ablage im Speicher erfolgt nicht codiert.

  ZESI verzichtet auf das Statusregister. Größtes Manko ist das Fehlen eines unbedingten Sprungbefehls. Eine Notlösung ist hier die Verwendung des Befehls `jxz xx`. Dabei handelt es sich um einen bedingten Sprung, abhängig vom Inhalt eines zusätzlich in der Zentraleinheit verfügbaren Datenregisters XR, das für die Simulation des `jump`-Befehls immer den Wert 0 beinhalten muss. Das Programm wirkt insgesamt noch nicht fertig implementiert, insbesondere ist der `mul`-Befehl nicht verfügbar, obwohl er in der Beschreibung angeführt ist.

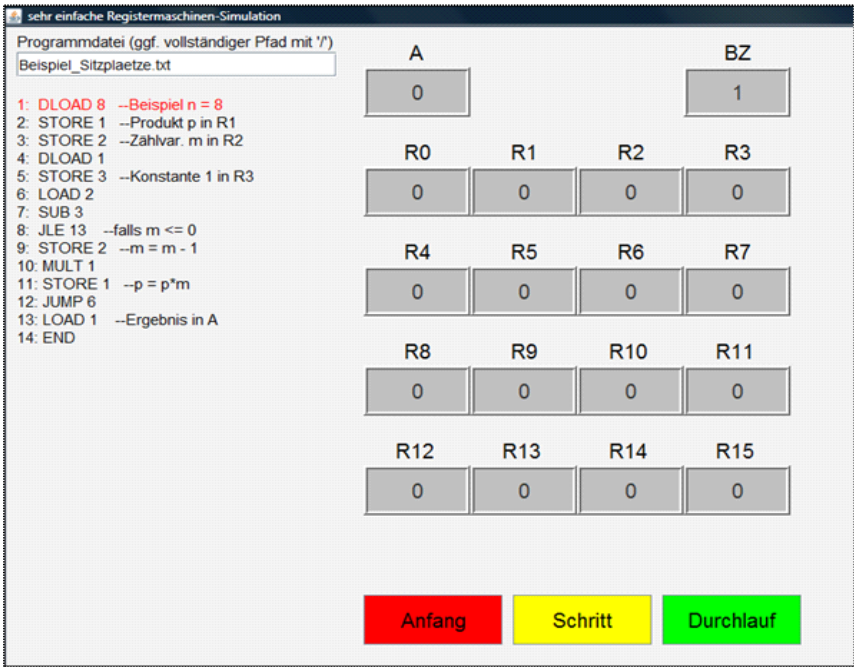
  Der Ablauf beim Befehlszyklus wird visualisiert, dabei wird zwischen Adress- und Datenbus unterschieden. Es wird aber nur ein Bus visualisiert.

  Es gibt keinen eigenen Editor, die Programme werden direkt in die Speicherzellen hineingeschrieben. Die Programme können gespeichert werden. Das Einfügen von Kommentaren ist (damit natürlich) nicht möglich.



– seRmS – Eine **s**ehr **e**infache **R**egister**m**aschinen-**S**imulation (Downloadmöglichkeit unter http://www.ddi.edu.tum.de/schule/materialien/)

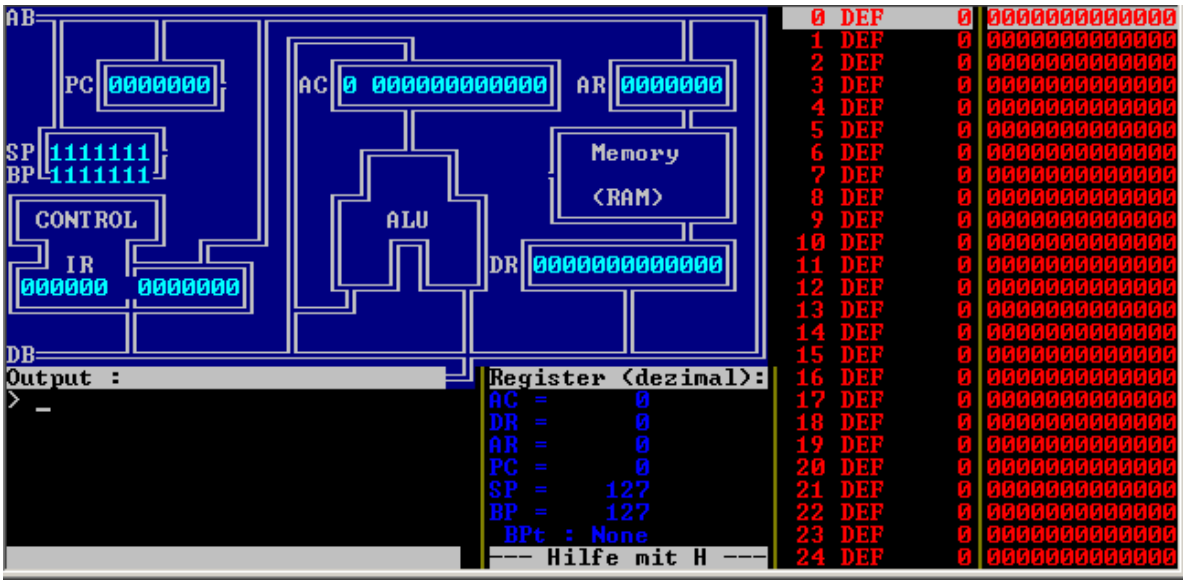
Dieses bewusst sehr einfach gehaltene Simulationsprogramm wurde von StD Ferdinand Winhard (unter Verwendung der JGUI-Toolbox von Hans Witt) entwickelt. Die hier simulierte Registermaschine ist kein Von-Neumann-Rechner. Programm und Daten liegen nicht in einem gemeinsamen Speicher.



Es stehen 16 Register zur Verfügung. Mithilfe dreier Tasten kann das Programm schrittweise oder ganz abgearbeitet und die Maschine auf den Anfangszustand zurückgesetzt werden. Dabei wird jeweils der nächste abzuarbeitende Befehl farbig hervorgehoben.

– DC - Didaktischer Computer (Download unter www.gierhardt.de/informatik/dc/index.html)

Diese Simulation überzeugt durch ihre Funktionalitäten, hat aber das Problem, dass die Benutzeroberfläche eher DOS-orientiert ist und für die Schülerinnen und Schüler aufgrund der fehlenden Mausbedienungsmöglichlichkeiten doch gewöhnungsbedürftig sein dürfte. Vorteilhaft ist aber die umfangreiche Dokumentation, die vom Autor Horst Gierhardt zur Verfügung gestellt wird.



Hinweis: Im Studienseminar Informatik 2008/2010 des Gymnasiums Vilshofen wurden zum Lehrplanpunkt 12.3 zwei Hausarbeiten erstellt, die unter www.gym-vilshofen.de → Fachbereiche → Informatik verfügbar sind. Wie oben schon erwähnt, stellt Stefan Wessely in seiner Hausarbeit das von ihm entwickelte Simulationsprogramm RiSa vor. Thomas Brandl befasst sich mit der Entwicklung einer Aufgabensequenz zur Umsetzung der Kontrollstrukturen in Maschinensprache. Er geht dabei u. a. auch kurz auf die 2009 verfügbaren Simulationsprogramme ein. Beide Arbeiten liefern interessante Ansätze und Ideen.

3.5.3 Möglicher Unterrichtsablauf

Ausgangspunkt für die Betrachtungen kann die eingesetzte Simulationssoftware sein. Nach einer Wiederholung der von-Neumann-Architektur werden in einer ersten oberflächlichen Betrachtung die Bestandteile der Registermaschine analysiert und mit dem Modell des von-Neumann-Rechners abgeglichen. Dabei wird noch nicht detaillierter auf das Bussystem eingegangen.

Ein erstes einfaches Programm wird vorgegeben, beispielsweise:

```
1 loadi 5
2 add 11
3 store 12
4 end
```

Bemerkung: Im Folgenden wird, insbesondere zur übersichtlicheren Umsetzung der bedingten Sprungbefehle, vor jedem Befehl die Adresse der Speicherzelle, die den Befehl enthält, angegeben. In den Simulationsprogrammen ist die Angabe der Adresse in der Regel nicht notwendig.

Dabei ist die Speicherzelle mit der Adresse 11 bereits mit einem Wert vorbelegt. Das Programm wird zuerst im Nonstop-Modus ausgeführt, dann evtl. noch im Einzelschritt-Modus (d.h. jeweils eine Befehlszyklus pro Schritt). Daraus wird die Wirkungsweise des Programms gefolgert. Im vorliegenden Fall werden der Inhalt des Akkumulators und der Inhalt der Speicherzelle 11 addiert und das Ergebnis in der Speicherzelle 12 abgelegt.

Die Schülerinnen und Schüler erhalten eine Befehlsliste mit der zur Verfügung stehenden bzw. für den Unterricht benötigten Befehle mit kurzer Beschreibung. Darauf ist in einer vorausgehenden Erklärung der grundsätzliche Aufbau eines Befehls (Operations- und Operandenteil) kurz dargestellt.

Der Aufbau der Befehlsliste kann sich beispielsweise an folgender Tabelle orientieren:

Befehl	Funktion	gesetzte Flags	Bemerkungen
Rechenbefehle			
`add xx`	addiert zum Wert im Akkumulator den Wert in der Speicherzelle xx	N, Z	
...	...	...	...

Hinweis: Insbesondere die dritte Spalte, also das Setzen des Flagregisters, ist sehr vom verwendeten Simulationsprogramm abhängig, da beispielsweise im Programm ZESI keine Flags gesetzt werden.

Die Ausführung des `add`-Befehls wird im nächsten Schritt im Mikrostep-Modus (Einzeltakt-Modus) untersucht, falls das verwendete Simulationsprogramm dieses Feature zur Verfügung stellt. Dies führt auf die grundsätzliche Struktur des Befehlszyklus. Die Schritte des Befehlszyklus werden fixiert und an einigen Beispielen noch einmal verifiziert.

In einer abschließenden Betrachtung wird auf den grundsätzlichen Unterschied zwischen einem Maschinen- und Assemblerprogramm bzw. einer Maschinen- und Assemblersprache eingegangen. Dabei ist eine wie in Kapitel 3.5.1.4 vorgestellte exemplarische Einführung für das grundlegende Verständnis durchaus ausreichend.

Eine weitere Variante zur Einführung in die Funktionsweise einer Registermaschine, die sich auch durchaus zur Vertiefung eignet, ist die Durchführung eines Rollenspiels, mit dem eine CPU, und damit eine Registermaschine, simuliert werden kann. Einen mögliche Umsetzungsvorschlag findet man auf der Homepage von Horst Gierhardt (www.oberstufeninformatik.de/dc/index.html).

3.6 Anwendung der Registermaschine: Umsetzung einfacher Algorithmen auf maschinennaher Ebene

> Lp: Beispiele zeigen den Schülern, wie einfache Algorithmen auf systemnaher Ebene durch Maschinenbefehle realisiert werden können.

In den Jahrgangsstufen 7 und 10 wurde den Schülerinnen und Schülern bewusst, dass sich die Algorithmik auf relativ wenige zentrale Bausteine (einfache Anweisung, Sequenz, bedingte Anweisung, Wiederholungsanweisung mit Zähler bzw. Bedingung) reduzieren lässt. Diese sind für die Schülerinnen und Schüler relativ leicht fassbar, weil sie auch bei umgangssprachlichen

Ablaufbeschreibungen, wie beispielsweise bei Kochrezepten oder Aufbauanleitungen, fast identisch, oft sogar mit synonymem Wortlaut, verwendet werden. Es stellt sich nun die zentrale Frage, wie diese Bausteine auf maschinennaher Ebene umgesetzt werden. Die Erschließung der Antwort macht den Jugendlichen bewusst, dass dazu im Vergleich zu den Realisierungen in einer höheren Programmsprache ein höherer Aufwand betrieben werden muss. Dies äußert sich beispielsweise in einfachster Weise darin, dass die in Assemblerprogramme umgesetzten Algorithmen meist umfangreicher als die in höhere Programmiersprachen umgesetzten sind.

> Lp: […] wird ihnen auch bewusst, dass Möglichkeiten und Grenzen theoretischer algorithmischer Berechnungsverfahren für die reale maschinelle Verarbeitung von Information ebenfalls gelten.

Die Umsetzung von Algorithmen führt schnell auf die nahe liegende Frage, ob sich auf Maschinenebene im Vergleich zu höheren Programmiersprachen erweiterte Möglichkeiten bei der Formulierung algorithmischer Verfahren bieten. Hier muss den Jugendlichen bewusst gemacht werden, dass dies nicht der Fall ist. Ein konkreter Beweis wird aber nicht durchgeführt.

> Lp: Die prinzipielle Automatisierbarkeit des Übersetzungsvorgangs von einer höheren Programmiersprache in eine Maschinensprache wird ihnen bei der Umsetzung einfacher Algorithmen mit einer maschinennahen Sprache deutlich.

In Lehrplankapitel 12.1 „Formale Sprachen“ hinterfragen die Jugendlichen das erste Mal die Kommunikation zwischen Mensch und Maschine. Dabei wird ihnen deutlich gemacht, dass diese Kommunikation, ähnlich der Kommunikation Mensch – Mensch, Regeln unterworfen ist, die beispielsweise in formalen Sprachen festgeschrieben werden.

In einem zweiten Schritt ist nun am konkreten Beispiel der höheren, für den Menschen gut handhabbaren Programmiersprachen zu überlegen, wie diese Sprachen vom Rechner „verstanden“ werden. Nahe liegend ist, dass das mit einer höheren Programmiersprache geschriebene Programm vom Rechner „erkannt“ und deshalb folglich in eine für die Maschine verständliche Form gebracht werden muss. Dabei ist leicht einsehbar, dass diese Art der Kommunikation nur zweckmäßig ist, wenn der Übersetzungsvorgang automatisiert werden kann.

Da ein Maschinenprogramm, wie in Abschnitt 3.5.1.4 kurz dargestellt, für den Menschen unleserlich ist, wird für die Umsetzung im Unterricht die für die Schülerinnen und Schüler verständlichere Assemblerebene genutzt.

3.6.2 Inhaltliche Überlegungen

3.6.2.1 Anforderungen des Lehrplans

Die Anforderungen des Lehrplans lassen sich wie folgt konkretisieren:

Behandelt werden sollen die grundlegenden Bausteine der Algorithmik. Hier bietet sich eine Orientierung an den entsprechenden Kapiteln der Jahrgangsstufen 7 und 10 an, was zu folgenden Grundbausteinen führt:

- Sequenz;
- bedingte Anweisung (einseitig bzw. zweiseitig bedingte Anweisung (Alternative));
- Wiederholung (mit fester Anzahl bzw. mit Bedingung)

Eine Einschränkung des Befehlsvorrates auf die wesentlichen Grundbefehle, also Rechenoperationen für die vier Grundrechenarten, einfache Sprungbefehle (direkt, mit Bedingung) und einfache Lade- und Speicherbefehle, genügt zur vollständigen Umsetzung der geforderten Algorithmik-Bausteine. Für die nachfolgenden Beispiele wird der in Abschnitt 3.5.1.2 festgelegte

Befehlssatz verwendet. Bei den unbedingten Sprungbefehlen wird dabei der Ansatz verwendet, dass bei Bedingungen der Akkumulatorinhalt überprüft wird.

Die Programmierung findet auf Assemblerebene statt. Die Vorgänge bei der Umsetzung von der Assembler- auf die Maschinenebene sollen kurz erläutert werden, sind aber für das eigentliche Verständnis der Umsetzung von einfachen Algorithmen nicht notwendig.

Die entwickelten Assemblerprogramme müssen mit einer Simulationssoftware getestet werden. Dies gilt insbesondere bei den Umsetzungen der bedingten Anweisung und der Wiederholungsanweisungen.

3.6.2.2 Veranschaulichung durch Zustandsmodelle

In Jahrgangsstufe 10 wurde bereits das Konzept der Zustandsmodellierung erarbeitet. Hierbei wurden Veränderungen von Objekten mithilfe von Zuständen und Übergängen zwischen diesen Zuständen beschrieben. Orientiert an den Vorgaben des Lehrplans ergab sich dabei folgende Vorgehensweise: Aus einer Problemstellung wurde ein Zustandsübergangsdiagramm entwickelt, das anschließend als einfacher Automat implementiert wurde. Grundlage der Implementierung war meist die Einführung eines Attributs *zustand*, dessen Attributwerte die Zustände des Modells repräsentierten. Zustandsübergänge wurden durch Methoden realisiert, die u.a. auch dem Attribut *zustand* den entsprechenden Wert des Zielzustands zuweisen.

Die zustandsorientierte Denkweise kann nun zum Verständnis von maschinennahen Programmabläufen, die auf den ersten Blick nicht einfach zu durchschauen sind, genutzt werden.

Dabei muss nun aber der allgemeine Zustandsbegriff verwendet werden, der den Zustand eines Objekts durch die Gesamtheit der momentanen Werte seiner Attribute (also nicht nur des Werts eines mehr oder weniger künstlich eingeführten Attributs) beschreibt.

In unserem speziellen Fall ist das Objekt die Registermaschine. Der Zustand der Registermaschine ist festgelegt durch den momentanen Inhalt der Speicherzellen und der Register. Eine Änderung des Zustands, also ein Zustandsübergang, wird dabei durch die Ausführung eines Maschinenbefehls bewirkt.

In der Realität ist meist nur ein kleiner Teil der Speicherzellen während eines Programmablaufs von Wertänderungen betroffen. Aus diesem Grund kann man zur Vereinfachung der Notation die Betrachtung der Speicherzellen vernachlässigen, deren Inhalte sich während einer Programmausführung nicht ändern. U.a. spielen deshalb also die Speicherzellen, in denen das eigentliche Programm abgespeichert ist, für den Zustand der Registermaschine keine Rolle. Auf der anderen Seite haben speziell der Akkumulator und der Programmzähler entscheidende Bedeutung für die Festlegung des Zustands. Das Flagregister spielt ggf. bei bedingten Sprüngen (oder bei der Gefahr von Überläufen) eine Rolle; es muss also bei Programmen, die entsprechende Sprünge enthalten, berücksichtigt werden.

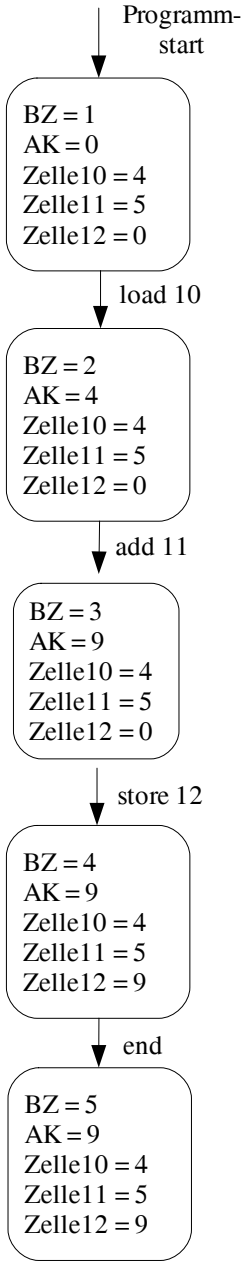
Das Befehlsregister enthält den Befehl, der für den Zustandsübergang verantwortlich ist. Dieser Registerinhalt ist für den eigentlichen Zustand damit eigentlich nicht relevant.

Beispiel: Folgendes Maschinenprogramm addiert zwei Zahlen, die in den Speicherzellen 10 und 11 abgelegt sind. Das Ergebnis befindet sich dann in der Speicherzelle 12.

```
1 load 10
2 add 11
3 store 12
4 end
```

Während der Programmausführung ändern sich lediglich die Inhalte des Akkumulators AK, des Befehlszählers BZ und der Speicherzelle mit der Adresse 12. Da die Inhalte der Speicherzellen 10 und 11 für die Berechnung benötigt werden, können auch diese aus Übersichtlichkeitsgründen mit einbezogen werden. Der Zustand der Registermaschine ist damit durch fünf Werte charakterisiert.

Orientiert an der Zustandsmodellierung der Jahrgangsstufe 10 könnte man damit das nebenstehende Zustandsübergangsdiagramm entwerfen.



Dabei kann man grundsätzlich von folgenden Voraussetzungen ausgehen:

- Den nicht mit konkreten Startwerten versehenen Attributen wird beim Programmstart grundsätzlich der Wert 0 zugewiesen.

  Hinweis: Man kann alternativ durch ein Symbol, beispielsweise x, ausdrücken, dass der momentane Wert unbekannt ist. Dabei muss aber beachtet werden, dass dieser unbekannte Wert beim Ablauf des Programms keine Rolle spielt, also Seiteneffekte ausgeschlossen sind.
- Der Akkumulator wird beim Programmstart auf den Wert 0 gesetzt.
- Die erste Anweisung des Programms befindet sich Speicherzelle 1, der Befehlszähler wird dementsprechend mit 1 initialisiert.

Hinweise:

- Für die Schreibweise der Attribut-Attributwert-Paare wird die in den Jahrgangsstufen 6 und 7 im Rahmen der Objektdiagramme eingeführte Notation *Attributname = Attributwert* verwendet. Das Gleichheitszeichen ist also hier nicht als Zuweisung zu verstehen. Das Verständnis als Zuweisung ist hier nicht zutreffend, da den Attributen, die bei einem Zustandsübergang nicht betroffen sind, die Werte nicht erneut zugewiesen werden.
- Falls man die Verwendung eines Symbols für einen unbekannten Attributwert verwenden möchte, würde man im obigen Beispiel bei dem Attribut *Zelle12*, das den Summenwert aufnimmt, schreiben: *Zelle 12 = x*.
- Beim letzten Übergang, also der Ausführung von `end`, wird nur noch der Befehlszähler aktualisiert. Dieser wird um einen vom Befehlsformat abhängigen Wert, meist 1, erhöht.

Bei einem Programmablauf gibt es immer eine eindeutige Zustandsreihenfolge. Aus diesem Grund können die Zustände und Zustandsübergänge auch übersichtlich in einer Tabelle dargestellt werden. Die 1. Spalte enthält dabei den momentanen Befehl, also die auslösende Aktion für einen Zustandsübergang. Für jede Speicher- bzw. Registerzelle, die von Änderungen zur Laufzeit des Programms betroffenen sind, wird eine eigene Spalte reserviert. In die Tabellenzellen werden dann die Werte der Speicherzellen und Register nach Abarbeitung des in der 1. Spalte angeführten Befehls protokolliert. Das obige Beispiel stellt sich damit folgendermaßen dar:

	BZ	AK	Zelle 10	Zelle 11	Zelle 12
	1	0	4	5	0
`load 10`	2	4	4	5	0
`add 11`	3	9	4	5	0
`store 12`	4	9	4	5	9
`end`	5	9	4	5	9

Zustand vor dem Programmstart

Zustand nach Abarbeitung des Befehls `add11`

3.6.3 Didaktische Überlegungen, möglicher Unterrichtsablauf

Die grundsätzliche Vorgehensweise liegt auf der Hand: Vorgegebene Algorithmen werden in Assemblersprache umgesetzt. Vor allem bei komplizierten Abläufen, insbesondere bei den Wiederholungen, werden Zustandsmodelle als Verständnishilfe genutzt.

Für die Formulierung der Algorithmen bieten sich folgende Varianten an, die in den Übungen variabel verwendet werden sollten:

- Umgangssprache
- Pseudocode
- Struktogramm
- (höhere) Programmiersprache

Im Folgenden wird nun eine mögliche Unterrichtsumsetzung vorgestellt.

Hinweis: Lösungsvorschläge für die Beispiele und Übungen werden auf der Begleit-CD im Ordner Kapitel_3 zur Verfügung gestellt. Bei den Programmieraufgaben werden dabei jeweils Lösungen für die Simulationsprogramme RiSa und MiniMaschine (siehe Abschnitt 3.5.2) angeboten.

3.6.3.1 Sequenz

Eine grundlegende Einführung in die verwendete Simulationssoftware kann im Rahmen der Behandlung von einfachen Sequenzen, hier verstanden als Sequenzen ohne weitere Kontrollstrukturen, stattfinden. Dazu sind nur die Rechen- und Transportbefehle notwendig. Gleichzeitig wird in diesem Abschnitt das aus der Jahrgangsstufe 10 bekannte Zustandsmodell wiederholt und das durch Tabellen dargestellte Zustandsmodell vorgestellt.

Unter Nutzung des Beispiels aus Abschnitt 3.6.2.2 (Addition zweier Zahlen) wäre folgender Ablauf vorstellbar:

Vorgegeben wird die einfache Aufgabenstellung: "Entwickeln Sie ein Assemblerprogramm, das zwei Zahlen addiert, die bereits in den Speicherzellen 10 und 11 abgespeichert sind. Der Wert der Summe soll in Speicherzelle 12 abgespeichert werden."

Die Jugendlichen erarbeiten mithilfe der verwendeten Befehlsliste (siehe Abschnitt 3.5.3) eine Lösung. Dabei sollten in einem ersten Schritt die Werte der Summanden per Hand in die Speicherzellen 10 und 11 eingegeben werden, d.h. es wird bei der Programmentwicklung davon ausgegangen, dass die Speicherzellen bereits mit den gewünschten Werten belegt sind. Dies ermöglicht ein unkompliziertes Testen der Schülerlösungen mit verschiedenen Werten.

Hinweise:

- Ist die direkte Eingabe von Werten in die Speicherzellen nicht möglich, so muss die Belegung der Speicherzellen über entsprechende Lade-Befehle, typischerweise `store xx`, erfolgen. In

diesem Fall kann folgende Aufgabenstellung vorgegeben werden: "Entwickeln Sie ein Assemblerprogramm, das zwei Zahlen addiert. Der 1. Summand mit dem Wert 5 soll zuerst in Speicherzelle 10 und der 2. Summand mit dem Wert 2 in Speicherzelle 11 abgespeichert werden. Nach der Addition beider Wert soll der Wert der Summe soll in Speicherzelle 12 abgespeichert werden."

Alternativ kann eine bereits vorbereitete Datei vorgegeben werden, in der die Speicherbelegung bereits realisiert ist. Diese Sequenz kann dann von den Schülerinnen und Schüler unmittelbar mithilfe der Befehlsübersicht analysiert werden.

– Bei der MiniMaschine können keine Werte direkt in den Speicher eingegeben werden. Man kann jedoch unmittelbar im Anschluss an das Programm im Editor die Datenwerte angeben. Bei der Assemblierung werden für diese Daten sukzessive die Speicherzellen unmittelbar nach dem letzten Befehl des Programms, d.h meist dem codierten Haltebefehl (Code: 0 0), verwendet.

Bei der Diskussion der Lösung stellt sich heraus, dass es sich bei dem Programm um eine Sequenz einfacher Befehle handelt, die durch die Registermaschine nacheinander abgearbeitet werden. Da das sukzessive Abarbeiten von im Speicher hintereinander abgelegten Befehlen die grundlegende Arbeitsweise der Registermaschine (Basis: von-Neumann-Rechner) ist, lässt sich damit die (triviale) Folgerung ziehen, dass der Algorithmik-Baustein „Sequenz“ durch ein Maschinenprogramm grundsätzlich realisierbar ist.

In einem nächsten Schritt wird das auf die Bedürfnisse der Thematik angepasste Zustandsmodell in Form von Tabellen eingeführt. Dabei ist es durchaus empfehlenswert, den in Abschnitt 3.6.2.2 dargestellten Erarbeitungsweg auszunutzen und damit auch wiederholend auf die Zustandsmodellierung der Jahrgangsstufe 10 einzugehen.

	BZ	AK	Zelle 10	Zelle 11	Zelle 12
	1	0	4	5	0
`load 10`	2	4	4	5	0
`add 11`	3	9	4	5	0
`store 12`	4	9	4	5	9
`end`	5	0	4	5	9

Das Programm wird, falls nicht bereits in der Einleitungsphase geschehen, in einer kleinen Sicherungsphase erweitert: "Die Werte der Summanden sollen nun über das Programm in den gewünschten Speicherzellen abgelegt werden." Gleichzeitig wird das Zustandsmodell angepasst.

Bemerkungen:

– Bei den Programmen wird nicht auf eine hinsichtlich der Befehlsanzahl optimierte Variante geachtet. Diese Optimierung ist auch nicht Ziel des Unterrichts. Wo es sich anbietet, kann dies aber durchaus in Übungen als propädeutische Vorbereitung des Lehrplankapitels 12.4 thematisiert werden.

– Bei folgendem Lösungsvorschlag sind die Speicherzellen mit x markiert, deren Inhalt zum Zeitpunkt des Programmstartes nicht bekannt sein müssen.

– Grundsätzlich empfiehlt sich die Verwendung von Kommentaren aus Strukturierungs- und Übersichtlichkeitsgründen auch dann, wenn die Simulationssoftware Kommentare nicht unterstützt.

	BZ	AK	Zelle 10	Zelle 11	Zelle 12
	1	x	x	x	x
// Vorbereitung des Speichers					
`loadi 4`	2	4	x	x	x
`store 10`	3	4	4	x	x
`loadi 5`	4	5	4	x	x
`store 11`	5	5	4	5	x
// Eigentliche Berechnung					
`load 10`	6	4	4	5	x
`...`					

Anhand einiger Übungen wird das Entwickeln einfacher Assemblerprogramme vertieft. Hier bieten sich beispielsweise folgende Aufgaben an.

1. Gegeben sind folgende Terme

   i) x - y;

   ii) x + y + z;

   iii) x - y + z.

   a) Geben Sie jeweils ein Assemblerprogramm zur Berechnung des Terms bei gegebenen Variablenwerten an. Die Variablenwerte sollen am Programmanfang in geeigneten Speicherzellen abgelegt werden, entsprechendes gilt für den berechneten Wert.

   b) Testen Sie Ihr Programm für einige Variablenbelegungen und geben Sie jeweils ein Zustandsmodell an.

2. Implementieren Sie ein Assemblerprogramm, das für vorgegebene Variablenwerte folgenden Term berechnet: v * x + y : z. Beachten Sie hier insbesondere die Regel "Punkt vor Strich".

3. Es soll der Mittelwert zweier Zahlen berechnet werden. Geben Sie ein entsprechendes Assemblerprogramm an.

   Testen Sie Ihr Programm und analysieren Sie insbesondere den Fall, dass der Mittelwert keine ganze Zahl ist.

4. Ein typisches Problem der Programmierung, mit dem Sie vermutlich schon im Informatikunterricht der Jahrgangsstufe 10 konfrontiert wurden, ist das Vertauschen zweier Variablenwerte:

   $$a = 4; b = 6 \quad \Rightarrow \quad a = 6; b = 4$$

   a) Erläutern Sie, warum folgender Algorithmus dieses Problem nicht löst.

   ```
   a = 4
   b = 6
   a = b
   b = a
   ```

   b) Zur Lösung des Problems benötigt man eine Hilfsvariable. Geben Sie in Pseudocode eine Lösung für das Problem unter Ausnutzung einer solchen Hilfsvariablen an.

c) Implementieren Sie Ihren in Teilaufgabe b) entwickelten Algorithmus in einem Assemblerprogramm.

3.6.3.2 Einseitig bedingte Anweisung

Nun folgt die Umsetzung der weiteren Kontrollstrukturen auf Assemblerebene, was im Vergleich zur Sequenz aus Abschnitt 3.6.4.1 einen höheren Aufwand erfordert. Als erste Kontrollstruktur bietet sich die einseitig bedingte Anweisung an, da sie am einfachsten umzusetzen ist. Erforderlich ist nur ein bedingter Sprung, der für folgende Idee eingesetzt wird: Liefert die Überprüfung der Bedingung falsch, so wird die im Falle von wahr auszuführende Sequenz übersprungen.

Ein Problem ist aber, dass das Maschinenprogramm nicht mit Wahrheitswerten umgehen kann, sondern lediglich den Datentyp GANZEZAHL kennt. Deshalb müssen die Schülerinnen und Schüler in einem ersten Schritt dafür sensibilisiert werden, wie man eine Bedingung und deren Auswertung auf Maschinensprachebene umsetzen kann.

Wie in Kapitel 3.5.1.2 dargestellt, gibt es grundsätzlich zwei verschiedene Varianten von bedingten Sprungbefehlen:

1. In der ersten Ausprägung des bedingten Sprungs wird bei der Auswertung der Bedingung der Akkumulatorwert zugrunde gelegt.

   Beispiel: `jumpn 12`: Falls der Akkumulatorwert negativ ist, springe zu Speicherzelle 12.

2. In der zweiten Ausprägung wird das Flagregister ausgewertet.
   Beispiel: `jumpn 12`: Falls das N-Flag gesetzt ist, springe zu Speicherzelle 12.

Für die Umsetzung von Programmen mit Simulationsprogrammen hat das aber keine Auswirkung, da der entscheidende Test intern abgewickelt wird. Man verwendet in beiden Fällen den „gleichen“ Befehl.

Als Einstieg bietet sich hier die Analyse eines vorgegebenen Codefragements an, also das Lernen am konkreten Beispiel.

Vorgehensweise für Sprungvariante 1:

Den Jugendlichen wird eine Einstiegsmotivation präsentiert, beispielsweise:

Im Folgenden sehen Sie, wie eine einseitig bedingte Anweisung (links im Pseudocode formuliert) auf Assemblerebene (rechts) umgesetzt werden kann:

```
wenn
    Wert_Zelle20 >= Wert_Zelle21
dann
    tausche Wert von Zelle 20 und 21
endewenn
```

```
1 load 20
2 sub 21
3 jumpn 10
4 load 20
5 store 22
6 load 21
7 store 20
8 load 22
9 store 21
10 end
```

In einem vorbereitenden Schritt wird erarbeitet, welche Aufgabe das Programm erledigt: Es sortiert die Werte von Zelle 20 und Zelle 21 aufsteigend.

Die Behauptung, dass das Assemblerprogramm die einseitig bedingte Anweisung richtig umsetzt, wird durch Tests mit verschiedenen Zellbelegungen überprüft und als korrekt erkannt.

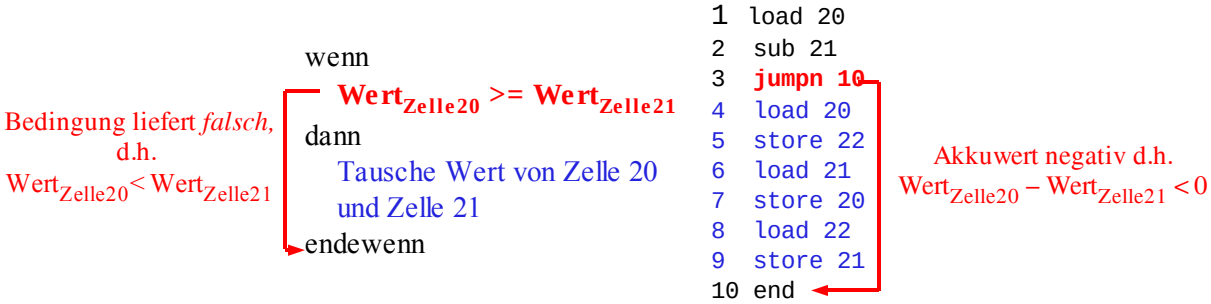
	Belegung vorher	
	Zelle 20	Zelle 21
a)	3	4
b)	3	-1
c)	-2	3
d)	-2	-5

Ausführung des Assemblerprogramms →

Belegung nachher	
Zelle 20	Zelle 21
3	4
-1	3
-2	3
-5	-5

Als erstes Zwischenergebnis ergibt sich: Eine einseitig bedingte Anweisung kann (anscheinend) durch ein Assemblerprogramm umgesetzt werden.

Es wird nun an Hand der vorgegebenen Lösung untersucht, wie das wenn-dann-Konstrukt umgesetzt wird. Dazu wird zuerst die Sequenz, die das Tauschen der Werte bewirkt, identifiziert. Die übrigen Befehle müssen für die Umsetzung der eigentlichen bedingten Anweisung verantwortlich sein. Das Überspringen der Tauschsequenz ist mit dem bedingten Sprung möglich. Der Sprung wird ausgeführt, wenn der Akkumulatorwert, also $Wert_{Zelle20} - Wert_{Zelle21}$, kleiner als 0 ist. Im Fall des Pseudocodes wird die Tauschsequenz übersprungen, wenn die Bedingung *falsch* liefert, d.h. $Wert_{Zelle20} < Wert_{Zelle21}$. Es ist sofort einsichtig, dass die Aussagen $Wert_{Zelle20} - Wert_{Zelle21} < 0$ und $Wert_{Zelle20} < Wert_{Zelle21}$ identische Bedeutung haben (Äquivalenzumformung).



Hinweis: In Abhängigkeit vom Befehlsvorrat der verwendeten Registermaschine gibt es natürlich verschiedene Umsetzungen. Falls der `comp`-Befehl zur Verfügung steht, wird der Aufwand etwas geringer, da man sich das wiederholte Laden von Werten in den Akkumulator (Programmzeile 4) sparen kann. In diesem Fall könnte man das obige Beispiel folgendermaßen umsetzen:

```
1 load 20
2 comp 21
3 jumpn 9
4 store 22
5 load 21
6 store 20
7 load 22
8 store 21
9 end
```

Das Prinzip zur Umsetzung kann schließlich verallgemeinert werden:

Eine Bedingung, speziell ein Vergleich zweier Werte x und y, kann durch ein Assemblerprogramm simuliert werden, indem zunächst die Differenz x – y berechnet wird. Das im Akkumulator befindliche Ergebnis dieser Berechnung wird dann von einem passenden Sprungbefehl ausgewertet.

Vorgehensweise für Sprungvariante 2:

Die Vorgehensweise ist identisch. Der verallgemeinerte Prinzip für eine Bedingung lautet:

Eine Bedingung, speziell ein Vergleich zweier Werte x und y, kann durch ein Assemblerprogramm simuliert werden, indem zunächst die Differenz x – y berechnet wird. Die durch diese Operation im Statusregister gesetzten Flags werden dann von einem passenden Sprungbefehl ausgewertet.

Die Umsetzung der einseitig bedingten Anweisung wird anhand von Aufgaben geübt und vertieft:

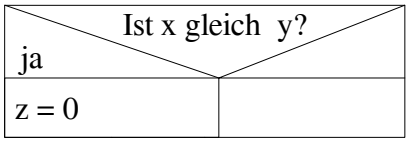
1. Welche der folgenden Assemblerprogramme realisieren die im Einführungsbeispiel geforderte einseitig bedingte Anweisung? Begründen Sie kurz!

a)
```
1 load 20
2 sub 21
3 jmpp 10
4 load 20
5 store 22
6 load 21
7 store 20
8 load 22
9 store 21
10 end
```

b)
```
1 load 21
2 sub 20
3 jumpn 10
4 load 20
5 store 22
6 load 21
7 store 20
8 load 22
9 store 21
10 end
```

c)
```
1 load 21
2 sub 20
3 jmpzn 10
4 load 20
5 store 22
6 load 21
7 store 20
8 load 22
9 store 21
10 end
```

2. Setzen Sie den durch folgendes Struktogramm dargestellten Algorithmus in einem Assemblerprogramm um.



3. Implementieren Sie folgende als Pseudocode gegebene Anweisung in einem Assemblerprogramm.

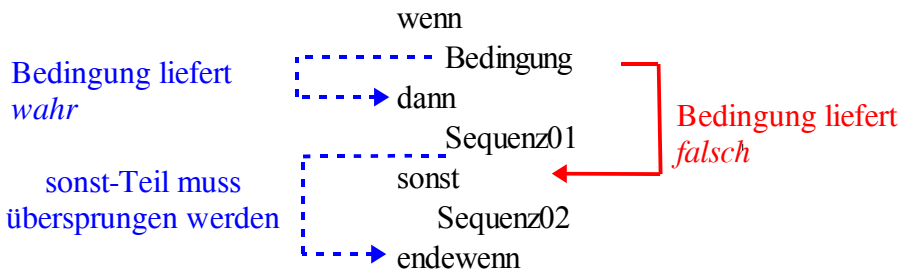
wenn $Wert_{Zelle23}$ <= $Wert_{Zelle24}$
dann weise der Speicherzelle 25 den Wert -1 zu
endewenn

3.6.3.3 Zweiseitig bedingte Anweisung

Hinweis: Bei den im Folgenden kurz skizzierten Vorgehensweisen (3.6.3.3 bis 3.6.3.5) wird zuerst das allgemeine Prinzip der jeweils betrachteten Kontrollstruktur analysiert, weil davon ausgegangen werden kann, dass die Schülerinnen und Schüler aufgrund ihrer Erfahrungen aus den Jahrgangsstufen 7 und 10 die Konzepte bereits gut verstehen. Dann werden die Erkenntnisse an Beispielen angewendet und überprüft. Genauso möglich ist ein induktiver Weg, bei dem ausgehend von einem oder mehreren konkreten Beispielen verallgemeinert wird.

Grundlegend für die Umsetzung der Kontrollstrukturen in ein Maschinenprogramm ist das Verständnis, wie Bedingungen mithilfe von Maschinenbefehlen umgesetzt werden. Dies sollte für die Schülerinnen und Schüler nach dem letzten Abschnitt gut klar geworden sein. Die Umsetzung der zweiseitig bedingten Anweisung ist nun relativ einfach.

Man betrachtet den Pseudocode einer zweiseitig bedingten Anweisung und diskutiert, an welchen Stellen unter welchen Bedingungen Sprünge stattfinden:



Relativ klar ist der Sprung zu *Sequenz01* bzw. *Sequenz02* in Abhängigkeit von der Auswertung der Bedingung.

Nach dem Abarbeiten von *Sequenz01* darf aber *Sequenz02* nicht durchlaufen werden, das Programm muss also auf alle Fälle *Sequenz02* überspringen. Die Aussage „auf alle Fälle“ bedeutet, dass dieser Sprung immer stattfinden muss, also von keiner Bedingung abhängt.

Die Übersetzung in ein Assemblerprogramm gelingt sehr einfach, da der unbedingte Sprung als Befehl zur Verfügung steht. Dieser muss unmittelbar nach *Sequenz01* stehen und auf die erste Speicherzelle nach der Sequenz des sonst-Teils führen.

Die abstrakte Betrachtung wird nun an einem konkreten Beispiel umgesetzt.

Aufgabe: Von zwei verschiedenen Zahlen a und b soll die größere Zahl bestimmt werden.

Ein erster Weg zum Assemblerprogramm führt über die Formulierung der Aufgabenstellung in der wenn-dann-Form, die in Pseudocode notiert wird:

```
wenn a < b
dann
    b ist die größere Zahl
sonst
    a ist die größere Zahl
endewenn
```

Für die Implementierung legt man fest, dass die Zahlen in zwei Speicherzellen, beispielsweise Zelle 20 und 21, abgespeichert wird. Das Ergebnis soll in Speicherzelle 22 abgelegt werden.

Eine kurze Vorüberlegung bereitet die Umsetzung vor: Im Pseudocode erfolgt der Sprung in den sonst-Teil genau dann, wenn die Bedingung $a < b$ falsch liefert. Da die beiden Zahlen als verschieden vorausgesetzt sind, ist dies für für $a > b$, anders ausgedrückt für $a - b > 0$ der Fall. Die Differenz $a - b$ muss also unmittelbar vor dem bedingten Sprung im Akkumulator berechnet werden. Bei positivem Ergebnis soll der Sprung erfolgen, also muss der entsprechende bedingte Sprung, bei unserem Befehlsvorrat also `jumpp`, verwendet werden.

Hinweis: Die gerade gezogenen Folgerungen sind natürlich von den zur Verfügung stehenden Sprungbefehlen abhängig. Ein Anpassung an die jeweiligen Vorgaben sollte aber unproblematisch sein.

Nun ist eine Implementierung leicht möglich:

Lösungsvorschlag	Kommentierung

`1 load 20`
`2 sub 21`
`3 jmpp 7`
`4 load 21`
	im Akku liegt der Wert der Differenz $a - b$
	wenn Akkuwert positiv, springe zu 7

```
5 store 22
6 jump 9          springe zu 9
7 load 20
8 store 22
9 end
```

Hinweis: Auch hier ist eine Umsetzung durch die Verwendung des `comp`-Befehls und der damit verbundenen Flags möglich: `sub 21` wird durch `comp 21` ersetzt. Auf das erneute Laden des Werts aus Speicherzelle 20 (Programmzeile 7) kann dann verzichtet werden.

Über eine Zustandstabelle werden nun mindestens zwei Fälle getestet, um die Richtigkeit der Umsetzung zu überprüfen:

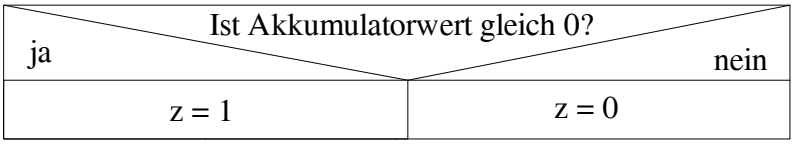
Fall 1: a = 4, b = 5, d.h. a < b

Befehl	BZ	AK	20	21	22
	1	x	4	5	x
`load 20`	2	4	4	5	x
`sub 21`	3	-1	4	5	x
`jmpp 7`	4	-1	4	5	x
`load 21`	5	5	4	5	x
`store 22`	6	5	4	5	5
`jump 9`	9	5	4	5	5
`end`	10	5	4	5	5

Fall 2: a = 5, b = 4, d.h. a > b: analoge Tabelle

Die zweiseitig bedingte Anweisung wird nun durch Übungen vertieft:

1. Die Speicherzelle z soll das Zero-Flag des Statusregisters simulieren. Implementieren Sie ein Assemblerprogramm, mit der man das Setzen dieses Flags gemäß folgender bedingten Anweisung simulieren kann.



2. Implementieren Sie für folgende Problemstellungen ein Assemblerprogramm:

   a) Zwei gegebene Zahlen sollen subtrahiert werden. Ist der Subtrahend größer als der Minuend, so soll das Ergebnis 0 sein.

   b) Zu einer vorgegebenen Zahl a soll der Betrag |a| berechnet werden.

3.6.3.4 Wiederholung mit (Anfangs-)Bedingung

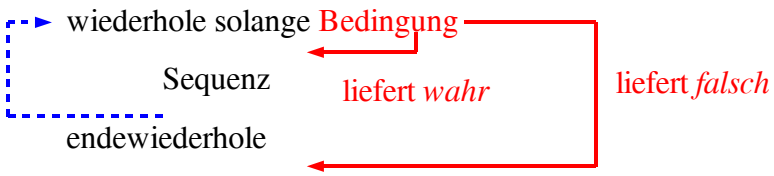
Der Lehrplan verlangt die Umsetzung von Wiederholungen. Explizit ist damit nicht ausgesagt, welche speziellen Typen der Wiederholungsanweisung im Unterricht behandelt werden müssen. Da die Wiederholung mit Zähler sowie die Wiederholung mit Endbedingung prinzipiell durch eine

Wiederholung mit Anfangsbedingung ausgedrückt werden kann, ist es nahe liegend, die Wiederholung mit Anfangsbedingung in den Fokus zu rücken.

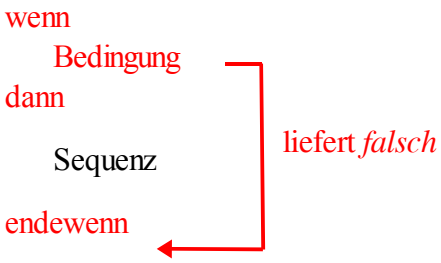
Insbesondere ist - analog zu den Algorithmik-Sequenzen der Jahrgangsstufe 7 und 10 - die Behandlung beider Versionen der Wiederholung mit Bedingung (Anfangs- und Endbedingung) nicht notwendig. Im Rahmen einer Vertiefung oder im Blick auf innere Differenzierung kann die Wiederholungsanweisung mit Endbedingung aber durchaus angesprochen werden.

Bei einer Wiederholung mit Anfangsbedingung wird zu Beginn jedes potentiellen Durchlaufs der zu wiederholenden Sequenz überprüft, ob eine Bedingung zutrifft und dementsprechend die Sequenz wiederholt wird oder nicht.

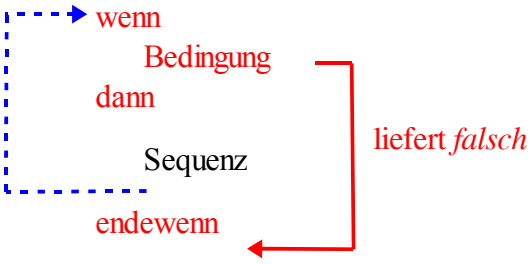
Dies kann man zur Veranschaulichung durch Pfeile verdeutlichen;



Die Auswertung der Bedingung kann über ein WENN-DANN-Konstrukt ausgedrückt werden:



Im Fall, das die Bedingung wahr liefert, wird die Sequenz abgearbeitet. Danach muss aber die Bedingung erneut geprüft werden, d.h. es muss auf jeden Fall an den Anfang zurück gesprungen werden. Damit kann die Wiederholung mit Anfangsbedingung auch folgendermaßen dargestellt werden:



Die Umsetzung der bedingten Anweisung ist den Schülerinnen und Schülern bereits bekannt. Der gestrichelt (blau) gezeichnete Sprung kann durch einen unbedingten Sprung umgesetzt werden, der nach der Sequenz gesetzt wird.

Ob diese Überlegungen richtig sind, wird nun an einem konkreten Beispiel, das vorgegeben wird, verifiziert. Es soll die Potenz 2^3 berechnet werden.

In einem ersten Schritt wird die Problemlösung in Pseudocode formuliert, beispielsweise:

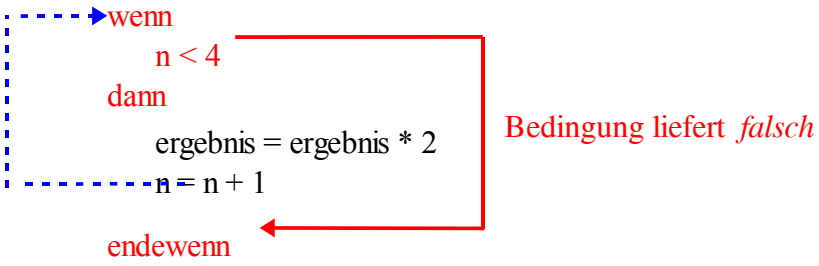
```
ergebnis = 1
 n = 1
```

```
wiederhole solange n < 4
    ergebnis = ergebnis * 2
    n = n + 1
endewiederhole
```

Mithilfe einer Zustandstabelle wird überprüft, ob der Algorithmus das Problem löst.

	ergebnis	n
Vor Beginn der Wiederholungsanweisung	1	1
Überprüfung der Bedingung: 1 < 4		
1. Durchlauf	2	2
Überprüfung der Bedingung: 2 < 4		
2. Durchlauf	4	3
Überprüfung der Bedingung: 3 < 4		
3. Durchlauf	8	4
Überprüfung der Bedingung: 4 < 4 → kein weiterer Durchlauf		

Die Wiederholungsanweisung kann nun gemäß der oben entwickelten Strategie "umformuliert"werden:



Diese Darstellung kann leicht in ein Assemblerprogramm umgesetzt werden. Letztendlich muss eine einseitig bedingte Anweisung implementiert werden, bei der nach der Sequenz ein unbedingter Sprung zum Anfang der bedingten Anweisung gesetzt wird.

Beim folgenden Lösungsvorschlag werden die Werte von *n* bzw. *ergebnis* in den Speicherzellen 20 und 21 abgespeichert. Es wird davon ausgegangen, dass diese Speicherzellen bereits jeweils mit dem Wert 1 und die Speicherzelle 22 mit dem Vergleichswert 4 belegt sind.

Lösungsvorschlag	Kommentierung
`1 load 20`	n laden
`2 sub 22`	im Akku liegt der Wert der Differenz n - 4
`3 jumpz 11`	wenn Akkuwert null ist, d.h. n = 4, springe zu 11
`4 loadi 2`	
`5 mul 21`	
`6 store 21`	
`7 loadi 1`	erhöhe n um 1
`8 add 20`	
`9 store 20`	
`10 jump 2`	springe zu 2 (im Akku liegt schon der aktuelle Wert von n)
`11 end`	

Hinweis: Hier gibt es eine große Palette von alternativen Umsetzungsmöglichkeiten, beispielsweise:

- Anstatt des Produkts 2 * ergebnis kann eine Summe ergebnis + ergebnis berechnet werden.
- Die Erhöhung von n kann auch mit Hilfe des `inc`- Befehls realisiert werden.

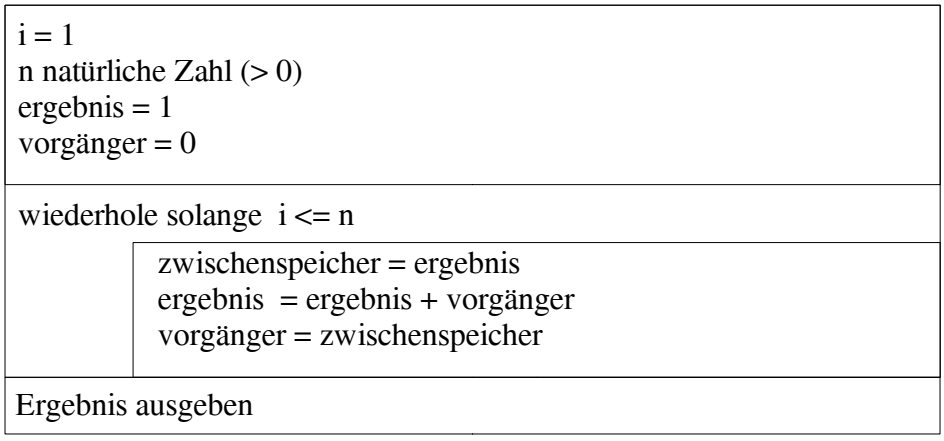
Die Richtigkeit des Programms wird mit einer Zustandstabelle überprüft:

	PC	AK	Zelle 20 (n)	Zelle 21 (ergebnis)
	1	0	1	1
`load 20`	2	1	1	1
`sub 22`	3	-3	1	1
`jumpz 11`	4	-3	1	1
`loadi 2`	5	2	1	1
`mul 21`	6	2	1	1
`store 22`	7	2	1	2
`loadi 1`	8	1	1	1
`add 20`	9	2	1	3
`store 20`	10	2	2	3
`jump 2`	2	2	2	3
`sub 22`	2	-2	2	3
`....`				

Die Wiederholung mit Bedingung wird an Hand von Übungen weiter vertieft:

1. Die Bedingung $n < 4$ in dem obigen Pseudocode wird nun durch $n <= 3$ ersetzt.
   a) Ändert sich damit die Funktionalität des Programms? Begründen Sie kurz.
   b) Passen Sie Ihr Assemblerprogramm an die neue Bedingung an, d.h. nutzen Sie bei der Formulierung der Bedingung konkret den Wert 3.

2. Gegeben ist das folgende Struktogramm:



   a) Implementieren Sie den Algorithmus in einem Assemblerprogramm.

b) Testen Sie Ihr Programm nun für n = 1, n =2, ..., n = 6 und notieren Sie sich für jedes n den Wert von *ergebnis*.

c) Welcher Zusammenhang besteht zwischen den notierten Ergebnissen aus Teilaufgabe b?

d) Überprüfen Sie Ihre Vermutung aus Teilaufgabe c), indem Sie den Wert von *ergebnis* für n = 7 und n = 8 „voraussagen“ und die Ergebnisse mit Ihrem Programm überprüfen.

e) Recherchieren Sie im Internet, unter welchem Namen diese Folge von Zahlen bekannt ist.

3. Es soll der Rest bei der Division einer vorgegebenen positiven Zahl *a* durch 3 bestimmt werden. Dabei können Sie folgende Idee ausnutzen: Von a wird solange der Wert 3 abgezogen, bis ein nicht negativer Wert übrig bleibt, der kleiner als 3 ist. Das muss dann der Rest sein.

a) Vollziehen Sie für a = 11 die Lösungsidee auf Papier nach.

b) Formulieren Sie die Idee als Algorithmus in Form eines Struktogramms.

c) Setzen Sie Ihr Struktogramm in ein Assemblerprogramm um.

d) Testen Sie Ihr Programm für a = 2, a = 9 und a = 11.

e) Fakultativ für Schnelle: Erweitern Sie Ihr Programm so, dass die Restberechnung auch bei negativen Zahlen möglich ist.

4. Der folgende Algorithmus ermittelt zu zwei vorgegebenen positiven natürlichen Zahlen a und b den größten gemeinsamen Teiler:

```
wiederhole solange b > 0
      wiederhole solange a >= b
            a = a – b
      endewiederhole
*     ggt = b
      b = a
      a = ggt
endewiederhole
```

a) Welches Ihnen bereits aus Jahrgangsstufe 10 bekanntes Problem wird durch die mit * markierte Sequenz gelöst?

b) Vollziehen Sie den Algorithmus mit einer Zustandstabelle für a = 15 und b = 6 nach.

c) Implementieren Sie ein entsprechendes Assemblerprogramm und testen Sie es für verschiedene Zahlen a und b.

3.6.3.5 Wiederholung mit Zähler

Die Behandlung dieses Algorithmik-Bausteins ist im Hinblick auf eine Vertiefung sinnvoll. Nachdem auf Assemblerebene die Wiederholung mit Zähler auf die bedingte Wiederholung zurückgeführt wird, ist damit außerdem die implizite Rekapitulation der bedingten Wiederholung verbunden.

Bei der Behandlung der Thematik bieten sich zwei Wege an:

1. Herleitung der Umsetzung aus dem Vergleich der Prinzipien der Wiederholung mit Zähler und der bedingten Wiederholung.

2. Grundsätzliche Analyse des notwendigen Sprungverhaltens bei der Wiederholung mit Zähler und darauf aufbauende Herleitung der Umsetzung.

Vorgehensweise 1:

In einem ersten Schritt wird das Prinzip der Wiederholung mit Zähler analysiert: Ein Durchlauf der zu wiederholenden Sequenz findet statt, solange die Zählvariable kleiner als die Anzahl der gewünschten Wiederholungen ist.

Aus der Formulierung geht hervor, dass also am Anfang der Wiederholung eine Bedingung abgeprüft wird. Damit kann eine Wiederholung mit Zähler offensichtlich als Wiederholung mit Anfangsbedingung dargestellt werden.

wiederhole n-mal
  Sequenz
endewiederhole
	→	wiederhole solange noch nicht n Durchgänge
		  Sequenz
		endewiederhole

Es stellt sich noch die Frage, wie die Zählung der bereits durchlaufenen Wiederholungen realisiert werden kann.

Ansatzpunkt kann beispielsweise die Syntax einer den Jugendlichen bereits aus Jahrgangsstufe 10 bekannten höheren Programmiersprache sein, aus der man die Erfordernisse gut herauslesen kann. Im Fall der Programmiersprache Java sieht der Anweisungskopf folgendermaßen aus:

```
for (int i = 0; i < n, i = i +1)
```

Zählvariable — Bedingung für Wiederholung — Anpassung der Zählvariablen

Daraus lassen sich drei Erkenntnisse schließen, die für die gewünschte Umsetzung nützlich sind:

1. Man braucht eine Zählvariable für die Anzahl der durchlaufenen Wiederholungen. Diese muss zu Beginn auf einen Startwert, hier 0, gesetzt werden.
2. Der Inhalt dieser Zählvariablen muss bei jedem Durchlauf mit der Anzahl der gewünschten Durchläufe (also n) verglichen werden. Für die Anzahl der gewünschten Durchläufe braucht man ebenfalls eine Speicherzelle.
3. Die Zählvariable muss bei jedem Durchlauf angepasst werden, d.h. um 1 erhöht werden.

Damit lässt sich die „Umformung“ von einer Wiederholung mit Zähler in eine Wiederholung mit Anfangsbedingung vervollständigen:

wiederhole n-mal
  Sequenz
endewiederhole
	→	zählvariable = 0
		wiederhole solange zählvariable < n
		  Sequenz
		  zählvariable = zählvariable + 1
		endewiederhole

Das wird nun an einem Beispiel verifiziert.

Hinweis: Je nach Unterrichtssituation kann entweder der folgende, links stehende Pseudocode direkt vorgegeben werden oder ausgehend von der Aufgabenstellung „Es soll die Summe 1 + 1 + … + 1 (n Summanden) berechnet werden. Geben Sie einen möglichen Algorithmus für n = 3 an, der leicht auf andere n angepasst werden kann.“ erst entwickelt werden.

```
ergebnis = 0;                          ergebnis = 0
wiederhole 3-mal                  →    zählvariable = 0
   ergebnis= ergebnis + 1              wiederhole solange  zählvariable < 3
endewiederhole                            ergebnis= ergebnis + 1;
                                          zählvariable = zählvariable + 1
                                       endewiederhole
```

Die Umsetzung in ein Assemblerprogramm ist nun mit dem Wissen aus Kapitel 3.6.3.4 relativ leicht möglich. Im folgenden Lösungsvorschlag wird davon ausgegangen, dass in Speicherzelle 20 der Wert von *ergebnis*, in Zelle 21 der Wert von *zählvariable* und in Zelle 22 der Wert von n bereits abgelegt ist. Zelle 20 und 21 sind bereits mit 0 vorbelegt.

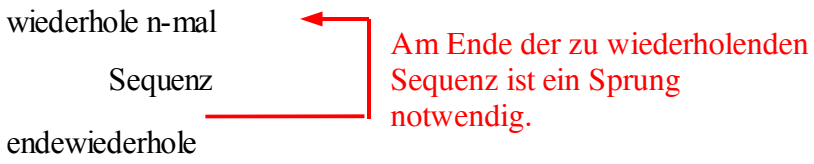
Lösungsvorschlag	Kommentierung
`1 load 22`	n laden
`2 sub 21`	Im Akku liegt der Wert der Differenz n - zählvariable
`3 jumpz 11`	Wenn Akkuwert null ist, d.h. n = zählvariable, dann springe zu 11
`4 loadi 1`	Erhöhe ergebnis um 1
`5 add 20`	
`6 store 20`	
`7 loadi 1`	Erhöhe zähler um 1
`8 add 21`	
`9 store 21`	
`10 jump 1`	Springe zu 1
`11 end`	

Vorgehensweise 2:

Diese Vorgehensweise basiert auf einer Analyse des Sprungverhaltens bei einer Wiederholung mit Zähler.

Hinweis: Bei der Analyse des Sprungverhaltens kann den Schülerinnen und Schülern bereits auffallen, dass die Vorgehensweise analog zuf Umsetzung der bedingten Wiederholung ist. In diesem Fall kann man dann direkt zur Vorgehensweise 1 gewechselt werden.

In einem ersten Schritt machen sich die Schülerinnen und Schüler klar, an welcher Stelle der Wiederholung mit Zähler der Sprung erfolgt und ob es sich dabei um einen bedingten Sprung handelt.



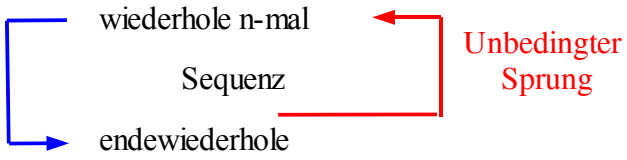
Bezüglich der Frage, ob dies ein unbedingter oder bedingter Sprung sein muss, sind nun prinzipiell zwei Umsetzungsideen möglich.

Die erste Variante (Umsetzung mit unbedingtem Sprung) wird von den Schülerinnen und Schülern wahrscheinlich schneller gefunden, da sie von der Denkweise her die Intuitivere ist. Im Folgenden wird diese Variante weiter ausgeführt.

Natürlich sollte die Wahl der Umsetzungsvariante davon abhängig gemacht werden, welchen Weg die Jugendlichen vorschlagen. Die jeweils andere Idee kann dann ggf. in Übungsaufgaben unter Abwägung eventueller Vor- und Nachteile thematisiert werden.

Variante 1: Umsetzung mit unbedingtem Sprung
Der in der obigen Abbildung herausgearbeitete Sprung wird als unbedingter Sprung umgesetzt. Damit muss am Beginn der Wiederholungsanweisung geprüft werden, ob die Anzahl der gewünschten Wiederholungen (bereits) erreicht ist. Je nach Auswertungsergebnis muss die Sequenz noch einmal durchlaufen bzw. übersprungen werden. Folglich ist am Beginn der Wiederholungsanweisung nochmals ein Sprung notwendig, jetzt aber ein bedingter Sprung.

Bedingter Sprung
Wenn die gewünschte Anzahl der Wiederholungen erreicht ist, dann springe zum Ende der Wiederholungsanweisung.



Hinweis: Hier wird die Bedingung für eine Wiederholung am Anfang geprüft wird. Das hat zwei Vorteile:

- Der Spezialfall der 0-maligen Wiederholung wird berücksichtigt.
- Die Idee ist sehr nahe an der Umsetzung einer Wiederholung mit Anfangsbedingung. Das kann im Abschnitt der bedingten Wiederholung genutzt werden.

Damit ist der grundsätzliche Ablauf bei der Wiederholung mit Zähler klar. In einem nächsten Schritt wird die Umsetzung der Durchlaufzählung thematisiert (analog zur Vorgehensweise 1).

Anschließend erfolgt nun die Umsetzung an einem konkreten Beispiel:

```
ergebnis = 0;
wiederhole 3-mal
        ergebnis= ergebnis + 1;
endewiederhole
```

Nach einer Interpretation des Programms (Berechnung der Summe 1 + 1 + 1 oder Mitprotokollierung der Anzahl der Wiederholungen) wird das Assemblerprogramm entsprechend der gewählten Lösungsstrategie entwickelt.

Dabei kann man sehr systematisch vorgehen:

1. Festlegung (und evtl. Vorbelegung) der notwendigen Variablen, u.a. auch der Zählvariablen
2. Realisierung des bedingten Sprunges
3. Programmierung der Sequenz
4. Inkrementierung der Zählvariablen, d.h. Erhöhung des Zählvariablenwertes um 1
5. Einbau des unbedingten Sprunges
6. Ende-Befehl
7. Festlegung der Sprungadresse beim Sprungbefehl aus Schritt 2

Hinweise:

- Es wird hier unbedingt empfohlen, die Lösung (wenigstens auf Papier) stark zu kommentieren, um den Überblick zu bewahren.

– Im zweiten Schritt kann die Sprungadresse noch nicht festgelegt werden, da zu diesem Zeitpunkt die Adresse des Ende-Befehls noch nicht bekannt ist. Aus diesem Grund darf die Festlegung des Sprungziels nicht vergessen werden (Schritt 7).

  Falls die Simulationssoftware symbolische Adressierung, also Sprungmarken, anbietet, ist deren Verwendung hier sehr zu empfehlen, da man den Namen der Sprungmarke bereits in Schritt 2 angeben kann.

Es könnte sich folgende Lösung ergeben, bei der auch die Belegung der Speicherzellen für die notwendigen Variablen berücksichtigt ist:

Lösungsvorschlag	Kommentierung
	Belegung der Speicherzellen
`1 loadi 0` `2 store 21`	Speicherzelle für Zählvariable. Startwert ist 0
`3 loadi 3` `4 store 22`	Wert und Speicherzelle für die Anzahl der gewünschten Wiederholungen
`5 loadi 0` `6 store 23`	Wert und Speicherzelle für das Ergebnis
`7 load 22` `8 sub 21`	Überprüfung, ob noch ein Durchlauf stattfindet.
`9 jumpz 17`	Wenn WertAdresse22 = WertAdresse21, also WertAdresse22 - WertAdresse22 = 0, dann springe zum Ende der Wiederholungsanweisung, ansonsten (WertAdresse22 > WertAdresse21) gehe linear weiter Ausführung der Sequenz

`10 loadi 1`
`11 add 23`
`12 store 23`	

`13 loadi 1`
`14 add 21`
`15 store 21`
	Inkrementierung des Zählers, d.h. Erhöhung um 1

`16 jump 7`	Unbedingter Sprungbefehl
`17 end`	

Zur Kontrolle und auch zum Verständnis der Vorgehensweise sollte nun unbedingt eine Zustandsbetrachtung erfolgen:

	BZ	AK	Zelle 21 (zähler)	Zelle 22 (n)	Zelle 23 (ergebnis)
	1	0	x	x	x
`loadi 0`	2	0	x	x	x
`store 21`	3	0	0	x	x
`loadi 3`	4	3	0	x	x
`store 22`	5	3	0	3	x
`loadi 0`	6	0	0	3	x
`store 23`	7	0	0	3	0
`load 22`	8	3	0	3	0
`sub 21`	9	3	0	3	0

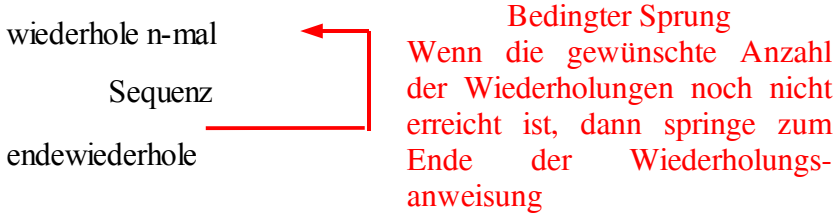
`jmpz 17`	10	3	0	3	0
`loadi 1`	11	1	0	3	0
`add 23`	12	1	0	3	1
`...`					

Ein abschließender Vergleich der Ergebnisse mit der Umsetzung bei der bedingten Wiederholung sollte die Betrachtungen abschließen. Es wird erkannt, dass die Strukturen der jeweiligen Assemblerprogramme identisch sind. Damit wird deutlich, dass es auf dieser Ebene der Programmierung keine unterschiedlichen Konstrukte für Wiederholungsanweisungen gibt. Eine Unterscheidung von Wiederholung mit Zähler und Wiederholung mit (Anfangs-) Bedingung erfolgt erst auf höherer Ebene. Es ermöglicht dort komfortableres Programmieren, da die Wiederholung mit Zähler doch einfacher und schneller verifizierbar ist als die Wiederholung mit Bedingung.

Hinweis: Bei der höheren Programmiersprache C gibt es beispielsweise keine explizit durch spezielle Syntax umgesetzte Wiederholung mit Zähler, wie dies z.B. in Java oder Pascal der Fall ist. Zur Umsetzung wird die bedingte Wiederholung unter Nutzung einer Zählvariablen verwendet.

Variante 2: Umsetzung mit bedingtem Sprung

Bei der zweiten Variante wird nun davon ausgegangen, dass es sich um einen bedingten Sprung handelt. In diesem Fall wird die weitere Durchführung der Wiederholung am Ende der Wiederholungsanweisung geprüft.



Es ist damit im Gegensatz zur 1. Variante nur ein Sprung erforderlich. Sie hat aber den Nachteil, dass die Sequenz mindestens einmal durchlaufen wird, die Wiederholung mit Anzahl 0 also nicht realisiert wird. Dies ist i.d.R. kein Problem, da man eine Wiederholungsanweisung mit Zähler dann einsetzt, wenn man die Anzahl der Wiederholungen vorher weiß. Es könnte aber sein, dass die Anzahl der gewünschten Wiederholungen zu einem früheren Zeitpunkt im Programm berechnet wird. In diesem Fall wäre es durchaus vorstellbar, dass die Anzahl der gewünschten Wiederholungen 0 ist.

Die weiteren Überlegungen sind analog zur Variante 1.

Hinweis: Ein grundsätzliches Problem ist bei der Umsetzung von Wiederholungen mit Zähler zu beachten. Wählt man die maximale Wiederholungsanzahl, die aufgrund der Größe einer Speicherzelle möglich ist, so wird aus der bedingten Wiederholung eine Endloswiederholung (Abschnitt 3.6.3.6).

Angeschlossen werden vertiefende Übungen, beispielsweise:

1. Durch Veränderung des Wertes für die Anzahl der gewünschten Wiederholungen kann das im Unterricht implementierte Programm die Summe mit einer beliebigen Anzahl von Summanden des Wertes 1 berechnet werden. Ändern Sie das Programm so ab, dass auch Summen wie 2 + 2 + 2 oder (-1)+ (-1) + (-1) + (-1) einfach berechnet werden können.

2. Gegeben ist folgender Algorithmus:

ergebnis = 0
anzahlDerBereitsErfolgtenDurchläufe= 0
wiederhole k-mal
ergebnis = ergebnis + anzahlDerBereitsErfolgtenDurchläufe

   a) Was berechnet der Algorithmus? Überprüfen Sie beispielhaft Ihre Vermutung mithilfe einer Zustandstabelle für k = 4.

   b) Implementieren Sie den Algorithmus in Assemblersprache.

3. Berechnet werden soll

   a) die n-te Potenz der ganzen Zahl a: an ($a \in Z$, $n \in N$);

   b) die n-te Potenz der ganzen Zahl a, wobei auch der Spezialfall $a^0 = 1$ berücksichtigt wird: an ($a \in Z$, $n \in N_0$);

   c) die Fakultät von n: n! ($n \in N_0$).

   Folgende Schritte sind dabei jeweils durchzuführen:

   – Angabe des Lösungsalgorithmus für das Assemblerprogramm in Pseudocode;
   – Entwicklung des Assemblerprogramms und Test mit der Simulationssoftware;
   – Nachvollziehen des Programmablaufes für konkrete Werte mithilfe einer Zustandstabelle.

3.6.3.6 Hinweis: Endloswiederholung

Die Umsetzung einer Wiederholung auf Assemblerebene kann ggf. zu einer Endloswiederholung führen, wenn die Zählvariable den größten Wert erreicht bzw. überschreitet, den man in einer Speicherzelle ablegen kann. Man sollte sich dieses Problems grundsätzlich bewusst sein. Eine Thematisierung ist bei sehr leistungsstarken Kursen im Unterricht durchaus vorstellbar.

Dies macht man sich am besten an einem Beispiel klar:

Die Registermaschine soll Speicherzellen der Größe 1 Byte besitzen; außerdem sollen darin aus Einfachheitsgründen nur natürliche Zahlen (eingeschlossen die Codierungen für die Befehle) abgespeichert werden können. Die größte Zahl, die damit in einer Speicherzelle abgelegt werden kann, ist $11111111_2 = 255$.

Folgende bedingte Wiederholung wird zum Problemfall:

```
a = 0
wiederhole solange a <= 255
      a = a + 1
endewiederhole
```

Hat a den Wert 255, so wird die Anweisung a=a+1 ausgeführt; a wird folglich der Wert 256 = 100000000_2 zugewiesen. Im Speicher abgelegt werden kann aber nur eine 8-stellige Dualzahl, es werden die letzten acht Bits verwendet, so dass faktisch der Wert $00000000_2 = 0$ abgespeichert wird. Damit beginnt die Wiederholungsanweisung von vorne und es entsteht eine Endloswiederholung.

Eine Möglichkeit, dieses Problem zu vermeiden, wäre eine Abfrage des Überlauf-Flags. Falls das Überlauf-Flag gesetzt ist, müsste das Programm abgebrochen werden. Dies führt aber weit über den Unterricht hinaus!

4 Grenzen der Berechenbarkeit (ca. 10 Std.)

Lp: Ein wichtiges Maß für die Realisierbarkeit von Algorithmen ist die Effizienz hinsichtlich des Zeitbedarfs. Bei der Untersuchung des Laufzeitverhaltens ausgewählter Algorithmen erkennen die Jugendlichen praktische Grenzen der Berechenbarkeit. Daneben gewinnen sie auch Einblicke in theoretische Grenzen der Berechenbarkeit, sodass sie die Einsatzmöglichkeiten automatischer Informationsverarbeitung realistischer einschätzen können.

Nachdem die Jugendlichen bereits erkannt haben, dass sich nicht formalisierbare Aufgabenstellungen der Bearbeitung durch eine Rechenanlage entziehen, bauen sie nun ihre Fähigkeit zur Beurteilung der Einsatzmöglichkeiten maschineller Informationsverarbeitung weiter aus. Sie vertiefen dabei ihr Bewusstsein für grundlegende Einschränkungen, denen jede auch noch so leistungsfähige Rechenanlage unterworfen ist.

Einen Einblick in die praktischen Grenzen der Berechenbarkeit gewinnen die Schüler mithilfe von Aufwandsbetrachtungen an Aufgabenstellungen wie der Wegesuche, die algorithmisch zwar vollständig lösbar sind, bei denen die Ausführung des jeweiligen Algorithmus aber nicht mit vertretbarem Zeitaufwand realisierbar ist. Den Jugendlichen wird deutlich, dass die Sicherheit moderner Verschlüsselungsverfahren auf den praktischen Grenzen der Berechenbarkeit beruht.

Anhand des Halteproblems lernen die Schüler schließlich auch prinzipielle Grenzen der Berechenbarkeit kennen. Sie sehen ein, dass es mathematisch exakt definierbare Probleme gibt, die algorithmisch und damit auch technisch unlösbar sind.

4.1 Experimentelle Betrachtung unterschiedlichen Laufzeitverhaltens

Lp: experimentelle Abschätzung des Laufzeitaufwands typischer Algorithmen und die damit verbundenen Grenzen der praktischen Anwendbarkeit

4.1.1 Didaktische Hinweise

Als Einstieg in das Thema „Laufzeitaufwand von Algorithmen“ eignen sich experimentelle Untersuchungen des Laufzeitverhaltens von Algorithmen. Die Jugendlichen erkennen, dass verschiedene Ansätze zur Lösung einer Aufgabe deutlich verschiedene Rechenzeiten (Laufzeiten) nach sich ziehen können. Geeignete Beispiele bringen dabei nahe, dass die Ursachen für den verschiedenen Zeitbedarf sowohl in der zu Grunde liegenden Datenstruktur als auch in den verwendeten Algorithmen liegen.

Typische Beispiele sollten – soweit möglich – aus den Unterrichtserfahrungen der Schülerinnen und Schüler stammen und einige typische Laufzeitordnungen (wie $\log(n)$, n, $n \cdot \log(n)$, n^2 und insbesondere $\exp(n)$) abdecken.

Zweckmäßig ist die Behandlung folgender Problemstellungen, die bis auf die Sortieralgorithmen aus Stoffgebieten der Jahrgangsstufe 11 stammen:

- Suche in Listen versus Suche in Bäumen
- Wegesuche in Graphen: Brute force gegen Dijkstra
- Sortieralgorithmen: SelectionSort (Auswählen) und MergeSort

Zur experimentellen Untersuchung des Laufzeitverhaltens von Algorithmen kann man den Jugendlichen fertige Programme an die Hand geben, sodass sie nicht die Algorithmen selbst zu implementieren, sondern lediglich deren Laufzeitverhalten zu untersuchen haben. Das ist insbesondere bei den Wegesuch- und Sortieralgorithmen von Vorteil, da diese Algorithmen vor einer Implementierung entwickelt werden müssen. In Abhängigkeit von der Problemgröße wird dann jeweils der Zeitbedarf des Algorithmus bestimmt. Mithilfe der so gewonnenen Ergebnisse lassen sich Wertetabellen erstellen, die anschließend ausgewertet und visualisiert werden können.

Die fertigen Programme sind als ausführbare jar-Dateien in der Begleit-CD im Ordner „Kapitel_4“ zu finden. Dort finden sich auch die zugehörigen BlueJ-Projekte zur Analyse des Quellcodes. Weiter enthält dieser Ordner erweiterte Varianten der Programme mit eingebautem Zähler für die zeitkritischen Operationen. Alle Programme verlangen als Eingabe die Anzahl der Elemente und ermöglichen die Auswahl der (des) Verfahren(s).

Ein typisches Problem moderner Maschinen ist bei dieser Auswertung die Tatsache, dass kein Prozess den Prozessor exklusiv erhält; Hintergrundprogramme können eine scheinbar zu hohe Rechenzeit verursachen. Um derartige Messfehler gering zu halten, helfen mehrfache Messungen und das Heranziehen der jeweils minimalen Zeit für die Auswertung.

4.1.2 Möglicher Unterrichtsablauf

Zur Motivation kann an einem der oben genannten Beispiele ohne weitere Kommentierung demonstriert werden, dass die Lösung einer bestimmten Aufgabe unterschiedliche Zeit in Anspruch nehmen kann. Anschließend werden die verschiedenen zu untersuchenden Aufgabenstellungen vorgestellt. Zieht man in die Betrachtungen auch Sortieralgorithmen mit ein, so sollte auf den Grundansatz und die Bedeutung dieser Algorithmen etwas näher eingegangen werden, da dieser Bereich für die Schülerinnen und Schüler völlig neu ist. Anschließend wird die generelle Vorgehensweise für die Untersuchung diskutiert (Gruppenweise Bearbeitung der einzelnen Themen, wie tastet man sich an sinnvolle Elementzahlen heran, wie wird die Messung durchgeführt, wie werden die Ergebnisse notiert) und die Gruppeneinteilung vorgenommen. Eventuell können schon erste Laufzeitmessungen gemacht werden.

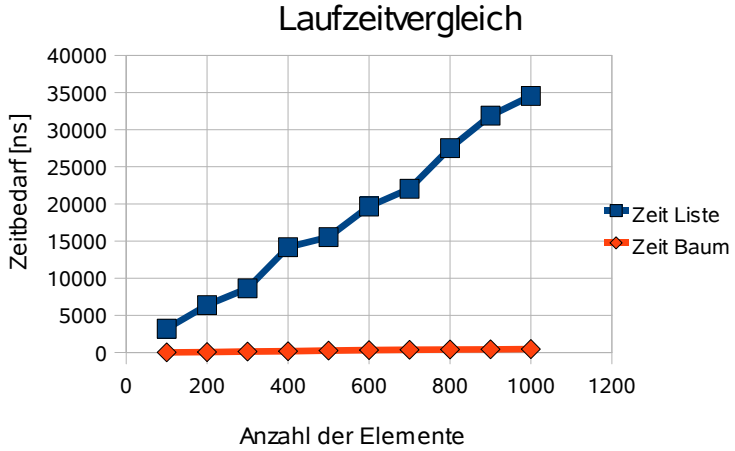
In der Folge arbeiten die Gruppen selbsttätig an ihren Teilaufgaben. Die Ergebnisse werden notiert und z. B. mithilfe von Tabellenkalkulationsprogrammen (Tabellen und Graphiken) aufgearbeitet.

Nach Durchführung der Messungen werden die Ergebnisse der Gruppen vorgestellt und verglichen. Dabei kann durchaus versucht werden, Kurvenformen herauszuarbeiten. Drei Gruppen sind hierbei sicher ansprechbar: steigt linear, steigt weniger als linear und steigt mehr als linear. Details sind wegen der großen Streuungen der Laufzeiten in der Regel nicht zu erkennen.

Mögliche Ergebnisse können sein:

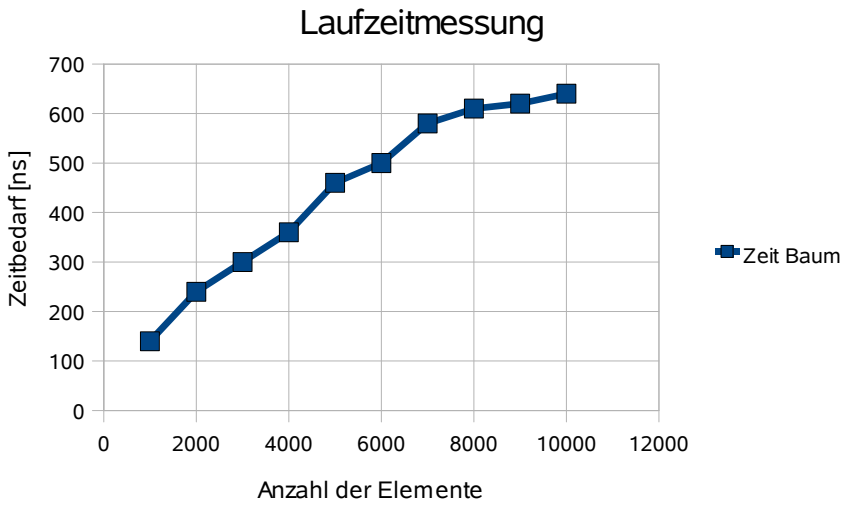
Laufzeitvergleich bei der Suche

Elementanzahl	100	200	300	400	500	600	700	800	900	1000
Zeit Liste [ns]	3200	6390	8640	14180	15539	19660	22040	27520	31883	34542
Zeit Baum [ns]	20	60	124	180	255	320	370	403	427	462



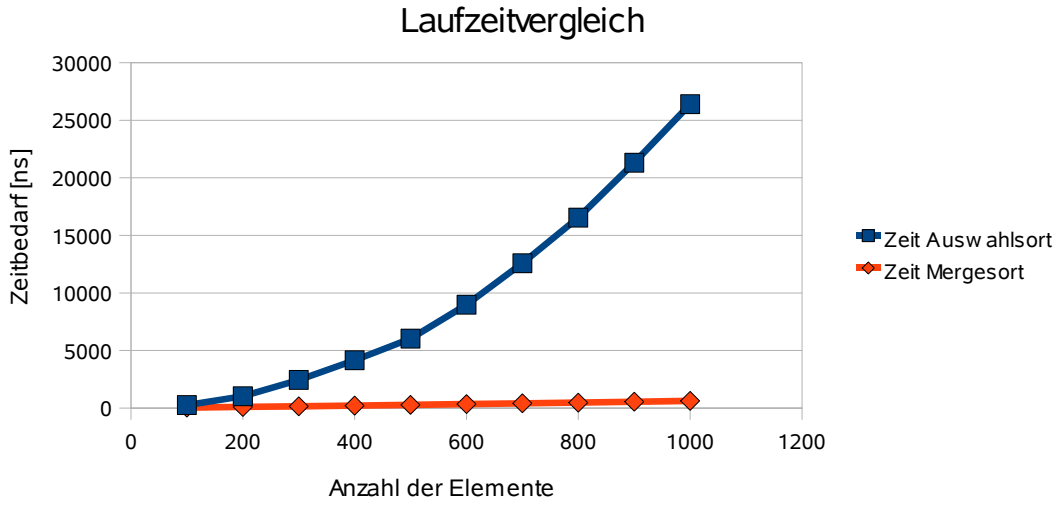
Suche im Binärbaum mit größerer Elementzahl

Elementzahl	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Zeit Baum [ns]	140	240	300	360	460	500	580	610	620	640



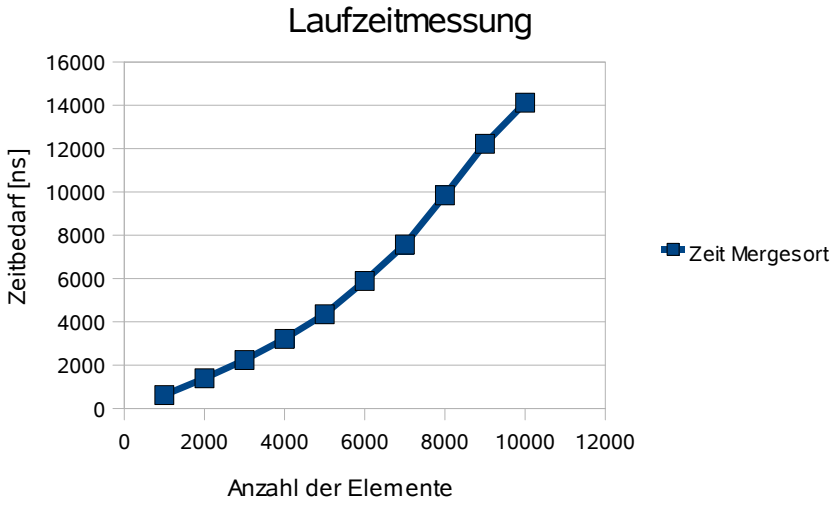
Vergleich der Sortieralgorithmen

Elementanzahl	100	200	300	400	500	600	700	800	900	1000
Zeit Auswahlsort [ns]	257	1026	2442	4154	6035	8975	12580	16541	21316	26391
Zeit Mergesort [ns]	44	99	151	213	276	343	411	479	549	619



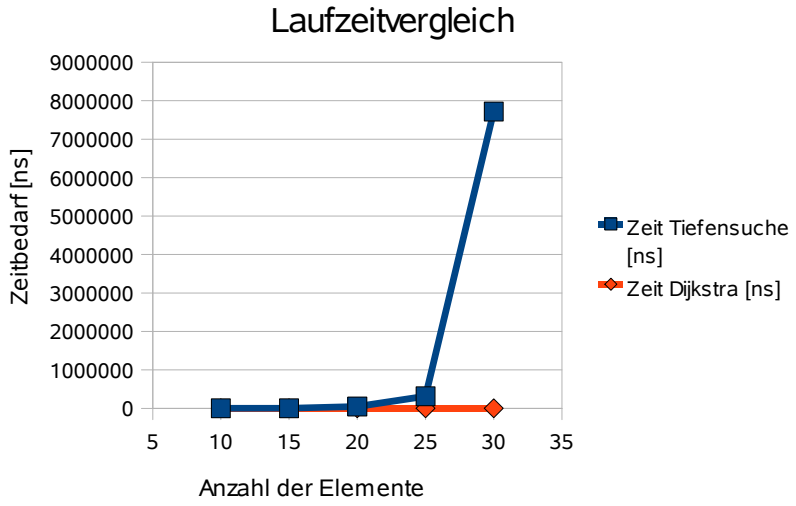
Mergesort bei größeren Datenmengen

Elementanzahl	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Zeit Mergesort [ns]	621	1390	2237	3214	4351	5887	7570	9847	12221	14120



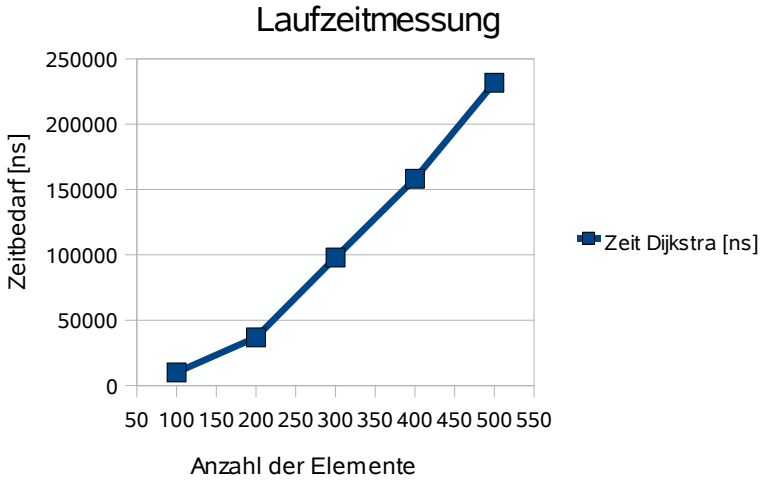
Vergleich der Wegesuchalorithmen

Elementanzahl	10	15	20	25	30
Zeit Tiefensuche [ns]	260	3297	47231	314391	7716470
Zeit Dijkstra [ns]	85	218	390	616	830



Dijkstra bei größeren Datenmengen

Elementanzahl	100	200	300	400	500
Zeit Dijkstra [ns]	10076	36815	97974	158203	231685



4.2 Analyse der Laufzeitordnung durch Abzählen zeitrelevanter Operationen

Lp: experimentelle Abschätzung des Laufzeitaufwands typischer Algorithmen und die damit verbundenen Grenzen der praktischen Anwendbarkeit

4.2.1 Didaktische Hinweise

Bei der Auswertung der Experimentalergebnisse zeigt sich, dass die Bestimmung der Elementanzahl-Zeit-Funktionen nur grob möglich ist; eine genaue Bestimmung des Funktionstyps und damit eine echte Vorhersage des Zeitbedarfs von Problemen mit mehr Elementen (wie längerer Listen oder umfangreicherer Bäume, mehr zu sortierender Werte, Graphen mit mehr Knoten) oder der Zeitersparnis durch schnellere Rechner ist damit aber nicht möglich. Daher ist es zum Erreichen des übergeordneten Lernziels „Fähigkeit zur Beurteilung der Einsatzmöglichkeiten maschineller Informationsverarbeitung“ notwendig, sich genauer mit dem Zeitbedarf der verschiedenen Beispiele auseinander zu setzen. Dass diese Auseinandersetzung notwendig ist, kann den Schülerinnen und Schülern gut nahe gebracht werden. Im Zuge dieser Überlegungen wird schnell deutlich, dass es nicht auf die Details der Elementanzahl-Zeit-Funktion ankommt, sondern nur auf die Art des am schnellsten wachsenden Teilterms, die so genannte Laufzeitordnung. Weiter ist es für eine (mögliche) Laufzeitoptimierung notwendig, die Gründe für die benötigte Rechenzeit genau zu kennen.

Der Lehrplan fordert keines der im folgenden genannten Verfahren explizit ein, so dass sowohl die Wahl der Verfahren wie auch die Tiefe der Behandlung der Unterrichtssituation angemessen gewählt werden können. Es werden exemplarisch gut brauchbare Verfahren genannt. In den Ausführungen zur Umsetzung werden Vorschläge für die Variation der Tiefe der Behandlung gegeben.

Bei der Analyse der Suchalgorithmen (BlueJ-Projekt Suchen als mögliche Vorlage) ist schnell klar, dass der wesentliche Zeitfaktor die Anzahl der Vergleiche ist, die benötigt werden, bis der gesuchte Wert gefunden ist. Auch bei den Sortieralgorithmen kann die Anzahl der Vergleiche als zeitkritische Operation leicht erkannt werden. Das ist um so offensichtlicher, wenn man statt Zahlen Zeichenketten sucht bzw. sortiert, da hier die Vergleichsoperation deutlich aufwendiger ist als alle anderen Operationen. Für die Analyse der Sortierverfahren ist es hilfreich, die jeweilige Grundidee den Schülerinnen und Schülern kurz vorzustellen. Angesichts der begrenzten Zeit müssen die betrachteten Verfahren gut verständlich und schnell analysierbar sein; relevante Unterschiede im Zeitbedarf können nur festgestellt werden, wenn mindestens ein Verfahren der Ordnung $O(n^2)$

sowie eines mit O(n·log(n)) betrachtet wird. Auf Zusatzfragen wie das Laufzeitverhalten bei vorsortierten bzw. umgekehrt vorsortierten Daten kann aus Zeitgründen nicht eingegangen werden.

Die Betrachtung der Wegesuche in Graphen bindet nicht nur dieses Kapitel an den Stoff der vorigen Jahrgangsstufe. Die „sture“ Untersuchung aller möglichen Wege (brute force) ist auch wichtig, um ein Verfahren mit exponentieller Laufzeit vorzustellen, denn auf diese Laufzeitordnung hebt die Untersuchung der Sicherheit von Verschlüsselungsverfahren ab. Bei der einfachen Wegesuche ist die für die Laufzeitordnung entscheidende Frage, wie oft jeder Knoten „besucht“ werden muss. Beim Dijkstra-Algorithmus ist diese Frage etwas kniffliger, da hier neben der Bearbeitung des aktuellen Knotens noch ein weiterer wesentlicher Zeitbedarf beim Suchen des nächsten Knotens in der Liste der zur Bearbeitung anstehenden Knoten entsteht.

Anhand von MergeSort lässt sich auch schnell auf die Ergänzung kommen, dass nicht nur der Zeit-, sondern auch der Platzbedarf von Interesse für den Anwender sein kann.

4.2.2 Möglicher Unterrichtsablauf

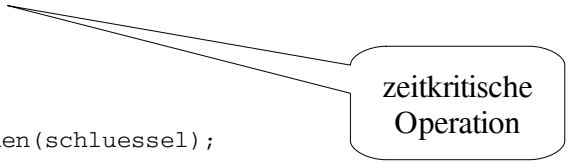
Der folgende Ablauf geht von einer maximalen Ausführung des Themas aus; bei in Frage kommenden Stellen ist angegeben, wie sie eingespart werden können.

Zeitkritische Operationen

Motiviert aus den unzureichenden Ergebnissen der Zeitmessungen werden zunächst die Suchalgorithmen näher untersucht. Dabei ist direkt klar, dass das „Besuchen“ jeden Knotens gleich viel Zeit benötigt; dass die Zeit hauptsächlich für die Schlüsselvergleiche benötigt wird, kann vermutet werden, ist aber für das Ergebnis hier nicht von Belang. Da der Besuch jedes Knotens nur aus dem Aufruf der Methode *Suchen* besteht, genügt es, diese Methode zu betrachten.

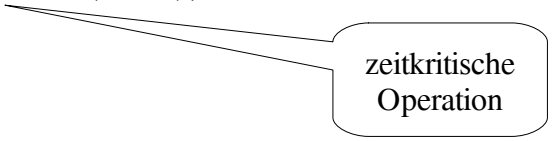
Suche bei der Liste:

```
DATENELEMENT Suchen(DATENELEMENT schluessel)
{
    if (schluessel.Vergleichen(daten) == 0)
    {
        return daten;
    }
    else
    {
        return nachfolger.Suchen(schluessel);
    }
}
```



Suche beim Baum

```
DATENELEMENT Suchen (DATENELEMENT schluessel)
{
    int vergleich;
    vergleich = schluessel.Vergleichen(daten);
    if (vergleich == 0)
    {
        return daten;
    }
    else if (vergleich < 0)
    {
        return linksnachfolger.Suchen(schluessel);
    }
    else
    {
        return rechtsnachfolger.Suchen(schluessel);
    }
}
```



Für eine Liste mit n Datenelementen ist sofort klar, dass für das Finden des ersten Elements eine Zeiteinheit, für das Finden den letzten Elements n Zeiteinheiten benötigt werden. Bei gleichmäßiger Streuung der Werte gibt das im Schnitt n/2 Zeiteinheiten pro Suchvorgang. Diskussionen über nicht vorkommende Elemente bzw. den Unterschied zwischen sortierten und nicht sortierten Listen können dahin kanalisiert werden, dass das nur den Faktor ändert, grundsätzlich aber gilt: Zeitbedarf ~ n. An dieser Stelle könnte die Landau-Notation O(n) erwähnt werden; eine Einführung ist laut Lehrplan jedoch nicht gefordert.

Für die Suche im Baum kann nun festgestellt werden, dass der schlimmste Fall das Suchen eines Blattelements ist. Bei einem ausgeglichenen Binärbaum mit n Elementen ist die Höhe eines Astes etwa $\log_2(n)$, nicht gut ausgeglichene Bäume ändern daran nur wenig; eine Verlängerung des längsten Astes um z. B. 20 % führt nur zu einem konstanten Faktor vor dem Logarithmus: $1{,}2 \cdot \log_2(n)$. Damit gilt hier: Zeitbedarf ~ log(n).

Beim Vorstellen der Sortierverfahren kann man sich auf das jeweilige Prinzip beschränken.

Beim Sortieren durch direktes Auswählen wird im unsortierten Teil des Feldes nach dem jeweils kleinsten Wert gesucht. Der Beginn des unsortierten Teils ist mit dem Pfeil in der Zeile „Trennung“ markiert. Zu Beginn ist noch nichts sortiert.

Index:	1	2	3	4	5	6
Wert:	2	8	7	1	7	4
Trennung:	↑					

Um den kleinsten Wert zu finden, merkt man sich zunächst den ersten Wert ab der Trennung.

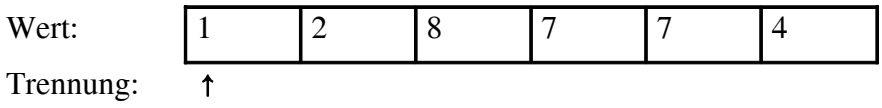
Index:	1	2	3	4	5	6
Wert:	2	8	7	1	7	4
Trennung:	↑					
kl. Wert:	↑					

Nun geht man das Feld ab der nächsten Position durch. Dabei kommt man zur „1“ und merkt sich deren Position:

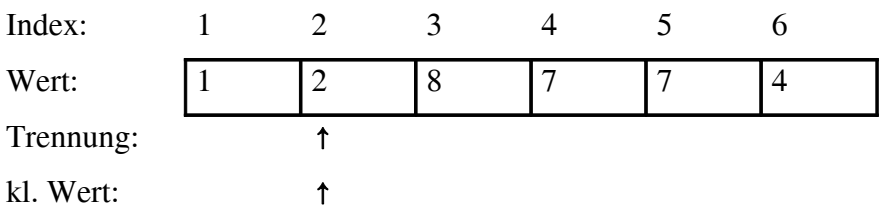
Index:	1	2	3	4	5	6
Wert:	2	8	7	1	7	4
Trennung:	↑					
kl. Wert:				↑		

Nun speichert man die „1“ in einer Hilfsvariablen und schiebt alle Feldelemente bis zur Trennposition um einen Platz nach oben. Die „1“ wird an der Trennposition wieder eingefügt.

Index:	1	2	3	4	5	6

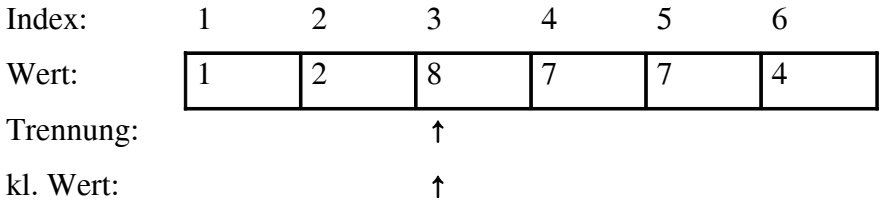


Das erste Element ist damit gefunden. Die Trennposition kann nun erhöht werden. Mit dem zweiten Element verfährt man genauso. Diesmal sind Trennelement und kleinstes Element identisch:

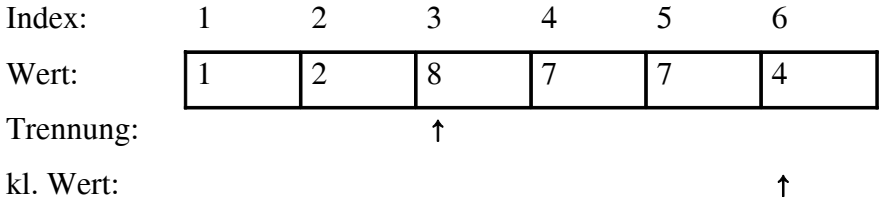


Dafür ist keine Sonderbehandlung im Algorithmus nötig, da durch das oben beschriebene Verfahren einfach die „2“ mit sich selbst „vertauscht“ wird.

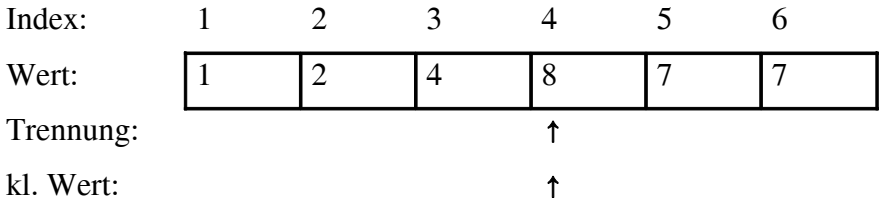
Für den nächsten Durchlauf wird der Wert des Trennindex wieder um 1 erhöht, man beginnt also auf Position 3:



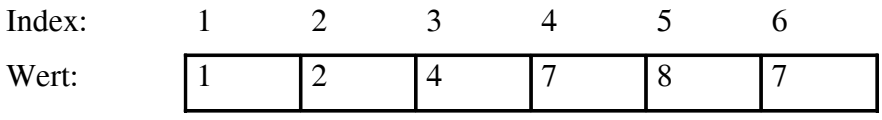
Diesmal steht der kleinste Wert, nämlich 4, auf Position 8:



Nach dem Hochschieben und Erhöhen des Trennindex ergibt sich:



Beim vierten Durchlauf wird die erste 7 gefunden, da die zweite 7 ja nicht kleiner ist:



Trennung:					↑	
kl. Wert:					↑	

Nach dem fünften Durchlauf ergibt sich folgendes Bild:

Index:	1	2	3	4	5	6
Wert:	1	2	4	7	7	8
Trennung:						↑
kl. Wert:						↑

Das letzte, übrig bleibende Element ist sowieso das größte und braucht nicht mehr behandelt zu werden.

Untersuchung der Wegesuchalgorithmen

Dieser Teil kann eingespart werden, wenn man den Schülerinnen und Schülern die Wegesuchalgorithmen mit eingebauter Zählung der relevanten Operationen vorgibt (BlueJ-Projekt Wegesuche_Zaehlen).

Die Analyse der Wegesuchalgorithmen ist aufwendig, insbesondere da das Verfahren nach Dijkstra erklärt werden müsste, was deutlich über den Lehrplan hinaus geht. Für gute Schülergruppen bietet sich hier aber ein spannendes Feld der Vertiefung an.

Zuerst wird die Suche des kürzesten Weges durch wiederholte Anwendung der Tiefensuche analysiert. Bei diesem Verfahren wird mittels Tiefensuche der Weg von A nach B ermittelt. Die Methode wird aber nicht beendet, wenn der erste Weg gefunden ist. Es wird nur der Weg gespeichert und die Tiefensuche dann fortgesetzt; beim Zurückgehen in der Rekursion werden besuchte Knoten wieder frei gegeben. Auf diese Weise werden alle existierenden Wege gegangen. Der nächste gefundene Weg wird mit dem bisher besten verglichen, der kürzere gespeichert usw.

Da alle möglichen Wege gegangen werden müssen, muss man von jedem der n Knoten aus etwa m weitere Knoten besuchen, wobei m die durchschnittliche Kantenanzahl pro Knoten ist. Für den ersten Knoten sind das m Folgeknoten, nach dem zweiten m·m , nach dem dritten m·m·m usw. Insgesamt müssen also m^n Knoten besucht werden. Mit $m^n = e^{\ln(m^n)} = e^{n \cdot \ln(m)}$ kommt man zu einem exponentiellen Zeitbedarf, also Zeitbedarf~ exp(n).

Das Arbeiten des Dijkstra-Algorithmus zeigt man am besten anhand von Folien (siehe Präsentationen im Ordner „Präsentationen für Dijkstra-Algorithmus“) mit den einzelnen Schritten der Suche in einem gegebenen Graphen. Entscheidender Punkt für die Laufzeit ist die Liste der „in Bearbeitung befindlichen“ Knoten, deren Länge proportional zur Gesamtzahl n der Knoten ist. Da in diese Liste (fast) jeder Knoten im Graphen einsortiert werden muss, ergibt sich der Zeitbedarf n · n, also Zeitbedarf ~n^2.

Implementierung der Zählung der zeitkritischen Operationen

Die Implementierung kann weggelassen werden, wenn den Jugendlichen die Projekte mit eingebauter Ausgabe der Zeitschritte (Vergleiche, Knoten besuchen etc.) vorgegeben werden (BlueJ-Projekte Suche_Zaehlen, Sortieren_Zaehlen, Wegesuche_Zaehlen).

Gemäß den Überlegungen der ersten (und zweiten Stunde) identifizieren die Schülerinnen und Schüler die zeitkritischen Operationen und erhöhen dort einen Zähler. Hierfür bieten sich Klassenvariable (statische Variable) an.

Vergleich der Zeitmessung mit der theoretischen Untersuchung

Der Vergleich kann weggelassen werden, wenn auf den Test der theoretisch ermittelten Laufzeitordnungen verzichtet wird.

Die Schülerinnen und Schüler führen die Tests durch (hier sind keine Wiederholungen nötig, da die Anzahl der Operationen ja fix ist) und stellen das Ergebnis auf die gewohnte Weise dar. Gegebenenfalls überprüfen sie auch das Ergebnis auf Übereinstimmung mit der vorhergesagten Ordnung (Wurzel ziehen, Logarithmieren etc., um eine Proportionalität n ~ f(Zeitbedarf) zu erhalten). Diese Überprüfung kann aber auch beim Zusammentragen der Ergebnisse durchgeführt werden.

4.3 Laufzeitordnung von Schlüsselangriffen

Lp: hoher Laufzeitaufwand als Schutz vor Entschlüsselung durch systematisches Ausprobieren aller Möglichkeiten (Brute-Force-Verfahren)

4.3.1 Didaktische Hinweise

Ziel dieses Kapitels ist es aufzuzeigen, dass die exponentielle Laufzeitordnung bestimmter Algorithmen auch Schutz vor Entschlüsseln von Nachrichten durch systematisches Ausprobieren aller möglichen Schlüssel bietet. Die Jugendlichen sollen hier in die Lage versetzt werden, die Sicherheit heute üblicher Schlüsselverfahren sowohl jetzt als auch über den typischen Zeitraum ihrer Nutzung abzuschätzen (elektronisch signierte Verträge können z.B. für 30 Jahre abgeschlossen werden). Dabei kann das Prinzip dieser Verfahren kurz angesprochen werden, eine allgemeinere Beschäftigung mit Verschlüsselungsmethoden ist durch den Lehrplan nicht impliziert.

Schülerinnen und Schülern (und Lehrerinnen und Lehrern), die sich aus Eigeninteresse näher mit Schlüsselverfahren beschäftigen wollen, können die Bücher „Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet.“ von Simon Singh sowie die Bücher „Kryptologie“ und weitere von Albrecht Beutelspacher empfohlen werden.

Ein Schlüsselverfahren besteht aus zwei Funktionen. Die erste hat als Eingabeparameter den Klartext und eine weitere Zeichenkette (den Schlüssel) und erzeugt daraus den verschlüsselten Text. Die zweite Funktion hat als Eingabeparameter den verschlüsselten Text und einen nicht notwendig gleichen Schlüssel und erzeugt daraus wieder den Klartext. Noch bis Mitte des vorigen Jahrhunderts war es üblich zu versuchen, den Schlüssel aus dem verschlüsselten Text zurück zu rechnen. Heutige Schlüsselverfahren arbeiten mit nicht umkehrbaren Funktionen ausreichender Komplexität, so dass ein Rückrechnen des Schlüssels im Allgemeinen nicht möglich ist. Daher ist es nur möglich, alle denkbaren Schlüssel systematisch auszuprobieren, die so genannten Brute-Force-Angriffe: Angriffe mit brutaler (Rechenleistungs-)Gewalt statt mit Überlegung. Dieser Ansatz hat aber exponentielle Laufzeitordnung: Bei einem Schlüssel mit Länge n und a möglichen verschiedenen Zeichen pro Stelle müssen a^n verschiedene Möglichkeiten ausprobiert werden.

Leistungsstarke Computer können heute pro Sekunde bis zu 400 Millionen Schlüssel testen. Bei einfachen Passwörtern (Schlüssel) mit z.B. 6 Kleinbuchstaben gibt es 26^6 = 308915776 Möglichkeiten; die Zeit, alle Passwörtern auszuprobieren, beträgt nur 308915776 / 400000000 s ≈ 0,77 s. Für ein 12-stelliges Passwort aus Groß- und Kleinbuchstaben, Ziffern und gängigen Sonderzeichen (etwa 20) ergeben sich 82^{12} = $9{,}24200562703 \cdot 10^{22}$ Möglichkeiten; der Test dauert dann etwa 7 Millionen Jahre. Bei einer angenommenen Verdoppelung der Rechnerleistung alle

zwei Jahre (Mooresches Gesetz) würde das Brechen dieses Passworts in 30 Jahren immer noch ca. 220 Jahre dauern.

Anmerkung: Da viele Anwender sich ihre Passwörter leicht merken können wollen, ist es beim Versuch, Schlüssel zu brechen, üblich, zuerst alle „typischen“ Passwörter aus der Umgangssprache, wie Vornamen oder Begriffe, auszuprobieren; dafür existieren bereits spezielle Passwortwörterbücher. Dieser Ansatz kann durch das Verwenden von Ziffern und / oder Sonderzeichen in den Passwörtern verhindert werden. Bei der Anmeldung an Computern, auch über das World-Wide-Web, wird heute als zusätzlicher Schutz nach mehreren Fehlversuchen eine Pause bis zur nächsten Passworteingabe erzwungen, so dass der Zeitbedarf für das Brechen des Passwortes schon indiskutabel wird, wenn nur eines dieser Passwortwörterbücher abgearbeitet werden soll. Derartige Mechanismen sind aber nicht möglich, um das Entschlüsseln verschlüsselter Dateien zu erschweren.

4.3.2 Möglicher Unterrichtsablauf

Motivation kann ein Ausflug in die Geschichte der Kryptographie sein. Bei einfachen Verschlüsselungsverfahren, die bestimmte Charakteristika der Sprache beibehalten, stellt die Analyse der Buchstabenhäufigkeit eine prinzipielle Angriffsmöglichkeit dar. Heutige Verfahren verwischen diese Charakteristika so weit, dass nur noch der zweite Weg bleibt, das systematische Ausprobieren aller Schlüssel.

Die Anzahl möglicher Schlüssel ist sehr schnell abgeschätzt. Bei m möglichen Zeichen und n Stellen des Schlüssel gilt: Anzahl = m^n; hier liegt also ein Problem mit exponentieller Laufzeitordnung vor. Die Zeit pro Entschlüsselungsversuch ist nur grob abschätzbar, da sie vom Verfahren und der Länge des Textes abhängt. Die Wirkung grundlegender Einflussfaktoren wie Größe des Alphabets und Länge des Schlüssels sind aber sofort einschätzbar.

An dieser Stelle kann auch thematisiert werden, warum „bevorzugte“ Schlüssel auch zuerst getestet werden und welchen geringen Bruchteil der möglichen Schlüssel diese Lieblingsschlüssel überhaupt ausmachen. So sind Namen nur bis zu ca. 10 Zeichen lang, beginnen mit einem Groß- oder Kleinbuchstaben (je nach Gewohnheit) und bestehen dann aus Kleinbuchstaben. Das ergibt $52 \cdot 26^9 = 282334191306752$ Möglichkeiten. 10-stellige Schlüssel auch aus Sonderzeichen lassen $82^{10} = 1{,}37448031336 \cdot 10^{19}$ Möglichkeiten zu, also etwa 50000 mal so viel Möglichkeiten; sie benötigen zum Brechen daher auch die 50000-fache Zeit.

In der Folge können tatsächlich benötigte Zeiten für Schlüsselangriffe berechnet werden. Beispielsweise durch Internetrecherche bekommen die Jugendlichen Daten über den aktuellen Zeitbedarf für das Ausprobieren eines Schlüssels bei heute gängigen Verfahren. Sie können daraus abschätzen, ab welcher Schlüssellänge damit von sicherer Verschlüsselung gesprochen werden kann. Eine Beispielrechnung ist oben (Kapitel 4.3.1) zu finden.

Unter der Annahme unbegrenzter Gültigkeit des Mooreschen Gesetzes (Verdopplung der Rechnerleistung alle zwei Jahre) können auf dieser Basis Vorhersagen über die Sicherheitsieh z. B. 30 Jahre gültiger, elektronisch unterzeichneter Verträge gemacht werden (Beispielrechnung e 4.3.1).

Ein kurzer Ausflug in die Möglichkeiten, Brute-Force-Angriffe durch eine maximale Zahl von Fehlversuchen zu unterbinden, rundet diese Einheit ab. Dabei werden auf Verfahren wie „Sperrung nach n Fehlversuchen“ (z. B. Handy-PIN, manche Webseiten) oder „Zwangspause nach n Fehlversuchen“ (z. B. Anmeldung am Computer) angesprochen. Es sollte den Schülerinnen und Schülern aber auch bewusst werden, dass solche Schutzmechanismen nicht zur Verfügung stehen, wenn verschlüsselte Texte angegriffen werden. Hier hilft allein die Güte des Verfahrens und des Schlüssels.

4.4 Das Halteproblem

Lp: prinzipielle Grenzen der Berechenbarkeit anhand von Plausibilitätsbetrachtungen zum Halteproblem

4.4.1 Vorüberlegungen

In der Öffentlichkeit herrscht häufig die Hoffnung, dass, wenn Computer immer „intelligenter“ werden, sie eines Tages auch in der Lage sein werden, sich selbst zu reparieren. Speziell wird erwartet, dass sie Fehler in ihren Programmen selbst erkennen und reparieren können. Diese auf die Informationsverarbeitung übertragene Idee eines modernen Perpetuum Mobile gibt Anlass, sich mit der Frage zu beschäftigen, was alles algorithmisch berechnet werden kann.

Bisher wurden bereits zwei Einschränkungen für die Berechenbarkeit gegebener Aufgaben erkannt:

- Ist die Fragestellung formal nicht beschreibbar, kann mit dem Computer keine Lösung berechnet werden (vergleiche Abschnitt 12.1 des Lehrplans).
- Es gibt Aufgabenstellungen, deren Berechnung mehr Zeit kostet, als aufgewendet werden kann.

Die allgemeine Aufgabe, eine Methode zu entwickeln, die die Fehler anderer Programme entdeckt (und gegebenenfalls korrigiert) ist nicht formalisiert, da dafür zuerst der Begriff „Fehler“ formal beschrieben werden müsste. Ein wesentlicher Teil dieser allgemeinen Fragestellung kann aber sehr wohl formal beschrieben werden:

Gesucht ist eine Methode, die überprüfen kann, ob eine beliebige Methode für eine bestimmte Eingabe terminiert (oder nicht).

Den Schülerinnen und Schülern sind „hängen gebliebene“ Programme bekannt, d. h. Programme, die mit ihrer Aufgabe nie fertig werden. Diese Fragestellung wird als relevant erkannt.

Für die Untersuchung dieser von Alan Touring 1936 als **Halteproblem** formulierten Frage muss zunächst die Fragestellung noch weiter präzisiert werden. Gibt es eine Methode

boolean TerminierungTesten (String methode, String eingabe),

die für beliebige Werte von *methode* und zugehöriger Eingabe (zugehörigen Parameterwerten) entscheiden kann, ob die gegebene Methode für diese Eingabe terminiert?

Hier sollte kurz die Idee des Widerspruchsbeweises aufgezeigt werden, da mathematische Beweise dieser Form den Jugendlichen nicht vertraut sind: Wenn man von einer Aussage vermuten kann, dass sie richtig ist, sucht man mithilfe logischer Folgerungen, deren Richtigkeit zu beweisen. Wenn man aus gutem Grund vermuten kann, dass eine Annahme falsch ist, ist es aber oft sehr schwierig, das direkt zu zeigen. Man geht dann lieber einen kleinen Umweg und zeigt, dass die Annahme, die Aussage sei richtig, zu einem Widerspruch führt; die Aussage kann also doch nicht richtig sein.

4.4.2 Möglicher Unterrichtsablauf

Zunächst wird, motiviert durch die obigen Überlegungen, das Halteproblem formuliert, beispielsweise:

Gibt es eine Methode

```
boolean TerminierungTesten (String methode, String eingabe),
```

die für beliebige Werte von Methode und zugehöriger Eingabe (zugehörigen Parameterwerten) entscheiden kann, ob die gegebene Methode für diese Eingabe terminiert?

Der Rückgabewert wahr bedeutet, dass die gegebene Methode mit der gegebenen Eingabe terminiert, falsch bedeutet, dass die gegebene Methode mit der gegebenen Eingabe nicht terminiert.

Es kann mitgeteilt werden, dass der Engländer Alan Touring das Halteproblem formulierte (Kurzbiographie) und aufgrund der Vermutung, dass eine solche Methode nicht existiere, den nachfolgend skizzierten Widerspruchsbeweis fand.

Zunächst folgt hier noch eine kurze Skizze, was ein Widerspruchsbeweis ist (siehe oben).

Die Basis des Beweises ist folgende Methode:

```
void Test(String methode)
{
    while (TerminierungTesten(methode, methode))
    {
        // keine Aktion
    }
}
```

Die Methode *Test* terminiert genau dann, wenn die übergebene Methode mit ihrem eigenen Text als Eingabedaten nicht terminiert.

Die Methode *Test* wird nun mit sich selbst als Eingabe aufgerufen:

```
Test (<Quelltext von Test>);
```

TerminierungTesten wird also mit dem Quelltext von Test für beide Eingabeparameter aufgerufen; sie muss daher überprüfen, ob *Test* mit sich selbst als Eingabe terminiert oder nicht. Die beiden möglichen Ergebnisse von *TerminierungTesten* werden untersucht.

a) *TerminierungTesten* liefert wahr.

Das bedeutet, dass *TerminierungTesten* „behauptet“, *Test* würde mit sich selbst als Eingabe terminieren. Wenn aber *TerminierungTesten* den Wert wahr liefert, lautet der Algorithmus von *Test* zusammengefasst aber

```
while (true) {};
```

d. h. *Test* enthält eine Endloswiederholung und terminiert nicht. Das ist ein Widerspruch zu der Behauptung von *TerminierungTesten*, dass *Test* terminieren würde.

b) *TerminierungTesten* liefert falsch.

Das bedeutet, dass *TerminierungTesten* behauptet, *Test* würde mit sich selbst als Eingabe nicht terminieren. Wenn aber *TerminierungTesten* den Wert falsch liefert, lautet der Algorithmus von *Test* aber

```
while (false) {};
```

d. h. *Test* ist sofort zu Ende, terminiert also. Das ist ein Widerspruch zu der Behauptung von *TerminierungTesten*, dass *Test* nicht terminieren würde.

Damit ergibt sich die Zusammenfassung, dass alle von *TermierungTesten* erzeugbaren Ergebnisse auf Widersprüche stoßen, diese Methode kann also nicht existieren.

Im Folgenden können weitere Konsequenzen aus dem Ergebnis des Halteproblems gezogen werden, z. B. die Erweiterung, dass kein Programm allgemein alle Fehler eines anderen Programms finden, geschweige denn beheben kann oder die Konsequenz, dass prinzipiell nicht alle formalisiert darstellbaren Probleme auch berechnet werden können.

Als Schlussfazit lassen sich die im Laufe dieses Schuljahres aufgetretenen Grenzen der Berechenbarkeit zusammenfassen:

1. Probleme, die nicht formalisiert werden können, können auch nicht berechnet werden.
2. Es gibt formal beschreibbare Probleme, die zwar prinzipiell berechnet werden können, bei denen die Berechnung aber viel zu lange dauert.
3. Es gibt formal beschreibbare Probleme, die prinzipiell nicht berechnet werden können.